**DATA STRUCTURES**

**EXERCISE 1**

**INVENTORY MANAGEMENT SYSTEM**

**UNDERSTANDING THE PROBLEM :**

Efficient data structures and algorithms are crucial in handling large inventories because proper data structures and algorithms ensure that operations like adding, updating, deleting, and searching for products are performed quickly, even as the inventory grows.

**TYPES OF DATA STRUCTURES SUITABLE FOR THIS PROBLEM :**

**Array List** is useful for maintaining an ordered list of products with fast random access.

**HashMap** provides fast access, addition, and deletion operations through hashing. It's ideal for associating unique product IDs with product details.

**Tree Map** is Similar to HashMap but maintains a sorted order of keys, which can be useful for range queries and ordered traversals.

**ANALYSIS :**

Add Product: The addProduct method uses the put operation of the HashMap, which has an average time complexity of O(1).

Update Product: The updateProduct method also uses the put operation, resulting in O(1) time complexity.

Delete Product: The deleteProduct method uses the remove operation of the HashMap, which has an average time complexity of O(1).

Get Product: The getProduct method uses the get operation of the HashMap, which has an average time complexity of O(1).

**EXERCISE 2**

**E-COMMERCE PLATFORM SEARCH FUNCTION**

**UNDERSTANDING ASYMPTOTIC NOTATION :**

Big O notation is a way of expressing how long it takes for an algorithm to run. It's focused on the maximum amount of time the algorithm could take, which helps us understand how it will perform as we work with larger and larger sets of data.

**Different Ways to Measure Search Operations:**

Best Case: This is when the algorithm finds what it's looking for quickly, like when the thing you want is right at the beginning of the list.

Average Case: This is a more realistic view of how the algorithm will perform, looking at what will happen over many different situations.

Worst Case: This is when the algorithm takes the longest to find what it's looking for, like when the thing you want is at the very end of the list or not there at all.

**ANALYSIS :**

**Time Complexity Comparison**

* **Linear Search**:
  + Best Case: O(1) (when the element is at the first position)
  + Average Case: O(n) (when the element is somewhere in the middle)
  + Worst Case: O(n) (when the element is at the last position or not present)
* **Binary Search**:

Best Case: O(1) (when the element is at middle position)

Average Case: O(log n)

**SUITABLE ALGORITHM FOR THE PLATFORM**

**Binary Search** is more suitable for the e-commerce platform because:

* **Efficiency**: Binary search has a significantly lower time complexity (O(log n)) compared to linear search (O(n)), making it faster for large datasets.
* **Sorted Data Requirement**: E-commerce platforms often sort products based on various attributes (price, popularity, etc.), making the data readily usable for binary search.
* **Scalability**: As the number of products grows, binary search ensures that search operations remain efficient.

**EXERCISE 3**

**SORTING CUSTOM ORDERS**

**UNDERSTANDING SORTING ALGORITHMS :**

**Sorting Algorithms:**

1. Bubble Sort: Compares adjacent elements and swaps them if they are in the wrong order.

2. Insertion Sort: Builds the final sorted array one item at a time.

3. Quick Sort: Efficiently divides and sorts the array using a pivot element.

4. Merge Sort: Divides the array into two halves, sorts them, and merges the sorted halves.

**Analysis**

**Time Complexity Comparison**

**Bubble Sort**:

Best Case: O(n) (when the array is already sorted)

Average Case: O(n^2)

Worst Case: O(n^2)

**Quick Sort**:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) (when the pivot is the smallest or largest element in each partition)

**Quick Sort is Preferred Over Bubble Sort :**

**Performance**: Quick Sort generally performs much better than Bubble Sort due to its average time complexity of O(n log n) compared to Bubble Sort's O(n^2).

**Efficiency**: Quick Sort efficiently handles larger datasets by dividing the problem into smaller sub-problems and solving them recursively.

**Use Cases**: Quick Sort is suitable for most practical use cases where average performance is critical, whereas Bubble Sort is often used for educational purposes or small datasets.

**EXERCISE 4**

**EMPLOYEE MANAGEMENT SYSTEM**

**UNDERSTANDING ARRAY REPRESENTATION :**

Arrays are contiguous blocks of memory where each element is stored at a specific index. The memory address of each element is calculated based on the starting address of the array and the size of each element. For an array starting at memory address base with element size, the address of the element at index i is calculated as base + (i \* size).

**Advantages of Arrays**

**Constant-Time Access**: Arrays provide O(1) time complexity for accessing elements by index.

**Cache-Friendly**: Due to contiguous memory allocation, arrays benefit from spatial locality, improving cache performance.

**Memory Efficiency**: Arrays have low memory overhead since they do not require additional structures for element storage.

**ANALYSIS :**

**Time Complexity of Operations**

* **Add Employee**: O(1) if the array is not full. If the array needs resizing (not implemented here), the time complexity would be O(n) due to copying elements to a new array.
* **Search Employee**: O(n) as it involves iterating through the array to find the employee.
* **Traverse Employees**: O(n) since it requires visiting each element in the array.
* **Delete Employee**: O(n) in the worst case as it involves shifting elements after deleting an employee.

**Limitations of Arrays and When to Use Them**

* **Fixed Size**: Arrays have a fixed size, which means they cannot dynamically grow or shrink. This can lead to wasted space if the array is too large or the need for resizing if the array is too small.
* **Inefficient Insertion and Deletion**: Inserting or deleting elements in the middle of the array requires shifting elements, leading to O(n) time complexity.
* **Better Alternatives**: Linked lists, dynamic arrays (like ArrayList in Java), and other data structures may be more suitable for scenarios requiring frequent insertions and deletions or unknown array sizes.

**Use Arrays** when:

* The size of the collection is known and does not change frequently.
* Fast access to elements by index is required.
* Memory overhead needs to be minimized.

**EXERCISE 5**

**TASK MANAGEMENT SYSTEM**

**UNDERSTANDING LINKED LISTS**

**Types of Linked Lists**

**Singly Linked List:**

* Structure: Each node contains data and a reference (or link) to the next node in the sequence.
* Operations:

Insertion: O(1) if inserting at the beginning, O(n) if inserting at the end or middle.

Deletion: O(1) if deleting the head, O(n) if deleting other nodes.

Search: O(n).

**Doubly Linked List:**

* Structure: Each node contains data, a reference to the next node, and a reference to the previous node.
* Operations:

Insertion: O(1) for inserting at the beginning, O(n) for inserting at the end or middle.

Deletion: O(1) if deleting from the beginning or end, O(n) for deleting from the middle.

Search: O(n).

**ANALYSIS**

**Time Complexity of Operations**

* **Add Task**: O(1) when adding at the beginning.
* **Search Task**: O(n) since it may need to traverse the entire list.
* **Traverse Tasks**: O(n) as each node must be visited.
* **Delete Task**: O(1) if the node is at the beginning, O(n) for other positions due to the traversal needed to find the node.

**Advantages of Linked Lists Over Arrays for Dynamic Data**

* **Dynamic Size**: Linked lists can grow or shrink dynamically, unlike arrays which have a fixed size.
* **Efficient Insertions/Deletions**: Linked lists allow efficient insertions and deletions, especially at the beginning or middle, without needing to shift elements as in arrays.
* **Memory Utilization**: Linked lists allocate memory as needed for new elements, while arrays may allocate more memory than necessary.

**EXERCISE 6**

**LIBRARY MANAGEMENT SYSTEM**

**UNDERSTANDING SEARCH ALGORITHMS**

**Linear Search**

Linear search is a straightforward algorithm that checks each element in the list sequentially until the desired element is found or the list ends.

* Best Case: O(1) (element is the first item)
* Average Case: O(n)
* Worst Case: O(n)

**Binary Search**

Binary search is a more efficient algorithm that works on sorted lists. It repeatedly divides the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Continue until the value is found or the interval is empty.

* Best Case: O(1) (element is the middle item)
* Average Case: O(log n)
* Worst Case: O(log n)

**ANALYSIS :**

**Time Complexity Comparison**

* Linear Search:
  + Best Case: O(1) (element is the first item)
  + Average Case: O(n)
  + Worst Case: O(n)
* Binary Search:
  + Best Case: O(1) (element is the middle item)
  + Average Case: O(log n)
  + Worst Case: O(log n)

**When to Use Each Algorithm**

* **Linear Search:**
  + Use when the dataset is small or unsorted.
  + Useful when the overhead of sorting is not justified by the number of searches.
  + Simpler to implement and does not require the data to be sorted.
* **Binary Search:**
  + Use when the dataset is large and sorted.
  + Much more efficient for large datasets due to its O(log n) complexity.
  + Requires the list to be sorted, which may involve an additional preprocessing step if the data is not already sorted.

By implementing both linear and binary search, we can see that binary search is more efficient for larger, sorted datasets. However, linear search is simpler and effective for small or unsorted datasets.

**EXERCISE 7**

**FINANCIAL FORECASTING**

**UNDERSTANDING RECURSIVE ALGORITHMS**

Recursion is a programming technique where a function calls itself in order to solve smaller instances of the same problem. It simplifies certain problems by breaking them down into smaller, more manageable sub-problems.

B**ase Case**: The condition under which the recursive function stops calling itself. This prevents infinite recursion.

**Recursive Case**: The part of the function that includes the recursive call, breaking the problem down into a smaller instance of the original problem.

**ANALYSIS :**

**Time Complexity of the Recursive Algorithm**

* **Time Complexity**: O(n)
  + The recursive algorithm makes n recursive calls, each reducing the number of periods by 1.
  + For each call, a single multiplication and addition operation are performed.