(Assignment - 01)

Name: Vilakshan roll : 2213256
Class: SY-4 Batch - B

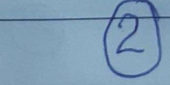
Ques: Construct an expression tree for following prefix expression +3*+592

(i) An expression tree is the tree representation of mathematical expression in this, the leaf represents operands & node as operators.

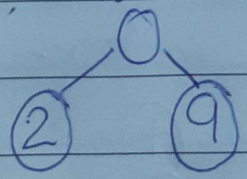
(ii) a) Read expression from right to left
b) If the symbol is an operand, add it to tree as leaf node.
c) If symbol is operator, add root as subtree formed by previous ~~two~~ symbol.

(iii) expression: +3*+592

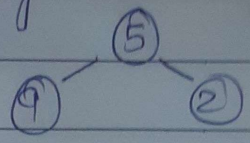
a) Read 2, add it as leaf node.



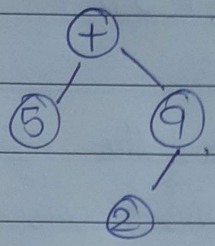
b) Read 9 and follow same steps



c) read 5 again

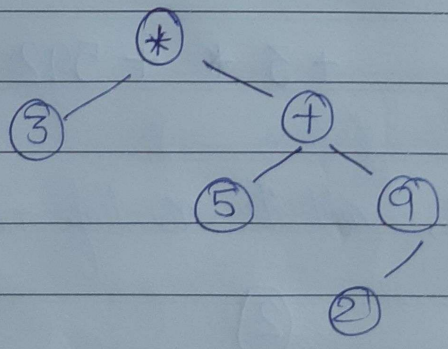


d) Read +, add it as root node to the two previous subtrees.

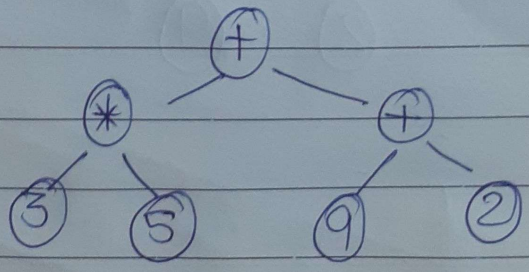


e) read 3.

f) read *, add it as root node to the two previous subtrees,



g) read +, add it as root to the previous two subtrees.



2. Write a non-recursive, in-order, pre-order, post-order traversal algorithms in a binary tree.

ans: (i) Pre-order :

(a) Create empty stack 'nodestack' and push root node to it.

(b) If stack is not empty,

1. pop an item from stack and print it.
2. push right child to popped item.
3. push left child to popped item.

(c) If stack is empty, break.

(ii) In-order -

(a) Create an empty stack 's'.
(b) Initialize current node as root.
(c) push current node to 's' = current → left, until NULL.
d) If current is NULL, then stack is empty, then,
1) pop from the stack
2) print popped item, set current = popped-item → right.
3) goto Step 3.

e) If stack is empty, break.

? (iii) Post-order:

a) Create an empty stack.

(b) do following if root is NOT NULL,

1) push root right child and root to stack.

2) set root as root's left child.

(c) Pop an item from stack and set it as root.

1) If popped item has a right child is at top of stack, push root back and set root as root right child.

2) Else print root data and set root as NULL.

(d) repeat steps (b) and (c) until stack is empty.

Ques1:

Accept prefix expressions, and construct a binary tree and perform recursive and non-recursive traversals.

CODE:

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

// Node structure for binary tree
struct Node {
    char data;
    Node* left;
    Node* right;
};

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

// Function to construct binary tree from prefix expression
Node* constructTree(string prefix) {
    stack<Node*> st;
    int n = prefix.length();

    for (int i = n - 1; i >= 0; i--) {
        char c = prefix[i];
        if (isdigit(c) || isalpha(c)) {
            Node* node = newNode(c);
            st.push(node);
        }
        else {
            Node* node = newNode(c);
            node->left = st.top();
            st.pop();
            node->right = st.top();
            st.pop();
            st.push(node);
        }
    }
    return st.top();
}

// Recursive preorder traversal
void preorder(Node* root) {
```

```
    if (root == NULL) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

// Recursive inorder traversal
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Recursive postorder traversal
void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

// Non-recursive preorder traversal
void iterativePreorder(Node* root) {
    if (root == NULL) return;
    stack<Node*> st;
    st.push(root);

    while (!st.empty()) {
        Node* node = st.top();
        st.pop();
        cout << node->data << " ";

        if (node->right) st.push(node->right);
        if (node->left) st.push(node->left);
    }
}

// Non-recursive inorder traversal
void iterativeInorder(Node* root) {
    if (root == NULL) return;
    stack<Node*> st;
    Node* curr = root;

    while (curr != NULL || !st.empty()) {
        while (curr != NULL) {
            st.push(curr);
            curr = curr->left;
```

```
        }
        curr = st.top();
        st.pop();
        cout << curr->data << " ";
        curr = curr->right;
    }
}

// Non-recursive postorder traversal
void iterativePostorder(Node* root) {
    if (root == NULL) return;
    stack<Node*> st1, st2;
    st1.push(root);

    while (!st1.empty()) {
        Node* node = st1.top();
        st1.pop();
        st2.push(node);

        if (node->left) st1.push(node->left);
        if (node->right) st1.push(node->right);
    }

    while (!st2.empty()) {
        Node* node = st2.top();
        st2.pop();
        cout << node->data << " ";
    }
}

// Driver code
int main() {
    string prefix = "+*AB-CD";
    Node* root = constructTree(prefix);

    cout << "Recursive Preorder Traversal: ";
    preorder(root);
    cout << endl;

    cout << "Recursive Inorder Traversal: ";
    inorder(root);
    cout << endl;

    cout << "Recursive Postorder Traversal: ";
    postorder(root);
    cout << endl;

    cout << "Non-Recursive Preorder Traversal: ";
```

```
    iterativePreorder(root);
    cout << endl;

    cout << "Non-Recursive Inorder Traversal: ";
    iterativeInorder(root);
    cout << endl;

    cout << "Non-Recursive Postorder Traversal: ";
    iterativePostorder(root);
    cout << endl;

return 0;
}
```

OUTPUT:

```
Tree.cpp -o PrefixBinaryTree } ; if ($?) { .\PrefixBinaryTree }
Recursive Preorder Traversal: + * A B - C D
Recursive Inorder Traversal: A * B + C - D
Recursive Postorder Traversal: A B * C D - +
Non-Recursive Preorder Traversal: + * A B - C D
Non-Recursive Inorder Traversal: A * B + C - D
Non-Recursive Postorder Traversal: A B * C D - +
```

MIT SCHOOL OF ENGINEERING
Rajbaug, Loni-Kalbhor, Pune

MIT-ADT UNIVERSITY
PUNE, INDIA
A leap towards World Class Education

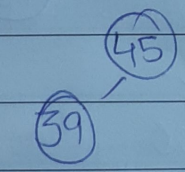
(Assignment - II)

Name: Vilakshan roll: 2213256
Class: SY-4 Batch: B

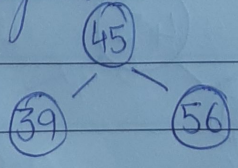
(1.) **Binary Search Tree:** It is a binary tree where each node contains only smaller value on left and larger value on right.
and thus inorder traversal of such a tree returns a sorted list.

1) select 45 as root node,

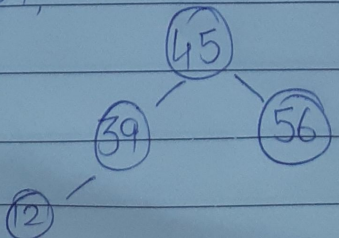
2) select 39, as it is smaller, place it at left.



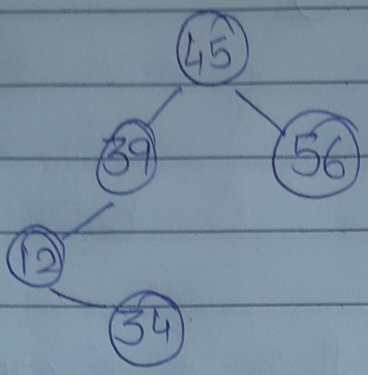
3) now, select 56. which is larger than 45. place on right.



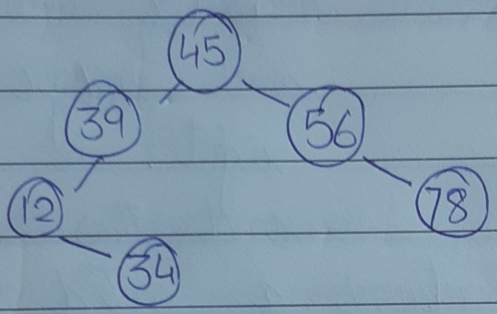
4) Select 12. which is smaller than 39, place on left of 39.



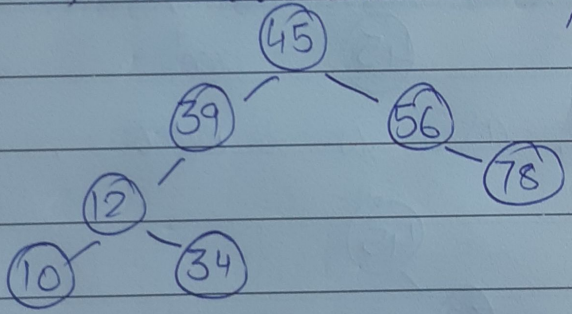
⑤ Selct 39. smaller than 45 and 39, but greater than 12.



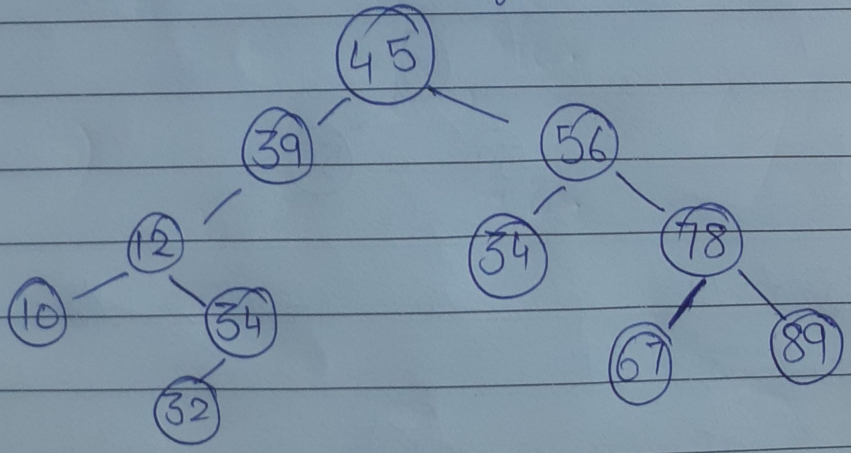
⑥ selct 78. greater than 56, place on right



⑦ Silct 10. smaller than 12, place on left most.



⑧ plac 67, 89, 54 accordingly in tree.



(2.) ① applications of BST (Binary Search Tree)

* used for indexing algorithm.
* to implement various search algorithms.
* to implement various data structures.
* can be used in fast auto complete systems.
* used in hoffman coding algorithm.
* used for data catching.
* used in spell checkness.

(3.) ① Creating a B.S.T -

1) add first node as root node.
2) for the following node compare the values with root node,
   - if less than root, put in left subtree.
   - if greater than root, put in right.

② Inserting an element in B.S.T -

1) start with root node.
2) Compare value of new node with current node,
   - If less,
     move to left child.
   - If more,
     move to right child

③ Searching an element.

1) Start at root node
2) Compare value of target with current node -
   - If value is equal to current node, return the node.
   - If less, then current move to left.
   - If greater, than current, move to right child.

④ modifying an element in B.S.T.

1) Start at root node
2) Compare value of target with current node.
   - If less than current, move to left child.
   - If greater than current, move to right child.
   - If equal to current, update value of node.
3) repeat until the target is found, or tree is over.

Ques2:

A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide a facility to display whole data sorted in ascending/ Descending order. Also, find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.

CODE:

```
/*
Name : Vilakshan
Roll : 2213256
Class: SY-CSE-4
*/
// A dictionary stores keywords and its meaning -
// Provide a facility for -
// 1. adding new keywords
// 2. deleting keywords
// Use binary Search Tree for implementing
#include <iostream>
using namespace std;

class Node {
    public:
    string keyword, meaning;
    Node *left, *right;
    Node() {
        keyword = ""; // to be used in traversing
        left = NULL;
        right = NULL;
    }
    Node(string kword, string mning) {
        keyword = kword;
        meaning = mning;
        left = NULL;
        right = NULL;
    }
};

Node* insertNode(Node *root) {
    string keyword, meaning;
    cout << "Enter the key : ";
    cin >> keyword;
    cout << "Enter the meaning : ";
    cin >> meaning;
    Node *node = new Node(keyword, meaning); // creating the node through
constructor
    Node *curr = root;
```

```
    bool notInserted = true;
    if (curr == NULL) {
        curr = node; // insert the node into root node
    }
    else {
        while (notInserted)
        {
            if (curr == NULL)
            {
                curr = node; // Inserted Successfully
                cout << keyword << " : " << meaning;
                cout << "---INSERTED SUCCESSFULLY---\n\n";
                notInserted = false; // INSERTED SUCCESSFULLY
            }
            else if (keyword < curr -> keyword) {
                // move to left child
                curr = curr -> left;
            }
            else if (keyword > curr -> keyword) {
                // move to right child
                curr = curr -> right;
            }
        }
    }

    return node;
}

void deleteNode(Node *root) {
    string kword;
    cout << "Enter the keyword to delete : ";
    cin >> kword;

    Node *curr = root, *bcurr = root;
    //bcurr will move one step behind curr

    int left;
    // will record left or right transition

    // traverse and find the keyword
    // to delete keyword -
    // 1. traverse to the left node if kword if smaller than the curr and
negation is true
    // 2. delete the node by using Free() method.

    while (curr -> left != NULL) {
        if (kword == curr -> keyword) {
            if (left == 1)
```

```
                bcurr -> left = curr -> right;
            else
                bcurr -> right = curr -> right;
            cout << curr -> keyword << " : " << curr -> meaning;
            delete(curr);
            cout << "---Deleted Successfully---\n\n";
            break; // Node deleted successfully
        }
        else if (kword < curr -> keyword) {
            bcurr = curr;
            curr = curr -> left;
            left = 1;
        }
        else if (kword > curr -> keyword) {
            bcurr = curr;
            curr = curr -> right;
            left = 0;
        }

    }
    cout << "---NO SUCH KEYWORD FOUND---\n\n"; // traversing returned NULL

}

void showDictionary (Node *root) { //In-order Traversal - sorting method for
B.S.T.
    cout << root;
    showDictionary(root->left);
    if (root == NULL)
    //if (root -> keyword == NULL || *root == NULL)
    {
        cout << root -> keyword << " : ";
        cout << root -> meaning << endl;
        return; // returning void, just for the sake of recursion
    }

    showDictionary(root->right);
}

int main() {
    cout << "\n\t\tAuthor : Vilakshan, 2213256\n\n\n";

    char code; // for menu
    int run = 0;

    Node *bstroot = new Node(); //The Dictionary root Node

    bool cont = true;
```

```
    while (cont)
    {
        if (run++ % 5 == 0) {
            string menu[] = {"Add a Keyword", "Delete a Keyword", "Show
Dictionary", "Exit"};
            cout << "\t---Menu---\nCode\tOperation\n";
            for (int i = 0; i < 3; i++)
            cout << i + 1 << "\t" << menu[i] << endl;
        }
        cout << "code : ";
        cin >> code;
        switch (code) {
            case '1':
                insertNode(bstroot);
                break;
            case '2':
                deleteNode(bstroot);
                break;
            case '3':
                showDictionary(bstroot);
                break;
            case '4':
                cout << "\n\t---exiting---";
                cont = false;
                break;
        }
    }



    return 0;
}
```

OUTPUT:

```
        Author : Vilakshan, 2213256


    ---Menu---
Code    Operation
1       Add a Keyword
2       Delete a Keyword
3       Show Dictionary
code : 1

Enter the key : v
Enter the meaning : vilakshan
v : vilakshan
---INSERTED SUCCESSFULLY---


code : 1

Enter the key : A
Enter the meaning : Anmol
A : Anmol
---INSERTED SUCCESSFULLY---


code : 1

Enter the key : D
Enter the meaning : Django
D : Django
---INSERTED SUCCESSFULLY---


code : 3

v : vilakshan
A : Anmol
D : Django


code : 2

Enter the keyword to delete : A
A : Anmol
---Deleted Successfully---

code : 3

v : vilakshan
D : Django
```

PRACTICAL - III

Name: Vilakshan
Roll: 2213256

- FAQ's:-

1> How we can find mirror image of Binary Search Tree?

→ To find the mirror image of a Binary Search tree, we need to swap the left & right children of every node in tree. This can be done recursively using following steps:

1. If the root of tree is NULL, then the mirror image is also NULL.
2. Otherwise, create a new node that has same value as the root node, and swap the left and right children of this new node.
3. Recursively find mirror image of left subtree of the original tree, and set it as the right subtree of new node created in step 2.
4. Recursively find mirror image of right subtree of original tree, and set it as left subtree of new node created in step 2.
5. Return the new node as the root of the mirror image tree.

2> What is the process to find height of Binary Search Tree?

→ The height of a binary search tree is defined as the number of edges on the longest path from the ~~root~~ root node to a leaf node in tree. To find height of binary search tree, we can use a recursive approach that traverses the tree and calculates the height of the left & right subtree at each node.

Here is algorithm to find height of binary search tree:-

1. If root node is NULL, then height of tree is 0.
2. Otherwise, recursively calculate the height of subtree by calling the height function on left child of root node.
3. Recursively calculate the height of right subtree by calling the height function on right child of root node.
4. Return the maximum of the heights of left and right subtrees, plus 1 for root node.

3. Write down steps to perform level order traversal of binary search tree.

→ Level order traversal also known as breadth first traversal, is a way of traversing a binary search tree where we visit each node in tree level by level. To perform level order traversal of BST we can use a queue to store the nodes at each level of tree, and process them in order they were added to queue. Here are the steps to perform level order traversal of a binary search tree:-

1. Create an empty queue and add the root node to queue.
2. While the queue is not empty, do the following
   a) Remove the node at front of queue and print its value.
   b) Add the left child of removed node to queue (if exists).
   c) Add the right child of removed node to queue (if exists).
3. Repeat step 2 until queue is empty.

Ques 3:

Create a Binary Search tree and find its mirror image. Print original & new tree level

wise. Find height & print leaf nodes.

CODE:

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

// Definition of a Binary Search Tree node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a node into a Binary Search Tree
Node* insert(Node* root, int val) {
    if (root == nullptr) {
        return new Node(val);
    }
    if (val < root->data) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }
    return root;
}

// Function to find the mirror image of a Binary Search Tree
Node* mirror(Node* root) {
    if (root == nullptr) {
        return nullptr;
    }
    Node* temp = root->left;
    root->left = mirror(root->right);
    root->right = mirror(temp);
    return root;
}

// Function to print the level-order traversal of a Binary Search Tree
void printLevelOrder(Node* root) {
```

```
    if (root == nullptr) {
        return;
    }
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        int n = q.size();
        for (int i = 0; i < n; i++) {
            Node* node = q.front();
            q.pop();
            cout << node->data << " ";
            if (node->left != nullptr) {
                q.push(node->left);
            }
            if (node->right != nullptr) {
                q.push(node->right);
            }
        }
        cout << endl;
    }
}

// Function to find the height of a Binary Search Tree
int height(Node* root) {
    if (root == nullptr) {
        return 0;
    }
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    return max(leftHeight, rightHeight) + 1;
}

// Function to print the leaf nodes of a Binary Search Tree
void printLeafNodes(Node* root) {
    if (root == nullptr) {
        return;
    }
    if (root->left == nullptr && root->right == nullptr) {
        cout << root->data << " ";
    }
    printLeafNodes(root->left);
    printLeafNodes(root->right);
}

// Driver code
int main() {
    // Create a Binary Search Tree
    Node* root = nullptr;
```

```
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 5);
    insert(root, 7);

    // Print the original tree level-wise
    cout << "Original tree level-wise:" << endl;
    printLevelOrder(root);

    // Find the mirror image of the Binary Search Tree
    Node* mirrorRoot = mirror(root);

    // Print the new tree level-wise
    cout << "New tree level-wise:" << endl;
    printLevelOrder(mirrorRoot);

    // Find the height of the original tree
    int treeHeight = height(root);
    cout << "Height of the tree: " << treeHeight << endl;

    // Print the leaf nodes of the original tree
    cout << "Leaf nodes of the tree: ";
    printLeafNodes(root);
    cout << endl;

    return 0;
}


// Original tree level-wise:
// 4
// 2 6
// 1 3 5 7
// New tree level-wise:
// 4
// 6 2
// 7 5 3 1
// Height of the tree: 3
// Leaf nodes of the tree: 1 3 5 7
```

OUTPUT:

```
Original tree level-wise:
4
2 6
1 3 5 7
New tree level-wise:
4
6 2
7 5 3 1
Height of the tree: 3
Leaf nodes of the tree: 7 5 3 1
```

PRACTICAL - IV

Name: Vilakshan
Roll: 2213256

- FAQ's:-

1) Create a Threaded binary tree using the following data values (step wise)

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67

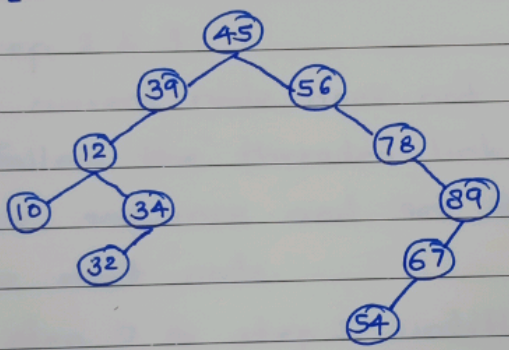
→ To create a threaded binary tree we need to add extra links to each node that allow us to traverse the tree without using recursion. The two types of threads that can be added to each node are:

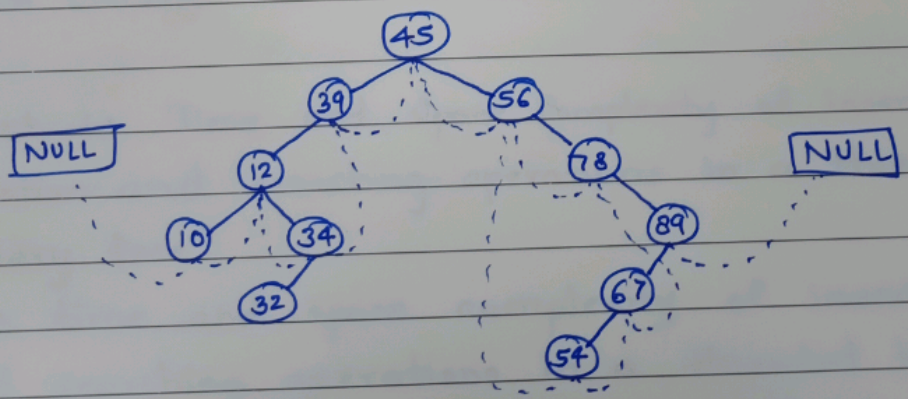
1. Inorder predecessor thread.
2. Inorder successor thread.

Steps to create threaded binary tree:

1. Create the root node with value of 45.
2. Add remaining nodes to tree one by one, starting with 39.
   a) Compare value of new node with current node.
   b) If new node is less than the current node, move to left child.
   c) If new node is greater than the current node, move to right child.
   d) Repeat process until we find an empty position to insert new node.

3. After inserting each node, set its left and right children to NULL.
4. Traverse tree in inorder, and add the inorder predecessor and successor threads to each node:
   a) If a node has left child, set its inorder predecessor thread to right most node in left subtree.
   b) If a node has right child, set its inorder successor thread to leftmost node in right subtree.
   c) Repeat process for each node in the tree.
5. Return the root node of threaded binary tree.





2> Write an algorithm for an Inorder Traversal of a Inorder threaded Binary Tree.

→ Inorder traversal of a inorder threaded binary tree can be performed using the following algorithm:

1. Initialize the current node as the leftmost node of the tree.
2. Traverse the left subtree by following the left child pointer until a null pointer is reached.
3. Process the current node.
4. If the current node has a right child pointer, set the current node to be the right child and repeat step 2 & 3.
5. If the current node does not have a right child pointer, follow the threaded link to the next node in inorder sequence and set the current node to be the next node.
6. Repeat step 2 to step 5 until all nodes have been visited.

3> What is Time and Space Complexity of insertion, deletion and searching operations in a threaded binary tree?

→ The time and space complexity of insertion, deletion and searching operations in a threaded binary tree depend on the type of threaded binary tree. There are two types of threaded binary tree:

i> Inorder threaded binary tree
ii> Preorder threaded binary tree

1. Inorder Threaded Binary Tree:
- Insertion : O(n)
- Deletion: O(n)
- Searching: O(n)

The space complexity of an inorder threaded binary tree is O(n), where n is no. of nodes in tree, as each node requires a thread link in addition to left and right child pointers.

2. Preorder Threaded Binary Tree:
- Insertion : O(1)
- Deletion : O(1)
- Searching: O(n)

The space complexity of preorder threaded binary tree is also O(n), where n is no. of nodes in tree, as each node requires a thread link in addition to left and right child pointers.

In general threaded binary tree provide faster traversal times compared to non-threaded binary tree but at the cost of increased space complexity.

Ques 4:

Create an in-order threaded binary search tree and perform the traversals.

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

// Node structure for threaded binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    bool isThreaded;

    Node(int data) {
        this->data = data;
        this->left = this->right = NULL;
        this->isThreaded = false;
    }
};

// Function to find the in-order successor of a node
Node* inOrderSuccessor(Node* node) {
    if (node == NULL) return NULL;

    // If node has a right child, then its in-order successor is the leftmost node in the
right subtree
    if (node->right != NULL) {
        node = node->right;
        while (node->left != NULL) {
            node = node->left;
        }
        return node;
    }

    // If node does not have a right child, then its in-order successor is the first
ancestor
    // that is threaded, i.e. the first ancestor that has a left child that is not its
direct predecessor
    while (node->isThreaded && node->right == NULL) {
        node = node->right;
    }
    return node->right;
}

// Function to insert a node into the threaded binary tree
Node* insertNode(Node* root, int data) {
    Node* node = new Node(data);

    if (root == NULL) {
```

```
        return node;
    }

    Node* curr = root;
    Node* prev = NULL;

    while (curr != NULL) {
        prev = curr;
        if (data < curr->data) {
            if (curr->isThreaded || curr->left == NULL) {
                node->left = curr->left;
                node->right = curr;
                curr->left = node;
                curr->isThreaded = false;
                return root;
            }
            curr = curr->left;
        }
        else {
            if (curr->isThreaded || curr->right == NULL) {
                node->left = curr;
                node->right = curr->right;
                curr->right = node;
                curr->isThreaded = false;
                return root;
            }
            curr = curr->right;
        }
    }

    if (data < prev->data) {
        node->left = prev->left;
        node->right = prev;
        prev->left = node;
    }
    else {
        node->left = prev;
        node->right = prev->right;
        prev->right = node;
    }
    return root;
}

// Function to perform in-order traversal of threaded binary tree
void inOrderTraversal(Node* root) {
    if (root == NULL) return;

    Node* curr = root;
```

```
    while (curr->left != NULL) {
        curr = curr->left;
    }

    while (curr != NULL) {
        cout << curr->data << " ";
        curr = inOrderSuccessor(curr);
    }
}

// Driver code
int main() {
    Node* root = NULL;

    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    cout << "In-order Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

OUTPUT:

```
ryTree }
Inorder Traversal : 20 30 40 50 60 70 80
```