Data Structures and Algorithms  **Recurrence Relation**

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

|  |  |
| --- | --- |
| LIST | ARRAY |
| It contain elements of multiple data types | It contains same data type |
| only allows sequential access to its elements | allow random access to the elements contained within them |
| used for large lists of data where the total number of items in the list is changing | better suited for small lists of data |
| There will be only used memory | There can be unused memory(used+unused) |
| 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 |
| Consume larger memory for easy addition of elements | Comparatively more compact in memory size |
| There is no need to explicitly import a module for declaration | We need to explicitly import a module for declaration |
|  |  |

2. What are the qualities of a binary tree?

1. The maximum number of nodes at level ‘n’ of a binary tree is 2n
2. Binary tree does not allow duplicate values.
3. when we create a binary tree and if an element is less than the value of its parent node then, it will placed on the left side of it else it will place on right side.
4. In Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children.

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

using stacks is the best way though this problem can be solved in multiple ways as it depends on the time/space complexity. Let us see below steps

1. Do inorder traversal of tree1 and store them in array1
2. Do inorder traversal of tree2 and store them in array2
3. Now merge array1 and array2 into array say array\_final that will contain the elements of both array1 and array2 into array\_final
4. Now do sorted array(array\_final) to build a balanced binary search tree, which is a merged version of the given trees.
5. The middle element of the array forms the root of the balanced BST and all the elements to the left of the middle element form the left sub-tree and all the elements to the right of the middle element form the right sub-tree.
6. Recursively do step 5 for the left subtree and attach it to the left of root.
7. Recursively do step 4 for the right subtree and attach it to the right of root.
8. Return root.

4. How would you describe Heap in detail?

* heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.
* A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.
* A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.
* 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 its 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 its 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?

* Hash table or a Hashmap is a type of data structure that maps keys to its value pairs (implement abstract array data types). It basically makes use of a function that computes an index value that in turn holds the elements to be searched, inserted, removed, etc. This makes it easy and fast to access data. In general, hash tables store key-value pairs and the key is generated using a hash function.
* Hash table or a Hashmap is a type of data structure that maps keys to its value pairs (implement abstract array data types). It basically makes use of a function that computes an index value that in turn holds the elements to be searched, inserted, removed, etc. This makes it easy and fast to access data. In general, hash tables store key-value pairs and the key is generated using a hash function.
* Hashing is the process of converting a given key into another smaller value for O(1) retrieval time.
* This is done by taking the help of some function or algorithm which is called as hash function to map data to some encrypted or simplified representative value which is termed as “hash code” or “hash”. This hash is then used as an index to narrow down search criteria to get data quickly.
* An example can be a mapping of student names and their student IDs or the names of students along with their student IDs.
* in python eg.

my\_dict\_date={'Ram' : '001' , 'Ajay': '002' , 'Neha': '003', 'Ronak': '004'}

print(my\_dict)

type(my\_dict)

* Output will be like

{'Ram' : '001' , 'Ajay': '002' , 'Neha': '003', 'Ronak': '004'}

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

* Space Complexity
* Space complexity is a measure of how efficient your code is in terms of memory used.
* Space complexity analysis happens almost in the same way time complexity analysis happens.
* For example, consider the following code :

vector<int> V;

for (int i = 0; i < N; i++) V.push\_back(i);

The code snippet ends up creating a vector of size N. So, space complexity of the code is O(N).

* Additional space / memory is measured in terms of the largest memory use by the program when it runs. That is to say, if you allocated O(N) memory, and later free it, that does not make the space complexity of your program O(1).
* Time Complexity
* Time complexity of an algorithm signifies the total time required by the program to run till its completion.
* The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.
* Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.
* And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size.
* For example, in case of addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c\*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.
* **While solving some questions I have seen that while decreasing time complexity space complexity increases and vice versa just like bias and variance trade-off**

7. How do you recursively sort a stack?

* The logic to solve above is to hold all values in function call stack until the stack becomes empty.
* once the stack becomes empty, insert all held items one by one in sorted order. Note that Here sorted order is important.
* eg. logic is

sortStack(stack S)

if stack is not empty:

temp = pop(S);

sortStack(S);

sortedInsert(S, temp);