**Unit-4**

**TREE**

**Tree terminologies with simple diagrams**

**1. General Tree**

* Definition: A tree structure where each node can have any number of children.
* Example: A company hierarchy where a manager can oversee multiple employees.

2. Binary Tree

* Definition: A tree where each node has at most two children, referred to as the left and right child.
* Example: A family tree where each person can have up to two children.

3. Binary Search Tree (BST)

* Definition: A binary tree where the left child contains values smaller than the parent node, and the right child contains values larger.
* Example: A collection of numbers sorted in ascending order, where each number is placed based on its value.

4. Left Skewed Binary Tree

* Definition: A binary tree where each node has only a left child, forming a linear structure.
* Example: A chain of nodes where each node points only to the next one on the left.

5. Right Skewed Binary Tree

* Definition: A binary tree where each node has only a right child, also forming a linear structure.
* Example: Similar to left skewed, but each node points only to the next one on the right.

6. Root Node

* Definition: The topmost node in a tree, from which all other nodes descend.
* Example: In a family tree, the oldest ancestor can be considered the root.

7. Parent Node

* Definition: A node that has one or more child nodes.
* Example: In a tree structure, any node that leads to another node below it is a parent.

8. Leaf Node

* Definition: A node that does not have any children.
* Example: The end nodes of a tree that do not lead to any further nodes.

9. Degree of a Node

* Definition: The number of children a particular node has.
* Example: A node with three children has a degree of 3.

10. Degree of a Tree

* Definition: The highest degree of any node in the tree.
* Example: If the node with the most children has 4, then the degree of the tree is 4.

11. Height of a Tree

* Definition: The length of the longest path from the root to any leaf node.
* Example: If the longest path from the root to a leaf node involves 4 edges, the height is 4.

12. Depth of a Tree

* Definition: The distance from the root to a specific node, measured in edges.
* Example: If a node is 2 edges away from the root, its depth is 2.

13. Complete Binary Tree

* Definition: A binary tree in which all levels, except possibly the last, are completely filled, and all nodes are as far left as possible.
* Example: A tree where all levels are filled except the last, which is filled from left to right.

14. Binary Search Tree (Duplicate Entry)

* Definition: Same as above. A binary tree where left children are less than the parent and right children are greater.

15. Expression Tree

* Definition: A binary tree used to represent expressions where each internal node represents an operator and each leaf node represents an operand.
* Example: For the expression (a+b)∗c(a + b) \* c(a+b)∗c, the root is ∗\*∗, with +++ as a left child and ccc as a right child.

16. Max Heap Tree

* Definition: A binary tree where the value of each node is greater than or equal to the values of its children.
* Example: The highest value is at the root, and every parent node is greater than its children.

17. Min Heap Tree

* Definition: A binary tree where the value of each node is less than or equal to the values of its children.
* Example: The smallest value is at the root, and every parent node is less than its children.

18. Successor of a Node

* Definition: The node that comes immediately after a given node in the in-order traversal of the tree.
* Example: In a BST, the successor of a node is the node with the smallest value in its right subtree.

19. Predecessor of a Node

* Definition: The node that comes immediately before a given node in the in-order traversal of the tree.
* Example: In a BST, the predecessor of a node is the node with the largest value in its left subtree.

**1. General Tree**

* **Example**:

A

/ | \

B C D

| \

E F

**2. Binary Tree**

* **Example**:

mathematica

Copy code

A

/ \

B C

/ / \

D E F

**3. Binary Search Tree (BST)**

* **Example**:

15

/ \

10 20

/ \ / \

5 12 18 25

**4. Left Skewed Binary Tree**

* **Example**:

10

/

9

/

8

**5. Right Skewed Binary Tree**

* **Example**:

5

\

10

\

15

**6. Root Node**

* **Example**:

A (Root)

/ \

B C

**7. Parent Node**

* **Example**:

A (Parent)

/ \

B C

**8. Leaf Node**

* **Example**:

A

/ \

B C

/ \

D E

*D and E are leaf nodes.*

**9. Degree of a Node**

* **Example**:

A (Degree 2)

/ \

B C

*Degree of A is 2 (has two children).*

**10. Degree of a Tree**

* **Example**:

A (Degree 3)

/ | \

B C D

*Degree of the tree is 3 (highest degree of node A).*

**11. Height of a Tree**

* **Example**:

A

/ \

B C

/

D

*Height is 3 (A to D).*

**12. Depth of a Tree**

* **Example**:

A

/ \

B C

/

D

*Depth of D is 2 (A to D has two edges).*

**13. Complete Binary Tree**

* **Example**:

A

/ \

B C

/ \ /

D E F

**14. Binary Search Tree (Duplicate Entry)**

* **Example**: Same as above.

15

/ \

10 20

/ \ / \

5 12 18 25

**15. Expression Tree**

* **Example**:

\*

/ \

+ c

/ \

a b

*(This represents the expression (a+b)∗c(a + b) \* c(a+b)∗c).*

**16. Max Heap Tree**

* **Example**:

20

/ \

15 18

/ \ / \

10 12 8 5

*(Each parent is greater than its children.)*

**17. Min Heap Tree**

* **Example**:

5

/ \

10 8

/ \ \

15 12 20

*(Each parent is less than its children.)*

**18. Successor of a Node**

* **Example**:

15

/ \

10 20

\

12 (Successor of 10)

**19. Predecessor of a Node**

* **Example**:

15

/ \

10 20

/ \

5 12 (Predecessor of 15)

**Representation of Binary Tree**

**1. Binary Tree Representation Using Array**

* **Definition**: In an array representation, a binary tree is stored in a contiguous block of memory, where each node corresponds to an index in the array.

**Properties:**

* The root node is stored at index 0.
* For any node at index i:
  + The left child is at index 2i + 1
  + The right child is at index 2i + 2
  + The parent node is at index (i - 1) / 2 (for non-root nodes)

**Example Diagram:**

**Binary Tree**:

A

/ \

B C

/ \ \

D E F

**Array Representation**:

Index: 0 1 2 3 4 5

Value: A B C D E F

**Mapping**:

* A is at index 0 (root).
* B is at index 1 (left child of A).
* C is at index 2 (right child of A).
* D is at index 3 (left child of B).
* E is at index 4 (right child of B).
* F is at index 5 (right child of C).

**2. Binary Tree Representation Using Linked List**

* **Definition**: In a linked list representation, each node of the binary tree is represented as a separate object or structure that contains pointers to its left and right children.

**Properties:**

* Each node consists of:
  + A data field to store the value of the node.
  + A pointer/reference to the left child.
  + A pointer/reference to the right child.

**Example Diagram:**

**Binary Tree**:

A

/ \

B C

/ \ \

D E F

**Linked List Representation**:

lua

Copy code

+---+---+ +---+---+ +---+---+

| A | \* | ---> | B | \* | ---> | C | \* |

+---+---+ +---+---+ +---+---+

| L | R | | L | R | | L | R |

+---+---+ +---+---+ +---+---+

| | | | | |

| | | | | |

+---+---+ +---+---+ +---+

| D | \* | | E | \* | | F | \* |

+---+---+ +---+---+ +---+

| L | R | | L | R | | L | R |

+---+---+ +---+---+ +---+

* **Node Structure**:

plaintext

Copy code

struct Node {

char data; // data field

Node\* left; // pointer to left child

Node\* right; // pointer to right child

};

**Comparison of Array and Linked List Representations**

| **Feature** | **Array Representation** | **Linked List Representation** |
| --- | --- | --- |
| **Memory Usage** | Fixed size (requires maximum size) | Dynamic size (grows as needed) |
| **Access Speed** | Faster access (O(1) for index) | Slower access (O(n) to find node) |
| **Space Efficiency** | Wastes space if tree is sparse | Efficiently uses space |
| **Child Indexing** | Easy to calculate using index | Requires pointers for child access |
| **Structure** | Contiguous block of memory | Nodes linked through pointers |

**Reconstruction of Binary Tree from Traversals**

**1. Understanding Tree Traversals**

* **In-Order Traversal**: Left, Root, Right
* **Pre-Order Traversal**: Root, Left, Right
* **Post-Order Traversal**: Left, Right, Root

**2. Reconstruction from Pre-Order and In-Order Traversals**

**Steps:**

1. **Identify the Root**: The first element of the pre-order traversal is the root of the tree.
2. **Find the Root in In-Order**: Locate the root in the in-order traversal. This splits the in-order list into left and right subtrees.
3. **Recursively Construct Subtrees**:
   * The elements to the left of the root in the in-order list are part of the left subtree.
   * The elements to the right of the root in the in-order list are part of the right subtree.
   * Use the corresponding elements in the pre-order list to construct the left and right subtrees recursively.

**Example:**

* **Given**:
  + **Pre-Order**: A, B, D, E, C, F
  + **In-Order**: D, B, E, A, C, F
* **Reconstruction Process**:

1. **Root**: A (from pre-order)
2. **In-Order Split**:
   * Left Subtree: D, B, E
   * Right Subtree: C, F
3. **Construct Left Subtree**:
   * **Pre-Order for Left**: B, D, E
   * **In-Order for Left**: D, B, E
   * Root is B → D (left), E (right)
4. **Construct Right Subtree**:
   * **Pre-Order for Right**: C, F
   * **In-Order for Right**: C, F
   * Root is C → F (right)

* **Final Tree**:

A

/ \

B C

/ \ \

D E F

**3. Reconstruction from Post-Order and In-Order Traversals**

**Steps:**

1. **Identify the Root**: The last element of the post-order traversal is the root of the tree.
2. **Find the Root in In-Order**: Locate the root in the in-order traversal. This splits the in-order list into left and right subtrees.
3. **Recursively Construct Subtrees**:
   * The elements to the left of the root in the in-order list are part of the left subtree.
   * The elements to the right of the root in the in-order list are part of the right subtree.
   * Use the corresponding elements in the post-order list to construct the left and right subtrees recursively.

**Example:**

* **Given**:
  + **Post-Order**: D, E, B, F, C, A
  + **In-Order**: D, B, E, A, C, F
* **Reconstruction Process**:

1. **Root**: A (from post-order)
2. **In-Order Split**:
   * Left Subtree: D, B, E
   * Right Subtree: C, F
3. **Construct Left Subtree**:
   * **Post-Order for Left**: D, E, B
   * **In-Order for Left**: D, B, E
   * Root is B → D (left), E (right)
4. **Construct Right Subtree**:
   * **Post-Order for Right**: F, C
   * **In-Order for Right**: C, F
   * Root is C → F (right)

* **Final Tree**:

A

/ \

B C

/ \ \

D E F

**Algorithm for Binary Tree Creation and Traversals:**

**1. Binary Tree Creation:**

* **Step 1**: Create a node with data and set its left and right child pointers to NULL.
* **Step 2**: Repeat Step 1 for each new node and link them to their parent nodes as left or right children.

**2. Inorder Traversal (Left, Root, Right):**

* **Step 1**: Traverse the left subtree.
* **Step 2**: Visit the root node (process or print data).
* **Step 3**: Traverse the right subtree.

**3. Preorder Traversal (Root, Left, Right):**

* **Step 1**: Visit the root node (process or print data).
* **Step 2**: Traverse the left subtree.
* **Step 3**: Traverse the right subtree.

**4. Postorder Traversal (Left, Right, Root):**

* **Step 1**: Traverse the left subtree.
* **Step 2**: Traverse the right subtree.
* **Step 3**: Visit the root node (process or print data).

**Steps to Create a Binary Search Tree (BST)**

1. **Start with an Empty Tree**: Begin with an empty tree where the root is null.
2. **Inserting a Value**:
   * **Check if the Tree is Empty**:
     + If the tree is empty, create a new node with the value to be inserted and make it the root of the tree.
   * **Traverse the Tree**:
     + Begin at the root and compare the value to be inserted with the value of the current node:
       - If the value is less than the current node’s value, move to the left child.
       - If the value is greater than the current node’s value, move to the right child.
   * **Repeat the Process**:
     + Continue this process until you find an empty position (where the left or right child is null), and insert the new node at that position.
3. **Maintaining BST Properties**:
   * Ensure that for every node:
     + All values in the left subtree are less than the node’s value.
     + All values in the right subtree are greater than the node’s value.

**Example of Insertion:**

1. Insert 10: The tree is empty, so 10 becomes the root.
2. Insert 5: 5 is less than 10, so it goes to the left of 10.
3. Insert 15: 15 is greater than 10, so it goes to the right of 10.
4. Insert 3: 3 is less than 10 (go left), and less than 5 (go left again), so it goes to the left of 5.
5. Insert 7: 7 is less than 10 (go left), and greater than 5 (go right), so it goes to the right of 5.
6. Insert 12: 12 is greater than 10 (go right), and less than 15 (go left), so it goes to the left of 15.
7. Insert 18: 18 is greater than 10 (go right), and greater than 15 (go right), so it goes to the right of 15.

**Visual Representation:**

After inserting the values 10, 5, 15, 3, 7, 12, 18, the BST will look like this:

10

/ \

5 15

/ \ / \

3 7 12 18

**Summary:**

* Start with an empty tree.
* Insert values by comparing and placing them in the correct position.
* Ensure the properties of the BST are maintained at each insertion.

**Notes for Splay Tree and B-Tree:**

**Splay Tree**

* **Definition**: A self-adjusting binary search tree where frequently accessed elements are moved to the root for faster future access.
* **Operations**:
  + **Splaying**: When a node is accessed, it is moved to the root using a series of rotations.
    - **Zig** (Single Rotation): If the node to be splayed is a left or right child of the root, one rotation brings it to the root.
    - **Zig-Zig** (Double Rotation): If the node and its parent are both left or both right children, two rotations are performed.
    - **Zig-Zag** (Double Rotation): If the node is a left child and its parent is a right child (or vice versa), two rotations are performed in opposite directions.
  + **Insert**: Insert the node like in a Binary Search Tree (BST), then splay the newly inserted node to the root.
  + **Search**: Search like in a BST, then splay the accessed node to the root.
  + **Delete**: Splay the node to be deleted to the root, then remove it like in a BST.
* **Example**:  
  Suppose we have the following operations on an initially empty splay tree:

**Insert**: 10, 20, 30, 40, 50

The splay tree looks like a regular BST:

10

\

20

\

30

\

40

\

50

**Search 30**: After searching for 30, we splay it to the root. The new tree will be:

30

/ \

20 40

/ \

10 50

* **Advantages**:
  + Frequently accessed elements are near the root, so future operations on them are faster.
  + Amortized time complexity is O(log⁡n)O(\log n)O(logn) for basic operations.

**B-Tree**

* **Definition**: A self-balancing search tree in which nodes can have multiple children and all leaf nodes are at the same level. It’s widely used in databases and file systems to store large datasets efficiently.
* **Properties**:
  + A B-tree of order mmm can have a maximum of m−1m-1m−1 keys and mmm children.
  + Internal nodes contain keys that act as separation values to determine which subtree a given key belongs to.
  + All leaf nodes are at the same level, maintaining balance.
* **Operations**:
  + **Insert**: Add the key in the correct place, splitting nodes if they become full.
  + **Delete**: Remove the key, merging or redistributing nodes if necessary.
  + **Search**: Traverse through the tree to find the correct key.
* **Example** (B-tree of order 3): Let's insert the following keys: 10, 20, 5, 6, 12, 30, 7, 17 into a B-tree of order 3.

**Step 1**: Insert 10, 20:

[10 20]

**Step 2**: Insert 5, 6:

[10]

/ \

[5 6] [20]

**Step 3**: Insert 12:

[10 20]

/ | \

[5 6] [12] [30]

**Step 4**: Insert 7:

* + After insertion, 7 is placed in the left node, causing it to split:

[10]

/ \

[5 6 7] [20]

* **Advantages**:
  + Efficient disk reads and writes due to wide structure, reducing the tree's height.