DSA-101

Learn with Approach & then code with explain
Only way to clear coding round.

1. DSA Top 40 Interview Questions
   - Includes Arrays, LinkedList, stack, Queue, Map, hashing, tree, Graph.

2. Search & Sort (Importance is approach & Code in your way to explain)
   - ~~Own~~ Printed notes + own notes

3. Stack, Queues & ~~Miscellaneous~~ Stack, Queues Miscellaneous.

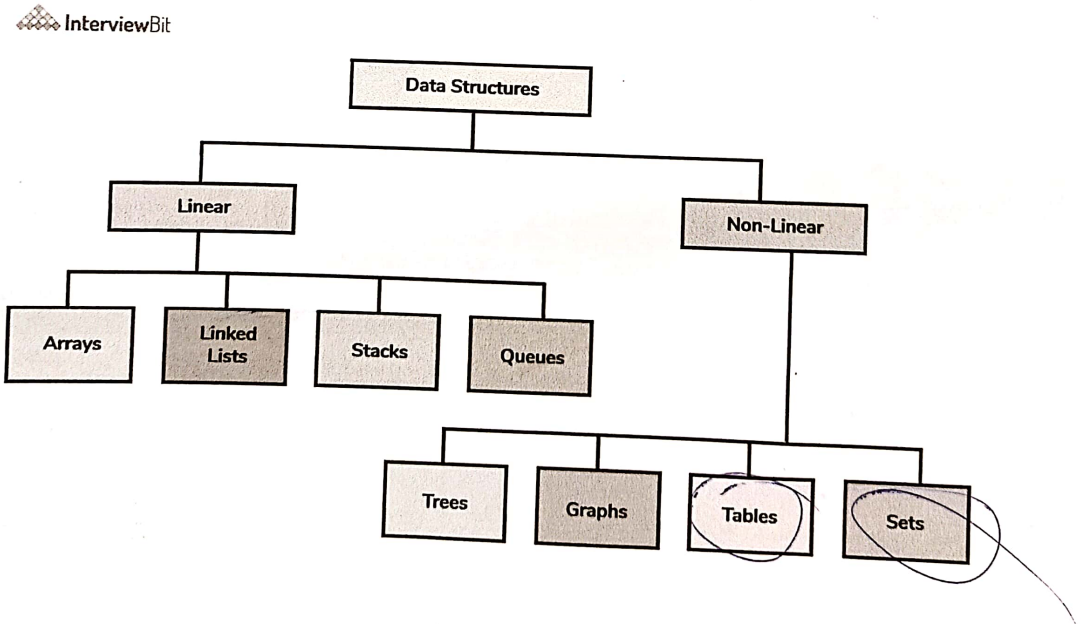
Data Structure Interview Questions

What is Data Structure?

- Data structure is a fundamental concept of any programming language, essential for algorithmic design.
- It is used for the efficient organization and modification of data.
- DS is how data and the relationship amongst different data is represented, that aids in how efficiently various functions or operations or algorithms can be applied.

Types

- There are two types of data structures:
  - Linear data structure: If the elements of a data structure result in a sequence or a linear list then it is called a linear data structure. Example: Arrays, Linked List, Stacks, Queues etc.
  - Non-linear data structure: If the elements of data structure results in a way that traversal of nodes is not done in a sequential manner, then it is a non linear data structure. Example: Trees, Graphs etc.



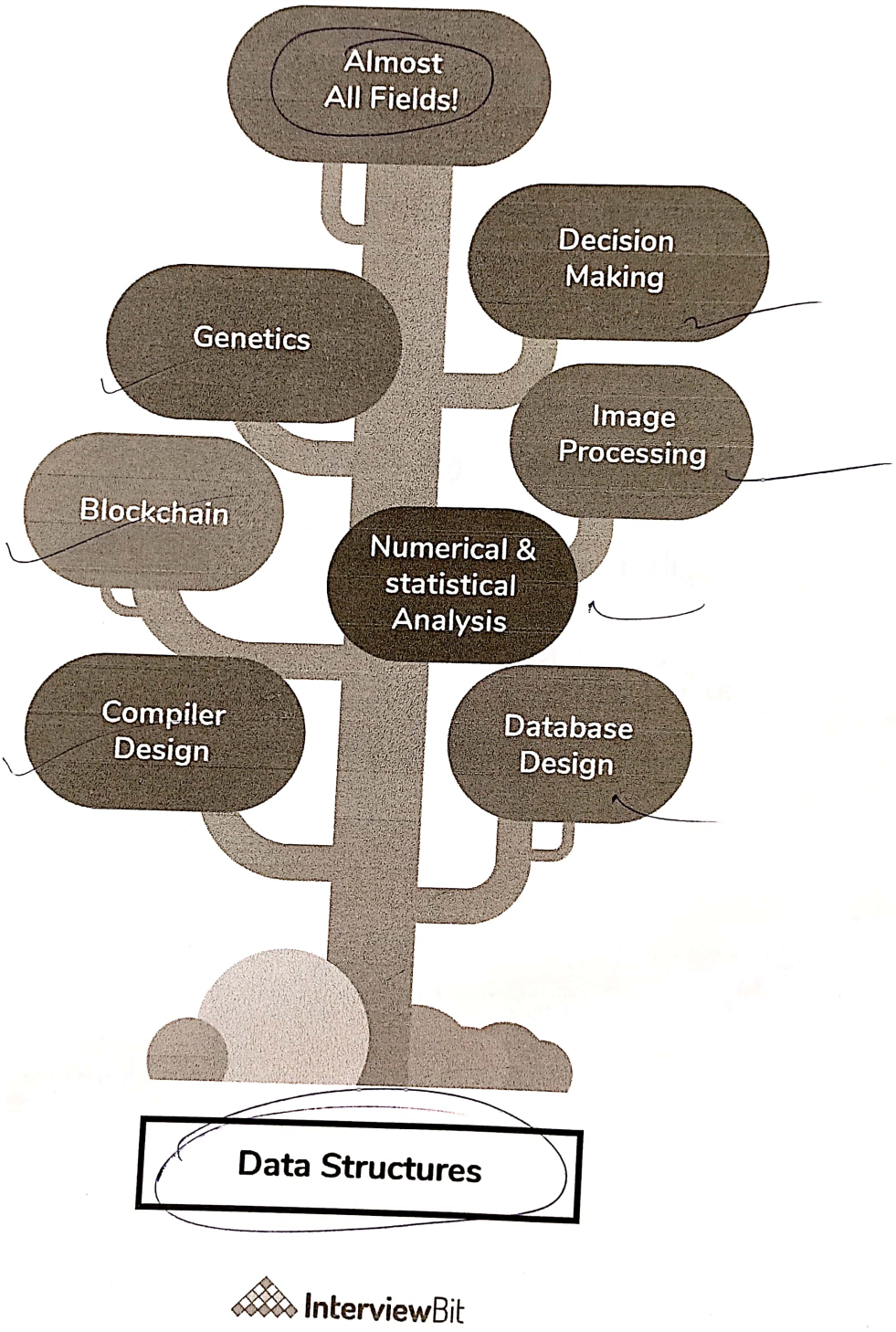
Applications

Data structures form the core foundation of software programming as any efficient algorithm to a given problem is dependent on how effectively a data is structured.

- Identifiers look ups in compiler implementations are built using hash tables.
- The B-trees data structures are suitable for the databases implementation.
- Some of the most important areas where data structures (https://www.scaler.com/courses/data-structures-and-algorithms) are used are as follows:

1. Artificial intelligence
2. Compiler design
3. Machine learning
4. Database design and management
5. Blockchain
6. Numerical and Statistical analysis
7. Operating system development
8. Image & Speech Processing
9. Cryptography

Applications of Data Structures



Benefits of Learning Data Structures

DSA Interview Questions

Top - 40 Data Structures Interview Question & Answers (2021) - Interview Bit

INTERVIEW QUESTIONS AND ANSWERS DATA STRUCTURE AND ALGORITHM

1. Can you explain the difference between file structure and storage structure?

File Structure: Representation of data into secondary or auxiliary memory say any device such as hard disk or pen drives that stores data which remains intact until manually deleted is known as a file structure representation.

Storage Structure: In this type, data is stored in the main memory i.e RAM, and is deleted once the function that uses this data gets completely executed.

The difference is that storage structure has data stored in the memory of the computer system, whereas file structure has the data stored in the auxiliary memory.

2. Can you tell how linear data structures differ from non-linear data structures?

If the elements of a data structure result in a sequence or a linear list then it is called a linear data structure. Whereas, traversal of nodes happens in a non-linear fashion in non-linear data structures.

Lists, stacks, and queues are examples of linear data structures whereas graphs and trees are the examples of non-linear data structures.

3. What is an array?

Arrays are the collection of similar types of data stored at contiguous memory locations.

It is the simplest data structure where the data element can be accessed randomly just by using its index number.

4. What is a multidimensional array?

Multi-dimensional arrays are those data structures that span across more than one dimension.

This indicates that there will be more than one index variable for every point of storage. This type of data structure is primarily used in cases where data cannot be represented or stored using only one dimension. Most commonly used multidimensional arrays are 2D arrays.

2D arrays emulates the tabular form structure which provides ease of holding the bulk of data that are accessed using row and column pointers.

8. Explain the scenarios where you can use linked lists and arrays.

Following are the scenarios where we use linked list over array:

- When we do not know the exact number of elements beforehand.
- When we know that there would be large number of add or remove operations.
- Less number of random access operations.
- When we want to insert items anywhere in the middle of the list, ~~such as when implementing~~ a priority queue, linked list is more suitable.

Below are the cases where we use arrays over the linked list:

- When we need to index or randomly access elements more frequently.
- When we know the number of elements in the array ~~beforehand in order to allocate~~ the right amount of memory.
- When we need speed while iterating over the elements in the sequence.
- When memory is a concern:

Due to the nature of arrays and linked list, it is safe to say that filled arrays use less memory than linked lists.

Each element in the array indicates just the data whereas each linked list node represents the data as well as one or more pointers or references to the other elements in the linked list.

To summarize, requirements of space, time, and ease of implementation are considered while deciding which data structure has to be used over what.

9. What is a doubly-linked list (DLL)? What are its applications.

This is a complex type of a linked list wherein a node has two references:

One that connects to the next node in the sequence

Another that connects to the previous node.

This structure allows traversal of the data elements in both directions (left to right and vice versa).

Applications of DLL are:

A music playlist with next song and previous song navigation options.

The browser cache with BACK-FORWARD visited pages

The undo and redo functionality on platforms such as word, paint etc, where you can reverse the node to get to the previous page.

10. What is a stack? What are the applications of stack?

Stack is a linear data structure that follows LIFO (Last In First Out) approach for accessing elements.

Push, pop, and top (or peek) are the basic operations of a stack.

Following are some of the applications of a stack:

Check for balanced parentheses in an expression

Evaluation of a postfix expression

Problem of Infix to postfix conversion

Reverse a string

11. What is a queue? What are the applications of queue?

A queue is a linear data structure that follows the FIFO (First In First Out) approach for accessing elements.

Dequeue from the queue, enqueue element to the queue, get front element of queue, and get rear element of queue are basic operations that can be performed.

Some of the applications of queue are:

CPU Task scheduling

BFS algorithm to find shortest distance between two nodes in a graph.

Website request processing

Used as buffers in applications like MP3 media player, CD player, etc.

Managing an Input stream

12. How is a stack different from a queue?

In a stack, the item that is most recently added is removed first whereas in queue, the item least recently added is removed first.

13. Explain the process behind storing a variable in memory.

A variable is stored in memory based on the amount of memory that is needed. Following are the steps followed to store a variable:

The required amount of memory is assigned first.

Then, it is stored based on the data structure being used.

Using concepts like dynamic allocation ensures high efficiency and that the storage units can be accessed based on requirements in real time.

14. How to implement a queue using stack?

A queue can be implemented using two stacks. Let q be the queue andstack1 and stack2 be the 2 stacks for implementing q. We know that stack supports push, pop, peek operations and using these operations, we need to emulate the operations of queue - enqueue and dequeue. Hence, queue q can be implemented in two methods (Both the methods use auxillary space complexity of O(n)):

⇒ By making enqueue operation costly:

Here, the oldest element is always at the top of stack1 which ensures dequeue operation to occur in O(1) time complexity.

To place element at top of stack1, stack2 is used.

Pseudocode:

Enqueue: Here time complexity will be O(n)

```
enqueue(q, data):
  While stack1 is not empty:
    Push everything from stack1 to stack2.
    Push data to stack1
    Push everything back to stack1.
```

Dequeue: Here time complexity will be O(1)

```
deQueue(q):
  If stack1 is empty then error
  else
    Pop an item from stack1 and return it
```

⇒ By making dequeue operation costly:

Here, for enqueue operation, the new element is pushed at the top of stack1. Here, the enqueue operation time complexity is O(1).

In dequeue, if stack2 is empty, all elements from stack1 are moved to stack2 and top of stack2 is the result. Basically, reversing the list by pushing to a stack and returning the first enqueued element. This operation of pushing all elements to new stack takes O(n) complexity.

Pseudocode:

Enqueue: Time complexity: O(1)

enqueue(q, data):

Push data to stack1

Dequeue: Time complexity: O(n)

dequeue(q):

If both stacks are empty then raise error.

If stack2 is empty:

While stack1 is not empty:

push everything from stack1 to stack2.

Pop the element from stack2 and return it.

15. How do you implement stack using queues?

A stack can be implemented using two queues. We know that a queue supports enqueue and dequeue operations. Using these operations, we need to develop push, pop operations.

Let stack be 's' and queues used to implement be 'q1' and 'q2'. Then, stack 's' can be implemented in two ways:

By making push operation costly:

This method ensures that newly entered element is always at the front of 'q1', so that pop operation just dequeues from 'q1'.

'q2' is used as auxillary queue to put every new element at front of 'q1' while ensuring pop happens in O(1) complexity.

Pseudocode:

Push element to stack s : Here push takes O(n) time complexity.

push(s, data):

Enqueue data to q2

Dequeue elements one by one from q1 and enqueue to q2.

Swap the names of q1 and q2

Pop element from stack s: Takes O(1) time complexity.

pop(s):

dequeue from q1 and return it.

By making pop operation costly:

In push operation, the element is enqueued to q1.

In pop operation, all the elements from q1 except the last remaining element, are pushed to q2 if it is empty. That last element remaining of q1 is dequeued and returned.

Pseudocode:

Push element to stack s : Here push takes O(1) time complexity.

```
push(s,data):
  Enqueue data to q1
```

Pop element from stack s: Takes O(n) time complexity.

```
pop(s):
  Step1: Dequeue every elements except the last element from q1 and enqueue to q2.
  Step2: Dequeue the last item of q1, the dequeued item is stored in result variable.
  Step3: Swap the names of q1 and q2 (for getting updated data after dequeue)
  Step4: Return the result.
```

16. What is hashmap in data structure?

Hashmap is a data structure that uses implementation of hash table data structure which allows access of data in constant time (O(1)) complexity if you have the key.

17. What is the requirement for an object to be used as key or value in HashMap?

The key or value object that gets used in hashmap must implement equals() and hashcode() method.

The hash code is used when inserting the key object into the map and equals method is used when trying to retrieve a value from the map.

18. How does HashMap handle collisions in Java?

The java.util.HashMap class in Java uses the approach of chaining to handle collisions. In chaining, if the new values with same key are attempted to be pushed, then these values are stored in a linked list stored in bucket of the key as a chain along with the existing value.

In the worst case scenario, it can happen that all key might have the same hashcode, which will result in the hash table turning into a linked list. In this case, searching a value will take O(n) complexity as opposed to O(1) time due to the nature of the linked list. Hence, care has to be taken while selecting hashing algorithm.

Best in Java 8 → It will convert LinkedList to Tree and hence from O(n) it becomes O(log n)

19. What is the time complexity of basic operations get() and put() in HashMap class?

The time complexity is O(1) assuming that the hash function used in hash map distributes elements uniformly among the buckets.

20. Which data structures are used for implementing LRU cache?

LRU cache or Least Recently Used cache allows quick identification of an element that hasn't been put to use for the longest time by organizing items in order of use. In order to achieve this, two data structures are used:

Queue – This is implemented using a doubly-linked list. The maximum size of the queue is determined by the cache size, i.e by the total number of available frames. The least recently used pages will be near the front end of the queue whereas the most recently used pages will be towards the rear end of the queue.

Hashmap – Hashmap stores the page number as the key along with the address of the corresponding queue node as the value.

21. What is a priority queue?

A priority queue is an abstract data type that is like a normal queue but has priority assigned to elements.

Elements with higher priority are processed before the elements with a lower priority.

In order to implement this, a minimum of two queues are required - one for the data and the other to store the priority.

22. Can we store a duplicate key in HashMap?

No, duplicate keys cannot be inserted in HashMap. If you try to insert any entry with an existing key, then the old value would be overridden with the new value. Doing this will not change the size of HashMap.

This is why the keySet() method returns all keys as a SET in Java since it doesn't allow duplicates.

23. What is a tree data structure?

Tree is a recursive, non-linear data structure consisting of the set of one or more data nodes where one node is designated as the root and the remaining nodes are called as the children of the root.

Tree organizes data into hierarchial manner.

The most commonly used tree data structure is a binary tree and its variants.

Some of the applications of trees are:

Filesystems —files inside folders that are inturn inside other folders.

Comments on social media — comments, replies to comments, replies to replies etc form a tree representation.

Family trees — parents, grandparents, children, and grandchildren etc that represents the family hierarchy.

24. What are Binary trees?

A binary Tree is a special type of tree where each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets, i.e. the root of the tree, left sub-tree and right sub-tree.

25. What is the maximum number of nodes in a binary tree of height k?

The maximum nodes are : 2k+1-1 where k >= 1

26. Write a recursive function to calculate the height of a binary tree in Java.

Consider that every node of a tree represents a class called Node as given below:

```
public class Node{
   int data;
   Node left;
   Node right;
}
```

Then the height of the binary tree can be found as follows:

```
int heightOfBinaryTree(Node node)
{
   if (node == null)
      return 0; // If node is null then height is 0 for that node.
   else
   {
      // compute the height of each subtree
      int leftHeight = heightOfBinaryTree(node.left);
      int rightHeight = heightOfBinaryTree(node.right);

      //use the larger among the left and right height and plus 1 (for the root)
      return Math.max(leftHeight, rightHeight) + 1;
   }
}
```

27. Write Java code to count number of nodes in a binary tree.

```
int countNodes(Node root)
{
   int count =  1;  //Root itself should be counted
   if (root ==null)
      return 0;
   else
   {
      count += countNodes(root.left);
      count += countNodes(root.right);
      return count;
   }
}
```

These are recursive calls.

28. What are tree traversals?

Tree traversal is a process of visiting all the nodes of a tree. Since root (head) is the first node and all nodes are connected via edges (or links) we always start with that node. There are three ways which we use to traverse a tree –

Inorder Traversal:

Algorithm:

Step 1. Traverse the left subtree, i.e., call Inorder(root.left)

Step 2. Visit the root.

Step 3. Traverse the right subtree, i.e., call Inorder(root.right)

Inorder traversal in Java:

```
// Print inorder traversal of given tree.
void printInorderTraversal(Node root)
{
   if (root == null)
      return;
```

```
        //first traverse to the left subtree
        printInorderTraversal(root.left);

        //then print the data of node
        System.out.print(root.data + " ");

        //then traverse to the right subtree
        printInorderTraversal(root.right);
    }
```

Uses: In binary search trees (BST), inorder traversal gives nodes in ascending order.

Preorder Traversal:

Algorithm:

Step 1. Visit the root.

Step 2. Traverse the left subtree, i.e., call Preorder(root.left)

Step 3. Traverse the right subtree, i.e., call Preorder(root.right)

Preorder traversal in Java:

```
    // Print preorder traversal of given tree.
    void printPreorderTraversal(Node root)
    {
        if (root == null)
            return;
        //first print the data of node
        System.out.print(root.data + " ");

        //then traverse to the left subtree
        printPreorderTraversal(root.left);

        //then traverse to the right subtree
        printPreorderTraversal(root.right);
```

}

Uses:

Preorder traversal is commonly used to create a copy of the tree.

It is also used to get prefix expression of an expression tree.

Postorder Traversal:

Algorithm:

Step 1. Traverse the left subtree, i.e., call Postorder(root.left)

Step 2. Traverse the right subtree, i.e., call Postorder(root.right)

Step 3. Visit the root.

Postorder traversal in Java:

```
// Print postorder traversal of given tree.
void printPostorderTraversal(Node root)
{
 if (root == null)
    return;

 //first traverse to the left subtree
 printPostorderTraversal(root.left);

 //then traverse to the right subtree
 printPostorderTraversal(root.right);

 //then print the data of node
 System.out.print(root.data + " ");

}
```

Uses:

Postorder traversal is commonly used to delete the tree.

It is also useful to get the postfix expression of an expression tree.

Consider the following tree as an example, then:

Inorder Traversal => Left, Root, Right : [4, 2, 5, 1, 3]

Preorder Traversal => Root, Left, Right : [1, 2, 4, 5, 3]

Postorder Traversal => Left, Right, Root : [4, 5, 2, 3, 1]

29. What is a Binary Search Tree?

A binary search tree (BST) is a variant of binary tree data structure that stores data in a very efficient manner such that the values of the nodes in the left sub-tree are less than the value of the root node, and the values of the nodes on the right of the root node are correspondingly higher than the root.

Also, individually the left and right sub-trees are their own binary search trees at all instances of time.

30. What is an AVL Tree?

AVL trees are height balancing BST. AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called Balance Factor and is calculates as.

BalanceFactor = height(left subtree) – height(right subtree)

31. Print Left view of any binary trees.

The main idea to solve this problem is to traverse the tree in pre order manner and pass the level information along with it. If the level is visited for the first time, then we store the information of the current node and the current level in the hashmap. Basically, we are getting the left view by noting the first node of every level.

At the end of traversal, we can get the solution by just traversing the map.

Consider the following tree as example for finding the left view:

Left view of a binary tree in Java:

```
import java.util.HashMap;


//to store a Binary Tree node
class Node
{
  int data;
```

```
    Node left = null, right = null;

    Node(int data) {
      this.data = data;
    }
  }
  public class InterviewBit
  {
    // traverse nodes in pre-order way
    public static void leftViewUtil(Node root, int level, HashMap<Integer, Integer> map)
    {
      if (root == null) {
        return;
      }

      // if you are visiting the level for the first time
      // insert the current node and level info to the map
      if (!map.containsKey(level)) {
        map.put(level, root.data);
      }

    leftViewUtil(root.left, level + 1, map);
    leftViewUtil(root.right, level + 1, map);
  }

// to print left view of binary tree
public static void leftView(Node root)
{
  // create an empty HashMap to store first node of each level
```

```
        HashMap<Integer, Integer> map = new HashMap<>();

        // traverse the tree and find out the first nodes of each level
        leftViewUtil(root, 1, map);

        // iterate through the HashMap and print the left view
        for (int i = 0; i <map.size(); i++) {
          System.out.print(map.get(i) + " ");
        }
      }

      public static void main(String[] args)
      {
        Node root = new Node(4);
        root.left = new Node(2);
        root.right = new Node(6);
        root.left.left = new Node(1);
        root.left.left = new Node(3);
        root.right.left = new Node(5);
        root.right.right = new Node(7);
        root.right.left.left = new Node(9);

        leftView(root);
      }
    }
```

32. What is a graph data structure?

Graph is a type of non-linear data structure that consists of vertices or nodes connected by edges or links for storing data. Edges connecting the nodes may be directed or undirected.

33. What are the applications of graph data structure?

Graphs are used in wide varieties of applications. Some of them are as follows:

Social network graphs to determine the flow of information in social networking websites like facebook, linkedin etc.

Neural networks graphs where nodes represent neurons and edge represent the synapses between them

Transport grids where stations are the nodes and routes are the edges of the graph.

Power or water utility graphs where vertices are connection points and edge the wires or pipes connecting them.

Shortest distance between two end points algorithms.

34. How do you represent a graph?

We can represent a graph in 2 ways:

Adjacency matrix: Used for sequential data representation

Adjacency list: Used to represent linked data

35. What is the difference between tree and graph data structure?

Tree and graph are differentiated by the fact that a tree structure must be connected and can never have loops whereas in the graph there are no restrictions.

Tree provides insights on relationship between nodes in a hierarchical manner and graph follows a network model.

36. What is the difference between the Breadth First Search (BFS) and Depth First Search (DFS)?

BFS and DFS both are the traversing methods for a graph. Graph traversal is nothing but the process of visiting all the nodes of the graph.

The main difference between BFS and DFS is that BFS traverses level by level whereas DFS follows first a path from the starting to the end node, then another path from the start to end, and so on until all nodes are visited.

Furthermore, BFS uses queue data structure for storing the nodes whereas DFS uses the stack for traversal of the nodes for implementation.

DFS yields deeper solutions that are not optimal, but it works well when the solution is dense whereas the solutions of BFS are optimal.

You can learn more about BFS here: Breadth First Search and DFS here: Depth First Search.

37. How do you know when to use DFS over BFS?

The usage of DFS heavily depends on the structure of the search tree/graph and the number and location of solutions needed. Following are the best cases where we can use DFS:

If it is known that the solution is not far from the root of the tree, a breadth first search (BFS) might be better.

If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.

If the tree is very wide, a BFS might need too much memory, so it might be completely impractical. We go for DFS in such cases.

If solutions are frequent but located deep in the tree we opt for DFS.

38. What is topological sorting in a graph?

Topological sorting is a linear ordering of vertices such that for every directed edge ij, vertex i comes before j in the ordering.

Topological sorting is only possible for Directed Acyclic Graph (DAG).

Applications:

jobs scheduling from the given dependencies among jobs.

ordering of formula cell evaluation in spreadsheets

ordering of compilation tasks to be performed in make files,

data serialization

resolving symbol dependencies in linkers.

Topological Sort Code in Java:

```
// V - total vertices
// visited - boolean array to keep track of visited nodes
// graph - adjacency list.
// Main Topological Sort Function.
void topologicalSort()
{
    Stack<Integer> stack = new Stack<Integer>();
```

```
        // Mark all the vertices as not visited
        boolean visited[] = new boolean[V];
        for (int j = 0; j < V; j++){
          visited[j] = false;
        }
        // Call the util function starting from all vertices one by one
        for (int i = 0; i < V; i++)
          if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

        // Print contents of stack -> result of topological sort
        while (stack.empty() == false)
          System.out.print(stack.pop() + " ");
    }

    // A helper function used by topologicalSort
    void topologicalSortUtil(int v, boolean visited[],
                             Stack<Integer> stack)
    {
      // Mark the current node as visited.
      visited[v] = true;
      Integer i;

      // Recur for all the vertices adjacent to the current vertex
      Iterator<Integer> it = graph.get(v).iterator();
      while (it.hasNext()) {
        i = it.next();
        if (!visited[i])
```

```
            topologicalSortUtil(i, visited, stack);
        }

        // Push current vertex to stack that saves result
        stack.push(new Integer(v));
    }
```

39 Given an m x n 2D grid map of '1's which represents land and '0's that represents water, return the number of islands (surrounded by water and formed by connecting adjacent lands in 2 directions - vertically or horizontally). Assume that the boundary cases - which is all four edges of the grid are surrounded by water.

Constraints are:

m == grid.length

n == grid[i].length

1 <= m, n <= 300

grid[i][j] can only be '0' or '1'.

Example:

Input: grid = [

["1" , "1" , "1" , "0" , "0"],

["1" , "1" , "0" , "0" , "0"],

["0" , "0" , "1" , "0" , "1"],

["0" , "0" , "0" , "1" , "1"]

]

Output: 3

Solution:

```
class InterviewBit {
  public int numberOfIslands(char[][] grid) {
    if(grid==null || grid.length==0||grid[0].length==0)
```

```
        return 0;

      int m = grid.length;
      int n = grid[0].length;

      int count=0;
      for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
          if(grid[i][j]=='1'){
            count++;
            mergeIslands(grid, i, j);
          }
        }
      }

    return count;
}

public void mergeIslands(char[][] grid, int i, int j){
  int m=grid.length;
  int n=grid[0].length;

 if(i<0||i>=m||j<0||j>=n||grid[i][j]!='1')
    return;

 grid[i][j]='X';

mergeIslands(grid, i-1, j);
mergeIslands(grid, i+1, j);
```

```
        mergeIslands(grid, i, j-1);
        mergeIslands(grid, i, j+1);
    }
}
```

40 What is a heap data structure?

Heap is a special tree-based non-linear data structure in which the tree is a complete binary tree. A binary tree is said to be complete if all levels are completely filled except possibly the last level and the last level has all elements towards as left as possible. Heaps are of two types:

Max-Heap:

In a Max-Heap the data element present at the root node must be greatest among all the data elements present in the tree.

This property should be recursively true for all sub-trees of that binary tree.

Min-Heap:

In a Min-Heap the data element present at the root node must be the smallest (or minimum) among all the data elements present in the tree.

This property should be recursively true for all sub-trees of that binary tree.

RESOURCE---https://www.interviewbit.com/data-structure-interview-questions/

Sortings

- Insertion
- Bubble
- Selection
- Quick
- Merge
- Heap

DATA STRUCTURE (BASICS)

[SORTING IN C++ & JAVA]

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

TOPICS COVERED-

- Introduction to Sorting
- Quick Sort
- Merge Sort
- Counting Sort
- Heap Sort
- sort() Function in C++ STL
- Sorting using Built-in methods in Java

Introduction to Sorting-

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

For Example, the array arr[] = {5, 4, 2, 1, 3} after *sorting in increasing order* will be: arr[] = {1, 2, 3, 4, 5}. The same array after *sorting in descending order* will be: arr[] = {5, 4, 3, 2, 1}.

In-Place Sorting: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

In this tutorial, we will see three of such in-place sorting algorithms, namely:

- Insertion Sort
- Selection Sort
- Bubble Sort

Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.

The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

In the below image you can see, how the array [4, 3, 2, 10, 12, 1, 5, 6] is being sorted in increasing order following the insertion sort algorithm.

Insertion Sort Execution Example

→ 'hold 5' and after '4' i.e. [10, 12, 5]

Algorithm:

```
Step 1: If the current element is 1st element of array,
        it is already sorted.
Step 2: Pick next element
Step 3: Compare the current element will all elements
        in the sorted sub-array before it.
Step 4: Shift all of the elements in the sub-array before
        the current element which are greater than the current
        element by one place and insert the current element
        at the new empty space.
Step 5: Repeat step 2-3 until the entire array is sorted.
```

Another Example:

arr[] = {**12**, 11, 13, 5, 6}

Let us loop for i = 1 (second element of the array) to 4 (Size of input array - 1).

- *i = 1*, Since 11 is smaller than 12, move 12 and insert 11 before 12.
  11, 12, 13, 5, 6
- *i = 2*, 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
  11, 12, 13, 5, 6
- *i = 3*, 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
  5, 11, 12, 13, 6
- *i = 4*, 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
  5, 6, 11, 12, 13

Function Implementation:

Function to sort an array using insertion sort

```
void insertionSort(int arr[], int n)
{
 int i, key, j;
 for (i = 1; i < n; i++)
 {
   key = arr[i];
   j = i-1;

/* Move elements of arr[0..i-1], that are
   greater than key, to one position ahead
```

key = arr[i]; ——→ Pick one element from unsorted list

j = i-1; ——→ Get the last index of sorted list.

```
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

shift all the elements to right in sorted list if it's greater.

Time Complexity: $O(N^2)$, where N is the size of the array.

Bubble Sort-

Bubble Sort is also an in-place sorting algorithm. This is the simplest sorting algorithm and it works on the principle that:

In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.

Example: Consider the array, arr[] = {5, 1, 4, 2, 8}.

- **First Pass:** (**5 1** 4 2 8) --> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 **5 4** 2 8) --> (1 **4 5** 2 8), Swap since 5 > 4
  (1 4 **5 2** 8) --> (1 4 **2 5** 8), Swap since 5 > 2
  (1 4 2 **5 8**) --> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
- **Second Pass:** (**1 4** 2 5 8) --> (**1 4** 2 5 8)
  (1 **4 2** 5 8) --> (1 **2 4** 5 8), Swap since 4 > 2
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

- **Third Pass:** (1 2 4 5 8) --> (1 2 4 5 8)
  (1 2 4 5 8) --> (1 2 4 5 8)
  (1 2 4 5 8) --> (1 2 4 5 8)
  (1 2 4 5 8) --> (1 2 4 5 8)

Function Implementation:

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
 int i, j;
 for (i = 0; i < n-1; i++)

  // Last i elements are already in place
  for (j = 0; j < n-i-1; j++)
   if (arr[j] > arr[j+1])
    swap(&arr[j], &arr[j+1]);
}
```

Note: The above solution can be further optimized by keeping a flag to check if the array is already sorted in the first pass itself and to stop any further iteration.

Time Complexity: $O(N^2)$

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
arr[] = 64 25 12 22 11.

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Function Implementation:

```
void selectionSort(int arr[], int n)
{
  int i, j, min_idx;

  // One by one move boundary of unsorted subarray
  for (i = 0; i < n-1; i++)
  {
    // Find the minimum element in unsorted array
    min_idx = i;
    for (j = i+1; j < n; j++)
      if (arr[j] < arr[min_idx])
        min_idx = j;

    // Swap the found minimum element with the first element
    swap(&arr[min_idx], &arr[i]);
  }
}
```

Time Complexity: $O(N^2)$

Quick Sort- → Assembly line sorting way

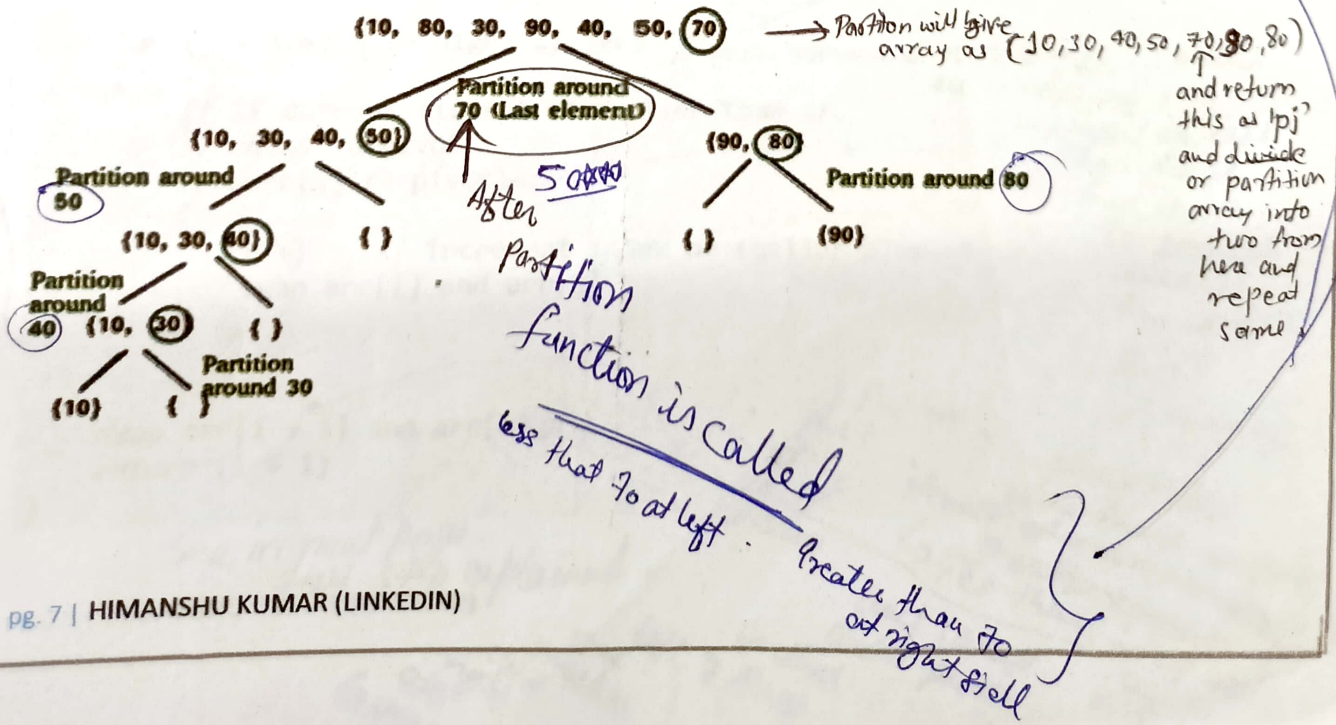
QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm:

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }

    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

I chose Right most element as Pivot

// Now from the rest (n-1) elements in array, if any element is less than Pivot then swap it to smaller index(i) from index(j)

(i<j) always

Putting pivot at correct position and at end.

Iterate when there is no more greater element. Get that (i), and swap with pivot

See in next page full step explained.

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes:  0   1   2   3   4   5   6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
```

```
// No change in i and arr[]
```

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, **40**, 90, **80**, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, **50**, 80, **90**, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, **70**, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

and return (i+1)

this is where we've put the pivot after partition

Implementation:

```
/* This function takes last element as pivot, places
 the pivot element at its correct position in sorted
 array, and places all smaller (smaller than pivot)
 to left of pivot and all greater elements to right
 of pivot */
int partition (int arr[], int low, int high)
```

Approach & corresponding code understanding

```
int partition (int[] arr, int low, int high)
{
    int pivot = arr[high];   // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    //At the end swap the higher(pivot) to its right position.
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

for larger element index pointer.

SOS

we are trying to bring smaller elements at initial or starting index left hand side.

Iterate from j=0 to high-1

arr[high] since is chosen as pivot

arr = {10, 80, 30, 90, 40, 50, 70}

pivot.

Smaller element index pointer

doesn't know what its making is

To understand check previous 2 pages well explained though.

```
/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
 low  --> Starting index,
 high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

arr[pi]

SOS

Final index of pivot after partitioning it.

SOS

Here no tree like recursion will happen like in merge sort.

BCOZ here we are just putting pi (pivot) to its right position & recursively doing for befor & after partitions of sub-arrays.

```
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

```
T(n) = T(k) + T(n-k-1) + Θ(n)
```

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

```
 T(n) = T(0) + T(n-1) + Θ(n)
which is equivalent to
 T(n) = T(n-1) + Θ(n)
```

The solution of above recurrence is $\Theta(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

```
 T(n) = 2T(n/2) + Θ(n)
```

The solution of above recurrence is **Θ(nLogn)**. It can be solved using case 2 of Master Theorem.***Average Case:*** To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

```
 T(n) = T(n/9) + T(9n/10) + Θ(n)
```

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like **Merge Sort** and **Heap Sort**, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most

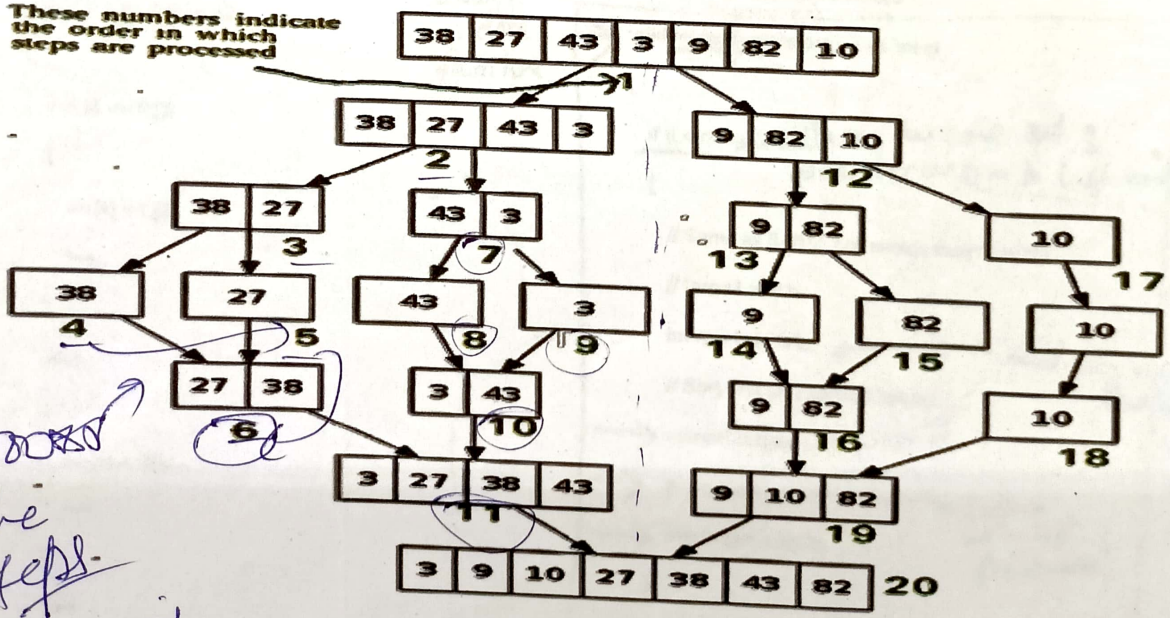
real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Merge Sort-

Merge Sort is a Divide and Conquer algorithm. It divides the input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one in a sorted manner. See following implementation for details:

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Learn briefly, its wookey

Implementation:

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
  int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;

  /* create temp arrays */
  int L[n1], R[n2];

  /* Copy data to temp arrays L[] and R[] */
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[m + 1+ j];

  /* Merge the temp arrays back into arr[l..r]*/
  i = 0; // Initial index of first subarray
  j = 0; // Initial index of second subarray
  k = l; // Initial index of merged subarray
  while (i < n1 && j < n2)
  {
    if (L[i] <= R[j])
    {
      arr[k] = L[i];
      i++;
    }
    else
    {
      arr[k] = R[j];
      j++;
    }
    k++;
  }

  /* Copy the remaining elements of L[], if there
     are any */
  while (i < n1)
  {
    arr[k] = L[i];
    i++;
    k++;
  }

  /* Copy the remaining elements of R[], if there
     are any */
  while (j < n2)
  {
    arr[k] = R[j];
    j++;
    k++;
  }
}

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
  if (l < r)
  {
    // Same as (l+r)/2, but avoids overflow for
    // large l and h
    int m = l+(r-l)/2;

    // Sort first and second halves
    mergeSort(arr, l, m);
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r);
  }
}
```

Need to understand these also.

Putting & (readily in Left array & Right array

Actually comparing and putting in sorted order from given two arrays is done from here

condition to come out of one recursion in left and go to R

need to understand these things also because it goes very deep

$$l + \frac{r-l}{2} = l + \frac{r}{2} - \frac{l}{2} = \frac{l}{2} + \frac{r}{2} = \left(\frac{l+r}{2}\right)$$

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \Theta(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(n\text{Log}n)$.

Time complexity of Merge Sort is **$\Theta(n\text{Log}n)$** in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

(Ignoring this)

Counting Sort-

It is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0  1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
the output sequence.

  3) Output each object from the input sequence followed by
  decreasing its count by 1.
  Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
  Put data 1 at index 2 in output. Decrease count by 1 to place
  next data 1 at an index 1 smaller than this index.
```

Implementation:

```
// The main function that sort the given string arr[] in
// alphabetical order
void countSort(char arr[])
{
  // The output character array that will have sorted arr
  char output[strlen(arr)];

  // Create a count array to store count of inidividul
  // characters and initialize count array as 0
  int count[RANGE + 1], i;
  memset(count, 0, sizeof(count));

  // Store count of each character
  for(i = 0; arr[i]; ++i)
    ++count[arr[i]];

  // Change count[i] so that count[i] now contains actual
  // position of this character in output array
  for (i = 1; i <= RANGE; ++i)
    count[i] += count[i-1];

  // Build the output character array
  for (i = 0; arr[i]; ++i)
  {
    output[count[arr[i]]-1] = arr[i];
    --count[arr[i]];
  }

  // Copy the output array to arr, so that arr now
  // contains sorted characters
  for (i = 0; arr[i]; ++i)
    arr[i] = output[i];
}
```

Time Complexity: O(N + K) where N is the number of elements in input array and K is the range of input.

Auxiliary Space: O(N + K)

The problem with the previous counting sort was that it could not sort the elements if we have negative numbers in the array because there are no negative array indices. So what we can do is, we can find the minimum element and store count of that minimum element at zero index.

Implementation:

```
void countSort(vector <int>& arr)
{
  int max = *max_element(arr.begin(),
arr.end());
  int min = *min_element(arr.begin(),
arr.end());
  int range = max - min + 1;
  vector<int> count(range),
output(arr.size());
  for(int i = 0; i < arr.size(); i++)
    count[arr[i]-min]++;

  for(int i = 1; i < count.size(); i++)
    count[i] += count[i-1];
```

```
for(int i = arr.size()-1; i >= 0; i--)
{
    output[ count[arr[i]-min] -1 ] = arr[i];
    count[arr[i]-min]--;
}
```

```
for(int i=0; i < arr.size(); i++)
    arr[i] = output[i];
}
```

Important Points:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. It's running time complexity is O(n) with space proportional to the range of data.
3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
5. Counting sort can be extended to work for negative inputs also.

Heap Sort-

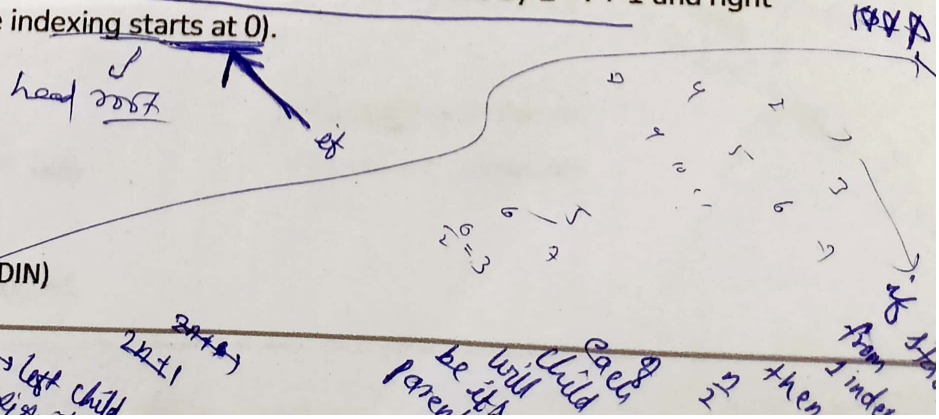
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (SourceWikipedia).

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Array based representation for Binary Heap: Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).



Heap Sort Algorithm for sorting an array in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: [4, 10, 3, 5, 1]
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)

The heapify procedure calls itself recursively to build heap
in top down manner.
```

Implementation:

```
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
int largest = i; // Initialize largest as root
int l = 2*i + 1; // left = 2*i + 1
int r = 2*i + 2; // right = 2*i + 2
// If left child is larger than root
if (l < n && arr[l] > arr[largest])
```

```
largest = l;

// If right child is larger than largest so far
if (r < n && arr[r] > arr[largest])
    largest = r;

// If largest is not root
if (largest != i)
{
    swap(arr[i], arr[largest]);

    // Recursively heapify the affected sub-tree
    heapify(arr, n, largest);
}
}

// Main function for heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Important Notes:

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable (See this).

Time Complexity: Time complexity of heapify is O(N*LogN). Time complexity of createAndBuildHeap() is O(N) and overall time complexity of Heap Sort is **O(N*LogN)** where N is the number of elements in the list or array.

Heap sort algorithm has limited use because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

sort() Function in C++ STL-

C++ STL provides a built-in function sort() that sorts a vector or array (items with random access).

Syntax to sort an Array:

```
sort(arr, arr+n);

Here, arr is the name or base address of the array
and, n is the size of the array.
```

Syntax to sort a Vector:

```
sort(vec.begin(), vec.end());

Here, vec is the name of the vector.
```

Below program illustrate the sort function:

```
// C++ program to demonstrate default behaviour of
// sort() in STL.
#include <bits/stdc++.h>
using namespace std;

int main()
{
  // Sorting Array
  int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
  int n = sizeof(arr)/sizeof(arr[0]);

sort(arr, arr+n);

cout << "Array after sorting is : \n";
for (int i = 0; i < n; ++i)
  cout << arr[i] << " ";

// Sorting Vector
vector<int> vec = {1,2,4,5,3};

sort(vec.begin(), vec.end());

cout << "\nVector after sorting is : \n";
for (int i = 0; i < vec.size(); ++i)
  cout << vec[i] << " ";

return 0;
}
```

Output :

```
Array after sorting is :
0 1 2 3 4 5 6 7 8 9
Vector after sorting is :
1 2 3 4 5
```

So by default, sort() function sorts an array in ascending order.

How to sort in descending order?

The sort() function takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass "greater()" function to sort in descending order. This function does comparison in a way that puts greater element before.

```
// C++ program to demonstrate descending order
// sort using greater<>().
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
  int n = sizeof(arr)/sizeof(arr[0]);
  sort(arr, arr+n, greater<int>());

  cout << "Array after sorting : \n";
  for (int i = 0; i < n; ++i)
    cout << arr[i] << " ";

  return 0;
}
```

Output:

```
Array after sorting :

9 8 7 6 5 4 3 2 1 0
```

How to sort in particular order?

We can also write our own comparator function and pass it as a third parameter.

```
// A C++ program to demonstrate STL sort() using
// our own comparator
#include<bits/stdc++.h>
using namespace std;

// An interval has start time and end time
struct Interval
{
  int start, end;
};

// Compares two intervals according to staring times.
bool compareInterval(Interval i1, Interval i2)
{
```

```
    return (i1.start < i2.start);
}

int main()
{
    Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };
    int n = sizeof(arr)/sizeof(arr[0]);

    // sort the intervals in increasing order of
    // start time
    sort(arr, arr+n, compareInterval);

    cout << "Intervals sorted by start time : \n";
    for (int i=0; i<n; i++)
        cout << "[" << arr[i].start << "," << arr[i].end
             << "] ";

    return 0;
}
```

Output:

```
Intervals sorted by start time :

[1,9] [2,4] [4,7] [6,8]
```

Sorting using Built-in methods in Java- Arrays.sort()

The Arrays.sort() is a built-in method in Java of Arrays class which is used to sort an array in ascending or descending or any other order specified by the user.

Syntax:

```
public static void sort(int[] arr, int from_Index, int to_Index)

arr - The array to be sorted.
from_Index - The index of the first element, inclusive, to be sorted.
to_Index - The index of the last element, exclusive, to be sorted.
```

Below are different ways of using the sort() method of Arrays class in Java to sort arrays differently.

```
// A sample Java program to sort an array of integers
// using Arrays.sort(). It by default sorts in
// ascending order
import java.util.Arrays;

public class SortExample
{
  public static void main(String[] args)
  {
    // Our arr contains 8 elements
    int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

    Arrays.sort(arr);

    System.out.printf("Modified arr[] : %s",
                      Arrays.toString(arr));
  }
}
```

- **Output:**

```
Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]
```

We can also use sort() to sort a subarray of arr[]

```
// A sample Java program to sort a subarray
// using Arrays.sort().
import java.util.Arrays;
public class SortExample
{
  public static void main(String[] args)
  {
    // Our arr contains 8 elements
    int[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

    // Sort subarray from index 1 to 4, i.e.,
    // only sort subarray {7, 6, 45, 21} and
    // keep other elements as it is.
    Arrays.sort(arr, 1, 5);

    System.out.printf("Modified arr[] : %s",
                      Arrays.toString(arr));
  }
}
```

- **Output:**

```
Modified arr[] : [13, 6, 7, 21, 45, 9, 2, 100]
```

We can also sort in descending order

```
// A sample Java program to sort a subarray
// in descending order using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
  public static void main(String[] args)
```

```
{
  // Note that we have Integer here instead of
  // int[] as Collections.reverseOrder doesn't
  // work for primitive types.
  Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

  // Sorts arr[] in descending order
  Arrays.sort(arr, Collections.reverseOrder());

  System.out.printf("Modified arr[] : %s",
                    Arrays.toString(arr));
  }
}
```

- **Output:**

```
Modified arr[] : [100, 45, 21, 13, 9, 7, 6, 2]
```

We can also sort strings in alphabetical order

```
// A sample Java program to sort an array of strings
// in ascending and descending orders using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
  public static void main(String[] args)
  {
    String arr[] = {"practice.geeksforgeeks.org",
                    "quiz.geeksforgeeks.org",
                    "code.geeksforgeeks.org"
                   };

    // Sorts arr[] in ascending order
    Arrays.sort(arr);
    System.out.printf("Modified arr[] : \n%s\n\n",
                      Arrays.toString(arr));

    // Sorts arr[] in descending order
    Arrays.sort(arr, Collections.reverseOrder());

    System.out.printf("Modified arr[] : \n%s\n\n",
                      Arrays.toString(arr));
  }
}
```

- **Output:**

```
Modified arr[] :
[code 1="practice.geeksforgeeks.org," 2="quiz.geeksforgeeks.org"
language=".geeksforgeeks.org,"][/code]

Modified arr[] :
[quiz.geeksforgeeks.org, practice.geeksforgeeks.org, code.geeksfo
rgeeks.org]
```

We can also sort an array according to user defined criteria: We use Comparator interface for this purpose. Below is an example.

```
// Java program to demonstrate working of Comparator
// interface
import java.util.*;

// A class to represent a student.
class Point
{
  int x, y;
  Point(int i, int j) {x = i; y = j;}
}

class MySort implements Comparator<Point>
{
  // Used for sorting in ascending order of
  // roll number
  public int compare(Point a, Point b)
  {
    return a.x - b.x;
  }
}

// Driver class
class Main
{
  public static void main (String[] args)
  {
    Point [] arr = {new Point(10, 20), new Point(3, 12),
new Point(5, 7)};
    Arrays.sort(arr, new MySort());
    for (int i=0; i<arr.length; i++)
      System.out.println(arr[i].x + " " + arr[i].y);
  }
}
```

- **Output:**

```
3 12
5 7
10 20
```

Collections.sort()

The **Collections.sort()** method is present in Collections class. It is used to sort the elements present in the specified list of Collection in ascending order.
It works similar to the Arrays.sort() method but it is better as it can sort the elements of Array as well as any collection interfaces like a linked list, queue and many more.
Syntax:

```
public static void sort(List myList)

myList : A List type object we want to sort.

This method doesn't return anything
```

Example:

```
Let us suppose that our list contains
{"Geeks For Geeks", "Friends", "Dear", "Is", "Superb"}

After using Collection.sort(), we obtain a sorted list as
{"Dear", "Friends", "Geeks For Geeks", "Is", "Superb"}
```

Below are some ways of using the Collections.sort() method in Java:

Sorting an ArrayList in ascending order

```
// Java program to demonstrate working of
Collections.sort()

import java.util.*;


public class Collectionsorting
{
  public static void main(String[] args)
  {
    // Create a list of strings
    ArrayList<String> al = new ArrayList<String>();
    al.add("Geeks For Geeks");
    al.add("Friends");
    al.add("Dear");
    al.add("Is");
    al.add("Superb");

    /* Collections.sort method is sorting the
    elements of ArrayList in ascending order. */
    Collections.sort(al);

    // Let us print the sorted list
    System.out.println("List after the use of" +
                       " Collection.sort() :\n" + al);
  }
}
```

- **Output:**

```
List after the use of Collection.sort() :

[Dear, Friends, Geeks For Geeks, Is, Superb]
```

```
// Java program to demonstrate working of
Collections.sort()

// to descending order.

import java.util.*;

public class Collectionsorting
{
  public static void main(String[] args)
  {
    // Create a list of strings
    ArrayList<String> al = new ArrayList<String>();
    al.add("Geeks For Geeks");
    al.add("Friends");
    al.add("Dear");
    al.add("Is");
    al.add("Superb");
```

```
/* Collections.sort method is sorting the
elements of ArrayList in ascending order. */
Collections.sort(al, Collections.reverseOrder());

// Let us print the sorted list
System.out.println("List after the use of" +
                   " Collection.sort() :\n" + al);
  }
}
```

- **Output:**

```
List after the use of Collection.sort() :

[Superb, Is, Geeks For Geeks, Friends, Dear]
```

Sorting an ArrayList according to user defined criteria: We can use Comparator Interface for this purpose

```
// Java program to demonstrate working of Comparator
// interface and Collections.sort() to sort according
// to user defined criteria.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student
{
  int rollno;
  String name, address;

  // Constructor
  public Student(int rollno, String name,
                 String address)
  {
    this.rollno = rollno;
    this.name = name;
    this.address = address;
  }

// Used to print student details in main()
  public String toString()
  {
    return this.rollno + " " + this.name +
                 " " + this.address;
  }
}
class Sortbyroll implements Comparator<Student>
{
  // Used for sorting in ascending order of
  // roll number
  public int compare(Student a, Student b)
  {
    return a.rollno - b.rollno;
  }
}
// Driver class
class Main
{
  public static void main (String[] args)
  {
    ArrayList<Student> ar = new ArrayList<Student>();
    ar.add(new Student(111, "bbbb", "london"));
    ar.add(new Student(131, "aaaa", "nyc"));
```

```
ar.add(new Student(121, "cccc", "jaipur"));

System.out.println("Unsorted");
for (int i=0; i<ar.size(); i++)
  System.out.println(ar.get(i));

Collections.sort(ar, new Sortbyroll());

System.out.println("\nSorted by rollno");
for (int i=0; i<ar.size(); i++)
  System.out.println(ar.get(i));
}}
```

Output :

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc
```

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

CREDITS- INTERNET

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.

Stack

Stack &
Queue Implementation is not
required. Just put its images.
Even coding is not required.
But sight on your own way
Coding, just no need of explanation

Data Structure and Algo In Java :-

In IntellijIdea IDE

1) Stack :-

⇒ We make a Mystack class as our own stack :-

In Main.java

```
package.com.company;
public class Main {
    public static void main(String[] args) {
        MyStack theStack = new MyStack(s:3); //size is passed here only.
        theStack.push(j:10);
        theStack.push(20);
        theStack.display();
        theStack.push(30);
        theStack.pop();
        theStack.push(40);
        theStack.display();
    }
}
```

output

```
the content of the stack :- 10 20
the pop item is 30
the content of the stack :- 10 20 40
```

{"syso" → short for System.out.println}

(2) In MyStack.java :-

```
package com.company
public class MyStack {

        private int maxSize;
        private int[] stackArray;
        private int top;

    //"size" as well as "top" is being initialized in constructor only
    public MyStack(int s){
        maxSize = s;
        stackArray = new int[maxSize];
        top = -1;
    }

    //Remember in this the top is changed everywhere within the methods
    //with stackArray. since stackArray is non-primitive type data so
    //the changes is kept. and its permanent.

    public int peek() {
        return stackArray[top];    //Returns the top element of the stack.
    }

    public boolean isEmpty() {
        return (top == -1);
    }

    public boolean isFull() {
        return (top == maxSize-1);
    }
}
```

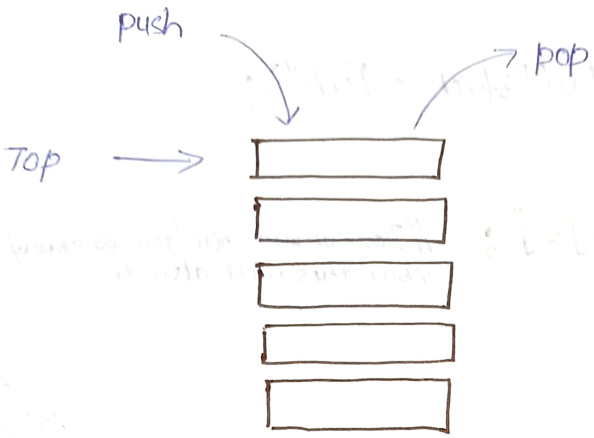
```
public void push(int j) {
    if (isFull()) {
        System.out.println("Stack is Full");
    }
    else {
        stackArray[++top]=j;   //You can write here "this.stackArray" also
    }                          //but this will also do.
}

public void pop() {
    if (isEmpty()) {
        System.out.println("Stack is Empty");
    }
    else {
        Syso("the pop item is " + stackArray[top--]);   //same comment as above.
    }
}

public void display() {
    Syso("The content of the stack is :-");
    if (this.maxSize == -1) {
        Syso("stack is empty");
    }
    else {
        for(int i=0; i<=top; i++) {
            Syso(stackArray[i]+" ");   //with no ln in print"
        }
        Syso(" ");
    }
}

}
```

Static → you actually don't need to write 'this' here in it's method because we know it's already a (static) method of (myStack) we know which class this all belong to.

only here we need to use 'this' because we've used maxSize in constructor so while making object for each object this maxSize will be different so we are specifically targetting that object by using 'this'.

(not exactly required)

Stack Data Structure



Last in first out (LIFO)
or
First in Last out (FILO)

⇒ Insertion and Deletion happen on same end.

⇒ Can be implement using two ways:-
1. Using Arrays (Array in class only)
2. Using LinkedList. (Instead of Array here we use linked list to store data)

Qweve

#2 Queue :- using class Queue

1. In Queue.java → just translate it from below C++ code

~~package com.company;~~

~~public class Queue~~

```
using namespace std;

int queue[100], n=100, front=-1, rear=-1

void enqueue(int val) {
    if (rear == n-1) {   // corner case
        cout<<"Queue Overflow"<<endl;   //(Means the size of Queue (Array) is all used up)
    } else {
        if (front==-1) {
            front=0;
        }
        rear++;
        queue[rear]=val;
    }
}

void dequeue() {
    if (front==-1 || front>rear) {   // corner case
        cout<<"Queue Underflow";   //(Means no element present in queue)
        return;
    } else {
        cout<<"Element deleted from queue is: "<<queue[front]<<endl;
        front++;
    }
}
```

```
void display() {
    if (front == -1) {
        cout << "Queue is empty" << endl;
    } else {
        cout << "Queue elements are: ";
        for (int i=front ; i <= rear ; i++) {
            cout << queue[i] << " ";
        }
        cout << endl;
    }
}

int main() {
    enqueue(5); display();
    enqueue(6); display();
    enqueue(2); display();
    dequeue(); display();
    return 0;
}
```

output

```
                    F,R
Queue elements are: 5
                    F   R
                 :  5   6
                    F       R
                 :  5   6   2
```

Element deleted from queue is : 5

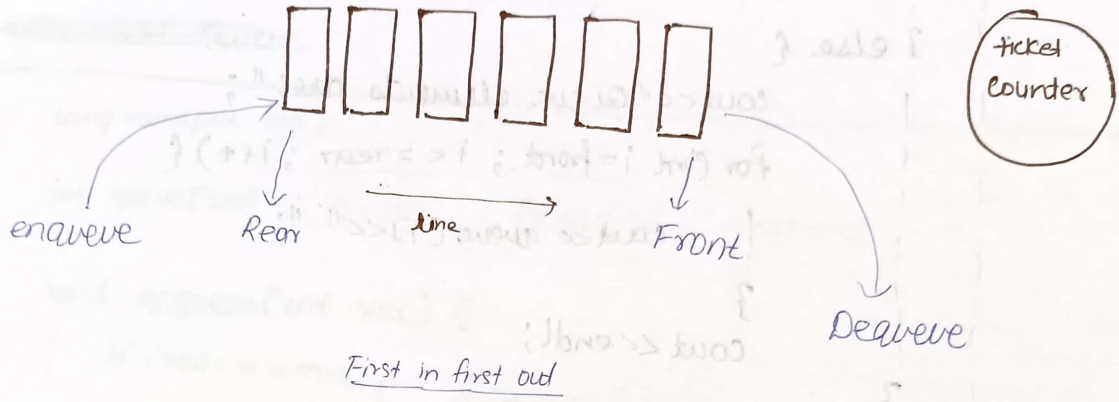
Queue elements are : 6 2 → Rear.

↑ front

F = queue[front]
R = queue[rear]

OK now while understood

Operations on queue



ticket counter

First in first out

⇒ Insertion and deletion happens on different ends.

↓ at Real

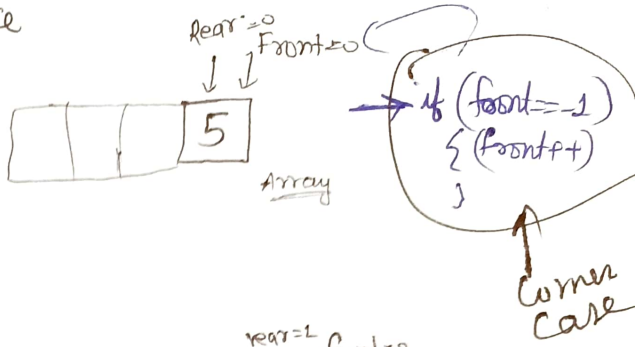
↓ at Front

⇒ can be i

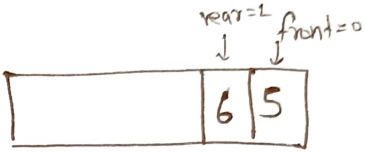
Initially front = -1, rear = -1

Also you might be thinking like

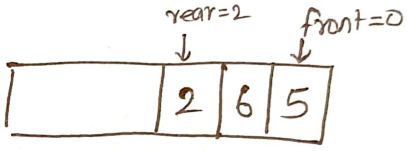
enqueue(5)



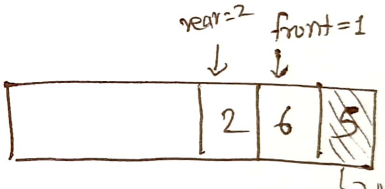
enqueue(6)



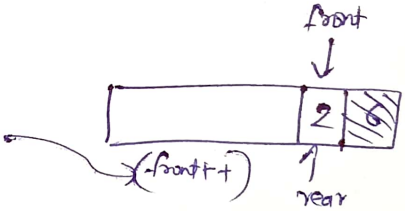
enqueue (2)



dequeue ()



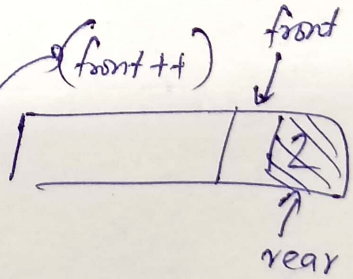
dequeue ()



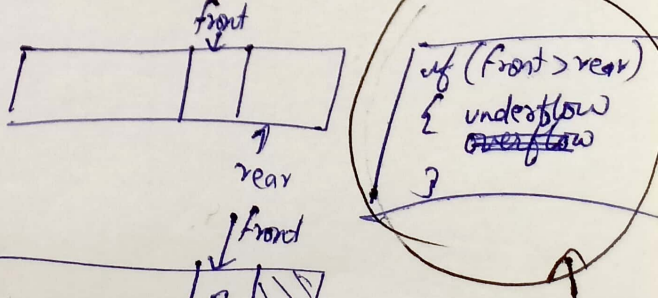
If you see this and start writing code you will cover all the corner cases also.

OK all understood

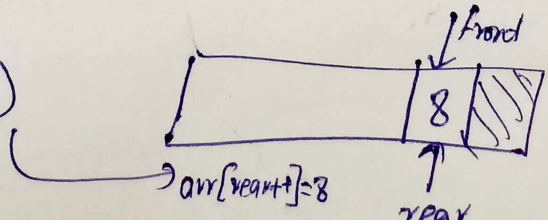
dequeue ()



dequeue()



enqueue(8)



Miscellaneous Stack Queues Questions

Implement Stack using 2 Queues

By Making Pop Method expensive

S20

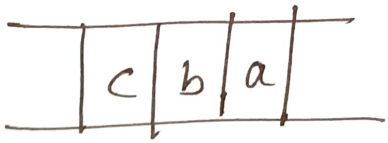
- Push(s, x) process
  - ↳ enqueue x to q_1

- Pop(s) process
  1. ↳ One by one dequeue everything except the last element from q_1 and enqueue to q_2.
  2. ↳ Dequeue the last element of q_1, the dequeue item is result, store it.
  3. ↳ Swap the names of q_1 and q_2.
  4. ↳ Return the item stored in step 2.

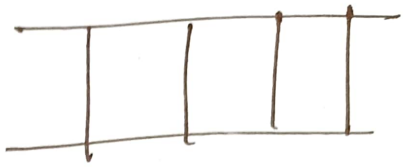
---

example

① Push (a)
② Push (b)
③ Push (c)



q_1



q_2

---

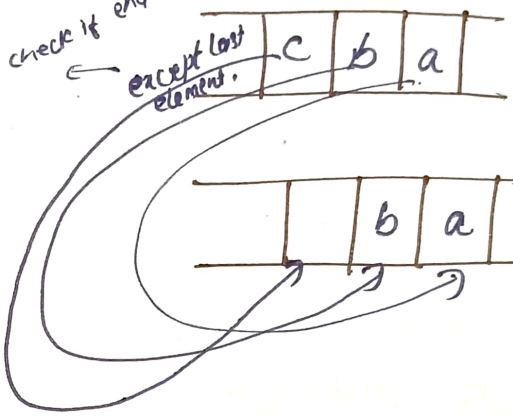
④ PoP()

So Pop

check if enque results empty queue then stop so i.e. if (rear == front) → last element

except last element.

Step-1

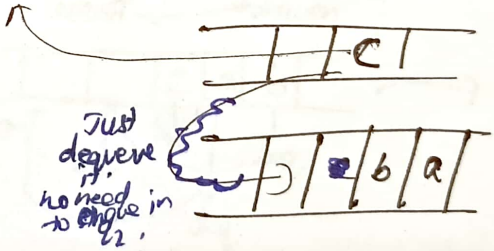


q_1

q_2

Step-2 → dequeue the last element but also store it.

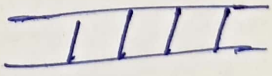
∴ int result = dequeue(q_1)



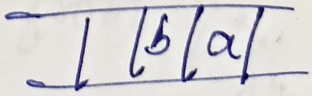
q_1

q_2

step-3 :- swap names :-



→ q_2



→ q_1

Implement stack using single queue :-

⇒ This solution assumes that we can find size of queue at any point.

⇒ The idea is to keep newly inserted element always at rear of queue, keeping order of previous elements same.

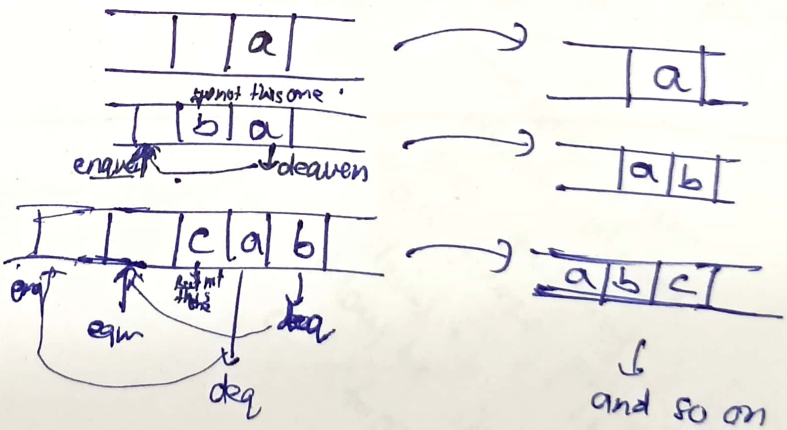
so it's easy pop & hard push

Push (s, x) :- Let size of q be s.

① Enqueue x to q

② One by one Dequeue (s) items (all already present) from queue and enque them.

push(a)
push(b)
push(c)



Pop (s) :-

② Dequeue an item from q.

Implement Queue using Single stack :-

(→ 2 stack ret Costs like.)

1) Enqueue :- fun enqueue(x)

⇒ straight away push the element into the stack.

2) Dequeue() :-

↳ Pop all the elements from Main stack recursively until stack size is equal to 1.

↳ If stack size = 1, pop item from stack, and return the same item

↳ Push all popped elements back to stack.

(→ back from where, where are you storing all those popped items, it's way? May be its stacks. & stacks)