Data Structures and Algorithms Assignment 7

1. What is the distinction between a list and an array?

List	Array
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for declaration	Need to explicitly import a module for declaration
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Can be nested to contain different type of elements	Must contain either all nested elements of same size
Preferred for shorter sequence of data items	Preferred for longer sequence of data items
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, deletion has to be done element wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of array
Consume larger memory for easy addition of elements	Comparatively more compact in memory size

2. What are the qualities of a binary tree?

A binary tree consists of a number of nodes that contain the data to be stored (or pointers to the data), and the following structural characteristics :

- Each node has up to two direct child nodes.

- There is exactly one node, called the root of the tree, that has no parent node. All other nodes have exactly one parent.

- Nodes in a binary tree are placed according to this rule: the value of a node is greater than or equal to the values of any descendant in its left branch, and less than the value of any descendant in its right branch.

3. What is the best way to combine two balanced binary search trees?

We know that any element on the left side of the root is always going to be smaller than the root. And in this problem, to merging the two tree nodes, we can compare the two nodes of both the trees while adding into the result array simultaneously.

So, at any point in time, the comparisons that will be required to construct the result array will only be from those nodes which lie in the path from the root to the current node.

We can create two stacks for each tree and will maintain the stack as the smallest element on the top. Now, considering we have two stacks where the stack is sorted in ascending order from top to bottom, we can simply compare the top of both the stacks and put smaller ones in the result array, thereby popping up that element from the stack.

In this way, we could reduce memory usage as the maximum size of the stack will reach equal to the height of the tree. If the BST is height-balanced BST, then this approach will significantly affect the space complexity.

Solution Steps:

- Create two stacks, stack1 and stack2
- Iterate until tree1 is not NULL or tree2 is not NULL or stack1 is not empty or stack2 is not empty.
- For each iteration:
- Push all the left nodes of tree1 inside stack1
- Push all the left nodes of tree2 inside stack2
- Compare the top of both the stacks, put the smaller one in the result while popping it out from the respective stack and move to its right child.

4. How would you describe Heap in detail?

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.
Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

5. In terms of data structure, what is a HashMap?

HashMap is a dictionary data structure . It’s a Map-based collection class that is used to store data in Key & Value pairs. In this article, we’ll be creating our own hashmap implementation.

The benefit of using this data structure is faster data retrieval. It has data access complexity of O(1) in the best case.
To implement this structure,

- We need a list to store all the keys
- Key — Value relationship to get value based on key

6. How do you explain the complexities of time and space?

Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it takes for each statement to complete. As a result, it is highly dependent on the size of the processed data. It also aids in defining an algorithm's effectiveness and evaluating its performance.

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does the algorithm become incredibly slow? Is it able to maintain its fast run time as the input size grows? You can answer these questions thanks to Asymptotic Notation.

You can't compare two algorithms head to head. It is heavily influenced by the tools and hardware you use for comparisons, such as the operating system, CPU model, processor generation, and so on. Even if you calculate time and space complexity for two algorithms running on the same system, the subtle changes in the system environment may affect their time and space complexity.

As a result, you compare space and time complexity using asymptotic analysis. It compares two algorithms based on changes in their performance as the input size is increased or decreased.

Asymptotic notations are classified into three types:

- Big-Oh (O) notation
- Big Omega (Ω) notation
- Big Theta (Θ) notation

7. How do you recursively sort a stack?

The idea is simple – recursively remove values from the stack until the stack becomes empty and then insert those values (from the call stack) back into the stack in a sorted position.

Code:

```
from collections import deque
# Insert the given key into the sorted stack while maintaining its sorted order.
# This is similar to the recursive insertion sort routine
```

```
def sortedInsert(stack, key):

    # base case: if the stack is empty or
    # the key is greater than all elements in the stack
    if not stack or key > stack[-1]:
        stack.append(key)
        return

    ''' We reach here when the key is smaller than the top element '''

    # remove the top element
    top = stack.pop()

    # recur for the remaining elements in the stack
    sortedInsert(stack, key)

    # insert the popped element back into the stack
    stack.append(top)


# Recursive method to sort a stack
def sortStack(stack):

    # base case: stack is empty
    if not stack:
        return

    # remove the top element
    top = stack.pop()

    # recur for the remaining elements in the stack
    sortStack(stack)

    # insert the popped element back into the sorted stack
    sortedInsert(stack, top)


if __name__ == '__main__':

    A = [5, -2, 9, -7, 3]

    stack = deque(A)

    print('Stack before sorting:', list(stack))
    sortStack(stack)
    print('Stack after sorting:', list(stack))
```