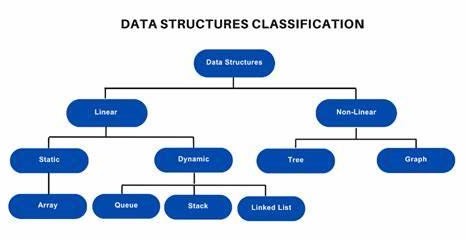
# ICS2105 DATA STRUCTURES AND ALGORITHMS PORFOLIO



**Name:ABDIWASA MOHAMED IBRAHIM.**

**RegNo:** SCT221-C004-0651/2021.

**Purpose Statement:**

This portfolio assignment is being presented as a partial requirement for obtaining the Bachelor of Information Technology degree from JKUAT (NCBD-Campus).

**Supervisor:** Mr. John Ahenda **May - August 2023**

## Contents

[ICS2105 DATA STRUCTURES AND ALGORITHMS PORFOLIO 1](#_bookmark0)

1. [Introduction Of Data Structures and Algorithms 3](#_bookmark1)
   1. [Overview of Data Structures and Algorithms 3](#_bookmark2)
   2. [Fundamental Data Structures 3](#_bookmark3)
      1. [Array Operations 3](#_bookmark4)
      2. [Linked List Operations 4](#_bookmark5)
      3. [Stack Operations 4](#_bookmark6)
      4. [Queue Operations 5](#_bookmark7)
   3. [Algorithm Design and Techniques 5](#_bookmark8)
2. [Advanced Data Structures 7](#_bookmark9)
   1. [Trees 7](#_bookmark10)
   2. [Heaps 7](#_bookmark11)
   3. [Graphs 8](#_bookmark12)
   4. [Priority Queues 8](#_bookmark13)
   5. [Hashing 9](#_bookmark14)
3. [Algorithm Analysis 9](#_bookmark15)
   1. [Time Complexity 9](#_bookmark16)
   2. [Space Complexity 10](#_bookmark17)
   3. [Asymptomatic Notation 10](#_bookmark18)
   4. [Worst-Case and Average-Case Analysis 11](#_bookmark19)
4. [Advanced Algorithms 11](#_bookmark20)
   1. [Sorting and Searching Algorithms 11](#_bookmark21)
   2. [Greedy Algorithms 12](#_bookmark22)
   3. [Dynamic Programming 13](#_bookmark23)
   4. [Divide and Conquer 13](#_bookmark24)

[Bibliography 14](#_bookmark25)

1. Introduction Of Data Structures and Algorithms

### Overview of Data Structures and Algorithms

##### **Importance of Data Structures and Algorithms in Computer Science**

Data structures and algorithms are fundamental concepts in computer science that play a crucial role in solving complex problems efficiently. They form the backbone of computer programs and are essential for developing efficient software systems. Here are some key reasons why data structures and algorithms are important in computer science:

i.Efficiency: Data structures and algorithms allow programmers to design efficient solutions to problems. By choosing appropriate data structures and implementing efficient algorithms, it is possible to optimize resource usage such as time and memory. Efficient algorithms can significantly reduce execution time, making software faster and more responsive.

ii.Problem Solving: Data structures and algorithms provide a framework for problem-solving. They enable programmers to represent and manipulate data effectively, facilitating the development of efficient algorithms for various tasks. Understanding different data structures and algorithms helps programmers analyze problems, devise appropriate strategies, and design effective solutions.

iii.Scalability: As the size of the data and complexity of problems increase, the efficiency of algorithms and data structures becomes critical. Scalable algorithms and data structures ensure that the performance of a program does not deteriorate significantly with increasing input sizes. This is crucial for handling large datasets, processing massive amounts of information, and building robust systems that can handle real-world demands.

iv.Code Reusability: Data structures and algorithms provide reusable building blocks that can be applied to various problems. Once implemented, they can be used across different applications, saving time and effort. Many algorithms and data structures are widely available as libraries or modules, allowing programmers to leverage existing solutions and focus on the specific problem at hand.

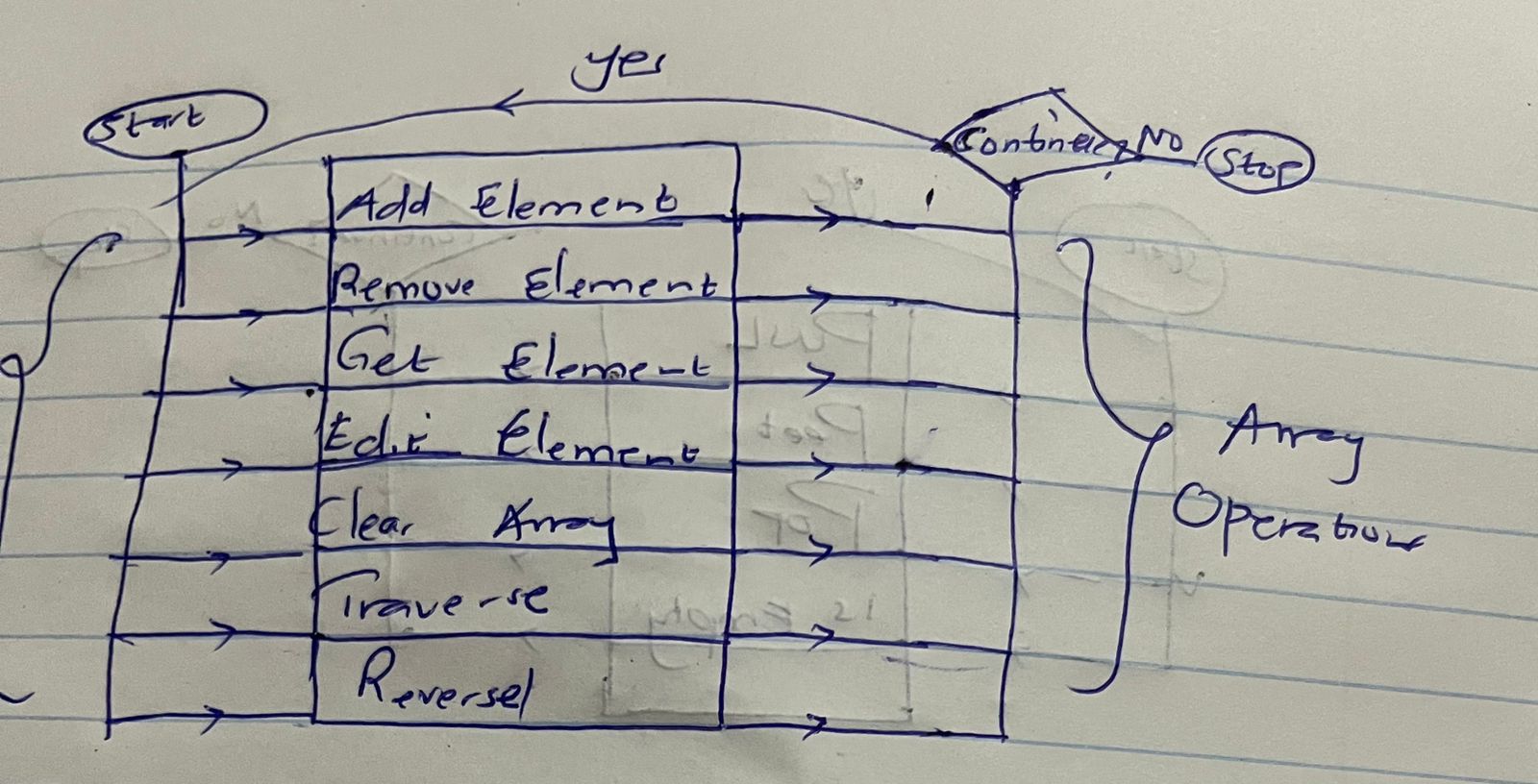
v.Optimization: Data structures and algorithms play a vital role in optimizing various aspects of software development. They help optimize memory usage, reduce computational complexity, and minimize redundant operations. Efficient algorithms can significantly impact the performance of software systems, making them more responsive, scalable, and resource-efficient.

vi.Core Knowledge in Computer Science: Data structures and algorithms are considered fundamental concepts in computer science education. They provide a solid foundation for understanding other advanced topics such as operating systems, databases, artificial intelligence, and more. Proficiency in data structures and algorithms is highly valued in the software industry and is often assessed in technical interviews and coding assessments.

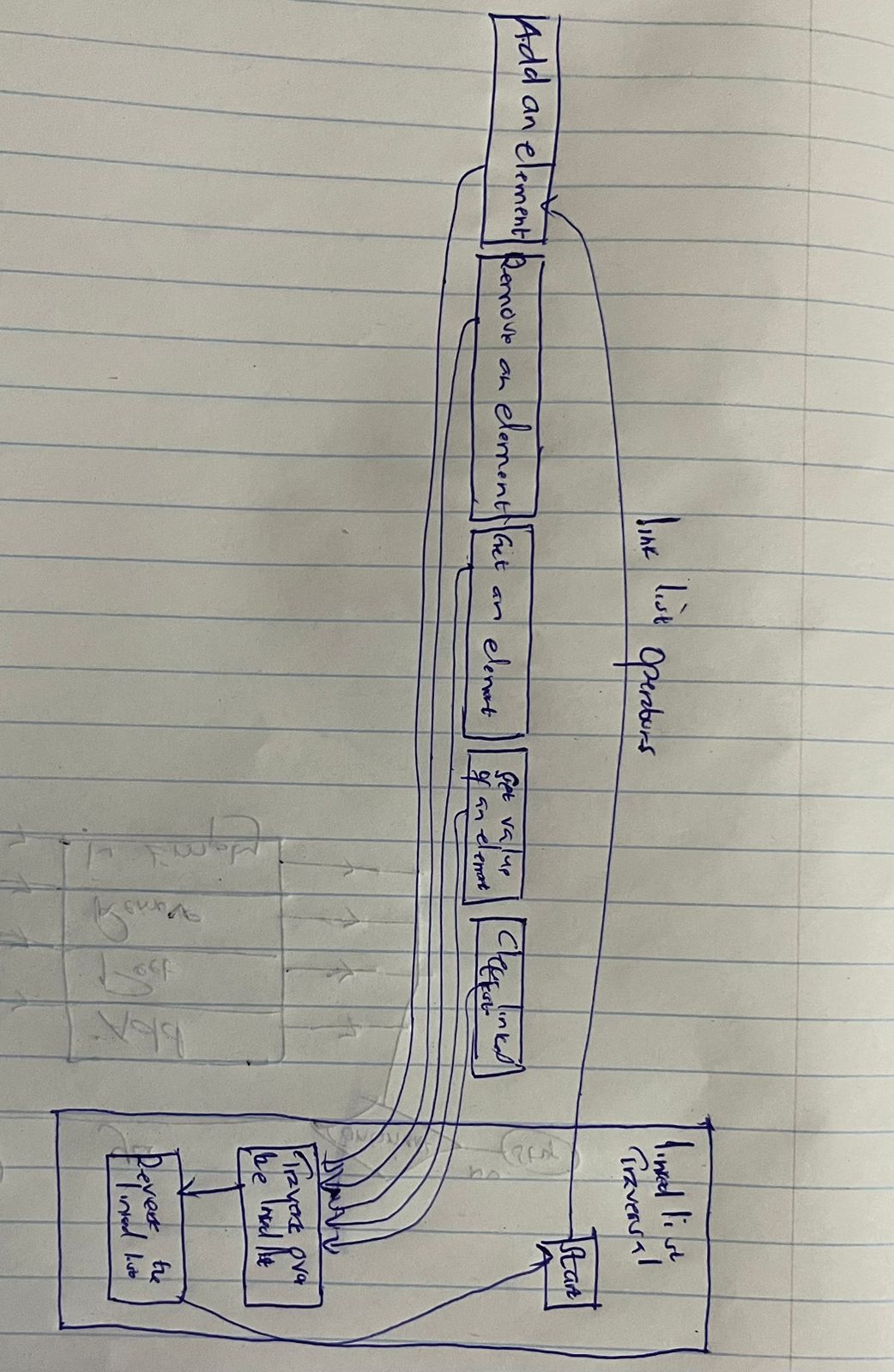
### Fundamental Data Structures

**flowchart illustrating the different operations that can be performed on each of the fundamental data structures: arrays, linked lists, stacks, queues.**

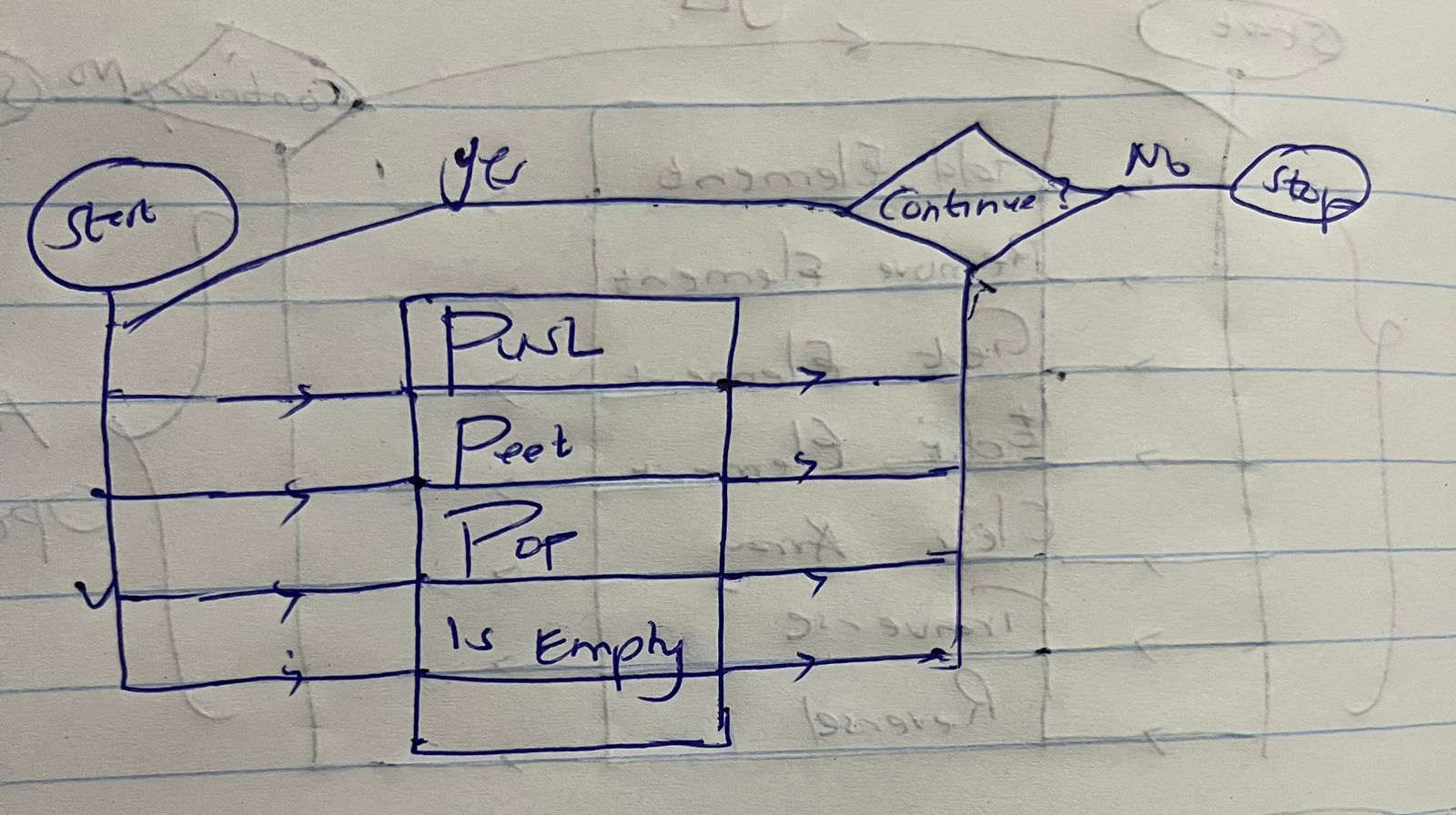
#### Array Operations



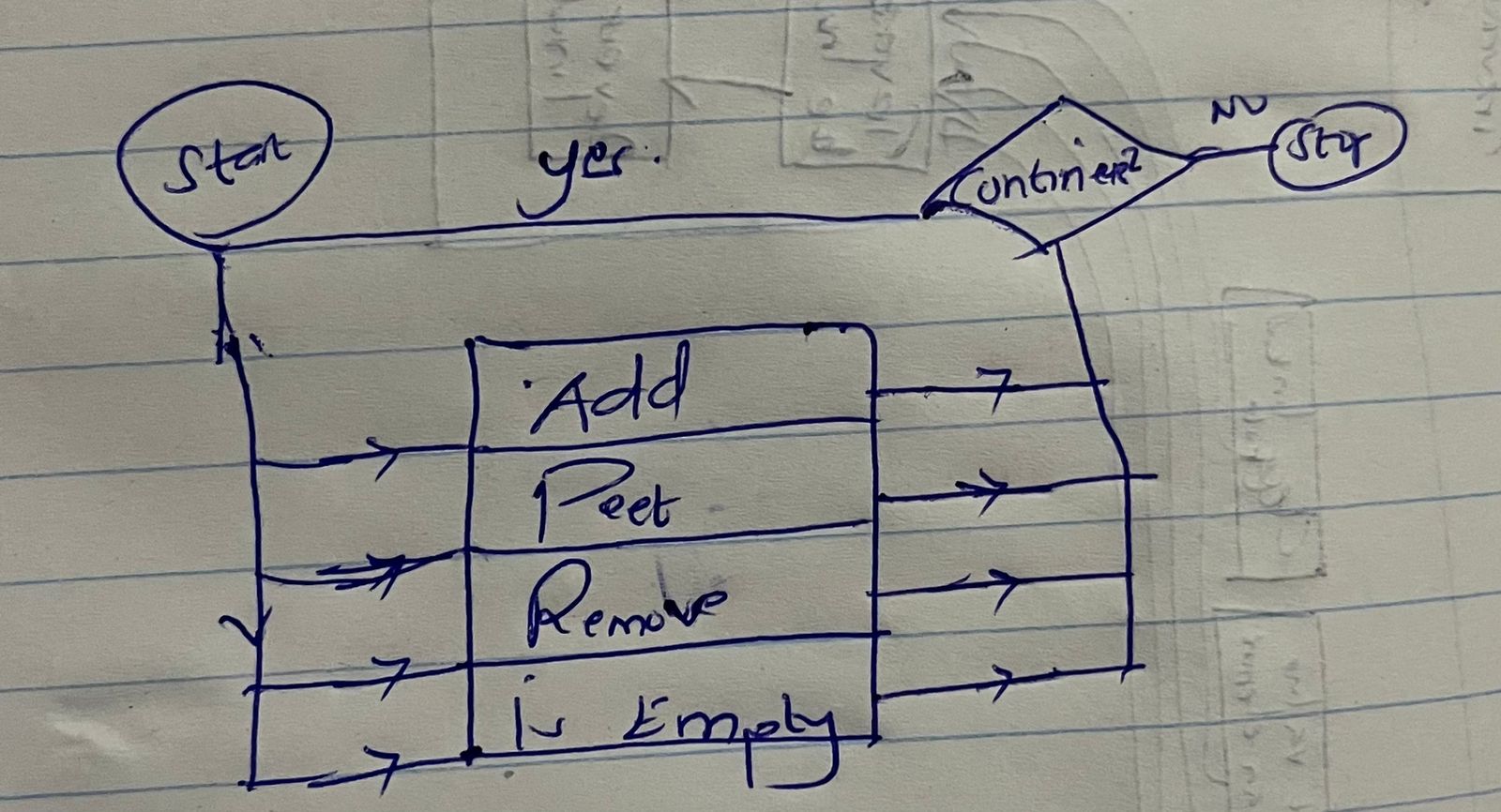
#### Linked List Operations



#### Stack Operations



#### Queue Operations



### Algorithm Design and Techniques

* + 1. **Greedy algorithms:** These algorithms make locally optimal choices at each step to achieve a globally optimal solution. For example, the minimum spanning tree algorithm adds edges with the lowest weights without creating cycles.
    2. **Divide and conquer algorithms:** They break down problems into smaller subproblems, solve them recursively, and then combine the solutions to solve the original problem. An example is the merge sort algorithm that divides an array, recursively sorts each half, and merges them together.
    3. **Dynamic programming:** This technique solves problems by breaking them into smaller subproblems and using the solutions to solve the original problem. It is useful for finding optimal solutions, like the shortest path between two points in a graph.
    4. **Branch and bound:** It explores a tree of potential solutions to optimization problems. Starting from the root, it explores each branch until a better solution is found. It is commonly used for finding minimum or maximum values of a function.
    5. **Backtracking:** This technique explores all possible solutions to a problem, backtracking when an invalid solution is encountered. It is employed to find all possible solutions, such as combinations of items in a set.
    6. **Randomized algorithms:** These algorithms leverage randomness to solve problems and can be more efficient than deterministic algorithms for certain cases.
    7. **Network flow:** It models the flow of resources through a network, representing problems like traffic flow or power distribution.
    8. **Linear programming:** This optimization technique solves problems represented as systems of linear equations and inequalities. It is used for finding minimum costs or maximum profits in scenarios such as transportation or production.
    9. **Integer programming:** This optimization technique handles problems where some

variables must be integers within a system of linear equations and inequalities. Examples include determining the minimum number of machines needed or the maximum number of students assigned to a class.

* + 1. **Approximation algorithms:** These algorithms provide solutions that are not necessarily optimal but are close to optimal. They are often employed for NP-hard problems, where finding an optimal solution in polynomial time is not known.
    2. **Brute force:** This technique attempts all possible solutions to a problem. It is suitable for small problems or cases where no better algorithm exists.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Pros** | **Cons** |
| Greedy algorithms | Easy to understand and  implement |  |
| Divide and conquer algorithms | Efficient for some problems |  |
| Dynamic programming | Can be very efficient for some  problems |  |
| Branch and bound | Can find the optimal solution for  some problems |  |
| Backtracking | Can find all possible solutions |  |
| Randomized algorithms | Can be more efficient than deterministic algorithms for  some problems |  |
| Network flow | Can be used to model problems  such as traffic flow, electrical power flow, and water flow |  |
| Linear programming | Can be used to solve problems such as finding the minimum cost of a transportation problem or the maximum profit of a  production problem |  |
| Integer programming | Can be used to solve problems such as finding the minimum number of machines needed to produce a product or the maximum number of students  that can be assigned to a class |  |
| Approximation algorithms | Can provide a solution to a problem that is not necessarily  optimal, but is close to optimal |  |
| Brute force | Can be used for problems that are small or for problems where there is no known better  algorithm |  |

**Bibliography**

<https://www.geeksforgeeks.org/fundamentals-of-algorithms/> accessed on 27/05/2022

**NOTES REFERENCE**

i) <https://plantuml.com/>

ii) <https://mermaid.js.org/>

## 2.Advanced Data Structures

### Trees

Trees optimize time complexity in algorithms in various ways. They can be used to represent and process data efficiently, store frequently accessed data, and implement algorithms like binary search, depth-first search, and breadth-first search.

For example, using a binary tree to represent a list of numbers enables quick finding and updating of the smallest and largest numbers. Storing a large number of words in a tree reduces access time and memory usage.

Examples:

**Binary search:** Implemented with a binary tree, this algorithm finds the index of an element in a sorted array in O(log n) time complexity.

**Depth-first search:** Implemented with a stack, it traverses all nodes in a tree in O(n) time complexity, exploring each node's children until no more are left.

**Breadth-first search:** Implemented with a queue, it traverses all nodes in a tree in a level-order manner, taking O(n) time complexity.

### Heaps

##### Advantages and Disadvantages of Using Heaps to Store Data Advantages:

* **Quick access to minimum or maximum element:** Heaps enable rapid retrieval of the minimum or maximum element in the data structure as it is always located at the root.
* **Efficient insertion and deletion:** Inserting and deleting elements in a heap can be accomplished in O(log n) time, where n represents the number of elements. This efficiency is due to the maintained heap structure after each operation.
* **Compact storage:** Heaps can be stored in a compact manner, optimizing space usage. The balanced nature of heaps ensures there are no empty nodes.

##### Disadvantages:

* **Unsuitability for certain data types:** Heaps can solely store data that can be compared, rendering them unsuitable for non-comparable data such as strings or objects.
* **Limited applicability for certain operations:** Heaps are not well-suited for all operations. For example, they are not designed to find the middle element within the data structure.
* **Higher complexity compared to other data structures:** Heaps possess a greater level of complexity compared to simpler data structures like arrays or linked lists. Consequently, implementing and comprehending heaps may require more effort.

### Graphs

##### Applications of Graph Theory That Can Be Used to Solve Real-World Problems

1. **Social networks:** Graph theory is employed to represent social networks, where individuals are nodes and relationships are edges. This modeling allows the

examination of information or disease propagation and the identification of influential individuals.

1. **Transportation networks:** Graph theory is utilized to model transportation networks, with cities or points of interest as nodes and roads or transportation links as edges. This facilitates finding the shortest path between locations and optimizing traffic routing.
2. **Computer networks:** Graph theory is applied to model computer networks, where computers are nodes and network connections are edges. This aids in determining efficient data routing and identifying potential security weaknesses.
3. **Scheduling problems:** Graph theory is employed in modeling scheduling problems, where tasks are nodes and precedence constraints are edges. This assists in finding optimal schedules and detecting potential conflicts.

Additionally, graph theory finds further applications such as:

* + Finding the shortest path between two points.
  + Determining the most efficient way to route traffic in a network.
  + Identifying potential security vulnerabilities in a network.
  + Finding the optimal schedule for a set of tasks.
  + Detecting potential conflicts in a schedule.
  + Modeling social networks.
  + Modeling transportation networks.
  + Modeling computer networks.

### Priority Queues

##### How to Use Priority Queues to Prioritize Tasks in an Efficient Manner

Priority queues offer an efficient method to prioritize tasks by storing them in a data structure that allows for quick sorting based on their priority. Consequently, the task with the highest priority is always processed next.

There are various approaches to implement priority queues, with one common method being the utilization of a binary heap. A binary heap organizes elements in a tree-like structure,

ensuring that they are consistently sorted according to their priority, with the highest priority element residing at the root of the tree.

Adding a new task to a priority queue involves simply inserting it into the tree, while removing the next task for processing entails removing the root of the tree and rebalancing it.

Priority queues find applicability across diverse domains for task prioritization. For instance, they are valuable in operating systems to prioritize processes awaiting execution. Additionally, they are utilized in web servers to prioritize requests awaiting processing.

### Hashing

##### Best Practices For Implementing Hashing Functions In a Production Environment

1. **Choose a reputable hashing function:** Not all hashing functions are equal in terms of security and efficiency. It is crucial to select a well-established and trusted hashing function that has undergone rigorous scrutiny for security vulnerabilities.
2. **Utilize a secure salt:** A salt, which is a random string appended to the input of a hashing function, aids in preventing collisions. Collisions occur when different inputs produce the same hash output. It is essential to use a unique and secure salt for each input to

enhance security.

1. **Opt for a secure hash output size:** The hash output size refers to the number of bits in the output of the hashing function. A larger hash output size increases the difficulty of brute-forcing the hash output. It is important to select a hash output size that is

sufficiently large to ensure adequate security.

1. **Safeguard the hash output storage:** Store the hash outputs in a secure location that is inaccessible to unauthorized individuals. This involves employing robust security

measures such as strong passwords and encryption when storing the hash outputs, such as in a protected database.

1. **Monitor hash outputs for compromises:** Vigilantly monitor the hash outputs for any

signs of compromise. This involves observing the hash outputs for unusual patterns that may indicate attacks against the hash function. In the event of suspicious activity, take

immediate action to safeguard your data.

## 3.Algorithm Analysis`

## a)Time Complexity

**Use Time Complexity To Optimize The Performance Of Our Algorithm.**

Time complexity is a measure used in algorithm analysis to estimate the amount of time an algorithm takes to run as a function of the input size. It provides a way to compare and evaluate the efficiency of different algorithms.

Time complexity is typically expressed using big O notation, which represents the upper bound on the growth rate of an algorithm's runtime. It provides an asymptotic approximation of the worst-case behavior of an algorithm.

Here are some common time complexity notations used in algorithm analysis, listed from best to worst:

**O(1) - Constant Time Complexity:**

An algorithm with constant time complexity takes the same amount of time regardless of the input size. It means that the algorithm's runtime is not dependent on the input size.

**O(log n) - Logarithmic Time Complexity**: Algorithms with logarithmic time complexity exhibit a runtime that grows logarithmically with the input size. These algorithms often divide the input in half at each step, such as in binary search.

**O(n) - Linear Time Complexity**:

Algorithms with linear time complexity have a runtime that grows linearly with the input size. These algorithms typically process each input element once.

**O(n log n) - Linearithmic Time Complexity**:

Algorithms with linearithmic time complexity have a runtime that grows linearly with the input size multiplied by the logarithm of the input size. Common examples include efficient sorting algorithms like Merge Sort and Quick Sort.

O(n^2) - Quadratic Time Complexity:

Algorithms with quadratic time complexity have a runtime that grows quadratically with the input size. These algorithms often involve nested loops.

### b)Space Complexity

##### **Strategies That Can Be Used To Minimize Space Complexity In Our Algorithm**

1. Utilize data structures with constant space requirements. Several data structures, such as linked lists and hash tables, have a constant space requirement. Employing these data structures can effectively reduce the space complexity of your algorithm.
2. Minimize the use of temporary variables. Temporary variables are utilized to store

intermediate results. By avoiding excessive use of temporary variables, you can reduce the space complexity of your algorithm.

1. Embrace in-place algorithms. In-place algorithms operate without requiring additional space to store intermediate results. Employing in-place algorithms can effectively

minimize the space complexity of your algorithm.

1. Opt for space-efficient algorithms. Numerous algorithms are specifically designed to be space-efficient. Employing these algorithms can effectively minimize the space complexity of your algorithm

### 

### c)Asymptomatic Notation

##### **How To Decide When to Use Which Asymptotic Notation**

1. **Big O notation:** Big O notation is utilized to portray the worst-case runtime of an algorithm. Use big O notation when you want to express the maximum time or space requirement of an algorithm, regardless of whether the actual performance matches this upper bound in every case.
2. **Theta notation:** Theta notation is utilized to portray the average-case runtime of an algorithm. Use big Theta notation when you want to express a tight range of time or space complexity, indicating that an algorithm's performance is both upper and lower bounded by the given function.
3. **Omega notation:** Omega notation is utilized to portray the best-case runtime of an algorithm. Use big Omega notation when you want to express the minimum time or space requirement of an algorithm, regardless of whether the actual performance matches this lower bound in every case.

When deciding which asymptotic notation to use, consider the following factors:

What information do you want to convey about the algorithm's performance? Upper bound (big O), lower bound (big Omega), or a tight range (big Theta)?

Do you need to describe the worst-case, best-case, or average-case behavior of the algorithm?

What level of precision and rigor is required in your analysis?

### d)Worst-Case and Average-Case Analysis

##### Advantages And Disadvantages of Using Worst-Case and Average-Case Analysis

**Worst-case analysis:**

##### Advantages:

* Worst-case analysis is a cautious approach to estimating the running time of an algorithm, making it more likely to provide an accurate estimate.
* It is a straightforward and easy-to-understand method of analysis.

##### Disadvantages:

* Worst-case analysis tends to be pessimistic and may overestimate the running time of the algorithm in many scenarios.
* It does not consider the distribution of the input data, which can lead to inaccuracies when the data is not evenly distributed**.**

##### Average-case analysis:

**Advantages:**

* Average-case analysis is often more accurate than worst-case analysis since it considers the distribution of the input data.
* It enables the selection of the best algorithm for a given problem by considering the expected performance.

##### Disadvantages:

* Average-case analysis can be more challenging to comprehend and implement compared to worst-case analysis.
* It may yield inaccurate results when the input data is not evenly distributed.

## 4.)Advanced Algorithms

### a)Sorting and Searching Algorithms

**Using Sorting and Searching Algorithms To Improve The Efficiency Of A Program**

Sorting and searching algorithms play a crucial role in improving the efficiency of programs that deal with large amounts of data. By utilizing efficient sorting and searching algorithms, you can optimize the performance and responsiveness of your program. Here are some ways in which sorting and searching algorithms can enhance program efficiency:

Sorting Algorithms:

Sorting algorithms arrange data in a particular order, making it easier to search, retrieve, and process. By using efficient sorting algorithms, you can improve the efficiency of subsequent operations that rely on sorted data.

a) Binary Search:

Sorting the data allows you to apply binary search, which is an efficient searching algorithm. Binary search operates on sorted data and has a time complexity of O(log n), significantly faster than linear search.

b) Merge Sort or Quick Sort:

Merge sort and quick sort are commonly used sorting algorithms with an average time complexity of O(n log n). These algorithms efficiently sort large datasets and enable faster subsequent operations.

Searching Algorithms:

Searching algorithms help locate specific elements or patterns within a dataset. By using efficient searching algorithms, you can reduce the time taken to find desired information, improving overall program efficiency.

1. Binary Search:

Binary search is a commonly used searching algorithm that operates on sorted data. It efficiently finds the target element by dividing the search space in half at each step, resulting in a time complexity of O(log n).

1. Hashing and Hash-based Searching:

Hashing is a technique that maps data to a specific index using a hash function. Hash-based searching, such as using hash tables or hash maps, enables quick lookup operations with an average time complexity of O(1) or close to it.

1. Tree-based Searching:

Tree-based searching algorithms, like binary search trees (BSTs) or balanced search trees (e.g., AVL trees, Red-Black trees), provide efficient search operations with a time complexity of O(log n) on average. These structures are particularly useful when dealing with dynamically changing or sorted data.

Optimized Data Structures:

Sorting and searching algorithms often rely on efficient data structures to enhance program efficiency. Choosing the right data structure for your specific requirements can significantly impact the overall performance of your program.

Priority Queues and Heaps:

Priority queues, implemented using heap data structures (e.g., binary heaps or Fibonacci heaps), facilitate efficient processing of elements based on priority. They are commonly used in sorting, scheduling, and graph algorithms, improving program efficiency.

1. Balanced Trees:

Balanced trees, such as AVL trees or Red-Black trees, provide efficient searching and insertion operations with a time complexity of O(log n). These structures ensure the tree remains balanced, enabling optimized searching and retrieval.

1. Hash Tables:

Hash tables provide constant-time average case lookup, insertion, and deletion operations (O(1)). They are efficient for storing and retrieving key-value pairs, enhancing the efficiency of various operations that rely on quick data access.

### b)Greedy Algorithms

##### Key Characteristics of a Greedy Algorithm

A greedy algorithm is an algorithm that makes the best possible choice at each step,

considering the immediate options, in the expectation of achieving the overall best outcome. These algorithms are commonly employed to address optimization problems, which involve tasks like determining the shortest route between two locations or identifying the minimum spanning tree in a graph.

The following are characteristics of a Greedy Algorithm: -

1. Local optimality refers to the characteristic of a greedy algorithm where it selects the best available option at each step based on the currently known information. It means that at each stage, the choice made by the algorithm is the most favorable among the available options.
2. While the greedy algorithm aims for local optimality, it does not guarantee finding the global optimum in all cases. However, it frequently manages to obtain a satisfactory solution to the problem at hand.
3. Moreover, greedy algorithms often demonstrate efficiency, enabling them to tackle substantial problems within a reasonable timeframe. They are capable of efficiently handling large-scale problems.

### c)Dynamic Programming

##### Difference Between Dynamic Programming and Other Programming Paradigms

Dynamic programming is a programming approach that involves dividing a problem into smaller subproblems and storing their solutions in a table. By referencing the precomputed solutions from the table, the program can efficiently determine the solution to the original problem.

Dynamic programming distinguishes itself from other programming approaches in several aspects. Firstly, it necessitates the breakdown of the problem into smaller subproblems. Secondly, it employs a table to store the solutions to these subproblems. Lastly, dynamic programming can tackle problems that would be challenging to solve using alternative programming paradigms.

### C)Divide and Conquer

##### Examples Of Problems That Can Be Solved Using Divide and Conquer Algorithms

1. **Multiplication:** Divide and conquer techniques can be employed to multiply two large numbers by breaking them down into smaller subproblems, recursively solving and combining these subproblems to obtain the final result.
2. **Sorting:** Divide and conquer algorithms can be utilized to sort a list of numbers by recursively dividing the list into halves, sorting each half independently, and then merging the sorted halves.
3. **Searching:** Divide and conquer strategies can be applied to search for an element in a list by recursively dividing the list into halves, narrowing down the search to the half where the element is more likely to be located.
4. **Finding the maximum or minimum element in a list:** Divide and conquer methods can be used to find the maximum or minimum element in a list by recursively dividing the

list into halves, finding the maximum or minimum in each half, and then comparing the results to determine the overall maximum or minimum.

1. **Finding the longest common subsequence of two strings:** Divide and conquer

algorithms can be utilized to find the longest common subsequence of two strings by recursively dividing the strings into halves, finding the longest common subsequences of the subparts, and combining them to obtain the longest common subsequence.

#### Bibliography

##### Sources

1. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. Algorithms by Robert Sedgewick and Kevin Wayne.
3. Grokking Algorithms by Aditya Bhargava.
4. The Algorithm Design Manual by Steven S. Skiena.
5. Data Structures and Algorithms in Java by Robert Lafore.

##### Websites

1. W3Schools: https:[//w](http://www.w3schools.com/)ww[.w3schools.com/](http://www.w3schools.com/)
2. freeCodeCamp: https:[//w](http://www.freecodecamp.org/)ww[.freecodecamp.org/](http://www.freecodecamp.org/)
3. GeeksforGeeks: https://[www.geeksforgeeks.org/](http://www.geeksforgeeks.org/)