Chapter 18 Binary Search Tree

Contents

Syllabus Learning Outcomes

1.3 Data Structures

Understand concept and write algorithms for stack, queue (linear and circular), linear linked list and binary tree (including binary search tree).

1.3.6 Create, update (edit, insert, delete) and search operations for a binary tree (including binary search tree).
Exclude: edit and deletion of nodes from binary search tree

1.3.7 Understand pre-order, in-order and post-order tree traversals; and application of in-order tree traversal for a binary tree.

2.3 Implementing Algorithms and Data Structures

Use programming language elements and constructs to implement sort and search algorithms such as insertion sort, bubble sort, quicksort, merge sort, linear search, binary search and hash table search, as well as data structures such as stacks, queues, linear linked lists and binary search trees.

2.3.2 Implement search programs.
- – Linear search
- – Binary search
- – Hash table search

2.3.3 Write programs to implement operations for stacks, queues (linear and circular), linear linked lists and binary search trees.
Exclude: doubly-linked list and circular linked list

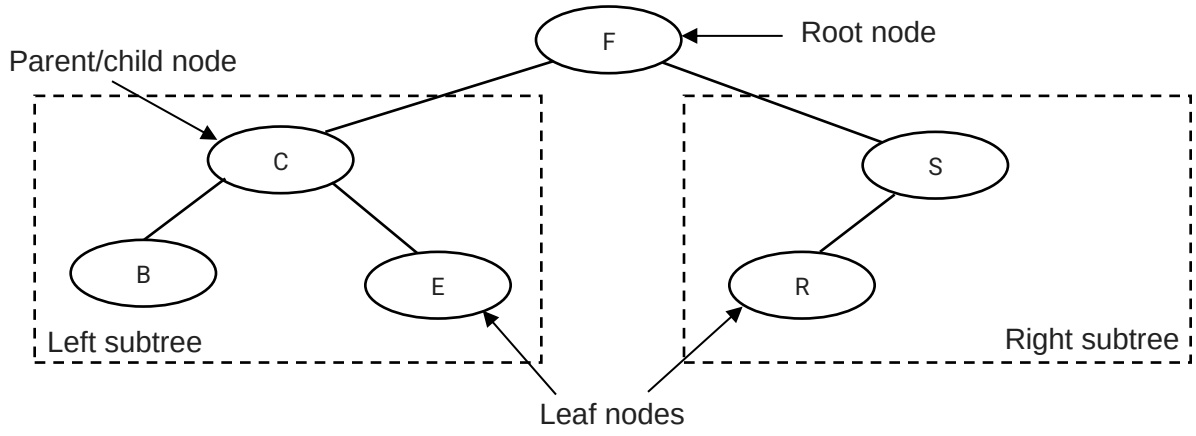
1 Binary Search Trees

In the real world, we draw tree structures to represent hierarchies. For example, we can draw a family tree showing ancestors and their children.

A **binary tree** is different from a family tree. It is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. This structure allows for efficient searching, insertion, and deletion operations.

A **binary search tree** is a specific type of binary tree with an additional property: for each node, the values of all nodes in its left subtree are less than or equal to the node's value, and the values of all nodes in its right subtree are greater than the node's value. This ordering property makes binary search trees particularly useful for searching and retrieval operations.

Let's look at a conceptual diagram of a binary search tree.



A binary search tree is a non-linear data structure represented using **nodes** connected by **edges**. A tree's topmost node is called a **root node**. Each node (apart from the root) in a tree that has at least one sub-node of its own is called a **parent node**. All nodes have one parent except the root node, which has no parent. Each node can have zero, one, or two children, typically named as the **left child** and **right child**. A node with no child, typically the bottom-most node is also called a **leaf node**. For each node, the values of its left descendent nodes are less than that of the current node, which in turn is less than the right descendent nodes (if any).

A binary search tree (BST) is built on the idea of the binary search algorithm which allows for fast lookup, insertion, and removal of nodes. Data stored in the whole tree can be printed out in sequence.

2 OOP implementation of a binary search tree

2.1 Node object

A binary search tree implementation is typically by object-oriented programming, where a node will have the following attributes:

- data
- pointer to left child
- pointer to right child

Node

2.2 Insertion operation

The rules to follow to create a binary search tree that can be easily and quickly searched for a given data:

- Place the first data in the root node.
- Take each subsequent data in turn.
- Start at the root node each time. If the data is less than data in the root node, branch to the left, and if it is greater than the data in the root node, branch to the right.
- Apply the rule at each node encountered.

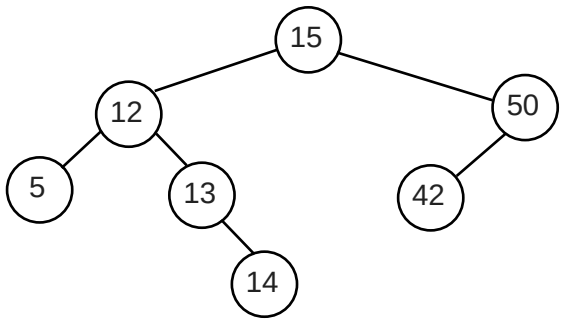
This automatically sorts the data as they are being added.

Pseudocode for insertion:

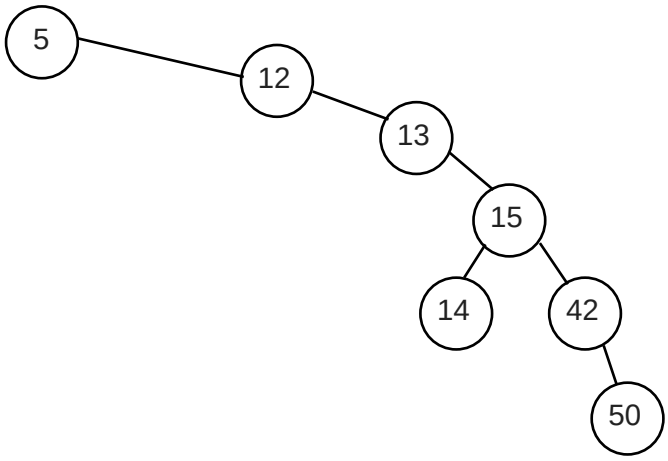
```
IF root is NONE
  THEN
    root ← Node(data)
  ELSE
    parent ← root

    WHILE data not inserted DO
      IF data < parent's data // branch left
        THEN
          IF left branch is empty
            THEN
              parent's left pointer ← Node(data)
            ELSE
              parent ← left node
          ENDIF

        ELSE // branch right, we assume no duplicate data
          IF right branch is empty
            THEN
              parent's right pointer ← Node(data)
            ELSE
              parent ← right node
          ENDIF
      ENDIF
    ENDWHILE
ENDIF
```

Insertion walkthrough:

Example 1

Values to be added	Binary Search Tree
1. 15 2. 12 3. 5 4. 50 5. 42 6. 13 7. 14	

Example 2

Values to be added	Binary Search Tree
1. 5 2. 12 3. 13 4. 15 5. 42 6. 50 7. 14	

2.3 Search operation

The search operation in a binary search tree is similar to the binary search algorithm. In binary search, we will be given a sorted array and we have to search for an element so we start with finding the middle element in the array and compare it with the element to be searched. If the element to be searched is equal to the middle element then we will stop and simply return that element. But if the element to be searched is smaller than the middle element we will discard the right subarray as the elements are in sorted form so we can easily say that the element will be present in the left subarray and if the element is found to be greater than the middle element we will search in right subarray discarding the left subarray. So in this way, we will keep on reducing the array size in half till we get our target element(element to be searched) or we are left with only one element.

The same procedure is applicable in the case of binary search tree. In this case, we will be given a tree and a target value that needs to be searched. We will start comparing that element with the root node's value. If the element to be searched is equal to the root node's value, we will simply stop and return that element as it is successfully searched. But if the element is smaller than the root node's value, we will discard the right subtree of the root node as after learning the properties of the binary search tree we can say that the element needs to be searched in the left subtree as all the node values in left subtree will be smaller than the root node value. If the element is greater than the root node's value, we will discard the left subtree of the root node because all the nodes in the right subtree have values greater than the root node value so we will search for the element in the right subtree. This way, we will keep on reducing the size of the binary search tree till we find the element that needs to be searched or we are left with one node only. We can see that the procedure is the same as what we have done in the binary search algorithm, and this is the reason for the name Binary Search Tree.

The search operation can be implemented using the following algorithm:

```
current_node ← root
REPEAT
  IF data < current_node.data
    THEN
      current_node ← current_node.left
    ELSE
      current_node ← current_node.right
  ENDIF

UNTIL current_node.data EQUALS data OR
      Current_node has no corresponding child node
```

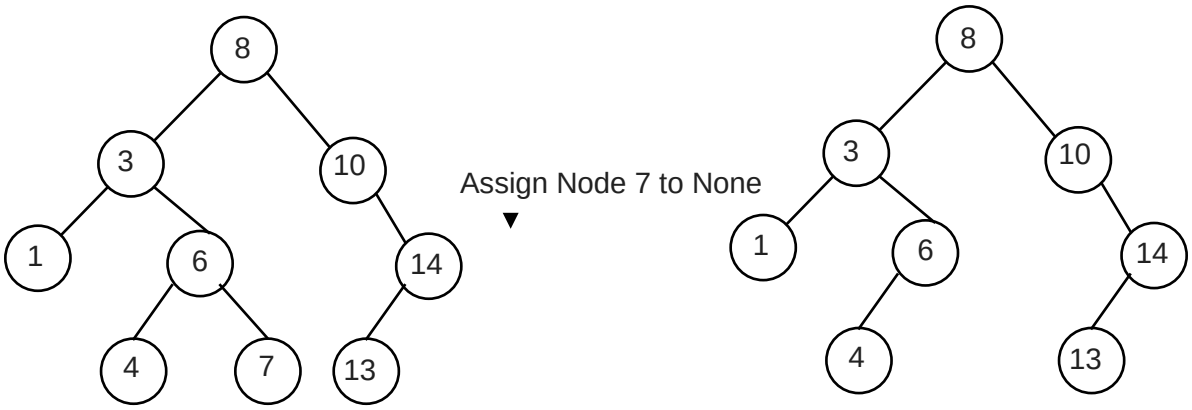
2.4 Deletion operation [no coding required]

Deletion of a node from the binary search tree is dependent on how many child nodes this node has.

There are 3 scenarios to consider.

Case 1: Deletion of a leaf node (no children)

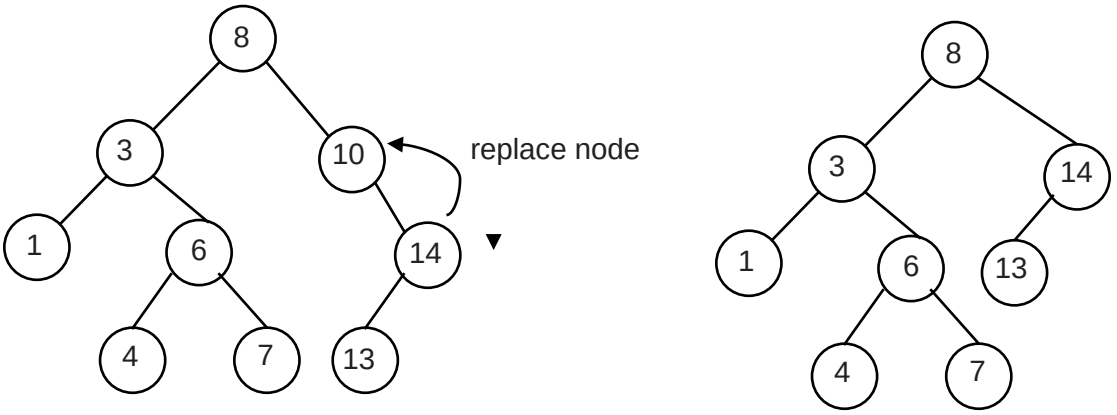
Suppose we want to remove a leaf node, say Node 7. We can simply remove Node 7 by assigning Node 7 to None without any extra steps.



Delete Node 7 Deleted Node 7

Case 2: Deletion of a node with a single child (1 child)

Suppose we want to remove a node with only 1 child node, say Node 10. We will replace Node 10 with its child node, Node 14. This includes Node 14 and its descendants.

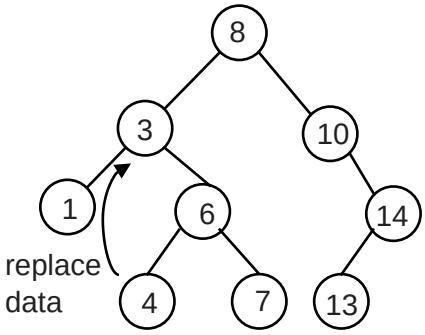


Case 3: Deletion of a node with 2 child nodes (2 children)

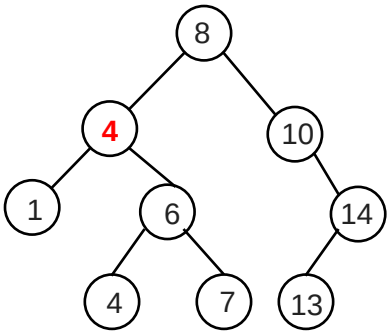
The trick is to look for the next larger value (in-order successor, the leftmost node in the right subtree). The next larger value can be obtained by finding the minimum value in the right child of the node. Replace the contents of the node with the next larger value and delete the node of the next larger value.

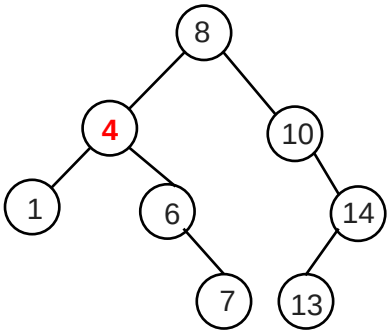
Suppose we want to remove Node 3 which has 2 children.

We will replace the data of Node 3 with data of Node 4 and delete Node 4 by applying the deletion algorithm from either case 1 or 2.



Replace 3 with 4

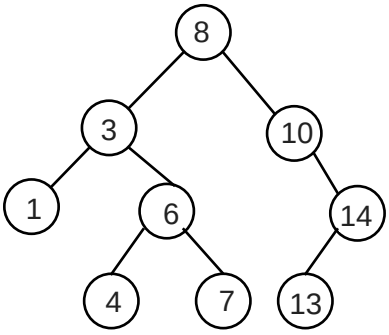




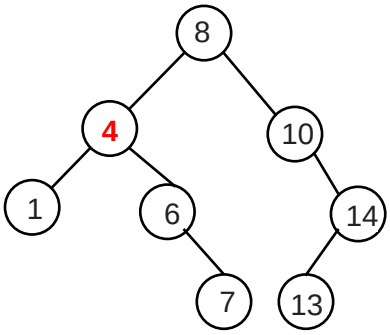
Delete Node 4 (case 1: leaf node)

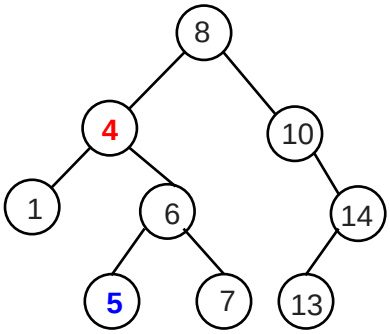
2.5 Update operation [no coding required]

To update the value of a node, say from 3 to 5, we can simply delete Node 3 and insert Node 5.



Delete Node 3





Insert Node 5

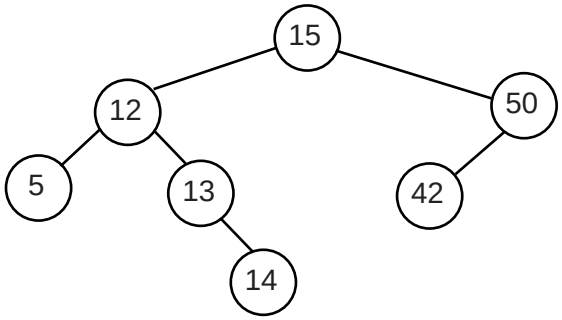
3 Array implementation of a binary tree

Similar to linked lists, we can construct a binary search tree using either OOP or array implementation.

In an array implementation, the pointers store the index of the array where the child nodes are being stored. None or -1 can be used to indicate no further nodes. The implementation using an array could be represented in a table.

Using None as null pointers, when adding 15, 12, 50, 5, 42, 13, 14 in this order to a BST, the table may look like this:

Index	Data	Left Pointer	Right Pointer
0	15	1	2
1	12	3	5
2	50	4	None
3	5	None	None
4	42	None	None
5	13	None	6
6	14	None	None



4 Time Complexity of Binary search tree for Search Operation

When there is a balanced binary search tree (a binary search tree is called balanced if height difference of nodes on left and right subtree is not more than one), so height becomes log *n* where *n* is the number of nodes in a tree.

In a search operation, we will keep on traversing through nodes one by one. Suppose if we find the element in the second level, there would have been 2 comparisons, if we get the element in the third level, there would have been 3 comparisons. In this way, we can say that the time taken to search for a target element in a binary search tree is the same as the height of the tree which is log *n*, so the time complexity for searching is Ω(log *n*).

If the tree is unbalanced or if it is a skewed binary search tree, we have to traverse from the root node to the deepest leaf node, and in that case, the height of the tree becomes *n,* and hence, time complexity in the worst case becomes O(*n*).

The difference between Ω(log *n*) and O(*n*) is significant for large *n*. To maintain the search performance, sometimes we may need to rebalance the tree. This involves re-creating the binary search tree with the same data items. The data items are inserted in an order such that the height difference of nodes on the left and right subtree is not more than one.

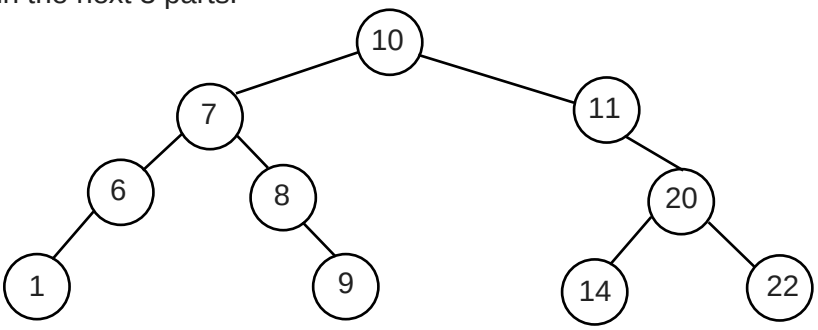
5 Tree traversal

Traversing a tree means visiting every node in the tree. There are different strategies or algorithms for traversing trees, and the choice of traversal order depends on the specific requirements of the task at hand. All traversals start at the root node.

The three main types of tree traversal are:

- pre-order
- in-order
- post-order

Let's use the binary search tree below to display the sequence of data using the different types of traversal in the next 3 parts.



5.1 Pre-order traversal

In a pre-order traversal, the current node is visited before exploring its left and right subtrees. This can be denoted as **Node**-**Left**-**Right**.

Pseudocode	Pre-order sequence

FUNCTION preorder(node)
  IF node is not null
  THEN
  OUTPUT node.value
  preorder(node.left)
  preorder(node.right)
  ENDIF
ENDFUNCTION
	10, 7, 6, 1, 8, 9, 11, 20, 14, 22

5.2 In-order traversal

In an in-order traversal, the left subtree is visited first, followed by the current node, and then the right subtree. This can be denoted as **Left**- **Node**-**Right**.

This results in nodes being visited in ascending order in the case of a binary search tree.

Pseudocode	In-order sequence

FUNCTION inorder(node)
  IF node is not null
  THEN
  inorder(node.left)
  OUTPUT node.value
  inorder(node.right)
  ENDIF
ENDFUNCTION
	1, 6, 7, 8, 9, 10, 11, 14, 20, 22

5.3 Post-order traversal

In a post-order traversal, the left and right subtrees are visited before the current node. This can be denoted as **Left**-**Right**-**Node**.

Pseudocode	Post-order sequence

FUNCTION postorder(node)
  IF node is not null
  THEN
  postorder(node.left)
  postorder(node.right)
  OUTPUT node.value
  ENDIF
ENDFUNCTION
	1, 6, 9, 8, 7, 14, 22, 20, 11, 10