**SWARNANDHRA COLLEGE OF ENGINEERING & TECHNOLOGY**

**[AUTONOMOUS]**

**Seetharampuram, NARSAPUR-534 280**

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING**

**PART-A**

**UNIT-1**

**1.Compare and contrast Time complexity and Space complexity with examples**

### **Time Complexity:**

* **Definition:** Time complexity measures the amount of time an algorithm takes to run as a function of the input size (n). It reflects how the execution time grows with the input size.
* **Example Metric:** The most common time complexity notations include O(1), O(n), O(n^2), O(log n), etc.
* **Focus:** It focuses on **how fast** an algorithm performs.
* **Measured In:** The number of **basic operations** (comparisons, assignments, etc.) that the algorithm performs.

**Example :**

* **Linear Search:** Searching for an element in an unsorted array.
  + **Worst-case time complexity:** O(n), because in the worst case, it may have to check all elements.

### **Space Complexity:**

* **Definition:** Space complexity measures the amount of memory an algorithm uses as a function of the input size. It includes both the memory needed for the input itself and any additional memory used by the algorithm.
* **Example Metric:** Similar to time complexity, the space complexity can be O(1), O(n), O(n^2), etc.
* **Focus:** It focuses on **how much memory** the algorithm needs.
* **Measured In:** The number of **memory units** used by the algorithm (e.g., arrays, variables, recursion stacks).

**Example :**

* **Merge Sort:** A sorting algorithm that requires extra space to store the divided sub-arrays during the merge phase.
  + **Space complexity:** O(n), since it needs additional arrays for merging.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Time Complexity** | **Space Complexity** |
| **Focus** | Measures the time taken to execute the algorithm. | Measures the memory used by the algorithm. |
| **Metric** | Counts the number of operations or steps. | Counts the memory units required. |
| **Optimization Goal** | Reduce the number of operations. | Reduce memory usage. |
| **Example of Trade-off** | Algorithms can be optimized for faster execution, often at the cost of more memory usage (e.g., dynamic programming). | Algorithms can be optimized for minimal memory usage, often at the cost of more processing time (e.g., iterative algorithms vs. recursion). |

### **Trade-off Example:**

* **Dynamic Programming** vs. **Recursion:** In dynamic programming, you trade space for time by storing the results of subproblems to avoid recomputation, improving time complexity. This, however, increases space complexity.

**Fibonacci Sequence:**

1. **Recursive approach:** Has time complexity O(2^n) due to repeated recalculations, and space complexity O(n) for the recursion stack.
2. **Dynamic Programming approach (using memoization):** Reduces the time complexity to O(n), but increases space complexity to O(n) due to the stored results.

**2 .Construct B-Tree of order-4 by inserting values with following set of data 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8.**

## **Constructing a B-Tree of Order 4**

**Understanding B-Trees:**

* **Order:** Determines the maximum number of children a node can have.
* **Keys:** Each node can hold up to m-1 keys, where m is the order.
* **Children:** A node with k keys has k+1 children.
* **Balanced:** All leaf nodes are at the same level.

**Insertion Process:**

1. **Start at the root:** Begin at the root node.
2. **Traverse:** If the node is a leaf, insert the key. If not, find the appropriate child to continue the search.
3. **Split if full:** If the node becomes full, split it into two nodes and promote the middle key to its parent.

**Data:** 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

**step1 :** Insert 5



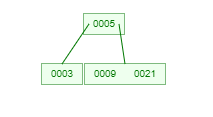
**step2 :** Insert 3



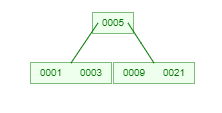
**step3 :** Insert 21



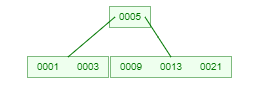
**step4:** Insert 9



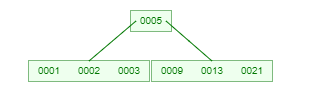
**step5:** Insert 1



**step6:** Insert 13



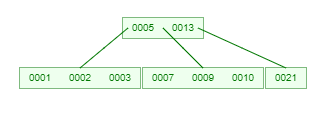
**step4:** Insert 2



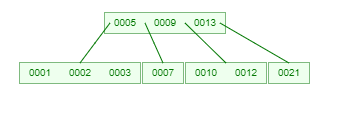
**step4:** Insert 7



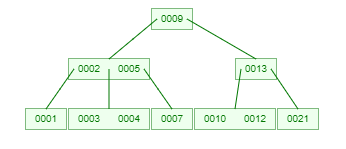
**step4:** Insert 10



**step4:** Insert 12



**step4:** Insert 4



**step4:** Insert 8



The final B - Tree :



**3.Compare and contrast about Omega, Theta and Big-O notation with examples**

## **Omega, Theta, and Big-O Notation: A Comparison**

These notations are used in computer science to describe the asymptotic behavior of algorithms, particularly in terms of time complexity. They provide a way to analyze how the running time of an algorithm scales with the input size.

### **Omega Notation (Ω)**

* **Lower bound:** Represents the minimum growth rate of a function.
* **Meaning:** A function f(n) is said to be Ω(g(n)) if there exist positive constants c and n₀ such that f(n) ≥ c \* g(n) for all n ≥ n₀.
* **Example:** If an algorithm's running time is Ω(n²), it means that the time complexity will grow at least as fast as n².

### **Theta Notation (Θ)**

* **Tight bound:** Represents both the upper and lower bounds of a function.
* **Meaning:** A function f(n) is said to be Θ(g(n)) if there exist positive constants c₁, c₂, and n₀ such that c₁ \* g(n) ≤ f(n) ≤ c₂ \* g(n) for all n ≥ n₀.
* **Example:** If an algorithm's running time is Θ(n²), it means that the time complexity will grow exactly as fast as n².

### **Big-O Notation (O)**

* **Upper bound:** Represents the maximum growth rate of a function.
* **Meaning:** A function f(n) is said to be O(g(n)) if there exist positive constants c and n₀ such that f(n) ≤ c \* g(n) for all n ≥ n₀.
* **Example:** If an algorithm's running time is O(n²), it means that the time complexity will grow no faster than n².

### **Comparison Table**

|  |  |  |
| --- | --- | --- |
| **Notation** | **Meaning** | **Example** |
| **Ω(g(n))** | f(n) grows at least as fast as g(n) | f(n) = n² + 3n is Ω(n²) |
| **Θ(g(n))** | f(n) grows exactly as fast as g(n) | f(n) = 2n² + 5n is Θ(n²) |
| **O(g(n))** | f(n) grows no faster than g(n) | f(n) = n² + 3n is O(n²) |

### **Key Points**

* **Ω** provides a lower bound on the running time.
* **Θ** provides a tight bound, indicating the exact growth rate.
* **O** provides an upper bound on the running time.
* In practice, **O** is often used to analyze algorithms, as it gives a worst-case scenario.

**Example:** Consider a sorting algorithm with the following time complexity:

* Best case: Θ(n)
* Average case: Θ(n log n)
* Worst case: O(n²)

**4.Explain in detail about Asymptotic Notations**

## **Asymptotic Notations: A Clear and Concise Explanation**

**Asymptotic notations** are mathematical tools used to describe the limiting behavior of functions. In computer science, they are primarily employed to analyze the efficiency of algorithms, particularly in terms of time complexity and space complexity.

### **Big O Notation (O)**

* **Definition:** A function f(n) is said to be O(g(n)) if there exist positive constants c and n₀ such that f(n) ≤ c \* g(n) for all n ≥ n₀.
* **Meaning:** It provides an **upper bound** on the growth rate of f(n). In simpler terms, f(n) grows no faster than g(n) for large values of n.
* **Example:** If an algorithm's time complexity is O(n²), it means that the running time will not exceed a constant multiple of n² for large input sizes.

### **Big Omega Notation (Ω)**

* **Definition:** A function f(n) is said to be Ω(g(n)) if there exist positive constants c and n₀ such that f(n) ≥ c \* g(n) for all n ≥ n₀.
* **Meaning:** It provides a **lower bound** on the growth rate of f(n). In simpler terms, f(n) grows at least as fast as g(n) for large values of n.
* **Example:** If an algorithm's time complexity is Ω(n²), it means that the running time will be at least a constant multiple of n² for large input sizes.

### **Theta Notation (Θ)**

* **Definition:** A function f(n) is said to be Θ(g(n)) if there exist positive constants c₁, c₂, and n₀ such that c₁ \* g(n) ≤ f(n) ≤ c₂ \* g(n) for all n ≥ n₀.
* **Meaning:** It provides a **tight bound** on the growth rate of f(n). In simpler terms, f(n) grows exactly as fast as g(n) for large values of n.
* **Example:** If an algorithm's time complexity is Θ(n²), it means that the running time will grow exactly as fast as n² for large input sizes.

### **Common Growth Rates**

* **Constant time:** O(1)
* **Logarithmic time:** O(log n)
* **Linear time:** O(n)
* **Linearithmic time:** O(n log n)
* **Quadratic time:** O(n²)
* **Cubic time:** O(n³)
* **Exponential time:** O(2^n)
* **Factorial time:** O(n!)

### **Why Use Asymptotic Notation?**

* **Simplifies analysis:** It allows us to focus on the overall behavior of an algorithm without getting bogged down in specific implementation details.
* **Compares algorithms:** It helps us compare the efficiency of different algorithms for a given problem.
* **Predicts performance:** It gives us an estimate of how an algorithm will perform for large input sizes.

**.5.Calculate and explain the time complexity of the following code:**

* 1. **int i, j, k = 0;**
  2. **for (i = n / 2; i <= n; i++) {**
  3. **for (j = 2; j <= n; j = j \* 2) {**
  4. **k = k + n / 2; }**
  5. **}**

**Outer Loop Analysis:**

* The outer loop iterates from n/2 to n, which is a total of n - (n/2) + 1 = n/2 + 1 iterations.

**Inner Loop Analysis:**

* The inner loop iterates from 2 to n, doubling j in each iteration. This is a geometric progression.

To find the number of iterations, we can use the formula for the sum of a geometric series:  
sum = a \* (r^n - 1) / (r - 1)

* where:
  + a is the first term (2)
  + r is the common ratio (2)
  + n is the number of terms
  + We want to find the smallest n such that sum >= n.

Solving for n, we get:  
n >= log2(n + 1)

* + Since n is an integer, we can round up the result to the nearest integer.

**Time Complexity:**

The total time complexity is the product of the iterations in both loops:  
T(n) = (n/2 + 1) \* log2(n + 1)

Using the properties of logarithms and big O notation, we can simplify this to:  
T(n) = O(n \* log n)

**Explanation:**

* The code has two nested loops.
* The outer loop iterates linearly with respect to n.
* The inner loop iterates logarithmically with respect to n.
* The overall time complexity is the product of these two factors, resulting in O(n \* log n).
* This means that the code's running time grows proportionally to n multiplied by the logarithm of n.

**6.Discuss how single and double rotations performed in AVL trees**

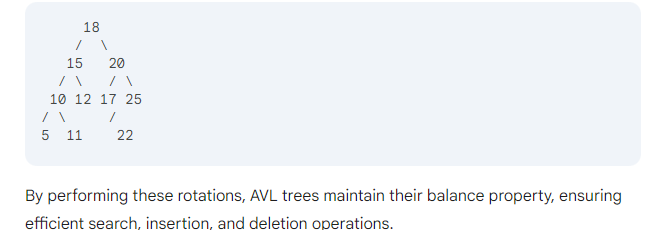
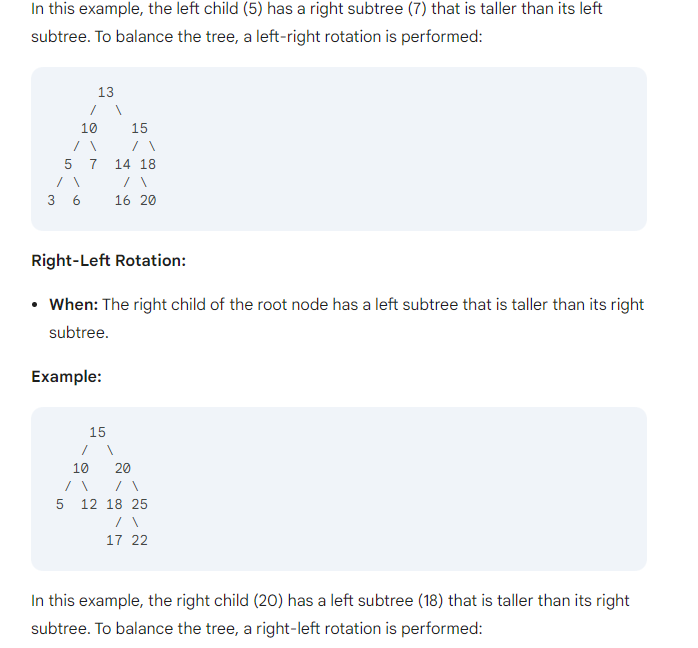
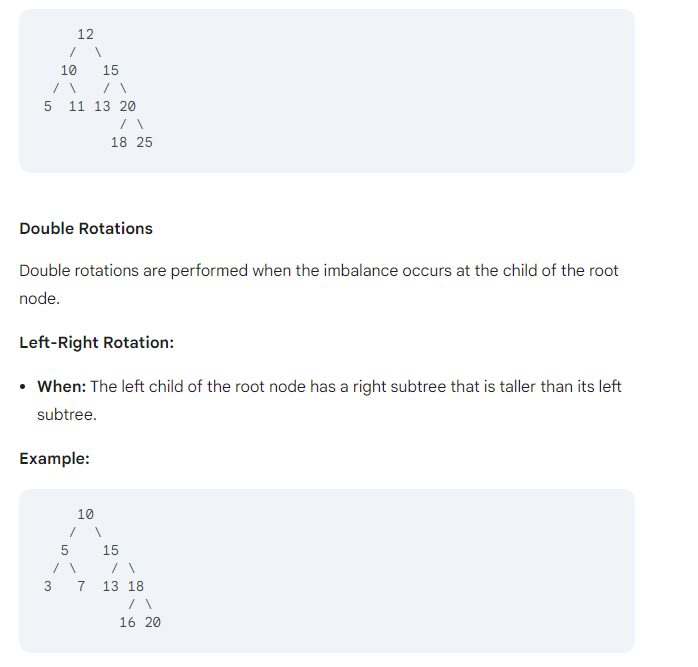
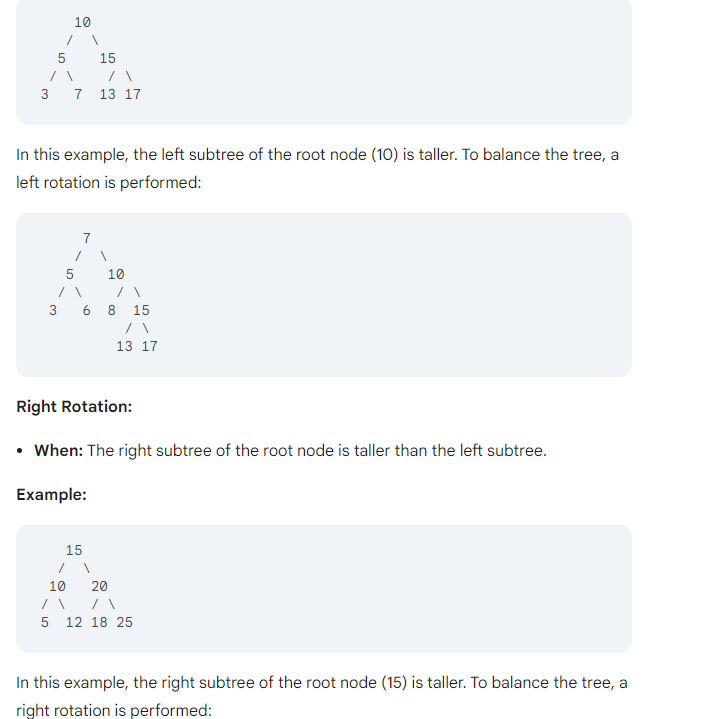
## **Single and Double Rotations in AVL Trees: A Deeper Dive with Examples**

**AVL trees** are self-balancing binary search trees that ensure the height difference between the left and right subtrees of any node is at most one. To maintain this balance property, AVL trees use rotations.

### **Single Rotations**

**Left Rotation:**

* **When:** The left subtree of the root node is taller than the right subtree.

**Example:**

### **Visual Representation**

### **Example**

Consider an AVL tree with the following nodes: 10, 20, 30, 40, 50.

* **Initial state:** The tree is unbalanced after inserting 50.
* **Rebalancing:**
  1. Perform a right-left rotation on the node with value 20.
  2. The tree becomes balanced.

### **Complexity**

The time complexity of a single or double rotation is O(1) as it involves a constant number of operations.

**7.Discuss about the deletion operation in B-Tree**

## **Deletion Operation in B-Trees: A Deeper Dive with Examples**

**B-trees** are self-balancing search trees that are optimized for disk I/O. They are widely used in databases and file systems due to their efficient performance for large datasets.

### **Deletion Operation**

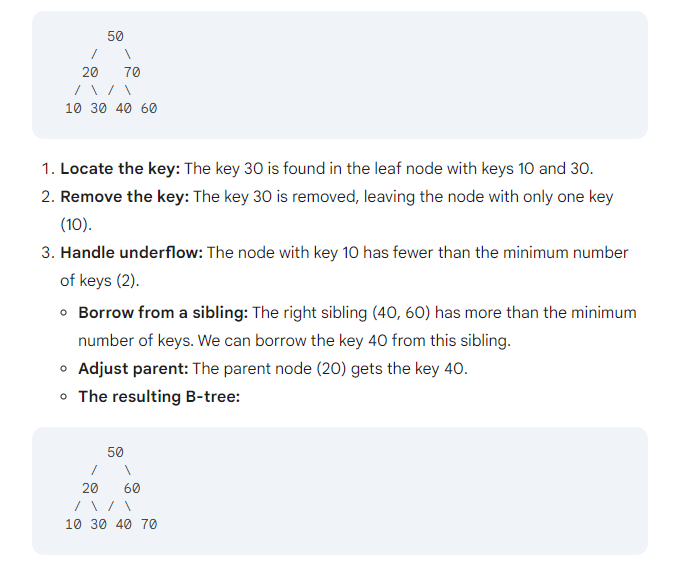
The deletion operation in a B-tree is slightly more complex than insertion due to the potential for underflow. Underflow occurs when a node has fewer than the minimum allowed number of keys.

**Steps involved in deleting a key from a B-tree:**

1. **Locate the Key:** Find the leaf node containing the key to be deleted.
2. **Remove the Key:** If the leaf node has more than the minimum number of keys, simply remove the key.
3. **Handle Underflow:** If the leaf node now has fewer than the minimum number of keys, perform one of the following:
   * **Borrow from a sibling:** If a sibling node has more than the minimum number of keys, borrow a key from it and adjust the parent node's key accordingly.
   * **Merge with a sibling:** If both siblings have the minimum number of keys, merge the node with one of its siblings. This creates a new node with the combined keys of the two merged nodes, and the parent node's key is adjusted accordingly.
   * **Propagate underflow:** If the parent node also has fewer than the minimum number of keys, recursively apply the above steps to the parent node.

**Example:**

Consider a B-tree with a minimum degree of 3. We want to delete the key 30.



**Key points to remember:**

* The deletion operation in B-trees ensures that the tree remains balanced and efficient for search, insertion, and deletion operations.
* Underflow handling is crucial to maintain the tree's structure.
* Borrowing from siblings and merging with siblings are common strategies to handle underflow.
* In some cases, the underflow may propagate up the tree, requiring recursive handling.

**8.Explain the insert and delete operations of AVL tree**

**Insertion and Deletion Operations in AVL Trees: A Deeper Dive with Examples**

## **AVL trees** are self-balancing binary search trees that ensure the height difference between the left and right subtrees of any node is at most one. To maintain this balance property, AVL trees use rotations.

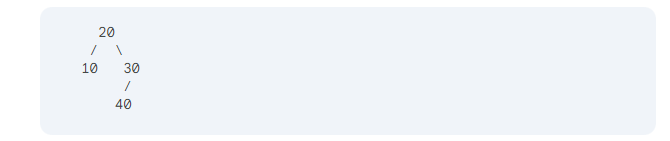
### **Insertion Operation**

1. **Insert the key:** Insert the key into the tree as in a regular binary search tree.
2. **Check for balance:** Check if the height difference between the left and right subtrees of any node has become greater than one.
3. **Rebalance:** If necessary, perform rotations to restore the balance property.

**Example:**

Consider an AVL tree with the following nodes: 10, 20, 30.

* **Insert 40:** The tree becomes unbalanced.
* **Rebalance:** Perform a right rotation on the node with value 20.
* **The resulting balanced tree:**

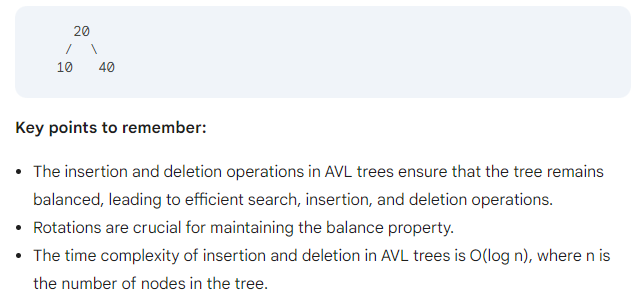
**Deletion Operation**

1. **Locate the key:** Find the node containing the key to be deleted.
2. **Remove the key:** If the node has two children, find its successor (the smallest key in the right subtree) or predecessor (the largest key in the left subtree) and swap it with the node to be deleted.
3. **Rebalance:** Check if the height difference between the left and right subtrees of any node has become greater than one. Perform rotations to restore the balance property.

**Example:**

Consider an AVL tree with the following nodes: 10, 20, 30, 40.

* **Delete 30:** The tree becomes unbalanced.
* **Rebalance:** Perform a left rotation on the node with value 20.
* **The resulting balanced tree:**



**9.Describe about the operations of AVL Tree**

## **Operations of AVL Trees**

AVL trees are self-balancing binary search trees that ensure the height difference between the left and right subtrees of any node is at most one. This balance property is maintained through rotations.

### **Search Operation**

* **Iterative Search:**
  + Start at the root node.
  + If the target key is less than the current node's key, move to the left child.
  + If the target key is greater than the current node's key, move to the right child.
  + Repeat until the target key is found or the current node is null.
* **Recursive Search:**
  + If the current node is null, return null.
  + If the target key is equal to the current node's key, return the current node.
  + If the target key is less than the current node's key, recursively search the left subtree.
  + If the target key is greater than the current node's key, recursively search the right subtree.

**Example:**

Consider an AVL tree with the following nodes: 10, 20, 30, 40.

* **Search for 30:**
  + Start at the root node (20).
  + Since 30 is greater than 20, move to the right child (30).
  + The target key is found.

### **Insertion Operation**

1. **Insert the key:** Insert the key into the tree as in a regular binary search tree.
2. **Check for balance:** Check if the height difference between the left and right subtrees of any node has become greater than one.
3. **Rebalance:** If necessary, perform rotations to restore the balance property.

**Example:**

Consider an AVL tree with the following nodes: 10, 20, 30.

* **Insert 40:** The tree becomes unbalanced.
* **Rebalance:** Perform a right rotation on the node with value 20.
* **The resulting balanced tree:**



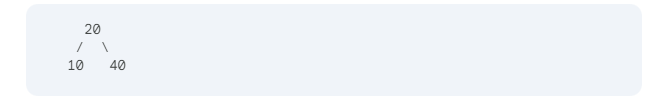
### **Deletion Operation**

1. **Locate the key:** Find the node containing the key to be deleted.
2. **Remove the key:** If the node has two children, find its successor (the smallest key in the right subtree) or predecessor (the largest key in the left subtree) and swap it with the node to be deleted.
3. **Rebalance:** Check if the height difference between the left and right subtrees of any node has become greater than one. Perform rotations to restore the balance property.

**Example:**

Consider an AVL tree with the following nodes: 10, 20, 30, 40.

* **Delete 30:** The tree becomes unbalanced.
* **Rebalance:** Perform a left rotation on the node with value 20.
* **The resulting balanced tree:**



### **Time Complexity**

* **Search:** O(log n)
* **Insertion:** O(log n)
* **Deletion:** O(log n)

**Key points to remember:**

* AVL trees are self-balancing binary search trees.
* Rotations are used to maintain the balance property.
* The time complexity of operations in AVL trees is logarithmic.

**10.Create an AVL tree using the following data entered as a sequential set. Show the balance factors in the resulting tree: 7, 10, 14, 23, 33, 56, 66, 70, 80.**

**11. Explain about the operations of B-Tree**

**B-trees** are self-balancing tree data structures that are widely used to efficiently store and retrieve data. They are particularly well-suited for applications that require frequent insertions, deletions, and searches of large datasets.

### **Key Characteristics of B-Trees:**

* **Order:** Each node in a B-tree has a maximum number of children, called the order of the tree.
* **Keys:** Each internal node contains a set of keys that separate the ranges of values stored in its children.
* **Children:** Each internal node has one more child than the number of keys it contains.
* **Leaf Nodes:** All leaf nodes are at the same depth, and they contain data records.

### **Operations on B-Trees:**

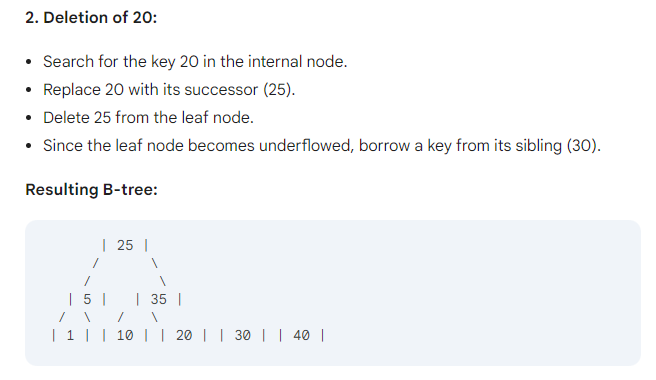
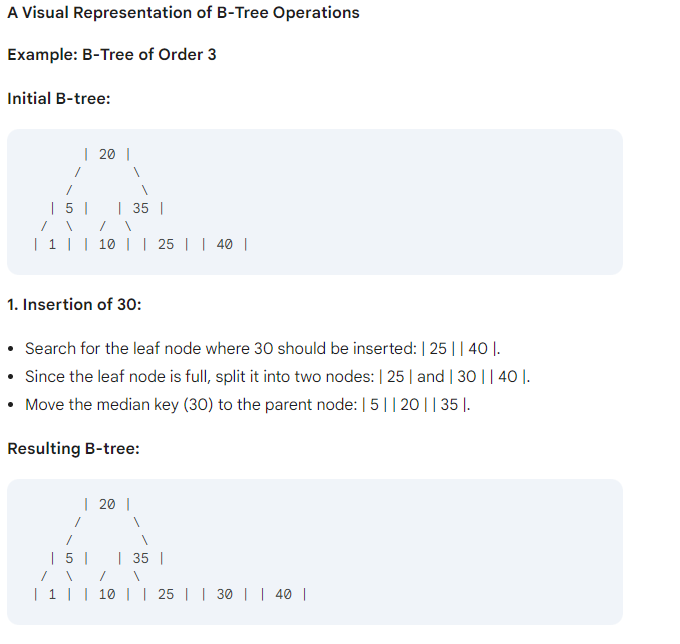
1. **Search:**
   * Start at the root node.
   * Compare the search key with the keys in the current node.
   * If the search key is found, return the corresponding data record.
   * If the search key is less than the smallest key in the current node, search the leftmost child.
   * If the search key is greater than the largest key in the current node, search the rightmost child.
   * Repeat the process until the search key is found or the end of the tree is reached.
2. **Insertion:**
   * Search for the appropriate leaf node where the new key should be inserted.
   * If there is enough space in the leaf node, insert the key and its associated data.
   * If the leaf node is full, split it into two nodes and redistribute the keys and data.
   * If the parent node is also full, split it and redistribute the keys.
   * Continue this process until the root node is reached.
3. **Deletion:**
   * Search for the key to be deleted.
   * If the key is found in a leaf node, delete it.
   * If the key is found in an internal node, replace it with its predecessor or successor.
   * If the node becomes underflowed (has fewer than the minimum number of keys), borrow or merge with a sibling node.
   * If the root node becomes underflowed and has only one child, the root becomes the new root.

### **Advantages of B-Trees:**

* **Efficient Searches:** B-trees are well-balanced, ensuring efficient searches even for large datasets.
* **Efficient Insertions and Deletions:** The self-balancing property of B-trees allows for efficient insertions and deletions.
* **Space Efficient:** B-trees minimize the number of disk accesses required, making them suitable for large databases.

### **Applications of B-Trees:**

* **Database Systems:** B-trees are commonly used in database systems to efficiently store and retrieve data.
* **File Systems:** Some file systems use B-trees to manage directory structures.
* **Indexing:** B-trees can be used to create indexes for large datasets, improving search performance.



**12. Compare and contrast AVL and B Tree?**

|  |  |  |
| --- | --- | --- |
| **Feature** | **AVL Trees** | **B-Trees** |
| **Balancing** | Height difference between subtrees | Splitting and merging of nodes |
| **Efficiency** | Excellent for smaller datasets | Optimized for large datasets and disk I/O |
| **Height** | Guaranteed to be logarithmic | Logarithmic, but depends on order and dataset size |
| **Use Cases** | Frequent insertions/deletions, smaller datasets | Large datasets, database systems, file systems |
| **Disk I/O** | Not optimized for disk I/O | Optimized for disk I/O |
| **Space Usage** | Generally less space than B-trees | May require more space due to larger nodes |
| **Implementation Complexity** | More complex due to rotations | Less complex due to simpler operations |
| **Typical Applications** | In-memory data structures, smaller databases | Large databases, file systems, indexing |

**13.Give an analysis of the B-Tree insertion process.**

## **Analysis of B-Tree Insertion**

The B-tree insertion process involves several steps to maintain the tree's balanced structure and efficient search capabilities. Here's a detailed analysis:

### **1. Search for the Appropriate Leaf Node:**

* The insertion process begins by searching for the leaf node where the new key should be inserted. This search is typically done using a top-down traversal of the tree.

### **2. Insert the Key:**

* If there is enough space in the leaf node, the new key and its associated data are simply inserted into the appropriate position.
* However, if the leaf node is full, a split operation is necessary.

### **3. Split the Leaf Node:**

* When a leaf node becomes full, it is split into two new leaf nodes.
* The median key of the original leaf node is moved up to the parent node.
* The remaining keys and data are distributed between the two new leaf nodes.

### **4. Propagate Splits Up the Tree:**

* If the parent node is also full after the split, it must be split as well. This process continues up the tree until a node with enough space is found.
* In the worst case, this can result in a series of splits, potentially reaching the root node.

### **5. Adjust Pointers:**

* As nodes are split and new nodes are created, the pointers between nodes need to be adjusted to maintain the tree's structure.

### **Analysis of Time Complexity:**

* The time complexity of B-tree insertion is logarithmic in the height of the tree.
* The height of a B-tree is typically much smaller than the number of nodes, especially for large datasets.
* Therefore, B-tree insertions are generally very efficient.

### **Key Factors Affecting Insertion Performance:**

* **Order of the Tree:** The order of a B-tree determines the maximum number of children a node can have. A higher order can reduce the number of levels in the tree, leading to faster insertions.
* **Fill Factor:** The fill factor is the percentage of keys that are actually stored in a node. A higher fill factor can reduce the frequency of splits, improving performance.
* **Disk I/O:** For large datasets stored on disk, the number of disk I/O operations can significantly impact performance. B-trees are designed to minimize disk I/O by storing multiple keys in a single node.

**UNIT-2**

1. **Draw and explain a MAX heap with example?**

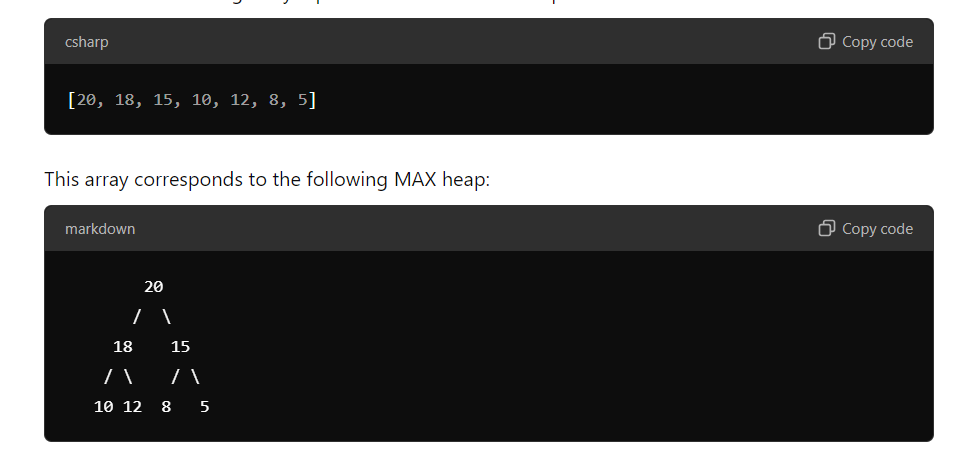
A **MAX heap** is a complete binary tree where the value of each node is greater than or equal to the values of its children. This property makes it useful for implementing priority queues, where the highest priority element (the largest value) can be accessed quickly.

### **Properties of a MAX Heap:**

1. **Complete Binary Tree**: All levels of the tree are fully filled except possibly for the last level, which is filled from left to right.
2. **Heap Property**: The value of each parent node is greater than or equal to the values of its child nodes.

### **Example of a MAX Heap**

Consider the following array representation of a MAX heap:



### **Explanation of the Example:**

1. **Root Node**: The root node is 20, which is the largest element in the heap.
2. **Children**:
   * The left child of 20 is 18 and the right child is 15.
   * For the node 18, its left child is 10 and its right child is 12.
   * For the node 15, its left child is 8 and its right child is 5.
3. **Heap Property**:
   * All parent nodes are greater than their children:
     + 20 > 18, 20 > 15
     + 18 > 10, 18 > 12
     + 15 > 8, 15 > 5

### **Operations on a MAX Heap:**

1. **Insertion**: When inserting a new element, it's added at the end of the heap (maintaining the complete tree property) and then "bubbled up" to maintain the heap property.
2. **Deletion (Extract Max)**: The maximum element (root) is removed, replaced with the last element, and then "bubbled down" to restore the heap property.

2.Describe about Graphs and their Representations

## **Graphs and Their Representations**

**Graphs** are mathematical structures used to represent relationships between objects. They consist of:

* **Vertices:** The objects or nodes.
* **Edges:** The connections between the vertices.

**Types of Graphs:**

* **Directed Graphs:** Edges have a direction (e.g., one-way streets).
* **Undirected Graphs:** Edges have no direction (e.g., two-way streets).
* **Weighted Graphs:** Edges have associated values (e.g., distances, costs).
* **Simple Graphs:** No multiple edges between the same pair of vertices and no self-loops.

**Representations of Graphs:**

1. **Adjacency Matrix:**
   * A 2D array where rows and columns represent vertices.
   * The value at A[i][j] indicates the weight of the edge from vertex i to vertex j.
   * For undirected graphs, it's symmetric.
   * Space complexity: O(V^2).
2. **Adjacency List:**
   * A list where each element is a list of adjacent vertices.
   * For undirected graphs, each edge is represented twice.
   * Space complexity: O(V + E).
3. **Incidence Matrix:**
   * A 2D array where rows represent vertices and columns represent edges.
   * The value at A[i][j] is 1 if vertex i is incident to edge j, 0 otherwise.
   * Space complexity: O(VE).

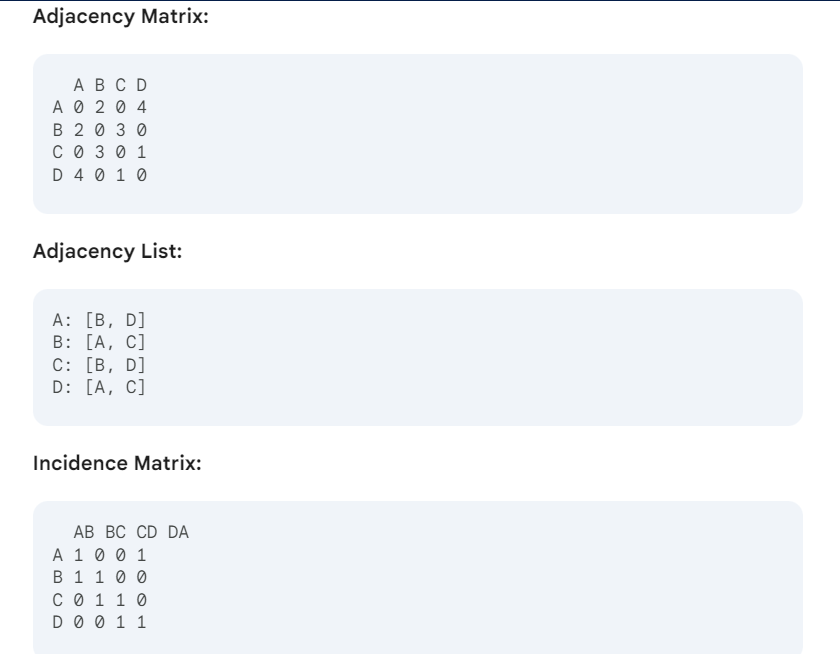
**Choosing the Right Representation:**

* **Adjacency Matrix:** Good for dense graphs (many edges) and operations like finding the degree of a vertex or checking if an edge exists.
* **Adjacency List:** Good for sparse graphs (few edges) and operations like traversing the graph (e.g., BFS, DFS).
* **Incidence Matrix:** Used less frequently, but useful for analyzing the structure of the graph.

**Example:**

Consider a graph with 4 vertices (A, B, C, D) and the following edges:

* A-B (weight 2)
* B-C (weight 3)
* C-D (weight 1)
* D-A (weight 4)



**3.Discuss about priority queues operations and its applications.**

A priority queue is a special type of queue where each element is associated with a priority. Elements are dequeued in order of their priority, with the highest-priority element being dequeued first. This makes priority queues useful for applications where elements need to be processed based on their importance or urgency.

### **Key Operations**

### **1. Insertion:**

* **Purpose:** Adds a new element to the queue with a specified priority.
* **Time complexity:**
  + **Binary heap:** O(log n)
  + **Fibonacci heap:** O(1) amortized

### **2. Deletion (or Extract-Max/Extract-Min):**

* **Purpose:** Removes and returns the highest (or lowest) priority element from the queue.
* **Time complexity:**
  + **Binary heap:** O(log n)
  + **Fibonacci heap:** O(log n) amortized

### **3. Peek (or Find-Max/Find-Min):**

* **Purpose:** Returns the highest (or lowest) priority element without removing it.
* **Time complexity:**
  + **Binary heap:** O(1)
  + **Fibonacci heap:** O(1)

### **4. Decrease-Key (or Increase-Key):**

* **Purpose:** Reduces (or increases) the priority of a given element.
* **Time complexity:**
  + **Binary heap:** O(log n)
  + **Fibonacci heap:** O(1) amortized

### **5. Delete:**

* **Purpose:** Removes a specific element from the queue.
* **Time complexity:**
  + **Binary heap:** O(log n)
  + **Fibonacci heap:** O(log n) amortized

### **Common Implementations**

1. **Binary Heap:** A complete binary tree where the parent node's key is always greater than or equal to (in a max heap) or less than or equal to (in a min heap) its children's keys.
2. **Fibonacci Heap:** A more complex data structure that offers better amortized time for decrease-key operations, making it suitable for algorithms like Dijkstra's shortest path algorithm.

### **Applications**

Priority queues are used in a wide range of applications, including:

* **Scheduling:** Task scheduling systems often use priority queues to determine the order in which tasks are executed based on their importance or deadlines.
* **Graph Algorithms:** Dijkstra's algorithm for finding the shortest path in a graph uses a priority queue to efficiently select the next vertex to explore.
* **Data Compression:** Huffman coding uses a priority queue to determine the optimal Huffman tree for compressing data.
* **Simulation:** Priority queues are used in event-driven simulations to handle events in the order in which they occur.
* **Operating Systems:** Process scheduling and memory management in operating systems often rely on priority queues to determine the order in which processes are executed or pages are swapped.

**4.Explain in detail about Connected Components and Biconnected Components**

### **Connected Components**

#### **Definition**

A connected component in an undirected graph is a subgraph in which any two vertices are connected by a path, and which is connected to no additional vertices in the graph. In simpler terms, it is a maximally connected subgraph, meaning no more vertices can be added to this subgraph without disconnecting it.

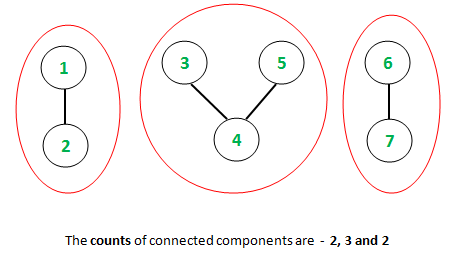
In a graph, if every node is reachable from every other node, the graph is said to be connected. If a graph is not connected, it can be broken down into several connected components, each of which is a subgraph where:

* All nodes are reachable from any other node in the same component.
* There is no path between any two nodes belonging to different components.

#### **Properties**

* Maximality: A connected component is a maximal set of nodes such that there is a path between any two nodes in the set.
* Disjoint: Connected components are disjoint sets of nodes; any node belongs to exactly one component.
* Graph Traversal: Algorithms like Depth First Search (DFS) or Breadth First Search (BFS) are commonly used to find all connected components.

#### **Example**

****

* There are two connected components in the graph:
  1. Component 1: {1, 2, 3, 4}
  2. Component 2: {5, 6}

In this graph, nodes {1, 2, 3, 4} are connected to each other, and nodes {5, 6} are connected to each other. However, there is no path between any node in {1, 2, 3, 4} and any node in {5, 6}.

#### **Algorithm to Find Connected Components**

* Depth First Search (DFS) or Breadth First Search (BFS) can be used to find connected components.
* Visit each unvisited node, and mark all reachable nodes as part of the same connected component.

**Steps:**

1. Start from any unvisited node.
2. Perform DFS or BFS from that node, marking all visited nodes.
3. All marked nodes belong to the same connected component.
4. Repeat for any unvisited nodes.

### **Biconnected Components (BCC)**

#### **Definition**

A **biconnected component** (also known as a **2-connected component** or **block**) is a maximal subgraph in which any two vertices are connected by at least two disjoint paths (ignoring the direction of edges). In other words, the graph remains connected even if any single vertex is removed.

A graph is **biconnected** if:

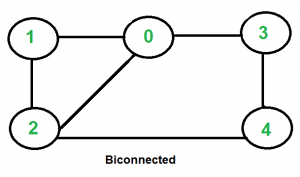
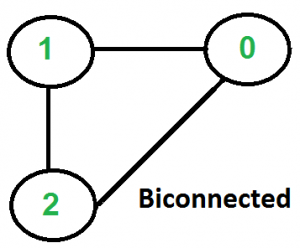
* It is connected.
* It has no **articulation points** (or cut vertices), i.e., removing any single vertex does not disconnect the graph.

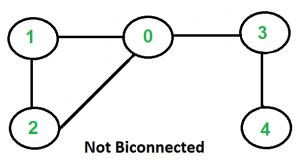
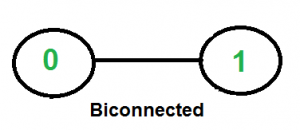
A **biconnected component** is a portion of a graph that does not contain an articulation point (removing any vertex from the component does not break the component into multiple pieces).

#### **Properties**

* **Articulation Points**: An articulation point (or cut vertex) is a vertex that, when removed, increases the number of connected components in the graph. Biconnected components are identified by the absence of articulation points.
* **Bridge**: A bridge (or cut edge) is an edge whose removal disconnects the graph. Biconnected components cannot contain a bridge within them.
* A single connected component can be composed of several biconnected components.

#### **Example :**





* **Articulation points**: {4} is an articulation point. Removing node 4 breaks the graph into two disconnected parts.
* **Biconnected components**:
  1. {1, 2, 3, 4} (removing any single vertex does not break the component)
  2. {4, 5, 6} (removing any single vertex does not break the component)

#### **Algorithm to Find Biconnected Components**

* **DFS-based algorithm** is commonly used to find biconnected components.
* This is done by calculating the **DFS discovery time** and the **low-link values** of each vertex.

**Steps**:

1. Perform a DFS traversal.
2. For each vertex, track the discovery time (the time when it was first visited) and the lowest point reachable from it.
3. Use these values to identify articulation points and hence the biconnected components.

**Low-link value** of a vertex is the minimum discovery time reachable from the vertex or its descendants in the DFS tree.

**5.Compare and contrast the MAX and MIN heap.**

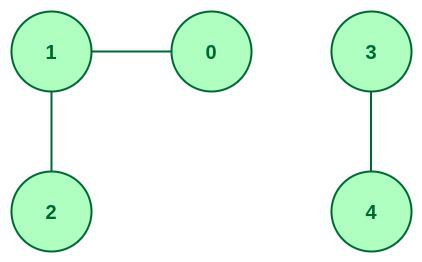
|  |  |  |
| --- | --- | --- |
| **Feature** | **MAX Heap** | **MIN Heap** |
| **Definition** | A complete binary tree where the value of any node is greater than or equal to the values of its children. | A complete binary tree where the value of any node is less than or equal to the values of its children. |
| **Key Operation** | Finding the maximum element (at the root). | Finding the minimum element (at the root). |
| **Common Use Cases** | Priority queues (e.g., Dijkstra's algorithm, Prim's algorithm), sorting (e.g., heapsort). | Scheduling algorithms (e.g., shortest job first), graph algorithms (e.g., Prim's algorithm). |
| **Insertion** | Add the new element to the end of the heap and bubble it up until the heap property is satisfied. | Same as MAX heap, but bubble up based on the MIN heap property. |
| **Deletion** | Remove the root (maximum element), replace it with the last element, and bubble it down until the heap property is satisfied. | Remove the root (minimum element), replace it with the last element, and bubble it down until the heap property is satisfied. |
| **Heapify** | Convert an array into a heap by starting from the last internal node and bubbling down elements until the heap property is satisfied. | Same as MAX heap, but use the MIN heap property for bubbling down. |
| **Time Complexity** | Insertion, deletion, and heapify operations have a time complexity of O(log n). | Same as MAX heap. |
| **Space Complexity** | Both MAX and MIN heaps are typically implemented as arrays, so their space complexity is O(n). | Same as MAX heap. |
| **Implementation Variants** | Binary heaps, d-ary heaps, Fibonacci heaps. | Same as MAX heap. |
| **Applications** | Priority queues, sorting, graph algorithms. | Scheduling, graph algorithms. |

**6.Explain the connected components.**

### **Connected Component in Data Structures and Algorithms (DSA)**

A **connected component** in an undirected graph refers to a set of vertices that are all connected to each other by edges, but not connected to any vertices outside this group.

For example, in the graph below:



* **{0, 1, 2}** form one connected component.
* **{3, 4}** form another connected component.

### **Characteristics of a Connected Component**

* A connected component is a subset of vertices where each vertex is reachable from every other vertex in that subset.
* A graph can have **multiple connected components**.
* **No vertex** in one component is reachable from any vertex in another component.

### **How to Identify Connected Components**

There are several algorithms used to identify connected components in a graph:

1. **Depth-First Search (DFS)**:
   * DFS explores each vertex and its adjacent vertices recursively, marking them as visited to form a connected component.
2. **Breadth-First Search (BFS)**:
   * BFS uses a queue to explore all vertices at the present depth before moving on to vertices at the next depth level, identifying connected components in the process.
3. **Union-Find Algorithm (Disjoint Set Union)**:
   * The Union-Find algorithm helps manage dynamic connectivity by grouping nodes into sets where each set represents a connected component.

### **Applications of Connected Components**

1. **Graph Theory**:
   * Used to find subgraphs or clusters of connected nodes in larger graphs.
2. **Computer Networks**:
   * Helps discover clusters of connected nodes or devices, such as groups with similar bandwidth or network characteristics.
3. **Image Processing**:
   * Used to identify connected regions in binary images, where each pixel in a region is connected to its neighboring pixels.

**By using DFS or BFS, we can efficiently identify the connected components of a graph.**

**7. Explain different types of Graphs with a neat sketch.**

**Types of Graphs with Example**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, A Graph consisting of a finite set of vertices(or nodes) and a set of edges that connect a pair of nodes.

1. **Undirected Graphs:** A graph in which edges have no direction, i.e., the edges do not have arrows indicating the direction of traversal. Example: A social network graph where friendships are not directional.
2. **Directed Graphs:** A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Example: A web page graph where links between pages are directional.
3. **Weighted Graphs:** A graph in which edges have weights or costs associated with them. Example: A road network graph where the weights can represent the distance between two cities.
4. **Unweighted Graphs:** A graph in which edges have no weights or costs associated with them. Example: A social network graph where the edges represent friendships.
5. **Complete Graphs:** A graph in which each vertex is connected to every other vertex. Example: A tournament graph where every player plays against every other player.
6. **Bipartite Graphs:** A graph in which the vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set. Example: A job applicant graph where the vertices can be divided into job applicants and job openings.
7. **Trees:** A connected graph with no cycles. Example: A family tree where each person is connected to their parents.
8. **Cycles:** A graph with at least one cycle. Example: A bike-sharing graph where the cycles represent the routes that the bikes take.
9. **Sparse Graphs:** A graph with relatively few edges compared to the number of vertices. Example: A chemical reaction graph where each vertex represents a chemical compound and each edge represents a reaction between two compounds.
10. **Dense Graphs:** A graph with many edges compared to the number of vertices. Example: A social network graph where each vertex represents a person and each edge represents a friendship.

#### **1. What is a Graph?**

A **graph** is a data structure used to represent relationships between pairs of objects. In simple terms, it consists of:

* **Vertices (or Nodes):** These are the objects or entities in a graph, like cities, people, or computers.
* **Edges:** These are the connections or relationships between the vertices. Edges can be **directed** (one-way) or **undirected** (two-way).

#### **2. Types of Graphs**

1. **Undirected Graph:**
   * In an undirected graph, edges have no direction, meaning the connection between two vertices goes both ways.
   * Example: A road between two cities where you can travel in both directions.
2. **Directed Graph (or Digraph):**
   * In a directed graph, edges have direction, meaning the connection between two vertices goes one way.
   * Example: One-way roads or communication from one person to another.
3. **Weighted Graph:**
   * In a weighted graph, edges have values (weights), which can represent things like distance, time, or cost.
   * Example: A graph representing cities where the edge weights show the distance between cities.

#### **3. Components in Graphs**

A **component** in a graph is a set of vertices that are connected to each other but not connected to vertices outside the set. There are two main types of components:

1. **Connected Graph:**
   * A graph where every vertex is connected to every other vertex, either directly or indirectly, is called a connected graph.
2. **Disconnected Graph:**
   * A graph with multiple **connected components** is called a disconnected graph. Each connected component is a subgraph where every vertex is reachable from every other vertex in that subgraph.

**Example:**

* Graph with vertices {1, 2, 3} all connected forms one connected component.
* Vertices {4, 5} form another connected component.

#### **4. Graph Representation Methods**

There are two primary ways to represent graphs: **Adjacency Matrix** and **Adjacency List**. Both methods have their advantages and are used based on the characteristics of the graph.

##### **a. Adjacency Matrix:**

* An **Adjacency Matrix** is a 2D array where the rows and columns represent vertices. The values in the matrix indicate whether there is an edge between the vertices.
* For **unweighted graphs**, a value of 1 indicates a connection, and 0 indicates no connection.
* For **weighted graphs**, the value in the matrix represents the weight of the edge.

**Example:**

Graph with 3 vertices and edges between (1-2) and (2-3):

Adjacency Matrix:

1 2 3

1 [ 0, 1, 0 ]

2 [ 1, 0, 1 ]

3 [ 0, 1, 0 ]

**Advantages:**

* Easy to implement and use.
* Fast to check if there is an edge between two vertices.

**Disadvantages:**

* Takes up a lot of space, especially if the graph has many vertices and few edges (sparse graphs).

##### **b. Adjacency List:**

* An **Adjacency List** represents a graph as a collection of lists. Each vertex has a list of all the vertices it is connected to.
* This is more memory efficient for **sparse graphs** (graphs with fewer edges compared to the number of vertices).

**Example:**

Graph with 3 vertices and edges between (1-2) and (2-3):

Adjacency List:

1 -> [2]

2 -> [1, 3]

3 -> [2]

**Advantages:**

* Space efficient, especially for large graphs with fewer edges.
* Faster to iterate over the neighbors of a vertex.

**Disadvantages:**

* Slower to check if an edge exists between two vertices, compared to the adjacency matrix.

#### **5. Special Types of Graphs**

1. **Complete Graph:**
   * A graph in which every pair of vertices is connected by an edge.
   * Example: A graph with 3 vertices where each vertex is connected to every other vertex.
2. **Cycle Graph:**
   * A graph where the vertices form a cycle, meaning each vertex is connected to exactly two other vertices.
   * Example: Vertices 1-2-3-1 form a cycle.
3. **Tree:**
   * A special type of graph where there are no cycles, and there is exactly one path between any pair of vertices.
   * Example: A family tree is a common example of a tree graph.

#### **6. Applications of Graphs**

Graphs are used in various real-world applications, such as:

* **Social Networks:** To represent connections between people, where vertices are individuals and edges are friendships or followings.
* **Computer Networks:** Representing how computers are connected to each other through routers or switches.
* **Maps and Navigation:** Cities or locations are vertices, and roads or paths are edges, sometimes with weights representing distances.
* **Recommendation Systems:** Where items are connected based on user preferences or similarities.

### **Summary**

* **Graphs** are structures used to model relationships between objects through vertices (nodes) and edges (connections).
* A graph can have multiple **connected components**, where vertices are connected within components but not to other components.
* Graphs can be represented using either an **Adjacency Matrix** (efficient for dense graphs) or an **Adjacency List** (efficient for sparse graphs).
* Special types of graphs include **complete graphs**, **cycle graphs**, and **trees**.
* Graphs have broad applications in social networks, computer networking, map routing, and more.

**11. Explain the Breadth first search with neat diagram.**

# **Breadth First Search**

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level. A lot of popular graph algorithms like Dijkstra’s shortest path, Kahn’s Algorithm, and Prim’s algorithm are based on BFS. BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweghted graph and many more problems.

## BFS from a Given Source:

The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to the [Breadth-First Traversal of a tree](https://www.geeksforgeeks.org/level-order-tree-traversal/). Like tree, we begin with the given source (in tree, we begin with root) and traverse vertices level by level using a queue data structure. The only catch here is that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

Initialization: Enqueue the given source vertex into a queue and mark it as visited.

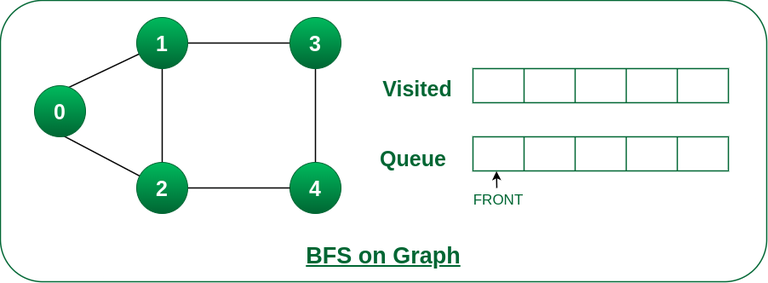
1. Exploration: While the queue is not empty:
   * Dequeue a node from the queue and visit it (e.g., print its value).
   * For each unvisited neighbor of the dequeued node:
     + Enqueue the neighbor into the queue.
     + Mark the neighbor as visited.
2. Termination: Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

### **How Does the BFS Algorithm Work?**

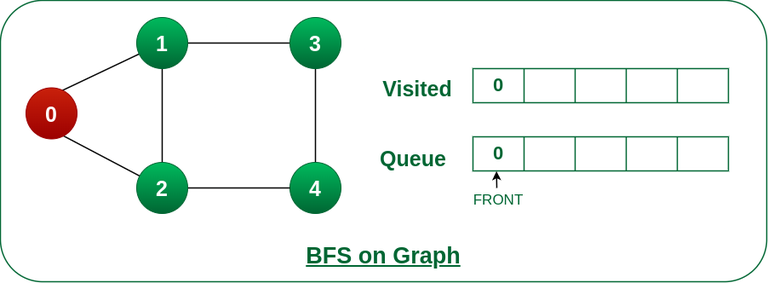
Let us understand the working of the algorithm with the help of the following example where the source vertex is 0.

*Step1: Initially queue and visited arrays are empty.*

**

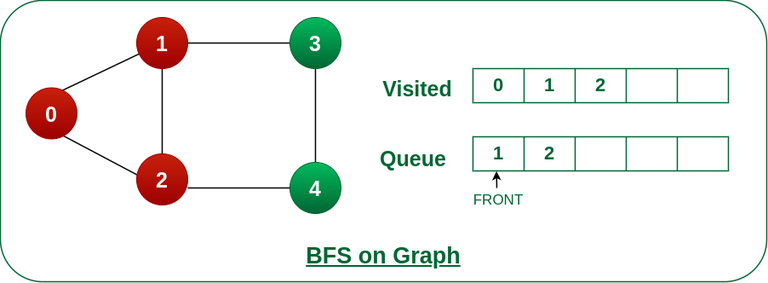
*Queue and visited arrays are empty initially.*

*Step2: Push 0 into queue and mark it visited.*

**

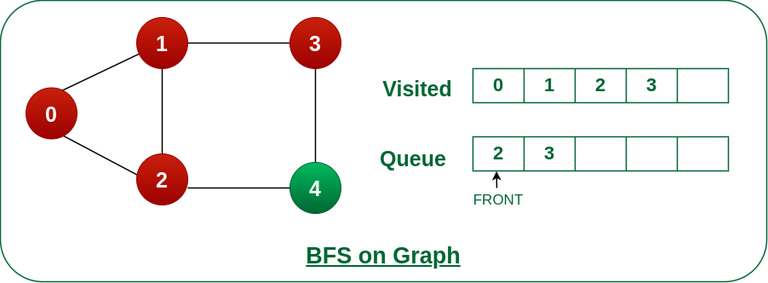
*Push node 0 into queue and mark it visited.*

*Step 3: Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.*

**

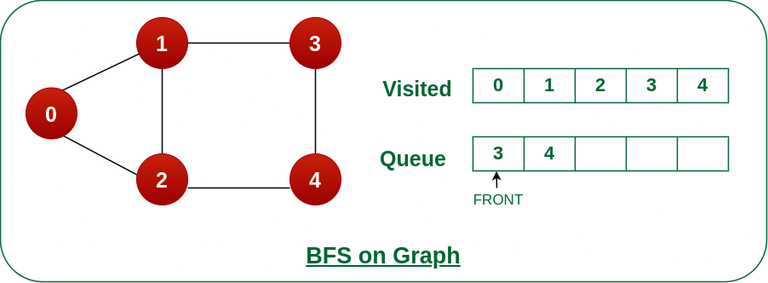
*Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.*

*Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*

**

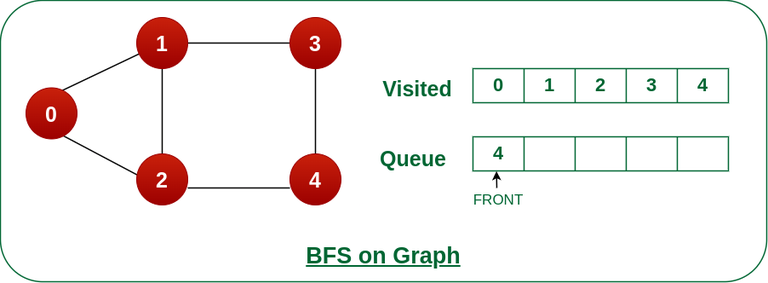
*Remove node 1 from the front of queue and visited the unvisited neighbours and push*

*Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

**

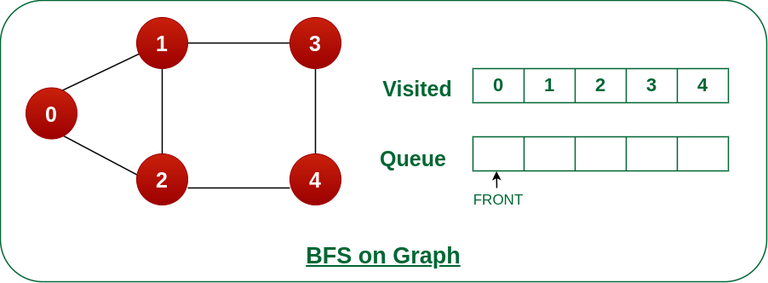
*Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

*Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*

**

*Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

*Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.  
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*

**

*Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

*Now, Queue becomes empty, So, terminate these process of iteration.*

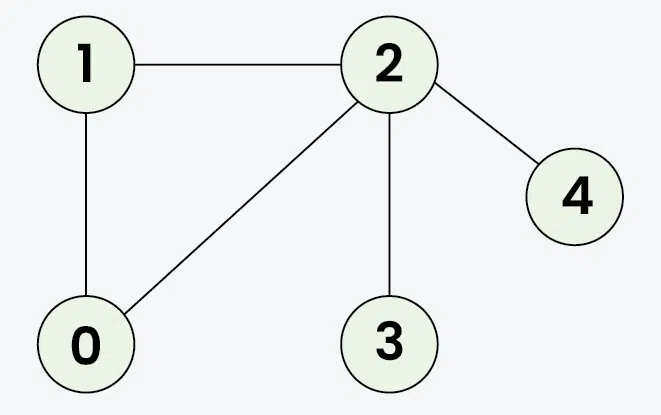
# **12. Explain the Depth first search with neat diagram.**

# **Depth First Search**

**Depth First Traversal (or DFS)** for a graph is similar to [Depth First Traversal of a tree.](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

**Example:**

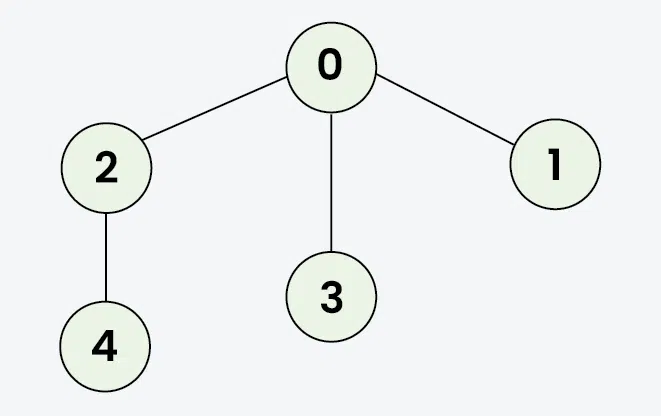
***Input:*** *V = 5, E = 5, edges = {{1, 2}, {1, 0}, {0, 2}, {2, 3}, {2, 4}}, source = 1*

**

***Output:*** *1 2 0 3 4****Explanation:*** *DFS Steps:*

* *Start at 1: Mark as visited. Output: 1*
* *Move to 2: Mark as visited. Output: 2*
* *Move to 0: Mark as visited. Output: 0 (backtrack to 2)*
* *Move to 3: Mark as visited. Output: 3 (backtrack to 2)*
* *Move to 4: Mark as visited. Output: 4 (backtrack to 1)*

***Input:*** *V = 5, E = 4, edges = {{0, 2}, {0, 3}, {0, 1}, {2, 4}}, source = 0*

**

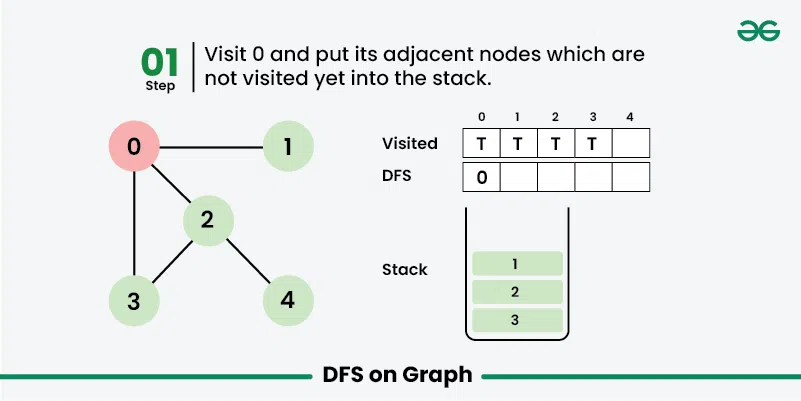
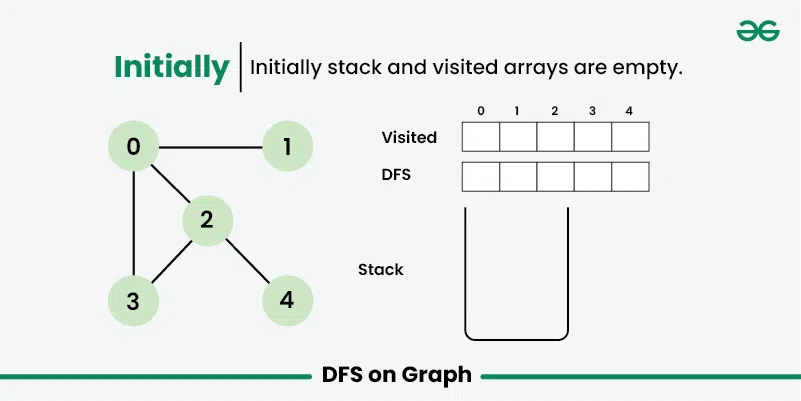
***Output:*** *0 2 4 3 1****Explanation:*** *DFS Steps:*

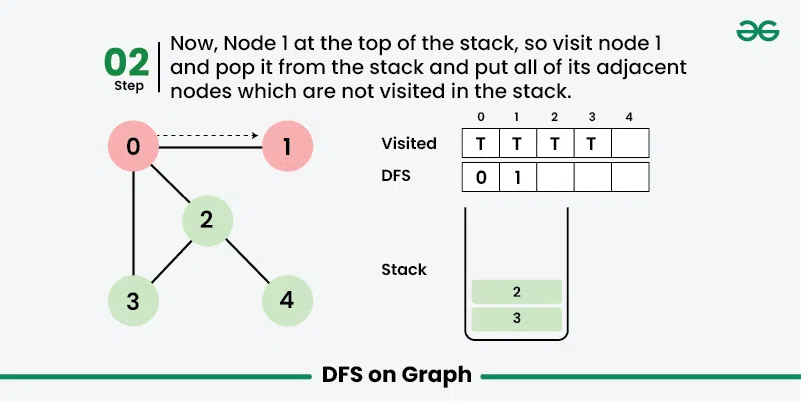
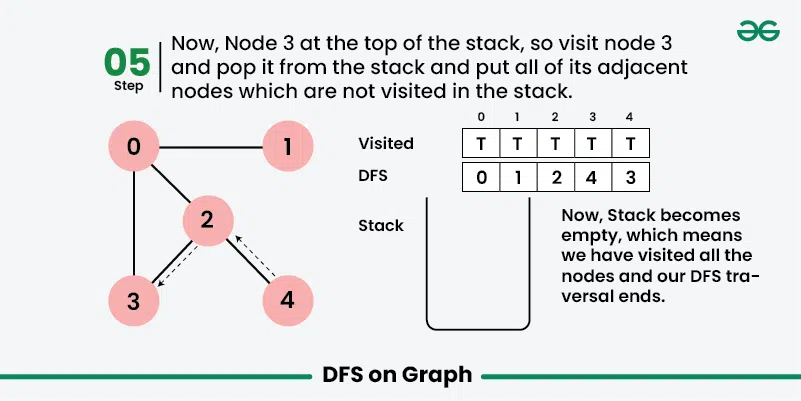
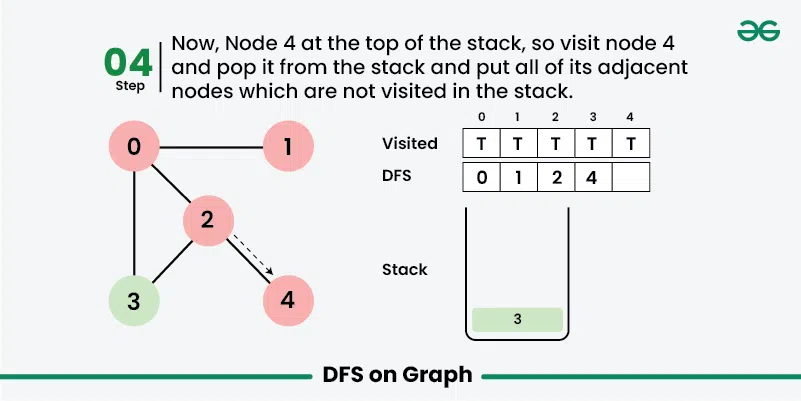
* *Start at 0: Mark as visited. Output: 0*
* *Move to 2: Mark as visited. Output: 2*
* *Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 0)*
* *Move to 3: Mark as visited. Output: 3 (backtrack to 0)*
* *Move to 1: Mark as visited. Output: 1*

## **DFS from a Given Source of Undirected Graph:**

The algorithm starts from a given source and explores all reachable vertices from the given source. It is similar to [Preorder Tree Traversal](https://www.geeksforgeeks.org/preorder-traversal-of-binary-tree/) where we visit the root, then recur for its children. In a graph, there maybe loops. So we use an extra visited array to make sure that we do not process a vertex again.

Let us understand the working of **Depth First Search** with the help of the following illustration: for the **source as 0**.



****

**PART-B**

**UNIT-1**

**1.Define the properties of an Algorithm**

An algorithm should be:

1. **Finite:** It must terminate after a finite number of steps.
2. **Definite:** Each step must be precisely defined.
3. **Effective:** Each step must be executable.
4. **Input:** It must take input.
5. **Output:** It must produce output.
6. **Correct:** It must produce correct results for all valid inputs.

**2.Calculate the big-O notation of 5n2 + n3/2**

To find the big-O notation of 5n^2 + n^(3/2), we need to identify the term that grows the fastest as n approaches infinity.

In this case, n^(3/2) grows faster than 5n^2. Therefore, the big-O notation of the given expression is:

**O(n^(3/2))**

**3.Differentiate Time and Space Complexity**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Time Complexity** | **Space Complexity** |
| **Measures** | Execution time | Memory usage |
| **Focus** | Number of operations | Memory cells |
| **Impact** | Efficiency | Resource consumption |

**4.Define a B-Tree**

A B-Tree is a self-balancing tree data structure that is optimized for disk-based storage. It is widely used in database systems and file systems due to its efficient search, insertion, and deletion operations.

**5.Define a AVL Tree**

An AVL tree is a self-balancing binary search tree that maintains a balanced structure through rotations. It is widely used in various applications due to its efficient search, insertion, and deletion operations.

**6.List the different rotations in AVL Tree.**

**Left Rotation:** A restructuring operation that moves the right child of a node up to become its parent.

**Right Rotation:** A restructuring operation that moves the left child of a node up to become its parent.

**7.Explain the Balancing factor of AVL Tree**

The balancing factor of a node in an AVL tree is the difference in height between its left and right subtrees. It is used to determine if the tree is balanced and if any rotations are necessary to maintain balance.

**8.List the different types of Algorithm notations.**

Big O (upper bound), Big Omega (lower bound), Theta (average case), Little o (strictly slower), Little omega (strictly faster).

**9.Discuss the properties of B-Tree**

B-Trees have a fixed order, store keys in ascending order, have leaf nodes at the same level, and are balanced.

**10.Difference between Tree and Graph.**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Tree** | **Graph** |
| **Structure** | Hierarchical | Arbitrary |
| **Cycles** | No | Allowed |
| **Edges** | Directed (implied) | Directed or undirected |

UNIT-2

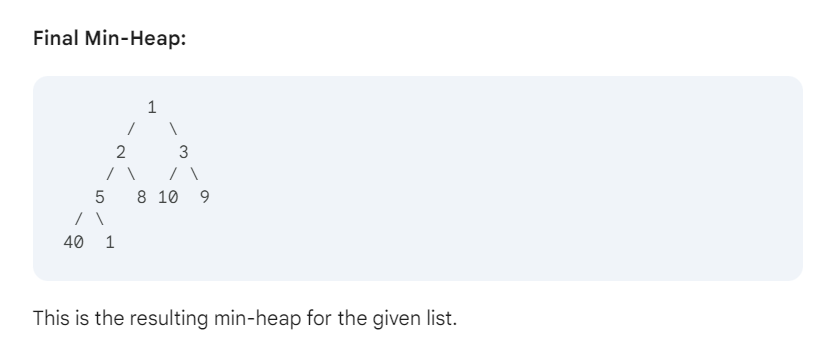
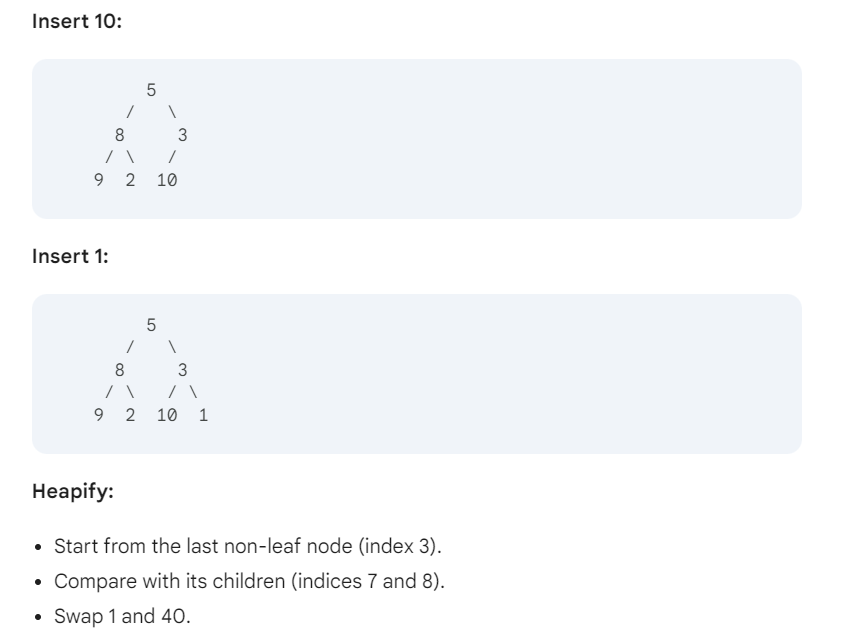
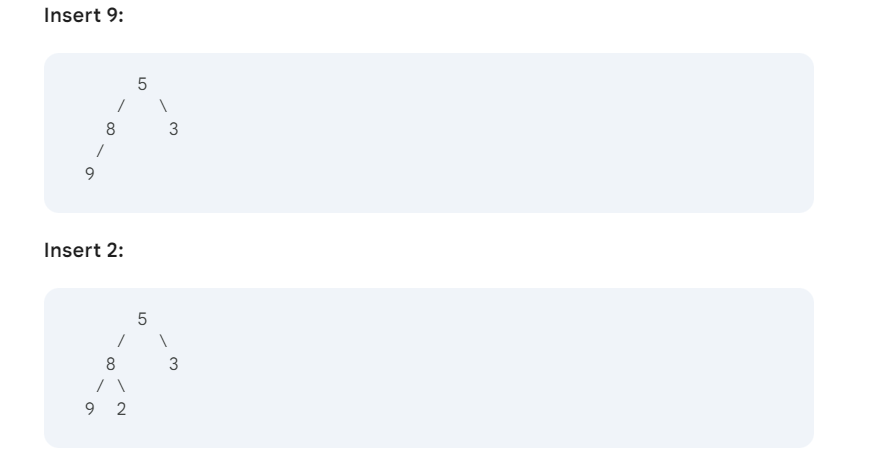
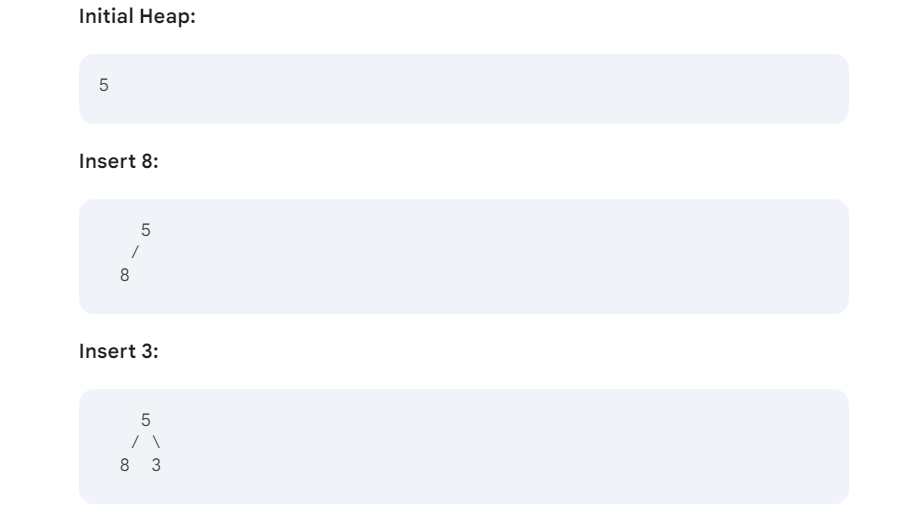
**1.Create a min-heap for the following list**

**(5, 8, 3, 9, 2, 10, 1, 40)**

**Creating a Min-Heap**

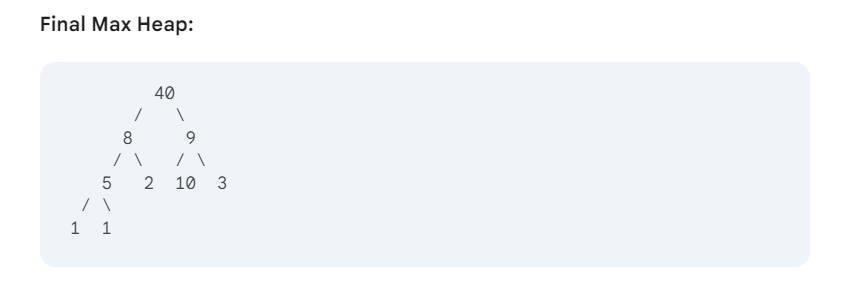
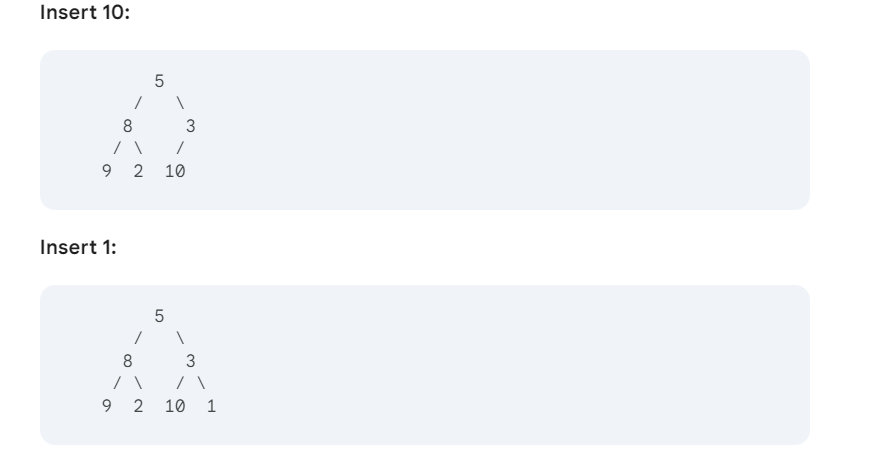
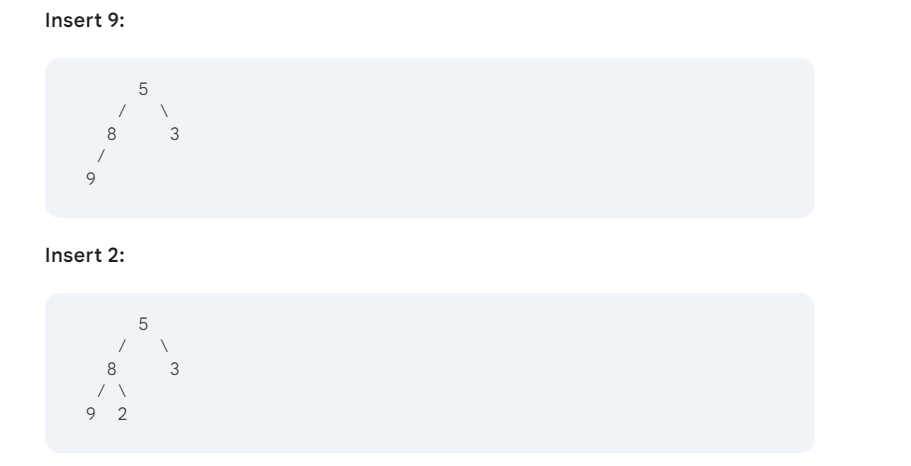
**Min-Heap Property:** In a min-heap, the value of a parent node is always less than or equal to the values of its children.

**Steps:**

1. **Insert elements:** Add the elements to the heap one by one.
2. **Heapify:** After each insertion, ensure the heap property is maintained by comparing the newly inserted element with its parent and swapping if necessary.

**2.How to construct a max heap?**

**Max Heap:** A complete binary tree where the value of each node is greater than or equal to its children. It's constructed by inserting elements and maintaining the heap property through heapification.



**3.Define a Priority Queue.**

**Priority Queue:** A data structure where elements are dequeued based on their priority, not their insertion order. Common implementations include binary heaps and Fibonacci heaps.

**4.Define Components in a Graph.**

**Components in a Graph:** A group of connected nodes that are isolated from the rest of the graph. They can be connected or strongly connected, depending on the type of graph.

**5.Define a Graph with diagram**

A **graph** is a simple way to show connections between things.

* **Vertices (or nodes)** are the points in the graph (like cities or people).
* **Edges** are the lines that connect the points (showing roads between cities or friendships between people).

### **Graph Example: Cities and Roads**

Imagine we have four cities: **A**, **B**, **C**, and **D**. These cities are connected by roads. We can represent this situation using a graph.

### **Graph Representation:**

(A) ---- (B)

| |

| |

(C) ---- (D)

### **Components:**

1. **Vertices (Nodes)**:
   * **A**: Represents City A.
   * **B**: Represents City B.
   * **C**: Represents City C.
   * **D**: Represents City D.
2. **Meaning**: Each vertex represents an entity (in this case, a city) in the graph.
3. **Edges (Connections)**:
   * **(A) -- (B)**: This edge shows there is a road between City A and City B.
   * **(A) -- (C)**: This edge indicates a road between City A and City C.
   * **(B) -- (D)**: This edge represents a road connecting City B and City D.
   * **(C) -- (D)**: This edge shows a road connecting City C and City D.
4. **Meaning**: Edges represent the relationships or connections between the vertices. In this example, they represent roads connecting cities.

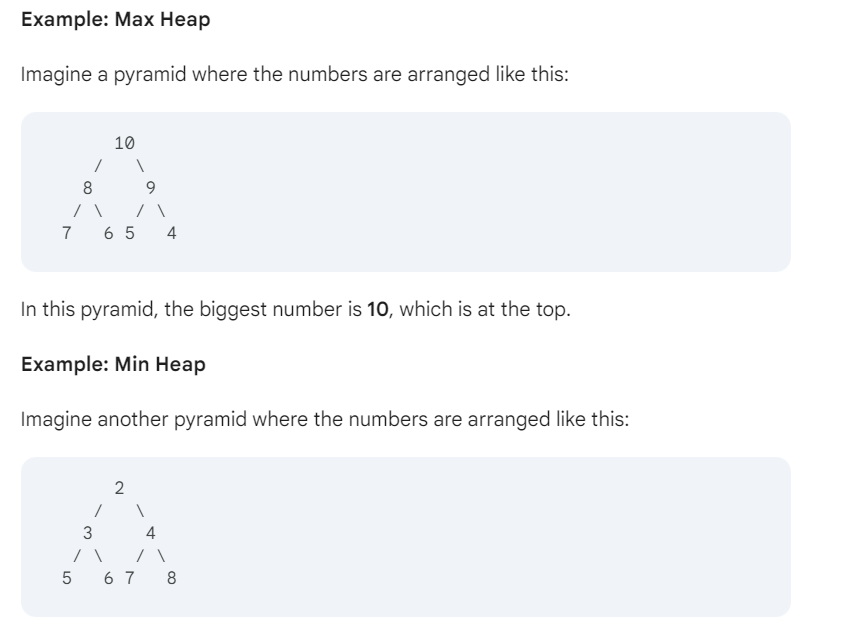
**6.Write about how to find the maximum element in the heap data structure**

**Imagine a heap like a pyramid.**

* **Max heap:** The biggest number is always at the **top** of the pyramid.
* **Min heap:** The smallest number is always at the **top** of the pyramid.

**So, to find the biggest number in a max heap, you just look at the top.**

**But to find the biggest number in a min heap, you have to check every number in the pyramid.**



In this pyramid, the smallest number is **2**, which is at the top.

**To find the biggest number in this min heap, you'd have to check every number.** You'd find that **8** is the biggest.

**7.Discuss the applications of Graphs**

**Social Networks**: Graphs represent users as nodes and relationships (friends, followers) as edges, helping platforms like Facebook and Twitter manage connections and recommend new friends.

**Transportation**: Cities or locations are nodes, and roads are edges. Graphs help find the shortest path or most efficient route in applications like Google Maps.

**Internet (Web Page Ranking)**: Webpages are nodes, and hyperlinks are edges. Graphs are used to rank websites and improve search results (e.g., Google’s PageRank).

**Biological Networks**: Graphs model protein interactions and genetic sequences to understand biological processes and study diseases.

**Telecommunication Networks**: Graphs help optimize data flow and design efficient communication networks.

**8.Difference between directed and undirected graph.**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Directed Graphs** | **Undirected Graphs** |
| **Edges** | Have a direction | Have no direction |
| **Examples** | Road networks with one-way streets, social networks with one-way connections | Road networks with two-way streets, social networks with mutual connections |

**9.Discuss the traversal of graphs.**

**Graph traversal** is the process of visiting all the nodes (vertices) in a graph systematically. There are two main algorithms for graph traversal:

* **Breadth-First Search (BFS):** Visits nodes level by level, starting from a given node.
* **Depth-First Search (DFS):** Explores as deeply as possible along a path before backtracking.

**Applications:**

* **Shortest path algorithms**
* **Cycle detection**
* **Connected components**
* **Topological sorting**
* **Spanning trees**

**10.Discuss the applications of Priority Queue**

**Priority queues** are data structures that allow elements to be stored with associated priorities. Elements with higher priorities are dequeued before elements with lower priorities.

**Common applications:**

* **Scheduling:** Process scheduling, task management
* **Graph algorithms:** Dijkstra's algorithm, Prim's algorithm, Kruskal's algorithm
* **Heap-based data structures:** Heaps, heap sort
* **Event-driven programming:** Event queues
* **Simulation and modeling:** Discrete event simulation
* **Other:** Huffman coding, load balancing, resource allocation