Data Structures & Algorithms for Problem Solving (CS1.304)

Lecture # 06

Avinash Sharma

Center for Visual Information Technology (CVIT),

IIIT Hyderabad

Binary Search Tree: Insert(x)

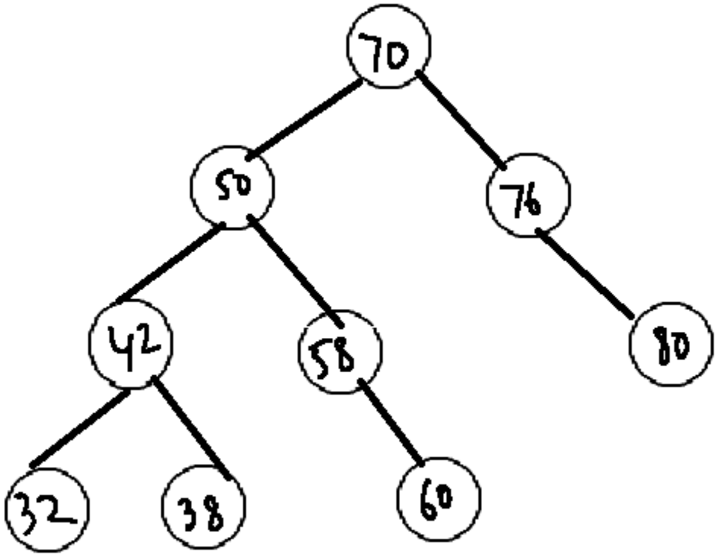
- Where should x be inserted?
- Should satisfy the search invariant.
  - So, if x is larger than the root, insert in the right subtree
  - if x is smaller than the root, insert in the left subtree.
- Repeat the above till we reach a deficient node.
- Can always add a new child to a deficient node.
- So, add node with value x as a child of some deficient node.

Insert(x)

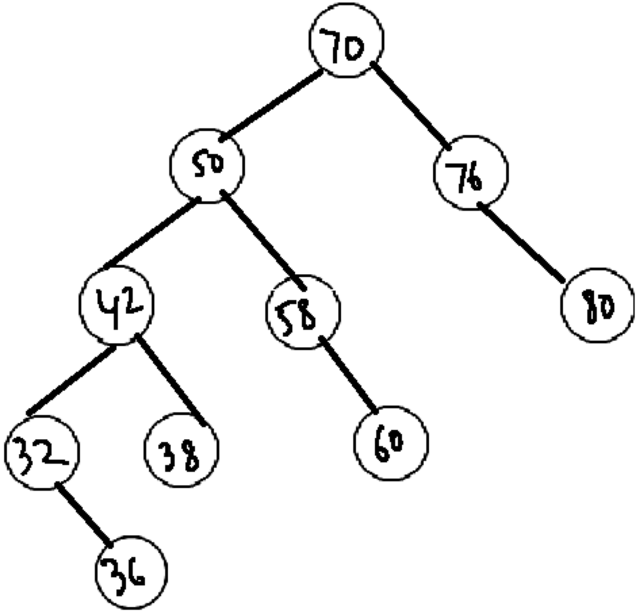
- Notice the analogy to Find(x)
- If x is not in the tree, Find(x) stops at a deficient node.
- Now, we are inserting x as a child of the deficient node last visited by Find(x).
- If the tree is presently empty, then x will be the new root.
- Let us consider a few examples.

Insert(x)

- Consider the tree shown and inserting 36.
- We travel the path 70 – 50 – 42 – 32.
- Since 32 is a leaf node, we stop at 32.



Insert(x)



- Now, 36 > 32. So 36 is inserted as a right child of 32.

Insert(x)

- Show the binary search tree obtained after inserting the following values in that order starting with an empty binary search tree.

  32, 28, 22, 38, 42, 51, 18, 37, 12

Insert(x)

```
Procedure insert(x)
begin
T’ = T;
if T’ =  NULL then
        T’ = new Node(x, Null, Null);
else
        while (1)
                if T’−> data > x then
                                If T'->left then T’ = T’−> left;
                                   else Add x as a left child of T'
                                        break;
                else
                                If T'->right then T’ = T’−> right;
                                  else Add x as a right child of T'
                                        break;
        end-while;
End.
```

Insert(x)

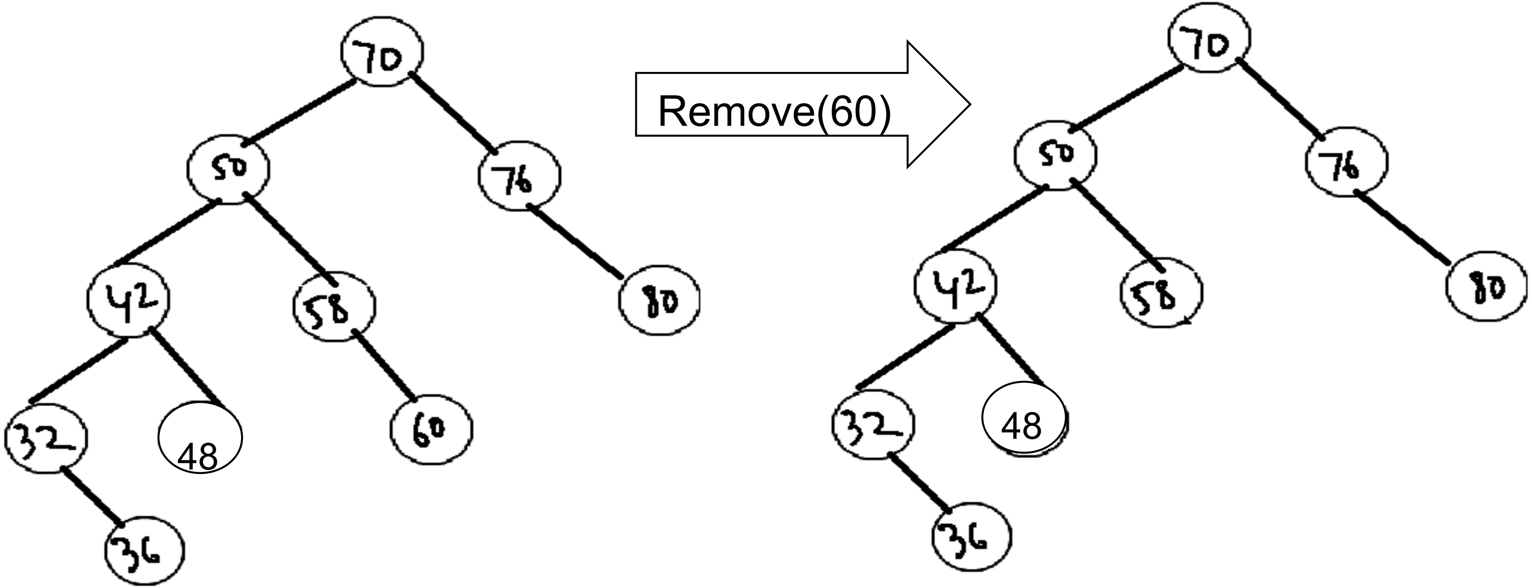
- New node always inserted as a leaf.
- To analyze the operation insert(x), consider the following.
  - Operation similar to an unsuccessful find operation.
  - After that, only O(1) operations to add x as a child.
- So, the time taken for insert is also proportional to the depth of the tree.

Binary Search Tree: Remove(x)

- Finally, the remove operation.
- Difficult compared to insert
  - new node inserted always as a leaf.
  - but can also delete a non-leaf node.
- We will consider several cases
  - when x is a leaf node
  - when x has only one child
  - when x has both children

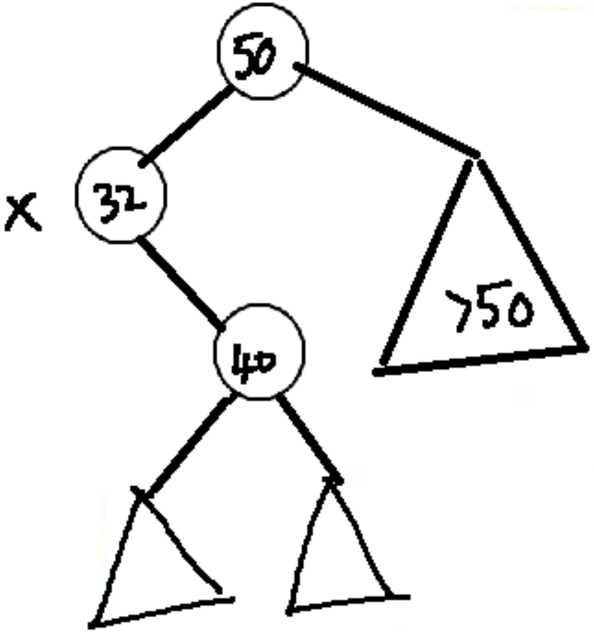
Remove(x)

- If x is a leaf node, then x can be removed easily.
  - parent(x) misses a child.

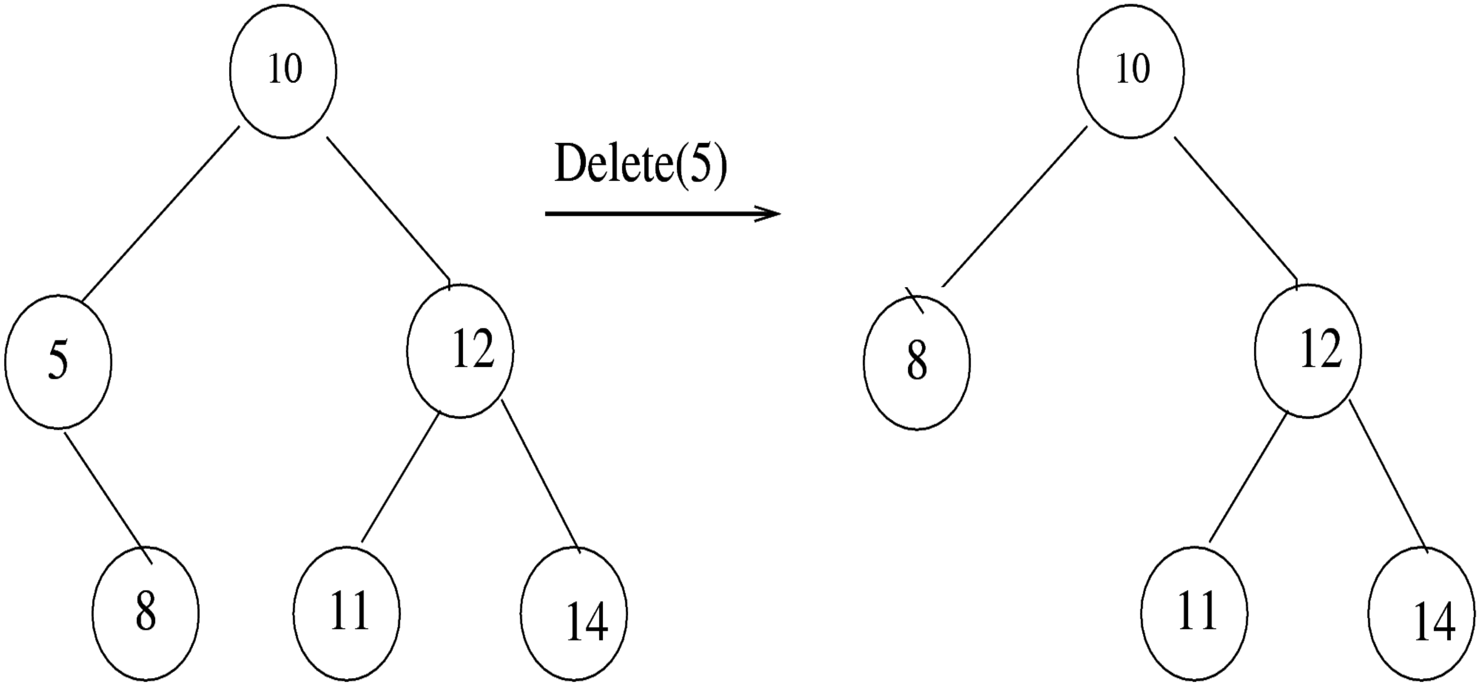


Remove(x)

- Suppose x has only one child, say right child.
- Say, x is a left child of its parent.
- Notice that x < parent(x) and child(x) > x, and also child(x) < parent(x).
- So, child(x) can be a left child of parent(x), instead of x.
- In essence, promote child(x) as a child of parent(x).



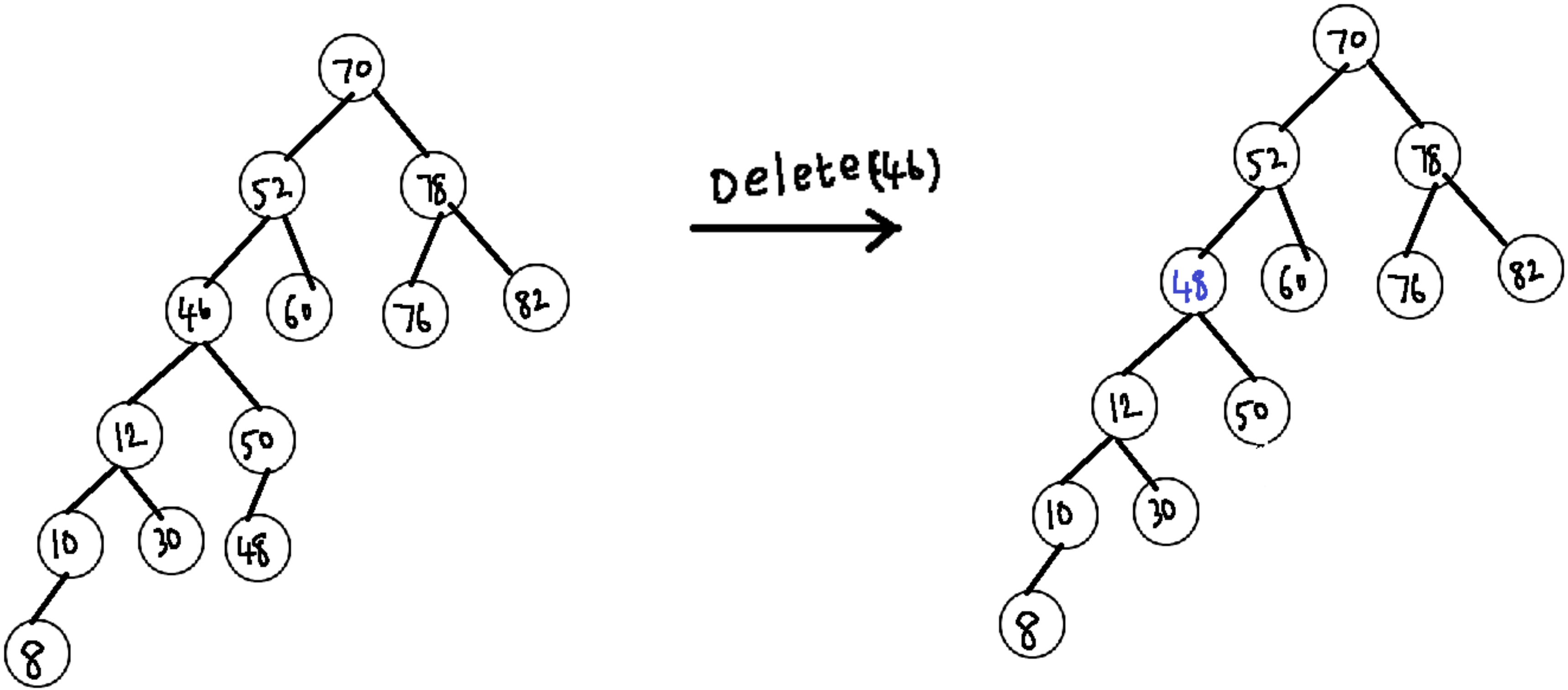
Remove(x)



Remove(x)

- One possibility is to consider the maximum valued node in the left subtree of x.
- Equivalently, can also consider the node with the minimum value in the right subtree of x.
- Notice that both these replacement nodes are deficient nodes. Hence easy to remove them.
- In a way, to remove x, we physically remove a deficient node.

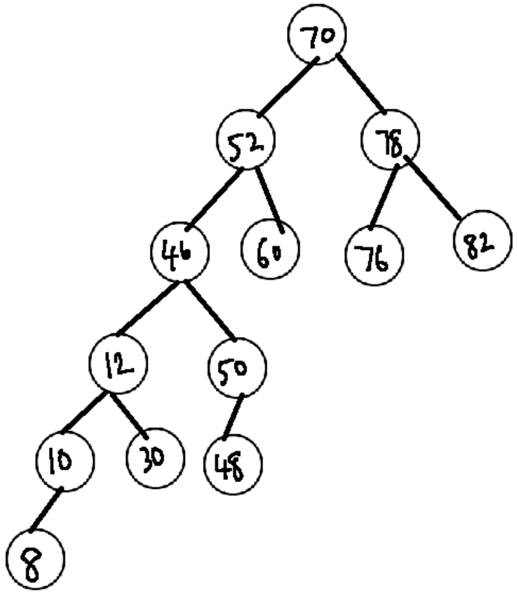
Remove(x)



70
52
78
46
60
76
82
12
50
10
30
48
8
Delete(46)
70
52
78
48
60
76
82
12
50
10
30
8

Remove(x)

- From the tree shown below, delete nodes 30, 78, and 12 in that order.



Remove(x)

```
Procedure Delete(x, T)
begin
if T = NULL then return NULL;
T′ = Find(x);
if T′ has only one child then
        adjust the parent of the remaining child;
else
        T′′ = FindMin(T′−> right);
        Remove T′′ from the tree;
        T′−> value = T′′−> value;
End-if
End.
```

Remove(x)

- Time taken by the remove() operation also proportional to the depth of the tree.

Depth of a Binary Search Tree

- Imagine that each internal node has exactly two children.
- A depth of $\log_2 n$ is the best possible.
- So the depth can be between $\log_2 n$ and n.
- What is the average depth?

Average Depth

- A good notion as most operations take time proportional on the depth of the binary search tree.
- Still, not a satisfactory measure as we wanted worst-case performance bounds.

Average Depth

- Let us analyze the average depth of a binary search tree.
- This average is on what?
  - Assume that all subtree sizes are equally likely.
- Under the above assumption, let us show that the average depth of a binary search tree is O(log n).

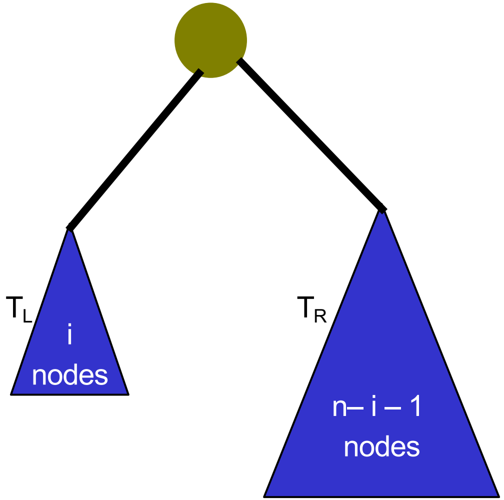
Average Depth

- Internal path length : The sum of the depths of all nodes in a tree.
- Let D(n) to be the internal path length of some binary search tree of n nodes.
  - $D(n) = {}^{n}\sum_{i=1} d(i)$, where d(i) is the depth of node i.
- Note that D(1) = 0.

Average Depth

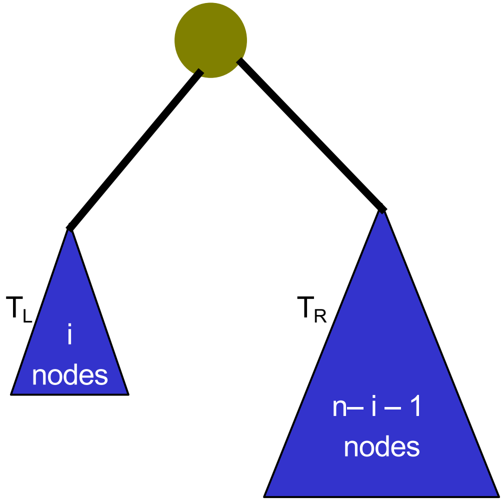
- In a tree with n nodes, there is one root node and a left subtree of i nodes and a right subtree of $n-i-1$ nodes.
- Using our notation, D(i) is the internal path length of the left subtree.
- D(n-i-1) is the internal path length of the right subtree.

Average Depth



- Further, if now these trees are attached to the root
  - the depth of each node in T_L and T_R increases by 1.

Average Depth



- So, D(N) = D(i) + D(n-i-1) + n-1

Solving the Recurrence Relation

- If all subtree sizes are equally likely then D(i) is the average over all subtree sizes.
  - That is, i ranges over 0 to n – 1.
  - Can hence see that $D(i) = (1/n) \sum_{j=0}^{n-1} D(j)$
- Similar is the case with the right subtree.
  - So, $D(n - i - 1) = (1/n) \sum_{j=0}^{n-1} D(j)$

Solving the Recurrence Relation

- The recurrence relation simplifies to

$$D(n) = (2/n) \left({}^{n-1}\sum_{j=0} D(j) \right) + n - 1$$

- Can be solved using known techniques as follows.

Solving the Recurrence Relation

- Consider $D(n) = (2/n) \left(\sum_{j=0}^{n-1} D(j) \right) + n - 1$.
- Rearrange as

  $$n\, D(n) = 2 \left(\sum_{j=0}^{n-1} D(j) \right) + n(n - 1)$$

- Now, write the equation with n-1 replacing n.

  $$(n-1)\, D(n-1) = 2 \left(\sum_{j=0}^{n-2} D(j) \right) + (n-1)(n - 2)$$

- Subtract the two equations to get:

  $$nD(n) - (n-1)\, D(n-1) = 2\, D(n - 1) + 2(n - 1)$$

- Rearrange as:

  $$nD(n) = (n+1)\, D(n-1) + 2(n-1)$$

Solving the Recurrence Relation

- Consider

$$nD(n) = (n+1)\ D(n-1) + 2(n-1).$$

- Try another telescoping but after some adjustment.

  Divide the whole equation by n(n+1) to get:

$$D(n) / (n+1) = D(n-1) / n + 2(n-1) / (n(n+1))$$

- Now, write the above equation for n, n-1, …, all the way to n = 2.

$$D(n-1) / n = D(n-2) / (n-1) + 2(n-2) / ((n-1)n)$$

$$D(n-2) / (n-1) = D(n-3) / (n-2) + 2(n-3) / ((n-2)(n-1))$$

$$\vdots$$

$$D(2) / 3 = D(1) / 2 + 2 /2*3$$

Solving the Recurrence Relation

- Summing these equation, we should get

$$D(n) \,/\, (n+1) = D(1)/2 + 2c \,{}^{n}\Sigma_{j=2}\, 1/j$$

where $c=(j-1)/(j+1)$ is close to 1

- Now, notice that the summation on the right is about $H(n) = O(\log n)$.
- Therefore, $D(n) = O(n\log n)$.

Solving the Recurrence Relation

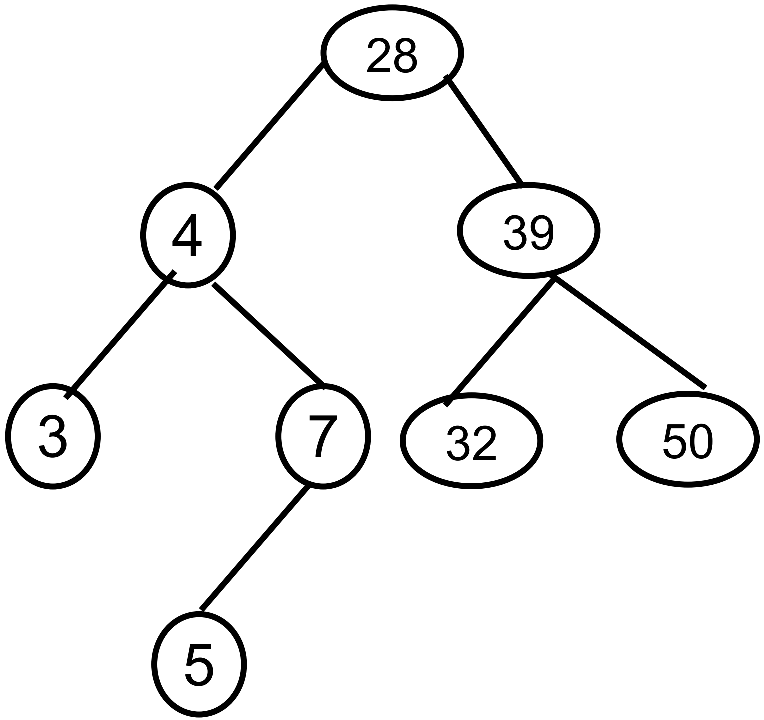
- The solution to D(n) is D(n) = O(n log n).
- How is D(n) related to the average depth of a binary search tree.
  - There are N paths in any binary search tree from the root.
  - So the average internal path length is O(log n).
- Does this mean that each operation has an average O(log n) runtime.
  - Not quite.

Average Runtime

- Now, remove() operation may introduce a skew.
- Replacement node can skew left or right subtree.
- Can pick the replacement node from the left or the right subtree uniformly at random.
  - Still not known to help.
- So, at best we can be satisfied with an average O(log n) runtime in most cases.
- Need techniques to restrict the height of the binary search tree.

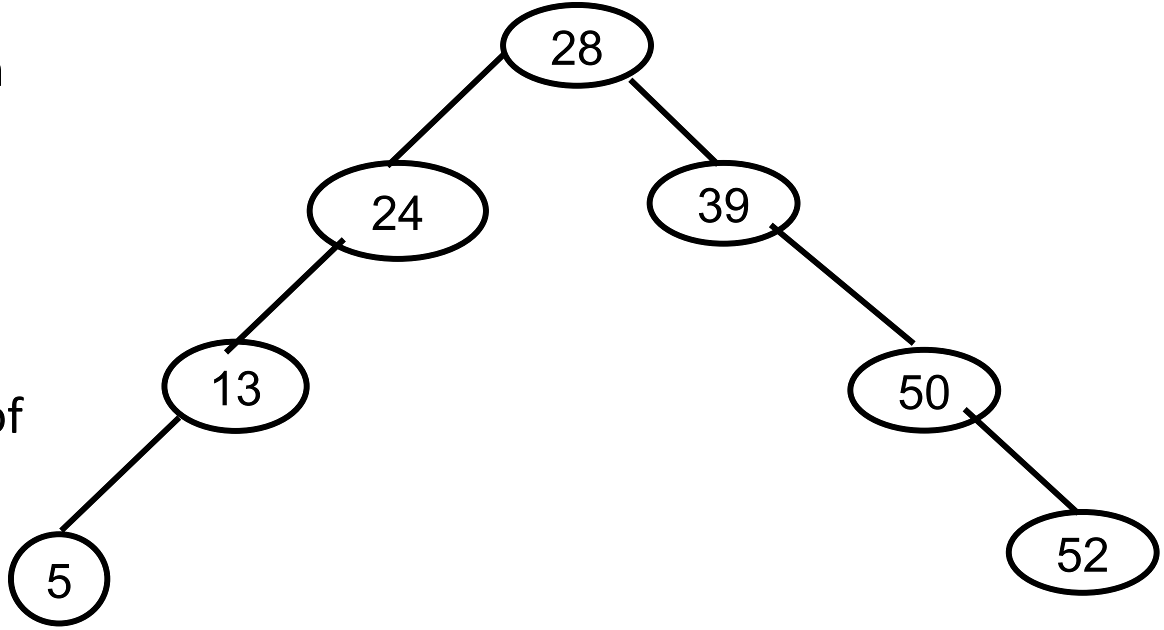
Towards Height Balanced Trees

- How can we control the height of a binary search tree?
  - should still maintain the search invariant
  - additional invariants required.
- What if the root of every subtree is the median of the elements in that subtree?
  - Difficult to maintain as median can change due to insertion/deletion.



Towards Height Balanced Trees

- Attempt 1: Would it suffice if we say that the root has both a left and a right subtree of equal height?
- Still, the depth of the tree is not O(log n).
- In the this tree, irrespective of values at the nodes, the root has left and right subtrees of equal height.



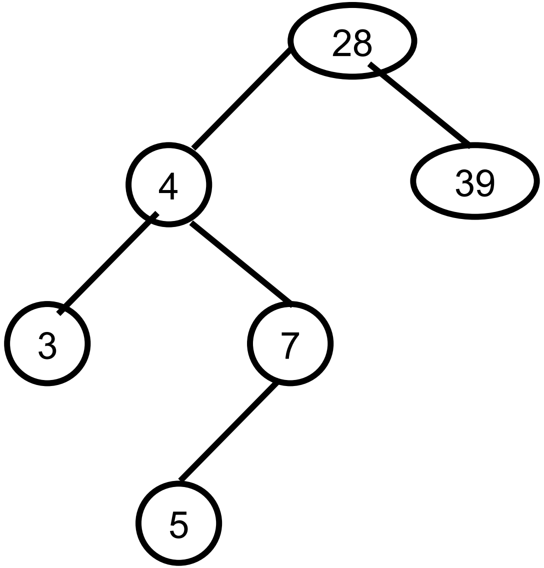
Towards Height Balanced Trees

- Our condition is too simple. Need more strict invariants.
- Consider the following modification.
- Attempt 2: For every node, its left and right subtrees should be of the same height.
- The condition ensures good balance, but
- The above condition may force us to keep the median as the root of every subtree.
  - Fairly difficult to maintain.

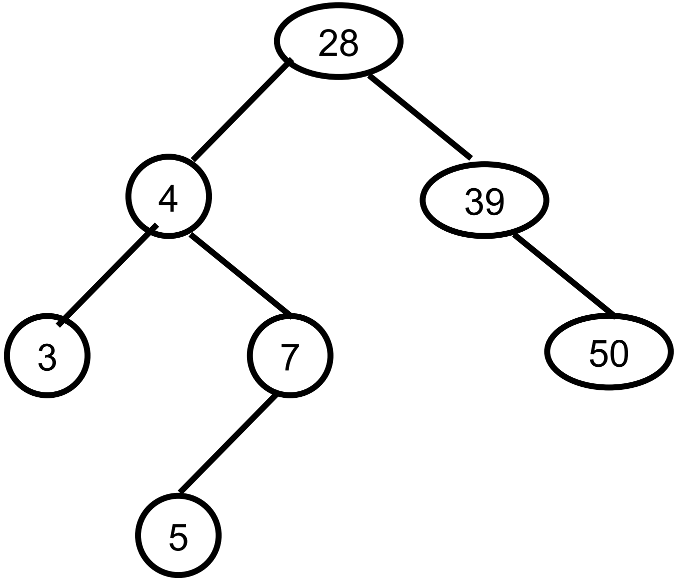
Towards Height Balanced Trees

- A small relaxation to Condition 2 works suprisingly well.
- The relaxed condition, Condition 3, is stated below.
- Height Invariant: For every node in the tree, its left and the right subtrees can have heights that differ by at most 1.

Towards Height Balanced Trees



Not a Height Balanced Tree



Height Balanced Tree

Thank You