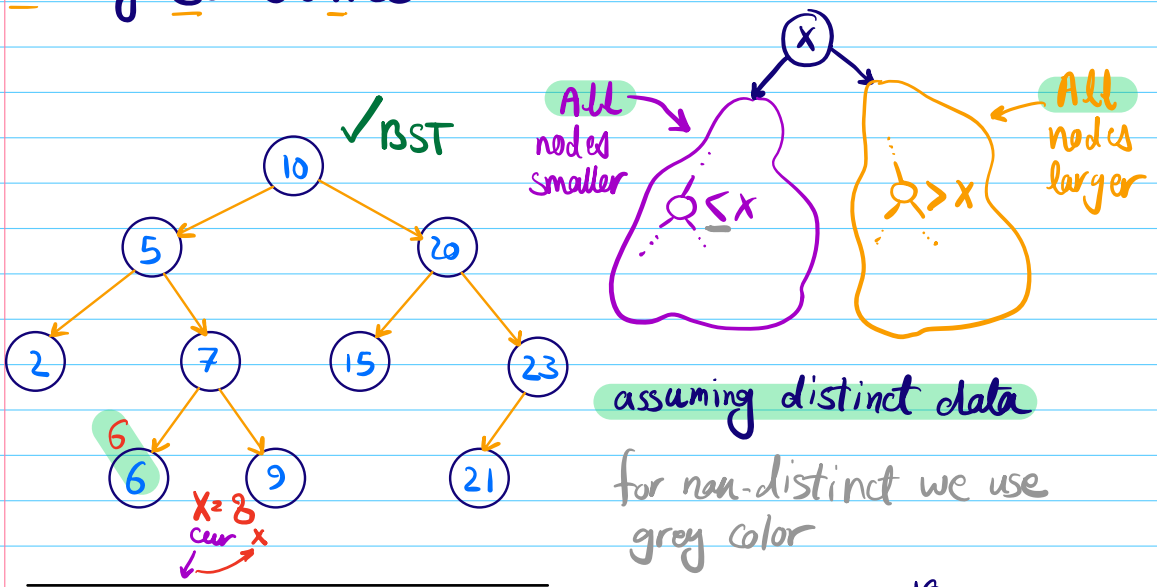
AVL trees
B-trees
fib trees
RB trees
↓
beyond the scope

Today Topics

BST
- intro
- search
- insert
- remove (optional)
- check BST
- make BST
- recover BST
- balanced BST

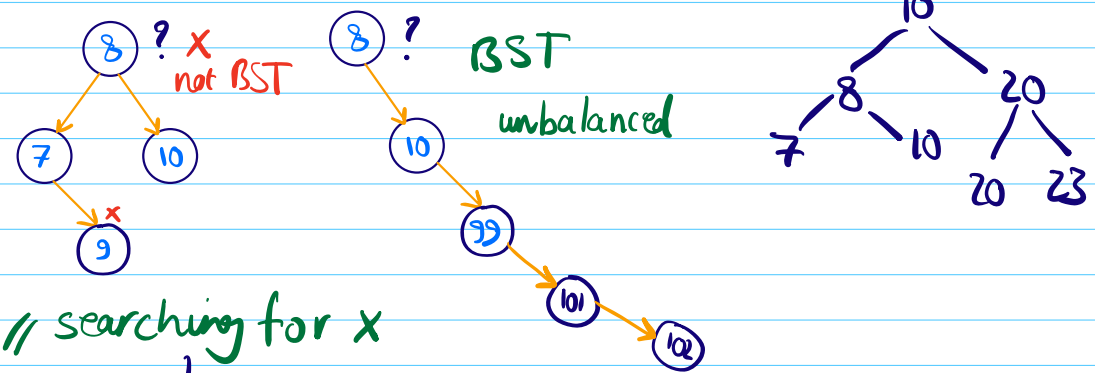
Binary Search Tree



height of tree
H

assuming distinct data

for non-distinct we use grey color



if tree is balanced
H ~ log(n)

search

// searching for x

```
cur = root
Parent = null
while( cur != null){
    if( cur.data == x ) ret (cur, Parent) // assuming distinct otherwise cont. to left
    Parent = cur
    if( x <= cur.data) cur = cur.left,
    else  cur = cur.right
}

ret null, Parent // does not exist
```

① add cur to ans list

// x > cur.data

SC
O(1)

TC search O(H)

cur = root

* Find min data node in BST?
  left most node
  while(cur.left != null)
      cur = cur.left

* Find max data node in BST?
  right most node
  while(cur.right != null)
      cur = cur.right

* for search, besides node w/ data == X
  Get parent node of the X as well.

keep BST status

insert new node in BST

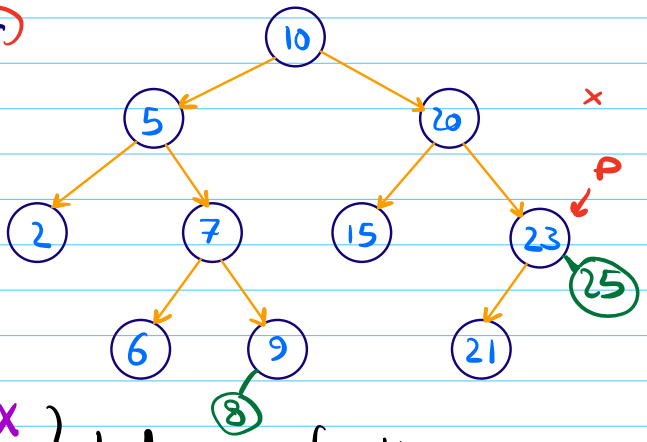
SC O(1)

TC search O(H)

insert(8); insert(3)

insert(25);

HW duplicate insert(7);

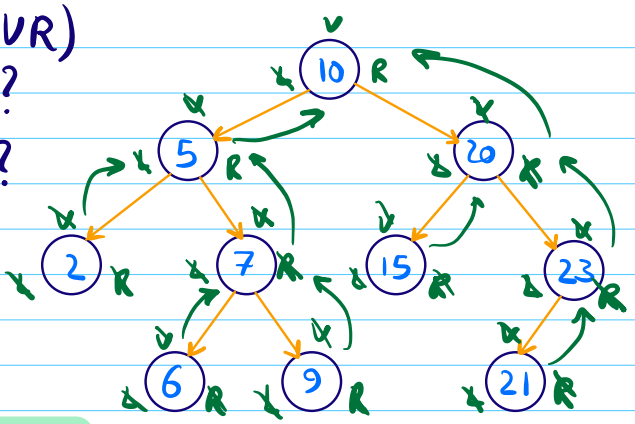


① search for new node X
② get pointer parent P
} last page function

if X ≤ P.data P.left = new TNode(X)
else P.right = new TNode(X) // X > P.data

⚠ check P != null

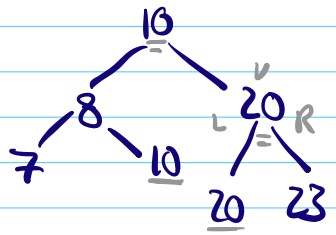
- What is the inorder (LVR) traversal of this BST?
- Any specific property?



2, 5, 6, 7, 9, 10, 15, 20, 21, 23,

→ inorder traversal of BST is sorted in increasing order

7, 8, 10, 10, 20, 20, 23



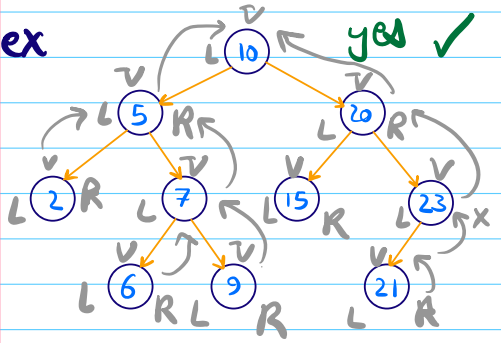
related Question in your assignments:
Find the largest subtree that is BST.

Max < 10 < min

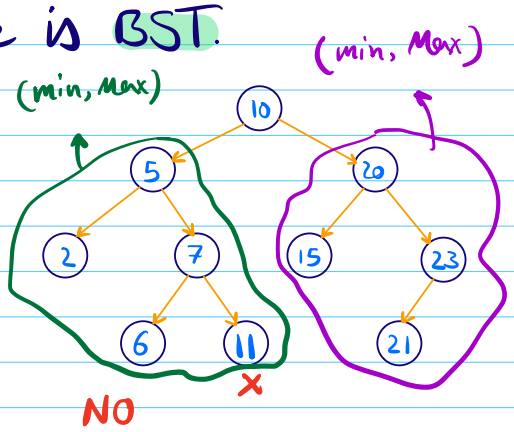
P1 check if a given binary tree is BST

true / false

ex



ex



LVR

sorted →

2, 5, 6, 7, 9, 10, 15, 20, 21, 23

idea1: inorder traversal (Step 1) of BST → sorted array? (Step 2) | true / false

TC: O(n+n) ~ O(n)

SC: O(H+n) ~ O(n)

idea2:

for root (10):

1. all elements on left smaller than root (root > O)
2. all elements on right larger than root (root < O)

2, 5, 6, 7, 11 < 10 < 15, 20, 21, 23

min Max min Max

post order LRV

idea2 Code

false

pass max min by ref

isBst = true;

Pair ← <min, max> ↔ bool checkBst(TreeNode r) {

if(root = null) {

ret int_max > int_min

}

Pair ← L = checkBst(r.left) // L

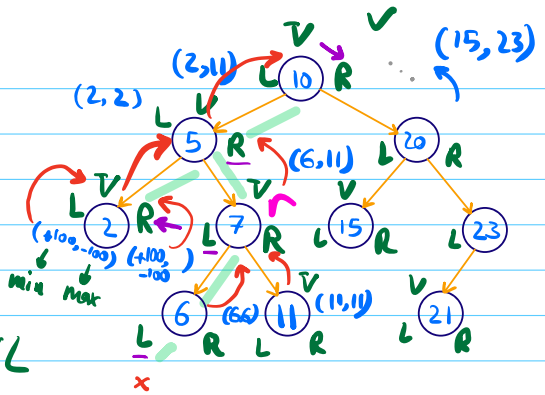
→ Pair ← R = checkBst(r.Right) // R

if (!(L.max <= r.data)) isBst = false

if (!(R.min > r.data)) isBst = false } // V

ret <MIN(L.min, R.min, r.data), MAX(L.max, R.max, r.data)>

}



break

P2 Recover sorted array : we swapped two elements in a sorted array. Recover the original sorted array

ex : {4, 10, 19, 14, 18, 12, 25, 28}

0 1 2 3 4 5 6 7

12 19

{ } → max, min → swap

ex : {4, 10, 19, 14, 20, 21, 25, 28}

P3 Recover original BST : value of two nodes are swapped

root = ...

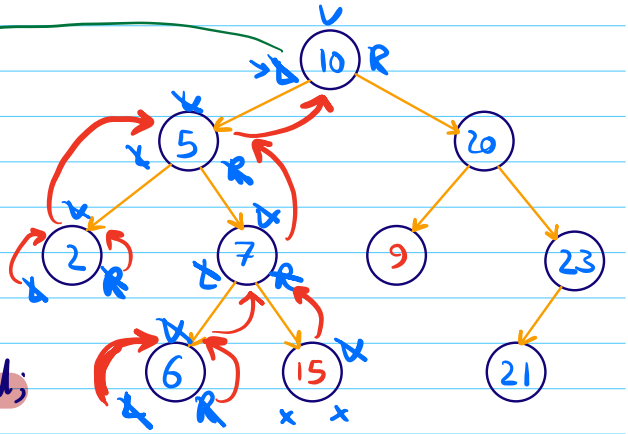
Tnode first, second; prev;

Main(){
  Inorder(root);
  swap_val(first, second)
}

a[i-1] < a[i]

Tnode Inorder(Tnode root){
  if(root == null) return null;
  Inorder(root.left);
  if(prev != null && prev.data > root.data){
    if(first == null){
      first = prev
      second = root
    }else{
      second = root
    }
  }
  prev = root // visit
  Inorder(root.right);
}

L V R



= first
= second

aut : 2 5 6 7 15 10

hypothetical

prev

f s

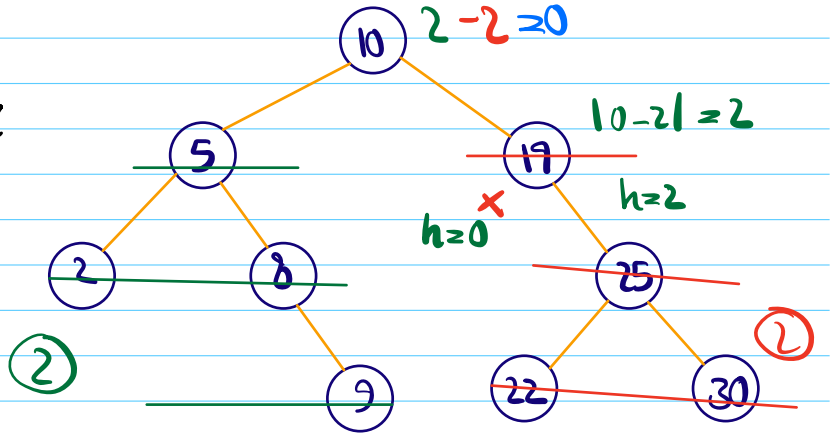
balanced BST:

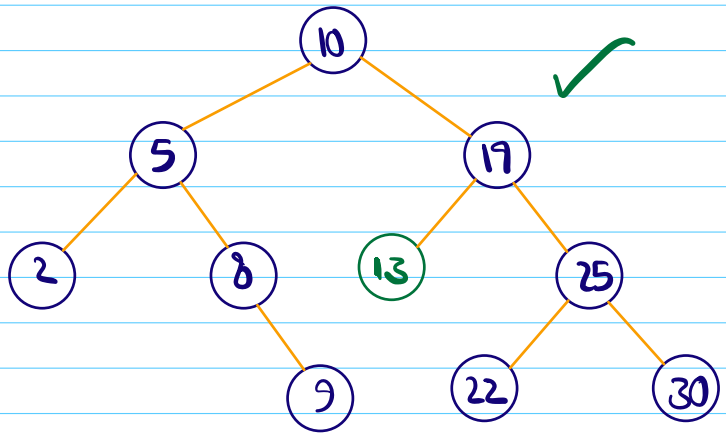
balanced tree $|h(\text{left sub tree}) - h(\text{right sub tree})| <= 1$

for all nodes

HW: how to check isBalanced?

assignment

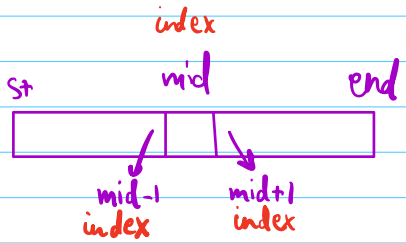


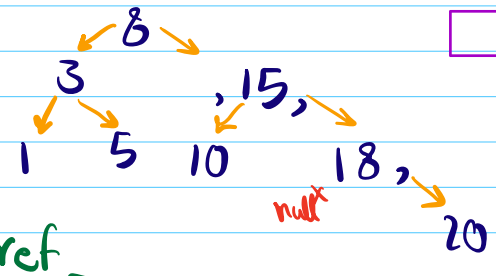


optional

P4 Construct a balanced BST from a sorted array.

ex a: { 0:1, 1:3, 2:5, 3:8, 4:10, 5:15, 6:18, 7:20 }





ref

TC: O(n)

SC: O(logn)

(10)

```
TNode buildBST(A[], st, end){
    if(st>end) ret null;
    mid=(st+end)/2
    r = new Tnode(A[mid])          // V
    r.left=buildBST(A[], st, mid-1)    // L   (n/2)
    r.right=buildBST(A[], mid+1, end)  // R   (n/2)
    ret r;
}
```

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 1$

TC: O(n)

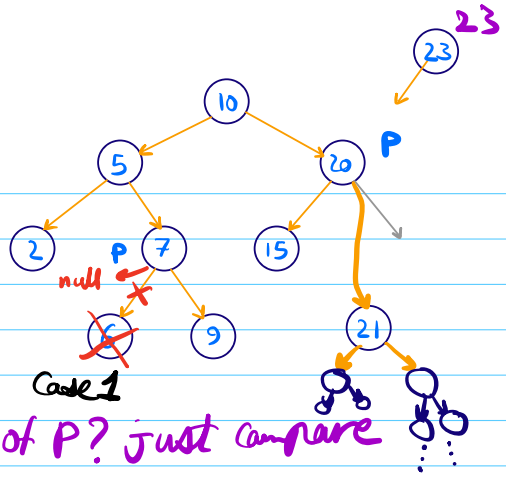
SC: O(log n)

Next page optional

bool lexist = cur.left != null

bool rexist = cur.right != null

delete node from BST



Case 1 node with data x is leaf

delete(6) step1 & search for x

P.left = null ← left or right of P? just compare

Case 2 node with data x has one child

delete(23)

step1 & search(23) → n

step 2. P.<left/right> = n.<left/right>

↓ ↓ Compare to 23

↓ ↓ which one is not null

Case 3 node with data x has two children

7 x

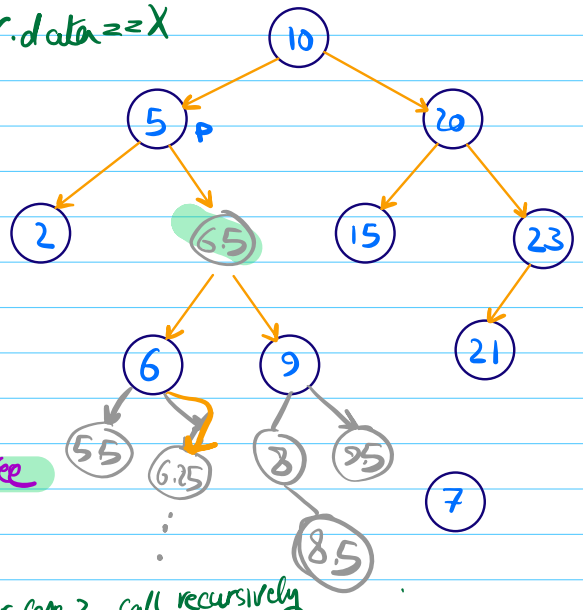
delete(7)

step1 & search(X) → cur.data == X

replace it with either

delete(20) { smallest from right subtree or largest from left subtree

step 2 → largest from left subtree



step3 temp (6.5)

delete(temp) // must be case 1 or case 2, call recursively

P.<left/right> = temp

temp.left = cur.left

temp.right = cur.right

// let cur be alone and die!