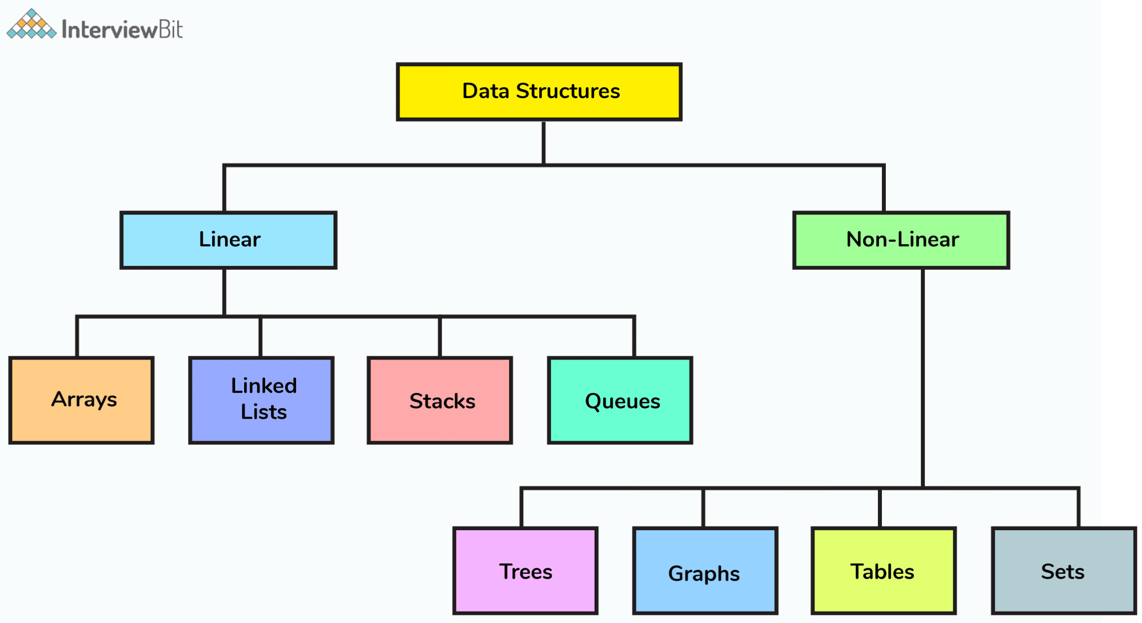
What is Data Structure?

- 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.



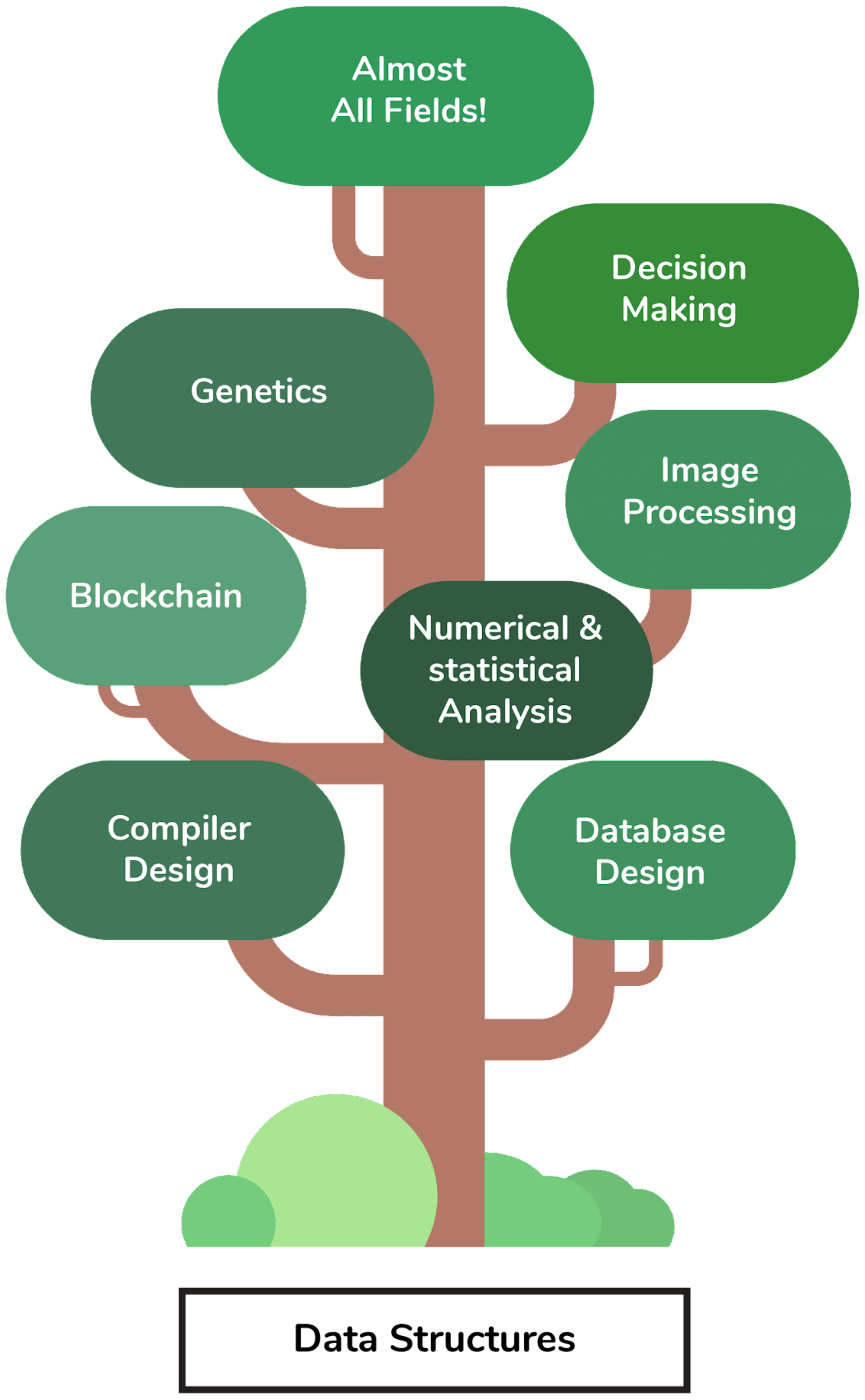
-

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 database implementation.
- Some of the most important areas where data structures 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



InterviewBit

Benefits of Learning Data Structures

Any given problem has constraints on how fast the problem should be solved (time) and how much less resources the problem consumes(space). That is, a problem is constrained by the space and time complexity within which it has to be solved efficiently.

- In order to do this, it is very much essential for the given problem to be represented in a proper structured format upon which efficient algorithms could be applied.
- Selection of proper data structure becomes the most important step before applying algorithms to any problem.
  Having knowledge of different kinds of data structures available helps the programmer in choosing which data structure suits the best for solving a problem efficiently. It is not just important to make a problem work, it is important how efficiently you make it work.

Data structures in C, Java

- The core concepts of data structures remain the same across all the programming languages. Only the implementation differs based on the syntax or the structure of the programming language.
  - The implementation in procedural languages like C is done with the help of structures, pointers, etc.
  - In an object oriented language like Java, data structures are implemented by using classes and objects.
- Having sound knowledge of the concepts of each and every data structure helps you to stand apart in any interviews as selecting the right data structure is the first step towards solving problems efficiently.

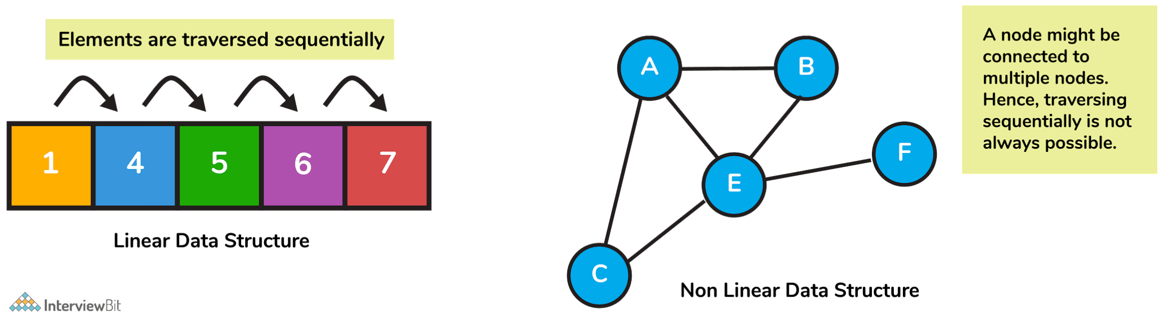
Interview Questions

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 examples of non-linear data structures.

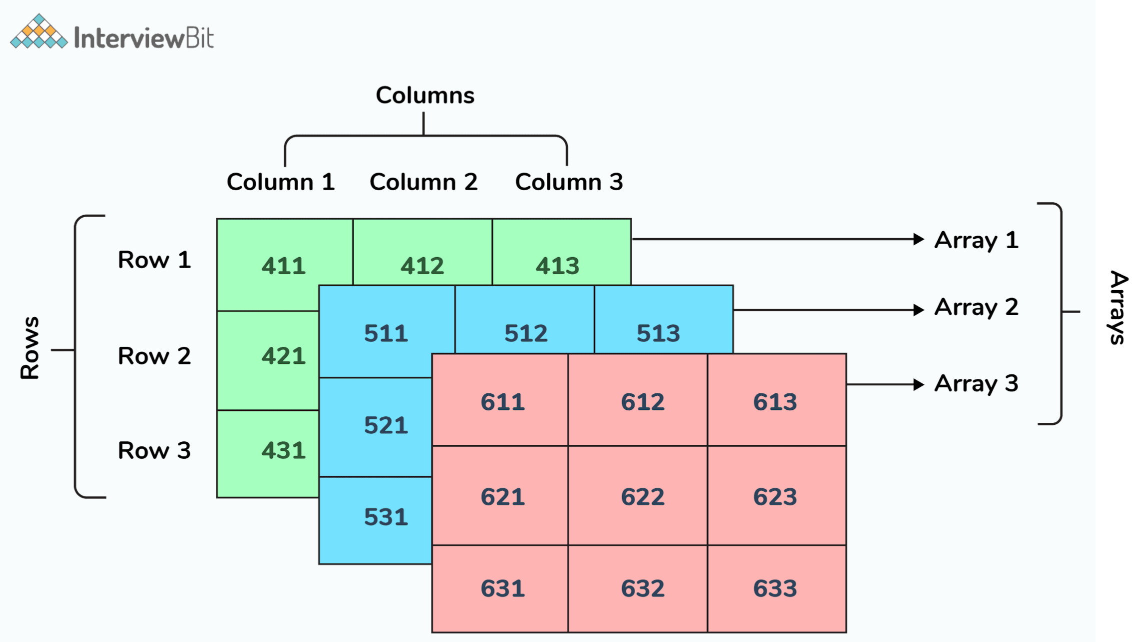

Elements are traversed sequentially
1
4
5
6
7
Linear Data Structure
InterviewBit
A
B
F
E
C
Non Linear Data Structure
A node might be connected to multiple nodes. Hence, traversing sequentially is not always possible.

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.

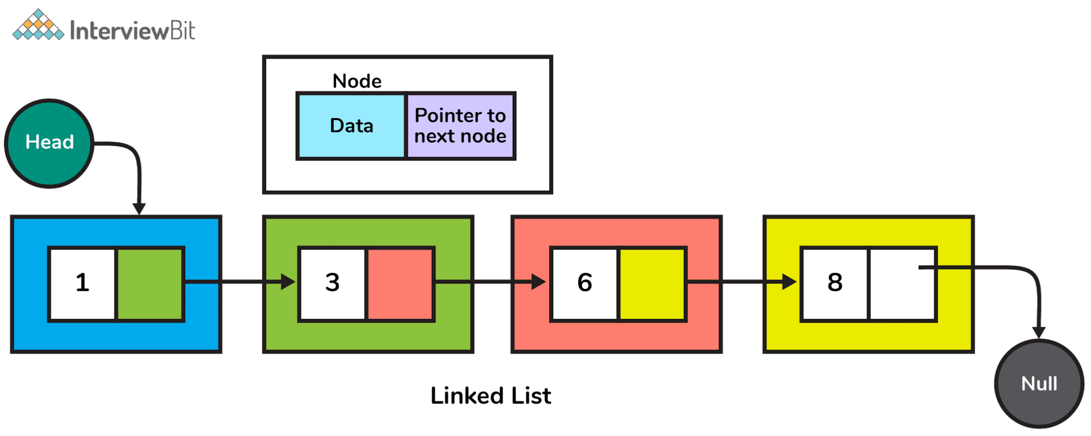


-

5. What is a linked list?

A linked list is a data structure that has a sequence **of nodes** where every node is connected to the next node by means of a reference pointer. The elements are **not stored in adjacent** memory locations. They are linked using pointers to form a chain. This forms a chain-like link for data storage.

- Each node element has two parts:
    - a data field
    - a reference (or pointer) to the next node.
- The first node in a linked list is called the head and the last node in the list has the pointer to NULL. Null in the reference field indicates that the node is the last node. When the list is empty, the head is a null reference.



6. Are linked lists of linear or non-linear types?

Linked lists can be considered both linear and non-linear data structures. This depends upon the application that they are used for.

- When a linked list is used for access strategies, it is considered as a linear data-structure. When they are used for data storage, it can be considered as a non-linear data structure.

7. How are linked lists more efficient than arrays?

1. **Insertion and Deletion**
    - Insertion and deletion process is expensive in an array as the room has to be created for the new elements and existing elements must be shifted.
    - But in a linked list, the same operation is an easier process, as we only update the address present in the next pointer of a node.
2. **Dynamic Data Structure**
    - Linked list is a dynamic data structure that means there is no need to give an initial size at the time of creation as it can grow and shrink at runtime by allocating and deallocating memory.
    - Whereas, the size of an array is limited as the number of items is statically stored in the main memory.
3. **No wastage of memory**
    - As the size of a linked list can grow or shrink based on the needs of the program, there is no memory wasted because it is allocated in runtime.
    - In arrays, if we declare an array of size 10 and store only 3 elements in it, then the space for 3 elements is wasted. Hence, chances of memory wastage is more in arrays.

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

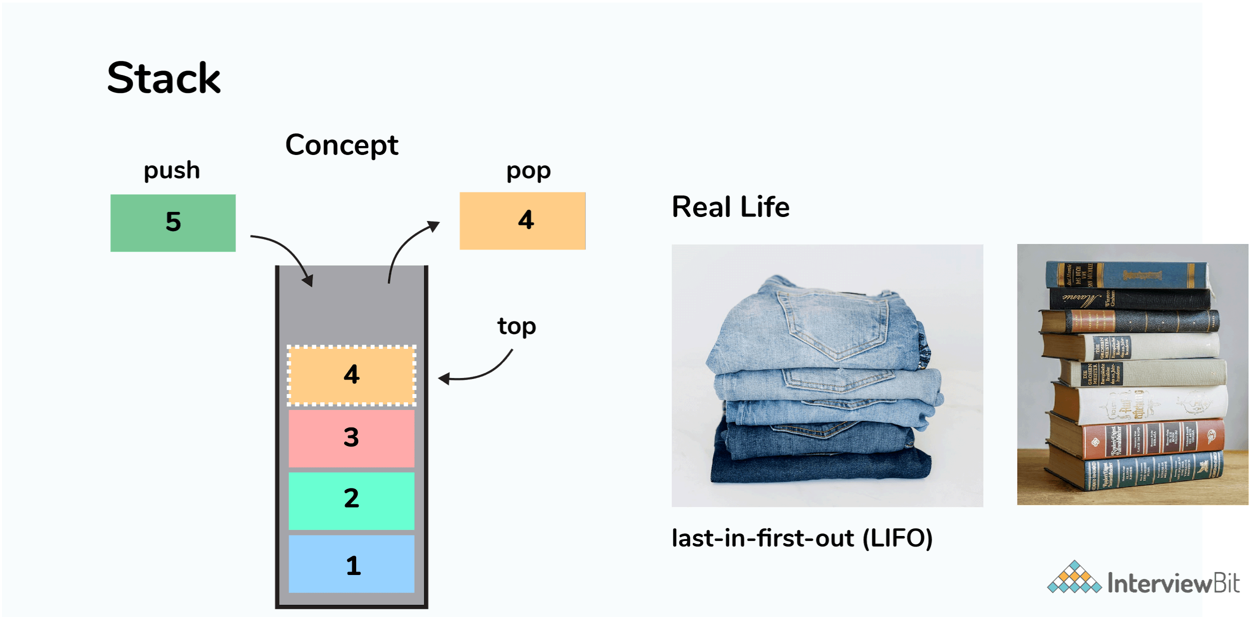
- Following are the scenarios where we use linked list over array:
    - When we do not know the exact number of elements beforehand.
    - When we know that there would be a large number of add or remove operations.
    - Less number of random access operations.
    - When we want to insert items anywhere in the middle of the list, such as when implementing a priority queue, a 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 lists, 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:
    1. One that connects to the next node in the sequence
    2. 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:
    1. A music playlist with next song and previous song navigation options.
    2. The browser cache with BACK-FORWARD visited pages
    3. 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?

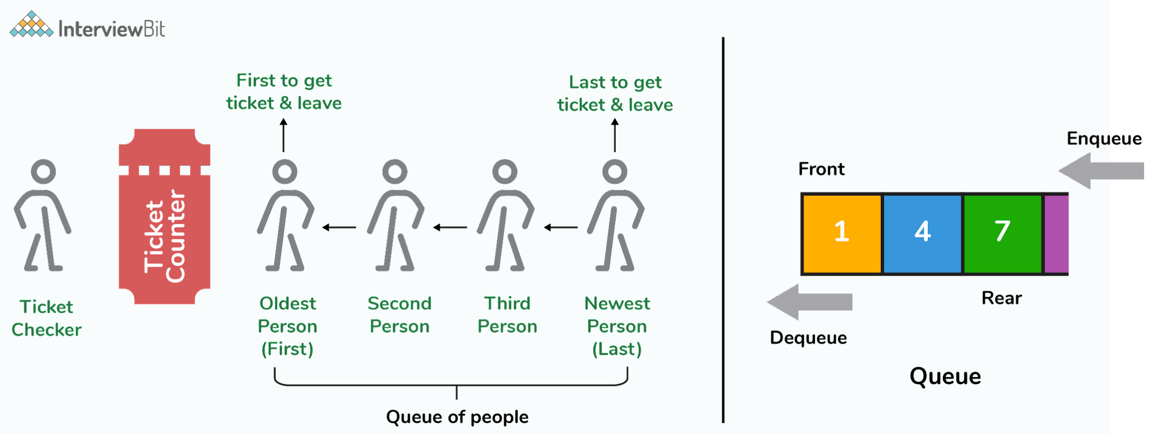
- Stack is a linear data structure that follows the LIFO (Last In First Out) approach for accessing elements.
- Push, pop, and top (or peek) are the basic operations of a stack.



-
  Following are some of the applications of a stack:
    1. Check for balanceid parentheses in an expression
    2. Evaluation of a postfix expression
    3. Problem of Infix to postfix conversion
    4. 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.

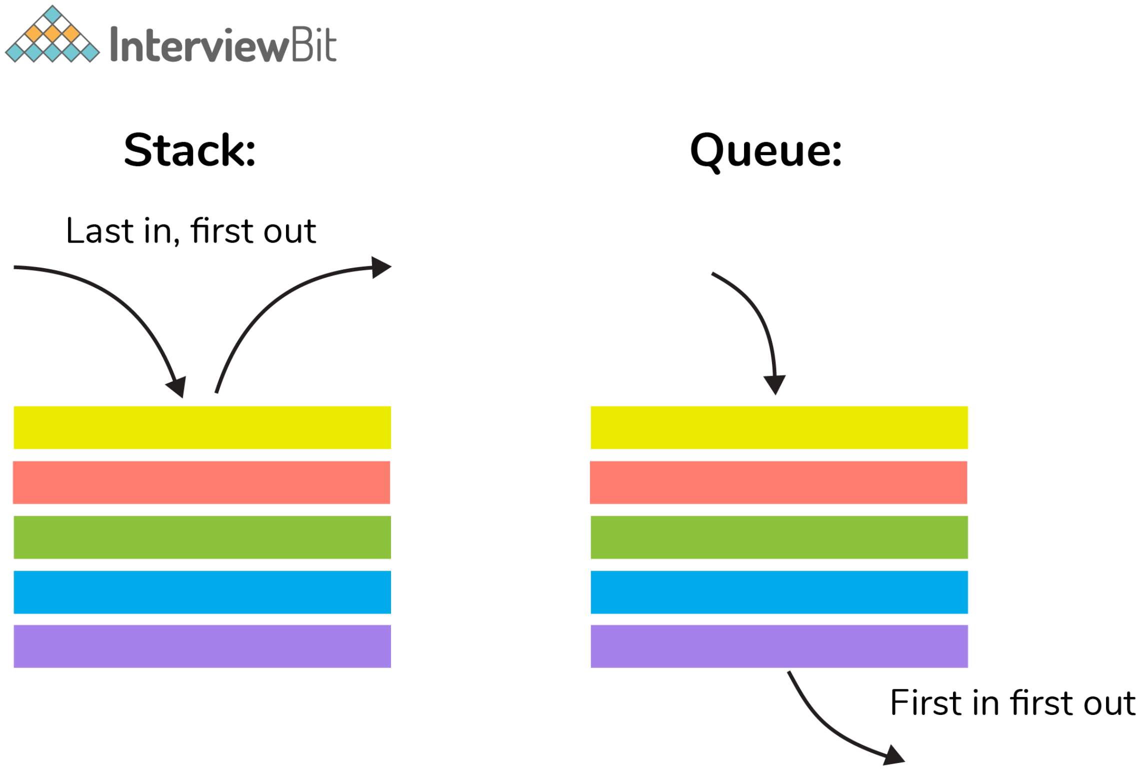


-
  Some of the applications of queue are:

1. CPU Task scheduling
2. BFS algorithm to find shortest distance between two nodes in a graph.
3. Website request processing
4. Used as buffers in applications like MP3 media player, CD player, etc.
5. 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 the 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 and `stack1` 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 auxiliary space complexity of O(n)):
    1. **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 an element at the 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
```

2. **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 a 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:
    1. **By making push operation costly:**
        - This method ensures that the newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’.
        - ‘q2’ is used as auxiliary 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.
```

■

2. **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.

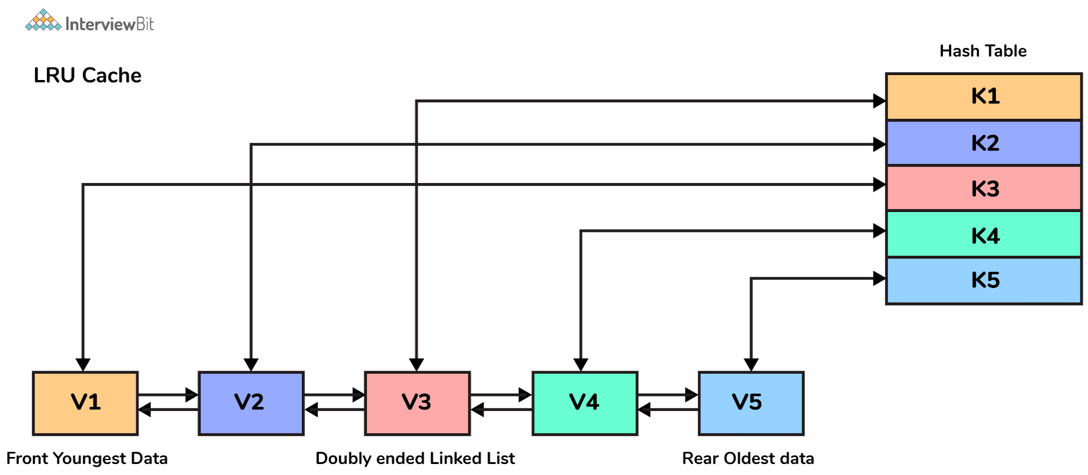
```
pop(s):
    Step1: Dequeue every element except the last element from q1 and enqueue to
q2.
    Step2: Dequeue the last item of q1, the dequeued item is stored in the
result variable.
    Step3: Swap the names of q1 and q2 (for getting updated data after dequeue)
    Step4: Return the result.
```

■

- **16. What is a hashmap in data structure?**
    1. 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?**
    1. The key or value object that gets used in hashmap must implement `equals()` and `hashcode()` methods.
    2. The hash code is used when inserting the key object into the map and the equals method is used when trying to retrieve a value from the map.
- **18. How does HashMap handle collisions in Java?**
    1. The `java.util.HashMap` class in Java uses the approach of chaining to handle collisions. In chaining, if the new values with the same key are attempted to be pushed, then these values are stored in a linked list stored in a bucket of the key as a chain along with the existing value.
    2. In the worst case scenario, it can happen that all keys 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 algorithms.
- **19. What is the time complexity of basic operations get() and put() in HashMap class?**
    1. The time complexity is O(1) **assuming** that the hash function used in the hash map distributes elements uniformly among the buckets.
- **20. Which data structures are used for implementing LRU cache?**
    1. 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.



2.

21. What is a priority queue?

1. A priority queue is an abstract data type that is like a normal queue but has priority assigned to elements.
2. Elements with higher priority are processed before the elements with a lower priority.
3. In order to implement this, a minimum of two queues are required - one for the data and the other to store the priority.

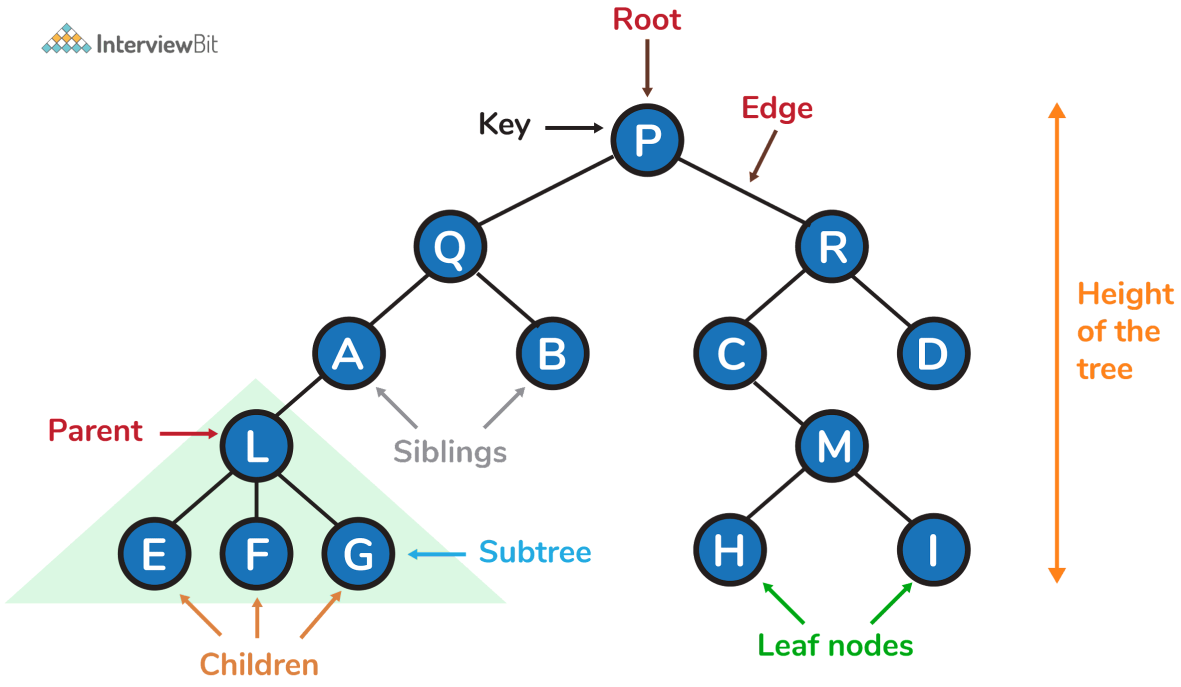
22. Can we store a duplicate key in HashMap?

1. **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?

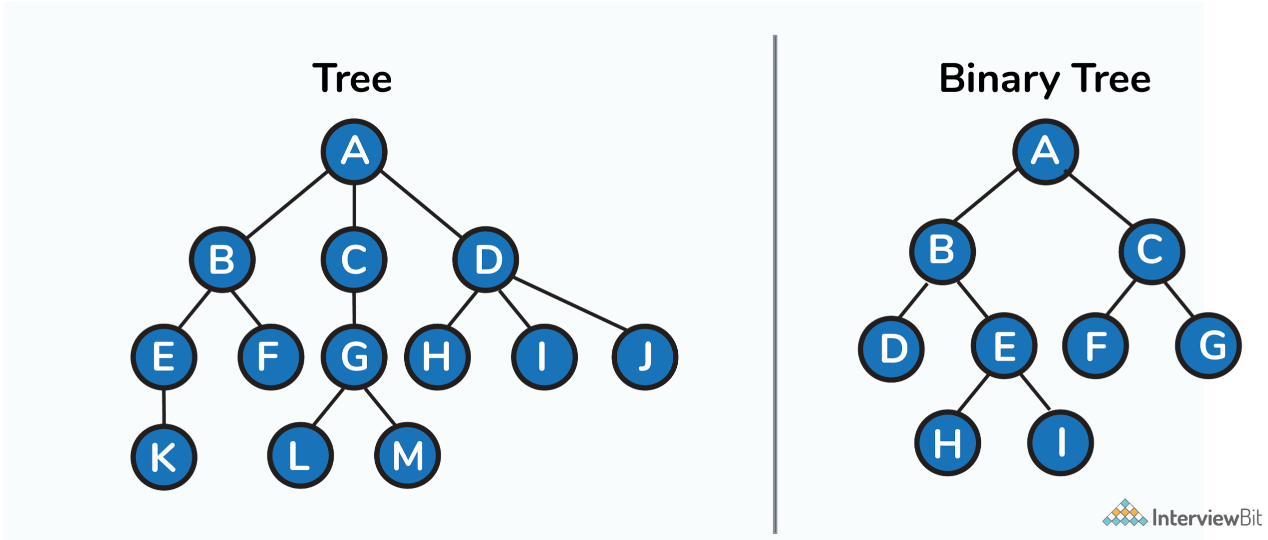
1. Tree is a recursive, non-linear data structure consisting of the set of one or more data nodes where one node is designated as the root and the remaining nodes are called as the children of the root.
2. Tree organizes data in a hierarchical manner.
3. The most commonly used tree data structure is a binary tree and its variants.
4. Some of the applications of trees are:
    - **Filesystems** —files inside folders that are in turn 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 represent the family hierarchy.



5.

- **24. What are Binary trees?**
  1. 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 subtree and right subtree.



- **25. What is the maximum number of nodes in a binary tree of height k?**
  1. The maximum nodes are : $2^{k+1}-1$ 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;
  }
```

1.

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;
          }
      }
```

2.

27. Write Java code to count the 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 += count(root.left);
        count += count(root.right);
        return count;
    }
}
```

-

  ## 28. What are tree traversals?

  1. 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 the 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.

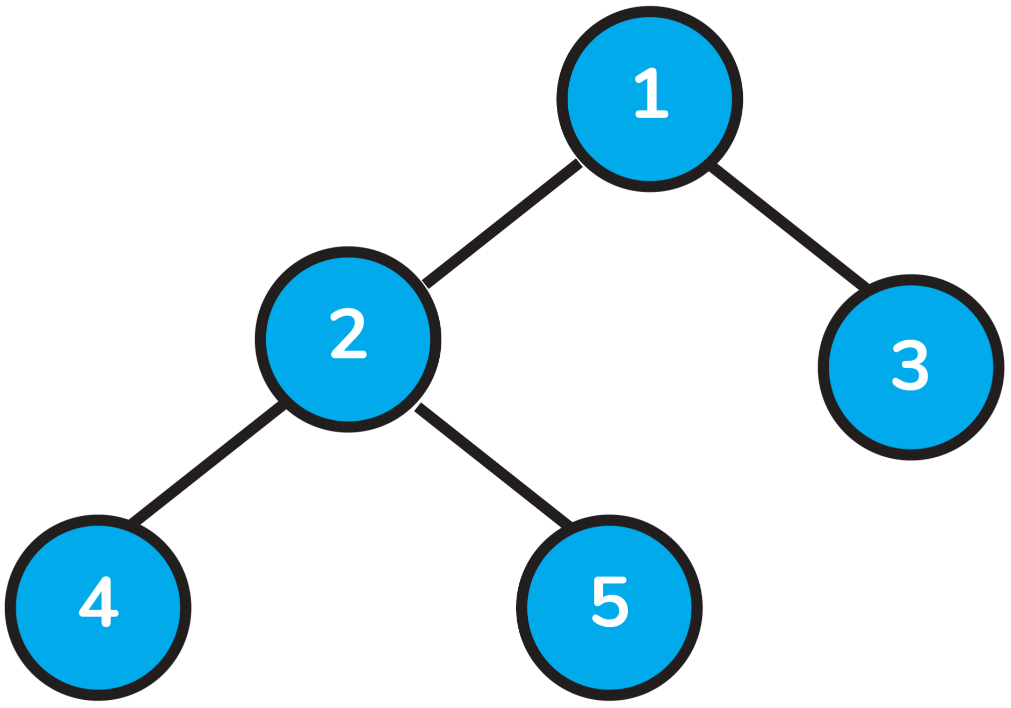
Postorder traversal in Java:

```
// 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.

2. Consider the following tree as an example, then:
3.

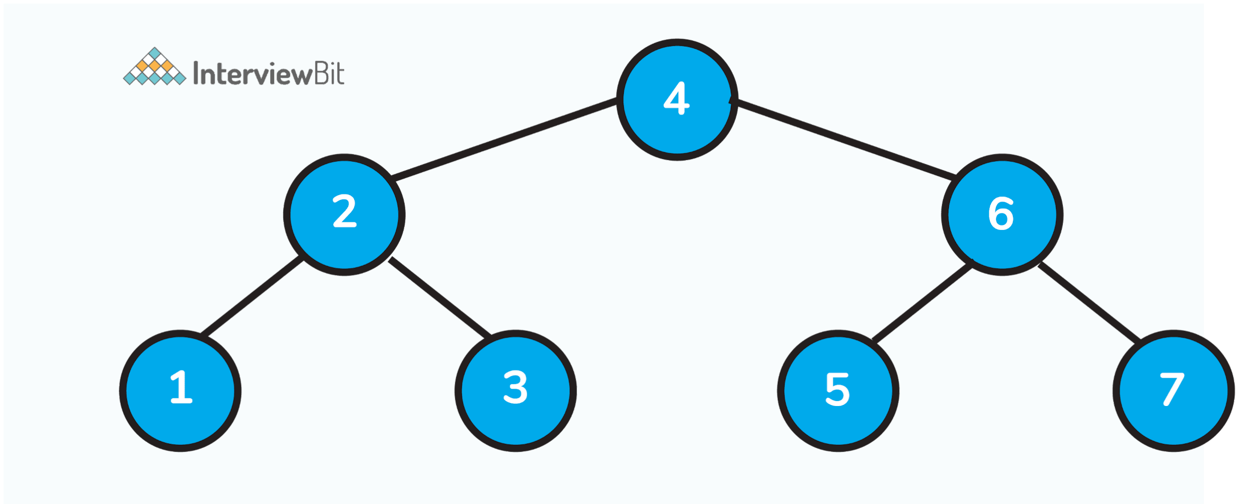


- Inorder Traversal => Left, Root, Right : [4, 2, 5, 1, 3]
- Preorder Traversal => Root, Left, Right : [1, 2, 4, 5, 3]
- Postorder Traversal => Left, Right, Root : [4, 5, 2, 3, 1]

- **29. What is a Binary Search Tree?**

1. 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

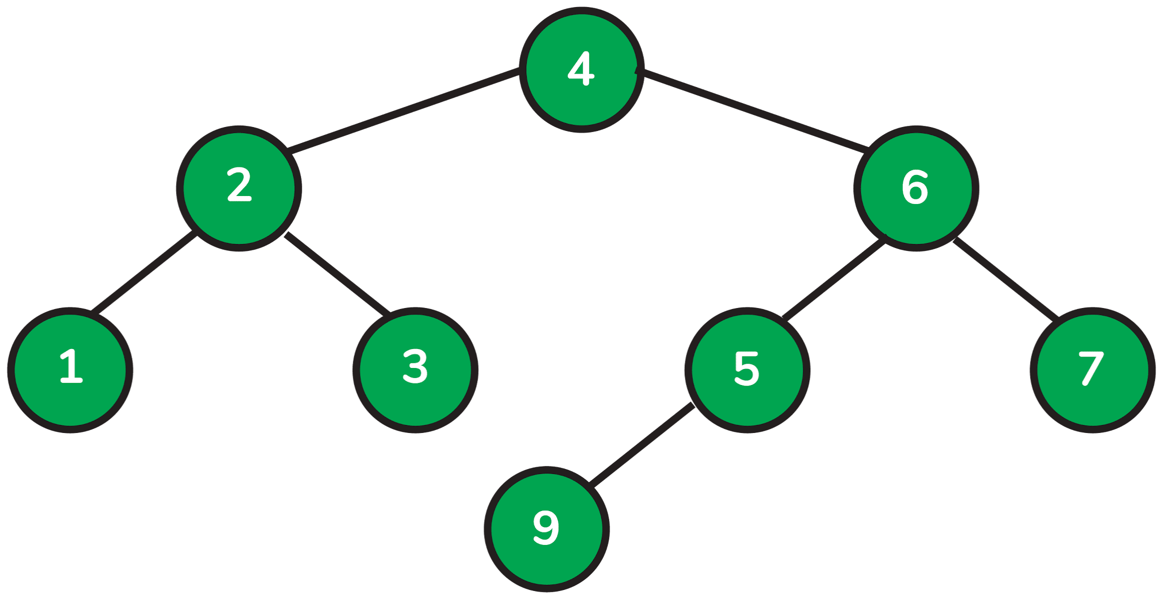
subtree are less than the value of the root node, and the values of the nodes on the right of the root node are correspondingly higher than the root.
2. Also, individually the left and right subtrees are their own binary search trees at all instances of time.



Example of Binary Search Tree

-
30. What is an AVL Tree?
1. AVL trees are **height balancing** BST. AVL tree checks the height of left and right subtrees and assures that the difference is **not more than 1**. This difference is called Balance Factor and is calculated as. `BalanceFactor = height(left subtree) – height(right subtree)`

- ## 31. Print Left view of any binary trees.
1. The main idea to solve this problem is to traverse the tree in preorder 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.
2. At the end of traversal, we can get the solution by just traversing the map.
3. Consider the following tree as example for finding the left view:



Left View of this tree: 4,2,1,9

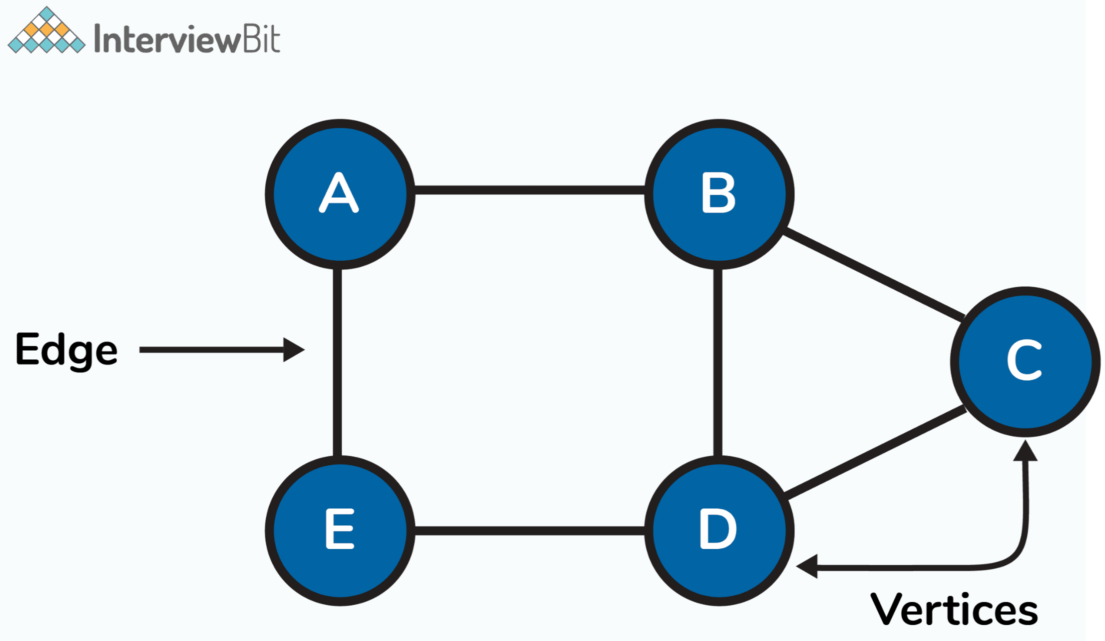
Left view of a binary tree in Java:

```
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);
    }
}
```

4.

- **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?

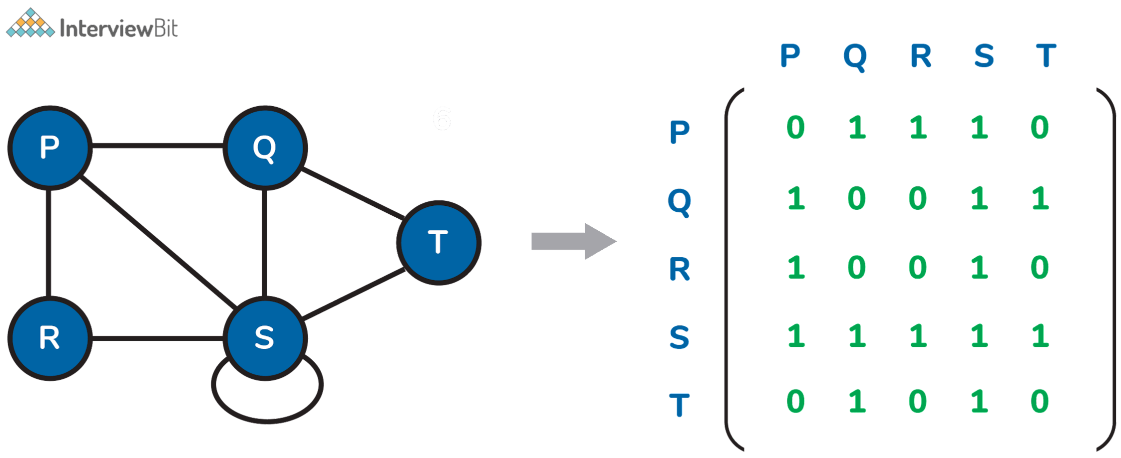
Graphs are used in wide varieties of applications. Some of them are as follows:

1. Social network graphs to determine the flow of information in social networking websites like facebook, linkedin etc.
2. Neural networks graphs where nodes represent neurons and edge represent the synapses between them
3. Transport grids where stations are the nodes and routes are the edges of the graph.
4. Power or water utility graphs where vertices are connection points and edge the wires or pipes connecting them.
5. Shortest distance between two endpoints algorithms.

- ### 34. How do you represent a graph?

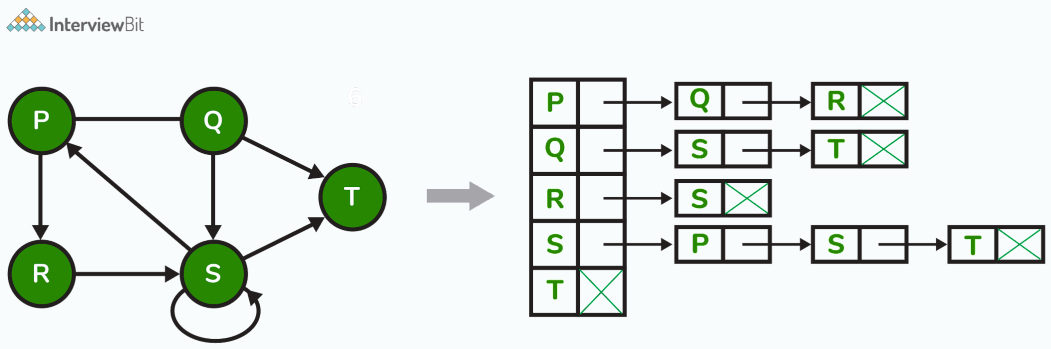
  We can represent a graph in 2 ways:

  1. Adjacency matrix: Used for sequential data representation



$$\begin{array}{c|ccccc} & P & Q & R & S & T \\ \hline P & 0 & 1 & 1 & 1 & 0 \\ Q & 1 & 0 & 0 & 1 & 1 \\ R & 1 & 0 & 0 & 1 & 0 \\ S & 1 & 1 & 1 & 1 & 1 \\ T & 0 & 1 & 0 & 1 & 0 \end{array}$$

2. Adjacency list: Used to represent linked data



- **35. What is the difference between tree and graph data structure?**
  1. 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.
  2. 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)?**
  1. 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.
  2. 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.
  3. Furthermore, BFS uses queue data structure for storing the nodes whereas DFS uses the stack for traversal of the nodes for implementation.

4. DFS yields deeper solutions that are not optimal, but it works well when the solution is dense whereas the solutions of BFS are optimal.
5. 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?
  1. 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?
  1. Topological sorting is a linear ordering of vertices such that for every directed edge ij, vertex i comes before j in the ordering.
  2. Topological sorting is only possible for Directed Acyclic Graph (DAG).
  3. 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.
  4. 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 are all four edges of the grid - are surrounded by water.**
  Constraints are:
  1. m == grid.length
  2. n == grid[i].length
  3. 1 <= m, n <= 300
  4. 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);
    }
}
```

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:
  1. 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 subtrees of that binary tree.

2. 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 subtrees of that binary tree.