**Binary Search Tree:**

**Binary Search Tree Insertion and Search:**

I totally remember this.

**BST Deletion:**

There are cases.

When node is leaf node. Simply remove it.

Node to be deleted has only one child: Copy the child to the node and delete the child.

**Node to be deleted has two children:**

Find inorder successor of the node.Copy contents of the inorder successor to the node (the current node to be deleted) and delete the inorder successor. Note that inorder predecessor can also be used. **(do it until we reach a situation when node is a leaf node or the node to be deleted has 1 child)**

Now, there’ s the challenge. You think which one is inorder successor? Now, for a node whose both children is present, the inorder successor is always the leftmost node of right subtree. Always. **(since, both children are present)** That makes this problem easy.

**Construct BST From Given PreOrder Traversal:**

**First Approach:**

The idea used here is inspired from method 3 of this post. 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;**

It must be incremented here. Think again. Because, a node is chosen based on some condition

**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;**

**}**

**Note:** check the terminal condition. Check where preIndex is being updated? Check in which condition we are constructing a root

**Second Approach:**

1. Create an empty stack.

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

3. 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.

**4.**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. (the last node will be smaller than stack’s top value.

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

**(this is a good problem)**

**Binary Tree To BST Conversation:**

Traverse the binary tree in inorder fashion.

Now, construct Balanced BST using that inorder traversal.

**Sorted Linked List To BST:**

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.

Now, the problem with this approach?

Every time we need to traverse upto middle node which is costly. **Note: this approach produced a balanced bst. You cannot produce a skewed BST.**

This can be modified.

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.

struct TNode\* sortedListToBSTRecur(struct LNode \*\*head\_ref, int n)

{

/\* Base Case \*/

/\*This will be base case since, we are constantly recurring fro first n/2 half

if (n <= 0)

return NULL;

/\* Recursively construct the left subtree \*/

struct TNode \*left = sortedListToBSTRecur(head\_ref, n/2);

/\*Now, consider the fact, where we reach n==0

Consider the linked list as 1->2->3->4->5->6->7

Now, we reach n=0 due to constant recursion for first n/2 elements

Now, head\_ref is being sent without assigning a value to it

So, head\_ref becomes 1

1,2,3,4,5,6,7

[123] [4567]

[1][23] [[45][67]]

[1] is set as head reference

And head\_ref++ or head\_ref=head\_ref->next

/\*Now, control is back to previous recursion call

Now, left node is sorted

Head\_ref is 2

Head\_ref=head\_ref->next;

Now, recursion goes deep again for element 3

This is how it works

\*/

/\* 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;

}

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

Reverse Inorder Traversal.

**Convert A BST To Min Heap Using Extra Space:**

If we are allowed to use extra space, we can perform inorder traversal of the tree and store the keys in an auxiliary array. As we’re doing inorder traversal on a BST, array will be sorted. Finally, we construct a complete binary tree from the sorted array. **We construct the binary tree level by level and from left to right by taking next minimum element from sorted array.** The constructed binary tree will be a min-Heap. This solution works in O(n) time, but is not in-place.

**Note that we construct the binary tree level by level. This is same as converting a sorted array to a binary tree level by level using queue. Now, binary tree here can be constructed as Complete binary tree( I.e. all leaf nodes are at last level and leaf nodes are as left as possible)**

**Convert A BST TO Min Heap WithOut Using Extra Space Or In Place:**

In order traversal then pre order traversal.

**Convert BST From It’s Given Level Order Traversal:**

**First trick** is to try the same way as from a given level order traversal we make Binary tree. I.e. using queue.

Second Trick is to set limit for right child and left child. So, If the tree is not complete binary tree, our program does not fail.

**Check If A Binary Tree Is BST Or Not:**

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.

**Kth Smallest Element In BST:**

**First Solution:In order traversal**

This will take O(n)

**Second Solution: O(h) or O(logn) solution:**

**But with the cost of an extra field for each bst node.** (left count. The number of nodes present in left subtree)

**Second Largest Element In BST:**

Do reverse inorder traversal.

**Kth Largest Element Without Maintaining Any Extra Information:**Do reverse inorder traversal.

**Kth Smallest Element in O(k) time Using O(1) Extra space:**

Convert it to in place DLL.

**Check whether BST contains Dead End or not:**

Given a Binary search Tree that contains positive integer values greater then 0. The task is to check whether the BST contains a dead end or not. Here Dead End means, we are not able to insert any element after that node.

If we take a closer look at problem, we can notice that we basically need to check if there is leaf node with value x such that x+1 and x-1 exist in BST with exception of x = 1. For x = 1, we can’t insert 0 as problem statement says BST contains positive integers only.

Now, we need to maintain two separate Hashtable. One for internal nodes and one for leaf nodes.

Now, we will traverse the hash table for leaf nodes and check the presence of x+1 and x-1 in the hashtable maintained for internal node.

**Check If Given Sorted Subsequence Exists In BST Or Not:**

An efficient solution is to match elements of sub-sequence while we are traversing BST in inorder fashion. We take index as a iterator for given sorted sub-sequence and start inorder traversal of given bst, if current node matches with seq[index] then move index in forward direction by incrementing 1 and after complete traversal of BST if index==n that means all elements of given sub-sequence have been matched and exist as a sorted sub-sequence in given BST.

**Check If An Array Can Represent The Inorder Traversal Of BST Or Not:**

The check whether the array is sorted or not.

**Check if two BSTs contain same set of elements**

**Approach 1: HashMap based Approach:**

This method is an optimization of above approach. If we observe carefully, we will see that in the above approach, search for element in the list takes linear time. We can optimize this operation to be done in constant time using a hashmap instead of list. We insert elements of both trees in different hash sets. Finally we compare if both hash sets contain same elements or not.

Now, the problem with it is It takes O(n) extra space.

Since, first bst is to put in hashmap. And, during traversal of second bst(inorder traversal) we will check for each element whether an element is present or not in the hashmap.

**Approach 2: O(logh) extra space and O(n) time complexity**

We will use inorder traversal of both bsts simultaneously using stack.

**Now, how to do tree inorder traversal of tree:**

1. Make curr as root
2. Continue
3. While curr->left!=NULL keep pushing curr in the stack and make curr=curr->left
4. If curr->left is Null, pop the stack top. Print it. Now do the following for popped node
   * + 1. Make it curr. If curr->right!=NULL make curr as curr->right. Now, do steps 2 and 3 and 4.
       2. Otherwise, if curr->right=NULL, keep repeating step 3.
5. Continue until both curr is NULL and stack is empty.

**Largest Number In BST Which Is Less Than Or Equal To n:**

We follow recursive approach for solving this problem. We start searching for element from root node. If we reach a leaf and its value is greater than N, element does not exist so return -1. Else if node’s value is less than or equal to N and right value is NULL or greater than N, then return the node value as it will be the answer.

Otherwise if node’s value is greater than N, then search for the element in the left subtree else search for the element in the right subtree by calling the same function by passing the left or right values accordingly.

For understanding the condition that current node’s value is less than n and current node’s right subtree (or right child ) is NULL, consider the tree:

18

/

14

/ \

10 15

/

7

And try to find largest number which is smaller than or equal to 12

If current node is greater than n, and left subtree does not exist there is no solution.

18

\

21

And try to find largest number which is smaller than or equal to 12

If current node is smaller than n and both left subtree and right subtree exists, we will visit right subtree. If right subtree’s value is greater than n, then current node is the answer.

If current node is bigger than n and both left subtree and right subtree exists, we will visit left subtree

(you can solve it with two other methods: keeping parent pointer with extra space or convert it to DLL in place and do that)

**Find The Node With Minimum Or Maximum Value In BST:**I can do it.

**Find Inorder Successor Of A Node In BST:**

We either need parent node (with extra space) or inorder traversal or

**converting the inorder traversal of BST into DLL and traverse upto given node.**

**Print BST Keys In Given Range:**

I can do it.

**Floor And Ceiling For A Value In BST:**already floor of a value in BST is discussed.

Now, similarly we can carefully think can observe critical test cases to think about ceiling value too.

**Find If There is A Triplet In The Balanced BST That Adds Up To 0:  
  
first, converting the balanced bst into DLL will be convenient.**

**Second,** 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.  
  
**Third,** we can keep track of last node of DLL and two pointer method on this sorted DLL to find the pair. ( in DLL with sum equal to key of current node multiplied by -1.)

However, in these two pointer, the starting first node will always be the next node of the current node with negative values and starting last node will always be the last node.

**What’s more benefiting/ beneficial?**

We need to do this as long as we find negative values in the DLL.

**Find If There Is A Pair With Given Value In The Given BST?  
  
Now, If space is not a problem,** traverse the bst in inorder fashion and store them in an array, then use two pointer method.

Now, another approach which takes very little space is to convert the bst into DLL then do the same two pointer method.

**Another approach** which will take 2\*O(h) space that traverse the same bst in inorder and reverse in order simultaneously using temp stack.

This is the mechanism to traverse inorder traversal using the stack:

1) Create an empty stack S.

2) Initialize current node as root

3) Push the current node to S and set current = current->left until current is NULL

4) If current is NULL and stack is not empty then

a) Pop the top item from stack.

b) Print the popped item, set current = popped\_item->right

c) Go to step 3.

5) If current is NULL and stack is empty then we are done.

**Find If There Is A Pair with given sum in two different BSTs?**

Save inorder traversals of them in two different arrays. And do two pointer method.

Converting them to DLL can be done too. To save additional memory space. (but, let them remain as separate DLL)

**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 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.

**Merge Two Balanced BSTs Using Extra Space:**

Store the Inorder traversals in two arrays. Now, merge them easily. Now, apply the program which converts a sorted array to balanced bst.

**Merge Two Balanced BSTs With Limited Extra Space: (It is basically printing Of Two Balanced BSTs with Limited Extra Space)**

Traverse two bsts simultaneously in inorder fashion using temporary stacks.

For each stack, the space complexity is O(h)

Now, in case of normal bst, O(h) could be O(n)

But, in case of balanced bst, O(h) would be O(log2(n+1))

**Two Nodes Of Balanced BSTs Are Swapped, Correct The BST?**Three nodes method. First, middle, last.

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

**Case 2: The Swapped nodes are adjacent in the inorder traversal of the bst.**

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.

**Sorted order printing of a given array that represents a Complete BST:**

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

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);

}

**Inorder Successor Or Predecessor Of A Tree:**

**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

This is hassle free (I.e. step 3). This eliminates the need of parent node.

**Find All Conflicting Appointments:**1) Create an Interval Tree, initially with the first appointment.

2) Do following for all other appointments starting from the second one.

a) Check if the current appointment conflicts with any of the existing

appointments in Interval Tree. If conflicts, then print the current

appointment. This step can be done O(Logn) time.

b) Insert the current appointment in Interval Tree. This step also can

be done O(Logn) time.

**A slight modification can be done to this approach with the cost of an extra variable.**

That is suppose two intervals are not overlapping. But, combines like, second interval’s start time=first interval’s end time+1 then if we store the combined endtime we need to make less traversals.

Also, the appointments should be sorted if it wants to find the maximum number of intervals non conflicting. (like activity selection problem)

**How To Handle Duplicates In Binary Tree:**

I can do that.

**Print Common Nodes In Two Binary Search Trees:**

**Method 1:** Use Hashmap.

**Method 2:** simultaneous inorder traversals of both trees using stacks.

**LCA In BST:**   
  
It will be slightly modified than the approach used in Binary Tree.

Consider node1 and node2 as values. Now, if both are present in the left subtree or right subtree then go for it(I mean we need to traverse deeper). If one is present in left subtree and one in right subtree then current node is LCA.

**How this approach is different than the approach used in Binary Tree?**

Because, In BST, we already know if an element can possibly present in left subtree or right subtree (based on it’s value)

**Data Structure Of Single Resource Reservations:**   
  
Only starting time Is given. And each job requires k time unit to finish. (since, each job requires k times unit to finish, we can choose this approach. Otherwise, it would be either interval tree or activity selection)

1) Every job requires exactly k time units of the machine.

2) The machine can do only one job at a time.

3) Time is part of the system. Future Jobs keep coming at different times. Reservation of a future job is done only if there is no existing reservation within k time frame (after and before)

4) Whenever a job finishes (or its reservation time plus k becomes equal to current time), it is removed from system.

(since, these are reservation request, this will already be sorted based on arrival time)

**Count BST Nodes That Lie In A Given Range:**

I can easily do that.

**Count BST Subtrees That Lie In A Given Range:**

I can easily do that.

**Find Closest Element In A BST:**The closest element in a BST would be either floor or ceiling of a value.

So, we need to find floor and ceil element simultaneously in a recursive manner.

How can we find ceil

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

1. Root data < key value, certainly the ceil value can’t be in left subtree.Proceed to search on right subtree as reduced problem instance.
2. Root Data is > key value. And no left subtree is present. Then root data is the ceiling of the key value.
3. Root data is > key value and left subtree is present. Store current root data as possible\_ceil value and move to the left subtree.

These small observations

Now, back to this problem. Here, we should maintain two values. Possible\_greater value and possible\_smaller\_value.

At any point if root->data matches with given value, It will be the result.

Otherwise, develop your own rules based on observations and make them work accordingly.

**Find The Largest Subtree In A Tree Which Is Largest BST Too:**

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.

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;

}

}

**Construct All Possible BSTs using Key From 1 to N:**

vector<struct tree\_node \*> construct\_bsts\_util(int low,int high)

{

vector <struct tree\_node \*> list;

if(low>high)

{

list.push\_back(NULL);

return list;

}

for(int i=low;i<=high;i++)

{

vector<struct tree\_node \*> left\_subtree\_nodes=construct\_bsts\_util(low,i-1);

vector<struct tree\_node \*>right\_subtree\_nodes=construct\_bsts\_util(i+1,high);

//left subtree contains all left subtree combination for current node as root

//right subtree contains all right subtree combination for current node as root

for (int j = 0; j < left\_subtree\_nodes.size(); j++)

{

//for current node all left subtree combination

struct tree\_node\* left = left\_subtree\_nodes[j];

for (int k = 0; k < right\_subtree\_nodes.size(); k++)

{

struct tree\_node \* right = right\_subtree\_nodes[k];

Struct tree\_node \*node = create\_new\_tree\_node(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;

}

**Find Distance Of Two Nodes In A BST:  
  
LCA application.**

**Find If All internal Nodes have Only One Child Or Not:**

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.

**How To Implement Decrease Key Or Change Key For BST:**

The idea is to delete it first. Then insert with new value.

**Maximum Element B/W Two Nodes In BST:**

LCA based problem.

**Remove All Leaf Nodes From BST:**

Do it in preorder.

**Find Median Of BST using O(n) Time Complexity and O(1) space complexity:**

**Convert bst into dll**