# Algorithms and Data Structures 2

# Table of contents

**Preface** 1

## I Mindmaps 3

**1 Analysis of algorithms** 5

1.1 What is the analysis of algorithms . . . 5
1.2 How to measure/estimate time and space requirements . . . 5
1.3 The RAM model . . . 6
1.4 Counting up time and space units - part 1 . . . 7
1.5 Counting up time and space units - part 2 . . . 7
1.6 Growth of functions - part 1 . . . 7
1.7 Growth of functions - part 2 . . . 7
1.8 Faster computer vs faster algorithm . . . 8
1.9 Worst and best cases . . . 8
1.10 Introduction to asymptotic analysis . . . 8
1.11 Big-O notation . . . 9
1.12 Omega notation . . . 9
1.13 Theta notation . . . 9

**2 Recursive algorithms** 11

2.1 Introduction . . . 12
2.2 The structure of recursive algorithms . . . 12
2.3 Tracing a recursive algorithm . . . 12
2.4 From iteration to recursion . . . 12
2.5 Writing a recursive algorithm - part 1 . . . 13
2.6 Writing a recursive algorithm - part 2 . . . 13
2.7 The time complexity of recursive algorithms . . . 13
2.8 Solving recurrence equations . . . 13
2.9 The master theorem . . . 14

**3 Comparison sorting algorithms** 15

3.1 Introduction . . . 15

3.2 Bubble sort . . . . . 17
3.2.1 Bubble sort demonstration . . . . . 17
3.2.2 Bubble sort pseudo code . . . . . 17
3.2.3 Bubble sort time complexity . . . . . 17
3.3 Insertion sort . . . . . 17
3.3.1 Insertion sort demonstration . . . . . 17
3.3.2 Insertion sort pseudocode . . . . . 18
3.4 Selection sort . . . . . 18
3.4.1 Selection sort demonstration . . . . . 18
3.4.2 Selection sort pseudo code . . . . . 18
3.5 Quicksort . . . . . 18
3.5.1 Quicksort demonstration . . . . . 18
3.5.2 Quicksort pseudocode . . . . . 19
3.6 Mergesort . . . . . 19
3.6.1 Mergesort demonstration . . . . . 19
3.6.2 Mergesort pseudocode . . . . . 20
3.7 Flash cards . . . . . 20

**4 Non-comparison sorting algorithms 23**
4.1 Introduction . . . . . 24
4.2 Counting sort . . . . . 24
4.3 Radix sort . . . . . 24
4.4 Bucket sort . . . . . 25

**5 Hashing 27**
5.1 What is hashing? . . . . . 28
5.2 Hash tables . . . . . 28
5.3 Collisions in hash tables . . . . . 28
5.4 Flash cards . . . . . 28
5.5 Flashcards . . . . . 29

**6 Linear data structures 31**
6.1 Introduction to data structures . . . . . 32
6.2 Linked lists . . . . . 32
6.2.1 Linked lists introduction . . . . . 32
6.2.2 Linked lists insert operation . . . . . 32
6.2.3 Linked lists delete operation . . . . . 32
6.2.4 Linked lists summary . . . . . 33
6.3 Stacks . . . . . 33
6.3.1 Stacks introduction . . . . . 33
6.3.2 Stacks implementation . . . . . 33
6.4 Queues . . . . . 34
6.4.1 Queues introducion . . . . . 34
6.4.2 Queues array based implementation . . . . . 34
6.4.3 Queues list based implementation . . . . . 34

**7 Trees** **35**
7.1 Binary trees implementation . . . 36
7.2 Binary tree traversal introduction . . . 36
7.3 Depth first traversal . . . 36
7.4 Breadth first traversal . . . 37
7.5 Binary search trees . . . 37
7.6 BST insert . . . 37
7.7 BST search . . . 38
7.8 BST delete . . . 38

**8 Heaps** **39**
8.1 Implementation . . . 40
8.2 Insert, element by element . . . 40
8.3 Deletion, extract maximum . . . 41
8.4 Build in place . . . 41
8.5 Heapsort . . . 41
8.6 Heapsort's complexity . . . 41

**9 Graphs** **43**
9.1 Introduction to graphs . . . 44
9.2 Graph representations . . . 44
9.3 Minimum spanning tree problem . . . 44
9.3.1 Prims algorithm . . . 45
9.3.2 Kruskal's algorithm . . . 45
9.4 Shortest path problem . . . 45
9.4.1 Dijkstra's algorithm . . . 45

**II Module goals and objectives** **47**
1. Choose and justify appropriate data structures and algorithms to solve specific problems . . . 49
2. Express time and space complexities of specific algorithms using big-O notation . . . 49
3. Implement standard searching, sorting and path finding algorithms 50
4. Implement different data structures, and describe the consequences of particular implementation choices . . . 52
5. Compare and contrast recursive and iterative expressions of solutions to problems . . . 53
6. Describe the abstraction of collections, relate this abstraction to linear collections, and recall the basic operations that each abstraction supports . . . 54

**10 Analysis of Algorithms** **57**
10.1 Theoretical vs. empirical algorithm analysis . . . 57
10.2 Memory and time requirements from pseudocode . . . 58
10.3 Growth function and asymptotic notation . . . 59

**11 Learning outcomes** **61**
11.1 Determine time and memory consumption of an algorithm described using pseudocode . . . 61
11.2 Determine the growth function of the running time or memory consumption of an algorithm . . . 62
11.3 Use big-O, big-Omega and big-Theta notations to describe the running time or memory consumption of an algorithm . . . 63

**12 Recursive algorithms** **65**
12.1 Algorithmic recursion . . . 65
12.2 The structure of recursive algorithms . . . 65
12.3 Recurrence equations . . . 66
12.4 Master Theorem . . . 67

**13 Learning outcomes** **69**
13.1 Trace and write recursive algorithms . . . 69
13.2 Write the recursive version of an iterative algorithm using pseudocode . . . 70
13.3 Calculate the time complexity of recursive algorithms . . . 71

**14 Comparison sorting algorithms** **73**
14.1 Comparison vs. non comparison sorts . . . 73
14.2 Bubble, Insertion and Selection sort . . . 74
14.2.1 Bubble Sort . . . 74
14.2.2 Insertion Sort . . . 75
14.2.3 Selection Sort . . . 76
14.3 Recursive sorts: Mergesort and Quicksort . . . 77
14.3.1 Mergesort . . . 78
14.3.2 Quicksort . . . 80

**15 Learning outcomes** **81**
15.1 Identify the different approaches of different comparison sorting algorithms . . . 81
15.2 Implement different comparison sorting algorithms . . . 82
15.3 Calculate the time complexity of different comparison sorting algorithms . . . 84

**16 Non-comparison sorting algorithms** **85**
16.1 The limits of comparison sorts . . . 85
16.2 Counting, radix and bucket sort . . . 86
16.2.1 Counting Sort . . . 86
16.2.2 Radix Sort . . . 87
16.2.3 Bucket Sort . . . 88

**17 Learning outcomes** **91**
17.1 Identify the different approaches of different non-comparison sorting algorithms . . . 91

17.2 Implement different non-comparison sorting algorithms . . . . . . 92
17.2.1 Counting Sort . . . . . . . . . . . . . . . . . . . . . . . . . 92
17.2.2 Radix Sort (LSD) . . . . . . . . . . . . . . . . . . . . . . . 92
17.2.3 Bucket Sort . . . . . . . . . . . . . . . . . . . . . . . . . . 93
17.3 Calculate the time complexity of different non-comparison sorting algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
17.3.1 Counting Sort . . . . . . . . . . . . . . . . . . . . . . . . . 94
17.3.2 Radix Sort . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
17.3.3 Bucket Sort . . . . . . . . . . . . . . . . . . . . . . . . . . 94

**18 Hashing 95**
18.1 The problem of searching . . . . . . . . . . . . . . . . . . . . . . 95
18.2 Hash tables, hash functions . . . . . . . . . . . . . . . . . . . . . 95
18.3 Collision resolution techniques . . . . . . . . . . . . . . . . . . . 96

**19 Learning outcomes 99**
19.1 Describe the different methods used to search for data . . . . . . 99
19.2 Describe different collision resolution methods . . . . . . . . . . . 100
19.3 Implement a hash table with linear probing collision resolution . 100

**20 Linear data structures 103**
20.0.1 Data structures . . . . . . . . . . . . . . . . . . . . . . . . 103
20.1 Dynamic memory allocation . . . . . . . . . . . . . . . . . . . . . 104
20.2 Linear data structures . . . . . . . . . . . . . . . . . . . . . . . . 104

**21 Learning outcomes 107**
21.1 Describe linear data structures and its operations using pseudocode 107
21.1.1 Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
21.1.2 Linked Lists . . . . . . . . . . . . . . . . . . . . . . . . . . 107
21.1.3 Stacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
21.1.4 Queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
21.1.5 Deques (double ended queues) . . . . . . . . . . . . . . . . 108
21.2 Understand array and linked list based implementations of stacks and queues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
21.2.1 Arrays for Stacks and Queues . . . . . . . . . . . . . . . . 109
21.2.2 Linked Lists for Stacks and Queues . . . . . . . . . . . . . 109
21.3 Implement a sorted linked list . . . . . . . . . . . . . . . . . . . . 110

**22 Trees 115**
22.0.1 Trees, tree representation . . . . . . . . . . . . . . . . . . 115
22.1 Binary trees, traversal . . . . . . . . . . . . . . . . . . . . . . . . 116
22.2 Binary search trees, insert, search, delete . . . . . . . . . . . . . . 117

**23 Learning outcomes 119**
23.1 Understand how to implement a tree . . . . . . . . . . . . . . . . 119
23.2 Describe and trace different types of binary tree traversals using pseudocode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123

23.2.1 Inorder Traversal . . . 123
23.2.2 Preorder Traversal . . . 123
23.2.3 Postorder Traversal . . . 123
23.3 Describe and trace binary search tree operations using pseudocode 124
23.3.1 Insertion . . . 124
23.3.2 Search . . . 124
23.3.3 Deletion . . . 125

**24 Heaps** **127**
24.1 Heaps, shape and heap properties . . . 127
24.2 Heap operations . . . 128
24.2.1 Insertion . . . 128
24.2.2 Deletion . . . 128
24.3 Heapsort, priority queues . . . 128

**25 Learning outcomes** **131**
25.1 Check heap and shape properties . . . 131
25.2 Describe heap operations using pseudocode . . . 131
25.2.1 Max-Heapify(A, i): . . . 132
25.2.2 Build-Max-Heap(A) . . . 132
25.2.3 Heap-Extract-Max(A) . . . 132
25.2.4 Heap-Increase-Key(A, i, key): . . . 132
25.2.5 Max-Heap-Insert(A, key) . . . 133
25.3 Implement heapsort using a heap . . . 133

**26 Graphs** **135**
26.1 Graph representations . . . 135
26.2 Minimum spanning tree . . . 136
26.3 Shortest path finding . . . 136

**27 Learning outcomes** **139**
27.1 Use different implementations of graphs (matrix adjacency, list adjacency. Edge list) . . . 139
27.2 Implement Prim's algorithm to find the minimum spanning tree . 140
27.3 Implement Dijkstra's algorithm to find the shortest path between two nodes . . . 141

# Preface

Download the PDF here.

# Part I

# Mindmaps

# Chapter 1

# Analysis of algorithms

In this lecture series on algorithm analysis, different techniques for analyzing algorithms to determine their processing and memory requirements are discussed. The lecture covers two approaches for analyzing algorithms: empirical and theoretical. The Random Access Machine (RAM) model is introduced as a simplified representation of a computer that is used to estimate the running time and memory requirements of algorithms. The lecture also explains how to count the number of time and space units of an algorithm using the RAM model. Asymptotic analysis is introduced as an alternative way to describe the time or memory requirements of an algorithm, based on the Big-O, Omega, and Theta notations. The focus of the lecture is on the Theta notation, which finds a single function that acts as both an upper and lower bound for an algorithm's running time or memory space function.

## 1.1 What is the analysis of algorithms

The topic is about the analysis of algorithms to determine the best algorithm for a given task based on their correctness, ease of understanding, and computational resource consumption. The focus is on the processing and memory requirements of algorithms, which are measured in terms of the number of operations and memory positions required during execution. By analyzing these factors, one can select the best algorithm for a given task. Techniques for analyzing algorithms will be discussed in the next lecture.

## 1.2 How to measure/estimate time and space requirements

The lecture discusses two approaches for analyzing algorithms to determine their processing and memory requirements: empirical and theoretical. The empirical

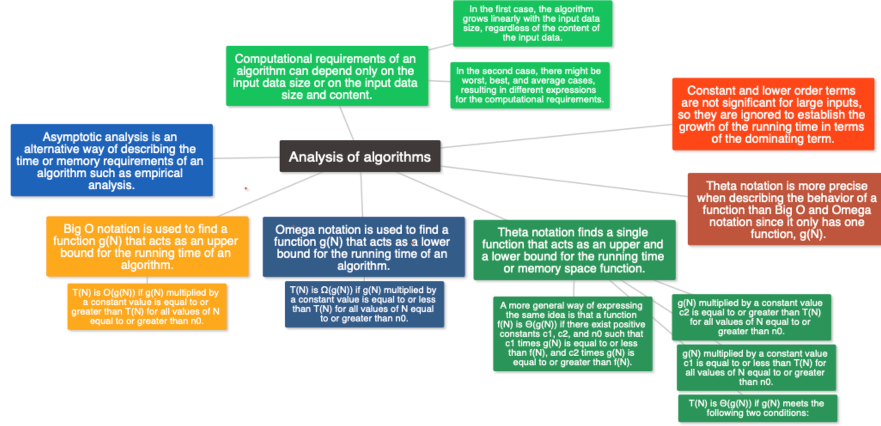



Figure 1.1: Analysis of algorithms mindmap

approach involves implementing the algorithm in a specific machine and measuring its execution time and memory consumption, while the theoretical approach involves making assumptions about the number of CPU operations and memory required based on a model of a generic computer. The advantages of the empirical approach include obtaining precise results and not needing to calculate the time and space required for each instruction, while its disadvantages include being machine-dependent and requiring implementation effort. The advantages of the theoretical approach include obtaining universal results and not needing to implement the algorithms beforehand, while its drawbacks include needing to simplify the machine model and requiring calculation effort. The course will use the theoretical approach, so learners need to learn the machine model, its assumptions and simplifications, and be prepared to make calculations.

## 1.3 The RAM model

The lecture introduces the Random Access Machine (RAM) model, which is a simplified representation of a computer that will be used to estimate the running time and memory requirements of algorithms in the course. The RAM model assumes a single CPU, where each simple operation takes one unit of time, including numerical operations, control instructions, writing or reading data from memory, and returning from a function. Loops and functions are not simple operations and need to be analyzed further. Memory is assumed to be unlimited, with no memory hierarchy, where every access takes one unit of time. The RAM model assumes that every simple variable uses one memory position, regardless of its type. In the next lecture, the RAM model will be used to practice calculating the number of operations and memory consumption of

algorithms.

## 1.4 Counting up time and space units - part 1

The video explains how to count the number of time and space units of an algorithm using the RAM model. The assumptions of the RAM model are reviewed, and a simple algorithm that returns the highest value from three input values is analyzed. The video walks through each instruction in the algorithm, and the number of time and space units required for each instruction is counted. The algorithm is made up of only simple operations, so the total number of time units is 16 and the total number of space units is one. In the next video, a more complex algorithm with a loop will be analyzed using the same process.

## 1.5 Counting up time and space units - part 2

In this video, the RAM model assumptions were revisited, and an algorithm with a for loop was analyzed to calculate the time and space units it requires. The linear search algorithm in an array containing N elements and one non-repeated element is used as an example. The loop instruction inside the algorithm was treated as not a simple operation, and its components were analyzed to determine the time units. The loop is executed N+1 times, and its time units are 7N+5. The rest of the instructions were analyzed to have a total time unit of 12N+6. The algorithm requires only one space unit. The video concludes that with this knowledge, one could determine the time and space units of other simple algorithms.

## 1.6 Growth of functions - part 1

In this video, the presenter explains that calculating the exact running time of an algorithm can be tedious and error-prone, and that it is sufficient to compare the growth rates of different algorithms to determine which is faster. By ignoring coefficients and lower order terms in the function describing the running time, it is possible to simplify calculations and still obtain meaningful results. The presenter shows how different growth functions can significantly impact the running time of an algorithm, and concludes that these principles also apply to memory requirements.

## 1.7 Growth of functions - part 2

In this video, it is shown how to calculate the growth of an algorithm's running time without counting every single time unit, instead using a generic constant to refer to the time units taken by simple operations. The example used is an algorithm that calculates the sum of the elements in the diagonal of a square

matrix. By checking if an instruction takes constant time or not, the algorithm's running time can be expressed as a simple expression and determined to be linear with n. This method allows for quick analysis of algorithms while still being able to compare the running time of two different algorithms in terms of their dominant terms.

## 1.8 Faster computer vs faster algorithm

In this lecture, the importance of knowing the growth of the running time of an algorithm is discussed. A comparison is made between the impact of a faster computer and a faster algorithm on the execution time of an algorithm with a quadratic growth. It is shown that investing time in a faster algorithm is always better than investing money on a faster computer, as the growth of the algorithm is not affected by a faster computer. It is emphasized that knowing the best growth expected from an algorithm for a specific problem is crucial for solving big instances of that problem. The lecture concludes with a teaser about the next lecture on calculating the growth of the running time of an algorithm for best and worst cases.

## 1.9 Worst and best cases

In this lecture, it is discussed that some algorithms' running time growth depends not only on the input data size but also on the content of the input data. When analyzing such algorithms, it is essential to specify whether the growth rate corresponds to the worst or the best case. The lecture provides two examples, one is the "Find Maximum" algorithm, which has a linear running time that only depends on the input data size, while the second is the "Linear Search" algorithm, which has a different running time for different inputs. In the worst case, it has a linear running time, and in the best case, it has a constant running time. The lecture summarizes that to analyze the worst case, we look at cases where the algorithm executes the maximum possible number of instructions, while for the best case, we look at cases in which the algorithm executes the minimum possible number of instructions.

## 1.10 Introduction to asymptotic analysis

In this lecture, the concept of asymptotic analysis is introduced as an alternative way to describe the time or memory requirements of an algorithm. Asymptotic analysis compares the growth of the running time functions of different algorithms as the input data size approaches infinity. It is based on three notations: Big-O, Omega, and Theta, which were originally devised by mathematicians to describe the behavior of a function when its argument tends to infinity. The use of these notations in analyzing algorithms when the input data is large is called asymptotic analysis.

## 1.11 Big-O notation

In this lecture, the Big O notation is introduced as a way to describe the time or space requirements of an algorithm. The goal is to find a function g(n) that acts as an upper bound for the algorithm's resource requirements, T(n). The Big O notation defines a set of functions that can act as an upper bound for T(n). The notation T(n) is O(g(n)) if g(n) is multiplied by a constant value, c, that is equal to or greater than T(n) for all values of n equal to or greater than n0. The lecture provides an example of finding a function g(n) such that T(n) is O(g(n)) for a given T(n) function. The viewer is then asked to find values of c and n0 that make T(n) O(n^3), O(n^4), and O(2^n).

## 1.12 Omega notation

In this lecture, the Omega notation is introduced, which is similar to the Big O notation but looks for functions that act as a lower bound for the running time of an algorithm. The function T(N) represents the running time of an algorithm, and the aim is to find a function g(N) that acts as a lower bound for T(N). For any function that satisfies this condition, we can say that T(N) is Omega g(N). The formal definition of the Omega notation is provided, and an example is given to demonstrate how to find a lower bound function g(N) that satisfies the condition. Finally, the audience is invited to demonstrate that T(N) is Omega logN and Omega 1 by finding values of c and n0 that make c times g(N) equal to or less than T(N) for all values of N equal to or greater than n0.

## 1.13 Theta notation

In this lecture, we learned about Theta notation which was created to overcome the limitations of Big O and Omega notations. Theta notation finds a single function that acts as both an upper and lower bound for an algorithm's running time function or memory space function. We revisited the example from the previous lectures, where t(n) represents the running time of an algorithm, and found that the function g(n) that acts as both an upper and lower bound for t(n) is n^2. We defined Theta notation formally, stating that T(n) is Theta of g(n) if the function g(n) meets two conditions: g(n) multiplied by a constant value c1 is less than or equal to T(n) for all values of n greater than or equal to n0, and g(n) multiplied by a constant value c2 is greater than or equal to T(n) for all values of n greater than or equal to n0. Unlike Big O and Omega notations, which correspond to a set of functions g(n), Theta notation only has one function, g(n), making it more precise when describing the behavior of a function.

# Chapter 2

# Recursive algorithms

This topic covers recursion in computer science, including understanding, creating, and analyzing recursive algorithms. The structure of a well-built recursive algorithm that executes a finite number of recursive calls is explained, and the importance of tracing recursive algorithms to understand their task is highlighted. The relationship between iterative and recursive algorithms is discussed, and the technique of doing a small part of the job and delegating the rest is explained for writing recursive algorithms. The time complexity of recursive algorithms is analyzed using recurrence equations, and the master theorem is introduced as a method for analyzing the time complexity of recursive algorithms with a specific structure.

Figure 2.1: Recursive algorithms mindmap

## 2.1 Introduction

This topic is about recursion in computer science, which is the use of an algorithm that calls itself. The topic is divided into three parts. The first part involves understanding recursion by studying different recursive algorithms and tracing their steps to determine their purpose. The second part involves creating your own recursive algorithms, while the third part involves analyzing recursive algorithms by determining their running time. An example of a recursive algorithm is given, which prints “hello” on the screen and then calls itself, resulting in infinite recursion. In the next lecture, a well-constructed recursive algorithm that performs a finite number of recursive calls will be studied.

## 2.2 The structure of recursive algorithms

This lecture is about the structure of a well-built recursive algorithm that executes a finite number of recursive calls. A modified version of the recursive printing algorithm is presented, which includes a base case to stop the recursion and an input argument that gets closer to the base case with every recursive call. An example is provided, and the algorithm is traced step-by-step to demonstrate how it prints the word “hello” on the screen a number of times equal to the input argument. It is recommended to implement the algorithm and experiment with different conditions for the base case and the recursive call.

## 2.3 Tracing a recursive algorithm

In this lecture, the instructor shows how to trace a recursive algorithm in order to understand the task performed by the algorithm. They use the example of a function F that takes two integer arguments, a and b, and returns the sum of a and b using recursion. The base case is when b is equal to zero, in which case a is returned. The instructor walks through the process of tracing the algorithm with input arguments of 7 and 3, showing how the recursive calls work and how the value of a is incremented with each call. The final result is 10, the sum of 7 and 3. The key takeaway is that tracing a recursive algorithm can help you understand what task it is performing, even if it is not immediately obvious.

## 2.4 From iteration to recursion

This lecture explains the relationship between iterative and recursive algorithms. Both types of algorithms repeat a set of instructions, but iterative algorithms use loops to do so, while recursive algorithms call themselves. Both types need an initial value and a condition to stop repetition, but for iterative algorithms, the condition must be true to continue repetition, while for recursive algorithms, it must be true to stop repetition. Recursive algorithms can be more concise, but they are more difficult to understand and use more memory.

## 2.5 Writing a recursive algorithm - part 1

In this lecture, the instructor explains how to ideate recursive algorithms from scratch. Instead of looking into the details of recursive processing, we should view recursive calls as black boxes for which we only know the result. The instructor uses the example of determining the number of marbles in a bag to illustrate the approach of doing a small part of the job and delegating the rest without caring about the details of how the delegated task is going to be executed. The base case occurs when the person receiving the bag is left with an empty bag. The instructor encourages the viewers to review the recursive algorithms studied so far and identify the small job the algorithm does by itself and the part that gets delegated.

## 2.6 Writing a recursive algorithm - part 2

In this lecture, the speaker explains how to apply the technique of doing a small part of the job and delegating the rest to write a recursive linear search algorithm. The algorithm takes an array, the number of elements in the array, and the number to search for as input. The small part of the job is checking if the number to search for is in the last position of the array, and if so, returning true. If not, the algorithm delegates the remaining positions of the array to a recursive call. The base case occurs when all positions of the array have been checked, and if the number to search for has not been found, false is returned. The algorithm is then ready for tracing to check its correctness.

## 2.7 The time complexity of recursive algorithms

In this lecture, we learn how to analyze the time complexity of recursive algorithms. We start with the recursive implementation of the factorial function and define t(n) as the running time of the algorithm for an input size of n. We break down the code into simple operations and use recurrence equations to express the running time of the algorithm in terms of the running time of a smaller input. We end up with an expression for t(n) that is itself recursive and describes the running time of the problem of size n in terms of the running time of a smaller input. This expression is known as a recurrence equation, and we will learn how to solve them in the next lecture.

## 2.8 Solving recurrence equations

In the previous lecture, we learned about recurrence equations that describe the running time of recursive algorithms. In this lecture, we learn how to solve recurrence equations. To solve a recurrence equation, we need to find a value of N for which the running time is known, and then we expand the right side of the recurrence equation until we can't replace the known value of T(N) on it.

We then apply a pattern that arises from the expansion of the equation, and we simplify it to get an explicit expression for the running time. Once we have an explicit expression for the running time, we perform an asymptotic analysis to determine the complexity class of the running time. We can express T(N) using the big O, omega, and theta notations.

## 2.9 The master theorem

In this lecture, the master theorem was introduced as a method to analyze the time complexity of recursive algorithms. The master theorem can be applied only if the recurrence equation has a specific structure, where the coefficient a is greater than or equal to one and the coefficient b is strictly greater than one. The theorem classifies the recurrence equation into one of three cases depending on whether the function f(n) is less than, equal to, or greater than n to the log base b of a. After identifying the case, the master theorem can be used to determine the theta notation for the running time of the algorithm. The three cases give three different theta notations, Theta of n to the log base b of a, Theta of n to the log base b of a multiplied by log n, and Theta of f(n). It is important to check whether the master theorem can be applied to a recurrence equation before attempting to solve it.

# Chapter 3

# Comparison sorting algorithms

In the lectures, the concept of Merge Sort, a comparison sorting algorithm, was introduced. The implementation of Merge Sort using arrays as data structures and an out-of-place implementation for the merge part was explained. The pseudocode for the recursive implementation of Merge Sort was analyzed, and the different parts of the algorithm and their tasks were defined. The execution of Merge Sort was simulated with a specific small input, and each step of the algorithm was explained, including the recursive calls and the merge function. The tasks given were to write the pseudocode for the merge function and to calculate the worst and best case time complexity of Merge Sort. The solution document was provided for these tasks, which can be used to check the correctness of the solutions.

## 3.1 Introduction

Topic 3 is about sorting in Computer Science, which involves arranging data in ascending or descending order. Sorting algorithms can be classified into two categories, comparison and non-comparison sorts. Comparison sorts, such as Bubble sort, Insertion sort, Selection sort, Quick sort, and Merge sort, compare pairs of elements to determine sorting order. Non-comparison sorts, such as Counting sort, Radix sort, and Bucket sort, do not make comparisons. The main point of learning non-comparison sorts is that comparison sorts have a limit to their running time, which cannot be faster than n times log n in the worst case. The lecture provides a summary of the worst and best case time complexity for the most known comparison sorts and explains that there is no comparison sort that can run faster than n times log n in the worst case. The lecture concludes by saying that non-comparison sorts can beat the lower bound

Figure 3.1: Comparison sorting algorithms

of n times log n of comparison sorts.

## 3.2 Bubble sort

### 3.2.1 Bubble sort demonstration

The lecture is about the bubble sort algorithm, which gets its name from how the highest numbers "bubble up" to the highest positions in the array during sorting. The lecture demonstrates the execution of bubble sort on an unsorted array of five numbers, explaining how the algorithm compares and swaps pairs of elements to sort the array from smallest to largest. The lecture goes through each pass of the algorithm until the array is fully sorted, indicating that no more swaps are needed.

### 3.2.2 Bubble sort pseudo code

The lecture reviews the pseudocode of the bubble sort algorithm, which has a while loop nested in a for loop. The while loop checks if a swap was made during the last pass through the array and if so, continues to execute the for loop. The for loop compares pairs of elements and swaps them if they are in the wrong order. The algorithm sorts the array from smallest to largest. After each pass, the value of n is decreased to avoid comparing already sorted numbers. The lecture includes a step-by-step simulation of the algorithm's execution on a small input array, demonstrating how the highest numbers bubble up to the highest positions during sorting. Finally, the lecture notes that the time complexity of bubble sort is still theta n to the square in the worst case, despite the fast best-case performance of this implementation.

### 3.2.3 Bubble sort time complexity

In this lecture, the time complexity of bubble sort is analyzed for both the best and worst cases. In the best case, the array is already sorted, and the time complexity is linear with the size of the array, represented as O(N), Omega(N), and Theta(N). In the worst case, when the array is sorted in reverse order, the time complexity is quadratic with the size of the array, represented as O(N^2), Omega(N^2), and Theta(N^2). The worst-case analysis shows that the algorithm's running time grows quadratically with the size of the array.

## 3.3 Insertion sort

### 3.3.1 Insertion sort demonstration

The lecture explains the concept of insertion sort, where each element in an array is picked and inserted into its correct position in a sorted part of the array. The lecture walks through an example of sorting an array using insertion sort,

demonstrating how elements are compared and moved to make space for the new element, until the array is fully sorted.

### 3.3.2 Insertion sort pseudocode

The lecture reviews the pseudocode for insertion sort and simulates its execution with a small input. The pseudocode involves a for-loop that selects the next element to be inserted in the sorted part of the array and a while loop that determines the correct position to insert the number and moves the numbers to the right to make space. The sorted part of the array is increased in each iteration of the for-loop. The simulation shows how the algorithm sorts an array of five elements. The lecture concludes by inviting the listener to analyze the time complexity of the algorithm for the best and worst cases.

## 3.4 Selection sort

### 3.4.1 Selection sort demonstration

In this lecture, the instructor explains the concept of selection sort. Selection sort involves finding the smallest number in the array and storing it in its correct position. The instructor demonstrates the sorting process by visually selecting the smallest number and swapping it with the number in the correct position. The process is repeated until the entire array is sorted. The instructor also notes that the process can be optimized by packing the function that finds the smallest number in one step.

### 3.4.2 Selection sort pseudo code

The lecture explains the pseudocode of selection sort and simulates its execution using a specific input array. The algorithm works by finding the minimum value in the unsorted portion of the array and placing it in its correct position. This process is repeated until the entire array is sorted. The lecture also answers two questions: 1) Selection sort is a comparison sort, and comparisons are made within the pos_min function. 2) The worst and best case time complexity of selection sort is O(n^2), the same as bubble sort.

## 3.5 Quicksort

### 3.5.1 Quicksort demonstration

This lecture covers the idea behind Quicksort, which is one of the two recursive comparison sorting algorithms. In Quicksort, the pivot is chosen in every call and all numbers lower than the pivot are stored to its left, and all numbers higher than the pivot are stored to its right. The execution of Quicksort starts with choosing the pivot, and in this implementation, the highest number is

chosen as the pivot. After that, every number lower than the pivot goes to the left part of the array, and every number higher than the pivot goes to the right part of the array. The Quicksort algorithm is then applied recursively on both parts of the array until the array is fully sorted.

### 3.5.2 Quicksort pseudocode

The lecture discusses quicksort, a comparison sorting algorithm that is easy to code using recursion. Quicksort calls itself twice and recursively processes two sections of the array: the left section from position low to p-1, and the right section from position p+1 to high. The function partition is called for each section and performs three tasks: selecting the pivot, rearranging the array to move all numbers lower than the pivot to its left and all numbers greater than the pivot to its right, and returning the position of the pivot. The execution of quicksort stops when low is no longer lower than high. The lecture provides a simulation of the algorithm execution for a specific input array.

The tasks assigned are to write pseudocode for the function partition, which selects the pivot and rearranges the array, and to calculate the worst and best case time complexity of quicksort using the master theorem.

## 3.6 Mergesort

The lecture explains the idea behind merge sort, which involves dividing an array into halves until there is only one element in each part. The algorithm then merges neighboring pairs of sorted elements, creating larger and larger sorted arrays until the full array is sorted. The lecture gives an example of a five element array being sorted using merge sort. It is explained how comparisons and merging are done at each step until a sorted array is obtained. The lecture concludes by mentioning that the implementation of merge sort will be covered in the next lecture.

### 3.6.1 Mergesort demonstration

In this lecture, the execution of Merge sort with a specific small input was simulated, and the step-by-step process was explained in detail. It was emphasized that after executing the first recursive calls, the left half of the array between the positions l and h is already sorted. The execution of the algorithm was completed with the merge function, which copies the left and right halves of the array into two new arrays, sorts them, and merges them back into the original array. The task of writing the pseudocode for the merge function was given, which involves copying already sorted elements into new arrays and comparing them to determine the order in which they should be copied back into the original array. The second task was to calculate the worst and best case time complexity of Merge sort using the master theorem.

### 3.6.2 Mergesort pseudocode

This lecture is about implementing Merge sort. Merge sort is a comparison sorting algorithm that can be coded easily using recursion. There are different implementations of Merge sort depending on the data structure used to store the numbers and the specific algorithm used for the merge part. This lecture focuses on the implementation that uses arrays as data structures and an out-of-place implementation for the merge part, which uses extra memory space to improve time complexity. The pseudocode for the recursive implementation of Merge sort is analyzed in detail, including the recursive structure, base case, and tasks assigned to each part of the code. The execution of the algorithm is demonstrated step-by-step using a specific example array. The merge function is also briefly introduced.

## 3.7 Flash cards

**Flashcard for "Merge Sort"**

Front: What is Merge Sort?

Back: Merge Sort is a comparison sorting algorithm that involves dividing an array into halves until there is only one element in each part. The algorithm then merges neighboring pairs of sorted elements, creating larger and larger sorted arrays until the full array is sorted.

Front: What is the time complexity of Merge Sort?

Back: The worst and best case time complexity of Merge Sort is O(n*log(n)), which is faster than bubble sort, insertion sort, and selection sort in the worst case.

**Flashcard for "Comparison and non-comparison sorting algorithms"**

Front: What is a comparison sorting algorithm?

Back: A comparison sorting algorithm compares pairs of elements to determine sorting order, such as Bubble sort, Insertion sort, Selection sort, Quick sort, and Merge sort.

Front: What is a non-comparison sorting algorithm?

Back: A non-comparison sorting algorithm does not make comparisons, such as Counting sort, Radix sort, and Bucket sort.

**Flashcard for "Bubble sort demonstration"**

Front: What is Bubble sort?

Back: Bubble sort is a comparison sorting algorithm that gets its name from how the highest numbers "bubble up" to the highest positions in the array during sorting.

Front: How does Bubble sort work?

Back: Bubble sort works by comparing pairs of elements and swapping them if they are in the wrong order, sorting the array from smallest to largest.

**Flashcard for "Insertion sort demonstration"**

Front: What is Insertion sort?

Back: Insertion sort is a comparison sorting algorithm that involves selecting each element in an array and inserting it into its correct position in a sorted part of the array.

Front: How does Insertion sort work?

Back: Insertion sort works by comparing elements and moving them to make space for the new element, until the array is fully sorted.

**Flashcard for "Selection sort demonstration"**

Front: What is Selection sort?

Back: Selection sort is a comparison sorting algorithm that involves finding the smallest number in the array and storing it in its correct position.

Front: How does Selection sort work? Back: Selection sort works by visually selecting the smallest number and swapping it with the number in the correct position, repeating the process until the entire array is sorted.

**Flashcard for "Quicksort demonstration"**

Front: What is Quicksort?

Back: Quicksort is a comparison sorting algorithm that involves choosing a pivot and storing all numbers lower than the pivot to its left and all numbers higher than the pivot to its right.

Front: How does Quicksort work?

Back: Quicksort works by recursively processing two sections of the array, the left section from position low to p-1, and the right section from position p+1 to high, until the array is fully sorted.

**Flashcard for "Mergesort demonstration"**

Front: What is the Merge sort algorithm?

Back: Merge sort is a comparison sorting algorithm that involves dividing an array into halves until there is only one element in each part. The algorithm then merges neighboring pairs of sorted elements, creating larger and larger sorted arrays until the full array is sorted.

Front: How does Merge sort work?

Back: Merge sort works by recursively dividing the array into halves, sorting each half, and merging the sorted halves back into the original array until the full array is sorted.

# Chapter 4

# Non-comparison sorting algorithms

These lectures covers the concept of decision trees for sorting algorithms and the worst-case time complexity of comparison sorts, followed by the introduction of non-comparison sorting algorithms such as counting sort, radix sort, and bucket sort. The lecture includes a review of the pseudocode for counting sort and a simulation of its execution with a small input, as well as an explanation and implementation challenge for radix sort using provided pseudocode. The lecture concludes with an overview of bucket sort, including a high-level pseudocode and worst and best case complexity analysis.

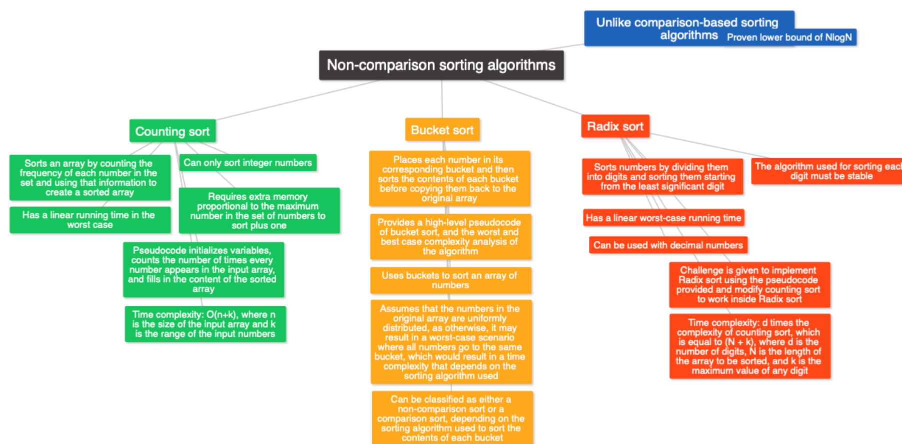



Figure 4.1: Non-comparison sorting algorithms

## 4.1 Introduction

This topic discusses the maximum number of comparisons a sorting algorithm must make to find the correct arrangement among all the different arrangements of an unsorted array of N elements, which is visualized as a decision tree. The length of the longest path in the decision tree is equal to the maximum number of comparisons made by a sorting algorithm, and it is proven that no comparison sort algorithm can be better than NlogN in the worst case. Then, three sorting algorithms that do not use comparisons in their core processing and have linear running times in the worst case are introduced: Counting sort, Radix sort, and Bucket sort.

## 4.2 Counting sort

Counting sort is a non-comparison sorting algorithm that has a linear running time in the worst case. It sorts an array by counting the frequency of each number in the set and using that information to create a sorted array. The algorithm works by first creating an array C with the correct number of elements based on the maximum value stored in A plus 1, then counting the frequency at which the elements in A appear and storing that information in C, and finally, creating the resulting sorted array R by visiting every position in C and copying the corresponding elements as many times as required. Counting sort can only sort integer numbers and requires extra memory proportional to the maximum number in the set of numbers to sort plus one.

In this lecture, the pseudocode for the counting sort algorithm was reviewed and the execution of the algorithm was simulated with a specific small input. The pseudocode initializes variables, counts the number of times every number appears in the input array, and fills in the content of the sorted array. The time complexity of counting sort is O(n+k), where n is the size of the input array and k is the range of the input numbers. The lecture concludes by asking the viewer to calculate the time complexity of counting sort.

## 4.3 Radix sort

In this lecture, you will learn about Radix sort, which is a non-comparison sort that sorts numbers by dividing them into digits and sorting them starting from the least significant digit. The numbers are sorted in passes, and the algorithm used for sorting each digit must be stable. Radix sort has a linear worst-case running time and can be used with decimal numbers. The time complexity of Radix sort is d times the complexity of counting sort, which is equal to (N + k), where d is the number of digits, N is the length of the array to be sorted, and k is the maximum value of any digit. A challenge is given to implement Radix sort using the pseudocode provided and modify counting sort to work inside Radix sort.

## 4.4 Bucket sort

The lecture introduces bucket sort, a non-comparison sorting algorithm that uses buckets to sort an array of numbers. The algorithm works by placing each number in its corresponding bucket and then sorting the contents of each bucket before copying them back to the original array. The algorithm assumes that the numbers in the original array are uniformly distributed, as otherwise, it may result in a worst-case scenario where all numbers go to the same bucket, which would result in a time complexity that depends on the sorting algorithm used. Bucket sort can be classified as either a non-comparison sort or a comparison sort, depending on the sorting algorithm used to sort the contents of each bucket. The lecture provides a high-level pseudocode of bucket sort, and the worst and best case complexity analysis of the algorithm.

# Chapter 5

# Hashing

The series lecture introduces hashing, which is the process of transforming alphanumeric characters into a value using a hash function. Hashing is used to efficiently search for elements in an array, with hash tables being widely used in computer science. The lectures covers four algorithmic solutions to searching for an element in an array, including linear search, binary search, direct addressing, and hashing. The lectures also discusses collision resolution in hash tables and presents three methods for dealing with collisions: extent and re-hash, linear probing, and separate chaining. The best case time complexity for hash table operations is theta 1, while the worst case time complexity depends on the method of collision resolution used.

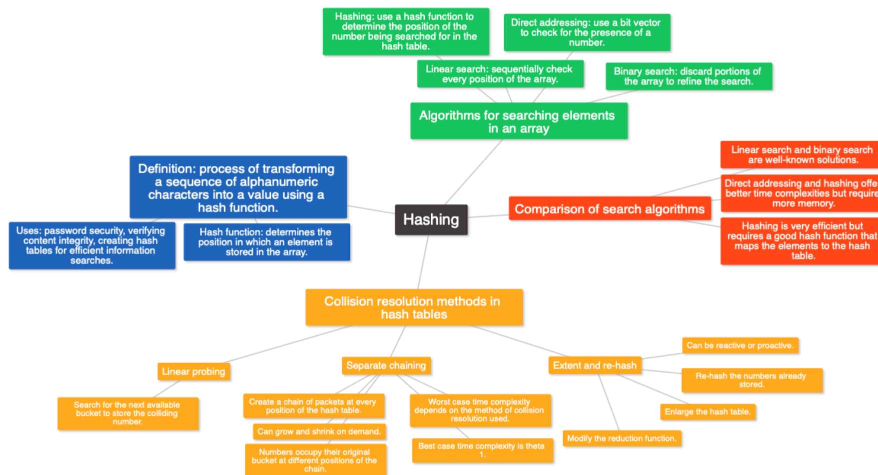



Figure 5.1: Hashing Mind Map

## 5.1 What is hashing?

Hashing is the process of transforming a sequence of alphanumeric characters into a value using a hash function. Hash functions are used for password security and verifying content integrity in security applications. Hashing is also used to create hash tables for efficient information searches. A hash function is used to determine the position in which an element is stored in the array, and when searching for an element, the hash function is applied to it to quickly determine its position in the array. Hash tables are widely used in computer science.

## 5.2 Hash tables

The lecture discusses the problem of searching for an element in an array and presents four algorithmic solutions to it: linear search, binary search, direct addressing, and hashing. Linear search and binary search are well-known solutions, with binary search being faster with a time complexity of theta log n in the worst case. Direct addressing and hashing offer even better time complexities of theta 1, but at the cost of memory usage. Hashing, in particular, is very efficient, but its effectiveness depends on having a good hash function that maps the elements to the hash table. The lecture concludes by acknowledging that the efficiency of hashing depends on knowing the elements that will be stored in the hash table beforehand, and that the next lecture will address the issues that arise when this is not the case.

## 5.3 Collisions in hash tables

This lecture discusses three methods for collision resolution in hash tables: extent and re-hash, linear probing, and separate chaining. Collision resolution is necessary when two or more numbers are hashed to the same position in the hash table. Extent and re-hash consists of enlarging the hash table, modifying the reduction function, and re-hashing the numbers already stored. Linear probing searches for the next available bucket to store the colliding number. Separate chaining creates a chain of packets at every position of the hash table, where the numbers occupy their original bucket at different positions of the chain. The best case time complexity for hash table operations is theta 1, while the worst case time complexity depends on the method of collision resolution used.

## 5.4 Flash cards

**What is hashing?**

Hashing is the process of transforming a sequence of alphanumeric characters into a value using a hash function.

**What are hash functions used for in security applications?**

Hash functions are used for password security and verifying content integrity in security applications.

**What are hash tables used for?**

Hashing is used to create hash tables for efficient information searches.

**What are the four algorithmic solutions for searching for an element in an array?**

Four algorithmic solutions for searching for an element in an array are linear search, binary search, direct addressing, and hashing.

**Which search algorithm is faster between binary search and linear search?**

Binary search is faster than linear search with a time complexity of theta log n in the worst case.

**Which two search algorithms offer the best time complexities, and what is the tradeoff?**

Direct addressing and hashing offer time complexities of theta 1, but at the cost of memory usage.

**What is necessary for the effectiveness of hashing?**

The effectiveness of hashing depends on having a good hash function that maps the elements to the hash table.

**What are the three methods for collision resolution in hash tables?**

Three methods for collision resolution in hash tables are extent and re-hash, linear probing, and separate chaining.

**What is the best and worst case time complexity for hash table operations?**

The best-case time complexity for hash table operations is theta 1, while the worst-case time complexity depends on the method of collision resolution used.

## 5.5 Flashcards

Question: What is recursion in computer science? Answer: Recursion is the use of an algorithm that calls itself.

Question: What are the three parts of recursion covered in this topic? Answer: The three parts of recursion covered in this topic are understanding recursion, creating recursive algorithms, and analyzing recursive algorithms.

Question: What is the structure of a well-built recursive algorithm that executes a finite number of recursive calls? Answer: A well-built recursive algorithm that executes a finite number of recursive calls includes a base case to stop the

recursion and an input argument that gets closer to the base case with every recursive call.

Question: What is the key takeaway from tracing a recursive algorithm? Answer: Tracing a recursive algorithm can help you understand what task it is performing, even if it is not immediately obvious.

Question: How do recursive algorithms differ from iterative algorithms? Answer: Recursive algorithms call themselves, while iterative algorithms use loops to repeat a set of instructions.

Question: What is the technique used to write recursive algorithms? Answer: The technique used to write recursive algorithms is doing a small part of the job and delegating the rest without caring about the details of how the delegated task is going to be executed.

Question: How is the time complexity of recursive algorithms analyzed? Answer: The time complexity of recursive algorithms is analyzed using recurrence equations.

Question: What is the master theorem? Answer: The master theorem is a method used to analyze the time complexity of recursive algorithms with a specific structure.

# Chapter 6

# Linear data structures

This module covers linear data structures, including linked lists, stacks, and queues, and their implementations using arrays and linked lists. Linked lists allow for non-contiguous memory allocation and access through memory addresses, and the main operations associated with linked lists are insert, delete, and search. Stacks allow only insertion and removal at the top and are useful for verifying balanced pairs of elements and undoing actions. Queues operate on a First-In-First-Out basis, with elements added to the back and removed from the front. Queue operations include enqueue, dequeue, peek, and isempty.

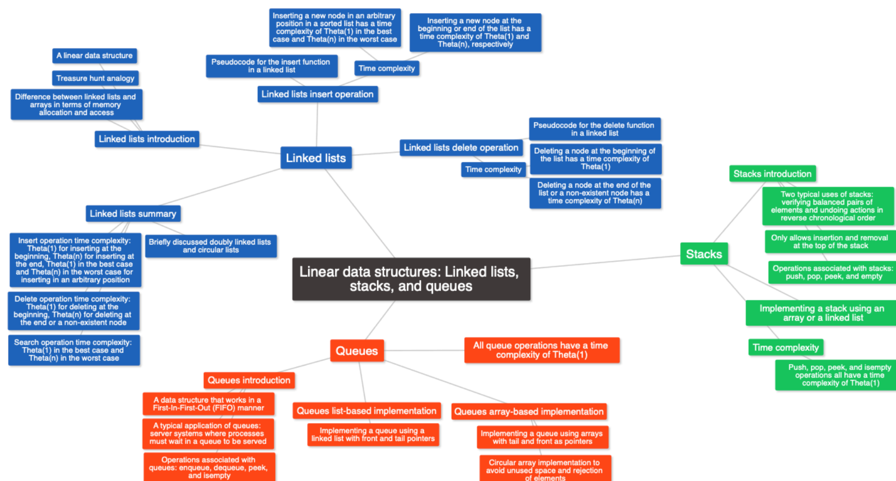



Figure 6.1: Linear Dat Structures Mindmap

## 6.1 Introduction to data structures

In this lecture, the speaker discusses data structures, which are containers that organize data in a specific way. The first half of the module covered algorithms, and this half will focus on data structures. The lecture covers linear data structures such as linked lists, stacks, and queues, as well as non-linear structures such as trees, heaps, and graphs. Each structure has specific operations or functions to manipulate the data stored within it. The lecture emphasizes that data structures and algorithms are closely related and that algorithms are necessary to operate the data stored in a data structure.

## 6.2 Linked lists

### 6.2.1 Linked lists introduction

In this lecture, the speaker introduces the concept of a linked list, a linear data structure in which each element is stored in a separate memory position and is linked to the next element through a memory address. The lecture explains the difference between linked lists and arrays in terms of their memory allocation and access, with linked lists being non-contiguous and accessed through following memory addresses. The lecture walks through the process of creating a linked list, inserting elements, and linking them. The speaker uses the analogy of a treasure hunt to explain how a linked list works. The lecture concludes with a discussion of why linked lists are considered linear data structures, despite their non-linear memory representation, and previews the next lecture, which will cover the main functions associated with linked lists.

### 6.2.2 Linked lists insert operation

The lecture discusses the pseudocode for the insert function in a linked list. The first step is to create a new node with the corresponding data, and then link the new node to the list by making one of the nodes or the head point to the new node, and the new node must point to some node of the list or to null. The pseudocode is shown for inserting a new node at the beginning of the list and for inserting a new node into an empty list. Two other versions of the insert function are also mentioned for inserting a new element at the end of the list and for inserting a new element in an arbitrary position in the list.

### 6.2.3 Linked lists delete operation

In this lecture, the delete function for linked lists is discussed. To delete a node, the node before the one to be deleted is made to point to the node after it. If the node to be deleted is the first node, the head is updated to point to the next node. The pseudocode for the delete function is explained, which involves initializing pointers tmp and prev, checking if the list is empty, and then traversing the list to find and delete the node with a given value. The execution of the pseudocode

is demonstrated step-by-step with an example of deleting node 2 from a list. The lecture also provides practice exercises to simulate executing the delete function for different cases.

### 6.2.4 Linked lists summary

In this lecture, the time complexity of the main operations associated with linked lists was discussed. The time complexity of the insert operation depends on whether the new node is inserted at the beginning, end or at an arbitrary position of the list. If it is inserted at the beginning, the time complexity is Theta 1. If it is inserted at the end, the time complexity is Theta n. If it is inserted at an arbitrary position in a sorted list, the time complexity is Theta 1 in the best case and Theta n in the worst case. The time complexity of the delete operation depends on where the number to be deleted is located in the list. If it is at the beginning, the time complexity is Theta 1, and if it is at the end or not in the list at all, the time complexity is Theta n. The search operation has the same time complexity as using a linear search in an array. The time complexity is Theta 1 in the best case and Theta n in the worst case. Two other types of linked lists, doubly linked lists and circular lists, were briefly discussed. The next lectures will cover stacks and queues.

## 6.3 Stacks

### 6.3.1 Stacks introduction

In this lecture, the second linear data structure, the stack, was introduced. A stack is a data structure that only allows insertion and removal at the top of the stack. The operations associated with the stack are push, pop, peek, and empty. Two typical uses of the stack data structure are verifying balanced pairs of elements, such as parentheses, and undoing actions in reverse chronological order. To implement a stack, an array or a linked list can be used. The linked list implementation is more flexible, while the array implementation has better performance. The lecture ended with a detailed explanation of how to implement a stack using an array.

### 6.3.2 Stacks implementation

In this lecture, we learned about the stack data structure, which works like a physical stack, where objects are inserted and removed from the top. A stack can be implemented using an array or a linked list. When implemented with an array, it may run out of space or have unused memory, but the push, pop, peek, and isempty operations all have a time complexity of Theta(1). When implemented with a linked list, the push, pop, peek, and isempty operations all have a time complexity of Theta(1) as well. We also learned about some common uses for stacks, such as verifying balanced pairs of elements and undoing

actions in inverse chronological order. In the next lecture, we will study the implementation of the queue data structure.

## 6.4 Queues

### 6.4.1 Queues introducion

In this lecture, the queue data structure is introduced, which works in a First-In-First-Out (FIFO) manner, where elements are added to the back and removed from the front. The enqueue operation adds an element to the back of the queue, the dequeue operation removes an element from the front of the queue, the peek operation returns the value of the element at the front of the queue, and the isempty operation returns whether the queue is empty or not. An example of a typical application of the queue data structure is in server systems where processes must wait in a queue to be served. The implementation of the queue data structure using arrays or linked lists is also discussed.

### 6.4.2 Queues array based implementation

In this lecture, the queue data structure was introduced, which works in a FIFO manner, and the operations associated with it were explained, including enqueue, dequeue, peek, and isempty. The implementation of a queue using arrays was demonstrated, where tail and front were used as pointers to add and remove elements. The circular array implementation was introduced to avoid the problem of not being able to use the empty space in the array. The pseudocode for enqueue, dequeue, peek, and isempty were provided, and it was shown that the time complexity of all operations of a queue implemented with arrays is theta one.

### 6.4.3 Queues list based implementation

In this lecture, the implementation of a queue data structure using a linked list is discussed. Unlike an array implementation, a linked list implementation does not have unused allocated memory or the need to reject elements when the queue exceeds a certain size. The front and tail pointers are used to insert elements at the tail and remove elements at the front. The enqueue operation adds a new node at the tail and updates the tail pointer, while the dequeue operation removes the front node and updates the front pointer. The operation peek returns the element at the front of the queue, and the operation isempty returns a Boolean value based on whether the front and tail pointers are null. All queue operations take constant time complexity of Theta one.

# Chapter 7

# Trees

The lecture series covers the concept of trees and hierarchical data structures, binary trees and their implementation, tree traversal techniques like breadth-first and depth-first traversal, binary search trees and their operations, including insert, search, and delete. The lectures also explain the time complexity of each operation and provide pseudocode and examples for implementation. The audience is challenged to think about modifying the algorithms for non-binary trees, maintaining the BST conditions, and balancing binary search trees.

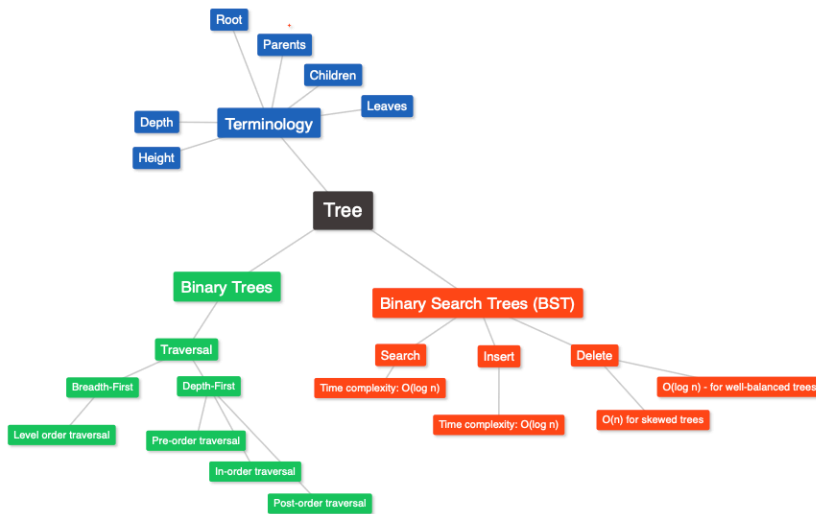



Figure 7.1: Trees Mindmap

The lecture introduces the concept of hierarchical data structures known as

trees, with the root at the top and leaves at the bottom. The nodes in the tree have parents, children, ancestors, and descendants, and each node has a depth and level. The height of the tree is the number of branches from the root to the deepest leaf. Binary trees are a type of tree where each node has at most two children, and they are drawn with the left child to the left of the parent and the right child to the right of the parent. Binary trees are useful for recursive algorithms, and the lecture introduces the topic of binary tree traversal and implementing binary search using binary trees.

## 7.1 Binary trees implementation

The lecture covers two ways of implementing a binary tree: using memory pointers or using arrays. Memory pointers involve defining a node with three memory spaces to store data and the memory addresses of the left and right children. Array implementation involves storing the root node at position 0 of the array and the elements at Level k using 2_k positions, starting at position 2_k minus 1. Memory pointers use more memory but allow for dynamic growth, while arrays use less memory but have a fixed size. The lecture provides examples of adding nodes to a binary tree using memory pointers and introduces the topic of tree traversal in the next lecture.

## 7.2 Binary tree traversal introduction

The lecture explains that traversing a binary tree involves visiting all nodes of the tree and is important for operations like insertion, deletion, and search. There are two main approaches to traversing a tree: breadth-first and depth-first. Breadth-first traversal involves visiting nodes from top to bottom and left to right, while depth-first traversal involves visiting nodes vertically, diving into the tree. Depth-first traversal has three types: pre-order, in-order, and post-order, which determine the order of visiting nodes relative to the root node. Pre-order visits the root node first, in-order visits the root node in the middle, and post-order visits the root node last. The lecture provides an overview of the traversal approaches and introduces the topic of implementing breadth-first and depth-first traversal in the next lectures.

## 7.3 Depth first traversal

In this lecture, the focus was on the implementation of depth-first traversal for binary trees, which includes pre-order, in-order, and post-order traversal. The lecture went through the pseudocode of the pre-order traversal and simulated its execution. It was noted that all three types of traversal are recursive functions and the prefix of their name indicates the order of visiting the root node. The lecture also highlighted the order in which nodes are visited in each type of traversal. Finally, the lecture pointed out that the same recursive structure

applies to the sub-trees. In the next lecture, the implementation of breadth-first traversal will be discussed.

## 7.4 Breadth first traversal

In this lecture, the implementation of breadth-first traversal of a binary tree is studied. The iterative algorithm uses a queue data structure to store the information about nodes to visit. The pseudocode shows how to enqueue nodes, visit them, and store their children information in the queue. The algorithm is executed step by step on a tree as an example. The lecture emphasizes that breadth-first traversal is a horizontal traversal, and it is more comfortable to follow for Western humans, but depth-first traversal is easier to code because it takes advantage of the vertical recursive structure of the tree. The lecture concludes by challenging the audience to think about how to modify the algorithm to work with non-binary trees.

## 7.5 Binary search trees

In this lecture, the use of binary trees for binary searches was discussed. It was explained that binary search is not efficient in a linked list, and that a binary tree is better suited for binary search. A binary search tree (BST) must meet two conditions: all nodes in the left subtree must store numbers that are lesser than the root, and all nodes in the right subtree must store numbers that are greater than the root. The lecture then ended with a discussion of the operations of a binary search tree, including insert, search, and delete, and the challenges of maintaining the BST conditions during these operations.

## 7.6 BST insert

In this lecture, the implementation of the insert operation in a binary search tree was discussed. The pseudocode for the insert function was provided, and the process of inserting nodes was demonstrated step by step. The time complexity of the insert operation was analyzed, and it was determined that the worst-case time complexity is theta(n) when the tree is not balanced. The importance of maintaining the rules of a binary search tree, where all nodes in the left subtree are less than the root and all nodes in the right subtree are greater than the root, was emphasized. Finally, it was mentioned that the other two important operations of a binary search tree, search and delete, will be discussed in future lectures.

## 7.7 BST search

In this lecture, the implementation of the operation search in a binary search tree was discussed. The key idea is that when searching for a number, you take advantage of the fact that all numbers less than the root are stored in the left subtree and all numbers greater than the root are stored in the right subtree. The search function starts by checking whether the number being searched for is in the root. If it is, the function returns true. If it is not, the function recursively searches on either the left or right subtree depending on whether the number being searched for is less than or greater than the root. The time complexity of the search operation is Theta log n, assuming a balanced binary search tree. However, for an unbalanced binary search tree, the worst-case time complexity can be Theta n if all nodes are on one side of the tree. Therefore, for the search operation to have a time complexity of Theta log n, the binary search tree must be balanced.

## 7.8 BST delete

In this lecture, the operation delete in a binary search tree is discussed. The three cases to consider when deleting a node from a binary search tree are explained. Case 1 is when the node to be deleted is a leaf node with no children. Case 2 is when the node to be deleted has only one child. Case 3 is when the node to be deleted has two children. In Case 3, the minimum value in the right subtree or the maximum value in the left subtree is selected and copied into the node to be deleted, and then the node where the maximum or minimum value was found is deleted. The pseudocode for each case is provided, and the structure of the delete function is discussed in detail. The lecture emphasizes that although deleting a node from a binary search tree is a lengthy function to implement, it becomes easier to understand once the cases are known and the pseudocode is familiar.

# Chapter 8

# Heaps

In this series of lectures on heaps, we have learned about various operations related to heaps and how they can be used to implement efficient algorithms.

We started by understanding what heaps are and their properties, including the shape property and the heap property. Then, we learned about the various operations on heaps, including insert, extract-max, max-heapify, and build-max-heap. We learned how to implement these operations in pseudocode and understand their time complexities. We also learned about the in-place technique for building a max-heap from an existing binary tree.

Next, we learned about the heapsort algorithm, which is a sorting algorithm built on the principles of heaps. We learned how to implement heapsort in pseudocode, and how to calculate its time complexity. We saw that the worst-case time complexity of heapsort is N log N, which is the same as mergesort and makes it one of the best comparison performance algorithms.

Finally, we learned about the use of heaps in implementing priority queues, where the items are served based on their allocated priority, and how extracting the maximum value from a heap can make this process efficient.

Overall, these lectures have provided a comprehensive understanding of heaps, their properties, and their various applications in computer science.

In this lecture, the speaker explains the concept of a heap, which is a hierarchical data structure that satisfies two properties: the Heap Property and the Shape Property. The Heap Property refers to the relationship between the values of parent and child nodes, which can be either greater or less than, depending on whether the heap is a max-heap or a min-heap. The Shape Property refers to the shape of the tree, which must be a perfect triangle or a triangle with a left-aligned rectangle in the base. The speaker provides examples of both max-heap and min-heap and mentions that heaps can have a maximum of n children. The speaker emphasizes the importance of these properties in implementing

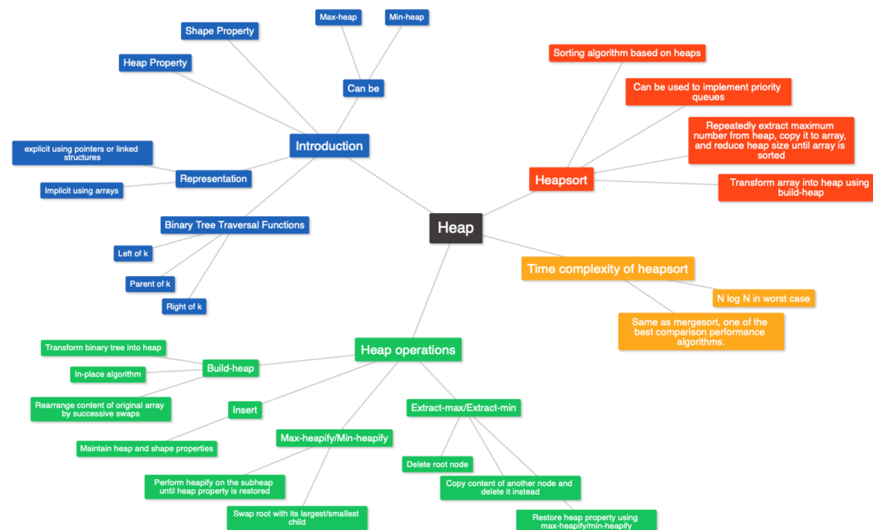



Figure 8.1: Heaps Mindmap

heap operations, such as inserting a new number into the heap. The speaker concludes by stating that the next lecture will cover how to implement heaps.

## 8.1 Implementation

In this lecture, the speaker discusses the implementation of heaps and how they can be represented using arrays, which is an implicit representation. The speaker demonstrates how to move through the heap when implemented using an array and provides algebraic expressions to find the position of a parent and children node in the array. The speaker also introduces three functions - parent of k, left of k, and right of k - that help to move through the implicit heap. The speaker concludes by stating that this knowledge will be useful in the next lecture when discussing the most common operations used to manipulate the data stored in a heap.

## 8.2 Insert, element by element

The lecture covers the implementation of heaps using arrays, particularly the implicit representation, and the movement through the heap by calculating the positions of the parent and children of a node using algebraic expressions. The lecture then focuses on the insertion operation in a max-heap, demonstrating how to maintain the heap and shape properties while inserting numbers. The lecture also provides pseudocode and step-by-step execution of the insertion operation. Lastly, the lecture tasks the listener with identifying the necessary

modifications to transform the function insert for a max-heap to a min-heap and hints about studying the function delete in the next lecture.

## 8.3 Deletion, extract maximum

The lecture discusses the heap operation that deletes the root node, called extract-max for a max heap or extract-min for a min heap. Deleting another node from the heap is not efficient due to the partially sorted nature of the heap. To delete the root node, we copy the content of another node and delete that node instead. After deleting the root node, we need to recover the heap property using the heapify process. The lecture provides pseudocode for the extract-max and max-heapify functions, which use recursion to swap the root with its largest child and perform heapify on the subheap until the heap property is restored. The lecture also mentions a task to write pseudocode for the index largest node function and the extract-mean and mean-heapify functions for a min heap.

## 8.4 Build in place

The lecture explains two ways to transform an existing binary tree into a max-heap, namely using either an out-of-place or an in-place algorithm. The focus is on the latter, which involves rearranging the content of the original array by doing successive swaps. The process starts from the bottom up and involves heapifying subtrees at each level, with the leaves being already heaps. The pseudocode for the in-place algorithm is presented, and a step-by-step simulation of it is demonstrated using an input array to show how it transforms the binary tree into a max-heap.

## 8.5 Heapsort

In this lecture, the HEAPSORT algorithm was introduced as a sorting algorithm built on heaps. The first step is to transform the array into a heap using the build-heap operation in place. Then, the algorithm repeatedly extracts the maximum number from the heap, copies it to the empty position in the array, and reduces the size of the heap until it disappears and the array is sorted. The lecture also mentioned that heaps can be used to implement priority queues, where the next person or process to serve is determined by extracting the maximum number from the heap. Finally, the lecture ended with a note on the time complexity of heapsort, which will be discussed in the next lecture.

## 8.6 Heapsort's complexity

In this lecture, the speaker reviews what has been covered so far in the topic of heaps, including the four heap operations (insert, extract-max, max-heapify, and

build-heap) and the heapsort algorithm. The speaker then examines the time complexity of heapsort and walks through the pseudocode to calculate the time complexity of build-max-heap and extract-max. The worst-case time complexity of heapsort is N log N, which is the same as that of mergesort, making it one of the best comparison performance algorithms.

# Chapter 9

# Graphs

Topic 10 is graphs, including their history, representation, topologies, and operations. The lectures discuss two algorithms, Prim's algorithm and Kruskal's algorithm, used to find the minimum spanning tree of a graph. Additionally, there are lectures explaining Dijkstra's algorithm, used to find the lowest cost route from one node to another in a graph, and finding the shortest path from a start node to an end node. All discussed algorithms are presented with walk throughs and pseudocode.

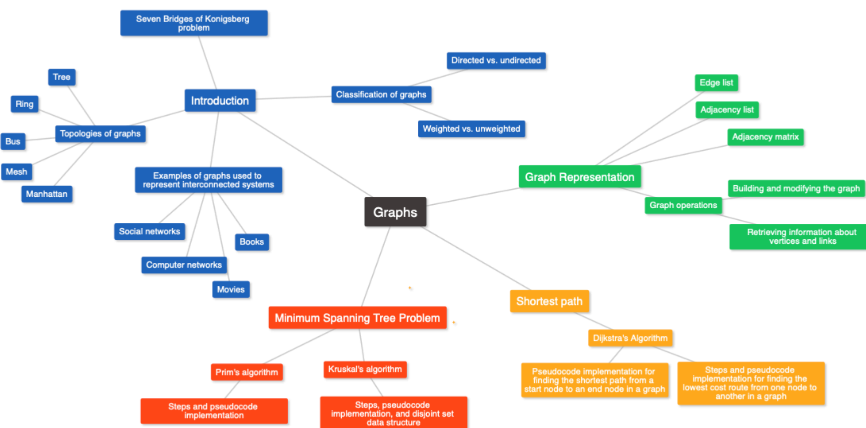



Figure 9.1: Graphs mindmap

## 9.1 Introduction to graphs

The lecture introduces the concept of graphs, a data structure used to represent interconnected systems. The history of the Seven Bridges of Konigsberg problem is presented as an example of how graphs were used to solve a mathematical problem. The transcript then presents various examples of graphs being used to represent different interconnected systems, including books, movies, computer networks, and social networks. The transcript also classifies graphs based on their directed or undirected nature and the weight associated with their links. Finally, the transcript mentions the different topologies of graphs, including the bus, ring, tree, Manhattan, and mesh topologies. The lecture concludes by highlighting that in the next lecture, graphs will be represented in a computer.

## 9.2 Graph representations

This lecture discusses the different ways of representing a graph data structure. Three representations are discussed: the edge list, adjacency matrix, and adjacency list. The edge list representation stores every link in the graph as a triplet; starting node, ending node, and the weight of the link. The adjacency matrix has the same number of rows and columns and contains numbers that represent the presence or absence of links and their weights. The adjacency list is an array of lists where each position represents a node, and the list contains information about outgoing edges. The lecture also covers the most common graph operations, including building and modifying the graph and retrieving information about vertices and links. The way graph operations are implemented depends on the selected graph representation. The lecture concludes by encouraging the audience to think about how to implement graph operations using the different representations.

## 9.3 Minimum spanning tree problem

The minimum spanning tree is a tree that includes all the nodes of a graph using the subset of edges with minimum total weight. This problem is useful in various scenarios, such as in the case of the men's football World Cup where the minimum spanning tree is used to efficiently send data through the network of servers.

Prim's algorithm starts by selecting any node in the graph to start building the tree and identifies all links that connect the nodes in the tree with the nodes not yet in the tree. The link with the minimum cost among them is selected, and the node connected to it is added to the minimum spanning tree. This process is repeated until all nodes are added to the tree.

Kruskal's algorithm starts with V different trees, each made of one node. These trees start to merge until they form a single spanning tree. The merging of the trees is done one step at a time, and at each step, the minimum weight link

connecting two different trees is chosen. This process is repeated until all trees are merged into one.

### 9.3.1 Prims algorithm

In this lecture, the pseudocode implementation of Prim's algorithm for building the minimum spanning tree was discussed. Prim's algorithm starts by initializing the tree with any vertex from the graph and then incrementally constructing the tree by finding the minimum weight link and adding it to the tree, as well as the node connected to that link. The pseudocode implementation includes initializing the minimum spanning tree, finding the set of links that connects nodes in the tree to nodes not yet in the tree, selecting the minimum cost link, and adding it along with the new node to the minimum spanning tree. The lecture also demonstrated the step-by-step execution of the pseudocode with a small graph. The pseudocode is kept general to apply to different representations of graphs. The lecture concludes by mentioning that the next lecture will review the pseudocode of Kruskal's algorithm.

### 9.3.2 Kruskal's algorithm

In this lecture, the pseudocode implementation of Kruskal's algorithm for finding the minimum spanning tree of a graph was discussed. The three main steps of Kruskal's algorithm were outlined, including the initialization of the tree with as many 1-node trees as there are nodes in the graph, sorting the edges of the graph in ascending order of weight, and selecting the next edge to add to the spanning tree by merging two trees with the lowest cost edge. The pseudocode implementation of Kruskal's algorithm was presented, and the disjoint set data structure was briefly introduced. The steps were then demonstrated with a small graph, and the execution was simulated step-by-step.

## 9.4 Shortest path problem

### 9.4.1 Dijkstra's algorithm

The lecture explains Dijkstra's algorithm, a graph algorithm used to find the lowest cost route from one node to another in a graph. The algorithm involves initializing a table and set of unexplored nodes, and then updating the table by analyzing each node and checking if a route passing through that node is better than a previously recorded one. The algorithm continues until all nodes have been explored. The lecture provides an example to demonstrate the algorithm's steps and shows how to read the routing table to obtain the shortest route from one node to another.

The lecture presents a pseudocode version of Dijkstra's algorithm that finds the shortest path from a start node to an end node in a graph. The algorithm

initializes a routing table and a set of unexplored nodes, selects the next unexplored node closest to the source node, calculates the distance from the source node to each of its neighbor nodes, updates distance and route information data if a shorter distance is found, and removes the explored node from the set of unexplored nodes. The input arguments for the pseudocode are the graph G, the start node, and the end node. The algorithm stops as soon as the route from start to end is found and pushes the sequence of previous nodes to a stack to return the route. A min-heap is used as the priority queue to select the next unexplored node with the shortest distance to the start node. The rest of the code processes the unexplored nodes, updates information, and returns the result.

# Part II

# Module goals and objectives

# 1. Choose and justify appropriate data structures and algorithms to solve specific problems

Choosing an appropriate data structure and algorithm to solve a specific problem involves considering several factors, including the problem's size, the input data type, the required output format, and the available computing resources.

Here are some general steps to follow when selecting a data structure and algorithm:

1. Understand the problem: Before selecting a data structure and algorithm, you must fully understand the problem. Determine what inputs the algorithm will receive, what the expected output is, and any constraints on time or space complexity.

2. Identify the input and output data types: The data type of the input and output data will affect the choice of data structure and algorithm. For example, if the input data is in the form of a graph, you may want to use graph algorithms such as breadth-first search or Dijkstra's algorithm.

3. Analyze the time and space complexity: Consider the time and space complexity of different data structures and algorithms. Choose the one with the lowest time and space complexity that can solve the problem.

4. Evaluate trade-offs: Consider the trade-offs between time complexity and space complexity, as well as any other factors that may affect the algorithm's performance, such as cache locality, branch prediction, or parallelism.

5. Test and refine: Test the selected algorithm on various input sizes and types. If the algorithm's performance is not satisfactory, refine the algorithm or choose a different data structure or algorithm.

To justify the chosen data structure and algorithm, you can provide an analysis of its time and space complexity and compare it with other potential solutions. You can also discuss any trade-offs and limitations of the chosen solution and explain why it is the best fit for the specific problem at hand.

# 2. Express time and space complexities of specific algorithms using big-O notation

Big-O notation is a mathematical notation that describes the asymptotic behavior of functions. In computer science, it is commonly used to express the time and space complexity of algorithms.

The basic idea of big-O notation is to represent the worst-case scenario of an algorithm, that is, how the algorithm's time or space requirements grow as the

input size grows. We use "O" to denote the upper bound of the function's growth rate.

For example, let's say we have an algorithm that iterates over a list of n items and performs a constant-time operation on each item. In this case, the time complexity of the algorithm would be O(n), meaning that the algorithm's running time increases linearly with the input size.

Here are some common examples of time complexities expressed using big-O notation:

- O(1) - constant time complexity, where the algorithm takes a constant amount of time to execute regardless of the input size.
- O(log n) - logarithmic time complexity, where the algorithm's running time grows slower than the input size. Common examples of such algorithms include binary search or some divide and conquer algorithms.
- O(n) - linear time complexity, where the algorithm's running time grows linearly with the input size.
- O(n log n) - linearithmic time complexity, where the algorithm's running time is proportional to n times the logarithm of n. Common examples of such algorithms include merge sort or quicksort.
- O(n^2) - quadratic time complexity, where the algorithm's running time is proportional to the square of the input size. Common examples of such algorithms include bubble sort or selection sort.
- O(2^n) - exponential time complexity, where the algorithm's running time doubles with every addition to the input size. This kind of algorithm can quickly become impractical for large input sizes.

Similarly, we can express space complexities using big-O notation. The space complexity of an algorithm is the amount of memory it requires to run as a function of the input size.

For example, let's say we have an algorithm that creates a new list of n items, and the size of the list grows with the input size. In this case, the space complexity of the algorithm would be O(n), meaning that the algorithm's memory requirements grow linearly with the input size.

Understanding and expressing time and space complexities using big-O notation is essential for analyzing and optimizing the performance of algorithms.

# 3. Implement standard searching, sorting and path finding algorithms

Implementing standard searching, sorting, and pathfinding algorithms requires a solid understanding of the algorithms and proficiency in a programming language of your choice. Here are the general steps to implement these algorithms:

1. Choose a programming language: Select a programming language that is

appropriate for the problem and algorithm you are implementing. Most commonly used programming languages for implementing algorithms include C++, Java, Python, and JavaScript.

2. Understand the algorithm: Before implementing any algorithm, ensure that you have a complete understanding of how the algorithm works, what its time and space complexities are, and its inputs and outputs.

3. Write the code: Once you understand the algorithm, start writing the code. Begin with a high-level overview of the algorithm and then break it down into smaller, more manageable functions.

4. Test the code: After writing the code, test it with various input sizes and types to ensure that it works as expected. Use a variety of test cases to test the algorithm's performance, such as edge cases or worst-case scenarios.

5. Optimize the code: Once the algorithm is functioning correctly, look for opportunities to optimize the code. This might include using more efficient data structures, reducing the number of operations, or improving cache locality.

Here are some specific steps for implementing common algorithms:

- **Searching algorithms**: Standard searching algorithms include linear search and binary search. Linear search scans the entire list sequentially to find the target element, while binary search only works for sorted lists and repeatedly divides the search interval in half. To implement these algorithms, use loops to iterate over the list and compare the target element to each element in the list.

- **Sorting algorithms**: Standard sorting algorithms include bubble sort, selection sort, insertion sort, merge sort, and quicksort. To implement these algorithms, use loops and comparisons to sort the list of elements according to some criteria. It is often helpful to use recursion to simplify the sorting algorithm's implementation.

- **Pathfinding algorithms**: Standard pathfinding algorithms include breadth-first search, depth-first search, and Dijkstra's algorithm. These algorithms find the shortest path between two points in a graph or network. To implement these algorithms, represent the graph as an adjacency list or matrix, and use loops and comparisons to traverse the graph and find the shortest path.

Remember, each algorithm has its strengths and weaknesses, and the best algorithm to use will depend on the specific problem and constraints you are working with.

# 4. Implement different data structures, and describe the consequences of particular implementation choices

Implementing data structures involves creating classes or structures that define how data is organized, stored, and accessed. Choosing the right implementation of a data structure is essential for efficient algorithmic performance. Here are the general steps to implement data structures:

1. Choose a programming language: Select a programming language that is appropriate for the data structure and problem you are implementing. Most commonly used programming languages for implementing data structures include C++, Java, Python, and JavaScript.

2. Choose a data structure: Choose a data structure that is appropriate for the problem you are trying to solve. There are many different types of data structures, including arrays, linked lists, stacks, queues, trees, and graphs.

3. Define the data structure: Define the data structure as a class or structure in your chosen programming language. Include member variables to store the data, and member functions to manipulate the data and access it.

4. Implement the data structure: Write the code for the member functions of the data structure. The implementation should be efficient and easy to use. Use best practices for your chosen programming language, such as memory management and error handling.

5. Test the data structure: After implementing the data structure, test it with various input sizes and types to ensure that it works as expected. Use a variety of test cases to test the data structure's performance, such as edge cases or worst-case scenarios.

Here are some specific considerations for implementing different data structures:

- **Arrays**: Arrays are a simple data structure that can be used to store a fixed number of elements of the same data type. Arrays are fast for random access and sequential iteration, but resizing an array can be expensive.

- **Linked lists**: Linked lists are a dynamic data structure that can be used to store a variable number of elements of the same or different data types. Linked lists are slow for random access, but they are fast for insertion and deletion operations.

- **Stacks and Queues**: Stacks and queues are abstract data types that can be implemented using arrays or linked lists. Stacks use a last-in, first-out (LIFO) ordering, while queues use a first-in, first-out (FIFO) ordering. Stacks are useful for undo/redo operations or function call management, while queues are useful for scheduling or buffer management.

- **Trees**: Trees are a hierarchical data structure that can be used to store and organize data in a way that supports efficient searching and sorting. Trees can be implemented using pointers or arrays. Binary trees are the most common type of tree, but other types include AVL trees, red-black trees, and B-trees.
- **Graphs**: Graphs are a flexible data structure that can be used to represent a wide range of data relationships. Graphs can be implemented using adjacency matrices or adjacency lists. Adjacency matrices are fast for random access, but they require a lot of memory. Adjacency lists are more memory-efficient but slower for random access.

Choosing the right data structure and implementation depends on the specific problem you are trying to solve and the trade-offs between performance and memory requirements. It's essential to understand these trade-offs to choose the most effective implementation of a data structure.

# 5. Compare and contrast recursive and iterative expressions of solutions to problems

Recursive and iterative solutions to problems are two common approaches used in programming to solve problems. Recursive solutions break down problems into smaller sub-problems and solve them using a recursive function, while iterative solutions use loops and iterations to solve the problem.

Here are some ways to compare and contrast recursive and iterative solutions:

- **Conceptual differences**: Recursive solutions break down a problem into smaller sub-problems, while iterative solutions use loops to repeat a process until a specific condition is met. Algorithmic complexity: Recursive solutions tend to have higher algorithmic complexity than iterative solutions because of the overhead of the function call stack. This can make recursive solutions slower and consume more memory.
- **Code readability**: Recursive solutions can be more readable and easier to understand, especially when the problem is naturally recursive. Iterative solutions, on the other hand, can be more verbose and difficult to understand, especially when the problem requires complex loops or branching.
- **Stack overflow**: Recursive solutions are prone to stack overflow errors if the function call stack becomes too deep, while iterative solutions are less likely to encounter this problem.
- **Tail recursion optimization**: In some programming languages, tail recursion optimization can make recursive solutions as efficient as iterative solutions by reusing the same stack frame for each recursive call.

In general, recursive solutions are suitable for problems that can be expressed in terms of smaller sub-problems, and where the cost of recursion is outweighed by the benefits of readability and simplicity. Iterative solutions are suitable for problems that require more complex logic or when the algorithm's complexity is a significant concern.

Choosing the right approach depends on the specific problem and its constraints. Experienced programmers often use a combination of both approaches to create efficient and maintainable code.

## 6. Describe the abstraction of collections, relate this abstraction to linear collections, and recall the basic operations that each abstraction supports

The abstraction of collections refers to a way of organizing and storing data in a structured manner, allowing for easy access, manipulation, and iteration. The collection abstraction is the foundation of many data structures used in programming, such as arrays, linked lists, stacks, queues, trees, and graphs.

Linear collections are one type of collection abstraction that store data in a linear sequence, where each element is assigned a unique index or position. The basic operations that linear collections support include:

- **Accessing elements**: Elements in a linear collection can be accessed using their index or position in the sequence. This operation has a constant time complexity (O(1)) for arrays and a linear time complexity (O(n)) for linked lists.
- **Inserting elements**: Elements can be inserted into a linear collection at a specified index or position, causing subsequent elements to shift. This operation has a linear time complexity (O(n)) for both arrays and linked lists.
- **Deleting elements**: Elements can be removed from a linear collection at a specified index or position, causing subsequent elements to shift. This operation has a linear time complexity (O(n)) for both arrays and linked lists.
- **Searching elements**: Elements in a linear collection can be searched for by iterating over the collection and comparing each element to the target value. This operation has a linear time complexity (O(n)) for both arrays and linked lists.
- **Iterating elements**: Elements in a linear collection can be iterated over by using loops or iterators to access each element in the sequence. This

operation has a linear time complexity (O(n)) for both arrays and linked lists.

Other collection abstractions include non-linear collections, such as trees and graphs, which organize data in hierarchical or network structures. The basic operations for non-linear collections depend on their specific structure and can include traversal, insertion, deletion, and searching.

Understanding the abstraction of collections and their basic operations is essential for selecting the most appropriate data structure for a given problem and optimizing algorithmic performance.

# Chapter 10

# Analysis of Algorithms

## 10.1 Theoretical vs. empirical algorithm analysis

Theoretical analysis and empirical analysis are two key concepts used to evaluate an algorithm's performance.

Theoretical analysis involves predicting an algorithm's time and space complexity based on its design and structure. The analysis uses mathematical models and tools to express an algorithm's complexity as a function of the input size. Big-O notation is commonly used to describe the worst-case time complexity of an algorithm, while big-Omega and big-Theta notations can be used to describe the best-case and average-case time complexity of an algorithm, respectively. Theoretical analysis provides bounds on an algorithm's performance, which helps estimate how the algorithm will behave as the input size grows larger.

Empirical analysis, on the other hand, involves measuring an algorithm's actual performance on real-world data. This type of analysis involves implementing the algorithm and running it on a variety of inputs with different sizes and properties, and then measuring the algorithm's execution time and space usage. Empirical analysis provides concrete data on how an algorithm performs in practice, which can help programmers validate and refine the algorithm's design, identify performance bottlenecks, and optimize the algorithm's performance.

The key concept of theoretical vs. empirical algorithm analysis is that theoretical analysis provides an upper bound on an algorithm's performance, while empirical analysis measures the actual performance of the algorithm. Theoretical analysis is useful for comparing different algorithms and selecting the most appropriate one for a given problem. Empirical analysis is useful for verifying the theoretical analysis, identifying real-world performance bottlenecks, and tuning

the algorithm for optimal performance.

Both theoretical and empirical analysis are important for evaluating algorithm performance. Theoretical analysis provides insights into the algorithm's design and behavior, while empirical analysis provides concrete data on how the algorithm performs in practice. By combining theoretical and empirical analysis, programmers can design and optimize algorithms that provide the best performance for a given problem and input size.

## 10.2 Memory and time requirements from pseudocode

Memory and time requirements refer to the amount of memory and time needed by an algorithm to execute successfully. These requirements are important because they determine how much space and time an algorithm needs to solve a particular problem, which in turn affects the efficiency and scalability of the algorithm.

Pseudocode is a high-level description of an algorithm that uses natural language and programming constructs to express the algorithm's logic. Pseudocode is often used to describe algorithms because it is easier to read and understand than programming code, while still providing a precise and unambiguous specification of the algorithm's behavior.

To analyze an algorithm's memory and time requirements from pseudocode, we need to evaluate the algorithm's code and logic, and determine its memory and time requirements. This process involves several steps:

1. **Analyzing the code and logic**: The pseudocode should be carefully analyzed to determine the algorithm's memory and time requirements. This involves identifying the data structures used by the algorithm and their memory requirements, as well as the number of operations performed by the algorithm and their time requirements.

2. **Determining memory requirements**: Memory requirements can be determined by analyzing the data structures used by the algorithm and their memory usage patterns. This involves identifying the total amount of memory needed to store the data structures, as well as any additional memory needed for intermediate computations.

3. **Determining time requirements**: Time requirements can be determined by analyzing the number of operations performed by the algorithm and their time complexity. This involves identifying the worst-case, best-case, or average-case running time of the algorithm, and expressing this time complexity using big-O notation.

Once the memory and time requirements have been determined, we can use this information to analyze the efficiency and scalability of the algorithm. For

example, if an algorithm has a high memory requirement, it may not be suitable for systems with limited memory, or if the algorithm has a high time complexity, it may not be suitable for real-time applications or other time-sensitive tasks.

Memory and time requirements are important concepts in algorithm analysis, and analyzing them from pseudocode involves evaluating the algorithm's code and logic, and determining its memory and time requirements. This information can be used to analyze the efficiency and scalability of the algorithm and identify potential performance bottlenecks.

## 10.3 Growth function and asymptotic notation

The growth function and asymptotic notation are key concepts used in the analysis of algorithms to describe how the time or space requirements of an algorithm grow as the size of the input increases.

The growth function of an algorithm describes the rate at which the time or space requirements of the algorithm increase as the input size grows. This growth function can be expressed using mathematical notation, such as a polynomial or exponential function, to represent the algorithm's time or space complexity.

Asymptotic notation is a way to express the growth rate of an algorithm's time or space complexity, using a notation such as big-O, big-Omega, or big-Theta. These notations provide a way to describe the upper, lower, and tight bounds of an algorithm's time or space complexity as the input size grows to infinity.

The big-O notation represents the upper bound of an algorithm's time or space complexity, describing how the time or space requirements of the algorithm grow no faster than a specific function. For example, an algorithm with a time complexity of $O(n)$ would take at most linear time to execute, as the size of the input grows.

The big-Omega notation represents the lower bound of an algorithm's time or space complexity, describing how the time or space requirements of the algorithm grow no slower than a specific function. For example, an algorithm with a time complexity of $\Omega(n)$ would take at least linear time to execute, as the size of the input grows.

The big-Theta notation represents the tight bound of an algorithm's time or space complexity, describing how the time or space requirements of the algorithm grow at the same rate as a specific function. For example, an algorithm with a time complexity of $\Theta(n)$ would take linear time to execute, as the size of the input grows.

Using the growth function and asymptotic notation, we can compare and analyze the time and space requirements of different algorithms and select the most appropriate algorithm for a given problem and input size. We can also

optimize algorithms by reducing their time or space complexity and improving their performance for large input sizes.

In conclusion, the growth function and asymptotic notation are key concepts in algorithm analysis, providing a way to describe and compare the time and space requirements of different algorithms as the size of the input grows. By understanding these concepts, we can select and optimize algorithms to provide the best performance for a given problem and input size.

# Chapter 11

# Learning outcomes

## 11.1 Determine time and memory consumption of an algorithm described using pseudocode

Here is some step-by-step guidance on how to analyze the time and memory complexity of an algorithm described using pseudocode:

1. **Identify the main operations in the pseudocode**: To analyze the time complexity of the algorithm, you need to identify the main operations that are executed by the algorithm in the pseudocode. Examples of operations include arithmetic operations, comparisons, loops, and function calls.

2. **Assign a time cost to each operation**: For each main operation, you need to assign a time cost that represents the number of basic operations needed to execute that operation. For example, arithmetic operations and comparisons typically have a time cost of 1, while loops have a time cost that depends on the number of iterations.

3. **Express the time complexity using asymptotic notation**: Once you have assigned a time cost to each operation, you can calculate the total time complexity of the algorithm by summing the time costs of all the main operations. You should then express the time complexity in terms of big-O notation, which provides an upper bound on the time required to execute the algorithm as a function of the input size.

4. **Identify the data structures used in the pseudocode**: To analyze the memory complexity of the algorithm, you need to identify the data structures that are used in the pseudocode, such as arrays, linked lists, and trees.

5. **Assign a memory cost to each data structure**: For each data structure, you need to assign a memory cost that represents the number of memory units needed to store the data structure. For example, an array of n elements typically requires n memory units.

6. **Express the memory complexity using asymptotic notation**: Once you have assigned a memory cost to each data structure, you can calculate the total memory complexity of the algorithm by summing the memory costs of all the data structures used in the algorithm. You should then express the memory complexity in terms of big-O, big-Omega or big-Theta notation, which provides bounds on the memory required to execute the algorithm as a function of the input size.

7. **Verify the time and memory complexity by testing the algorithm**: To verify the time and memory complexity of the algorithm, you should test the algorithm on different input sizes and measure its actual running time and memory usage. You can then compare the actual running time and memory usage to the theoretical time and memory complexity calculated in steps 3 and 6 to ensure that they are consistent.

By following these steps, you can analyze the time and memory complexity of an algorithm described using pseudocode. This information can help you understand the performance characteristics of the algorithm and select the most appropriate algorithm for a given problem and input size. You can also optimize algorithms by reducing their time and memory complexity and improving their performance for large input sizes.

## 11.2 Determine the growth function of the running time or memory consumption of an algorithm

Follow the steps above, then:

8. **Simplify the growth function(s)**: The growth function(s) may be simplified by removing lower-order terms or constant factors, as these become negligible as the input size grows. The simplified growth function provides a more concise representation of the algorithm's time or memory complexity.

BThis information can help us compare and analyze the efficiency and scalability of different algorithms and select the most appropriate algorithm for a given problem and input size. We can also optimize algorithms by reducing their time or memory complexity and improving their performance for large input sizes.

## 11.3 Use big-O, big-Omega and big-Theta notations to describe the running time or memory consumption of an algorithm

We can use big-O, big-Omega, and big-Theta notations to describe the running time or memory consumption of an algorithm. These notations provide a way to describe the upper, lower, and tight bounds of the algorithm's time or memory complexity as the input size grows to infinity.

Here are the ways to use these notations to describe the running time or memory consumption of an algorithm:

1. **Big-O notation**: We use big-O notation to describe the upper bound of the algorithm's running time or memory consumption. We express the growth rate of the algorithm's time or memory complexity using O(f(n)), where f(n) is a mathematical function that represents the upper bound of the algorithm's time or memory requirements. For example, if an algorithm takes at most n^2 operations to execute, we can express its time complexity as O(n^2).

2. **Big-Omega notation**: We use big-Omega notation to describe the lower bound of the algorithm's running time or memory consumption. We express the growth rate of the algorithm's time or memory complexity using Ω(f(n)), where f(n) is a mathematical function that represents the lower bound of the algorithm's time or memory requirements. For example, if an algorithm takes at least n^2 operations to execute, we can express its time complexity as Ω(n^2).

3. **Big-Theta notation**: We use big-Theta notation to describe the tight bound of the algorithm's running time or memory consumption. We express the growth rate of the algorithm's time or memory complexity using Θ(f(n)), where f(n) is a mathematical function that represents the tight bound of the algorithm's time or memory requirements. For example, if an algorithm takes exactly n^2 operations to execute, we can express its time complexity as Θ(n^2).

By using these notations to describe the running time or memory consumption of an algorithm, we can compare and analyze the efficiency and scalability of different algorithms and select the most appropriate algorithm for a given problem and input size. We can also optimize algorithms by reducing their time or memory complexity and improving their performance for large input sizes.

# Chapter 12

# Recursive algorithms

## 12.1 Algorithmic recursion

Algorithmic recursion refers to the process of defining a problem or solution in terms of a smaller version of itself. In other words, a function or subroutine is designed to call itself repeatedly until a base case is reached. This is known as a recursive function or a recursive algorithm.

The basic idea behind algorithmic recursion is to break down a complex problem into simpler subproblems and solve them recursively. Each recursive call solves a smaller instance of the same problem until the base case is reached, at which point the solution is returned.

A classic example of a recursive algorithm is the factorial function, which calculates the product of all positive integers up to a given number. The factorial of a number n is defined as n * (n-1) * (n-2) * ... * 1. The recursive implementation of the factorial function is straightforward: if n is 0 or 1, the function returns 1; otherwise, it multiplies n by the factorial of (n-1).

Recursive algorithms are widely used in computer science, especially in data structures and algorithms, and can be very efficient in solving problems that can be broken down into smaller subproblems. However, they can also be tricky to implement correctly and efficiently, and may require careful management of memory usage and stack overflow.

## 12.2 The structure of recursive algorithms

The structure of a recursive algorithm typically consists of two parts: the base case and the recursive case.

- **Base case**: This is the simplest form of the problem that can be solved

without recursion. It is the termination condition that stops the recursion from continuing indefinitely. The base case is important because without it, the recursive algorithm would run forever.

- **Recursive case**: This is the more complex form of the problem that can be broken down into simpler subproblems. In the recursive case, the problem is solved by recursively calling the same function with smaller instances of the problem until the base case is reached.

In general, a recursive algorithm follows the following steps:

1. Check if the current problem instance is a base case. If it is, return the solution directly.
2. If the current problem instance is not a base case, break it down into smaller subproblems.
3. Solve each subproblem recursively by calling the same function on the subproblem.
4. Combine the solutions of the subproblems to get the solution to the original problem.
5. Return the final solution.

It's important to note that recursive algorithms can have multiple base cases and multiple recursive cases depending on the complexity of the problem. In addition, care must be taken to ensure that the recursive algorithm terminates for all possible inputs.

## 12.3 Recurrence equations

Recurrence equations, also known as recurrence relations, are equations that describe a sequence of numbers in terms of one or more of the preceding terms in the sequence. They are often used in computer science and mathematics to describe the time or space complexity of recursive algorithms.

In the context of recursive algorithms, recurrence equations are used to analyze the performance of the algorithm and determine its time or space complexity. The recurrence equation expresses the time or space required to solve a problem of size n in terms of the time or space required to solve smaller subproblems.

For example, let's consider the recursive implementation of the Fibonacci sequence:

```
fibonacci(n):
  if n == 0 or n == 1:
    return n
  else:
```

```
    return fibonacci(n-1) + fibonacci(n-2)
```

The recurrence equation for the time complexity of this algorithm can be expressed as follows:

```
T(n) = T(n-1) + T(n-2) + O(1)
```

where T(n) is the time required to compute the nth Fibonacci number, and O(1) represents the constant time required for the base case.

Solving the recurrence equation involves finding a closed-form expression for T(n) in terms of n. In this case, the solution is O(2^n), which means that the time required to compute the nth Fibonacci number grows exponentially with n.

Recurrence equations can be useful for understanding the performance of recursive algorithms and for comparing different algorithms for the same problem. However, solving them can be challenging and often requires advanced mathematical techniques such as generating functions or the master theorem.

## 12.4 Master Theorem

The master theorem is a mathematical tool used to analyze the time complexity of divide-and-conquer algorithms, which are a common type of recursive algorithm. It provides a formula for the time complexity of such algorithms in terms of their recursion depth and the size of their subproblems.

The master theorem has three cases, depending on the relative sizes of the subproblems and the amount of work done in combining their solutions:

1. If the size of the subproblems is a constant fraction of the original problem size, and the amount of work done in combining their solutions is also constant, then the time complexity is given by a simple formula: T(n) = a * T(n/b) + O(n^d), where a is the number of subproblems, b is the size of the subproblems, and d is the amount of work done in combining their solutions. The time complexity is then given by:

- If d < log_b(a), then T(n) = O(n^log_b(a)).
- If d = log_b(a), then T(n) = O(n^d * log(n)).
- If d > log_b(a), then T(n) = O(n^d).

2. If the size of the subproblems is a constant fraction of the original problem size, but the amount of work done in combining their solutions is greater than constant, then the time complexity is given by T(n) = a * T(n/b) + O(n^d), where a, b, and d are as above, but d > log_b(a). The time complexity is then given by T(n) = O(n^d).

3. If the size of the subproblems decreases faster than a constant factor of the original problem size, then the time complexity is dominated by the cost of the leaves of the recursion tree, and is given by $T(n) = O(f(n))$, where $f(n)$ is the cost of the leaves of the recursion tree.

The master theorem is a powerful tool for analyzing the time complexity of divide-and-conquer algorithms, and can simplify the process of determining their time complexity. However, it is important to note that the master theorem only applies to a specific class of recursive algorithms and may not be applicable to all types of recursive algorithms.

# Chapter 13

# Learning outcomes

## 13.1 Trace and write recursive algorithms

Tracing and writing recursive algorithms can be a bit challenging, but there are some general steps you can follow to help you understand and write recursive algorithms:

1. **Identify the base case**: The first step in writing a recursive algorithm is to identify the base case, which is the simplest form of the problem that can be solved without recursion. The base case is important because it stops the recursion from continuing indefinitely.

2. **Define the recursive case**: Once you have identified the base case, you need to define the recursive case, which is the more complex form of the problem that can be broken down into simpler subproblems. In the recursive case, the problem is solved by recursively calling the same function with smaller instances of the problem until the base case is reached.

3. **Write the function signature**: After you have identified the base case and recursive case, you can write the function signature, which includes the function name, the input parameters, and the return type. Write the function body: With the function signature in place, you can now write the function body. The function body should start by checking whether the current problem instance is a base case. If it is, the function should return the solution directly. If not, the function should break down the problem into smaller subproblems and solve each subproblem recursively by calling the same function on the subproblem.

4. **Combine the solutions**: Once the recursive calls have returned their solutions, the final step is to combine the solutions of the subproblems to get the solution to the original problem. This may involve some additional processing or computation, depending on the nature of the problem.

5. **Test the algorithm**: Finally, it is important to test the recursive algorithm with a range of inputs to ensure that it produces correct and expected results.

When tracing recursive algorithms, it can be helpful to draw a recursion tree to visualize the flow of execution and the order in which the recursive calls are made. This can also be useful for identifying potential problems with the algorithm, such as infinite recursion or redundant computations.

## 13.2 Write the recursive version of an iterative algorithm using pseudocode

Here's an example of an iterative algorithm that computes the sum of the elements in an array:

```
// Iterative algorithm for computing the sum of an array
sum_array(array):
    sum = 0
    for element in array:
        sum = sum + element
    return sum
```

To convert this iterative algorithm into recursive form, we can use a recursive helper function that takes the current index of the array as a parameter and computes the sum of the remaining elements in the array. The base case occurs when the current index is greater than or equal to the length of the array, at which point the function returns 0. The recursive case involves computing the sum of the current element and the sum of the remaining elements using a recursive call to the same function with the next index as a parameter. The sum of the array is then given by the sum of the current element and the sum of the remaining elements.

Here's the recursive version of the algorithm:

```
// Recursive algorithm for computing the sum of an array
sum_array_helper(array, index):
    if index >= len(array):
        return 0
    else:
        return array[index] + sum_array_helper(array, index + 1)

sum_array(array):
    return sum_array_helper(array, 0)
```

In this recursive algorithm, `sum_array_helper` is the recursive helper function

that takes an array and an index as input and computes the sum of the remaining elements in the array. The `sum_array` function is the main recursive function that calls `sum_array_helper` with an initial index of 0.

When converting an iterative algorithm to a recursive algorithm, the key is to identify the base case and the recursive case, just as we did in this example. In general, the base case should be the simplest form of the problem that can be solved directly, while the recursive case should be a more complex form of the problem that can be broken down into smaller subproblems. Once you have identified the base case and the recursive case, you can write a recursive helper function that calls itself with smaller instances of the problem until the base case is reached.

It's worth noting that not all iterative algorithms can be easily converted to recursive form, and in some cases the resulting recursive algorithm may be less efficient or more difficult to implement. However, for certain types of problems, the recursive approach can offer a more elegant and intuitive solution.

I hope this clarifies how to convert an iterative algorithm to recursive form. Please let me know if you have any further questions or concerns.

## 13.3 Calculate the time complexity of recursive algorithms

Calculating the time complexity of recursive algorithms can be more challenging than for iterative algorithms, since the time complexity depends on the number of recursive calls and the size of the subproblems at each level of the recursion. However, there are some general steps you can follow to calculate the time complexity of recursive algorithms:

1. **Identify the base case**: The first step is to identify the base case, which is the simplest form of the problem that can be solved directly. The base case is important because it stops the recursion from continuing indefinitely.

2. **Determine the number of recursive calls**: Once you have identified the base case, you need to determine the number of recursive calls made by the algorithm at each level of the recursion. This depends on the size of the subproblems being solved and the specific algorithm being used.

3. **Analyze the time complexity of each recursive call**: For each recursive call, you need to analyze the time complexity of the operations being performed. This may involve analyzing loops, conditional statements, and other operations.

4. **Combine the time complexity of each level of the recursion**: Once you have analyzed the time complexity of each recursive call, you need to combine the time complexity of each level of the recursion. This may

involve using recurrence relations or other mathematical tools to compute the total time complexity.

5. **Simplify the time complexity if possible**: Finally, you may be able to simplify the time complexity if there is a pattern or formula that describes the time complexity of each level of the recursion.

It's worth repeating that calculating the time complexity of recursive algorithms can be more challenging than for iterative algorithms, since the time complexity can depend on many factors, including the number of recursive calls, the size of the subproblems, and the specific operations being performed. In some cases, it may be necessary to use more advanced mathematical tools, such as the master theorem, to analyze the time complexity of recursive algorithms.

# Chapter 14

# Comparison sorting algorithms

## 14.1 Comparison vs. non comparison sorts

In the context of sorting algorithms, comparison sorts are algorithms that sort a sequence of elements by comparing pairs of elements and swapping them if they are in the wrong order, until the sequence is sorted. Comparison sorts are generally based on comparison-based sorting algorithms, such as merge sort, quicksort, and heapsort.

On the other hand, non-comparison sorts are algorithms that sort a sequence of elements without comparing pairs of elements. Non-comparison sorts are generally based on specialized sorting algorithms, such as counting sort, radix sort, and bucket sort.

Comparison sorts have a lower theoretical time complexity of O(nlogn) for the worst-case scenario. This is due to the fact that any comparison-based algorithm must compare a pair of elements at least once in order to determine their relative order, and the number of such comparisons that need to be made to sort a sequence of n elements grows at least logarithmically with n.

Non-comparison sorts can achieve linear time complexity for the average or worst-case scenario, which is faster than comparison sorts. However, non-comparison sorts are generally less versatile and have more restrictive assumptions about the nature of the input sequence, such as that the elements be integers within a certain range.

In practice, the choice between comparison and non-comparison sorts depends on the specific requirements of the problem being solved, such as the size of the input sequence, the range of possible values, and the desired level of efficiency

and stability.

## 14.2 Bubble, Insertion and Selection sort

Bubble sort, insertion sort, and selection sort are three simple and widely used sorting algorithms. Bubble sort repeatedly compares adjacent elements and swaps them if they are in the wrong order until the entire list is sorted. Insertion sort builds the final sorted list by iterating through the list and inserting each unsorted element into its correct position in the sorted portion of the list. Selection sort selects the smallest unsorted element and swaps it with the leftmost unsorted element, moving the boundary of the sorted portion of the list one element to the right. While these sorting algorithms are easy to implement and understand, they generally have a worst-case time complexity of O(n^2), making them less efficient than more advanced sorting algorithms for large or complex lists.

### 14.2.1 Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements in a list and swaps them if they are in the wrong order. The algorithm iterates through the list until no more swaps are needed, indicating that the list is sorted. Bubble sort has a worst-case and average time complexity of O(n^2), where n is the number of elements in the list.

Here is the pseudocode for Bubble Sort in iterative form:

```
// Iterative Bubble Sort algorithm
bubble_sort(array):
    n = length(array)
    for i from 0 to n-1:
        for j from 0 to n-i-1:
            if array[j] > array[j+1]:
                swap(array[j], array[j+1])
```

It is also possible to implement bubble sort recursively.

Here is the pseudocode for Bubble Sort in recursive form:

```
// Recursive Bubble Sort algorithm
bubble_sort(array, n):
    if n == 1:
        return

    for i from 0 to n-1:
        if array[i] > array[i+1]:
```

```
            swap(array[i], array[i+1])

    bubble_sort(array, n-1)
```

In this recursive algorithm, the base case occurs when n is equal to 1, indicating that the array is already sorted. Otherwise, the algorithm iterates through the array and swaps adjacent elements if they are in the wrong order. The algorithm then makes a recursive call to bubble_sort with n-1, which reduces the size of the array by 1 and repeats the process until the entire array is sorted.

Note that the recursive version of Bubble Sort has a worst-case time complexity of O(n^2), the same as the iterative version, since both implementations make the same number of comparisons and swaps.

I hope this helps! Let me know if you have any further questions.

### 14.2.2 Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted list one element at a time. The algorithm iterates through the list and repeatedly inserts each unsorted element into its correct position in the sorted portion of the list. The sorted portion of the list grows until all elements are sorted. Insertion sort has a worst-case and average time complexity of O(n^2), but can perform better than other quadratic sorting algorithms on small or nearly sorted lists.

Here is the pseudocode for Insertion Sort in iterative form:

```
// Iterative Insertion Sort algorithm
insertion_sort(array):
    n = length(array)
    for i from 1 to n-1:
        key = array[i]
        j = i-1
        while j >= 0 and array[j] > key:
            array[j+1] = array[j]
            j = j-1
        array[j+1] = key
```

Here is the pseudocode for Insertion Sort in recursive form:

```
// Recursive Insertion Sort algorithm
insertion_sort(array, n):
    if n <= 1:
        return
```

```
    insertion_sort(array, n-1)

    key = array[n-1]
    j = n-2

    while j >= 0 and array[j] > key:
        array[j+1] = array[j]
        j = j-1

    array[j+1] = key
```

In this recursive algorithm, the base case occurs when n is less than or equal to 1, indicating that the array is already sorted. Otherwise, the algorithm makes a recursive call to insertion_sort with n-1, which sorts the first n-1 elements of the array. The algorithm then places the nth element in its correct position in the sorted subarray by iterating through the subarray and shifting elements to the right until the correct position is found.

Note that the recursive version of Insertion Sort has a worst-case time complexity of O(n^2), the same as the iterative version, since both implementations make the same number of comparisons and swaps.

### 14.2.3 Selection Sort

Selection sort is a simple sorting algorithm that repeatedly selects the smallest unsorted element and swaps it with the leftmost unsorted element, moving the boundary of the sorted portion of the list one element to the right. The algorithm iterates through the list until the entire list is sorted. Selection sort has a worst-case and average time complexity of O(n^2), and is generally less efficient than other quadratic sorting algorithms.

Here is the pseudocode for Selection Sort in iterative form:

```
// Iterative Selection Sort algorithm
selection_sort(array):
    n = length(array)
    for i from 0 to n-1:
        min_idx = i
        for j from i+1 to n-1:
            if array[j] < array[min_idx]:
                min_idx = j
        swap(array[i], array[min_idx])
```

And here is the pseudocode for Selection Sort in recursive form:

```
// Recursive Selection Sort algorithm
selection_sort(array, start_idx):
    n = length(array)
    if start_idx >= n-1:
        return

    min_idx = start_idx
    for i from start_idx+1 to n-1:
        if array[i] < array[min_idx]:
            min_idx = i

    swap(array[start_idx], array[min_idx])
    selection_sort(array, start_idx+1)
```

In this recursive algorithm, the base case occurs when the start index is greater than or equal to n-1, indicating that the array is already sorted. Otherwise, the algorithm selects the smallest element in the unsorted portion of the array by iterating through the array from the start index and comparing each element to the current minimum. The algorithm then swaps the smallest element with the leftmost unsorted element and makes a recursive call to selection_sort with start_idx+1, which sorts the remaining unsorted elements.

Note that the recursive version of Selection Sort has a worst-case time complexity of O(n^2), the same as the iterative version, since both implementations make the same number of comparisons and swaps.

Overall, these sorting algorithms are simple and easy to implement, but they are generally less efficient than more advanced sorting algorithms for large or complex lists.

## 14.3 Recursive sorts: Mergesort and Quicksort

In the context of comparison sorting algorithms, recursive sorts are algorithms that use a recursive approach to divide the input list into smaller sublists, sort the sublists recursively, and combine the sorted sublists to obtain the final sorted list. Recursive sorts are often based on divide-and-conquer strategies, which can lead to more efficient sorting algorithms than those based on simple comparison-based approaches.

Mergesort and Quicksort are two of the most popular recursive sorting algorithms, and they are particularly relevant in this category because they both have a worst-case time complexity of O(nlogn) and are widely used in practice.

### 14.3.1 Mergesort

Mergesort divides the input list into two halves, sorts each half recursively using mergesort, and then merges the sorted halves into a single sorted list. Mergesort is a stable sorting algorithm, which means that it preserves the relative order of equal elements in the input list, and it is often used in situations where stability is important.

Here is the pseudocode for recursive mergesort:

```
// Recursive Mergesort algorithm
mergesort(array):
    n = length(array)
    if n == 1:
        return array

    mid = n // 2
    left = mergesort(array[0:mid])
    right = mergesort(array[mid:n])

    return merge(left, right)

// Merge function
merge(left, right):
    result = []
    i = 0
    j = 0

    while i < length(left) and j < length(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    while i < length(left):
        result.append(left[i])
        i += 1

    while j < length(right):
        result.append(right[j])
        j += 1

    return result
```

In this recursive algorithm, the input list is first divided into halves recursively until the base case of n = 1 is reached. Then, the sorted halves are merged back together using the `merge` function, which iterates through the two sorted subarrays and compares their elements to create a new sorted array. The `merge` function works by copying the two subarrays into temporary arrays, comparing their elements, and copying them back into the original array in sorted order.

Note that the recursive mergesort algorithm has a worst-case time complexity of O(nlogn) and is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the input list.

It is also possible to implement mergesort iteratively using an algorithm known as iterative mergesort or bottom-up mergesort. This algorithm does not use recursion, and instead works by iteratively merging adjacent subarrays of the input list until the entire list is sorted.

The basic idea behind iterative mergesort is to start by merging adjacent pairs of elements in the input list to create sorted subarrays of size 2. The algorithm then merges adjacent pairs of subarrays to create sorted subarrays of size 4, and so on, until the entire input list is sorted.

Here is the pseudocode for iterative mergesort:

```
// Iterative Mergesort algorithm
mergesort(array):
    n = length(array)
    subarray_size = 1
    while subarray_size < n:
        for i from 0 to n-subarray_size-1 step subarray_size*2:
            left = i
            right = min(i+subarray_size, n)
            end = min(i+subarray_size*2, n)
            merge(array, left, right, end)
        subarray_size *= 2
```

In this iterative algorithm, the input list is first divided into subarrays of size 1, and then adjacent subarrays are merged iteratively until the entire list is sorted. The merge function is used to merge adjacent subarrays, and it works by copying the two subarrays into temporary arrays, comparing their elements, and copying them back into the original array in sorted order.

Note that the iterative mergesort algorithm has the same worst-case time complexity of O(nlogn) as the recursive mergesort algorithm, but it can be more efficient in practice due to its better cache locality and lack of recursion overhead.

### 14.3.2 Quicksort

Quicksort also uses a divide-and-conquer approach, but it sorts the input list in place by selecting a pivot element and partitioning the list into two sublists, one containing elements smaller than the pivot and the other containing elements greater than or equal to the pivot. Quicksort then recursively sorts each sublist using quicksort, and combines the sorted sublists to obtain the final sorted list. Quicksort is often faster than mergesort in practice due to its cache efficiency and low constant factors, but it can have a worst-case time complexity of O(n^2) if the pivot selection is not optimal.

Here is the pseudocode for the Recursive Quicksort algorithm:

```
// Recursive Quicksort algorithm
quicksort(array, start, end):
    if start < end:
        pivot = partition(array, start, end)
        quicksort(array, start, pivot-1)
        quicksort(array, pivot+1, end)

// Partition function
partition(array, start, end):
    pivot = array[end]
    i = start - 1

    for j from start to end-1:
        if array[j] < pivot:
            i += 1
            swap(array[i], array[j])

    swap(array[i+1], array[end])
    return i+1
```

In this recursive algorithm, the quicksort function sorts the input array by first partitioning the array around a pivot element, and then recursively sorting the resulting sublists. The partition function selects the rightmost element of the subarray as the pivot element, and rearranges the elements such that all elements less than the pivot are to its left, and all elements greater than the pivot are to its right.

The `quicksort` function then recursively sorts the subarray to the left and right of the pivot by calling itself on each of the subarrays. This process continues until the subarrays have a size of 1 or 0, at which point they are considered sorted.

Note that the time complexity of recursive Quicksort is O(nlogn) on average, but can be O(n^2) in the worst case if the pivot selection is not optimal.

# Chapter 15

# Learning outcomes

## 15.1 Identify the different approaches of different comparison sorting algorithms

Comparison sorting algorithms are algorithms that sort a list of elements by comparing each element to one another. They can be broadly categorized into two main approaches: comparison-based and non-comparison-based.

Comparison-based algorithms are the most common approach to sorting and work by comparing pairs of elements in the input list and swapping them if they are out of order. These algorithms have a worst-case time complexity of O(nlogn) and are generally slower than non-comparison-based algorithms for large datasets. Some popular comparison-based sorting algorithms include Quicksort, Mergesort, Heapsort, Insertion Sort, and Selection Sort.

Non-comparison-based algorithms, on the other hand, do not compare individual elements to one another, but instead rely on other properties of the input data to sort the list. These algorithms can achieve linear time complexity, but are often more complex and less flexible than comparison-based algorithms. Examples of non-comparison-based sorting algorithms include Counting Sort, Radix Sort, and Bucket Sort.

It is worth noting that some sorting algorithms, such as Introsort and Timsort, combine elements of both comparison-based and non-comparison-based approaches to achieve a balance between performance and simplicity.

Overall, the choice of sorting algorithm depends on several factors, including the size of the input list, the distribution of the input values, and the specific requirements of the application.

## 15.2 Implement different comparison sorting algorithms

See the text above.

Here is the implementation of some of the popular comparison sorting algorithms in Python:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Insertion Sort:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr
```

Selection Sort:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Merge Sort:

```
def merge_sort(arr):
    if len(arr) > 1:
```

```
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr
```

Quick Sort:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = []
    right = []
    for i in range(1, len(arr)):
        if arr[i] < pivot:
            left.append(arr[i])
        else:
            right.append(arr[i])
    return quick_sort(left) + [pivot] + quick_sort(right)
```

Note that these are just sample implementations and there are many variations and optimizations that can be made to these algorithms depending on the spe-

cific requirements of the application. Additionally, these implementations may not be the most efficient or optimal for all cases.

## 15.3 Calculate the time complexity of different comparison sorting algorithms

The time complexity of different comparison sorting algorithms can vary depending on the specific implementation and the properties of the input data. Here are the time complexities of some common comparison sorting algorithms:

Bubble Sort:

- Best Case: O(n)
- Worst Case: O(n^2)
- Average Case: O(n^2)

Insertion Sort: - Best Case: O(n) - Worst Case: O(n^2) - Average Case: O(n^2)

Selection Sort: - Best Case: O(n^2) - Worst Case: O(n^2) - Average Case: O(n^2)

Merge Sort: - Best Case: O(nlogn) - Worst Case: O(nlogn) - Average Case: O(nlogn)

Quick Sort: - Best Case: O(nlogn) - Worst Case: O(n^2) - Average Case: O(nlogn)

Note that these time complexities are based on the number of comparisons or operations required to sort an input list of n elements. The actual running time of an algorithm may depend on factors such as memory access times, cache behavior, and the specific implementation of the algorithm. Additionally, there are other factors to consider when choosing a sorting algorithm, such as the stability of the algorithm, the space complexity, and the ease of implementation.

# Chapter 16

# Non-comparison sorting algorithms

## 16.1 The limits of comparison sorts

While comparison-based sorting algorithms are widely used and have many advantages, there are some limits to their performance and scalability. Here are some of the limitations of comparison-based sorting algorithms:

1. **Lower Bound**: The worst-case time complexity of comparison-based sorting algorithms is O(nlogn), which is the lower bound for any comparison-based sorting algorithm. This means that there is a limit to how fast a comparison-based algorithm can sort a list of elements.
2. **Performance on Certain Data**: Some comparison-based sorting algorithms, such as Quicksort and Mergesort, are particularly well-suited for sorting randomly ordered input data. However, they may perform poorly on already sorted or nearly sorted data, resulting in unnecessary comparisons and swaps.
3. **Comparison Overhead**: Comparison-based sorting algorithms require comparisons between elements to determine their relative order. In some cases, the cost of these comparisons may be higher than other operations, such as swaps or moves. This can impact the performance of the algorithm.
4. **Non-Adaptive**: Comparison-based sorting algorithms are not adaptive, meaning that they do not take advantage of any existing order in the input data. This can result in unnecessary comparisons and swaps when sorting partially sorted data.

Non-comparison-based sorting algorithms, on the other hand, can overcome some of these limitations. For example, Counting Sort, Radix Sort, and Bucket

Sort all have linear time complexity, making them faster than comparison-based sorting algorithms for certain types of input data. These algorithms also do not require comparisons between individual elements, making them more efficient in some cases. However, non-comparison-based sorting algorithms also have their own limitations and may not be suitable for all types of input data.

The choice of sorting algorithm depends on the specific requirements of the application and the properties of the input data. While comparison-based sorting algorithms are widely used and have many advantages, there are limits to their performance and scalability, which may make non-comparison-based sorting algorithms a better choice in certain situations.

## 16.2 Counting, radix and bucket sort

Counting Sort, Radix Sort, and Bucket Sort are all examples of non-comparison-based sorting algorithms.

### 16.2.1 Counting Sort

Counting Sort is a linear time sorting algorithm that is used to sort a list of integers in a specific range. The algorithm works by counting the number of occurrences of each integer value in the input list, and using this count to determine the sorted order of the values. Counting Sort assumes that the input data consists of integers in a known range, making it a useful algorithm for certain types of input data. Counting Sort has a time complexity of O(n+k), where n is the number of elements in the input list and k is the range of the input values.

Here is some pseudocode for Counting Sort:

```
countingSort(array A, int k)
    let C[0..k] be a new array
    let B[0..n-1] be a new array
    for i = 0 to k
        C[i] = 0
    for j = 0 to n-1
        C[A[j]] = C[A[j]] + 1
    for i = 1 to k
        C[i] = C[i] + C[i-1]
    for j = n-1 down to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] - 1
    return B
```

The `countingSort` function takes as input an array `A` of integers and an integer `k`, which represents the maximum value in `A`. The function creates two new

arrays `C` and `B`, with sizes `k+1` and `n` respectively, where `n` is the length of `A`.

The algorithm then iterates through `A` to count the number of occurrences of each value in `C`. It then updates `C` to hold the prefix sum of the counts. Finally, it iterates through `A` in reverse order and uses `C` to place each element into its correct sorted position in `B`.

The sorted array `B` is then returned as the output of the function.

The time complexity of Counting Sort is O(n+k), where `n` is the length of the input array and `k` is the maximum value in the input array.

### 16.2.2 Radix Sort

Radix Sort is a linear time sorting algorithm that is used to sort a list of integers by their digits. The algorithm works by sorting the input list based on the values of their digits, starting from the least significant digit to the most significant digit. Radix Sort can be implemented using either a least significant digit (LSD) or most significant digit (MSD) approach. Radix Sort has a time complexity of O(nk), where n is the number of elements in the input list and k is the maximum number of digits in the input values.

Here is the pseudocode for Radix Sort (LSD):

```
radixSortLSD(array A)
    let B[0..n-1] be a new array
    let maxDigit be the maximum number of digits in the numbers in A
    for d = 0 to maxDigit-1
        let C[0..9] be a new array
        for i = 0 to 9
            C[i] = 0
        for j = 0 to n-1
            digit = (A[j] / (10^d)) mod 10
            C[digit] = C[digit] + 1
        for i = 1 to 9
            C[i] = C[i] + C[i-1]
        for j = n-1 down to 0
            digit = (A[j] / (10^d)) mod 10
            B[C[digit]-1] = A[j]
            C[digit] = C[digit] - 1
        A = B
    return A
```

The `radixSortLSD` function takes as input an array `A` of integers. The function first determines the maximum number of digits in the numbers in `A`. It then iterates through each digit position from the least significant digit to the most significant digit.

For each digit position, the algorithm creates a new array `C` to hold the counts of each digit value (0-9) in `A`. It then iterates through `A` to count the number of occurrences of each digit value, updates `C` to hold the prefix sum of the counts, and uses `C` to place each element into its correct sorted position in a temporary array `B`.

Finally, the sorted array `B` is assigned back to `A` and the algorithm moves on to the next digit position.

The time complexity of Radix Sort (LSD) is O(d(n+k)), where `n` is the length of the input array, `k` is the maximum value in the input array, and `d` is the maximum number of digits in the numbers in `A`.

### 16.2.3 Bucket Sort

Bucket Sort is a linear time sorting algorithm that is used to sort a list of elements by dividing them into buckets and then sorting the individual buckets. The algorithm works by dividing the input values into a fixed number of equally sized buckets, and then sorting each bucket using another sorting algorithm, such as Insertion Sort or Quicksort. Bucket Sort assumes that the input data is uniformly distributed across a known range, making it a useful algorithm for certain types of input data. Bucket Sort has a time complexity of O(n+k), where n is the number of elements in the input list and k is the number of buckets.

These non-comparison-based sorting algorithms offer advantages over comparison-based sorting algorithms in certain situations. For example, Counting Sort and Radix Sort have linear time complexity, which can be faster than the O(nlogn) time complexity of comparison-based sorting algorithms. However, these algorithms also have specific requirements and limitations, such as the need for input data in a known range or the need for uniformly distributed input data. It is important to choose the appropriate sorting algorithm based on the specific requirements of the application and the properties of the input data.

Here is the pseudocode for Bucket Sort:

```
bucketSort(array A)
    let B[0..n-1] be a new array
    let buckets[0..n-1] be a new array of empty lists
    for i = 0 to n-1
        bucketIndex = floor(n*A[i])
        append A[i] to buckets[bucketIndex]
    for i = 0 to n-1
        sort buckets[i] using any sorting algorithm
        concatenate buckets[i] onto B
    return B
```

The `bucketSort` function takes as input an array `A` of floating-point numbers between 0 and 1. The function creates a new array `B` of the same length as `A`, as well as an array of `n` empty lists called `buckets`. The `n` value here refers to the number of buckets, which is usually chosen to be the same as the length of `A`.

The algorithm then iterates through `A`, calculates the index of the corresponding bucket for each element using the formula `floor(n*A[i])`, and appends each element to the appropriate bucket.

Next, the algorithm sorts each non-empty bucket individually using any sorting algorithm, and concatenates the sorted buckets together into `B`.

Finally, the sorted array `B` is returned as the output of the function.

The time complexity of Bucket Sort depends on the sorting algorithm used to sort the individual buckets, but it can achieve linear time complexity in the average case if the elements are uniformly distributed among the buckets. In the worst case, where all elements fall into the same bucket, the time complexity is the same as the sorting algorithm used for sorting the buckets.

# Chapter 17

# Learning outcomes

## 17.1 Identify the different approaches of different non-comparison sorting algorithms

There are different non-comparison sorting algorithms, each with its own approach. Here are the main approaches of some popular non-comparison sorting algorithms:

1. **Counting Sort**: This algorithm works by counting the number of occurrences of each element in the input array, and then using this count to determine the position of each element in the output array. Counting Sort is efficient when the range of values in the input array is small.

2. **Radix Sort**: This algorithm sorts the input elements by their digits, from the least significant to the most significant. Radix Sort can be implemented using LSD (Least Significant Digit) or MSD (Most Significant Digit) approach, depending on the sorting order.

3. **Bucket Sort**: This algorithm divides the input elements into a number of buckets and then sorts the elements within each bucket. Bucket Sort is efficient when the input elements are uniformly distributed over a range.

4. **Pigeonhole Sort**: This algorithm is a variation of Bucket Sort, where each bucket represents a pigeonhole, and the input elements are distributed among these pigeonholes according to their values. Pigeonhole Sort is efficient when the range of values in the input array is small and the values are integers.

Each non-comparison sorting algorithm has its own strengths and weaknesses, and the choice of algorithm depends on the specific characteristics of the input data.

## 17.2 Implement different non-comparison sorting algorithms

Here are implementations of some popular non-comparison sorting algorithms in Python:

### 17.2.1 Counting Sort

```
def counting_sort(arr):
    # Find the maximum value in the array
    k = max(arr)

    # Initialize an array to hold the counts of each element
    counts = [0] * (k+1)

    # Count the number of occurrences of each element in the array
    for elem in arr:
        counts[elem] += 1

    # Calculate the cumulative counts of the elements
    for i in range(1, k+1):
        counts[i] += counts[i-1]

    # Initialize the output array
    output = [0] * len(arr)

    # Place each element in its sorted position
    for elem in arr:
        index = counts[elem] - 1
        output[index] = elem
        counts[elem] -= 1

    return output
```

### 17.2.2 Radix Sort (LSD)

```
def radix_sort_lsd(arr):
    # Find the maximum number of digits in the array
    max_digit = len(str(max(arr)))

    # Sort the array by each digit, from LSD to MSD
    for digit in range(max_digit):
        # Initialize the bucket array
        buckets = [[] for _ in range(10)]
```

```
        # Distribute the elements into the buckets based on the current digit
        for elem in arr:
            bucket_index = (elem // (10**digit)) % 10
            buckets[bucket_index].append(elem)

        # Concatenate the elements in the buckets to form the sorted array
        arr = [elem for bucket in buckets for elem in bucket]

    return arr
```

### 17.2.3 Bucket Sort

```
def bucket_sort(arr):
    # Determine the number of buckets
    num_buckets = len(arr)

    # Initialize the buckets
    buckets = [[] for _ in range(num_buckets)]

    # Distribute the elements into the buckets
    for elem in arr:
        bucket_index = int(elem * num_buckets)
        buckets[bucket_index].append(elem)

    # Sort the elements within each bucket
    for i in range(num_buckets):
        buckets[i].sort()

    # Concatenate the elements in the buckets to form the sorted array
    sorted_arr = [elem for bucket in buckets for elem in bucket]

    return sorted_arr
```

These implementations assume that the input is a list of integers or floating-point numbers. The time complexity of these non-comparison sorting algorithms depends on the input size and the range of values in the input. Counting Sort and Bucket Sort have linear time complexity in the average case, while Radix Sort has linear time complexity in the worst case.

## 17.3 Calculate the time complexity of different non-comparison sorting algorithms

Here are the time complexities of some popular non-comparison sorting algorithms:

### 17.3.1 Counting Sort

Counting Sort has a time complexity of O(n+k), where n is the number of elements in the input array, and k is the range of values in the input array. Counting Sort is efficient when the range of values is small, such as when sorting integers or characters.

### 17.3.2 Radix Sort

Radix Sort has a time complexity of O(d(n+k)), where n is the number of elements in the input array, k is the range of values in the input array, and d is the number of digits in the maximum value in the input array. Radix Sort is efficient when the input values have a fixed number of digits, such as when sorting integers.

### 17.3.3 Bucket Sort

Bucket Sort has a time complexity of O(n+k), where n is the number of elements in the input array, and k is the number of buckets used for sorting. Bucket Sort is efficient when the input elements are uniformly distributed over a range.

These non-comparison sorting algorithms have better time complexity than comparison-based sorting algorithms, which have a lower bound of O(n log n) in the average case. However, the performance of non-comparison sorting algorithms depends on the specific characteristics of the input data, and they may not be efficient in all cases.

# Chapter 18

# Hashing

## 18.1 The problem of searching

In the context of hashing, the problem of searching refers to the task of finding a specific element in a set of elements that have been stored in a hash table. The goal of searching in a hash table is to quickly find the location of the element in the table, so that its value can be accessed or modified.

The problem of searching in a hash table can be divided into two main subproblems: first, determining the hash value of the element to be searched, and second, using this hash value to locate the element in the hash table. The hash function used to determine the hash value should be designed to minimize collisions, so that the search can be performed quickly and efficiently.

In order to search for an element in a hash table, the same hash function that was used to store the element in the table must be used to compute its hash value. Once the hash value is computed, the element can be found by using a search algorithm that takes into account the collision resolution method used by the hash table.

There are several search algorithms that can be used to search for an element in a hash table, including linear probing, quadratic probing, and chaining. The choice of search algorithm depends on the specific implementation of the hash table and the characteristics of the input data. The goal of these algorithms is to find the location of the element in the hash table in as few operations as possible, so that the search can be performed quickly and efficiently.

## 18.2 Hash tables, hash functions

In the context of hashing, a hash table is a data structure that is used to store a set of elements, with the goal of providing efficient insertion, deletion, and

search operations. A hash table uses a hash function to map each element to a unique location in the table, based on its key.

A hash function is a function that takes an element as input and computes a hash code, which is an integer that represents the location of the element in the hash table. A good hash function should be designed to minimize collisions, which occur when two or more elements are mapped to the same location in the table. Collisions can lead to slower insertion, deletion, and search operations in the hash table, so a good hash function is crucial to the performance of a hash table.

Once the hash code is computed, the element is stored in the corresponding location in the hash table. If another element is already stored in that location, a collision occurs, and a collision resolution method is used to resolve the conflict. There are several collision resolution methods that can be used, including linear probing, quadratic probing, and chaining.

Hash tables are widely used in computer science and software engineering, due to their efficiency in storing and retrieving data. Hash tables are used in applications such as database indexing, symbol tables in compilers and interpreters, and network routing tables.

## 18.3 Collision resolution techniques

In the context of hashing, collision resolution techniques are methods used to handle the situation when two or more elements in a hash table are mapped to the same location, also known as a collision. There are several techniques for resolving collisions, including the following:

1. **Open addressing**: In this technique, when a collision occurs, the hash function is used to compute a new location for the element in the table, based on a predefined rule. There are several rules that can be used for open addressing, including linear probing, quadratic probing, and double hashing.

2. **Chaining**: In this technique, each location in the hash table contains a linked list of elements that have the same hash code. When a collision occurs, the new element is added to the linked list at the corresponding location.

3. **Cuckoo hashing**: In this technique, each element is assigned two hash functions and can be stored in two different locations in the hash table. When a collision occurs, one of the elements is evicted from its location and moved to the alternate location, recursively updating the hash codes until a free location is found.

The choice of collision resolution technique depends on the specific requirements of the application and the characteristics of the input data. The goal of these

techniques is to minimize the number of collisions and the time required to resolve them, in order to maintain the performance of the hash table.

# Chapter 19

# Learning outcomes

## 19.1 Describe the different methods used to search for data

There are different methods used to search for data in a hash table, depending on the specific implementation and the characteristics of the data being searched. Here are some common methods:

1. **Direct Addressing**: In this method, the hash function is used to compute the index of the location in the hash table where the element is stored. When searching for an element, the hash function is used to compute the index, and the element is retrieved from that location.

2. **Linear Probing**: In this method, if a collision occurs, the search algorithm checks the next location in the table, and so on, until an empty location is found. When searching for an element, the search algorithm uses the hash function to compute the index of the first location where the element might be stored, and then checks each subsequent location until the element is found.

3. **Quadratic Probing**: This method is similar to linear probing, but instead of checking the next location in the table, the search algorithm checks a location that is a quadratic offset from the original location. For example, if the original location is i, the next location checked is (i + 1^2) % table_size, followed by (i + 2^2) % table_size, and so on.

4. **Chaining**: In this method, each location in the hash table contains a linked list of elements that have the same hash code. When searching for an element, the search algorithm uses the hash function to compute the index of the location where the element might be stored, and then searches the linked list at that location to find the element.

The choice of search method depends on the specific implementation of the hash table and the characteristics of the input data. The goal of these methods is to find the location of the element in the hash table as quickly and efficiently as possible, in order to maintain the performance of the hash table.

## 19.2 Describe different collision resolution methods

Collision resolution methods are used to handle the situation when two or more elements in a hash table are mapped to the same location, also known as a collision. There are several collision resolution methods, including the following:

1. **Open Addressing**: In this method, when a collision occurs, the hash function is used to compute a new location for the element in the table, based on a predefined rule. There are several rules that can be used for open addressing, including linear probing, quadratic probing, and double hashing.

2. **Chaining**: In this method, each location in the hash table contains a linked list of elements that have the same hash code. When a collision occurs, the new element is added to the linked list at the corresponding location.

3. **Robin Hood Hashing**: In this method, the hash table is filled sequentially with elements, and when a collision occurs, the element with the smaller distance from its original location is moved closer to the location where it would have been inserted if there were no collision.

4. **Cuckoo Hashing**: In this method, each element is assigned two hash functions and can be stored in two different locations in the hash table. When a collision occurs, one of the elements is evicted from its location and moved to the alternate location, recursively updating the hash codes until a free location is found.

The choice of collision resolution method depends on the specific requirements of the application and the characteristics of the input data. The goal of these methods is to minimize the number of collisions and the time required to resolve them, in order to maintain the performance of the hash table.

## 19.3 Implement a hash table with linear probing collision resolution

Here's an implementation of a hash table with linear probing collision resolution in Python:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.keys = [None] * size
        self.values = [None] * size

    def hash_function(self, key):
        return hash(key) % self.size

    def put(self, key, value):
        index = self.hash_function(key)
        while self.keys[index] is not None and self.keys[index] != key:
            index = (index + 1) % self.size
        self.keys[index] = key
        self.values[index] = value

    def get(self, key):
        index = self.hash_function(key)
        while self.keys[index] is not None:
            if self.keys[index] == key:
                return self.values[index]
            index = (index + 1) % self.size
        return None

    def remove(self, key):
        index = self.hash_function(key)
        while self.keys[index] is not None:
            if self.keys[index] == key:
                self.keys[index] = None
                self.values[index] = None
                return
            index = (index + 1) % self.size

    def print_table(self):
        for i in range(self.size):
            if self.keys[i] is not None:
                print(self.keys[i], self.values[i])
```

This implementation uses a hash function to compute the index of the location in the hash table where the key-value pair should be stored. If there is a collision, it uses linear probing to find the next available location in the table.

The `put()` method inserts a key-value pair into the hash table. It first computes the index of the location in the table where the key-value pair should be stored using the hash function. If that location is already occupied, it uses linear

probing to find the next available location. Once it finds an empty location, it stores the key-value pair there.

The `get()` method retrieves the value associated with a given key from the hash table. It first computes the index of the location in the table where the key-value pair should be stored using the hash function. If that location is already occupied and the key at that location is not the desired key, it uses linear probing to find the next occupied location. If it finds the key, it returns the corresponding value. If it reaches an empty location or the end of the table without finding the key, it returns `None`.

The `remove()` method removes a key-value pair from the hash table. It first computes the index of the location in the table where the key-value pair should be stored using the hash function. If that location is already occupied and the key at that location is not the desired key, it uses linear probing to find the next occupied location. If it finds the key, it sets the key and value at that location to `None`.

The `print_table()` method is a helper method that prints the contents of the hash table.

# Chapter 20

# Linear data structures

### 20.0.1 Data structures

Linear data structures are a type of data structure where the data elements are arranged in a linear order, with each element connected to its predecessor and/or successor. There are several key linear data structures, including the following:

1. **Arrays**: An array is a collection of elements that are stored in a contiguous block of memory, with each element accessed using an index. Arrays are fixed in size, which means that the number of elements they can store must be known in advance.

2. **Linked Lists**: A linked list is a collection of elements where each element, called a node, contains both data and a reference to the next node in the list. Linked lists can be singly linked, where each node contains a reference to the next node, or doubly linked, where each node contains references to both the next and previous nodes.

3. **Stacks**: A stack is a data structure that stores a collection of elements and supports two main operations: push, which adds an element to the top of the stack, and pop, which removes and returns the top element from the stack. Stacks follow a Last-In-First-Out (LIFO) order, which means that the last element pushed onto the stack is the first one to be popped off.

4. **Queues**: A queue is a data structure that stores a collection of elements and supports two main operations: enqueue, which adds an element to the back of the queue, and dequeue, which removes and returns the element at the front of the queue. Queues follow a First-In-First-Out (FIFO) order, which means that the first element enqueued is the first one to be dequeued.

5. **Deques**: A deque, short for double-ended queue, is a data structure that supports operations at both ends. Elements can be added and removed from both the front and back of the deque. Deques can be thought of as a combination of stacks and queues.

These key linear data structures are fundamental building blocks for many algorithms and data processing tasks in computer science. Choosing the right data structure for a particular problem can have a significant impact on the efficiency and performance of the algorithm or program.

## 20.1 Dynamic memory allocation

Dynamic memory allocation is the process of allocating memory at runtime, rather than at compile-time. This allows programs to use memory as needed, without requiring the amount of memory to be known beforehand.

In the context of linear data structures, dynamic memory allocation is often used to create linked lists and other structures where the size of the structure may change during program execution. For example, a linked list may initially contain only a few nodes, but as more data is added, the list may need to grow in size.

In languages like C and C++, dynamic memory allocation is typically done using the `malloc()` and `free()` functions. `malloc()` is used to allocate a block of memory of a specified size, while `free()` is used to release the memory once it is no longer needed.

In languages like Python, Java, and C#, dynamic memory allocation is handled automatically by the language's memory management system. These languages use garbage collection to automatically free up memory that is no longer being used by the program.

Dynamic memory allocation can be a powerful tool, but it can also introduce new problems such as memory leaks and fragmentation. It is important for developers to carefully manage dynamic memory allocation to avoid these issues and ensure efficient use of memory.

## 20.2 Linear data structures

Linear data structures are a type of data structure where the data elements are arranged in a linear order, with each element connected to its predecessor and/or successor. This means that the elements are arranged in a sequence, with one element following another in a linear fashion.

Some examples of linear data structures include arrays, linked lists, stacks, queues, and deques. Arrays store elements in a contiguous block of memory,

with each element accessed using an index. Linked lists use nodes to store elements, with each node containing both data and a reference to the next node in the list. Stacks and queues are collections of elements that support operations for adding and removing elements in a specific order, with stacks following a Last-In-First-Out (LIFO) order and queues following a First-In-First-Out (FIFO) order. Deques are similar to queues, but they allow for elements to be added and removed from both the front and back of the collection.

Linear data structures are often used in computer science and programming because they provide a simple and efficient way to organize and access data. The choice of which linear data structure to use for a given problem depends on the specific requirements of the problem, including the type and amount of data being processed, and the operations that need to be performed on that data.

# Chapter 21

# Learning outcomes

## 21.1 Describe linear data structures and its operations using pseudocode

Here's an overview of linear data structures and their operations using pseudocode:

### 21.1.1 Arrays

a. Initializing an array:

```
n = 10 # size of array
array = [0] * n # initialize array with all elements set to 0
```

b. Accessing an element:

```
x = array[i] # access element i in array
```

c. Updating an element:

```
array[i] = x # set element i in array to x
```

### 21.1.2 Linked Lists

a. Initializing a linked list

```
head = None # initialize head pointer to null
```

b. Inserting a node at the beginning:

```
node = Node(data) # create new node with data
node.next = head # set next pointer of new node to current head
head = node # update head pointer to new node
```

c. Deleting a node from the beginning:

```
if head is not None:
    head = head.next # update head pointer to next node
```

### 21.1.3 Stacks

a. Initializing a stack:

```
stack = [] # initialize an empty list
```

b. Pushing an element onto the stack:

```
stack.append(x) # add element x to end of list
```

c. Popping an element from the stack:

```
x = stack.pop() # remove and return last element in list
```

### 21.1.4 Queues

a. Initializing a queue:

```
queue = [] # initialize an empty list
```

b. Enqueuing an element:

```
queue.append(x) # add element x to end of list
```

c. Dequeuing an element:

```
x = queue.pop(0) # remove and return first element in list
```

### 21.1.5 Deques (double ended queues)

a. Initializing a deque:

```
deque = [] # initialize an empty list
```

b. Adding an element to the front:

```
deque.insert(0, x) # insert element x at the beginning of list
```

c. Removing an element from the front:

```
x = deque.pop(0) # remove and return first element in list
```

d. Adding an element to the back:

```
deque.append(x) # add element x to end of list
```

e. Removing an element from the back:

```
x = deque.pop() # remove and return last element in list
```

Note that this is just a basic overview of the operations for each linear data structure, and the actual implementation may vary depending on the specific programming language and data structure library being used.

## 21.2 Understand array and linked list based implementations of stacks and queues

### 21.2.1 Arrays for Stacks and Queues

Arrays can be used to implement both stacks and queues. In an array-based implementation, a fixed-size array is used to store the elements, and a pointer (often called "top" for stacks and "front" and "rear" for queues) is used to keep track of the current position of the element(s) in the data structure.

For a stack implemented using an array, new elements are added to the end of the array and removed from the end of the array. The "top" pointer points to the last element added to the stack, and is updated with each push or pop operation.

For a queue implemented using an array, new elements are added to the end of the array and removed from the front of the array. The "front" and "rear" pointers point to the first and last elements of the queue, respectively, and are updated with each enqueue or dequeue operation.

One limitation of array-based implementations is that the size of the array is fixed, which means that it cannot be dynamically resized if more elements need to be added than the array can hold. This can lead to inefficiencies in memory usage if the array is larger than necessary, or errors if the array is not large enough to hold all the required elements.

### 21.2.2 Linked Lists for Stacks and Queues

Linked lists can also be used to implement both stacks and queues. In a linked list-based implementation, a series of nodes are used to store the elements, with

each node containing both the element and a pointer to the next node in the list.

For a stack implemented using a linked list, new elements are added to the beginning of the list and removed from the beginning of the list. The "top" pointer points to the first element in the list, and is updated with each push or pop operation.

For a queue implemented using a linked list, new elements are added to the end of the list and removed from the beginning of the list. The "front" pointer points to the first element in the list, and the "rear" pointer points to the last element in the list. Both pointers are updated with each enqueue or dequeue operation.

One advantage of linked list-based implementations is that they can dynamically resize to accommodate any number of elements. However, linked lists can be less efficient than array-based implementations in terms of memory usage and cache locality, since each element is stored in a separate node and requires additional memory for the pointer to the next node.

## 21.3 Implement a sorted linked list

Here's an implementation of a sorted linked list in Python:

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None

class SortedLinkedList:
    def __init__(self):
        self.head = None

    def add(self, val):
        # create a new node with the given value
        new_node = Node(val)

        # if the list is empty, add the new node as the head
        if not self.head:
            self.head = new_node
            return

        # if the new node's value is less than the head's value,
        # insert it at the beginning of the list
        if val < self.head.val:
            new_node.next = self.head
```

```
            self.head = new_node
            return

        # find the appropriate place to insert the new node
        curr = self.head
        while curr.next and curr.next.val < val:
            curr = curr.next

        # insert the new node after the current node
        new_node.next = curr.next
        curr.next = new_node

    def remove(self, val):
        # if the list is empty, do nothing
        if not self.head:
            return

        # if the value to remove is at the head, remove it
        if self.head.val == val:
            self.head = self.head.next
            return

        # find the node with the value to remove
        curr = self.head
        while curr.next and curr.next.val != val:
            curr = curr.next

        # if the value was not found, do nothing
        if not curr.next:
            return

        # remove the node with the given value
        curr.next = curr.next.next

    def print_list(self):
        # print the values in the list
        curr = self.head
        while curr:
            print(curr.val, end=' ')
            curr = curr.next
        print()
```

Here's an example implementation of a sorted linked list in TypeScript:

```typescript
class Node {
  public value: number;
  public next: Node | null;

  constructor(value: number) {
    this.value = value;
    this.next = null;
  }
}

class SortedLinkedList {
  private head: Node | null;

  constructor() {
    this.head = null;
  }

  // insert a value into the linked list in sorted order
  insert(value: number): void {
    const newNode = new Node(value);

    if (this.head === null || value < this.head.value) {
      // if the linked list is empty or the new value is smaller than the head value
      // insert the new node at the beginning of the list
      newNode.next = this.head;
      this.head = newNode;
    } else {
      // otherwise, find the appropriate position to insert the new node
      let current = this.head;
      while (current.next !== null && current.next.value < value) {
        current = current.next;
      }
      newNode.next = current.next;
      current.next = newNode;
    }
  }

  // print the values in the linked list
  print(): void {
    let current = this.head;
    while (current !== null) {
      console.log(current.value);
      current = current.next;
    }
```

```
  }
}

// example usage
const sortedList = new SortedLinkedList();
sortedList.insert(3);
sortedList.insert(1);
sortedList.insert(4);
sortedList.insert(2);
sortedList.print(); // output: 1 2 3 4
```

Here is an implementation of a sorted linked list in C:

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for the linked list node
typedef struct node {
    int data;
    struct node* next;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a new node into the sorted linked list
void insertNode(Node** head, int data) {
    Node* current;
    Node* newNode = createNode(data);

    // If the list is empty or the new node is smaller than the head node, insert at the begin
    if (*head == NULL || (*head)->data >= data) {
        newNode->next = *head;
        *head = newNode;
    }
    // Otherwise, iterate through the list to find the correct position to insert the new node
    else {
        current = *head;
        while (current->next != NULL && current->next->data < data) {
```

```
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}

// Function to print the linked list
void printList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Main function to test the sorted linked list implementation
int main() {
    Node* head = NULL;

    // Insert elements into the sorted linked list
    insertNode(&head, 5);
    insertNode(&head, 2);
    insertNode(&head, 8);
    insertNode(&head, 1);
    insertNode(&head, 10);

    // Print the sorted linked list
    printList(head);

    return 0;
}
```

# Chapter 22

# Trees

### 22.0.1 Trees, tree representation

The key concepts in the context of trees and tree representation are:

1. **Node**: The basic building block of a tree, which contains some data and one or more links to other nodes.
2. **Root**: The topmost node in a tree, which has no parent node.
3. Parent and child nodes: Nodes in a tree can have one or more child nodes and a single parent node.
4. **Leaf nodes**: Nodes in a tree that have no child nodes.
5. **Depth**: The number of edges from the root to a given node in the tree.
6. **Height**: The number of edges from a given node to the deepest leaf node in the tree.
7. **Binary tree**: A tree in which each node can have at most two child nodes.
8. **Binary search tree**: A binary tree in which the left child of a node contains only values less than the node's value, and the right child contains only values greater than the node's value.
9. **Traversal**: The process of visiting all nodes in a tree in a specific order, such as in-order, pre-order, or post-order traversal.
10. **Tree representation**: Trees can be represented using various data structures, such as arrays, linked lists, or dynamically allocated memory.

Trees are a widely used data structure in computer science that are used to represent a hierarchical structure of data. A tree consists of nodes that are connected by edges, with the topmost node being called the root node. Each node can have zero or more child nodes, and a node that has no child nodes is called a leaf node.

The depth of a node in a tree is the number of edges from the root to the node, while the height of a node is the number of edges from the node to the deepest leaf node. In a binary tree, each node can have at most two child nodes, and

a binary search tree is a type of binary tree in which the left child of a node contains only values less than the node's value, and the right child contains only values greater than the node's value.

There are various ways to traverse a tree, such as in-order, pre-order, or post-order traversal, which determine the order in which nodes are visited. Traversing a tree is often necessary to perform operations such as searching, inserting, or deleting nodes in the tree.

Trees can be represented using various data structures, such as arrays, linked lists, or dynamically allocated memory. Each method has its own advantages and disadvantages, and the choice of data structure depends on the specific use case and requirements of the application. For example, if the size of the tree is known in advance and memory efficiency is a concern, an array-based representation might be a good choice. On the other hand, if the size of the tree is not known in advance and nodes need to be added or removed dynamically, a linked list-based representation might be more appropriate.

## 22.1 Binary trees, traversal

A binary tree is a type of tree data structure in which each node has at most two child nodes, which are referred to as the left child and the right child. The structure of a binary tree is such that each node can have zero, one or two children.

Binary trees can be used to solve a wide range of problems such as searching, sorting, and dynamic programming. They are also used as the foundation for more complex data structures such as heaps and binary search trees.

There are three common methods for traversing a binary tree, which are:

1. **In-order traversal**: In this method, the left subtree is first traversed recursively, followed by the root node, and then the right subtree is traversed recursively. The result is a sorted list of the values in the tree.

2. **Pre-order traversal**: In this method, the root node is visited first, followed by the left subtree recursively, and then the right subtree recursively.

3. **Post-order traversal**: In this method, the left subtree is first traversed recursively, followed by the right subtree recursively, and then the root node is visited.

Each traversal method has its own uses and applications, depending on the problem being solved. For example, in-order traversal is often used for searching and sorting, while pre-order traversal is useful for creating a copy of the tree.

Binary trees can also be used to represent a wide range of data structures such as expression trees, binary search trees, and heaps. They are a fundamental data

structure in computer science and are used extensively in many applications.

## 22.2 Binary search trees, insert, search, delete

Binary search tree is a binary tree data structure in which each node has a value and two child nodes, referred to as the left and right subtrees. The left subtree of a node contains only values less than the node's value, while the right subtree contains only values greater than the node's value.

To *insert* a new node into a binary search tree, we start at the root node and compare the value of the new node with the value of the current node. If the new node's value is less than the current node's value, we move to the left subtree, and if the new node's value is greater than the current node's value, we move to the right subtree. We continue this process until we reach a leaf node where we can insert the new node.

To *search* for a node in a binary search tree, we start at the root node and compare the value of the node with the value of the current node. If the value of the node is less than the current node's value, we move to the left subtree, and if the value of the node is greater than the current node's value, we move to the right subtree. We continue this process until we find the node or reach a leaf node where the node is not present in the tree.

To *delete* a node from a binary search tree, we first search for the node to be deleted. If the node has no children, we simply remove the node from the tree. If the node has only one child, we replace the node with its child. If the node has two children, we replace the node with its successor, which is the smallest value in its right subtree, and then delete the successor node.

Deleting a node from a binary search tree can also lead to imbalanced trees, where one subtree is much larger than the other. To maintain balance in the tree, we can use techniques such as tree rotation or AVL trees.

# Chapter 23

# Learning outcomes

## 23.1 Understand how to implement a tree

Here is the pseudocode implementation of a binary search tree:

```
class Node
    value
    left
    right

class BinarySearchTree
    root

    function insert(value)
        new_node = Node(value)
        if root is null
            root = new_node
            return
        current_node = root
        while true
            if value == current_node.value
                return
            if value < current_node.value
                if current_node.left is null
                    current_node.left = new_node
                    return
                current_node = current_node.left
            else
                if current_node.right is null
```

```
                current_node.right = new_node
                return
            current_node = current_node.right

function search(value)
    if root is null
        return false
    current_node = root
    while current_node is not null
        if value == current_node.value
            return true
        if value < current_node.value
            current_node = current_node.left
        else
            current_node = current_node.right
    return false

function delete(value)
    if root is null
        return null

    function find_min(node)
        while node.left is not null
            node = node.left
        return node

    function remove(node, value)
        if node is null
            return null
        if value == node.value
            if node.left is null and node.right is null
                return null
            if node.left is null
                return node.right
            if node.right is null
                return node.left
            temp = find_min(node.right)
            node.value = temp.value
            node.right = remove(node.right, temp.value)
        else if value < node.value
            node.left = remove(node.left, value)
        else
            node.right = remove(node.right, value)
        return node
```

```
        root = remove(root, value)
```

Here is an example implementation of a binary search tree in Python:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        new_node = Node(value)
        if not self.root:
            self.root = new_node
            return
        current_node = self.root
        while True:
            if value == current_node.value:
                return
            if value < current_node.value:
                if not current_node.left:
                    current_node.left = new_node
                    return
                current_node = current_node.left
            else:
                if not current_node.right:
                    current_node.right = new_node
                    return
                current_node = current_node.right

    def search(self, value):
        if not self.root:
            return False
        current_node = self.root
        while current_node:
            if value == current_node.value:
                return True
            if value < current_node.value:
                current_node = current_node.left
            else:
```

```
                current_node = current_node.right
        return False

    def delete(self, value):
        if not self.root:
            return None

        def find_min(node):
            while node.left:
                node = node.left
            return node

        def remove(node, value):
            if not node:
                return None
            if value == node.value:
                if not node.left and not node.right:
                    return None
                if not node.left:
                    return node.right
                if not node.right:
                    return node.left
                temp = find_min(node.right)
                node.value = temp.value
                node.right = remove(node.right, temp.value)
            elif value < node.value:
                node.left = remove(node.left, value)
            else:
                node.right = remove(node.right, value)
            return node

        self.root = remove(self.root, value)
```

In this implementation, each node in the tree is represented by an instance of the `Node` class, which contains a `value` attribute and references to its left and right child nodes. The `BinarySearchTree` class contains methods for inserting a new node into the tree, searching for a node in the tree, and deleting a node from the tree.

The `insert` method first creates a new node with the given value and then traverses the tree, starting at the root node, to find the appropriate place to insert the new node based on the binary search tree property. The `search` method also traverses the tree to find the node with the given value, returning `True` if the node is found and `False` otherwise.

The `delete` method first searches for the node with the given value in the tree,

and then removes it based on the type of children the node has. If the node has no children, it is simply removed. If the node has one child, the child node replaces the removed node. If the node has two children, the node is replaced with its successor and then the successor is removed.

This implementation demonstrates how to create and manipulate a binary search tree, a fundamental tree data structure in computer science.

## 23.2 Describe and trace different types of binary tree traversals using pseudocode

### 23.2.1 Inorder Traversal

```
function inorder(node):
    if node is not null:
        inorder(node.left)
        visit(node)
        inorder(node.right)
```

In this traversal, the left subtree is recursively traversed first, then the current node is visited, and finally the right subtree is recursively traversed.

### 23.2.2 Preorder Traversal

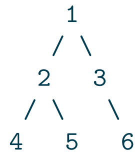
```
function preorder(node):
    if node is not null:
        visit(node)
        preorder(node.left)
        preorder(node.right)
```

### 23.2.3 Postorder Traversal

```
function postorder(node):
    if node is not null:
        postorder(node.left)
        postorder(node.right)
        visit(node)
```

In this traversal, the left subtree is recursively traversed first, then the right subtree is recursively traversed, and finally the current node is visited.

To trace these traversals, let's use the following binary tree as an example:




Using the `inorder` traversal, we would visit the nodes in the order: 4, 2, 5, 1, 3, 6.

Using the `preorder` traversal, we would visit the nodes in the order: 1, 2, 4, 5, 3, 6.

Using the `postorder` traversal, we would visit the nodes in the order: 4, 5, 2, 6, 3, 1.

These pseudocode implementations demonstrate how to traverse binary trees, a fundamental data structure in computer science.

## 23.3 Describe and trace binary search tree operations using pseudocode

Here is the pseudocode for the different operations on a binary search tree:

### 23.3.1 Insertion

```
function insert(node, value):
    if node is null:
        return createNode(value)
    if value < node.value:
        node.left = insert(node.left, value)
    else:
        node.right = insert(node.right, value)
    return node
```

In this operation, we traverse the tree recursively and insert the new value in the correct position based on its value compared to the values of the nodes already in the tree.

### 23.3.2 Search

```
function search(node, value):
    if node is null or node.value == value:
        return node
    if value < node.value:
```

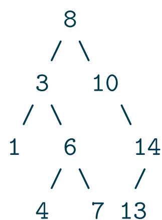
```
        return search(node.left, value)
    else:
        return search(node.right, value)
```

### 23.3.3 Deletion

```
function delete(node, value):
    if node is null:
        return null
    if value < node.value:
        node.left = delete(node.left, value)
    else if value > node.value:
        node.right = delete(node.right, value)
    else:
        if node.left is null:
            return node.right
        if node.right is null:
            return node.left
        temp = minNodeValue(node.right)
        node.value = temp
        node.right = delete(node.right, temp)
    return node
```

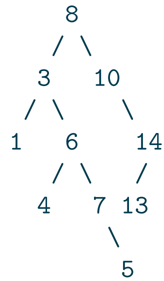
In this operation, we traverse the tree recursively and delete the node with the given value if it exists in the tree. There are three cases: the node has no children, the node has one child, or the node has two children. In the third case, we replace the node with the smallest value in the right subtree, and then delete that node.

To trace these operations, let's use the following binary search tree as an example:



```
       8
      / \
     3   10
    / \    \
   1   6    14
      / \   /
     4   7 13
```

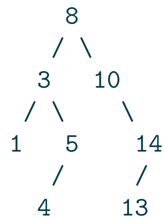
Using the `insert` operation, let's insert the value 5 into the tree. This would result in the following binary search tree:



```
        8
       / \
      3   10
     / \    \
    1   6    14
       / \   /
      4   7 13
           \
            5
```

Using the `search` operation, let's search for the value 7 in the tree. This would return the node with the value 7.

Using the `delete` operation, let's delete the value 6 from the tree. This would result in the following binary search tree:



```
        8
       / \
      3   10
     / \    \
    1   5    14
       /     /
      4     13
```

These pseudocode implementations demonstrate how to perform operations on a binary search tree, a widely used data structure in computer science.

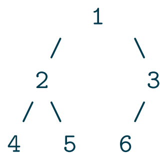
# Chapter 24

# Heaps

## 24.1 Heaps, shape and heap properties

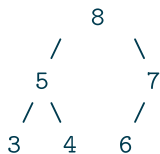
A heap is a specialized tree-based data structure that satisfies two properties: the shape property and the heap property.

The shape property of a heap requires that it is a complete binary tree. This means that all levels of the tree are completely filled, except for possibly the last level, which is filled from left to right. For example, the following is a complete binary tree:



```
        1
      /   \
     2     3
    / \   /
   4   5 6
```

The heap property of a heap depends on whether it is a max-heap or a min-heap. In a max-heap, every node is greater than or equal to its children, while in a min-heap, every node is less than or equal to its children. For example, the following is a max-heap:



```
        8
      /   \
     5     7
    / \   /
   3   4 6
```

In this tree, every node is greater than or equal to its children.

These properties make heaps useful for implementing priority queues, as they allow for efficient access to the maximum or minimum element in the data structure. Additionally, heaps can be implemented efficiently using an array, where the left child of a node is at index `2i + 1` and the right child is at index `2i + 2`, where `i` is the index of the node in the array.

## 24.2 Heap operations

The most common operations on heaps are insertion and deletion of elements. There are two types of heaps: max-heap and min-heap, and the operations differ slightly depending on which type of heap is used.

### 24.2.1 Insertion

To insert an element into a heap, the element is added to the next available position at the bottom level of the heap (which will maintain the shape property), and then it is "bubbled up" to its proper position to maintain the heap property. For a max-heap, this means swapping the new element with its parent until it is larger than its parent (or until it reaches the root), while for a min-heap, it means swapping the new element with its parent until it is smaller than its parent (or until it reaches the root).

### 24.2.2 Deletion

To delete an element from a heap, the element at the root is removed, and then the heap is restructured to maintain the shape property and heap property. For a max-heap, this means replacing the root with the last element in the heap, then "bubbling down" this element to its proper position by swapping it with its larger child until it is larger than both its children (or until it reaches a leaf node), while for a min-heap, it means replacing the root with the last element in the heap, then "bubbling down" this element to its proper position by swapping it with its smaller child until it is smaller than both its children (or until it reaches a leaf node).

Heaps can also support other operations such as finding the minimum or maximum element in the heap, and merging two heaps together. These operations can be implemented efficiently with the help of the heap properties.

## 24.3 Heapsort, priority queues

Heapsort is a sorting algorithm that uses a heap data structure to sort an array. The algorithm consists of two main steps:

1. Build a max-heap from the input array.

2. Repeatedly extract the maximum element from the heap and insert it into the output array, until the heap is empty.

The resulting output array will be sorted in ascending order. Heapsort has a time complexity of O(n log n) in the worst case, and it is an in-place sorting algorithm (i.e., it uses only a constant amount of additional memory).

Priority queues are abstract data types that allow efficient access to the minimum (or maximum) element in a set of elements. Priority queues can be implemented using heaps, where the minimum (or maximum) element is always at the root of the heap. Priority queues are used in a variety of applications, such as task scheduling, graph algorithms (e.g., Dijkstra's shortest path algorithm), and Huffman coding.

To insert an element into a priority queue, it is added to the bottom level of the heap, and then "bubbled up" to its proper position to maintain the heap property. To extract the minimum (or maximum) element from a priority queue implemented with a heap, the root element is removed, and then the heap is restructured to maintain the heap property. These operations can be performed in O(log n) time, where n is the number of elements in the heap.

# Chapter 25

# Learning outcomes

## 25.1 Check heap and shape properties

The heap property and shape property are two key properties that must be maintained by a binary heap.

The heap property states that for a max-heap, every parent node has a value greater than or equal to its children, while for a min-heap, every parent node has a value less than or equal to its children. To check the heap property of a binary heap, we can compare the value of each parent node with the values of its children. If the heap property is violated, we can swap the parent node with its larger (or smaller) child, and continue the comparison until the heap property is restored.

The shape property states that a binary heap is a complete binary tree, where all levels except possibly the last level are completely filled, and all nodes in the last level are as far left as possible. To check the shape property of a binary heap, we can count the number of nodes at each level of the tree, and verify that the number of nodes at the last level is between 1 and 2^(h-1), where h is the height of the tree. If the shape property is violated, we can swap nodes in the tree to restore the complete binary tree structure.

Overall, maintaining the heap property and shape property requires careful manipulation of the binary heap structure during insertion, deletion, and other heap operations, to ensure that the desired properties are maintained at all times.

## 25.2 Describe heap operations using pseudocode

Here is pseudocode for the main heap operations:

### 25.2.1 Max-Heapify(A, i):

```
left = 2i
right = 2i + 1
largest = i

if left <= A.heapsize and A[left] > A[largest]:
    largest = left

if right <= A.heapsize and A[right] > A[largest]:
    largest = right

if largest != i:
    swap A[i] and A[largest]
    Max-Heapify(A, largest)
```

### 25.2.2 Build-Max-Heap(A)

```
A.heapsize = A.length
for i = floor(A.length/2) downto 1:
    Max-Heapify(A, i)
```

### 25.2.3 Heap-Extract-Max(A)

```
if A.heapsize < 1:
    error "Heap underflow"

max = A[1]
A[1] = A[A.heapsize]
A.heapsize = A.heapsize - 1
Max-Heapify(A, 1)

return max
```

### 25.2.4 Heap-Increase-Key(A, i, key):

```
if key < A[i]:
    error "New key is smaller than current key"

A[i] = key
while i > 1 and A[i/2] < A[i]:
```

```
    swap A[i/2] and A[i]
    i = i/2
```

### 25.2.5 Max-Heap-Insert(A, key)

```
A.heapsize = A.heapsize + 1
A[A.heapsize] = -infinity
Heap-Increase-Key(A, A.heapsize, key)
```

These operations allow for the manipulation of a binary heap data structure, maintaining the heap property and shape property as elements are added and removed from the heap.

## 25.3 Implement heapsort using a heap

Heapsort is an efficient in-place sorting algorithm that uses the binary heap data structure to sort an array of elements. Here's how to implement heapsort using a heap:

1. Build a max-heap from the array by calling Build-Max-Heap(array).
2. Swap the first element of the array (which is the largest element) with the last element.
3. Decrement the heap size of the array by 1.
4. Call Max-Heapify(array, 1) to maintain the heap property of the remaining elements in the heap.
5. Repeat steps 2-4 for the remaining elements in the array (from n-1 down to 2).
6. The array is now sorted in ascending order.

Here is the pseudocode for heapsort:

```
Heapsort(A):
    Build-Max-Heap(A)
    for i = A.length downto 2:
        swap A[1] and A[i]
        A.heapsize = A.heapsize - 1
        Max-Heapify(A, 1)
```

This implementation of heapsort has a time complexity of O(n log n), where n is the number of elements in the array. The space complexity of heapsort is O(1), as it sorts the array in-place without requiring any additional memory.

# Chapter 26

# Graphs

## 26.1 Graph representations

A graph is a collection of nodes (also known as vertices) and edges that connect these nodes. Graphs are used to model relationships between objects or data points. The key concepts in a graph include:

- **Nodes or vertices**: These are the points in the graph that represent objects or data points.
- **Edges**: These are the connections between nodes that represent relationships between objects or data points. Edges can be directed (pointing in a specific direction) or undirected (bi-directional).
- **Weight**: Some graphs have weights assigned to their edges to represent the cost, distance, or other quantitative measure of the relationship between nodes.
- **Paths**: A path in a graph is a sequence of edges that connect a sequence of nodes.
- **Cycles**: A cycle is a path that starts and ends at the same node.

Graphs can be represented in various ways, including:

- **Adjacency matrix**: A two-dimensional array where the rows and columns represent nodes, and the values in the matrix indicate whether there is an edge between the nodes.
- **Adjacency list**: A list of lists where each list represents a node, and the elements in each list are the nodes that are adjacent to that node.
- **Edge list**: A list of tuples where each tuple represents an edge and contains the two nodes that the edge connects.

Each representation has its own strengths and weaknesses depending on the size and nature of the graph, as well as the specific use case.

## 26.2 Minimum spanning tree

A minimum spanning tree (MST) is a subset of the edges in an undirected, weighted graph that connects all of the vertices and has the minimum possible total edge weight. In other words, a minimum spanning tree is a tree that connects all the vertices of the graph while minimizing the total weight of the edges.

MSTs have many practical applications, such as in network design, transportation planning, and resource allocation. Finding the minimum spanning tree of a graph can be done using algorithms such as Kruskal's algorithm, Prim's algorithm, and Boruvka's algorithm.

MSTs have several properties, such as:

- An MST is always a tree, which means it does not contain any cycles.
- An MST must connect all the vertices in the graph.
- If the graph has multiple MSTs, then they will have the same total weight.
- If the graph has unique edge weights, then the MST is also unique.

Finding the minimum spanning tree of a graph is an important problem in graph theory with many practical applications.

## 26.3 Shortest path finding

Shortest path finding is the problem of finding the shortest path between two nodes in a graph. This problem has many applications in areas such as transportation planning, network routing, and computer networking.

In a weighted graph, the shortest path between two nodes is the path with the lowest total weight. The weight of a path is the sum of the weights of all the edges in the path. Shortest path finding algorithms can be classified as either single-source or all-pairs algorithms.

Single-source algorithms find the shortest path from a single source node to all other nodes in the graph. Common single-source algorithms include Dijkstra's algorithm and Bellman-Ford algorithm.

All-pairs algorithms find the shortest path between all pairs of nodes in the graph. The most well-known all-pairs algorithm is the Floyd-Warshall algorithm.

Shortest path finding algorithms can be implemented using a variety of data structures, such as priority queues, heaps, and adjacency lists. The choice of

data structure depends on the specific algorithm being used and the characteristics of the graph being analyzed.

Overall, shortest path finding is an important problem in graph theory and has many practical applications in various fields.

# Chapter 27

# Learning outcomes

## 27.1 Use different implementations of graphs (matrix adjacency, list adjacency. Edge list)

Here are some examples of how to use different implementations of graphs:

**Matrix adjacency**: In this implementation, the graph is represented as a 2D matrix where each row and column represents a vertex, and the entries in the matrix represent the edges between vertices. For example, if we have a graph with 4 vertices and edges (1,2), (1,3), and (2,4), the matrix representation would look like this:

```
    1   2   3   4
1   0   1   1   0
2   1   0   0   1
3   1   0   0   0
4   0   1   0   0
```

**List adjacency**: In this implementation, the graph is represented as a list of lists, where each vertex has a list of its adjacent vertices. For example, if we have the same graph as before, the list representation would look like this:

```
graph = [[2,3],
         [1,4],
         [1],
         [2]]
```

**Edge list**: In this implementation, the graph is represented as a list of tuples,

where each tuple represents an edge in the graph. For example, if we have the same graph as before, the edge list representation would look like this:

```
graph = [(1,2),         (1,3),         (2,4)]
```

These are just a few examples of how to implement and use different representations of graphs. The choice of implementation depends on the specific problem being solved and the characteristics of the graph being analyzed.

## 27.2 Implement Prim's algorithm to find the minimum spanning tree

Here is an example of how to implement Prim's algorithm in pseudocode to find the minimum spanning tree of a graph:

```
Prim's algorithm (graph G, starting node s):
    Let S be the set of visited vertices, initially empty
    Let Q be the set of vertices in G, sorted by distance from s
    Let D be an array of distances, initialized to infinity for all vertices except
    Let P be an array of parent vertices, initially null for all vertices

    While Q is not empty:
        Let u be the vertex in Q with the minimum distance
        Remove u from Q and add it to S
        For each vertex v adjacent to u:
            If v is not in S and the weight of (u,v) is less than D[v]:
                Set D[v] to the weight of (u,v)
                Set P[v] to u
                Update the position of v in Q based on its new distance

    Return the set of edges {P[v], v} for all vertices v
```

In this algorithm, we maintain three data structures: S, which represents the set of visited vertices; Q, which is a priority queue of vertices sorted by distance from the starting node s; and D and P, which are arrays used to keep track of the distances and parent vertices for each vertex.

The algorithm starts by initializing the data structures, and then repeatedly selects the vertex with the minimum distance from Q, adds it to S, and updates the distances and parent vertices for its adjacent vertices if necessary. The algorithm terminates when all vertices have been visited.

Finally, the algorithm returns the set of edges {P[v], v} for all vertices v, which represents the minimum spanning tree of the graph.

Note that the specific implementation details, such as the choice of priority queue and the method for updating the position of a vertex in Q, may vary depending on the programming language and the specific use case.

## 27.3 Implement Dijkstra's algorithm to find the shortest path between two nodes

Here's an example of how to implement Dijkstra's algorithm in pseudocode to find the shortest path between two nodes in a graph:

"'textDijkstra's algorithm (graph G, starting node s, ending node t): Let Q be a priority queue of vertices, sorted by distance from s Let D be an array of distances, initialized to infinity for all vertices except s, which is initialized to 0 Let P be an array of parent vertices, initially null for all vertices Add s to Q

```
While Q is not empty:
    Let u be the vertex in Q with the minimum distance
    Remove u from Q
    If u is t, return the path from s to t using P
    For each vertex v adjacent to u:
        Let w be the weight of the edge (u, v)
        If D[u] + w < D[v]:
            Set D[v] to D[u] + w
            Set P[v] to u
            If v is not in Q, add it to Q

If t was not reached, return null (there is no path from s to t)
```

"'

In this algorithm, we maintain three data structures: Q, which is a priority queue of vertices sorted by distance from the starting node s; D, which is an array used to keep track of the distances for each vertex; and P, which is an array used to keep track of the parent vertices for each vertex.

The algorithm starts by initializing the data structures and adding the starting node s to the priority queue. It then repeatedly selects the vertex with the minimum distance from Q, removes it from Q, and updates the distances and parent vertices for its adjacent vertices if necessary. The algorithm terminates when the ending node t is reached or when Q is empty.

Finally, if the ending node t was reached, the algorithm returns the path from s to t using the parent vertices in P. Otherwise, it returns null to indicate that there is no path from s to t.

Note that the specific implementation details, such as the choice of priority queue and the method for updating the position of a vertex in Q, may vary depending on the programming language and the specific use case.