Inorder predecessor and successor for a given key in BST

Difficulty Level : Medium • Last Updated : 04 Jul, 2022

There is BST given with root node with key part as integer only. The structure of each node is as follows:

C++

```
struct Node
{
    int key;
    struct Node *left, *right ;
};
```

Java

```
static class Node
{
    int key;
```

```
// This code is contributed by gauravrajput1
```

Python3

```
class Node:

    def __init__(self, key):

        self.key = key
        self.left = None
        self.right = None

# This code is contributed by harshitkap00r
```

C#

```
public class Node
{
    public int key;
    public Node left, right ;
};

// This code is contributed by gauravrajput1
```

Javascript

```
<script>
```

```
        this.key = 0;
        this.left = null;
        this.right = null;
      }
    }

</script>
```

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.

Recommended PracticePredecessor and SuccessorTry It!

Following is the algorithm to reach the desired result. It is a recursive method:

```
Input: root node, key
output: predecessor node, successor node

1. If root is NULL
      then return
2. if key is found then
    a. If its left subtree is not null
        Then predecessor will be the right most
        child of left subtree or left child itself.
```

```
        of right subtree or right child itself.
    return
3. If key is smaller then root node
        set the successor as root
        search recursively into left subtree
    else
        set the predecessor as root
        search recursively into right subtree
```

Following is the implementation of the above algorithm:

C++

```cpp
// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

// BST Node
struct Node
{
    int key;
    struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
```

```
// If key is present at root
if (root->key == key)
{
    // the maximum value in left subtree is predecessor
    if (root->left != NULL)
    {
        Node* tmp = root->left;
        while (tmp->right)
            tmp = tmp->right;
        pre = tmp ;
    }

    // the minimum value in right subtree is successor
    if (root->right != NULL)
    {
        Node* tmp = root->right ;
        while (tmp->left)
            tmp = tmp->left ;
        suc = tmp ;
    }
    return ;
}

// If key is smaller than root's key, go to left subtree
if (root->key > key)
{
    suc = root ;
    findPreSuc(root->left, pre, suc, key) ;
}
else // go to right subtree
{
    pre = root ;
    findPreSuc(root->right, pre, suc, key) ;
```

```
// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp =  new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65;    //Key to be searched in BST

   /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
    Node *root = NULL;
    root = insert(root, 50);
```

```
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);


    Node* pre = NULL, *suc = NULL;

    findPreSuc(root, pre, suc, key);
    if (pre != NULL)
      cout << "Predecessor is " << pre->key << endl;
    else
      cout << "No Predecessor";

    if (suc != NULL)
      cout << "Successor is " << suc->key;
    else
      cout << "No Successor";
    return 0;
}
```

Java

```
// Java program to find predecessor
// and successor in a BST
class GFG{

// BST Node
static class Node
{
    int key;
    Node left, right;
```

```
    public Node(int key)
    {
        this.key = key;
        this.left = this.right = null;
    }
};

static Node pre = new Node(), suc = new Node();

// This function finds predecessor and
// successor of key in BST. It sets pre
// and suc as predecessor and successor
// respectively
static void findPreSuc(Node root, int key)
{

    // Base case
    if (root == null)
        return;

    // If key is present at root
    if (root.key == key)
    {

        // The maximum value in left
        // subtree is predecessor
        if (root.left != null)
        {
            Node tmp = root.left;
            while (tmp.right != null)
                tmp = tmp.right;
```

```
            // The minimum value in
            // right subtree is successor
            if (root.right != null)
            {
                Node tmp = root.right;

                while (tmp.left != null)
                    tmp = tmp.left;

                suc = tmp;
            }
            return;
        }

        // If key is smaller than
        // root's key, go to left subtree
        if (root.key > key)
        {
            suc = root;
            findPreSuc(root.left, key);
        }

        // Go to right subtree
        else
        {
            pre = root;
            findPreSuc(root.right, key);
        }
    }

    // A utility function to insert a
    // new node with given key in BST
    static Node insert(Node node, int key)
```

```
        if (key < node.key)
            node.left = insert(node.left, key);
        else
            node.right = insert(node.right, key);

        return node;
    }

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

        // Key to be searched in BST
        int key = 65;

        /*
         * Let us create following BST
         *           50
         *         /  \
         *        30   70
         *       /  \ /  \
         *      20 40 60  80
         */

        Node root = new Node();
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);
```

```
        else
            System.out.println("No Predecessor");

        if (suc != null)
            System.out.println("Successor is " + suc.key);
        else
            System.out.println("No Successor");
    }
    }

    // This code is contributed by sanjeev2552
```

Python

```
# Python program to find predecessor and successor in a BST

# A BST node
class Node:

    # Constructor to create a new node
    def __init__(self, key):
        self.key  = key
        self.left = None
        self.right = None

# This function finds predecessor and successor of key in BST
# It sets pre and suc as predecessor and successor respectively
def findPreSuc(root, key):

    # Base Case
    if root is None:
```

```
        if root.key == key:

            # the maximum value in left subtree is predecessor
            if root.left is not None:
                tmp = root.left
                while(tmp.right):
                    tmp = tmp.right
                findPreSuc.pre = tmp


            # the minimum value in right subtree is successor
            if root.right is not None:
                tmp = root.right
                while(temp.left):
                    tmp = tmp.left
                findPreSuc.suc = tmp

            return

        # If key is smaller than root's key, go to left subtree
        if root.key > key :
            findPreSuc.suc = root
            findPreSuc(root.left, key)

        else: # go to right subtree
            findPreSuc.pre = root
            findPreSuc(root.right, key)

    # A utility function to insert a new node in with given key in BST
    def insert(node , key):
        if node is None:
            return Node(key)
```

```
        else:
            node.right = insert(node.right, key)

        return node


# Driver program to test above function
key = 65 #Key to be searched in BST

""" Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80
"""
root = None
root = insert(root, 50)
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

# Static variables of the function findPreSuc
findPreSuc.pre = None
findPreSuc.suc = None

findPreSuc(root, key)

if findPreSuc.pre is not None:
    print "Predecessor is", findPreSuc.pre.key
```

```
if findPreSuc.suc is not None:
    print "Successor is", findPreSuc.suc.key
else:
    print "No Successor"

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

C#

```
// C# program to find predecessor
// and successor in a BST
using System;
public class GFG
{

  // BST Node
  public

    class Node
    {
      public
        int key;
      public
        Node left, right;
      public Node()
      {}

      public Node(int key)
      {
        this.key = key;
```

```
static Node pre = new Node(), suc = new Node();

// This function finds predecessor and
// successor of key in BST. It sets pre
// and suc as predecessor and successor
// respectively
static void findPreSuc(Node root, int key)
{

  // Base case
  if (root == null)
    return;

  // If key is present at root
  if (root.key == key)
  {

    // The maximum value in left
    // subtree is predecessor
    if (root.left != null)
    {
      Node tmp = root.left;
      while (tmp.right != null)
        tmp = tmp.right;

      pre = tmp;
    }

    // The minimum value in
    // right subtree is successor
    if (root.right != null)
    {
```

```
            tmp = tmp.left;

          suc = tmp;
        }
        return;
      }

      // If key is smaller than
      // root's key, go to left subtree
      if (root.key > key)
      {
        suc = root;
        findPreSuc(root.left, key);
      }

      // Go to right subtree
      else
      {
        pre = root;
        findPreSuc(root.right, key);
      }
    }

    // A utility function to insert a
    // new node with given key in BST
    static Node insert(Node node, int key)
    {
      if (node == null)
        return new Node(key);
      if (key < node.key)
        node.left = insert(node.left, key);
      else
        node.right = insert(node.right, key);
```

```
// Driver code
public static void Main(String[] args)
{

  // Key to be searched in BST
  int key = 65;

  /*
   * Let us create following BST
   *           50
   *         /  \
   *        30   70
   *       /  \ /  \
   *      20 40 60  80
   */

  Node root = new Node();
  root = insert(root, 50);
  insert(root, 30);
  insert(root, 20);
  insert(root, 40);
  insert(root, 70);
  insert(root, 60);
  insert(root, 80);

  findPreSuc(root, key);
  if (pre != null)
    Console.WriteLine("Predecessor is " + pre.key);
  else
    Console.WriteLine("No Predecessor");

  if (suc != null)
```

```
  }
}

// This code is contributed by aashish1995
```

Javascript

```
<script>

// JavaScript program to find predecessor
// and successor in a BST// BST Node
 class Node
{
     constructor(key)
    {
        this.key = key;
        this.left = this.right = null;
    }
}

var pre = new Node(), suc = new Node();

// This function finds predecessor and
// successor of key in BST. It sets pre
// and suc as predecessor and successor
// respectively
function findPreSuc(root , key)
{

    // Base case
    if (root == null)
```

```
if (root.key == key)
{

    // The maximum value in left
    // subtree is predecessor
    if (root.left != null)
    {
        var tmp = root.left;
        while (tmp.right != null)
            tmp = tmp.right;

        pre = tmp;
    }

    // The minimum value in
    // right subtree is successor
    if (root.right != null)
    {
        var tmp = root.right;

        while (tmp.left != null)
            tmp = tmp.left;

        suc = tmp;
    }
    return;
}

// If key is smaller than
// root's key, go to left subtree
if (root.key > key)
{
    suc = root;
```

```
    // Go to right subtree
    else
    {
        pre = root;
        findPreSuc(root.right, key);
    }
}

// A utility function to insert a
// new node with given key in BST
function insert(node , key)
{
    if (node == null)
        return new Node(key);
    if (key < node.key)
        node.left = insert(node.left, key);
    else
        node.right = insert(node.right, key);

    return node;
}

// Driver code


    // Key to be searched in BST
    var key = 65;

    /*
     * Let us create following BST
     *          50
     *         /  \
     *        30   70
```

```
        var root = new Node();
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);

        findPreSuc(root, key);
        if (pre != null)
            document.write("Predecessor is " + pre.key);
        else
            document.write("No Predecessor");

        if (suc != null)
            document.write("<br/>Successor is " + suc.key);
        else
            document.write("<br/>No Successor");

    // This code contributed by gauravrajput1

    </script>
```

Output:

```
Predecessor is 60
Successor is 70
```

Complexity Analysis:

Time Complexity: *O(h), where h is the height of the tree. In the worst case as explained above we travel the whole height of the tree.*
Auxiliary Space: *O(1)*

Another Approach:

We can also find the inorder successor and inorder predecessor using inorder traversal. Check if the current node is smaller than the given key for the predecessor and for a successor, check if it is greater than the given key. If it is

C++

```cpp
// CPP code for inorder successor
// and predecessor of tree
#include<iostream>
#include<stdlib.h>

using namespace std;

struct Node
{
    int data;
    Node* left,*right;
};

// Function to return data
Node* getnode(int info)
{
    Node* p = (Node*)malloc(sizeof(Node));
    p->data = info;
    p->right = NULL;
    p->left = NULL;
    return p;
}

/*
since inorder traversal results in
ascending order visit to node , we
can store the values of the largest
no which is smaller than a (predecessor)
and smallest no which is large than
```

```
                Node** p, Node** q)
{
    // If root is null return
    if(!root)
        return ;

    // traverse the left subtree
    find_p_s(root->left, a, p, q);

    // root data is greater than a
    if(root&&root->data > a)
    {

        // q stores the node whose data is greater
        // than a and is smaller than the previously
        // stored data in *q which is successor
        if((!*q) || (*q) && (*q)->data > root->data)
                *q = root;
    }

    // if the root data is smaller than
    // store it in p which is predecessor
    else if(root && root->data < a)
    {
        *p = root;
    }

    // traverse the right subtree
    find_p_s(root->right, a, p, q);
}

// Driver code
int main()
```

```
    root1->right = getnode(60);
    root1->left->left = getnode(10);
    root1->left->right = getnode(30);
    root1->right->left = getnode(55);
    root1->right->right = getnode(70);
    Node* p = NULL, *q = NULL;

    find_p_s(root1, 55, &p, &q);

    if(p)
        cout << p->data;
    if(q)
        cout << " " << q->data;
    return 0;
}
```

Java

```
// JAVA code for inorder successor
// and predecessor of tree

import java.util.*;

class GFG{

static class Node
{
    int data;
    Node left,right;
};
```

```
        Node p = new Node();
        p.data = info;
        p.right = null;
        p.left = null;
        return p;
    }

    /*
    since inorder traversal results in
    ascending order visit to node , we
    can store the values of the largest
    no which is smaller than a (predecessor)
    and smallest no which is large than
    a (successor) using inorder traversal
    */
    static Node p,q;
    static void find_p_s(Node root,int a)
    {
        // If root is null return
        if(root == null)
            return ;

        // traverse the left subtree
        find_p_s(root.left, a);

        // root data is greater than a
        if(root != null && root.data > a)
        {

            // q stores the node whose data is greater
            // than a and is smaller than the previously
            // stored data in *q which is successor
            if((q == null) || (q != null) && q.data > root.data)
```

```
        // if the root data is smaller than
        // store it in p which is predecessor
        else if(root != null && root.data < a)
        {
            p = root;
        }

        // traverse the right subtree
        find_p_s(root.right, a);
    }

    // Driver code
    public static void main(String[] args)
    {
        Node root1 = getnode(50);
        root1.left = getnode(20);
        root1.right = getnode(60);
        root1.left.left = getnode(10);
        root1.left.right = getnode(30);
        root1.right.left = getnode(55);
        root1.right.right = getnode(70);
         p = null;
         q = null;

        find_p_s(root1, 55);

        if(p != null)
            System.out.print(p.data);
        if(q != null)
            System.out.print(" " +  q.data);
    }
    }
```

Python3

```
""" Python3 code for inorder successor
and predecessor of tree """

# A Binary Tree Node
# Utility function to create a new tree node
class getnode:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

"""
since inorder traversal results in
ascendingorder visit to node , we
can store the values of the largest
o which is smaller than a (predecessor)
and smallest no which is large than
a (successor) using inorder traversal
"""
def find_p_s(root, a, p, q):

    # If root is None return
    if(not root):
        return

    # traverse the left subtree
    find_p_s(root.left, a, p, q)
```

```
            # q stores the node whose data is greater
            # than a and is smaller than the previously
            # stored data in *q which is successor
            if((not q[0]) or q[0] and
                    q[0].data > root.data):
                q[0] = root

        # if the root data is smaller than
        # store it in p which is predecessor
        elif(root and root.data < a):
            p[0]= root

        # traverse the right subtree
        find_p_s(root.right, a, p, q)

# Driver Code
if __name__ == '__main__':

    root1 = getnode(50)
    root1.left = getnode(20)
    root1.right = getnode(60)
    root1.left.left = getnode(10)
    root1.left.right = getnode(30)
    root1.right.left = getnode(55)
    root1.right.right = getnode(70)
    p = [None]
    q = [None]

    find_p_s(root1, 55, p, q)

    if(p[0]) :
        print(p[0].data, end = "")
    if(q[0]) :
```

```
# SHUBHAMSINGH10
```

C#

```
// C# code for inorder successor
// and predecessor of tree
using System;

public class GFG {

  public class Node {
    public int data;
    public Node left, right;
  };

  // Function to return data
  static Node getnode(int info) {
    Node p = new Node();
    p.data = info;
    p.right = null;
    p.left = null;
    return p;
  }

  /*
     * since inorder traversal results in ascending order visit to node , we can
     * store the values of the largest no which is smaller than a (predecessor) and
     * smallest no which is large than a (successor) using inorder traversal
     */
  static Node p, q;
```

```
  // If root is null return
  if (root == null)
    return;

  // traverse the left subtree
  find_p_s(root.left, a);

  // root data is greater than a
  if (root != null && root.data > a) {

    // q stores the node whose data is greater
    // than a and is smaller than the previously
    // stored data in *q which is successor
    if ((q == null) || (q != null) && q.data > root.data)
      q = root;
  }

  // if the root data is smaller than
  // store it in p which is predecessor
  else if (root != null && root.data < a) {
    p = root;
  }

  // traverse the right subtree
  find_p_s(root.right, a);
}

// Driver code
public static void Main(String[] args) {
  Node root1 = getnode(50);
  root1.left = getnode(20);
  root1.right = getnode(60);
  root1.left.left = getnode(10);
```

```
        p = null;
        q = null;

        find_p_s(root1, 55);

        if (p != null)
          Console.Write(p.data);
        if (q != null)
          Console.Write(" " + q.data);
      }
    }

    // This code is contributed by Rajput-Ji
```

Javascript

```
<script>

class Node
{
    constructor(data)
    {
        this.data = data;
        this.left = this.right = null;
    }
}

function find_p_s(root, a, p, q)
{
    // If root is None return
    if(root == null)
```

```
    find_p_s(root.left, a, p, q)

    // root data is greater than a
    if(root && root.data > a)
    {
        // q stores the node whose data is greater
        // than a and is smaller than the previously
        // stored data in *q which is successor
        if((q[0] == null) || q[0] != null && q[0].data > root.data)

            q[0] = root

    }

    // if the root data is smaller than
    // store it in p which is predecessor
    else if(root && root.data < a)
    {    p[0] = root

     }

    // traverse the right subtree
    find_p_s(root.right, a, p, q)
}

// Driver Code
let root1 = new Node(50)
root1.left = new Node(20)
root1.right = new Node(60)
root1.left.left = new Node(10)
root1.left.right = new Node(30)
root1.right.left = new Node(55)
root1.right.right = new Node(70)
```

```
find_p_s(root1, 55, p, q)

if(p[0] != null)
    document.write(p[0].data, end = " ")
if(q[0] != null)
    document.write(" ", q[0].data)

// This code is contributed by patel2127
</script>
```

Output :

```
50 60
```

Complexity Analysis:

Time Complexity: *O(n), where n is the total number of nodes in the tree. In the worst case as explained above we travel the whole tree.*

Auxiliary Space: *O(n).*

?list=PLqM7alHXFySHCXD7r1J0ky9Zg_GBB1dbk

This article is contributed by **algoLover**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Previous

Threaded Binary Search Tree | Deletion

Next

How to handle duplicates in Binary Search Tree?

RECOMMENDED ARTICLES

Page : 1 2 3

01 Inorder predecessor and successor for a given key in BST | Iterative Approach
25, Jun 18

02 Replace each node in binary tree with the sum of its inorder predecessor and successor
07, Jul 17

03 Modify Binary Tree by replacing each node with the sum of its Preorder Predecessor and Successor

05 Inorder Successor in Binary Search Tree
11, Feb 11

06 Inorder Successor of a node in Binary Tree
04, Oct 17

07 Postorder predecessor of a Node in Binary Search Tree
03, Jun 18

26, Jan 12 03, Jun 18

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Improved By : SHUBHAMSINGH10, sanjeev2552, aashish1995, GauravRajput1, surinderdawra388, itwasme, patel2127, harshitkap00r, Rajput-Ji, kapilag, polymatir3j, adityap3055

Article Tags : Ola Cabs, Binary Search Tree, Tree

Practice Tags : Ola Cabs, Binary Search Tree, Tree

Improve Article Report Issue

GeeksforGeeks
A-143, 9th Floor, Sovereign Corporate Tower,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org