# Binary Search Tree:

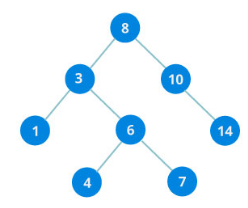
**Uses**: Linked List to perform and why?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | Array | | Linked List | |
| Time Complexity | Space complexity | Time Complexity | Space complexity |
| Creation | O (1) | O (n) | O (1) | O (1) |
| Insertion | O (1) | O (1) | O (n) | O (n) |
| Deletion | O (n) | O (1) | O (n) | O (n) |
| Searching | O (n) | O (1) | O (n) | O (n) |
| Traversal | O (n) | O (1) | O (n) | O (n) |
| Space Efficient |  | No |  | **Yes** |

## What is Binary search tree?

Binary search tree follows similar properties to Binary tree with some additional features

* Left sub-trees of a node should always be lesser than the node
* Right sub-trees of a node should always be more than the node



## Why Binary Search tree?

Reduction of time to Log n.

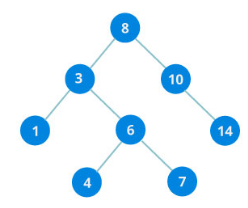
# Creation and Searching of BST

**Creation**

CreateBST()

Create root with null

**Time Complexity: O(1); Space Complexity: O(1)**



## Search in BST:

SearchBST(root, value)

If Root = null

Return Error

Else if root = value

Print root

Else if root < value

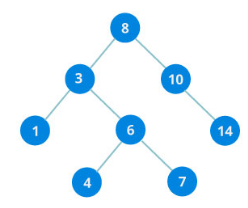
SearchBST(root.left, value)

Else if root > value

SearchBST(root.right , value)

**Time Complexity: O( log n); Space Complexity: O(log n)**

# Traversal of BST



**Logic**: Pre, post, In Order ; Also Level order.

Preorder(root)

If root=null

Return error

Else

Print root

Preorder(root.left)

Preorder(root.right)

LevelOrder(root)

Create enqueuer(Q)

Enqueue (root)

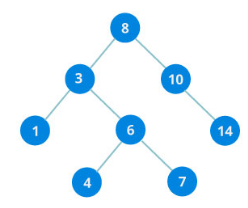
While queue is not empty

Enqueuer() # Child of first element

Dequeuer() and print

**Time Complexity: O(n); Space Complexity: O(n)**

# Insertion of BST



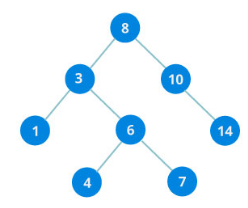
**Conditions:**

BST is blank

BST is not blank

**Example Insert**: 2, 12

# Insertion of BST



**Algorithm**

**InsertBST(current, value)**

**If current = null**

**Create a node and inset a value**

**Else if value <= current**

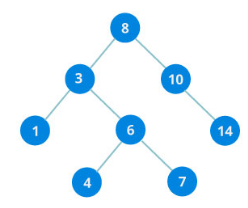
**Current.left = InsertBST(current.left, value)**

**Else**

**Current.right = InsertBST(current. right, value)**

**Return current**

# Deletion of BST

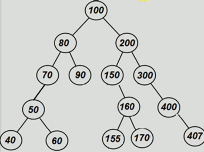


**Conditions:**

Node deleted is a left node

Node having 1 children

Node having 2 childern



BST is not blank

**Example Delete**: 1, 8, 10

**Algorithm:**

DeleteBST(root, node)

If root = null

Return Null

Else if node< root

Then root.left = DeleteBST (root.left, node)

Else if node> root

Then root.right = DeleteBST(root.right, node)

Else

If root have children, find minimum element from right sub-tree

Replace currentnode with min node at the right and delte the min node from right

Else if node has only one child on left

Then root = root.left

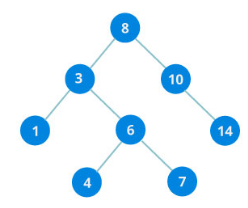
Else if node has only one child on right

Then root = root.right

Else

Root=null

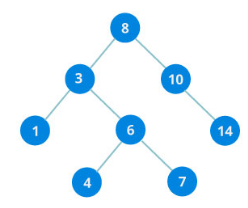
Return root



# Delete Entire Tree

DeleteBST()

Root=null



|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Space complexity |
| Creation | O (1) | O (1) |
| Insertion | O (Log n) | O (Log n) |
| Deletion | O (Log n) | O (Log n) |
| Searching | O (Log n) | O (Log n) |
| Traversal | O (n) | O (n) |