DSA question solve

13 batch

1-a define data structure

A **data structure** is a way of organizing, storing, and managing data in a computer so that it can be used efficiently. It defines the layout or arrangement of data in memory and allows for efficient access and modification of that data.

There are two main types of data structures:

1. **Linear Data Structures**: Elements are arranged in a sequence, and each element is connected to its previous and next element. Examples include:
   - **Array**: A fixed-size sequence of elements of the same data type.
   - **Linked List**: A collection of elements (nodes) where each node contains the data and a reference to the next node.
   - **Stack**: Follows the Last In First Out (LIFO) principle.
   - **Queue**: Follows the First In First Out (FIFO) principle.
2. **Non-linear Data Structures**: Elements are not arranged sequentially. Examples include:
   - **Tree**: A hierarchical structure with nodes connected by edges, where each node can have zero or more child nodes (e.g., binary trees).
   - **Graph**: A collection of nodes (vertices) connected by edges, which can represent various types of relationships.
   - **Hash Table**: Stores key-value pairs and allows for fast lookups using a hash function.

Data structures are fundamental in programming and computer science as they affect the efficiency of algorithms and overall system performance.

a-what is the purpose of data structure in problem solving

ans: The purpose of data structures in problem solving is to provide efficient ways to organize, store, and manipulate data. They are essential because they allow for optimal performance in terms of time and memory when solving computational problems. Here are some key purposes of using data structures in problem solving:

1. Efficient Data Access and Modification

Data structures allow for quick access and manipulation of data. For instance:

- Arrays provide constant-time access to elements via indexing.
- Hash tables offer efficient data retrieval using keys (average O(1) time complexity).
- Trees and graphs allow for hierarchical or networked relationships between elements.

2. Handling Complex Relationships

Data structures help model complex relationships within data:

- **Graphs** model networks (e.g., social networks, transportation routes).
- **Trees** are used to represent hierarchical data, such as file systems or organizational charts.

3. Memory Efficiency

Some data structures are designed to save memory. For example:

- Linked lists dynamically allocate memory for each element, avoiding wasted space.
- Trees and graphs allow for sparse representations, reducing memory overhead.

4. Improving Algorithm Performance

The right data structure ensures that operations like searching, inserting, and deleting are as efficient as possible. For example:

- Sorting algorithms can be optimized using data structures like **heaps** or **binary search trees**.
- Graph algorithms (like Dijkstra's shortest path) rely on efficient graph representations such as adjacency lists or matrices.

5. Simplifying Problem Solving

Data structures allow problems to be broken down into smaller, more manageable parts:

- **Stacks** are useful for solving problems involving backtracking (e.g., depth-first search, parsing).
- **Queues** can be used in breadth-first search or scheduling problems.

6. Enabling Reusability and Modularity

By using well-defined data structures, developers can build reusable components. Many common problems (e.g., sorting, searching) can be addressed by implementing standard algorithms on these data structures.

In essence, the choice of data structure can significantly impact the time complexity, space complexity, and overall feasibility of solving a problem efficiently.

Example:

Example 3: Finding the Shortest Path in a City Map

Problem:

You need to find the shortest path between two locations in a city with multiple roads.

Data Structures:

1. **Graph**: A city map can be represented as a **graph**, where intersections are **nodes** and roads are **edges** with weights (distances). Algorithms like **Dijkstra's** or *A search** can be applied to find the shortest path.

b. write down the difference between linear and non linear datastructure? why graph is a nonlinear data structure

ans: Difference Between Linear and Non-Linear Data Structures

Feature	Linear Data Structure	Non-Linear Data Structure
Definition	Data elements are arranged in a sequential manner, one after the other.	Data elements are arranged in a hierarchical or interconnected structure.

Feature	Linear Data Structure	Non-Linear Data Structure
Memory utilization	Memory is often allocated in a contiguous block.	Memory is utilized in a non-contiguous manner.
Traversal	Traversed in a single run (one element after another).	Traversal requires hierarchical or multi-level visits.
Complexity	Generally simpler to implement and understand.	More complex due to hierarchical relationships and pointers.
Examples	Arrays, Linked Lists, Stacks, Queues	Trees, Graphs, Heaps
Access Time	Accessing elements can be done in sequential order (O(n)) for most linear structures, except arrays where random access is possible (O(1)).	Access depends on the structure (e.g., O(log n) for trees, O(V + E) for graphs).
Data Relationship	Every element has a direct relationship to its previous and next element.	Elements can have multiple relationships (one-to-many, many-to-many).
Insertion and Deletion	Easier in structures like linked lists (O(1)), but more challenging in arrays (O(n) for insertion or deletion).	More complex, as it may involve re-arranging sub-elements or adjusting pointers.
Examples of Real-world Uses	Queues for task scheduling, arrays for static storage.	Graphs for network connections, trees for hierarchical data like file systems.

Why Graph is a Non-Linear Data Structure?

A **graph** is a non-linear data structure because it represents data in a more complex way, where data points (called **vertices**) are connected by edges. Unlike linear structures like arrays or linked lists where elements are stored in a strict sequence (one after another), a graph allows any element (vertex) to be connected to one or more other elements (vertices) in no specific order. This interconnection allows for complex relationships, such as:

- **Many-to-many relationships**: A node in a graph can be connected to multiple other nodes, representing multiple relationships (e.g., in a social network, one user can have many friends).
- **Hierarchical or cyclic relationships**: A graph can represent both hierarchical relationships (like trees) and cyclic ones (where nodes are connected in loops).

This interconnected, non-sequential nature makes graphs non-linear. They are often used to represent networks, maps, and other structures that require flexible connections between elements.

Examples of Graph Usage:

- **Social networks**: Users are nodes, and friendships are edges connecting them.
- **Road networks**: Intersections are nodes, and roads between them are edges.
- **Internet/web**: Websites are nodes, and hyperlinks between them are edges.

Since graphs do not follow a strict sequential order and can have multiple connections and cycles, they fall under the category of non-linear data structures.

c. write a program to create a singly linklist that insert each element in the head of the current list

```
#include<bits/stdc++.h>
using namespace std;
struct node{
  int data;
```

```
  struct node *next;
};
void Addnode(node *&head,int data)
{
   node *temp=new node;
   temp->data=data;
   temp->next=NULL;
   if(head==NULL)
   {
      head=temp;
   }
   else
   {
      node *temp2=new node;
      temp2=head;
      while(temp2->next!=NULL)
      {
         temp2=temp2->next;
      }
      temp2->next=temp;
   }
}
void Insert_at_First(node *&head,int data)
{
   if(head==NULL)
   {
      cout<<"List is Empty create a node First ";
```

```
    }
    node *temp=new node;
    temp->data=data;
    temp->next=head;
    head=temp;
}
void print(node *head)
{
    node *ptr=new node;
    ptr=head;
    cout<<"List are : ";
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" ";
        ptr=ptr->next;
    }
    cout<<endl;
}
int main()
{
     struct node *head=NULL;
    while (true)
    {
        cout<<" 1 Input Element in List "<<endl;
        cout<<" 2 Insert At First"<<endl;
        cout<<" 3 Exit"<<endl;
        cout<<"Enter choise :";
```

```
int ch;
cin>>ch;
switch (ch)
{
    case 1:
{
    cout<<"Enter data : ";
   int x;
   cin>>x;
   Addnode(head,x);
   system("CLS");
   print(head);
   break;

}
case 2:
{
    cout<<"Enter data : ";
   int y;
   cin>>y;
   Insert_at_First(head,y);
   system("CLS");
   print(head);
   break;

}
case 3:
```

```
    {
      system("CLS");
      print(head);
      exit(1);
      break;
    }

    default:
      break;
    }
  }
```

```
  return 0;
}
```

Question 2

a. what do you mean by time complexity of algorithm? suppose an algorithm has O(n^2) time complexity another one has o(n) time complexity which algorithm perform better describe with proper example

ans: **Time complexity** is a way to express the efficiency of an algorithm in terms of how the running time or the number of operations it requires grows as the input size (usually denoted as `n`) increases. It helps in evaluating the performance of an algorithm and predicting its scalability.

Time complexity is generally expressed using **Big-O notation** (e.g., $O(n)$, $O(n^2)$, $O(\log n)$, etc.), which gives an upper bound on the growth rate of the algorithm's time requirement as the input size increases. This allows us to compare algorithms based on their worst-case performance.

$O(n^2)$ vs. $O(n)$: Which Algorithm Performs Better?

If one algorithm has **$O(n^2)$** time complexity and another has **$O(n)$** time complexity, the algorithm with **$O(n)$** performs better, especially as the input size grows.

Here's why:

- **$O(n)$** means the algorithm's running time grows linearly with the input size.
- **$O(n^2)$** means the algorithm's running time grows quadratically (much faster) as the input size increases.

For smaller inputs, the difference might not be noticeable. However, as `n` grows larger, the **$O(n^2)$** algorithm will take significantly more time to run compared to the **$O(n)$** algorithm.

Example

Problem: Finding the Maximum Element in an Array

- **Algorithm 1** ($O(n^2)$) - A **brute force** approach:
  - For each element in the array, compare it with every other element to check if it is the maximum.
  - For an array of size `n`, you will make `n * n = n²` comparisons in the worst case.
- python
- Copy code
- `# Brute force approach (O(n²))`
- `def find_max_brute_force(arr):`
- `    n = len(arr)`

- for i in range(n):
- is_max = True
- for j in range(n):
- if arr[j] > arr[i]:
- is_max = False
- break
- if is_max:
- return arr[i]

Time Complexity:

- **Outer loop** runs `n` times.
- **Inner loop** also runs `n` times for each iteration of the outer loop.
- Total time complexity = **O(n²)**.

Algorithm 2 (O(n)) - A linear scan:

- Iterate through the array once, keeping track of the largest element found so far.
- int max_element = arr[0];
- for (size_t i = 1; i < arr.size(); ++i) {
- if (arr[i] > max_element) {
- max_element = arr[i]; }
- }
- return max_element;

Time Complexity:

- The algorithm scans the array once, comparing each element to the current maximum.
- Total time complexity = **O(n)**.

Performance Comparison

Let’s compare these algorithms with different input sizes:

Example with small input:

```
python
Copy code
arr = [5, 3, 8, 2, 7]
```

Both algorithms will perform similarly, since the input size is small. You might not notice a significant difference in execution time.

Example with large input:

```
python
Copy code
arr = [i for i in range(10000)]
```

Here, the **$O(n^2)$** algorithm will perform significantly worse because:

- $O(n^2)$ means that for 10,000 elements, the brute force algorithm will make $10{,}000^2 = 100{,}000{,}000$ comparisons.
- $O(n)$ means the linear scan algorithm will make only 10,000 comparisons.

Visualization:

Input Size (n)	$O(n^2)$ Comparisons	$O(n)$ Comparisons
10	100	10
100	10,000	100
1,000	1,000,000	1,000
10,000	100,000,000	10,000

As you can see, the **$O(n^2)$** algorithm becomes increasingly inefficient as the input size grows, while the **$O(n)$** algorithm scales much better.

Conclusion:

- **O(n)** performs better than **O(n²)**, especially for large input sizes.
- While both algorithms may perform similarly on small inputs, the **O(n²)** algorithm will take much longer to run for larger inputs due to its quadratic growth, whereas the **O(n)** algorithm grows linearly with the input size. Thus, for efficiency, **O(n)** is preferable.

b. write down a scenerio where using dublly linklist is advantageous

ans: A **doubly linked list** is advantageous in scenarios where you need to efficiently navigate and manipulate data in both forward and backward directions. Here's a specific scenario where a doubly linked list is particularly useful:

Scenario: Implementing an Undo/Redo Feature in a Text Editor

Problem:

A text editor needs to support an "undo" and "redo" feature, allowing users to revert and reapply changes to the document.

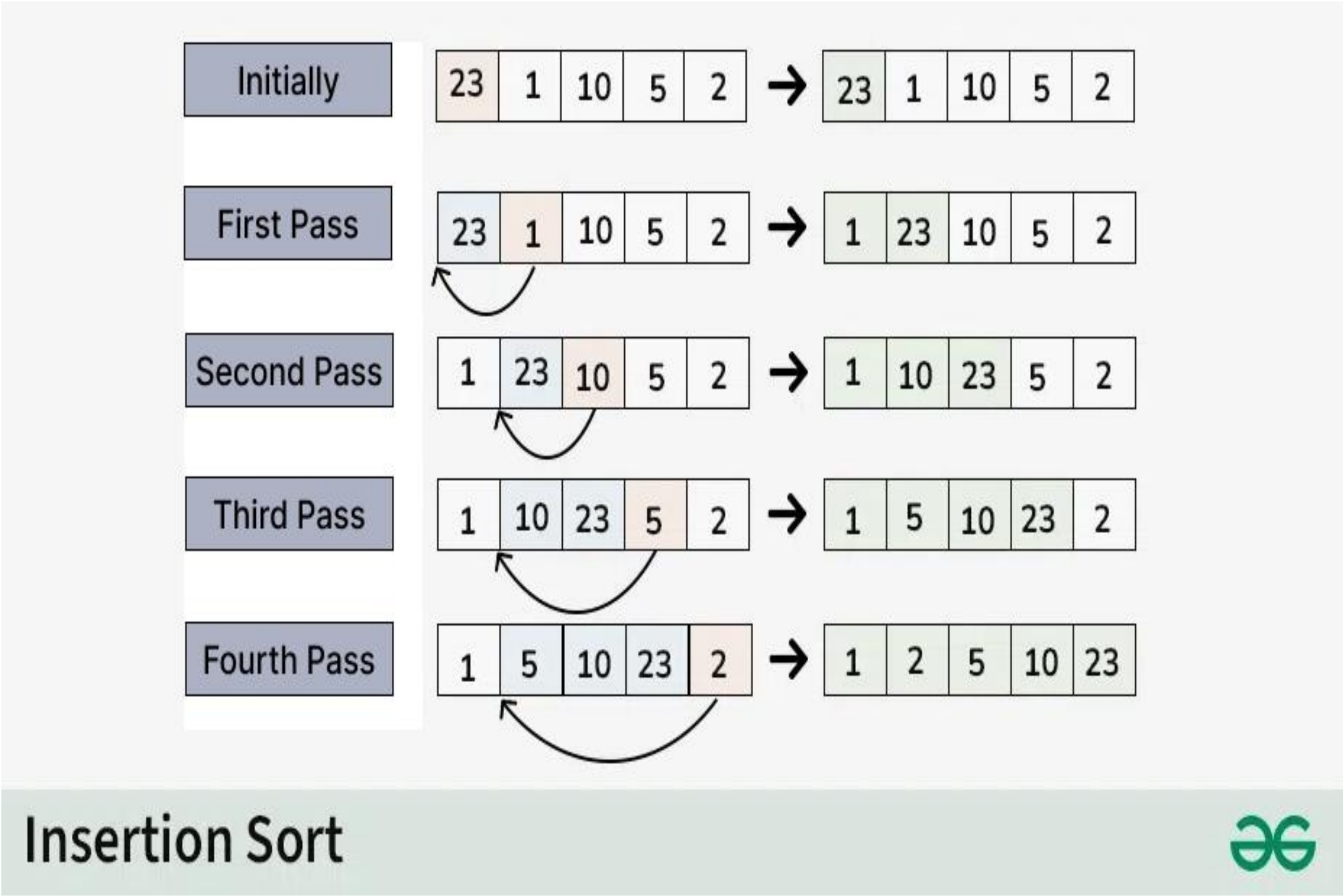
Requirements:

1. **Efficient Navigation**: Users should be able to quickly move between different states of the document (both past and future states).
2. **Quick Insertions and Deletions**: The list of changes should support quick insertions and deletions as users make changes to the document.

Why Doubly Linked List is Advantageous:

1. **Bidirectional Traversal**:
   - **Undo**: Allows moving back to previous states. The doubly linked list allows you to traverse backwards to access previous states.
   - **Redo**: After undoing an action, you can traverse forwards to redo the actions. The doubly linked list's backward and forward pointers make it easy to move in both directions.
2. **Efficient Operations**:
   - **Insertion and Deletion**: In a doubly linked list, adding or removing a state (node) from the middle of the list is efficient, as it involves only updating a few pointers. This is crucial for managing the history of document changes.
3. **State Management**:
   - Each node in the doubly linked list can represent a state of the document. By maintaining pointers to both the previous and next states, you can easily traverse the list to find or revert to any state.

c.sorting using insertion sort.



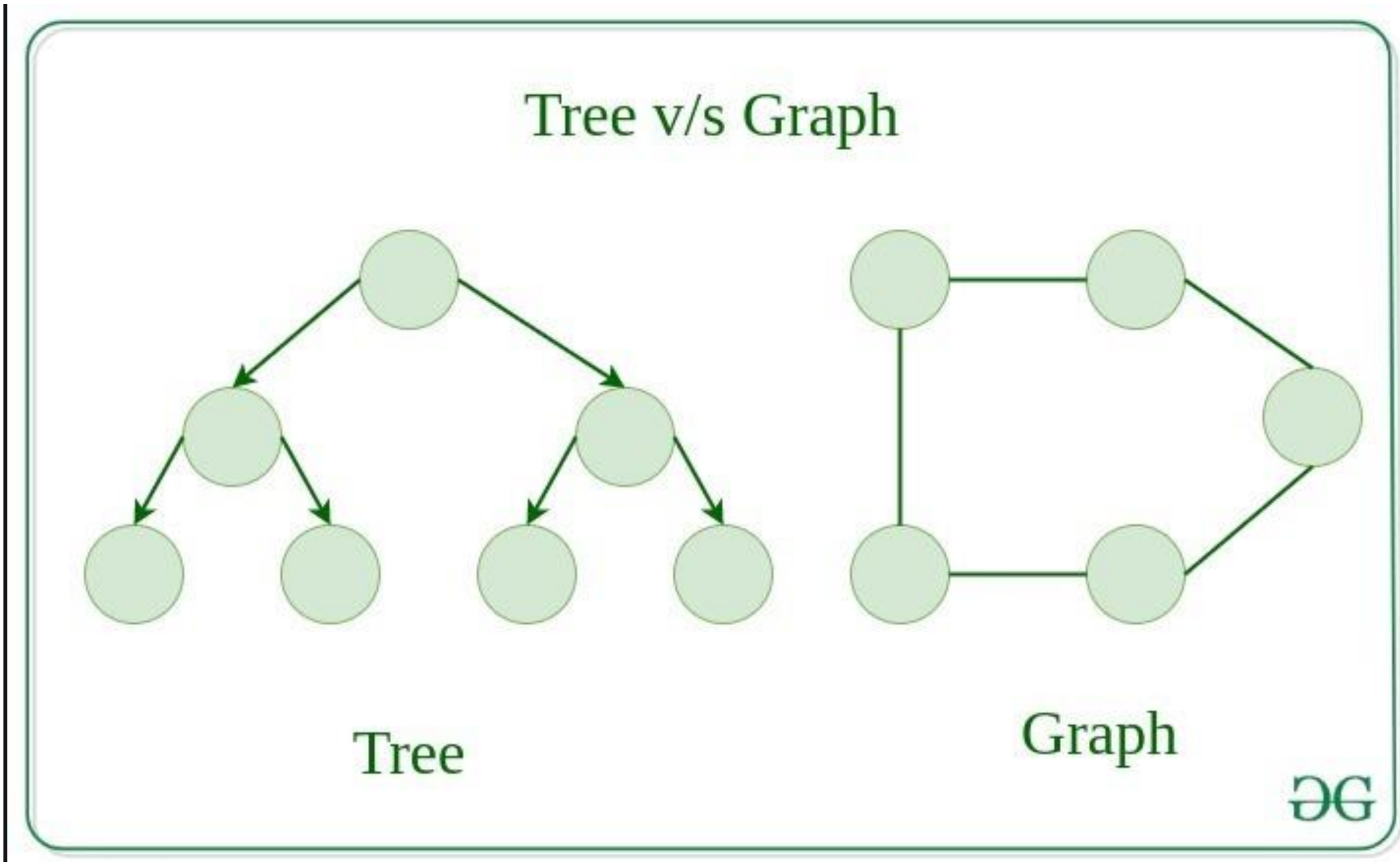

```
/ C++ program for implementation of Insertion Sort
#include <iostream>
using namespace std;

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
```

```
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver method
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

3 a. write down the difference between tree and graph .describe two scenario where using graph and tree data structure is advantageous.

Ans.

Graphs and **trees** are two fundamental data structures used in computer science to represent relationships between objects. While they share some similarities, they also have distinct differences that make them suitable for different applications.



Difference Between Graph and Tree

What is Graph?

A **graph data structure** is a collection of **nodes** (also called vertices) and **edges** that connect them. Nodes can represent entities, such as people, places, or things, while edges represent relationships between those entities.

Graphs are used to model a wide variety of real-world systems, such as **social networks**, **transportation networks**, and **computer networks**.

What is Tree?

A **tree data structure** is a hierarchical data structure that consists of nodes connected by edges. Each node can have multiple child nodes, but only one parent node. The topmost node in the tree is called the **root node**.

Trees are often used to represent hierarchical data, such as file systems, **XML documents**, and **organizational charts**.

Difference Between Graph and Tree

Key Differences Between Graph and Tree

- **Cycles:** Graphs can contain cycles, while trees cannot.
- **Connectivity:** Graphs can be disconnected (i.e., have multiple components), while trees are always connected.

- **Hierarchy:** Trees have a hierarchical structure, with one vertex designated as the root. Graphs do not have this hierarchical structure.
- **Applications:** Graphs are used in a wide variety of applications, such as social networks, transportation networks, and computer science. Trees are often used in hierarchical data structures, such as file systems and XML documents.
  - **Structure**:
    - **Tree**: A tree is a type of graph that is hierarchical and connected. It consists of nodes connected by edges with a strict parent-child relationship. Trees have a single root node and no cycles.
    - **Graph**: A graph is a more generalized data structure consisting of nodes (vertices) and edges. It doesn't have to follow a hierarchical structure and may contain cycles, loops, or disconnected components.
  - **Edges**:
    - **Tree**: In a tree with `n` nodes, there are always `n-1` edges.
    - **Graph**: There can be any number of edges, including `n`, fewer than `n-1`, or more. Graphs can have loops or multiple edges between the same set of nodes.
  - **Cyclic Nature**:
    - **Tree**: A tree is acyclic, meaning there are no loops or cycles.
    - **Graph**: A graph can be cyclic or acyclic, directed or undirected.
  - **Root**:
    - **Tree**: A tree has a designated root node from which all other nodes descend.
    - **Graph**: A graph doesn’t have a designated root node unless explicitly defined, such as in a directed acyclic graph (DAG).
  - **Direction**:
    - **Tree**: Typically has directed edges where the direction represents the parent-child relationship.
    - **Graph**: Can have either directed or undirected edges depending on the use case.

Scenarios Where Using a Graph is Advantageous:

1. **Social Networks**: In a social network, people are represented as nodes, and friendships or connections between them as edges. A graph is suitable here because the connections are non-hierarchical, mutual, and can be bidirectional. This allows modeling relationships such as who follows whom on a social platform or how people are connected in a professional network (LinkedIn).
2. **Transportation Networks**: In road or railway networks, intersections (nodes) are connected by roads or tracks (edges), where there may be multiple paths between two points. Using a graph allows modeling different routes and calculating the shortest path between destinations using algorithms like Dijkstra’s or A*.

Scenarios Where Using a Tree is Advantageous:

1. **File System Structure**: The hierarchical structure of directories and subdirectories in an operating system is best represented as a tree. Each folder (node) can contain subfolders (children), but there is a single parent directory leading up to the root (e.g., root directory `/` in Linux).
2. **Binary Search Trees (BSTs) in Databases**: Trees are advantageous in databases where fast searching, insertion, and deletion are needed. In a binary search tree, data is stored hierarchically, and searching for an item can be done in O(log n) time due to the binary division of elements at each level.

c. suppose you need to solve a problem using stack but your designer solved it using queue data structure now write down possible approach to solve it

2. Using a Single Queue (Reversing Elements on Push)

This approach involves using just one queue but reversing the order of elements each time a new element is pushed. The idea is that when you push a new element, you rotate the queue so that the newly added element appears at the front of the queue (mimicking the top of a stack).

Algorithm:

- **Push operation**:
  1. Enqueue the new element into the queue.
  2. Dequeue all previous elements from the front of the queue and enqueue them back, so the new element moves to the front of the queue (acting as the stack's top).
- **Pop operation**: Simply dequeue the front element from the queue, as it simulates popping from the top of the stack.

Example:

- Push 1 → Queue: `[1]`
- Push 2 → Queue: `[2, 1]` (after rotating 1 to the back)
- Push 3 → Queue: `[3, 2, 1]` (after rotating 1 and 2 to the back)
- Pop → Dequeue from the front, returns 3.

This method ensures that the order of elements is reversed every time you push, making the queue behave like a stack.

Set b

5 a. define linklist. write down the application of linklist.

Ans. A **Linked List** is a linear data structure in which elements, called **nodes**, are stored in a sequence, but unlike arrays, they are not stored in contiguous memory locations. Each node in a linked list contains two parts:

1. **Data**: The value or information the node holds.
2. **Pointer** (or **Reference**): A reference to the next node in the sequence. The last node points to `NULL` (or `None` in some programming languages), indicating the end of the list.

There are different types of linked lists:

- **Singly Linked List**: Each node points to the next node in the sequence.
- **Doubly Linked List**: Each node contains references to both the next node and the previous node.
- **Circular Linked List**: The last node points back to the first node, forming a loop.

Applications of Linked List

1. **Dynamic Memory Allocation**: Linked lists allow for dynamic memory allocation where the size of data structures can grow or shrink during runtime, unlike arrays which have a fixed size.
2. **Implementing Data Structures**: Linked lists are used to implement various abstract data types such as:
    - Stacks
    - Queues
    - Hash tables
    - Graphs (Adjacency list representation)
3. **Efficient Insertions/Deletions**: Linked lists allow efficient insertions and deletions (O(1) time) at the beginning or end without the need for shifting elements, unlike arrays.
4. **Handling Large Data**: When you don't know the size of the dataset in advance, linked lists are useful as their size can grow dynamically.
5. **Memory Utilization**: Linked lists can use memory more efficiently in some cases by allocating memory only when needed. This is especially beneficial when dealing with sparse datasets.
6. **Undo/Redo Functionality in Applications**: Linked lists are used in applications like text editors, where undo/redo actions are implemented using linked lists to store different states of the content.
7. **Navigating Files or Web Pages**: Doubly linked lists are used to store the history of web pages visited, or navigating through files where forward and backward traversal is required.
8. **Polynomial Arithmetic**: In applications involving polynomial arithmetic (e.g., adding or multiplying polynomials), linked lists can efficiently represent polynomials as each term can be treated as a node.
9. **Music/Video Playlists**: Circular linked lists can be used to implement playlist functionality where the last song/video loops back to the first one.
10. **OS Process Management**: Linked lists are used in process scheduling, where processes are linked and executed in a queue.

The flexibility and dynamic nature of linked lists make them suitable for a variety of real-world applications where efficient memory usage and dynamic resizing are essential.

b. write a program to implement a queue using linklist with enqueue ,dequeue .

ans.

```
#include<bits/stdc++.h>
using namespace std;

struct node{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;

void enque(int data){
    struct node *newnode=new node;
    if(newnode==NULL){
        cout<<"Queue is full";
    }
    else{
        newnode->data=data;
        newnode->next=NULL;
        if(front==NULL&& rear==NULL){
            front=rear=newnode;
        }
        else{
            rear->next=newnode;
            rear=newnode;
        }
```

```
    }
}
int dequeue(){
    int val=-1;
    struct node *temp;
    temp=front;

    if(front==NULL){
        cout<<"Queue is empty";
    }
    else{
        front=front->next;
        val=temp->data;
        delete(temp);
        //c free(temp);
    }
    return val;
}

void display(){
    struct node *ptr;
    if(front==NULL&& rear==NULL){
            cout<<"Queue is empty";
    }
    else{
        ptr=front;
        while(ptr!=NULL){
            cout<<ptr->data<<"  ";
            ptr=ptr->next;
        }
        cout<<endl;
```

```cpp
    }

}


int main(){
    while(true){
        cout<<"1 to enque data "<<endl;
        cout<<"2 deque data in the queue"<<endl;
        cout<<"3 exit"<<endl;
        cout<<"Enter coice :"<<endl;
        int ch;
        cin>>ch;
        switch(ch){
            case 1:
            {
                cout<<" enter data :";
                int x;
                cin>>x;
                system("CLS");
                enque(x);
                display();
                break;
            }
            case 2:
            {
                int xx;
                xx=dequeue();
               // system("CLS");
                cout<<"dequed element:"<<xx<<endl;
                display();
                break;
```

```
            }
            case 3:
            {
                 while(front!=nullptr){
                dequeue();
            }
            exit(0);

            }
            default:{
                cout<<"Invalid coice!"<<endl;
                break;
            }

        }

    }
}
```

c. describe the scenario for a practical example where using priority queue is advantageous

ans. **Scenario: Operating System Task Scheduling**

In an operating system, multiple processes or tasks run concurrently. These tasks can have different levels of priority based on their urgency or importance. For instance, real-time tasks like handling input from a keyboard or mouse might have higher priority than background tasks like updating software or syncing files.

Using a **Priority Queue** helps the operating system efficiently manage these processes by ensuring that higher-priority tasks are executed before lower-priority ones.

Steps in this Scenario:

1. **Arrival of Processes**: Processes arrive at the CPU with different priorities. For example:

    - A real-time system interrupt (e.g., keyboard input) has the highest priority.
    - A video streaming process has medium priority.
    - A background file syncing task has low priority.
2. **Inserting Processes into the Queue**:
    - Each process is inserted into the **priority queue** based on its priority value.
    - The priority queue keeps the highest-priority process at the front, ensuring efficient retrieval.
3. **Execution of Processes**:
    - The operating system scheduler retrieves the highest-priority process from the front of the queue and assigns the CPU to it.
    - Once the high-priority process is completed, the next highest-priority process is executed.
4. **Dynamic Priority Management**:
    - Priorities can change dynamically. For example, if a background task becomes more urgent, its priority can be increased and re-inserted into the queue.
    - If new high-priority tasks (like system interrupts) arrive, they are placed at the front of the queue.

Why is Priority Queue Advantageous in This Scenario?

- **Efficient Task Management**: Using a priority queue allows the operating system to handle multiple tasks efficiently. Tasks that are critical (e.g., handling user input) are executed first, minimizing response time.
- **Dynamic Rescheduling**: Tasks that change in priority or new high-priority tasks are immediately recognized by the queue, allowing dynamic and flexible scheduling.
- **Fairness in Resource Allocation**: By assigning higher priority to more critical tasks, the system ensures that resources (CPU time) are allocated in a fair and effective way according to urgency.
- **Preemptive Scheduling**: In preemptive scheduling algorithms, the priority queue is used to preempt a running task if a higher-priority task arrives. The running task is pushed back into the queue, ensuring that no time is wasted.

Applications of Priority Queue in Task Scheduling:

- **Real-Time Systems**: Where deadlines are strict, and certain tasks must be executed within a specific time frame.
- **Job Scheduling in Cloud Computing**: Allocating resources to cloud-based tasks with different priority levels.
- **Network Packet Routing**: Where some packets (e.g., video/audio streaming) are given higher priority than others (e.g., file downloads).

In this example, a priority queue ensures optimal use of CPU resources, providing a smooth and efficient multitasking environment.

6.b. write down one practical example of each using preorder,post order and inorder traversal;

Here's a practical example for each type of tree traversal:

1. Preorder Traversal (Root -> Left -> Right)

- **Practical Example**: **File System Traversal**

  Imagine you're traversing through a file system where each directory is a node, and files or subdirectories are children nodes. In **preorder traversal**, you'd first visit the directory (root), then its left subtree (files/subdirectories), and finally the right subtree (other files/subdirectories).

  Example:

  ```
  arduino
  Copy code
  /home
     ├── user1
     │      ├── doc1.txt
     │      └── doc2.txt
     └── user2
           └── doc3.txt
  ```

  Preorder traversal: `/home, user1, doc1.txt, doc2.txt, user2, doc3.txt`

2. Inorder Traversal (Left -> Root -> Right)

- **Practical Example**: **Binary Search Tree (BST) for Sorting**

  In a binary search tree (BST), **inorder traversal** gives nodes in a sorted order. This is useful when you need to retrieve sorted data from a BST, such as searching or ordering numbers.

  Example:
  For a BST:

  ```
  markdown
  Copy code
       5
      / \
     3   7
    / \   \
   2   4   8
  ```

  Inorder traversal (sorted): `2, 3, 4, 5, 7, 8`

3. Postorder Traversal (Left -> Right -> Root)

- **Practical Example**: **Expression Tree Evaluation**

In an expression tree, the leaves are operands and internal nodes are operators. **Postorder traversal** is used to evaluate the expression because it processes operands before operators (left, right, then root).

Example:
For the expression `(3 + 2) * (5 - 1)`:

```
javascript
Copy code
      *
     / \
    +   -
   / \  / \
  3   2 5  1
```

Postorder traversal: `3, 2, +, 5, 1, -, *`

In preorder traversal, the root node is always processed first, then you recursively traverse the left subtree, and finally, the right subtree.

c. In preorder traversal, the root node is always processed first, then you recursively traverse the left subtree, and finally, the right subtree.

Data structures exist in a variety of forms because they are optimized for different tasks and scenarios. While it may seem that every data structure ultimately serves the same goal—storing and organizing data—the **efficiency** with which they achieve specific tasks (like searching, inserting, or deleting data) varies greatly. Here are some reasons why multiple data structures are used instead of relying on just one:

1. Efficiency for Different Operations

Each data structure has strengths and weaknesses for particular types of operations. Depending on what the program needs to prioritize, a certain data structure will perform better than others:

- **Arrays** allow fast random access to elements but are slow for insertion and deletion.
- **Linked Lists** allow fast insertions and deletions but are slow for random access.
- **Binary Search Trees** allow fast searching, insertion, and deletion if balanced, but can degrade in performance if unbalanced.

In complex systems, different operations need to be optimized for speed, memory use, or both, depending on the situation.

2. Memory Usage

Some data structures use memory more efficiently than others:

- **Arrays** use a fixed amount of contiguous memory, which is ideal for systems with limited memory but can lead to memory waste if the array is larger than needed.
- **Linked Lists** use dynamic memory, allocating memory only as needed, but the memory overhead of storing pointers (links) can add up.

In memory-constrained environments (like embedded systems), efficient memory usage is critical, and the choice of data structure can significantly impact performance.

3. Data Organization and Retrieval

Different data structures organize data in ways that optimize specific types of retrieval:

- **Hash Tables** allow constant-time retrieval of data with keys but don't maintain any order.
- **Heaps** are useful for efficiently finding the maximum or minimum value, often used in priority queues.
- **Trees (like Binary Search Trees)** keep elements in a sorted order, making range queries efficient.

The way data is organized impacts how quickly and efficiently you can access or modify it.

4. Flexibility and Use Cases

Different use cases require different levels of flexibility:

- **Stacks** follow the Last In, First Out (LIFO) principle, making them ideal for managing function calls, recursion, or backtracking algorithms.
- **Queues**, on the other hand, follow the First In, First Out (FIFO) principle and are ideal for scheduling processes, managing tasks, or breadth-first search.

Some algorithms and applications require specific behavior that is well-suited to certain data structures.

5. Concurrency and Multithreading

In multi-threaded environments, certain data structures provide better guarantees for safe access:

- **Concurrent queues** or **lock-free data structures** allow multiple threads to access them without needing heavy locking mechanisms, ensuring better performance in highly concurrent systems.

6. Real-World Constraints and Trade-offs

In practice, developers face many constraints, such as time complexity, space complexity, and ease of implementation. Choosing the right data structure is about balancing those constraints based on the specific problem:

- A **graph** is the right structure for networked data like social networks or road maps, where nodes and edges represent connections.
- A **tree** might be more appropriate for hierarchical data like organization charts or file systems.

7. Algorithmic Needs

Certain algorithms rely on specific data structures:

- **Dijkstra's shortest path algorithm** requires a priority queue, which is often implemented using a heap.
- **Depth-first search** and **breadth-first search** algorithms are easily implemented using stacks and queues, respectively.

Although the final goal of most data structures is to store and retrieve data, the variety of data structures exists because no single structure optimally handles all situations. The choice depends on the specific needs of the problem, whether it's speed, memory usage, ease of maintenance, or type of operations required. Different data structures allow us to tailor the approach for maximum efficiency and effectiveness.

7.a. write down the practical scenario where using array is array is advantegeous than linklist.

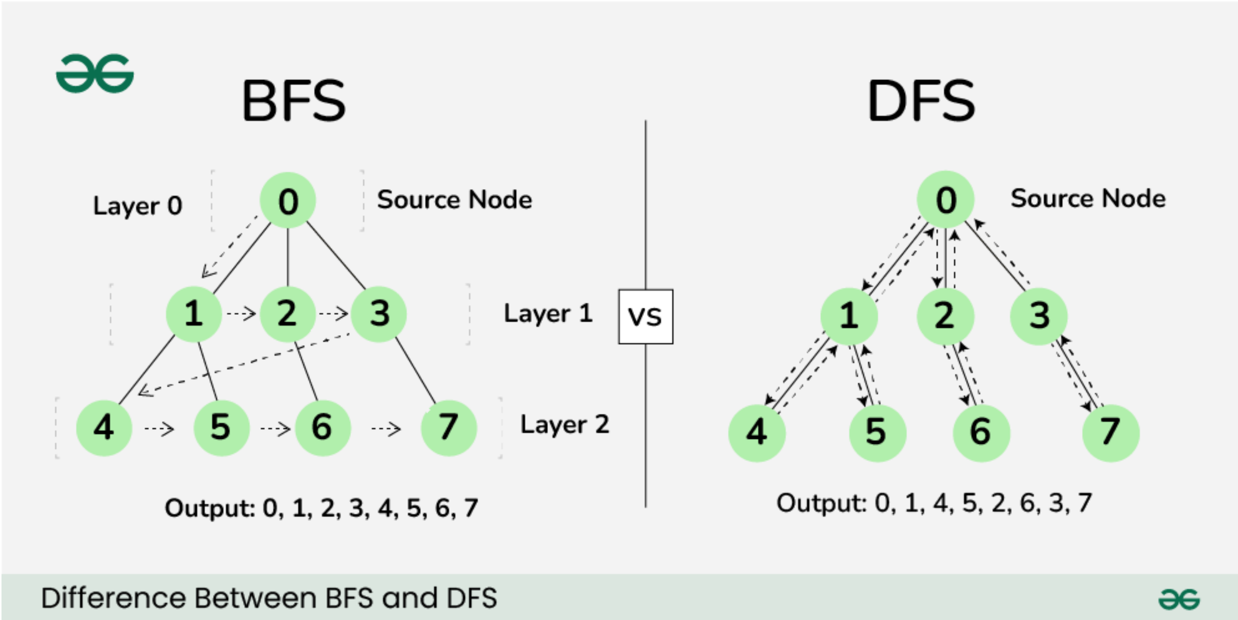
Summary of Advantages of Arrays over Linked Lists:

- **Fast random access** with constant time complexity (`O(1)`).
- **Efficient memory usage** when the number of elements is fixed.
- **Lower memory overhead** because there are no extra pointers.
- **Cache-friendly** due to contiguous memory allocation.
- **Simpler and faster** for iteration and accessing elements sequentially.

In scenarios where fast random access, fixed size, or memory efficiency is critical, array outperform linked lists significantly.

b.bfs dfs ans.

ans. Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental algorithms used for traversing or searching graphs and trees. This article covers the basic difference between Breadth-First Search and Depth-First Search.



7.c. 50,10,13,45,55,110,5,64,47 1.draw the binary search tree 2.show output value of preorder inorder and postorder traversal 3.show the resulting tree after we delate 47,110,50(each deletation is applied on the original tree).

Ans: 1. Drawing the Binary Search Tree (BST)

We'll insert the values one by one into the binary search tree, following the BST property:

- **Left child** is smaller than the parent node.
- **Right child** is larger than the parent node.

Given values: `50, 10, 13, 45, 55, 110, 5, 64, 47`

Insertions:

1. `50` becomes the root.
2. `10` is smaller than `50`, so it goes to the left of `50`.
3. `13` is larger than `10`, so it goes to the right of `10`.
4. `45` is larger than `13`, so it goes to the right of `13`.
5. `55` is larger than `50`, so it goes to the right of `50`.
6. `110` is larger than `55`, so it goes to the right of `55`.
7. `5` is smaller than `10`, so it goes to the left of `10`.
8. `64` is larger than `55` but smaller than `110`, so it goes to the left of `110`.
9. `47` is larger than `45` but smaller than `50`, so it goes to the left of `50`.

Binary Search Tree (BST):

```
markdown
```

```
Copy code
        50
       /  \
     10    55
    /  \     \
   5    13    110
         \    /
          45  64
            \
             47
```

2. Preorder, Inorder, and Postorder Traversals

a. Preorder Traversal (Root → Left → Right):

- Start at the root (50), then explore the left subtree before the right subtree.
- **Preorder Traversal**: `50, 10, 5, 13, 45, 47, 55, 110, 64`

b. Inorder Traversal (Left → Root → Right):

- Explore the left subtree first, then the root, and finally the right subtree.
- **Inorder Traversal**: `5, 10, 13, 45, 47, 50, 55, 64, 110`
    - Note: Inorder traversal of a binary search tree always gives a **sorted list** of elements.

c. Postorder Traversal (Left → Right → Root):

- Explore both subtrees first, then the root.
- **Postorder Traversal**: `5, 47, 45, 13, 10, 64, 110, 55, 50`

---

3. Resulting Tree After Deletions

We'll apply the deletions as specified. Each deletion happens **on the original tree**, one by one.

a. Deletion of 47

1. Find `47` in the tree (it's the right child of `45`).
2. `47` is a **leaf node**, so we can simply remove it.

Tree after deleting 47:

```
markdown
Copy code
        50
       /  \
     10    55
    /  \     \
   5    13    110
```

```
          \     /
          45   64
```

b. Deletion of 110

1. Find `110` (it's the right child of `55`).
2. `110` has one child (`64`), so we remove `110` and replace it with its child `64`.

Tree after deleting 110:

```
markdown
Copy code
        50
       /  \
     10    55
    /  \     \
   5    13    64
         \
          45
```

c. Deletion of 50

1. `50` is the **root node**, and it has two children (`10` and `55`).
2. When deleting a node with two children, we find its **inorder successor** (the smallest node in the right subtree). The inorder successor of `50` is `55`.
3. Replace `50` with `55`, and then remove the original `55` node.

Tree after deleting 50:

```
markdown
Copy code
        55
       /  \
     10    64
    /  \
   5    13
         \
          45
```

Summary of the Results:

1. **Original Binary Search Tree (BST)**:

   ```
   markdown
   Copy code
        50
       /  \
     10    55
    /  \     \
   5    13    110
         \     /
          45   64
   ```

```
\
 47
```

2. **Preorder Traversal**: `50, 10, 5, 13, 45, 47, 55, 110, 64` **Inorder Traversal**: `5, 10, 13, 45, 47, 50, 55, 64, 110` **Postorder Traversal**: `5, 47, 45, 13, 10, 64, 110, 55, 50`
3. **Tree After Deletions**:
   - After deleting 47:

     ```
     Copy code
        50
       /  \
     10    55
     ```

4. / \
   5 13 110 \ / 45 64
5. diff
6. Copy code
7. - After deleting 110:
8. Copy code
9. 50
10. / \
11. 10 55 / \
    5 13 64
    45
12. diff
13. Copy code
14. - After deleting 50:
15. Copy code
16. 55
17. / \
18. 10 64 /
    5 13
    45
19. Copy code

4o

You said:

50,10,13,45,55,110,5,31,64,47 1.draw the binary search tree 2.show output value of preorder inorder and postorder traversal 3.show the resulting tree after we delate 47,110,50(each deletetation is applied on the original tree)

ChatGPT said:

ChatGPT

Let's go step by step as per your requirements.

1. Drawing the Binary Search Tree (BST)

We insert the values one by one into the binary search tree, following the **BST property**:

- **Left child** is smaller than the parent node.
- **Right child** is larger than the parent node.

Given values: `50, 10, 13, 45, 55, 110, 5, 31, 64, 47`

Insertions:

1. `50` becomes the root.
2. `10` is smaller than `50`, so it goes to the left of `50`.
3. `13` is larger than `10`, so it goes to the right of `10`.
4. `45` is larger than `13`, so it goes to the right of `13`.
5. `55` is larger than `50`, so it goes to the right of `50`.
6. `110` is larger than `55`, so it goes to the right of `55`.
7. `5` is smaller than `10`, so it goes to the left of `10`.
8. `31` is smaller than `45` but larger than `13`, so it goes to the left of `45`.
9. `64` is larger than `55` but smaller than `110`, so it goes to the left of `110`.
10. `47` is smaller than `50` but larger than `45`, so it goes to the right of `45`.

Binary Search Tree (BST):

```
markdown
Copy code
        50
       /   \
     10     55
    /  \      \
   5    13     110
          \    /
           45   64
          /  \
        31   47
```

---

2. Preorder, Inorder, and Postorder Traversals

a. Preorder Traversal (Root → Left → Right):

- Start at the root (50), then explore the left subtree before the right subtree.
- **Preorder Traversal**: `50, 10, 5, 13, 45, 31, 47, 55, 110, 64`

b. Inorder Traversal (Left → Root → Right):

- Explore the left subtree first, then the root, and finally the right subtree.
- **Inorder Traversal**: `5, 10, 13, 31, 45, 47, 50, 55, 64, 110`
    - Note: Inorder traversal of a binary search tree always gives a **sorted list** of elements.

c. Postorder Traversal (Left → Right → Root):

- Explore both subtrees first, then the root.

- **Postorder Traversal:** `5, 31, 47, 45, 13, 10, 64, 110, 55, 50`

---

3. Resulting Tree After Deletions

We'll apply the deletions as specified. Each deletion happens **on the original tree**, one by one.

a. Deletion of 47

1. Find `47` in the tree (it's the right child of `45`).
2. `47` is a **leaf node**, so we can simply remove it.

Tree after deleting 47:

```
markdown
Copy code
        50
       /   \
     10      55
    /  \       \
   5    13      110
         \      /
          45   64
         /
        31
```

b. Deletion of 110

1. Find `110` (it's the right child of `55`).
2. `110` has one child (`64`), so we remove `110` and replace it with its child `64`.

Tree after deleting 110:

```
markdown
Copy code
        50
       /   \
     10      55
    /  \       \
   5    13      64
         \
          45
         /
        31
```

c. Deletion of 50

1. `50` is the **root node**, and it has two children (`10` and `55`).
2. When deleting a node with two children, we find its **inorder successor** (the smallest node in the right subtree). The inorder successor of `50` is `55`.
3. Replace `50` with `55`, and then remove the original `55` node.

Tree after deleting 50:

```
markdown
Copy code
        55
       /   \
    10      64
   /  \
  5    13
         \
          45
         /
       31
```

Summary of the Results:

1. **Original Binary Search Tree (BST)**:

   ```
   markdown
   Copy code
        50
       /   \
     10      55
    /  \       \
   5    13      110
          \     /
           45  64
          /  \
        31   47
   ```

2. **Traversals**:
   - **Preorder**: `50, 10, 5, 13, 45, 31, 47, 55, 110, 64`
   - **Inorder**: `5, 10, 13, 31, 45, 47, 50, 55, 64, 110`
   - **Postorder**: `5, 31, 47, 45, 13, 10, 64, 110, 55, 50`
3. **Tree After Deletions**:
   - After deleting `47`:

     ```
     Copy code
        50
       /   \
     10     55
     ```

4. / \
   5 13 110 \ / 45 64 / 31

```
5. markdown
6. Copy code
7. - After deleting `110`:
8. Copy code
9.  50
10.  /   \
```

11. 10 55 / \
    5 13 64
    45 / 31
12. markdown
13. Copy code
14. - After deleting `50`:
15. Copy code
16. 55
17. / \
18. 10 64 /
    5 13
    45 / 31

8.a. write down three practical example of using stack and queue data structure;

Here are three practical examples for both **stack** and **queue** data structures, highlighting their real-world applications:

1. Stack:

A **stack** follows the **Last In, First Out (LIFO)** principle, where the last element added is the first to be removed.

a. Undo/Redo Functionality in Text Editors

- **Scenario**: Most text editors (e.g., Microsoft Word, Notepad, or programming IDEs) have an undo/redo feature that allows users to revert or repeat their recent actions.
- **How Stack is Used**:
    - Every time a change is made, it is pushed onto the **undo stack**.
    - When you press undo, the action is popped from the stack and applied in reverse.
    - Similarly, redo operations are managed by a separate **redo stack** that stores undone actions.

b. Function Call Stack in Programming

- **Scenario**: When a function is called in a program, the current function's state is saved so the program can return to it after the called function finishes execution.
- **How Stack is Used**:
    - The function call stack stores the return addresses, local variables, and function arguments when a function is invoked.
    - Once the called function completes, its state is popped from the stack, and control is returned to the previous function.

c. Expression Evaluation (Postfix/Prefix)

- **Scenario**: Stack is used to evaluate mathematical expressions written in postfix (Reverse Polish Notation) or prefix form.
- **How Stack is Used**:

    - In postfix evaluation, operators are applied to operands on the stack.
    - Operands are pushed onto the stack, and when an operator is encountered, the appropriate number of operands is popped, and the result is pushed back onto the stack.

2. Queue:

A **queue** follows the **First In, First Out (FIFO)** principle, where the first element added is the first to be removed.

a. Printer Queue

- **Scenario**: When multiple print jobs are sent to a printer, they are processed in the order they were received.
- **How Queue is Used**:
    - Print jobs are enqueued in the order they are submitted.
    - The printer processes the jobs in the same order, dequeuing one job at a time once printing is complete.

b. Customer Service Call Center (FIFO)

- **Scenario**: In a customer service call center, incoming calls are handled in the order they are received.
- **How Queue is Used**:
    - Calls are enqueued as customers dial in.
    - Agents answer the calls in the same order they were added to the queue, ensuring fairness and order.

c. Task Scheduling in Operating Systems

- **Scenario**: Operating systems use queues to manage the scheduling of tasks (or processes) that need CPU time.
- **How Queue is Used**:
    - Processes that need to be executed are enqueued in the task queue.
    - The operating system schedules these tasks using different queue-based algorithms (e.g., Round Robin), ensuring that each task gets its share of CPU time in order.

Summary of Real-World Uses:

- **Stack**:
    1. **Undo/Redo** operations in text editors.
    2. **Function Call Stack** during program execution.
    3. **Expression Evaluation** in compilers or calculators.
- **Queue**:
    1. **Printer Job Queue** for managing print requests.

2. **Call Center** queue for managing customer calls.
3. **Task Scheduling** in operating systems for process management.

-

b.what is circular queue

ans. A **circular queue** is a type of **queue** data structure in which the last position is connected back to the first position, forming a circle. Unlike a regular (linear) queue, in which elements are inserted from the rear and removed from the front, a circular queue efficiently utilizes space by connecting the end of the queue back to the beginning, preventing unused spaces in scenarios where elements are dequeued.

```
#include <iostream>
using namespace std;

#define n 5
int queue1[n];
int front = -1;
int rear = -1;
void enqueue(){
    int x;
    cout<<"Enter value you want to enqueue"<<endl;
    cin>>x;
    if((rear+1)%n==front){
        cout<<"overflow"<<endl;
    }
    else if(front==-1&&rear==-1){
        front=rear=0;
        queue1[rear]=x;
    }
    else{
        rear=(rear+1)%n;
        queue1[rear]=x;
    }
```

```
}

void dequeue(){
    if(front==-1&&rear==-1){
        cout<<"empty"<<endl;
    }
    else if(front==rear){
        front=rear=-1;
    }
    else{
        front=(front+1)%n;
    }
}

void display(){
    int i=front;
    if(front==-1&&rear==-1){
        cout<<"empty"<<endl;
    }
    else{

        while(i!=rear){
            cout<<queue1[i]<<" ";
            i=(i+1)%n;
        }
        cout<<queue1[rear]<<endl;
    }
}

void peak(){
    if(front==-1&&rear==-1){
        cout<<"empty"<<endl;
```

```
    }
    else{
        cout<<queue1[front]<<endl;
    }
}
int main()
{
    int ch;
    do
    {
        cout << "1.for enqueue" << endl;
        cout << "2.for dequeue" << endl;
        cout << "3.show peak" << endl;
        cout << "4.DISPLAY ELEMENT" << endl;
        cout<<"-1 exict"<<endl;

        cin >> ch;
        switch (ch)
        {
        case 1:
            enqueue();
            display();
            break;
        case 2:
            dequeue();
            display();
            break;
        case 3:
            peak();
            break;
        case 4:
            display();
```

```
            break;
        case -1:
            cout << "Exictint program" << endl;
            break;
        default:
            cout << "invalid option" << endl;
        }

    } while (ch != -1);
}
```