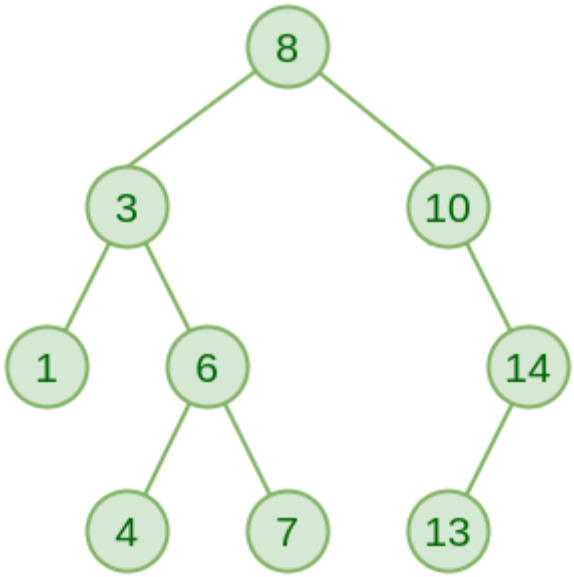
Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

Handling approach for Duplicate values in the Binary Search tree:

- You can not allow the duplicated values at all.
- We must follow a consistent process throughout i.e either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.
- We can keep the counter with the node and if we found the duplicate value, then we can increment the counter.

Below are the various operations that can be performed on a BST:

- **Insert a node into a BST:** A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

```
import java.io.*;

// Java program for Inserting a node
```

```
class GFG {



    // Given Node
    static class node {
        int key;
        node left, right;
    };



    // Function to create a new BST node
    static node newNode(int item)
    {
        node temp = new node();
        temp.key = item;
        temp.left = temp.right = null;
        return temp;
    }



    // Function to insert a new node with
    // given key in BST
    static node insert(node node, int key)
    {
        // If the tree is empty, return a new node
```

```
    if (node == null)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node.key) {
        node.left = insert(node.left, key);
    }
    else if (key > node.key) {
        node.right = insert(node.right, key);
    }

    // Return the node
    return node;
}

// Function to do inorder traversal of BST
static void inorder(node root)
{
    if (root != null) {
        inorder(root.left);
        System.out.print(" " + root.key);
        inorder(root.right);
```

```
    }
}


// Driver Code
public static void main(String[] args)
{


    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80
     */
    node root = null;


    // inserting value 50
    root = insert(root, 50);


    // inserting value 30
    insert(root, 30);
```

```
        // inserting value 20
        insert(root, 20);

        // inserting value 40
        insert(root, 40);

        // inserting value 70
        insert(root, 70);

        // inserting value 60
        insert(root, 60);

        // inserting value 80
        insert(root, 80);

        // print the BST
        inorder(root);
    }
}
```

Output

```
20 30 40 50 60 70 80
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Inorder traversal: In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. We visit the left child first, then the root, and then the right child.

```
import java.io.*;

 // Java program for Inorder Traversal

class GFG {



    // Given Node node

    static class node {

        int key;

        node left, right;

    };



    // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }
```

```
// Function to insert a new node with

// given key in BST

static node insert(node node, int key)

{

    // If the tree is empty, return a new node

    if (node == null)

        return newNode(key);



    // Otherwise, recur down the tree

    if (key < node.key) {

        node.left = insert(node.left, key);

    }

    else if (key > node.key) {

        node.right = insert(node.right, key);

    }



    // Return the node

    return node;

}

 // Function to do inorder traversal of BST

static void inorder(node root)
```

```
{
    if (root != null) {
        inorder(root.left);
        System.out.print(" " + root.key);
        inorder(root.right);
    }
}


// Driver Code
public static void main(String[] args)
{

    /* Let us create following BST
             50
          /     \
         30      70
        /  \    /  \
      20   40  60   80
 */
    node root = null;

    // inserting value 50
```

```
root = insert(root, 50);


// inserting value 30

insert(root, 30);


// inserting value 20

insert(root, 20);


// inserting value 40

insert(root, 40);


// inserting value 70

insert(root, 70);


// inserting value 60

insert(root, 60);


// inserting value 80

insert(root, 80);


// print the BST
```

```
        inorder(root);

    }

}
```

Output

```
20 30 40 50 60 70 80
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Preorder traversal: Preorder traversal first visits the root node and then traverses the left and the right subtree. It is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

```
import java.io.*;

 // Java program for Preorder Traversal

class GFG {

     // Given Node node

    static class node {

        int key;

        node left, right;

    };



    // Function to create a new BST node

    static node newNode(int item)
```

```
{
    node temp = new node();
    temp.key = item;
    temp.left = temp.right = null;
    return temp;
}

// Function to insert a new node with
// given key in BST
static node insert(node node, int key)
{
    // If the tree is empty, return a new node
    if (node == null)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node.key) {
        node.left = insert(node.left, key);
    }
    else if (key > node.key) {
        node.right = insert(node.right, key);
    }
```

```
    // Return the node
    return node;
}

// Function to do preorder traversal of BST
static void preOrder(node root)
{
    if (root != null) {
        System.out.print(root.key + " ");
        preOrder(root.left);
        preOrder(root.right);
    }
}

// Driver Code
public static void main(String[] args)
{

    /* Let us create following BST
             50
          /     \
```

```
          30      70
        /  \    /  \
      20   40  60   80
*/
    node root = null;

    // inserting value 50
    root = insert(root, 50);

    // inserting value 30
    insert(root, 30);

    // inserting value 20
    insert(root, 20);

    // inserting value 40
    insert(root, 40);

    // inserting value 70
    insert(root, 70);
```

```
        // inserting value 60

        insert(root, 60);



        // inserting value 80

        insert(root, 80);



        // print the BST

        preOrder(root);

    }

}
```

Output

```
 50 30 20 40 70 60 80
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Postorder traversal: Postorder traversal first traverses the left and the right subtree and then visits the root node. It is used to delete the tree. In simple words, visit the root of every subtree last.

Output

```
 20 40 30 60 80 70 50
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Level order traversal: Level order traversal of a BST is breadth first traversal for the tree. It visits all nodes at a particular level first before moving to the next level.

```
import java.io.*;
```

```
// Java program for Level Order Traversal

class GFG {

    // Given Node node

    static class node {

        int key;

        node left, right;

    };



    // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }

    // Function to insert a new node with

    // given key in BST

    static node insert(node node, int key)

    {

        // If the tree is empty, return a new node

        if (node == null)
```

```
            return newNode(key);

        // Otherwise, recur down the tree

        if (key < node.key) {

            node.left = insert(node.left, key);

        }

        else if (key > node.key) {

            node.right = insert(node.right, key);

        }

        // Return the node

        return node;

    }

    // Returns height of the BST

    static int height(node node)

    {

        if (node == null)

            return 0;

        else {

            // Compute the depth of each subtree

            int lDepth = height(node.left);

            int rDepth = height(node.right);
```

```
            // Use the larger one
            if (lDepth > rDepth)
                return (lDepth + 1);
            else
                return (rDepth + 1);
        }
    }

    // Print nodes at a given level
    static void printGivenLevel(node root, int level)
    {
        if (root == null)
            return;
        if (level == 1)
            System.out.print(" " + root.key);
        else if (level > 1) {
            // Recursive Call
            printGivenLevel(root.left, level - 1);
            printGivenLevel(root.right, level - 1);
        }
    }

    // Function to line by line print
    // level order traversal a tree
```

```
static void printLevelOrder(node root)
{
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++) {
        printGivenLevel(root, i);
        System.out.println();
    }
}
 // Driver Code
public static void main(String[] args)
{
     /* Let us create following BST
              50
           /     \
         30       70
        /  \     /  \
      20   40   60   80
 */
    node root = null;
     // inserting value 50
    root = insert(root, 50);
```

```
        // inserting value 30
        insert(root, 30);
         // inserting value 20
        insert(root, 20);
         // inserting value 40
        insert(root, 40);
         // inserting value 70
        insert(root, 70);
         // inserting value 60
        insert(root, 60);
         // inserting value 80
        insert(root, 80);
         // Function Call
        printLevelOrder(root);
    }
}
```

Output

```
50
30  70
20  40  60  80
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Print nodes at given Level : It prints all the nodes at a particular level of the BST.
-

```
import java.io.*;

// Java program for Printing nodes at given level

class GFG {

    // Given Node node

    static class node {

        int key;

        node left, right;

    };

    // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }

    // Function to insert a new node with

    // given key in BST

    static node insert(node node, int key)
```

```
{
    // If the tree is empty, return a new node
    if (node == null)
        return newNode(key);
     // Otherwise, recur down the tree
    if (key < node.key) {
        node.left = insert(node.left, key);
    }
    else if (key > node.key) {
        node.right = insert(node.right, key);
    }
     // Return the node
    return node;
}
 // Print nodes at a given level
static void printGivenLevel(node root, int level)
{
    if (root == null)
        return;
    if (level == 1)
        System.out.print(" " + root.key);
    else if (level > 1) {
```

```
        // Recursive Call
        printGivenLevel(root.left, level - 1);
        printGivenLevel(root.right, level - 1);
    }
}
 // Driver Code
public static void main(String[] args)
{
     /* Let us create following BST
              50
           /     \
         30      70
        /  \    /  \
      20   40  60   80
 */
    node root = null;
     // inserting value 50
    root = insert(root, 50);
     // inserting value 30
    insert(root, 30);
```

```
        // inserting value 20

        insert(root, 20);

        // inserting value 40

        insert(root, 40);

        // inserting value 70

        insert(root, 70);

        // inserting value 60

        insert(root, 60);

        // inserting value 80

        insert(root, 80);

        // Function Call

        printGivenLevel(root, 2);

    }

}
```

Output

```
30 70
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Print all leaf nodes: A node is a leaf node if both left and right child nodes of it are NULL.
-

```
import java.io.*;

// Java program for Printing all leaf nodes
```

```
class GFG {

    // Given Node node
    static class node {
        int key;
        node left, right;
    };

    // Function to create a new BST node
    static node newNode(int item)
    {
        node temp = new node();
        temp.key = item;
        temp.left = temp.right = null;
        return temp;
    }

    // Function to insert a new node with
    // given key in BST
    static node insert(node node, int key)
    {
        // If the tree is empty, return a new node
        if (node == null)
            return newNode(key);

        // Otherwise, recur down the tree
```

```
        if (key < node.key) {
            node.left = insert(node.left, key);
        }
        else if (key > node.key) {
            node.right = insert(node.right, key);
        }
        // Return the node
        return node;
    }
    // Function to print leaf nodes
    // from left to right
    static void printLeafNodes(node root)
    {
        // If node is null, return
        if (root == null)
            return;
        // If node is leaf node,
        // print its data
        if (root.left == null && root.right == null) {
            System.out.print(" " + root.key);
            return;
        }
```

```
        // If left child exists,
        // check for leaf recursively
        if (root.left != null)
            printLeafNodes(root.left);
        // If right child exists,
        // check for leaf recursively
        if (root.right != null)
            printLeafNodes(root.right);
    }
    // Driver Code
    public static void main(String[] args)
    {
        /* Let us create following BST
                  50
               /     \
              30      70
             /  \    /  \
            20   40  60   80
     */
        node root = null;
```

```
        // inserting value 50
        root = insert(root, 50);
         // inserting value 30
        insert(root, 30);
         // inserting value 20
        insert(root, 20);
         // inserting value 40
        insert(root, 40);
         // inserting value 70
        insert(root, 70);
         // inserting value 60
        insert(root, 60);
         // inserting value 80
        insert(root, 80);
         // Function Call
        printLeafNodes(root);
    }
}
```

Output

```
20 40 60 80
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Print all non leaf node: A node is a non leaf node if either of its left or right child nodes are not NULL.

```
import java.io.*;

// Java program for Printing all non leaf nodes

class GFG {

    // Given Node node

    static class node {

        int key;

        node left, right;

    };

    // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }

    // Function to insert a new node with

    // given key in BST

    static node insert(node node, int key)
```

```
{
    // If the tree is empty, return a new node
    if (node == null)
        return newNode(key);
     // Otherwise, recur down the tree
    if (key < node.key) {
        node.left = insert(node.left, key);
    }
    else if (key > node.key) {
        node.right = insert(node.right, key);
    }
     // Return the node
    return node;
}
 // Function to print all non-leaf
// nodes in a tree
static void printNonLeafNode(node root)
{
    // Base Cases
    if (root == null
        || (root.left == null && root.right == null))
        return;
```

```
    // If current node is non-leaf,
    if (root.left != null || root.right != null) {
        System.out.print(" " + root.key);
    }
     // If root is Not NULL and its one
    // of its child is also not NULL
    printNonLeafNode(root.left);
    printNonLeafNode(root.right);
}
 // Driver Code
public static void main(String[] args)
{
     /* Let us create following BST
              50
           /     \
         30      70
        /  \    /  \
      20   40  60   80
 */
    node root = null;
```

```
        // inserting value 50
        root = insert(root, 50);
        // inserting value 30
        insert(root, 30);
        // inserting value 20
        insert(root, 20);
        // inserting value 40
        insert(root, 40);
        // inserting value 70
        insert(root, 70);
        // inserting value 60
        insert(root, 60);
        // inserting value 80
        insert(root, 80);
        // Function Call
        printNonLeafNode(root);
    }
}
```

Output

```
 50 30 70
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Height of BST: It is recursively calculated using height of left and right subtrees of the node and assigns height to the node as max of the heights of two children plus 1.

```
import java.io.*;

 // Java program for Height of BST

class GFG {

     // Given Node node

    static class node {

        int key;

        node left, right;

    };

     // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }

     // Function to insert a new node with

    // given key in BST

    static node insert(node node, int key)
```

```
{

    // If the tree is empty, return a new node

    if (node == null)

        return newNode(key);

     // Otherwise, recur down the tree

    if (key < node.key) {

        node.left = insert(node.left, key);

    }

    else if (key > node.key) {

        node.right = insert(node.right, key);

    }

     // Return the node

    return node;

}

 // Returns height of the BST

static int height(node node)

{

    if (node == null)

        return 0;

    else {

         // Compute the depth of each subtree

        int lDepth = height(node.left);
```

```
            int rDepth = height(node.right);

            // Use the larger one

            if (lDepth > rDepth)

                return (lDepth + 1);

            else

                return (rDepth + 1);

        }

    }

    // Driver Code

    public static void main(String[] args)

    {

        /* Let us create following BST

                  50

               /     \

              30      70

             /  \    /  \

            20   40  60   80

     */

        node root = null;

        // inserting value 50

        root = insert(root, 50);

        // inserting value 30
```

```
        insert(root, 30);

         // inserting value 20

        insert(root, 20);

         // inserting value 40

        insert(root, 40);

         // inserting value 70

        insert(root, 70);

         // inserting value 60

        insert(root, 60);

         // inserting value 80

        insert(root, 80);

         // Function Call

        System.out.println(" " + height(root));

    }

}
```

Output

```
3
```

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Delete a Node of BST: It is used to delete a node with specific key from the BST and return the new BST.

Different scenarios for deleting the node:

1. Node to be deleted is the leaf node : Its simple you can just null it out.
2. Node to be deleted has one child : You can just replace the node with the child node.
3. Node to be deleted has two child :

- Need to figure out what will be the replacement of the node to be deleted.
- Want minimal disruption to the existing tree structure
- Can table the replacement node from the deleted nodes left or right subtree.
- If taking if from the left subtree, we have to take the largest value in the left subtree.
- If taking if from the right subtree, we have to take the smallest value in the right subtree.
- Choose one approach and stick to it.

```
import java.io.*;

 // Java program for Delete a Node of BST

class GFG {

     // Given Node node

    static class node {

        int key;

        node left, right;

    };

     // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }
```

```
 // Function to insert a new node with

// given key in BST

static node insert(node node, int key)

{

    // If the tree is empty, return a new node

    if (node == null)

        return newNode(key);

     // Otherwise, recur down the tree

    if (key < node.key) {

        node.left = insert(node.left, key);

    }

    else if (key > node.key) {

        node.right = insert(node.right, key);

    }

     // Return the node

    return node;

}

 // Function to do inorder traversal of BST

static void inorder(node root)

{

    if (root != null) {

        inorder(root.left);
```

```
            System.out.print(" " + root.key);
            inorder(root.right);
        }
    }
    // Function that returns the node with minimum
    // key value found in that tree
    static node minValueNode(node node)
    {
        node current = node;
         // Loop down to find the leftmost leaf
        while (current != null && current.left != null)
            current = current.left;
         return current;
    }
     // Function that deletes the key and
    // returns the new root
    static node deleteNode(node root, int key)
    {
        // base Case
        if (root == null)
            return root;
```

```
// If the key to be deleted is

// smaller than the root's key,

// then it lies in left subtree

if (key < root.key) {

    root.left = deleteNode(root.left, key);

}

 // If the key to be deleted is

// greater than the root's key,

// then it lies in right subtree

else if (key > root.key) {

     root.right = deleteNode(root.right, key);

}

 // If key is same as root's key,

// then this is the node

// to be deleted

else {

     // Node with only one child

    // or no child

    if (root.left == null) {

        node temp = root.right;

        return temp;

    }
```

```
            else if (root.right == null) {
                node temp = root.left;
                return temp;
            }
             // Node with two children:
            // Get the inorder successor(smallest
            // in the right subtree)
            node temp = minValueNode(root.right);
             // Copy the inorder successor's
            // content to this node
            root.key = temp.key;
             // Delete the inorder successor
            root.right = deleteNode(root.right, temp.key);
        }
        return root;
    }


// Driver Code
    public static void main(String[] args)
   {
         /* Let us create following BST
```

```
            50
          /     \
         30      70
        /  \    /  \
      20   40  60   80
*/
   node root = null;
    // inserting value 50
   root = insert(root, 50);
    // inserting value 30
   insert(root, 30);
    // inserting value 20
   insert(root, 20);
    // inserting value 40
   insert(root, 40);
    // inserting value 70
   insert(root, 70);
    // inserting value 60
   insert(root, 60);
    // inserting value 80
   insert(root, 80);
```

```
        // Function Call

        root = deleteNode(root, 60);

        inorder(root);

    }

}
```

Output

```
20 30 40 50 70 80
```

Time Complexity: O(log N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Smallest Node of the BST: It is used to return the node with the smallest value in the BST.

```
import java.io.*;

// Java program for Smallest Node in the BST

class GFG {

    // Given Node node

    static class node {

        int key;

        node left, right;

    };

    // Function to create a new BST node

    static node newNode(int item)

    {
```

```
    node temp = new node();

    temp.key = item;

    temp.left = temp.right = null;

    return temp;

}

 // Function to insert a new node with

// given key in BST

static node insert(node node, int key)

{

    // If the tree is empty, return a new node

    if (node == null)

       return newNode(key);

     // Otherwise, recur down the tree

    if (key < node.key) {

        node.left = insert(node.left, key);

    }

    else if (key > node.key) {

        node.right = insert(node.right, key);

    }

     // Return the node

    return node;

}
```

```
 // Function that returns the node with minimum
// key value found in that tree
static node minValueNode(node node)
{
    node current = node;
     // Loop down to find the leftmost leaf
    while (current != null && current.left != null)
        current = current.left;
     return current;
}
 // Driver Code
public static void main(String[] args)
{
     /* Let us create following BST
              50
           /     \
         30       70
        /  \     /  \
      20   40   60   80
 */
    node root = null;
```

```
        // inserting value 50
        root = insert(root, 50);
         // inserting value 30
        insert(root, 30);
         // inserting value 20
        insert(root, 20);
         // inserting value 40
        insert(root, 40);
         // inserting value 70
        insert(root, 70);
         // inserting value 60
        insert(root, 60);
         // inserting value 80
        insert(root, 80);
        // Function Call
        System.out.println(" " + minValueNode(root).key);
    }
}
```

Output

```
20
```

Time Complexity: O(log N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

- Total number of nodes in a BST: The function returns the total count of nodes in the BST.

```
import java.io.*;

 // Java program for Total number of nodes in BST

class GFG {

     // Given Node node

    static class node {

        int key;

        node left, right;

    };

     // Function to create a new BST node

    static node newNode(int item)

    {

        node temp = new node();

        temp.key = item;

        temp.left = temp.right = null;

        return temp;

    }

     // Function to insert a new node with

    // given key in BST

    static node insert(node node, int key)

    {
```

```
    // If the tree is empty, return a new node
    if (node == null)
        return newNode(key);
     // Otherwise, recur down the tree
    if (key < node.key) {
        node.left = insert(node.left, key);
    }
    else if (key > node.key) {
        node.right = insert(node.right, key);
    }
     // Return the node
    return node;
}
 // Function to get the total count of
// nodes in a binary tree
static int nodeCount(node node)
{
    if (node == null)
        return 0;
     else
        return nodeCount(node.left)
            + nodeCount(node.right) + 1;
```

```
}

 // Driver Code

public static void main(String[] args)

{

     /* Let us create following BST

              50

           /     \

         30       70

        /  \     /  \

      20   40   60   80

 */

    node root = null;

     // inserting value 50

    root = insert(root, 50);

     // inserting value 30

    insert(root, 30);

     // inserting value 20

    insert(root, 20);

     // inserting value 40

    insert(root, 40);

     // inserting value 70
```

```
        insert(root, 70);

         // inserting value 60

        insert(root, 60);

         // inserting value 80

        insert(root, 80);

         // print the BST

        System.out.print(" " + nodeCount(root));

    }

}
```

Output
7

Time Complexity: O(N), where N is the number of nodes of the BST
Auxiliary Space: O(1)

Applications of BST:

- **Graph algorithms:** BSTs can be used to implement graph algorithms, such as in minimum spanning tree algorithms.
- **Priority Queues:** BSTs can be used to implement priority queues, where the element with the highest priority is at the root of the tree, and elements with lower priority are stored in the subtrees.
- **Self-balancing binary search tree:** BSTs can be used as a self-balancing data structures such as AVL tree and Red-black tree.
- **Data storage and retrieval:** BSTs can be used to store and retrieve data quickly, such as in databases, where searching for a specific record can be done in logarithmic time.

Advantages:

- **Fast search:** Searching for a specific value in a BST has an average time complexity of O(log n), where n is the number of nodes in the tree. This is much faster than searching for an element in an array or linked list, which have a time complexity of O(n) in the worst case.
- **In-order traversal:** BSTs can be traversed in-order, which visits the left subtree, the root, and the right subtree. This can be used to sort a dataset.
- **Space efficient:** BSTs are space efficient as they do not store any redundant information, unlike arrays and linked lists.

Disadvantages:

- **Skewed trees:** If a tree becomes skewed, the time complexity of search, insertion, and deletion operations will be O(n) instead of O(log n), which can make the tree inefficient.
- **Additional time required:** Self-balancing trees require additional time to maintain balance during insertion and deletion operations.
- **Efficiency:** BSTs are not efficient for datasets with many duplicates as they will waste space.