ITSC 2214 - Data Structures and Algorithms - Summer 2023

2023-05-22

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 On Computers and Computing

1.1 Introduction to Computer Science

1.2 Topics that will be covered:

- What do these 'Computer Science' words mean?
- What exactly does it mean to compute?
- What's the history of Computing?

1.3 Science

- A particular area of science
- A systematically organized body of knowledge on particular subject
- knowledge of any kind
- The intellectual and practical activity encompassing the systematic study of the structure and behavior of the physical and natural world through observation and experiment

1.4 Computer

- The term "computer", in use from the early 17th century (the first known written reference dates from 1613) meant "one who computes": a person performing mathematical calculations, before electronic computers became commercially available.
- Alan Turing described the "human computer" as someone who is "supposed to be following fixed rules; he has no authority to deviate from them in any detail." Teams of people, from the late nineteenth century onwards, were used to undertake long and often tedious calculations; the work was divided so that this could be done in parallel. The same calculations were frequently performed independently by separate teams to check the correctness of the results.
- Since the end of the 20th century, the term "human computer" has also been applied to individuals with prodigious powers of mental arithmetic, also known as mental calculators.

2 NASA High Speed Flight Station "Computer Room"

Early "computers" at work, summer 1949. In the terminology of that period, computers were employees who performed the arduous task of transcribing raw data from rolls of celluloid film and strips of oscillograph paper and then, using slide rules and electric calculators, reducing into standard engineering units.

3 The History of Computing

- **14th C - Abacus** - an instrument for performing calculations by sliding counters along rods or in grooves.
- **17th C - Slide rule** - a manual device used for calculation that consists in its simple form of a ruler and a movable middle piece which are graduated with similar logarithmic scales.
- **1642 - Pascaline -** a mechanical calculator built by Blaise Pascal, a 17th century mathematician, for whom the Pascal computer programming language was named.
- **1804 Jacquard loom -** a loom programmed with punched cards invented by Joseph Marie Jacquard.
- **1850-Difference Engine, Analytical Engine-**Charles Babbage and Ada Byron. Babbage's description, in 1837, of the Analytical Engine, a hand cranked, mechanical digital computer anticipated virtually every aspect of present-day computers. It wasn't until over a 100years later that another all purpose computer was conceived. Sketch of the Engine and notes by Ada Byron King, Countess of Lovelace.
- **1939-1942- Atanasoff Berry Computer**-built at Iowa State by Prof. JohnV. Atanas off and graduate student Clifford Berry.Represented several"firsts" in computing, including a binary system of arithmetic, parallel processing, regenerative memory, separation of memory and computing functions, and more. Weighed 750 lbs. and had a memory storage of3,000 bits (0.4K).Recorded numbers by scorching marks into cards as it worked through a problem.
- **1940s - Colossus -** a vacuum tube computing machine which broke Hitler's codes during WW II. It was instrumental in helping Turing break the German's codes during WW II to turn the tide of the war. In the summer of 1939, a small group of scholars became codebreakers, working at Bletchley Part in England. This group of pioneering codebreakers helped shorten the war and changed the course of history.
- **1946 - ENIAC -** World's first electronic, large scale, general purpose computer, built by Mauchly and Eckert, and activated at the University of Pennsylvania in 1946. ENIAC recreated on a modern computer chip.The ENIAC is a 30 ton machine that measured 50 x 30 feet. It contained 19,000 vacuum tubes, 6000 switches, and could add 5,000 numbers in a second, a remarkable accomplishment at the time. A reprogrammable machine, the ENIAC performed initial calculations for the H-bomb. It was also used to prepare artillery shell trajectory tables and perform other military and scientific calculations. Since there was no software to reprogram the computer, people had to rewire it to get it to perform different functions. The human programmers had to read wiring diagrams

and know what each switch did. J. Presper Eckert, Jr. and John W. Mauchly drew on Alansoff's work to create the ENIAC, the Electronic Numerical Integrator and Computer

4 Computer Science

It's the science of computers; it answers questions like

- What are the fundamental parts of a computer? What basic building blocks do you need?
- What can be computed, what cannot be?
- How easy or how fast can something be computed? What governs these metrics

5 DataStructures & Algorithms

- How easy or how fast can something be computed? What governs these metrics
- What are the basic building blocks or tools in our programming toolkit we need to write programs that are not inefficient
- What common pattern problems/patterns exist in programs and what are the known ways of speeding those up

5.1 Let's say there is an alternate Universe

Where two guys live in adjacent houses, Cory and Jamal. There's no phones or computers yet in this universe, so Cory and Jamal do things like playing scrabble in their free time. Cory is the only one that owns a scrabble set, and since you can't play alone the only way they can play is if Jamal goes to Cory's place. But Cory's parents are rather religious, and their religion's scripture says that there will one day be a game that youngster's play that will bring about the end of the world. Cory's parents thus never want Cory to go near Scrabble.

It then becomes a game of cat and mouse between Cory and his parents, with him needing to hide his playing of scrabble. Cory's father is a nurse who sometimes must do whole night shifts at the hospital. Cory's mother is a software engineer that sometimes works very late and is not home till 2am. These are they times Cory and Jamal get to play Scrabble. But they don't have a fixed schedule, so they cannot be relied on to be absent every night. Cory has two table lamps at his desk which can be seen from Jamal's window. So they come up with a scheme. If Cory's mother isn't home, Cory turns on the right lamp. If Cory's father isn't home he turns on the left lamp. Meaning that–if both lamps are on–it's like an invite for Jamal to come play Scrabble if he feels like.

Now, sometimes Cory's sister who's usually at the University is at home, and on such days, they can only play till 1 am, which is when she comes out of her room to grab something to eat. And so Cory and Jamal decide that if his sister is home, he'll turn on the ceiling fan in his room and Jamal will know that he can come only for a little while. For every extra piece of information Cory wanted to give to Jamal, they needed on more thing. If it's was only about his mom, just one lamp being on or off gives the requisite information – one thing. If you add his dad into the mix, you need two lamps-two things. If you add his sister to the mix, you need two lamps and a ceiling fan– three things.

5.1.1 A processor in a computer kind of works in similar ways

- A processor in a computer has a set of operations it can do. Copy data, move data, Add data, subtract, multiply, divide, jump to another instruction, branch conditionally, etc.
- When you want to tell computer to do a certain operation, you need several things to convey this information about what to do to the Processor.
- Instead of lamps or fans being on or off, computers have very, very small wires with or without electricity.
- Presence or absence of Electricity conveys information about what to do. This, together, is known as one "Instruction
- Sometimes, just telling a processor to do something isn't enough. Like for jumping, you also need to tell the processor where to jump to, So you need additional wires to provide information on what Data to do the instruction on.
- Sometimes, like for say adding. One piece of information isn't enough. You need to know two numbers to add them together.
- Sometimes, along with adding, you also need to tell the processor where to store the result of whatever it is we are adding.
- Each of these 0 or 1 is known as a "bit". 8 such bits together is known as a "byte". The "size" of an instruction (the number of bits it has) depends on the kind of processor. Most modern CPUs use 64 bits.

5.1.2 Consider a Rube Goldberg machine

- It is a giant, very, very, complex contraption that has a bunch of lanes (each line below is a lane) which takes in the presence or absence of marbles, and the marbles rolling through the contraption causes some sort of side effect and this side effect is designed to be the intended operation.
- In the processor's case, presence or absence of electricity rolls through the processor on a bunch of lanes and triggers some side effect and the processor is designed such that the side effect is our intended operation.
- It just so happens that the processor itself is very tiny and everything happens so fast that there is no good tools available to us that would let us see the operation of this rube Goldberg machine.

5.1.3 Memory

- Memory is just a giant grid of cells which holds "electricity".
- If we consider our previous example, think of it as a giant grid which as a bunch of balls stored in it and that you can open lanes from any cell to let the marble (electricity) roll on into the processor to do stuff with it.

- The processor has instructions which lets it select which cells to "open" roll marbles (data) from and use this data as instructions or operands, whichever.
- When I say Memory I mean RAM or Active memory.

5.1.4 A program

- Is a region of your memory which holds instructions and data. When the first instruction from this region of memory starts going to the Processor we say that the program has started execution.

5.1.5 Java Compiler

- Is a program that takes the "code" you write as text and turns it into a Program.
- That is, turns your code into a bunch of "bits" that are VALID sequence of instructions for your processor to execute to do the thing that your code wants to do.

5.1.6 Program

- When we write and compile a program to "solve" an assignment, essentially we are creating sequences in memory filled with instructions that carry out, for example, the objective of assignment 1.
- Once the processor is done carrying out all these instructions, the assignment is complete.
- This is not exactly true, but the "GHz" or the gigahertz you see next to a processors specifications refers to the number of instructions the processor can execute in one second.
- If you have a 1GHz processor, you can execute 1 billion (10^12) instructions per second.

6 The Graph Data Structure

6.1 Background and Motivation

Imagine you want to represent the connections between you and your Instagram followers in a data structure. The diagram below (Figure 6.1) shows a simple representation of your followers as numbered vertices and the edges represent the connections between you and your followers.

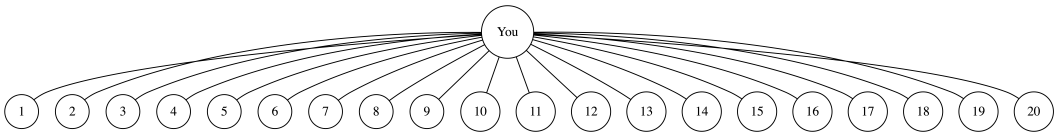


Figure 6.1: A representation of your Instagram followers.

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 6.2).

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 6.3 visualizes this directed relationship.

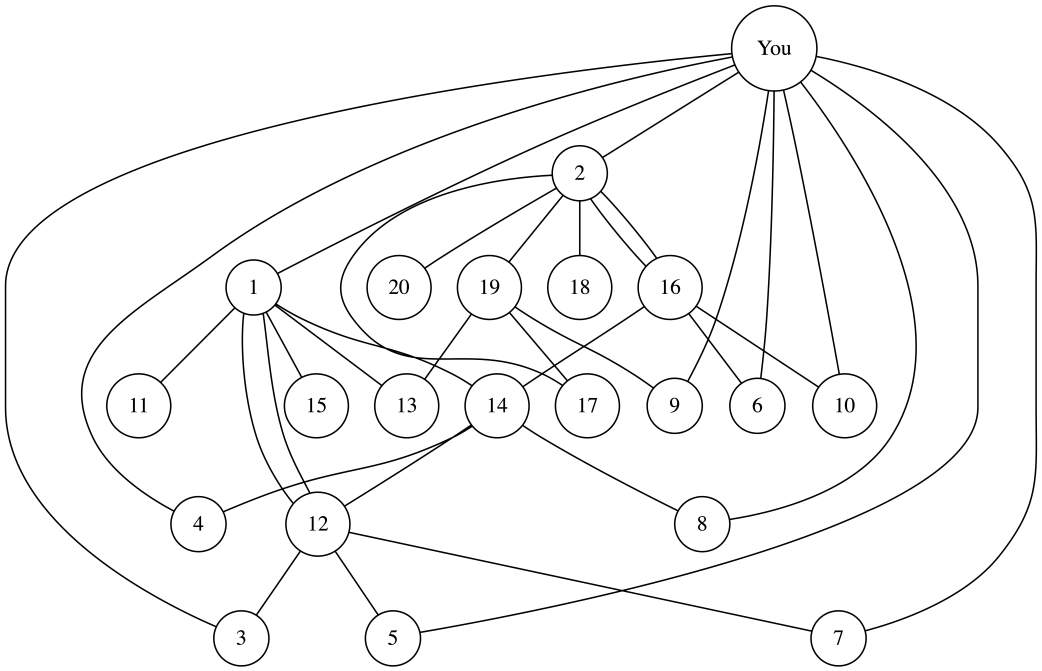


Figure 6.2: A representation of your instagram followers where they're allowed to follow other people too.

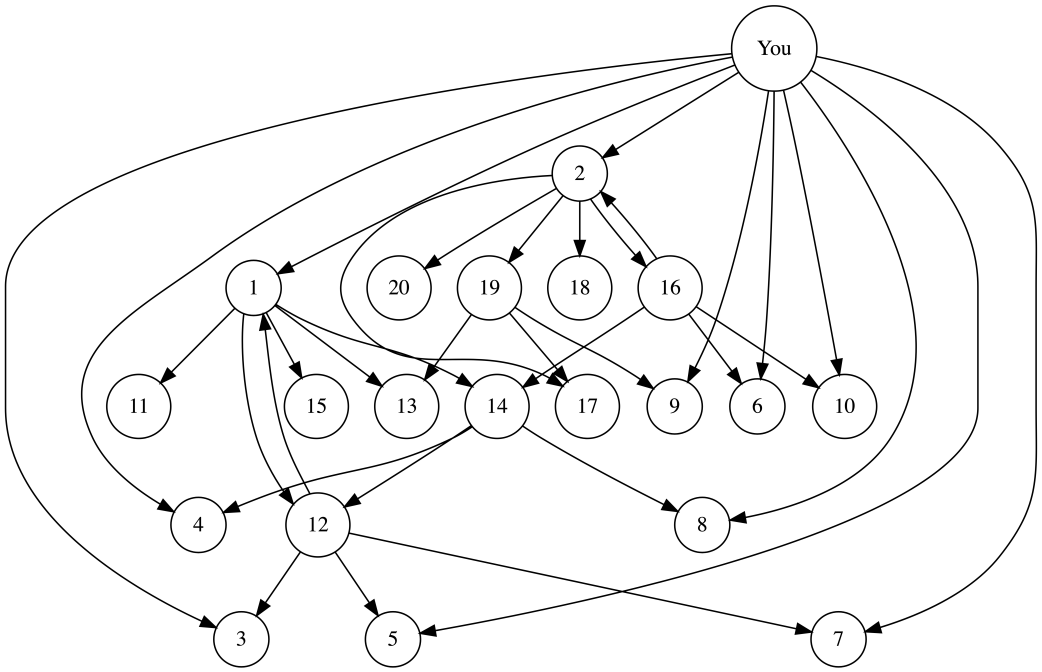


Figure 6.3: A directed representation of your Instagram followers. Here, an arrow going from vertex A to vertex B indicates that A follows B, but B does not necessarily follow A.

6.2 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

6.3 Graph Terminology

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

6.3.1 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 6.4 and Figure 6.5 for examples.)

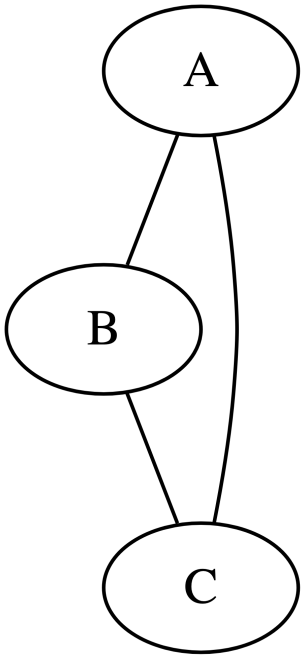


Figure 6.4: Example of an undirected graph.

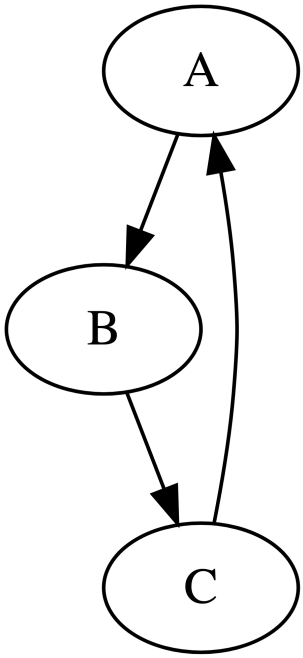


Figure 6.5: Example of a directed graph.

- **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 6.6 and Figure 6.7 for examples.)

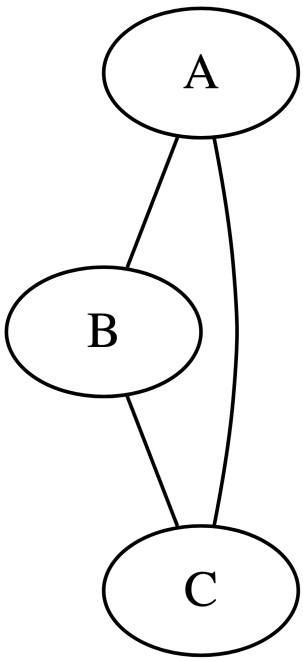


Figure 6.6: Example of an unweighted graph.

- **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 6.8 and Figure 6.9 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 6.10 for an example.

- **Path**: A path in a graph is a sequence of vertices connected by edges. See Figure 6.11 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 6.12 for an example.

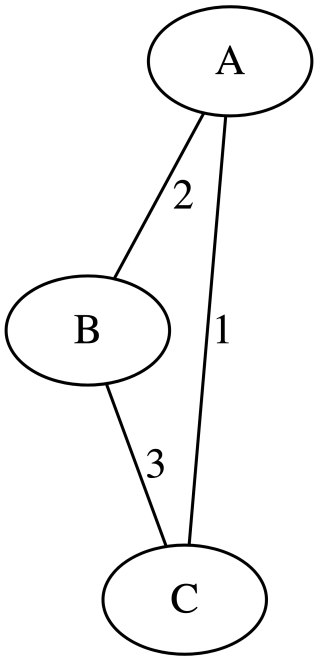


Figure 6.7: Example of a weighted graph.

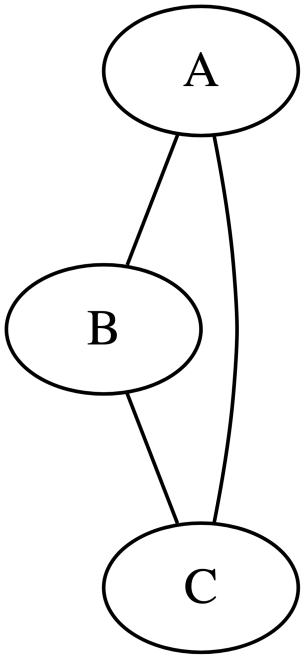


Figure 6.8: Example of a simple graph.

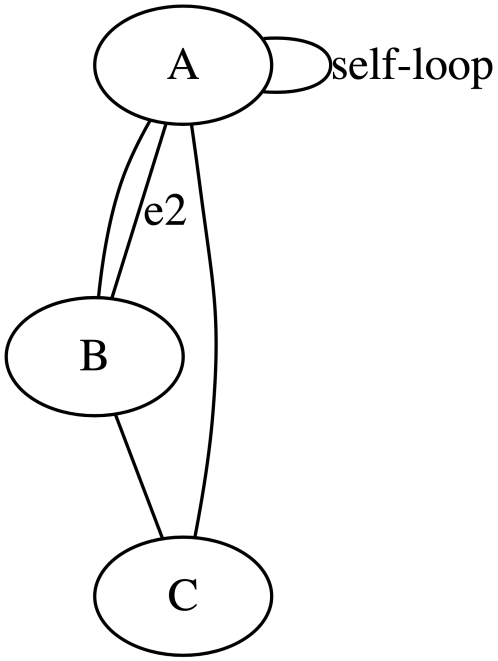


Figure 6.9: Example of a multigraph.

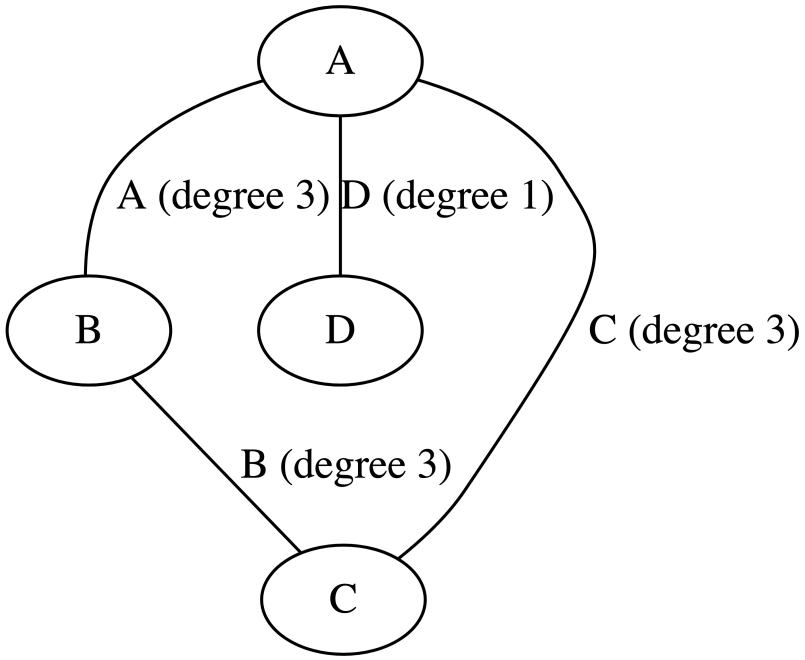


Figure 6.10: Example graph with vertex degrees.

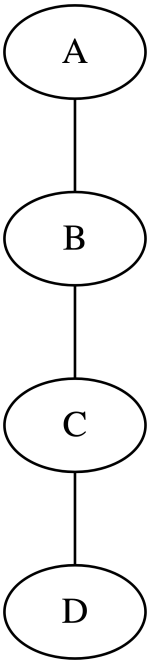


Figure 6.11: Example graph with a path from A to D.

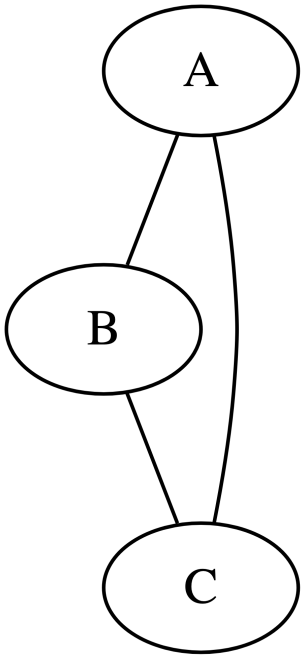


Figure 6.12: Example graph with a cycle. (A-B-C)

- **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 6.13 and Figure 6.14 for examples.

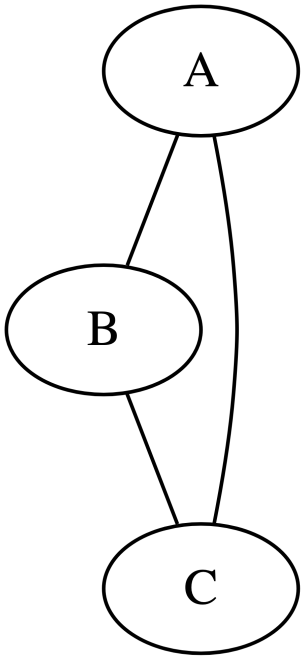


Figure 6.13: Example of a connected graph.

6.3.2 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.

6.3.3 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 6.15 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 6.16 for an example.

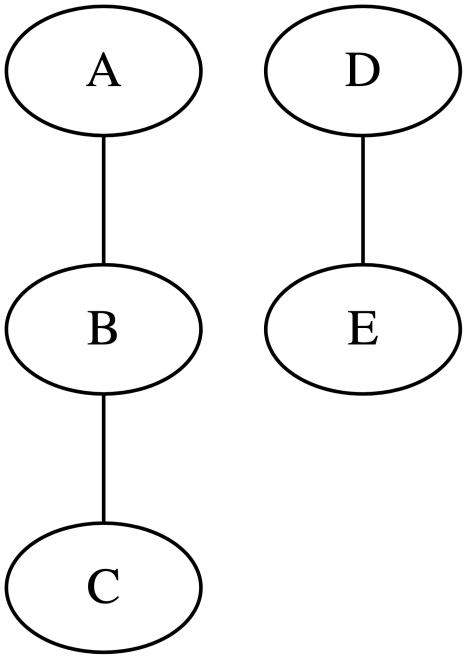


Figure 6.14: Example of a disconnected graph.

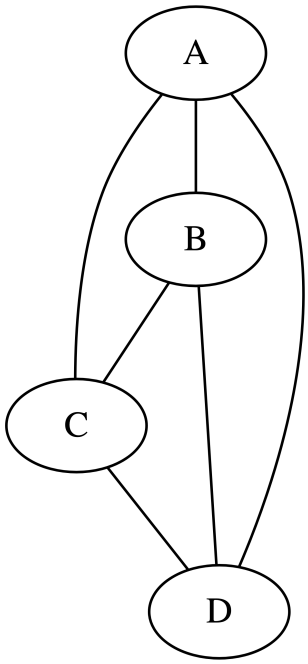


Figure 6.15: Example of a complete graph.

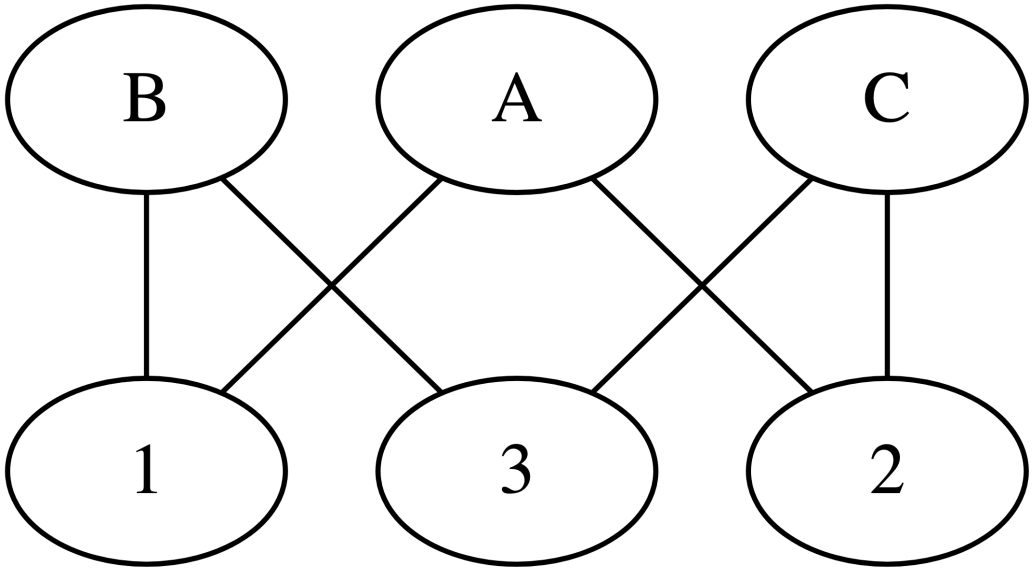


Figure 6.16: Example of a bipartite graph.

- **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 6.17 for an example.

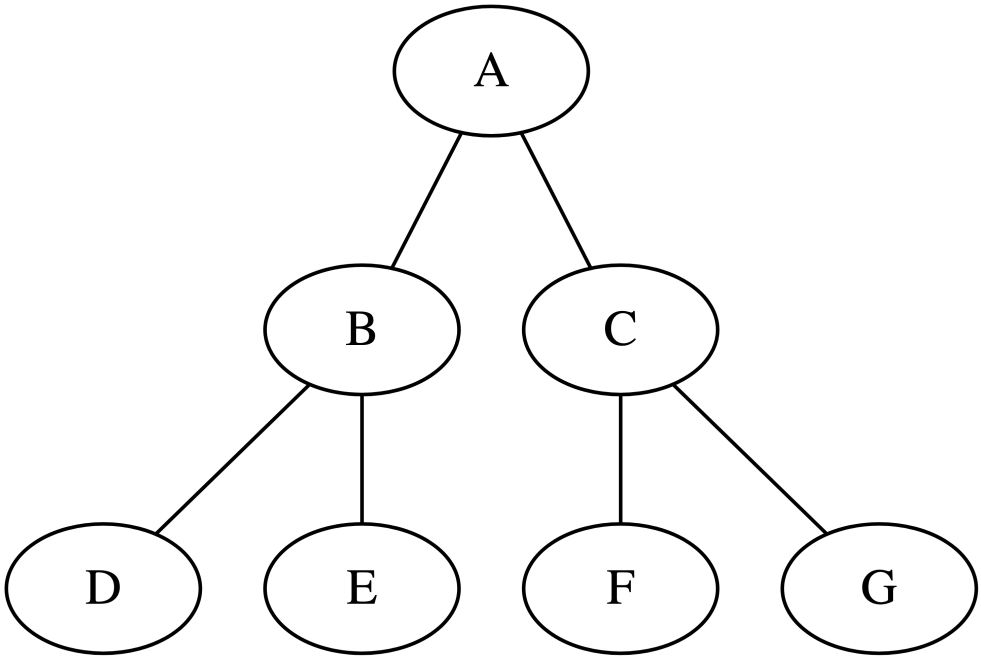


Figure 6.17: Example of a tree.

6.4 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.

6.4.1 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.

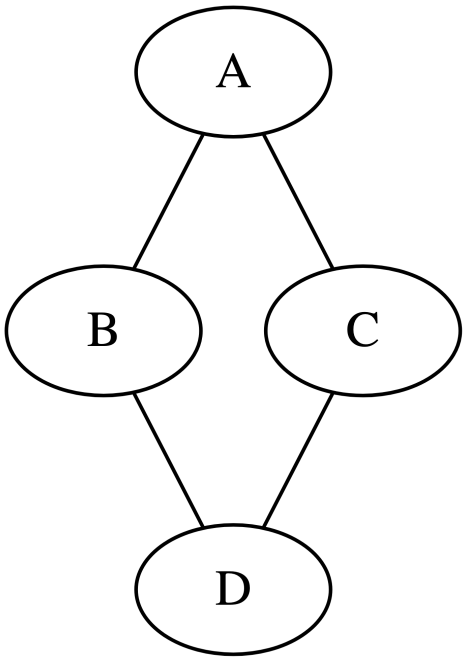


Figure 6.18: See adjacency list for this example Listing 6.1

Adjacency list representation for figure Figure 6.18:

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.

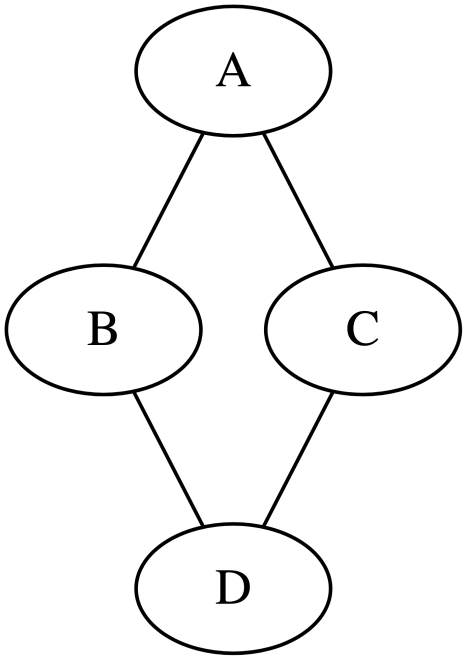
6.4.2 Adjacency Matrix

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 6.1 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 6.2

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 6.2 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
```

6.4.3 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).

6.5 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.

6.5.1 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 6.19):

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: []

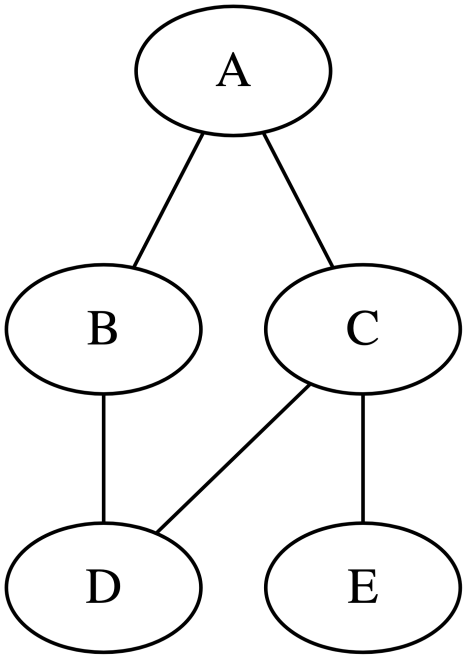


Figure 6.19: Example graph for BFS traversal.

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
```

6.5.2 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 6.20):

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:
```

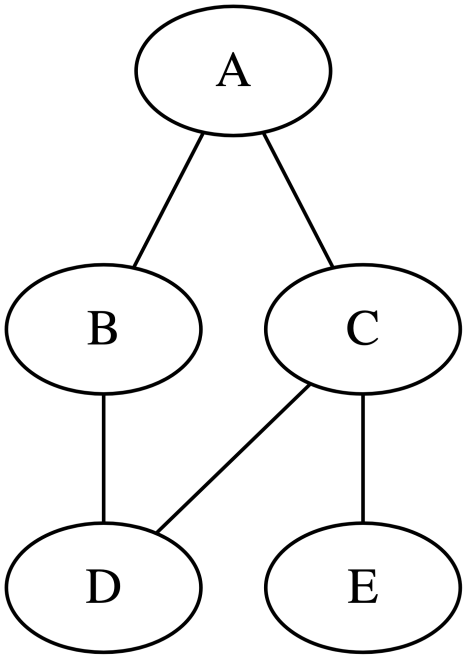


Figure 6.20: Example graph for DFS traversal.

```
    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
```

6.5.3 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.

7 Hashing, Hash Tables, and Hash Maps

7.1 Background and Motivation

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 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.

7.2 Hash Functions

7.2.1 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.

7.2.2 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.

7.2.3 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
```

7.2.4 Other Types of Hash Functions

7.2.4.1 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.

7.2.4.2 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
```

7.2.4.3 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)
```

7.2.4.4 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)
```

7.2.4.5 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).

7.2.5 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.

7.3 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**.

7.4 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.

7.4.1 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.

7.4.2 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.

7.5 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.

7.5.1 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.

7.5.2 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.

7.5.3 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.

7.6 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.

7.6.1 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).

7.6.2 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.

7.6.3 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).

7.6.4 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.

7.7 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.

7.7.1 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.

7.7.2 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.

7.8 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.

7.9 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.

7.10 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.

7.11 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.

7.12 `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.

7.12.1 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.

7.12.2 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.

7.12.3 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