Summer of Science 2022

Mid-Term Report

Title: Exploring Time Complexity Analysis, Basic Data Structures, STL, and Sorting

Introduction: Hey there! In this mid-term report, I'm diving into the exciting world of data structures and algorithms and the topics I have understood are time complexity analysis, basic data structures, the Standard Template Library (STL), and sorting algorithms!

1. Time Complexity Analysis: Unveiling the Efficiency of Algorithms

    - What is Time Complexity? Time complexity is a measure of the efficiency of an algorithm. It helps us understand how the running time of an algorithm grows as the input size increases. In simpler terms, it tells us how well an algorithm can handle larger data sets.

    - Big O Notation: Big O notation is a mathematical notation used to describe the upper bound of an algorithm's time complexity. It helps us compare and analyse algorithms based on their growth rates. Common notations include O(1), O(n), O(log n), and O(n^2), among others.

      Formal Definition: Let T(n) and f(n) be two positive functions. We write **T(n) ∈ *O*(f(n))**, and say that T(n) has order of f(n), if there are positive constants M and n_0 such that T(n) ≤ M·f(n) for all $n \geq n_0$.

    - Understanding the Growth Rate: O(1), O(n), O(log n), and More Let's delve deeper into the different growth rates seen in time complexity analysis. O(1) represents constant time complexity, meaning the algorithm's running time remains constant regardless of the input size. O(n) denotes linear time complexity, where the running time grows linearly with the input size. O(log n) signifies logarithmic time complexity, showing that the running time grows slowly as the input size increases. Additionally, there are other complexities such as O(n^2) for quadratic time complexity and O(2^n) for exponential time complexity.

    - Analysing Time Complexity Examples: To better understand time complexity analysis, let's consider a few examples. Suppose we have an algorithm that performs a simple operation on a single element. This algorithm would have a time complexity of O(1) since the running time remains constant regardless of the input size. On the other hand, if we have an algorithm that loops through an array of size n and performs an operation on each element, the time complexity would be O(n) since the running time grows linearly with the input size. By analyzing such examples, we can gain insights into how different algorithms perform.

    - Best, Worst, and Average Case Analysis Time complexity analysis also considers the best, worst, and average case scenarios. The best-case scenario represents the minimum possible time required for an algorithm to run, while the worst-case scenario represents the maximum time required. The average-case analysis

considers the expected running time over a range of inputs. Understanding these cases helps us anticipate the performance of an algorithm under different conditions.

2. Basic Data Structures: Building Blocks of Efficient Programs

    - Introduction to Data Structures: More than Just Containers Data structures are essential tools for organizing and managing data efficiently. They provide a way to store and manipulate data in a structured manner, enabling us to perform operations effectively.
    - Arrays: The Simple Yet Powerful Data Structure Arrays are one of the fundamental data structures. They allow us to store a collection of elements of the same type in contiguous memory locations. Arrays provide fast access to elements through their indices but have a fixed size. It's important to note that arrays in some programming languages, like C++, can be dynamically resized using vectors.
    - Linked Lists: Connecting the Dots Linked lists are dynamic data structures that consist of nodes linked together via pointers. Each node contains data and a reference to the next node. Unlike arrays, linked lists allow for efficient insertion and deletion operations but have slower access times since elements need to be traversed sequentially.
    - Stacks: Last In, First Out (LIFO) Stacks are data structures that follow the Last In, First Out (LIFO) principle. Elements are added and removed from the same end, typically called the top. Stacks are used in various scenarios, such as function call management, expression evaluation, and undo-redo functionality.
    - Queues: First In, First Out (FIFO) Queues adhere to the First In, First Out (FIFO) principle. Elements are added at one end, known as the rear, and removed from the other end, known as the front. Queues are commonly used in scenarios like process scheduling, printing tasks, and breadth-first search algorithms.
    - Trees: Nature's Data Structure Trees are hierarchical data structures composed of nodes connected by edges. They have a root node at the top and offer efficient searching, insertion, and deletion operations. Trees find applications in file systems, hierarchical data organization, and data representation.
    - Graphs: The Web of Connections Graphs are versatile data structures that represent interconnected nodes (vertices) through edges. They are used to model relationships between objects and play a crucial role in network analysis, social networks, and transportation systems.

3. Standard Template Library (STL) Containers: Versatile Data Structures

    The Standard Template Library (STL) in C++ provides a rich set of container classes that offer various data storage and manipulation capabilities. In this section, we will explore three commonly used STL containers: vectors, lists, sets ,stacks, linked lists, and queues.

- Stacks: Last-In-First-Out (LIFO) Structure Stacks are data structures that follow the Last-In-First-Out (LIFO) principle. Elements are inserted and removed from the same end, known as the top of the stack. The stack offers constant-time insertion and deletion at the top, making it efficient for managing function calls, expression evaluation, and backtracking algorithms. STL provides the **std::stack** container adapter that uses a deque or a list as the underlying container. Stacks are commonly used in applications such as text editors (undo/redo operations), depth-first search algorithms, and parentheses balancing.

- Vectors: Dynamic Arrays, Vectors are dynamic arrays that allow for efficient random access and dynamic resizing. They provide a contiguous block of memory that stores elements in a linear order. Vectors are especially useful when the size of the container needs to change dynamically during runtime. They offer constant-time access to elements and have a time complexity of O(1) for insertion and deletion at the end. However, inserting or deleting elements in the middle of a vector can be costly, requiring elements to be shifted. Vectors are suitable for scenarios where fast random access is required, such as when implementing dynamic arrays or implementing algorithms that rely on indexing.

- Lists: Doubly Linked Lists, Lists are doubly linked lists that provide efficient insertion and deletion at both ends and in the middle. Unlike vectors, lists do not offer constant-time random access. However, they excel at constant-time insertion and deletion operations regardless of the position within the container. Lists use nodes connected through pointers to maintain their structure, which allows for efficient element removal and insertion. They are suitable for scenarios that involve frequent insertion and removal operations, such as implementing queues or maintaining a sorted collection of elements.

- Sets: Unique Sorted Elements, Sets are containers that store unique elements in a sorted order. They provide fast searching, insertion, and deletion operations based on the value of the elements. Sets automatically maintain their sorted order, making them useful for scenarios where maintaining a collection of unique elements with efficient searching is required. Sets are implemented as binary search trees or hash tables, depending on the underlying implementation. Binary search tree-based sets have a time complexity of O(log n) for search, insert, and delete operations, while hash-based sets offer constant-time operations on average.

- Linked Lists: Dynamic and Flexible Linked lists are dynamic data structures that consist of nodes linked together via pointers. Each node holds data and a pointer to the next node, forming a sequence. Unlike arrays, linked lists can efficiently insert and delete elements at any position. However, accessing elements in a linked list requires traversing the list sequentially. STL provides the **std::list** container, which implements a doubly linked list. Linked lists are useful in scenarios where frequent

insertion and deletion at any position are required, such as implementing hash tables, maintaining a sorted collection, or representing sparse data structures.

- Queues: First-In-First-Out (FIFO) Structure Queues are data structures that follow the First-In-First-Out (FIFO) principle. Elements are inserted at the rear (enqueue) and removed from the front (dequeue). Queues are ideal for scenarios where the order of elements matters, such as simulating real-world scenarios, handling requests, or implementing breadth-first search algorithms. The STL provides the **std::queue** container adapter, which internally uses a deque or a list as the underlying container. Queues efficiently support enqueue and dequeue operations with a time complexity of O(1). Additionally, STL offers the **std::priority_queue** container, which implements a priority queue using a binary heap.

- Choosing the Right STL Container, The choice of STL container depends on the specific requirements of the problem at hand. Vectors are suitable when fast random access and dynamic resizing are needed. Lists are preferred for scenarios involving frequent insertion and deletion operations, especially in the middle of the container. Sets are ideal for maintaining unique elements in a sorted order with efficient searching capabilities. It is essential to consider the trade-offs between time complexity, memory usage, and the specific operations required by your application.

Practical Examples:

- Suppose you are implementing a dynamic array that needs to grow or shrink dynamically based on user input. In that case, a vector would be a suitable choice due to its efficient random access and dynamic resizing capabilities.
- If you are implementing a task scheduler that requires frequent insertion and deletion of tasks in the middle of the list, a list would be a better option due to its constant-time insertion and deletion at any position.
- Consider a scenario where you need to maintain a collection of unique words in alphabetical order. In this case, a set would be an appropriate choice, as it provides automatic sorting and efficient searching for unique elements.

Summary: The STL containers offer versatile data structures with different strengths and capabilities. Vectors provide dynamic arrays with efficient random access and dynamic resizing. Lists offer doubly linked lists with fast insertion and deletion at any position. Sets store unique elements in sorted order with efficient searching operations. Understanding the characteristics and appropriate use cases of each container allows you to leverage their power in designing efficient and effective data structures for your applications. Understanding the intricacies of stacks, linked lists, and queues allows you to leverage the power of these versatile data structures. Stacks provide Last-In-First-Out (LIFO) behavior and are commonly used for function call management and expression evaluation. Linked lists offer flexibility with efficient insertion and deletion at any position, making them suitable for

various scenarios. Queues follow the First-In-First-Out (FIFO) principle and are used in simulations, request handling, and breadth-first search algorithms. By choosing the right container for your specific needs, you can design efficient and robust algorithms and data structures.

4. Sorting Algorithms: Putting Things in Order

    - The Importance of Sorting ,Sorting is a fundamental operation in computer science, allowing us to arrange elements in a specific order. Sorted data facilitates efficient searching, retrieval, and analysis, leading to optimized algorithms and improved program performance.
    - Bubble Sort: Surfing the Waves of Swapping Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. The algorithm progresses through the list, gradually moving larger elements towards the end. Bubble sort has a time complexity of O(n^2) in the worst and average case, and O(n) in the best case when the list is already sorted. It is straightforward to implement but not the most efficient sorting algorithm for large datasets.
    - Insertion Sort: The Right Place at the Right Time Insertion sort builds a sorted portion of the list by inserting elements into their correct positions. It iterates over the unsorted portion, comparing each element with the sorted portion and placing it in its appropriate place. Insertion sort performs well for small lists or partially sorted data. It has a time complexity of O(n^2) in the worst and average case, and O(n) in the best case for already sorted or nearly sorted lists.
    - Selection Sort: Picking the Right One Selection sort divides the list into sorted and unsorted portions. It repeatedly selects the smallest (or largest) element from the unsorted portion and places it at the beginning of the sorted portion. Selection sort has a time complexity of O(n^2) in all cases, making it less efficient compared to other sorting algorithms. However, it has the advantage of fewer swaps compared to bubble sort and insertion sort.
    - Quick Sort: Divide and Conquer Quick sort is a popular divide-and-conquer sorting algorithm that partitions the list around a pivot element. It divides the list into two sublists , one with elements smaller than the pivot and the other with elements larger than the pivot. The process is recursively applied to the sublists until the entire list is sorted. Quick sort has an average time complexity of O(n log n), making it one of the fastest sorting algorithms. However, in the worst case, it can degrade to O(n^2) when the pivot selection is unbalanced.
    - Merge Sort: Divide, Sort, and Merge: Merge sort is another divide-and-conquer algorithm that recursively divides the list into halves, sorts them, and then merges them back together. It guarantees a stable sort and performs well for large datasets. Merge sort has a time complexity of O(n log n) in all cases, making it an efficient sorting algorithm. However, it requires additional memory for merging the sublists, which can be a consideration when working with limited resources.
    - Understanding Trade-Offs and Choosing the Right Sorting Algorithm Each sorting algorithm has its strengths and weaknesses. Bubble sort, insertion sort, and

selection sort are simple and easy to understand but not efficient for large datasets. Quick sort and merge sort offer better performance but may have higher memory requirements. When choosing a sorting algorithm, consider the input size, data characteristics, stability requirements, and available resources.

Practical Examples

- Bubble sort and insertion sort can be suitable for small lists or situations where simplicity and ease of implementation are prioritized.
- Selection sort can be useful when minimizing the number of swaps is crucial, such as in scenarios where writing to memory is costly.
- Quick sort is often favoured for its average-case efficiency and can be used in a wide range of applications, including general-purpose sorting.
- Merge sort is suitable when stability and guaranteed O(n log n) performance are required, such as in external sorting or applications dealing

Conclusion: I've covered the foundations of time complexity analysis, basic data structures, the STL, and sorting algorithms. Remember, this report provides a starting point of my exploration of these topics.

MENTEE : G.NIVEDHA

MENTOR : PREETHI MALYALA

TOPIC : DATA STRUCTURES AND ALGORITHMS

TENTATIVE DEADLINES:

WEEK ONE: Time complexity Analysis

WEEK TWO: Basic Data Structures

WEEK THREE: STL, Sorting

WEEK FOUR: (End-sems)

WEEK FIVE: Divide and Conquer

WEEK SIX: Graph Algorithms

WEEK SEVEN: Dynamic Programming

WEEK EIGHT: Greedy Algorithms

EXPLORE MORE AND LEFTOVER TOPICS, REPORT- MAKING

WEEK NINE:

WEEK TEN:

WEEK ELEVEN:

WEEK TWELVE: