DATA STRUCTURES

MODULE-4:

1. Introduction to Trees

A **tree** is a hierarchical data structure that consists of nodes connected by edges. It is a non-linear data structure used for efficient organization and retrieval of data.

Basic Terminologies in Trees

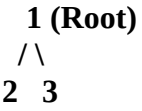
1. **Node**: A fundamental unit of a tree that stores data and references to its children.
2. **Root**: The topmost node of the tree.
3. **Parent**: A node that has one or more child nodes.
4. **Child**: A node that descends from another node.
5. **Sibling**: Nodes that share the same parent.
6. **Leaf Node**: A node with no children.
7. **Degree**: The number of children a node has.
8. **Depth**: The number of edges from the root node to a specific node.
9. **Height**: The longest path from a node to a leaf.
10. **Subtree**: A tree formed by a node and its descendants.
11. **Path**: A sequence of nodes connected by edges.
12. **Ancestor**: Any node in the path from a given node to the root.
13. **Descendant**: Any node that comes after a given node in the hierarchy.

Tree Representation in C

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

int main() {
   struct Node* root = createNode(1);
   root->left = createNode(2);
   root->right = createNode(3);
```

```
    printf("Root Node: %d\n", root->data);
    printf("Left Child: %d\n", root->left->data);
    printf("Right Child: %d\n", root->right->data);
    return 0;
}
```

Graphical Representation of a Tree:



Explanation of Code:

- We define a Node structure with integer data and pointers to left and right children.
- The createNode function dynamically allocates memory for a new node and initializes its data and pointers.
- We create a small tree with a root node and two children, then print their values.

Binary Tree and Its Types

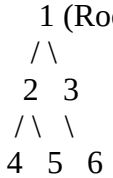
1. Introduction to Binary Tree

A **binary tree** is a hierarchical data structure in which each node has at most two children, referred to as the left and right child.

Properties of a Binary Tree

1. Each node has at most two children.
2. The left child comes before the right child.
3. The height of the binary tree is the longest path from the root to a leaf node.
4. The depth of a node is the number of edges from the root to that node.

Graphical Representation of a Binary Tree

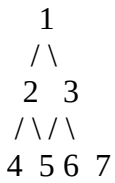


2. Types of Binary Trees

1. Full Binary Tree

A binary tree where each node has either 0 or 2 children.

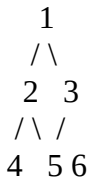
Example:



2. Complete Binary Tree

A binary tree in which all levels are fully filled except possibly the last level, which is filled from left to right.

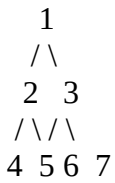
Example:



3. Perfect Binary Tree

A binary tree in which all internal nodes have two children, and all leaf nodes are at the same level.

Example:



4. Balanced Binary Tree

A binary tree where the height difference between the left and right subtrees of any node is at most one.

5. Degenerate (Skewed) Tree

A tree where each parent has only one child, making it resemble a linked list.

Example (Left-Skewed):

```
    1
   /
  2
 /
3
```

Example (Right-Skewed):

```
1
 \
  2
   \
    3
```

3. Binary Tree Implementation in C

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->right = createNode(6);

    printf("Binary Tree Created Successfully!\n");
    return 0;
}
```

Explanation of Code:

- We define a Node structure that contains an integer value and two pointers to represent left and right children.
- The createNode function dynamically allocates memory for a new node.
- We create a sample binary tree with root and child nodes and print a success message.

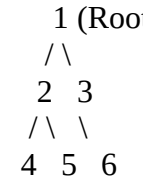
Binary Tree Operations and Implementation in C

1. Introduction to Binary Tree Operations

A **binary tree** is a hierarchical data structure in which each node has at most two children. The basic operations performed on a binary tree include:

- **Insertion**: Adding a node to the tree.
- **Deletion**: Removing a node from the tree.
- **Searching**: Finding a node in the tree.
- **Traversal**: Visiting nodes in a specific order.

Graphical Representation of a Binary Tree



2. Basic Binary Tree Operations

1. Insertion in a Binary Tree

Insertion can be done in different ways:

- **In a Binary Search Tree (BST)**: Insert a node in such a way that the left child is smaller, and the right child is greater.
- **In a Complete Binary Tree**: Insert in level order.

C Implementation of Insertion in a BST

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int data) {
    if (root == NULL) return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
}
```

2. Deletion in a Binary Tree

Deletion involves three cases:

1. **Node to be deleted is a leaf node**: Simply remove it.
2. **Node has one child**: Replace the node with its child.
3. **Node has two children**: Replace with the in-order successor (smallest node in the right subtree).

C Implementation of Deletion in a BST

```
struct Node* findMin(struct Node* root) {
    while (root->left != NULL) root = root->left;
    return root;
}

struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) return root;
    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else {
        if (root->left == NULL) return root->right;
        else if (root->right == NULL) return root->left;
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

3. Searching in a Binary Tree

Searching follows the same rules as insertion in a BST.

C Implementation of Searching in a BST

```
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    return search(root->right, key);
}
```

3. Tree Traversal Techniques

Tree traversal is the process of visiting all nodes of the tree in a specific order. There are three main types of traversal:

1. **Inorder (Left, Root, Right)**
2. **Preorder (Root, Left, Right)**
3. **Postorder (Left, Right, Root)**

C Implementation of Inorder Traversal

```
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

Graphical Representation of Traversal

Inorder Traversal (Left, Root, Right)

Inorder Traversal Output: 4 2 5 1 3 6

Preorder Traversal (Root, Left, Right)

Preorder Traversal Output: 1 2 4 5 3 6

Postorder Traversal (Left, Right, Root)

Postorder Traversal Output: 4 5 2 6 3 1

4. Complete Binary Tree Implementation in C

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
```

```
    struct Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->right = createNode(6);

    printf("Inorder Traversal: ");
    inorder(root);
    return 0;
}
```

Output of the Program:

Inorder Traversal: 4 2 5 1 3 6

5. Summary

- Binary trees support operations like insertion, deletion, searching, and traversal.
- Traversal types include inorder, preorder, and postorder.
- A Binary Search Tree (BST) allows efficient searching and sorting.
- The C implementation provides practical examples of these operations.

Expression Tree in C Programming

1. Introduction to Expression Tree

An **Expression Tree** is a binary tree used to represent arithmetic expressions. In an expression tree:

- **Operands (numbers/variables)** are represented as leaf nodes.
- **Operators (+, -, , /)* are represented as internal nodes.
- The tree follows **postfix notation** where left and right children represent sub-expressions.

Graphical Representation of an Expression Tree

*Example: Expression (3 + 5) * (6 - 2)*

```
      *
     / \
    +   -
   / \ / \
  3  5 6  2
```

2. Construction of an Expression Tree
Steps to Construct an Expression Tree

1. Read the postfix expression from left to right.
2. If the character is an operand, create a node and push it onto the stack.
3. If the character is an operator, pop two nodes from the stack, make them children of the operator node, and push the new node back.
4. The final node in the stack is the root of the expression tree.

3. C Implementation of Expression Tree

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

// Define the structure for a tree node
struct Node {
    char data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(char data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Stack implementation for tree nodes
struct Node* stack[100];
int top = -1;
```

```
void push(struct Node* node) {
   stack[++top] = node;
}

struct Node* pop() {
   return stack[top--];
}

// Function to construct an expression tree from postfix expression
struct Node* constructExpressionTree(char postfix[]) {
   for (int i = 0; postfix[i] != '\0'; i++) {
      char ch = postfix[i];
      struct Node* newNode = createNode(ch);

      if (isalnum(ch)) {
         push(newNode);
      } else {
         newNode->right = pop();
         newNode->left = pop();
         push(newNode);
      }
   }
   return pop();
}

// Inorder Traversal (prints infix expression)
void inorder(struct Node* root) {
   if (root != NULL) {
      if (!isalnum(root->data)) printf("(");
      inorder(root->left);
      printf("%c", root->data);
      inorder(root->right);
      if (!isalnum(root->data)) printf(")");
   }
}

int main() {
   char postfix[] = "35+62-*"; // (3+5)*(6-2)
   struct Node* root = constructExpressionTree(postfix);
   printf("Inorder traversal (Infix expression): ");
   inorder(root);
   printf("\n");
   return 0;
}
```

Output of the Program:

Inorder traversal (Infix expression): ((3+5)*(6-2))

4. Summary

- An **Expression Tree** is a binary tree used to represent arithmetic expressions.
- **Operands are leaf nodes**, and **operators are internal nodes**.
- **Postfix notation** is used to construct the tree efficiently.
- The **C implementation** constructs an expression tree from a postfix expression and displays the infix notation using inorder traversal.

Advanced Tree Structures in C Programming

1. AVL Tree

Introduction to AVL Tree

An **AVL Tree** is a self-balancing binary search tree (BST) where the difference between the heights of left and right subtrees cannot be more than **1** for all nodes.

Graphical Representation of an AVL Tree

```
      30
     /  \
    20   40
   /  \    \
 10  25    50
```

C Implementation of AVL Tree

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
   int key;
   struct Node* left;
   struct Node* right;
   int height;
};

int max(int a, int b) {
   return (a > b) ? a : b;
}

int height(struct Node* N) {
   return (N == NULL) ? 0 : N->height;
}

struct Node* createNode(int key) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->key = key;
   node->left = node->right = NULL;
```

```
    node->height = 1;
    return node;
}

struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

int getBalance(struct Node* N) {
    return (N == NULL) ? 0 : height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int key) {
    if (node == NULL) return createNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else return node;

    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key) return rightRotate(node);
    if (balance < -1 && key > node->right->key) return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
```

```
    }
    return node;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 10);
    root = insert(root, 25);
    root = insert(root, 50);
    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");
    return 0;
}
```

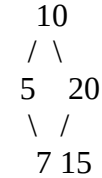
Output:

Inorder traversal: 10 20 25 30 40 50

2. Threaded Binary Tree

A **Threaded Binary Tree** is a variation of the binary tree where NULL pointers are replaced by threads to optimize traversal.

Graphical Representation of a Threaded Binary Tree



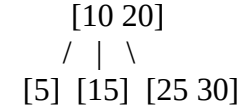
Advantages:

- Faster traversal.
- Reduced memory overhead.

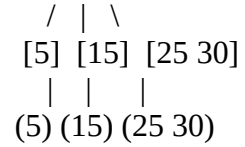
3. Multiway Search Tree (B and B+)

A **B-Tree** is a self-balancing search tree where nodes can have multiple children. A **B+ Tree** is a variation where only leaf nodes store actual data.

Graphical Representation of a B-Tree (Order 3)



Graphical Representation of a B+ Tree (Order 3)



4. Applications of Trees

1. Binary Search Trees (BSTs)

- Used in databases and indexing.
- Implementations in language parsers.

2. AVL Trees

- Used in memory management.
- Provides efficient searching.

3. B-Trees and B+ Trees

- Used in file systems.
- Used in database indexing.

4. Expression Trees

- Used in compilers for arithmetic expressions.

Conclusion

- **AVL Trees** maintain balance for efficient searching and insertion.
- **Threaded Binary Trees** improve traversal speed and reduce memory waste.
- **B-Trees and B+ Trees** optimize data storage for databases and filesystems.