IAN OMWOYO CAT 2

PART 1

A:

I. **Linked List:** The insertBeginning() method insert number 20 at the beginning of the linked list. The insertEnd() method inserts number 10 at the end of the linked list. The delete() method deletes a given data from the linked list. The searchNode() method searches for a given data if it is in the list or not. It returns a Boolean of True or False. The printTheList() method which is also the transvers method, it prints each data from the Linked list or from each node.

II. **Stack:** Implementation uses an array to store the stack data and provides methods for the stack operations like push, pop transvers and search. Push method adds a data at the top of the stack. Pop method removes an element from the top of the stack. Search method checks if a data is present in the stack. The traverse method prints out the data in the stack.

III. **Queue:** The enqueuer method add a data to the end of the queue. The dequeuer method removes an element from the front of the queue. The search method checks if a given data is present in the queue. The traverse method prints out all the data from the queue.

IV. **Binary Search Tree:** Inserting a new node into the binary tree, the correct location for the new node must be found. A recursive approached is used. Traversal has three main types (InOrder, Preorder and Postorder). Inorder visits the left subtree, the current node and the right subtree. Preorder visits the current subtree, the left subtree and the right subtree. Postorder visits the left subtree, right subtree and the current node.

B: (Linked list and stack)

I. ***Linked list*** is a dynamic data structure where each node points to the next node in the sequence. This allows efficient insertion and deletion of nodes at any position in the list. *ADVANTAGES:* Linked list is efficient in insertion and of elements of task at any position and can implement priority scheduling by inserting task at specific positions. *DISADVANTAGES:* Linked list can be slow in retrieving tasks especially for large lists and more memory is needed to store pointers between nodes. ***Stack*** is a Last in First out data structure that follows the principle of last element inserted being the first to be removed. *ADVANTAGES:* Stack is efficient at retrieving tasks and uses less memory unlike linked list. *DISADVANTAGES:* Insertion and deletion of tasks are limited to the top of the stack and not suitable for priority scheduling. I would choose a linked list as the data structure for managing a task scheduler.

II. ***Linked list*** requires less memory while ***stack*** required more memory. Stack is more efficient at Undo. Undoing an action is a simple O(1) operation which is popping the top element. A linked list requires traversing the list from the end to the beginning, which is O(n). Unlike linked list, Stack is generally simpler to implement especially when it comes to handling edge cases. Linked list can be used to implement undo functionality but the overhead of node management and traversal makes it less efficient than stack so I would choose to use tack in this scenario.

III. Both linked list and Stack can be used to handle print jobs in a printer queue but a linked list is a more suitable choice because in order of processing, a linked list print jobs are processed in the order they are received which is FIFO, which is needed in this scenario. However, in a stack the print jobs are processed in reverse order which is LIFO which does not meet the requirements. In scalability, a linked list can handle a large number of print jobs without significant

performance degradation. Stack can become inefficient as the number of elements grow. Linked list is a more suitable data structure for handling print jobs in a printer queue.

IV. A linked list allows for efficient insertion and deletion of elements but searching for an element can take time because it needs traversing the whole list. In a contact list, if the list is large, searching for a contact by name could take time. In a stack the last element added to the stack is the first one to be removed. However a stack is not optimized for searching, searching for a contact would need traversing the entire stack which could be time consuming

V. ***Linked list:*** More efficient at insertion and deletion of nodes which can be useful when navigating back and forth in the browser history. It is more dynamic meaning linked list can grow or shrink as websites are added or removed from the history. However, Linked list would require more memory to store the next node's reference. It is slower in searching for a specific website. ***Stack:*** Efficient at insertion and removal of elements from the top which is perfect for navigating back in the browser history and searching is very fast because we only need to access the top element. However stack is fixed size which can lead to overflow issues if the browser history exceeds the allocated size and it is less flexible because only the top elements are accessible. I would choose stack over linked list because it is memory efficient. Stack uses less memory hence better performance.

C: By managing task scheduler, linked list is good at retrieving task is a specific order but use more memory due to overhead of node pointers.
Binary trees are fast at searching of tasks with an average time complexity of O(log n) but implementing a self-balancing BST can be complex to ensure efficient operations.

PART 2

A:

I.
```
FUNCTION MergeSort(transactions):
    IF length(transactions) <= 1:
        RETURN transactions
    Mid = length(transactions)/2
    Left = transactions[0…Mid]
    Right = transactions[Mid…length(transactions)]
    Left = MergeSort(Left)
    Right = MergeSort(Right)
    RETURN Merge(Left,Right)
FUNCTION Merge(Left,Right):
    Result = []
    While length(left) > 0 AND length(Right) > 0:
        IF Left[0].amount > Right[0].amaount:
            Result.append(Left[0])
            Left = Left[1…length(Left)]
        ELSE:
            Result.append(Right[0])
            Right = Right[1…length(Right)]
    Resutl = Result + Left + Right
    RETURN Result
FUNCTION GetTop10Trans(sortedTrans):
    Top10 = []
    FOR i = 0 TO 9:
        Top10.append(sortedTrans[i])
```

```
    RETURN top10

FUNCTION ProcessTrans(transactions):
    sortedTrans = MergeSort(transactions)
    top10Trans = GetTop10Trans(sortedTrans)
    Return top10Trans
```

II. ***OMEGA NOTATION:*** In the best case scenario, the quicksort algorithm has a time complexity of Omega (n log no), where n is the number of transactions. This occurs when the pivot is chosen such that it divides the array into two halves of equal size.
THETA NOTATION: In the average case scenario, the quicksort algorithm has a time complexity of Theta (n log n) because the algorithm is designed to minimize the number of comparisons needed to sort the array
BIG O NOTATION: In the worst case scenario, the quicksort algorithm has a time complexity of O (n^2) and occurs when the pivot is chosen poorly which results in a highly unbalanced partition.

III. (Code in Transaction.java file)

PART 3

A:

I. ***Bubble Sort:*** Generates an array of 100,000 random numbers between 0 to 100,000. Sorts it using Bubble Sort and print all the 100,000 elements of the unsorted and sorted arrays. Bubble sort works by repeatedly swapping the adjacent elements if they are in wrong order.

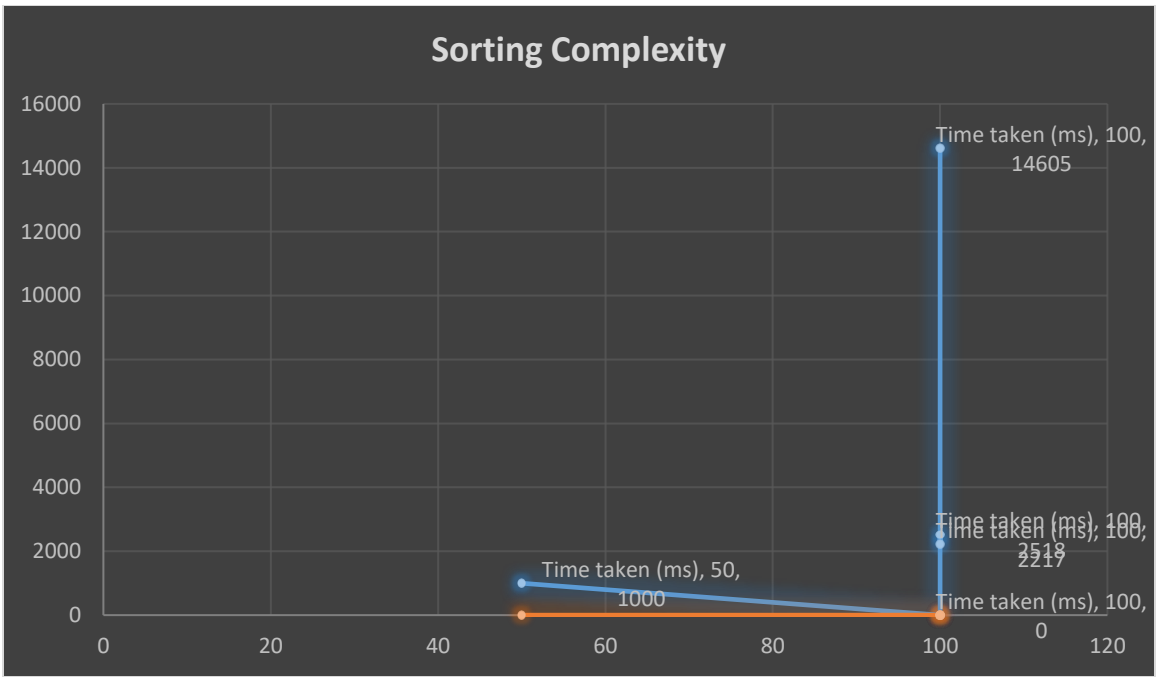
II. ***Selection Sort:*** The generateRandomArray() method generates an array of random numbers with a specific size. The selectionSort method sorts an array using the selection sort algorithm. It searches through the array and finds the minimum element in the unsorted part and swaps it with the current element. The printArray() method prints all the 100k elements of the array.

III. ***Insertion Sort:*** Insertions sort has a time complexity of O(n^2), which makes it inefficient for large datasets like the one used in the example of 100k elements.

IV. ***Quick Sort:*** Generates an array of 100k random numbers and sorts it using the quick sort algorithm. It recursively partitions the array around a pivot element and returns the index of the pivot element. The swap() method swaps two elements in the array.

V. ***Merge Sort:*** 100k random numbers are generated. The merge sort algorithm is implemented. It recursively divides the array into two halves until each subarray has only one element, and then merges the subarrays back together in sorted order. Two sorted arrays are merged into one sorted array.

B:



C:

Bubble sort: took the longest time to complete with a time complexity of 14.605 seconds but completed the algorithm. The performance of bubble sort is not impressive.

Selection sort: time complexity of only 2.217 seconds which is faster than bubble sort and completed the algorithm. Performance is good

Insertion sort: Time complexity of 2.518 seconds which is slightly slower than selection sort but also has good performance. Completed the algorithm.

Quick sort: (Did not work)

Merge sort: Time complexity of around 1 second which makes it the fastest but according to my results it did not complete the algorithm.