

Design and Analysis of Binary Search Trees

Implementation, Time complexity, and Real world applications

Team Members:

Subha Laxmi-AP24110010029

Lopa Mudra-AP24110010832

T. Venkata Harshitha - AP24110010901

N. Sreeja - AP24110010902

K. Meghana - AP24110010914

K. Manasa - AP24110010943

TABLE OF CONTENT

01

**PROBLEM
STATEMENT**

02

CREATING BST

03

INSERTION

04

SEARCHING

05

DELETION

06

**TIME
COMPLEXITY
ANALYSIS**

07

**IMPACT ON
TREE SHAPES**

08

**REAL TIME
SCENARIOS**

PROBLEM STATEMENT

OBJECTIVE & OPERATIONS

- Objective: Create a Binary Search Tree (BST) and implement core operations from scratch.
- Insertion
- Deletion
- Searching

ANALYSIS & APPLICATION

- Analysis: Compare Average-Case vs. Worst-Case time complexities.
- Discussion: Analyze how 'Tree Shape' impacts performance.
- Application: Identify real-time scenarios for this data structure

WHAT IS BINARY SEARCH TREE?

01

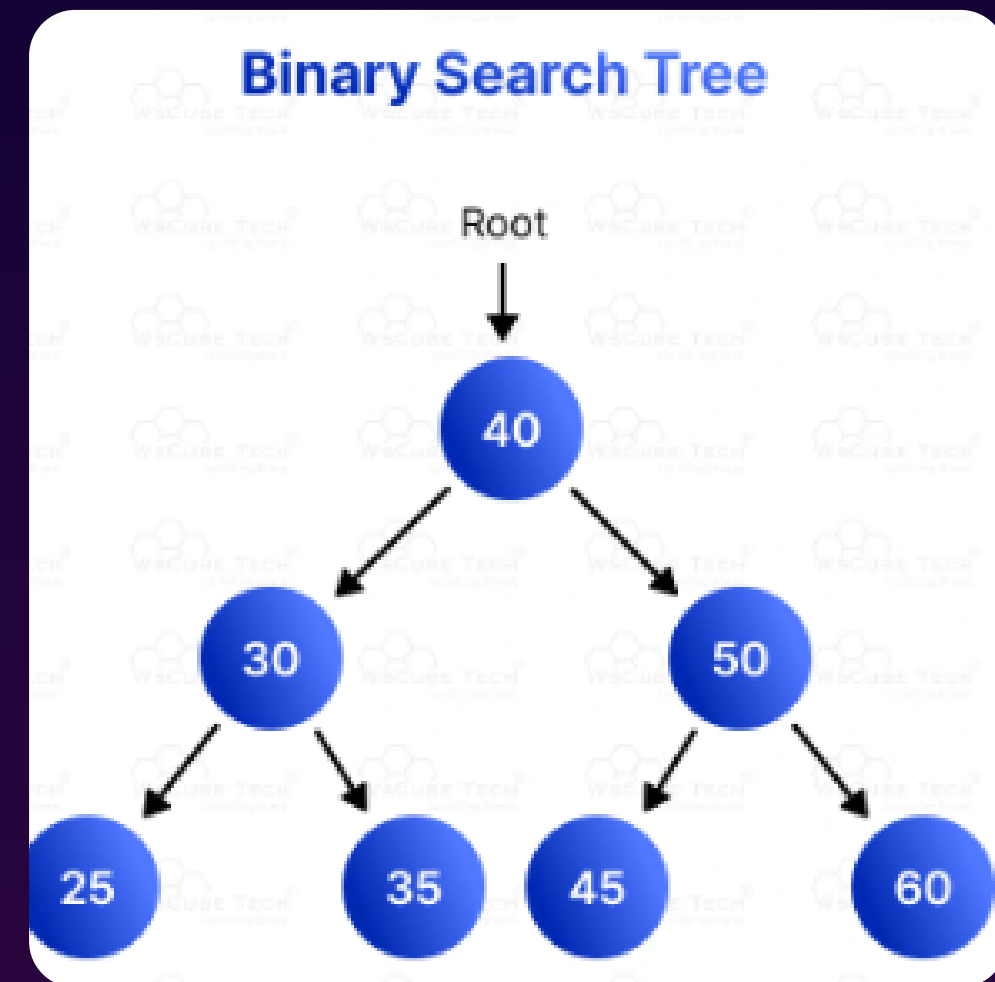
Definition:

A hierarchical data structure consisting of nodes.

02

The BST Property:

- **Left Subtree:** Contains values smaller than the root.
- **Right Subtree:** Contains values larger than the root.



IMPLEMENTATION:

CREATING BST:

The Node Class

Before we perform operations, we must define the building block: the Node.

We also initialize the Tree by setting the root pointer to NULL.

- **Data:** Stores the integer value.
- **Left/Right:** Pointers to child nodes.
- **Constructor:** Initializes pointers to NULL.

```
1 struct Node {  
2     int data;  
3     Node* left;  
4     Node* right;  
5  
6     Node(int val) {  
7         data = val;  
8         left = NULL;  
9         right = NULL;  
10    }  
11 };
```

INSERTION

```
1 Node* insert(Node* node, int val) {
2     if (node == NULL) {
3         return new Node(val);
4     }
5
6     if (val < node->data) {
7         node->left = insert(node->left,
8 val);
9     }
10    else if (val > node->data) {
11        node->right = insert(node->right
12, val);
13    }
14    return node;
15 }
```

- **Algorithm:**

- a. Start at Root.
- b. Compare New Value vs CurrentNode.
- c. If Smaller -> Left. If Larger -> Right.
- d. Repeat until NULL.

- **Example: Inserting 35**

- a. Compare 35 vs Root(40): Smaller -> Go Left.
- b. Compare 35 vs Node(30): Larger -> Go Right.
- c. Right is NULL -> Insert 35 here.

SEARCHING

- **Algorithm:**

- a. Start at Root.
- b. If Target == Node.Data: Found.
- c. If Target < Node.Data: Move Left.
- d. If Target > Node.Data: Move Right.

- **Example:** Search for 60

- a. $60 > 40 \rightarrow$ Right.
- b. $60 > 50 \rightarrow$ Right.
- c. Found 60! (Only 2 comparisons needed).

```
1 Node* search(Node* root, int key) {
2
3     if (root == NULL || root->data ==
4         key) {
5         return root;
6     }
7     if (root->data > key) {
8         return search(root->left, key
9     );
10    }
11    return search(root->right, key);
12 }
```

DELETION

- **Case 1:** Leaf Node (e.g., 25)
- Simply remove it. No structure change.
- **Case 2:** One Child
- Bypass the node (link Grandparent to Child).
- **Case 3:** Two Children (e.g., deleting 30)
- Node 30 has children 25 and 35.
- **Solution:** Replace 30 with its In-Order Successor (35).
- Delete the original 35 (which is now a leaf).

```
1 Node* deleteNode(Node* root, int key) {
2     if (root == NULL) return root;
3
4     if (key < root->data) {
5         root->left = deleteNode(root->left, key);
6     }
7
8     else if (key > root->data) {
9         root->right = deleteNode(root->right, key
10    );
11    }
12    else {
13        if (root->left == NULL) {
14            Node* temp = root->right;
15            delete root;
16            return temp;
17        }
18        else if (root->right == NULL) {
19            Node* temp = root->left;
20            delete root;
21            return temp;
22        }
23        Node* temp = minValueNode(root->right);
24        root->data = temp->data;
25        root->right = deleteNode(root->right, temp
26        ->data);
27    }
28    return root;
29 }
```

TIME COMPLEXITY ANALYSIS

Operation	Average Case	Worst Case	Best Case
Search	$O(\log n)$	$O(n)$	$O(1)$
Insertion	$O(\log n)$	$O(n)$	$O(\log n)$
Deletion	$O(\log n)$	$O(n)$	$O(\log n)$

AVERAGE CASE

S Scenario

Data is inserted in random order
resulting in a "bushy" or
Balanced tree

T The Math

The tree splits roughly in half at
each level.

Height $h \approx \log_2(n)$

C

Since operations are proportional
to height, Complexity = $O(\log n)$.

E

Example: Searching 1 item in
1,000,000 nodes takes only ~ 20
comparisons.

WORST CASE:

Scenario

Data is inserted in Sorted Order (e.g., 25, 30, 35, 40, 45..).

- **Tree Shape:** Skewed
- Every node has only one child, effectively becoming a Linked List.
- **Height $h=n$**
- We must traverse every node to reach the end.
- **Complexity = $O(n)$**

AFFECT OF TREE SHAPES

BEST SHAPE (BALANCED):

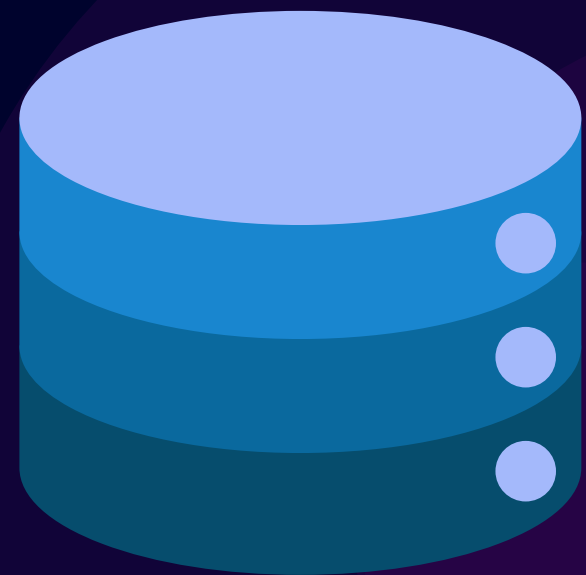
- Nodes fill up level by level.
- Height is minimized:
 $h \approx \log_2 n$.
- Result: Efficient operations.

- Time complexity is not fixed; it is Height dependent($O(h)$).

WORST SHAPE (SKEWED/DEGENERATE):

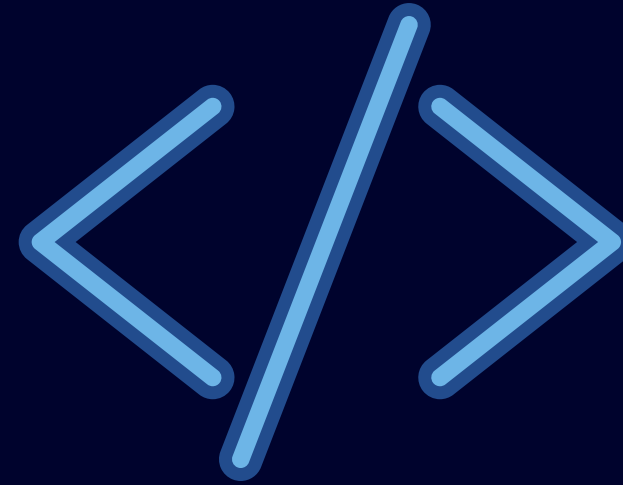
- Nodes form a single line (Left-Skewed or Right-Skewed).
- Height is maximized: $h \approx n$.
- Result: Performance degrades to that of a Linked List.

REAL-TIME SCENARIOS



DATABASE INDEXING

Used to locate records on a drive without scanning the entire disk.



COMPILERS

Abstract Syntax Trees (AST) help understand program structure.



ROUTERS

Routing tables used to determine packet paths based on IP prefixes.

CONCLUSION

- **Summary:**

- BSTs offer efficient $O(\log n)$ search/insert/delete for random data.
- Implementation relies on recursive pointer logic.
- Solves the problem of slow searching in unsorted lists $O(n)$ vs slow insertion in sorted arrays $O(n)$.

- **Solution to Skewing:**

- To fix the worst-case $O(n)$ issue, we use Self-Balancing Trees (like AVL or Red-Black Trees) which automatically rotate nodes to maintain an optimal height of $\log n$.



**THANK
YOU**

FOR YOUR ATTENTION
