Data Structures and Algorithms

Lecture 15

Aniket Basu Roy

2026-02-18 Wed

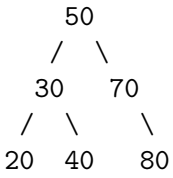
Contents

1 Agenda

1.1 Data Structures

1.2 Binary Search Trees

2 Binary Search Trees



Left Subtree $<$ Node $<$ Right Subtree

3 Binary Search Trees

Functions	Time Complexity
Insert(x)	$O(h)$
Delete(x)	$O(h)$
Find(x)	$O(h)$
ListAllElements()	$\Theta(n)$

4 Today's Topic: Deletion from BST

4.1 Overview

Deletion is more complex than insertion because we need to maintain the BST property after removing a node. We must consider three cases based on the node's children.

4.2 The Three Cases of Deletion

4.2.1 Case 1: Node to be deleted is a leaf (no children)

- Simply remove the node

- Update parent's pointer to NULL

4.2.2 Case 2: Node to be deleted has one child

- Remove the node
- Connect the parent directly to the node's child
- Bypass the deleted node

4.2.3 Case 3: Node to be deleted has two children

- Find the inorder successor (smallest node in right subtree) OR
- Find the inorder predecessor (largest node in left subtree)
- Replace the node's value with successor/predecessor value
- Delete the successor/predecessor (which falls into Case 1 or Case 2)

4.3 Pseudocode for BST Deletion

```
ALGORITHM: DELETE(root, key)
INPUT: root - pointer to root of BST, key - value to delete
OUTPUT: root - pointer to root of modified BST

1. // Base case: empty tree
2. IF root == NULL THEN
3.     RETURN NULL
4. END IF
5.
6. // Recursive search for the node
7. IF key < root.data THEN
8.     root.left = DELETE(root.left, key)
9. ELSE IF key > root.data THEN
10.     root.right = DELETE(root.right, key)
11. ELSE
12.     // Node found! Now handle three cases
13.
14.     // Case 1: Leaf node (no children)
15.     IF root.left == NULL AND root.right == NULL THEN
16.         DELETE root
```

```
17.         RETURN NULL
18.
19.     // Case 2a: Only right child exists
20.     ELSE IF root.left == NULL THEN
21.         temp = root.right
22.         DELETE root
23.         RETURN temp
24.
25.     // Case 2b: Only left child exists
26.     ELSE IF root.right == NULL THEN
27.         temp = root.left
28.         DELETE root
29.         RETURN temp
30.
31.     // Case 3: Both children exist
32.     ELSE
33.         // Find inorder successor (minimum in right subtree)
34.         temp = FIND_MIN(root.right)
35.
36.         // Copy successor's value to current node
37.         root.data = temp.data
38.
39.         // Delete the successor
40.         root.right = DELETE(root.right, temp.data)
41.     END IF
42. END IF
43.
44. RETURN root
```

4.4 Helper Function: Find Minimum

```
ALGORITHM: FIND_MIN(node)
INPUT: node - pointer to a node in BST
OUTPUT: pointer to node with minimum value

1. WHILE node.left != NULL DO
2.     node = node.left
3. END WHILE
4. RETURN node
```

4.5 Time Complexity Analysis

4.5.1 Best Case: $O(logn)$

- Occurs when tree is balanced
- We traverse height of tree once
- Height of balanced BST = $\log_2(n)$

4.5.2 Worst Case: $O(n)$

- Occurs when tree is skewed (resembles linked list)
- We may need to traverse all nodes
- Height of skewed BST = n

4.6 Detailed Complexity Breakdown

Operation	Best Case	Worst Case
Search	$O(logn)$	$O(n)$
Delete	$O(logn)$	$O(n)$

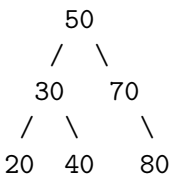
4.6.1 Why $O(h)$ where h is height?

1. Search for node: $O(h)$
2. Find successor (Case 3 only): $O(h)$
3. Delete successor: $O(h)$

Total: $O(h) + O(h) + O(h) = O(h)$

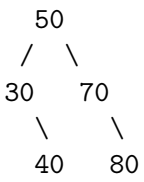
4.7 Example Walkthrough

Let's delete node with value 50 from:



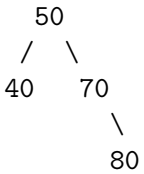
4.7.1 Deleting 20 (Case 1: Leaf)

Result:



4.7.2 Deleting 30 (Case 2: One child)

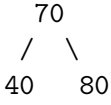
Result:



4.7.3 Deleting 50 (Case 3: Two children)

- Find inorder successor: 70
- Replace 50 with 70
- Delete old 70 node

Result:



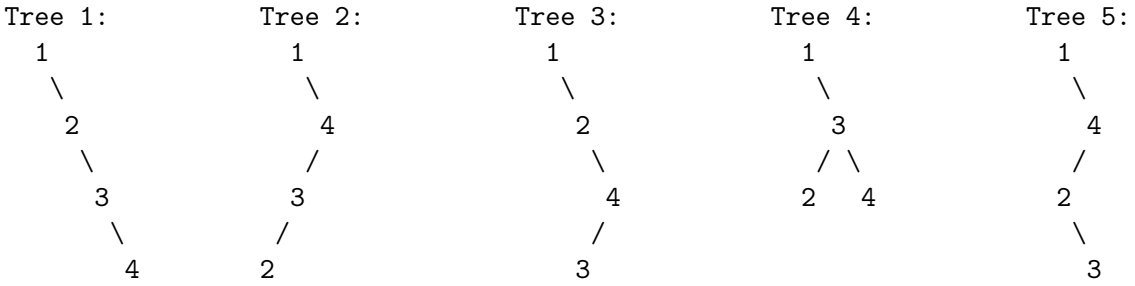
4.8 Enumeration of All Possible BSTs with Keys {1, 2, 3, 4}

The number of structurally different BSTs with n nodes is given by the n^{th} Catalan number: $C_n = (2n)!/((n+1)!n!)$

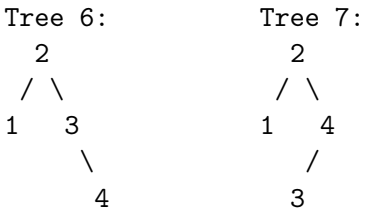
For $n = 4$: $C_4 = 14$

Below are all 14 possible BSTs:

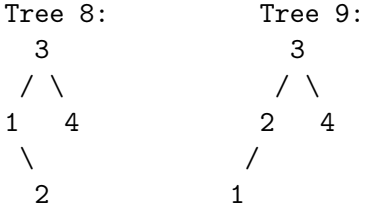
4.8.1 Trees with Root = 1 (5 trees)



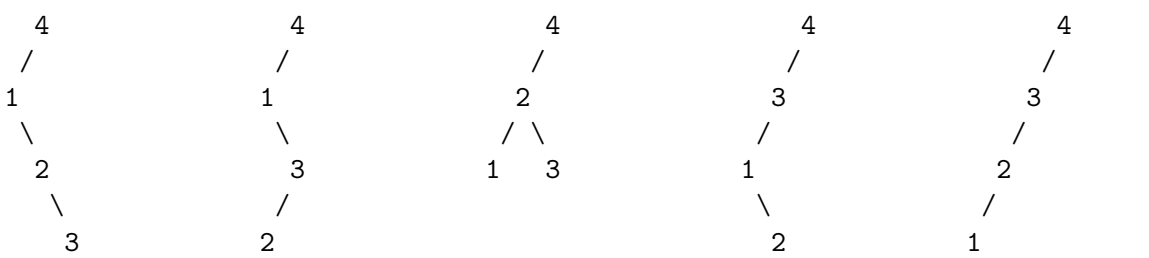
4.8.2 Trees with Root = 2 (2 trees)



4.8.3 Trees with Root = 3 (2 trees)



4.8.4 Trees with Root = 4 (5 trees)



5 Questions

1. Given a preorder sequence, can we uniquely construct the corresponding BST?

2. Show bijection between preorder sequences and balanced parantheses?