Trees and Binary Search Trees

Data Structures and Algorithms

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

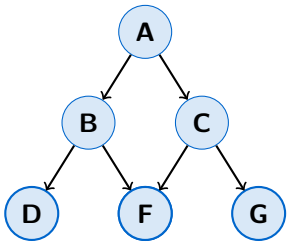
Outline

What is a Tree?

- A **tree** is a non-linear, hierarchical data structure
- Consists of **nodes** connected by **edges**
- Has a single **root** node (topmost)
- Each node can have zero or more **children**
- Nodes with no children are called **leaves**

Key Property

A tree with n nodes has exactly $n - 1$ edges.



Tree Terminology

- **Root**: Top node (no parent)
- **Parent**: Node directly above
- **Child**: Node directly below
- **Siblings**: Nodes with same parent
- **Leaf**: Node with no children
- **Internal node**: Node with at least one child
- **Subtree**: A node and all its descendants

- **Depth** of a node: Number of edges from root to that node
- **Height** of a node: Number of edges on the longest path from that node to a leaf
- **Height of tree**: Height of the root node
- **Level**: Set of nodes at same depth
- **Degree**: Number of children of a node

Real-World Examples of Trees

File Systems

```
/
  home/
    student/
      file.c
  usr/
    bin/
```

Organization Charts

CEO $\rightarrow$ Managers $\rightarrow$ Employees

HTML DOM

```
<html>
  <head>
    <title>
  <body>
    <div>
```

Other Examples

- Family trees
- Decision trees (AI)
- Expression trees
- Network routing

Types of Trees

General Tree Each node can have any number of children

Binary Tree Each node has **at most 2** children (left and right)

Binary Search Tree Binary tree with ordering property

Full Binary Tree Every node has 0 or 2 children

Complete Binary Tree All levels filled except possibly the last (filled left to right)

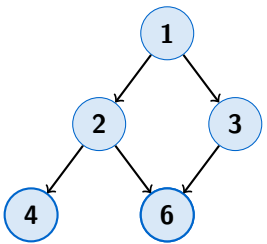
Perfect Binary Tree All internal nodes have 2 children, all leaves at same level

Balanced Tree Height difference of left and right subtrees ≤ 1

- **AVL Tree** Self-balancing BST

Binary Tree

- Each node has **at most two children**
- Children are referred to as **left child** and **right child**
- A binary tree of height h has:
  - At most $2^{h+1} - 1$ nodes
  - At least $h + 1$ nodes
- A binary tree with n nodes has height:
  - At least $\lfloor \log_2 n \rfloor$
  - At most $n - 1$

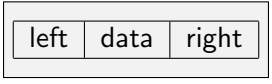


Height $= 2$, Nodes $= 6$

Binary Tree Node in C

```c
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;
}
```



Each node stores data + two pointers

Tree Traversal Methods

Traversal = visiting every node in the tree exactly once.

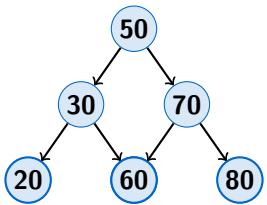
Depth-First (DFS)

1. **Inorder**: Left → Root → Right
2. **Preorder**: Root → Left → Right
3. **Postorder**: Left → Right → Root

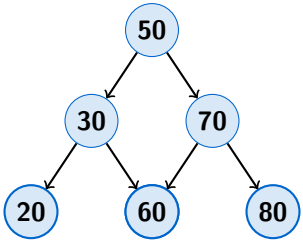
Breadth-First (BFS)

1. **Level Order**: Visit level by level, left to right

Uses a **queue** data structure.



Traversal Example



Traversal	**Output**
Inorder (L, Root, R)	20, 30, 40, **50**, 60, 70, 80
Preorder (Root, L, R)	**50**, 30, 20, 40, 70, 60, 80
Postorder (L, R, Root)	20, 40, 30, 60, 80, 70, **50**
Level Order	**50**, 30, 70, 20, 40, 60, 80

Traversal Code in C

Inorder

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

Preorder

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

Postorder

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

Key Observation

All three DFS traversals use **recursion**. The only difference is *when* we process the root node.

Level Order Traversal (BFS)

```
void levelOrder(struct Node *root) {
    if (root == NULL) return;

    struct Node *queue[100];
    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear) {
        struct Node *current = queue[front++];
        printf("%d ", current->data);

        if (current->left != NULL)
            queue[rear++] = current->left;
        if (current->right != NULL)
            queue[rear++] = current->right;
    }
}
```

How it works

Uses a **queue** (FIFO). Enqueue root, then repeatedly dequeue a node, print it, and enqueue its

What is a BST?

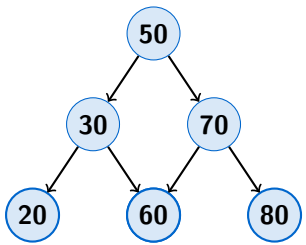
A **Binary Search Tree** is a binary tree where:

1. All values in the **left subtree** are **less than** the root
2. All values in the **right subtree** are **greater than** the root
3. Both left and right subtrees are also BSTs

Key Property

Inorder traversal of a BST gives elements in **sorted order**.

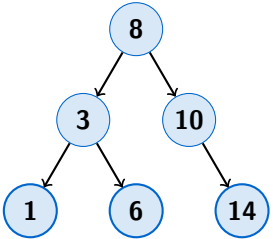
Valid BST:



Inorder: 20, 30, 40, 50, 60, 70, 80

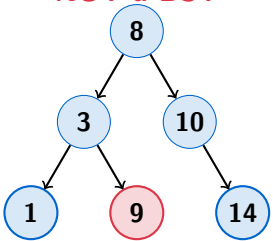
BST vs Non-BST

Valid BST



Left $<$ Root $<$ Right at every node

NOT a BST



9 is in left subtree of 8, but $9 > 8$

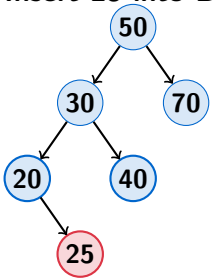
BST Insert

```c
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 if (data > root->data)
        root->right =
            insert(root->right, data);

    return root;
}
```

Insert 25 into BST:



$25 < 50 \rightarrow$ left
$25 < 30 \rightarrow$ left
$25 > 20 \rightarrow$ right
Insert here!

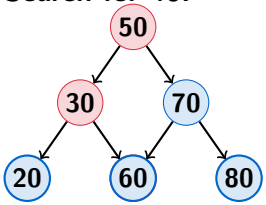
BST Search

```c
struct Node* search(
    struct Node *root, int key)
{
    // Base cases
    if (root == NULL)
        return NULL;    // not found

    if (root->data == key)
        return root;    // found!

    // Recurse left or right
    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}
```

Search for 40:



$40 < 50 \rightarrow$ go left
$40 > 30 \rightarrow$ go right
$40 = 40 \rightarrow$ Found!

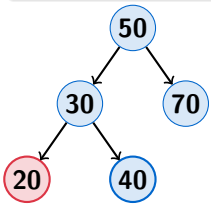
Comparisons: 3

Much fewer than searching all 7 nodes!

BST Find Min and Max

Find Minimum

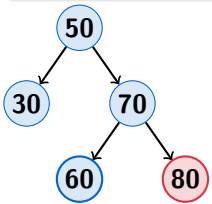
Go to the **leftmost** node.



Min = 20

Find Maximum

Go to the **rightmost** node.



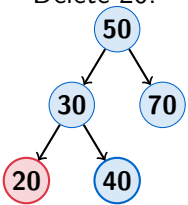
Max = 80

Time Complexity: $O(h)$ where h = height of tree

BST Deletion – Three Cases

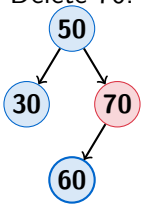
Case 1: Leaf Node
Simply remove it.

Delete 20:



Case 2: One Child
Replace with child.

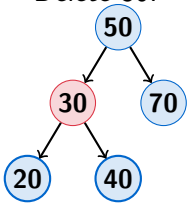
Delete 70:



60 takes 70's place

Case 3: Two Children
Replace with **inorder successor**.

Delete 30:



Replace 30 with 40 (inorder successor)

BST Deletion Code

```
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 {
        // Case 1 & 2: No child or one child
        if (root->left == NULL) {
            struct Node *temp = root->right;
            free(root);
            return temp;
        }
        if (root->right == NULL) {
            struct Node *temp = root->left;
            free(root);
            return temp;
        }
        // Case 3: Two children - find inorder successor
        struct Node *successor = findMin(root->right);
        root->data = successor->data;
        root->right = deleteNode(root->right, successor->data);
```

BST Time Complexity

Operation	Best Case	Average Case	Worst Case
Search	$O(1)$	$O(\log n)$	$O(n)$
Insert	$O(1)$	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(\log n)$	$O(n)$
Find Min	$O(1)$	$O(\log n)$	$O(n)$
Find Max	$O(1)$	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$	$O(n)$

Best: Balanced BST

Height $= O(\log n)$
Insert 4, 2, 6, 1, 3, 5, 7

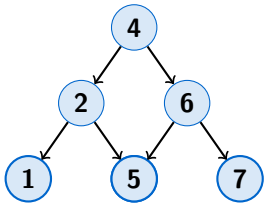
Worst: Skewed BST

Height $= O(n)$
Insert 1, 2, 3, 4, 5, 6, 7

Balanced vs Skewed BST

Balanced BST

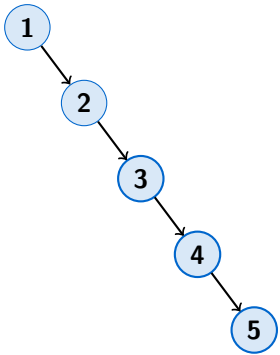
Height $= O(\log n)$



Search for any value: max 3 comparisons

Skewed BST (Degenerate)

Height $= O(n)$



This is essentially a linked list!

Height of BST

```
int height(struct Node *root) {
    if (root == NULL)
        return -1;  // empty tree has height -1

    int leftHeight  = height(root->left);
    int rightHeight = height(root->right);

    if (leftHeight > rightHeight)
        return leftHeight + 1;
    else
        return rightHeight + 1;
}
```

Example

Tree: 50, 30, 70, 20, 40, 60, 80
Height = 2

Recursive idea

Height = 1 + max(height of left, height of right)

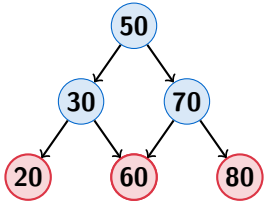
Count Nodes and Leaves

Count All Nodes

```
int countNodes(Node *root) {
    if (root == NULL)
        return 0;
    return 1
        + countNodes(root->left)
        + countNodes(root->right);
}
```

Count Leaf Nodes

```
int countLeaves(Node *root) {
    if (root == NULL)
        return 0;
    if (root->left == NULL &&
        root->right == NULL)
        return 1;
    return countLeaves(root->left)
         + countLeaves(root->right);
}
```



Total nodes = 7, Leaf nodes = 4 (highlighted)

Applications of BST

1. **Databases**: Indexing and fast lookups
2. **File Systems**: Directory organization
3. **Compilers**: Symbol tables for variable lookup
4. **Auto-complete**: Dictionary with prefix search
5. **Priority Queues**: Using heap (special tree)
6. **Routing**: Network routing tables
7. **Expression Evaluation**: Parse trees

Why not just use sorted arrays?

- Arrays: Insert/Delete = $O(n)$, Search = $O(\log n)$
- BST: Insert/Delete/Search = $O(\log n)$ (balanced)
- BSTs are better when data changes frequently

Summary

- Trees are **hierarchical, non-linear** data structures
- Binary trees have at most **2 children** per node
- BST maintains the **ordering property**: left $<$ root $<$ right
- Four traversals: **Inorder, Preorder, Postorder, Level Order**
- BST operations (search, insert, delete) are $O(\log n)$ on average
- **Worst case** $O(n)$ when tree becomes skewed
- Balanced trees (AVL, Red-Black) guarantee $O(\log n)$

What's Next?

Practice: Lab 4 (Trees & BST), Exercises, Homework 7 & 8

Questions?

Practice: `dsacode/binary_search_tree.c`