

MTH 4300/4299

More on Trees: Iterative traversals, Tries, and more problems

Jaime Abbariao

Binary Search Trees

BST Properties

- Left subtree contains only nodes with values **less than** the root
- Right subtree contains only nodes with values **greater than** the root
- Both left and right subtrees are also BSTs
- No duplicate values (in this implementation)

Binary Search Trees (ii)

BST Node Structure

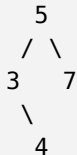
```
struct BSTNode {  
    int val;  
    BSTNode* left;  
    BSTNode* right;  
  
    BSTNode(int x) : val(x), left(nullptr), right(nullptr) {}  
};
```

Binary Search Trees (iii)

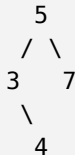
BST Insert Operation

Visual example - inserting 4 into existing BST:

Before:



After inserting 4:

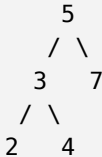
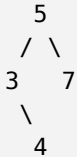


Steps:

1. Start at root (5): $4 < 5$, go left
2. At node (3): $4 > 3$, go right
3. At node (4): $4 == 4$, no insertion (no duplicates)

Binary Search Trees (iv)

Inserting 2:



Binary Search Trees (v)

```
BSTNode* insert(BSTNode* root, int val) {  
    if (root == nullptr) {  
        return new BSTNode(val);  
    }  
  
    if (val < root->val) {  
        root->left = insert(root->left, val);  
    } else if (val > root->val) {  
        root->right = insert(root->right, val);  
    }  
    // If val == root->val, do nothing (no duplicates)  
  
    return root;  
}
```

Binary Search Trees (vi)

BST Search Operation

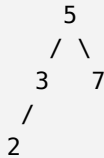
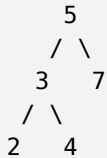
```
bool search(BSTNode* root, int val) {  
    if (root == nullptr) {  
        return false;  
    }  
  
    if (val == root->val) {  
        return true;  
    } else if (val < root->val) {  
        return search(root->left, val);  
    } else {  
        return search(root->right, val);  
    }  
}
```

Binary Search Trees (vii)

BST Delete Operation

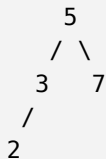
Visual examples of three deletion cases:

Case 1 - Delete leaf (4):



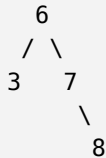
Binary Search Trees (viii)

Case 2 - Delete node with one child (3):



Binary Search Trees (ix)

Case 3 - Delete node with two children (5):



(replace with successor)

Binary Search Trees (x)

```
BSTNode* deleteNode(BSTNode* root, int val) {  
    if (root == nullptr) return nullptr;  
  
    if (val < root->val) {  
        root->left = deleteNode(root->left, val);  
    } else if (val > root->val) {  
        root->right = deleteNode(root->right, val);  
    } else {  
        // Node to delete found  
        if (root->left == nullptr) {  
            BSTNode* temp = root->right;  
            delete root;  
            return temp;  
        } else if (root->right == nullptr) {  
            BSTNode* temp = root->left;  
            delete root;  
            return temp;  
        }  
    }  
}
```

Binary Search Trees (xi)

```
    }

    // Node with two children
    BSTNode* temp = findMin(root->right);
    root->val = temp->val;
    root->right = deleteNode(root->right, temp->val);
}
return root;
}

BSTNode* findMin(BSTNode* root) {
    while (root && root->left) {
        root = root->left;
    }
    return root;
}
```

Binary Search Trees (xii)

BST Time Complexity

- **Average case:** $O(\log n)$ for search, insert, delete
- **Worst case:** $O(n)$ when tree becomes a linked list
- **Best case:** $O(\log n)$ when tree is balanced

Binary Search Trees (xiii)

BST Applications

- Database indexing
- Expression parsing
- Priority queues
- File system organization
- Auto-complete systems

Tries

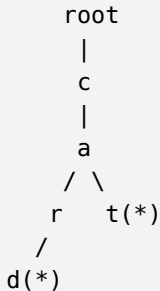
What is a Trie?

- Also called **prefix tree** or **digital tree**
- Tree data structure for storing strings
- Each path from root represents a string prefix
- Efficient for prefix-based operations

Tries (ii)

Trie Structure

Visual representation of Trie storing ["cat", "car", "card"]:



(*) = end of word marker

Path for "cat": root → c → a → t (isEndOfWord = true)

Tries (iii)

Path for "car": root → c → a → r (isEndOfWord = true)

Path for "card": root → c → a → r → d (isEndOfWord = true)

Tries (iv)

```
struct TrieNode {
    std::map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() : isEndOfWord(false) {}
};

class Trie {
private:
    TrieNode* root;

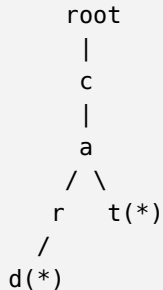
public:
    Trie() {
        root = new TrieNode();
    }
};
```

Tries (v)

Insert Operation

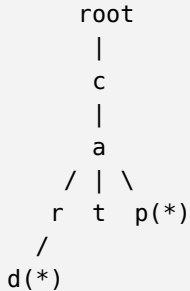
Visual example - inserting “cap” into existing Trie:

Before (has "cat", "car", "card"):



Tries (vi)

After inserting "cap":



Steps for "cap":

1. Follow $c \rightarrow a$ (existing path)
2. Create new node for 'p'
3. Mark 'p' node as end of word

Tries (vii)

```
void insert(const std::string& word) {
    TrieNode* current = root;

    for (char ch : word) {
        if (current->children.find(ch) == current-
>children.end()) {
            current->children[ch] = new TrieNode();
        }
        current = current->children[ch];
    }

    current->isEndOfWord = true;
}
```

Tries (viii)

Search Operation

```
bool search(const std::string& word) {  
    TrieNode* current = root;  
  
    for (char ch : word) {  
        if (current->children.find(ch) == current-  
>children.end()) {  
            return false;  
        }  
        current = current->children[ch];  
    }  
  
    return current->isEndOfWord;  
}
```

Tries (ix)

StartsWith Operation

```
bool startsWith(const std::string& prefix) {
    TrieNode* current = root;

    for (char ch : prefix) {
        if (current->children.find(ch) == current-
>children.end()) {
            return false;
        }
        current = current->children[ch];
    }

    return true;
}
```

Tries (x)

Common Applications

- Auto-complete systems
- Spell checkers
- Word games (Scrabble, Boggle)
- IP routing tables
- Phone directory lookups

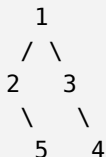
Lab

General tree traversal

Given a `TreeNode*`, return a `std::vector<int>` containing the elements from a binary tree if you were viewing it from the right-side

Visual example:

Example 1:



Right side view: [1, 3, 4]

Explanation: Looking from right side, we see:

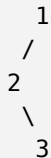
- Level 0: node 1 (rightmost)

Lab (ii)

- Level 1: node 3 (rightmost)
- Level 2: node 4 (rightmost)

Lab (iii)

Example 2:



Right side view: [1, 2, 3]

Explanation: Looking from right side, we see:

- Level 0: node 1 (only node)
- Level 1: node 2 (only node)
- Level 2: node 3 (only node)

Lab (iv)

BST

Given a BSTNode, find the sum of all values within the range [low, high]

```
struct BSTNode {  
    int data;  
    BSTNode *left;  
    BSTNode *right;  
  
    BSTNode(int data): data(data), left(nullptr), right(nullptr)  
    {}  
};  
  
int rangeSumBST(BSTNode *root, int low, int high) {}
```