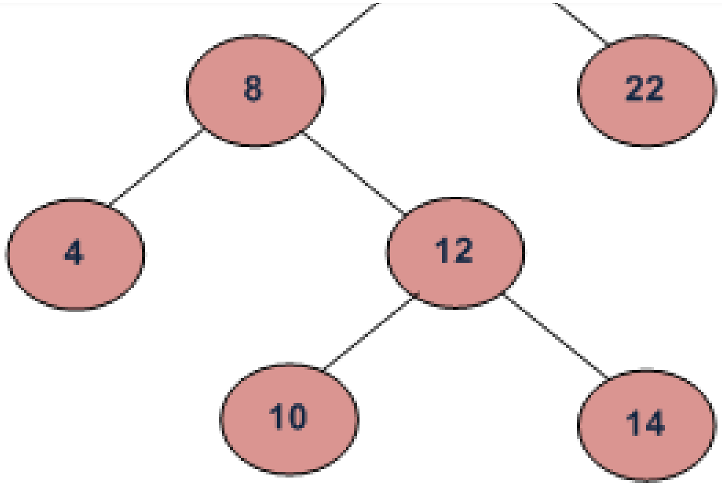
Inorder Successor in Binary Search Tree

Difficulty Level : Medium • Last Updated : 17 Jun, 2022

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inorder traversal.

In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of the input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

Recommended PracticeInorder Successor in BSTTry It!

Method 1 (Uses Parent Pointer)

In this method, we assume that every node has a parent pointer.
The Algorithm is divided into two cases on the basis of the right subtree of the input node being empty or not.

Input: *node, root // node* is the node whose Inorder successor is needed.

Output: *succ // succ* is Inorder successor of *node*.

1. If right subtree of *node* is not *NULL*, then succ lies in right subtree. Do the following.
   Go to right subtree and return the node with minimum key value in the right subtree.
2. If right subtree of *node* is NULL, then *succ* is one of the ancestors. Do the following.
   Travel up using the parent pointer until you see a node which is left child of its parent. The parent of such a node is the *succ*.

Implementation:

Note that the function to find InOrder Successor is highlighted (with gray background) in below code.

C++

```
#include <iostream>
using namespace std;

/* A binary tree node has data,
```

```
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
};

struct node* minValue(struct node* node);

struct node* inOrderSuccessor(
    struct node* root,
    struct node* n)
{
    // step 1 of the above algorithm
    if (n->right != NULL)
        return minValue(n->right);

    // step 2 of the above algorithm
    struct node* p = n->parent;
    while (p != NULL && n == p->right) {
        n = p;
        p = p->parent;
    }
    return p;
}

/* Given a non-empty binary search tree,
    return the minimum data
    value found in that tree. Note that
    the entire tree does not need
    to be searched. */
struct node* minValue(struct node* node)
{
    struct node* current = node;
```

```
        current = current->left;
    }
    return current;
}

/* Helper function that allocates a new
    node with the given data and
    NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(
            struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;

    return (node);
}

/* Give a binary search tree and
   a number, inserts a new node with
    the given number in the correct
    place in the tree. Returns the new
    root pointer which the caller should
    then use (the standard trick to
    avoid using reference parameters). */
struct node* insert(struct node* node,
                    int data)
{
    /* 1. If the tree is empty, return a new,
      single node */
```

```
        struct node* temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data) {
            temp = insert(node->left, data);
            node->left = temp;
            temp->parent = node;
        }
        else {
            temp = insert(node->right, data);
            node->right = temp;
            temp->parent = node;
        }

        /* return the (unchanged) node pointer */
        return node;
    }
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = NULL, *temp, *succ, *min;

    // creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;
```

```
        cout << "\n Inorder Successor of " << temp->data<< " is "<< succ->data;
    else
        cout <<"\n Inorder Successor doesn't exit";

    getchar();
    return 0;
}

// this code is contributed by shivanisinghss2110
```

C

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data,
   the pointer to left child
   and a pointer to right child */
struct node {
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
};

struct node* minValue(struct node* node);

struct node* inOrderSuccessor(
    struct node* root,
    struct node* n)
{
```

```
    // step 2 of the above algorithm
    struct node* p = n->parent;
    while (p != NULL && n == p->right) {
        n = p;
        p = p->parent;
    }
    return p;
}

/* Given a non-empty binary search tree,
    return the minimum data
    value found in that tree. Note that
    the entire tree does not need
    to be searched. */
struct node* minValue(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) {
        current = current->left;
    }
    return current;
}

/* Helper function that allocates a new
    node with the given data and
    NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(
```

```
    node->right = NULL;
    node->parent = NULL;

    return (node);
}

/* Give a binary search tree and
   a number, inserts a new node with
    the given number in the correct
    place in the tree. Returns the new
    root pointer which the caller should
    then use (the standard trick to
    avoid using reference parameters). */
struct node* insert(struct node* node,
                    int data)
{
    /* 1. If the tree is empty, return a new,
      single node */
    if (node == NULL)
        return (newNode(data));
    else {
        struct node* temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data) {
            temp = insert(node->left, data);
            node->left = temp;
            temp->parent = node;
        }
        else {
            temp = insert(node->right, data);
            node->right = temp;
            temp->parent = node;
```

```
        return node;
    }
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = NULL, *temp, *succ, *min;

    // creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    succ = inOrderSuccessor(root, temp);
    if (succ != NULL)
        printf(
            "\n Inorder Successor of %d is %d ",
            temp->data, succ->data);
    else
        printf("\n Inorder Successor doesn't exit");

    getchar();
    return 0;
}
```

```
// Java program to find minimum
// value node in Binary Search Tree

// A binary tree node
class Node {

    int data;
    Node left, right, parent;

    Node(int d)
    {
        data = d;
        left = right = parent = null;
    }
}

class BinaryTree {

    static Node head;

    /* Given a binary search tree and a number,
     inserts a new node with the given number in
     the correct place in the tree. Returns the new
     root pointer which the caller should then use
     (the standard trick to avoid using reference
     parameters). */
    Node insert(Node node, int data)
    {

        /* 1. If the tree is empty, return a new,
         single node */
        if (node == null) {
            return (new Node(data));
```

```
            Node temp = null;

            /* 2. Otherwise, recur down the tree */
            if (data <= node.data) {
                temp = insert(node.left, data);
                node.left = temp;
                temp.parent = node;
            }
            else {
                temp = insert(node.right, data);
                node.right = temp;
                temp.parent = node;
            }

            /* return the (unchanged) node pointer */
            return node;
        }
    }

    Node inOrderSuccessor(Node root, Node n)
    {

        // step 1 of the above algorithm
        if (n.right != null) {
            return minValue(n.right);
        }

        // step 2 of the above algorithm
        Node p = n.parent;
        while (p != null && n == p.right) {
            n = p;
            p = p.parent;
        }
```

```
/* Given a non-empty binary search
   tree, return the minimum data
   value found in that tree. Note that
   the entire tree does not need
   to be searched. */
Node minValue(Node node)
{
    Node current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null) {
        current = current.left;
    }
    return current;
}

// Driver program to test above functions
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    Node root = null, temp = null, suc = null, min = null;
    root = tree.insert(root, 20);
    root = tree.insert(root, 8);
    root = tree.insert(root, 22);
    root = tree.insert(root, 4);
    root = tree.insert(root, 12);
    root = tree.insert(root, 10);
    root = tree.insert(root, 14);
    temp = root.left.right.right;
    suc = tree.inOrderSuccessor(root, temp);
    if (suc != null) {
        System.out.println(
            "Inorder successor of "
```

```
            System.out.println(
                "Inorder successor does not exist");
        }
    }
}

// This code has been contributed by Mayank Jaiswal
```

Python3

```
# Python program to find the inorder successor in a BST

# A binary tree node
class Node:

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

def inOrderSuccessor(n):

    # Step 1 of the above algorithm
    if n.right is not None:
        return minValue(n.right)

    # Step 2 of the above algorithm
    p = n.parent
    while( p is not None):
        if n != p.right :
```

```
    return p

# Given a non-empty binary search tree, return the
# minimum data value found in that tree. Note that the
# entire tree doesn't need to be searched
def minValue(node):
    current = node

    # loop down to find the leftmost leaf
    while(current is not None):
        if current.left is None:
            break
        current = current.left

    return current


# Given a binary search tree and a number, inserts a
# new node with the given number in the correct place
# in the tree. Returns the new root pointer which the
# caller should then use( the standard trick to avoid
# using reference parameters)
def insert( node, data):

    # 1) If tree is empty then return a new singly node
    if node is None:
        return Node(data)
    else:

        # 2) Otherwise, recur down the tree
        if data <= node.data:
            temp = insert(node.left, data)
            node.left = temp
```

```
            node.right = temp
            temp.parent = node

        # return  the unchanged node pointer
        return node


# Driver program to test above function

root = None

# Creating the tree given in the above diagram
root = insert(root, 20)
root = insert(root, 8);
root = insert(root, 22);
root = insert(root, 4);
root = insert(root, 12);
root = insert(root, 10);
root = insert(root, 14);
temp = root.left.right.right

succ = inOrderSuccessor(temp)
if succ is not None:
    print ("\nInorder Successor of % d is % d "%(temp.data, succ.data))
else:
    print ("\nInorder Successor doesn't exist")

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

C#

```
// A binary tree node
public
  class Node
  {
    public
      int data;
    public
      Node left, right, parent;
    public
      Node(int d)
    {
      data = d;
      left = right = parent = null;
    }
  }

public class BinaryTree
{
  static Node head;

  /* Given a binary search tree and a number,
     inserts a new node with the given number in
     the correct place in the tree. Returns the new
     root pointer which the caller should then use
     (the standard trick to avoid using reference
     parameters). */
  Node insert(Node node, int data)
  {

    /* 1. If the tree is empty, return a new,
          single node */
    if (node == null)
```

```
        else
        {

          Node temp = null;

          /* 2. Otherwise, recur down the tree */
          if (data <= node.data)
          {
            temp = insert(node.left, data);
            node.left = temp;
            temp.parent = node;
          }
          else
          {
            temp = insert(node.right, data);
            node.right = temp;
            temp.parent = node;
          }

          /* return the (unchanged) node pointer */
          return node;
        }
      }

      Node inOrderSuccessor(Node root, Node n)
      {

        // step 1 of the above algorithm
        if (n.right != null)
        {
          return minValue(n.right);
        }
```

```
  {
    n = p;
    p = p.parent;
  }
  return p;
}

/* Given a non-empty binary search
     tree, return the minimum data
     value found in that tree. Note that
     the entire tree does not need
     to be searched. */
Node minValue(Node node)
{
  Node current = node;

  /* loop down to find the leftmost leaf */
  while (current.left != null)
  {
    current = current.left;
  }
  return current;
}

// Driver program to test above functions
public static void Main(String[] args)
{
  BinaryTree tree = new BinaryTree();
  Node root = null, temp = null, suc = null, min = null;
  root = tree.insert(root, 20);
  root = tree.insert(root, 8);
  root = tree.insert(root, 22);
  root = tree.insert(root, 4);
```

```
        temp = root.left.right.right;
        suc = tree.inOrderSuccessor(root, temp);
        if (suc != null) {
          Console.WriteLine(
            "Inorder successor of "
            + temp.data + " is " + suc.data);
        }
        else {
          Console.WriteLine(
            "Inorder successor does not exist");
        }
      }
    }

    // This code is contributed by aashish1995
```

Javascript

```
<script>

// JavaScript program to find minimum
// value node in Binary Search Tree

// A binary tree node
class Node {
    constructor(val) {
        this.data = val;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
```

```
var head;

/*
 * Given a binary search tree and a number,
 inserts a new node with the given
 * number in the correct place in the tree.
 Returns the new root pointer which
 * the caller should then use
 (the standard trick to afunction using reference
 * parameters).
 */
function insert(node , data) {

    /*
     * 1. If the tree is empty,
     return a new, single node
     */
    if (node == null) {
        return (new Node(data));
    } else {

        var temp = null;

        /* 2. Otherwise, recur down the tree */
        if (data <= node.data) {
            temp = insert(node.left, data);
            node.left = temp;
            temp.parent = node;
        } else {
            temp = insert(node.right, data);
            node.right = temp;
            temp.parent = node;
        }
```

```
    }
}

function inOrderSuccessor(root,  n) {

    // step 1 of the above algorithm
    if (n.right != null) {
        return minValue(n.right);
    }

    // step 2 of the above algorithm
    var p = n.parent;
    while (p != null && n == p.right) {
        n = p;
        p = p.parent;
    }
    return p;
}

/*
 * Given a non-empty binary search tree,
 return the minimum data value found in
 * that tree. Note that the entire tree
 does not need to be searched.
 */
function minValue(node) {
    var current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null) {
        current = current.left;
    }
    return current;
```

```
        var root = null, temp = null,
        suc = null, min = null;
        root = insert(root, 20);
        root = insert(root, 8);
        root = insert(root, 22);
        root = insert(root, 4);
        root = insert(root, 12);
        root = insert(root, 10);
        root = insert(root, 14);
        temp = root.left.right.right;
        suc = inOrderSuccessor(root, temp);
        if (suc != null) {
            document.write("Inorder successor of " +
            temp.data + " is " + suc.data);
        } else {
            document.write(
            "Inorder successor does not exist"
            );
        }

// This code contributed by gauravrajput1

</script>
```

Output

```
Inorder Successor of 14 is 20
```

Complexity Analysis:

As in the second case(suppose skewed tree) we have to travel all the way towards the root.

- **Auxiliary Space:** O(1).
  Due to no use of any data structure for storing values.

Method 2 (Search from root)

Parent pointer is NOT needed in this algorithm. The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.

Output: *succ* // *succ* is Inorder successor of *node*.

1. If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do the following.
   Go to right subtree and return the node with minimum key value in the right subtree.
2. If right subtree of *node* is NULL, then start from the root and use search-like technique. Do the following.
   Travel down the tree, if a node's data is greater than root's data then go right side, otherwise, go to left side.

Below is the implementation of the above approach:

C++

```
// C++ program for above approach
#include <iostream>
using namespace std;

/* A binary tree node has data,
   the pointer to left child
   and a pointer to right child */
```

```
    struct node* left;
    struct node* right;
    struct node* parent;
};

struct node* minValue(struct node* node);

struct node* inOrderSuccessor(struct node* root,
                              struct node* n)
{

    // Step 1 of the above algorithm
    if (n->right != NULL)
        return minValue(n->right);

    struct node* succ = NULL;

    // Start from root and search for
    // successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
            break;
    }

    return succ;
```

```
// return the minimum data value found
// in that tree. Note that the entire
// tree does not need to be searched.
struct node* minValue(struct node* node)
{
    struct node* current = node;

    // Loop down to find the leftmost leaf
    while (current->left != NULL)
    {
        current = current->left;
    }
    return current;
}

// Helper function that allocates a new
// node with the given data and NULL left
// and right pointers.
struct node* newNode(int data)
{
    struct node* node = (struct node*)
    malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;

    return (node);
}

// Give a binary search tree and a
// number, inserts a new node with
// the given number in the correct
```

```
// avoid using reference parameters).
struct node* insert(struct node* node,
                    int data)
{

    /* 1. If the tree is empty, return a new,
       single node */
    if (node == NULL)
        return (newNode(data));
    else
    {
        struct node* temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
        {
            temp = insert(node->left, data);
            node->left = temp;
            temp->parent = node;
        }
        else
        {
            temp = insert(node->right, data);
            node->right = temp;
            temp->parent = node;
        }

        /* Return the (unchanged) node pointer */
        return node;
    }
}

// Driver code
```

```
    // Creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    // Function Call
    succ = inOrderSuccessor(root, temp);
    if (succ != NULL)
        cout << "\n Inorder Successor of "
             << temp->data << " is "<< succ->data;
    else
        cout <<"\n Inorder Successor doesn't exit";

    getchar();
    return 0;
}

// This code is contributed by shivanisinghss2110
```

C

```
// C program for above approach
#include <stdio.h>
#include <stdlib.h>
```

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
};

struct node* minValue(struct node* node);

struct node* inOrderSuccessor(
    struct node* root,
    struct node* n)
{

    // step 1 of the above algorithm
    if (n->right != NULL)
        return minValue(n->right);

    struct node* succ = NULL;

    // Start from root and search for
    // successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
```

```
    return succ;
}

/* Given a non-empty binary search tree,
    return the minimum data
    value found in that tree. Note that
    the entire tree does not need
    to be searched. */
struct node* minValue(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
    {
        current = current->left;
    }
    return current;
}

/* Helper function that allocates a new
    node with the given data and
    NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(
            struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;
```

```
/* Give a binary search tree and
   a number, inserts a new node with
    the given number in the correct
    place in the tree. Returns the new
    root pointer which the caller should
    then use (the standard trick to
    avoid using reference parameters). */
struct node* insert(struct node* node,
                    int data)
{
    /* 1. If the tree is empty, return a new,
      single node */
    if (node == NULL)
        return (newNode(data));
    else
    {
        struct node* temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
        {
            temp = insert(node->left, data);
            node->left = temp;
            temp->parent = node;
        }
        else
        {
            temp = insert(node->right, data);
            node->right = temp;
            temp->parent = node;
        }

        /* return the (unchanged) node pointer */
```

```
/* Driver program to test above functions*/
int main()
{
    struct node *root = NULL, *temp, *succ, *min;

    // creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    // Function Call
    succ = inOrderSuccessor(root, temp);
    if (succ != NULL)
        printf(
            "\n Inorder Successor of %d is %d ",
            temp->data, succ->data);
    else
        printf("\n Inorder Successor doesn't exit");

    getchar();
    return 0;
}

// Thanks to R.Srinivasan for suggesting this method.
```

```
// Java program for above approach
class GFG
{

/* A binary tree node has data,
   the pointer to left child
   and a pointer to right child */
static class node
{
    int data;
    node left;
    node right;
    node parent;
};

static node inOrderSuccessor(
    node root,
    node n)
{

    // step 1 of the above algorithm
    if (n.right != null)
        return minValue(n.right);

    node succ = null;

    // Start from root and search for
    // successor down the tree
    while (root != null)
    {
        if (n.data < root.data)
        {
            succ = root;
```

```
                root = root.right;
            else
                break;
        }
        return succ;
    }

    /* Given a non-empty binary search tree,
        return the minimum data
        value found in that tree. Note that
        the entire tree does not need
        to be searched. */
    static node minValue(node node)
    {
        node current = node;

        /* loop down to find the leftmost leaf */
        while (current.left != null)
        {
            current = current.left;
        }
        return current;
    }

    /* Helper function that allocates a new
        node with the given data and
        null left and right pointers. */
    static node newNode(int data)
    {
        node node = new node();
        node.data = data;
        node.left = null;
        node.right = null;
```

```
    }

    /* Give a binary search tree and
       a number, inserts a new node with
        the given number in the correct
        place in the tree. Returns the new
        root pointer which the caller should
        then use (the standard trick to
        astatic void using reference parameters). */
    static  node insert(node node,
                        int data)
    {

        /* 1. If the tree is empty, return a new,
          single node */
        if (node == null)
            return (newNode(data));
        else
        {
            node temp;

            /* 2. Otherwise, recur down the tree */
            if (data <= node.data)
            {
                temp = insert(node.left, data);
                node.left = temp;
                temp.parent = node;
            }
            else
            {
                temp = insert(node.right, data);
                node.right = temp;
                temp.parent = node;
```

```
            return node;
        }
    }

    /* Driver program to test above functions*/
    public static void main(String[] args)
    {
        node root = null, temp, succ, min;

        // creating the tree given in the above diagram
        root = insert(root, 20);
        root = insert(root, 8);
        root = insert(root, 22);
        root = insert(root, 4);
        root = insert(root, 12);
        root = insert(root, 10);
        root = insert(root, 14);
        temp = root.left.right.right;

        // Function Call
        succ = inOrderSuccessor(root, temp);
        if (succ != null)
            System.out.printf(
                "\n Inorder Successor of %d is %d ",
                temp.data, succ.data);
        else
            System.out.printf("\n Inorder Successor doesn't exit");
    }
    }

    // This code is contributed by gauravrajput1
```

```
# Python program to find
# the inorder successor in a BST

# A binary tree node
class Node:

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

def inOrderSuccessor(root, n):

    # Step 1 of the above algorithm
    if n.right is not None:
        return minValue(n.right)

    # Step 2 of the above algorithm
    succ=Node(None)


    while( root):
        if(root.data<n.data):
            root=root.right
        elif(root.data>n.data):
            succ=root
            root=root.left
        else:
            break
    return succ

# Given a non-empty binary search tree,
```

```
def minValue(node):
    current = node

    # loop down to find the leftmost leaf
    while(current is not None):
        if current.left is None:
            break
        current = current.left

    return current


# Given a binary search tree
# and a number, inserts a
# new node with the given
# number in the correct place
# in the tree. Returns the
# new root pointer which the
# caller should then use
# (the standard trick to avoid
# using reference parameters)
def insert( node, data):

    # 1) If tree is empty
    # then return a new singly node
    if node is None:
        return Node(data)
    else:

        # 2) Otherwise, recur down the tree
        if data <= node.data:
            temp = insert(node.left, data)
            node.left = temp
```

```
            node.right = temp
            temp.parent = node

        # return  the unchanged node pointer
        return node


# Driver program to test above function
if __name__ == "__main__":
  root = None

  # Creating the tree given in the above diagram
  root = insert(root, 20)
  root = insert(root, 8);
  root = insert(root, 22);
  root = insert(root, 4);
  root = insert(root, 12);
  root = insert(root, 10);
  root = insert(root, 14);
  temp = root.left.right

  succ = inOrderSuccessor( root, temp)
  if succ is not None:
      print("Inorder Successor of" ,
               temp.data ,"is" ,succ.data)
  else:
      print("InInorder Successor doesn't exist")
```

C#

```
// C# program for above approach
```

```
{

  /* A binary tree node has data,
   the pointer to left child
   and a pointer to right child */
  public
    class node
    {
      public
        int data;
      public
        node left;
      public
        node right;
      public
        node parent;
    };

  static node inOrderSuccessor(
    node root,
    node n)
  {

    // step 1 of the above algorithm
    if (n.right != null)
      return minValue(n.right);

    node succ = null;

    // Start from root and search for
    // successor down the tree
    while (root != null)
    {
```

```
        root = root.left;
      }
      else if (n.data > root.data)
        root = root.right;
      else
        break;
    }
    return succ;
  }

  /* Given a non-empty binary search tree,
    return the minimum data
    value found in that tree. Note that
    the entire tree does not need
    to be searched. */
  static node minValue(node node)
  {
    node current = node;

    /* loop down to find the leftmost leaf */
    while (current.left != null)
    {
      current = current.left;
    }
    return current;
  }

  /* Helper function that allocates a new
    node with the given data and
    null left and right pointers. */
  static node newNode(int data)
  {
    node node = new node();
```

```
  node.parent = null;

  return (node);
}

/* Give a binary search tree and
 a number, inserts a new node with
  the given number in the correct
  place in the tree. Returns the new
  root pointer which the caller should
  then use (the standard trick to
  astatic void using reference parameters). */
static  node insert(node node,
                    int data)
{

  /* 1. If the tree is empty, return a new,
    single node */
  if (node == null)
    return (newNode(data));
  else
  {
    node temp;

    /* 2. Otherwise, recur down the tree */
    if (data <= node.data)
    {
      temp = insert(node.left, data);
      node.left = temp;
      temp.parent = node;
    }
    else
    {
```

```
        }

        /* return the (unchanged) node pointer */
        return node;
      }
    }

    /* Driver program to test above functions*/
    public static void Main(String[] args)
    {
      node root = null, temp, succ;

      // creating the tree given in the above diagram
      root = insert(root, 20);
      root = insert(root, 8);
      root = insert(root, 22);
      root = insert(root, 4);
      root = insert(root, 12);
      root = insert(root, 10);
      root = insert(root, 14);
      temp = root.left.right.right;

      // Function Call
      succ = inOrderSuccessor(root, temp);
      if (succ != null)
        Console.Write(
        "\n Inorder Successor of {0} is {1} ",
        temp.data, succ.data);
      else
        Console.Write("\n Inorder Successor doesn't exit");
    }
  }
```

Javascript

```
<script>

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

function inOrderSuccessor(root,n)
{
    // step 1 of the above algorithm
    if (n.right != null)
        return minValue(n.right);

    let succ = null;

    // Start from root and search for
    // successor down the tree
    while (root != null)
    {
        if (n.data < root.data)
        {
            succ = root;
            root = root.left;
        }
        else if (n.data > root.data)
            root = root.right;
```

```
        return succ;
    }

    function minValue(node)
    {
        let current = node;

        /* loop down to find the leftmost leaf */
        while (current.left != null)
        {
            current = current.left;
        }
        return current;
    }


    function insert(node,data)
    {
        /* 1. If the tree is empty, return a new,
          single node */
        if (node == null)
            return (new Node(data));
        else
        {
            let temp;

            /* 2. Otherwise, recur down the tree */
            if (data <= node.data)
            {
                temp = insert(node.left, data);
                node.left = temp;
                temp.parent = node;
            }
```

```
            node.right = temp;
            temp.parent = node;
        }

        /* return the (unchanged) node pointer */
        return node;
    }
}

let root = null, temp, succ, min;

// creating the tree given in the above diagram
root = insert(root, 20);
root = insert(root, 8);
root = insert(root, 22);
root = insert(root, 4);
root = insert(root, 12);
root = insert(root, 10);
root = insert(root, 14);
temp = root.left.right.right;

// Function Call
succ = inOrderSuccessor(root, temp);
if (succ != null)
    document.write(
"<br> Inorder Successor of "+temp.data+"  is "+
 succ.data);
else
    document.write("<br> Inorder Successor doesn't exit");

// This code is contributed by unknown2108

</script>
```

```
Inorder Successor of 14 is 20
```

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).
  Due to no use of any data structure for storing values.

Method 3 (Inorder traversal) An inorder transversal of BST produces a sorted sequence. Therefore, we perform an inorder traversal. The first encountered node with value greater than the node is the inorder successor.

Input: node, root // node is the node whose ignorer successor is needed.

Output: succ // succ is Inorder successor of node.

Below is the implementation of the above approach:

C++

```
// C++ program for above approach
#include <iostream>
using namespace std;

/* A binary tree node has data,
   the pointer to left child
   and a pointer to right child */
```

```
    struct node* left;
    struct node* right;
    struct node* parent;
};
struct node* newNode(int data);

void inOrderTraversal(struct node* root,
                              struct node* n,
                              struct node* succ)
{
    if(root==nullptr) { return; }

    inOrderTraversal(root->left, n, succ);
    if(root->data>n->data && !succ->left) { succ->left = root; return; }
    inOrderTraversal(root->right, n, succ);
}

struct node* inOrderSuccessor(struct node* root,
                              struct node* n)
{
    struct node* succ = newNode(0);
    inOrderTraversal(root, n, succ);
    return succ->left;
}

// Helper function that allocates a new
// node with the given data and NULL left
// and right pointers.
struct node* newNode(int data)
{
    struct node* node = (struct node*)
    malloc(sizeof(struct node));
    node->data = data;
```

```
    return (node);
}

// Give a binary search tree and a
// number, inserts a new node with
// the given number in the correct
// place in the tree. Returns the new
// root pointer which the caller should
// then use (the standard trick to
// avoid using reference parameters).
struct node* insert(struct node* node,
                    int data)
{

    /* 1. If the tree is empty, return a new,
       single node */
    if (node == NULL)
        return (newNode(data));
    else
    {
        struct node* temp;

        /* 2. Otherwise, recur down the tree */
        if (data <= node->data)
        {
            temp = insert(node->left, data);
            node->left = temp;
            temp->parent = node;
        }
        else
        {
            temp = insert(node->right, data);
```

```
        /* Return the (unchanged) node pointer */
        return node;
    }
}

// Driver code
int main()
{
    struct node *root = NULL, *temp, *succ, *min;

    // Creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    // Function Call
    succ = inOrderSuccessor(root, temp);
    if (succ != NULL)
        cout << "\n Inorder Successor of "
             << temp->data << " is "<< succ->data;
    else
        cout <<"\n Inorder Successor doesn't exist";

    //getchar();
    return 0;
}
```

Java

```
// Java program for above approach
import java.util.*;

class GFG {

  /*
     * A binary tree node has data, the pointer to left child and a pointer to right
     * child
     */
  static class node {
    int data;
    node left;
    node right;
    node parent;
  };
  static void inOrderTraversal(node root) {
    if (root == null) {
      return;
    }

    inOrderTraversal(root.left);
    System.out.print(root.data);
    inOrderTraversal(root.right);
  }
  static void inOrderTraversal(node root, node n, node succ) {
    if (root == null) {
      return;
    }

    inOrderTraversal(root.left, n, succ);
```

```
    }
    inOrderTraversal(root.right, n, succ);
  }

  static node inOrderSuccessor(node root, node n) {
    node succ = newNode(0);
    inOrderTraversal(root, n, succ);
    return succ.left;
  }

  // Helper function that allocates a new
  // node with the given data and null left
  // and right pointers.
  static node newNode(int data) {
    node node = new node();

    node.data = data;
    node.left = null;
    node.right = null;
    node.parent = null;

    return (node);
  }

  // Give a binary search tree and a
  // number, inserts a new node with
  // the given number in the correct
  // place in the tree. Returns the new
  // root pointer which the caller should
  // then use (the standard trick to
  // astatic void using reference parameters).
  static node insert(node node, int data) {
```

```
    if (node == null)
      return (newNode(data));
    else {
      node temp;

      /* 2. Otherwise, recur down the tree */
      if (data <= node.data) {
        temp = insert(node.left, data);
        node.left = temp;
        temp.parent = node;
      } else {
        temp = insert(node.right, data);
        node.right = temp;
        temp.parent = node;
      }

      /* Return the (unchanged) node pointer */
      return node;
    }
  }

  // Driver code
  public static void main(String[] args) {
    node root = null, temp, succ, min;

    // Creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
```

```
        succ = inOrderSuccessor(root, temp);
        if (succ != null)
          System.out.print("\n Inorder Successor of " + temp.data + " is " + succ.data);
        else
          System.out.print("\n Inorder Successor doesn't exist");

      }
    }

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

C#

```
// C# program for above approach
using System;
using System.Collections.Generic;

public class GFG {

  /*
     * A binary tree node has data, the pointer to left child and a pointer to right
     * child
     */
 public  class node {
   public  int data;
   public  node left;
   public  node right;
   public  node parent;
  };
  static void inOrderTraversal(node root) {
    if (root == null) {
```

```
  inOrderTraversal(root.left);
  Console.Write(root.data);
  inOrderTraversal(root.right);
}
static void inOrderTraversal(node root, node n, node succ) {
  if (root == null) {
    return;
  }

  inOrderTraversal(root.left, n, succ);
  if (root.data > n.data && succ.left == null) {
    succ.left = root;
    return;
  }
  inOrderTraversal(root.right, n, succ);
}

static node inOrderSuccessor(node root, node n) {
  node succ = newNode(0);
  inOrderTraversal(root, n, succ);
  return succ.left;
}

// Helper function that allocates a new
// node with the given data and null left
// and right pointers.
static node newNode(int data) {
  node node = new node();

  node.data = data;
  node.left = null;
  node.right = null;
  node.parent = null;
```

```
// Give a binary search tree and a
// number, inserts a new node with
// the given number in the correct
// place in the tree. Returns the new
// root pointer which the caller should
// then use (the standard trick to
// astatic void using reference parameters).
static node insert(node node, int data) {

  /*
       * 1. If the tree is empty, return a new, single node
       */
  if (node == null)
    return (newNode(data));
  else {
    node temp;

    /* 2. Otherwise, recur down the tree */
    if (data <= node.data) {
      temp = insert(node.left, data);
      node.left = temp;
      temp.parent = node;
    } else {
      temp = insert(node.right, data);
      node.right = temp;
      temp.parent = node;
    }

    /* Return the (unchanged) node pointer */
    return node;
  }
}
```

```
        node root = null, temp, succ, min;

        // Creating the tree given in the above diagram
        root = insert(root, 20);
        root = insert(root, 8);
        root = insert(root, 22);
        root = insert(root, 4);
        root = insert(root, 12);
        root = insert(root, 10);
        root = insert(root, 14);
        temp = root.left.right.right;

        // Function Call
        succ = inOrderSuccessor(root, temp);
        if (succ != null)
          Console.Write("\n Inorder Successor of " + temp.data + " is " + succ.data);
        else
          Console.Write("\n Inorder Successor doesn't exist");

      }
    }

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

Javascript

```
<script>
// javascript program for above approach

    /*
     * A binary tree node has data, the pointer to left child and a pointer to right
```

```
 constructor(){
    this.data = 0;
    this.left = null;
    this.right = null;
    this.parent = null;
}
}

function inOrderTraversal( root) {
    if (root == null) {
        return;
    }

    inOrderTraversal(root.left);
    document.write(root.data);
    inOrderTraversal(root.right);
}

function inOrderTraversal( root,  n,  succ) {
    if (root == null) {
        return;
    }

    inOrderTraversal(root.left, n, succ);
    if (root.data > n.data && succ.left == null) {
        succ.left = root;
        return;
    }
    inOrderTraversal(root.right, n, succ);
}

 function inOrderSuccessor( root,  n) {
    var succ = newNode(0);
```

```
// Helper function that allocates a new
// node with the given data and null left
// and right pointers.
 function newNode(data) {
    var node = new Node();

    node.data = data;
    node.left = null;
    node.right = null;
    node.parent = null;

    return (node);
}

// Give a binary search tree and a
// number, inserts a new node with
// the given number in the correct
// place in the tree. Returns the new
// root pointer which the caller should
// then use (the standard trick to
// afunction using reference parameters).
 function insert( node , data) {

    /*
     * 1. If the tree is empty, return a new, single node
     */
    if (node == null)
        return (newNode(data));
    else {
        var temp;

        /* 2. Otherwise, recur down the tree */
```

```
                temp.parent = node;
            } else {
                temp = insert(node.right, data);
                node.right = temp;
                temp.parent = node;
            }

            /* Return the (unchanged) node pointer */
            return node;
        }
    }

// Driver code

    var root = null, temp, succ, min;

    // Creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root.left.right.right;

    // Function Call
    succ = inOrderSuccessor(root, temp);
    if (succ != null)
        document.write("\n Inorder Successor of " + temp.data +
        " is " + succ.data);
    else
        document.write("\n Inorder Successor doesn't exist");
```

Output

```
Inorder Successor of 14 is 20
```

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). Due to no use of any data structure for storing values.

Method 4 (Inorder traversal iterative) this method is inspired from the method 3 but with iterative and easy to understand approach.

Input: node, root // node is the node whose inorder successor is needed.
Output: succ // succ is Inorder successor of node.

Below is the implementation of the above approach:

Java

```
// Java program for above approach
import java.util.*;

class GFG {

  /*
```

```
  static class node {
    int data;
    node left;
    node right;
    node parent;
  };
  static void inOrderTraversal(node root) {
    if (root == null) {
      return;
    }

    inOrderTraversal(root.left);
    System.out.print(root.data);
    inOrderTraversal(root.right);
  }

 public static node inOrderSuccessor(node root, int key) {
        Deque<node> stack = new ArrayDeque<>();
        while(root != null || !stack.isEmpty()){
            while(root != null){
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            if(root.data > key)
                return root;
            root = root.right;
        }
        return null;
    }

  // Helper function that allocates a new
  // node with the given data and null left
```

```
    node.data = data;
    node.left = null;
    node.right = null;
    node.parent = null;

    return (node);
  }

  // Give a binary search tree and a
  // number, inserts a new node with
  // the given number in the correct
  // place in the tree. Returns the new
  // root pointer which the caller should
  // then use (the standard trick to
  // astatic void using reference parameters).
  static node insert(node node, int data) {

    /*
         * 1. If the tree is empty, return a new, single node
         */
    if (node == null)
      return (newNode(data));
    else {
      node temp;

      /* 2. Otherwise, recur down the tree */
      if (data <= node.data) {
        temp = insert(node.left, data);
        node.left = temp;
        temp.parent = node;
      } else {
        temp = insert(node.right, data);
```

```
      /* Return the (unchanged) node pointer */
      return node;
    }
  }

  // Driver code
  public static void main(String[] args) {
    node root = null, temp, succ, min;

    // Creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root.left.right.right;

    // Function Call
    succ = inOrderSuccessor(root, temp.data);
    if (succ != null)
      System.out.print("\n Inorder Successor of " + temp.data + " is " + succ.data);
    else
      System.out.print("\n Inorder Successor doesn't exist");

  }
}

// This code is contributed by Nitin Dhamija
```

```
Inorder Successor of 14 is 20
```

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). Due to no use of any data structure for storing values.

Next

Inorder predecessor and successor for a given key in BST

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

02 **Inorder predecessor and successor for a given key in BST**
25, Jul 14

03 **Pre-Order Successor of all nodes in Binary Search Tree**
14, Aug 19

04 **Check if an array represents Inorder of Binary Search tree or not**
29, Jun 17

05 **Construct a Binary Tree from Postorder and Inorder**
26, May 16

06 **Binary Tree to Binary Search Tree Conversion using STL set**
22, Mar 18

07 **Binary Tree to Binary Search Tree Conversion**
15, Jun 12

08 **Difference between Binary Tree and Binary Search Tree**
31, Oct 19

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Improved By : bidibaaz123, vipinyadav15799, reapedjuggler, pancudaniel, aashish1995, GauravRajput1, unknown2108, surinderdawra388, shivanisinghss2110, as5853535, jaisw7, amartyaghoshgfg, Rajput-Ji, nitin dhamija, hardikkoriintern

Article Tags : Amazon, Morgan Stanley, Binary Search Tree

Practice Tags : Morgan Stanley, Amazon, Binary Search Tree

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

feedback@geeksforgeeks.org