ITSC 2214 - Data Structures and Algorithms - Summer 2023

2023-05-22

Table of contents

Preface

This website contains a set of readings for ITSC 2214 - Data Structures and Algorithms.

Part I

On Computers and Computing

1 Background on Computer Science and Data Structures

Let's break down the term "Computer Science" into its two parts: **computer** and **science**. What do these words mean?

The word **science** means "knowledge" in Latin. Science is a way of finding out things about the world by asking questions, doing experiments, and looking for evidence.

The word **computer** means "to calculate" in Latin. A computer is a machine that can do math problems and store information very fast. The word **computing** means using computers or other machines to solve problems or do tasks.

But did you know that computers were not always machines? Before electronic computers were invented, there were people who did math problems by hand for scientists and engineers. They were called **human computers**. They followed fixed rules and had no authority to deviate from them in any detail. They worked in teams and checked each other's results for accuracy.

1.1 Computing Capabilities

Computers, in the broadest sense, are devices that can perform calculations or manipulate information. Throughout history, humans have invented and used various types of computers, each with increasing capabilities and complexity. Here are some examples of how computing capabilities have evolved over time:

- **Tally stick** - One of the earliest forms of computers, dating back to prehistoric times. A tally stick is a piece of wood or bone with notches carved into it to record numbers or events. The computations it could do were incrementing and retrieving one piece of information. For example, a shepherd could use a tally stick to keep track of his sheep by making a notch for each one.
- **Abacus** - A manual device used for calculations by sliding counters along rods or in grooves. The abacus was invented in ancient times and is still used today in some parts of the world. It could store one set of numbers, add, subtract, multiply, divide, and perform other arithmetic operations to the stored information which can later be retrieved. For example, a merchant could use an abacus to keep track of his transactions and profits.

NASA Dryden Flight Research Center Photo Collection
http://www.dfrc.nasa.gov/gallery/photo/index.html
NASA Photo: E49-54 Date: 1949

NACA High Speed Flight Station "Computer Room"

Figure 1.1: For example, in 1959, NASA had a team of women who worked as human computers to help launch rockets into space. They used pencils, paper, and calculators to do complex calculations that machines could not do at that time.

- **Astrolabe** - A sophisticated instrument used for astronomy and navigation by measuring the positions and movements of celestial bodies. The astrolabe was developed in ancient Greece and reached its peak in the Islamic Golden Age. It could perform complex calculations such as determining the time, latitude, longitude, and direction based on the observation of stars and planets. For example, a sailor could use an astrolabe to find his way across the sea by aligning it with the sun or the pole star.
- **Antikythera mechanism** - A mechanical device that simulated the motions of the sun, moon, and planets according to a geocentric model. The Antikythera mechanism was discovered in a shipwreck near the Greek island of Antikythera in 1901. It is estimated to date back to the 2nd century BC and is considered one of the first analog computers. It could predict astronomical phenomena such as eclipses, phases of the moon, and positions of the zodiac signs. For example, a priest could use the Antikythera mechanism to plan religious ceremonies and festivals based on the celestial calendar.
- **Difference engine** - A mechanical calculator that could compute polynomial functions using the method of finite differences. The difference engine was designed by Charles Babbage in the early 19th century but was never fully completed due to technical and financial difficulties. It could generate accurate tables of values for various mathematical functions such as logarithms, trigonometry, and navigation. For example, a mathematician could use the difference engine to check his calculations and avoid errors.
- **Analytical engine** - A proposed mechanical computer that could perform any calculation given a set of instructions or a program. The analytical engine was also designed by Charles Babbage in the mid-19th century but was never built due to his death and lack of funding. It is considered the first general-purpose computer and the precursor of modern computers. It could store data in memory, process data using arithmetic and logical operations, control the flow of execution using conditional branching and looping, and output data using a printer or a punch card. For example, Ada Lovelace, who wrote the first algorithm for the analytical engine, envisioned that it could compose music based on mathematical rules.

Here is a summary of the computing capabilities of some of these devices -

- **Tally stick**: Store and retrieve one piece of data
- **Abacus**: Store and retrieve one piece of data, and perform basic arithmetical operations on them with another operand.
- **System of Gears**: Store and retrieve one piece of data. Each gear can use stored data and scale it up or down (multiply or divide) by a fixed constant determined by the gear ratio.
- **Difference engine**: Can perform complex, but non-programmable computations. Produced tables of input-output pairs.
- **Analytical engine**: Can perform complex, programmable computations. Can store data in memory, process data using arithmetic and logical operations, control the flow of execution using conditional branching and looping, and output data using a printer or a punch card.

1.2 What is Computer Science then?

The science of computers, or Computer Science, seeks to answer fundamental questions like: What are the essential parts of a computer? What can be computed, and what cannot? What determines the ease and speed of computation? This field provides the foundation for understanding the principles that drive computational systems.

1.3 Data Structures and Algorithms

The field of Data Structures and Algorithms expands upon the principles of Computer Science. It determines the efficiency of computation by answering the question: How easily or quickly can something be computed, and what factors influence these metrics? It outlines the necessary building blocks or tools required for programming, and explores the common patterns and problems in programs, offering known ways to enhance their speed. It is this branch of Computer Science that gives us the skills to design, write and analyze the efficiency of our programs. This understanding is crucial to becoming proficient in the broader field of Computer Science.

In essence, the intertwined journey of computers and computing science is a testament to human ingenuity and the relentless pursuit of understanding and harnessing the principles that underlie our world. As we delve deeper into the concepts of data structures and algorithms, we continue to contribute to this exciting journey.

1.4 Review

We talked a little about what computers are, and what capabilities a device needs to have to be able to compute certain types of problems. The next chapter will be about "general purpose, programmable computers", and how they work.

Here's a fun exercise for you -

- Are analog wristwatches computers? What do they compute?
- Can you design a system of gears that can convert Fahrenheit to Celcius?

2 Modern General-Purpose, Programmable Computers

In this chapter, we will explore the functioning of modern, general-purpose computers. We will approach this topic by drawing parallels with a relatable example from an alternate universe, where two boys, Cory and Jamal, develop a unique system of communication.

In this parallel universe, Cory and Jamal are two teenage neighbors that enjoy playing Scrabble, but there's a hitch. Cory's parents are quite religious and in this universe, there exists a prophecy from their scriptures that a game played by youngsters will one day end the world; and so they disapprove of Cory playing Scrabble.

To get around this, Cory and Jamal come up with a clever plan. Cory happens to have two desk lamps that are visible from Jamal's window. They decide that if Cory's mother isn't home, Cory will switch on the right lamp, and if his father isn't home, he will turn on the left one. If both lamps are on, it essentially signals to Jamal that he can come over to play Scrabble.

Cory also has a sister who is usually at the University, but also at home on some weekends. When she is home, she tends to wake up for late-night snacks around 1 am. And so when Cory's sister is home, they can only play until 1 am. To work around this, they add another signal. If Cory switches on his ceiling fan, it will signal that he can come to play, but only until 1 am.

As we see, each additional piece of information that we need to share requires an additional indicator.

2.1 The Working of a Processor

The way Cory and Jamal communicate parallels how a computer's processor operates. A processor has a set of operations it can perform - like copying, moving, adding, subtracting data, jumping to another instruction, conditional branching, and more.

To tell a computer to perform a certain operation, we need several indicators, just like Cory and Jamal's system. Computers use tiny wires that may or may not have electricity running through them. The presence or absence of electricity indicates what the processor should do.

We refer to the absence as a `0` and the presence of electricity as a `1`. Each wire conveying either a `0` or a `1` is said to convey one `bit` of information.

This is a fixed group of wires going into the processor to convey such information on what needs to be done. This grouping is known as "Instruction." The number of bits in an instruction tells us the 'size" of the instruction.

A processor usually has a fixed instruction size (64 for a 64-bit processor, 32 for a 32-bit processor, etc.).

Sometimes even more information is needed. For example, for operations like jumping to a different instruction, the processor needs to know where to jump to. For operations like addition, two numbers are needed. Sometimes, along with the operation, the processor also needs to be instructed about where to store the result. We call these pieces of data on which we perform instructions as `operands`. We often refer to instructions as operations.

We cannot have an extra set of wires for each time we need to share another operand with the processor, so we share this data in sequence.

Say `0010 1010` is the code for the `add` operation, `0000 0000`` is the number 0, and`0000 0001' is the number 1. If we want an 8-bit processor to add these together, we might share the following data through the wires -

```
...
0000 0001 // third cycle - second operand
0000 0000 // second cycle - first operand
0010 1010 // first cycle - add operation
```

There is a "clock" mechanism within the processor to signal the start and end of "cycles". Generally the processor "pipelines" its operation such that it can complete one instruction in each cycle. However, many complex instructions like division or square roots take >20 cycles to complete.

A computer processor can be compared to a complex Rube Goldberg machine, with wires that take in the presence or absence of electricity similar to how many Rube Goldberg machines' mechanism is kicked off by a rolling ball or marbles. The "marbles" of electricity roll through the processor, along each wire, triggering complex chain reactions or side effects that are designed to elicit the intended operation. Consider that a processor, as small as it is, frequently contains billions of transistors - use this information to imagine how complex these "Rube Goldberg machines" are.

2.2 Memory

Let's compare the computer's memory to a vast grid filled with tiny cells. These cells can each store some electricity - and again the presence or absence of electricity conveys a `bit` of information. If we continue the previous analogy, it can be compared to a large grid storing balls (data).

The processor has instructions that allow it to select which cells to "read" and transfer those bits to the processor - to be interpreted either as data or instructions. The processor also has instructions that allow it to write to memory.

It is this interaction of a processor being able to "write" instructions and data to memory, and then conditionally "read" and execute them, and then write new instructions or data that makes a processor "programmable".

What we describe in this section is Active memory or RAM. It is the data in RAM that is readily available for the processor to read and write.

2.3 Programs

Programs are essentially detailed sets of instructions that command a computer to execute specific operations. They can be likened to a recipe that a computer follows to achieve a particular task. Programs dictate what steps the computer must take and in what sequence to reach the desired outcome.

The program is stored on disk in a format that is standard for that operating system, and where the first instruction is stored in this format is always known - say, the first instruction in a "main" section. When a program is executed, it is loaded from the computer's storage into RAM, and then the processor reads and executes the first instruction; and so the execution begins.

The complexity of programs can vary greatly, from a simple one that performs basic arithmetic to an intricate operating system like Windows or Linux that manages every aspect of a computer. However, the core characteristic of all programs is the same: they are sequences of instructions that the computer follows.

2.4 The Java Compiler

As instructions are hard to write directly, we make use of programs called compilers that take in "code" in human-readable text format and output a list of instructions. The program is designed in such a way that the produced list of valid instructions always carries out the task described in "code" faithfully.

The Java compiler is such a program for Java code. When a Java program is compiled, the compiler reviews the code for syntax errors and then translates it into bytecode, a type of intermediate language closer to machine language. The Java Virtual Machine (JVM) then interprets this bytecode into machine code that your computer's processor can execute.

Syntax errors or other kinds of errors essentially refer to situations where the compiler doesn't know how to, or cannot produce a valid set of instructions that can carry out what the code is describing.

Compilers like the Java compiler don't just translate code. They also optimize it, making it more efficient so that the resulting program runs faster and consumes less memory. For example, they may look at your `for` loop that is adding up the first `N` natural numbers and decide to replace the loop with the formula for this computation instead.

2.5 Operating Systems

With a sound understanding of the fundamental components of modern computing, it's important to highlight the role of the operating system.

An operating system (OS) is a type of system software that manages computer hardware and software resources and provides various services for computer programs. It acts as a mediator between users and the computer hardware. Users interact with the operating system through user interfaces such as a command-line interface (CLI) or a graphical user interface (GUI).

Operating systems bear the responsibility of managing the computer's resources, including the processor, memory, disk space, and input/output devices. They coordinate tasks, ensuring that the processor's time is used judiciously, and manage the memory, keeping track of which parts are in use and which are available.

In essence, the operating system provides the platform on which all other software runs. It is the environment in which programs, written in languages like Java and then compiled, operate.

This fundamental understanding of modern computing components helps elucidate the intricate operations that are continuously happening within our laptops, desktops, and even our smartphones. These fundamental aspects form the backbone of the digital age we live in.

Part II

Algorithmic Analysis

3 Algorithmic Analysis

In this chapter, we will delve into the exciting world of algorithmic analysis. The objectives of this chapter are to learn to communicate the speed of an algorithm effectively and to appraise the performance of an algorithm from pseudocode or Java code.

3.1 Problems, Algorithms, and Programs

Before understanding algorithmic analysis, it's essential to differentiate between problems, algorithms, and computer programs. These are three distinct concepts that are interrelated.

3.1.1 Problems

A problem in computer science refers to a specific task that needs to be solved. It can be thought of in terms of inputs and matching outputs. For instance, to solve the problem of finding the youngest student in our class, the input would be the names and ages of all students in the class. The output would be the name of the youngest student.

It's helpful to perceive problems as functions in a mathematical sense. In mathematics, a function is a relationship or correspondence between two sets — the input set (domain) and the output set (range).

3.1.2 Algorithms

An algorithm, on the other hand, is a method or a process followed to solve a problem. If we perceive the problem as a function, then an algorithm can be seen as an implementation of this function that transforms an input into the corresponding output.

Since there are typically numerous ways to solve a problem, there could be many different algorithms for the same problem. Having multiple solutions is advantageous because a specific solution might be more efficient than others for certain variations of the problem or specific types of inputs.

For instance, one sorting algorithm might be best suited for sorting a small collection of integers. Another might excel in sorting a large collection of integers, while a third might be ideal for sorting a collection of variable-length strings.

By definition, a sequence of steps can only be called an algorithm if it fulfills the following properties:

- It must be correct.
- It consists of a series of concrete steps.
- There is no ambiguity about the step to be performed next.
- It must comprise a finite number of steps.
- It must terminate.

3.1.3 Programs and Their Building Blocks

Before discussing programs in detail, let's briefly review two essential components that make a program run: the CPU (Central Processing Unit) and memory.

A CPU is the electronic circuitry within a computer that has the ability to execute certain instructions. Its primary function is to fetch, decode, and execute instructions. It has slots to store data, referred to as registers. Communicating with a CPU involves telling it what operation you want to perform and on which data. For instance, you might instruct the CPU to add two numbers stored in registers `A` and `B` and store the result in register `C`.

CPU instructions are binary codes that specify which operation the CPU should perform. Here's an example of what they look like:

```
10010011001100111110000111011111
```

Some bits in the instruction form the `opcode`, the operation code. The opcode is a unique identifier for an operation, like adding integers. Other bits form the `operand`(s), the data on which to operate. The operand can be where the data is stored (the name of a register or an address in memory), or where to store the result of the operation (again, the name of a register or an address in memory).

For human readability, there are notations to represent these binary instructions. Here is an example of a set of instructions in a human-readable form:

```
.global main
main:
addi   sp, sp, -16
sd     t0, 0(sp)
sd     t1, 8(sp)
call   some_function
ld     t0, 0(sp)
ld     t1, 8(sp)
# Use t0 and t1 here as if nothing happened.
```

```
addi   sp, sp, 16
```

Programs are structured into sections. They include code sections, which contain a list of instructions, and data sections, which hold binary data such as text, images, or numbers that the program needs to use. Typically, there is a designated "main" section that contains the instructions to be executed first.

When you initiate an executable (with the exception of Mac "applications"), the binary data ("bits") are read from the hard disk and transferred to the main memory (RAM). The execution of the program begins when the first instruction from the "main" section is transferred to the CPU.

In this context, a computer program's code can be seen as an instance, or concrete representation, of an algorithm. Although the terms "algorithm" and "program" are distinct, they are often used interchangeably for simplification.

3.2 Comparing the Performance of Programs

When you compile or build a program, its code is converted into a series of instructions and data in memory. However, the execution time of the same program can vary across different machines due to differences in the processor's capabilities.

As each machine can potentially have a different processor, comparing the speed of programs can be a complex task, and is only meaningful when the processor is the same or standardized. Even when the processor is standardized, many factors affect performance -

- Background tasks on one machine can interfere with performance measurements.
- Even if you have the exact same processor, differences in manufacturing mean each can run at a different clock frequency.
- Small differences in ambient temperature affect how high a processor can clock.
- The mounting pressure of a cooler can affect heat transfer and in turn how high a processor can clock.
- Many cooling solutions involve vapor chambers. The orientation of vapor chambers can affect heat transfer and in turn how high a processor can clock.
- Even cosmic radiation can affect processors and memory.

Therefore, we usually prefer to compare algorithms instead. The methods of comparing algorithms will be discussed in the following sections of this chapter.

3.3 Analyzing Algorithms

One of the key components of this course is to provide a framework for predicting the performance of algorithms just by inspecting their structure. Let's dive into the process of analyzing an algorithm's time complexity.

3.3.1 Predicting Execution Time

Consider a simple method that adds two numbers:

```
public int add(int lhs, int rhs)
```

Suppose calling `add(2, 4)` takes 1 second. How long would `add(40, 50)` or `add(343245634, 32432423)` take? As you might expect, all these operations, despite the difference in magnitude of the numbers involved, take approximately the same time. That's because the time complexity of an addition operation does not depend on the values of the numbers but on the number of operations involved, which, in this case, is a single addition.

3.3.2 Impact of Input Size

Now, let's examine a slightly more complex method that sums up a list of numbers:

```
public int sumOfList(List<int> l)
```

Assuming that adding two numbers takes one second, how long would summing a list of 10 numbers take? We can infer that the time taken by `sumOfList` depends on the size of the list we provide. For instance, summing up a list of 10 numbers would take about half the time needed to sum up a list of 20 numbers.

Here's an implementation of `sumOfList`:

```
import java.util.ArrayList;

class Square {
  static int sumOfList(ArrayList<Integer> l) {
    int sum = 0;
    for (int i : l) {
      sum += i;
    }
    return sum;
  }
```

```
}
```

The key insight is that the execution time of this method depends on the number of elements in the list - which is the size of the input. The time complexity is directly proportional to the number of times the addition statement is executed, which is equal to the size of the list.

3.3.3 Iterations and Input Size

Let's take it a step further. If you're summing a list of N items, each of which is another list of M items, the operation would take N * M addition statements. Here, the time complexity is a function of both N and M.

3.3.4 Gauging Relative Execution Time

The crux of analyzing an algorithm's performance lies in understanding how many times statements in the program run as a function of the size of the input. This approach enables us to estimate how long two invocations of the same method will take relative to each other, given the size of the input for each.

Understanding this concept will allow you to better predict the performance of algorithms, which is a crucial skill in efficient programming and system design.

3.4 Algorithm Complexities

In order to evaluate an algorithm's efficiency, we analyze its time complexity and space complexity, both of which describe how the algorithm's performance scales with the size of the input.

3.4.1 Time Complexity

Time complexity measures how the execution time of an algorithm increases with the size of the input. For instance, counting how many times the number 5 appears in a list requires checking each number in the list. Hence, the time complexity is directly proportional to the size of the list.

3.4.2 Space Complexity

Space complexity quantifies the amount of memory an algorithm requires relative to the size of the input. Using the previous example, we would need enough space to store the list and an additional space to store the counter. Thus, the space complexity is proportional to the size of the list, or more precisely, $N + 1$.

3.4.3 Conditional Statements and Complexity

In cases where conditional statements are present, the number of executed statements depends on which branch the program takes. One branch may contain more statements than the other. To handle such scenarios, we introduce the concept of Big-O, Big-Ω, and Big-Θ notations.

3.4.3.1 Big-O Notation

The Big-O notation describes the worst-case time complexity of an algorithm, essentially providing an upper bound on the time taken. This notation considers the scenario where the program consistently takes the path with the most statements. While Big-O is commonly used for time complexity, it can also describe space complexity.

The notation comprises two parts: the function itself and the variable representing the input size. Generally, 'n' is used to represent the input size, and constants and coefficients are typically ignored. For instance, the following functions are all $O(n)$:

- $n + 1$
- $2n$
- $103n + 124$

  Only the highest degree of 'n' is considered when determining Big-O notation. Therefore, functions such as n^2, $n/2$, or $\sqrt{n}$ are not considered O(n).

3.4.3.2 Big-Ω Notation

The Big-Ω notation represents the best-case complexity of an algorithm. It follows the same format as Big-O notation but focuses on the scenario where the program consistently takes the path with the fewest statements.

3.4.3.3 Big-Θ Notation

The Big-Θ notation is used to denote the average-case complexity of an algorithm. It again follows the same structure as Big-O, but it considers both the best and worst-case scenarios to provide an average estimate of the algorithm's performance.

Remember, these notations and complexities are pivotal in estimating the performance of an algorithm based on the size or other properties of the input, correlating to the number of steps taken by the algorithm. This understanding is crucial when designing efficient and effective solutions in computer science.

3.5 Common Big-O Complexities

Big-O notation is a way of expressing the worst-case time complexity of an algorithm. It describes how the running time of an algorithm changes as the size of its input grows. The most common Big-O complexities are:

3.5.1 Constant Complexity and Growth

Constant complexity, often represented as $O(1)$, occurs when the running time of an algorithm or the amount of work needed does not change with the size of the input (N). This means the algorithm takes a fixed amount of time, regardless of how many elements it is processing.

For example, accessing an element in an array by its index is an operation of constant complexity. This is because it takes roughly the same amount of time, regardless of the size of the array:

- $f(n) = 1$ for all n

In this case, no matter how large or small our input size is, the amount of work we have to do remains the same. This is the most efficient complexity an algorithm can have.

3.5.2 Linear Complexity and Growth

Linear complexity, often represented as $O(n)$, occurs when the running time of an algorithm or the amount of work needed scales proportionally with the size of the input (N). For every additional element in the input, a fixed amount of work is added.

For example, finding an element in an unsorted list is an operation of linear complexity, as the algorithm might need to look at every element once:

- $f(n) = n$

In this case, if we add one more element to our input size, we add one more unit of work. This is because every additional element requires the same amount of work.

3.5.3 Quadratic Complexity and Growth

Quadratic complexity, often represented as $O(n^2)$, occurs when the running time of an algorithm or the amount of work needed scales with the square of the size of the input (N). For every additional element in the input, the work increases by a factor of n.

For example, a simple nested loop for comparing pairs of elements in a list has quadratic complexity. This is because each element is compared to every other element:

- $f(n) = n^2$

In this case, if we add one more element to our input size, we add n units of work, as we have to compare this new element with every other element already in the list.

3.5.4 Exponential Functions and Growth

An exponential function is one that includes a variable in the exponent. To illustrate, the function 2^n is an exponential function. Here, with each unit increase in the input, the output is multiplied (or scaled up) by a factor of 2. For instance:

- $f(1) = 2^1 = 2$
- $f(2) = 2^2 = 4$
- $f(n + 1) = f(n) * 2$

Exponential growth is characterized by a constant factor scaling up the running time of the algorithm, or the amount of work needed, for each unit increase in the size of the input (often denoted as N). This constant factor is the base of the exponent. This means that for an algorithm with a time complexity of 2^N, adding one more element to the input could potentially double the amount of work required. Examples of algorithms exhibiting exponential growth include certain solutions to problems like the Towers of Hanoi and calculations of the Fibonacci sequence.

3.5.5 Logarithmic Functions and Growth

Logarithmic functions serve as the inverse of exponential functions. For example, $log_2(n)$ is a logarithmic function. Here, the output is decremented by 1 each time the input is divided (or scaled down) by a factor of 2. For instance:

- $f(16) = log_2(16) = 4$

- $f(8) = log_2(8) = 3$
- $f(n/2) = f(n) - 1$

Logarithmic growth is characterized by a decrement of 1 in the running time of the algorithm or the amount of work needed, each time the size of the input (denoted as N) is divided by a constant factor. This constant factor is the base of the logarithm. For an algorithm with a time complexity of $log_2(n)$, if we have an input size of 16, doubling the input size will only increase the amount of work by 1 unit. Similarly, halving the input size will decrease the work by 1 unit.

Algorithms often display complexities such as $N * log_2(N)$, which represents a combination of linear and logarithmic growth. Examples of such algorithms that include logarithmic growth are binary search and certain sorting algorithms.

3.5.6 Examples

Let's look at some examples of algorithms and their Big-O complexities.

```
static int findMin(x, y) {
  if (x < y) {
    return x;
  } else {
    return y;
  }
}
```

This algorithm finds the minimum of two numbers x and y. It does not depend on the input size, since it only performs one comparison and one return statement. Therefore, its worst-case complexity is $O(1)$. Its best case and average case are also $O(1)$ since they are the same as the worst case.

```
static int linearSearch(numbers[], target)
  for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == target) {
      return i;
    }
  }
  return -1;
}
```

This algorithm performs a linear search on an array of numbers to find a target value. It iterates through each element of the array until it finds the target or reaches the end of the array. In the worst case, it has to check every element of the array, which means its worst-case

complexity is $O(n)$, where n is the length of the array. In the best case, it finds the target in the first element, which means its best-case complexity is $O(1)$. In the average case, it finds the target somewhere in the middle of the array, which means its average-case complexity is also $O(n)$.

We will talk about more examples of other common worst-case complexities throughout this course.

3.6 Growth Rate

The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The cost can be measured in terms of time, space, or other resources. The worst-case complexity notation essentially denotes the growth rate of the time complexity with respect to the size of a worst-case input.

The table below summarizes how different Big-O complexities compare in terms of their growth rates.

Complexity	Growth Rate
O(1)	Constant
O($log(n)$)	Logarithmic
O(n)	Linear
O(n^2)	Quadratic
O(2^n)	Exponential

As we can see, constant and logarithmic complexities have very low growth rates, meaning that they are very efficient and scalable algorithms. Linear complexity has a moderate growth rate, meaning that it can handle reasonably large inputs but may become slow for very large inputs. Quadratic and exponential complexities have very high growth rates, meaning that they are very inefficient and unscalable algorithms that can only handle small inputs.

3.7 Summary / Review

In this section, we learned how to analyze the performance of algorithms using Big-O notation. We saw that measuring the actual running time of an algorithm on real hardware is difficult and impractical because it depends on many factors such as the hardware specifications, the programming language, the compiler, the input size, and the input distribution. Therefore, we need a way to simplify and standardize the performance analysis across different hardware and software platforms.

We learned that one way to simplify the performance analysis is to count the number of steps or instructions that an algorithm needs to execute before finishing. We saw that these steps are analogous to the instructions generated by a compiler when it translates our code into machine code. We also learned that different steps may have different costs depending on their complexity, but we can ignore these differences for simplicity and focus on the overall number of steps.

We learned that Big-O notation is a mathematical tool that allows us to express the worst-case complexity of an algorithm. It describes how the number of steps grows as a function of the input size in the worst possible scenario. It gives us an upper bound on the performance of an algorithm, meaning that it tells us how slow an algorithm can get in the worst case. We also learned that Big-O notation ignores constant factors and lower-order terms because they become insignificant as the input size grows.

We learned about some common Big-O complexities and their growth rates, such as constant, logarithmic, linear, quadratic, and exponential. We saw some examples of algorithms and their Big-O complexities, and how to analyze them using simple rules such as adding complexities for sequential steps, multiplying complexities for nested steps, and taking the maximum complexity for conditional steps.

We learned that Big-O notation helps us compare different algorithms and choose the most efficient one for a given problem. It also helps us estimate how well an algorithm can scale to larger inputs and how it can affect the performance of our applications.

Part III

Graphs

4 The Graph Data Structure

4.1 Background and Motivation

Imagine you want to represent the connections between you and your Instagram followers in a data structure. The diagram below (Figure 4.1) shows a simple representation of your followers as numbered vertices and the edges represent the connections between you and your followers.

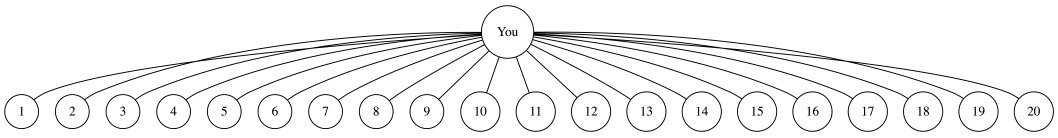


Figure 4.1: A representation of your Instagram followers.

At first glance, it might appear to be a tree structure, but that is not the case. Your followers can follow other people, who in turn can have their followers. This creates a recursive relationship that cannot be represented using a tree data structure (see Figure 4.2).

To accurately capture this complex relationship, we need to use a graph data structure. Graphs consist of a set of vertices (or nodes) and a set of edges that connect them. In the context of Instagram followers, the vertices represent the users, and the edges represent the connections between them.

Using a graph data structure allows us to represent the recursive nature of the relationship between you and your followers, enabling us to model and analyze the complex network of connections in a more accurate way.

Sure! I will translate the given block according to the specifications you provided. Here's the updated version:

Furthermore, the relationship between you and your followers is even more complex. For instance, you can follow someone who does not follow you back, creating a directed relationship where the edges have a direction. In this case, the edges represent the connections from you to your followers, but not the other way around. Such relationships are often modeled using directed graphs, which are a common use case for graphs. Figure 4.3 visualizes this directed relationship.

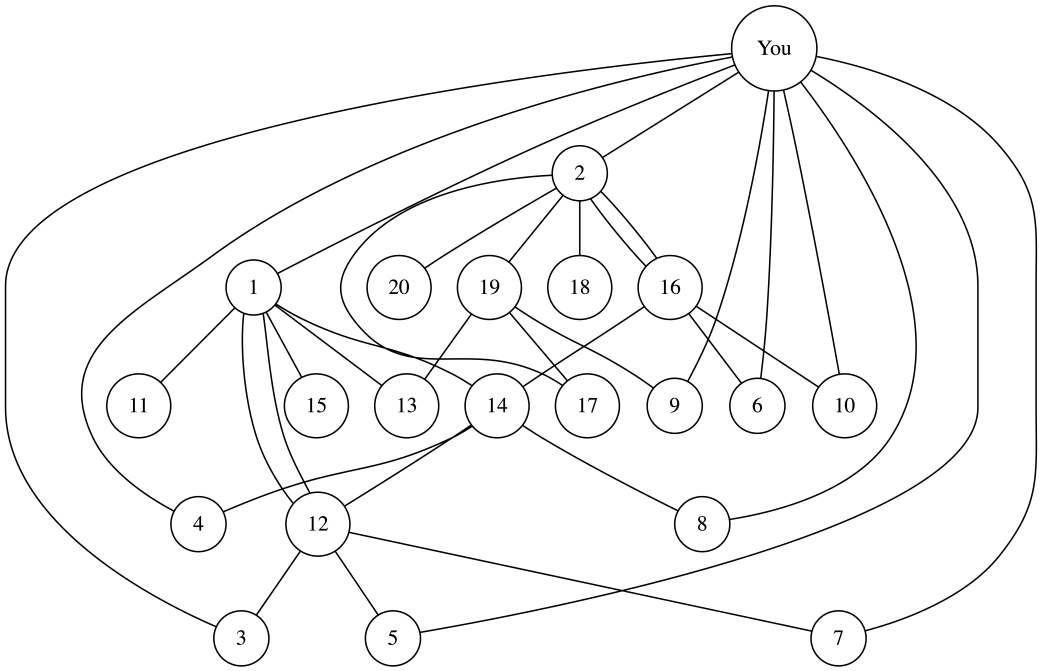


Figure 4.2: A representation of your instagram followers where they're allowed to follow other people too.

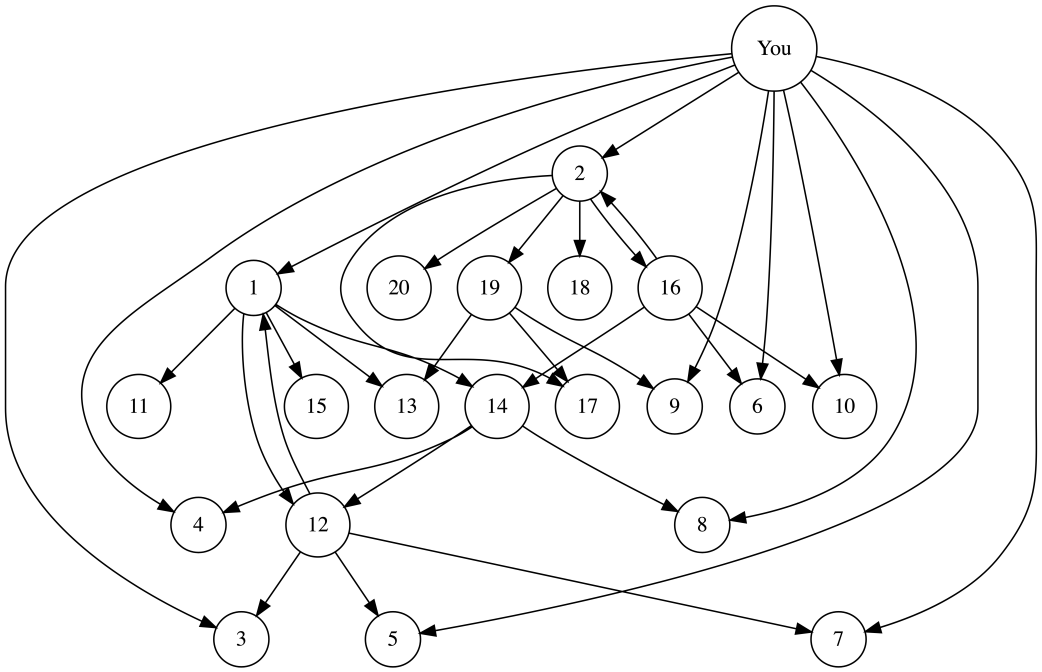


Figure 4.3: A directed representation of your Instagram followers. Here, an arrow going from vertex A to vertex B indicates that A follows B, but B does not necessarily follow A.

4.2 Introduction

A **graph** is a non-linear data structure that consists of a set of vertices (also called nodes) and a set of edges (or connections) that connect these vertices. In this data structure, the arrangement of vertices and edges allows for a more flexible and complex representation of relationships between data elements compared to linear data structures like arrays, lists, or queues.

The concept of **adjacency** refers to the relationship between two vertices in a graph. If there is an edge connecting two vertices, they are said to be adjacent. **Incidence** is the relationship between a vertex and an edge. A vertex is said to be incident to an edge if it is one of the two vertices connected by that edge.

Graphs have numerous real-life applications, and some examples include:

- Social networks, where vertices represent people and edges represent friendships or connections
- Transportation networks, where vertices represent locations and edges represent roads or routes
- Coronavirus transmission networks, where vertices represent individuals and edges represent transmission paths

4.3 Graph Terminology

Before diving into the implementation of graph data structures, let's discuss some basic terms and properties of graphs.

4.3.1 Basic Terms and Properties

- A **graph** is a data structure for representing connections among items and consists of **vertices** connected by **edges**.
- A **vertex** (or node) represents an item in a graph.
- An **edge** represents a connection between two vertices in a graph.
- Two vertices are **adjacent** if connected by an **edge**.
- **Directed vs Undirected**: In an **undirected graph**, the edges have no specific direction, meaning that if there is an edge between vertices A and B, the connection is mutual. In a **directed graph** (also called a digraph), the edges have a direction, indicating an asymmetrical relationship between vertices. (See Figure 4.4 and Figure 4.5 for examples.)

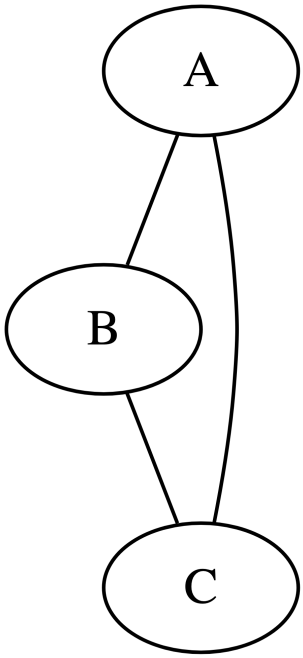


Figure 4.4: Example of an undirected graph.

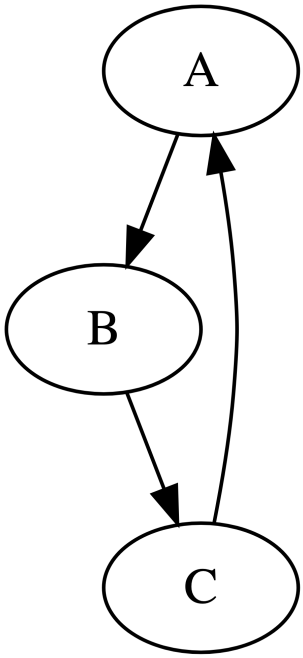


Figure 4.5: Example of a directed graph.

- **Weighted vs Unweighted**: In an **unweighted graph**, all edges have equal importance, while in a **weighted graph**, each edge is assigned a value (or weight), representing the importance, cost, or distance between the connected vertices. (See Figure 4.6 and Figure 4.7 for examples.)

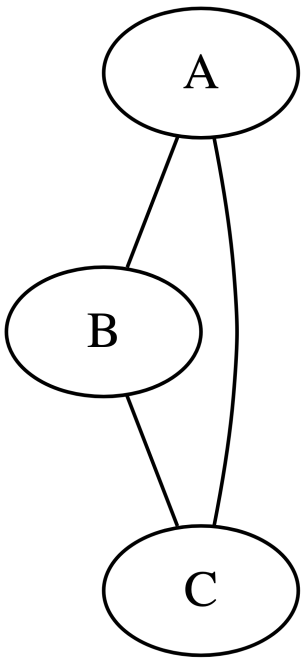


Figure 4.6: Example of an unweighted graph.

- **Simple vs Multigraph**: A **simple graph** has no more than one edge between any pair of vertices and does not contain any self-loops (edges that connect a vertex to itself). A **multigraph** can have multiple edges between the same pair of vertices and may include self-loops. (See Figure 4.8 and Figure 4.9 for examples.)

- **Degree**: The degree of a vertex is the number of edges incident to it. In a directed graph, we can distinguish between in-degree (the number of edges directed towards the vertex) and out-degree (the number of edges directed away from the vertex). See Figure 4.10 for an example.

- **Path**: A path in a graph is a sequence of vertices connected by edges. See Figure 4.11 for an example.

- **Cycle**: A cycle is a closed path, where the first and last vertices in the path are the same, and no vertex is visited more than once. See Figure 4.12 for an example.

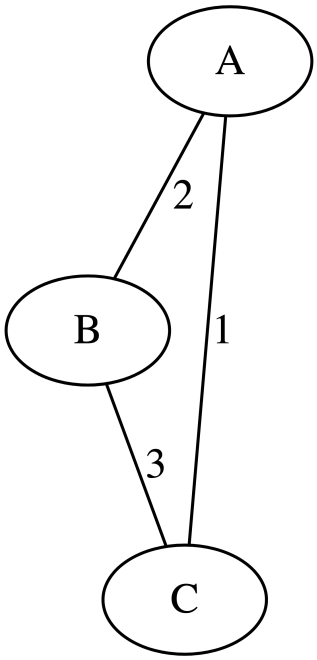


Figure 4.7: Example of a weighted graph.

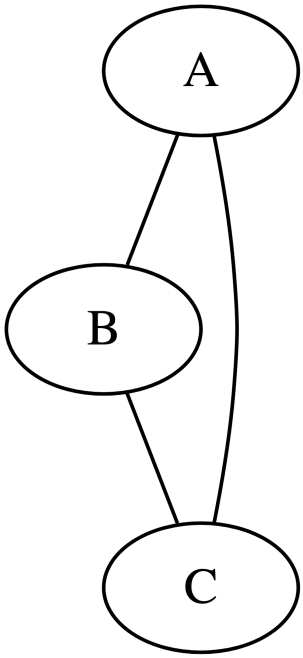


Figure 4.8: Example of a simple graph.

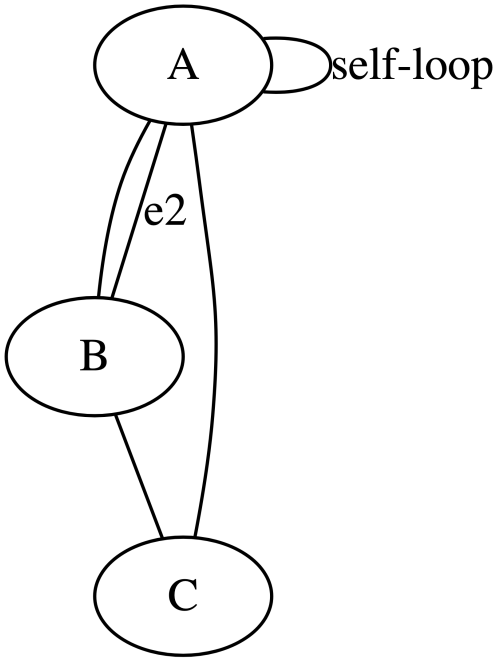


Figure 4.9: Example of a multigraph.

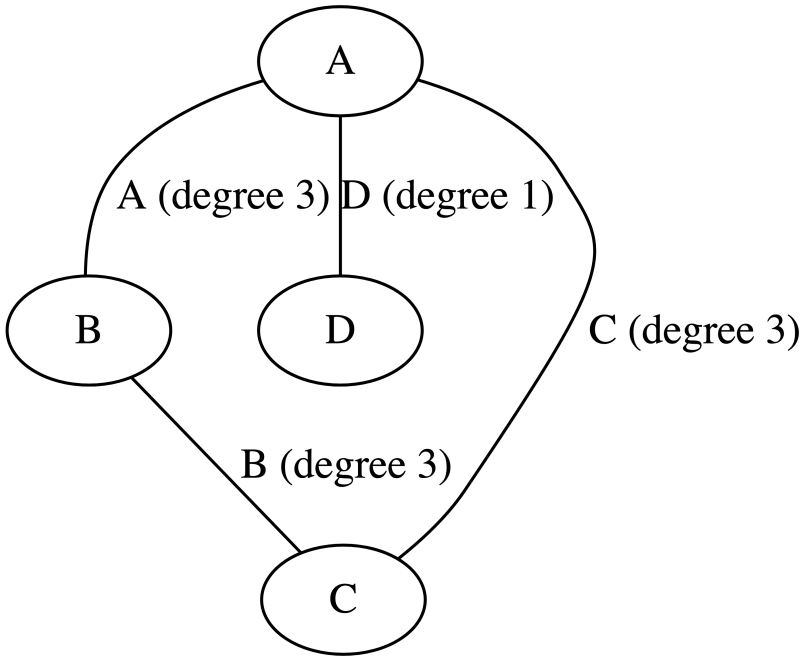


Figure 4.10: Example graph with vertex degrees.

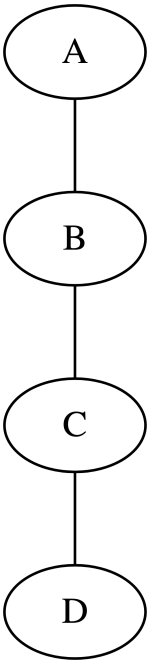


Figure 4.11: Example graph with a path from A to D.

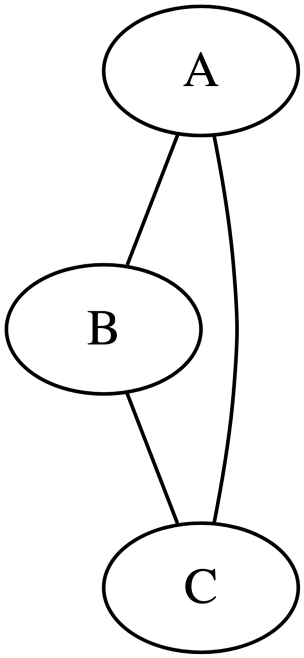


Figure 4.12: Example graph with a cycle. (A-B-C)

- **Connected vs Disconnected**: A graph is connected if there is a path between every pair of vertices. If there is at least one pair of vertices with no path between them, the graph is disconnected. See Figure 4.13 and Figure 4.14 for examples.

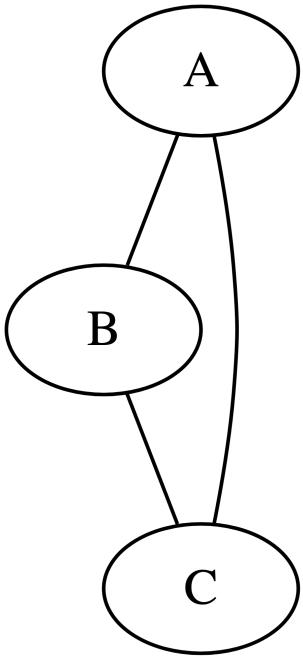


Figure 4.13: Example of a connected graph.

4.3.2 Graph Notation

We can use a notation like $G(V, E)$, where V is the set of vertices and E is the set of edges, to represent a graph.

4.3.3 Special Types of Graphs

- **Complete Graph**: A complete graph is a simple graph in which every pair of vertices is connected by a unique edge. See Figure 4.15 for an example.

- **Bipartite Graph**: A bipartite graph is a graph whose vertices can be divided into two disjoint sets such that all edges connect vertices from one set to the other, with no edges connecting vertices within the same set. See Figure 4.16 for an example.

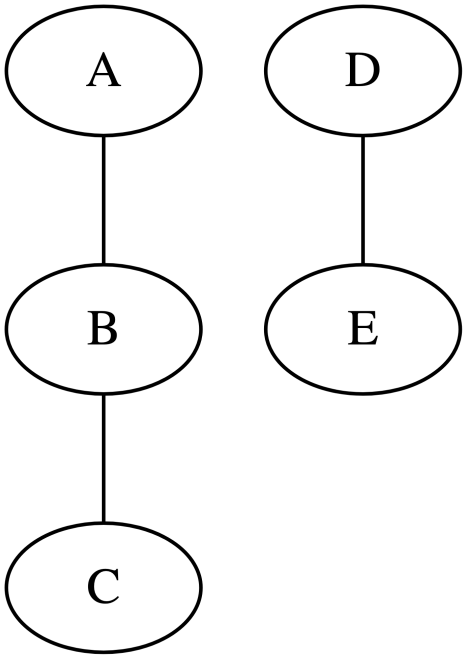


Figure 4.14: Example of a disconnected graph.

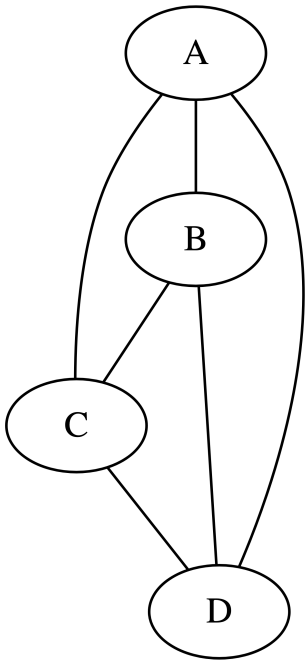


Figure 4.15: Example of a complete graph.

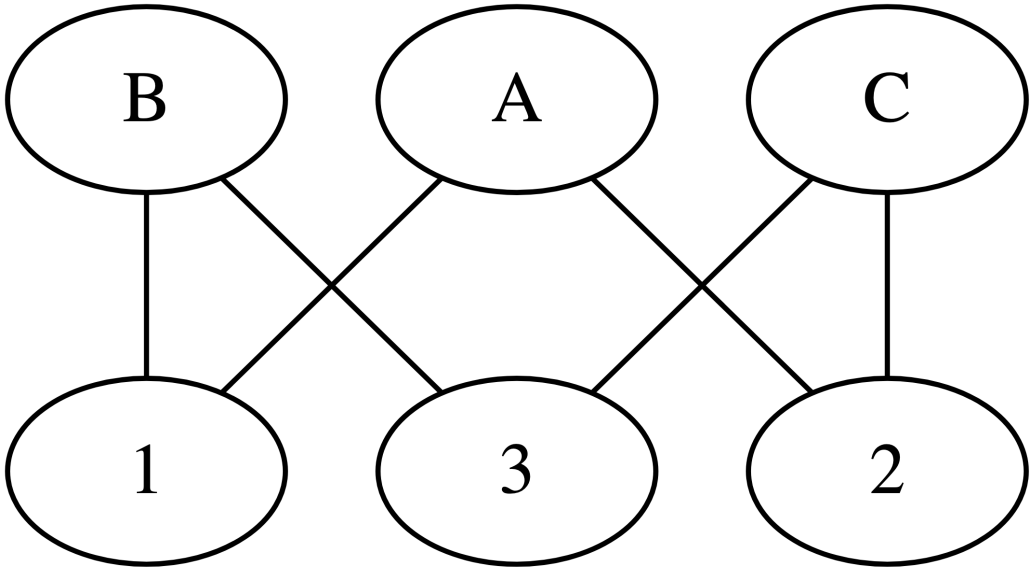


Figure 4.16: Example of a bipartite graph.

- **Tree**: A tree is an undirected graph with no cycles, and all vertices are connected. It has a hierarchical structure, with one vertex acting as the root, and the other vertices connected in a parent-child relationship. See Figure 4.17 for an example.

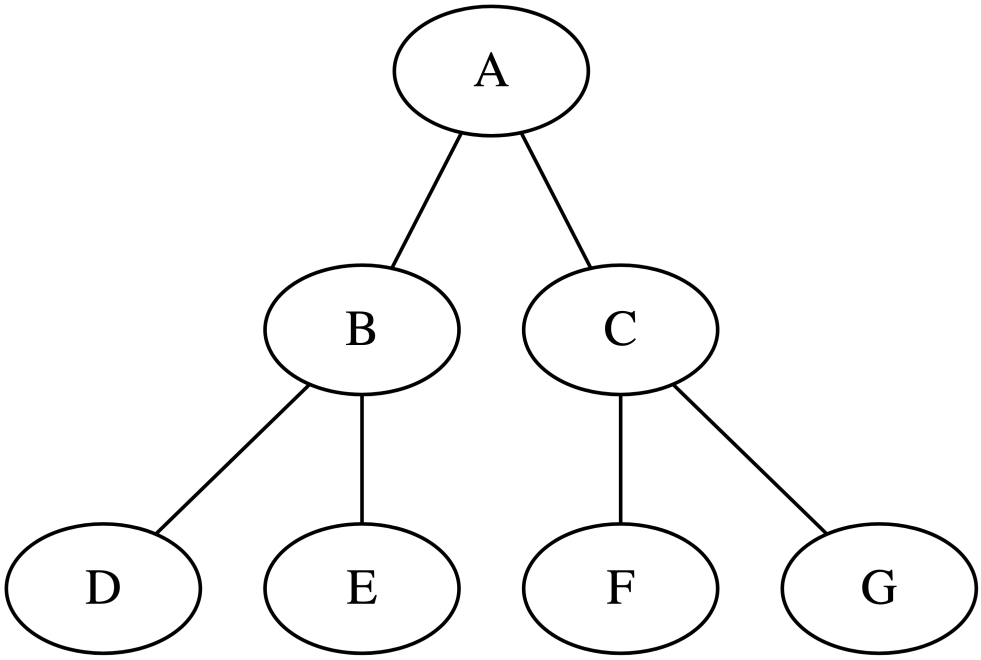


Figure 4.17: Example of a tree.

4.4 Graph Representation

In order to work with graphs in code or store them in memory, we need efficient ways to represent them. There are multiple methods to represent graphs, and the choice of representation depends on factors such as the density of the graph, the operations to be performed, and memory constraints.

In this section, we will discuss two common methods to represent a graph: adjacency list and adjacency matrix.

4.4.1 Adjacency List

An **adjacency list** represents a graph by storing a list of adjacent vertices for each vertex in the graph. This can be implemented using an array of lists or a hash table, where the index

or key corresponds to a vertex, and the value is a list of adjacent vertices.

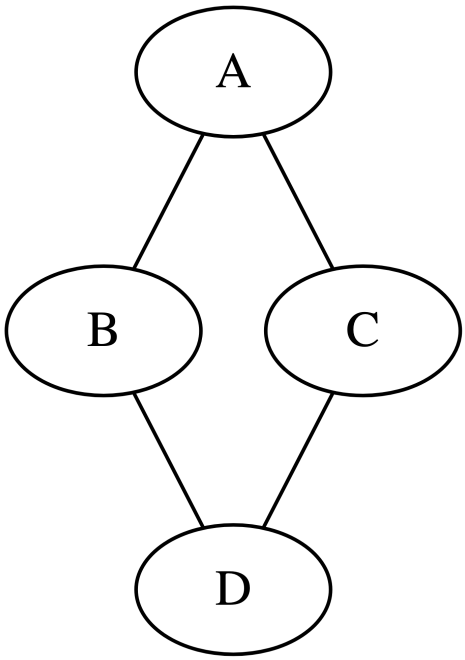


Figure 4.18: See adjacency list for this example Listing 4.1

Adjacency list representation for figure Figure 4.18:

The adjacency list representation is efficient for sparse graphs (graphs with relatively few edges) as it only stores the existing edges, reducing memory usage. This representation also allows for faster traversal of a vertex's neighbors.

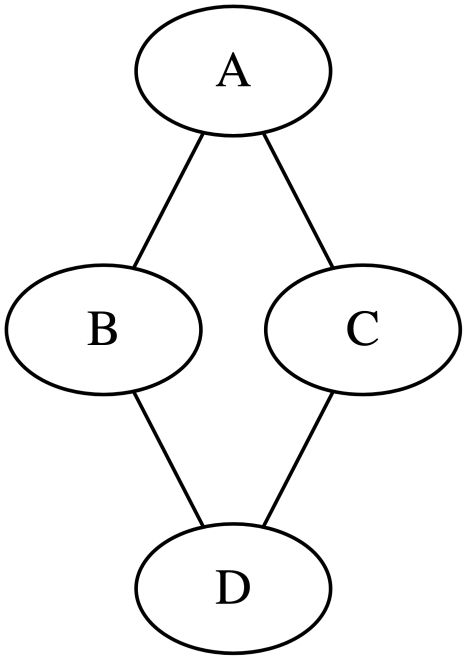
4.4.2 Adjacency Matrix

An **adjacency matrix** is a two-dimensional array (or matrix) where the cell at the i-th row and j-th column represents the edge between vertex i and vertex j. For an undirected graph, the adjacency matrix is symmetric. For a weighted graph, the values in the cells represent the weights of the edges; for an unweighted graph, the cells contain either 1 (edge exists) or 0 (no edge).

Listing 4.1 Adjacency list representation.

```
A: [B, C]
B: [A, D]
C: [A, D]
D: [B, C]

// or as an arraylist of arraylists -
[[B, C], [A, D], [A, D], [B, C]]
// here, the index of the outer arraylist represents the vertex.
// in order for this to work, the order of the vertices must be
// fixed, and stored separately.
```



See adjacency matrix for this example Listing 4.2

Adjacency matrix representation (unweighted):

The adjacency matrix representation is suitable for dense graphs (graphs with many edges) or when checking for the presence of an edge between two vertices needs to be fast. However, this representation can be inefficient in terms of memory usage, especially for large, sparse graphs, as it stores information for all possible edges, even if they do not exist.

Listing 4.2 Adjacency matrix representation.

```
  A B C D
A 0 1 1 0
B 1 0 0 1
C 1 0 0 1

D 0 1 1 0
```

4.4.3 Converting Between Representations

To convert a graph diagram or notation into an adjacency list or an adjacency matrix, follow these steps:

1. Identify the vertices and edges in the graph.
2. For an adjacency list, create an empty list or hash table for each vertex. For each edge, add the adjacent vertices to the corresponding lists.
3. For an adjacency matrix, create a square matrix with dimensions equal to the number of vertices. For each edge, set the corresponding cells in the matrix to 1 (or the edge weight for weighted graphs).

To convert an adjacency list or an adjacency matrix back into a graph diagram or notation, follow these steps:

1. Identify the vertices based on the keys (for an adjacency list) or indices (for an adjacency matrix).
2. For an adjacency list, iterate through the lists and draw an edge for each adjacent vertex.
3. For an adjacency matrix, iterate through the matrix cells and draw an edge for each non-zero value (or the corresponding weight for weighted graphs).

4.5 Graph Traversal

Imagine you want to find the average age of all users on Facebook. With billions of users, it is infeasible to hold the entire graph of the friend network in memory. Ideally, we would want to find out information on each user one at a time, on a per-need basis. To achieve this, we can use graph traversal algorithms, which allow us to visit each user, add up their ages, and then calculate the average. A simple way to do this is to load information on a user, add all their friends to a stack, and then keep popping from the stack and requesting data from Facebook for each friend. When we receive the data, we mark that user as visited to avoid recounting their age if we reach the same user again. We then add friends of each loaded user to our stack and keep repeating until we run out of users in our stack.

This problem illustrates the importance of graph traversal, a fundamental operation in graph theory. Graphs are a powerful and versatile data structure that can model various kinds of relationships and networks, such as social networks, computer networks, transportation networks, web pages, games, and many other domains. Graph traversal allows us to explore and manipulate graphs in various ways, with applications in domains like searching for specific nodes, finding the shortest path between nodes, and analyzing the structure of a graph.

Graph traversal algorithms typically begin with a start node and attempt to visit the remaining nodes from there. They must deal with several troublesome cases, such as unreachable nodes, revisited nodes, and choosing which node to visit next among several options. To handle these cases, graph traversal algorithms use different strategies and data structures to keep track of which nodes have been visited and which nodes are still pending. The most common graph traversal algorithms are breadth-first search (BFS) and depth-first search (DFS), which differ in the order in which they visit the nodes.

In some situations, we may not know the entire graph at once and instead only have access to a node object and its adjacent nodes. As demonstrated in the Facebook example, graph traversal algorithms can be used to solve problems that involve large and dynamic graphs by visiting each user and analyzing their information on a per-need basis.

There are two common methods to traverse a graph:

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)

By understanding and implementing these graph traversal methods, you can efficiently explore and manipulate complex graphs to solve a wide range of problems.

4.5.1 Breadth-First Search (BFS)

Breadth-First Search explores a graph by visiting all the neighbors of the starting vertex before moving on to their neighbors. BFS uses a queue data structure to keep track of the vertices to visit.

Here's a step-by-step example of BFS traversal (for the graph in example Figure 4.19):

BFS traversal starting from vertex A:

1. Visit A and add its neighbors B and C to the queue: [B, C]
2. Visit B and add its unvisited neighbor D to the queue: [C, D]
3. Visit C and add its unvisited neighbor E to the queue: [D, E]
4. Visit D: [E]
5. Visit E: []

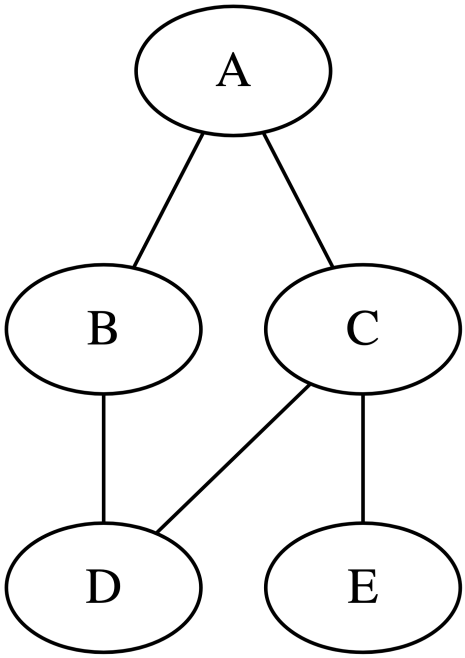


Figure 4.19: Example graph for BFS traversal.

BFS traversal order: `A, B, C, D, E`

BFS pseudocode:

```
BFS(graph, start):
  Initialize an empty queue Q
  Mark start as visited
  Enqueue start into Q

  while Q is not empty:
    vertex = Dequeue(Q)
    Visit vertex

    for each neighbor of vertex:
      if neighbor is not visited:
        Mark neighbor as visited
        Enqueue neighbor into Q
```

4.5.2 Depth-First Search (DFS)

Depth-First Search explores a graph by visiting a vertex and its neighbors as deeply as possible before backtracking. DFS can be implemented using recursion or an explicit stack data structure.

Here's a step-by-step example of DFS traversal (for the graph in example Figure 4.20):

DFS traversal starting from vertex A:

1. Visit A and recurse on its first neighbor B
2. Visit B and recurse on its first neighbor D
3. Visit D and backtrack (no unvisited neighbors)
4. Backtrack to A and recurse on its next neighbor C
5. Visit C and recurse on its first neighbor E
6. Visit E and backtrack (no unvisited neighbors)

DFS traversal order: `A, B, D, C, E`

DFS pseudocode (recursive):

```
DFS(graph, vertex):
  Mark vertex as visited
  Visit vertex

  for each neighbor of vertex:
```

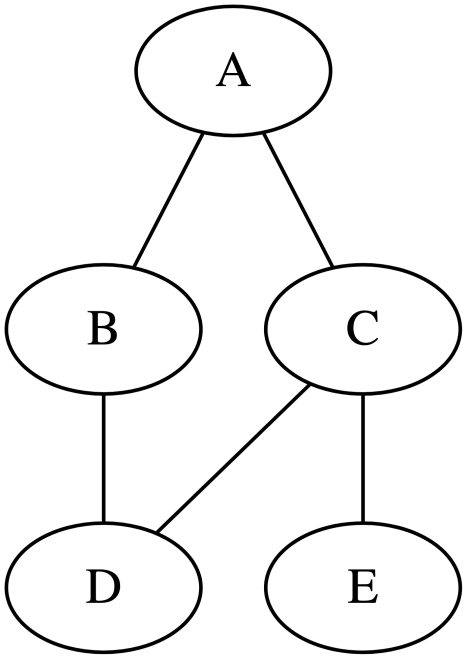


Figure 4.20: Example graph for DFS traversal.

```
    if neighbor is not visited:
      DFS(graph, neighbor)
```

DFS pseudocode (iterative with a stack):

```
DFS(graph, start):
  Initialize an empty stack S
  Mark start as visited
  Push start onto S

  while S is not empty:
    vertex = Pop(S)
    Visit vertex

    for each neighbor of vertex:
      if neighbor is not visited:
        Mark neighbor as visited
        Push neighbor onto S
```

4.5.3 Applications and Variations of BFS and DFS

Both BFS and DFS have numerous applications and can be adapted to solve various graph-related problems:

- **Shortest path**: BFS can be used to find the shortest path between two vertices in an unweighted graph. The algorithm can be modified to keep track of the path length or the actual path itself.
- **Connected components**: Both BFS and DFS can be used to find connected components in an undirected graph. By running the traversal algorithm and marking visited vertices, we can identify the set of vertices reachable from a starting vertex. Repeating this process for all unvisited vertices will find all connected components in the graph.
- **Topological sorting**: DFS can be adapted to perform a topological sort on a directed acyclic graph (DAG). A topological ordering is a linear ordering of the vertices such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. This can be useful in scheduling tasks with dependencies or determining the order of courses in a curriculum.
- **Bipartite graph check**: BFS or DFS can be used to check if a graph is bipartite. The algorithm can be modified to color vertices while traversing the graph. If at any point during the traversal, two adjacent vertices have the same color, the graph is not bipartite.

- **Graph cycle detection**: DFS can be used to detect cycles in a graph. By keeping track of the recursion stack, we can determine if a vertex is visited more than once in the same path, indicating a cycle.

In summary, graph traversal is a fundamental operation in graph theory with various applications. Breadth-First Search (BFS) and Depth-First Search (DFS) are two common techniques to traverse a graph, each with its own advantages and use cases. Understanding these algorithms and their variations can help solve a wide range of graph-related problems.

Part IV

Hashing, HashMaps and HashTables

5 Hashing, Hash Tables, and Hash Maps

5.1 Background and Motivation

5.1.1 Indexing

Indexing refers to the idea of accessing a certain element of an array by referring to it using a specific number, called the index. Using the same index always returns the same element, as long as the array remains unchanged. For example, to access the 12th element of an array `arr`, we use `arr[11]`.

The math for finding the address of an element in the array works out to be:

```
baseAddress + (index * sizeOfElement)
```

Now, keeping that in mind, let's explore the limitations of indexing.

5.1.2 Limitations of Indexing

Suppose we want to access an element in the array using a string as an index, such as `arr["dhruv"]`. What is stopping us?

The problem is that we cannot calculate `baseAddress + ("dhruv" * sizeOfElement)` because the index, in this case, is a string, not a number. The operation is not defined, and therefore, we can't directly use strings as indices in an array.

5.1.3 Mapping Strings to Numbers

Let's consider an array of strings. We can use it to map a number to a string:

```
0 -> "Alice"
1 -> "Bob"
2 -> "Charlie"
```

If we can use an array to map a number to a string, can we also use it to map strings to numbers? Yes, we can!

One way to do this is by searching linearly for a string in the array to find its index. For example, we can find the string "Dhruv" at index 5:

```
0 -> "Alice"
1 -> "Bob"
2 -> "Charlie"
...
5 -> "Dhruv"
```

The index at which we found the string "Dhruv" (in this case, 5) can be used as a key in a different array to find the data related to "Dhruv". However, this method of linear searching can be quite slow for large datasets.

5.1.4 Hashing: A Better Solution

This is where **hashing** comes into play. Hashing allows us to efficiently map strings (or any other non-numeric keys) to numbers. By using a hash function, we can convert a string into a number that represents the index in the array.

A **hash function** takes a key as input and outputs an index in the hash table's array. A good hash function has the following criteria:

- **Uniform distribution**: The hash function should distribute keys evenly across the array to minimize collisions (when multiple keys map to the same index).
- **Minimal collisions**: A good hash function should minimize the chance of collisions.
- **Fast computation**: The hash function should be fast to compute, allowing for quick insertion, deletion, and retrieval of data.
- **Deterministic output**: The hash function should produce the same output for the same input every time it is called.

For example, let's consider a simple hash function that converts the first character of a string into its ASCII code:

```
hash("dhruv") = ASCII('d') = 100
```

The output of the hash function is 100, which we can use as an index in an array to store or retrieve data related to "dhruv". This allows us to use strings (and other non-numeric keys) as indices, achieving our goal of efficient mapping.

5.2 Hash Functions

5.2.1 Introduction

Previously, we managed to map a string "D" to some data, just like an index maps to some data in an array. The resulting data structure that can map any string to any data is called a **hash table**. The function used to map strings to data is called a **hash function**. The concept of mapping "D" to some data can be referred to as **hashing** "D" to an index 3, which is where we found the value corresponding to "D".

Now, we'll learn about a way of mapping any object (called a key) to any other object (called the record, or the value). For instance, your student ID can be a key, and all the data about you on your ID can be stored in a record object.

5.2.2 Hashing

Hashing can be thought of as a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is far better than the $O(logn)$ average cost required to do a binary search on a sorted array of n records, or the $O(logn)$ average cost required to do an operation on a binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system.

A hash system stores records in an array called a **hash table**, which we will call `HT`. Hashing works by performing a computation on a search key `K` in a way that is intended to identify the position in `HT` that contains the record with key `K`. The function that does this calculation is called the **hash function**, and will be denoted by the letter `h`. Since hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, records are not ordered by value. A position in the hash table is also known as a **slot**. The number of slots in hash table `HT` will be denoted by the variable `M` with slots numbered from 0 to M−1. The goal for a hashing system is to arrange things such that, for any key value `K` and some hash function `h`, $i = h(K)$ is a slot in the table such that $0 \leq i < M$, and we have the key of the record stored at `HT[i]` equal to `K`.

Hashing is not good for applications where multiple records with the same key value are permitted. Hashing is not a good method for answering range searches. In other words, we cannot easily find all records (if any) whose key values fall within a certain range. Nor can we easily find the record with the minimum or maximum key value or visit the records in key order. Hashing is most appropriate for answering the question, 'What record, if any, has

key value K?' For applications where all search is done by exact-match queries, hashing is the search method of choice because it is extremely efficient when implemented correctly.

Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots. Since keys have a large range and values have smaller, limited slots for storage – A hash function might sometimes end up hashing two keys to the same slot. We refer to such an event as a **collision**.

To illustrate, consider a classroom full of students. What is the probability that some pair of students shares the same birthday (i.e., the same day of the year, not necessarily the same year)? If there are 23 students, then it is unlikely that more than one student will share the same birthday. There are 365 "slots" or possible days a student can have a birthday on; but only 23 "keys". As the number of students increases, the probability of a "collision" or two students sharing a birthday increases. To be practical, a database organized by hashing must store records in a hash table that is not so large that it wastes space.

We would like to pick a hash function that maps keys to slots in a way that makes each slot in the hash table have equal probability of being filled for the actual set keys being used. Unfortunately, we normally have no control over the distribution of key values for the actual records in a given database or collection. So how well any particular hash function does depends on the actual distribution of the keys used within the allowable key range. In some cases, incoming data are well distributed across their key range. For example, if the input is a set of random numbers selected uniformly from the key range, any hash function that assigns the key range so that each slot in the hash table receives an equal share of the range will likely also distribute the input records uniformly within the table.

However, in many applications the incoming records are highly clustered or otherwise poorly distributed. When input records are not well distributed throughout the key range it can be difficult to devise a hash function that does a good job of distributing the records throughout the table, especially if the input distribution is not known in advance. For example, If the input is a collection of English words, the beginning letter will be poorly distributed. A dictionary of words mapped to their frequency is often used in rudimentary natural language processing algorithms.

In conclusion, anything can be a hash function (i.e., map a value to an index), but not everything can be a good hash function. A function that always returns the index 0 is a hash function that maps everything to 0. It's no good but it's still a hash function. An example of a commonly used hash function is the modulus operator! It is common for `N`-sized hash tables to use the modulus of `N` as a hash function. If `N` is 20, data for 113 will be hashed to index $113\%20 = 13$.

But if we use the modulo operator as a hash function, what do we do when multiple pieces of data map to the same index? $53\%20 = 13$, $73\%20 = 13$, etc. But if you think about it, we can store everything at 13! By using nested data structures… More on this later.

5.2.3 Simple Hash Functions

Let's apply a simple hash function to a set of keys and compute their indices. In this example, we'll use the modulo operation as the hash function. Given a hash table with a size of 5, we can compute the indices for the keys as follows:

```
HashTable size: 5
HashFunction: key % size

Keys: 15, 28, 47, 10, 33

Indices:
15 % 5 = 0
28 % 5 = 3
47 % 5 = 2
10 % 5 = 0
33 % 5 = 3
```

5.2.4 Other Types of Hash Functions

5.2.4.1 Direct Hashing

A direct hash function uses the item's key as the bucket index. For example, if the key is 937, the index is 937. A hash table with a direct hash function is called a direct access table. Given a key, a direct access table search algorithm returns the item at index key if the bucket is not empty, and returns null (indicating item not found) if empty.

Limitations:

A direct access table has the advantage of no collisions: Each key is unique (by definition of a key), and each gets a unique bucket, so no collisions can occur. However, a direct access table has two main limitations:

1. All keys must be non-negative integers, but for some applications, keys may be negative.
2. The hash table's size equals the largest key value plus 1, which may be very large.

Similarly, there are other hash functions each with their own characteristics.

5.2.4.2 Modulo Hash

A modulo hash function computes the index by taking the remainder of the key divided by the table size `M`. This is a simple and effective way to convert a large key range into a smaller index range. The hash function can be defined as:

```
h(K) = K % M
```

5.2.4.3 Mid-Square Hash

A mid-square hash function computes the index by first squaring the key, and then extracting a portion of the squared value as the index. This approach is especially useful when the keys are not uniformly distributed. The hash function can be defined as:

```
h(K) = middle_digits(K^2)
```

5.2.4.4 Mid-Square Hash with Base 2

A mid-square hash function with base 2 is a variation of the mid-square hash function, where the key is first squared, and then the middle bits of the binary representation of the squared value are extracted as the index. This approach is especially useful for binary keys. The hash function can be defined as:

```
h(K) = middle_bits(K^2)
```

5.2.4.5 Multiplicative String Hashing

A multiplicative string hashing function computes the index by treating the characters in the string as numbers and combining them using a multiplication and a constant. This approach can help achieve a good distribution of string keys in the hash table. The hash function can be defined as:

```
h(K) = (c1 * a^(n-1) + c2 * a^(n-2) + ... + cn) % M
```

where `c1, c2, ..., cn` are the character codes of the string, `a` is a constant, `n` is the length of the string, and `M` is the size of the hash table.

Here's the ASCII representation of the resulting hash table:

```
Index | Key
-------------
  0   | 15
  1   | -
  2   | 47
  3   | 28
  4   | -
```

In this example, we can see that the keys 15 and 10, as well as 28 and 33, have collided, as they both map to the same indices (0 and 3, respectively).

5.2.5 Trade-offs Between Different Hash Functions

There are trade-offs between different hash functions in terms of performance and complexity:

- A simple hash function, like the modulo operation, is fast to compute but may not distribute keys uniformly, leading to more collisions and reduced performance.
- More complex hash functions, such as cryptographic hash functions, can provide a better distribution of keys but may be slower to compute.

In practice, the choice of a hash function depends on the specific requirements of the application and the data being stored. The goal is to find a balance between uniform distribution, minimal collisions, fast computation, and deterministic output.

5.3 Hash Collisions

Hash collisions occur when two or more keys map to the same index in the hash table. Due to the pigeonhole principle, hash collisions are inevitable, as there are typically more possible keys than available indices in the array. Collisions negatively impact the efficiency of hashing, as they can lead to longer access times for insertion, deletion, and retrieval of key-value pairs.

There are two primary methods to resolve hash collisions: **chaining** and **open addressing**.

5.4 Chaining

Chaining is a collision resolution technique that uses a linked list or another data structure to store multiple key-value pairs at the same index. When a collision occurs, the new key-value pair is simply added to the data structure at the index.

5.4.1 Insertion, Search, and Deletion

Here's how to perform insertion, search, and deletion operations using chaining:

1. **Insertion**: Calculate the index using the hash function. If the index is empty, create a new data structure (e.g., linked list) and insert the key-value pair. If the index is not empty, add the key-value pair to the existing data structure.
2. **Search**: Calculate the index using the hash function. If the index is empty, the key is not in the hash table. If the index is not empty, search the data structure at the index for the key.
3. **Deletion**: Calculate the index using the hash function. If the index is empty, the key is not in the hash table. If the index is not empty, search the data structure at the index for the key and remove it if found.

5.4.2 Advantages and Disadvantages of Chaining

Chaining has several advantages and disadvantages:

- **Advantages**:
  - Easy implementation: Chaining can be easily implemented using existing data structures like linked lists.
  - Dynamic size: The data structure at each index can grow or shrink as needed, allowing for efficient use of space.
- **Disadvantages**:
  - Extra space: Chaining requires additional space for the data structure at each index, which can increase memory overhead.
  - Variable access time: The access time for key-value pairs depends on the length of the data structure at the index, which can vary.

Chaining is a popular method for resolving hash collisions due to its simplicity and dynamic size. However, it may not be the most efficient option for all use cases, especially when memory overhead and variable access times are critical factors.

5.5 Open Addressing

Open addressing is a collision resolution technique that finds an alternative index for a key-value pair if the original index is occupied. When a collision occurs, the algorithm searches for the next available index using a probing technique. There are three common types of probing techniques: linear probing, quadratic probing, and double hashing.

5.5.1 Probing Techniques

1. **Linear probing**: When a collision occurs, search the hash table linearly (one index at a time) until an empty slot is found.
2. **Quadratic probing**: When a collision occurs, search the hash table quadratically (by increasing the index by the square of the probe number) until an empty slot is found.
3. **Double hashing**: When a collision occurs, use a secondary hash function to compute a new index for the key-value pair, and repeat this process until an empty slot is found.

5.5.2 Insertion, Search, and Deletion

Here's how to perform insertion, search, and deletion operations using open addressing:

1. **Insertion**: Calculate the index using the hash function. If the index is empty, insert the key-value pair. If the index is occupied, use the chosen probing technique to find the next available index and insert the key-value pair there.
2. **Search**: Calculate the index using the hash function. If the index is empty, the key is not in the hash table. If the index is occupied, check if the key matches the stored key. If not, use the chosen probing technique to search for the next index until the key is found or an empty index is encountered.
3. **Deletion**: Calculate the index using the hash function. If the index is empty, the key is not in the hash table. If the index is occupied and the key matches the stored key, remove the key-value pair and mark the index as deleted. Continue searching using the chosen probing technique to handle cases where the removed key-value pair was part of a cluster.

5.5.3 Advantages and Disadvantages of Open Addressing

Open addressing has several advantages and disadvantages:

- **Advantages**:
  - No extra space: Open addressing does not require additional space for data structures at each index, making it more memory-efficient.
  - Fixed size: The hash table has a fixed size, which can be useful when memory is limited.
- **Disadvantages**:
  - Clustering: Probing techniques can cause clusters of key-value pairs to form, leading to increased access times.

– Deletion issues: Deleting key-value pairs can create complications, as it may leave "holes" in clusters that need to be addressed.

Open addressing is an alternative method for resolving hash collisions that can be more memory-efficient than chaining. However, it may not be the best option for all use cases, especially when clustering and deletion issues are critical factors.

5.6 Complexity and Load Factor

When analyzing the complexity of hash functions and hash tables, we need to consider the time taken for searching, inserting, or deleting an element. There are two main steps involved in these operations:

1. Computing the hash function for the given key.
2. Traversing the list of key-value pairs present at the computed index.

5.6.1 Time Complexity of Hash Computation

For the first step, the time taken depends on the key and the hash function. For example, if the key is a string "abcd", then its hash function may depend on the length of the string. But for very large values of `n`, the number of entries into the map, the length of the keys is almost negligible in comparison to `n`, so hash computation can be considered to take place in constant time, i.e., O(1).

5.6.2 Time Complexity of List Traversal

For the second step, traversal of the list of key-value pairs present at that index needs to be done. In the worst case, all the `n` entries are at the same index, resulting in a time complexity of O(n). However, enough research has been done to make hash functions uniformly distribute the keys in the array, so this almost never happens.

5.6.3 Load Factor

On average, if there are `n` entries and `b` is the size of the array, there would be `n/b` entries at each index. This value `n/b` is called the load factor, which represents the load on our map. The load factor is denoted by the symbol :

```
= n/b
```

This load factor needs to be kept low so that the number of entries at one index is less, and the complexity remains almost constant, i.e., O(1).

5.6.4 Balancing Load Factor and Complexity

To maintain the load factor at an acceptable level, the hash table can be resized when the load factor exceeds a certain threshold. This helps to keep the complexity of hash table operations near O(1) by redistributing the keys uniformly across a larger array.

In conclusion, understanding the complexity and load factor of hash functions is crucial for designing efficient hash tables. By carefully choosing a suitable hash function and managing the load factor, it's possible to achieve near-constant time complexity for various hash table operations.

5.7 Rehashing

Rehashing, as the name suggests, means hashing again. When the load factor increases to more than its pre-defined value (the default value of the load factor is 0.75), the complexity increases. To overcome this issue, the size of the array is increased (typically doubled) and all the values are hashed again and stored in the new, larger array. This helps maintain a low load factor and low complexity.

5.7.1 Why?

Rehashing is done because whenever key-value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases, as explained earlier. This might not provide the desired time complexity of O(1). Hence, rehashing must be performed, increasing the size of the `bucketArray` to reduce the load factor and the time complexity.

5.7.2 How?

Rehashing can be done as follows:

1. For each addition of a new entry to the map, check the load factor.
2. If the load factor is greater than its pre-defined value (or the default value of 0.75 if not given), then perform rehashing.
3. To rehash, create a new array of double the previous size and make it the new `bucketArray`.
4. Traverse each element in the old `bucketArray` and call the `insert()` method for each, to insert it into the new larger `bucketArray`.

The following diagram illustrates the rehashing process:

```
Initial bucketArray (size = 4):
+---+---+---+---+
|   | K1|   | K2|
+---+---+---+---+

After inserting a new key K3 (load factor > 0.75):

New bucketArray (size = 8):
+---+---+---+---+---+---+---+---+
|   | K1|   | K2|   |   |   | K3|
+---+---+---+---+---+---+---+---+
```

By rehashing, the hash table maintains its desired time complexity of O(1) even as the number of elements increases. It is important to note that rehashing can be a costly operation, especially if the number of elements in the hash table is large. However, since rehashing is done infrequently and only when the load factor surpasses a certain threshold, the amortized cost of rehashing remains low, allowing the hash table operations to maintain near-constant time complexity.

5.8 Hash Tables vs Hash Maps

Hash tables and **hash maps** differ in their implementation and functionality.

- **Hash tables** use direct hashing, where the key is an integer or can be directly converted to an integer (e.g., a string of digits). The integer is then used to compute the index in the hash table.
- **Hash maps** use indirect hashing, where the key can be any data type. A separate hash function is needed to convert the key into an index in the hash table.

When deciding whether to use a hash table or a hash map, consider the problem domain and the data type of the keys:

- If the keys are integers or can be directly converted to integers, a **hash table** may be a more suitable choice. For example, if you're working with student IDs as keys, a hash table would be a good fit.
- If the keys are of any other data type or cannot be directly converted to integers, a **hash map** would be more appropriate. For example, if you're working with strings, such as usernames or URLs, a hash map would be a better choice.

5.9 HashMaps in Java

A **HashMap** is a collection in Java that implements the Map interface and uses a hash table for storage. It stores key-value pairs, where each key is unique, and the keys are not ordered.

Here's how to use a HashMap in Java:

1. **Import the HashMap class**: To use the HashMap class in your Java code, you'll need to import it from the `java.util` package:

```
import java.util.HashMap;
```

2. **Create a HashMap**: To create a new HashMap, use the following syntax:

```
HashMap<String, Integer> myMap = new HashMap<String, Integer>();
```

3. **Add elements**: To add key-value pairs to the HashMap, use the `put()` method:

```
myMap.put("apple", 3);
myMap.put("banana", 5);
myMap.put("orange", 2);
```

4. **Access elements**: To access the value associated with a key, use the `get()` method:

```
int apples = myMap.get("apple"); // 3
int oranges = myMap.get("orange"); // 2
```

5. **Remove elements**: To remove a key-value pair from the HashMap, use the `remove()` method:

```
myMap.remove("banana");
```

6. **Check if a key exists**: To check if a key is in the HashMap, use the `containsKey()` method:

```
boolean hasApple = myMap.containsKey("apple"); // true
boolean hasGrape = myMap.containsKey("grape"); // false
```

7. **Iterate over keys**: To iterate over the keys in a HashMap, you can use a for-each loop with the `keySet()` method:

```
for (String fruit : myMap.keySet()) {
    System.out.println(fruit + ": " + myMap.get(fruit));
}
```

8. **Iterate over values**: To iterate over the values in a HashMap, you can use a for-each loop with the `values()` method:

```
for (Integer count : myMap.values()) {
    System.out.println(count);
}
```

9. **Iterate over key-value pairs**: To iterate over the key-value pairs in a HashMap, you can use a for-each loop with the `entrySet()` method:

```
for (HashMap.Entry<String, Integer> entry : myMap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

A HashMap can be a useful data structure when you need to store key-value pairs efficiently. It provides constant-time performance for common operations like put, get, and remove, making it an ideal choice for various applications.

5.10 HashTables in Java

A **HashTable** is a collection in Java that implements the Map interface and uses a hash table for storage. It is similar to a HashMap but with some differences, such as being synchronized, which makes it thread-safe. HashTable stores key-value pairs, where each key is unique, and the keys are not ordered.

Here's how to use a HashTable in Java:

1. **Import the HashTable class**: To use the HashTable class in your Java code, you'll need to import it from the `java.util` package:

```
import java.util.Hashtable;
```

2. **Create a HashTable**: To create a new HashTable, use the following syntax:

```
Hashtable<String, Integer> myTable = new Hashtable<String, Integer>();
```

3. **Add elements**: To add key-value pairs to the HashTable, use the `put()` method:

```
myTable.put("apple", 3);
myTable.put("banana", 5);
myTable.put("orange", 2);
```

4. **Access elements**: To access the value associated with a key, use the `get()` method:

```
int apples = myTable.get("apple"); // 3
int oranges = myTable.get("orange"); // 2
```

5. **Remove elements**: To remove a key-value pair from the HashTable, use the `remove()` method:

```
myTable.remove("banana");
```

6. **Check if a key exists**: To check if a key is in the HashTable, use the `containsKey()` method:

```
boolean hasApple = myTable.containsKey("apple"); // true
boolean hasGrape = myTable.containsKey("grape"); // false
```

7. **Iterate over keys**: To iterate over the keys in a HashTable, you can use a for-each loop with the `keySet()` method:

```
for (String fruit : myTable.keySet()) {
    System.out.println(fruit + ": " + myTable.get(fruit));
}
```

8. **Iterate over values**: To iterate over the values in a HashTable, you can use a for-each loop with the `values()` method:

```
for (Integer count : myTable.values()) {
    System.out.println(count);
}
```

9. **Iterate over key-value pairs**: To iterate over the key-value pairs in a HashTable, you can use a for-each loop with the `entrySet()` method:

```
for (Hashtable.Entry<String, Integer> entry : myTable.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

A HashTable can be a useful data structure when you need to store key-value pairs and require thread-safe operations. However, it has some performance overhead due to synchronization, so if thread safety is not a concern, a HashMap is generally a more efficient choice.

5.11 HashSets in Java

A **HashSet** is a collection in Java that implements the Set interface and uses a hash table for storage. It does not store key-value pairs like hash tables or hash maps, but instead stores unique elements. The elements in a HashSet are not ordered, and duplicate values are not allowed.

Here's how to use a HashSet in Java:

1. **Import the HashSet class**: To use the HashSet class in your Java code, you'll need to import it from the `java.util` package:

```
import java.util.HashSet;
```

2. **Create a HashSet**: To create a new HashSet, use the following syntax:

```
HashSet<String> mySet = new HashSet<String>();
```

3. **Add elements**: To add elements to the HashSet, use the `add()` method:

```
mySet.add("apple");
mySet.add("banana");
mySet.add("orange");
```

4. **Remove elements**: To remove elements from the HashSet, use the `remove()` method:

```
mySet.remove("banana");
```

5. **Check if an element exists**: To check if an element is in the HashSet, use the `contains()` method:

```
boolean hasApple = mySet.contains("apple"); // true
boolean hasGrape = mySet.contains("grape"); // false
```

6. **Iterate over elements**: To iterate over the elements in a HashSet, you can use a for-each loop:

```
for (String fruit : mySet) {
    System.out.println(fruit);
}
```

A HashSet can be a useful data structure when you need to store a collection of unique elements without any specific order. It provides constant-time performance for common operations like add, remove, and contains, making it an efficient choice for many applications.

5.12 `hasCode` and `equals` in Java

In Java, the `hashCode` method is part of the `Object` class, which is the superclass of all Java classes. The purpose of the `hashCode` method is to provide a default implementation for generating hash codes, which are integer values that represent the memory address of an object.

5.12.1 The `hashCode` Method

The `hashCode` method has the following signature:

```
public int hashCode()
```

This method returns an integer hash code for the object on which it is called. By default, it returns a hash code that is based on the object's memory address, but this behavior can be overridden in subclasses to provide custom hash code generation.

A well-implemented `hashCode` method should follow these general rules:

1. If two objects are equal according to their `equals()` method, they must have the same hash code.
2. If two objects have the same hash code, they are not necessarily equal according to their `equals()` method.
3. The hash code of an object should not change over time unless the information used in the `equals()` method also changes.

5.12.2 Overriding the `hashCode` Method

When creating custom classes, it is important to override the `hashCode` method if the `equals()` method is also overridden. This ensures that the general contract of the `hashCode` method is maintained, which is essential for the correct functioning of hash-based data structures like `HashSet` and `HashMap`.

Here's an example of a custom `Person` class that overrides both the `equals()` and `hashCode()` methods:

```
public class Person {
    private String name;
    private int age;

    // Constructor, getters, and setters
```

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

In this example, the `equals()` method checks if two `Person` objects have the same name and age. The `hashCode()` method uses the `Objects.hash()` utility method, which generates a hash code based on the name and age fields.

5.12.3 Using `hashCode` with Java Collections

The `hashCode` method plays a crucial role in the performance of Java's hash-based data structures, such as `HashSet`, `HashMap`, and `HashTable`. These data structures rely on the `hashCode` method to efficiently store and retrieve objects based on their hash codes.

When working with these collections, it is important to ensure that the `hashCode` method is correctly implemented for the objects being stored. Failing to do so can lead to poor performance or incorrect behavior.

In summary, the `hashCode` method in Java is a critical part of the `Object` class that provides a default implementation for generating hash codes. When creating custom classes, it is essential to override the `hashCode` method if the `equals()` method is also overridden, ensuring the correct functioning of hash-based data structures like `HashSet` and `HashMap`.

References