Today.

lowest common ancestor. (3)

print BST elements in given range (2).

check whether BST contains Dead End (2).

inorder:

↳ New way

Total (4) ways.

1) Recursive
2) Iterative
3) Morris traversal. — Optimised Space Complexity O(1).
4)

common Node in BST. Solve (2).

+ Algo.

Sorted LL to BST. (balanced). (1).

(3)+1 Merge two BST. (2). — NlogN vector sort
— O(N) Recursive inorder way
— O(N) inorder iterative stack all left Elements Approach.

Fixing two Nodes of a BST.

Fixing two Nodes of a BST. (Recover BST)

↳ Basic inorder Vector Sort NlogN.

↳ using Stack inorder, print & pop & insert elements all this should be right & all its' left Elements

↳ Element popped.

↳ First, second + Morris traversal

O(1) optimised space.

Largest BST (2)

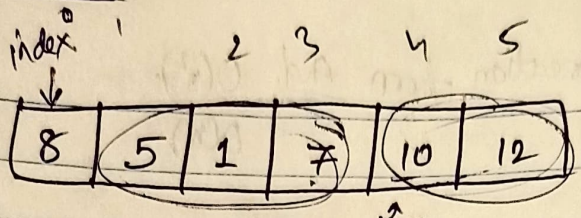
Maximum Sum BST in Binary Tree

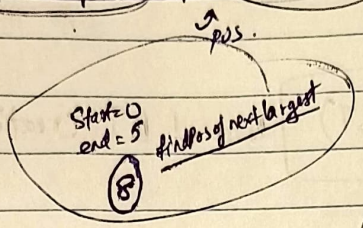
: Construct BST from preorder (3)

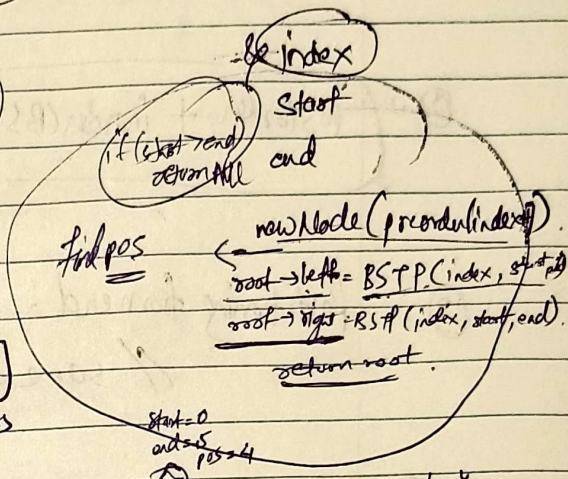
Construct BST from postorder (3)

preorder traversal & BST ← could be wrong/right, validate. (2)

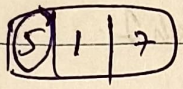
*BST from Preorder

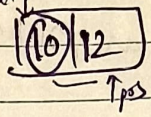






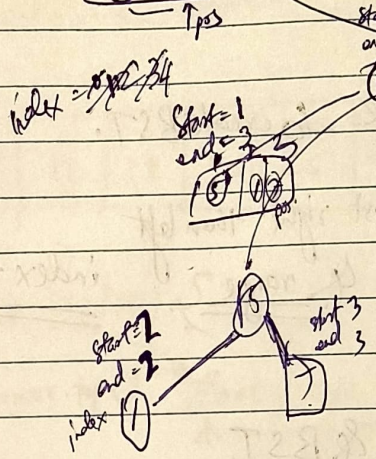
① Partition:

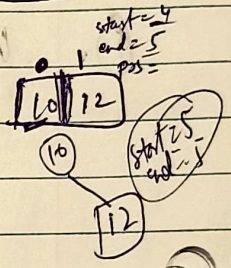




T.C: $O(n^2)$

S.C: $O(n)$.

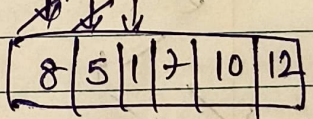




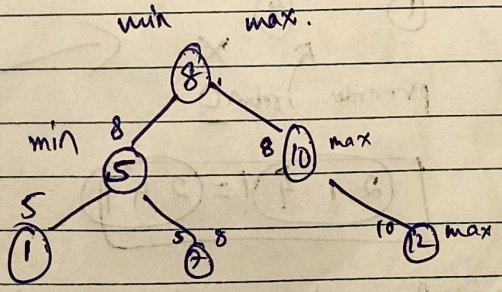
② Normal BST : $O(n^2)$. $O(n)$. BST Creation.

③. inorder preorder BT Approach: $O(n)$ but extra space $O(n)$. for inorder vector.

④. using range to create BST. : $O(n)$.



index=0



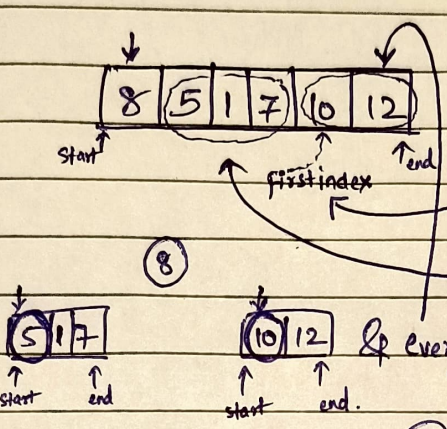

```
&preorder &index  lower, upper.
if (index >= preorder.size() || out of range)
    return NULL
Node* root = newNode(preorder[index++]);
root->left = BSTPre(pre, index, lower, root->data);
root->right = BSTPre(pre, index, root->data, upper);
return root;
```

~~if(index != preorder.size())~~

Lecture 16 BST.

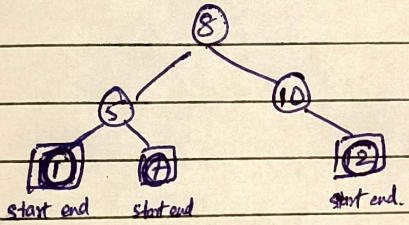
P(1): Construct BST from Preorder.

Approach 1:



Take the first element of the preorder array as the root, then in each subarray (start, end), find the first index where the value becomes greater than the root; Everything before it goes to the left subtree & everything from there to the end goes to the right subtree.

Recursively repeat this process on those left and right segments until the segment becomes empty, which constructs the entire BST.




```
int findPosNextLargest(int pre[], int start, int end, int rootVal) {
    int i = start;
    while(i<=end && pre[i]<=rootVal)
        i++;
    return i; //maybe n+1 if none greater
}
```

```
Node* BSTPreorder(int pre[], int start, int end) {
    if(start > end) return NULL;
    Node* root = newNode(pre[start]);
    if(start == end) return root;
    int split = findPosNextLargest(pre, start+1, end, pre[start]);
    root->left = BSTPreorder(pre, start+1, split-1);
    root->right = BSTPreorder(pre, split, end);
    return root;
}
```

T & S Complexity?: worst case: skewed tree / strictly inc/decr

Time: $O(n^2)$. Space: $O(n) \approx O(n)$

For each node, the algorithm scan forward through a portion of array to find the split point b/w left & right subtrees.
In a worst case (strictly increasing/strictly decreasing preorder, tree is skewed), the split search for each node take $O(n)$ — n nodes $\times O(n) \approx O(n^2)$.

Construct BST from Preorder

Approach 2: Normal BST insertion using preorder (insert 1-by-1)

Process the preorder arr left to right, & for each value,
start at the root & walk down the tree (go left if smaller, right if larger)
until you hit a NULL & insert there.

Time complexity:
Average (balanced ish tree): each insert is $O(\log n)$, for n elements → $O(n \log n)$
Worst case (sorted preorder, tree becomes a chain): each insert is $O(n)$, → overall $O(n^2)$.

Space complexity:

- Recursive insertion: $O(h)$
  Average: $O(\log n)$
  Worst: $O(n)$.
- Iterative insertion: $O(1)$.
  inserting into bst using loops instead of recursion,
  implemented using a while loop & a couple of pointers (curr, parent)
  instead of function calling itself. (no call-stack overhead)

Approach 3: Sort to get inorder,
then build BT using preorder + inorder.

Time complexity:
↓ Linear search of ele in inorder
n nodes × $O(n)$
$O(n^2)$

↓ Binary Search or hashmap
n nodes × $\log n$
$O(n \log n)$.

Space complexity:
recursive stack (+ hashmap) (if).
$O(h)$
$\approx O(n)$. worst case.

Approach 4: Finally Optimized Approach - using range to validate the node, & then creating it,

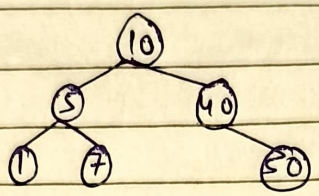
T.C: Every ele read exactly once & placed, without searching / scanning.
$O(n)$
S.C: $O(h) \approx O(n)$.

if index is out of range, or if index == preorder.size()
return NULL;

(L R N)

P2: Construct BST from Postorder.

0	1	2	3	4	5
1	7	5	50	40	10



A1: Normal BST Creation from End

T.C: $O(n^2)$ worstcase

S.C: $O(n)$. if recursive
or $O(1)$ if iterative.

A2: (Postorder + inorder) → Normal BT Creation.
(sort (postorder))

T.C: $O(n \log n)$ or $O(n^2)$ worst case + linear search.

S.C: $O(h) \approx O(n)$.

A3: Partitioning from end // Same as Preorder Approach 1.

T.C: $O(n^2)$

S.C: $O(n)$.

A4: using range to create BST. (Optimised).

order: first right then left side.

validate range

& at each node creation

index--.

* Construct BST from Postorder.

① Normal BST creation from end. $O(n^2)$. $O(n)$

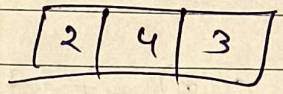
② Postorder + inorder(BST) → Normal BT creation.

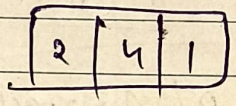
③ Partitioning from end: $O(n^2)$.
// same as preorder Approach 1.

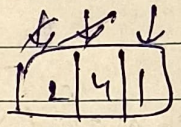
④ Using range to create BST.

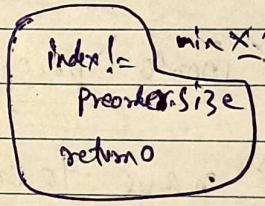
first right then left
range index--.

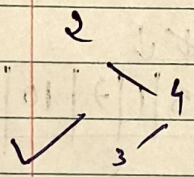
** Preorder & BST.

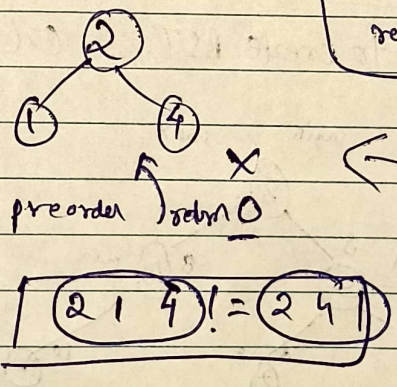












① Create BST, Evaluate Preorder & compare it with Question. $O(n^2)$.

② Using range to create BST. (By just looking at index, we can confirm if it is BT or not).

T.C. O(n)
S.C. O(n)

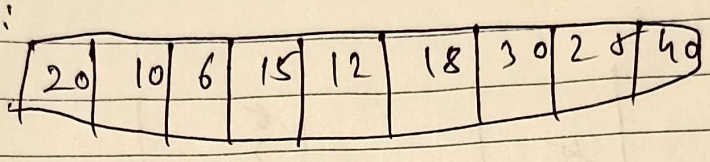
if (index == preorder.size) return 1 else return 0. check

② — But Segmentation fault GFG why (stack & heap) issue.

recursive stack uses → stack DS which is limited & not Heap DS which is comparatively more.

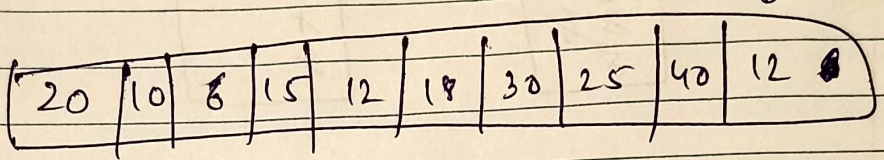
③. iterative Approach. 2 stack (STL). [Memory Allocated In Heap].
range check & BST Creation.

Dry Run this:



valid ✓

&

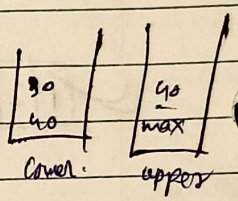


invalid.

SUDOCODE:

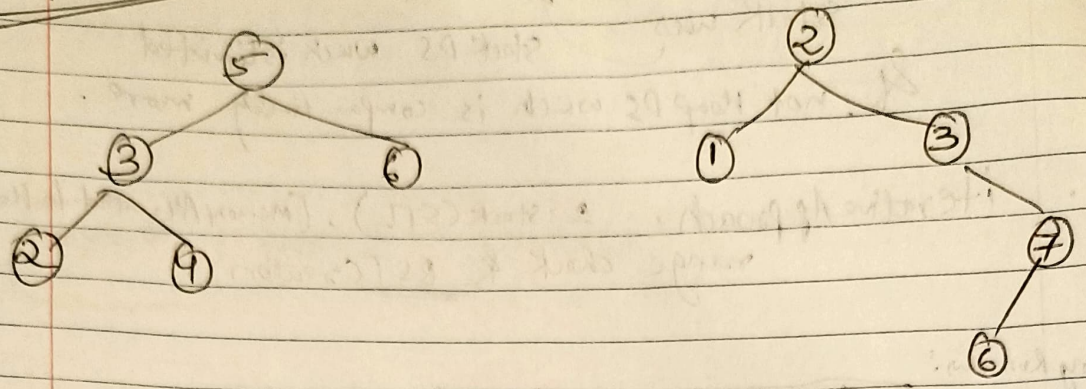
```
for(int i=0; i<N; i++) {
① . arr[i] < lower.top()   // out of range.
    return 0.                & ans not possible

② . while (arr[i] > upper.top())   // out of range
      upper & lower.pop().          // pop until it satisfies the range.

③ .   left = lower.top
      right = upper.top
      lower & upper.pop.
      lower.push(arr[i])
      upper.push(right)
      lower.push(left)
      upper.push(arr[i])
}
```

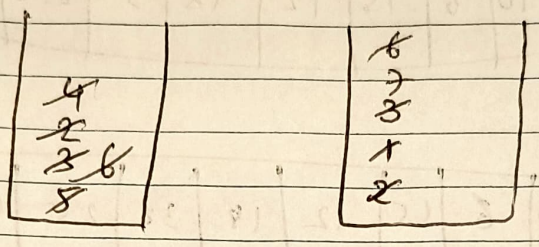
within the range.
Make changes to right & left range based on curr node.

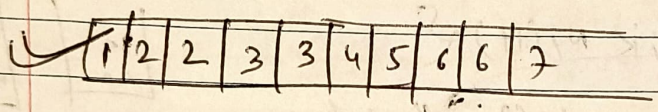


Merge 2 BST

3rd Approach :



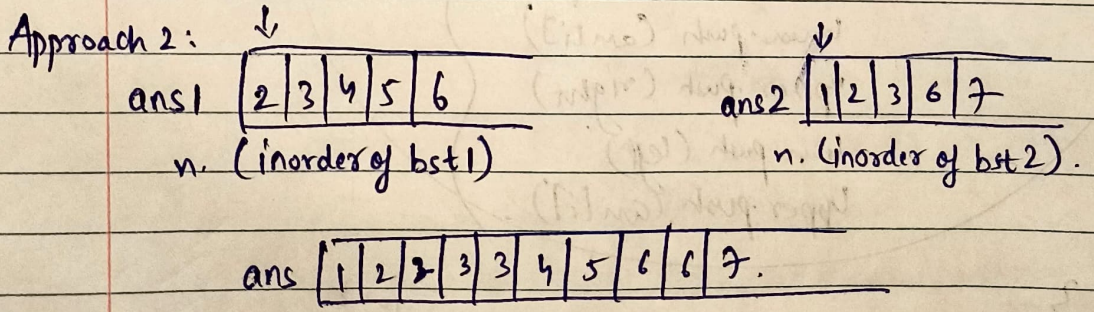




Approach 1: create a vector,
inorder traversal of bst1
inorder traversal of bst2
sort the final vector nlogn & return that as an ans.

T.C: $O(n \log n)$. S.C: $O(h) \approx O(n)$.

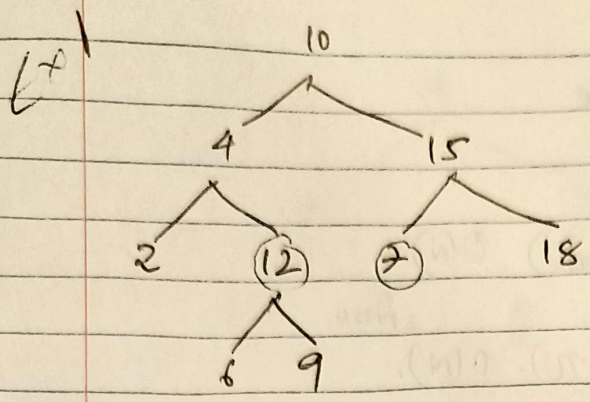
Approach 2:



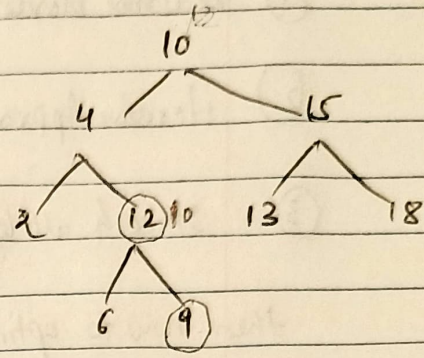
T.C: $O(n) \approx O(n)$.

SC: $O(h)$.

Fixing 2 Nodes BST



Ex: 2 — it will never be the case.



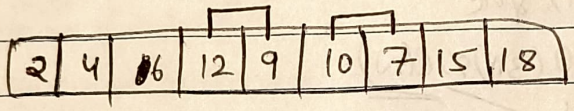
(1)

inorder

+ sort

+ inorder (to make changes to corresponding nodes)

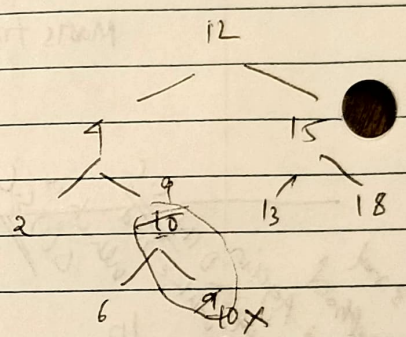
it is fixed only 2 nodes are at wrong place.



First Mistake — first: 12, second: 9

Second Mistake — third: 10, fourth: 7

swap always.



3 shuffles 3 wrong nodes ×.

first: 12 → 7

second: 9 → 10

swap

Ex3:

Now based on Ex1, Ex3, & Ex4 what we can do is.

We can also rely on just 2 variables.

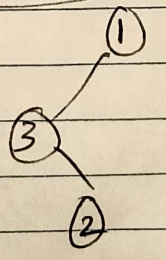
initialising

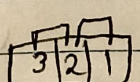
first: -1

second: -1

if 1st time mistake update first & second

if 2nd time occurs only update second variable



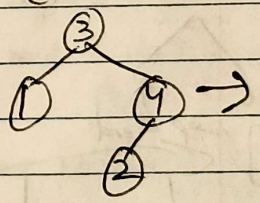


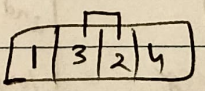
1st Mistake { first = 3, second = 2 }

2nd mistake { third = 2, fourth = 1 }

swap (always)

Ex4:





first: 3

second: 2

Only 1st Mistake no 2nd mistake in that case swap first & second.

fixing 2 nodes in a BST.

Now in any case:

it is going to consume space

① recursive inorder (recursive stack). O(N)

② iterative Approach (stack STL). O(N). $2N \approx N$.

③ Stack & all left Elements (stack STL) O(N).

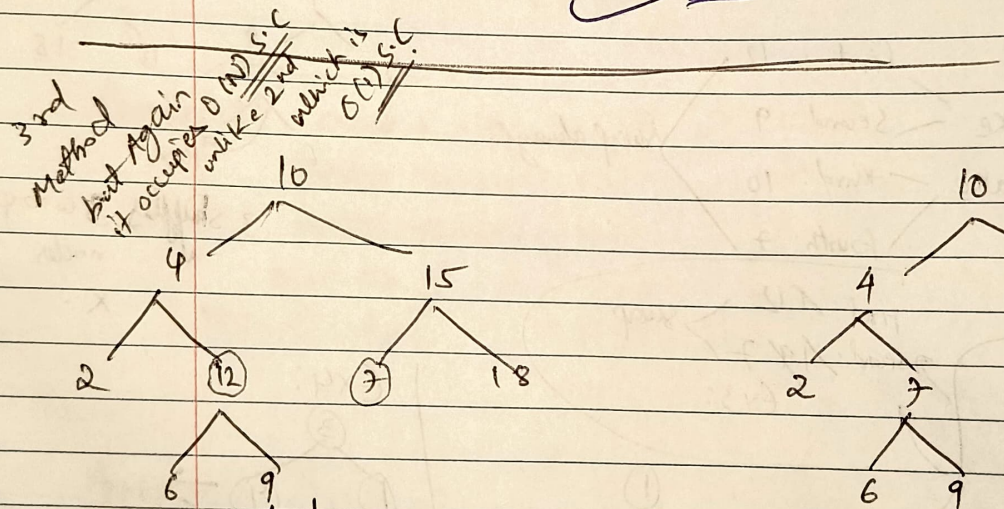
then how to optimize space complexity ?

that's where

④th Approach for inorder calculation comes

Morris traversal O(1) space

T: $O(3N) \approx O(N)$



first: 12
second: ~~9~~ 7

swap at end

12 popped
9 inserted
9 ⊅ 12 ×

violates 1st time

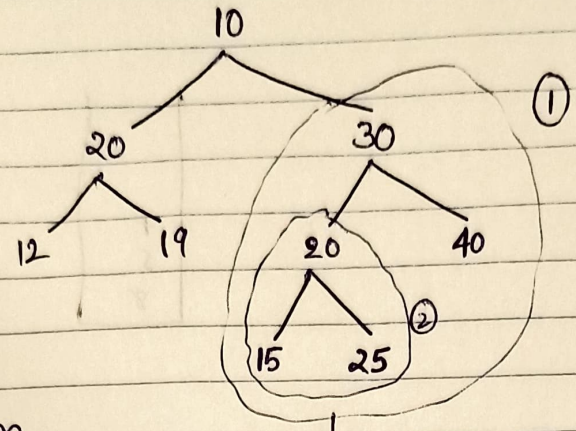
2nd violation.
10 popped
15, 7 inserted
7 ⊅ 10 so changes

No violation.

Rule: whenever pop & insert all the elements inserted should be > element popped.

Exception: initial insertion.

Largest BST



Approach 1:

BST we can confirm using inorder,
& for largest bst, we need to check for every node.

& if ans is yes, calculate total no of nodes for that root node & store it in global variable

largest = max (largest, currTotalNoNodes).
return largest.

isBST()
1 → O(n).
2 → . .
}
n → O(n)
n x O(n)
$O(n^2)$.

Now, I know ① is BST,
do I seperately need to check for it's child nodes?
2 is subset of 1
& if 1 is BST, then 2 is obviously BST, &
size of 1 > size of 2

But Again, how are we going to do so?
for this too,
we need to check,
10, 20, 12, 19, (30) ← it will give us expected ans. then no need to check further.

but suppose if the largest BST size = 1 (leaf nodes),
then again we atleast need to check for all nodes & that will take $O(N^2)$.

How can we do it in one iteration? 5 steps

1 leftBST
2 RightBST
3 leftSide → max < root → data } if this
4 Rightside → min > root → data } return to parent node, it's
5 Size = leftSide + Rightside + 1 ← else 0 (check parent's other side subtree).