Trees

Generic Trees

Intro :

```
struct TreeNode {
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *secondChild;
    struct TreeNode *thirdChild;
    struct TreeNode *fourthChild;
    struct TreeNode *fifthChild;
    struct TreeNode *sixthChild;
}
```

Simplified to :

```
struct TreeNode {
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
}
```

Problems

Sum of all elements of the tree :

```
int FindSum(struct TreeNode *root) {
    if(!root) {
        return 0;
    }
    return root->data + FindSum(root->firstChild) +
FindSum(root->nextSibling);
}
```

Given a parent array P, where P[i] indicates the parent of ith node in the tree (assume parent of root node is indicated with -1). Give an algorithm for finding the height or depth of the tree :

```
int FindDepthInGenericTree(int P[], int n) {
    int maxDepth = -1, currentDepth = -1, j;
    for (int i = 0; i < n; i++) {
        currentDepth = 0;
        j = i;

        while (P[j] != -1) {
            currentDepth++;
```

```
            j = P[j];
        }

        if (currentDepth > maxDepth)
            maxDepth = currentDepth;
    }
    return maxDepth;
}
```

Count no of siblings of a given node :

```
int SiblingsCount(TreeNode* current) {
    int count = 0;
    while (current) {
        count++;
        current = current->nextSibling;
    }
    return count;
}
```

Count no of children for a given node :

```
int ChildrenCount(TreeNode* current) {
    int count = 0;
    current = current->firstChild;
    while (current) {
        count++;
        current = current->nextSibling;
    }
    return count;
}
```

Check whether two trees are isomorphic or not? :

```
bool IsIsomorphic(TreeNode* root1, TreeNode* root2) {
    if (!root1 && !root2)
        return true;
    if ((!root1 && root2) || (root1 && !root2))
        return false;

    return IsIsomorphic(root1->left, root2->left) &&
IsIsomorphic(root1->right, root2->right);
}
```

Check whether two trees are quasi-isomorphic or not? :
[Two trees root1 and root2 are quasi-isomorphic if root1 can be transformed into root2 by swapping the left and right children of some of the nodes of root1.]

```
bool QuasiIsomorphic(TreeNode* root1, TreeNode* root2) {
    if (!root1 && !root2)
        return true;
    if (!root1 || !root2)
        return false;
    return (QuasiIsomorphic(root1->left, root2->left) &&
QuasiIsomorphic(root1->right, root2->right)) ||
           (QuasiIsomorphic(root1->left, root2->right) &&
QuasiIsomorphic(root1->right, root2->left));
}
```

A full k-ary tree is a tree where each node has either 0 or k children. Given an array that contains the preorder traversal of a full k-ary tree, write an algorithm to construct the full k-ary tree :

```
struct KaryTreeNode {
    char data;
    KaryTreeNode** child;
};

int Ind = 0;

KaryTreeNode* BuildKaryTree(char A[], int n, int k) {
    if (n <= 0 || Ind >= n)
        return NULL;

    KaryTreeNode* newNode = new KaryTreeNode;
    if (!newNode)
        return NULL;

    newNode->child = new KaryTreeNode*[k];
    if (!newNode->child)
        return NULL;

    newNode->data = A[Ind];

    for (int i = 0; i < k; i++) {
        if (Ind + 1 < n) {
            Ind++;
            newNode->child[i] = BuildKaryTree(A, n, k);
        } else {
            newNode->child[i] = NULL;
        }
```

```
    }

    return newNode;
}

void DisplayTree(KaryTreeNode* root, int k) {
    if (!root)
        return;

    cout << root->data << " ";
    for (int i = 0; i < k; i++) {
        DisplayTree(root->child[i], k);
    }
}
```

—---

Threaded Binary Trees
Stack or Queue-less Traversals

Intro :

```
struct ThreadedBTNode {
    strcut ThreadedBTNode *left;
    int LTag;
    int data;
    int RTag;
    struct ThreadedBTNode *right;
}
```

Finding Inorder successor in Inorder Threaded Binary Tree :

```
struct ThreadedBinaryTreeNode* InorderSuccessor(struct ThreadedBinaryTreeNode* P) {
    struct ThreadedBinaryTreeNode* Position;
    if (P->RTag == 0) {
        return P->right;
    } else {
        Position = P->right;
        while (Position->LTag == 1) {
            Position = Position->left;
        }
        return Position;
    }
}
```

Inorder traversal in Inorder Threaded Binary Tree :

Approach 1 :

```
void InorderTraversal(struct ThreadedBinaryTreeNode* root) {
    struct ThreadedBinaryTreeNode* P = InorderSuccessor(root);
    while (P != root) {
        P = InorderSuccessor(P);
        cout << P->data << " ";
    }
}
```

Approach 2 :

```
void InorderTraversalAlternative(struct ThreadedBinaryTreeNode*
root){
    struct ThreadedBinaryTreeNode* P = root;
    while (true) {
        P = InorderSuccessor(P);
        if (P == root) return;
        cout << P->data << " ";
    }
}
```

Finding Pre-order successor in Inorder Threaded Binary Tree :

```
struct ThreadedBinaryTreeNode* PreorderSuccessor(struct ThreadedBinaryTreeNode* P) {
    struct ThreadedBinaryTreeNode* Position;
    if (P->LTag == 1)
        return P->left;
    else {
        Position = P;
        while (Position->RTag == 0)
            Position = Position->right;
        return Position->right;
    }
}
```

Pre-order traversal in Inorder Threaded Binary Tree :

Approach 1 :

```
void PreorderTraversal(struct ThreadedBinaryTreeNode* root) {
    struct ThreadedBinaryTreeNode* P;
    P = PreorderSuccessor(root);
    while (P != root) {
        P = PreorderSuccessor(P);
        cout << P->data << " ";
    }
}
```

Approach 2 :

```
void PreorderTraversal(ThreadedBinaryTreeNode* root) {
    struct ThreadedBinaryTreeNode* P = root;
    while (true) {
        P = PreorderSuccessor(P);
        if (P == root) {
            return ;
        }
        cout << P->data << " ";
    }
}
```

Insertion of nodes in Inorder Threaded Binary Tree :

```
void InsertRightInInorderTBT(ThreadedBinaryTreeNode *P,
ThreadedBinaryTreeNode *Q) {
    ThreadedBinaryTreeNode *Temp;

    Q->right = P->right;
    Q->RTag = P->RTag;
    Q->left = P;
    Q->LTag = 0;
    P->right = Q;
    P->RTag = 1;

    if (Q->RTag == 1) { // Case-2
        Temp = Q->right;
        while (Temp->LTag) {
            Temp = Temp->left;
        }
        Temp->left = Q;
    }
}
```

Problems

For a given binary tree (not threaded), how do we find the preorder predecessor

```
BinaryTreeNode* PreorderSuccessor(BinaryTreeNode *node) {
    static BinaryTreeNode *P = nullptr;
    static Stack *S = CreateStack();
    if (node != nullptr) {
        P = node;
    }
    if (P->left != nullptr) {
        S->Push(P); // Push equivalent
        P = P->left;
```

```
    } else {
        while (P->right == nullptr) {
            P = Pop(S);
        }
        P = P->right;
    }
    return P;
}
```

For a given binary tree (not threaded), how do we find the inorder predecessor

```
BinaryTreeNode* InorderSuccessor(BinaryTreeNode *node) {
    static BinaryTreeNode *P = nullptr;
    static Stack *S = CreateStack();

    if (node != nullptr) {
        P = node;
    }

    if (P->right == nullptr) {
        P = Pop(S);
    } else {
        P = P->right;
        while (P->left != nullptr) {
            S->Push(P); // or Push(S, P)
            P = P->left;
        }
    }

    return P;
}
---------------------------------------------------------------------
-
```

Expression Trees

Building Expression Tree from Postfix Expression

```
BinaryTreeNode* BuildExprTree(const char postfixExpr[], int size) {
    Stack *S = new Stack(size);

    for (int i = 0; i < size; i++) {
        if (isalnum(postfixExpr[i])) { // Check if it is an operand
            BinaryTreeNode *newNode =
(BinaryTreeNode*)malloc(sizeof(BinaryTreeNode));
            if (!newNode) {
                std::cerr << "Memory Error\n";
```

```
                return nullptr;
            }
            newNode->data = postfixExpr[i];
            newNode->left = newNode->right = nullptr;
            S->Push(newNode);
        } else {
            BinaryTreeNode *T2 = S->Pop();
            BinaryTreeNode *T1 = S->Pop();
            BinaryTreeNode *newNode =
(BinaryTreeNode*)malloc(sizeof(BinaryTreeNode));
            if (!newNode) {
                std::cerr << "Memory Error\n";
                return nullptr;
            }
            newNode->data = postfixExpr[i];
            newNode->left = T1;
            newNode->right = T2;
            S->Push(newNode);
        }
    }

    BinaryTreeNode *root = S->Pop();
    delete S;
    return root;
}
```

———
-

<u>XOR Trees</u>

Similar to memory efficient DLL

(no codes)