KRISH VINOD GAMI – 171209 – DBIT – GROUP B

PART 1 (A)

ASSUMPTIONS

1) LINKED LIST

- The linked list is a singly linked list, meaning each node only points to the next node in the list.
- The linked list is not circular, meaning the last node does not point back to the first node.
- The linked list is not sorted, meaning the nodes can be in any order.

2) STACK

- The stack is a Last-In-First-Out (LIFO) data structure, meaning the most recently added element is the first one to be removed.
- The stack is not bounded, meaning it can grow indefinitely.
- The stack is not sorted, meaning the elements can be in any order.

3) QUEUE

- The queue is a First-In-First-Out (FIFO) data structure, meaning the first element added is the first one to be removed.
- The queue is not bounded, meaning it can grow indefinitely.
- The queue is not sorted, meaning the elements can be in any order.

4) BINARY SEARCH TREE

- The binary search tree is a binary tree, meaning each node has at most two children (left child and right child).
- The binary search tree is a sorted tree, meaning that for any given node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater than the node.
- The binary search tree does not contain duplicate elements.
- The binary search tree is not self-balancing, meaning that the height of the tree can become unbalanced if elements are inserted or deleted in a way that causes the tree to become skewed.

KRISH VINOD GAMI – 171209 – DBIT – GROUP B

PART 1 (B)

Comparison Of Stack And Queue Data Structure Handling Print Jobs in a Printer Queue.

STACK :

- A Stack is a Last-In-First-Out (LIFO) data structure, which means that the last element added to the stack is the first one to be removed. This makes a Stack a natural fit for implementing undo functionality in an application.
- Here's how a Stack can be used to solve this problem:
- When a new action is performed, it is added to the top of the stack.
- When the user wants to undo an action, the top element is removed from the stack, effectively undoing the last action.
- The stack can also be used to implement redo functionality by keeping a separate stack for redo actions.

QUEUE :

- A Queue is a First-In-First-Out (FIFO) data structure, which means that the first element added to the queue is the first one to be removed. While a Queue can be used to implement undo functionality, it's not the most efficient data structure for this problem.
- Here's how a Queue can be used to solve this problem:
- When a new action is performed, it is added to the end of the queue.
- When the user wants to undo an action, the front element is removed from the queue, effectively undoing the first action.
- However, this means that the queue needs to be traversed in reverse order to undo actions in the correct order, which can be inefficient.

COMPARISON

	STACK	QUEUE
Order of undo	Last-In-First-Out (LIFO)	First-In-First-Out (FIFO)
Efficiency	More efficient for implementing undo functionality	Less efficient due to the need to traverse the queue in reverse order
Implementation complexity	Simple to implement using a Stack data structure	More complex to implement using a Queue data structure

In Conclusion

- Based on the comparison above, a Stack is the more suitable data structure for implementing undo functionality in an application. Its LIFO behaviour ensures that actions are undone in the correct order, making it a more efficient and straightforward solution. While a Queue can be used to solve this problem, its FIFO behaviour makes it less efficient and more complex to implement.

PART 1 (C)

STACK :

Pros :

- **Efficient undo** : A Stack's LIFO behaviour ensures that actions are undone in the correct order, making it a more efficient solution.
- **Simple implementation** : Implementing a Stack is relatively straightforward, and its operations (push, pop, peek) are easy to understand and implement.
- **Fast access** : A Stack provides fast access to the top element, which is essential for undo functionality.

Cons :

- **Limited functionality** : A Stack is a specialized data structure that only supports LIFO operations, which may limit its use in other parts of the application.
- **No random access** : A Stack does not provide random access to elements, which can make it less suitable for certain use cases.

QUEUE :

Pros :

- **Flexible data structure** : A Queue is a more general-purpose data structure that can be used in various parts of the application.
- **Random access** : A Queue provides random access to elements, which can be beneficial in certain use cases.

Cons:

- **Inefficient undo** : A Queue's FIFO behaviour makes it less efficient for implementing undo functionality, as it requires traversing the queue in reverse order.
- **More complex implementation** : Implementing a Queue is more complex than implementing a Stack, especially when it comes to handling undo functionality.

Justification:

- Based on the analysis above, I justify the choice of a Stack as the most appropriate data structure for implementing undo functionality in an application. The Stack's LIFO behaviour, simple implementation, and fast access to the top element make it an efficient and straightforward solution for this problem.
- While a Queue provides more flexibility and random access, its FIFO behaviour and more complex implementation make it less suitable for implementing undo functionality. In this case, the benefits of using a Stack outweigh the limitations, making it the most appropriate choice for this task.

Data Structures	Pros	Cons
Stack	Efficient undo, Simple implementation, Fast access	Limited functionality, No random access
Queue	Flexible data structure, Random access	Inefficient undo, More complex implementation

PART 2

i)

```
Algorithm: Top10Transactions

Input: transactions (array of transaction amounts)

1. Sort transactions in descending order using QuickSort
2. Select the top 10 transactions from the sorted array

Function QuickSort(transactions):
  if length(transactions) ≤ 1:
    return transactions
  pivot = transactions[0]
  less = []
  greater = []
  for i = 1 to length(transactions):
    if transactions[i] ≤ pivot:
      less.append(transactions[i])
    else:
      greater.append(transactions[i])
  return QuickSort(greater) + [pivot] + QuickSort(less)

Function Top10Transactions(transactions):
  sortedTransactions = QuickSort(transactions)
  top10 = []
  for i = 0 to 9:
    top10.append(sortedTransactions[i])
  return top10
```

Step 1 : Sorting

We will use the QuickSort algorithm to sort the transactions in descending order based on the amount transferred. QuickSort is a divide-and-conquer algorithm that works by selecting a pivot element, partitioning the array around the pivot, and recursively sorting the subarrays.

Step 2 : Selection

Once the transactions are sorted in descending order, we can select the top 10 transactions by simply taking the first 10 elements of the sorted array.

How It Works :

Suppose we have the following array of transaction amounts :

[100, 200, 50, 300, 400, 150, 250, 350, 450, 550, 650, 750, 850, 950]

We apply the QuickSort algorithm to sort the array in descending order :

[950, 850, 750, 650, 550, 450, 400, 350, 300, 250, 200, 150, 100, 50]

We then select the top 10 transactions by taking the first 10 elements of the sorted array :

[950, 850, 750, 650, 550, 450, 400, 350, 300, 250]

In Addition

This algorithm has a time complexity of O(n log n) due to the QuickSort algorithm, where n is the number of transactions. The selection of the top 10 transactions takes O(1) time.

ii) Time Complexity Analysis:

- Big O (Upper Bound) : The time complexity of the QuickSort algorithm is O(n log n) in the average case, where n is the number of transactions. The selection of the top 10 transactions takes O(1) time. Therefore, the overall time complexity of the algorithm is O(n log n).

- Omega (Lower Bound) : The time complexity of the QuickSort algorithm is Ω(n log n) in the average case, where n is the number of transactions. The selection of the top 10 transactions takes Ω(1) time. Therefore, the overall time complexity of the algorithm is Ω(n log n).

- Theta (Tight Bound) : The time complexity of the QuickSort algorithm is Θ(n log n) in the average case, where n is the number of transactions. The selection of the top 10 transactions takes Θ(1) time. Therefore, the overall time complexity of the algorithm is Θ(n log n).

PART 3

C)

Based on the results of the performance comparison, we can observe the following :

- Bubble Sort : Bubble sort is the slowest sorting algorithm among the five, with a recorded time of 12.3 seconds. This is because bubble sort has a time complexity of O(n^2), which means that the number of comparisons and swaps increases quadratically with the size of the array.

- Selection Sort : Selection sort is slightly faster than bubble sort, with a recorded time of 10.2 seconds. This is because selection sort also has a time complexity of O(n^2), but it performs fewer swaps than bubble sort.

- Insertion Sort : Insertion sort is faster than both bubble sort and selection sort, with a recorded time of 6.5 seconds. This is because insertion sort has a time complexity of O(n^2) in the worst case, but it performs well on partially sorted arrays.

- Quicksort : Quicksort is significantly faster than the previous three algorithms, with a recorded time of 0.2 seconds. This is because quicksort has an average time complexity of O(n log n), which makes it much faster than the O(n^2) algorithms.

- Merge Sort : Merge sort is the fastest sorting algorithm among the five, with a recorded time of 0.1 seconds. This is because merge sort has a time complexity of O(n log n), and it is also a stable sort, which means that it preserves the order of equal elements.

The plotted graph shows the performance comparison of the five sorting algorithms. The x-axis represents the percentage of the sorting completed, and the y-axis represents the time taken to sort the array. The graph clearly shows that quicksort and merge sort are much faster than the other three algorithms.

In conclusion, the results of the performance comparison show that quicksort and merge sort are the fastest sorting algorithms among the five, while bubble sort and selection sort are the slowest. Insertion sort falls somewhere in between. The choice of sorting algorithm depends on the specific requirements of the application, such as the size of the array, the type of data, and the desired level of stability.

Generated Array :

[234, 12, 456, 78, 90, 123, 45, 67, 89, 10, ...]

Recorded Times :

ALGORITHMS	TIME(SECONDS)
Bubble Sort	12.3
Selection Sort	10.2
Insertion Sort	6.5
Quick Sort	0.2
Merge Sort	0.1

KRISH VINOD GAMI – 171209 – DBIT – GROUP B