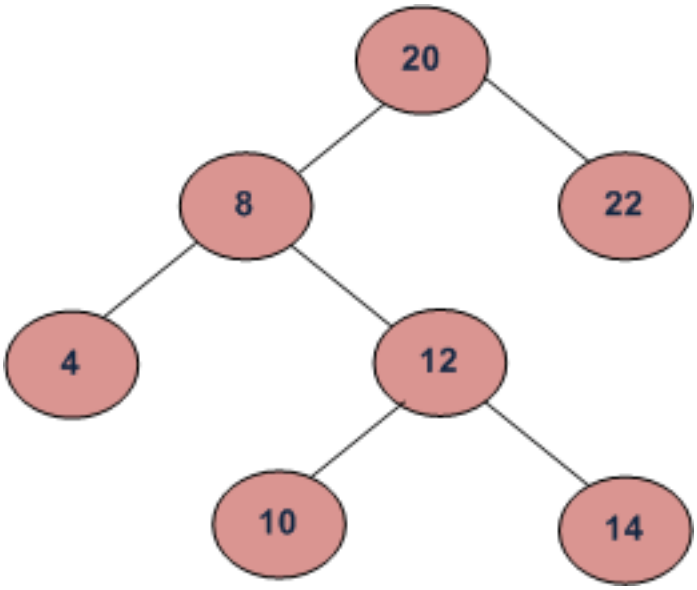
Contents

Chapter 1

Find the node with minimum value in a Binary Search Tree

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

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

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

```
};

/* 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;

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

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

/* 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. */
int minValue(struct node* node) {
  struct node* current = node;

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

```
/* Driver program to test sameTree function*/
int main()
{
  struct node* root = NULL;
  root = insert(root, 4);
  insert(root, 2);
  insert(root, 1);
  insert(root, 3);
  insert(root, 6);
  insert(root, 5);

  printf("\n Minimum value in BST is %d", minValue(root));
  getchar();
  return 0;
}
```

Time Complexity: O(n) Worst case happens for left skewed trees.

Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

References:
http://cslibrary.stanford.edu/110/BinaryTrees.html

Source

http://www.geeksforgeeks.org/find-the-minimum-element-in-a-binary-search-tree/

Category: Trees

Chapter 2

Inorder predecessor and successor for a given key in BST

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question:

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

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

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.

Following is the algorithm to reach the desired result. Its 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.
    b. If its right subtree is not null
        The successor will be the left most child
        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 C++ implementation of the above algorithm:

```
// 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)
{
    // 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)
                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, 30);
    insert(root, 20);
    insert(root, 40);
    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;
}
```

Output:

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

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.

Source

http://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/

Category: Trees

Post navigation

← An Interesting Method to Generate Binary Numbers from 1 to n Amazon Interview | Set 97 (On-Campus for SDE1) →

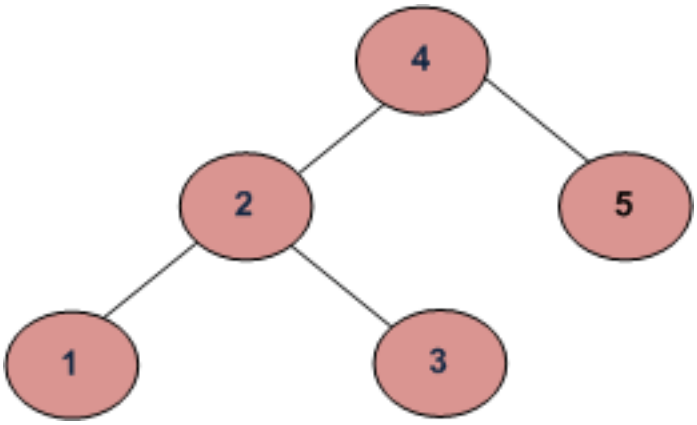
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 3

A program to check if a binary tree is BST or not

A binary search tree (BST) is a node based binary tree data structure which has the following properties.
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:
- Each node (item in the tree) has a distinct key.

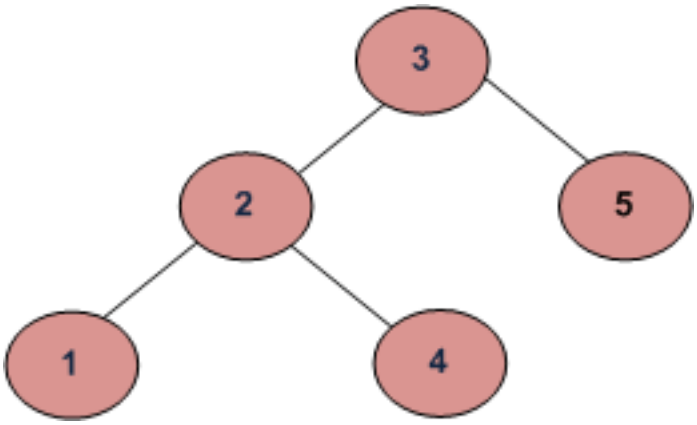


METHOD 1 (Simple but Wrong)
Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```
int isBST(struct node* node)
{
  if (node == NULL)
    return 1;

  /* false if left is > than node */
  if (node->left != NULL && node->left->data > node->data)
    return 0;
```

```
  /* false if right is < than node */
  if (node->right != NULL && node->right->data < node->data)
    return 0;

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return 0;

  /* passing all that, it's a BST */
  return 1;
}
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



METHOD 2 (Correct but not efficient)
For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
  if (node == NULL)
    return(true);

  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
    return(false);

  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) < node->data)
    return(false);

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  /* passing all that, it's a BST */
```

```
  return(true);
}
```

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)
Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTUtil(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

```
/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 values are >= min and
Implementation:

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

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

int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{

  /* an empty tree is BST */
```

```
  if (node==NULL)
     return 1;

  /* false if this node violates the min/max constraint */
  if (node->data < min || node->data > max)
     return 0;

  /* otherwise check the subtrees recursively,
   tightening the min or max constraint */
  return
    isBSTUtil(node->left, min, node->data-1) &&  // Allow only distinct values
    isBSTUtil(node->right, node->data+1, max);  // Allow only distinct values
}

/* 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;

  return(node);
}

/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(4);
  root->left        = newNode(2);
  root->right       = newNode(5);
  root->left->left  = newNode(1);
  root->left->right = newNode(3);

  if(isBST(root))
    printf("Is BST");
  else
    printf("Not a BST");

  getchar();
  return 0;
}
```

Time Complexity: O(n)
Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

METHOD 4(Using In-Order Traversal)
Thanks to LJW489 for suggesting this method.
1) Do In-Order Traversal of the given tree and store the result in a temp array.
3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to ygosfor this space optimization.

```
bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
          return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
          return false;

        prev = root;

        return isBST(root->right);
    }

    return true;
}
```

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to thispost).

Sources:
http://en.wikipedia.org/wiki/Binary_search_tree
http://cslibrary.stanford.edu/110/BinaryTrees.html

Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/

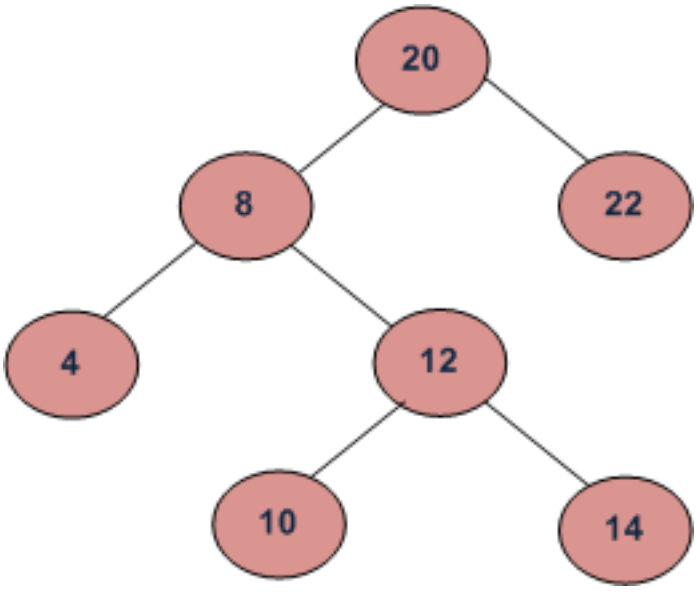
Chapter 4

Lowest Common Ancestor in a Binary Search Tree.

Given values of two nodes in a Binary Search Tree, write a c program to find the **L**owest **C**ommon **A**ncestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

```
struct node *lca(node* root, int n1, int n2)
n1 and n2 are two given values in the tree with given root.
```



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

Following is definition of LCA from Wikipedia:
Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining

the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source Wiki)

Solutions:
If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can **recursively traverse** the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 // A recursive C program to find LCA of two nodes n1 and n2. #include <stdio.h> #include <stdlib.h> struct node { int data; struct node* left, *right; }; /* Function to find LCA of n1 and n2. The function assumes that both n1 and n2 are present in BST */ struct node *lca(struct node* root, int n1, int n2) { if (root == NULL) return NULL; // If both n1 and n2 are smaller than root, then LCA lies in left if (root->data > n1 && root->data > n2) return lca(root->left, n1, n2); // If both n1 and n2 are greater than root, then LCA lies in right if (root->data < n1 && root->data < n2) return lca(root->right, n1, n2); return root; } /* Helper function that allocates a new node with the given data.*/ struct node* newNode(int data) { struct node* node = (struct node*)malloc(sizeof(struct node)); node->data = data; node->left = node->right = NULL; return(node); } /* Driver program to test mirror() */ int main() { // Let us construct the BST shown in the above figure struct node *root = newNode(20); root->left = newNode(8); root->right = newNode(22); root->left->left = newNode(4); root->left->right = newNode(12); root->left->right->left = newNode(10); root->left->right->right = newNode(14); int n1 = 10, n2 = 14; struct node *t = lca(root, n1, n2); printf("LCA of %d and %d is %d \n", n1, n2, t->data); n1 = 14, n2 = 8; t = lca(root, n1, n2); printf("LCA of %d and %d is %d \n", n1, n2, t->data); n1 = 10, n2 = 22; t = lca(root, n1, n2); printf("LCA of %d and %d is %d \n", n1, n2, t->data); getchar(); return 0; }

Output:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time complexity of above solution is O(h) where h is height of tree. Also, the above solution requires O(h) extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```
/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
         // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
           root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
           root = root->right;

        else break;
    }
```

```
    return root;
}
```

See thisfor complete program.

You may like to see Lowest Common Ancestor in a Binary Tree also.

Exercise
The above functions assume that n1 and n2 both are in BST. If n1 and n2 are not present, then they may return incorrect result. Extend the above solutions to return NULL if n1 or n2 or both not present in BST.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

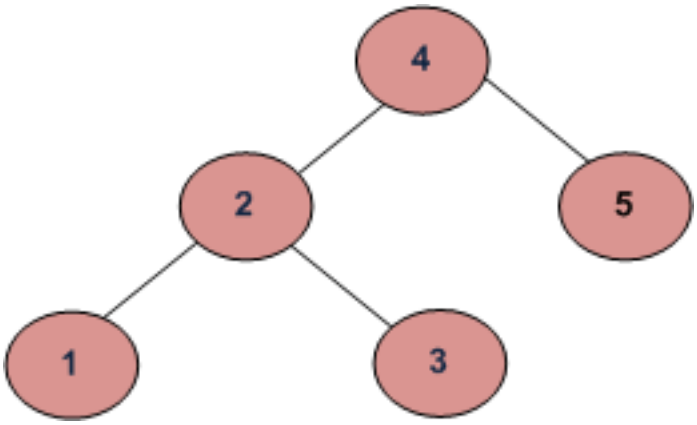
http://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-search-tree/

Chapter 5

Sorted order printing of a given array that represents a BST

Given an array that stores a complete Binary Search Tree, write a function that efficiently prints the given array in ascending order.

For example, given an array [4, 2, 5, 1, 3], the function should print 1, 2, 3, 4, 5



Solution:
Inorder traversal of BST prints it in ascending order. The only trick is to modify recursion termination condition in standard Inorder Tree Traversal.

Implementation:

```
#include<stdio.h>

void printSorted(int arr[], int start, int end)
{
  if(start > end)
    return;

  // print left subtree
  printSorted(arr, start*2 + 1, end);

  // print root
```

```
  printf("%d  ", arr[start]);

  // print right subtree
  printSorted(arr, start*2 + 2, end);
}

int main()
{
  int arr[] = {4, 2, 5, 1, 3};
  int arr_size = sizeof(arr)/sizeof(int);
  printSorted(arr, 0, arr_size-1);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Please write comments if you find the above solution incorrect, or find better ways to solve the same problem.

Source

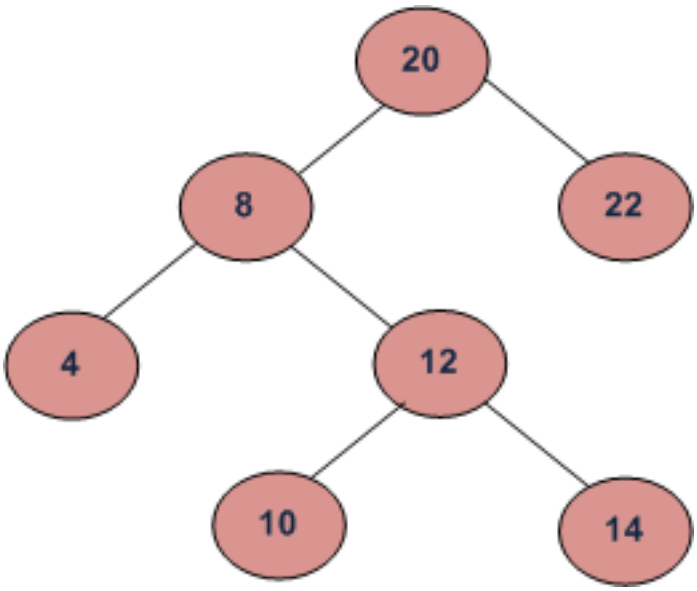
http://www.geeksforgeeks.org/sorted-order-printing-of-an-array-that-represents-a-bst/

Category: Trees

Chapter 6

Inorder Successor in Binary Search Tree

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 Inoorder 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 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**.

Method 1 (Uses Parent Pointer)
In this method, we assume that every node has parent pointer.

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 following.
Go to right subtree and return the node with minimum key value in right subtree.
2) If right sbtree of *node* is NULL, then *succ* is one of the ancestors. Do following.

Travel up using the parent pointer until you see a node which is left child of it's 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.

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

/* A binary tree node has data, 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 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;

  /* 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 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 */
    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;
}
```

Output of the above program:
Inorder Successor of 14 is 20

Time Complexity: O(h) where h is height of tree.

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 following.
Go to right subtree and return the node with minimum key value in right subtree.
2) If right sbtree of *node* is NULL, then start from root and us search like technique. Do 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.

```
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;
```

```
}
```

Thanks to R.Srinivasan for suggesting this method.

Time Complexity: O(h) where h is height of tree.

References:

http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap13.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

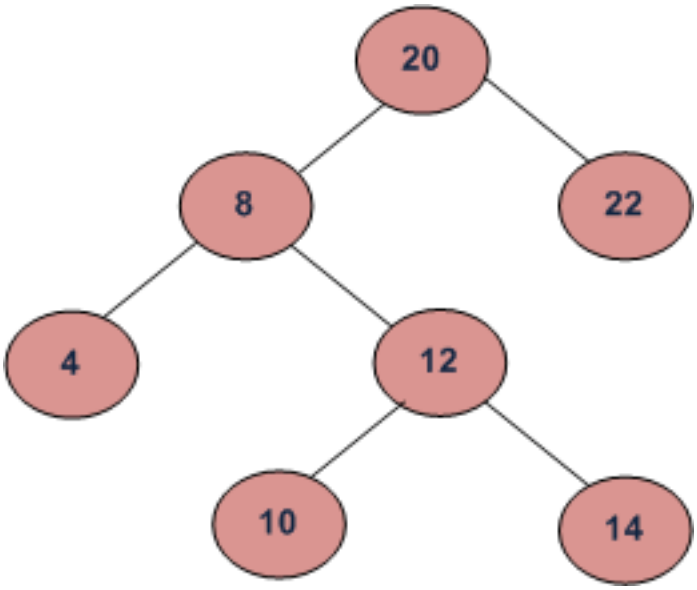
http://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/

Chapter 7

Find k-th smallest element in BST (Order Statistics in BST)

Given root of binary search tree and K as input, find K-th smallest element in BST.

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



Method 1: Using Inorder Traversal.

Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes of tree (threaded tree avoids stack and recursion for traversal, see this post). The idea is to keep track of popped elements which participate in the order statics. Hypothetical algorithm is provided below,

Time complexity: O(n) where n is total nodes in tree..

Algorithm:

```
 /* initialization */
pCrawl = root
set initial stack element as NULL (sentinal)
```

```
/* traverse upto left extreme */
while(pCrawl is valid )
   stack.push(pCrawl)
   pCrawl = pCrawl.left

/* process other nodes */
while( pCrawl = stack.pop() is valid )
   stop if sufficient number of elements are popped.
   if( pCrawl.right is valid )
      pCrawl = pCrawl.right
      while( pCrawl is valid )
         stack.push(pCrawl)
         pCrawl = pCrawl.left
```

Implementation:

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

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

/* just add elements to test */
/* NOTE: A sorted array results in skewed tree */
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };

/* same alias */
typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;

    node_t* left;
    node_t* right;
};

/* simple stack that stores node addresses */
typedef struct stack_t stack_t;

/* initial element always NULL, uses as sentinal */
struct stack_t
{
    node_t*  base[ARRAY_SIZE(ele) + 1];
    int      stackIndex;
};

/* pop operation of stack */
node_t *pop(stack_t *st)
{
    node_t *ret = NULL;

    if( st && st->stackIndex > 0 )
```

```
    {
        ret = st->base[st->stackIndex];
        st->stackIndex--;
    }

    return ret;
}

/* push operation of stack */
void push(stack_t *st, node_t *node)
{
    if( st )
    {
        st->stackIndex++;
        st->base[st->stackIndex] = node;
    }
}

/* Iterative insertion
   Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
```

```
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data   = keys[iterator];
        new_node->left   = NULL;
        new_node->right  = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

node_t *k_smallest_element_inorder(stack_t *stack, node_t *root, int k)
{
    stack_t *st = stack;
    node_t *pCrawl = root;

    /* move to left extremen (minimum) */
    while( pCrawl )
    {
        push(st, pCrawl);
        pCrawl = pCrawl->left;
    }

    /* pop off stack and process each node */
    while( pCrawl = pop(st) )
    {
        /* each pop operation emits one element
           in the order
        */
        if( !--k )
        {
            /* loop testing */
            st->stackIndex = 0;
            break;
```

```
        }

        /* there is right subtree */
        if( pCrawl->right )
        {
            /* push the left subtree of right subtree */
            pCrawl = pCrawl->right;
            while( pCrawl )
            {
                push(st, pCrawl);
                pCrawl = pCrawl->left;
            }

            /* pop off stack and repeat */
        }
    }

    /* node having k-th element or NULL node */
    return pCrawl;
}

/* Driver program to test above functions */
int main(void)
{
    node_t* root = NULL;
    stack_t stack = { {0}, 0 };
    node_t *kNode = NULL;

    int k = 5;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    kNode = k_smallest_element_inorder(&stack, root, k);

    if( kNode )
    {
        printf("kth smallest elment for k = %d is %d", k, kNode->data);
    }
    else
    {
        printf("There is no such element");
    }

    getchar();
    return 0;
}
```

Method 2: Augmented Tree Data Structure.

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If K = N + 1, root is K-th node. If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element. Note that we need the count of elements in left subtree only.

Time complexity: O(h) where h is height of tree.

Algorithm:

```
 start:
if K = root.leftElement + 1
   root node is the K th node.
   goto stop
else if K > root.leftElements
   K = K - (root.leftElements + 1)
   root = root.right
   goto start
else
   root = root.left
   goto srart

stop:
```

Implementation:

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

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;
    int lCount;

    node_t* left;
    node_t* right;
};

/* Iterative insertion
   Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
```

```
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* We are branching to left subtree
               increment node count */
            pTraverse->lCount++;
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data   = keys[iterator];
        new_node->lCount = 0;
        new_node->left   = NULL;
        new_node->right  = NULL;

        /* insert into BST */
```

```
        root = insert_node(root, new_node);
    }

    return root;
}

int k_smallest_element(node_t *root, int k)
{
    int ret = -1;

    if( root )
    {
        /* A crawling pointer */
        node_t *pTraverse = root;

        /* Go to k-th smallest */
        while(pTraverse)
        {
            if( (pTraverse->lCount + 1) == k )
            {
                ret = pTraverse->data;
                break;
            }
            else if( k > pTraverse->lCount )
            {
                /*  There are less nodes on left subtree
                    Go to right subtree */
                k = k - (pTraverse->lCount + 1);
                pTraverse = pTraverse->right;
            }
            else
            {
                /* The node is on left subtree */
                pTraverse = pTraverse->left;
            }
        }
    }

    return ret;
}

/* Driver program to test above functions */
int main(void)
{
    /* just add elements to test */
    /* NOTE: A sorted array results in skewed tree */
    int ele[] = { 20, 8, 22, 4, 12, 10, 14 };
    int i;
    node_t* root = NULL;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    /*  It should print the sorted array */
```

```
    for(i = 1; i <= ARRAY_SIZE(ele); i++)
    {
        printf("\n kth smallest elment for k = %d is %d",
                 i, k_smallest_element(root, i));
    }

    getchar();
    return 0;
}
```

Thanks to **Venki** for providing post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-k-th-smallest-element-in-bst-order-statistics-in-bst/

Category: Trees

Post navigation

← Data Structures and Algorithms | Set 26 C Language | Set 8 →

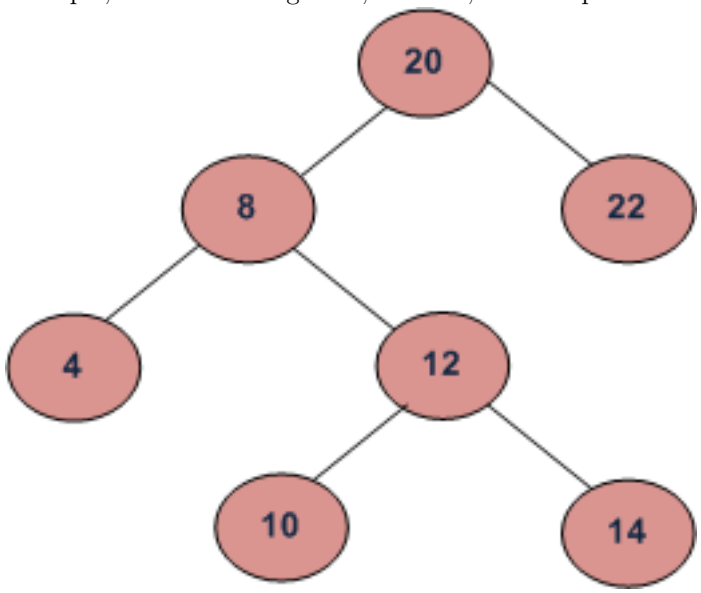
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 8

K'th smallest element in BST using O(1) Extra Space

Given a Binary Search Tree (BST) and a positive integer k, find the k'th smallest element in the Binary Search Tree.

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



We have discussed two methods in thispost and one method in thispost. All of the previous methods require extra space. How to find the k'th largest element without extra space?

We strongly recommend to minimize your browser and try this yourself first. Implementation

The idea is to use Morris Traversal. In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree. See thisfor more details.

Below is C++ implementation of the idea.

```
// C++ program to find k'th largest element in BST
#include<iostream>
#include<climits>
```

```
using namespace std;

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

// A function to find
int KSmallestUsingMorris(Node *root, int k)
{
    // Count to iterate over elements till we
    // get the kth smallest number
    int count = 0;

    int ksmall = INT_MIN; // store the Kth smallest
    Node *curr = root; // to store the current node

    while (curr != NULL)
    {
        // Like Morris traversal if current does
        // not have left child rather than printing
        // as we did in inorder, we will just
        // increment the count as the number will
        // be in an increasing order
        if (curr->left == NULL)
        {
            count++;

            // if count is equal to K then we found the
            // kth smallest, so store it in ksmall
            if (count==k)
                ksmall = curr->key;

            // go to current's right child
            curr = curr->right;
        }
        else
        {
            // we create links to Inorder Successor and
            // count using these links
            Node *pre = curr->left;
            while (pre->right != NULL && pre->right != curr)
                pre = pre->right;

            // building links
            if (pre->right==NULL)
            {
                //link made to Inorder Successor
                pre->right = curr;
                curr = curr->left;
            }
```

```
            // While breaking the links in so made temporary
            // threaded tree we will check for the K smallest
            // condition
            else
            {
                // Revert the changes made in if part (break link
                // from the Inorder Successor)
                pre->right = NULL;

                count++;

                // If count is equal to K then we found
                // the kth smallest and so store it in ksmall
                if (count==k)
                    ksmall = curr->key;

                curr = curr->right;
            }
        }
    }
    return ksmall; //return the found value
}

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

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
```

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

    for (int k=1; k<=7; k++)
       cout << KSmallestUsingMorris(root, k) << " ";

    return 0;
}
```

Output:

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

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

Source

http://www.geeksforgeeks.org/kth-largest-element-in-bst-using-o1-extra-space/

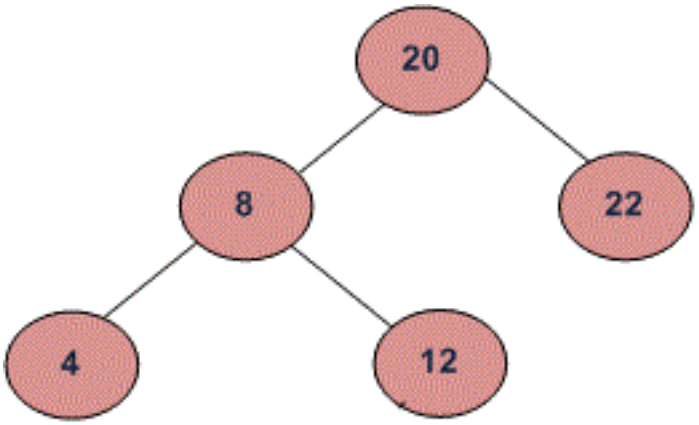
Category: Trees

Chapter 9

Print BST keys in the given range

Given two values k1 and k2 (where k1

For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.



Thanks to bhasker for suggesting the following solution.

Algorithm:
1) If value of root's key is greater than k1, then recursively call in left subtree.
2) If value of root's key is in range, then print the root's key.
3) If value of root's key is smaller than k2, then recursively call in right subtree.

Implementation:

```
 #include<stdio.h>

/* A tree node structure */
struct node
{
  int data;
  struct node *left;
  struct node *right;
};

/* The functions prints all the keys which in the given range [k1..k2].
```

```
    The function assumes than k1 < k2 */
void Print(struct node *root, int k1, int k2)
{
   /* base case */
   if ( NULL == root )
      return;

   /* Since the desired o/p is sorted, recurse for left subtree first
      If root->data is greater than k1, then only we can get o/p keys
      in left subtree */
   if ( k1 < root->data )
     Print(root->left, k1, k2);

   /* if root's data lies in range, then prints root's data */
   if ( k1 <= root->data && k2 >= root->data )
     printf("%d ", root->data );

  /* If root->data is smaller than k2, then only we can get o/p keys
      in right subtree */
   if ( k2 > root->data )
     Print(root->right, k1, k2);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
  struct node *temp = new struct node;
  temp->data = data;
  temp->left = NULL;
  temp->right = NULL;

  return temp;
}

/* Driver function to test above functions */
int main()
{
  struct node *root = new struct node;
  int k1 = 10, k2 = 25;

  /* Constructing tree given in the above figure */
  root = newNode(20);
  root->left = newNode(8);
  root->right = newNode(22);
  root->left->left = newNode(4);
  root->left->right = newNode(12);

  Print(root, k1, k2);

  getchar();
  return 0;
}
```

Output:
12 20 22

Time Complexity: O(n) where n is the total number of keys in tree.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/print-bst-keys-in-the-given-range/

Category: Trees

Chapter 10

Sorted Array to Balanced BST

Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.

Examples:

```
Input:  Array {1, 2, 3}
Output: A Balanced BST
     2
   /  \
  1    3

Input: Array {1, 2, 3, 4}
Output: A Balanced BST
      3
    /  \
   2    4
 /
1
```

Algorithm
In the previous post, we discussed construction of BST from sorted Linked List. Constructing from sorted array in O(n) time is simpler as we can get the middle element in O(1) time. Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the array and make it root.
2) Recursively do same for left half and right half.
      a) Get the middle of left half and make it left child of the root
          created in step 1.
      b) Get the middle of right half and make it right child of the
          root created in step 1.
```

Following is C implementation of the above algorithm. The main code which creates Balanced BST is highlighted.

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

/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);

/* A function that constructs Balanced Binary Search Tree from a sorted array */
struct TNode* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
      return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct TNode *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */
    root->left =  sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}

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

    return node;
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
```

```
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    /* Convert List to BST */
    struct TNode *root = sortedArrayToBST(arr, 0, n-1);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Time Complexity: O(n)
Following is the recurrance relation for sortedArrayToBST().

```
  T(n) = 2T(n/2) + C
  T(n) -->  Time taken for an array of size n
   C   -->  Constant (Finding middle of array and linking root to left
                      and right subtrees take constant time)
```

The above recurrence can be solved using Master Theorem as it falls in case 2.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sorted-array-to-balanced-bst/

Category: Trees

Chapter 11

Find the largest BST subtree in a given Binary Tree

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

```
Input:
      5
    /  \
   2    4
 /  \
1    3

Output: 3
The following subtree is the maximum size BST subtree
   2
 /  \
1    3


Input:
       50
     /    \
  30       60
 /  \     /  \
5   20   45    70
              /  \
            65    80
Output: 5
The following subtree is the maximum size BST subtree
      60
     /  \
   45    70
        /  \
      65    80
```

Method 1 (Simple but inefficient)
Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees and return the maximum of values returned by left and right subtrees.

```
/*
  See http://www.geeksforgeeks.org/archives/632 for implementation of size()

  See Method 3 of http://www.geeksforgeeks.org/archives/3042 for
  implementation of isBST()

  max() returns maximum of two integers
*/
int largestBST(struct node *root)
{
   if (isBST(root))
     return size(root);
   else
    return max(largestBST(root->left), largestBST(root->right));
}
```

Time Complexity: The worst case time complexity of this method will be O(n^2). Consider a skewed tree for worst case analysis.

Method 2 (Tricky and Efficient)
In method 1, we traverse the tree in top down manner and do BST test for every node. If we traverse the tree in bottom up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or O(1) time). A left subtree need to tell the parent whether it is BST or not and also need to pass maximum value in it. So that we can compare the maximum value with the parent's data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.
1) Whether the subtree itself is BST or not (In the following code, is_bst_ref is used for this purpose)
2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right subtree then minimum value in it.
3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTtil() is used for this purpose)

max_ref is used for passing the maximum value up the tree and min_ptr is used for passing minimum value up the tree.

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

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

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

/* 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;

  return(node);
}

int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                           int *max_size_ref, bool *is_bst_ref);

/* Returns size of the largest BST subtree in a Binary Tree
  (efficient version). */
int largestBST(struct node* node)
{
  // Set the initial values for calling largestBSTUtil()
  int min = INT_MAX;  // For minimum value in right subtree
  int max = INT_MIN;  // For maximum value in left subtree

  int max_size = 0;  // For size of the largest BST
  bool is_bst = 0;

  largestBSTUtil(node, &min, &max, &max_size, &is_bst);

  return max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree.   Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                           int *max_size_ref, bool *is_bst_ref)
{

  /* Base Case */
  if (node == NULL)
  {
     *is_bst_ref = 1; // An empty tree is BST
     return 0;    // Size of the BST is 0
  }

  int min = INT_MAX;
```

```
  /* A flag variable for left subtree property
     i.e., max(root->left) < root->data */
  bool left_flag = false;

  /* A flag variable for right subtree property
     i.e., min(root->right) > root->data */
  bool right_flag = false;

  int ls, rs; // To store sizes of left and right subtrees

  /* Following tasks are done by recursive call for left subtree
    a) Get the maximum value in left subtree (Stored in *max_ref)
    b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
    c) Get the size of maximum size BST in left subtree (updates *max_size) */
  *max_ref = INT_MIN;
  ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref, is_bst_ref);
  if (*is_bst_ref == 1 && node->data > *max_ref)
     left_flag = true;

  /* Before updating *min_ref, store the min value in left subtree. So that we
     have the correct minimum value for this subtree */
  min = *min_ref;

  /* The following recursive call does similar (similar to left subtree)
    task for right subtree */
  *min_ref =  INT_MAX;
  rs = largestBSTUtil(node->right, min_ref, max_ref, max_size_ref, is_bst_ref);
  if (*is_bst_ref == 1 && node->data < *min_ref)
     right_flag = true;

  // Update min and max values for the parent recursive calls
  if (min < *min_ref)
     *min_ref = min;
  if (node->data < *min_ref) // For leaf nodes
     *min_ref = node->data;
  if (node->data > *max_ref)
     *max_ref = node->data;

  /* If both left and right subtrees are BST. And left and right
     subtree properties hold for this node, then this tree is BST.
     So return the size of this tree */
  if(left_flag && right_flag)
  {
     if (ls + rs + 1 > *max_size_ref)
         *max_size_ref = ls + rs + 1;
     return ls + rs + 1;
  }
  else
  {
    //Since this subtree is not BST, set is_bst flag for parent calls
     *is_bst_ref = 0;
     return 0;
  }
}
```

```
/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
          50
        /      \
      10        60
     /  \       /  \
   5   20    55    70
            /     /  \
          45     65    80
  */

  struct node *root = newNode(50);
  root->left        = newNode(10);
  root->right       = newNode(60);
  root->left->left  = newNode(5);
  root->left->right = newNode(20);
  root->right->left  = newNode(55);
  root->right->left->left  = newNode(45);
  root->right->right = newNode(70);
  root->right->right->left = newNode(65);
  root->right->right->right = newNode(80);

  /* The complete tree is not BST as 45 is in right subtree of 50.
     The following subtree is the largest BST
        60
      /  \
    55    70
   /     /  \
 45     65    80
  */
  printf(" Size of the largest BST is %d", largestBST(root));

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst/

Category: Trees

Post navigation

← Complicated declarations in C Analysis of Algorithms | Set 2 (Worst, Average and Best Cases) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 12

Check for Identical BSTs without building the trees

Given two arrays which represent a sequence of keys. Imagine we make a Binary Search Tree (BST) from each array. We need to tell whether two BSTs will be identical or not without actually constructing the tree.

Examples
For example, the input arrays are {2, 4, 3, 1} and {2, 1, 4, 3} will construct the same tree

```
  Let the input arrays be a[] and b[]

Example 1:
a[] = {2, 4, 1, 3} will construct following tree.
   2
 /  \
1    4
    /
   3
b[] = {2, 4, 3, 1} will also also construct the same tree.
   2
 /  \
1    4
    /
   3
So the output is "True"

Example 2:
a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13}
b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13}

They both construct the same following BST, so output is "True"
            8
          /    \
        3       10
      /  \        \
    1     6       14
        /   \     /
       4     7   13
```

Solution:
According to BST property, elements of left subtree must be smaller and elements of right subtree must be greater than root.
Two arrays represent sane BST if for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees.
The idea is to check of if next smaller and greater elements are same in both arrays. Same properties are recursively checked for left and right subtrees. The idea looks simple, but implementation requires checking all conditions for all elements. Following is an interesting recursive implementation of the idea.

```
// A C program to check for Identical BSTs without building the trees
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>

/* The main function that checks if two arrays a[] and b[] of size n construct
  same BST. The two values 'min' and 'max' decide whether the call is made for
  left subtree or right subtree of a parent element. The indexes i1 and i2 are
  the indexes in (a[] and b[]) after which we search the left or right child.
  Initially, the call is made for INT_MIN and INT_MAX as 'min' and 'max'
  respectively, because root has no parent.
  i1 and i2 are just after the indexes of the parent element in a[] and b[]. */
bool isSameBSTUtil(int a[], int b[], int n, int i1, int i2, int min, int max)
{
   int j, k;

   /* Search for a value satisfying the constraints of min and max in a[] and
      b[]. If the parent element is a leaf node then there must be some
      elements in a[] and b[] satisfying constraint. */
   for (j=i1; j<n; j++)
       if (a[j]>min && a[j]<max)
           break;
   for (k=i2; k<n; k++)
       if (b[k]>min && b[k]<max)
           break;

   /* If the parent element is leaf in both arrays */
   if (j==n && k==n)
       return true;

   /* Return false if any of the following is true
      a) If the parent element is leaf in one array, but non-leaf in other.
      b) The elements satisfying constraints are not same. We either search
         for left child or right child of the parent element (decinded by min
         and max values). The child found must be same in both arrays */
   if (((j==n)^(k==n)) || a[j]!=b[k])
       return false;

   /* Make the current child as parent and recursively check for left and right
      subtrees of it. Note that we can also pass a[k] in place of a[j] as they
      are both are same */
   return isSameBSTUtil(a, b, n, j+1, k+1, a[j], max) &&  // Right Subtree
          isSameBSTUtil(a, b, n, j+1, k+1, min, a[j]);    // Left Subtree
}
```

```
// A wrapper over isSameBSTUtil()
bool isSameBST(int a[], int b[], int n)
{
   return isSameBSTUtil(a, b, n, 0, 0, INT_MIN, INT_MAX);
}

// Driver program to test above functions
int main()
{
   int a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13};
   int b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13};
   int n=sizeof(a)/sizeof(a[0]);
   printf("%s\n", isSameBST(a, b, n)?
             "BSTs are same":"BSTs not same");
   return 0;
}
```

Output:

```
BSTs are same
```

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

Source

http://www.geeksforgeeks.org/check-for-identical-bsts-without-building-the-trees/

Category: Trees

Post navigation

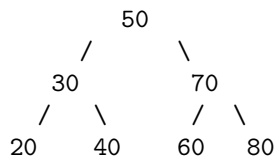
← Dynamic Programming | Set 33 (Find if a string is interleaved of two other strings) Can we Overload or Override static methods in java ? →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

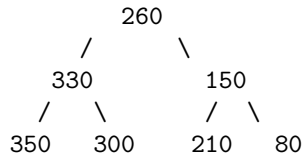
Chapter 13

Add all greater values to every node in a given BST

Given a **B**inary **S**earch **T**ree (BST), modify it so that all greater values in the given BST are added to every node. For example, consider the following BST.



The above tree should be modified to following



We strongly recommend you to minimize the browser and try this yourself first.

A **simple method** for solving this is to find sum of all greater values for every node. This method would take O(n^2) time.
We can do it **using a single traversal**. The idea is to use following BST property. If we do reverse Inorder traversal of BST, we get all nodes in decreasing order. We do reverse Inorder traversal and keep track of the sum of all nodes visited so far, we add this sum to every node.

```
// C program to add all greater values in every node of BST
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
```

```
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to add all greater values in every node
void modifyBSTUtil(struct node *root, int *sum)
{
    // Base Case
    if (root == NULL)  return;

    // Recur for right subtree
    modifyBSTUtil(root->right, sum);

    // Now *sum has sum of nodes in right subtree, add
    // root->data to sum and update root->data
    *sum = *sum + root->data;
    root->data = *sum;

    // Recur for left subtree
    modifyBSTUtil(root->left, sum);
}

// A wrapper over modifyBSTUtil()
void modifyBST(struct node *root)
{
    int sum = 0;
    modifyBSTUtil(root, &sum);
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given data in BST */
struct node* insert(struct node* node, int data)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(data);

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

```
    if (data <= node->data)
        node->left  = insert(node->left, data);
    else
        node->right = insert(node->right, data);

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

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    modifyBST(root);

    // print inoder tarversal of the modified BST
    inorder(root);

    return 0;
}
```

Output

```
 350 330 300 260 210 150 80
```

Time Complexity: O(n) where n is number of nodes in the given BST.

As a side note, we can also use reverse Inorder traversal to find kth largest element in a BST.

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

Source

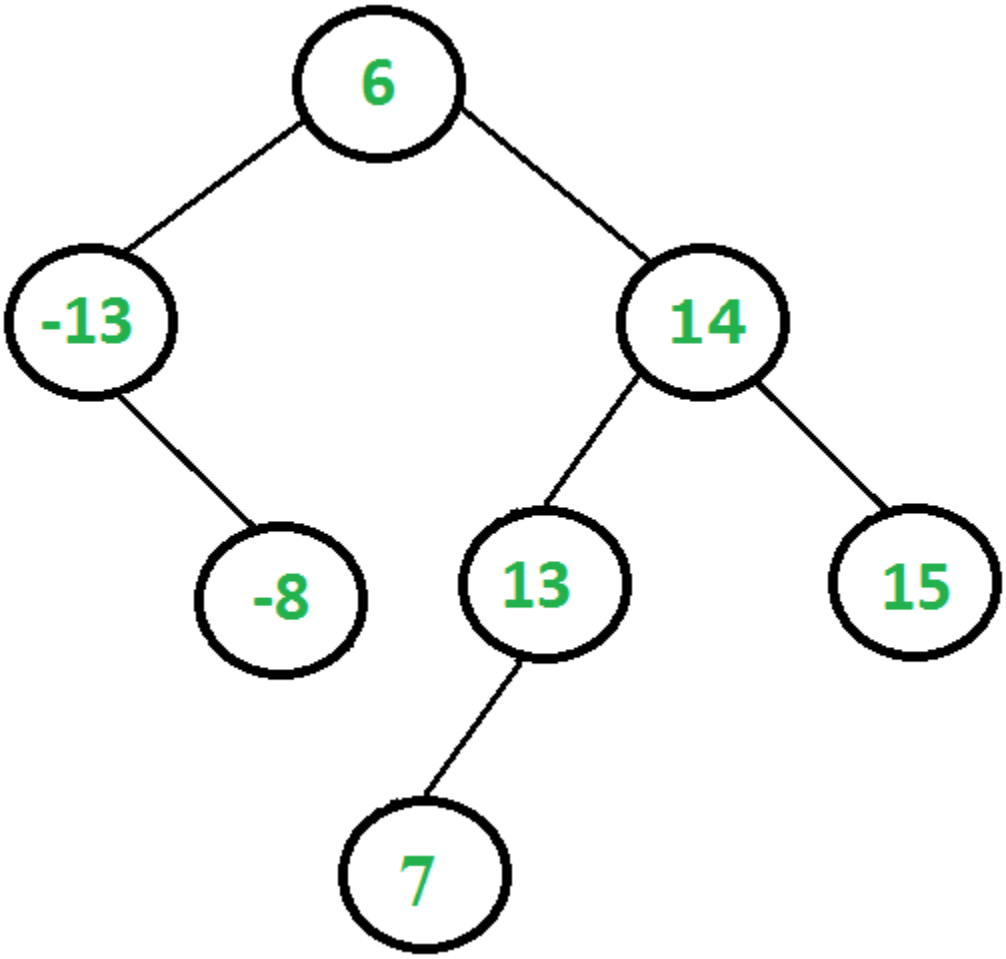
http://www.geeksforgeeks.org/add-greater-values-every-node-given-bst/

Category: Trees

Chapter 14

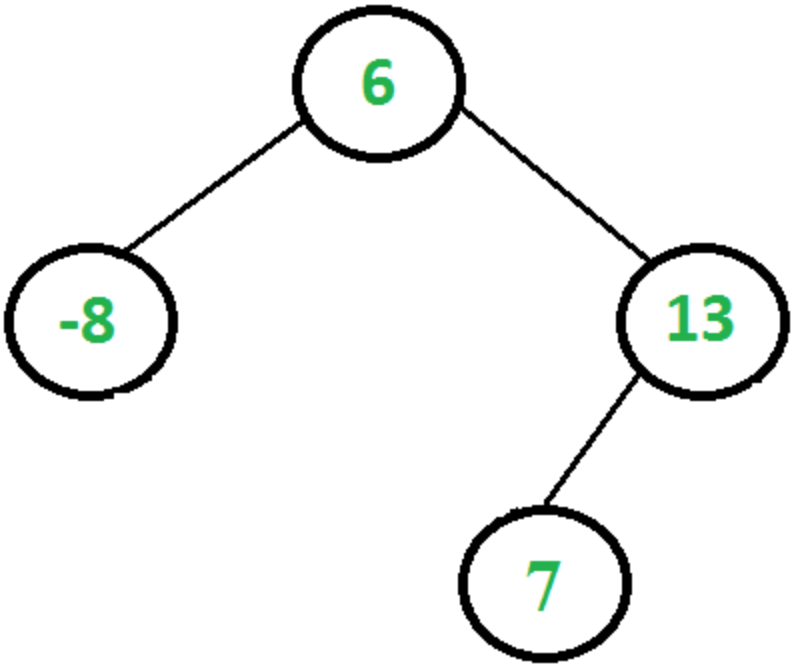
Remove BST keys outside the given range

Given a Binary Search Tree (BST) and a range [min, max], remove all keys which are outside the given range. The modified tree should also be BST. For example, consider the following BST and range [-10, 13].



The given tree should be changed to following. Note that all keys outside the range [-10, 13] are removed

and modified tree is BST.



There are two possible cases for every node.
1) Node's key is outside the given range. This case has two sub-cases.
.....**a)** Node's key is smaller than the min value.
.....**b)** Node's key is greater that the max value.
2) Node's key is in range.

We don't need to do anything for case 2. In case 1, we need to remove the node and change root of sub-tree rooted with this node.
The idea is to fix the tree in Postorder fashion. When we visit a node, we make sure that its left and right sub-trees are already fixed. In case 1.a), we simply remove root and return right sub-tree as new root. In case 1.b), we remove root and return left sub-tree as new root.

Following is C++ implementation of the above approach.

```
// A C++ program to remove BST keys outside the given range
#include<stdio.h>
#include <iostream>

using namespace std;

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// Resmoves all nodes having value outside the given range and returns the root
```

```
// of modified tree
node* removeOutsideRange(node *root, int min, int max)
{
   // Base Case
   if (root == NULL)
      return NULL;

   // First fix the left and right subtrees of root
   root->left =  removeOutsideRange(root->left, min, max);
   root->right =  removeOutsideRange(root->right, min, max);

   // Now fix the root.  There are 2 possible cases for toot
   // 1.a) Root's key is smaller than min value (root is not in range)
   if (root->key < min)
   {
       node *rChild = root->right;
       delete root;
       return rChild;
   }
   // 1.b) Root's key is greater than max value (root is not in range)
   if (root->key > max)
   {
       node *lChild = root->left;
       delete root;
       return lChild;
   }
   // 2. Root is in range
   return root;
}

// A utility function to create a new BST node with key as given num
node* newNode(int num)
{
    node* temp = new node;
    temp->key = num;
    temp->left = temp->right = NULL;
    return temp;
}

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

// Utility function to traverse the binary tree after conversion
void inorderTraversal(node* root)
{
```

```
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->key << " ";
        inorderTraversal( root->right );
    }
}

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);

    cout << "Inorder traversal of the given tree is: ";
    inorderTraversal(root);

    root = removeOutsideRange(root, -10, 13);

    cout << "\nInorder traversal of the modified tree is: ";
    inorderTraversal(root);

    return 0;
}
```

Output:

```
 Inorder traversal of the given tree is: -13 -8 6 7 13 14 15
Inorder traversal of the modified tree is: -8 6 7 13
```

Time Complexity: O(n) where n is the number of nodes in given BST.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/remove-bst-keys-outside-the-given-range/

Category: Trees

Chapter 15

Check if each internal node of a BST has exactly one child

Given Preorder traversal of a BST, check if each non-leaf node has only one child. Assume that the BST contains unique entries.

Examples

```
Input: pre[] = {20, 10, 11, 13, 12}
Output: Yes
The give array represents following BST. In the following BST, every internal
node has exactly 1 child. Therefor, the output is true.
        20
       /
      10
       \
        11
          \
           13
           /
          12
```

In Preorder traversal, descendants (or Preorder successors) of every node appear after the node. In the above example, 20 is the first node in preorder and all descendants of 20 appear after it. All descendants of 20 are smaller than it. For 10, all descendants are greater than it. In general, we can say, if all internal nodes have only one child in a BST, then all the descendants of every node are either smaller or larger than the node. The reason is simple, since the tree is BST and every node has only one child, all descendants of a node will either be on left side or right side, means all descendants will either be smaller or greater.

Approach 1 (Naive)
This approach simply follows the above idea that all values on right side are either smaller or larger. Use two loops, the outer loop picks an element one by one, starting from the leftmost element. The inner loop checks if all elements on the right side of the picked element are either smaller or greater. The time complexity of this method will be O(n^2).

Approach 2
Since all the descendants of a node must either be larger or smaller than the node. We can do following for

every node in a loop.
1. Find the next preorder successor (or descendant) of the node.
2. Find the last preorder successor (last element in pre[]) of the node.
3. If both successors are less than the current node, or both successors are greater than the current node, then continue. Else, return false.

```
#include <stdio.h>

bool hasOnlyOneChild(int pre[], int size)
{
    int nextDiff, lastDiff;

    for (int i=0; i<size-1; i++)
    {
        nextDiff = pre[i] - pre[i+1];
        lastDiff = pre[i] - pre[size-1];
        if (nextDiff*lastDiff < 0)
            return false;;
    }
    return true;
}

// driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre, size) == true )
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Output:

```
  Yes
```

Approach 3
1. Scan the last two nodes of preorder & mark them as min & max.
2. Scan every node down the preorder array. Each node must be either smaller than the min node or larger than the max node. Update min & max accordingly.

```
#include <stdio.h>

int hasOnlyOneChild(int pre[], int size)
{
    // Initialize min and max using last two elements
    int min, max;
    if (pre[size-1] > pre[size-2])
```

```
    {
        max = pre[size-1];
        min = pre[size-2]):
    else
    {
        max = pre[size-2];
        min = pre[size-1];
    }


    // Every element must be either smaller than min or
    // greater than max
    for (int i=size-3; i>=0; i--)
    {
        if (pre[i] < min)
            min = pre[i];
        else if (pre[i] > max)
            max = pre[i];
        else
            return false;
    }
    return true;
}

// Driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre,size))
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Output:

```
 Yes
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

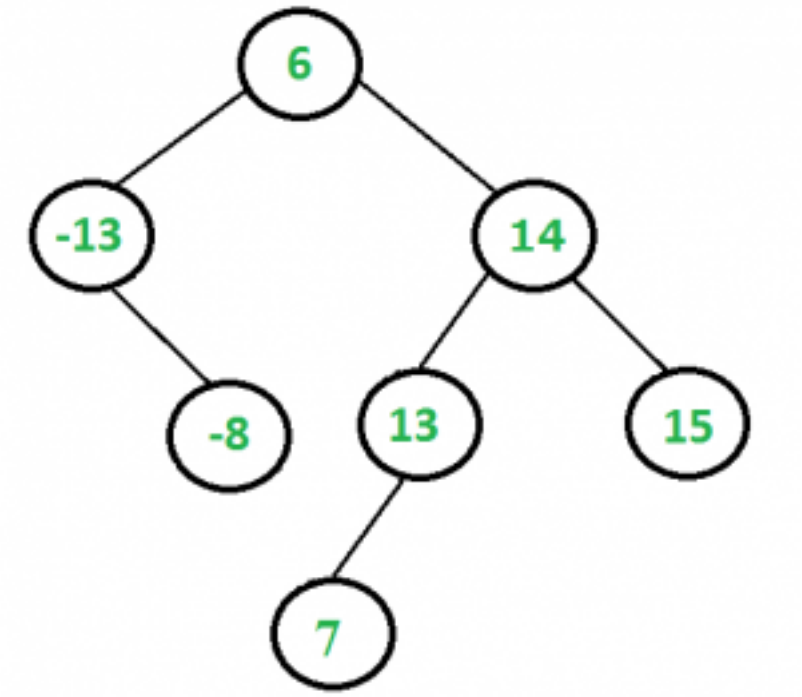
Source

http://www.geeksforgeeks.org/check-if-each-internal-node-of-a-bst-has-exactly-one-child/

Chapter 16

Three numbers in a BST that adds upto zero

Given a Balanced Binary Search Tree (BST), write a function isTripletPresent() that returns true if there is a triplet in given BST with sum equals to 0, otherwise returns false. Expected time complexity is O(n^2) and only O(Logn) extra space can be used. You can modify given Binary Search Tree. Note that height of a Balanced BST is always O(Logn)
For example, isTripletPresent() should return true for following BST because there is a triplet with sum 0, the triplet is {-13, 6, 7}.



The Brute Force Solution is to consider each triplet in BST and check whether the sum adds upto zero. The time complexity of this solution will be O(n^3).

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can use method 2 of thispost to find the triplet with sum equals to 0. This solution works in O(n^2) time, but requires O(n) auxiliary space.

Following is the solution that works in O(n^2) time and uses O(Logn) extra space:
1) Convert given BST to Doubly Linked List (DLL)
2) Now iterate through every node of DLL and if the key of node is negative, then find a pair in DLL with sum equal to key of current node multiplied by -1. To find the pair, we can use the approach used in hasArrayTwoCandidates() in method 1 of thispost.

```
// A C++ program to check if there is a triplet with sum equal to 0 in
// a given BST
#include<stdio.h>

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// A function to convert given BST to Doubly Linked List. left pointer is used
// as previous pointer and right pointer is used as next pointer. The function
// sets *head to point to first and *tail to point to last node of converted DLL
void convertBSTtoDLL(node* root, node** head, node** tail)
{
    // Base case
    if (root == NULL)
        return;

    // First convert the left subtree
    if (root->left)
        convertBSTtoDLL(root->left, head, tail);

    // Then change left of current root as last node of left subtree
    root->left = *tail;

    // If tail is not NULL, then set right of tail as root, else current
    // node is head
    if (*tail)
        (*tail)->right = root;
    else
        *head = root;

    // Update tail
    *tail = root;

    // Finally, convert right subtree
    if (root->right)
        convertBSTtoDLL(root->right, head, tail);
}

// This function returns true if there is pair in DLL with sum equal
// to given sum. The algorithm is similar to hasArrayTwoCandidates()
// in method 1 of http://tinyurl.com/dy6palr
bool isPresentInDLL(node* head, node* tail, int sum)
```

```
{
    while (head != tail)
    {
        int curr = head->key + tail->key;
        if (curr == sum)
            return true;
        else if (curr > sum)
            tail = tail->left;
        else
            head = head->right;
    }
    return false;
}

// The main function that returns true if there is a 0 sum triplet in
// BST otherwise returns false
bool isTripletPresent(node *root)
{
    // Check if the given  BST is empty
    if (root == NULL)
       return false;

    // Convert given BST to doubly linked list.  head and tail store the
    // pointers to first and last nodes in DLLL
    node* head = NULL;
    node* tail = NULL;
    convertBSTtoDLL(root, &head, &tail);

    // Now iterate through every node and find if there is a pair with sum
    // equal to -1 * heaf->key where head is current node
    while ((head->right != tail) && (head->key < 0))
    {
        // If there is a pair with sum equal to  -1*head->key, then return
        // true else move forward
        if (isPresentInDLL(head->right, tail, -1*head->key))
            return true;
        else
            head = head->right;
    }

    // If we reach here, then there was no 0 sum triplet
    return false;
}

// A utility function to create a new BST node with key as given num
node* newNode(int num)
{
    node* temp = new node;
    temp->key = num;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to insert a given key to BST
```

```
node* insert(node* root, int key)
{
    if (root == NULL)
       return newNode(key);
    if (root->key > key)
       root->left = insert(root->left, key);
    else
       root->right = insert(root->right, key);
    return root;
}

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);
    if (isTripletPresent(root))
        printf("Present");
    else
        printf("Not Present");
    return 0;
}
```

Output:

```
Present
```

Note that the above solution modifies given BST.

Time Complexity: Time taken to convert BST to DLL is O(n) and time taken to find triplet in DLL is O(n^2).

Auxiliary Space: The auxiliary space is needed only for function call stack in recursive function convertBSTtoDLL(). Since given tree is balanced (height is O(Logn)), the number of functions in call stack will never be more than O(Logn).

We can also find triplet in same time and extra space without modifying the tree. See nextpost. The code discussed there can be used to find triplet also.

This article is compiled by Ashish Anand and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-if-there-is-a-triplet-in-bst-that-adds-to-0/

Chapter 17

Merge two BSTs with limited extra space

Given two Binary Search Trees(BST), print the elements of both BSTs in sorted form. The expected time complexity is O(m+n) where m is the number of nodes in first tree and n is the number of nodes in second tree. Maximum allowed auxiliary space is O(height of the first tree + height of the second tree).

Examples:

```
First BST
       3
    /     \
   1       5
Second BST
    4
  /   \
2       6
Output: 1 2 3 4 5 6
```

```
First BST
          8
         / \
        2   10
       /
      1
Second BST
          5
         /
        3
       /
      0
Output: 0 1 2 3 5 8 10
```

Source: Google interview question

A similar question has been discussed earlier. Let us first discuss already discussed methods of the previous post which was for Balanced BSTs. The method 1 can be applied here also, but the time complexity will

be O(n^2) in worst case. The method 2 can also be applied here, but the extra space required will be O(n) which violates the constraint given in this question. Method 3 can be applied here but the step 3 of method 3 can't be done in O(n) for an unbalanced BST.

Thanks to Kumar for suggesting the following solution.

The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to print the elements in sorted form, whenever we get a smaller element from any of the trees, we print it. If the element is greater, then we push it back to stack for the next iteration.

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

// Structure of a BST Node
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//..................... START OF STACK RELATED STUFF...................
// A stack node
struct snode
{
    struct node  *t;
    struct snode *next;
};

// Function to add an elemt k to stack
void push(struct snode **s, struct node *k)
{
    struct snode *tmp = (struct snode *) malloc(sizeof(struct snode));

    //perform memory check here
    tmp->t = k;
    tmp->next = *s;
    (*s) = tmp;
}

// Function to pop an element t from stack
struct node *pop(struct snode **s)
{
    struct  node *t;
    struct snode *st;
    st=*s;
    (*s) = (*s)->next;
    t = st->t;
    free(st);
    return t;
}

// Fucntion to check whether the stack is empty or not
int isEmpty(struct snode *s)
```

```
{
    if (s == NULL )
        return 1;

    return 0;
}
//..................... END OF STACK RELATED STUFF...................


/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* A utility function to print Inoder traversal of a Binary Tree */
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// The function to print data of two BSTs in sorted order
void  merge(struct node *root1, struct node *root2)
{
    // s1 is stack to hold nodes of first BST
    struct snode *s1 = NULL;

    // Current node of first BST
    struct node  *current1 = root1;

    // s2 is stack to hold nodes of second BST
    struct snode *s2 = NULL;

    // Current node of second BST
    struct node  *current2 = root2;

    // If first BST is empty, then output is inorder
    // traversal of second BST
    if (root1 == NULL)
    {
        inorder(root2);
        return;
    }
    // If second BST is empty, then output is inorder
    // traversal of first BST
```

```
if (root2 == NULL)
{
    inorder(root1);
    return ;
}

// Run the loop while there are nodes not yet printed.
// The nodes may be in stack(explored, but not printed)
// or may be not yet explored
while (current1 != NULL || !isEmpty(s1) ||
      current2 != NULL || !isEmpty(s2))
{
    // Following steps follow iterative Inorder Traversal
    if (current1 != NULL || current2 != NULL )
    {
        // Reach the leftmost node of both BSTs and push ancestors of
        // leftmost nodes to stack s1 and s2 respectively
        if (current1 != NULL)
        {
            push(&s1, current1);
            current1 = current1->left;
        }
        if (current2 != NULL)
        {
            push(&s2, current2);
            current2 = current2->left;
        }

    }
    else
    {
        // If we reach a NULL node and either of the stacks is empty,
        // then one tree is exhausted, ptint the other tree
        if (isEmpty(s1))
        {
            while (!isEmpty(s2))
            {
                current2 = pop (&s2);
                current2->left = NULL;
                inorder(current2);
            }
            return ;
        }
        if (isEmpty(s2))
        {
            while (!isEmpty(s1))
            {
                current1 = pop (&s1);
                current1->left = NULL;
                inorder(current1);
            }
            return ;
        }
```

```
            // Pop an element from both stacks and compare the
            // popped elements
            current1 = pop(&s1);
            current2 = pop(&s2);

            // If element of first tree is smaller, then print it
            // and push the right subtree. If the element is larger,
            // then we push it back to the corresponding stack.
            if (current1->data < current2->data)
            {
                printf("%d ", current1->data);
                current1 = current1->right;
                push(&s2, current2);
                current2 = NULL;
            }
            else
            {
                printf("%d ", current2->data);
                current2 = current2->right;
                push(&s1, current1);
                current1 = NULL;
            }
        }
    }
}

/* Driver program to test above functions */
int main()
{
    struct node  *root1 = NULL, *root2 = NULL;

    /* Let us create the following tree as first tree
            3
          /  \
        1    5
     */
    root1 = newNode(3);
    root1->left = newNode(1);
    root1->right = newNode(5);

    /* Let us create the following tree as second tree
            4
          /  \
        2    6
     */
    root2 = newNode(4);
    root2->left = newNode(2);
    root2->right = newNode(6);

    // Print sorted nodes of both trees
    merge(root1, root2);

    return 0;
}
```

Time Complexity: O(m+n)
Auxiliary Space: O(height of the first tree + height of the second tree)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/merge-two-bsts-with-limited-extra-space/

Category: Trees

Chapter 18

Two nodes of a BST are swapped, correct the BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

```
Input Tree:
         10
        /  \
       5    8
      / \
     2   20

In the above tree, nodes 20 and 8 must be swapped to fix the tree.
Following is the output tree
         10
        /  \
       5    20
      / \
     2   8
```

The inorder traversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiilary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is O(nLogn) and auxiliary space needed is O(n).

We can solve this in O(n) time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.

```
For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5
```

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two node's values.

2. The swapped nodes are adjacent in the inorder traversal of BST.

```
For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 5 8 7 10 15 20 25
```

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve? *We will maintain three pointers, first, middle and last. When we find the first point where current node value is smaller than previous node value, we update the first with the previous node & middle with the current node. When we find the second point where current node value is smaller than previous node value, we update the last with the current node. In case #2, we will never find the second point. So, last pointer will not be updated. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.*

Following is C implementation of the given code.

```
// Two nodes in the BST's swapped, correct the BST.
#include <stdio.h>
#include <stdlib.h>

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

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* 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;
    return(node);
}

// This function does inorder traversal to find out the two swapped nodes.
```

```
// It sets three pointers, first, middle and last.  If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                     struct node** middle, struct node** last,
                     struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

// A function to fix a given BST where two nodes are swapped.  This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the poiters to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );
```

```
    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inoder traversal */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    /*   6
        /  \
       10    2
      / \   / \
     1   3 7  12
     10 and 2 are swapped
    */

    struct node *root = newNode(6);
    root->left        = newNode(10);
    root->right       = newNode(2);
    root->left->left  = newNode(1);
    root->left->right = newNode(3);
    root->right->right = newNode(12);
    root->right->left = newNode(7);

    printf("Inorder Traversal of the original tree \n");
    printInorder(root);

    correctBST(root);

    printf("\nInorder Traversal of the fixed tree \n");
    printInorder(root);

    return 0;
}
```

Output:

```
 Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12
```

Time Complexity: O(n)

See thisfor different test cases of the above code.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/fix-two-swapped-nodes-of-bst/

Category: Trees

Post navigation

← Output of C++ Program | Set 15 Private Destructor →

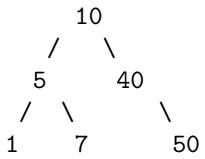
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 19

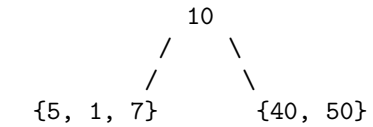
Construct BST from given preorder traversal | Set 1

Given preorder traversal of a binary search tree, construct the BST.

For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.



Method 1 (O(n^2) time complexity)
The first element of preorder traversal is always root. We first construct the root. Then we find the index of first element which is greater than root. Let the index be 'i'. The values between root and 'i' will be part of left subtree, and the values between 'i+1 and 'n-1 will be part of right subtree. Divide given pre[] at index "i" and recur for left and right sub-trees.
For example in {10, 5, 1, 7, 40, 50}, 10 is the first element, so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as following.



We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.

```
/* A O(n^2) program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
```

```
   and a pointer to right child */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// A recursive function to construct Full from pre[]. preIndex is used
// to keep track of index in pre[].
struct node* constructTreeUtil (int pre[], int* preIndex,
                                int low, int high, int size)
{
    // Base case
    if (*preIndex >= size || low > high)
        return NULL;

    // The first node in preorder traversal is root. So take the node at
    // preIndex from pre[] and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    *preIndex = *preIndex + 1;

    // If the current subarry has only one element, no need to recur
    if (low == high)
        return root;

    // Search for the first element greater than root
    int i;
    for ( i = low; i <= high; ++i )
        if ( pre[ i ] > root->data )
            break;

    // Use the index of element found in preorder to divide preorder array in
    // two parts. Left subtree and right subtree
    root->left = constructTreeUtil ( pre, preIndex, *preIndex, i - 1, size );
    root->right = constructTreeUtil ( pre, preIndex, i, high, size );

    return root;
}

// The main function to construct BST from given preorder traversal.
// This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int size)
```

```
{
    int preIndex = 0;
    return constructTreeUtil (pre, &preIndex, 0, size - 1, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

Output:

```
  1 5 7 10 40 50
```

Time Complexity: O(n^2)

Method 2 (O(n) time complexity)
The idea used here is inspired from method 3 of thispost. The trick is to set a range {min .. max} for every node. Initialize the range as {INT_MIN .. INT_MAX}. The first node will definitely be in range, so create root node. To construct the left subtree, set the range as {INT_MIN …root->data}. If a values is in the range {INT_MIN .. root->data}, the values is part part of left subtree. To construct the right subtree, set the range as {root->data..max .. INT_MAX}.

```
/* A O(n) program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

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

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

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// A recursive function to construct BST from pre[]. preIndex is used
// to keep track of index in pre[].
struct node* constructTreeUtil( int pre[], int* preIndex, int key,
                                int min, int max, int size )
{
    // Base case
    if( *preIndex >= size )
        return NULL;

    struct node* root = NULL;

    // If current element of pre[] is in range, then
    // only it is part of current subtree
    if( key > min && key < max )
    {
        // Allocate memory for root of this subtree and increment *preIndex
        root = newNode ( key );
        *preIndex = *preIndex + 1;

        if (*preIndex < size)
        {
            // Contruct the subtree under root
            // All nodes which are in range {min .. key} will go in left
            // subtree, and first such node will be root of left subtree.
            root->left = constructTreeUtil( pre, preIndex, pre[*preIndex],
                                        min, key, size );

            // All nodes which are in range {key..max} will go in right
            // subtree, and first such node will be root of right subtree.
            root->right = constructTreeUtil( pre, preIndex, pre[*preIndex],
                                         key, max, size );
        }
    }

    return root;
}

// The main function to construct BST from given preorder traversal.
// This function mainly uses constructTreeUtil()
```

```
struct node *constructTree (int pre[], int size)
{
    int preIndex = 0;
    return constructTreeUtil ( pre, &preIndex, pre[0], INT_MIN, INT_MAX, size );
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

Output:

```
  1 5 7 10 40 50
```

Time Complexity: O(n)

We will soon publish a O(n) iterative solution as a separate post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversa/

Category: Trees

Post navigation

← Shuffle a given array Construct BST from given preorder traversal | Set 2 →

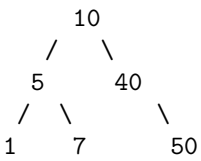
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 20

Construct BST from given preorder traversal | Set 2

Given preorder traversal of a binary search tree, construct the BST.

For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.



We have discussed O(n^2) and O(n) recursive solutions in the previous post. Following is a stack based iterative solution that works in O(n) time.

1. Create an empty stack.

2. Make the first value as root. Push it to the stack.

3. Keep on popping while the stack is not empty and the next value is greater than stack's top value. Make this value as the right child of the last popped node. Push the new node to the stack.

4. If the next value is less than the stack's top value, make this value as the left child of the stack's top node. Push the new node to the stack.

5. Repeat steps 2 and 3 until there are items remaining in pre[].

```
/* A O(n) iterative program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
typedef struct Node
{
    int data;
```

```
    struct Node *left, *right;
} Node;

// A Stack has array of Nodes, capacity, and top
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;

// A utility function to create a new tree node
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a stack of given capacity
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
    return stack;
}

// A utility function to check if stack is full
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
{
    return stack->top == -1;
}

// A utility function to push an item to stack
void push( Stack* stack, Node* item )
{
    if( isFull( stack ) )
        return;
    stack->array[ ++stack->top ] = item;
}

// A utility function to remove an item from stack
Node* pop( Stack* stack )
{
    if( isEmpty( stack ) )
```

```
        return NULL;
    return stack->array[ stack->top-- ];
}

// A utility function to get top node of stack
Node* peek( Stack* stack )
{
    return stack->array[ stack->top ];
}

// The main function that constructs BST from pre[]
Node* constructTree ( int pre[], int size )
{
    // Create a stack of capacity equal to size
    Stack* stack = createStack( size );

    // The first element of pre[] is always root
    Node* root = newNode( pre[0] );

    // Push root
    push( stack, root );

    int i;
    Node* temp;

    // Iterate through rest of the size-1 items of given preorder array
    for ( i = 1; i < size; ++i )
    {
        temp = NULL;

        /* Keep on popping while the next value is greater than
           stack's top value. */
        while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
            temp = pop( stack );

        // Make this greater value as the right child and push it to the stack
        if ( temp != NULL)
        {
            temp->right = newNode( pre[i] );
            push( stack, temp->right );
        }

        // If the next value is less than the stack's top value, make this value
        // as the left child of the stack's top node. Push the new node to stack
        else
        {
            peek( stack )->left = newNode( pre[i] );
            push( stack, peek( stack )->left );
        }
    }

    return root;
}
```

```
// A utility function to print inorder traversal of a Binary Tree
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    Node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

Output:

```
  1 5 7 10 40 50
```

Time Complexity: O(n). The complexity looks more from first look. If we take a closer look, we can observe that every item is pushed and popped only once. So at most 2n push/pop operations are performed in the main loops of constructTree(). Therefore, time complexity is O(n).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversal-set-2/

Category: Trees

Chapter 21

Floor and Ceil from a BST

There are numerous applications we need to find floor (ceil) value of a key in a binary search tree or sorted array. For example, consider designing memory management system in which free nodes are arranged in BST. Find best fit for the input request.

Ceil Value Node: Node with smallest data larger than or equal to key value.

Imagine we are moving down the tree, and assume we are root node. The comparison yields three possibilities,

A) Root data is equal to key. We are done, root data is ceil value.

B) Root data $<$ key value, certainly the ceil value can't be in left subtree. Proceed to search on right subtree as reduced problem instance.

C) Root data $>$ key value, the ceil value *may be* in left subtree. We may find a node with is larger data than key value in left subtree, if not the root itself will be ceil node.

Here is code in C for ceil value.

```
// Program to find ceil of a given value in BST
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

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

```
// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(node *root, int input)
{
    // Base case
    if( root == NULL )
        return -1;

    // We found equal key
    if( root->key == input )
        return root->key;

    // If root's key is smaller, ceil must be in right subtree
    if( root->key < input )
        return Ceil(root->right, input);

    // Else, either left subtree or root has the ceil value
    int ceil = Ceil(root->left, input);
    return (ceil >= input) ? ceil : root->key;
}

// Driver program to test above function
int main()
{
    node *root = newNode(8);

    root->left = newNode(4);
    root->right = newNode(12);

    root->left->left = newNode(2);
    root->left->right = newNode(6);

    root->right->left = newNode(10);
    root->right->right = newNode(14);

    for(int i = 0; i < 16; i++)
        printf("%d  %d\n", i, Ceil(root, i));

    return 0;
}
```

Output:

```
 0  2
1  2
2  2
3  4
4  4
5  6
6  6
7  8
8  8
```

```
9  10
10  10
11  12
12  12
13  14
14  14
15  -1
```

Exercise:

1. Modify above code to find floor value of input key in a binary search tree.

2. Write neat algorithm to find floor and ceil values in a sorted array. Ensure to handle all possible boundary conditions.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/floor-and-ceil-from-a-bst/

Category: Trees

Chapter 22

Convert a BST to a Binary Tree such that sum of all greater keys is added to every ke

Given a Binary Search Tree (BST), convert it to a Binary Tree such that every key of the original BST is changed to key plus sum of all greater keys in BST.

Examples:

```
Input: Root of following BST
              5
            /   \
           2     13

Output: The given BST is converted to following Binary Tree
              18
            /   \
          20     13
```

Source: Convert a BST

Solution: Do reverse Inoorder traversal. Keep track of the sum of nodes visited so far. Let this sum be *sum*. For every node currently being visited, first add the key of this node to *sum*, i.e. *sum = sum + node->key*. Then change the key of current node to *sum*, i.e., *node->key = sum*.
When a BST is being traversed in reverse Inorder, for every key currently being visited, all keys that are already visited are all greater keys.

```
// Program to change a BST to Binary Tree such that key of a node becomes
// original key plus sum of all greater keys in BST
#include <stdio.h>
#include <stdlib.h>

/* A BST node has key, left child and right child */
struct node
```

```
{
    int key;
    struct node* left;
    struct node* right;
};

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

// A recursive function that traverses the given BST in reverse inorder and
// for every key, adds all greater keys to it
void addGreaterUtil(struct node *root, int *sum_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    // Recur for right subtree first so that sum of all greater
    // nodes is stored at sum_ptr
    addGreaterUtil(root->right, sum_ptr);

    // Update the value at sum_ptr
    *sum_ptr = *sum_ptr + root->key;

    // Update key of this node
    root->key = *sum_ptr;

    // Recur for left subtree so that the updated sum is added
    // to smaller nodes
    addGreaterUtil(root->left, sum_ptr);
}

// A wrapper over addGreaterUtil().  It initializes sum and calls
// addGreaterUtil() to recursivel upodate and use value of sum
void addGreater(struct node *root)
{
    int sum = 0;
    addGreaterUtil(root, &sum);
}

// A utility function to print inorder traversal of Binary Tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
```

```
    printf("%d ", node->key);
    printInorder(node->right);
}

// Driver program to test above function
int main()
{
    /* Create following BST
              5
            /   \
           2     13  */
    node *root = newNode(5);
    root->left = newNode(2);
    root->right = newNode(13);

    printf(" Inorder traversal of the given tree\n");
    printInorder(root);

    addGreater(root);

    printf("\n Inorder traversal of the modified tree\n");
    printInorder(root);

    return 0;
}
```

Output:

```
  Inorder traversal of the given tree
2 5 13
 Inorder traversal of the modified tree
20 18 13
```

Time Complexity: O(n) where n is the number of nodes in given Binary Search Tree.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/convert-bst-to-a-binary-tree/

Category: Trees Tags: Amazon, BST

Chapter 23

Sorted Linked List to Balanced BST

Given a Singly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Linked List.

Examples:

```
Input:  Linked List 1->2->3
Output: A Balanced BST
     2
   /  \
  1    3


Input: Linked List 1->2->3->4->5->6->7
Output: A Balanced BST
        4
      /   \
     2     6
   /  \   / \
  1   3  4   7

Input: Linked List 1->2->3->4
Output: A Balanced BST
      3
    /  \
   2    4
 /
1

Input:  Linked List 1->2->3->4->5->6
Output: A Balanced BST
      4
    /   \
   2     6
 /  \   /
1   3  5
```

Method 1 (Simple)
Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
       a) Get the middle of left half and make it left child of the root
          created in step 1.
       b) Get the middle of right half and make it right child of the
          root created in step 1.
```

Time complexity: O(nLogn) where n is the number of nodes in Linked List.

See thisforum thread for more details.

Method 2 (Tricky)
The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as the appear in Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we allocate memory for root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.
While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.
Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

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

/* Link list node */
struct LNode
{
    int data;
    struct LNode* next;
};

/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);
int countLNodes(struct LNode *head);
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n);


/* This function counts the number of nodes in Linked List and then calls
```

```
   sortedListToBSTRecur() to construct BST */
struct TNode* sortedListToBST(struct LNode *head)
{
    /*Count the number of nodes in Linked List */
    int n = countLNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of linked list
       n  --> No. of nodes in Linked List */
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct TNode *left = sortedListToBSTRecur(head_ref, n/2);

    /* Allocate memory for root, and link the above constructed left
       subtree with root */
    struct TNode *root = newNode((*head_ref)->data);
    root->left = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) which is n-n/2-1*/
    root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}



/* UTILITY FUNCTIONS */

/* A utility function that returns count of nodes in a given Linked List */
int countLNodes(struct LNode *head)
{
    int count = 0;
    struct LNode *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}
```

```
/* Function to insert a node at the beginging of the linked list */
void push(struct LNode** head_ref, int new_data)
{
    /* allocate node */
    struct LNode* new_node =
        (struct LNode*) malloc(sizeof(struct LNode));
    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct LNode *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

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

    return node;
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
```

```
    struct LNode* head = NULL;

    /* Let us create a sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6->7 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given Linked List ");
    printList(head);

    /* Convert List to BST */
    struct TNode *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sorted-linked-list-to-balanced-bst/

Category: Linked Lists

Chapter 24

In-place conversion of Sorted DLL to Balanced BST

Given a Doubly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Doubly Linked List. The tree must be constructed in-place (No new node should be allocated for tree conversion)

Examples:

```
Input:  Doubly Linked List 1  2  3
Output: A Balanced BST
     2
   /  \
  1    3


Input: Doubly Linked List 1  2 3  4 5  6  7
Output: A Balanced BST
        4
      /   \
     2     6
   /  \   / \
  1   3  4   7

Input: Doubly Linked List 1  2  3  4
Output: A Balanced BST
      3
    /  \
   2    4
 /
1

Input:  Doubly Linked List 1  2  3  4  5  6
Output: A Balanced BST
      4
    /   \
   2     6
 /  \   /
```

```
1   3  5
```

The Doubly Linked List conversion is very much similar to this Singly Linked List problem and the method 1 is exactly same as the method 1 ofprevious post. Method 2 is also almost same. The only difference in method 2 is, instead of allocating new nodes for BST, we reuse same DLL nodes. We use prev pointer as left and next pointer as right.

Method 1 (Simple)
Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
       a) Get the middle of left half and make it left child of the root
          created in step 1.
       b) Get the middle of right half and make it right child of the
          root created in step 1.
```

Time complexity: O(nLogn) where n is the number of nodes in Linked List.

Method 2 (Tricky)
The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as the appear in Doubly Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.
While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.
Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

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

/* A Doubly Linked List node that will also be used as a tree node */
struct Node
{
    int data;

    // For tree, next pointer can be used as right subtree pointer
    struct Node* next;

    // For tree, prev pointer can be used as left subtree pointer
    struct Node* prev;
};


// A utility function to count nodes in a Linked List
int countNodes(struct Node *head);
```

```
struct Node* sortedListToBSTRecur(struct Node **head_ref, int n);

/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct Node* sortedListToBST(struct Node *head)
{
    /*Count the number of nodes in Linked List */
    int n = countNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of Doubly linked list
       n  --> No. of nodes in the Doubly Linked List */
struct Node* sortedListToBSTRecur(struct Node **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct Node *left = sortedListToBSTRecur(head_ref, n/2);

    /* head_ref now refers to middle node, make middle node as root of BST*/
    struct Node *root = *head_ref;

    // Set pointer to left subtree
    root->prev = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) */
    root->next = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a given Linked List */
int countNodes(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
```

```
    return count;
}

/* Function to insert a node at the beginging of the Doubly Linked List */
void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
            (struct Node*) malloc(sizeof(struct Node));

    /* put in the data  */
    new_node->data  = new_data;

    /* since we are adding at the begining,
      prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) !=  NULL)
      (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct Node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->prev);
    preOrder(node->next);
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
```

```
    /* Let us create a sorted linked list to test the functions
     Created linked list will be 7->6->5->4->3->2->1 */
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given Linked List ");
    printList(head);

    /* Convert List to BST */
    struct Node *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

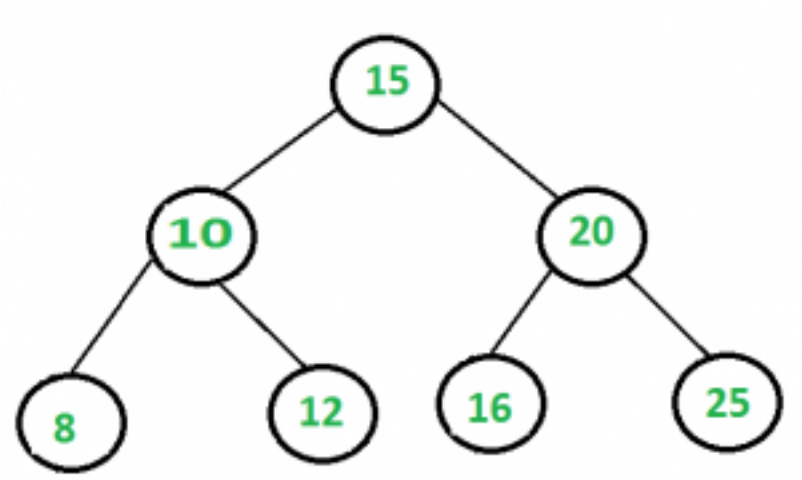
http://www.geeksforgeeks.org/in-place-conversion-of-sorted-dll-to-balanced-bst/

Category: Linked Lists

Chapter 25

Find a pair with given sum in a Balanced BST

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is O(n) and only O(Logn) extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always O(Logn).



This problem is mainly extension of the previous post. Here we are not allowed to modify the BST.

The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X. The time complexity of this solution will be O(n^2).

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in O(n) time (See this for details). This solution works in O(n) time, but requires O(n) auxiliary space.

A **space optimized solution** is discussed in previous post. The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in O(n) time. This solution takes O(n) time and O(Logn) extra space, but it modifies the given BST.

The **solution discussed below takes O(n) time, O(Logn) space and doesn't modify BST**. The idea is same as finding the pair in sorted array (See method 1 of this for details). We traverse BST in Normal

Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node. In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum. If the sum is same as target sum, we return true. If the sum is more than target sum, we move to next node in reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false. Following is C++ implementation of this approach.

```
/* In a balanced binary search tree isPairPresent two element which sums to
   a given value time O(n) space O(logn) */
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// A BST node
struct node
{
    int val;
    struct node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct node* *array;
};

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct node**) malloc(stack->size * sizeof(struct node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{   return stack->top - 1 == stack->size;  }

int isEmpty(struct Stack* stack)
{   return stack->top == -1;   }

void push(struct Stack* stack, struct node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}
```

```
struct node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// Returns true if a pair with target sum exists in BST, otherwise false
bool isPairPresent(struct node *root, int target)
{
    // Create two stacks. s1 is used for normal inorder traversal
    // and s2 is used for reverse inorder traversal
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE, we can find the tree size and
    // fix stack size as O(Logn) for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

    // done1, val1 and curr1 are used for normal inorder traversal using s1
    // done2, val2 and curr2 are used for reverse inorder traversal using s2
    bool done1 = false, done2 = false;
    int val1 = 0, val2 = 0;
    struct node *curr1 = root, *curr2 = root;

    // The loop will break when we either find a pair or one of the two
    // traversals is complete
    while (1)
    {
        // Find next node in normal Inorder traversal. See following post
        // http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
        while (done1 == false)
        {
            if (curr1 != NULL)
            {
                push(s1, curr1);
                curr1 = curr1->left;
            }
            else
            {
                if (isEmpty(s1))
                    done1 = 1;
                else
                {
                    curr1 = pop(s1);
                    val1 = curr1->val;
                    curr1 = curr1->right;
                    done1 = 1;
                }
            }
        }

        // Find next node in REVERSE Inorder traversal. The only
```

```
        // difference between above and below loop is, in below loop
        // right subtree is traversed before left subtree
        while (done2 == false)
        {
            if (curr2 != NULL)
            {
                push(s2, curr2);
                curr2 = curr2->right;
            }
            else
            {
                if (isEmpty(s2))
                    done2 = 1;
                else
                {
                    curr2 = pop(s2);
                    val2 = curr2->val;
                    curr2 = curr2->left;
                    done2 = 1;
                }
            }
        }

        // If we find a pair, then print the pair and return. The first
        // condition makes sure that two same values are not added
        if ((val1 != val2) && (val1 + val2) == target)
        {
            printf("\n Pair Found: %d + %d = %d\n", val1, val2, target);
            return true;
        }

        // If sum of current values is smaller, then move to next node in
        // normal inorder traversal
        else if ((val1 + val2) < target)
            done1 = false;

        // If sum of current values is greater, then move to next node in
        // reverse inorder traversal
        else if ((val1 + val2) > target)
            done2 = false;

        // If any of the inorder traversals is over, then there is no pair
        // so return false
        if (val1 >= val2)
            return false;
    }
}

// A utility function to create BST node
struct node * NewNode(int val)
{
    struct node *tmp = (struct node *)malloc(sizeof(struct node));
    tmp->val = val;
    tmp->right = tmp->left =NULL;
```

```
    return tmp;
}

// Driver program to test above functions
int main()
{
    /*
                    15
                  /    \
                10      20
               / \     /  \
              8  12   16  25    */
    struct node *root =  NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        printf("\n No such values are found\n");

    getchar();
    return 0;
}
```

Output:

```
  Pair Found: 8 + 25 = 33
```

This article is compiled by Kumarand reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-a-pair-with-given-sum-in-bst/

Category: Trees

Post navigation

← Find if there is a triplet in a Balanced BST that adds to zero Reverse Level Order Traversal →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 26

Total number of possible Binary Search Trees with n keys

Total number of possible Binary Search Trees with n different keys = Catalan number Cn = (2n)!/(n+1)!*n!

See references for proof and examples.

References:
http://en.wikipedia.org/wiki/Catalan_number

Source

http://www.geeksforgeeks.org/g-fact-18/

Category: Trees Tags: GFacts

Chapter 27

Merge Two Balanced Binary Search Trees

You are given two balanced binary search trees e.g., AVL or Red Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take O(m+n) time.

In the following solutions, it is assumed that sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree (See this).

Method 1 (Insert elements of first tree to second)
Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes Logn time (See this) where n is size of the BST. So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1). The value of this expression will be between mLogn and mLog(m+n-1). As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals)
1) Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.
2) Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.
3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.
4) Construct a balanced tree from the merged array using the technique discussed in this post. This step takes O(m+n) time.

Time complexity of this method is O(m+n) which is better than method 1. This method takes O(m+n) time even if the input BSTs are not balanced.
Following is C++ implementation of this method.

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

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

```
    struct node* right;
};

// A utility unction to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n);

// A helper function that stores inorder traversal of a tree in inorder array
void storeInorder(struct node* node, int inorder[], int *index_ptr);

/* A function that constructs Balanced Binary Search Tree from a sorted array
   See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end);

/* This function merges two balanced BSTs with roots as root1 and root2.
   m and n are the sizes of the trees respectively */
struct node* mergeTrees(struct node *root1, struct node *root2, int m, int n)
{
    // Store inorder traversal of first tree in an array arr1[]
    int *arr1 = new int[m];
    int i = 0;
    storeInorder(root1, arr1, &i);

    // Store inorder traversal of second tree in another array arr2[]
    int *arr2 = new int[n];
    int j = 0;
    storeInorder(root2, arr2, &j);

    // Merge the two sorted array into one
    int *mergedArr = merge(arr1, arr2, m, n);

    // Construct a tree from the merged array and return root of the tree
    return sortedArrayToBST (mergedArr, 0, m+n-1);
}

/* 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;

    return(node);
}

// A utility function to print inorder traversal of a given binary tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
```

```
    printInorder(node->left);

    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

// A utility unction to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n)
{
    // mergedArr[] is going to contain result
    int *mergedArr = new int[m + n];
    int i = 0, j = 0, k = 0;

    // Traverse through both arrays
    while (i < m && j < n)
    {
        // Pick the smaler element and put it in mergedArr
        if (arr1[i] < arr2[j])
        {
            mergedArr[k] = arr1[i];
            i++;
        }
        else
        {
            mergedArr[k] = arr2[j];
            j++;
        }
        k++;
    }

    // If there are more elements in first array
    while (i < m)
    {
        mergedArr[k] = arr1[i];
        i++; k++;
    }

    // If there are more elements in second array
    while (j < n)
    {
        mergedArr[k] = arr2[j];
        j++; k++;
    }

    return mergedArr;
}

// A helper function that stores inorder traversal of a tree rooted with node
void storeInorder(struct node* node, int inorder[], int *index_ptr)
{
    if (node == NULL)
        return;
```

```
    /* first recur on left child */
    storeInorder(node->left, inorder, index_ptr);

    inorder[*index_ptr] = node->data;
    (*index_ptr)++;  // increase index for next entry

    /* now recur on right child */
    storeInorder(node->right, inorder, index_ptr);
}

/* A function that constructs Balanced Binary Search Tree from a sorted array
   See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
      return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct node *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */
    root->left =  sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}

/* Driver program to test above functions*/
int main()
{
    /* Create following tree as first balanced BST
           100
         /  \
        50    300
       / \
      20  70
    */
    struct node *root1  = newNode(100);
    root1->left         = newNode(50);
    root1->right        = newNode(300);
    root1->left->left   = newNode(20);
    root1->left->right  = newNode(70);

    /* Create following tree as second balanced BST
            80
           /  \
         40   120
```

```
    */
    struct node *root2  = newNode(80);
    root2->left         = newNode(40);
    root2->right        = newNode(120);

    struct node *mergedTree = mergeTrees(root1, root2, 5, 3);

    printf ("Following is Inorder traversal of the merged tree \n");
    printInorder(mergedTree);

    getchar();
    return 0;
}
```

Output:

```
Following is Inorder traversal of the merged tree
20 40 50 70 80 100 120 300
```

Method 3 (In-Place Merge using DLL)
We can use a Doubly Linked List to merge trees in place. Following are the steps.

1) Convert the given two Binary Search Trees into doubly linked list in place (Refer this post for this step).
2) Merge the two sorted Linked Lists (Refer this post for this step).
3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer this post for this step)

Time complexity of this method is also O(m+n) and this method does conversion in place.

Thanks to Dheeraj and Ronzii for suggesting this method.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/merge-two-balanced-binary-search-trees/

Category: Trees

Post navigation

← Greedy Algorithms | Set 1 (Activity Selection Problem) Union and Intersection of two Linked Lists →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 28

Binary Tree to Binary Search Tree Conversion

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

Examples.

```
Example 1
Input:
          10
         /  \
        2    7
       / \
      8   4
Output:
          8
         /  \
        4    10
       / \
      2   7


Example 2
Input:
          10
         /  \
        30   15
       /      \
      20       5
Output:
          15
         /  \
       10    20
       /      \
      5        30
```

Solution
Following is a 3 step solution for converting Binary tree to Binary Search Tree.
1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes O(n) time.
2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in O(nLogn) time using Heap Sort or Merge Sort.
3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes O(n) time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

```
/* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* A helper function that stores inorder traversal of a tree rooted
  with node */
void storeInorder (struct node* node, int inorder[], int *index_ptr)
{
    // Base Case
    if (node == NULL)
        return;

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++;  // increase index for next entry

    /* finally store the right subtree */
    storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
{
    if (root == NULL)
     return 0;
    return countNodes (root->left) +
           countNodes (root->right) + 1;
}

// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
```

```
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
   This functon basically does Inorder traversal of Binary Tree and
   one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
      return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}

// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if(root == NULL)
       return;

    /* Count the number of nodes in Binary Tree so that
       we know the size of temporary array to be created */
    int n = countNodes (root);

    // Create a temp array arr[] and store inorder traversal of tree in arr[]
    int *arr = new int[n];
    int i = 0;
    storeInorder (root, arr, &i);

    // Sort the array using library function for quick sort
    qsort (arr, n, sizeof(arr[0]), compare);

    // Copy array elements back to Binary Tree
    i = 0;
    arrayToBST (arr, root, &i);

    // delete dynamically allocated memory to avoid meory leak
    delete [] arr;
}

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
```

```
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* Utility function to print inorder traversal of Binary Tree */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure
          10
         /  \
        30   15
       /      \
      20       5   */
    root = newNode(10);
    root->left = newNode(30);
    root->right = newNode(15);
    root->left->left = newNode(20);
    root->right->right = newNode(5);

    // convert Binary Tree to BST
    binaryTreeToBST (root);

    printf("Following is Inorder Traversal of the converted BST: \n");
    printInorder (root);

    return 0;
}
```

Output:

```
Following is Inorder Traversal of the converted BST:
```

```
5 10 15 20 30
```

We will be covering another method for this problem which converts the tree using O(height of tree) extra space.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

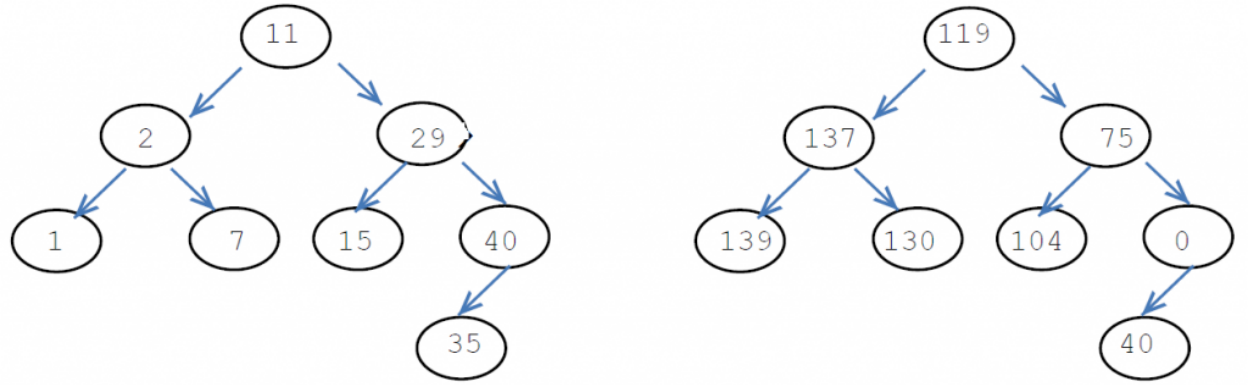
http://www.geeksforgeeks.org/binary-tree-to-binary-search-tree-conversion/

Category: Trees

Chapter 29

Transform a BST to greater sum tree

Given a BST, transform it into greater sum tree where each node contains sum of all nodes greater than that node.



We strongly recommend to minimize the gbrowser and try this yourself first.

Method 1 (Naive):
This method doesn't require the tree to be a BST. Following are steps.
1. Traverse node by node(Inorder, preorder, etc.)
2. For each node find all the nodes greater than that of the current node, sum the values. Store all these sums.
3. Replace each node value with their corresponding sum by traversing in the same order as in Step 1.
This takes O(n^2) Time Complexity.

Method 2 (Using only one traversal)
By leveraging the fact that the tree is a BST, we can find an O(n) solution. The idea is to traverse BST in reverse inorder. Reverse inorder traversal of a BST gives us keys in decreasing order. Before visiting a node, we visit all greater nodes of that node. While traversing we keep track of sum of keys which is the sum of all the keys greater than the key of current node.

```
// C++ program to transform a BST to sum tree
#include<iostream>
using namespace std;

// A BST node
struct Node
```

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

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp =  new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to transform a BST to sum tree.
// This function traverses the tree in reverse inorder so
// that we have visited all greater key nodes of the currently
// visited node
void transformTreeUtil(struct Node *root, int *sum)
{
   // Base case
   if (root == NULL)  return;

   // Recur for right subtree
   transformTreeUtil(root->right, sum);

   // Update sum
   *sum = *sum + root->data;

   // Store old sum in current node
   root->data = *sum - root->data;

   // Recur for left subtree
   transformTreeUtil(root->left, sum);
}

// A wrapper over transformTreeUtil()
void transformTree(struct Node *root)
{
    int sum = 0; // Initialize sum
    transformTreeUtil(root, &sum);
}

// A utility function to print indorder traversal of a
// binary tree
void printInorder(struct Node *root)
{
    if (root == NULL) return;

    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}
```

```
// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(11);
    root->left = newNode(2);
    root->right = newNode(29);
    root->left->left = newNode(1);
    root->left->right = newNode(7);
    root->right->left = newNode(15);
    root->right->right = newNode(40);
    root->right->right->left = newNode(35);

    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    transformTree(root);

    cout << "\n\nInorder Traversal of transformed tree\n";
    printInorder(root);

    return 0;
}
```

Output:

```
 Inorder Traversal of given tree
1 2 7 11 15 29 35 40

Inorder Traversal of transformed tree
139 137 130 119 104 75 40 0
```

Time complexity of this method is O(n) as it does a simple traversal of tree.

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

Source

http://www.geeksforgeeks.org/transform-bst-sum-tree/

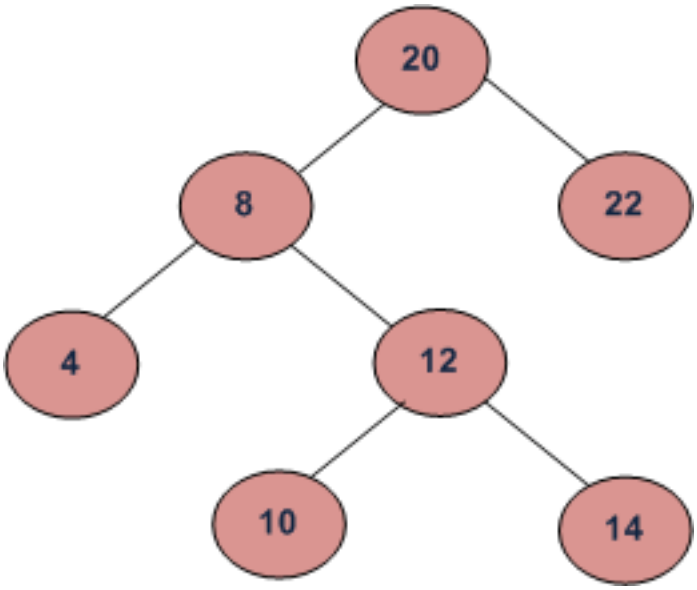
Category: Trees

Chapter 30

K'th Largest Element in BST when modification to BST is not allowed

Given a Binary Search Tree (BST) and a positive integer k, find the k'th largest element in the Binary Search Tree.

For example, in the following BST, if k = 3, then output should be 14, and if k = 5, then output should be 10.



We have discussed two methods in thispost. The method 1 requires O(n) time. The method 2 takes O(h) time where h is height of BST, but requires augmenting the BST (storing count of nodes in left subtree with every node).

Can we find k'th largest element in better than O(n) time and no augmentation?

We strongly recommend to minimize your browser and try this yourself first.

In this post, a method is discussed that takes O(h + k) time. This method doesn't require any change to BST.

The idea is to do reverse inorder traversal of BST. The reverse inorder traversal traverses all nodes in decreasing order. While doing the traversal, we keep track of count of nodes visited so far. When the count becomes equal to k, we stop the traversal and print the key.

```
// C++ program to find k'th largest element in BST
#include<iostream>
using namespace std;

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

// 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 function to find k'th largest element in a given tree.
void kthLargestUtil(Node *root, int k, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    kthLargestUtil(root->right, k, c);

    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the k'th largest
    if (c == k)
    {
        cout << "K'th largest element is "
             << root->key << endl;
        return;
    }

    // Recur for left subtree
    kthLargestUtil(root->left, k, c);
}

// Function to find k'th largest element
void kthLargest(Node *root, int k)
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
```

```
    kthLargestUtil(root, k, c);
}

/* A utility function to insert a new node with given key in BST */
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 (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    int c = 0;
    for (int k=1; k<=7; k++)
        kthLargest(root, k);

    return 0;
}


K'th largest element is 80
K'th largest element is 70
K'th largest element is 60
K'th largest element is 50
K'th largest element is 40
K'th largest element is 30
K'th largest element is 20
```

Time complexity: The code first traverses down to the rightmost node which takes O(h) time, then traverses

k elements in O(k) time. Therefore overall time complexity is O(h + k).

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

Source

http://www.geeksforgeeks.org/kth-largest-element-in-bst-when-modification-to-bst-is-not-allowed/

Category: Trees

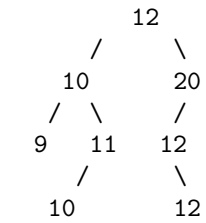
Chapter 31

How to handle duplicates in Binary Search Tree?

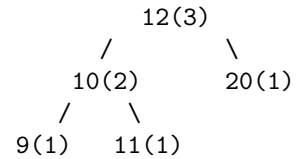
In a Binary Search Tree (BST), all keys in left subtree of a key must be smaller and all keys in right subtree must be greater. So a Binary Search Treeby definition has distinct keys.

How to allow duplicates where every insertion inserts one more key with a value and every deletion deletes one occurrence?

A **Simple Solution** is to allow same keys on right side (we could also choose left side). For example consider insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree



A **Better Solution** is to augment every tree node to store count together with regular fields like key, left and right pointers.
Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.




```
Count of a key is shown in bracket
```

This approach has following advantages over above simple approach.

1) Height of tree is small irrespective of number of duplicates. Note that most of the BST operations (search, insert and delete) have time complexity as O(h) where h is height of BST. So if we are able to keep the height small, we get advantage of less number of key comparisons.

2) Search, Insert and Delete become easier to do. We can use same insert, search and delete algorithms with small modifications (see below code).

3) This approach is suited for self-balancing BSTs (AVL Tree, Red-Black Tree, etc) also. These trees involve rotations, and a rotation may violate BST property of simple solution as a same key can be in either left side or right side after rotation.

Below is C implementation of normal Binary Search Tree with count with every key. This code basically is taken from code for insert and delete in BST. The changes made for handling duplicates are highlighted, rest of the code is same.

```
// C program to implement basic operations (search, insert and delete)
// on a BST that handles duplicates by storing count with every node
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int key;
    int count;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    temp->count = 1;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d(%d) ", root->key, root->count);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);
```

```
    // If key already exists in BST, icnrement count and return
    if (key == node->key)
    {
       (node->count)++;
        return node;
    }

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

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

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

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

    return current;
}

/* Given a binary search tree and a key, this function
   deletes a given key and returns root of modified tree */
struct node* deleteNode(struct 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
    else
    {
        // If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
```

```
        {
           (root->count)--;
           return root;
        }

        // ElSE, delete the node

        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct 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 Program to test above functions
int main()
{
    /* Let us create following BST
              12(3)
            /        \
       10(2)      20(1)
       /   \
    9(1)  11(1)   */
    struct node *root = NULL;
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 9);
    root = insert(root, 11);
    root = insert(root, 10);
    root = insert(root, 12);
    root = insert(root, 12);

    printf("Inorder traversal of the given tree \n");
```

```
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 12\n");
    root = deleteNode(root, 12);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 9\n");
    root = deleteNode(root, 9);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```

Output:

```
 Inorder traversal of the given tree
9(1) 10(2) 11(1) 12(3) 20(1)
Delete 20
Inorder traversal of the modified tree
9(1) 10(2) 11(1) 12(3)
Delete 12
Inorder traversal of the modified tree
9(1) 10(2) 11(1) 12(2)
Delete 9
Inorder traversal of the modified tree
10(2) 11(1) 12(2)
```

We will soon be discussing AVL and Red Black Trees with duplicates allowed.

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

Source

http://www.geeksforgeeks.org/how-to-handle-duplicates-in-binary-search-tree/

Category: Trees

Post navigation

← Amazon Interview Experience | Set 188 (For SDE1) Flipkart Interview Experience | Set 25 →

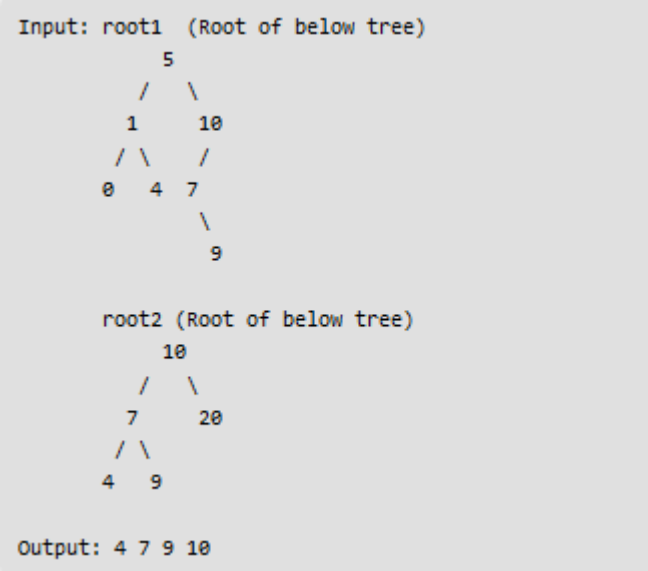
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 32

Print Common Nodes in Two Binary Search Trees

Given two Binary Search Trees, find common nodes in them. In other words, find intersection of two BSTs.

Example:



We strongly recommend you to minimize your browser and try this yourself first.

Method 1 (Simple Solution) A simple way is to one by once search every node of first tree in second tree. Time complexity of this solution is O(m * h) where m is number of nodes in first tree and h is height of second tree.

Method 2 (Linear Time) We can find common elements in O(n) time.
1) Do inorder traversal of first tree and store the traversal in an auxiliary array ar1[]. See sortedInorder() here.
2) Do inorder traversal of second tree and store the traversal in an auxiliary array ar2[]
3) Find intersection of ar1[] and ar2[]. See thisfor details.

Time complexity of this method is O(m+n) where m and n are number of nodes in first and second tree respectively. This solution requires O(m+n) extra space.

Method 3 (Linear Time and limited Extra Space) We can find common elements in O(n) time and O(h1 + h2) extra space where h1 and h2 are heights of first and second BSTs respectively.
The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element, we print it.

```
// Iterative traversal based method to find common elements
// in two BSTs.
#include<iostream>
#include<stack>
using namespace std;

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

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

// Function two print common elements in given two trees
void printCommon(Node *root1, Node *root2)
{
    // Create two stacks for two inorder traversals
    stack<Node *> stack1, s1, s2;

    while (1)
    {
        // push the Nodes of first tree in stack s1
        if (root1)
        {
            s1.push(root1);
            root1 = root1->left;
        }

        // push the Nodes of second tree in stack s2
        else if (root2)
        {
            s2.push(root2);
            root2 = root2->left;
        }

        // Both root1 and root2 are NULL here
        else if (!s1.empty() && !s2.empty())
        {
            root1 = s1.top();
```

```
            root2 = s2.top();

            // If current keys in two trees are same
            if (root1->key == root2->key)
            {
                cout << root1->key << " ";
                s1.pop();
                s2.pop();

                // move to the inorder successor
                root1 = root1->right;
                root2 = root2->right;
            }

            else if (root1->key < root2->key)
            {
                // If Node of first tree is smaller, than that of
                // second tree, then its obvious that the inorder
                // successors of current Node can have same value
                // as that of the second tree Node. Thus, we pop
                // from s2
                s1.pop();
                root1 = root1->right;

                // root2 is set to NULL, because we need
                // new Nodes of tree 1
                root2 = NULL;
            }
            else if (root1->key > root2->key)
            {
                s2.pop();
                root2 = root2->right;
                root1 = NULL;
            }
        }

        // Both roots and both stacks are empty
        else  break;
    }
}

// A utility function to do inorder traversal
void inorder(struct Node *root)
{
    if (root)
    {
        inorder(root->left);
        cout<<root->key<<" ";
        inorder(root->right);
    }
}

/* A utility function to insert a new Node with given key in BST */
struct Node* insert(struct 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 (unchanged) Node pointer */
    return node;
}

// Driver program
int main()
{
    // Create first tree as shown in example
    Node *root1 = NULL;
    root1 = insert(root1, 5);
    root1 = insert(root1, 1);
    root1 = insert(root1, 10);
    root1 = insert(root1,  0);
    root1 = insert(root1,  4);
    root1 = insert(root1,  7);
    root1 = insert(root1,  9);

    // Create second tree as shown in example
    Node *root2 = NULL;
    root2 = insert(root2, 10);
    root2 = insert(root2, 7);
    root2 = insert(root2, 20);
    root2 = insert(root2, 4);
    root2 = insert(root2, 9);

    cout << "Tree 1 : ";
    inorder(root1);
    cout << endl;

    cout << "Tree 2 : ";
    inorder(root2);

    cout << "\nCommon Nodes: ";
    printCommon(root1, root2);

    return 0;
}
```

Output:

```
 4 7 9 10
```

This article is contributed by **Ekta Goel**. Please write comments if you find anything incorrect, or you want

to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/print-common-nodes-in-two-binary-search-trees/

Category: Trees

Post navigation

← Directi Programming Questions Infosys Interview Experience Set (0n-Campus for Specialist Programmer) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 33

Construct all possible BSTs for keys 1 to N

In this article, first count of possible BST (Binary Search Trees)s is discussed, then construction of all possible BSTs.

How many structurally unique BSTs for keys from 1..N?

```
For example, for N = 2, there are 2 unique BSTs
     1               2
      \             /
       2           1

For N = 3, there are 5 possible BSTs
  1              3        3         2      1
    \           /        /        /  \      \
     3        2         1        1    3      2
    /        /           \                    \
   2      1               2                    3
```

We strongly recommend you to minimize your browser and try this yourself first.

We know that all node in left subtree are smaller than root and in right subtree are larger than root so if we have ith number as root, all numbers from 1 to i-1 will be in left subtree and i+1 to N will be in right subtree. If 1 to i-1 can form x different trees and i+1 to N can from y different trees then we will have x*y total trees when ith number is root and we also have N choices for root also so we can simply iterate from 1 to N for root and another loop for left and right subtree. If we take a closer look, we can notice that the count is basically n'th Catalan number. We have discussed different approaches to find n'th Catalan number here.

How to construct all BST for keys 1..N?
The idea is to maintain a list of roots of all BSTs. Recursively construct all possible left and right subtrees. Create a tree for every pair of left and right subtree and add the tree to list. Below is detailed algorithm.

```
1) Initialize list of BSTs as empty.
```

```
2) For every number i where i varies from 1 to N, do following
......a)  Create a new node with key as 'i', let this node be 'node'
......b)  Recursively construct list of all left subtrees.
......c)  Recursively construct list of all right subtrees.
3) Iterate for all left subtrees
   a) For current leftsubtree, iterate for all right subtrees
        Add current left and right subtrees to 'node' and add
        'node' to list.
```

Below is C++ implementation of above idea.

```
 // A C++ prgroam to contrcut all unique BSTs for keys from 1 to n
#include <iostream>
#include<vector>
using namespace std;

//  node structure
struct node
{
    int key;
    struct node *left, *right;
};

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

// A utility function to do preorder traversal of BST
void preorder(struct node *root)
{
    if (root != NULL)
    {
        cout << root->key << " ";
        preorder(root->left);
        preorder(root->right);
    }
}

//  function for constructing trees
vector<struct node *> constructTrees(int start, int end)
{
    vector<struct node *> list;

    /*  if start > end   then subtree will be empty so returning NULL
        in the list */
    if (start > end)
    {
        list.push_back(NULL);
```

```
        return list;
    }

    /*  iterating through all values from start to end  for constructing\
        left and right subtree recursively  */
    for (int i = start; i <= end; i++)
    {
        /*  constructing left subtree   */
        vector<struct node *> leftSubtree  = constructTrees(start, i - 1);

        /*  constructing right subtree  */
        vector<struct node *> rightSubtree = constructTrees(i + 1, end);

        /*  now looping through all left and right subtrees and connecting
            them to ith root  below  */
        for (int j = 0; j < leftSubtree.size(); j++)
        {
            struct node* left = leftSubtree[j];
            for (int k = 0; k < rightSubtree.size(); k++)
            {
                struct node * right = rightSubtree[k];
                struct node * node = newNode(i);// making value i as root
                node->left = left;              // connect left subtree
                node->right = right;            // connect right subtree
                list.push_back(node);           // add this tree to list
            }
        }
    }
    return list;
}

// Driver Program to test above functions
int main()
{
    // Construct all possible BSTs
    vector<struct node *> totalTreesFrom1toN = constructTrees(1, 3);


    /*  Printing preorder traversal of all constructed BSTs   */
    cout << "Preorder traversal of all constructed BSTs is \n";
    for (int i = 0; i < totalTreesFrom1toN.size(); i++)
    {
        preorder(totalTreesFrom1toN[i]);
        cout << endl;
    }
    return 0;
}
```

Output:

```
 Preorder traversal of all constructed BSTs is
1 2 3
1 3 2
```

```
2 1 3
3 1 2
3 2 1
```

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

Source

http://www.geeksforgeeks.org/construct-all-possible-bsts-for-keys-1-to-n/

Category: Trees