INTERNSHIP REPORT

A report submitted in partial fulfillment of the requirements for the Award of Degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING
by
ROBERT LOITONGBAM
Regd. No.: 963321104043

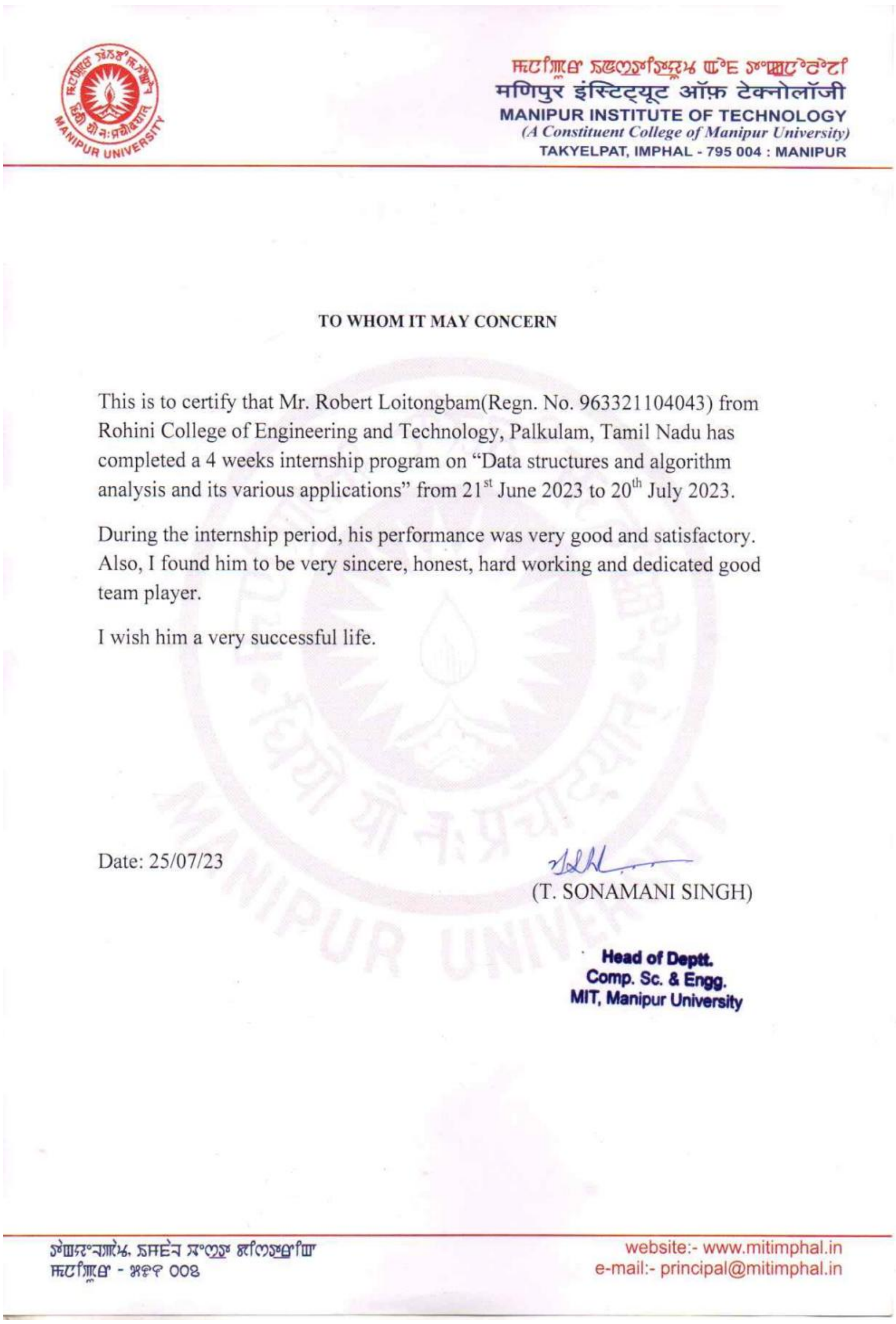
Under Supervision of
Mr. T.Sonamani Singh, HoD
Comp.Sc.&Engg.,MIT,
Manipur University,Imphal.
(Duration: 21 June, 2023 to 20 July, 2023)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
(An ISO Certified Institution)
Aproved by AICTE and affiliated to Anna University,Chennai
KANYAKUMARI,TAMIL NADU
2021 - 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY
(An ISO Certified Institution)
PALKULAM, KANYAKUMARI

CERTIFICATE

This is to certify that the "**Internship report**" submitted by **ROBERT LOITONGBAM (Regd. No.:963321104043)** is work done by him and submitted during 2022 – 2023 academic year, in partial fulfillment of the requirements for the award of the degree of **BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING,** at **Manipur Institute of Technology, Takyelpat, Imphal .**



ꯃꯅꯤꯄꯨꯔ ꯏꯅꯁ꯭ꯇꯤꯇ꯭ꯌꯨꯠ ꯑꯣꯐ ꯇꯦꯛꯅꯣꯂꯣꯖꯤ
मणिपुर इंस्टिट्यूट ऑफ़ टेक्नोलॉजी
MANIPUR INSTITUTE OF TECHNOLOGY
(A Constituent College of Manipur University)
TAKYELPAT, IMPHAL - 795 004 : MANIPUR

TO WHOM IT MAY CONCERN

This is to certify that Mr. Robert Loitongbam(Regn. No. 963321104043) from Rohini College of Engineering and Technology, Palkulam, Tamil Nadu has completed a 4 weeks internship program on "Data structures and algorithm analysis and its various applications" from 21st June 2023 to 20th July 2023.

During the internship period, his performance was very good and satisfactory. Also, I found him to be very sincere, honest, hard working and dedicated good team player.

I wish him a very successful life.

Date: 25/07/23

[illegible]
(T. SONAMANI SINGH)

Head of Deptt.
Comp. Sc. & Engg.
MIT, Manipur University

[illegible]
ꯃꯅꯤꯄꯨꯔ - ꯷꯹꯵ ꯰꯰꯴

website:- www.mitimphal.in
e-mail:- principal@mitimphal.in

ACKNOWLEDGEMENT

First I would like to thank Mr.T.Sonamani, Head of Deptt., of **Comp. Sc. Engg.**, **MIT, Manipur University, Imphal** for giving me the opportunity to do an internship within the organization.

I also would like all the people that worked along with me **MIT, Manipur University, Imphal** with their patience and openness they created an enjoyable working environment.

It is indeed with a great sense of pleasure and immense sense of gratitude that I acknowledge the help of these individuals.

I am highly indebted to Director **Rashtra Vibhusan Dr.N. Neela Vishnu,MBA** and Principal **Dr.R.Rajesh M.E.,Ph.D.**, for the facilities provided to accomplish this internship.

I would like to thank my Head of the Department **Dr.R.Shahila Devi M.Tech.MISTE(Ph.D)** for her constructive criticism throughout my internship.

I would like to thank **Dr. B. Rajesh,** College internship coordinator and Sri.**L.V Satyanarayana** internship coordinator Department of CSE for their support and advices to get and complete internship in above said organization.

I am extremely great full to my department staff members and friends who helped me in successful completion of this internship.

ROBERT LOITONGBAM
(963321104043)

ABSTRACT

During my internship, I focused on the analysis and application of data structures and algorithms to optimize performance in real-world problems. The objective was to enhance computational efficiency through the use of basic algorithmic techniques and appropriate data structures. I worked with a variety of structures such as trees, graphs, and hash maps, and implemented algorithms for sorting, searching, and optimization. The results included significant improvements in time complexity and memory usage for specific applications. This work demonstrated the importance of choosing the right data structures and algorithms to solve complex computational problems efficiently. Future work may explore further refinements and new applications in related areas.

Learning Objectives/Internship Objectives

- Internships are generally thought of to be reserved for college students looking to gain experience in a particular field. However, a wide array of people can benefit from Training Internships in order to receive real world experience and develop their skills.

- An objective for this position should emphasize the skills you already possess in the area and your interest in learning more

- Internships are utilized in a number of different career fields, including architecture, engineering, healthcare, economics, advertising and many more.

- Some internship is used to allow individuals to perform scientific research while others are specifically designed to allow people to gain first-hand experience working.

- Utilizing internships is a great way to build your resume and develop skills that can be emphasized in your resume for future jobs. When you are applying for a Training Internship, make sure to highlight any special skills or talents that can make you stand apart from the rest of the applicants so that you have an improved chance of landing the position.

INTRODUCTION TO DATA STRUCTURE AND ALGORITHM

Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms.

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Linear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
Example: Array, Stack, Queue, Linked List, etc.

Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
Example: array.

Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
Example: Queue, Stack, etc.

Non-Linear Data Structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can’t traverse all the elements in a single run only.
Examples: Trees and Graphs.

IMPORTANCE OF DATA STRUCTURE AND ALGORITM

Data structures and algorithms are foundational concepts in computer science and software development, playing a critical role in designing efficient, scalable, and maintainable applications. Here's why they are important:

1. Efficiency in Problem Solving

Optimal Use of Resources: Well-designed algorithms and data structures ensure that programs use memory and computational power efficiently. They help solve problems faster and with fewer resources, especially when dealing with large datasets.

Time Complexity Reduction: Understanding algorithmic time complexity helps developers choose the most efficient method to handle tasks. For example, using a linear-time algorithm ($O(n)$) versus a quadratic-time algorithm ($O(n^2)$) can be the difference between a program running in seconds or hours.

2. Scalability

Handling Large Data: With the growing amount of data in applications, selecting the right data structure (e.g., hash maps, trees, heaps) ensures that programs can handle large volumes of data efficiently without compromising performance.

Managing Complexity: Algorithms allow us to break down complex tasks into manageable parts, ensuring that applications can scale as data or user demands increase.

3. Improved Application Performance

Faster Execution: Efficient algorithms directly contribute to faster application performance. For instance, sorting algorithms like Quick Sort or Merge Sort outperform less efficient ones (e.g., Bubble Sort) for large datasets.

Reduced Memory Footprint: Appropriate data structures can minimize memory usage. For example, using linked lists over arrays can save memory when handling dynamic data sizes.

4. Foundation for Advanced Concepts

Enables Complex Operations: Many advanced algorithms in areas such as machine learning, artificial intelligence, and database management systems rely on foundational data structures (e.g., graphs, trees, queues) and algorithms (e.g., search, dynamic programming).

Basis for System Design: Understanding data structures and algorithms helps in designing complex systems such as operating systems, compilers, and distributed systems where performance and resource management are critical.

5. Enabling Code Reusability and Modularity

Reusable Solutions: Many data structures and algorithms are implemented as reusable components. Libraries and frameworks often provide standardized implementations (e.g., Java’s Collection framework

or Python's built-in data structures), allowing developers to focus on higher-level design.
Cleaner and Modular Code: By using appropriate data structures, developers can create modular, clean, and easy-to-maintain code. For example, a stack or queue can cleanly encapsulate operations that follow a Last-In-First-Out (LIFO) or First-In-First-Out (FIFO) pattern, respectively.

6. Enhancing Decision-Making in Design
Informed Trade-offs: Different algorithms and data structures come with trade-offs in terms of time complexity, space complexity, and ease of implementation. Understanding these trade-offs helps developers make informed decisions about which approaches to use in different contexts.
Tailoring Solutions to Specific Problems: By understanding different algorithms and data structures, developers can choose the best solution for specific problems. For example, when fast retrieval is needed, a hash map might be more appropriate than a list.

7. Enabling Competitive Advantage in Software Development
Better User Experience: Efficient algorithms ensure that applications remain responsive, even under heavy loads. For example, a poorly designed search algorithm can significantly slow down a search engine, affecting user experience.
Competitive Edge: Many tech giants (e.g., Google, Facebook, Amazon) use complex algorithms and data structures to maintain fast services and efficient systems. Mastery of these concepts allows developers to build competitive and high-performance systems.

8. Algorithm Analysis and Problem-Solving Mindset
Logical Thinking and Problem Solving: Studying data structures and algorithms fosters a problem-solving mindset. Developers learn to break down problems logically and solve them step by step, enhancing their analytical skills.
Algorithmic Thinking: Understanding algorithms instills a mindset of designing solutions that are not just correct but also optimal in terms of time and space.

9. Real-World Applications
Database Management: Data structures like B-trees and hash tables are essential for indexing in databases, enabling fast query retrieval and storage optimization.
Networking: Algorithms such as shortest path and spanning tree are vital in routing data efficiently across networks.

METHODS AND TECHNOLOGIES

What is an Array?

An array is a collection of items of the same variable type that are stored at contiguous memory locations. It's one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0 . Each element in an array is accessed through its index.

Need of Array Data Structures

Arrays are a fundamental data structure in computer science. They are used in a wide variety of applications, including:

- Storing data for processing
- Implementing data structures such as stacks and queues
- Representing data in tables and matrices
- Creating dynamic data structures such as linked lists and trees

Types of Array

There are two main types of arrays:

- One-dimensional arrays: These arrays store a single row of elements.
- Multidimensional arrays: These arrays store multiple rows of elements.

Array Operations

Common operations performed on arrays include:

- Traversal : Visiting each element of an array in a specific order (e.g., sequential, reverse).
- Insertion : Adding a new element to an array at a specific index.
- Deletion : Removing an element from an array at a specific index.
- Searching : Finding the index of an element in an array.

Applications of Array

Arrays are used in a wide variety of applications, including:

- Storing data for processing
- Implementing data structures such as stacks and queues
- Representing data in tables and matrices
- Creating dynamic data structures such as linked lists and trees

What is a String?

String is considered a data type in general and is typically represented as arrays of bytes (or words) that store a sequence of characters. String is defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

Some examples of strings are: “geeks” , “for”, “geeks”, “GeeksforGeeks”, “Geeks for Geeks”, “123Geeks”, “@123 Geeks”.

String Data Type:
In most programming languages, strings are treated as a distinct data type. This means that strings have their own set of operations and properties. They can be declared and manipulated using specific string-related functions and methods.
Note: In some languages, strings are implemented as arrays of characters, making them a derived data type.

String Operations:
Strings support a wide range of operations, including concatenation, substring extraction, length calculation, and more. These operations allow developers to manipulate and process string data efficiently.

Below are fundamental operations commonly performed on strings in programming.

- Concatenation: Combining two strings to create a new string.
- Length: Determining the number of characters in a string.
- Access: Accessing individual characters in a string by index.
- Substring: Extracting a portion of a string.
- Comparison: Comparing two strings to check for equality or order.
- Search: Finding the position of a specific substring within a string.
- Modification: Changing or replacing characters within a string.

Applications of String:

- Text Processing: Strings are extensively used for text processing tasks such as searching, manipulating, and analyzing textual data.
- Data Representation: Strings are fundamental for representing and manipulating data in formats like JSON, XML, and CSV.
- Encryption and Hashing: Strings are commonly used in encryption and hashing algorithms to secure sensitive data and ensure data integrity.
- Database Operations: Strings are essential for working with databases, including storing and querying text-based data.
- Web Development: Strings are utilized in web development for constructing URLs, handling form data, processing input from web forms, and generating dynamic content.

LINKED LIST DATA STRUCTURE

A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.

A linked list is a linear data structure that consists of a series of nodes connected by pointers (in C or C++) or references (in Java, Python and JavaScript). Each node contains data and a pointer/reference to the next node in the list. Unlike arrays, linked lists allow for efficient insertion or removal of elements from any position in the list, as the nodes are not stored contiguously in memory.

Linked Lists vs Arrays
Here's the comparison of Linked List vs Arrays
Linked List:
- Data Structure: Non-contiguous
- Memory Allocation: Typically allocated one by one to individual elements
- Insertion/Deletion: Efficient
- Access: Sequential

Array:
- Data Structure: Contiguous
- Memory Allocation: Typically allocated to the whole array
- Insertion/Deletion: Inefficient
- Access: Random

Types of Linked List
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List
- Header Linked List

Operations of Linked Lists:
- Linked List Insertion
- Search an element in a Linked List (Iterative and Recursive)
- Find Length of a Linked List (Iterative and Recursive)
- Reverse a linked list
- Linked List Deletion (Deleting a given key)
- Linked List Deletion (Deleting a key at given position)

STACK DATA STRUCTURE

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It behaves like a stack of plates, where the last plate added is the first one to be removed.

Think of it this way:

- Pushing an element onto the stack is like adding a new plate on top.
- Popping an element removes the top plate from the stack.

Key Operations on Stack Data Structures

- Push: Adds an element to the top of the stack.
- Pop: Removes the top element from the stack.
- Peek: Returns the top element without removing it.
- IsEmpty: Checks if the stack is empty.
- IsFull: Checks if the stack is full (in case of fixed-size arrays).

Applications of Stack Data Structures

- Recursion
- Expression Evaluation and Parsing
- Depth-First Search (DFS)
- Undo/Redo Operations
- Browser History
- Function Calls

Queue in Data Structure

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It operates like a line where elements are added at one end (rear) and removed from the other end (front).

Basic Operations of Queue Data Structure

- Enqueue (Insert): Adds an element to the rear of the queue.
- Dequeue (Delete): Removes and returns the element from the front of the queue.
- Peek: Returns the element at the front of the queue without removing it.
- Empty: Checks if the queue is empty.
- Full: Checks if the queue is full.

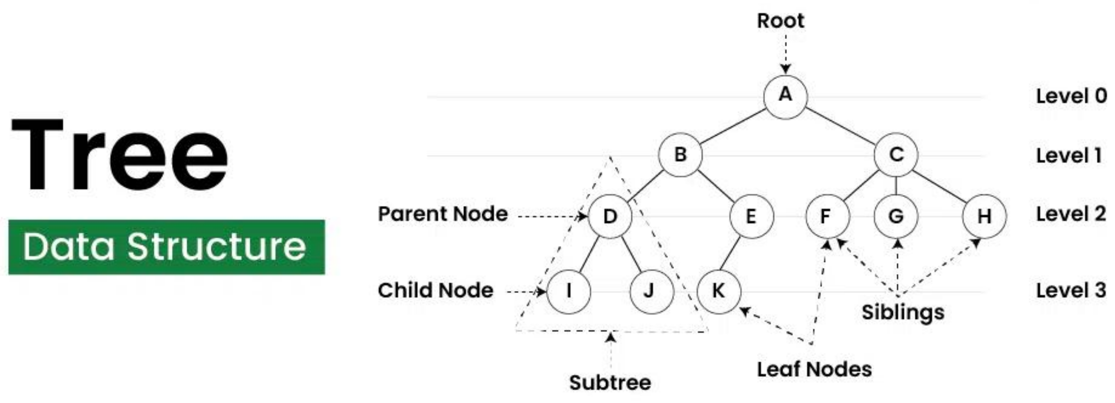
Applications of Queue

- Task scheduling in operating systems
- Data transfer in network communication
- Simulation of real-world systems (e.g., waiting lines)
- Priority queues for event processing queues for event processing

TREE DATA STRUCTURE

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



Terminologies In Tree Data Structure

Parent Node: The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.

Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.

Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.

Leaf Node or External Node: The nodes which do not have any child nodes are called leaf nodes. {I, J, K, F, G, H} are the leaf nodes of the tree.

Ancestor of a Node: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}

Descendant: A node x is a descendant of another node y if and only if y is an ancestor of x.

Sibling: Children of the same parent node are called siblings. {D,E} are called siblings.

Level of a node: The count of edges on the path from the root node to that node. The root node has level 0.

Internal node: A node with at least one child is called Internal Node.
Neighbor of a Node: Parent or child nodes of that node are called neighbors of that node.
Subtree: Any node of the tree along with its descendant.

HEAP DATA STRUCTURE

A heap is a binary tree-based data structure that follows the heap property. In a heap, the value of each node is compared to the values of its children in a specific way:

Max-Heap: The value of each node is greater than or equal to the values of its children, ensuring that the root node contains the maximum value. As you move down the tree, the values decrease.
Min-Heap: The value of each node is less than or equal to the values of its children, ensuring that the root node contains the minimum value. As you move down the tree, the values increase.

This property guarantees that the largest (in a max-heap) or smallest (in a min-heap) value is always at the root, and the tree maintains a specific order based on the type of heap.

Types of Heaps

There are two main types of heaps:
Max Heap: The root node contains the maximum value, and the values decrease as you move down the tree.
Min Heap: The root node contains the minimum value, and the values increase as you move down the tree.

Heap Operations

Common heap operations are:
Insert: Adds a new element to the heap while maintaining the heap property.
Extract Max/Min: Removes the maximum or minimum element from the heap and returns it.
Heapify: Converts an arbitrary binary tree into a heap.

GRAPH DATA STRUCTURE

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).
Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly

useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.

Components of a Graph:

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.

Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

Operations on Graphs:

Basic Operations:

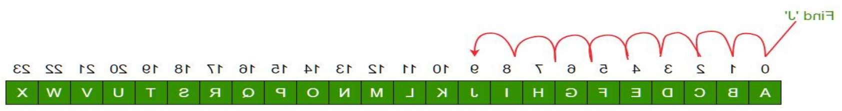
- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph.

What is Searching?

Searching is the fundamental process of locating a specific element or item within a collection of data. This collection of data can take various forms, such as arrays, lists, trees, or other structured representations. The primary objective of searching is to determine whether the desired element exists within the data, and if so, to identify its precise location or retrieve it. It plays an important role in various computational tasks and real-world applications, including information retrieval, data analysis, decision-making processes, and more.

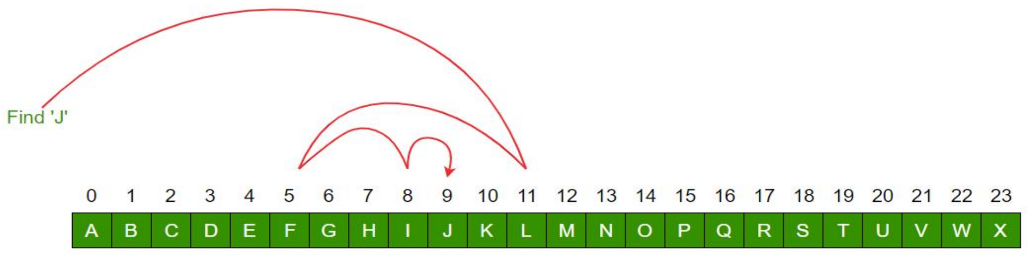
LINEAR SEARCH

Assume that item is in an array in random order and we have to find an item. Then the only way to search for a target item is, to begin with, the first position and compare it to the target. If the item is at the same, we will return the position of the current item. Otherwise, we will move to the next position. If we arrive at the last position of an array and still can not find the target, we return -1. This is called the Linear search or Sequential search.



BINARY SEARCH

In a binary search, however, cut down your search to half as soon as you find the middle of a sorted list. The middle element is looked at to check if it is greater than or less than the value to be searched. Accordingly, a search is done to either half of the given list



Sorting Algorithms:

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure. Sorting means reordering of all the elements either in ascending or in descending order.

- **Selection Sort**
- **Bubble Sort**
- **Insertion Sort**
- **Merge Sort**
- **Quick Sort**
- **Heap Sort**

Sorting Terminology:

In-place Sorting: An in-place sorting algorithm uses constant space for producing the output (modifies the given array only) or copying elements to a temporary storage. Examples: Selection Sort, Bubble Sort Insertion Sort and Heap Sort.

Internal Sorting: Internal Sorting is when all the data is placed in the main memory or internal memory. In internal sorting, the problem cannot take input beyond its size. Example: heap sort, bubble sort, selection sort, quick sort, shell sort, insertion sort.

External Sorting : External Sorting is when all the data that needs to be sorted cannot be placed in memory at a time, the sorting is called external sorting. External Sorting is used for the massive amount of data. Examples: Merge sort, Tag sort, Polyphase sort, Four tape sort, External radix sort, etc.

Stable sorting: When two same items appear in the same order in sorted data as in the original array called stable sort. Examples: Merge Sort, Insertion Sort, Bubble Sort.

Unstable sorting: When two same data appear in the different order in sorted data it is called unstable sort. Examples: Selection Sort, Quick Sort, Heap Sort, Shell Sort.

Recursion

Recursion is a programming technique where a function calls itself within its own definition. This allows a function to break down a problem into smaller subproblems, which are then solved recursively.

How Does Recursion Work?

Recursion works by creating a stack of function calls. When a function calls itself, a new instance of the function is created and pushed onto the stack. This process continues until a base case is reached, which is a condition that stops the recursion. Once the base case is reached, the function calls start popping off the stack and returning their results.

What is a Recursive Algorithm?

A recursive algorithm is an algorithm that uses recursion to solve a problem. Recursive algorithms typically have two parts:

- Base case: Which is a condition that stops the recursion.
- Recursive case: Which is a call to the function itself with a smaller version of the problem.

Examples of Recursion

Here are some common examples of recursion:

Example 1: Factorial: The factorial of a number n is the product of all the integers from 1 to n. The factorial of n can be defined recursively as:

factorial(n) = n * factorial(n-1)

Example 2: Fibonacci sequence: The Fibonacci sequence is a sequence of numbers where each number is the sum of the two preceding numbers. The Fibonacci sequence can be defined recursively as:

fib(n) = fib(n-1) + fib(n-2)

CONCLUSION

In conclusion, my internship experience in data structure and algorithm analysis provided valuable insights into optimizing computational processes and solving complex problems efficiently. By working with various data structures, such as trees and graphs, and implementing key algorithms like sorting, searching, and dynamic programming, I was able to achieve significant improvements in system performance and resource management. This experience has deepened my understanding of algorithmic complexity and its real-world applications. Additionally, I enhanced my problem-solving and analytical skills, which will be instrumental in my future endeavors in software development and system design. Overall, this internship has been an enriching learning experience, preparing me for more advanced challenges in the field of computer science.