Computer Science II

Dr. Chris Bourke
Department of Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588, USA
http://chrisbourke.unl.edu
cbourke@cse.unl.edu

2019/08/15 13:02:17
Version 0.2.0

This book is a *draft* covering Computer Science II topics as presented in CSCE 156 (Computer Science II) at the University of Nebraska—Lincoln.

Contents

1 Introduction **1**

2 Object Oriented Programming **3**
2.1 Introduction 3
2.2 Objects 4
2.3 The Four Pillars 4
2.3.1 Abstraction 4
2.3.2 Encapsulation 4
2.3.3 Inheritance 4
2.3.4 Polymorphism 4
2.4 SOLID Principles 4
2.4.1 Inversion of Control 4

3 Relational Databases **5**
3.1 Introduction 5
3.2 Tables 9
3.2.1 Creating Tables 10
3.2.2 Primary Keys 16
3.2.3 Foreign Keys & Relating Tables 18
3.2.4 Many-To-Many Relations 22
3.2.5 Other Keys 24
3.3 Structured Query Language 26
3.3.1 Creating Data 28
3.3.2 Retrieving Data 30
3.3.3 Updating Data 36
3.3.4 Destroying Data 36
3.3.5 Processing Data 37
3.4 Normalization & Database Design 47
3.4.1 Design Example 50
3.5 Miscellaneous 52
3.6 Exercises 53

4 List-Based Data Structures **57**
4.1 Array-Based Lists 58
4.1.1 Designing a Java Implementation 58

4.2 Linked Lists . . . 65
4.2.1 Designing a Java Implementation . . . 71
4.2.2 Variations . . . 74
4.3 Stacks & Queues . . . 77
4.3.1 Stacks . . . 77
4.3.2 Queues . . . 82
4.3.3 Variations . . . 86

5 Algorithm Analysis 91
5.1 Introduction . . . 91
5.1.1 Example: Computing a Sum . . . 97
5.1.2 Example: Computing a Mode . . . 99
5.2 Pseudocode . . . 102
5.3 Analysis . . . 105
5.4 Asymptotics . . . 109
5.4.1 Big-O Analysis . . . 109
5.4.2 Other Notations . . . 111
5.4.3 Observations . . . 113
5.4.4 Limit Method . . . 116
5.5 Examples . . . 118
5.5.1 Linear Search . . . 118
5.5.2 Set Operation: Symmetric Difference . . . 120
5.5.3 Euclid's GCD Algorithm . . . 121
5.5.4 Selection Sort . . . 123
5.6 Other Considerations . . . 124
5.6.1 Importance of Input Size . . . 124
5.6.2 Control Structures are Not Elementary Operations . . . 127
5.6.3 Average Case Analysis . . . 128
5.6.4 Amortized Analysis . . . 129
5.7 Analysis of Recursive Algorithms . . . 130
5.7.1 The Master Theorem . . . 131
5.8 Exercises . . . 134

6 Trees 137
6.1 Introduction . . . 137
6.2 Definitions & Terminology . . . 137
6.3 Implementation . . . 146
6.4 Tree Traversal . . . 147
6.4.1 Preorder Traversal . . . 148
6.4.2 Inorder Traversal . . . 150
6.4.3 Postorder Traversal . . . 153
6.4.4 Tree Walk Traversal . . . 157
6.4.5 Breadth-First Search Traversal . . . 159

6.5 Binary Search Trees 161
6.5.1 Retrieval 162
6.5.2 Insertion 163
6.5.3 Removal 165
6.5.4 In Practice 167
6.6 Heaps 169
6.6.1 Operations 170
6.6.2 Implementations 175
6.6.3 Variations 179
6.6.4 Applications 180
6.7 Exercises 183

Appendix **185**
1 Author-Book Database SQL 185

Glossary **187**

Acronyms **191**

Index **195**

References **195**

List of Algorithms

1 Insert-At-Head Linked List Operation . . . 66
2 Insert Between Two Nodes Operation . . . 67
3 Index-Based Retrieval Operation . . . 69
4 Key-Based Delete Operation . . . 71

5 Computing the Mean . . . 103
6 Computing the Mode . . . 104
7 Trivial Sorting (Bad Pseudocode) . . . 104
8 Trivially Finding the Minimal Element . . . 105
9 Finding the Minimal Element . . . 105
10 Linear Search . . . 119
11 Symmetric Difference of Two Sets . . . 120
12 Euclid's GCD Algorithm . . . 121
13 Selection Sort . . . 123
14 Sieve of Eratosthenes . . . 125
15 Fibonacci(n) . . . 131
16 Binary Search – Recursive . . . 133
17 Merge Sort . . . 133

18 Stack-based Preorder Tree Traversal . . . 149
19 preOrderTraversal(u): Recursive Preorder Tree Traversal . . . 150
20 Stack-based Inorder Tree Traversal . . . 151
21 inOrderTraversal(u): Recursive Inorder Tree Traversal . . . 153
22 Stack-based Postorder Tree Traversal . . . 154
23 postOrderTraversal(u): Recursive Postorder Tree Traversal . . . 157
24 Tree Walk based Tree Traversal . . . 160
25 Queue-based BFS Tree Traversal . . . 161
26 Search algorithm for a binary search tree . . . 163
27 Finding the maximum key value in a node's left subtree. . . . 167
28 Heapify . . . 171
29 Find Next Open Spot - Numerical Technique . . . 179
30 Heap Sort . . . 181

List of Code Samples

3.1 A simple table definition. We use this as an illustrative example, it is not necessarily a well-designed table. 10
3.2 A many-to-many Author/Book table design 23
3.3 Full Author/Book table database 27

4.1 Parameterized Array-Based List in Java 88
4.2 A linked list node Java implementation. Getter and setter methods have been omitted for readability. A convenience method to determine if a node has a next element is included. This implementation uses `null` as its terminating value. 89

5.1 Summing a collection of integers 94
5.2 Summation Algorithm 1 97
5.3 Summation Algorithm 2 97
5.4 Summation Algorithm 3 97
5.5 Mode Finding Algorithm 1 99
5.6 Mode Finding Algorithm 2 100
5.7 Mode Finding Algorithm 3 101
5.8 Naive Exponentiation 126
5.9 Computing an Average 127

List of Figures

3.1 CSV Formatted Data 6
3.2 JSON and XML data representation examples 8
3.3 Parent Child Table Relation 20
3.4 Entity-Relation Diagram 21
3.5 A many-to-many relationship using a join table. 24
3.6 Author/Book Many-to-Many Data 25
3.7 Final Author-Book Database 28
3.8 A 3-D cube projected onto a 2-D surface producing a square. 38
3.9 Enrollment Database 52
3.10 Enrollment Data 53
3.11 Simple Video Game Database 54

4.1 A simple linked list containing 3 nodes. 66
4.2 Insert-at-head Operation in a Linked List. We wish to insert a new element, 42 at the head of the list. 67
4.3 Inserting Between Two Nodes in a Linked List. Here, we wish to insert a new element 42 between the given two nodes containing 8 and 6. 68
4.4 Delete Operation in a Linked List 70
4.5 Key-Based Find and Remove Operation. We wish to remove the first node we find containing 42. 72
4.6 A Doubly Linked List Example 75
4.7 A Circularly Linked List Example 76
4.8 An Unrolled Linked List Example 76
4.9 A stack holding integer elements. Push and pop operations are depicted as happening at the “top” of the stack. In actuality, a stack stored in a computer’s memory is not really oriented but this visualization is consistent with a physical stack growing “upwards.” 79
4.10 An example of a queue. Elements are enqueued at the end of the queue and dequeued from the front of the queue. 82
4.11 Array-Based Queue 85
4.12 Producer Consumer Pattern 86

5.1 Quadratic Regression of Index-Based Linked List Performance 96
5.2 Plot of two functions. 108
5.3 Expected number of comparisons for various success probabilities p. 129

6.1 An undirected graph with labeled vertices. 138

6.2 Several examples of trees. It doesn't matter how the tree is depicted or organized, only that it is acyclic. The final example, 6.2(d) represents a disconnected tree, called a *forest*. 140
6.3 Several possible rooted orientations for the tree from Figure 6.2(a). . . . 141
6.4 A Tree Node's Relations. 143
6.5 A Binary Tree 143
6.6 A complete tree of depth $d = 3$ which has $1 + 2 + 4 + 8 = 15$ nodes. . . . 144
6.7 A summation of nodes at each level of a complete binary tree up to depth d.145
6.8 A small binary tree. 148
6.9 A walkthrough of a preorder traversal on the tree from Figure 6.5. 149
6.10 A walkthrough of a inorder traversal on the tree from Figure 6.5. 152
6.11 A walkthrough of a postorder traversal on the tree from Figure 6.5. . . . 156
6.12 A tree walk on the tree from Figure 6.8. 158
6.13 The three general cases of when to process a node in a tree walk. 158
6.14 A Breadth First Search Example 159
6.15 A Binary Search Tree 162
6.16 Various Search Examples on a Binary Search Tree 164
6.17 Binary Search Tree Insertion Operation 165
6.18 Binary Search Tree Deletion Operation Exampless 168
6.19 A degenerate binary search tree. 168
6.20 A min-heap 169
6.21 A Max-heap 170
6.22 An Invalid Max-heap 171
6.23 Insertion and Heapification 172
6.24 Another Invalid Heap 173
6.25 Removal of the root element (getMax) and Heapification 174
6.26 Heap Node's Index Relations 176
6.27 An array implementation of the heap from Figure 6.21 along with the generalized parent, left, and right child relations. 176
6.28 Tree-based Heap Analysis 178

1 Introduction

> “Smart data structures and dumb code works a lot better than the other way around”
>
> — Eric S. Raymond [8]

To Come.

2 Object Oriented Programming

2.1 Introduction

To Come.

Alan Kay (Pronounced oooo-p): ”the main thing about doing oop work or any kind of programming work is that there has to be some exquisite blend between beauty and practicality. There is no reason to sacrifice either one of those and people who are willing to sacrifice either one of those I think, don’t really get what computing is all about. (OOPSLA 1997); Alan Kay created smalltalk and coined ”OOP”

As things become more complex, architecture outstrips material.

maybe; General Responsibility Assignment Software Patterns (GRASP)

2.2 Objects

2.3 The Four Pillars

2.3.1 Abstraction

2.3.2 Encapsulation

2.3.3 Inheritance

2.3.4 Polymorphism

2.4 SOLID Principles

2.4.1 Inversion of Control

solId idea: Inversion of Control: 3 GPS data provider (Android provider using Location + float for lat/lon + certain method names, 2nd that is more accurate, but uses doubles, no location object, etc.: problems with adapting to "set" a marker on a map; then introduce a 3rd provider to show that even more code needs to change; then invert the control by defining a single interface and adapters for each provider

3 Relational Databases

3.1 Introduction

Consider the data in Table 3.1 for a course enrollment system.

Course	Course Title	Student	NUID	Email
		waits, tom	10001949	tomwaits@hotmail.com
CSCE 156	Intro to CSII	Lou Reed	10001942	reed@gmail.com
CSCE 156	Introduction to CS2	Tom Waits	10001949	twaits@email.com
CSCE 230	Computer Hardware	Student, J.	12345678	jstudent@unl.edu
CSCE 156	Intro to CSII	Student, John	12345678	jstudent@unl.edu
MATH 479	Wonton Burrito Meals	Philip J. Fry	10002000	fry@unl.edu
CSCE 235	Discrete Math	Student, John	12345678	jstudent@unl.edu
CSCE 235	Discrete Math	Student, John	12345678	jstudent@cse.unl.edu
CSCE 235	Discrete Math	Tom Waits	10001942	twaits@email.com
CSCE 235	Discrete Math	Lou Reed	10001942	reed@gmail.com
NONE	null	Lou Reed	10001942	reed@gmail.com

Table 3.1: Course Enrollment Data

The data in this table is essentially a flat file. Though we've presented it in a nicely formatted table, this data is essentially no different from a Comma-Separated Value (CSV) file in which each line represents a record and individual column values are delimited with a comma. The same data as a CSV file is presented in Figure 3.1. Flat file representations of data are called such because they take related pieces of data, such as which courses a student is enrolled, and *flatten* them into a single table representation. Even ancillary data that have a *completely different* relationship (such as emails) is included in the "flattened" representation.

Flat file representations can lead to a lot of problems, limitations and can lead to data anomalies. To motivate the use of a proper database system, we identify several of these here.

- There is **no semantic meaning** to the data fields. Though the first row contains *headers* of what each column represents, there is no way to *enforce* those associations. Since this data format is merely a plaintext file, we could accidentally swap columns,

```
Course,Course Title,Student,NUID,Email
,,waits,tom,10001949,tomwaits@hotmail.com
CSCE 156,Intro to CSII,Lou Reed,10001942,reed@gmail.com
CSCE 156,Introduction to CS2,Tom Waits,10001949,twaits@email.com
CSCE 230,Computer Hardware,Student,J.,12345678,jstudent@unl.edu
CSCE 156,Intro to CSII,Student,John,12345678,jstudent@unl.edu
MATH 479,Wonton Burrito Meals,Philip J. Fry,10002000,fry@unl.edu
CSCE 235,Discrete Math,Student,John,12345678,jstudent@unl.edu
CSCE 235,Discrete Math,Student,John,12345678,jstudent@cse.unl.edu
CSCE 235,Discrete Math,Tom Waits,10001942,twaits@email.com
CSCE 235,Discrete Math,Lou Reed,10001942,reed@gmail.com
NONE,null,Lou Reed,10001942,reed@gmail.com
```

Figure 3.1: CSV Formatted Data

omit columns or provide more tokens than expected. Alternatively, some data representation formats are designed for Electronic Data Interchange (EDI). In order to transfer data between two different systems (that may be on different hardware, use different data models or are written in different languages) data needs to be translated (a process called serialization) into a universal format in which each data field is *semantically marked up* to indicate *what* it represents. Two examples can be found in Figure 3.2 which contains samples of the same enrollment data represented as JavaScript Object Notation (JSON) and Extensible Markup Language (XML) respectively.

- There is a lot of **repetition** of data. Every record that models a student enrolled in a course repeats all the student's data including their name, ID, email, etc. Likewise, the course information is also repeated over every enrollment record.
- Repeated data can lead to different and conflicting **representations and formatting problems**. In some of the student records the name is similar, but represented differently, some are last name first, others are first name first, others have a middle initial or only have an initial for the first name, one doesn't use capitalization, etc.
- There is **missing or incomplete data**. The first and last record do not have any course data and they represent this missing data inconsistently: with empty string values in the first record, a placeholder value ("NONE"), and a null value. These may be records that once held data but that was changed. However, so as not to lose the (still relevant) email data, they were modified.
- There is **inconsistent** data. The third from last record for example has a different NUID for Tom Waits as it does from the other records with a similar name. This NUID is also associated with Lou Reed in the final two records. Without any rules to enforce consistency, there is no data integrity and incorrect data like this is

allowed to occur.

- There are **organizational and efficiency** issues with processing a single data file. Any aggregate reports, such as producing a roster for a particular class or a schedule for a single student, would require processing every record in the entire file. Similarly, updating a single record becomes extremely difficult. For example, if one student change their email, we have to find and replace every instance with several contingencies (dealing with multiple email records) and side effects to deal with.
- Keeping data in a single file also has **concurrency Issues**. If we want to design a system to be multi-user or even simply multithreaded or have several programs access and process our data then each process or thread may have to do so by placing a file lock on the data file. This precludes the possibility of concurrent and parallel processing, severely limiting a system's efficiency and essentially making it a single user system.

The solution to (most) of these problems is to use a proper Relational Database Management System (RDMS). A relational database stores data in *tables*. Tables delineate data in columns and rows. Each column holds a specific type of data (integers, strings, etc.) that define fields of *records*. Each row in a table corresponds to a single record.

Moreover, each row can be uniquely identified using a unique *primary key* which can be automatically managed by the database. Data between tables can be related to each other using *foreign keys*. A database enables you to define rules and *constraints* to enforce *data integrity*. Each column has a particular type and can be made to be required with invalid values rejected. Separating data into different tables and relating data between tables reduces the failure points, reduces redundancy and minimizes the potential for data integrity. Databases are sophisticated programs that structure data and provide better organization. They provide features for concurrency, authentication, security, and more that address the problems identified previously.

The key features that an RDMS offers can be summed up with the Atomicity, Concurrency, Isolation, Durability (ACID) principles. In a database, the basic unit of work is known as a transaction. A transaction may consist of one or more queries which are individual requests for data (retrieving current records) or requests for operations on data in a database (creating new records, updating or deleting current records). Databases treat transactions as an all-or-nothing operation that is either *committed* (changes are permanently made to the data in a database) or *rolled back* (there was a problem with the query or a potential data integrity issue that would make the operation(s) invalid). The ACID principles ensure that every transaction has the following four properties.

- **Atomicity**: transactions must be an all-or-nothing operation. Suppose a transaction consists of multiple queries that modify the data and that the third query would violate the constraints (rules) of the database. The transaction would fail and rollback as if the first two queries never happened. Atomic transactions are

```json
[
  {
    "Course": "",
    "Course Title": "",
    "Student": "waits",
    "NUID": "tom",
    "Email": 10001949
  },
  {
    "Course": "CSCE 156",
    "Course Title": "Intro to CSII",
    "Student": "Lou Reed",
    "NUID": 10001942,
    "Email": "reed@gmail.com"
  },
  ...
  {
    "Course": "NONE",
    "Course Title": null,
    "Student": "Lou Reed",
    "NUID": 10001942,
    "Email": "reed@gmail.com"
  }
]
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<enrollments>
  <enrollment>
    <Course></Course>
    <Student>waits</Student>
    <NUID>tom</NUID>
    <Email>10001949</Email>
    <Course_Title></Course_Title>
  </enrollment>
  <enrollment>
    <Course>CSCE 156</Course>
    <Student>Lou Reed</Student>
    <NUID>10001942</NUID>
    <Email>reed@gmail.com</Email>
    <Course_Title>Intro to CSII</Course_Title>
  </enrollment>
  ...
  <enrollment>
    <Course>NONE</Course>
    <Student>Lou Reed</Student>
    <NUID>10001942</NUID>
    <Email>reed@gmail.com</Email>
    <Course_Title/>
  </enrollment>
</enrollments>
```

Figure 3.2: JSON and XML data representation examples

not divisible or decomposable.

- **Consistency**: all transactions will retain a state of *consistency* so that all constraints and requirements of the database will be preserved and all data will conform to them before and *after* the transaction is committed. For efficiency, a database may temporarily violate constraints *during* a transaction, but consistency is guaranteed afterwards.
- **Isolation**: no transaction interferes with or is even aware of another. This ensures that we have concurrency and that different threads, programs or users will not interfere with each other.
- **Durability**: once committed, a transaction remains so, preserving the constraints, rules and integrity of the data.

The relational data model was originally developed by Edgar Codd [5] in 1974. Today there are many relational database systems ranging from Free and Open Source Software (FOSS) (SQLite, PostgreSQL, MariaDB) to commercial software (SQLServer, DB2, Oracle) costing hundreds or even millions of dollars. The Structured Query Language (SQL) is the defacto standard for interacting with these databases (designed by Chamberlin & Boyce [4] at IBM). Each one of these systems has their own *dialect* of SQL (PostgreSQL uses Psql, SQLServer uses Transact-SQL) that offers vendor-specific functionality and features. However, most of them do support the core features of the language which we'll focus on.

3.2 Tables

The first step to creating a database is to define tables that will hold data. When creating tables, you specify the table's name as well as its columns and the type of data each column represents. Many tools exist that allow you to create a database using a graphical user interface or even to programmatically generate a database using existing data. We'll focus on using basic SQL to create our tables. Typically, scripts to create tables are stored in Document Description Language (DDL) files which are simply plaintext files containing the commands to generate a database. You can also add comments to your SQL scripts using two hyphens, `--` or sometimes the hash symbol `#`.

```sql
-- This table will be used to hold data on books
-- ISBN = International Standard Book Number
-- dewey corresponds to the Dewey Decimal number
create table Book (
  bookId integer primary key auto_increment not null,
  title  varchar(255) not null,
  author varchar(255),
  isbn   varchar(255) not null default '',
  dewey  float,
  numCopies integer default 0
);
```

Code Sample 3.1: A simple table definition. We use this as an illustrative example, it is not necessarily a well-designed table.

3.2.1 Creating Tables

To create a table, we use the following basic syntax.

```sql
create table TableName (
  columnA columnType [options],
  columnB columnType [options],
  ...
);
```

We use the keyword/phrase `create table` and provide a table name. The columns of the table are defined within opening and closing parentheses and the entire statement is ended with a semicolon. Column definitions are delimited with commas and each one has a name, data type and a series of options that we'll discuss later. In order to have something to work with, we provide a full example as Code Sample 3.1.

Style

Typically, the naming rules for table names and column names are less strict than many programming languages. Most databases allow you to use special characters, combinations of numbers, lower and uppercase letters, underscores, and even spaces.[1] Some databases treat these as case insensitive and others may treat them as case sensitive. Or, it may even be dependent on the configuration of a particular database setup. In

[1]To include spaces, you typically need to encapsulate table/column names using backticks: `` `My Favorite Books` ``. You can sometimes even used otherwise reserved SQL keywords for table names as long as you escape them using backticks. However, this is generally discouraged.

any case, to avoid many of these issues we'll follow a "modern" naming convention of upper camel casing for table names and lower camel casing for column names. We prefer this style because ultimately, our data will be associated with objects and variables in a programming language. Following the same naming convention makes the one-to-one connection between data and objects convenient and straightforward.

Many style guides prefer a more traditional naming convention of lower casing for table names and lower underscore casing for column names. For example, the table in Code Sample 3.1 using this convention might have a table name of `book` while the first and last columns would have the names `book_id` and `num_copies` instead. If you have a choice of convention, whichever you choose is fine as long as you are consistent throughout all of your tables.

Another notable item in our styling is that we use the modern convention of using lower case SQL keywords. Though table names and columns may or may not be case sensitive, SQL keywords such as `create table` or `varchar` or `not null` are case *insensitive*.

The more traditional way of writing SQL is to use all uppercase letters in order to make the SQL keywords distinct from the table/column names. For example, the SQL in Code Sample 3.1 may instead look like the following.

```sql
CREATE TABLE Book (
  book_id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL,
  title  VARCHAR(255) NOT NULL,
  author VARCHAR(255),
  isbn   VARCHAR(255) NOT NULL DEFAULT '',
  dewey  FLOAT,
  num_copies INTEGER DEFAULT 0
);
```

However, we'll prefer the modern convention of using lower casing for all of our SQL keywords. This has the advantage of not having to hold the shift key down while typing half of our characters, saving our pinky fingers in the process. Moreover, the traditional convention dates back to a time when monitors were monochromatic and using different casing made SQL code easier to read. We live in the modern era with IDEs and syntax highlighting (as in our examples, SQL code is highlighted in green) which arguably makes the upper casing convention obsolete.

In addition, it is best practice to avoid pluralizations in table names. After all, it is the `Book` table, not the "books table." The table contains book records, but the table itself is a single entity. English is not the most consistent language when it comes to pluralization anyway. Best to avoid these issues altogether by consistently using singular forms for all of your tables.

Basic Data Types

Most SQL databases support a wide variety of column data types. For simplicity, we'll restrict our attention to a few of the most useful data types.

- `integer` (or `int`) is typically a traditional 32-bit signed integer allowing you to store values in the range $[-2, 147, 483, 648, 2, 147, 483, 647]$. You can insert and modify data within this range. However, if you try to insert a value outside this range, your query may result in an error or a warning. Depending on the particular database, your data may end up getting modified; a value greater than the maximum value may get rest to the maximum value for example.

  Some database systems support variations on this basic integer data type. For example, MySQL also supports `tinyint` , `smallint` `mediumint` and `bigint` which are 1, 2, 3, and 8 byte integer values respectively.

- `double` or `float` (or `real`) are floating-point numeric types. The exact supported keywords are highly dependent on the database being used, but typically implement the same floating-point types as most programming languages (as IEEE 754 floating-point numbers). `double` is usually an 8-byte number offering 15-17 digits of precision and `float` is a 4-byte floating-point number giving 6-8 digits of precision.

  Many databases allow custom precision floating-point datatypes using syntax such as `decimal(p,s)` or `numeric(p,s)` where p is the precision (number of total digits of precision) and s is the scale (number of factional digits of precision or number of digits to the right of the decimal place). For example, `decimal(65,30)` would give you 65 total digits of precision with up to 30 digits in the factional part. Though it may seem like you can have arbitrarily large numbers, many databases will still limit the maximum precision either internally or by configuration.

- `varchar(n)` can be considered as the basic string type. The keyword stands for **var**iable **char**acter column in which you can store basic ASCII characters (or Unicode if the database supports it). The n is typically specified as the *maximum* number of characters that can be stored in the column. For example, `varchar(255)` [2] would allow the column to store any text value of up to 255 characters. Attempts to insert or modify data that exceeds the limit usually result in *truncation*: the first n characters of the data will be stored, but any data exceeding the limit is cutoff and lost. Truncation may also result in a warning.

  In SQL, string *literals* can be denoted with either single or double quotes: `'foo'` `"bar"` and `'hello world'` are all examples.

[2]You'll often see the magic number 255 pop up in many examples as a column size that is "big enough" for many columns. This number has some historic meaning: it was the internal limit in earlier MySQL versions. The 255 would also allow the database to use at most one byte as metadata to store the actual length of the stored string.

> Some databases include extended support for "large" text fields. MySQL for example supports `text` (64KB), `mediumtext` (16MB), or `longtext` (4GB) while PostgreSQL supports `text` which allows an arbitrarily large text field.
>
> There may be other issues involved with `varchar`. Collation refers to how data is stored and compared in a database. For example, `varchar` data may be stored as ASCII or Unicode or some other *character set*. When retrieving or ordering data, comparisons may be made in a case sensitive or insensitive matter. The configuration of the database, or individual table or column or even how a query is made may all determine how data is stored or compared or and may result in unexpected or unintended results. For this reason, it is essential to choose sensible defaults and to be consistent about them.

In addition to those described, databases may support many other column data types or *aliases*. Aliases are simply convenient names or abbreviations for other data types. As mentioned above, `int` is an alias for `integer` for example.

Databases may also support other non-numeric data types such as date/time values (though these are rarely consistent and their use is often discouraged) or binary types (often referred to as Binary Large Object (BLOB)) that allow you to store binary files such as image, audio, and video files directly in a database table. Some databases support enumerated types such that column values may be limited to a few predefined values or even `boolean` types that may only be set to `true` or `false`. PostgreSQL even allows you to store JSON formatted data as a column type and even allows you to query these this sub-formatted data directly!

Databases may support each of these types simply as an alias to another simpler type. For example MySQL supports `boolean` but does not have an actual `boolean` type. Instead, it uses a `tinyint` value and uses the convention that 0 is false and any non-zero value is true. For more details you would need to Read The Manual (RTM) of your database system as many of these data types are non-standard and may not be portable from one database to another.

Options

Depending on the type of data stored in a column, you may be able to apply several *options* to specify additional rules or constraints to the values in the column. As with data types, there are dozens of different options, but we'll only focus on the most useful here.

- `primary key auto_increment` defines a primary key that is automatically assigned by the database system for new records. We'll cover keys in more detail in Section 3.2.2.
- `not null` specifies that column values are not allowed to be null. By default, column values are *nullable* which means that specific data values do not need to be

specified and instead can take on a value of `null` (which is a recognized keyword in SQL). Nullable fields essentially make a column value optional and allow for situations in which data may be "missing" or undefined. To disallow this, you can use the option `not null`; any attempt to insert a record without specifying the a valid value for the column (or alternatively, any attempt to change a column value to `null`) will result in an SQL error and the record may be rejected. In general, column values should be made `not null` unless there is a good reason to allow them to be nullable.

- `default (value)` allows you to define a default value for a column value. If a user inserts a new record without defining a value for a column with this option, then the default value will be used for this new record. Of course, if the user inserts a new record with a specified value then that value will be used instead. This option is typically used in conjunction with the `not null` option though they do not have to be used together.

To illustrate the effect of these options reconsider the example given in Code Sample 3.1, repeated here for convenience. Let's make some observations about each of the columns.

```sql
create table Book (
  bookId integer primary key auto_increment not null,
  title  varchar(255) not null,
  author varchar(255),
  isbn   varchar(255) not null default '',
  dewey  float,
  numCopies integer default 0
);
```

- The `title` column may take on any string value of up to 255 characters, but it may *not* be set to `null`. Consequently any new record must have a title specified. Moreover, any existing record may not have its title changed to `null`.
- The `author` column has no options so by default it is allowed to be `null`. New records inserted without an author value will default to `null`. Existing records can have their author value changed to `null`.
- The `isbn` column is not allowed to be `null` but a default value is specified. In this case, the default is the empty string (which is not the same thing as `null`), string literals being denoted with single or double quotes. Existing records may not have their ISBN value changed to `null`. However, new records may be inserted without specifying a value for the ISBN column as the database will use the defined default.
- The `dewey` column is specified as a `float` (which suffices for the Dewey Decimal System) but with no further options, it is allowed to be `null` and has no default value. Equivalently, you can think of `null` as being the default value for new

records that get inserted with no dewey value specified.

- The `numCopies` column is allowed to be nullable, but defines a default value of zero. This means that newly inserted records that do not specify a value for `numCopies` will not get set to `null`, but instead will get set to 0. However, you could modify existing records and set the `numCopies` column value to `null` if you wished.

Other Table Operations

Just as you can create a new table, you can also get rid of an existing table. The syntax for doing this is simply

```
drop table TableName
```

This will remove the table entirely including all records, potentially losing a lot of data. For this reason, many SQL clients have a "safe mode" that may be enabled by default or may be turned on. Safe mode typically disallows you from performing potentially dangerous operations like this. To remove a table, you may need to disable safe mode and then drop it. This provides an extra level of safety so that you cannot absentmindedly screw up the database.

Both `create table` and `drop table` will only work if the table does not exist/does exist respectively. Attempts to (re)create an existing table or to drop a non-existent table will likely result in an error. For convenience, SQL supports conditional create/drop statements. For example,

```
create table if not exists TableName ...
```

and

```
drop table if exists TableName;
```

would only create or remove a table if they did not/did exist respectively. The queries may still issue a warning, but not an error. The typical use case for these conditional statements is when developing and troubleshooting a database creation script. It saves you from having to manually re-create/re-drop tables as you debug and test your script.

Finally, you can also modify an existing table using `alter table`. You may want to do this instead of dropping/recreating the table if some data exists in the table that you do not want to lose. For details, you will need to refer to your particular database documentation, but typically you can use syntax similar to the following.

```
alter table Book add column numPages int;
```

would add a column to the `Book` table as specified.

```
alter table Book drop column dewey;
```

would remove the `dewey` column from the `Book` table as well as deleting all column values associated with records in that table.

```
alter table Book modify column title varchar(100);
```

would change the `title` column in the `Book` table. In this case, it would shorten the field to be at most 100 characters instead of the original 255. This would have the consequence of truncating the column value of any record that exceeded this 100 character limit. Thus, modifying a column may also modify data stored in that column. It also modifies the column to be nullable. If you attempt to modify the data type of a column, the database may attempt to coerce the data types (changing numbers to strings or vice versa) which may lead to unexpected results and data loss. For this reason, it is always best practice to rigorously test and verify your SQL scripts and operations on a test copy before executing them on "production data." Even then, it is always best practice to keep multiple (daily, weekly) database backups or snapshots.

3.2.2 Primary Keys

In our book example (Code Sample 3.1), the first we column we defined was `bookId` and we designated it as a `primary key`. A *primary key* (often referred to as Primary Key (PK)) is an identifier for records. Designating a primary key column ensures that all records in that table have a unique value. Attempts to insert "duplicate" records with the same primary key value will result in an error.

A primary key is important in a table because it provides a unique identity to every record in the table and all future records as well. It is not necessary for a table to have a primary key, but it is generally (very) poor design not to do so. Further, each table can have at most *one* primary key (otherwise it would not be *primary*). However, tables may have more than one *key* and there are several variations and other considerations (see Section 3.2.5).

In general, a *key* is a way for a database to organize and retrieve records and data. Defining a key on a column (or group of columns) means that the database will internally organize records according to that key. This organization is known as *indexing* (which is why a key is equivalently referred to as an *index*). This allows a database to search and retrieve data much more efficiently. This is similar to how binary search can be used to speed up searching an array but only if the array is sorted (or *indexed*).[3] A primary key is just a key with the additional *constraint* that all values are unique.

[3]Typically databases do not simply use arrays to store data, but instead "smart" data structures such as B-trees.

Best Practices

As previously mentioned, it is best practice to have a primary key in every table of your database for sake of consistency, identity, and most importantly to be able to reliably relate records between tables (see Section 3.2.3). In addition, below we identify several other best practices when using primary keys. Though there may be a healthy debate about the specifics, these are generally accepted as good practices.

- Only use a single integer data column. Any data type can be designated as a primary key, but integer data types are best. Floating-point data types are not ideal because of all the problems inherent in using inexact numeric types: floating-point errors, round-off errors, inexact comparisons, etc. Varchar fields are also less than ideal for several reasons. As previously noted, varchar fields are highly dependent on the database, table, and query character sets used which can lead to inconsistencies with case sensitivity, encodings, etc. Using varchar fields can also be less efficient. Comparisons become string comparisons and may require comparing the entire field (dozens or hundreds of characters) instead of one simple number. On the other hand, integer types have none of these problems: they are simple, exact, and efficient.
- Use `auto_increment` for all primary keys. Ensuring that every record has a unique primary key value, called *key management* is a very difficult problem in general. If you were to do this manually, you would immediately face a lot of difficult challenges. Before inserting any new record you'd have to check that the value was not already being used by any other record. Then, you'd have to "reserve" that value and hope that no other program, thread, or transaction came in and used that value before you were able to insert the new record. This is a classic "race condition": two separate programs or threads may find that the key "1234" is unused and think it is safe to use 1234 in their new record. One of these programs may "win" the race and insert their record using this value. This would either end up in a duplicate record or an error in the second program.

  Key management is a very difficult problem that is well-solved by a relational database. Let it do its job and generate and assign key values for you. Using the `auto_increment` option for your primary key field ensures that the database will take care of the key management problem for you.
- Do not allow primary keys to be nullable. Most databases do not allow this anyway, but some do. A nullable primary key is of very limited use. Since primary key values must be unique, at most one record can have a null primary key value anyway. A null primary key essentially means that the the record has no identity which is not only of little value but it is highly questionable if a record with no identity should be stored in a database to begin with. Most databases will implicitly add `not null` to a field designated as a primary key, but explicitly doing so ensures portable code.

- Use a consistent naming scheme. In following with using lower camel casing for fields, it is recommended that you name all of your primary key fields after the table name appending an `Id`. For example, the primary key in the `Book` table was named `bookId`. Following a consistent naming convention eliminates the guess work and the need to continually reference the database schema.
- Use surrogate keys. Database tables often model real-world entities and often these entities have a *natural key* or a natural identity. Books, for example have an ISBN (International Standard Book Number) that uniquely identifies every book published. This would be considered a "natural key" and we may be tempted to use it as a primary key in our database. However, we instead used a *surrogate key* which was a number generated by our database, identifying the ISBN as a separate column value. Other examples may include Social Security Numbers (SSN) for people, NUIDs for UNL students, a SKU (Stock Keeping Unit) for products, etc.

  The reason to use surrogate keys over natural keys for your database is because you have little to no control over the key management of natural keys. If you don't have control over the keys, then you should not allow the keys to control your database. Organizations can (and often do) make mistakes. It is entirely possible to issue the same SSN to two different people or to issue multiple student NUIDs to the same person. These mistakes can be rectified, but imagine having to correct the data in our database. If we've used the natural key throughout several tables then it may be very complicated to update the value. Natural keys may also not be simple integers, leading to the problems with varchar fields identified above.

  Instead, we can use auto-generated surrogate keys and avoid problems that are external to our database. We still have the option of storing natural keys values in separate columns and we can even index and make them unique (see Section 3.2.5), but we can do so without hamstringing our database design with potential problems that we have no control over.

3.2.3 Foreign Keys & Relating Tables

Once again consider the `Book` table in Code Sample 3.1. We identified the author of a book using a single `varchar` field. This can lead to several potential data anomalies, redundancies and other problems. For example, consider the (abridged) `Book` table records in Table 3.2.

All four books were clearly written by the same author, Douglas Adams. However, because we decided to model the author with a single `varchar` field, it introduced the potential for inconsistent data representation. In some records we ordered it first name-last name, in others it was reversed. In one record we included his middle name. These data anomalies were possible because though each book has a unique identity (`bookId`), authors do not. Even if we had consistently used the same name representation for Douglas Adams across all records, we would still have the problem of data redundancy:

bookId	title	author
1	Long Dark Tea-Time of the Soul	Douglas Adams
2	Dirk Gently's Holistic Detective Agency	Douglas Adams
3	The Hitchhiker's Guide to the Galaxy	Douglas Noel Adams
4	The Pirate Planet	Adams, Douglas

Table 3.2: Sample Book Data With Anomalies

the same string would be repeated for every book he wrote. This would be little different from a flat file data representation.

To give author records a unique identity, we need to define another, separate table for authors and then *relate* records between the two tables. To achieve this, we need to use Foreign Key (FK). The process of separating data out into different tables is referred to as *normalization* which we discuss in detail in Section **??**.

A foreign key is a column whose value matches or *refers* to a value in a column in another table. Generally, let A and B be tables. Further, let B have a foreign key column b that references a column a in A. This sets up a parent-child relationship between the two tables. We say that A is the parent table and B is the child table. Several records in table B may have the same foreign key value b, meaning that one parent can have multiple children. This defines a *one-to-many* relationship between the two tables.

To make this more concrete, let's reconsider the relationship between book and author data. For simplicity we'll consider the situation in which one author may write multiple books.[4] In this scenario, there is a potential one-to-many relationship between an author and her books. Equivalently, there is a many-to-one relationship between books and an author. We say that an author table should be the parent table and the book table should be the child table, so we need a foreign key in the book table that refers back to the author that wrote it.

Let's rewrite a simplified `Book` table and introduce a new `Author` table as follows.

```sql
create table Author (
  authorId  int primary key auto_increment not null,
  firstName varchar(255) not null,
  lastName  varchar(255) not null
);
```

```sql
create table Book (
  bookId int primary key auto_increment not null,
  title  varchar(255) not null,
```

[4]Of course one book may have multiple authors, but we'll save that for later.

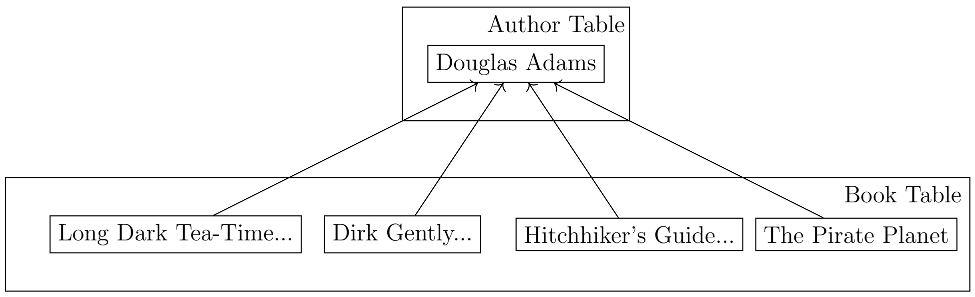


Figure 3.3: A parent-child relationship between the Author and Book tables. Book records in the child table, refer back up to the parent Author record via a foreign key.

```
  authorId int not null,
  foreign key (authorId) references Author(authorId)
);
```

We've included an `authorId` column in the `Book` table that is designated as a foreign key that refers to a unique record in the `Author` table. We're guaranteed that the record is unique because the foreign key references the primary key. Foreign keys are not *required* to refer to a primary key but it is best practice to do so (foreign keys *are* required to refer to a unique field, however). Note the syntax for defining a foreign key: we use the keywords `foreign key` followed by the column in the table designating the foreign key (surrounded by parentheses). We then specify which table and which column in that table to which the foreign key refers. A visualization of this relationship can be found in Figure 3.3.

Once this many-to-one relationship is defined and author data and book data has been separated, we can now see how the potential for data anomalies and redundancy is reduced. Consider several authors and several book records as in the following two tables.

Now each author has a unique record and thus identity (`authorId`). The first name and last name are separated and there is only one instance of each for each author solving our representation problem. In the book table, the author is represented by the primary key value of the corresponding author record. Arguably, there is still some redundancy here, but it is far less. Only a single number is repeated rather than an entire string (or other data that we may want to include relevant to an author).

Though Figure 3.3 depicts a parent-child relationship between the two tables, database tables are usually represented using an Entity Relation (ER) diagram which has several established conventions for visualizing tables and their relationships. An example can be found in Figure 3.4.

Though tools that generate ER diagrams can vary in specifics, in this example each table is represented separately. Primary keys are indicated with a key graphic, foreign

authorId	firstName	lastName
42	Douglas	Adams
8	Terry	Pratchett
17	Neil	Gaiman
23	Cory	Doctorow
97	Octavia	Butler

Table 3.3: Normalized Author Table

bookId	title	authorId
1	Long Dark Tea-Time of the Soul	42
2	Dirk Gently's Holistic Detective Agency	42
3	The Hitchhiker's Guide to the Galaxy	42
4	The Pirate Planet	42
5	Anansi Boys	17
6	Neverwhere	17
7	Coraline	17
8	Color of Magic	8
9	Small Gods	8
10	Kindred	97

Table 3.4: Normalized Author Table

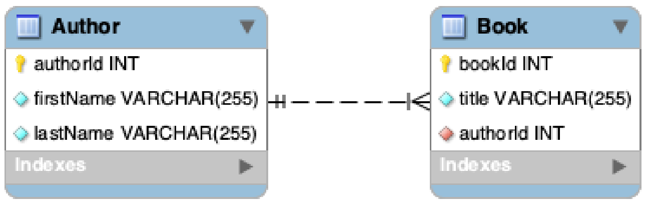


Figure 3.4: An Entity-Relation Diagram for the Author/Book tables indicating a one-to-many relationship. The relation is represented by a connection between the two tables with the vertical lines representing a "one" and the "chicken foot" representing "many".

keys are represented with red diamonds, and the data type of each column is included. A diamond that is filled in indicates that the field is not nullable (likewise, non-filled diamonds indicate nullable fields). Finally, the one-to-many relation is indicated with the line between the tables with the one symbol (two vertical lines) and the many symbol (the chicken foot) on each table respectively.

Finally, take note of the naming conventions we've used: the foreign key column name exactly matches the primary key name to which it refers. Using this consistent naming scheme will reduce the guesswork when we start formulating queries to our database.

Foreign Key Constraints

Consider again the data in Tables 3.4 and 3.3. Though we've defined a one-to-many relationship, that does not necessarily imply that every record has a relationship. For example, there is a record for the author Cory Doctorow, but he has no associated book records. Similarly, Octavia Butler only has one associated book record. Nevertheless, there is still a *potential* one-to-many relationship even if we do not have multiple records stored in the book table.

A foreign key represents a *constraint* in the relation of data that must always be satisfied. For example, suppose we were to insert a new book record for the novel *Station Eleven* into the book table but without a corresponding author (Emily St. John Mandel) record in the author table. This has several immediate problems. First, the foreign key `authorId` in the book table is not nullable, so inserting such a record would fail. From the parent-child relationship perspective, we would have a child record (book) without a parent (author), giving us an "orphan" record. This violates the relationship that we've defined between the two tables. For a child record to exist, the parent record it refers to must exist *first* so that we have a valid primary key value for our foreign key to refer to. If we were to delete records (say we wanted to remove the Douglas Adams record from the `Author` table) we must do so in reverse. In order to remove a parent record, all of its children records that refer to it must be deleted first.

This order also applies when creating tables. Since the `Book` table refers to the `Author` table, we must create the `Author` table first. Likewise, if we were to drop these tables, we must do so in reverse order: the child table must be removed before the parent table.

Attempts to manipulate data or tables in the wrong order will result in database errors.

3.2.4 Many-To-Many Relations

Let us consider extending our two table database further. It is entirely possible for one book to have multiple authors. For example, *Good Omens* was written by both Terry Pratchett and Neil Gaiman. Our current model only allows for a one-to-many relationship between authors and books. What we really want is a many-to-many relationship. This

```sql
create table Author (
  authorId  int primary key auto_increment not null,
  firstName varchar(255) not null,
  lastName  varchar(255) not null
);

create table Book (
  bookId int primary key auto_increment not null,
  title varchar(255) not null
);

create table AuthorBook (
  authorBookId int primary key auto_increment not null,
  authorId int not null,
  bookId int not null,
  foreign key (authorId) references Author(authorId),
  foreign key (bookId) references Book(bookId)
);
```

Code Sample 3.2: A many-to-many Author/Book table design

would enable one author to be associated with several books and one book to be written by several authors.

A one-to-many relation from table A to table B is the same thing as a many-to-one relation from table B to table A. To enable a many-to-many relationship we define a third table called a *join table* between these two tables. We then define a one-to-many relation from table A to the join table and a one-to-many relation from table B to the join table, resulting in a many-to-many relation between tables A and B. We do this by defining *two* foreign keys in the join table. From the parent-child relation perspective, the join table is a child of both tables (and so has two parents). This modified design is presented as Code Sample 3.2 and the resulting ER diagram is depicted in Figure 3.5.

As with a simple one-to-many relationship, all of the same rules and constraints apply. In this case, both an author and a book record must exist before you can associated them with each other by inserting a record in the join table. To delete data in either table, this association record in the join table must be removed first. We've named our join table `AuthorBook` after the two tables that it "joins" together. Alternatively, we could have named this table after the relation it represents, `WrittenBy` for example.

Data for the books from a previous example using this new many-to-many relation model is depicted in Figure 3.6. We can now model the fact that *Good Omens* was written by both Terry Pratchett and Neil Gaiman. Entries 11 and 12 in the `AuthorBook` join table have the same `bookId` corresponding to *Good Omens* but different `authorId` values.

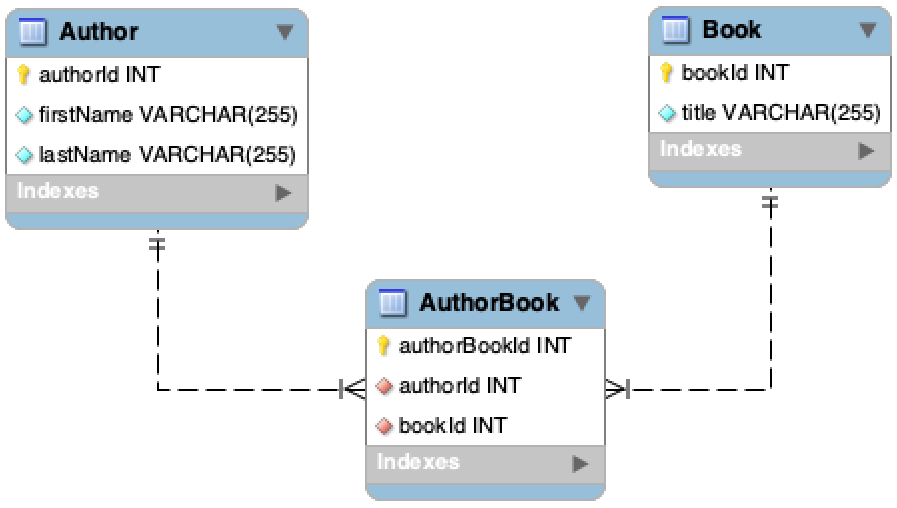


Figure 3.5: A many-to-many relationship using a join table.

The remaining entries remain similar to our old design, modeling that each of the other book records only have one author each. Another consequence of this design change is that we can now have book records without having to have an author record, similar to how we could have an author with no book records before. In particular, *Confederacy of Dunces* has no join record indicating its author.

3.2.5 Other Keys

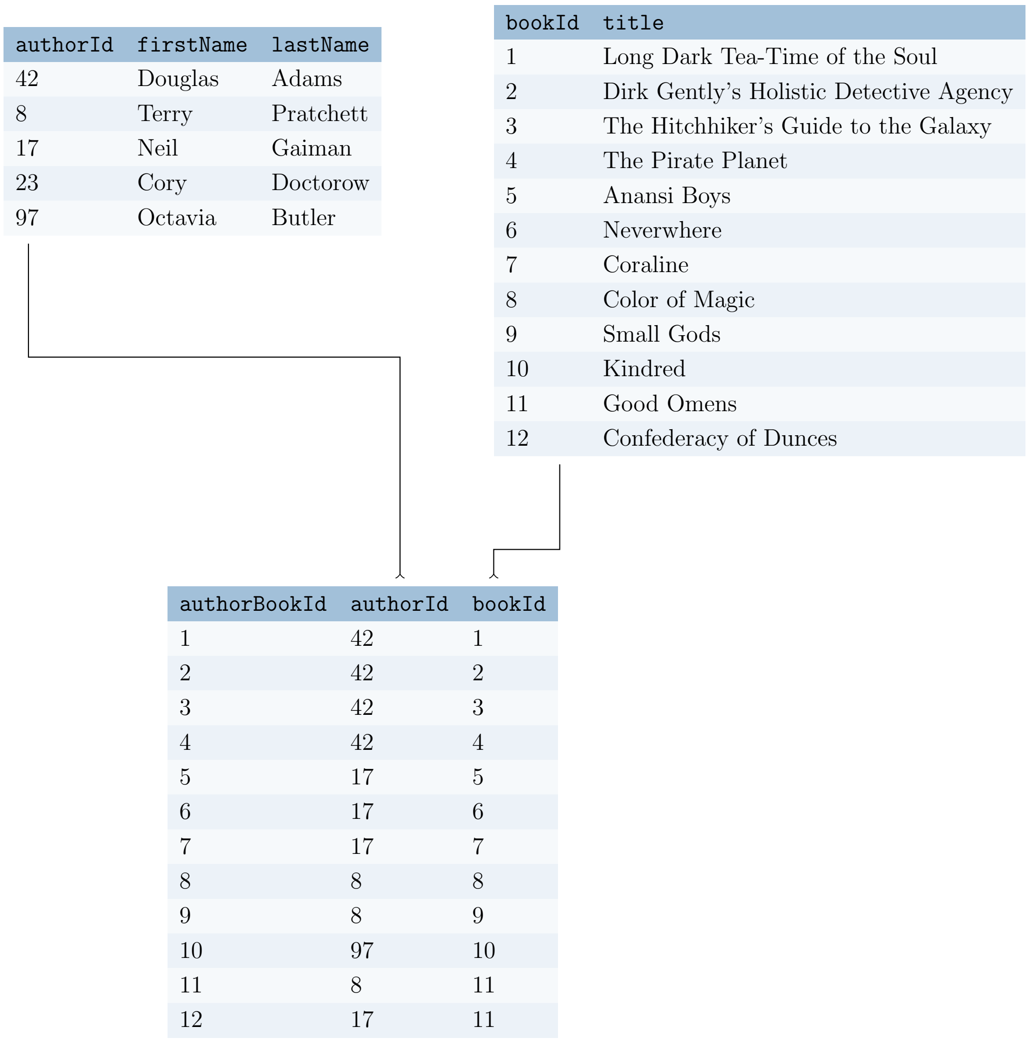
Primary keys and foreign keys are two specific types of keys. You can define a more general key on any column in any table of even a combination of columns in any table. Equivalently, a key is referred to as an *index* which makes the database maintain an ordering on the column values for fast search and retrieval.

Suppose that you'll be making frequent search queries to the author table based on (say) the last name. For example, you may enable users to search for authors whose last name begins with "T" or to perform a (partial) keyword search. Placing a key or index on the `lastName` column can greatly increase performance by several orders of magnitude. To do this, you can use the following syntax:

```
key (columnName)
```

Or, equivalently, `index (columnName)` (`key` and `index` are synonymous) and the database will keep the data (internally) organized according to values in that column.

Keys can also be defined for a *combination* of columns. Such keys are called *composite* keys because they are composed of more than one column. To define such a key you can simply provide a comma delimited list of column values. For example:



authorId	firstName	lastName
42	Douglas	Adams
8	Terry	Pratchett
17	Neil	Gaiman
23	Cory	Doctorow
97	Octavia	Butler

bookId	title
1	Long Dark Tea-Time of the Soul
2	Dirk Gently's Holistic Detective Agency
3	The Hitchhiker's Guide to the Galaxy
4	The Pirate Planet
5	Anansi Boys
6	Neverwhere
7	Coraline
8	Color of Magic
9	Small Gods
10	Kindred
11	Good Omens
12	Confederacy of Dunces

authorBookId	authorId	bookId
1	42	1
2	42	2
3	42	3
4	42	4
5	17	5
6	17	6
7	17	7
8	8	8
9	8	9
10	97	10
11	8	11
12	17	11

Figure 3.6: Normalized Author/Book table data with a many-to-many relation.

```
key (columnA,columnB)
```

Finally, you can make keys or composite keys unique without forcing them to be primary keys (primary keys are unique by definition). Recall that we made the design decision to use surrogate keys in our book table instead of the "natural" ISBN key. However, we may want to ensure that all ISBN values are unique so that we don't have two book entries with the same ISBN. To do this, we can use the following syntax:

```
unique key (columnName)
```

We can also combine these two concepts and put a unique composite key on a combination of columns; for example:

```
unique key (columnA,columnB)
```

Let's apply these features to our author-book database. The final version can be found in Code Sample 3.3. Observe that we've placed a key (index) on the `lastName` column for performance. We also placed a unique key on the `isbn` column in the `Book` table to prevent duplicate entries.

We also placed a unique composite key on the combination of foreign keys in the `AuthorBook` join table. This creates a constraint so that only one book/author join record is possible. Without this constraint we might be able to insert multiple author records for the same book:

authorBookId	authorId	bookId
1	42	1
1	42	1
1	42	1
1	42	1

Four records modeling the fact that Douglas Adams wrote *Long Dark Tea-Time of the Soul* are redundant and not necessary.

3.3 Structured Query Language

A relational database provide a means for storing, representing, and modeling data. We still need a way to interact with and operate on that data. In order to interact with data SQL provides queries for basic Create-Retrieve-Update-Destroy (CRUD). These four operations are sufficient to perform any task we may want to on our data. Each operation is supported by the query statement keywords `insert`, `select`, `update`, and `delete` respectively (though ISUD is a less compelling acronym).

To illustrate examples for each of these queries, we'll frequently refer to the Author-Book database as presented in Figure 3.7.

```sql
create table Author (
  authorId  int primary key auto_increment not null,
  firstName varchar(255) not null,
  lastName  varchar(255) not null,
  key (lastName)
);

create table Book (
  bookId int primary key auto_increment not null,
  title varchar(255) not null,
  isbn varchar(100) not null,
  numCopies int not null default 0,
  key (title),
  unique key (isbn)
);

create table AuthorBook (
  authorBookId int primary key auto_increment not null,
  authorId int not null,
  bookId int not null,
  foreign key (authorId) references Author(authorId),
  foreign key (bookId) references Book(bookId),
  unique key (authorId,bookId)
);
```

Code Sample 3.3: Full Author/Book table database

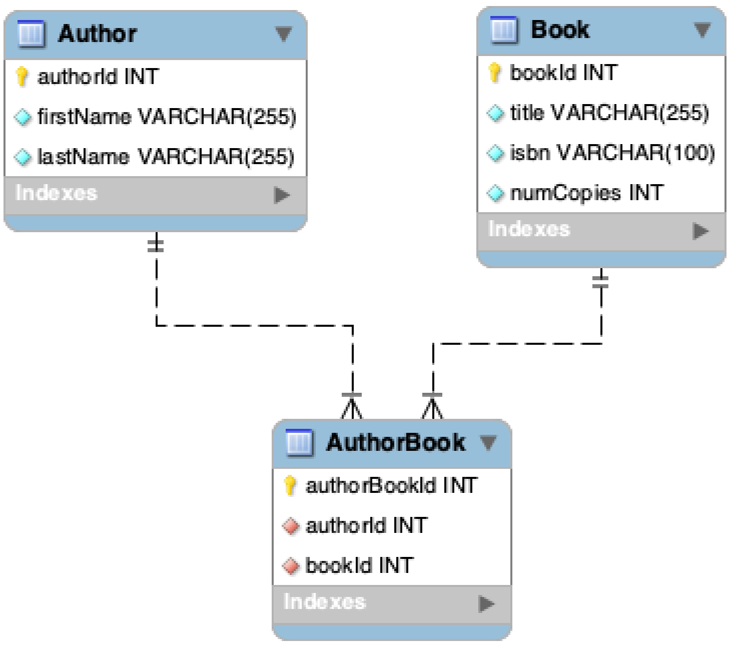


Figure 3.7: Final Author-Book Database

3.3.1 Creating Data

Before we can process data, we have to actually create some to work with. To do this, we can `insert` new records into a table. The general syntax for doing this is:

```
insert into TableName (columnA, columnB, ...) values (valueA, valueB, ...);
```

We identify the table we wish to insert a record into and then provide a comma-delimited list of column names we wish to specify values for. Another comma-delimited list of values (string literals or numbers for example) which has a one-to-one correspondence with the columns identified, thus the order in which we specify columns matters. These are often referred to as *tuples*: a group of data delimited by commas and enclosed in parentheses which correspond to a single record in a table.

As a concrete example, let's insert a new record into our `Author` table for Isaac Asimov:

```
insert into Author (firstName, lastName) values ("Isaac", "Asimov");
```

The order of the columns doesn't matter, but the correspondence with the values does. Equivalently we could have written:

```
insert into Author (lastName, firstName) values ("Asimov", "Isaac");
```

and it would have resulted in the same record. Recall that neither field was nullable and no default values were specified, so if we omitted one of them, say:

```
insert into Author (lastName) values ("Asimov");
```

it may result in an error or a warning and may result in either no record being inserted or a system-wide default value assigned to the `firstName` field depending on the database and configuration.

In these examples we omitted a value for the primary key `authorId`. As a result the database will automatically generate and assign a unique value for this field for us. If, instead we wished to specify a value for the primary key we could do so:

```
insert into Author (authorId, firstName, lastName) values
  (1920, "Isaac", "Asimov");
```

Recall, however, that key management is best left to the database. Nevertheless there are use cases for (manually) defining your own keys. For example, when you insert test data and need to make associations (foreign keys) between tables. Providing your own key values makes this much easier. For example, let's also insert a record for his novel *Pebble in the Sky* (omitting the ISBN for brevity):

```
insert into Book (title, isbn) values ("Pebble in the Sky", "0765319136");
```

Now we want to associate these two records with each other. To do so, we need to insert a record into the `AuthorBook` table which requires both the `authorId` and `bookId` as foreign keys. The problem is we don't know what the `bookId` is as it was generated by the database. The more verbose (and fragile) solution would be to use `select` statements (see Section 3.3.2) in *nested queries* which might look something like the following.

```
insert into AuthorBook (authorId, bookId) values (
  (select authorId from Author where lastName = "Asimov"),
  (select bookId from Book where title = "Pebble in the Sky")
);
```

This solution essentially pulls the generated key values from previously inserted records. It is fragile because it *assumes* the data has successfully been created (and still exists) and that it is *unique*. That is, it assumes there is one and only one author whose last name is Asimov. If those assumptions do not hold, this query will fail.

Instead, we could provide our own key values as follows.

```
insert into Author (authorId, firstName, lastName) values
  (1920, "Isaac", "Asimov");

insert into Book (bookId, title, isbn) values
  (123, "Pebble in the Sky", "0765319136");

insert into AuthorBook (authorId, bookId) values
```

```
  (1920, 123);
```

This still runs the risk that the key values are not already being used by other records in the database which is why providing your own key values should be restricted to test data or to initialize a database with some “seed” data.

If you want to insert multiple records with a single query you can provide a comma-delimited list of tuples:

```
insert into Author (firstName, lastName) values
  ("Charles", "Dickens"),
  ("Jane", "Austen"),
  ("Frank", "Herbert"),
  ("Robert", "Heinlein");
```

In some of the above examples you’ll note that we saved on horizontal space by breaking the query to the next line and using indentation. As with most programming languages, whitespace does not matter and so breaking long queries up into multiple lines is good style.

Recall that string literals can be denoted with either single quotes or double quotes. Good style would have you choose one and use it consistently, not mixing the two ways. Either way you choose, if you need to use a single or double quote in your string literals you can escape them with a backslash.

```
insert into Author (firstName, lastName) values
  ("Stan \"The Man\"", "Lee");

insert into Book (title,isbn) values
  ('Foundation\'s Edge', '0586058397');
```

3.3.2 Retrieving Data

Once a table has data in it, of course you’ll want to retrieve and process it. To retrieve data you use a `select` statement which has the following syntax.

`select columnA, columnB, ... from TableName;`

This will retrieve the specified column values for *every* record in the table you are querying.[5] The collection of records this query returns is generally referred to as a

[5]Depending on your setup, the database may use *pagination* so that a limited number of records are returned and the client will need to ask for the “next page” of results.

result set. Specifying a comma delimited list allows you to retrieve a subset of specific column values that you are interested in. Alternatively, you can use the *wildcard* or "star" operator to retrieve every column value:

`select * from TableName;`

For example, to get only the last name and first name from the `Author` table we could execute the following query:

`select lastName, firstName from Author;`

The order of the columns has been switched in this query from how we defined the table (though the order of columns is generally irrelevant). To retrieve all 5 columns from the `Book` table we could use:

`select * from Book;`

Using the wildcard operator is fine when working directly with a database via an SQL client or when debugging/developing. Ultimately, you'll likely want to connect to your database *programmatically*. When connecting to a database programmatically, using the wildcard is highly discouraged since it forces a lot of data to be returned in the result set that you may not necessarily use. Since this data is typically being transmitted over a network connection, sending unnecessary data can have a negative impact on performance.

Aliases

When you query a table you can rename the column names in the result set by specifying an *alias*. This doesn't change the column names in the table, only in the result set. The syntax for doing this is to use the keyword `as` followed by the alias. In the following example, we query the `Book` database, aliasing the first two columns but not the third.

```
select title as bookTitle,
       isbn as ISBN,
       numCopies
       from Book;
```

As we'll see later, you can also alias table names for convenience and to simplify complicated queries. In many databases the `as` keyword is optional. However, most style guides have you explicitly use the keyword for column aliases while omitting it for table aliases and we'll also follow this convention.

Where Clause

The the previous `select` queries were not limited or conditioned on anything and so the result set contained *every* record in the table being queried. If you want to limit the results of a `select` query, you need to qualify it with a `where` clause and provide some *condition* on the data.

For example, suppose that we want to query all books written by Isaac Asimov (whose `authorId` is 1920). To do this we append a `where` clause to the end of our query with a boolean condition.

```
select * from Book where authorId = 1920;
```

The result set of this query will only contain book records that satisfy the `authorId = 1920` condition. Note that unlike most programming languages, the equality operator is only a single quote. However, you do use the traditional `!=` for the inequality operator. So for example:

```
select * from Book where authorId != 1920;
```

would result in *every* book record *not* written by Isaac Asimov. Numerical *and* varchar field values can use both of these operators as well as the traditional comparison operators, `<` (strictly less than), `<=` (less than or equal to), `>` (strictly greater than) and `>=` (greater than or equal to). When applied to varchar fields, the comparison will use lexicographic ordering, for example,

```
select * from Book where title >= "M";
```

will result in all books whose title starts with any letter M or later in the alphabet. Note that lexicographic ordering is not the same thing as dictionary ordering so this query would also include any book whose title begins with any lower case letter (since lower case letters follow upper case letters in the ASCII text table).

The equality and inequality operators, however, cannot be used to test for nullity. Instead, you need to use the keywords `is null` or `is not null`. For example:

```
select * from Book where numCopies is null;
```

would return all book records where the `numCopies` field is null.

As with other programming languages you can also combine conditions with logical connectives using the keywords `and` and `or` as well as using parentheses and negations to modify your conditional statements. Some examples:

```
select * from Book where
  numCopies > 10 and
  (title <= "D" or title >= "Q");

-- the following are equivalent
```

```
select * from Book where authorId != 1920;
select * from Book where !(authorId = 1920);
```

Distinct Clause

A select query may return many duplicate entries. If we want to limit our results to only distinct values we can use a `distinct` clause after the `select` keyword to exclude duplicates. For example, if we queried all the last names of authors, there may be many instances of the last name "Smith" as it is quite common. If we used the following query

```
select distinct lastName from Author;
```

then we would only get (at most) one instance of each distinct last name in the result set. This clause will be particularly useful when we process more complex queries in Section 3.3.5.

Like Operator

Using equality and comparison operators for varchar columns only enables us to formulate queries for *ranges* of values. Often we want to make partial string comparisons. For example, retrieving all strings that begin with a certain letter or strings that contain a certain substring.

To perform partial string matching searches you can use the `like` clause in conjunction with the string wildcard symbol, `%` (which is not the same as the column wildcard or "star" operator). For example, to search for all authors whose last name begins with "A" we would write

```
select * from Author where lastName like "A%";
```

This string wildcard matches *any* string including the empty string, so this query would also match any author record whose last name was *only* a single "A" character. You can also define *groups* of characters similar to regular expressions. The following query matches all authors whose last name begins with an "A" a "B" or a "C":

```
select * from Author where lastName like "[A-C]%";
```

or:

```
select * from Author where lastName like "[ALZ]%";
```

would match any last name beginning with an "A" an "L" or a "Z". You can *invert* the group using an exclamation point:

```
select * from Author where lastName like "[!ALZ]%";
```

would match all records that *don't* begin with an A, L or Z. This is necessary as there is no `not like` clause.

You can place the wildcard anywhere you like so that

`select * from Author where lastName like "%e";`

matches last names that end in “e” and

`select * from Author where lastName like "%e%";`

match last names that *contain* an “e”. You can expand the string value so that you can perform substring searches.

`select * from Author where lastName like "%the%";`

will match any record whose last name contains the *entire* character sequence “the”.

The string wildcard, `%` matches any number of characters (including none). To match *exactly* one character you can use the underscore wildcard which matches any single character.

`select * from Author where lastName like "_s%";`

will match all last names that begin with any letter but that are immediately followed by a lowercase “s” (followed by any other string). You can use any number of combinations of character wildcards, string wildcards, and string literals to do partial string matching.

In Operator

If you have a lot of values that you want to match you could write a very large query with a lot of `or` conditions:

```
select * from Author where
  authorId = 1920 or
  authorId = 3213 or
  authorId = 5345 or
  authorId = 4324;
```

This can be written in a more concise manner using the `in` operator and specifying a comma delimited list of values:

```
select * from Author where
  authorId in (1920, 3213, 5345, 4324);
```

which works equally well for varchar fields.

```
select * from Author where
  lastName in ("Asimov", "Clarke", "Ellison");
```

The `in` operator is especially useful when you combine it with nested queries. For example:

```
select * from Author where
  authorId in (select authorId from AuthorBook);
```

would give us all authors who have at least one corresponding record in the `AuthorBook` table. As we'll see later, this is better done with a `join` query, but you can also negate the operator to get all author records that do *not* have any corresponding book records:

```
select * from Author where
  authorId not in (select authorId from AuthorBook);
```

Order By Clause

In general the order of records in a result set has no meaning. The database stores and returns results in whatever manner is the most efficient. If you want records to be return in a specific order you can do so by appending an `order by` clause and specifying which column or columns to order the data by in either ascending (`asc`) or descending order (`desc`).

Some examples:

```
-- order authors by last name in ascending order (default)
select * from Author order by lastName;

-- order authors by last name in ascending order (explicitly)
select * from Author order by lastName asc;

-- order authors by last name in descending order (Z to A)
select * from Author order by lastName desc;

-- order authors by last name then by first name if they
-- have the same last name
select * from Author order by lastName, firstName;

-- order books by most number of copies first (descending),
-- then by title (ascending)
select * from Book order by numCopies desc, title;
```

3.3.3 Updating Data

Records that already exist in a database can be updated using an (surprise!) `update` statement. The general syntax is as follows.

```
update TableName set columnA = valueA, columnB = valueB, ... where [expression]
```

The `where` *clause* part of the query is extremely important as it will limit the effect of our statement.

As an example, suppose that we messed up the insertion of Isaac Asimov by mixing up his first/last name:

```
insert into Author (authorId, lastName, firstName) values
  (1920, "Isaac", "Asimov");
```

To update the record we would so something like the following:

```
update Author
  set firstName = "Isaac", lastName = "Asimov"
  where authorId = 1920;
```

Of course we could have omitted the `where` clause and executed a statement like:

```
update Author set lastName = "Asimov";
```

which would have some extreme unintended consequences. If this query were to successfully executed, *every* author record's last name would now be "Asimov". Most (sane) SQL clients won't let you do this by default. Recall that we mentioned *safe mode* in which certain dangerous queries, queries that alter data without restrictions are not allowed. If you really do intend to alter every record, you can shut off safe mode and proceed at your own peril. The `where` clause in the correct example limits the effects of the `update` statement to only one record. In particular, the record whose `authorId` is 1920. In general, any subset of column values can be changed with an `update` query and the order does not matter.

3.3.4 Destroying Data

Removing data from a table can be done with a `delete` query. The general syntax is as follows.

```
delete from TableName where [expression]
```

As with the `update` statement, the `where` clause limits the effect of a `delete` statement. Without a `where` clause, it would effectively remove every record in the table. However, it would not delete the table (that is done using `drop table`, see Section 3.2.1).

If we wanted to delete Isaac Asimov from our `Author` table we would use the following.

```
delete from Author where authorId = 1920;
```

Of course if there were any references to this record in the `AuthorBook` table, those would need to be deleted first as the foreign key constraints would prevent us from deleting a parent record that still had child records.

3.3.5 Processing Data

SQL not only allows us to perform basic CRUD, but it has many powerful queries that allow you to process and aggregate data directly in the database. In this section we cover only a small selection of these capabilities.

Aggregates

Aggregate functions are functions that allow you to summarize data such as summing values, taking the average, finding the minimum/maximum values etc.

The `count` function can be used to determine how many records would be in the result set of your query. For example,

```
select count(*) from Book;
```

would return a count of the number of book records. The result of this query is only a single record: the number of book records. It does not include the actual book records in the result. Further, the "column" is not an actual column in the database. In this case the column "name" is `count(*)`. We can use an alias to give this column a more sensible name:

```
select count(*) as numberOfBooks from Book;
```

You can also qualify aggregate functions with a `where` clause.

```
select count(*) as numberMissingBooks from Book where numCopies = 0;
```

The `sum` function can be used to sum values in a particular column. Some examples:

```
-- find the total number of copies of all books
select sum(numCopies) as totalNumberOfBooks from Book;

-- find the total number of copies of all books
select sum(numCopies) as numberOfCommonBooks
  from Book
  where numCopies >= 10;
```

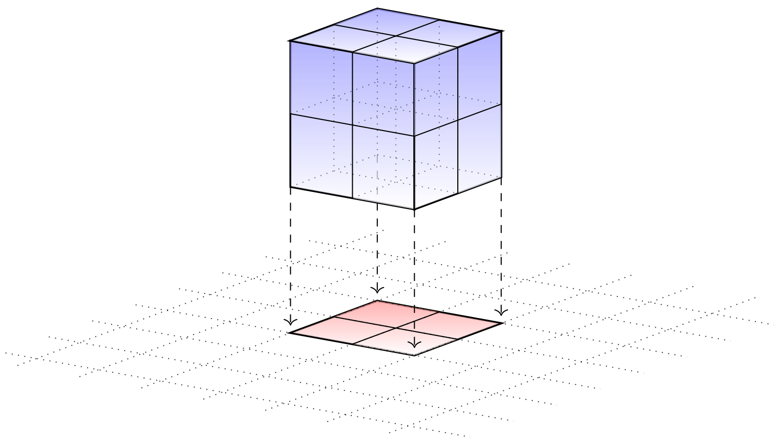


Figure 3.8: A 3-D cube projected onto a 2-D surface producing a square.

You can also find the average using `avg`:

```
select avg(numCopies) as averageNumCopies from Book;
```

Finally, you can use `min` and `max` to find the minimum and maximum values respectively.

```
select min(numCopies) from Book;
select min(numCopies) from Book where numCopies > 0;

select max(numCopies) from Book;
```

Both of these aggregate functions work for varchar fields as well:

```
select max(title) as lastBookTitle from Book;
```

would return the lexicographically maximum (last in alphabetic order) title of a book.

Projections

In mathematics you can take an n-dimensional object and remove one of the dimensions to *project* that object into an $(n-1)$ dimensional space. For example, you can take a 3-D cube and project it down onto a 2-D plane. This is essentially removing one of the coordinates, mapping points $(x, y, z) \to (x, y)$. Depending on the orientation of the cube you may get a variety of shapes on the 2-D surface. One possibility results in a square as depicted in Figure 3.8.

Recall that a data record (a single row) is called a *tuple* which is essentially the same thing as a point (in fact, we even use the same notation, (x, y, z) or `(author, title, numCopies)`). With respect to data, we can perform a similar projection, removing one of the "dimensions" by removing one (or more) of the columns. In addition, rather than simply "loosing" information as with the cube-to-square example, we can use this project to produce *more* aggregate information. The way we do this is by *grouping* collections of data and collapsing them into a single record. In SQL we use the `group by` clause.

Before we get into the specific syntax, let's use a motivating example to understand how it works. Suppose we have a (simplified) book table with only three columns: the book title, author name, and number of copies of the book in our library.

The table may look something like this.

title	author	numCopies
Naked and the Dead	Norman Mailer	10
Dirk Gently's Holistic Detective Agency	Douglas Adams	4
Barbary Shore	Norman Mailer	3
The Hitchhiker's Guide to the Galaxy	Douglas Adams	2
The Long Dark Tea-Time of the Soul	Douglas Adams	1
Kindred	Octavia Butler	5

Our goal is to produce a report of how many total copies of books by each author we have in our collection. We have 3 books by Douglas Adams with $4 + 2 + 1 = 7$ copies. As we saw in the previous section we can easily get a *total* number of copies using the `sum` aggregate function, but that would only result in one number. In this example, we want a report of three numbers: the total for each author. Conceptually, we need to *group* the records in this table *by* the author. Giving us the following result.

title	author	numCopies
Naked and the Dead	Norman Mailer	10
Barbary Shore	Norman Mailer	3
Dirk Gently's Holistic Detective Agency	Douglas Adams	4
The Hitchhiker's Guide to the Galaxy	Douglas Adams	2
The Long Dark Tea-Time of the Soul	Douglas Adams	1
Kindred	Octavia Butler	5

After grouping the data we can then project it down by eliminating one of the irrelevant columns (the book titles) and summing up the `numCopies` values for each group. That results in the following:

author	totalNumCopies
Norman Mailer	13
Douglas Adams	7
Octavia Butler	5

To perform this data projection in SQL, you can use the `group by` clause. In the example above we could have written the following.

```sql
select author, sum(numCopies) as totalNumCopies
  from Book
  group by author;
```

The `group by` clause grouped our data for us, and the `sum` aggregate was responsible for projecting the data and producing the new value. Without the `sum` aggregate we would only get the collapsed data records (likely with the first value in the group for `numCopies` instead of the projected sum).

Because the `totalNumCopies` is not a value in the original table, we cannot use a `where` clause if we wanted to further winnow this data. Instead, SQL provides another keyword to restrict the results *after* a data projection using a `having` clause. For example, we could further restrict the result in the example above as follows.

```sql
select author, sum(numCopies) as totalNumCopies
  from Book
  group by author
  having totalNumCopies > 5;
```

This would end up excluding the Octavia Butler record:

`author`	`totalNumCopies`
Norman Mailer	13
Douglas Adams	7

Joins

In our previous example we used a simplified table to illustrate data projection. However, in our author-book database, the data we're interested in is normalized and separated out into 3 different tables. In order to bring data together from 3 separate tables, we need to *join* them together (which is why we called the table between them a "join" table). To do this, we use a `join` clause.

Conceptually, a join corresponds to a *cartesian product* of tuples in mathematics combined with a condition by which tuples are joined. Without a condition, this corresponds to a regular cartesian product:

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

For simplicity, suppose that $A = \{10, 20\}$ and $B = \{5, 15, 25\}$ are simple sets of single values (instead of rows). The cartesian product would consist of the following pairs:

$$(10, 5), (10, 15), (10, 25), (20, 5), (20, 15), (20, 25)$$

Each element in the first set is paired with each element in the second set. In general, if the sets are of *cardinality* (size) n and m respectively, then there would be $n \cdot m$ total elements in the cartesian product (above, we have $2 \cdot 3 = 6$ pairs in the result).

We can limit a cartesian product by adding a condition by which a pair may be included. This corresponds to a traditional *relation* on sets in mathematics:

$$R \subseteq A \times B = \{(a, b) \mid a \in A, b \in B, P(a, b)\}$$

where P is some predicate that is either true or false depending on the elements a, b. In fact, this is why we call them *relational* databases. From the example above, suppose we put a condition that the pair (a, b) be included only if $a < b$. This would result in the smaller result

$$(10, 15), (10, 25), (20, 25)$$

excluding the other 3 elements.

With respect to a database, a `join` clause combines records (rows) in one table (table A) with records in another table (table B) usually with some *condition* that limits the combinations and defines *how* rows in each table get paired up. Further, we can and will perform joins with multiple tables just as you can with cartesian products and multiple sets (for example: $A \times B \times C$ would give the obvious[6] result).

The result of a join is still a "table" in that it has columns and rows as the result set, but it is not a table that exists in the database. In general you can join tables together using any criteria you want, but by design they are usually joined together using foreign keys. There are also several types of joins (left/right, inner/outer, cross joins, full outer joins, etc.) but to keep things simple we'll focus on two of the most useful: inner joins and left outer joins.

To illustrate how this works, let's consider the following (simplified) `Author`, `Book`, and `AuthorBook` tables with the following data. The SQL used to generate this data is present in Appendix 1 if you want to follow along with your own database.

[6] If not obvious, then $A \times B \times C = \{(a, b, c) \mid a \in A, b \in B, c \in C\}$

authorId	firstName	lastName
1	Norman	Mailer
2	Douglas	Adams
3	Octavia	Butler
4	Cory	Doctorow

bookId	title	numCopiess
1	Naked and the Dead	10
2	Dirk Gently's Holistic Detective Agency	4
3	The Hitchhiker's Guide to the Galaxy	2
4	The Long Dark Tea-Time of the Soul	1
5	Barbary Shore	3
6	Kindred	5

authorBookId	authorId	bookId
1	1	1
2	2	2
3	2	3
4	2	4
5	1	5
6	3	6

An `inner join` combines records from table *A* with records from table *B* based on some criteria. This is one of the most common joins and so the `inner` keyword is optional and we'll use the abbreviated `join`. For example, let's join records in our `Author` table with the join table `AuthorBook`. By our design, records in these two tables are related by the foreign key `authorId`. We could do this using the following query.

```
select * from Author a
  join AuthorBook ab
  on a.authorId = ab.authorId;
```

A few things of note about this query.

- We join *from* the `Author` table *to* the `AuthorBook` table
- For both tables, we provide an alias (without using the `as` keyword) so that we can refer to them in the `on` clause

- The `on` keyword is used to provide the condition by which records should be joined. Without this, *each* record in the first table would be paired with *every* record in the second table.

The result of this query would be a table that would look like the following.

authorId	firstName	lastName	authorBookId	authorId	bookId
1	Norman	Mailer	1	1	1
2	Douglas	Adams	2	2	2
2	Douglas	Adams	3	2	3
2	Douglas	Adams	4	2	4
1	Norman	Mailer	5	1	5
3	Octavia	Butler	6	3	6

The first three columns correspond to records in the `Author` table and each record is paired with its referencing record in the `AuthorBook` table (the last three columns). If we were generating a report there is some redundant information that we don't necessarily need (the `authorId` is repeated for example and the `bookAuthorId` is not useful for the report). We can use this as an opportunity to limit our query to only the relevant columns.

```sql
select a.firstName, a.lastName, ab.bookId from Author a
  join AuthorBook ab
  on a.authorId = ab.authorId;
```

In the above query we used the table aliases when specifying columns. Though it is not necessary in this *particular* example, it would be necessary to disambiguate the columns if we had repeated column names. The result would be the following simpler report:

firstName	lastName	bookId
Norman	Mailer	1
Douglas	Adams	2
Douglas	Adams	3
Douglas	Adams	4
Norman	Mailer	5
Octavia	Butler	6

This only gives us the internal `bookId` primary key value which is of little help in a report intended for human consumption. To get the book title into this report, we need to join to yet another table, the `Book` table.

firstName	lastName	title
Norman	Mailer	Naked and the Dead
Douglas	Adams	Dirk Gently's Holistic Detective Agency
Douglas	Adams	The Hitchhiker's Guide to the Galaxy
Douglas	Adams	The Long Dark Tea-Time of the Soul
Norman	Mailer	Barbary Shore
Octavia	Butler	Kindred

Table 3.5: Final results for the join of our author-book database.

```sql
select a.firstName, a.lastName, b.title from Author a
  join AuthorBook ab on a.authorId = ab.authorId
  join Book b on ab.bookId = b.bookId;
```

which gives us the final result:

Other Joins

In the example we joined tables according to their design by using foreign key values. If we did not use any conditions (no `on` clause) then we would have gotten a complete "cross join". For example:

```sql
select * from Author join AuthorBook;
```

would have produced a result similar to the following.

Which is essentially a full cartesian product. Since there were 4 records in the `Author` table (table A) and 6 records in the `AuthorBook` table (table B), there are $4 \cdot 6 = 24$ records in the "cross join" of the two tables. However, the interpretation of the data is meaningless, not every author authored every book. This illustrates the importance of both good database design and sensible queries.

Another mistake we could make is to join the two tables along mismatched column values. For example, in our design there is no direct relationship between the `authorId` (author primary key) and the `authorBookId` (the join table primary key) which are both independently generated by the database and have no real connection with each other. Nevertheless, if we tried to join the two tables together using these column values as in the following query,

authorId	firstName	lastName	authorBookId	authorId	bookId
1	Norman	Mailer	1	1	1
1	Norman	Mailer	2	2	2
1	Norman	Mailer	3	2	3
1	Norman	Mailer	4	2	4
1	Norman	Mailer	5	1	5
1	Norman	Mailer	6	3	6
2	Douglas	Adams	1	1	1
2	Douglas	Adams	2	2	2
2	Douglas	Adams	3	2	3
2	Douglas	Adams	4	2	4
2	Douglas	Adams	5	1	5
2	Douglas	Adams	6	3	6
3	Octavia	Butler	1	1	1
3	Octavia	Butler	2	2	2
3	Octavia	Butler	3	2	3
3	Octavia	Butler	4	2	4
3	Octavia	Butler	5	1	5
3	Octavia	Butler	6	3	6
4	Cory	Doctorow	1	1	1
4	Cory	Doctorow	2	2	2
4	Cory	Doctorow	3	2	3
4	Cory	Doctorow	4	2	4
4	Cory	Doctorow	5	1	5
4	Cory	Doctorow	6	3	6

```
select * from Author a
  join AuthorBook ab on a.authorId = ab.authorBookId;
```

then we would get something like the following result:

authorId	firstName	lastName	authorBookId	authorId	bookId
1	Norman	Mailer	1	1	1
2	Douglas	Adams	2	2	2
3	Octavia	Butler	3	2	3
4	Cory	Doctorow	4	2	4

If you examine the values in the `Book` table, the four results have nothing to do with the results. The first two records match (coincidence), but the third and fourth record do not: Octavia Butler did not write *Hitchhiker's Guide...* and there are no book records at all for Cory Doctorow. Again, it is necessary to write queries that conform to the intended design of your database.

Left Joins

The second type of join that we'll cover is a `left outer join` or simply just `left join`. Take another closer look at the "final" result of a regular `join` clause on our database as presented in Table 3.5. A close examination will show that Cory Doctorow has a record in the `Author` table but no corresponding book records. As such, his record did not appear in the results.

With a regular `join` clause, records in table A that do not have a matching record in table B (according to the criteria that you specify in the `on` clause) are *not* included in the result set. A `left join`, however, will *preserve* records in table A even if they do not have a matching record in table B. For example, the following modified `left join` query:

```
select a.firstName, a.lastName, b.title from Author a
  left join AuthorBook ab on a.authorId = ab.authorId
  left join Book b on ab.bookId = b.bookId;
```

The `left join` clauses ultimately preserve the Cory Doctorow record:

firstName	lastName	title
Norman	Mailer	Naked and the Dead
Douglas	Adams	Dirk Gently's Holistic Detective Agency
Douglas	Adams	The Hitchhiker's Guide to the Galaxy
Douglas	Adams	The Long Dark Tea-Time of the Soul
Norman	Mailer	Barbary Shore
Octavia	Butler	Kindred
Cory	Doctorow	`null`

We had to use a `left join` on *both* tables as there were no records in the join table for Cory Doctorow and thus, by design, were no records in the `Book` table either. For records that have no matches, the resulting column values are `null`.

The term `left` is used in this type of join because we are joining from table A to table B. You can also use a `right join` if you wanted to join from table B to table A instead, but that would be equivalent to a `left join` with the order of the tables reversed in the query. It is much easier to read from left to right and so it is common to simply reverse the tables if you want a `right join` and use `left join` exclusively.

One instance in which you may want to do *both* a `left join` *and* a `right join` at the same time is if you want to join two tables but preserve unmatched records in both. For example, if we wanted to preserve all authors with no book records in our join table but *also* include all books with no author record in our join table. This can be done using a `full outer join` query. For example:

```sql
select * from TableA a
  full outer join TableB b on a.colId = b.colId;
```

In our particular design using a join table, a better approach would be to "simulate" a full outer join using a `union` or `union all` operator.[7]

Consider the following query.

```sql
select * from Author a
  left join AuthorBook ab on a.authorId = ab.authorId
  left join Book b on b.bookId = ab.bookId
union
select * from Author a
  right join AuthorBook ab on a.authorId = ab.authorId
  right join Book b on b.bookId = ab.bookId;
```

The first `select` statement is similar to our previous example. The second `select` statement uses the same order but a `right join` to preserve book records. We use the same ordering in both statements so that the columns will match up. Between them we use a `union` operator to combine the two result sets. Though we use the keyword `union` this will not include duplicates (which is what we usually want). To preserve duplicates from both result sets you can use `union all`.

3.4 Normalization & Database Design

Recall that the purpose of creating a relational database is to separate data out into different tables to minimize the potential for data anomalies and enforce data integrity. This process is formalized as database normalization. Informally, normalization is simply just common sense. When you design a database you generally identify the *entities* that you wish to model and design a table for each of them. Entities usually correspond to real-world entities and have a natural "decomposition" that lend themselves to individual columns.

In our running example we naturally identified a table for books and one for authors.

[7] In some databases, in particular MySQL, full outer joins are not even supported, so you *must* use these operators.

Though these entities are related, they don't have an *is-a* relationship (a book is not an author and an author is not a book). Therefore, we designed a table for each of them. We naturally identified the components of each (a person has a first name, last name; a book has a title, ISBN, etc.). We then identified the relationship between these tables that we wanted to model. From this perspective the process of normalization is straightforward and natural.

More formally, however, there are several levels of normal form that formalize how tables are designed to best minimize redundancy and failure points. Though several "advanced" normal forms have been developed, we'll focus on the original three developed by Edgar Codd. Each normal form builds on the previous and identifies a characteristic about the column attributes/fields in a table and their relationship to the table's primary key.

First Normal Form

For a database to conform to first normal form, it may not have any column that represents more than one value. Each attribute in each table must have only *atomic* values that pertain to the key. Tables that violate first normal form would have column(s) that capture a list or series of values. For example, suppose we wanted to store multiple email addresses for a customer. Suppose further that we defined a column named `emails` and stored a comma-delimited list of values in a single varchar field so as to support as many email addresses we wanted. Such a column does not represent an atomic value, but instead should be modeled using a one-to-many relationship between a customer and an email. To conform to first normal form, we should separate this out into a second table and define an appropriate foreign key.

Using a comma-delimited "list" of values causes many practical problems as well. First, it requires additional processing to deserialize (split out individual email records) and serialize (formatting a list with delimiters, potentially escaping delimiters, etc.) every time you interact with records. Using a simple varchar field also precludes enforcing data integrity (foreign keys needing to refer to an existing record) and loses semantics (it represents elements only as strings instead of entities/tables). Cases where a record has no referencing record or only one referencing record (no emails or only one email) become corner cases that need to be dealt with separately (no delimited list or usage of a flag or special value may be necessary).

In general, violating first normal form defeats one of the main purposes of using a relational database: it removes formal relations!

Second Normal Form

For a database to conform to second normal (2NF) it must conform to first normal form *and* no non-prime attribute may be dependent on any proper subset of prime attributes. This mostly pertains to tables that use a composite key, a primary key that consists of multiple columns. If you take the design approach of using surrogate keys, you are essentially guaranteed second normal form.

Consider the following example that violates second normal form: a purchase record may have a composite key consisting of both a `customerId` and `storeId`. If the table also contained the `storeAddress`, this would be a violation because the store address would only depend on the store (and thus the `storeId`) and not the customer. The solution is again to split out these two separate entities into their own tables and relate them.

Third Normal Form

For a database to conform to third normal (3NF) it must conform to second normal form (and thus, first normal form as well) *and* no non-prime column is transitively dependent on the key. Put another way, no non-prime column may depend on any other non-prime column. All columns must only depend on the key.

An example of a table that would violate third normal form would be a purchase table. Suppose we stored a price-per-unit, quantity, and total columns in this table. The total value would be dependent on the other two since total = price-per-unit × quantity. Given the other two columns we could easily *derive* the total without having to store it explicitly. Storing the total can lead to data anomalies if either of the other two columns were ever changed.

A common saying that summarizes these normal forms is that "every non-key attribute must provide a fact about the key (1NF), the whole key (2NF), and nothing but the key (3NF), so help me Codd."

Denormalization

Normalization provides good general guidance when designing a database, but normalization alone does not guarantee a perfect design nor does it guarantee good performance in a database. Adhering to the normal forms (at least as a starting point) can provide a good means to design tables and identify the relationships you want to model, but there are instances in which you may want to *denormalize* a database for better performance or to simplify a design.

Reconsider the email example where we want to support multiple emails. It is reasonable that a customer or person would have more than one email address. However, it is likely not the case that they would have (say) dozens or hundreds of emails and even if they did, the business use case for supporting such a model may not be justifiable. As a simplification we could hardcode a fixed number of email columns (`Email1`, `Email2`, etc.) effectively creating a one-to-k relationship for a fixed constant k. This is a (slight) violation of first normal form, but it eliminates the need to join to another table.[8]

[8]Indeed, if you ever find yourself creating a table-to-table relationship that is essentially a one-to-one

Violating normal forms is not a critical design flaw, but should be done only after careful consideration and with very good justification.

3.4.1 Design Example

Let's apply what we've covered so far and design a simple database to support a course enrollment system similar to the data we started with in Section 3.1. Our goal will be to design a database to support a course roster system. The design should be able to model students, courses, and their relation to each other. The system will also need to email students about updates in enrollment.

In general, we'll want a table for each entity in this problem. For each one we'll want to identify the fields (columns) that define that entity and identify the relationships between tables. For this problem we'll need tables for both students and courses. We'll start with the student table.

We'll use the stylistic elements we identified before: UpperCamelCasing and singular forms for tables, lowerCamelCasing for column names, and `tableNameId` for primary key columns. Here's the full SQL for this table.

```sql
create table Student (
  studentId int not null auto_increment primary key,
  firstName varchar(255) not null,
  middleName varchar(255),
  lastName varchar(255) not null,
  nuid varchar(8) not null,
  --ensure that NUIDs are unique:
  constraint `uniqueNuids` unique index(nuid)
);
```

We've made both the `firstName` and `lastName` required by making them non nullable. However, the `middleName` is nullable, making it optional (some cultures do not have middle names). We've also included an `nuid` (Nebraska University ID) as natural key but have not made it our primary key. We've done this by giving it an index and a uniqueness constraint. We've created a surrogate key (auto generated `studentId` field) as our primary key instead.

To support multiple email addresses for students, we'll model a full one-to-many relationship, which requires (by 1NF) a separate email table.

```sql
create table Email (
  emailId int not null auto_increment primary key,
```

relationship you may want to rethink your design and collapse the columns to one table.

```
  studentId int not null,
  address varchar(255) not null,
  foreign key(studentId) references Student(studentId)
);
```

We've used the convention that the foreign key has the same name as the key it references. Furthermore, its type (int) must match the key type it references (yet another reason to prefer integers for primary keys). Keep in mind that order is important here: since the `Email` table references the `Student` table, we have to create the `Student` first.

A course table is straightforward and we've included larger text field to support full course descriptions.

```
create table Course (
  courseId int not null auto_increment primary key,
  subject varchar(4) not null,
  number varchar(4) not null,
  title varchar(255) not null,
  description varchar(4096)
);
```

We still need to model the actual enrollment of students into courses. Clearly this is a many-to-many relationship and we should thus create a join table to bring them together. We'll use two foreign keys referencing a student and a course. However, we also want to model *when* the student enrolled in the course. For this we can include a semester column; but what type should it be? We could make it a varchar to support representations like "Fall 2019" or "Spring 2020". However, such lexicographic representations are not well-ordered. When sorted alphanumerically, "Fall 2019" would come before "Spring 2020" even though they are out of order temporally. It is common to instead establish an encoding that uses integers (which are well-ordered). These are often referred to as *sort keys*. The drawback to this is that the encoding is not necessarily readable to end-users and should/must be converted appropriately.

```
create table Enrollment (
  enrollmentId int not null auto_increment primary key,
  studentId int not null,
  courseId int not null,
  semester int not null,
  foreign key(studentId) references Student(studentId),
  foreign key(courseId) references Course(courseId)
);
```

There is still a problem though. There is nothing that prevents us from inserting multiple

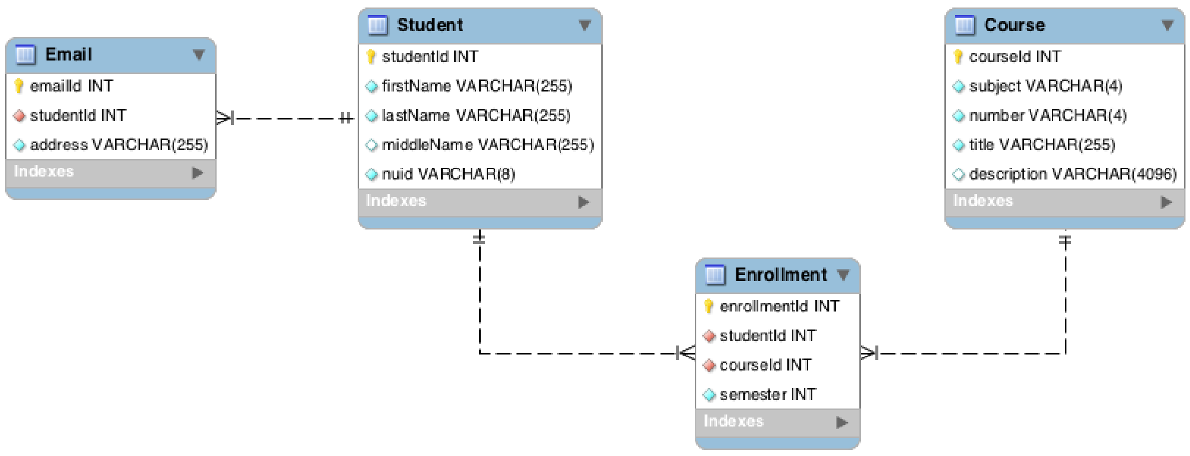


Figure 3.9: Enrollment Database

records for the same student, same course, in the same semester. Though a student could take the same course multiple times (say if they withdrew the first time), they can't take the same course in the same semester. This should be considered bad data and we should define a constraint to prevent it. Specifically we could add a uniqueness constraint on the combination of all three of these fields:

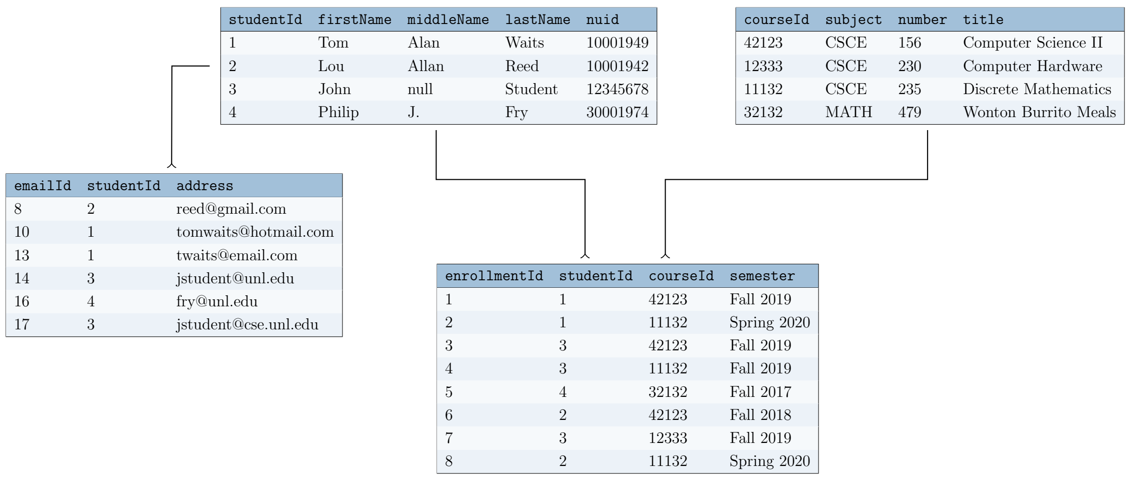
```
constraint `uniqueEnrollment` unique index(studentId,courseId,semester)
```

There are lots of other things we could add and model (offerings, instructors, etc.) and many more rules and constraints we could define. We leave these as Exercise 3.4. The final ER digram for this database is presented as Figure 3.9.

The same data as in Table 3.1 is depicted as an ER Digram in Figure 3.10. Observe that we've now solved most of the problems we identified that motivated using a relational database. Each piece of data is now organized and have specific types. There is minimal duplication of data. Entities are represented by unique IDs, ensuring identity (Tom Waits is now the same as t. Waits). Data integrity rules are now enforced (students only have one unique NUID) and relationships are well-defined.

3.5 Miscellaneous

Data, data representations, and databases are a huge topic that could not only span a course but an entire series of courses. We've only touched on the core topics there are dozens of other issues with data and databases that are beyond the scope of this chapter. In actual organizations, there may be entire *teams* devoted to Database Administrator (DBA). DBAs handle data maintenance, backups, migration, redundancy and reliance, quality assurance, tune performance, etc.



studentId	firstName	middleName	lastName	nuid
1	Tom	Alan	Waits	10001949
2	Lou	Allan	Reed	10001942
3	John	null	Student	12345678
4	Philip	J.	Fry	30001974

courseId	subject	number	title
42123	CSCE	156	Computer Science II
12333	CSCE	230	Computer Hardware
11132	CSCE	235	Discrete Mathematics
32132	MATH	479	Wonton Burrito Meals

emailId	studentId	address
8	2	reed@gmail.com
10	1	tomwaits@hotmail.com
13	1	twaits@email.com
14	3	jstudent@unl.edu
16	4	fry@unl.edu
17	3	jstudent@cse.unl.edu

enrollmentId	studentId	courseId	semester
1	1	42123	Fall 2019
2	1	11132	Spring 2020
3	3	42123	Fall 2019
4	3	11132	Fall 2019
5	4	32132	Fall 2017
6	2	42123	Fall 2018
7	3	12333	Fall 2019
8	2	11132	Spring 2020

Figure 3.10: Enrollment Data (some fields have been removed for presentation and others have been modified to be more readable)

With respect to relational databases, there are dozens of other commands, operations and features that we've not covered here such as temporary tables, views, triggers, stored procedures and more. There is also the issue of programmatically connecting to a database which most programming languages support either natively or through a library. Java for example provides the Java Database Connectivity API (JDBC) which defines interfaces to interact with a generic database. You write Java code to connect to a database, execute queries (possibly) receive a result set back to process. Using and API and interfaces like this is a form of dependency inversion as discussed in Section **??**. Many programming languages also provide Object-Relational Mapping (ORM) functionality in which you can define a mapping from your objects/member variables to tables/columns in a database and let the library generate the queries for you. Java's Java Persistence API (JPA) is one such example.

Beyond relational databases are non-relational databases, often called No-SQL that offer performance improvements at the cost of the ACID principles. Databases such as Apache's Cassandra and Amazon's DynamoDB store data as Key Value Pair (KVP) instead of related tables. Redis is a popular general purpose database that allows you to store data as *data structures* (so entire list or Tree data structures are directly stored in the database and can be queried).

3.6 Exercises

Exercise 3.1. Consider a database to model a video game library as depicted in the ER diagram in Figure 3.11. As designed, video games have a single publisher (a publisher may have published more than one game) and games may be available on more than one platform (while platforms may certainly have more than one game).

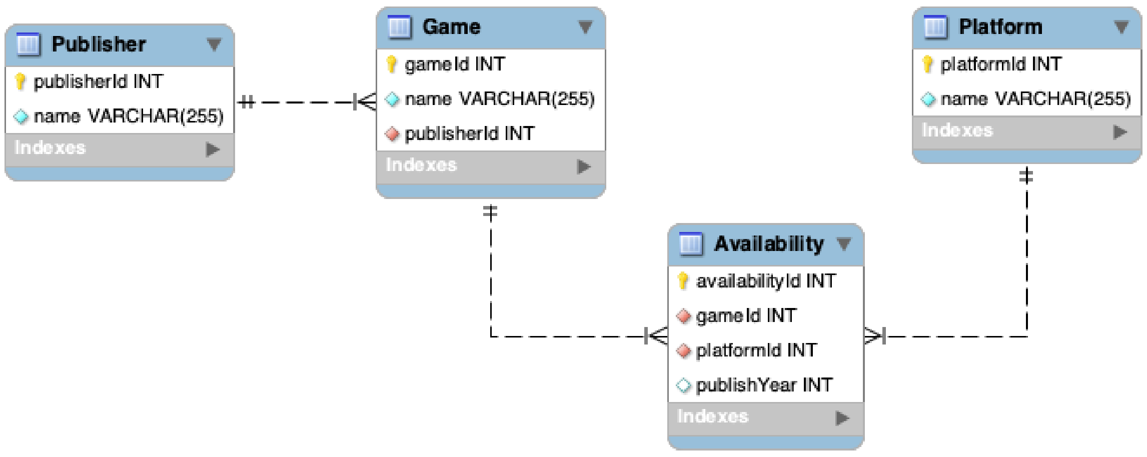


Figure 3.11: Simple Video Game Database

Reverse engineer this database and write SQL code to create all four tables. Use a tool to reproduce the ER diagram and identify and fix any mistakes.

Exercise 3.2. Write SQL queries to produce the following reports for data from the video game database.

1. List all video games in the database
2. List all video games that start with "G"
3. List all publishers in the database
4. List all video games along with their publishers
5. List all video games along with their publishers, but only the relevant fields
6. List all publishers in the database along with all their games, even if they don't have any
7. List all publishers with a count of how many games they have
8. List all games and all systems that they are available on
9. List all games that are not available on any system
10. List the oldest game(s) and its platform(s)
11. Flatten the entire data model by returning all data on all games
12. Insert a new game, *Assassin's Creed*, published by Ubisoft
13. Make the new game available on at least two platforms
14. Update the record for Megaman 3: the publisher should be Capcom, not Eidos
15. Delete the publisher Eidos

16. Write a query to return all games along with the number of platforms they are available on

Exercise 3.3. Add constraints to your video game database to prevent multiple game records for the same platform (in the same year).

Exercise 3.4. Modify the enrollment database so that instead of a single course table, there are specific *offerings* for each semester that students can enroll in. Add support for instructors that teach particular offerings of courses.

Exercise 3.5. Write SQL to insert the example data into the enrollment tables.

Exercise 3.6. Add a few book records to the `Book` table without any authors. Then execute the `union` and `union all` queries and observe the results.

4 List-Based Data Structures

Most programming languages provide some sort of support for storing collections of similar elements. The most common way is to store elements in an *array*. That is, elements are stored together in a contiguous chunk of memory and individual elements are accessed using an *index*. An index represents an *offset* with respect to the first element which is usually stored at index 0 (referred to as *zero-indexing*).

There are several disadvantages to using arrays, however. In particular, once allocated, the capacity of an array is fixed. It cannot grow to accommodate new elements and it cannot shrink if we end up removing elements. Moreover, we may not need the full capacity of the array at any given point in a program, leading to wasted space. Though libraries may provide convenience functions, in general all of the "bookkeeping" in an array is up to us. If we remove an element in the middle of the array, our data may no longer be contiguous. If we add an element in the array, we have to make sure to find an available spot. Essentially, all of the organization of the array falls to the user.

A much better solution is to use a dynamic data structure called a *List*. A List is an Abstract Data Type (ADT) that stores elements in an *ordered* manner. That is, there is a notion of a "first" element, a "second" element, etc. This is *not* necessarily the same thing as being *sorted*. A list containing the elements `10, 30, 5` is not sorted, but it is ordered (`10` is the first element, `30` is the second, and `5` is the third and final element). In contrast to an array, the list automatically organizes the elements in some underlying structure and provides an *interface* to the user that provides some set of core functionality, including:

- A way to add elements to the list
- A way to retrieve elements from the list
- A way to remove elements from the list

in some manner. We'll examine the specifics later on, but the key aspect to a list is that, in contrast to an array, it dynamically expands and contracts automatically as the user adds/removes elements.

How a list supports this core functionality may vary among different implementations. For example, the list's interface may allow you to add an element to the beginning of the list, or to the end of the list, or to add the new element at a particular index; or any combination of these options. The retrieval of elements could be supported by providing an index-based retrieval method or an iterator pattern that would allow a user to conveniently iterate over every element in the list.

In addition, a list may provide secondary functionality as a convenience to users, making the implementation more flexible. For example, it may be useful for a user to tell how many elements are in the list; whether or not it is empty or full (if it is designed to have a constrained capacity). A list might also provide batch methods to allow a user to add a collection of elements to the list rather than just one at a time.

Most languages will provide a list implementation (or several) as part of their standard library. In general, the best practice is to use the built-in implementations unless there is a very good reason to “roll your own” and create your own implementation. However, understanding how various implementations of lists work and what properties they provide is very important. Different implementations provide advantages and disadvantages and so using the correct one for your particular application may mean the difference between an efficient algorithm and an inefficient or even infeasible one.

4.1 Array-Based Lists

Our first implementation is an obvious extension of basic arrays. We’ll still use a basic array to store data, but we’ll build a data structure around it to implement a full list. The details of how the list works will be encapsulated inside the list and users will interact with the list through publicly available methods.

The basic idea is that the list will own an array (via composition) with a a certain *capacity*. When users add elements to the list, they will be stored in the array. When the array becomes full, the list will automatically reallocate a new, larger array (giving the list a larger capacity), copy over all the elements in the old array and then switch to this new array. We can also implement the opposite functionality and shrink the array if we desire.

4.1.1 Designing a Java Implementation

To illustrate this design, consider the basic the following code sample in Java.

```java
public class IntegerArrayList {

  private int arr[];
  private int size;

}
```

This array-based list is designed to store integers in the `arr` array. The second member variable, `size` will be used to track the number of elements stored in `arr`. Note that this is not the same thing as the size of the array (that is, `arr.length`). Elements may or may not be stored in each array position. To distinguish between the size of the array-based list and the size of the internal array, we'll refer to them as the *size* and *capacity*

To initialize an empty array list, we'll create a default constructor and instantiate the array with an initial *capacity* of 10 elements with an initial size of `0`.

```java
public IntegerArrayList() {
  this.arr = new int[10];
  this.size = 0;
}
```

Adding Elements

Now let's design a method to provide a way to add elements. As a first attempt, let's allow users to add elements to the *end* of the list. That is, if the list currently contains the elements `8, 6, 10` and the user adds the element `42` the list will then contain the elements `8, 6, 10, 42`.

Since the `size` variable keeps track of number of elements in the list, we can use it to determine the index at which the element should be inserted. Moreover, once we insert the element, we'll need to be sure to increment the `size` variable since we are increasing the number of elements in the list. Before we do any of this, however, we need to check to ensure that the underlying array has enough room to hold the new element and if it doesn't, we need to increase the capacity by creating a new array and copying over the old elements. This is all illustrated in the following code snippet.

```java
public void addAtEnd(int x) {

  //if the array is at capacity, resize it
  if(this.size == this.arr.length) {
    //create a new array with a larger capacity
    int newArr = new int[this.arr.length + 10];
    //copy over all the old elements
    for(int i=0; i<this.arr.length; i++) {
      newArr[i] = this.arr[i];
    }
    //use the new array
    this.arr = newArr;
  }
  this.arr[size] = x;
15    this.size++;
16  }
```

The astute Java programmer will note that lines 5–12 can be improved by utilizing methods provided by Java's `Arrays` class. Adhering to the Don't Repeat Yourself (DRY) principle, a better version would be as follows.

```java
public void addAtEnd(int x) {

  if(this.size == this.arr.length) {
    this.arr = Arrays.copyOf(this.arr, this.arr.length + 10);
  }
  this.arr[size] = x;
  this.size++;
}
```

As another variation, we could allow users to add elements at an arbitrary index. That is, if the array contained the elements `8, 6, 10`, we could allow the user to insert the element `42` at any index 0 through 3. The list would automatically shift elements down to accommodate the new element. For example:

- Adding at index 0 would result in `42, 8, 6, 10`
- Adding at index 1 would result in `8, 42, 6, 10`
- Adding at index 2 would result in `8, 6, 42, 10`
- Adding at index 3 would result in `8, 6, 10, 42`

Note that though there is no element (initially) at index 3, we still allow the user to "insert" at that index to allow the user to insert the element at the end. However, any other index should be considered invalid as it would either be invalid (negative) or it would mean that the data is no longer contiguous. For example, adding `42` at index 5 may result in `8, 6, 10, null, null, 42`. Thus, we do some basic index checking and throw an exception for invalid indices.

```java
public void insertAtIndex(int x, int index) {

  if(index < 0 || index > this.size) {
    throw new IndexOutOfBoundsException("invalid index: " + index);
  }

  if(this.size == this.arr.length) {
    this.arr = Arrays.copyOf(this.arr, this.arr.length + 10);
  }
```

```

    //start at the end; shift elements to the right to accommodate x
    for(int i=this.size-1; i>=index; i--) {
      this.arr[i+1] = this.arr[i];
    }
    this.arr[index] = element;
    this.size++;
  }
```

At this point we note that these two methods have a lot of code in common. In fact, one could be implemented in terms of the other. Specifically, `insertAtIndex` is the more general of the two and so `addToEnd` should use it:

```
public void addAtEnd(int x) {

  this.insertAtIndex(x, this.size);
}
```

This is a big improvement as it reduces the complexity of our code and thus the complexity of testing, maintenance, etc.

Retrieving Elements

In a similar manner, we can allow users to retrieve elements using an index-based retrieval method. We will again take care to do index checking, but otherwise returning the element is straightforward.

```
public void getElement(int index) {

  if(index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("invalid index: " + index);
  }
  return this.arr[index];
}
```

Removing Elements

When a user removes an element, we want to take care that all the elements remain contiguous. If the array contains the elements `8, 6, 10` and the user removes `8`. we want to ensure that the resulting list is `6, 10` and not `null, 6, 10`. Likewise, we'll want to make sure to *decrement* the size when we remove an element. We'll also return the value that is being removed. The user is free to ignore it if they don't want it, but this makes the list interface a bit more flexible as it means that the user doesn't have to make two method calls to retrieve-then-delete the elements. This is a common idiom with many collection data structures.

```java
public int removeAtIndex(int index) {

  if(index < 0 || index > this.size) {
    throw new IndexOutOfBoundsException("invalid index: " + index);
  }

  //start at index and shift elements to the left
  for(int i=index; i<size-1; i++) {
    this.arr[i] = this.arr[i+1];
  }
  this.size--;
}
```

Note that we omitted automatically "shrinking" the underlying array if its unused capacity became too big. We leave that and other variations on the core functionality as an exercise.

Secondary Functionality

In addition to the core functionality of a list, you could extend our design to provide more convenience methods to make the list implementation even more flexible. For example, you may implement the following methods, the details of which are left as an exercise.

- `public boolean isEmpty()` – this method would return `true` if no elements were stored in the list, `false` otherwise.
- `public int size()` – more generally, a user may want to know how many elements are in the list especially if they wanted to avoid an `IndexOutOfBoundsException`.
- `public void addAtBeginning(int element)` – the companion to the `addAtEnd(int)` method
- `public int replaceElementAt(int element, int index)` – a variation on the remove method that replaces rather than remove the element, returning the replaced element.
- `public void addAll(int arr[], int index)` – a *batch* method that allows a user to add entire array to the list with one method call. Similarly, you could allow the user to add elements stored in another list instance,

`public void addAll(IntegerArrayList list, int index)`. Sometimes this operation is referred to as "splicing."

- `public void clear()` – another batch method that allows a user to remove all elements at once.

Many other possible variations exist. In addition to the interface, one could vary how the underlying array expands or shrinks to accommodate additions and deletions. In the example above, we increased the size by a constant size of 10. Variations may include expanding the array by a certain percentage or doubling it in size each time. Each strategy has its own advantages and disadvantages.

A Better Implementation

The preceding list design was still very limited in that it only allowed the user to store integers. If we wanted to design a list to hold floating point numbers, strings, or a user defined type, we would need to create an implementation for every possible type that we wanted a list for. Obviously this is not a good approach. Each implementation would only differ in its name and the type of elements it stored in its array. This is a quintessential example of when to use parameterized polymorphism. Instead of designing a list that holds a particular type of element, we can parameterize it to hold *any* type of element. Thus, only one array-based list implementation is needed.

In the example we designed before for integers, we never actually examined the content of the array inside the class. We never used the fact that the underlying array held integers and the only time we referred to the `int` type was in the method signatures which can all be parameterized to accept and return the same type.

Code Sample 4.1 contains a parameterized version of the array-based list we implemented before.

A few things to note about this parameterized implementation. First, with the original implementation since we were storing primitive `int` elements, `null` was not an issue. Now that we are using parameterized types, a user would be able to store `null` elements in our list. We could make the design decision to allow this or disallow it (by adding null pointer checks on the add/insert methods).

Another issue, particular to Java, is the instantiation of the array on line 7. We *cannot* invoke the `new` keyword on an array with an indeterminate type. This is because different types require a different number of bytes (integers take 4 bytes, `double`s take 8 bytes). The number of bytes may even vary between Java Virtual Machine (JVM)s (32-bit vs. 64-bit). Without knowing how many bytes each element takes, it would be impossible for the JVM to allocate the right amount of memory. Thus, we are forced to use a *raw type* (the `Object` type) and do an explicit type cast. This is not much of an issue because the parameterizations guarantee that the user would only ever be able to add elements of type `T`.

Another useful feature that we could add would be an iterator pattern. An iterator allows you to iterate over each element in a collection and process them. With a traditional array, a simple for loop can be used to iterate over elements. An iterator pattern relieves the user of the need to write such boilerplate code and instead use a *foreach* loop.[1]

In Java, an iterator pattern is achieved by implementing the `Iterable<T>` interface (which is also parameterized). The interface requires the implantation of a public method that returns an `Iterator<T>` which has several methods that need to be implemented. The following is an example that could be included in our `ArrayList` implementation.

```java
public Iterator<T> iterator() {
  return new Iterator<T>() {
    private int currentIndex = 0;
    @Override
    public boolean hasNext() {
      return (this.currentIndex < size);
    }

    @Override
    public T next() {
      this.currentIndex++;
      return arr[currentIndex-1];
    }

  };
}
```

Essentially, the iterator (an anonymous class declaration/definition) has an internal index, `currentIndex` that is initialized to 0 (the first element). Each time `next()` is called, the "current" element is returned, but the method also sets itself up for the next iteration by incrementing `currentIndex`. The main advantage of implementing an iterator is that you can then use a foreach loop (which Java calls an "enhanced for-loop"). For example:

```java
ArrayList<Integer> list = new ArrayList<Integer>();
list.addAtEnd(8);
list.addAtEnd(6);
list.addAtEnd(10);

//prints "8 6 10"
```

[1]Note that this is not necessarily mere syntactic sugar. As we will see later, some collections are unordered and would *require* the use of such a pattern. Yet still, some list implementations, in particular linked lists, an index-based get method is actually very *inefficient*. An iterator pattern allows you to encapsulate the most efficient logic for iterating over a particular list implementation.

```java
for(Integer x : list) {
  System.out.print(x + " ");
}
```

4.2 Linked Lists

The array-based list implementation offers a lot of advantages and improvements over a primitive array. However, it comes at some cost. In particular, when we need to expand the underlying array, we have to create a new array and copy over every last element. If there were 1 million elements in the underlying array and we wanted to add one more, we would essentially be performing 1 million copy operations. In general, if there are n elements, we would be performing n copy operations (or a copy operation proportional to n). That is, some operations will induce a *linear* amount of work with respect to how many elements are already in the list. Even with different strategies for increasing the size of the underlying array (increasing the size by a percentage or doubling it), we still have some built-in overhead cost associated with the basic list operations.

For many applications this cost is well-worth it. A copy operation can be performed quite efficiently and the advantages that a list data structure provide to development may outweigh the efficiency issues (and in any case, the same issues would be present even if we used a primitive array). In some applications, however, an alternative implementation may be more desirable. In particular, a *linked list* implementation avoids the expansion/shrinking of an underlying array as it uses a series of linked *nodes* to store elements rather than a primitive array.

A simple linked list is depicted in Figure 4.1. Each element is stored in a node. Each node contains both an element (in this example, an integer) and a reference to the next node in the list. In this way, each node is *linked* together to form a chain. The start of the list is usually referred to as the *head* of the list. Similarly, the end of the list is usually referred to as the *tail* (in this case, the node containing 10). A special symbol or value is usually used to denote the end of the list so that that tail node does not point to another node. In this case, we've used the value ϕ to indicate the end of the list. In practice, a `null` value could be used or a special *sentinel* node can be created to indicate the end of a list.

To understand how a linked list works, we'll design several algorithms to support the core functionality of a list data structure. In order to refer to nodes and their elements, we'll use the following notation. Suppose that u is a node, then its value will be referred to as $u.value$ and the next node it is linked to will be referred to as $u.next$.

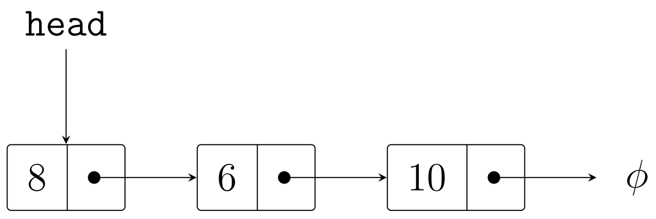


Figure 4.1: A simple linked list containing 3 nodes.

Adding Elements

With a linked list, there is no underlying array of a fixed size. To add an element, we simply need to create a new node and link it somewhere in the chain. Since there is no fixed array space to fill up, we no longer have to worry about expanding/shrinking it to accommodate new elements.

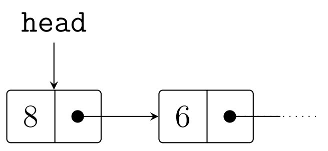
To start, an empty list will be represented by having the head reference refer to the special end-of-list symbol. That is, $head \rightarrow \phi$. To add a new element to an empty list, we simply create a new node and have the head reference point to it. In fact, we can more generally support an *insert at head* operation with the same idea. To insert at the head of the list, we create a new node containing the inserted value, make it point to the "old head" node of the list and then update the head reference to the new node. This operation is depicted in Algorithm 1.

```
INPUT : A linked list $L$ with head, $L.head$ and a new element to insert $x$ at the
        head
1 $u \leftarrow$ a new node
2 $u.value \leftarrow x$
3 $u.next \leftarrow L.head$
4 $L.head \leftarrow u$
```

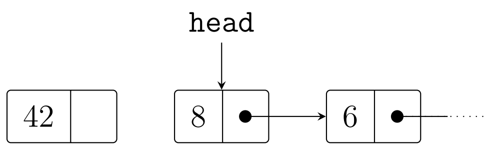
Algorithm 1: Insert-At-Head Linked List Operation

Just as with the array-based list, we could also support a more general insert-at-index method that would allow the user to insert a new node at any position in the list. The first step, of course, would be to find the two nodes between which you wanted to insert the new node. We will save the details for this procedure as it represents a more general retrieval method. For now, suppose we have two nodes, a, b and we wish to insert a new node, u between them. To do this, we simply need to make a refer to the new node and to make the new node refer to b. However, we must be careful with the order in which we do this so as not to lose a's reference to b. The general procedure is depicted in Algorithm 2.

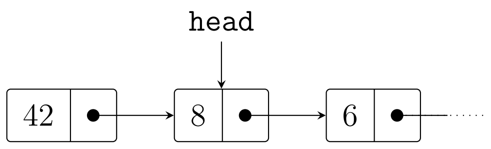
One special corner case occurs when we want to insert at the end of a list. In this



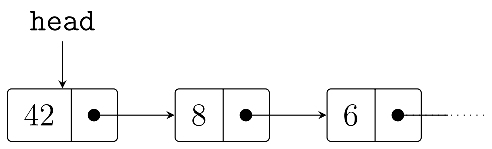
(a) Initially the linked list has element 8 as its head.



(b) We create a new node containing 42



(c) We then make the new node refer to the old head node.

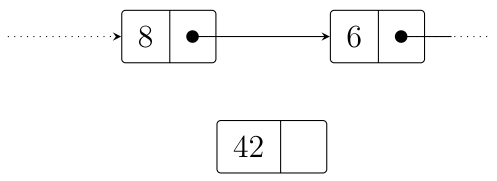


(d) And finally update the head reference to point to the new node instead.

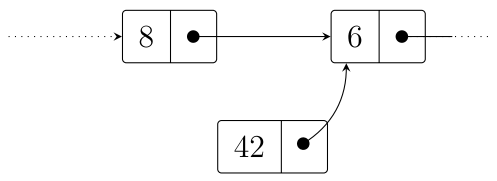
Figure 4.2: Insert-at-head Operation in a Linked List. We wish to insert a new element, 42 at the head of the list.

```
INPUT  : Two linked nodes, $a, b$ in a linked list and a new value, $x$ to insert
         between them
1 $u \leftarrow$ a new node
2 $u.value \leftarrow x$
3 $u.next \leftarrow a.next$
4 $a.next \leftarrow u$
```

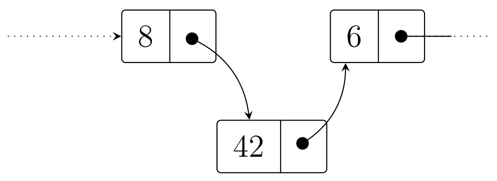
Algorithm 2: Insert Between Two Nodes Operation



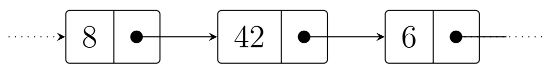
(a) We create a new node containing 42



(b) We then make the new node point to the second node



(c) And reassign the first node's *next* reference to the new node.



(d) Resulting in the new node being inserted between the two given nodes.

Figure 4.3: Inserting Between Two Nodes in a Linked List. Here, we wish to insert a new element 42 between the given two nodes containing 8 and 6.

scenario, b would not exist and so $a.next$ would end up referring to ϕ. This ends up working out with the algorithm presented. We never actually made any explicit reference to b in Algorithm 2. When we assigned $u.next$ to refer to $a.next$, we took care of both the possibility that it was an actual node and the possibility that it was ϕ.

Another corner case occurs if we wish to insert at the head of the list using this algorithm. In that scenario, a would not refer to an actual node while b would refer to the *head* element. In this case, lines 3–4 would be invalid as $a.next$ would be an invalid reference. For this corner case, we would need to either fall back to our first insert-at-head (Algorithm 1) operation or we would need to handle it separately.

Retrieving Elements

As previously noted, a linked list avoids the cost of expanding/shrinking of an underlying array. However, this does come at a cost. With an array-based list, we had "free" random access to elements. That is, if we wanted the element stored at index i it is a simple matter to compute a memory offset and "jump" to the proper memory location. With a linked list, we do *not* have the advantages of random access.

Instead, to retrieve an element, we must sequentially search through the list, starting from the head, until we find the element that we are trying to retrieve. Again, many variations exist, but we'll illustrate the basic functionality by describing the same index-based retrieval method as before.

```
INPUT  : A linked list $L$ with head, $L.head$ and in index $i$, $0 \leq i < n$ where $n$ is
         the number of elements in $L$
OUTPUT : The $i$-th node in $L$
1 $currentNode \leftarrow L.head$
2 $currentIndex \leftarrow 0$
3 WHILE $currentIndex < i$ DO
4     $currentNode \leftarrow currentNode.next$
5     $currentIndex \leftarrow (currentIndex + 1)$
6 END
7 output $currentNode$
```

Algorithm 3: Index-Based Retrieval Operation

The key to this algorithm is to simply keep track of the "current" node. Each iteration we traverse to the next node in the chain and iterate a counter. When we have traversed i times, we stop as that is the node we are looking for.

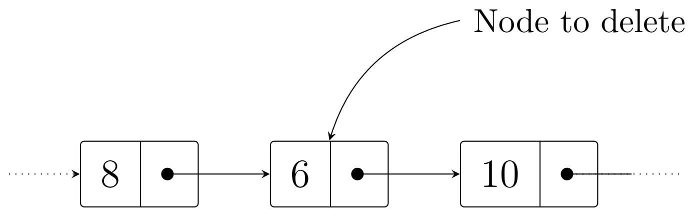
In contrast to the "free" index-based retrieval method with an array-based list, the operation of finding the i-th node in a linked list is much more expensive. We have to perform i operations to get the i-th node. In the worst case, when we are trying to retrieve the last (tail) element, we would end up performing n traversal operations. If the linked list held a lot of elements, say 1 million, then this could be quite expensive. In this manner, we see some clear trade-offs in the two implementations.

Removing Elements

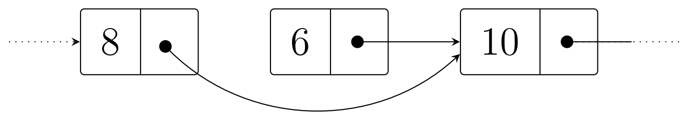
Like the insertion operation, the removal of an element begins by retrieving the node that contains it. Suppose we have found a node, u and wish to remove it. It actually suffices to simply circumvent the node, by making u's predecessor point to u's successor. This is illustrated in Figure 4.4.

Since we are changing a reference in u's predecessor, however, we must make appropriate changes to the retrieval operation we developed before. In particular, we must keep track of two nodes: the current node as before but also its predecessor, or more generally, a *previous* node. Alternatively, if we were performing an index-based removal operation, we could easily find the predecessor by adjusting our index. If we wanted to delete the i-th node, we could retrieve the $(i-1)$-th node and delete the next one in the list.

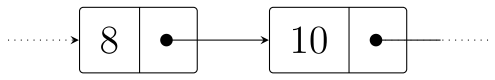
Again, we may need to deal with corner cases such as if we are deleting the head of the list or the tail of the list. As with the insertion, the case in which we delete the tail is essentially the same as the general case. If the successor of a tail node is ϕ, thus when we make u's predecessor refer to u's successor, we are simply making it refer to ϕ.



(a) We wish to delete the node containing 6.



(b) We make its predecessor node point to its successor.



(c) Though the node containing 6 still refers to the next node, from the perspective of the list, it has been removed.

Figure 4.4: Delete Operation in a Linked List

The more complicated part is when we delete the head of the list. By definition, there is no predecessor to the head, so handling it as the general case will result in an invalid reference. Instead, if we wanted to delete the head, we must actually change the list's head itself. This is easy enough, we simply need to make *head* refer to $head- > next$.

An example of a removal operation is presented in Algorithm 4. In contrast to previous examples, this operation is a *key-based* removal operation variation. That is, we search for the first instance of a node whose value matches a given key element and delete it. As another corner case for this variation, we also must deal with the situation in which no node matches the given key. In this case, we've decided to leave it as a no-operation ("noop"). This is achieved in lines 10–12 which will not delete the last node if the key does not match.

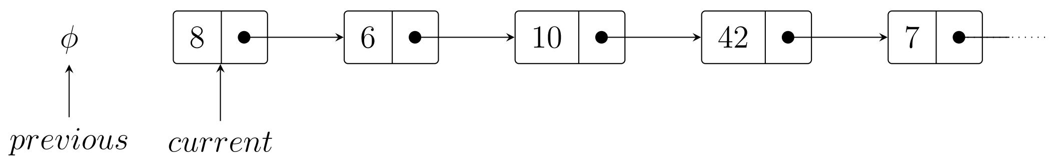
```
    INPUT  : A linked list $L$ with head, $L.head$ and a key element $k$
    OUTPUT : $L$ but with the first node $u$ such that $u.value = k$ removed if one exists
 1  IF $L.head.value = k$ THEN
 2      $L.head. \leftarrow L.head.next$
 3  ELSE
 4      $previousNode \leftarrow \phi$
 5      $currentNode \leftarrow L.head$
 6      WHILE $currentNode.value \neq k$ and $currentNode.next \neq \phi$ DO
 7          $previousNode \leftarrow currentNode$
 8          $currentNode \leftarrow currentNode.next$
 9      END
10      IF $currentNode.value = k$ THEN
11          $previousNode.next \leftarrow currentNode.next$
12      END
13  END
14  output $L$
```

Algorithm 4: Key-Based Delete Operation

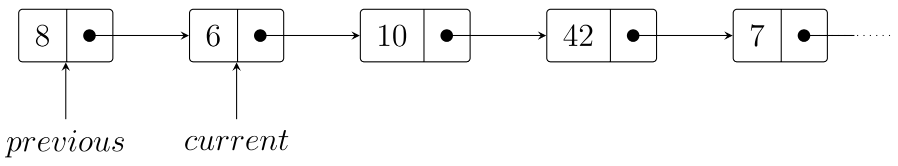
4.2.1 Designing a Java Implementation

We now adapt these operations and algorithms to design a Java implementation of a linked list. Note that the standard collections library has both an array-based list (`java.util.ArrayList<E>`) as well as a linked list implementation (`java.util.LinkedList<E>`) that should be used in general.

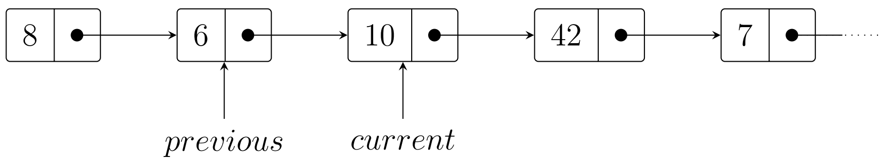
First, we need a node class in order to hold elements as well as a reference to another node. Code Sample 4.2 gives a basic implementation with several convenience methods.



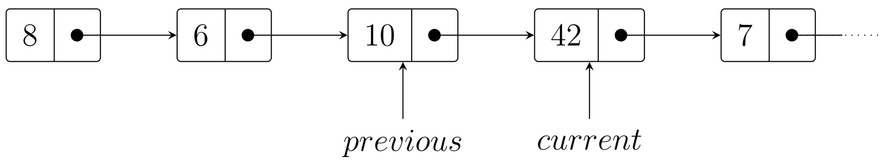
(a) Initialization of the previous and current references.



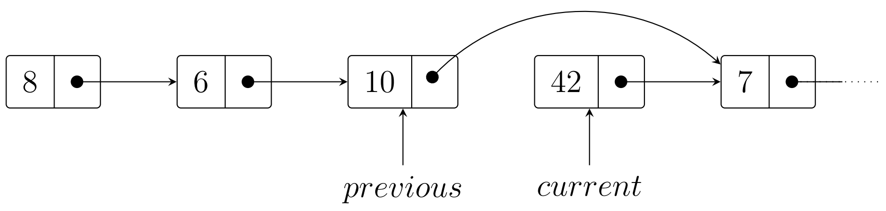
(b) After the first iteration of the while loop, the references have been updated.



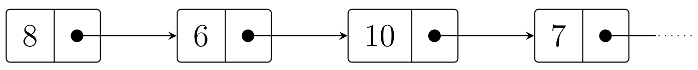
(c) After the second iteration of the while loop.



(d) After the third iteration, the current node matches our key value, 42 and the loop terminates.



(e) Since the current node matches the key, we remove it by circumventing it.



(f) Resulting in the node's removal.

Figure 4.5: Key-Based Find and Remove Operation. We wish to remove the first node we find containing 42.

The class is parameterized so that nodes can hold any type.

Given this `Node` class, we can now define the basic state of our linked list implementation:

```java
public class LinkedList<T> {

  private Node<T> head;
  private int size;

  public LinkedList() {
    this.head = null;
    this.size = 0;
  }

  ...

}
```

We keep track of the size of the list and increment/decrement it on each add/delete operation so that we do not have to recompute it. To keep things simple, we will implement two general purpose node retrieval methods, one index-based and one key based. These can be used as basic steps in other, more specific operations such as insertion, retrieval, and deletion methods. Note that both of these methods are `private` as they are intended for "internal" use by the class and not for external use. If we had made these methods `public` we would be exposing the internal structure of our linked list to outside code. Such a design would be a typical example of a leaky abstraction and is, in general, considered bad practice and bad design.

```java
private Node<T> getNodeAtIndex(int index) {

  if(index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("invalid index: " + index);
  }
  Node<T> curr = this.head;
  for(int i=0; i<index; i++) {
    curr = curr.getNext();
  }
  return curr;

}

private Node<T> getNodeWithValue(T key) {

  Node<T> curr = head;
  while(curr != null && !curr.getItem().equals(key)) {
    curr = curr.getNext();
  }
  return curr;
}
```

As previously mentioned, these two general purpose methods can be used or adapted to implement other, more specific operations. For example, index-based retrieval and removal methods become very easy to implement using these methods as subroutines.

```
public T getElement(int index) {
  return getNodeAtIndex(index).getItem();
}

public T removeAtIndex(int index) {

  if(index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("invalid index: " + index);
  }
  Node<T> previous = getNodeAtIndex(index-1);
  Node<T> current  = previous.getNext();
  T removedItem = current.getItem();
  previous.setNext(current.getNext());
  return removedItem;
}
```

Adapting these methods and using them to implement other operations is left as an exercise.

4.2.2 Variations

In addition to the basic linked list data structure design, there are several variations that have different properties that may be useful in various applications. For example, one simple variation would be to not only keep track of the head element, but also a tail element. This would allow you to efficiently add elements to either end of the list without having to traverse the entire list. Other variations are more substantial and we will not look at several of them.

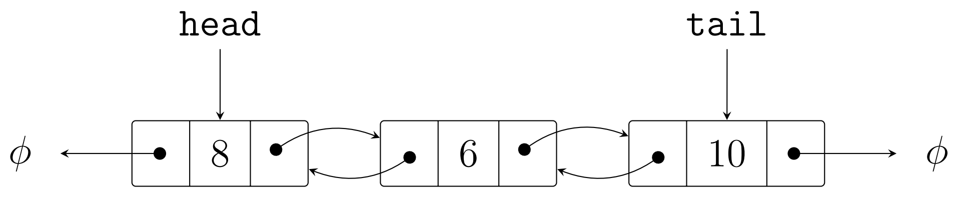


Figure 4.6: A Doubly Linked List Example

Doubly Linked Lists

As presented, a linked list is "one-way." That is, we can only traverse forward in the list from the head toward the tail. This is because our tree nodes only kept track of a *next* element, referring to the node's predecessor. As an alternative, our list nodes could keep track of both the *next* element as well as a *previous* element so that it has access to both a node's predecessor as well as its successor. This allows us to traverse the list two ways, forwards and backwards. This variation is referred to as a *doubly linked list*.

An example of a doubly linked list is depicted in Figure 4.6. In this example, we've also established a reference to the *tail* element to enable us to start at the end of the list and traverse backwards.

A doubly linked list may provide several advantages over a singly linked list. For example, when we designed our key-based delete operation we had to be sure to keep track of a previous element in order to manipulate its reference to the next node. In a doubly linked list, however, since we can always traverse to the previous node this is not necessary.

A doubly linked list has the potential to simplify a lot of algorithms, however it also means that we have to take greater care when we manipulate the next and previous references. For example, to insert a new node, we would have to modify up to four references rather than just two. Greater care must also be taken with respect to corner cases.

Circularly Linked Lists

Another variation is a *circularly linked list*. Instead of a ending the list using a sentinel value, ϕ, the "last" node points back to the first node, closing the list into a large loop. An example is given in Figure 4.7. With such a structure, it is less clear that there is a head or tail to the list. Certain operations such as index-based operations may no longer make sense with such an implementation. However, we could still designate an arbitrary node in the list as the "head" as in the figure.

Alternatively, we could instead simply have a reference to a *current* node. At any point during the data structure's life cycle the current node may reference any node in the list. The core operations may iterate through the list and end up at a different node each

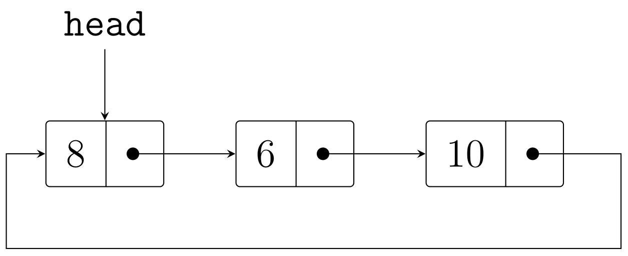


Figure 4.7: A Circularly Linked List Example

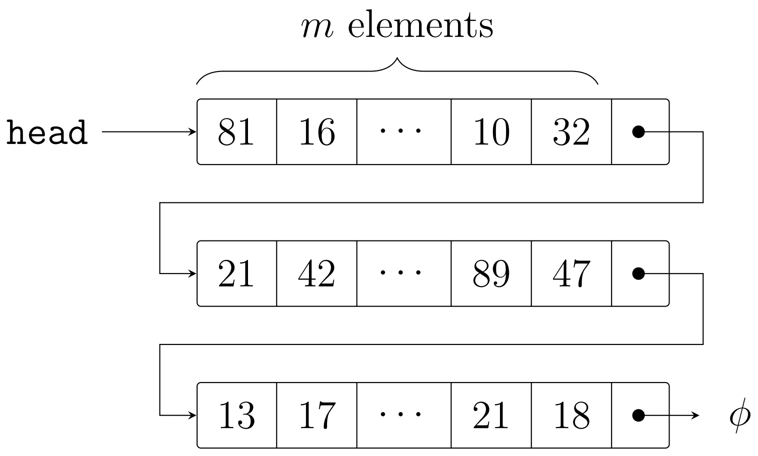


Figure 4.8: An Unrolled Linked List Example

time. Care would have to be taken to ensure that operations such as searching do not result in an infinite loop, however. An unsuccessful key-based search operation would need to know when to terminate (when it has gone through the entire circle once and returned to where it started). It is easy enough to keep track of the size of the list and to ensure that no circular operations exceed this size.

Circularly linked lists are useful for applications where elements must be processed over and over each in turn. For example, an operating system may give time slices to running applications. Instead of a well-defined beginning and end, a continuous poll loop is run. When the "last" application has exhausted its time slot, the operating system returns to the "first" in a continuous loop.

Unrolled Linked Lists

As presented, each node in a linked list holds a single element. However, we could design a node to hold any number of elements, in particular an array of m elements. Nodes would still be linked together, but each node would be a mini array-based list as well. An example is presented in Figure 4.8.

This hybrid approach is intended to achieve better performance with respect to memory. The size of each node may be designed to be limited to fit in a particular *cache line*, the amount of data typically transferred from main memory to a processor's cache. The goal is to improve cache performance while reducing the overhead of a typical linked list. In a typical linked list, every node has a reference (or two in the case of doubly linked lists) which can double the amount of memory required to store elements. By storing more elements in each node, the overall number of nodes is reduced and thus the number of references is reduced. At the same time, an unrolled linked list does not require large chunks of contiguous storage like an array-based list but provides some of the advantages (such as random access within a node). Overall, the goal of an unrolled linked list is to reduce the number of cache misses required to retrieve a particular element.

4.3 Stacks & Queues

Collection data structures such as lists are typically *unstructured*. A list, whether array-based, a linked list, or some other variation simply hold elements in an ordered manner. That is, there is a notion of a first element, second element, etc. However that ordering is not necessarily structured, it is simply the order in which the elements were added to the collection. In contrast, you can *impose* a structured ordering by sorting a list (thus, "sorted" is not the same thing as "ordered"). Sorting a list, however, does not give the data structure itself any more structure. The interface would still allow a user to insert or rearrange the elements so that they are no longer sorted. Sorting a list only change's the collection's *state*, not its behavior.

We now turn our attention to a different kind of collection data structure whose structure is defined by its behavior rather than its state. In particular, we will look at two data structures, stack and queues, that represent *restricted access data structures*. These data structures still store elements, but the general core functionality of a list (the ability to add, retrieve, and remove arbitrary elements) will be restricted in a very particular manner so as to give the data structure behavioral properties. This restriction is built into the data structure as part of the object's interface. That is, users may only interact with the collection in very specific ways. In this manner, structure is imposed through the collection's behavior.

4.3.1 Stacks

A stack is a data structure that stores elements in a last-in, first-out (or Last-In First-Out (LIFO) manner. That is, the last element to be inserted into a stack is the first element that will come out of the stack. A stack data structure can be described as a stack of dishes. When dealing with such a stack, we can add a dish to it, but only at the *top* of the stack lest we risk causing the entire stack of dish to fall and break. Likewise, when removing a dish, we remove the top-most dish rather than pulling a dish from the middle or bottom of the stack for the same reason. Thus, the *last* dish that we added to the stack will be the *first* dish that we take off the stack. It may also be helpful to visualize such a stack of dishes as in a cafeteria where a spring-loaded cart holds the stack. When we add a dish, the entire stack moves down into the cart and when we remove one, the next one "pops" up to the top of the stack.

You are probably already familiar with the concept of stacks in the context of a program's *call stack*. As a program invokes functions, a new stack frame is created that contains all the "local" information (parameters, local variables, etc.) and is placed on top of the call stack. When a function is done executing and returns control back to the calling function the stack frame is removed from the top of the call stack, restoring the stack frame right below it. In this manner, information can be saved and restored from function call to function call efficiently. This idea goes all the way back to the very earliest computers and programming languages (specifically, the Information Processing Language in 1956).

Core Functionality

In order to have achieve the LIFO behavior, access to a stack's elements are restricted through its interface by only allowing two core operations:

- *push* adds a new element to the stop of the stack and
- *pop* removes the element at the top of the stack

The element removed at the top of the stack may also be "returned" in the context of a method call.

Similar to lists, stacks also have several corner cases that need to be considered. First, when implementing a stack you must consider what will happen when a user performs a pop operation on an empty stack. Obviously there is nothing to remove from the stack, but how might we handle this situation? In general, such an invalid operation is referred to as a stack *underflow*. In programming languages that support error handling via exceptions, we could choose to throw an exception. If we were to make this design decisions then we should, in addition, provide a way for the user to check that such an operation may result in an exception by providing a way for them to check if the stack is empty or not (see Secondary Functionality below). Alternatively, we could instead return a "flag" value such as `null` to indicate an empty stack. Though this design decision has consequences as well: we would either need to disallow the user from pushing a `null` value onto the stack (and decide how again to handle that) or we would need to provide a way for the user to distinguish the situation where `null` was actually popped off the stack or was returned because the stack was empty.

In addition, we could design our stack to be either *bounded* or *unbounded*. An unbounded stack means that there would be no practical restrictions on how large the stack could grow. A program's use of our stack would only be limited by the amount of system memory available. A user could continue to push as many elements onto the stack as

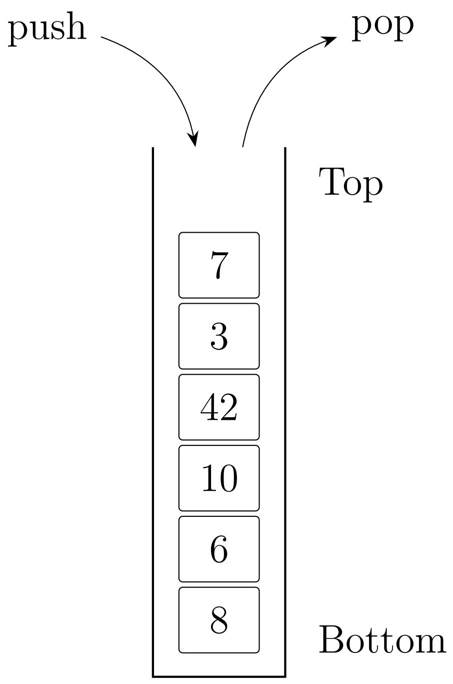


Figure 4.9: A stack holding integer elements. Push and pop operations are depicted as happening at the "top" of the stack. In actuality, a stack stored in a computer's memory is not really oriented but this visualization is consistent with a physical stack growing "upwards."

they like and a problem would only arise when the program or system itself runs out of memory.

A bounded stack means that we could design our stack to have a fixed capacity or limit of (say) n elements. If we went with such a design we would again have to deal with the corner case of a user pushing an element to the stack when it is full referred to as a *stack overflow*. Solutions similar to the popping from an empty stack could be used here as well. We could throw an exception (and give the user the ability to check if the stack is full or not) or we could make such an operation a no-op: we would not push the element to the stack, leaving it as it was before and then report the no operation to the user. Typically a boolean value is used, *true* to indicate that the operation was valid and had some side effect on the stack (the element was added) or *false* to indicate that the operation resulted in no side effects.

Secondary Functionality

To make a stack more versatile it is common to include secondary functionality such as the following.

- A *peek* method that allows a user to access the element at the top of the stack without removing it. The same effect can be achieved with a pop-then-push operation, but it may be more convenient to allow such access directly. This is useful if a particular algorithm needs to make a decision based on what will be popped off the stack next.
- A means to iterate over all the elements in the stack or to allow read-only access arbitrary elements. We would not want to allow arbitrary write access as that would violate the LIFO behavior of a stack and defeat the purpose of using this particular data structure.
- A way to determine how many elements are on the stack and, related whether or not the stack is empty and, if it is bounded, whether or not it is full or its remaining capacity. Such methods would allow the user to use a more defensive-style programming approach and not make invalid operations.
- A way to empty or "clear" the stack of all its elements with a single method call.
- General find or contains methods that would allow the user to determine if a particular element was already in the stack and more generally, if it is, how far down in the stack it is (its "index").

Implementations

A straightforward and efficient implementation for a stack is to simply use a list data structure "under the hood" and to restrict access to it through the stack's interface. The push and pop operations can then be achieved in terms of the list's add and remove operations, taking care that both work from the same "end" of the list. That is, if the push operation adds an element at the beginning of the list, then the pop operation must remove (and return) the element at the beginning of the list as well.

A linked list is ideal for bounded and unbounded stacks as adding and removing from the head of the list are both very efficient operations, requiring only the creation of a new node and the shuffling of a couple of references. There is also no expensive copy-and-expand operation over the life of the stack. For unbounded stacks, the capacity can be constrained by simply checking the size of the underlying list and handling the stack overflow appropriately. If the underlying linked list also keeps track of the tail element, adding and removing from the tail would also be an option.

An array-based list may also be a good choice for bounded stacks if the list is initialized to have a capacity equal to the capacity of the stack so as to avoid any copy-and-expand operations. An array-based list is less than ideal for unbounded stacks as expensive copy-and-expand operations may be common. However, care must be taken to ensure that the push and pop operations are efficient. If we designate the "first" element (the element at index 0) as the top of the stack, we would constantly be shifting elements with each and every push and pop operation which can be quite expensive. Instead, it would be more efficient to keep track of the top element and add elements to the "end" of the array.

In detail, we would keep track of a current index, say top that is initialized to -1 indicating an empty stack. Then as elements are pushed, we would add them to the list at index $(top + 1)$ and increment top. As elements are popped, we return the element at index top and decrement the index. In this manner, each push and pop operation requires a constant number of operations rather than shifting up to n elements.

Applications

As previously mentioned, stacks are used extensively in computer architecture. They are used to keep track of local variables and parameters as functions are called using a program stack. In-memory stacks may also be used to simulate a series of function/method calls in order to avoid using (or misusing) the program stack. A prime example of such a use case is avoiding recursion. Instead of a sequence of function calls that may result in a stack overflow of the program stack (which is generally limited) an in-memory stack data structure, which is generally able to accommodate many more elements, can be used.

Stacks are also the core data structures used in many fundamental algorithms. Stacks are used extensively in algorithms related to parsing and processing of data such as in the Shunting Yard Algorithm used by compilers to evaluate an Abstract Syntax Tree (AST). Stacks are also used in many graph algorithms such as Depth First Search (DFS) and in a preorder processing of binary trees (see Chapter 6). Essentially any application in which something needs to be "tracked" or remembered in a particular LIFO ordering,

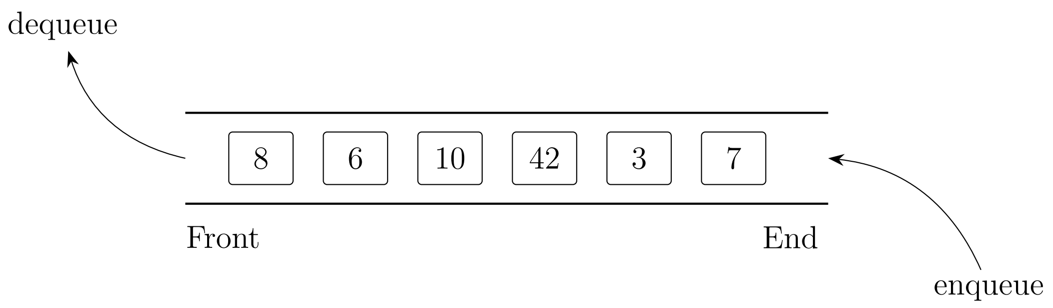


Figure 4.10: An example of a queue. Elements are enqueued at the end of the queue and dequeued from the front of the queue.

a stack is an ideal data structure to use.

4.3.2 Queues

A similar data structure is a *queue* which provides a First-In First-Out (FIFO) ordering of elements. As its name suggests, a queue can be thought of as a line. Elements enter the queue at one end (the "end" of the line) and are removed from the other end (the "front" of the line). Thus, the first element to enter a queue is the first element to be removed from the queue. Each element in the "line" is served in the order in which they entered the line. Similar to a stack, a queue's structure is defined by its interface.

Core Functionality

The two core operations in a queue are:

- *enqueue* which adds an element to the queue at its end and
- *dequeue* which removes the element at the front of the queue

An example of a queue and its operations is depicted in Figure 4.10. Some programming languages and data structure implementations may use different terminology to describe these two operations. Some use the same terms as a stack ("push" would correspond to enqueue and "pop" would correspond to a dequeue operation). However, using the same terminology for fundamentally different data structures is confusing.[2] Some implementations use the terms "offer" (we are offering an element to the queue if it is able to handle it) and "poll" (we are asking or "polling" the queue to see if it has any elements to remove).

[2] `<opinion>and wrong</opinion>`

Secondary Functionality

Secondary functionality is pretty much the same as with a stack. We could choose to make our queue bounded or unbounded and deal with corner cases (enqueuing to a full queue, dequeueing from an empty queue) similarly. In fact, we would probably want to design our queue with the same behavior for consistency. We could include methods to determine the size of the queue, its remaining capacity (if bounded), a way to determine if (and where) a certain element may be in the queue, etc.

As with stacks, in general we would not want to allow a user to arbitrarily insert elements into a queue. Doing so would be allowing "line jumpers" to jump ahead of other elements, violating FIFO. In some applications this does make sense and we discuss these variations in Section 4.3.3. However, we *could* allow arbitrary removal of certain elements. Strictly speaking, this would not violate FIFO as the remaining elements would still be processed in the order in which they were enqueued. This could model situations where those waiting in line got impatient and left (a process or request timed-out for example).

Implementations

The obvious implementation for a queue is a linked list. Since we have to work from both ends, however, our linked list implementation will need to keep track of both the head element and the tail element so that adding a new node at either end has the same constant cost (creating a new node, shuffling some references). If our linked list only keeps track of the head, then to enqueue an element, we would need to traverse the entire list all the way to the end in order to add an element to the tail.

A linked list that offers constant-time add and remove methods to both the head and the tail can be oriented either way. We could add to the head and remove from the tail or we could add to the tail and remove from the head. As long as we are consistently adding to one end and removing from the other, there really is no difference. Our design decision may have consequences, however, on the secondary functionality. If we design the array with an iterator, it makes the most sense to start at the front of the queue and iterate toward the end. If our linked list is a doubly linked list, then we can easily iterate in either direction. However, if our list is singly linked then we need to make sure that the front of our queue corresponds to the head of the list.

An array-based list can also be used to implement a queue. As with stacks, it is most appropriate to use array-based lists when you want a bounded queue so as to avoid expensive expand-and-copy operations. However, you need to be a bit clever in how you enqueue and dequeue items to the array in order to ensure efficient operations.

A naive approach would be to enqueue elements at one end of the array and dequeue them from the front (index 0). However, this would mean we need to shift elements down on every dequeue operation which is potentially very inefficient. A clever workaround would be to keep track of the $front$ and *end* of the queue using two index variables.

Initially, an empty queue would have both of these variables initialized to 0. As we add elements, the *end* index gets incremented (we add left-to-right). As elements are removed, the $front$ index variable gets incremented. At some point, these variables will read the right-end of the array at which point we simply reset them back to zero. The queue is empty when $front = end$ and it is full when $(end - front) \bmod n = n - 1$ where n is the size of the array. The various states of such a queue are depicted in Figure 4.11. Using an array-based list may save a substantial amount of memory. However, the added complexity of implementation (and thus increased opportunities for bugs and errors) may not justify the savings.

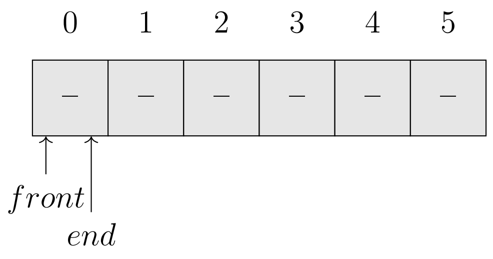
Applications

A queue is ideal for any application in which elements must be stored in order to be handled or processed in a particular order. For example, a queue is a natural data structure to implement buffers in which data is received but cannot be processed immediately (it may not be possible or it would be inefficient to process the data in small chunks). A queue ensures that the data remains in the order in which it was received.

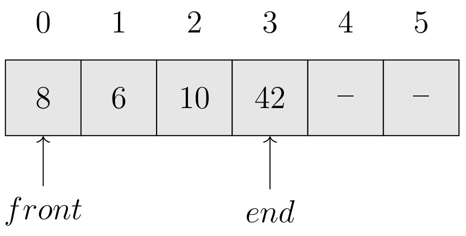
Another typical example is when requests are received and must be processed. This is typical in a webserver for example where requests for resources or webpages are stored in a queue and processed in the order in which they are received. In an operating system, threads (or processes) may be stored in a job queue and then woken/executed.

More generally, queues can facilitate communication in a Producer Consumer Pattern (see Figure 4.12). In this scenario we have independent producers and consumers. Producers may produce requests, tasks, works, or a resource that needs to be processed. For example, a producer may be a web browser requesting a particular web page, or it may be a thread making a request for a resource, etc. Consumers handle or service each of these requests. Each consumer and each producer acts independently and asynchronously. To facilitate thread-safe communication between the two groups, each request is enqueued to a *blocking queue*. As producers enqueue requests, they are stored in the order they are received. Then, as each consumer becomes available to service a request, it *polls* the queue for the next request.

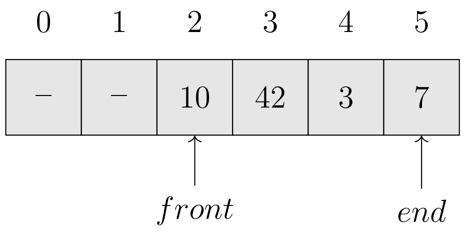
A naive approach would be to do *busy polling* where a consumer keeps polling the queue over and over for another request even if it is empty. Using a thread-safe blocking queue means that we don't do busy polling. Instead, if no request is available for a consumer, the queue *blocks* the consumer (typically, the consumer is a thread and this puts the thread to sleep) until a request becomes available so that it is not continuously polling for more work.



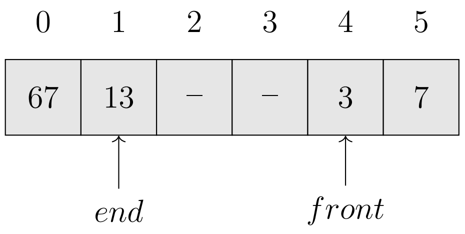
(a) An initially empty queue. Both index variables refer to index 0.



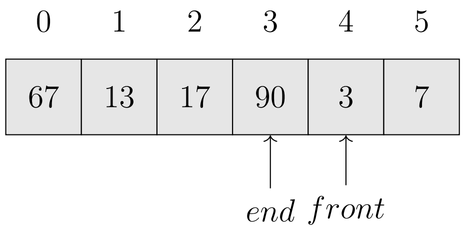
(b) As elements are enqueued the *end* index variable is incremented.



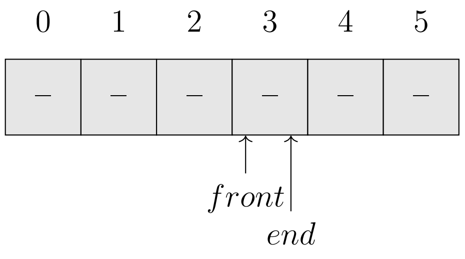
(c) More elements may be enqueued as well as dequeued, moving both index variables.



(d) At some point the index variables wrap around to the beginning of the array and the queue may "straddle" the array.



(e) The queue may also become full, at which point the two index variables are beside each other.



(f) The queue may again become empty and the two index variables refer to the same value, but not necessarily index 0.

Figure 4.11: Various states of an array-based queue. As the index variables, $front$ and *end* become n or larger, they wrap around to the beginning of the array using a modulus operation.

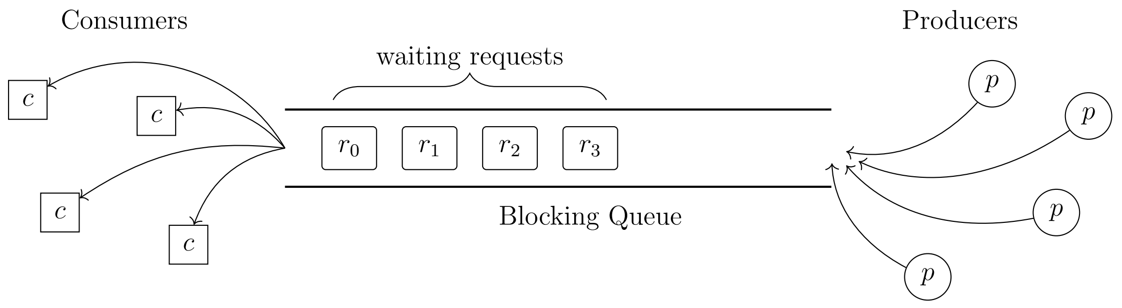


Figure 4.12: Producer Consumer Pattern. Requests (or tasks or resources) are enqueued into a blocking (or otherwise thread safe queue) by producers. Independently (asynchronously) consumers handle requests. Once done, consumers *poll* the queue for another request. If the queue is empty it *blocks* consumers until a new request becomes available.

4.3.3 Variations

In addition to the fundamental stack and queue data structures, there are numerous useful variations. Some simple variations involve allowing the user to perform more than just the two core operations on each.

For example, we could design a double-ended stack which, in addition to the push and pop operations at the top, we could allow a pop operation at the bottom. This provides a variation on the usual bounded stack where the "oldest" elements at the bottom drop out when the stack becomes full instead of rejecting the "newest" items at the top of the queue. A prime example of this is how a typical "undo" operation is facilitated in many applications. Take for example a word processor in which the user performs many actions in sequence (typing, highlighting, copy-paste, delete, etc.). A word processor typically allows the user to undo previous operations but in the reverse sequence that they were performed. However, at the same time we don't want to keep track of *every* change as it would start to take more and more memory and may impact performance. Using a double-ended stack means that the oldest action performed is removed at the bottom, allowing the last n operations to be tracked.

Relatedly, we could design a queue in which we are allowed to enqueue elements at *either* end, but only remove from one. This would allow the aforementioned "line jumpers" to jump ahead to the front of the line. This establishes a sort-of "fast lane" for high priority elements. As an example, consider system processes in an operating system waiting for their time slice to execute. User processes may be preempted by system-level processes as they have a high priority. More generally, we can establish more than two levels of priority (high/low) and build what is known as a *priority queue* (see below).

Deques

A logical extension is to allow insert and remove operations at both ends. Such a generalized data structure is known as a *deque* (pronounced “deck”, also called a double-ended queue). In fact, the primary advantage to creating this generalized data structure is that it can be used as a stack, queue, etc. all with only a single implementation.

In fact in Java (as of version 6), it is recommended to use its `java.util.Deque<E>` interface rather than the stack and queue implementations. The `Deque<E>` interface is extremely versatile, offering several different versions of each of the insert and remove methods to both ends (head and tail). One set of operations will throw exceptions for corner cases (inserting into a full deque and removing from an empty deque) and another set will return special values (`null` or `false`). Java also provides several different implementations including an array-based deque (`ArrayDeque<E>`) and its usual `LinkedList<E>` implementation as well as several concurrent and thread-safe versions.

Priority Queues

Another useful variation is a priority queue in which elements are stored not in a FIFO manner, but with respect to some priority. When elements are dequeued, the highest priority element is removed from the queue first. When elements are enqueued, conceptually they are placed in the queue according to their priority. This may mean that the new element jumps ahead to the front of the queue (if it has a priority higher than all other elements) or it may mean it ends up at the end of the queue (if it has the lowest priority) or somewhere in between. In general, any scheme can be used to define priority but it is typical to use integer values.

A naive implementation of a priority queue would implement the enqueue operation by making comparisons and inserting the new element at the appropriate spot, requiring up to n comparisons/operations in a queue with n elements. There are much better implementations that we’ll look at later on (in particular, a heap implementation, see Chapter 6. Again, most programming languages will have a built-in implementation. Java provides an efficient heap-based `PriorityQueue<E>` implementation. Priority is defined using either a natural ordering or a custom `Comparator` object.

```java
public class ArrayList<T> {

  private T[] arr;
  private int size;

  public ArrayList() {
    this.arr = (T[]) new Object[10];
    this.size = 0;
  }

  public T getElement(int index) {

    if(index < 0 || index >= this.size) {
      throw new IndexOutOfBoundsException("invalid index: " + index);
    }
    return this.arr[index];
  }

  public void removeAtIndex(int index) {

    if(index < 0 || index >= size) {
      throw new IndexOutOfBoundsException("invalid index: " + index);
    }

    for(int i=index; i<size-1; i++) {
      this.arr[i] = this.arr[i+1];
    }
    this.size--;
  }

  public void insertAtIndex(T x, int index) {
    if(index < 0 || index > size) {
      throw new IndexOutOfBoundsException("invalid index: " + index);
    }

    if(this.size == arr.length) {
      this.arr = Arrays.copyOf(this.arr, this.arr.length + 10);
    }

    for(int i=this.size-1; i>=index; i--) {
      this.arr[i+1] = this.arr[i];
    }
    this.arr[index] = element;
    this.size++;
  }

  public void addAtEnd(T x) {
    this.insertAtIndex(x, this.size);
  }
}
```

Code Sample 4.1: Parameterized Array-Based List in Java

```java
public class Node<T> {

  private final T item;
  private Node<T> next;

  public Node(T item) {
    this.item = item;
    next = null;
  }

  //getters and setters omitted

  public boolean hasNext() {
    return (this.next == null);
  }

}
```

Code Sample 4.2: A linked list node Java implementation. Getter and setter methods have been omitted for readability. A convenience method to determine if a node has a next element is included. This implementation uses `null` as its terminating value.

5 Algorithm Analysis

5.1 Introduction

An *algorithm* is a procedure or description of a procedure for solving a problem. An algorithm is a step-by-step specification of operations to be performed in order to compute an output, process data, or perform a function. An algorithm must always be *correct* (it must always produce a valid output) and it must be *finite* (it must terminate after a finite number of steps).

Algorithms are not code. Programs and code in a particular language are *implementations* of algorithms. The word, "algorithm" itself is derived from the latinization of Abū ‘Abdalāh Muhammad ibn Mūsā al-Khwārizmī, a Persian mathematician (c. 780 – 850). The concept of algorithms predates modern computers by several thousands of years. Euclid's algorithm for computing the greatest common denominator (see Section 5.5.3) is 2,300 years old.

Often, to be useful an algorithm must also be *feasible*: given its input, it must execute in a reasonable amount of time using a reasonable amount of resources. Depending on the application requirements our tolerance may be on the order of a few milliseconds to several days. An algorithm that takes years or centuries to execute is certainly not considered feasible.

Deterministic An algorithm is deterministic if, when given a particular input, will always go through the exact same computational process and produce the same output. Most of the algorithms you've used up to this point are deterministic.

Randomized An algorithm that is randomized is an algorithm that involves some form of random input. The random source can be used to make decisions such as random selections or to generate random state in a program as candidate solutions. There are many types of randomized algorithms including Monte-Carlo algorithms (that may have some error with low probability), Las Vagas algorithms (whose results are always correct, but may fail with a certain probability to produce any results), etc.

Optimization Many algorithms seek not only to find a solution to a problem, but to find the *best*, optimal solution. Many of these type of algorithms are *heuristics*: rather than finding the actual best solution (which may be infeasible), they can approximate a solution (*Approximation* algorithms). Other algorithms simulate biological processes (Genetic algorithms, Ant Colony algorithms, etc.) to search for an optimal solution.

Parallel Most modern processors are multicore, meaning that they have more than one processor on a chip. Many servers have dozens of processors that work together. Multiple processors can be utilized by designing parallel algorithms that can split work across multiple processes or threads which can be executed in parallel to each other, improving overall performance.

Distributed Computation can also be distributed among completely separate devices that may be located half way across the globe. Massive distributed computation networks have been built for research such as simulating protein folding (Folding@Home).

An algorithm is a more abstract, generalization of what you might be used to in a typical programming language. In an actual program, you may have functions/methods, subroutines or procedures, etc. Each one of these pieces of code could be considered an algorithm in and of itself. The combination of these smaller pieces create more complex algorithms, etc. A program is essentially a concrete implementation of a more general, theoretical algorithm.

When a program executes, it expends some amount of resources. For example:

Time The most obvious resource an algorithm takes is time: how long the algorithm takes to finish its computation (measured in seconds, minutes, etc.). Alternatively, time can be measured in how many CPU cycles or floating-point operations a particular piece of hardware takes to execute the algorithm.

Memory The second major resource in a computer is memory. An algorithm requires memory to store the input, output, and possibly extra memory during its execution. How much memory an algorithm uses in its execution may be even more of an important consideration than time in certain environments or systems where memory is extremely limited such as embedded systems.

Power The amount of power a device consumes is an important consideration when you have limited capacity such as a battery in a mobile device. From a consumer's perspective, a slower phone that offered twice the battery life may be preferable. In certain applications such as wireless sensor networks or autonomous systems power may be more of a concern than either time or memory.

Bandwidth In computer networks, efficiency is measured by how much data you can transmit from one computer to another, called *throughput*. Throughput is generally limited by a network's bandwidth: how much a network connection can transmit under ideal circumstances (no data loss, no retransmission, etc.)

Circuitry When designing hardware, resources are typically measured in the number of gates or wires are required to implement the hardware. Fewer gates and wires means you can fit more chips on a silicon die which results in cheaper hardware. Fewer wires and gates also means faster processing.

Idleness Even when a computer isn't computing anything, it can still be "costing" you something. Consider purchasing hardware that runs a web server for a small user base. There is a substantial investment in the hardware which requires maintenance and eventually must be replaced. However, since the user base is small, most of the time it sits idle, consuming power. A better solution may be to use the same hardware to serve multiple virtual machines (VMs). Now several small web serves can be served with the same hardware, increasing our utilization of the hardware. In scenarios like this, the lack of work being performed is the resource.

Load Somewhat the opposite of idleness, sometimes an application or service may have occasional periods of high demand. The ability of a system to service such high *loads* may be considered a resource, even if the capacity to handle them goes unused most of the time.

These are all very important engineering and business considerations when designing systems, code, and algorithms. However, we'll want to consider the complexity of algorithms in a more abstract manner.

Suppose we have two different programs (or algorithms) A and B. Both of those algorithms are correct, but A uses fewer of the above resources than B. Clearly, algorithm A is the better, more *efficient* solution. However, how can we better quantify this efficiency?

List Operations

To give a concrete example, consider the list data structures from Chapter 4. The list could be implemented as an array-based list (where the class owns a static array that is resized/copied when full) or a linked list (with nodes containing elements and linking to the next node in the list). Some operations are "cheap" on one type of list while other operations may be more "expensive."

Consider the problem of inserting a new element into the list at the beginning (at index 0). For a linked list this involves creating a new node and shuffling a couple of references. The number of operations in this case is not contingent on the *size* of the the list. In contrast, for an array-based list, if the list contains n elements, each element will need to be *shifted* over one position in the array in order to make room for the element to be inserted. The number of shifts is proportional to the number of elements in the array, n. Clearly for this operation, a linked list is better (more efficient).

Now consider a different operation: given an index i, retrieve the i-th element in the list. For an array-based list we have the advantage of having random access to the array. When we index an element, `arr[i]`, it only takes one memory address computation to "jump" to the memory location containing the i-th element. In contrast, a linked list would require us to start at the head, and traverse the list until we reach the i-th node. This requires i traversal operations. In the worst case, retrieving the last element, the n-th element, would require n such operations. A summary of these operations can be

List Type	Insert at start	Index-based Retrieve
Array-based List	n	1
Linked List	2	$i \approx n$

Table 5.1: Summary of the Complexity of List Operations

```java
1 public static int sum(List<Integer> items) {
2
3   int total = 0;
4   for(int i=0; i<items.size(); i++) {
5     total += items.get(i);
6   }
7   return total;
8 }
```

Code Sample 5.1: Summing a collection of integers

found in Table 5.1.

We will now demonstrate the consequences of this difference in performance for an index-based retrieval operation. Consider the Java code in Code Sample 5.1. This method is using a naive index-based retrieval (line 5) to access each element.

Suppose that the `List` passed to this method is an `ArrayList` in which each `get(i)` method call take a constant number of operations and thus roughly a constant amount of time. Since we perform this operation once for each element in the list, we can assume that the amount of time that the entire algorithm will take will be proportional to n, the number of elements in the list. That is, the time t that the algorithm will take to execute will be some linear function of n, say

$$t_{lin}(n) = an + b$$

Here, the constants a and b are placeholders for some values that are dependent on the particular machine that we run it on. These two values may vary depending on the speed of the machine, the available memory, etc. and will necessarily change if we run the algorithm on a different machine with different specifications. However, the overall complexity of the algorithm, the fact that it takes a linear amount of work to execute, will not change.

Now let's contrast the scenario where a `LinkedList` is passed to this method instead. On each iteration, each `get(i)` method will take i node traversals to retrieve the i-th element. On the first iteration, only a single node traversal will be required. On the second, 2 traversals are required, etc. Adding all of these operations together gives us the following

$$1 + 2 + 3 + 4 + \cdots + (n-1) + n$$

That is, the sum of natural numbers 1 up to n. We can use the well-known Gauss's Formula to get a closed form of this summation.

Theorem 1 (Gauss's Formula)**.**

$$\sum_{i=1}^{n} i = \frac{n^2 + n}{2}$$

Thus, the total number of operations will be proportional to some *quadratic* function,

$$t_{quad}(n) + an^2 + bn + c$$

Again, we don't know what the constants, a, b, c are as they may vary depending on the languages, system and other factors. However, we can compute them experimentally.

The code in Code Sample was run on a laptop and its performance was timed. The experiment generated random lists of various sizes, starting at $n = 50{,}000$ up to 1 million by increments of 50,000. The time to execute was recorded in seconds. The experiment was repeated for both types of lists multiple times and an average was taken to reduce the influence of other factors (such as other processes running on the machine at the same time).

To find the coefficients in both functions, a linear and quadratic regression was performed, giving us the following functions

$$\begin{aligned} t_{lin}(n) &= 5.138\text{e}{-5}n + 0.004 \\ t_{quad}(n) &= 6.410\text{e}{-4}n^2 - 0.112n + 9.782 \end{aligned}$$

Both had very high correlation coefficients (the quadratic regression was nearly perfect at 0.994) which gives strong empirical evidence for our theoretical analysis. The quadratic regression along with the experimental data points is given in Figure 5.1. The linear data is not graphed as it would remain essentially flat on this scale.

These experimental results allow us to estimate and project how this code will perform on larger and larger lists. For example, we can plug $n = 10$ million into $t_{lin}(n)$ and find that it will still take less than 1 second for the method to execute. Likewise, we can find out how long it would take for the linked list scenario. For a list of the same size, $n = 10$ million, the algorithm would take 17.498 *hours*! More values are depicted in Table 5.2.

The contrast between the two algorithms gets extreme even with moderately large lists of integers. With 10 billion numbers, the quadratic approach is not even feasible. Arguably, even at 10 million, a run time of over 17 hours is not acceptable, especially when an alternative algorithm can perform the same calculation in less than 1 second.

Being able to identify the complexity of algorithms and the data structures they use in order to avoid such inefficient solutions is an essential skill in Computers Science.[1] In the following examples, we'll begin to be a little bit more formal about this type of analysis.

[1]Using an index-based iteration is a common mistake in Java code. The proper solution would have been to use an enhanced for-loop or iterator pattern so that the method would perform the same on either array-based or linked lists.

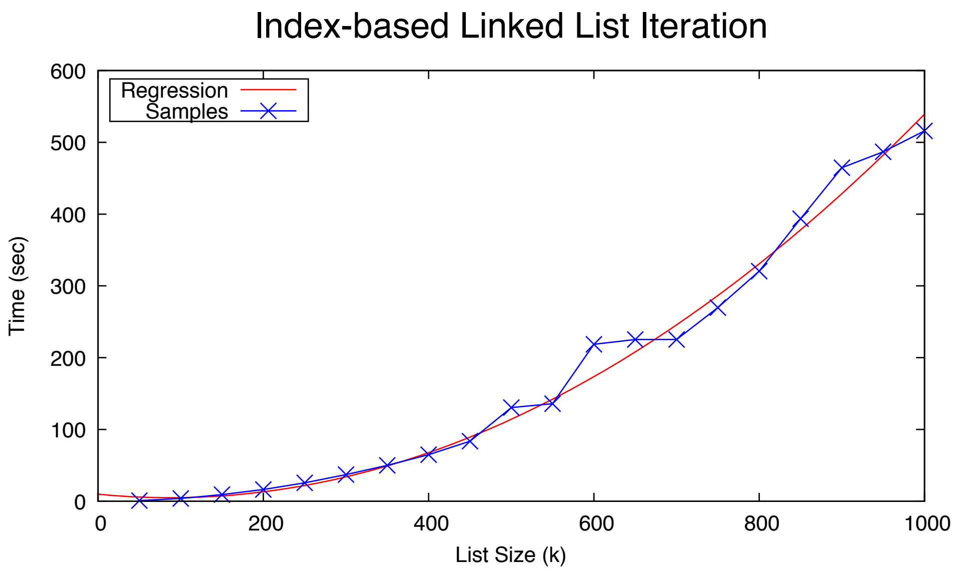


Figure 5.1: Quadratic Regression of Index-Based Linked List Performance.

Table 5.2: Various Projected Runtimes for Code Sample 5.1.

List Size	Execution Time	
	Linear	Quadratic
10 million	1 second	17.498 hours
100 million	10 seconds	74.06 days
1 billion	2 minutes	20.32 years
10 billion	8.56 minutes	2031.20 years

```
int result = 0;
for(int i=1; i<=n; i++) {
  for(int j=1; j<=i; j++) {
    result = result + 1;
  }
}
```

Code Sample 5.2: Summation Algorithm 1

```
int result = 0;
for(int i=1; i<=n; i++) {
  result = result + i;
}
```

Code Sample 5.3: Summation Algorithm 2

5.1.1 Example: Computing a Sum

The following is a toy example, but its easy to understand and straightforward. Consider the following problem: given an integer $n \geq 0$, we want to compute the arithmetic series,

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n$$

As a naive approach, consider the algorithm in Code Sample 5.2. In this algorithm, we iterate over each possible number i in the series. For each number i, we count 1 through i and add one to a result variable.

As an improvement, consider the algorithm in Code Sample 5.3. Instead of just adding one on each iteration of the inner loop, we omit the loop entirely and simply just add the index variable i to the result.

Can we do even better? Yes. Recall Theorem 1 that gives us a direct closed form solution for this summation:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Code Sample 5.4 uses this formula to directly compute the sum without any loops.

```
int result = n * (n + 1) / 2;
```

Code Sample 5.4: Summation Algorithm 3

Algorithm	Number of Additions	Input Size					
		10	100	1,000	10,000	100,000	1,000,000
1	$\approx n^2$	0.003ms	0.088ms	1.562ms	2.097ms	102.846ms	9466.489ms
2	n	0.002ms	0.003ms	0.020ms	0.213ms	0.872ms	1.120ms
3	1	0.002ms	0.001ms	0.001ms	0.001ms	0.001ms	0.000ms

Table 5.3: Empirical Performance of the Three Summation Algorithms

All three of these algorithms were run on a laptop computer for various values of n from 10 up to 1,000,000. Table 5.3 contains the resulting run times (in milliseconds) for each of these three algorithms on the various input sizes.

With small input sizes, there is almost no difference between the three algorithms. However, that would be a naive way of analyzing them. We are more interested in how each algorithm performs as the input size, n increases. In this case, as n gets larger, the differences become very stark. The first algorithm has two nested for loops. On average, the inner loop will run about $\frac{n}{2}$ times while the outer loop runs n times. Since the loops are nested, the inner loop executes about $\frac{n}{2}$ times *for each* iteration of the outer loop. Thus, the total number of iterations, and consequently the total number of additions is about

$$n \times \frac{n}{2} \approx n^2$$

The second algorithm saves the inner for loop and thus only makes n additions. The final algorithm only performs a constant number of operations.

Observe how the running time grows as the input size grows. For Algorithm 1, increasing n from 100,000 to 1,000,000 (10 times as large) results in a running time that is about 100 times as slow. This is because it is performing n^2 operations. To see this, consider the following. Let $t(n)$ be the time that Algorithm 1 takes for an input size of n. From before we know that

$$t(n) \approx n^2$$

Observe what happens when we increase the input size from n to $10n$:

$$t(10n) \approx (10n)^2 = 100n^2$$

which is 100 times as large as $t(n)$. The running time of Algorithm 1 will grow quadratically with respect to the input size n.

Similarly, Algorithm 2 grows linearly,

$$t(n) \approx n$$

Thus, a 10 fold increase in the input,

$$t(10n) \approx 10n$$

```java
public static int mode01(int arr[]) {

  int maxCount = 0;
  int modeIndex = 0;
  for(int i=0; i<arr.length; i++) {
    int count = 0;
    int candidate = arr[i];
    for(int j=0; j<arr.length; j++) {
      if(arr[j] == candidate) {
        count++;
      }
    }
    if(count > maxCount) {
      modeIndex = i;
      maxCount = count;
    }
  }
  return arr[modeIndex];
}
```

Code Sample 5.5: Mode Finding Algorithm 1

leads to a 10 fold increase in the running time. Algorithm 3's runtime does not depend on the input size, and so its runtime does not grow as the input size grows. It essentially remains flat–constant.

Of course, the numbers in Table 5.3 don't follow this trend exactly, but they are pretty close. The actual experiment involves a lot more variables than just the algorithms: the laptop may have been performing other operations, the compiler and language may have optimizations that change the algorithms, etc. Empirical results only provide general evidence as to the runtime of an algorithm. If we moved the code to a different, faster machine or used a different language, etc. we would get different numbers. However, the general trends in the rate of growth *would* hold. Those rates of growth will be what we want to analyze.

5.1.2 Example: Computing a Mode

As another example, consider the problem of computing the *mode* of a collection of numbers. The mode is the most common element in a set of data.[2]

[2] In general there may be more than one mode, for example in the set $\{10, 20, 10, 20, 50\}$, 10 and 20 are both modes. The problem will simply focus on finding *a* mode, not all modes.

```java
1  public static int mode02(int arr[]) {
2    Arrays.sort(arr);
3    int i=0;
4    int modeIndex = 0;
5    int maxCount = 0;
6    while(i < arr.length-1) {
7      int count=0;
8      while(i < arr.length-1 && arr[i] == arr[i+1]) {
9        count++;
10       i++;
11     }
12     if(count > maxCount) {
13       modeIndex = i;
14       maxCount = count;
15     }
16     i++;
17   }
18   return arr[modeIndex];
19 }
```

Code Sample 5.6: Mode Finding Algorithm 2

Consider the strategy as illustrated in Code Sample 5.5. For each element in the array, we iterate through all the other elements and count how many times it appears (its *multiplicity*). If we find a number that appears more times than the candidate mode we've found so far, we update our variables and continue. As with the previous algorithm, the nested nature of our loops leads to an algorithm that performs about n^2 operations (in this case, the comparison on line 9).

Now consider the following variation in Code Sample 5.6. In this algorithm, the first thing we do is *sort* the array. This means that all equal elements will be contiguous. We can exploit this to do less work. Rather than going through the list a second time for each possible mode, we can count up contiguous runs of the same element. This means that we need only examine each element exactly once, giving us n comparison operations (line 8).

We can't, however, ignore the fact that to exploit the ordering, we needed to first "invest" some work upfront by sorting the array. Using a typical sorting algorithm, we would expect that it would take about $n \log(n)$ comparisons. Since the sorting phase and mode finding phase were separate, the total number of comparisons is about

$$n \log(n) + n$$

The highest order term here is the $n \log(n)$ term for sorting. However, this is still lower than the n^2 algorithm. In this case, the investment to sort the array pays off! To compare

```java
public static int mode03(int arr[]) {
  Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
  for(int i=0; i<arr.length; i++) {
    Integer count = counts.get(arr[i]);
    if(count == null) {
      count = 0;
    }
    count++;
    counts.put(arr[i], count);
  }
  int maxCount = 0;
  int mode = 0;
  for(Entry<Integer, Integer> e : counts.entrySet()) {
    if(e.getValue() > maxCount) {
      maxCount = e.getValue();
      mode = e.getKey();
    }
  }
  return mode;
}
```

Code Sample 5.7: Mode Finding Algorithm 3

with our previous analysis, what happens when we increase the input size 10 fold? For simplicity, let's only consider the highest order term:

$$t(n) = n \log{(n)}$$

Then

$$t(10n) = 10n \log{(10n)} = 10n \log{(n)} + 10n \log{(10)}$$

Ignoring the lower order term, the increase in running time is essentially linear! We cannot discount the additive term in general, but it is so close to linear that terms like $n \log{(n)}$ are sometimes referred to as *quasilinear*.

Yet another solution, presented in Code Sample 5.7, utilizes a map data structure to compute the mode. A map is a data structure that allows you to store key-value pairs. In this case, we map elements in the array to a counter that represents the element's multiplicity. The algorithm works by iterating over the array and entering/updating the elements and counters.

There is some cost associated with inserting and retrieving elements from the map, but this particular implementation offers *amortized* constant running time for these operations. That is, some particular entries/retrievals may be more expensive (say

Algorithm	Number of Additions	Input Size					
		10	100	1,000	10,000	100,000	1,000,000
1	$\approx n^2$	0.007ms	0.155ms	11.982ms	45.619ms	3565.570ms	468086.566ms
2	n	0.143ms	0.521ms	2.304ms	19.588ms	40.038ms	735.351ms
3	n	0.040ms	0.135ms	0.703ms	10.386ms	21.593ms	121.273ms

Table 5.4: Empirical Performance of the Three Mode Finding Algorithms

linear), but when averaged over the life of the algorithm/data structure, each operation only takes a constant amount of time.

Once built, we need only go through the elements in the map (at most n) and find the one with the largest counter. This algorithm, too, offers essentially linear runtime for all inputs. Similar experimental results can be found in Table 5.4.

The difference in performance is even more dramatic than in the previous example. For an input size of 1,000,000 elements, the n^2 algorithm took nearly *8 minutes*! This is certainly unacceptable performance for most applications. If we were to extend the experiment to $n = 10{,}000{,}000$, we would expect the running time to increase to about *13 hours*! For perspective, input sizes in the millions are *small* by today's standards. Algorithms whose runtime is quadratic are *not* considered feasible for today's applications.

5.2 Pseudocode

We will want to analyze algorithms in an abstract, general way independent of any particular hardware, framework, or programming language. In order to do this, we need a way to specify algorithms that is also independent of any particular language. For that purpose, we will use *pseudocode*.

Pseudocode ("fake" code) is similar to some programming languages that you're familiar with, but does not have any particular syntax rules. Instead, it is a higher-level description of a process. You may use familiar control structures such as loops and conditionals, but you can also utilize natural language descriptions of operations.

There are no established rules for pseudocode, but in general, good pseudocode:

- Clearly labels the algorithm
- Identifies the input and output at the top of the algorithm
- Does not involve any language or framework-specific syntax–no semicolons, declaration of variables or their types, etc.
- Makes liberal use of mathematical notation and natural language for clarity

Good pseudocode abstracts the algorithm by giving enough details necessary to understand the algorithm and subsequently implement it in an actual programming language. Let's look at some examples.

```
INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
OUTPUT : The *mean*, $\mu$ of the values in $A$
$sum \leftarrow 0$
FOREACH $a_i \in A$ DO
    $sum \leftarrow sum + a_i$
END
$\mu \leftarrow \frac{sum}{n}$
output $\mu$
```

Algorithm 5: Computing the Mean

Algorithm 5 describes a way to compute the average of a collection of numbers. Observe:

- The input does not have a specific *type* (such as `int` or `double`), it uses set notation which also indicates how large the collection is.
- There is no language-specific syntax such as semicolons, variable declarations, etc.
- The loop construct doesn't specify the details of incrementing a variable, instead using a "foreach" statement with some set notation[3]
- Code blocks are not denoted by curly brackets, but are clearly delineated by using indentation and vertical lines.
- Assignment and compound assignment operators do not use the usual syntax from C-style languages, instead using a left-oriented arrow to indicate a value is assigned to a variable.[4]

Consider another example of computing the mode, similar to the second approach in a previous example.

Some more observations about Algorithm 6:

- The use of natural language to specify that the collection should be sorted and in what order
- The usage of $-\infty$ as a placeholder so that any other value would be greater than it
- The use of natural language to specify that an iteration takes place over contiguous elements (line 3) or that a sub-operation such as a count/summation (line 4) is performed

[3] To review, $a_i \in A$ is a predicate meaning the element a_i is *in* the set A.
[4] Not all languages use the familiar single equals sign `=` for the assignment operator. The statistical programming language R uses the left-arrow operator, `<-` and Maple uses `:=` for example.

```
   INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
   OUTPUT : A *mode* of $A$
 1 Sort the elements in $A$ in non-decreasing order
 2 $multiplicity \leftarrow -\infty$
 3 FOREACH *run of contiguous equal elements* $a$ DO
 4     $m \leftarrow$ count up the number of times $a$ appears
 5     IF $m > multiplicity$ THEN
 6         $mode \leftarrow a$
 7         $multiplicity \leftarrow m$
 8     END
 9 END
10 output $m$
```

Algorithm 6: Computing the Mode

In contrast, bad pseudocode would have the opposite elements. Writing a full program or code snippet in Java for example. Bad pseudocode may be unclear or it may overly simplify the process to the point that the description is trivial. For example, suppose we wanted to specify a sorting algorithm, and we did so using the pseudocode in Algorithm 7. This trivializes the process. There are many possible sorting algorithms (insertion sort, quick sort, etc.) but this algorithm doesn't specify *any* details for how to go about sorting it.

On the other hand, in Algorithm 6, we *did* essentially do this. In that case it was perfectly fine: sorting was a side operation that could be achieved by a separate algorithm. The point of the algorithm was not to specify how to sort, but instead how sorting could be used to solve another problem, finding the mode.

```
   INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
   OUTPUT : $A'$, sorted in non-decreasing order
 1 $A' \leftarrow$ Sort the elements in $A$ in non-decreasing order
 2 output $A'$
```

Algorithm 7: Trivial Sorting (Bad Pseudocode)

Another example would be if we need to find a minimal element in a collection. Trivial pseudocode may be like that found in Algorithm 8. No details are presented on *how* to find the element. However, if finding the minimal element were an operation used in a larger algorithm (such as selection sort), then this terseness is perfectly fine. If the primary purpose of the algorithm is to find the minimal element, then details *must* be presented as in Algorithm 9.

```
INPUT   : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
OUTPUT  : The minimal element of $A$
1 $m \leftarrow$ minimal element of $A$
2 output $m$
```

Algorithm 8: Trivially Finding the Minimal Element

```
INPUT   : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
OUTPUT  : The minimal element of $A$
1 $m \leftarrow \infty$
2 FOREACH $a_i \in A$ DO
3     IF $a_i < m$ THEN
4         $m \leftarrow a_i$
5     END
6 END
7 output $m$
```

Algorithm 9: Finding the Minimal Element

5.3 Analysis

Given two competing algorithms, we could empirically analyze them like we did in previous examples. However, it may be infeasible to implement both just to determine which is better. Moreover, by analyzing them from a more abstract, theoretical approach, we have a better more mathematically-based *proof* of the relative complexity of two algorithm.

Given an algorithm, we can analyze it by following this step-by-step process.

1. Identify the input
2. Identify the input size, n
3. Identify the *elementary operation*
4. Analyze how many times the elementary operation is executed with respect to the input size n
5. Characterize the algorithm's complexity by providing an asymptotic (Big-O, or Theta) analysis

Identifying the Input

This step is pretty straightforward. If the algorithm is described with good pseudocode, then the input will already be identified. Common types of inputs are single numbers, collections of elements (lists, arrays, sets, etc.), data structures such as graphs, matrices, etc.

However, there may be some algorithms that have *multiple* inputs: two numbers or a collection and a key, etc. In such cases, it simplifies the process if you can, without loss of generality, restrict attention to a single input value, usually the one that has the most relevance to the elementary operation you choose.

Identifying the Input Size

Once the input has been identified, we need to identify its size. We'll eventually want to characterize the algorithm as a function $f(n)$: given an input size, how many resources does it take. Thus, it is important to identify the number corresponding to the domain of this function.

This step is also pretty straightforward, but may be dependent on the type of input or even its representation. Examples:

- For collections (sets, lists, arrays), the most natural is to use the number of elements in the collection (cardinality, size, etc.). The size of individual elements is not as important as number of elements since the size of the collection is likely to grow more than individual elements do.
- An $n \times m$ matrix input could be measured by one or both nm of its dimensions.
- For graphs, you could count either the number of vertices or the number of edges in the graph (or both!). How the graph is represented may also affect its input size (an adjacency matrix vs. an adjacency list).
- If the input is a number x, the input size is typically the number of bits required to represent x. That is,
  $$n \approx \log_2(x)$$
  To see why, recall that if you have n bits, the maximum unsigned integer you can represent is $2^n - 1$. Inverting this expression gives us $\lceil \log_2(x+1) \rceil$.

Some algorithms may have multiple inputs. For example, a collection and a number (for searching) or two integers as in Euclid's algorithm. The general approach to analyzing such algorithms is to simplify things by only considering *one* input. If one of the inputs is larger, such as a collection vs. a single element, the larger one is used in the analysis. Even if it is not clear which one is larger, it may be possible to assume, without loss of generality, that one is larger than the other (and if not, the inputs may be switched). The input size can then be limited to one variable to simplify the analysis.

Identifying the Elementary Operation

We also need to identify what part of the algorithm does the actual work (where the most resources will be expended). Again, we want to keep the analysis simple, so we generally only identify one *elementary operation*. There may be several reasonable candidates for the elementary operation, but in general it should be the most common or most expensive operation performed in the algorithm. For example:

- When performing numeric computations, arithmetic operations such as additions, divisions, etc.
- When sorting or searching, comparisons are the most natural elementary operations. Swaps may also be a reasonable choice depending on how you want to analyze the algorithm.
- When traversing a data structure such as a linked list, tree, or graph a node traversal (visiting or processing a node) may be considered the elementary operation.

In general, operations that are necessary to control structures (such as loops, assignment operators, etc.) are not considered good candidates for the elementary operation. An extended discussion of this can be found in Section 5.6.2.

Analysis

Once the elementary operation has been identified, the algorithm must be analyzed to count the number of times it is executed with respect to the input size. That is, we analyze the algorithm to find a function $f(n)$ where n is the input size and $f(n)$ gives the number of times the elementary operation is executed.

The analysis may involve deriving and solving a summation. For example, if the elementary operation is performed within a for loop and the loop runs a number of times that depends on the input size n.

If there are multiple loops in which the elementary operation is performed, it may be necessary to setup multiple summations. If two loops are separate and independent (one executes *after* the other), then the *sum rule* applies. The total number of operations is the sum of the operations of each loop.

If two loops are nested, then the *product rule* applies. The inner loop will execute fully *for each* iteration of the outer loop. Thus, the number of operations are multiplied with each other.

Sometimes the analysis will not be so clear cut. For example, a while loop may execute until some condition is satisfied that does not directly depend on the input size but also on the nature of the input. In such cases, we can simplify our analysis by considering the *worst-case* scenario. In the while loop, what is the *maximum* possible number of iterations for any input?

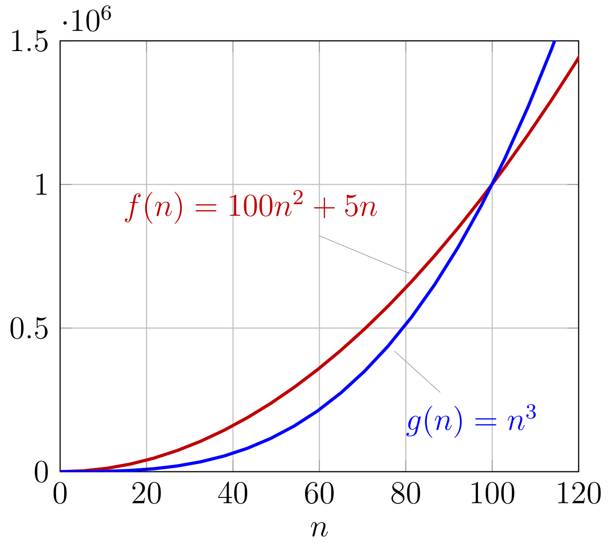


Figure 5.2: Plot of two functions.

Asymptotic Characterization

As computers get faster and faster and resources become cheaper, they can process more and more information in the same amount of time. However, the characterization of an algorithm should be *invariant* with respect to the underlying hardware. If we run an algorithm on a machine that is twice as fast, that doesn't mean that the algorithm has improved. It still takes the same *number* of operations to execute. Faster hardware simply means that the time it takes to execute those operations is half as much as it was before.

To put it in another perspective, performing Euclid's algorithm to find the GCD of two integers took the same number of steps 2,300 years ago when he performed them on paper as it does today when they are executed on a digital computer. A computer is obviously faster than Euclid would have been, but both Euclid and the computer are performing the same number of steps when executing the same algorithm.

For this reason, we characterize the number of operations performed by an algorithm using *asymptotic analysis*. Improving the hardware by a factor of two only affects the "hidden constant" sitting outside of the function produced by the analysis in the previous step. We want our characterization to be invariant of those constants.

Moreover, we are really more interested in how our algorithm performs for larger and larger input sizes. To illustrate, suppose that we have two algorithms, one that performs

$$f(n) = 100n^2 + 5n$$

operations and one that performs

$$g(n) = n^3$$

operations. These functions are graphed in Figure 5.2. For inputs of size less than 100, the first algorithm performs *worse* than the second (the graph is higher indicating "more" resources). However, for inputs of size greater than 100, the first algorithm is better. For small inputs, the second algorithm may be better, but small inputs are not the norm for any "real" problems.[5] In any case, on modern computers, we would expect small inputs to execute fast anyway as they did in our empirical experiments in Section 5.1.1 and 5.1.2. There was essentially no discernible difference in the three algorithms for sufficiently small inputs.

We can rigorously quantify this by providing an asymptotic characterization of these functions. An asymptotic characterization essentially characterizes the *rate of growth* of a function or the *relative* rate of growth of functions. In this case, n^3 grows much faster than $100n^2 + 5n$ as n grows (tends toward infinity). We formally define these concepts in the next section.

5.4 Asymptotics

5.4.1 Big-O Analysis

We want to capture the notion that one function grows faster than (or at least as fast as) another. Categorizing functions according to their growth rate has been done for a long time in mathematics using *big-O notation*.[6]

Definition 1. Let f and g be two functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in O(g(n))$$

read as "f is big-O of g," if there exist constants $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

First, let's make some observations about this definition.

- The "O" originally stood for "order of", Donald Knuth referred to it as the capital greek letter omicron, but since it is indistinguishable from the Latin letter "O" it makes little difference.

[5]There are problems where we can apply a "hybrid" approach: we can check for the input size and choose one algorithm for small inputs and another for larger inputs. This is typically done in hybrid sorting algorithms such as when merge sort is performed for "large" inputs but switches over to insertion sort for smaller arrays.

[6]The original notation and definition are attributed to Paul Bachmann in 1894 [2]. Definitions and notation have been refined and introduced/reintroduced over the years. Their use in algorithm analysis was first suggested by Donald Knuth in 1976 [7].

- Some definitions are more general about the nature of the functions f, g. However, since we're looking at these functions as characterizing the resources that an algorithm takes to execute, we've restricted the domain and codomain of the functions. The domain is restricted to non-negative integers since there is little sense in negative or factional input sizes. The codomain is restricted to nonnegative reals as it doesn't make sense that an algorithm would potentially consume a negative amount of resources.
- We've used the set notation $f(n) \in O(g(n))$ because, strictly speaking, $O(g(n))$ is a *class* of functions: the set of all functions that are asymptotically bounded by $g(n)$. Thus the set notation is the most appropriate. However, you will find many sources and papers using notation similar to

  $$f(n) = O(g(n))$$

  This is a slight abuse of notation, but common nonetheless.

The intuition behind the definition of big-O is that f is *asymptotically* less than or equal to g. That is, the rate of growth of g is *at least as fast* as the growth rate of f. Big-O provides a means to express that one function is an *asymptotic upper bound* to another function.

The definition essentially states that $f(n) \in O(g(n))$ if, after some point (for all $n \geq n_0$), the value of the function $g(n)$ will always be larger than $f(n)$. The constant c possibly serves to "stretch" or "compress" the function, but has no effect on the growth rate of the function.

Example

Let's revisit the example from before where $f(n) = 100n^2 + 5n$ and $g(n) = n^3$. We want to show that $f(n) \in O(g(n))$. By the definition, we need to show that there exists a c and n_0 such that

$$f(n) \leq cg(n)$$

As we observed in the graph in Figure 5.2, the functions "crossed over" somewhere around $n = 100$. Let's be more precise about that. The two functions cross over when they are equal, so we setup an equality,

$$100n^2 + 5n = n^3$$

Collecting terms and factoring out an n (that is, the functions have one crossover point at $n = 0$), we have

$$n^2 - 100n - 5 = 0$$

The values of n satisfying this inequality can be found by applying the quadratic formula, and so

$$n = \frac{100 \pm \sqrt{10000 + 20}}{2}$$

Which is $-0.049975\ldots$ and $100.0499\ldots$. The first root is negative and so irrelevant. The second is our cross over point. The next largest integer is 101. Thus, for $c = 1$ and $n_0 = 101$, the inequality is satisfied.

In this example, it was easy to find the intersection because we could employ the quadratic equation to find roots. This is much more difficult with higher degree polynomials. Throw in some logarithmic functions, exponential functions, etc. and this approach can be difficult.

Revisit the definition of big-O: the inequality doesn't have to be tight or precise. In the previous example we essentially fixed c and tried to find n_0 such that the inequality held. Alternatively, we could fix n_0 to be small and then find the c (essentially compressing the function) such that the inequality holds. Observe:

$$\begin{aligned}
100n^2 + 5n &\leq 100n^2 + 5n^2 && \text{since } n \leq n^2 \text{ for all } n \geq 0 \\
&= 105n^2 \\
&\leq 105n^3 && \text{since } n^2 \leq n^3 \text{ for all } n \geq 0 \\
&= 105g(n)
\end{aligned}$$

By adding positive values, we make the equation larger until it looks like what we want, in this case $g(n) = n^3$. By the end we've got our constants: for $c = 105$ and $n_0 = 0$, the inequality holds. There is nothing special about this c, $c = 1000000$ would work too. The point is we need only find at least one c, n_0 pair that the inequality holds (there are an infinite number of possibilities).

5.4.2 Other Notations

Big-O provides an asymptotic upper bound characterization of two functions. There are several other notations that provide similar characterizations.

Big-Omega

Definition 2. Let f and g be two functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \Omega(g(n))$$

read as "f is big-Omega of g," if there exist constants $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \geq cg(n)$$

Big-Omega provides an asymptotic lower bound on a function. The only difference is the inequality has been reversed. Intuitively f has a growth rate that is bounded below by g.

Big-Theta

Yet another characterization can be used to show that two functions have the *same* order of growth.

Definition 3. Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \Theta(g(n))$$

read as "f is Big-Theta of g," if there exist constants $c_1, c_2 \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Big-Θ essentially provides an asymptotic equivalence between two functions. The function f is bounded above *and* below by g. As such, both functions have the same rate of growth.

Soft-O Notation

Logarithmic factors contribute very little to a function's rate of growth especially compared to larger order terms. For example, we called $n \log(n)$ *quasi*linear since it was nearly linear. Soft-O notation allows us to simplify terms by removing logarithmic factors.

Definition 4. Let f, g be functions such that $f(n) \in O(g(n) \cdot \log^k(n))$. Then we say that $f(n)$ is *soft-O* of $g(n)$ and write

$$f(n) \in \tilde{O}(g(n))$$

For example,

$$n \log(n) \in \tilde{O}(n)$$

Little Asymptotics

Related to big-O and big-Ω are their corresponding "little" asymptotic notations, little-o and little-ω.

Definition 5. Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in o(g(n))$$

read as "f is little-o of g," if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

The little-o is sometimes defined as for every $\epsilon > 0$ there exists a constant N such that

$$|f(n)| \leq \epsilon |g(n)| \quad \forall n \geq N$$

but given the restriction that $g(n)$ is positive, the two definitions are essentially equivalent.

Little-o is a much stronger characterization of the relation of two functions. If $f(n) \in o(g(n))$ then not only is g an asymptotic upper bound on f, but they are *not* asymptotically equivalent. Intuitively, this is similar to the difference between saying that $a \leq b$ and $a < b$. The second is a stronger statement as it implies the first, but the first does not imply the second. Analogous to this example, little-o provides a "strict" asymptotic upper bound. The growth rate of g is *strictly* greater than the growth rate of f.

Similarly, a little-ω notation can be used to provide a *strict* lower bound characterization.

Definition 6. Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \omega(g(n))$$

read as "f is little-omega of g," if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

5.4.3 Observations

As you might have surmised, big-O and big-Ω are duals of each other, thus we have the following.

Lemma 1. Let f, g be functions. Then

$$f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$$

Because big-Θ provides an asymptotic equivalence, both functions are big-O *and* big-Θ of each other.

Lemma 2. Let f, g be functions. Then

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

Equivalently,

$$f(n) \in \Theta(g(n)) \iff g(n) \in O(f(n)) \text{ and } g(n) \in \Omega(f(n))$$

With respect to the relationship between little-o and little-ω to big-O and big-Ω, as previously mentioned, little asymptotics provide a stronger characterization of the growth rate of functions. We have the following as a consequence.

Lemma 3. Let f, g be functions. Then

$$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$$

and

$$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$$

Of course, the converses of these statements do not hold.

Common Identities

As a direct consequence of the definition, constant coefficients in a function can be ignored.

Lemma 4. For any constant c,

$$c \cdot f(n) \in O(f(n))$$

In particular, for $c = 1$, we have that

$$f(n) \in O(f(n))$$

and so any function is an upper bound on itself.

In addition, when considering the sum of two functions, $f_1(n), f_2(n)$, it suffices to consider the one with a larger rate of growth.

Lemma 5. Let $f_1(n), f_2(n)$ be functions such that $f_1(n) \in O(f_2(n))$. Then

$$f_1(n) + f_2(n) \in O(f_2(n))$$

In particular, when analyzing algorithms with independent operations (say, loops), we only need to consider the operation with a higher complexity. For example, when we presorted an array to compute the mode, the presort phase was $O(n \log(n))$ and the mode finding phase was $O(n)$. Thus the total complexity was

$$n \log(n) + n \in O(n \log(n))$$

When dealing with a polynomial of degree k,

$$c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \cdots + c_1 n + c_0$$

The previous results can be combined to conclude the following lemma.

Lemma 6. Let $p(n)$ be a polynomial of degree k,

$$p(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \cdots + c_1 n + c_0$$

then

$$p(n) \in \Theta(n^k)$$

Class Name	Asymptotic Characterization	Algorithm Examples
Constant	$O(1)$	Evaluating a formula
Logarithmic	$O(\log(n))$	Binary Search
Polylogarithmic	$O(\log^k(n))$	
Linear	$O(n)$	Linear Search
Quasilinear	$O(n\log(n))$	Mergesort
Quadratic	$O(n^2)$	Insertion Sort
Cubic	$O(n^3)$	
Polynomial	$O(n^k)$ for any $k > 0$	
Exponential	$O(2^n)$	Computing a powerset
Super-Exponential	$O(2^{f(n)})$ for $f(n) \in \Omega(n)$ For example, $n!$	Computing permutations

Table 5.5: Common Algorithmic Efficiency Classes

Logarithms

When working with logarithmic functions, it suffices to consider a single base. As Computer Scientists, we always work in base-2 (binary). Thus when we write $\log(n)$, we implicitly mean $\log_2(n)$ (base-2). It doesn't really matter though because all logarithms are the same to within a constant as a consequence of the change of base formula:

$$\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$$

That means that for any valid bases a, b,

$$\log_b(n) \in \Theta(\log_a(n))$$

Another way of looking at it is that an algorithm's complexity is the same regardless of whether or not it is performed by hand in base-10 numbers or on a computer in binary.

Other logarithmic identities that you may find useful remembering include the following:

$$\log(n^k) = k\log(n)$$

$$\log(n_1 n_2) = \log(n_1) + \log(n_2)$$

Classes of Functions

Table 5.5 summarizes some of the complexity functions that are common when doing algorithm analysis. Note that these classes of functions form a hierarchy. For example, linear and quasilinear functions are also $O(n^k)$ and so are polynomial.

5.4.4 Limit Method

The method used in previous examples directly used the definition to find constants c, n_0 that satisfied an inequality to show that one function was big-O of another. This can get quite tedious when there are many terms involved. A much more elegant proof technique borrows concepts from calculus.

Let $f(n), g(n)$ be functions. Suppose we examine the limit, as $n \rightarrow \infty$ of the ratio of these two functions.

$$\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)}$$

One of three things could happen with this limit.

The limit could converge to 0. If this happens, then by Definition 5 we have that $f(n) \in o(g(n))$ and so by Lemma 3 we know that $f(n) \in O(g(n))$. This makes sense: if the limit converges to zero that means that $g(n)$ is growing much faster than $f(n)$ and so f is big-O of g.

The limit could diverge to infinity. If this happens, then by Definition 6 we have that $f(n) \in \omega(g(n))$ and so again by Lemma 3 we have $f(n) \in \Omega(g(n))$. This also makes sense: if the limit diverges, $f(n)$ is growing much faster than $g(n)$ and so $f(n)$ is big-Ω of g.

Finally, the limit could converge to some positive constant (recall that both functions are restricted to positive codomains). This means that both functions have essentially the same order of growth. That is, $f(n) \in \Theta(g(n)$. As a consequence, we have the following Theorem.

Theorem 2 (Limit Method)**.** Let $f(n)$ and $g(n)$ be functions. Then if

$$\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) \in O(g(n)) \\ c > 0 & \text{then } f(n) \in \Theta(g(n)) \\ \infty & \text{then } f(n) \in \Omega(g(n)) \end{cases}$$

Examples

Let's reuse the example from before where $f(n) = 100n^2 + 5n$ and $g(n) = n^3$. Setting up our limit,

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{100n^2 + 5n}{n^3} \\ &= \lim_{n\to\infty} \frac{100n + 5}{n^2} \\ &= \lim_{n\to\infty} \frac{100n}{n^2} + \lim_{n\to\infty} \frac{5}{n^2} \\ &= \lim_{n\to\infty} \frac{100}{n} + 0 \\ &= 0\end{aligned}$$

And so by Theorem 2, we conclude that

$$f(n) \in O(g(n))$$

Consider the following example: let $f(n) = \log_2 n$ and $g(n) = \log_3 (n^2)$. Setting up our limit we have

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \frac{\log_2 n}{\log_3 n^2} \\ &= \frac{\log_2 n}{\frac{2\log_2 n}{\log_2 3}} \\ &= \frac{\log_2 3}{2} \\ &= .7924\ldots > 0\end{aligned}$$

And so we conclude that $\log_2 (n) \in \Theta(\log_3 (n^2))$.

As another example, let $f(n) = \log (n)$ and $g(n) = n$. Setting up the limit gives us

$$\lim_{n\to\infty} \frac{\log (n)}{n}$$

The rate of growth might seem obvious here, but we still need to be mathematically rigorous. Both the denominator and numerator are monotone increasing functions. To solve this problem, we can apply l'Hôpital's Rule:

Theorem 3 (l'Hôpital's Rule)**.** *Let f and g be functions. If the limit of the quotient $\frac{f(n)}{g(n)}$ exists, it is equal to the limit of the derivative of the denominator and the numerator. That is,*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Applying this to our limit, the denominator drops out, but what about the numerator? Recall that $\log(n)$ is the logarithm base-2. The derivative of the natural logarithm is well known, $\ln'(n) = \frac{1}{n}$. We can use the change of base formula to transform $\log(n) = \frac{\ln(n)}{\ln(2)}$ and then take the derivative. That is,

$$\log'(n) = \frac{1}{\ln(2)n}$$

Thus,

$$\begin{aligned} \lim_{n\to\infty} \frac{\log(n)}{n} &= \lim_{n\to\infty} \frac{\log'(n)}{n'} \\ &= \lim_{n\to\infty} \frac{1}{\ln(2)n} \\ &= 0 \end{aligned}$$

Concluding that $\log(n) \in O(n)$.

Pitfalls

l'Hôpital's Rule is not always the most appropriate tool to use. Consider the following example: let $f(n) = 2^n$ and $g(n) = 3^n$. Setting up our limit and applying l'Hôpital's Rule we have

$$\begin{aligned} \lim_{n\to\infty} \frac{2^n}{3^n} &= \lim_{n\to\infty} \frac{(2^n)'}{(3^n)'} \\ &= \lim_{n\to\infty} \frac{(\ln 2)2^n}{(\ln 3)3^n} \end{aligned}$$

which doesn't get us anywhere. In general, we should look for algebraic simplifications *first*. Doing so we would have realized that

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n$$

Since $\frac{2}{3} < 1$, the limit of its exponent converges to zero and we have that $2^n \in O(3^n)$.

5.5 Examples

5.5.1 Linear Search

As a simple example, consider the problem of searching a collection for a particular element. The straightforward solution is known as Linear Search and is featured as

Algorithm 10

```
INPUT  : A collection $A = \{a_1, \ldots, a_n\}$, a key $k$
OUTPUT : The first $i$ such that $a_i = k$, $\phi$ otherwise
1 FOR $i = 1, \ldots, n$ DO
2     IF $a_i = k$ THEN
3         output $i$
4     END
5 END
6 output $\phi$
```

Algorithm 10: Linear Search

Let's follow the prescribed outline above to analyze Linear Search.

1. Input: this is clearly indicated in the pseudocode of the algorithm. The input is the collection A.
2. Input Size: the most natural measure of the size of a collection is its cardinality; in this case, n
3. Elementary Operation: the most common operation is the comparison in line 2 (assignments and iterations necessary for the control flow of the algorithm are not good candidates).
4. How many times is the elementary operation executed with respect to the input size, n? The situation here actually depends not only on n but also the contents of the array.
    - Suppose we get lucky and find k in the first element, a_1: we've only made one comparison.
    - Suppose we are unlucky and find it as the last element (or don't find it at all). In this case we've made n comparisons
    - We could also look at the *average* number of comparisons (see Section 5.6.3)

   In general, algorithm analysis considers at the worst case scenario unless otherwise stated. In this case, there are $C(n) = n$ comparisons.
5. This is clearly linear, which is why the algorithm is called Linear Search,

$$\Theta(n)$$

5.5.2 Set Operation: Symmetric Difference

Recall that the *symmetric difference* of two sets, $A \oplus B$ consists of all elements in A *or* B but not both. Algorithm 11 computes the symmetric difference.

```
INPUT  : Two sets, $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_m\}$
OUTPUT : The symmetric difference, $A \oplus B$
1  $C \leftarrow \emptyset$
2  FOREACH $a_i \in A$ DO
3      IF $a_i \notin B$ THEN
4          $C \leftarrow C \cup \{a_i\}$
5      END
6  END
7  FOREACH $b_j \in B$ DO
8      IF $b_j \notin A$ THEN
9          $C \leftarrow C \cup \{b_j\}$
10     END
11 END
12 output $C$
```

Algorithm 11: Symmetric Difference of Two Sets

Again, following the step-by-step process for analyzing this algorithm,

1. Input: In this case, there are two sets as part of the input, A, B.
2. Input Size: As specified, each set has cardinality n, m respectively. We could analyze the algorithm with respect to both input sizes, namely the input size could be $n + m$. For simplicity, to work with a single variable, we could also define $N = n + m$.

   Alternatively, we could make the following observation: without loss of generality, we can assume that $n \geq m$ (if not, switch the sets). If one input parameter is bounded by the other, then

   $$n + m \leq 2n \in O(n)$$

   That is, we could simplify the analysis by only considering n as the input size. There will be no difference in the final asymptotic characterization as the constants will be ignored.
3. Elementary Operation: In this algorithm, the most common operation is the set membership query ($\notin$). Strictly speaking, this operation may not be trivial depending on the type of data structure used to represent the set (it may entail a

series of $O(n)$ comparisons for example). However, as our pseudocode is concerned, it is sufficient to consider it as our elementary operation.

4. How many times is the elementary operation executed with respect to the input size, n? In the first for-loop (lines 2–6) the membership query is performed n times. In the second loop (lines 7–11), it is again performed m times. Since each of these loops is independent of each other, we would *add* these operations together to get

$$n + m$$

total membership query operations.

5. Whether or not we consider $N = n + m$ or n to be our input size, the algorithm is clearly linear with respect to the input size. Thus it is a $\Theta(n)$-time algorithm.

5.5.3 Euclid's GCD Algorithm

The greatest common divisor (or GCD) of two integers a, b is the largest positive integer that divides both a and b. Finding a GCD has many useful applications and the problem has one of the oldest known algorithmic solutions: Euclid's Algorithm (due to the Greek mathematician Euclid c. 300 BCE).

```
INPUT  : Two integers, $a, b$
OUTPUT : The greatest common divisor, $\gcd(a, b)$
1 WHILE $b \neq 0$ DO
2     $t \leftarrow b$
3     $b \leftarrow a \bmod b$
4     $a \leftarrow t$
5 END
6 Output $a$
```

Algorithm 12: Euclid's GCD Algorithm

The algorithm relies on the following observation: any number that divides a, b must also divide the remainder of a since we can write $b = a \cdot k + r$ where r is the remainder. This suggests the following strategy: iteratively divide a by b and retain the remainder r, then consider the GCD of b and r. Progress is made by observing that b and r are necessarily smaller than a, b. Repeating this process until we have a remainder of zero gives us the GCD because once we have that one evenly divides the other, the larger must be the GCD. Pseudocode for Euclid's Algorithm is provided in Algorithm 12.

The analysis of Euclid's algorithm is seemingly straightforward. It is easy to identify the division in line 3 as the elementary operation. But how many times is it executed with respect to the input size? What is the input size?

When considering algorithms that primarily execute numerical operations the input is usually a number (or in this case a pair of numbers). How big is the input of a number? The input size of 12,142 is not 12,142. The number 12,142 has a compact representation when we write it: it requires 5 digits to express it (in base 10). That is, the input size of a number is the number of symbols required to represent its magnitude. Considering a number's input size to be equal to the number would be like considering its representation in unary where a single symbol is used and repeated for as many times as is equal to the number (like a prisoner marking off the days of his sentence).

Computers don't "speak" in base-10, they speak in binary, base-2. Therefore, the input size of a numerical input is the number of bits required to represent the number. This is easily expressed using the base-2 logarithm function:

$$\lceil \log{(n)} \rceil$$

But in the end it doesn't really matter if we think of computers as speaking in base-10, base-2, or any other integer base greater than or equal to 2 because as we've observed that all logarithms are equivalent to within a constant factor using the change of base formula. In fact this again demonstrates again that algorithms are an abstraction independent of any particular platform: that the same algorithm will have the same (asymptotic) performance whether it is performed on paper in base-10 or in a computer using binary!

Back to the analysis of Euclid's Algorithm: how many times does the while loop get executed? We can first observe that each iteration reduces the value of b, but by how much? The exact number depends on the input: some inputs would only require a single division, other inputs reduce b by a different amount on each iteration. The important thing to realize is that we want a general characterization of this algorithm: it suffices to consider the worst case. That is, at maximum, how many iterations are performed? The number of iterations is maximized when the reduction in the value of b is minimized at each iteration. We further observe that b is reduced by at least half at each iteration. Thus, the number of iterations is maximized if we reduce b by at most half on each iteration. So how many iterations i are required to reduce n down to 1 (ignoring the last iteration when it is reduced to zero for the moment)? This can be expressed by the equation:

$$n \left(\frac{1}{2}\right)^i = 1$$

Solving for i (taking the log on either side), we get that

$$i = \log{(n)}$$

Recall that the size of the input is $\log{(n)}$. Thus, Euclid's Algorithm is *linear* with respect to the input size.

5.5.4 Selection Sort

Recall that Selection Sort is a sorting algorithm that sorts a collection of elements by first finding the smallest element and placing it at the beginning of the collection. It continues by finding the smallest among the remaining $n-1$ and placing it second in the collection. It repeats until the “first” $n-1$ elements are sorted, which by definition means that the last element is where it needs to be.

The Pseudocode is presented as Algorithm 13.

```
    INPUT  : A collection $A = \{a_1, \ldots, a_n\}$
    OUTPUT : $A$ sorted in non-decreasing order
 1  FOR $i = 1, \ldots, n-1$ DO
 2      $min \leftarrow a_i$
 3      FOR $j = (i+1), \ldots, n$ DO
 4          IF $min < a_j$ THEN
 5              $min \leftarrow a_j$
 6          END
 7          swap $min, a_i$
 8      END
 9  END
10  output $A$
```

Algorithm 13: Selection Sort

Let’s follow the prescribed outline above to analyze Selection Sort.

1. Input: this is clearly indicated in the pseudocode of the algorithm. The input is the collection A.
2. Input Size: the most natural measure of the size of a collection is its cardinality; in this case, n
3. Elementary Operation: the most common operation is the comparison in line 4 (assignments and iterations necessary for the control flow of the algorithm are not good candidates). Alternatively, we could have considered swaps on line 7 which would lead to a different characterization of the algorithm.
4. How many times is the elementary operation executed with respect to the input size, n?
   - Line 4 does one comparison each time it is executed
   - Line 4 itself is executed multiple times for each iteration of the for loop in line 3 (for j running from $i+1$ up to n inclusive.

- line 3 (and subsequent blocks of code) are executed multiple times for each iteration of the for loop in line 1 (for i running from 1 up to $n-1$

This gives us the following summation:

$$\underbrace{\sum_{i=1}^{n-1} \underbrace{\sum_{j=i+1}^{n} \underbrace{1}_{\text{line 4}}}_{\text{line 3}}}_{\text{line 1}}$$

Solving this summation gives us:

$$\begin{aligned}\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1} n - i \\ &= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\ &= n(n-1) - \frac{n(n-1)}{2} \\ &= \frac{n(n-1)}{2}\end{aligned}$$

Thus for a collection of size n, Selection Sort makes

$$\frac{n(n-1)}{2}$$

comparisons.

5. Provide an asymptotic characterization: the function determined is clearly $\Theta(n^2)$

5.6 Other Considerations

5.6.1 Importance of Input Size

The second step in our algorithm analysis outline is to identify the input size. Usually this is pretty straightforward. If the input is an array or collection of elements, a reasonable input size is the cardinality (the number of elements in the collection). This is usually the case when one is analyzing a sorting algorithm operating on a list of elements.

Though seemingly simple, sometimes identifying the appropriate input size depends on the nature of the input. For example, if the input is a data structure such as a graph, the input size could be either the number of vertices *or* number of edges. The most appropriate measure then depends on the algorithm and the details of its analysis. It may even depend on how the input is represented. Some graph algorithms have different efficiency measures if they are represented as adjacency lists or adjacency matrices.

Yet another subtle difficulty is when the input is a single numerical value, n. In such instances, the input size is *not* also n, but instead the number of symbols that would be needed to represent n. That is, the number of digits in n or the number of bits required to represent n. We'll illustrate this case with a few examples.

Sieve of Eratosthenes

A common beginner mistake is made when analyzing the Sieve of Eratosthenes (named for Eratosthenes of Cyrene, 276 BCE – 195 BCE). The Sieve is an ancient method for prime number factorization. A brute-force algorithm, it simply tries every integer up to a point to see if it is a factor of a given number.

```
   INPUT  : An integer $n$
   OUTPUT : Whether $n$ is *prime* or *composite*
1  FOR $i = 2, \ldots, n$ DO
2      IF $i$ *divides* $n$ THEN
3          Output *composite*
4      END
5  END
6  Output *prime*
```

Algorithm 14: Sieve of Eratosthenes

The for-loop only needs to check integers up to $\sqrt{n}$ because any factor greater than $\sqrt{n}$ would necessarily have a corresponding factor $\sqrt{n}$. A naive approach would observe that the for-loop gets executed $\sqrt{n} - 1 \in O(\sqrt{n})$ times which would lead to the (incorrect) impression that the Sieve is a polynomial-time (in fact sub-linear!) running time algorithm. Amazing that we had a primality testing algorithm over 2,000 years ago! In fact, primality testing was a problem that was not known to have a deterministic polynomial time running algorithm until 2001 (the AKS Algorithm [1]).

The careful observer would realize that though n is the input, the actual input size is again $\log(n)$, the number of bits required to represent n. Let $N = \log(n)$ be a placeholder for our actual input size (and so $n = 2^N$). Then the running time of the Sieve is actually

$$O(\sqrt{n}) = O(\sqrt{2^N})$$

which is exponential with respect to the input size N.

```java
int a = 45, m = 67;
int result = 1;
for(int i=1; i<=n; i++) {
  result = (result * a % m);
}
```

Code Sample 5.8: Naive Exponentiation

This distinction is subtle but crucial: the difference between a polynomial-time algorithm and an exponential algorithm is huge even for modestly sized inputs. What may take a few milliseconds using a polynomial time algorithm may take billions and billions of years with an exponential time algorithm as we'll see with our next example.

Computing an Exponent

As a final example, consider the problem of computing a modular exponent. That is, given integers a, n, and m, we want to compute

$$a^n \bmod m$$

A naive (but common!) solution might be similar to the Java code snippet in Code Sample 5.8.

Whether one chooses to treat multiplication or integer division as the elementary operation, the for-loop executes exactly n times. For "small" values of n this may not present a problem. However, for even moderately large values of n, say $n \approx 2^{256}$, the performance of this code will be terrible.

To illustrate, suppose that we run this code on a 14.561 petaFLOP (14 quadrillion floating point operations per second) super computer cluster (this throughput was achieved by the Folding@Home distributed computing project in 2013). Even with this power, to make 2^{256} floating point operations would take

$$\frac{2^{256}}{14.561 \times 10^{15} \cdot 60 \cdot 60 \cdot 24 \cdot 365.25} \approx 2.5199 \times 10^{53}$$

or 252 sexdecilliion *years* to compute!

For context, this sort of operation is performed by millions of computers around the world every second of the day. A 256-bit number is not really all that "large". The problem again lies in the failure to recognize the difference between an input, n, and the input's *size*. The real solution to this problem is an algorithm known as Repeated Squaring where values are squared,

$$(a^2)^2 = a^4, (a^4)^2 = a^8, (a^8)^2 = a^16, \ldots$$

```
1 public static double average(double arr[]) {
2   double sum = 0.0;
3   for(int i=0; i<arr.length; i++) {
4     sum = sum + arr[i];
5   }
6   return sum / arr.length;
7 }
```

Code Sample 5.9: Computing an Average

allowing the exponent to grow exponentially and reducing the number of total multiplications.

5.6.2 Control Structures are Not Elementary Operations

When considering which operation to select as the elementary operation, we usually do *not* count operations that are necessary to the control structure of the algorithm. For example, assignment operations, or operations that are necessary to execute a loop (incrementing an index variable, a comparison to check the termination condition).

To see why, consider the method in Code Sample 5.9. This method computes a simple average of an array of `double` variables. Consider some of the minute operations that are performed in this method:

- An assignment of a value to a variable (lines 2, 3, and 4)
- An increment of the index variable `i` (line 3)
- A comparison (line 3) to determine if the loop should terminate or continue
- The addition (line 4) and division (line 6)

A proper analysis would use the addition in line 4 as the elementary operation, leading to a $\Theta(n)$ algorithm. However, for the sake of argument, let's perform a detailed analysis with respect to each of these operations. Assume that there are n values in the array, thus the for loop executes n times. This gives us

- $n + 2$ total assignment operations
- n increment operations
- n comparisons
- n additions and
- 1 division

Now, suppose that each of these operations take time t_1, t_2, t_3, t_4, and t_5 milliseconds each (or whatever time scale you like). In total, we have a running time of

$$t_1(n+2) + t_2n + t_3n + t_4n + t_5 = (t_1 + t_2 + t_3 + t_4)n + (2t_1 + t_5) = cn + d$$

Which doesn't change the asymptotic complexity of the algorithm: considering the additional operations necessary for the control structures only changed the constant sitting out front (as well as some additive terms).

The amount of resources (in this case time) that are expended for the assignment, increment and comparison for the loop control structure are proportional to the true elementary operation (addition). Thus, it is sufficient to simply consider the most common or most expensive operation in an algorithm. The extra resources for the control structures end up only contributing constants which are ultimately ignored when an asymptotic analysis is performed.

5.6.3 Average Case Analysis

The behavior of some algorithms may depend on the nature of the input rather than simply the input size. From this perspective we can analyze an algorithm with respect to its best-, average-, and worst-case running time.

In general, we prefer to consider the worst-case running time when comparing algorithms.

Example: searching an array

Consider the problem of searching an array for a particular element (the array is unsorted, contains n elements). You could get lucky (best-case) and find it immediately in the first index, requiring only a single comparison. On the other hand, you could be unlucky and find the element in the last position or not at all. In either case n comparisons would be required (worst-case).

What about the average case? A naive approach would be to average the worst and best case to get an average of $\frac{n+1}{2}$ comparisons. A more rigorous approach would be to define a probability of a successful search p and the probability of an unsuccessful search, $1 - p$.

For a successful search, we further define a uniform probability distribution on finding the element in each of the n indices. That is, we will find the element at index i with probability $\frac{p}{n}$. Finding the element at index i requires i comparisons. Thus the total number of expected comparisons in a successful search is

$$\sum_{i=1}^{n} i \cdot \frac{p}{n} = \frac{p(n+1)}{2}$$

For an unsuccessful search, we would require n comparisons, thus the number of expected comparisons would be

$$n(1-p)$$

p	C
0	n
$\frac{1}{4}$	$\frac{7}{8}n + \frac{1}{8}$
$\frac{1}{2}$	$\frac{3}{4}n + \frac{1}{4}$
$\frac{3}{4}$	$\frac{5}{8}n + \frac{3}{8}$
1	$\frac{n+1}{2}$

Table 5.6: Expected number of comparisons C for various values of the probability of a successful search p.

Since these are mutually exclusive events, we sum the probabilities:

$$\frac{p(n+1)}{2} + n(1-p) = \frac{p - pn + 2n}{2}$$

We cannot remove the p terms, but we can make some observations for various values (see Table 5.6.3). When $p = 1$ for example, we have the same conclusion as the naive approach. As p decreases, the expected number of comparisons grows to n.

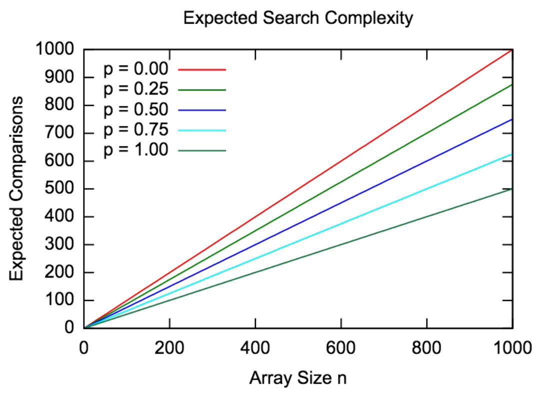


Figure 5.3: Expected number of comparisons for various success probabilities p.

5.6.4 Amortized Analysis

Sometimes it is useful to analyze an algorithm not based on its worst-case running time, but on its *expected* running time. On average, how many resources does an algorithm use? This is a more practical approach to analysis. Worst-case analysis assumes that all inputs will be difficult, but in practice, difficult inputs may be rare. The average input may require fewer resources.

Amortized algorithm analysis is similar to the average-case analysis we performed before, but focuses on how the *cost* of operations may change over the course of an algorithm. This is similar to a loan from a bank. Suppose the loan is taken out for \$1,000 at a 5% interest rate. You don't actually end up paying \$50 the first year. You pay slightly less than that since you are (presumably) making monthly payments, reducing the balance and thus reducing the amount of interest accrued each month.

We will not go into great detail here, but as an example, consider Heap Sort. This algorithm uses a *Heap* data structure. It essentially works by inserting elements into the heap and then removing them one by one. Due to the nature of a heap data structure, the elements will come out in the desired order.

An amortized analysis of Heap Sort may work as follows. First, a heap requires about $\log(n)$ comparisons to insert an element (as well as remove an element, though we'll focus on insertion), where n is the size of the heap (the number of elements currently in the heap. As the heap grows (and eventually shrinks), the cost of inserts will change. With an initially empty heap, there will only be 0 comparisons. When the heap has 1 element, there will be about $\log(1)$ comparisons, then $\log(2)$ comparisons and so on until we insert the last element. This gives us

$$C(n) = \sum_{i=1}^{n-1} \log(i)$$

total comparisons.

5.7 Analysis of Recursive Algorithms

Recursive algorithms can be analyzed with the same basic 5 step process. However, because they are recursive, the analysis (step 4) may involve setting up a recurrence relation in order to characterize how many times the elementary operation is executed.

Though cliche, as a simple example consider a naive recursive algorithm to compute the n-th Fibonacci number. Recall that the Fibonacci sequence is defined as

$$F_n = F_{n-1} + F_{n-2}$$

with initial conditions $F_0 = F_1 = 1$. That is, the n-th Fibonacci number is defined as the sum of the two previous numbers in the sequence. This defines the sequence

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

The typical recursive solution is presented as Algorithm 15

```
    INPUT    : An integer $n$
    OUTPUT : The $n$-th Fibonacci Number, $F_n$
1 IF $n = 0$ or $n = 1$ THEN
2   | output 1
3 ELSE
4   | output Fibonacci($n - 1$) + Fibonacci($n - 2$)
5 END
```

Algorithm 15: Fibonacci(n)

The elementary operation is clearly the addition in line 4. However, how do we analyze the number of additions performed by a call to Fibonacci(n)? To do this, we setup a recurrence relation. Let $A(n)$ be a function that counts the number of additions performed by Fibonacci(n). If n is zero or one, the number of additions is zero (the base case of our recursion performs no additions). That is, $A(0) = A(1) = 0$. But what about $n > 1$?

When $n > 1$, line 4 executes. Line 4 contains one addition. However, it also contains two recursive calls. How many additions are performed by a call to Fibonacci($n - 1$) and Fibonacci($n - 2$)? We defined $A(n)$ to be the number of additions on a call to Fibonacci(n), so we can reuse this function: the number of additions is $A(n - 1)$ and $A(n - 2)$ respectively. Thus, the total number of additions is

$$A(n) = A(n - 1) + A(n - 2) + 1$$

This is a recurrence relation,[7] in particular, it is a second-order linear non-homogeneous recurrence relation. This particular relation can be solved. That is, $A(n)$ can be expressed as a non-recursive closed-form solution.

The techniques required for solving these type of recurrence relations are beyond the scope of this text. However, for many common recursive algorithms, we can use a simple tool to characterize their running time, the "Master Theorem."

5.7.1 The Master Theorem

Suppose that we have a recursive algorithm that takes an input of size n. The recursion may work by dividing the problem into a subproblems each of size $\frac{n}{b}$ (where a, b are constants). The algorithm may also perform some amount of work before or after the recursion. Suppose that we can characterize this amount of work by a polynomial function, $f(n) \in \Theta(n^d)$.

[7]Also called a *difference equation*, which are sort of discrete analogs of differential equations.

This kind of recursive algorithm is common among "divide-and-conquer" style algorithms that *divide* a problem into subproblems and *conquers* each subproblem. Depending on the values of a, b, d we can categorize the runtime of this algorithm using the Master Theorem.

Theorem 4 (Master Theorem)**.** Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{array}{rcl} T(n) & = & aT(\frac{n}{b}) + f(n) \\ T(1) & = & c \end{array}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

"Master" is a bit of a misnomer. The Master Theorem can only be applied to recurrence relations of the particular form described. It can't, for example, be applied to the previous recurrence relation that we derived for the Fibonacci algorithm. However, it can be used for several common recursive algorithms.

Example: Binary Search

As an example, consider the Binary Search algorithm, presented as Algorithm 16. Recall that binary search takes a *sorted* array (random access is required) and searches for an element by checking the middle element m. If the element being searched for is larger than m, a recursive search on the upper half of the list is performed, otherwise a recursive search on the lower half is performed.

Let's analyze this algorithm with respect to the number of comparisons it performs. To simplify, though we technically make two comparisons in the pseudocode (lines 5 and 7), let's count it as a single comparison. In practice a comparator pattern would be used and logic would branch based on the result. However, there would still only be one invocation of the comparator function/object. Let $C(n)$ be a function that equals the number of comparisons made by Algorithm 16 while searching an array of size n *in the worst case* (that is, we never find the element or it ends up being the last element we check).

As mentioned, we make one comparison (line 5, 7) and then make one recursive call. The recursion roughly cuts the size of the array in half, resulting in an array of size $\frac{n}{2}$. Thus,

$$C(n) = C\left(\frac{n}{2}\right) + 1$$

Applying the master theorem, we find that $a = 1$, $b = 2$. In this case, $f(n) = 1$ which *is* bounded by a polynomial: a polynomial of degree $d = 0$. Since

$$1 = a = b^d = 2^0$$

```
INPUT  : A *sorted* collection of elements $A = \{a_1, \ldots, a_n\}$, bounds $1 \leq l, h \leq n$,
         and a key $e_k$
OUTPUT : An element $a \in A$ such that $a = e_k$ according to some criteria; $\phi$ if no
         such element exists
IF $l > h$ THEN
    output $\phi$
END
$m \leftarrow \lfloor \frac{h+l}{2} \rfloor$
IF $a_m = e_k$ THEN
    output $a_m$
ELSE IF $a_m < e_k$ THEN
    BinarySearch$(A, m+1, h, e)$
ELSE
    BinarySearch$(A, l, m-1, e)$
END
```

Algorithm 16: Binary Search – Recursive

by case 2 of the Master Theorem applies and we have

$$C(n) \in \Theta(\log(n))$$

Example: Merge Sort

Recall that Merge Sort is an algorithm that works by recursively splitting an array of size n into two equal parts of size roughly $\frac{n}{2}$. The recursion continues until the array is trivially sorted (size 0 or 1). Following the recursion back up, each subarray half is sorted and they need to be *merged*. This basic divide and conquer approach is presented in Algorithm 17. We omit the details of the merge operation on line 4, but observe that it can be achieved with roughly $n-1$ comparisons in the worst case.

```
INPUT  : An array, sub-indices $1 \leq l, r \leq n$
OUTPUT : An array $A'$ such that $A[l, \ldots, r]$ is sorted
1  IF $l < r$ THEN
2      MergeSort$(A, l, \lfloor \frac{r+l}{2} \rfloor)$
3      MergeSort$(A, \lceil \frac{r+l}{2} \rceil, r)$
4      Merge sorted lists $A[l, \ldots, \lfloor \frac{r+l}{2} \rfloor]$ and$A[\lceil \frac{r+l}{2} \rceil, \ldots, r]$
5  END
```

Algorithm 17: Merge Sort

Let $C(n)$ be the number of comparisons made by Merge Sort. The main algorithm makes *two* recursive calls on subarrays of size $\frac{n}{2}$. There are also $n+1$ comparisons made after the recursion. Thus we have

$$C(n) = 2C\left(\frac{n}{2}\right) + (n-1)$$

Again, applying the Master Theorem we have that $a = 2, b = 2$ and that $f(n) = (n+1) \in \Theta(n)$ and so $d = 1$. Thus,

$$2 = a = b^d = 2^1$$

and by case 2,

$$C(n) \in \Theta(n \log(n))$$

We will apply the Master Theorem to several other recursive algorithms later on.

5.8 Exercises

When asked to design and analyze an algorithm, be sure to provide the following:

1. Complete pseudocode
2. Identify the input and the input size, n
3. Identify the elementary operation
4. Compute how many times the elementary operation is executed with respect to the input size n
5. Provide a Big-O asymptotic characterization for the algorithm's complexity

Exercise 5.1. Let P be an image represented as an $(n \times m)$ 2-dimensional array of pixels. Design an algorithm that given an image P will rotate it clockwise by 90 degrees.

Exercise 5.2. Let C be a set of circles each represented as a triple (x, y, r) where x, y is its center and r is its radius. Design and analyze an algorithm that given a set C of n circles determines if any of the circles intersect.

Exercise 5.3. Let $A = [a_1, a_2, \ldots, a_n]$ be a collection of integers. A pair (i, j) is called an *inversion* if $i < j$ but $a_i > a_j$. For example, if $A = [2, 3, 8, 6, 1]$ then the list of inversions is $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$. Design an algorithm that, given a collection of integers A outputs a list of its inversions.

For the next few questions, consider defining a *time function* with respect to n. That is, if the algorithm is linear, we could write the time that it takes as a function of n:

$$t = f(n) = cn$$

If it is quadratic, we could write it as a quadratic function:

$$t = f(n) = cn^2$$

Both of these instances ignore lower order terms. If we know the time it takes for a particular value of n then we can compute the constant c and consequently predict the time t it takes for other values of n or vice versa.

Exercise 5.4. An algorithm takes 1.5 ms for an input size 100. How long will it take for input size 1500 (assuming that low-order terms are negligible) if the running time is

1. linear
2. $O(n \log n)$
3. quadratic
4. cubic
5. exponential

Exercise 5.5. An algorithm takes 0.5 ms for input size 100. How large can an input size be if a problem can be solved in 1 minute (assuming that low-order terms are negligible) if the running time is:

1. linear
2. $O(n \log n)$
3. quadratic
4. cubic
5. exponential

Exercise 5.6. Prove each of the following statements by applying the definition of Big-O. That is, derive an inequality (show your work) and clearly identify the c, n_0 constants you derive as per the definition of Big-O.

1. $3n = O(n)$
2. $500\sqrt{n} = O(n)$
3. $n^2 + 2n + 1 = O(n^2)$
4. $2048n + 1234 = O(n^2)$
5. $n \log(32n) = O(n \log(n)))$
6. $12n^3 + 50n^2 - 12n - 60 = O(n^3)$
7. $n2^n = O(3^n)$
8. $\log(n!) = O(n \log n)$

6 Trees

6.1 Introduction

One of our fundamental goals is to design a data structure to store elements that offers efficient and arbitrary retrieval (search), insertion, and deletion operations. We've already seen several examples of list-based data structures that offer different properties. Array-based lists offer fast index-based retrieval and, if sorted even more efficient, $O(\log n)$ key-based retrieval using binary search. However, arbitrary insertion and deletion may result in shifts leading to $O(n)$ behavior. In contrast, linked lists offer efficient, $O(1)$ insertion and deletion at the head/tail of the list, but inefficient, $O(n)$ arbitrary search/insert/delete.

Stacks and queues are efficient in that their core functionality offers $O(1)$ operations (push/pop, enqueue/dequeue) but they are restricted-access data structures and do not offer arbitrary and efficient operations.

We how turn our attention to another fundamental data structure based on *trees*. Trees are a very special type of graph. Graphs are useful because they can model many types of relations and processes in physics, biology, finance, and pretty much every other discipline. Moreover, we have over two centuries of foundational work on graph results and algorithms to work with.[1] General graphs, however, are not necessarily as structured as they need to be for an efficient data structure. Trees, however, are highly structured and have several useful properties that we can exploit to design an efficient data structure. As we will see, trees have the potential to offer efficient $O(\log n)$ behavior for all three fundamental operations.

6.2 Definitions & Terminology

A *graph* is a collection of nodes such that pairs of nodes may be connected by edges. More formally,

Definition 7 (Undirected Graph)**.** A *graph* is a two-tuple, $G = (V, E)$ where

- $V = \{v_1, \ldots, v_n\}$ is a set of vertices

[1]Graph Theory usually credited to Euler who studied the *Seven Bridges of Königsberg* [6] in the 18th century.

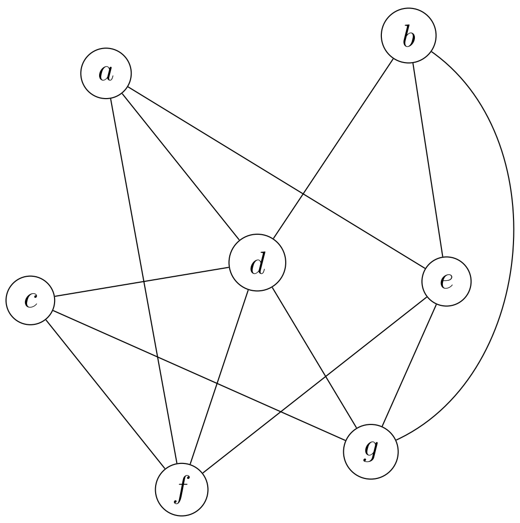


Figure 6.1: An undirected graph with labeled vertices.

- $E \subseteq V \times V$ is a set of edges connecting nodes

In general, the edges connecting two vertices may be directed or undirected. However, we'll only focus on undirected edges so that if two vertices u, v are connected, then the edge that connects them can be written as an unordered pair, $e = (u, v) = (v, u)$. An example of a graph can be found in Figure 6.1.

In the example, the nodes have been labeled with single characters. In general, we'll want to store elements of any type in a graph's vertices. In the context of data structures when vertices hold elements, they are usually referred to as *nodes* and we'll use this terminology from here on. Another point to highlight is that the size of a graph is usually measured in terms of the number of nodes in it. In the definition we have designated the variable n to denote this. In general, a graph may have up to

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

edges connecting pairs of nodes.[2]

Trees

General graphs may model very complex binary relations. Other than the graph structure itself (nodes possibly connected by edges) there is not much structure to exploit. For this reason, we will instead focus on a subclass of graphs called trees.

[2]This notation is from combinatorics, $\binom{n}{k}$ is a binomial coefficient and can be read as "n choose k." This operation counts the number of ways there are to choose an unordered subset of k elements from a set of n unique elements. In this case, $\binom{n}{2}$ is counting the number of ways there are two choose a pair of vertices from a set of n vertices.

Definition 8 (Trees)**.** A *tree* is an acyclic graph.

A *cycle* in a graph is any sequence of pairwise connected nodes that begin and end at the same node. For example, the sequence of vertices $aegcda$ in the graph in Figure 6.1 forms a cycle. Likewise, the sequences $begdb, bgdb$, and $cdgc$ also form cycles as well as many others. A graph in which there are no cycles is referred to as an *acyclic* graph and has a much simpler structure. This simplicity gives a tree several properties that can be exploited. An example of several trees can be found in Figure 6.2.

The example in Figure 6.2(d) is actually a *disconnected* tree in that there are several components whose nodes are not connected by any edges. Such graphs can be seen as a collection of trees and are usually called *forests*. Though forests are useful in modeling some problems, we will again restrict our attention to *connected trees* such that there is only *one* connected component.

Structural Properties

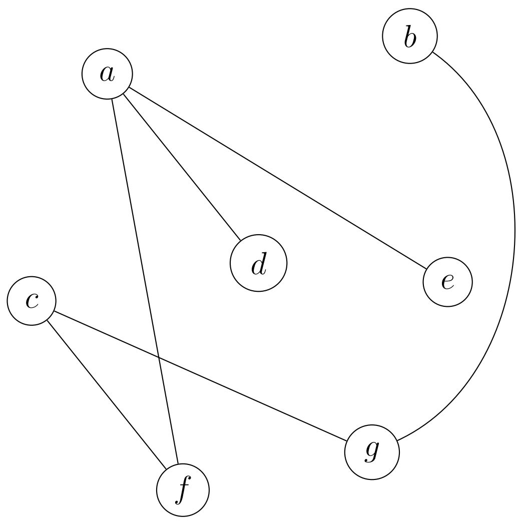
By definition, trees are acyclic which already gives them a high degree of structure. There are several related properties that follow from this structure. First, since we are restricting attention to connected trees, it will always be the case that the number of edges in a tree will be equal to one less than the number of vertices. That is:

Lemma 7. In any tree,
$$|E| = n - 1$$
where $n = |V|$.

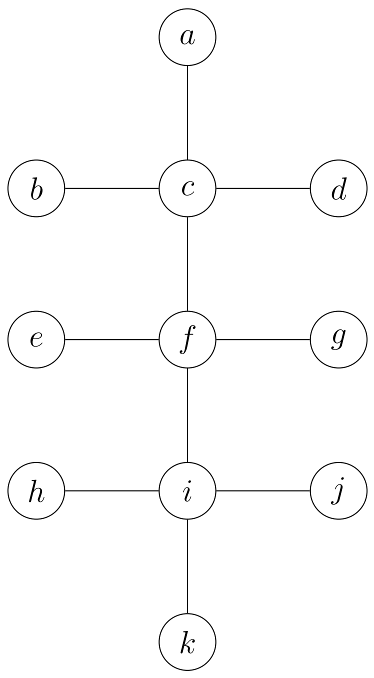
Recall that in general, graphs may have as many as $O(n^2)$ vertices. However, trees will necessarily have fewer $O(n)$ edges. This makes trees *sparse graphs*. Many graph problems that are hard or do not admit themselves to efficient solutions become easy or even trivial on sparse graphs and trees in particular.

Another consequence of a lack of cycles means that any two nodes in a tree will be connected by exactly one, unique *path*. A path is much like a cycle except that it need not begin/end at the same node. In Figure 6.2(a), the sequence of vertices $afcgb$ form a path. This is, in fact, the only path connecting the vertices a and b. Furthermore, the *length* of a path is defined as the number of edges on it. The aforementioned path $afcgb$ has length 4. Note that the length of.a path is also equal to the number of nodes in the path minus one. In the same tree in Figure 6.2(a), $afcgcfad$ is also technically a path, however, it traverses several edges more than once, backtracking from g. Such a path is referred to as a non-simple path. A path that does not traverse an edge more than once is a *simple* path and we will restrict our consideration to simple paths.

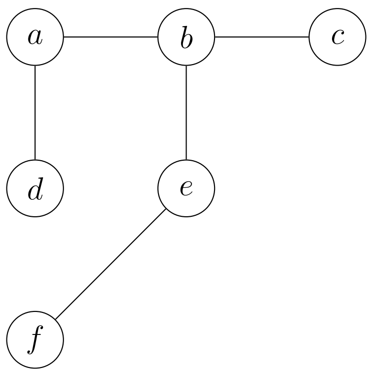
Lemma 8. Let $T = (V, E)$ be a connected tree and let $u, v \in V$ be two nodes in T. Then there exists exactly one single unique path connecting u, v.



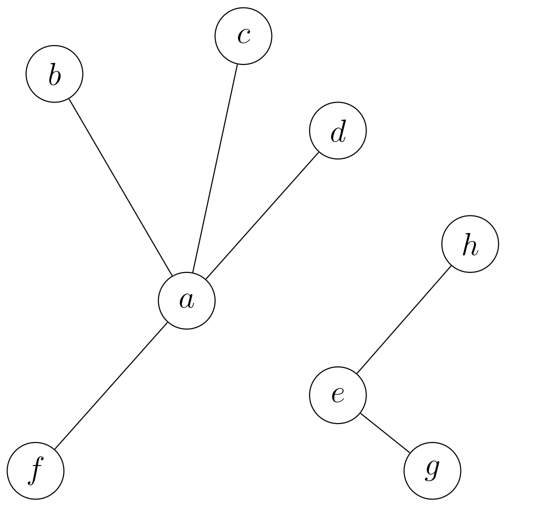
(a) The previous graph from Figure 6.1 with enough edges removed to make it a tree.



(b) Another tree drawn in a more organized manner.

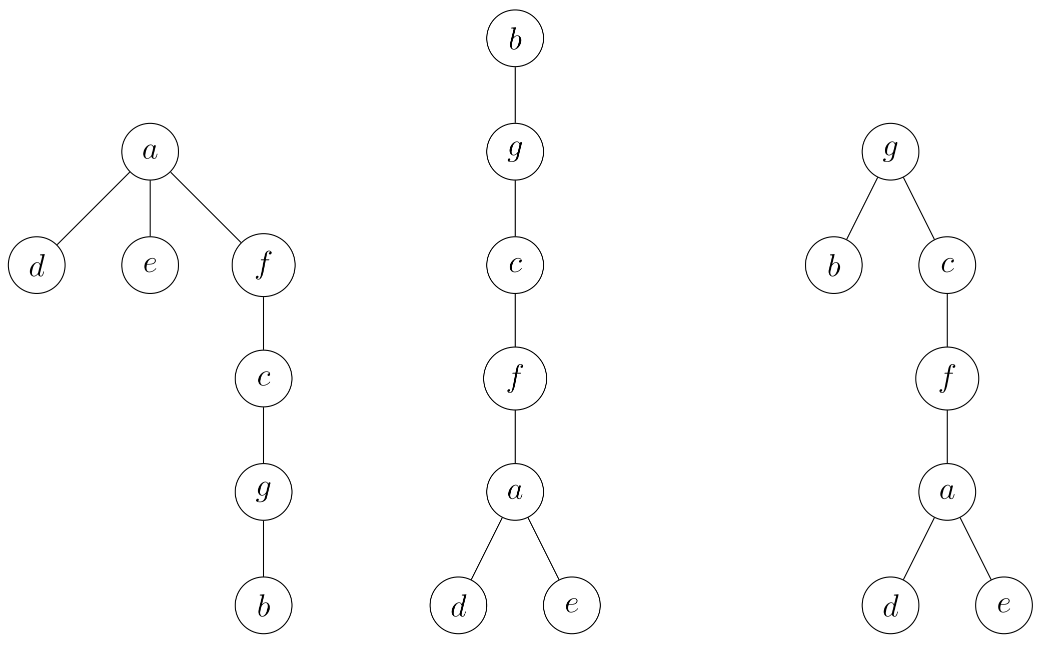


(c) Yet another little tree, happier.



(d) A disconnected tree with two components.

Figure 6.2: Several examples of trees. It doesn't matter how the tree is depicted or organized, only that it is acyclic. The final example, 6.2(d) represents a disconnected tree, called a *forest*.



(a) Tree rooted at a. (b) Tree rooted at b. (c) Tree rooted at g.

Figure 6.3: Several possible rooted orientations for the tree from Figure 6.2(a).

Proof. First we observe that there has to be at least one path between u, v. Otherwise, if there were not, then T would not be a connected tree.

Now suppose, by way of contradiction that there exist at least two paths p and p' connecting u, v such that $p \neq p'$. However, this means that we can now form a cycle. Starting at u and taking the first path p to v and then from v returning to u via p'. Since this forms a cycle, it contradicts the fact that we started with a tree. Thus, there cannot be two distinct paths between two vertices. □

These properties will prove useful in adapting trees as collection data structures.

Orienting Trees

To give trees even more structure, we can *orient* them. The first way that we'll do this is by *rooting* them. Imagine that you could take a tree (say any tree in Figure 6.2) and lift it from the page[3] by grabbing a single node. Then shake the tree and let gravity dangle the remaining nodes downward. Then place the tree back onto the page. The node that you shook the tree from will be designated as the tree's *root* and all other nodes are oriented downward. An example can be found in Figure 6.3 where we have done this operation on the tree in Figure 6.2(a) by dangling from various nodes.

[3]Or screen if you will.

The alert reader and professional arborists[4] alike will notice that we've drawn our trees with the root at the top and its "branches" growing downward, the exact opposite of how most real, organic trees grow. Trees in Computer Science "grow" downward because that is how we read. Since trees will ultimately be used to store data it is easier for a human to read them top-to-bottom, left-to-right. Of course, when represented in a computer, there is no such orientation; it is only a convention that we humans use when drawing and describing them. This convention is quite old; in fact Arthur Cayley, the original mathematician who first studied trees drew them like this in his original paper [3].[5]

The vertical orientation of a trees lends itself to some obvious terminology, borrowed from the concept of a family tree. Let u be a tree node. The node immediately above it is called its *parent*. Any node(s) immediately below it are called u's *children*. Likewise, all nodes on the path from u all the way back up to the root are known as u's ancestors; u's children and all nodes connected below them are called u's descendants. Suppose we were to remove all nodes in a tree except u and u's descendant(s). The new tree would form a *subtree* rooted at u. Other tree-related terminology will be used. For example, the root of the tree will be the only node without a parent. If a node has no children, it will be referred to as a *leaf* node.

Binary Trees

We can place more useful restrictions on our tree. An oriented tree such that every node has *at most* two children is a *binary tree*. This means that every node in a binary tree may have 0 children (a leaf), 1 child, or 2 children. We can further horizontally orient the children in a binary tree. Since the number of children is restricted to at most 2, we can refer to them as a left child and right child. All the identified relations in a binary tree node are depicted in Figure 6.4. Since a node may only have 1 child, using this orientation means that it may have a left child and missing its right child, or it may have a right child and missing its left child.

A larger binary tree is depicted in Figure 6.5. In this tree, the nodes k, l, n, o, p, q, r are all leaf nodes. Many nodes have both a left and right child (b, g, j as examples) while some have a left child, but no right child (m, i for example) and some have a right child, but no left child (e, h for example).

Given this orientation, it is natural to define the *depth* of a node. Let u be a node in a binary tree with root r. The depth of u is the length of the unique path from r to u. The depth of the root, r itself is 0 by convention. The depth of a tree T itself is defined as the maximal depth (i.e. "deepest") of any node in the tree. Thus, if a tree consists of a single root node, its depth is zero.[6]. It is possible to have an empty tree (where $|V| = 0$) and so by convention and for consistency, the depth of an empty tree is usually

[4] Or dendrologists.

[5] Specifically in part LVIII. *On the Analytical Forms called Trees.*–Part II in which he essentially defines binary trees.

[6] The author proposes to call this a *seed*; let's hope it catches on.

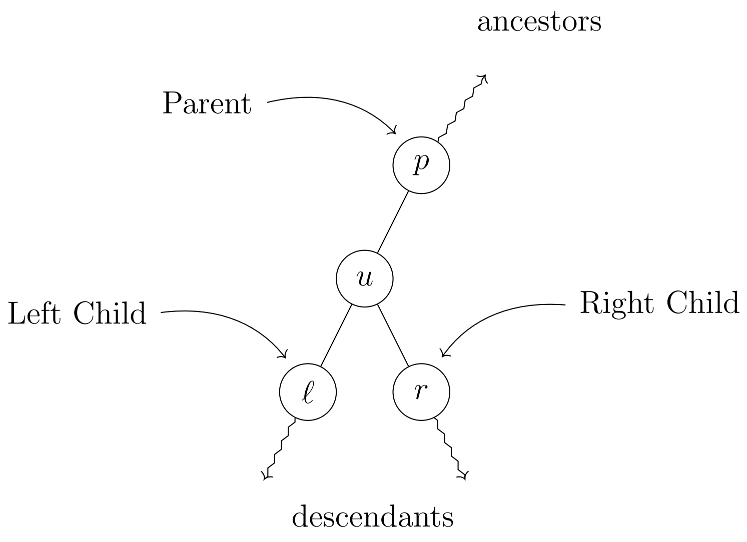


Figure 6.4: The various relations of a tree node. For a given node u, its parent, left and right child are labelled. In addition, nodes above u can collectively be referred to as ancestors (including its parent) and nodes in the subtree at u can collectively be referred to as descendants (including its immediate children).

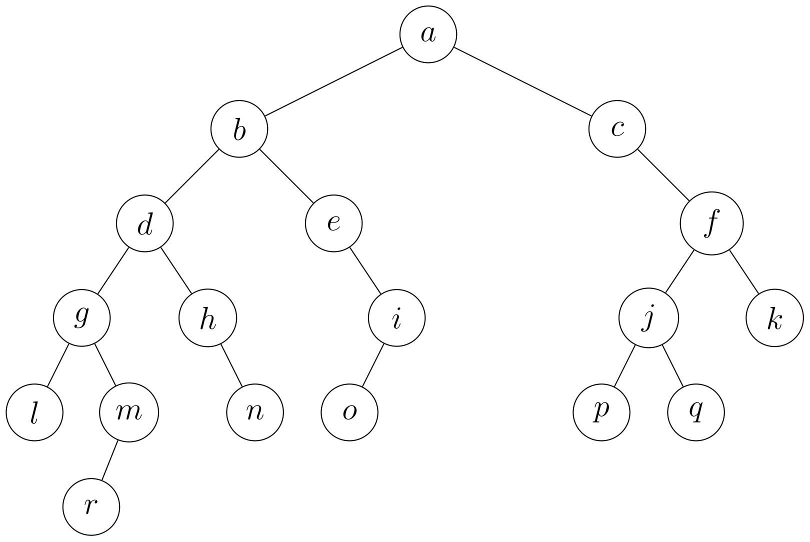


Figure 6.5: A Binary Tree

Table 6.1: Depths of nodes in Figure 6.5

Node(s)	Depth
a	0
b, c	1
d, e, f	2
g, h, i, j, k	3
l, m, n, o, p, q	4
r	5

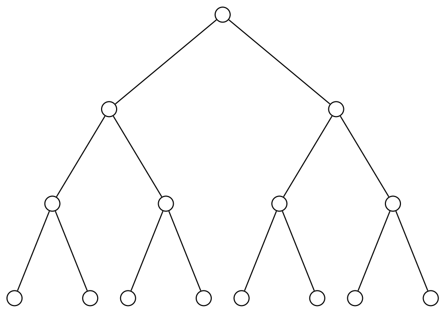


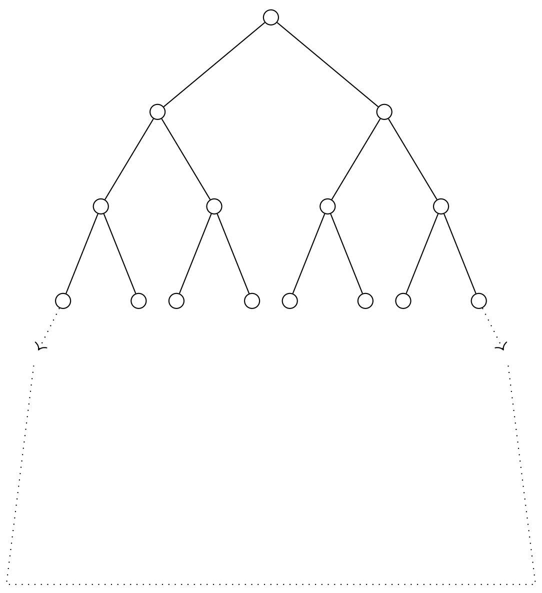
Figure 6.6: A complete tree of depth $d = 3$ which has $1 + 2 + 4 + 8 = 15$ nodes.

-1. The depth of every node in the tree in Figure 6.5 is given in Table 6.1. Since r is the deepest node, the depth of the tree itself is 5.

When depicting trees, we draw all nodes of equal depth on the same horizontal line. All nodes on this line are said to be at the same *level* in the tree. Levels are numbered as the depth so the root is at level 0, the next depth nodes are at level 1, etc. Structurally, we could draw the tree however we want. However, placing nodes of a different depth at different levels makes the tree less readable.

Now, consider a *complete* (also called *full* or *perfect*) binary tree of depth d. That is, a binary tree in which all nodes are present at all levels; from level 0 up to level d, the depth of the tree. How many nodes total are there in such a tree? At level 0, there would only be the root. At level 1, there would be 2 children from the root. Since all nodes are present, at level 2, both children from the previous 2 nodes would be present giving a total of 4, etc. That is, at each level, the number of nodes *on that level* doubles from the previous level. A concrete example for $d = 3$ is given in Figure 6.6 which has a total of 15 nodes.

We can generalize this by adding up powers of 2. At level 0, there are 2^0 nodes, level 1, there are 2^1 nodes, level 2 there are 2^2 nodes, etc. Summing all of these nodes up to



Level	Number of Nodes
0	2^0
1	2^1
2	2^2
3	2^3
4	2^4
$\vdots$	$\vdots$
d	2^d
total	$\sum_{i=0}^{d} i = 2^{d+1} - 1$

Figure 6.7: A summation of nodes at each level of a complete binary tree up to depth d.

depth d gives us the following geometric series which has a well-known solution.[7]

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^{d-1} + 2^d = \sum_{k=0}^{d} 2^k = 2^{d+1} - 1$$

This is visualized in Figure 6.7.

Taken from another perspective, if we have a complete binary tree with n nodes, its depth is

$$d = \log(n+1) - 1$$

That is, the depth is logarithmic, $d \in O(\log n)$. This observation gives us our motivation for using binary trees as a collection data structure. If we can develop a tree-based data structure such that insertion, retrieval, and removal operations are all proportional to

[7]As a Computer Scientist, you should also find it familiar–it represents the maximum integer representable by d bits.

the depth d then we have the *potential* for a very efficient, generalized collection data structure. Of course it is not necessarily a simple matter as we will see shortly.

6.3 Implementation

A binary tree implementation is fairly straightforward and similar to that of a linked list. Each tree node holds references to its two children, its parent (which is optional, but simplifies a lot of operations) and the key element stored in the tree node. In pseudocode we'll use the same dot operator syntax as with a linked list, so that each of these components can be referenced as follows. Let u be a tree node, then

- $u.key$ is its key,
- $u.parent$ is its parent,
- $u.leftChild$ is its left child, and
- $u.rightChild$ is its right child.

The tree itself only needs a reference to the root element, represented using $T.root$. Note that we do *not* keep a reference to every single tree node. Keeping track of and updating every tree node would bring us right back to a linked list or array-based list to store all the nodes, defeating the purpose of exploiting the tree structure. We could, however, keep track of how many nodes are in the tree as this could easily be updated with each insert/remove operation just as with a linked list.

As a concrete example, a pair of Java classes may be written to implement these two objects. In particular, the tree node class may look something like the following (getters, setters and other standard methods are omitted).

```java
public class BinaryTreeNode<T> {

  private T key;
  private TreeNode<T> parent;
  private TreeNode<T> leftChild;
  private TreeNode<T> rightChild;

  ...
}
```

While a tree itself would look something like the following.

```java
public class BinaryTree<T> {

  private TreeNode<T> root;
  private int size;

  ...
}
```

6.4 Tree Traversal

Before we develop the basic operations that will give us a collection data structure we need to ensure that we have a way to process all the elements in a general binary tree. With a linked list, we could easily iterate over the elements stored in it by starting at the head and iterating over each element until the tail. With a binary tree the order in which we iterate over elements is not all that clear. It is natural to start at the root node, but where do we go from there? To the left child or the right? If we go to the left, then we must remember to eventually come back and iterate over the right child.

More generally given a node u and its children, $u.leftChild, u.rightChild$ in which order do we enumerate them? In fact, there are several tree traversal strategies each with its own advantages and applications. Each one traverses the three different nodes in a different order but all are based on a more general graph traversal algorithm called DFS which attempts to explore a graph as deeply as possible before backtracking to explore the other areas. In general we will prefer to descend left in the tree before we backtrack and explore the right subtree. There is nothing particularly special about this preference (other than reading left-to-right is more natural). We *could* prefer to to right before going left. However, this would be equivalent to going left while also "reversing" the tree (reversing the left/right child at each node).

To understand the traversal strategies better, we will present the traversal choices locally. At any particular node, u, we'll refer to this node as the root (think of the subtree rooted at u) and its left and right child respectively. Given that we'll always go left-before-right, this gives three possible orders in which to enumerate the three nodes:

- Preorder: root-left-right
- Inorder: left-root-right
- Postorder: left-right-root

Of course, any given node may be missing one or both of its children in which case we do not enumerate it, but otherwise, we will preserve these orderings.

When we describe a traversal strategy, it is not restricted to merely enumerating all the elements in a tree. The purpose of using a binary tree as a data structure is to store data in each of its nodes. When visiting a node, in general we can perform any operation or process on the data stored in it rather than simply enumerating the element. The details of how to process a node would be specific to the type of data being stored and

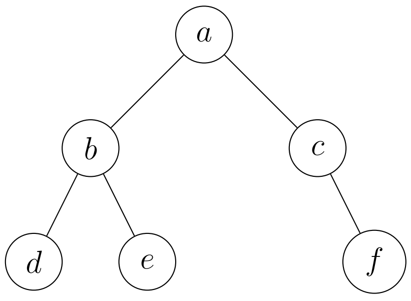


Figure 6.8: A small binary tree.

the overall goal of the algorithm being executed.

6.4.1 Preorder Traversal

A preorder traversal enumerates nodes in a root-left-right manner. This means that we process a node immediately when we first visit it, then after processing it, we next process its left child. Only after we have processed its left child and *all of its descendants*, do we return to process the right child and its descendants. However, we need a way to "remember" that the right child and its subtree still need to be traversed. As a small example, consider the tree in Figure 6.8. Starting at the root, we enumerate a. We next visit its left subtree rooted at b. When we visit b, we immediately enumerate it and again traverse left and enumerate its left child, d. Since d has no children, we backtrack to b. Having already enumerated b and its left child d, we next traverse to its right child, e. We enumerate e and since it has no children, we backtrack all the way back up to the root a and continue to its right child, c. We immediately enumerate c, but since it has no left child to visit, we traverse to its right child f as the final node. Altogether, the order of enumeration was

$$a, b, d, e, c, f$$

We can implement a preorder traversal algorithm using a stack to "remember" each right child that we need to visit and what order to visit them. Initially, we push the tree's root onto the stack. We then go into a loop that executes until the stack is emptied and we have processed all the nodes in the tree. On each iteration, we pop the node on the top of the stack and process it. We then need to process its children before we continue with other elements on the stack. To set up the next iteration, we push these child nodes onto the stack so that they are the next ones to be processed. We need to take care to exploit the LIFO ordering of the stack properly, however. We want the left child to be processed before the right child, so we push the right child first, then the left child second. The full push a

(enter loop)
pop, $node = a$
push c
push b
print a

(enter loop)
pop, $node = b$
push e
push d
print b

(enter loop)
pop, $node = d$
push h
push g
print d

(enter loop)
pop, $node = g$
push m
push l
print g

(enter loop)
pop, $node = l$
(no push)
(no push)
print l

(enter loop)
pop, $node = m$
(no push)
push r
print m

(enter loop)
pop, $node = r$
(no push)
(no push)
print r

(enter loop)
pop, $node = h$
push n
(no push)
print h

(enter loop)
pop, $node = n$
(no push)
(no push)
print n

(enter loop)
pop, $node = e$
push i
(no push)
print e

(enter loop)
pop, $node = i$
(no push)
push o
print i

(enter loop)
pop, $node = o$
(no push)
(no push)
print o

(enter loop)
pop, $node = c$
push f
(no push)
print c

(enter loop)
pop, $node = f$
push k
push j
print f

(enter loop)
pop, $node = j$
push q
push p
print j

(enter loop)
pop, $node = p$
(no push)
(no push)
print p

(enter loop)
pop, $node = q$
(no push)
(no push)
print q

(enter loop)
pop, $node = k$
(no push)
(no push)
print k

Figure 6.9: A walkthrough of a preorder traversal on the tree from Figure 6.5.

traversal is presented as Algorithm 18.

```
INPUT  : A binary tree, $T$
OUTPUT : A preorder traversal of the nodes in $T$
1 $S \leftarrow$ empty stack
2 push $T.root$ onto $S$
3 WHILE $S$ is not empty DO
4     $u \leftarrow S.pop$
5     process $u$
6     push $u.rightChild$ onto $S$
7     push $u.leftChild$ onto $S$
8 END
```

Algorithm 18: Stack-based Preorder Tree Traversal. If a node does not exist, we implicitly do not push it onto the stack.

A full preorder traversal on the binary tree in Figure 6.5 would result in the following ordering of vertices.

$$a, b, d, g, l, m, r, h, n, e, i, o, c, f, j, p, q, k$$

A full walkthrough of the algorithm on this example is presented in Figure 6.9.

A common alternative way of presenting a preorder traversal is to use recursion. This alternative version is presented as Algorithm 19. We are still implicitly using a stack to keep track of where we've come from in the tree, but instead of using a stack data structure, we are exploiting the system call stack to do this.

```
INPUT  : A binary tree node $u$
OUTPUT : A preorder traversal of the nodes in the subtree rooted at $u$
1 IF $u = null$ THEN
2     return
3 ELSE
4     process $u$
5     preOrderTraversal($u.leftChild$)
6     preOrderTraversal($u.rightChild$)
7 END
```

Algorithm 19: preOrderTraversal(u): Recursive Preorder Tree Traversal

Applications

A preorder traversal is straightforward and simple to implement. It is the most common traversal strategy for many applications that do not require tree nodes to be traversed in any particular order. It can be used to build a tree, enumerate elements in a tree collection, copy a tree, etc.

6.4.2 Inorder Traversal

An inorder traversal strategy visits nodes in a left-root-right order. This means that instead of immediately processing a node the first time we visit it, we wait until we have processed its left child and all of its descendants before processing it and moving on to the right subtree. From another we process a node when we backtrack to it (after having processed the left subtree.

Again, consider the small tree in Figure 6.8. We again start at the root a, but do not immediately enumerate it. Instead, we traverse to its left child, b. However, again we do not enumerate b until after we have enumerated its left subtree. We traverse to d and since there is no left child, we enumerate d and return to b. Now that we have enumerated b's left subtree, we now enumerate b and traverse down to its right child e and enumerate it. We now backtrack all the way back up to a. Since we have enumerated its entire left subtree, we can now enumerate a and continue to its right subtree. Since c has no left child, c is enumerated next, and then f. Altogether, the order of enumeration was

$$d, b, e, a, c, f$$

We can use the same basic idea of using a stack to keep track of tree nodes, however we want to delay processing the node until we've explored the left subtree. To do this, we need a way to tell if we are visiting the node for the first time or returning from exploring the left subtree (so that we may process it). To achieve this, we allow the node to be *null* as a flag to indicate that we are backtracking from the left subtree. Thus, if the current node is not null, we push it back onto the stack for later processing and explore the left subtree. If it is null, we pop the node on the top of the stack and process it, then push its right child to setup the next iteration. The full process is described in Algorithm 21.

```
   INPUT  : A binary tree, $T$
   OUTPUT : An inorder traversal of the nodes in $T$
1  $S \leftarrow$ empty stack
2  $u \leftarrow T.root$
3  WHILE $S$ is not empty OR $u \neq null$ DO
4      IF $u \neq null$ THEN
5          push $u$ onto $S$
6          $u \leftarrow u.leftChild$
7      ELSE
8          $u \leftarrow S.pop$
9          process $u$
10         $u \leftarrow u.rightChild$
11     END
12 END
```

Algorithm 20: Stack-based Inorder Tree Traversal

In lines 6 and 10, if no such child exists, u becomes null, indicating that we are backtracking on the next iteration.

A full inorder traversal on the binary tree in Figure 6.5 would result in the following order of vertices.

$$l, g, r, m, d, h, n, b, e, o, i, a, c, p, j, q, f, k$$

Again, a full walkthrough of the algorithm on this example is presented in Figure 6.10.

As with a preorder traversal, we can use recursion to implicitly use the call stack to keep track of the ordering. The only difference is is when we process the given node. In the preorder case, we did it before the two recursive calls. In the inorder case, we process it

(enter loop, $u = a$)
`push` a
update $u = b$

(enter loop, $u = b$)
`push` b
update $u = d$

(enter loop, $u = d$)
`push` d
update $u = g$

(enter loop, $u = g$)
`push` g
update $u = l$

(enter loop, $u = l$)
`push` l
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` l, update $u = l$
process l
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` g, update $u = g$
process g
update $u = m$

(enter loop, $u = m$)
`push` m
update $u = r$

(enter loop, $u = r$)
`push` r
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` r, update $u = r$
process r
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` m, update $u = m$
process m
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` d, update $u = d$
process d
update $u = h$

(enter loop, $u = h$)
`push` h
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` h, update $u = h$
process h
update $u = n$

(enter loop, $u = n$)
`push` n
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` n, update $u = n$
process n
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` b, update $u = b$
process b
update $u = e$

(enter loop, $u = e$)
`push` e
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` e, update $u = e$
process e
update $u = i$

(enter loop, $u = i$)
`push` i
update $u = o$

(enter loop, $u = o$)
`push` o
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` o, update $u = o$
process o
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` i, update $u = i$
process i
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` i, update $u = i$
process i
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` a, update $u = a$
process a
update $u = c$

(enter loop, $u = c$)
`push` c
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` c, update $u = c$
process c
update $u = f$

(enter loop, $u = f$)
`push` f
update $u = j$

(enter loop, $u = j$)
`push` j
update $u = p$

(enter loop, $u = p$)
`push` p
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` p, update $u = p$
process p
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` j, update $u = j$
process j
update $u = q$

(enter loop, $u = q$)
`push` q
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` q, update $u = q$
process q
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` f, update $u = f$
process f
update $u = k$

(enter loop, $u = k$)
`push` k
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` k, update $u = k$
process k
update $u =$ `null`

(done)

Figure 6.10: A walkthrough of a inorder traversal on the tree from Figure 6.5.

between them. The recursive version is presented as Algorithm 21.

```
Input   : A binary tree node $u$
Output  : An inorder traversal of the nodes in the subtree rooted at $u$
1 if $u = null$ then
2     return
3 else
4     inOrderTraversal($u.leftChild$)
5     process $u$
6     inOrderTraversal($u.rightChild$)
7 end
```

Algorithm 21: inOrderTraversal(u): Recursive Inorder Tree Traversal

Applications

The primary application of an inorder tree traversal is from where it derives its name. If we perform an inorder traversal on a binary search tree (see the next section), then we get an enumeration of elements that is sorted or "in order."

6.4.3 Postorder Traversal

The final depth-first-search based traversal strategy is a postorder traversal in which nodes are visited in a left-right-root manner. That is, we hold off on processing a node until both of its children and all of their respective descendants have been processed.

We turn once again to the small binary tree in Figure 6.8. We start at the root a and descend all the way to the left and process d since it has no children. Backtracking to b we again hold off on processing it until we've traversed its right child, e. Only after we've enumerated both d and e do we finally enumerate b. At this point, a's left subtree has been enumerated, but again we hold off on processing a until we are done with its right subtree. c has no left child, so we enumerate its right child, f first and only then do we process c. Finally, the last node to be processed is the root itself, a.

We can again utilize a stack data structure to perform a postorder traversal, but it gets a bit more complicated as we need to distinguish 3 different cases. When visiting a node, we need to know if it is the first time we've traversed it, the second time (returning from its left subtree) or the third time (returning from its right subtree, thus it needs to be processed). To do this, we keep track of not only the current node but also the previous node to know *where* we came from, either the parent, the left child, or the right child. That way, we can tell if it is the first time we've visited the node (the previous node would be the parent), the second time (the previous node is the left child) or the third time (the previous node is the right child). The full postorder traversal is presented in Algorithm 22.

```
INPUT  : A binary tree, $T$
OUTPUT : A postorder traversal of the nodes in $T$
$S \leftarrow$ empty stack
$prev \leftarrow null$
push $T.root$ onto $S$
WHILE $S$ is not empty DO
    $curr \leftarrow S.peek$
    IF $prev = null$ OR $prev.leftChild = curr$ OR $prev.rightChild = curr$ THEN
        IF $curr.leftChild \neq null$ THEN
            push $curr.leftChild$ onto $S$
        ELSE IF $curr.rightChild \neq null$ THEN
            push $curr.rightChild$ onto $S$
        END
    ELSE IF $curr.leftChild = prev$ THEN
        IF $curr.rightChild \neq null$ THEN
            push $curr.rightChild$ onto $S$
        END
    ELSE
        process $curr$
        $S.pop$
    END
    $prev \leftarrow curr$
END
```

Algorithm 22: Stack-based Postorder Tree Traversal

A full postorder traversal on the binary tree in Figure 6.5 would result in the following order of vertices.

$$l, r, m, g, n, h, d, o, i, e, b, p, q, j, k, f, c, a$$

Again, a full walkthrough of the algorithm on this example is presented in Figure 6.11.

$prev =$ `null`
`push` a

(enter loop)
update $curr = (a)$
check: $prev =$ `null`
`push` (b)
update $prev = a$

(enter loop)
update $curr = (b)$
check:
$prev.leftChild = curr$
$((b).leftChild = (b))$
`push` (d)
update $prev = b$

(enter loop)
update $curr = (d)$
check:
$prev.leftChild = curr$
$((d).leftChild = (d))$
`push` (g)
update $prev = d$

(enter loop)
update $curr = (g)$
check:
$prev.leftChild = curr$
$((g).leftChild = (g))$
`push` (l)
update $prev = g$

(enter loop)
update $curr = (l)$
check:
$prev.leftChild = curr$
$((l).leftChild = (l))$
(noop)
update $prev = l$

(enter loop)
update $curr = (l)$
check:
$prev.rightChild = curr$
$(null.rightChild = (l))$
process l
update $prev = l$

(enter loop)
update $curr = (g)$
check:
$prev.rightChild = curr$
$(null.rightChild = (g))$
check:
$curr.leftChild = prev$
$((l).leftChild = (l))$
`push` (m)
update $prev = g$

(enter loop)
update $curr = (m)$
check:
$prev.rightChild = curr$
$((m).rightChild = (m))$
`push` (r)
update $prev = m$

(enter loop)
update $curr = (r)$
check:
$prev.leftChild = curr$
$((r).leftChild = (r))$
(noop)
update $prev = r$

(enter loop)
update $curr = (r)$
check:
$prev.rightChild = curr$
$(null.rightChild = (r))$
process r
update $prev = r$

(enter loop)
update $curr = (m)$
check:
$prev.rightChild = curr$
$(null.rightChild = (m))$
check:
$curr.leftChild = prev$
$((r).leftChild = (r))$
update $prev = m$

(enter loop)
update $curr = (m)$
check:
$prev.rightChild = curr$
$(null.rightChild = (m))$
process m
update $prev = m$

(enter loop)
update $curr = (g)$
check:
$prev.rightChild = curr$
$(null.rightChild = (g))$
process g
update $prev = g$

(enter loop)
update $curr = (d)$
check:
$prev.rightChild = curr$
$((m).rightChild = (d))$
check:
$curr.leftChild = prev$
$((g).leftChild = (g))$
`push` (h)
update $prev = d$

(enter loop)
update $curr = (h)$
check:
$prev.rightChild = curr$
$((h).rightChild = (h))$
`push` (n)
update $prev = h$

(enter loop)
update $curr = (n)$
check:
$prev.rightChild = curr$
$((n).rightChild = (n))$
(noop)
update $prev = n$

(enter loop)
update $curr = (n)$
check:
$prev.rightChild = curr$
$(null.rightChild = (n))$
process n
update $prev = n$

(enter loop)
update $curr = (h)$
check:
$prev.rightChild = curr$
$(null.rightChild = (h))$
process h
update $prev = h$

(enter loop)
update $curr = (d)$
check:
$prev.rightChild = curr$
$((n).rightChild = (d))$
process d
update $prev = d$

(enter loop)
update $curr = (b)$
check:
$prev.rightChild = curr$
$((h).rightChild = (b))$
check:
$curr.leftChild = prev$
$((d).leftChild = (d))$
`push` (e)
update $prev = b$

(enter loop)
update $curr = (e)$
check:
$prev.rightChild = curr$
$((e).rightChild = (e))$
`push` (i)
update $prev = e$

(enter loop)
update $curr = (i)$
check:
$prev.rightChild = curr$
$((i).rightChild = (i))$
`push` (o)
update $prev = i$

(enter loop)
update $curr = (o)$
check:
$prev.leftChild = curr$
$((o).leftChild = (o))$
(noop)
update $prev = o$

(enter loop)
update $curr = (o)$
check:
$prev.rightChild = curr$
$(null.rightChild = (o))$
process o
update $prev = o$

(enter loop)
update $curr = (i)$
check:
$prev.rightChild = curr$
$(null.rightChild = (i))$
check:
$curr.leftChild = prev$
$((o).leftChild = (o))$
update $prev = i$

(enter loop)
update $curr = (i)$
check:
$prev.rightChild = curr$
$(null.rightChild = (i))$
process i
update $prev = i$

(enter loop)
update $curr = (e)$
check:
$prev.rightChild = curr$
$(null.rightChild = (e))$
process e
update $prev = e$

(enter loop)
update $curr = (b)$
check:
$prev.rightChild = curr$
$((i).rightChild = (b))$
process b
update $prev = b$

(enter loop)
update $curr = (a)$

check:
$prev.rightChild = curr$
$((e).rightChild = (a))$
check:
$curr.leftChild = prev$
$((b).leftChild = (b))$
`push` (c)
update $prev = a$

(enter loop)
update $curr = (c)$
check:
$prev.rightChild = curr$
$((c).rightChild = (c))$
`push` (f)
update $prev = c$

(enter loop)
update $curr = (f)$
check:
$prev.rightChild = curr$
$((f).rightChild = (f))$
`push` (j)
update $prev = f$

(enter loop)
update $curr = (j)$
check:
$prev.leftChild = curr$
$((j).leftChild = (j))$
`push` (p)
update $prev = j$

(enter loop)
update $curr = (p)$
check:
$prev.leftChild = curr$
$((p).leftChild = (p))$
(noop)
update $prev = p$

(enter loop)
update $curr = (p)$
check:
$prev.rightChild = curr$
$(null.rightChild = (p))$
process p
update $prev = p$

(enter loop)
update $curr = (j)$
check:
$prev.rightChild = curr$
$(null.rightChild = (j))$
check:
$curr.leftChild = prev$
$((p).leftChild = (p))$
`push` (q)
update $prev = j$

(enter loop)
update $curr = (q)$
check:
$prev.rightChild = curr$
$((q).rightChild = (q))$
(noop)
update $prev = q$

(enter loop)
update $curr = (q)$
check:
$prev.rightChild = curr$
$(null.rightChild = (q))$
process q
update $prev = q$

(enter loop)
update $curr = (j)$
check:
$prev.rightChild = curr$
$(null.rightChild = (j))$
process j
update $prev = j$

(enter loop)
update $curr = (f)$
check:
$prev.rightChild = curr$
$((q).rightChild = (f))$
check:
$curr.leftChild = prev$
$((j).leftChild = (j))$
`push` (k)
update $prev = f$

(enter loop)
update $curr = (k)$
check:
$prev.rightChild = curr$
$((k).rightChild = (k))$
(noop)
update $prev = k$

(enter loop)
update $curr = (k)$
check:
$prev.rightChild = curr$
$(null.rightChild = (k))$
process k
update $prev = k$

(enter loop)
update $curr = (f)$
check:
$prev.rightChild = curr$
$(null.rightChild = (f))$
process f
update $prev = f$

(enter loop)
update $curr = (c)$
check:
$prev.rightChild = curr$
$((k).rightChild = (c))$
process c
update $prev = c$

(enter loop)
update $curr = (a)$
check:
$prev.rightChild = curr$
$((f).rightChild = (a))$
process a
update $prev = a$

Figure 6.11: A walkthrough of a postorder traversal on the tree from Figure 6.5.

A recursive version would follow the same pattern as before. With a postorder traversal, we would process the node after *both* of the recursive calls. A recursive postorder traversal is presented in Algorithm 23.

```
INPUT  : A binary tree node $u$
OUTPUT : A postorder traversal of the nodes in the subtree rooted at $u$
1 IF $u = null$ THEN
2     return
3 ELSE
4     postOrderTraversal($u.leftChild$)
5     postOrderTraversal($u.rightChild$)
6     process $u$
7 END
```

Algorithm 23: postOrderTraversal(u): Recursive Postorder Tree Traversal

Applications

- Topological sorting
- Destroying a tree when manual memory management is necessary (roots are the last thing that get cleaned up)
- Reverse polish notation (operand-operand-operator, unambiguous, used in old HP calculators)
- PostScript (Page Description Language)

6.4.4 Tree Walk Traversal

It turns out that we can perform any of the three depth-first-search based tree traversals without using any extra space at all (that is, a stack is not necessary) by exploiting the oriented structure of a binary tree. As with the postorder traversal, we simply need to keep track of where we came from (a previous node) and where we are (a current node) to determine where to go next. By only keeping track of two variables, we can also determine when a node should be processed. This traversal is generally called a "tree walk" and it is illustrated on the small tree example we've been using in Figure 6.12. The traversal is essentially a "walk" around the perimeter of the tree.

The rules that we follow are quite simple and only require *local* information. In particular, there are only three cases that we need to distinguish. Suppose we are at a node u as our current node with parent p, and left/right child as ℓ, r respectively as in Figure 6.13. The rules are outlined in Table 6.2.

The full tree walk is presented as Algorithm 24 and includes all three traversal strategies.

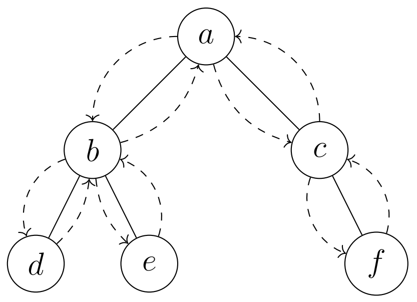


Figure 6.12: A tree walk on the tree from Figure 6.8.

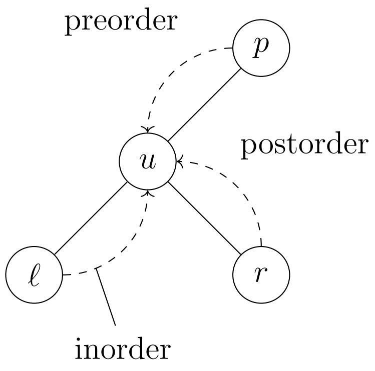


Figure 6.13: The three general cases of when to process a node in a tree walk.

Table 6.2: Rules for Tree Walk

If previous is...	Then traverse to...	And process u if order is...
p	ℓ	Preorder
ℓ	r	Inorder
r	p	Postorder

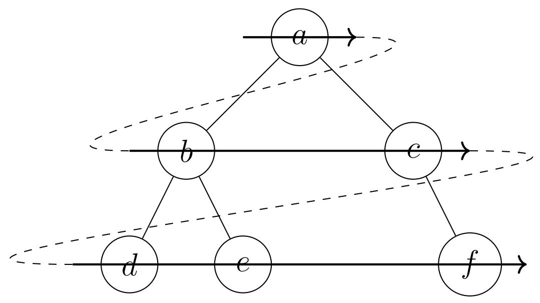


Figure 6.14: A Breadth First Search Example

The algorithm actually only keeps track of the previous node's *type* (whether it was a parent, left, or right child) rather than the node itself. It is a lot more complex than the three simple rules in Table 6.2 because we need to take care of many corner cases where the left and/or right children may not exist.

6.4.5 Breadth-First Search Traversal

An alternative to the depth-first-search based traversal strategies, we can explore a binary tree using what is known as a Breadth First Search (BFS) traversal. BFS is also a general graph traversal algorithm that explores a graph by visiting the closest vertices first before venturing deeper into the graph. When applied to an oriented binary tree, BFS ends up exploring a graph in a top-to-bottom, left-to-right ordering. In this manner, all nodes at the same depth are enumerated before moving on to the next deepest level. The basic enumeration is depicted in Figure 6.14 which represents a BFS traversal on the same graph we've been using in previous examples. Unsurprisingly, the order of enumeration matches the labeling which was chosen to be top-bottom/left-right for readability.

The visualization of the ordering should give a hint as to the implementation of BFS. If we were to stretch out the ordering it would be clear that the nodes are enumerated in one straight long line, suggesting a queue should be used. Indeed, we start by enqueueing the root node. Then on each iteration, we dequeue the next node and enqueue its children to be processed later. Since a queue is FIFO, we enqueue in the left child first. This is

```
INPUT   : A binary tree, $T$
OUTPUT : A Tree Walk around $T$
1  $curr \leftarrow T.root$
2  $prevType \leftarrow parent$
3  WHILE $curr \neq$ null DO
4      IF $prevType = parent$ THEN
           //preorder: process $curr$ here
5          IF $curr.leftChild$ $exists$ THEN
               //Go to the left child:
6              $curr \leftarrow curr.leftChild$
7              $prevType \leftarrow parent$
8          ELSE
9              $curr \leftarrow curr$
10             $prevType \leftarrow left$
11     ELSE IF $prevType = left$ THEN
           //inorder: process $curr$ here
12         IF $curr.rightChild$ $exists$ THEN
               //Go to the right child:
13             $curr \leftarrow curr.rightChild$
14             $prevType \leftarrow parent$
15         ELSE
16             $curr \leftarrow curr$
17             $prevType \leftarrow right$
18     ELSE IF $prevType = right$ THEN
           //postorder: process $curr$ here
19         IF $curr.parent =$ null THEN
               //root has no parent, we're done traversing
20             $curr \leftarrow curr.parent$
           //are we at the parent's left or right child?
21         ELSE IF $curr = curr.parent.leftChild$ THEN
22             $curr \leftarrow curr.parent$
23             $prevType \leftarrow left$
24         ELSE
25             $curr \leftarrow curr.parent$
26             $prevType \leftarrow right$
27 END
```

Algorithm 24: Tree Walk based Tree Traversal

presented as Algorithm 25.

```
INPUT  : A binary tree, $T$
OUTPUT : A BFS traversal of the nodes in $T$
1 $Q \leftarrow$ empty queue
2 enqueue $T.root$ into $Q$
3 WHILE $Q$ is not empty DO
4     $u \leftarrow Q.dequeue$
5     process $u$
6     enqueue $u.leftChild$ into $Q$
7     enqueue $u.rightChild$ into $Q$
8 END
```

Algorithm 25: Queue-based BFS Tree Traversal

Contrast the BFS algorithm with the original stack-based preorder algorithm (Algorithm 18). In fact, they are nearly identical. The only difference is that we use a queue instead of a stack (and its corresponding operations) and reverse the operations on the left/right child. This calls back to one of the major themes of this text on the importance of "smart" data structures. Simply by changing the underlying data structure used in an algorithm, it results in a fundamentally different but complementarily useful property.

6.5 Binary Search Trees

Binary trees have a lot of structure but there is still not enough to make them efficient data structures. As we've presented them so far, we can store elements and retrieve them, but without any additional ordered structure, using any one of the general traversal strategies we've developed so far is still going to be $O(n)$ in the worst case. We will now add some addition structure to our binary trees in order to attempt to make the general operations of insertion, retrieval and deletion more efficient. In particular, it will be our goal to make all three of these operations have complexity that is proportional to the depth of the tree, $O(d)$. To do that, we introduce Binary Search Trees (BSTs).

Definition 9 (Binary Search Tree)**.** A *binary search tree* is a binary tree such that every node u has an associated key, $u.key$ which satisfies the *binary search tree property*:

1. Every node in the left subtree of u has a key value *less* than $u.key$

2. Every node in the right subtree of u has a key value *greater* than $u.key$

Implicitly, this definition does not allow for duplicate key values in a binary tree. In general, we could amend this definition to allow duplicate keys but we would need to be consistent on if we place duplicates in the left or right subtree. In any case, it is not that big of a deal. In general, we can “break ties” using some unique value to each objects to induce a *total ordering* on all elements stored in a binary search tree.[8]

Furthermore, though we will use integer key values in all of our examples, in practice binary search trees need not be restricted to such values as they are intended to hold any type of object or data that we wish to store. Typically, an implementation will use a *hash value* of an object or data as a key value. A hash value is simply the result of applying a hash function that maps an object to an integer value. This is such a common operation it is built into many programming languages.

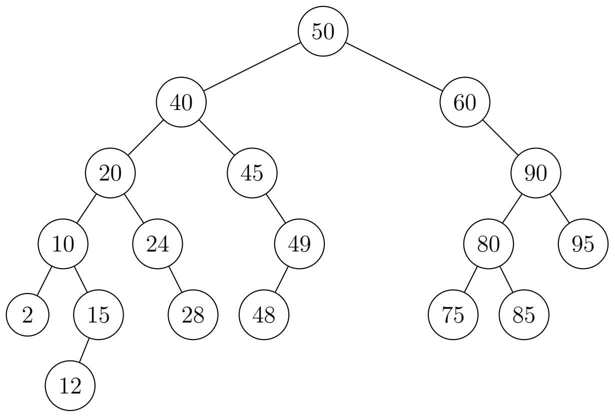


Figure 6.15: A Binary Search Tree

The binary search tree property is an *inductive* property. Since it holds for all nodes, it follows that every rooted subtree in a BST is also a binary search tree. A full example can be found in Figure 6.15. Note that a binary search tree need not be full or complete as in the example. Many children (and thus entire subtrees) are not present. The binary search tree property merely guarantees that all keys in the less subtree are less and those in the right subtree are greater than the key stored in the root respectively.

6.5.1 Retrieval

We can exploit the binary search tree property similar to how binary search exploits a sorted array. With binary search, we check the middle element and either find what we were searching for or we have effectively cut the array in half as we know that we should either search in the lower half or the upper half. Similarly, if with a binary search tree, we start a search for a particular key k at the root. If the root's key value is greater than k ($k < T.root.key$) then we know that the key lies in its left subtree while if the search key value is greater ($T.root.key < k$) then we know that it must lie in the right subtree. In either case, we continue the search on the tree rooted at the left or right child respectively. If we ever find a node such that the key value matches we can stop

[8]In practice, all things being equal, we could use the memory address of two objects stored in a tree to break ties.

the search and return the node's value. Of course, not every search will necessarily be successful. If we search for a key value that does not exist in the tree, we will eventually reach the end of the tree at some leaf node. If this happens, we can stop the process and return a flag indicating an unsuccessful search. Several examples are illustrated in Figure 6.16.

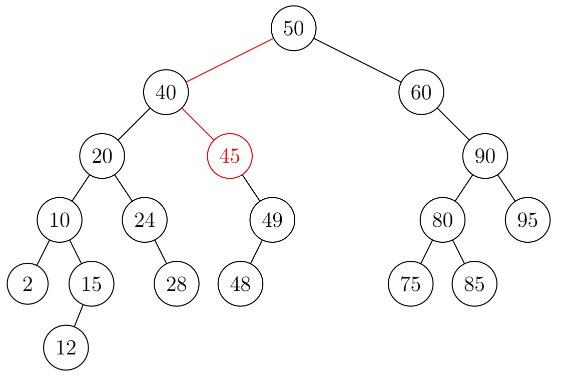
In some of the examples we got lucky and did not have to search too deeply in the tree. In fact, if we had searched for the key value $k = 50$ only a single key comparison would have been required. Thus, in the best case only $O(1)$ comparisons are necessary for a search/retrieval operation. In the worst case, however, we had to search to the deepest part of the tree as in the case where we successfully searched for $k = 12$ and unsuccessfully searched for $k = 13$. In both cases we made 6 key comparisons before ending the search. In general, we may need to make d comparisons where d is the depth of the tree. This achieves what we set out to do as a retrieval operation is $O(d)$. The full search process is presented as Algorithm 26.

```
INPUT  : A binary search tree, $T$, a key $k$
OUTPUT : The tree node $u \in T$ such that $u.key = k$, $\phi$ if no such node exists.
1  $u \leftarrow T.root$
2  WHILE $u \neq \phi$ DO
3      IF $u.key = k$ THEN
4          output $u$
5      ELSE IF $u.key > k$ THEN
6          $u \leftarrow u.leftChild$
7      ELSE IF $u.key < k$ THEN
8          $u \leftarrow u.rightChild$
9      END
10 END
11 output $\phi$
```

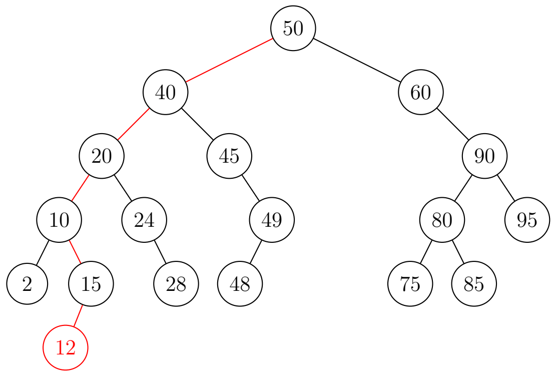
Algorithm 26: Search algorithm for a binary search tree

6.5.2 Insertion

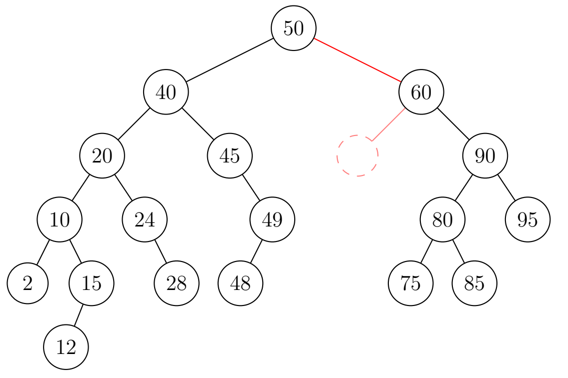
Inserting a new node into a binary search tree is a simple extension to searching for a node. In fact, it is exactly the same process. Since we will not allow duplicate key values, if we perform a search and find that a node with the given key value already exists, we can make the same design decisions we used with lists (throw an exception, return an error flag, or treat it as a no op). Otherwise, we perform the same search and when we reach the end of the tree, we insert the node at the spot where we would otherwise expect it to be.



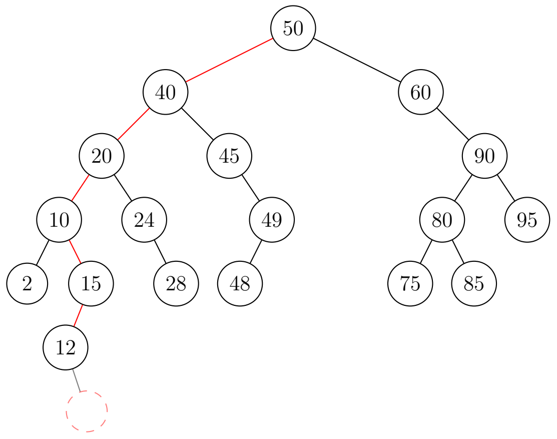
(a) Searching a BST Example 1. In this example, we search for the key value $k = 45$. The search starts at the root. Since $45 < 50$ we traverse to the left child. The next comparison finds that $40 < 45$ so we traverse right. Finally, the last comparison finds that the key value matches at which point the search process terminates.



(b) Searching a BST Example 2. In this example, we search for the key value $k = 12$. The search starts at the root. Since $12 < 50$ we traverse to the left child. The search then traverses left twice more. When compared to 10, it traverses right and finally left, finding the key value at the deepest leaf in the tree. A total of 6 comparisons was necessary.



(c) Searching a BST Example 3. In this example, we search for the key value $k = 55$ which will result in an unsuccessful search. Starting at the root, we traverse right, then left at which point we are no longer in the tree. With only 2 comparisons, we are able to conclude and return a flag value indicating an unsuccessful search.



(d) Searching a BST Example 4. Another unsuccessful search when we search for the key value $k = 13$. Similar to example 6.16(b). With the same 6 comparisons, we've traversed off the tree and return our flag value.

Figure 6.16: Various Search Examples on a Binary Search Tree

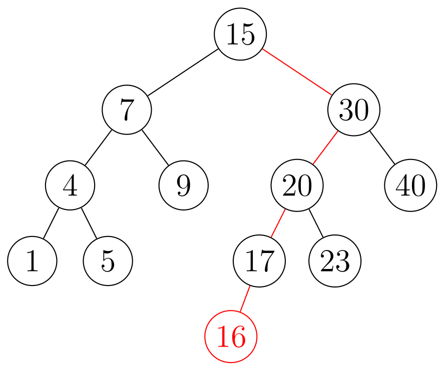


Figure 6.17: Insertion of a new node into a binary search tree. A search is performed for the key $k = 16$, when the open slot is found, the node is inserted.

This is illustrated in Figures 6.16(c) and 6.16(d) where a dashed "phantom" node is indicated where the key values would otherwise have been. When performing a search we will always be guaranteed to be inserting the new node as a leaf node. We'll never insert a new node in the middle of the tree so no other considerations are necessary. Once we have performed the search, creating a new node and inserting it only requires a few reference shuffles. Thus, using the same analysis as before, in the worst case, insertion is also $O(d)$. Developing the pseudocode for the insertion operation is left as an exercise (see Exercise 6.16). A small example is demonstrated in Figure 6.17.

6.5.3 Removal

The removal of keys also follows a similar initial search operation. Given a key k to delete, we traverse the binary search tree for the node containing k which is an $O(d)$ operation. Suppose the node we wish to delete is u. There are several cases to take care of depending on how many children u has.

Case 1: Suppose that u is a leaf node and has no children. It is a simple matter to delete it by simply changing its parent node's reference. Though we do need to check if u is the left or right child. In particular,

- If $u = u.parent.leftChild$ then set $u.parent.leftChild \leftarrow \phi$; otherwise
- If $u = u.parent.rightChild$ then set $u.parent.rightChild \leftarrow \phi$.

Case 2: Suppose that u has only one child (left or right) and is missing the other child. We don't want to "prune" the subtree of its child, but at the same time we want to preserve the binary search tree property. To do this we simply "promote" the only child up to u's position. Specifically, suppose that u's child is a left child. The steps to to promote it are nearly exactly the same as used to delete (circumvent) a node in a linked list.

If u is a left child then:

- $u.parent.leftChild \leftarrow u.leftChild$
- $u.leftChild.parent \leftarrow u.parent$

Otherwise if u is a right child then:

- $u.parent.rightChild \leftarrow u.rightChild$
- $u.rightChild.parent \leftarrow u.parent$

The operations for the case that u only has a right child is analogous. An example of this operation is depicted in Figures 6.18(c) and 6.18(d).

Case 3: Suppose that u has both of its children. Our first instinct may be to promote one of its children up to its place similar to the previous case. In some instances this might work. For example, consider the tree in Figure 6.15. If we wanted to delete the root, 50, we could promote its right child, 60 up to its place. However, that is only because 60 has no left child. If it had a left child (say 55), then it would not be possible as promoting 60 up would mean it would then have to have 3 children (40, 55, 90).

There are many potential solutions to this problem, but we want to ensure that the operation, in addition to preserving the binary search tree property, is simple, efficient, and causes a minimal change to the tree's structure. Instead of immediately deleting the node, let us instead remove the key, resulting in an "empty" node. We will want to backfill this empty node with a key value in the tree that preserves the BST property.

Again, we exploit the binary search tree property. Because the subtree rooted at u is also a BST, every key in its left subtree is less than its key and every key in its right subtree is greater. Thus, there are two candidates that we could "promote": the maximum value in the left subtree or the minimum value in the right subtree. Suppose we go with the first option, By replacing u with the the maximum value in the left subtree, we still ensure that all elements in the left subtree are less than it (because it was the maximum value) and all nodes in the right subtree are still greater than it (since it was less than u to begin with).

There are two issues that we need to deal with, however. One is finding the maximum (or minimum) value in a subtree and the other is how do we delete the node containing the maximum value from the tree so that it can take the place of u? We'll deal with the second problem first. Once we've found the node containing the maximum, if it has zero or two children, we already know how to deal with that from the two previous cases. The only case we really have to deal with is if it has two children. However, we only have to think for a moment that this is, in fact, impossible! Suppose that a node, x contained the maximum key value and it had both children. By the binary search tree property, its right child, $x.rightChild$ necessarily has a key value greater than i, thus x cannot contain the maximum key value! That is, necessarily the maximum must only have at most 1 child and we know how to deal with that situation.

How do we actually find the maximum value? Again, we exploit the binary search tree property. We first traverse to the left child. From there, we only have to keep traversing right until there is no longer a right child. Conversely, to find the minimum value in the right subtree, we traverse to the right child and then keep traversing left until there is no left child. The process for the former choice is presented as Algorithm 27.

```
INPUT  : A node $u$ in a binary search tree with two children.
OUTPUT : The node containing the maximum key value in $u$'s left subtree.
1 $curr \leftarrow u.leftChildt$
2 WHILE $curr.rightChild \neq \phi$ DO
3     $curr \leftarrow curr.rightChild$
4 END
5 output $curr$
```

Algorithm 27: Finding the maximum key value in a node's left subtree.

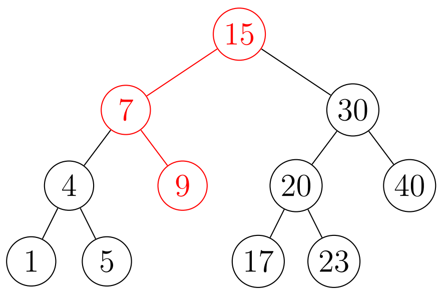
Examples of cases 2 and 3 can be found in Figure 6.18. The full delete operation has involves several subroutines. Finding the node to be deleted (or determining it does not exist) is $O(d)$. Finding the minimum in the left subtree is also at most $O(d)$. Ultimately, swapping the keys and/or references is $O(1)$. Since these are all independent operations, we have the total complexity as $O(d) + O(d) + O(1) = O(d)$.

6.5.4 In Practice

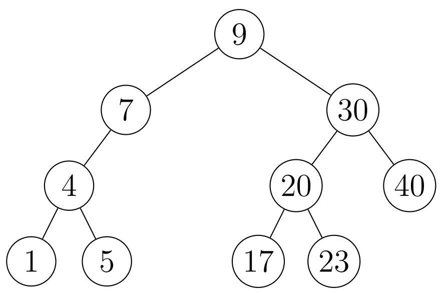
We achieved our goal of developing a tree-based data structure that offered $O(d)$ insert, retrieval, and update operations. However, we have still fallen short of our main goal of ensuring that these operations are all $O(\log n)$. The reason for this is that a binary search tree may become *skewed* or *degenerate*. Consider the binary search tree that would be formed if we inserted the elements $10, 20, 30, \ldots, 100$ in that order. The resulting tree would look that that in Figure 6.19. This tree is degenerate because its depth is linear with respect to the number of nodes in the tree; $d = n - 1 \in O(n)$. This should look familiar: a degenerate binary search tree is essentially a linked list!

In practice trees will not always be degenerate. However, there is no guarantee that the depth with be logarithmic. Yet more structure would be necessary to make this guarantee. There are plenty of examples of *balanced binary search trees* such that insert and remove operations reorder or *rebalance* a tree so that for every node the left/right subtrees are roughly equal in depth, ensuring that $d \in O(\log n)$. Each one has its own advantages and complexity in operations such as AVL trees, 2-3/B-trees, Red-Black trees, splay trees, and treaps. We will not examine such data structures here, but know that it is possible to guarantee that the depth, and thus the basic operations are all efficient, $O(\log n)$.

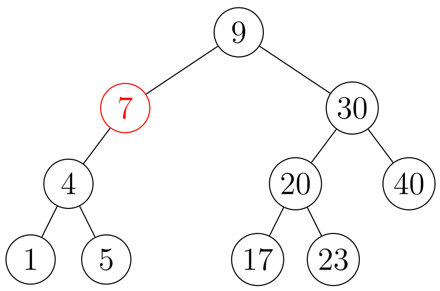
Many programming languages also offer standard implementations for binary search trees, in particular balanced BSTs. Java, for example, provides a `TreeSet` (as well as a



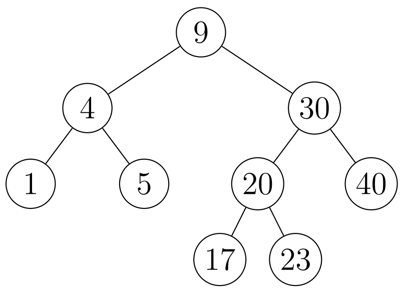
(a) Deletion of a node with two children (15). First step: find the maximum node in the left-sub-tree (lesser elements).



(b) Node 15 is replaced with the extremal node, preserving the BST property



(c) Deletion a node with only one child (7).



(d) Removal is achieved by simply promoting the single child/subtree.

Figure 6.18: BST Deletion Operation Examples. Figures 6.18(a) and 6.18(b) depict the deletion of a node (15) with two children. Figures 6.18(c) and 6.18(d) depict the deletion of a node with only one child (7).

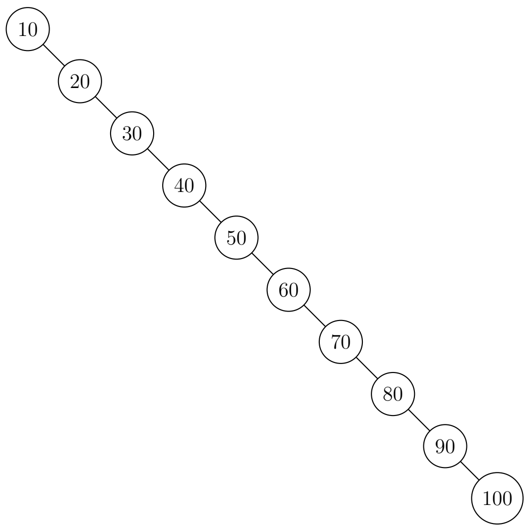


Figure 6.19: A degenerate binary search tree.

`TreeMap`) which is a red-black tree implementation with guaranteed $O(\log n)$ operations. Moreover, the default iterator pattern is an inorder traversal, providing a sorted ordering. It is usually used with one of its interfaces, a `SortedSet` which allows you to use a `Comparator` for ordering.

6.6 Heaps

We may have fallen short of presenting a guaranteed efficient general tree data structure, however we will present a related data structure that does provide efficiency guarantees. Like stack and queues, however, this data structure's efficiency will come at the cost of restricting access to its elements.

Definition 10 (Heap)**.** A *heap* is a binary tree of depth d that satisfies the following properties.

1. It is a *full* up to level $d-1$. That is, every node is present at every level up to and including level $d-1$.
2. At level d all nodes are full-to-the-left. That is, all nodes at the deepest level are all as far left as possible.
3. It satisfies the *heap property*; every node has a key value that is greater than *both* of its children.

This definition actually defines what is referred to as a *max-heap*. This is because as a consequence of the heap property, the maximum element is always guaranteed to be at the root of the heap. Alternatively, we could redefine a heap so that every node's children has key values that are greater than it. This would define a *min-heap*. In practice the distinction is unimportant and a comparator is usually used to define order. Thus, you can get a min-heap implementation using a max-heap with a comparator that reverses the usual ordering. A larger min-heap example can be found in Figure 6.20.

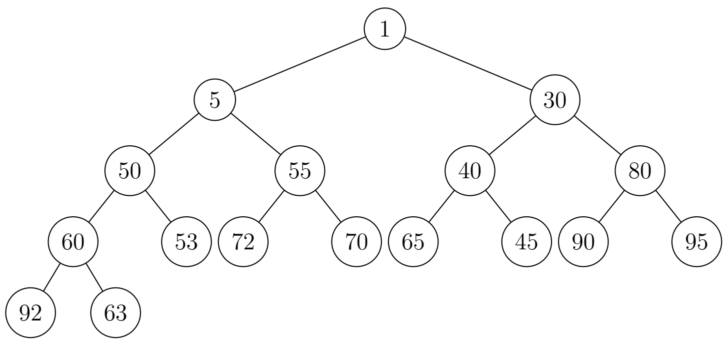


Figure 6.20: A min-heap

As depicted it is easy to see the fullness properties. Every node is present at every level except the last one. In the deepest level, the two remaining nodes are as far left as possible (the children of 60). Essentially there are no "gaps" in any of the levels. This fullness property guarantees that a heap's depth will always be logarithmic, $O(\log n)$.

However, it is easy to see that a heap does not provide too much additional structure on the ordering of elements other than the fact that the maximum element is at the top of the heap. In fact, the structure of a heap resembles a real heap: it is an unorganized pile of stuff. If we were to throw something on top of a real heap, we may not have much control on where it ends up. The operations on a heap data structure are similar. We cannot perform efficient arbitrary searches on a heap because of this lack of structure. We can, however, support two core restricted access operations. We can add elements to the heap and we can remove the top most (getMax) element.

6.6.1 Operations

To develop the two core operations, we'll assume that we are working with a max-heap as in Definition 10. We'll use the smaller max-heap example in Figure 6.21 in our examples.

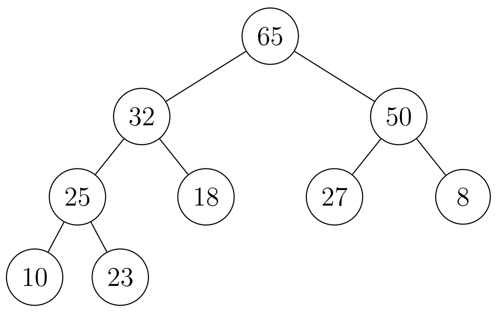


Figure 6.21: A Max-heap

Insert Key

The first operation is to insert a new key into the heap. Our first instinct may be to start at the root as we did with binary search trees. We could then move down left/right in the heap in some manner to insert a new key, but how? Suppose we were to insert a new key, $k = 90$ into the max-heap in Figure 6.21. Clearly it would need to become the new root of the heap, displacing 65. We could continue to shove 65 down displacing either its left or right child. However, what criteria would we use? With only local information to go on, we may choose to exchange with the larger of the two children. Following this criteria we would then exchange 65 and 50, 50 and 27, resulting in something that looks like Figure 6.22.

The problem is that this is not a valid heap as it does not fulfill the fullness property.

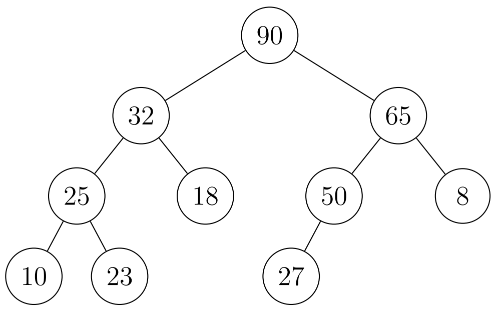


Figure 6.22: An Invalid Max-heap

One might be tempted to modify this approach and instead always exchange the inserted node with the lesser of the two children. This strategy would work with this *particular* example, but we could easily come up with a counter example in which it fails (in fact, the same example, just swap the keys 32 and 50 and it will still fail).

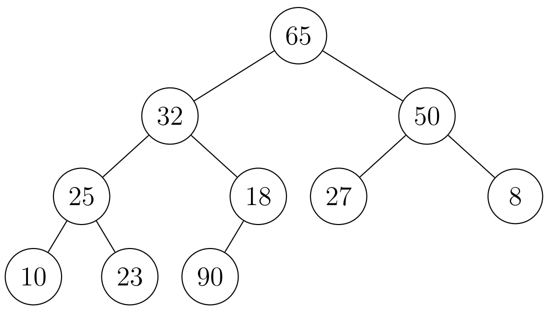
Let's take a step back and redevelop an approach that first ensures that the fullness property is satisfied but the heap property need not necessarily be preserved (at first at least). Preserving the fullness property means that we would necessarily insert the key at the deepest level at the left-most available spot. For the example in Figure 6.21, this would mean inserting as the left child of 18. Inserting 90 in this manner clearly violates the heap heap property, however, so we need to "fix" the heap. This process is known as *heapifying*. We exchange the inserted key with its parent until either 1) the heap property is satisfied or 2) we've reached the root node and the inserted key has become the new root of the heap. This process is fully illustrated in Figure 6.23.

The heapify process is presented as Algorithm 28 which assumes that the new key has already been inserted in the next available spot. The algorithm makes key comparisons, continually exchanging keys until the heap property is satisfied or it reaches the root. This pseudocode, as presented, assumes a tree-based implementation.

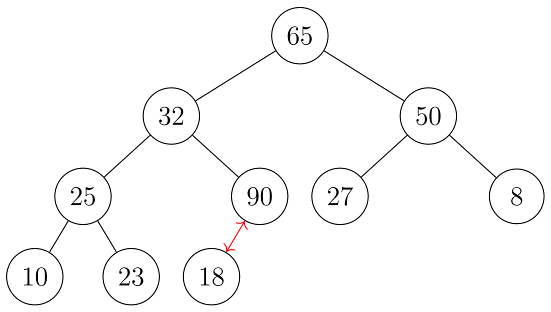
```
   INPUT   : A heap $H$ and an inserted node $u$
1  $curr \leftarrow u$
2  WHILE $curr.parent \neq \phi$ and $curr.key > curr.parent.key$ DO
3      swap $curr.key$, $curr.parent.key$
4      $curr \leftarrow curr.parent$
5  END
```

Algorithm 28: Heapify: fixing a heap data structure after the insertion of a new node, u.

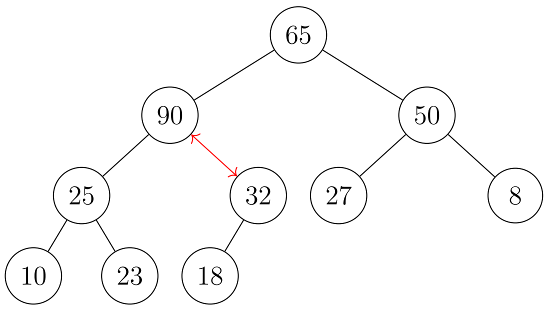
For the moment we will assume that we have an easy way to insert a node at the next available spot. The heapify algorithm itself makes at most d comparisons and swaps in the worst case (as was the case in the example in Figure 6.23). If we don't need to



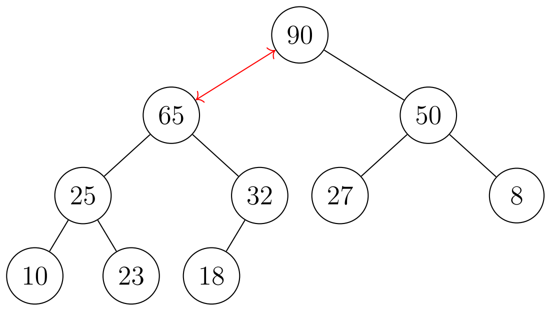
(a) The new key, $k = 90$ is inserted at the next available spot.



(b) The first comparison finds that 90 and its parent 18 are out of order and exchanges them.



(c) The second comparison ends up exchanging 90 and 32.



(d) The third and final comparison ends up promoting 90 up to the new root.

Figure 6.23: Insertion and Heapification.

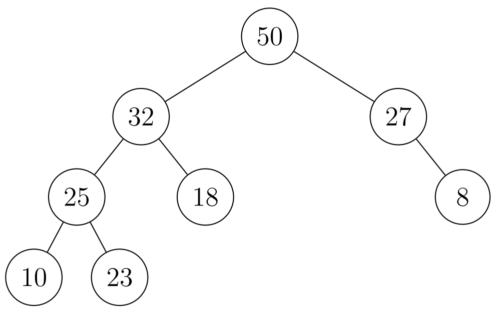


Figure 6.24: Another Invalid Heap

heapify all the way up to the root node, then even fewer comparisons and swaps would be necessary, but ultimately the process is $O(d)$. However, since a heap guarantees the fullness property, $d = O(\log n)$, the process only requires a logarithmic number of comparisons/swaps in the worst case.

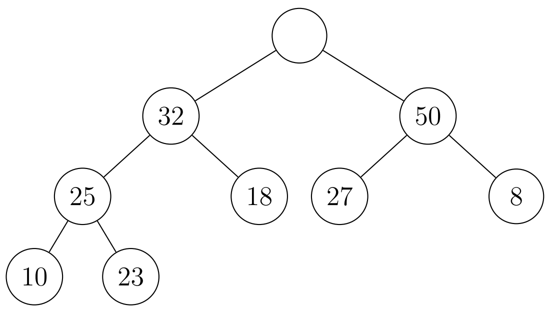
Retrieve Max

The other core operation is to get the maximum element. Because of the heap property, this is guaranteed to be the root element. First, we remove the key/value from the root node (which we have immediate access to) and save it off to eventually return it. However, this leaves a gap in the tree that must be filled to preserve the fullness property.

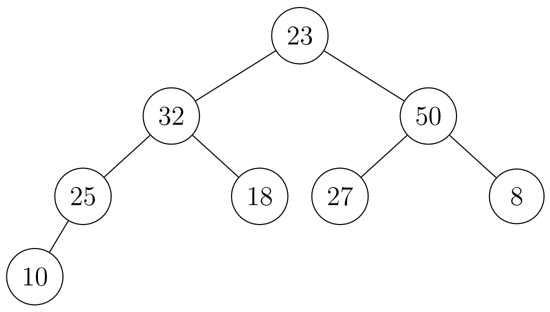
Once again we may be tempted promote one of the root's children to fill its place. To preserve the heap property, we must promote the larger of the two children. Of course we would have to continue down the heap as the promotion leaves another gap eventually promoting a leaf element up. This idea alone will not suffice however. Again, consider the heap from Figure 6.21. Were we to remove the root element 65 and proceed with the operation as described we would move up 50 then 27 resulting in a "heap" as in Figure 6.24. Obviously this does not fulfill the fullness property as there is a gap (27's left child is missing).

As before, the solution is to prioritize the fullness property. Once we remove the root element, we need to backfill it with a node that will preserve the fullness property, specifically the "last" node (the rightmost node at the deepest level). Replacing the root element with the last node may not necessarily satisfy the heap property, but we can perform a top-down heapify process similar to what we did with the insert operation. This process is illustrated properly in Figure 6.25.

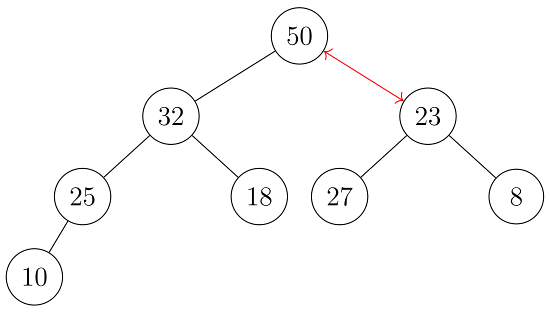
We leave the development of the algorithm and pseudocode as an exercise (see Exercise 6.17). Assuming, as before, that we have easy access to the last element, then swapping it and the the root is an $O(1)$ operation. The heapification may again require up to d comparisons and swaps if the last element must be exchanged all the way down to the bottom of the heap. As before, since $d = O(\log n)$ this operation is $O(\log n)$.



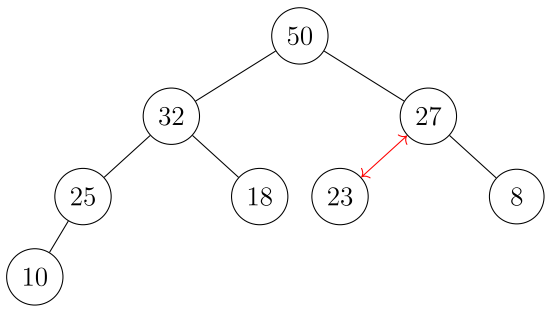
(a) The root element, 65 is saved off in a temporarily variable.



(b) The last element, 23 replaces the root node. However, the heap property is not satisfied.



(c) The node is exchanged with the *greater* of the two children to preserve the heap property.



(d) Another exchange must take place before the heap is fixed.

Figure 6.25: Removal of the root element (getMax) and Heapification

Secondary Operations

The heap properties don't allow us to do general operations such as arbitrary search and remove efficiently. We can achieve them using any of our tree traversal strategies, but all of these operations have the potential to be $O(n)$.

However, there are algorithms in which a heap is used that may require that keys in a heap be changed (either increase or decrease their key values). A primary example is when a heap is used to implement a priority queue and we wish to change the priority of an element that is already enqueued.

First, we assume that we have "free" access to the node whose key we want to change as finding an arbitrary node is going to be $O(n)$ as previously noted. Once we have the node u, we can increase or decrease its key value. Suppose we increase it. Its key value was already greater than all of its descendants, so the heap property with respect to the subtree rooted at u is still satisfied. However, we may have increased the key value such it is now greater than its parent. We will need to once again, heapify and exchange keys with its parents and ancestors until the heap property is once again satisfied. This is exactly the same process as when we inserted a new node at the bottom.

Likewise, we can support a decrease key operation. Since the key value was already smaller than its parent, the heap property is unaffected with respect to u's ancestors. In this case, however, the heap property could be violated with its children and/or descendants. We simply exchange the key with the larger of its two children if the heap property is violated and continue *downward* in the heap just as we did with the retrieve (and remove) the maximum element. Both of.these operations are simply $O(\log n)$ as before.

6.6.2 Implementations

So far we have presented heaps as binary trees. It is possible to implement heaps using the same tree nodes and structure as with binary trees, but it does present some challenges that we'll deal with later. Another, easier and simpler implementation, however is to come full circle back to array-based lists.

Array-Based Implementation

Recall that the fullness property of a heap essentially means that all nodes are *contiguous*. It makes sense, then that we can store them in an array. In particular, we will store the root element at index 1 (we will leave index 0 unused for this presentation though you can easily shift all of the elements by one index if you'd rather not waste it).

Now, suppose a node u is stored at index i. Given this index, we need to locate u's parent and left and right child. These will each be stored at:

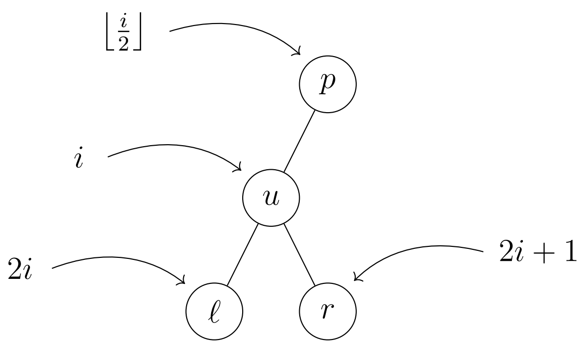


Figure 6.26: Heap Node's Index Relations

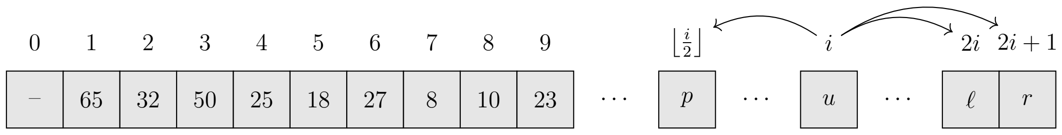


Figure 6.27: An array implementation of the heap from Figure 6.21 along with the generalized parent, left, and right child relations.

- The left child is at index $2i$
- The right child is at index $2i + 1$
- The parent is at index $\left\lfloor \frac{i}{2} \right\rfloor$

These relations are illustrated in Figure 6.26.

A full example is presented in Figure 6.27 which implements the same small max-heap example we've been using. It also demonstrates the backward and forward relationships to the parent and children nodes. Observe that the order in the array matches the order from the tree if we were to perform a breadth first search on it (recall Figure 6.14).

A clear advantage to this implementation is that it is extremely easy and we can even reuse an array-based list implementation. It is a simple matter to implement the heapify algorithms in terms of indices given the mappings above (see Exercise 6.18). One disadvantage is that it is costly to increase the size of the array when we need to add more elements. However, amortized over the life of the heap and given the "savings" in a simpler implementation, this may not be that big of a deal.

Tree-Based Implementation

Though not common, you can implement a heap using the same binary tree structure using a node with references to a parent, left child, and right child. However, to ensure efficient operation for the heapify algorithms, we necessarily have to keep track of a parent element for every node.

One problem is that we do not have obvious access to the "last" element or next available spot in the heap as we did with an array-based implementation. With an array, we could easily keep track of how many elements have been stored in the heap, say n. Then the last element is necessarily at index n and the next available spot is at index $n + 1$. Since we have random access, we can easily jump to either location to get the last element or to insert at the next available spot.

With a tree structure, however, we do not have such access. Though we keep track of the root, it is not straightforward to also keep track of the last element (or the next available spot). Searching for either using BFS or another tree traversal algorithm would kill our $O(\log n)$ efficiency.

Fortunately, using a bit of mathematical analysis and exploiting the fullness property of a heap will allow us to find either the last element or the first available spot in $O(\log n)$ time. We'll focus on finding the first available open spot as the same technique can be used to find the last element with minor modifications.

Without loss of generality, we'll assume that we've kept track of the number of nodes in the heap, n and thus we can compute the depth,

$$d = \lfloor \log n \rfloor$$

Due to the fullness property, simply knowing n and d give us a great deal of information. In particular, we know that there are

$$\sum_{k=0}^{d-1} 2^k = 2^d - 1$$

nodes in the first d levels (levels 0 up to $d - 1$). Computing a simple difference,

$$m = n - (2^d - 1)$$

tells us how many nodes are in the last (deepest) level, level d. We also know that if the last level were full, it would have 2^d nodes. This tell us whether or not the next available spot is in the left subtree or the right subtree by making a simple comparison:

- If $m < \frac{2^d}{2} = 2^{d-1}$ then the left subtree is not "full" at level d and so it contains the next available spot.
- Otherwise if $m \geq 2^{d-1}$ then the left subtree is full and the right subtree must contain the next available spot.

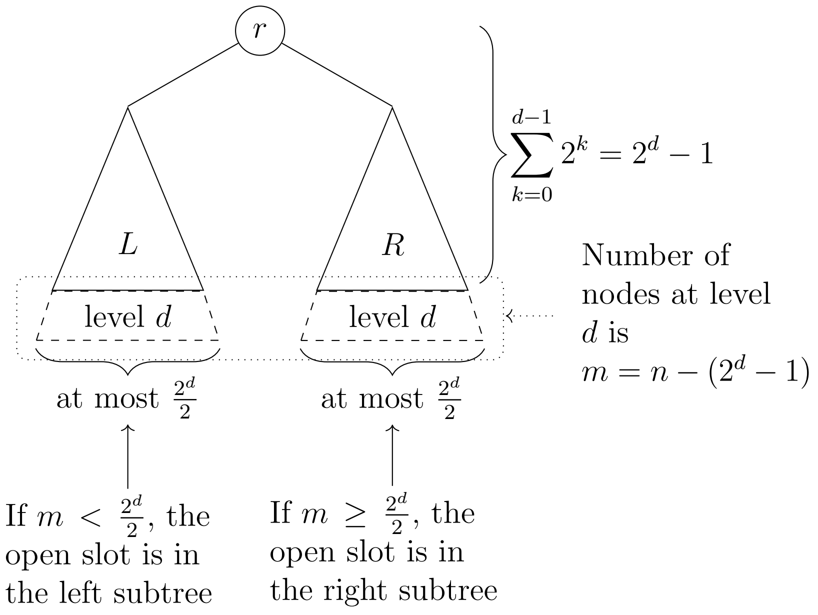

r
L
R
$\sum_{k=0}^{d-1} 2^k = 2^d - 1$
level d
level d
Number of nodes at level d is $m = n - (2^d - 1)$
at most $\frac{2^d}{2}$
at most $\frac{2^d}{2}$
If $m < \frac{2^d}{2}$, the open slot is in the left subtree
If $m \geq \frac{2^d}{2}$, the open slot is in the right subtree

Figure 6.28: Tree-based Heap Analysis. Because of the fullness property, we can determine which subtree (left or right) the "open" spot in a heap's tree is by keeping track of the number of nodes, n. This can be inductively extended to each subtree until the open spot is found.

In each case we can traverse down to the left or right child respectively (which ever's tree contains the next available spot) and update both n and m (the number of elements at level d) and repeat this process until we've found the next available spot. This analysis is visualized in Figure 6.28 and the full process is presented as Algorithm 29.

It is not difficult to see that the complexity of this algorithm is $O(d) = O(\log n)$ since we iterate down the depth of the heap using simple arithmetic operations. Thus the problem of finding the last element or the next available spot does not significantly increase the complexity of the heap's core operations.

```
    INPUT  : A tree-based heap $H$ with $n$ nodes
    OUTPUT : The node whose child is the next available open spot in the heap
 1  curr ← $T.head$
 2  $d \leftarrow \lfloor \log n \rfloor$
 3  $m \leftarrow n$
 4  WHILE $curr$ has both children DO
 5      IF $m = 2^{d+1} - 1$ THEN
            //remaining tree is full, traverse all the way left
 6          WHILE $curr$ has both children DO
 7              $curr \leftarrow curr.leftChild$
 8          END
 9      ELSE
            //remaining tree is not full, determine if the next open
              spot is in the left or right sub-tree
10          IF $m \geq \frac{2^d}{2}$ THEN
                //left sub-tree is full
11              $d \leftarrow (d - 1)$
12              $m \leftarrow (m - \frac{2^d}{2})$
13              $curr \leftarrow curr.rightChild$
14          ELSE
                //left sub-tree is not full
15              $d \leftarrow (d - 1)$
16              $m \leftarrow m$
17              $curr \leftarrow curr.leftChild$
18          END
19      END
20  END
21  output $curr$
```

Algorithm 29: Find Next Open Spot - Numerical Technique

6.6.3 Variations

As presented in Definition 10, the heaps we have been describing are sometimes referred to as *binary heaps* because they are based on a binary tree. There are other variations of heaps that uses different structures and offer different properties.

For example, Binomial Heaps are heaps that are a collection of binomial trees. A binomial tree of order k is a tree such that its children are binomial trees of order $k-1, k-2, \ldots, 2, 1, 0$ (that is, it has k children). This is an inductive definition so that the base case of $k = 0$ is a single node tree. A binomial tree of order k has 2^k nodes and is of depth k. The core operations are a bit more complicated but have the same complexity of $O(\log n)$. The advantage of this implementation is that a *merge* operation of two heaps is also efficient ($O(\log n)$).

A Fibonacci heap is a collection of trees that satisfy the heap property. However, the structure is a lot more flexible than binary heaps or binomial heaps. The main advantage is that some of the operations to keep track of elements are delayed until they are necessary. Thus, some operations may be quite expensive, but when looked at from an *amortized analysis*, the expected or average running time of the operations can be interpreted as constant, $O(1)$. This is similar to when we examined array-based lists. Sometimes (though not often) we may need to expand the underlying array. Though this is an expensive operation, since it is not too common, when you average the running time of the core operations over the lifetime of the data structure, it all "evens out." and looks to be constant.

6.6.4 Applications

A heap is used in many algorithms as a data structure to efficiently hold elements to be processed. For example, several graph algorithms such as Prim's Minimum Spanning Tree or Dijkstra's Shortest Path algorithms use heaps as their fundamental building block. A heap is also used in Huffman's Coding algorithm to efficiently compress a file without loss of information.

Priority Queue

From the core operations (insert and getMax) it might be obvious that a heap is an efficient way to implement a priority queue. The enqueue operation is a simple insert and a dequeue operation is always guaranteed to result in the maximum (highest priority) element. Since both operations are $O(\log n)$, this is far more efficient than a list-based priority queue. Note that the Java Collections library provides a heap-based priority queue implementation in the `java.util.PriorityQueue<E>` class.

Heap Sort

Suppose we have a collection of elements. Now suppose we threw them all into a heap and then, one-by-one, pulled them out using the getMax operation. As we pull them out, what order would they be in? Sorted of course! By using a sophisticated data structure like a heap, we can greatly simplify the code to achieve a more complex data operation, sorting. This is referred to as Heap Sort and is presented as Algorithm 30.

```
   INPUT   : A collection of elements $A = \{a_1, \ldots, a_n\}$
   OUTPUT : $A$, sorted
 1 $H \leftarrow$ empty min-heap
 2 FOREACH $a \in A$ DO
 3     insert $a$ into $H$
 4 END
 5 $i \leftarrow 1$
 6 WHILE $H$ is not empty DO
 7     $a_i \leftarrow H.getMin$
 8     $i \leftarrow (i + 1)$
 9 END
10 output $A$
```

Algorithm 30: Heap Sort

Examine at the code in this algorithm. It is extremely simple; we put stuff into a data structure, then we pull it out. It doesn't get much simpler than that. The real magic is in the data structure we used and exploited. This is a perfect illustration of one of the themes of this book, borrowed from Eric S. Raymond [8] that "Smart data structures and dumb code are a lot better than the other way around." There are many sophisticated sorting algorithms (Quick Sort, Merge Sort, etc.) that use clever code (recursion, partitioning, etc.) and simple arrays. In contrast, Heap Sort uses very dumb code: put stuff in, take stuff out and a very smart data structure. Beautiful.

Of course, this beauty is all for nothing if it is not also efficient. The complexity of the insertion and getMin operations in Heap Sort actually changes on each iteration of the algorithm because the data structure we're using is growing and shrinking. Let's first analyze the "put stuff in" phase (the foreach loop, lines 2–3). On the first iteration, no comparisons or swaps are made as H is initially empty. On the second iteration where we insert the second element, H has size 1 and so 1 comparison (and potentially 1 swap) is made. In general on the i-th iteration, H has $i-1$ elements in it, thus the insertion requires roughly $\log i - 1$ (the depth of the heap) comparisons and/or swaps. This is all summarized in Table 6.3.

Thus, the total number of comparisons (or swaps) made by Heap Sort is the summation of the 4th column, or

$$\sum_{i=2}^{n-1} \log i = \log 2 + \log 3 + \cdots + \log n - 1$$

Note that we start the index at $i = 2$, since the first iteration does not require any comparisons. Using logarithm identities, we can further simplify and bound this summation:

Table 6.3: Analysis of Heap Sort

Iteration	Size of H	Depth of H	Number of Comparisons
1	0	–	0
2	1	0	1
3	2	1	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$i-1$	$\log(i-1)$	$\log(i-1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$n-1$	$\log(n-1)$	$\log(n-1)$

$$\begin{aligned}
\sum_{i=2}^{n-1} \log i &= \log(2) + \log(3) + \cdots + \log(n-1) \\
&= \log(2 \cdot 3 \cdot 4 \cdots (n-1)) \\
&= \log((n-1)!) \\
&\leq \log(n!) \\
&\leq \log(n^n) \\
&= n \log n
\end{aligned}$$

That is, the first phase is only $O(n \log n)$, matching the best case running time of Quick Sort and other fast sorting algorithms. The analysis for the "take stuff out" phase is similar, but in reverse. The size of the heap will be diminishing from $n, n-1, \ldots, 1$ and will again contribute another factor of $n \log n$. Since these are independent phases, the entire algorithm is an efficient $O(n \log n)$ sorting algorithm. Simple, efficient, elegant and beautiful.

6.7 Exercises

Exercise 6.1. Develop an algorithm to determine if a given tree T is a binary search tree or not.

Exercise 6.2. Develop an algorithm to determine if a given tree T is a max heap or not.

Exercise 6.3. Adapt and implement the Stack-Based Preorder Traversal (Algorithm 18) for the Java `BinaryTree<T>` class.

Exercise 6.4. Adapt and implement the Stack-Based Inorder Traversal (Algorithm 21) for the Java `BinaryTree<T>` class.

Exercise 6.5. Adapt and implement the Stack-Based Postorder Traversal (Algorithm 22) for the Java `BinaryTree<T>` class.

Exercise 6.6. Adapt and implement the Tree Walk Algorithm (Algorithm 24) for the Java `BinaryTree<T>` class.

Exercise 6.7. Let T be a binary tree of depth d. What is the maximum number of leaves that T could have with respect to d?

Exercise 6.8. Develop an algorithm that inserts a new node into a binary tree at the shallowest available spot.

Exercise 6.9. Develop an algorithm that, given a binary tree T, counts the total number of nodes in the tree (assume it is not kept track of as part of the data structure).

Exercise 6.10. Develop an algorithm that, given a binary tree T, counts the number of leaves in it.

Exercise 6.11. Develop an algorithm that given a node u in a binary tree T determines its depth.

Exercise 6.12. Develop an algorithm that given a binary tree T, computes its depth.

Exercise 6.13. Given an example of two trees with three nodes each that have a different structure, but the same preorder traversal. Do the same for inorder and postorder traversals.

Exercise 6.14. Write an algorithm that, given two binary trees, T_1, T_2, determines if $T_1 = T_2$. Two trees are equal if they have the same node structure *and* the same keys in each node.

Exercise 6.15. Let T be a binary tree with n nodes. What is the maximum number of leaves that T could have with respect to n? Provide an example for $n = 15$

Exercise 6.16. Develop an algorithm (write pseudocode) to insert a given key k into a binary search tree T.

Exercise 6.17. Develop an algorithm (write pseudocode) fix a heap after its root element has been removed.

Exercise 6.18. Rewrite Algorithm 28 (heapify) to work on an array instead of a tree structure.

Exercise 6.19. Design an algorithm to determine if two given binary trees T_1, T_2 are "equivalent" where the condition for equivalency is:

1. The two trees contain the same set of keys.
2. The two trees contain the same set of keys with the same structure.
3. Repeat the first two conditions but when both T_1 and T_2 are binary search trees.

Appendix

1 Author-Book Database SQL

```sql
drop table if exists AuthorBook;
drop table if exists Book;
drop table if exists Author;

create table Author (
  authorId  int primary key auto_increment not null,
  firstName varchar(255) not null,
  lastName  varchar(255) not null,
  key (lastName)
);

create table Book (
  bookId int primary key auto_increment not null,
  #isbn varchar(100) not null,
  title  varchar(255) not null,
  numCopies int not null default 0,
  key (title)
  #unique key (isbn)
);

create table AuthorBook (
  authorBookId int primary key auto_increment not null,
  authorId int not null,
  bookId int not null,
  foreign key (authorId) references Author(authorId),
  foreign key (bookId) references Book(bookId),
  unique (authorId,bookId)
);

insert into Author (authorId, firstName, lastName) values
  (1, "Norman", "Mailer"),
  (2, "Douglas", "Adams"),
  (3, "Octavia", "Butler"),
  (4, "Cory", "Doctorow");

insert into Book (bookId, title, numCopies) values
  (1, "Naked and the Dead", 10),
  (2, "Dirk Gently's Holisitc Detective Agency", 4),
  (3, "The Hitchhiker's Guide to the Galaxy", 2),
  (4, "The Long Dark Tea-Time of the Soul", 1),
  (5, "Barbary Shore", 3),
  (6, "Kindred", 5);

insert into AuthorBook (authorBookId, authorId, bookId) values
  (1, 1, 1),
  (2, 2, 2),
  (3, 2, 3),
  (4, 2, 4),
  (5, 1, 5),
  (6, 3, 6);
```

Glossary

algorithm a process or method that consists of a specified step-by-step set of operations.

corner case a scenario that occurs outside of typical operating parameters or situations; an exceptional case or situation that may need to be dealt with in a unique or different manner. 66

data anomaly Redundant or inconsistent data in a database that violates the intended integrity of the data.. 5

dependency inversion The decoupling of software modules by defining an interface between them. Instead of one module depending directly on the other, both end up depending on the interface. One module *implements* the interface and the other (called a *client*) *uses* or consumes the interface. This allows client code to not have to depend on a particular implementation. Different implementations can be swapped out and changed with minimal code changes in the client code.. 53

duck typing In dynamically typed languages an object may not have a declared type and so to determine its type you rely on what method(s) and/or member variable(s) it has; "if it walks like a duck and talks like a duck" then it is a duck. That is, if it has the methods and variables that you expect of a particular type, then for all intents and purposes it *is* that type..

flat file A manner in which data is stored typically in a single file where the data model is "flattened" into a single table with many columns and each row representing a record.. 5

idiom in the context of software an idiom is a common code or design pattern. 62

interface segregation The principle that no client should be forced to depend on methods or functionality it does not use. This principle essentially states that interfaces should be very minimal in their design and that many small interfaces should be combined to create more complex behavior..

iterator a pattern that allows a user to more easily and conveniently iterate over the elements in a collection. 64

leaky abstraction a design that exposes details and limitations of an implementation that should otherwise be hidden through encapsulation.. 73

Liskov substitution principle The principle that if T is a subtype of S then any instance of S can be replaced with any subtype T without breaking a program..

mixin A mixin is a class that contains methods that other classes may use without a direct inheritance relationship. The functionality is "mixed in" to the class..

normalization the process of restructuring data and tables in a database, separating related data into their own tables in order to reduce redundancy and minimize the potential for data anomalies.. 47

open/closed principle The principle that "software entities should be open for extension, but closed for modification." This generally refers to inheritance in OOP: that a class's functionality can be extended through a subclass, but that the class itself should not be modifiable (so that other classes that depend on it don't have the rug pulled out from under them)..

parameterized polymorphism a means in which you can make a piece of code (a method, class, or variable) generic so that its type can vary depending on the context in which the code is used. 63

polymorphism A mechanism by which a piece of code (class, method, variable) can be written generically so that it can "take on many forms" or used on different types in different places of a program..

query A request for data or an operation on data in a database. 7

queue a collection data structure that holds elements in a first-in first-out manner.

random access a mechanism by which elements in an array can be accessed by simply computing a memory offset of each element relative to the beginning of the array. 68, 93

serialization translation of data into alternative formats, usually to plaintext data interchange formats such as JSON or XML. 6

single responsibility principle a general guideline that every module, class, function, etc. in code should have only a single responsibility or represent a single entity. More complex code can be written by composing these together..

stack a collection data structure that holds elements in a last-in first-out manner. 77

syntactic sugar syntax in a language or program that is not absolutely necessary (that is, the same thing can be achieved using other syntax), but may be shorter, more convenient, or easier to read/write. In general, such syntax makes the language "sweeter" for the humans reading and writing it. 64

transaction The basic unit of work in a database that is treated as an atomic or an all-or-nothing event. A transaction may consist of one or more queries.. 7

tuple An ordered sequence of elements. Typically the notation $(x_1, x_2, \ldots, x_n)$ is used. Tuples correspond to rows or records in a database..

Acronyms

ACID Atomicity, Concurrency, Isolation, Durability. 7, 53

ADT Abstract Data Type. 57

API Application Programmer Interface.

AST Abstract Syntax Tree. 81

BFS Breadth First Search. 159

BLOB Binary Large Object. 13

BST Binary Search Tree. 161, 162

CRUD Create-Retrieve-Update-Destroy. 26, 37

CSV Comma-Separated Value. 5

DAG Directed Acyclic Graph.

DBA Database Administrator. 52

DDL Document Description Language. 9

DFS Depth First Search. 81, 147

DIP Dependency Inversion Principle.

DRY Don't Repeat Yourself. 60

EDI Electronic Data Interchange. 6

ER Entity Relation. 20

FIFO First-In First-Out. 82, 83, 159

FK Foreign Key. 19

FOSS Free and Open Source Software. 9

GRASP General Responsibility Assignment Software Patterns. 3

ISP Interface Segregation Principle.

JDBC Java Database Connectivity API. 53

JPA Java Persistence API. 53

JSON JavaScript Object Notation. 6

JVM Java Virtual Machine. 63

KVP Key Value Pair. 53

LIFO Last-In First-Out. 77, 148

LSP Liskov Substitution Principle.

OCP Open-Closed Principle.

ORM Object-Relational Mapping. 53

PK Primary Key. 16

RDMS Relational Database Management System. 7

RTM Read The Manual. 13

SOLID SOLID Principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.

SQL Structured Query Language. 9, 26

SRP Single Responsibility Principle.

XML Extensible Markup Language. 6

Index

aggregates, 37
algorithm analysis, 105
    recursive algorithms, 130
aliases, 31
amortized analysis, 129
analysis of recursive algorithms, 130
array-based lists, 58
asymptotic analysis, 108
average case analysis, 128

balanced binary search trees, 167
Big-O, 109
Big-Omega, 111
Big-Theta, 112
binary search, 132
binary search tree, 161
    definition, 161
binary search trees
    balanced binary search trees, 167
binary tree, 142
binomial heaps, 179
breadth first search, 159

cartesian product, 40
composite key, 24
consumer producer pattern, 84
cross join, 44

database, 5
depth first search, 147
deque, 87
`distinct` clause, 33

elementary operation, 107
entity-relation diagram, 20

fibonacci heaps, 180
first normal form, 48
foreign key, 18
full outer join, 47

Gauss's formula, 95
`group by` clause, 38

heap sort, 180
heaps, 169
    binomial heaps, 179
    definition, 169
    fibonacci heaps, 180
    heap sort, 180

`in` operator, 34
inner join, *see* join
inorder traversal, 150

join
    cross join, 44
    full outer join, 47
    left join, 46
`join` statement, 40
join table, 23

key
    composite, 24
    foreign, 18
    natural, 17
    primary, 16
    surrogate, 17
    unique, 26

left join, 46
`like` clause, 33
linked lists, 65
lists, 57

array lists, 58
linked lists, 65

many-to-many, 22
Master Theorem, 131
merge sort, 133

natural key, 17
nested query, 29
normal form
first normal form, 48
second normal form, 48
third normal form, 49
normalization, 19, 47
nullability, 13

object oriented programming, 3
one-to-many, 19
`order by` clause, 35

postorder traversal, 153
preorder traversal, 148
primary key, 16
priority queue, 87
projections, 38
pseudocode, 102

queue
priority queue, 87
queues, 82

relation, 41
relational database, 5

second normal form, 48
`select` statement, 30
SQL, *see* Structured Query Language
stacks, 77
peek, 80
Structured Query Language, 9, 26
surrogate key, 17

tables, 9
altering, 15
creation, 9
data types, 11
dropping, 15
third normal form, 49
tree, 137, 138
binary, 142
binary search tree, 161
definition, 139
forest, 139
inorder traversal, 150
postorder traversal, 153
preorder traversal, 148
traversal, 147

union, 47
union all, 47
unique key, 26

varchar, 12

`where` clause, 32

Bibliography

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math*, 2:781–793, 2002.

[2] P. Bachmann. *Die analytische Zahlentheorie.* Number v. 2 in Die analytische Zahlentheorie. Teubner, 1894.

[3] Arthur Cayley. On the theory of the analytical forms called trees. *Philosophical Magazine Series 4*, 13(85):172–176, 1857.

[4] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[6] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741.

[7] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, April 1976.

[8] Eric S. Raymond. *The Cathedral and the Bazaar.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.