Search Trees in C

Course: Introduction to Programming and Data Structures

Dr. Laltu Sardar

Institute for Advancing Intelligence (IAI),
TCG Centres for Research and Education in Science and Technology (TCG Crest)

- Analysis of Recursive Approaches

Binary Search Trees

What is a Tree?

- A Tree is a data structure consisting of nodes.
- Each node contains a value or data, and links to child nodes.
- The topmost node is called the **root**.
- A node without children is called a **leaf**.

Tree Terminology

- Root: The topmost node in the tree.
- Leaf: A node with no children.
- Parent: A node that has children.
- Subtree: A tree consisting of a node and its descendants.
- Binary Tree: A tree where each node has at most two children.

Types of Trees

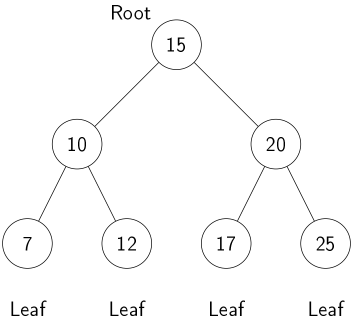
- **Binary Tree**: Each node has at most two children.
- **Binary Search Tree (BST)**: A binary tree with ordered nodes.
- **AVL Tree**: A self-balancing binary search tree.
- m-**ary Tree**: Each node has at most m children.
- **Heap**: A tree where the parent node is greater (or smaller) than its children.

Binary Search Tree

- **left** subtree contains **lesser** values, by convention
- **right** subtree contains **higher** values, by convention

tcg crest
Inventing Harmonious Future

Binary Search Tree (BST) Example



- **Root**: The topmost node (15).
- **Leaf**: Nodes without children (7, 12, 17, 25).
- **Parent and Child**: Relationships between nodes (e.g., 10 is parent, 7 and 12 are children).
- **Subtree**: A smaller part of the tree, e.g., (10, 7, 12).

Binary Search Tree (BST) Implementation in C

Node structure

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

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

Insert Function in C

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

```

tcg crest
Inventing Harmonious Future

Searching in a BST

```
struct Node* search(struct Node* root, int key)
{

    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

```

Tree Traversal Algorithms in C

- **Inorder Traversal** (Left, Root, Right):
  - Traverse the left subtree.
  - Visit the root.
  - Traverse the right subtree.
- **Preorder Traversal** (Root, Left, Right):
  - Visit the root.
  - Traverse the left subtree.
  - Traverse the right subtree.
- **Postorder Traversal** (Left, Right, Root):
  - Traverse the left subtree.
  - Traverse the right subtree.
  - Visit the root.

Tree Traversal Examples

Sample Tree:

```
      1
     / \
    2   3
   / \
  4   5
```

- **Inorder Traversal**: 4, 2, 5, 1, 3
- **Preorder Traversal**: 1, 2, 4, 5, 3
- **Postorder Traversal**: 4, 5, 2, 3, 1

Inorder Traversal in C

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

Preorder Traversal in C

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

```

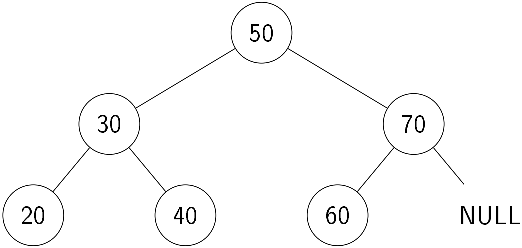
Postorder Traversal in C

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

Deletion in a Binary Search Tree

- **Case 1:** Deleting a leaf node (no children).
- **Case 2:** Deleting a node with one child.
- **Case 3:** Deleting a node with two children (find in-order successor or predecessor).

Example: Deletion in a BST



- **Case 1**: Deleting a Leaf Node, Delete 20
- **Case 2**: Deleting a Node with One Child, Delete 70
- **Case 3**: Deleting a Node with Two Children, Delete 50

Detailed Explanation of Deletion Cases

- **Case 1: Deleting a Leaf Node**
  Example: Delete node 20. Since it has no children, simply remove the node.
- **Case 2: Deleting a Node with One Child**
  Example: Delete node 70. Replace the node with its only child (80).
- **Case 3: Deleting a Node with Two Children**
  Example: Delete node 50. Replace the node with its in-order successor (60), and adjust the tree.

Deletion in Binary Search Tree

```
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) return root;

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

minValueNode() Function in C

```c
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;

    // Find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;

    return current;
}
```

Iterative Algorithms
in
Binary Search Trees

tcg crest
Inventing Harmonious Future

Iterative Inorder Traversal

- Inorder Traversal: Left subtree, Root, Right subtree.
- Use an explicit stack to simulate the recursive behavior.

```
void iterativeInorder(struct Node* root) {
    struct Node* current = root;
    struct Stack* stack = createStack(MAX_HEIGHT);

    while (!isEmpty(stack) || current != NULL) {
        if (current != NULL) {
            push(stack, current);
            current = current->left;
        } else {
            current = pop(stack);
            printf("%d ->", current->data);
            current = current->right;
        }
    }
}

```

Iterative Preorder Traversal

- Preorder Traversal: Root, Left subtree, Right subtree.
- Use an explicit stack to simulate recursive preorder traversal.

```
void iterativePreorder(struct Node* root) {
    if (root == NULL) return;
    struct Stack* stack = createStack(MAX_HEIGHT);
    push(stack, root);

    while (!isEmpty(stack)) {
        struct Node* current = pop(stack);
        printf("%d -> ", current->data);

        if (current->right != NULL) push(stack, current->right);
        if (current->left != NULL) push(stack, current->left);
    }
}
```

tcg crest
Inventing Harmonious Future

Iterative Postorder Traversal

- Postorder Traversal: Left subtree, Right subtree, Root.
- Use two stacks to simulate the recursive behavior.

```c
void iterativePostorder(struct Node* root) {
    if (root == NULL) return;
    struct Stack* s1 = createStack(MAX_HEIGHT);
    struct Stack* s2 = createStack(MAX_HEIGHT);

    push(s1, root);
    while (!isEmpty(s1)) {
        struct Node* current = pop(s1);
        push(s2, current);

        if (current->left != NULL) push(s1, current->left);
        if (current->right != NULL) push(s1, current->right);
    }

    while (!isEmpty(s2)) {
        struct Node* node = pop(s2);
        printf("%d ->", node->data);
    }
}
```

Iterative Search in BST

- Search for a key in the Binary Search Tree using a loop.
- Traverse left if the key is smaller than the current node.
- Traverse right if the key is larger.

```
struct Node* iterativeSearch(struct Node* root, int key) {
    while (root != NULL) {
        if (root->data == key)
            return root;
        else if (key < root->data)
            root = root->left;
        else
            root = root->right;
    }
    return NULL;
}
```

Iterative Deletion in BST

- Delete a node in a Binary Search Tree iteratively.
- Handle three cases: node to be deleted has no child, one child, or two children.

Iterative Deletion Algorithm in BST I

```c
struct Node* deleteNodeIterative(struct Node* root, int key) {
    struct Node* parent = NULL;
  struct Node* * current = root;

    while (current != NULL && current->data != key) {
        parent = current;
        if (key < current->data)
      current = current->left;
        else
      current = current->right;
    }
    if (current == NULL) return root;  // Node not found
    if (current->left == NULL || current->right == NULL) {
        struct Node* newCurr = (current->left) ? current->left : current->right;
        if (parent == NULL)
            return newCurr;
        if (current == parent->left)
            parent->left = newCurr;
        else
```

Iterative Deletion Algorithm in BST II

```
            parent->right = newCurr;
        free(current);
    } else {
        struct Node* successor = minValueNode(current->right);
        int successorData = successor->data;
        deleteNodeIterative(root, successorData);
        current->data = successorData;
    }
    return root;
}
```

minValueNode() Function in BST

- The ‘minValueNode()‘ function finds the smallest node in a subtree.
- This is useful when deleting a node with two children.

```c
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

```

Complexity Analysis
of
of BST algorithms

Complexities of Inorder, Preorder, and Postorder Traversals

- **Time Complexity**: $O(n)$
  - Each node is visited once.
- **Space Complexity**:
  - **Recursive Traversal**: $O(h)$, where h is the height of the tree.
  - **Iterative Traversal**: $O(h)$, as an explicit stack is used to simulate recursion.
- **Best Case**:
  - For a balanced BST, the height h is $O(\log n)$, making the space complexity $O(\log n)$.
- **Worst Case**:
  - In a skewed tree, the height h can be $O(n)$, leading to $O(n)$ space complexity.

Complexity of Search in a BST

- **Time Complexity**:
  - **Best Case**: $O(1)$, when the key is found at the root.
  - **Average Case**: $O(\log n)$, for a balanced tree.
  - **Worst Case**: $O(n)$, for a skewed tree.
- **Space Complexity**:
  - **Recursive Search**: $O(h)$, due to the call stack, where h is the height of the tree.
  - **Iterative Search**: $O(1)$, no extra space is required aside from the traversal.

Complexity of Deletion in a BST

- **Time Complexity**:
  - **Best Case**: $O(1)$, when deleting a node with no children.
  - **Average Case**: $O(\log n)$, for a balanced tree.
  - **Worst Case**: $O(n)$, for a skewed tree.
- **Space Complexity**:
  - **Recursive Deletion**: $O(h)$, due to the call stack, where h is the height of the tree.
  - **Iterative Deletion**: $O(1)$, as there is no need for recursion.

Advantages of Recursive Algorithms

Advantages:

- **Simplicity**:
  - Recursive code is more compact and easier to write for tree-based operations.
  - Natural fit for trees due to the recursive structure of trees (hierarchical data).
- **Clarity**:
  - Recursive code is often clearer and easier to understand, particularly for beginners.

tcg crest
Inventing Harmonious Future

Drawbacks of Recursive Algorithms

Drawbacks:

- **Memory Overhead**:
  - Each recursive call adds a new frame to the call stack. For deep trees (e.g., skewed trees), this could lead to stack overflow.
- **Performance**:
  - Recursion may add some overhead due to function calls, and managing the call stack.
- **Stack Limitations**:
  - Recursive solutions can fail for very deep trees if the depth exceeds the stack size, leading to stack overflow errors.

tcg crest
Inventing Harmonious Future

tcg crest
Inventing Harmonious Future

Dr. Laltu Sardar
laltu.sardar@tcgcrest.org
https://laltu-sardar.github.io.