Another Application – Dictionary Operations

- Consider designing a data structure for primarily three operations:
  - insert,
  - delete, and
  - search.
- Why not use a hash table?
  - a hash table can only give an average O(1) performance
  - Need worst case performance guarantees.

Dictionary Operations

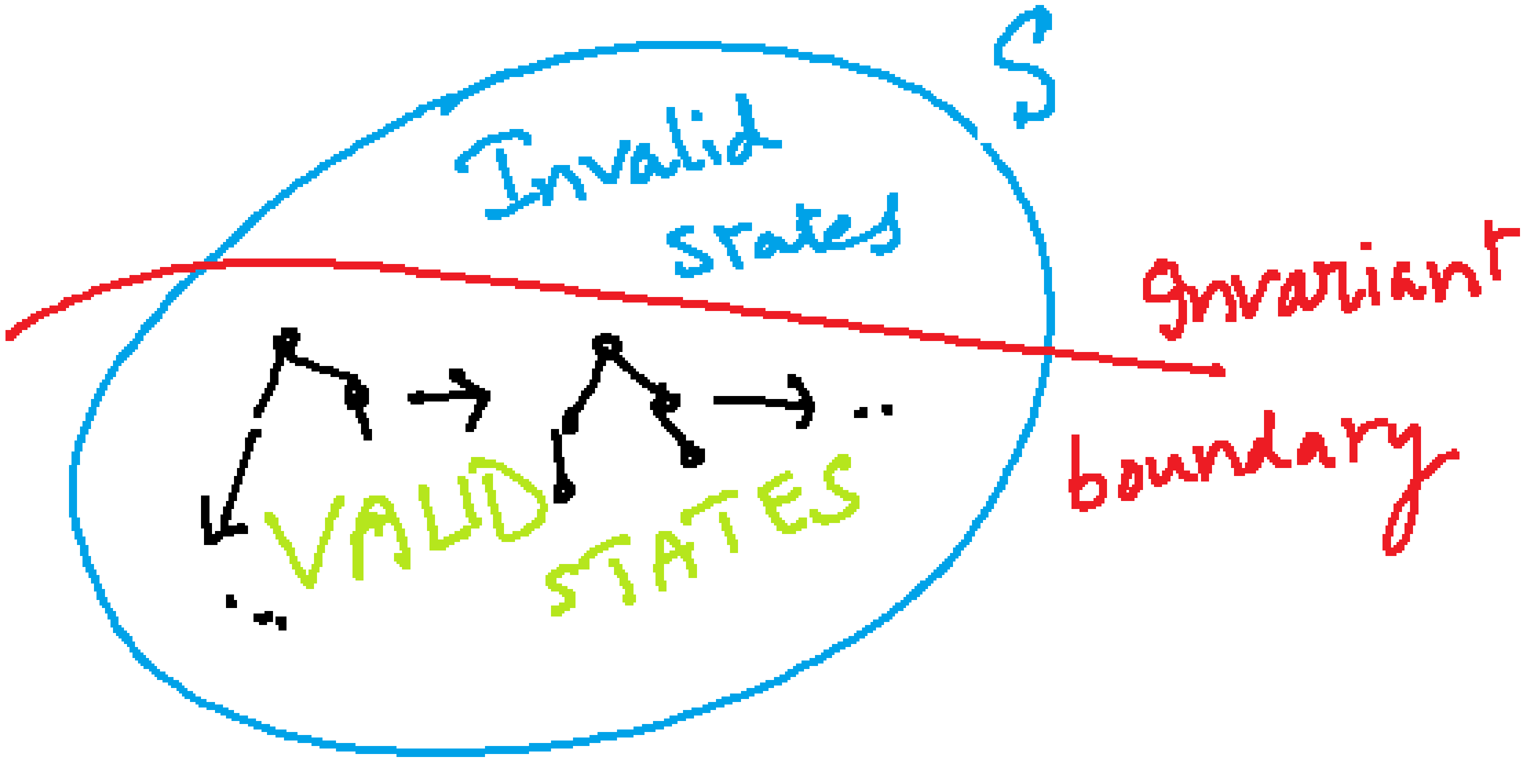
- Further extend the repertoire of operations to standard dictionary operations also such as findMin and findMax.
  - Hash tables may not help in findMin and findMax.
- Specifically, our data structure shall support the following operations.
  - Create()
  - Insert()
  - FindMin()
  - FindMax()
  - Delete(), and
  - Find()

Binary Search Tree

- Our data structure shall be a binary tree with a few modifications.
- Assume that the data is integer valued for now.
- Search Invariant:

  The data at the root of any binary search tree is larger than all elements in the left subtree and is smaller than all elements in the right subtree.

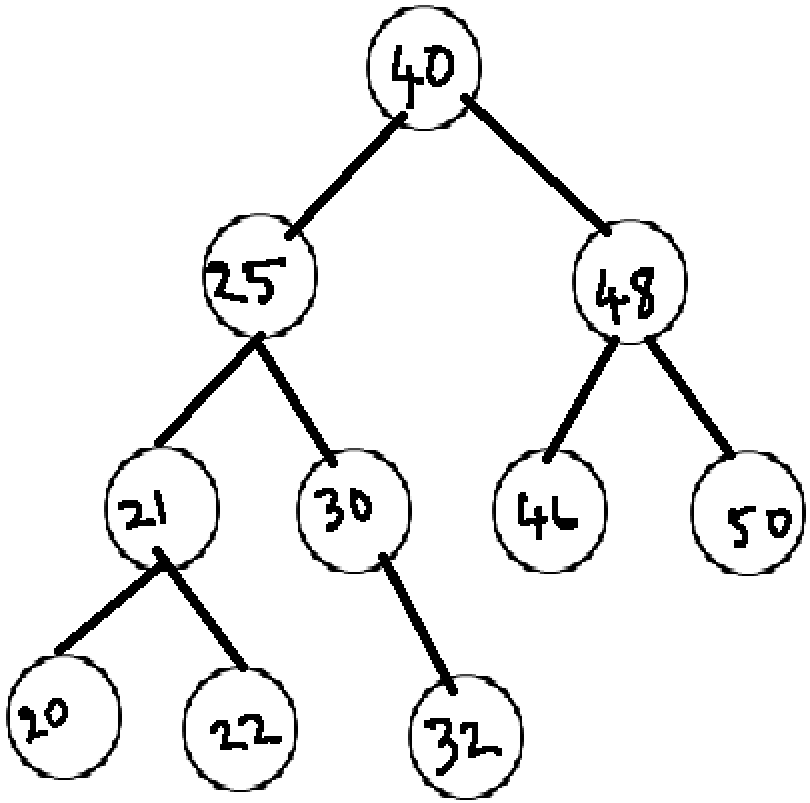
How to Understand Invariants



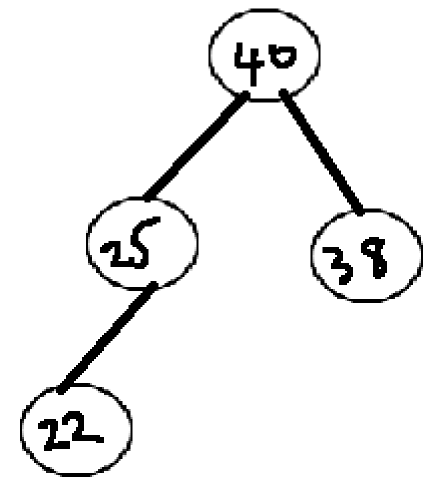
Binary Search Tree

- The search invariant has to be maintained at all times, after any operation.
- This invariant can be used to design efficient operations, and
- Also obtain bounds on the runtime of the operations.

Binary Search Tree – Example



A binary search tree



Not a binary search tree

Operations

- Let us start with the operation Find(x).
- We are given a binary search tree T.
- Answer YES if x is in T, and answer NO otherwise.
- Throughout, let us call a node deficient, if it misses at least one child.
  - So a leaf node is also deficient.
  - So is an internal node with only one child.

Find(x)

- Let us compare x with the data at the root of T.
- There are three possibilities
  - x = T->data : Answer YES. Easy case.
  - x < T->data : Where can x be if it is in T? Left subtree
  - x > T->data : Where can x be if it is in T? Right subtree
- So, continue search in the left/right subtree.
- When to stop?

Find(x)

- Let us compare x with the data at the root of T.
- There are three possibilities
  - x = T->data : Answer YES. Easy case.
  - x < T->data : Where can x be if it is in T? Left subtree
  - x > T->data : Where can x be if it is in T? Right subtree
- So, continue search in the left/right subtree.
- When to stop?
  - Successful search stops when we find x.

Find(x)

- Let us compare x with the data at the root of T.
- There are three possibilities
  - x = T->data : Answer YES. Easy case.
  - x < T->data : Where can x be if it is in T? Left subtree
  - x > T->data : Where can x be if it is in T? Right subtree
- So, continue search in the left/right subtree.
- When to stop?
  - Successful search stops when we find x.
  - Unsuccessful search stops when we reach a deficient node without finding x.

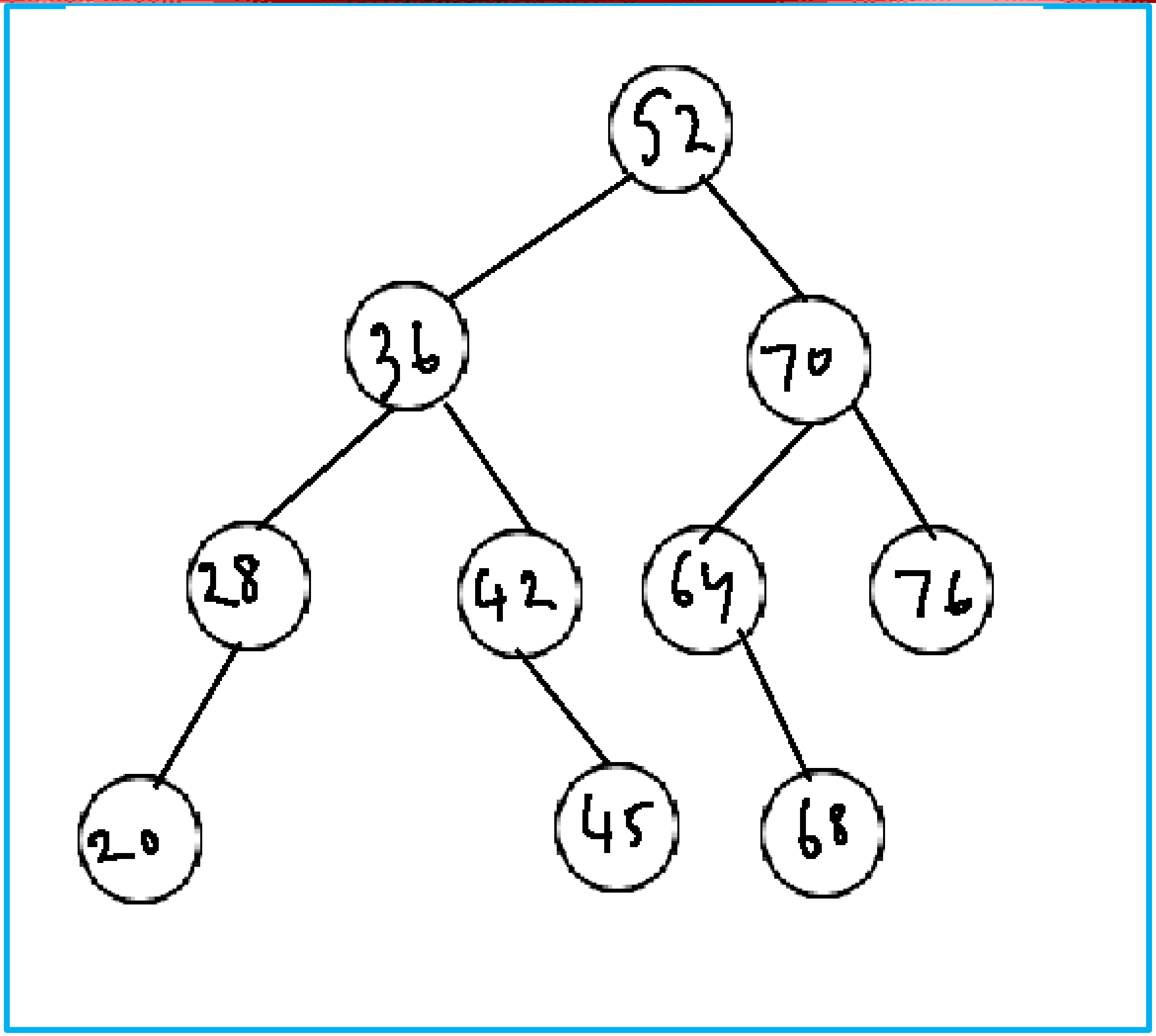
Find(x)

- Notice the similarity to binary search.
- In both cases, we continue search in a subset of the data.
  - In the case of binary search the subset size is exactly half the size of the current set.
  - Is that so in the case of a binary search tree also?
  - May not always be true.

Find(x)

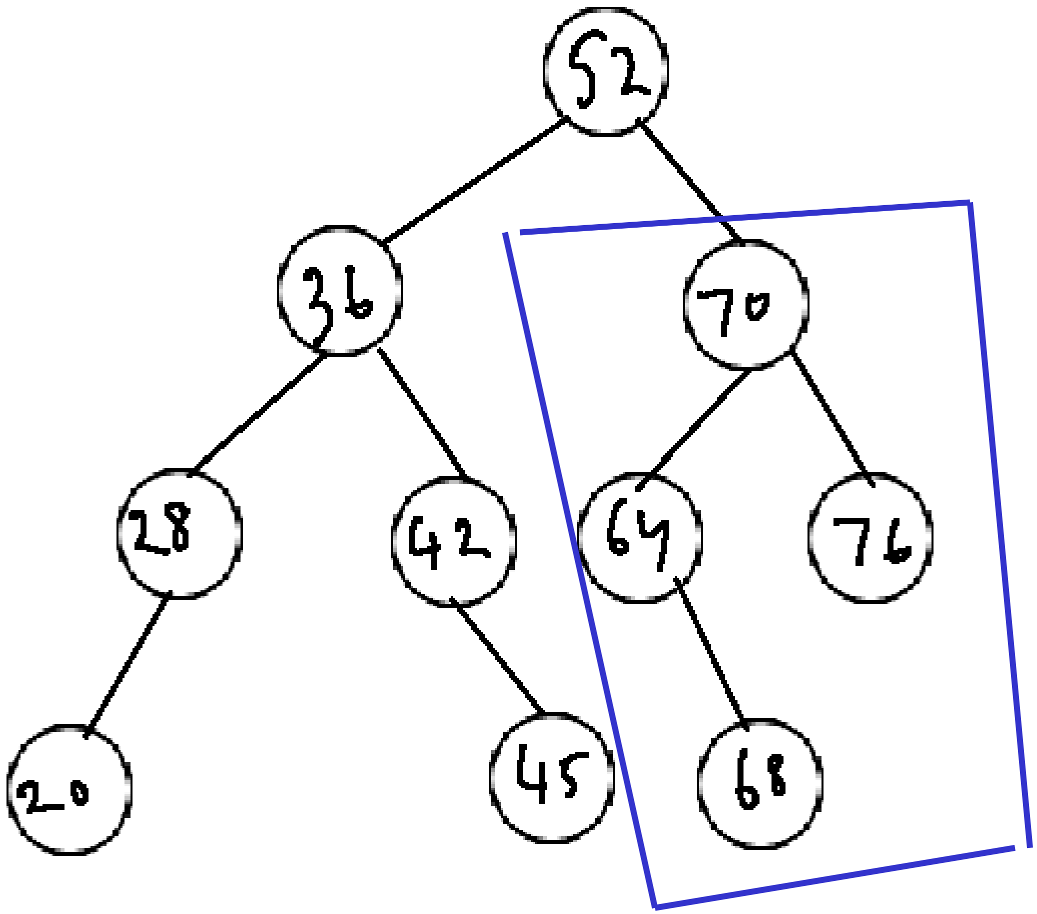
- How to analyze the runtime?
- Number of comparisons is a good metric.
- Notice that for a successful or an unsuccessful search, the worst case number of comparisons is equal to the height of the tree.
- What is the height of a binary search tree?
  - We'll postpone this question for now.

Example – Find(x)



- Search for 64.
- Since 52 < 64, we search in the right subtree.

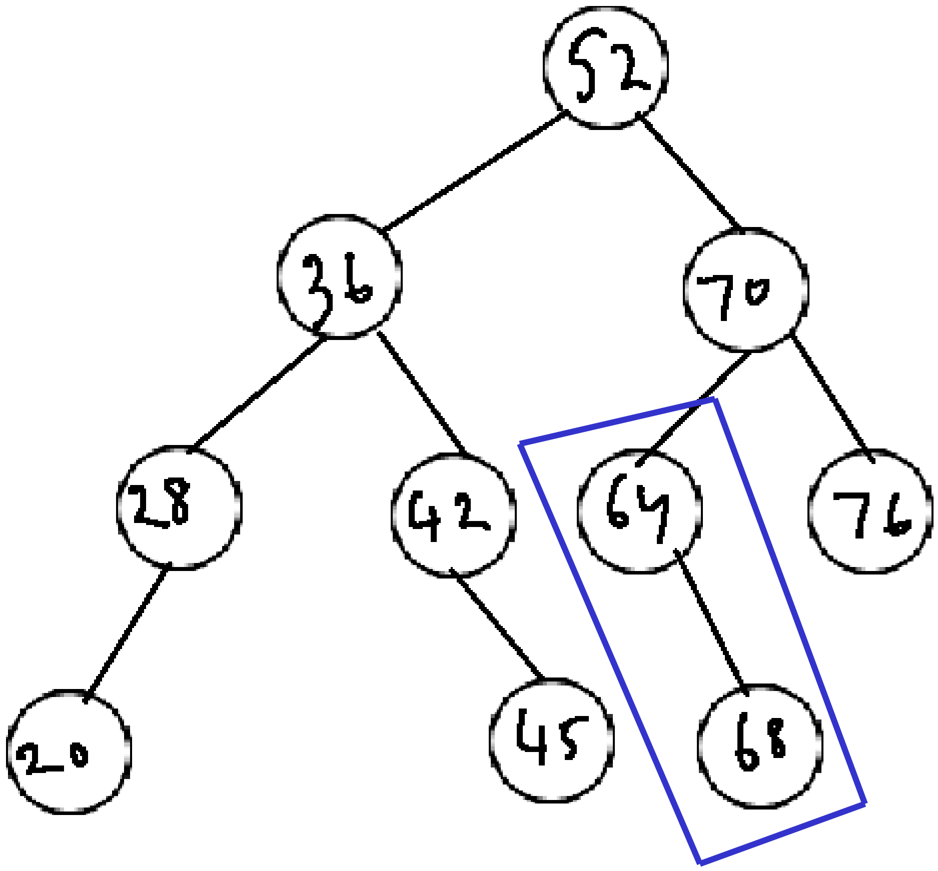
Example – Find(x)



- Search for 68
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.

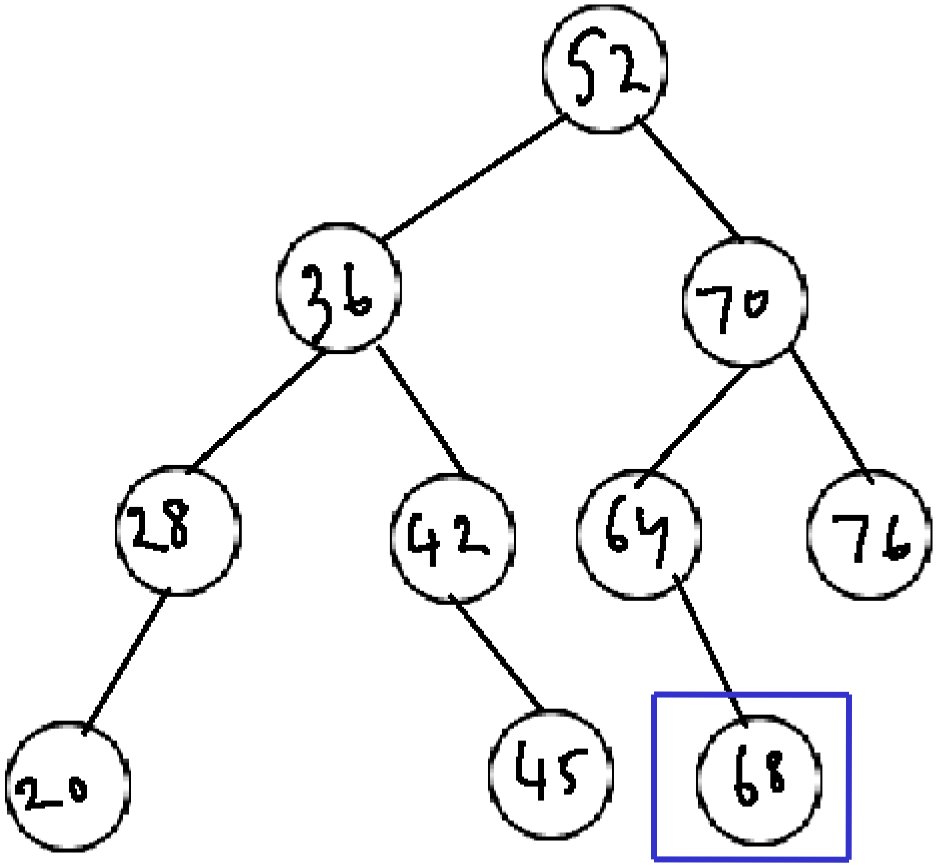
Example – Find(x)

- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.
- Since 64 < 65, again search in the right subtree.

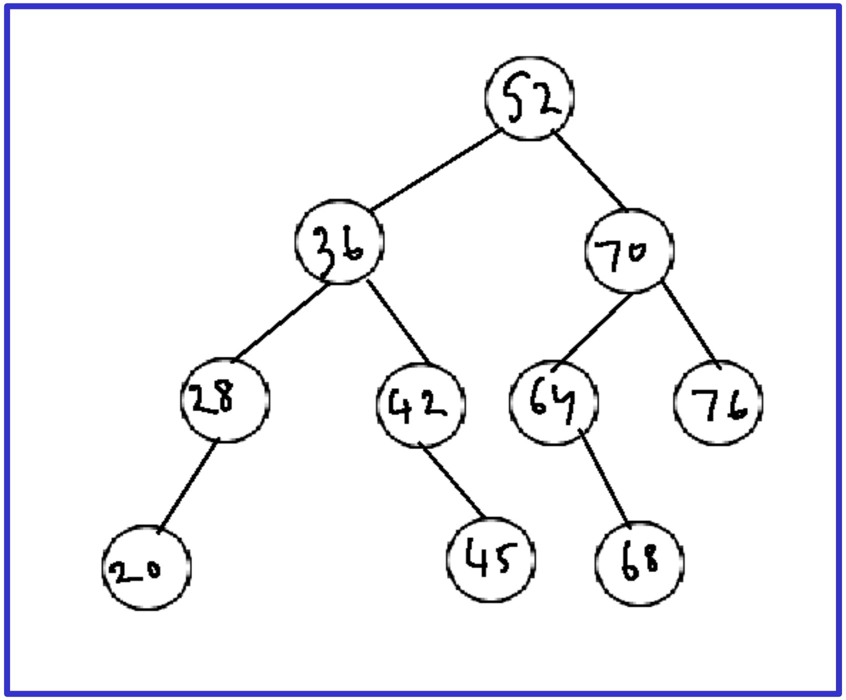


Example – Find(x)

- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.
- Since 64 < 68, again search in the right subtree.
- Finally, find 68 as a leaf node.

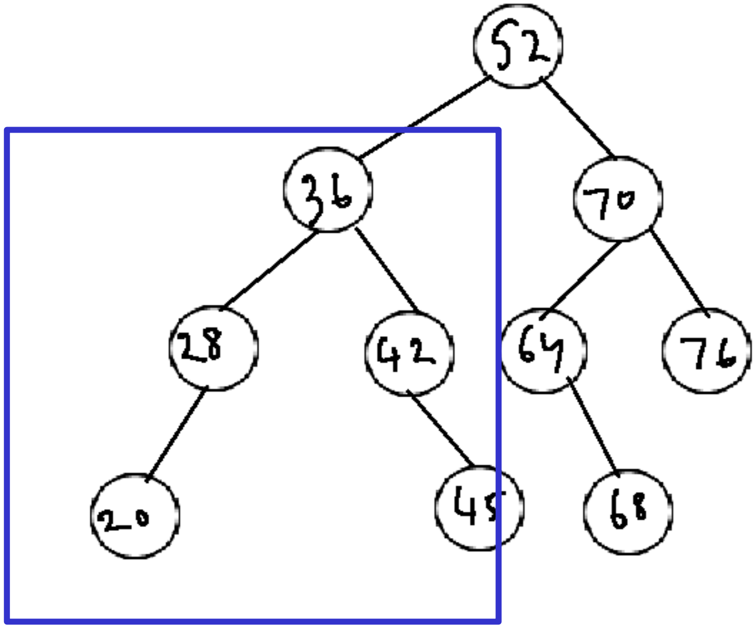


Example -- Find(x)



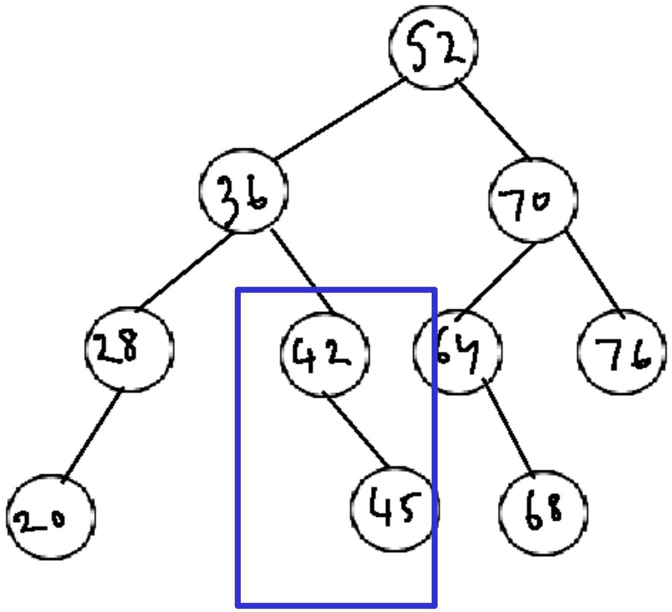
- Consider the same tree and Find(48).
- Since 52 > 48, we search in the left subtree.

Example -- Find(x)



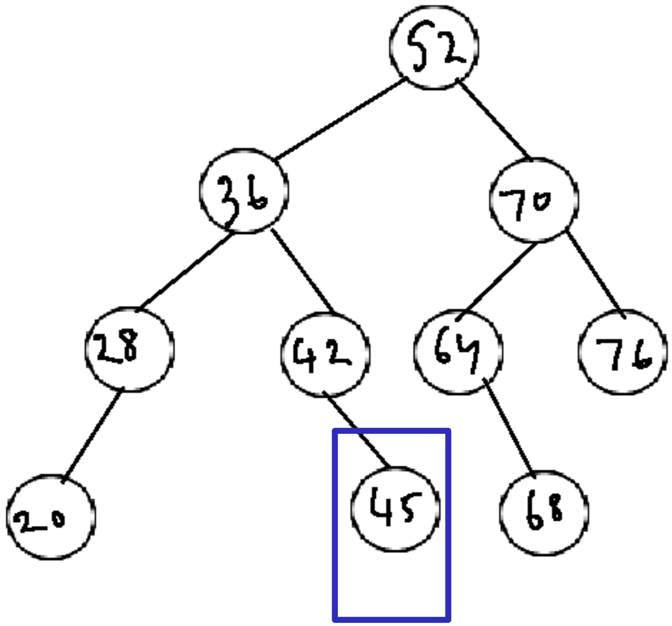
- Consider the same tree and Find(48).
- Since 52 > 48, we search in the left subtree.

Example -- Find(x)



- Consider the same tree and Find(48).
- Since 52 > 48, we search in the left subtree.
- Since 36 < 48, search in the right subtree.

Example – Find(x)



- Consider the same tree and Find(48).
- Since 52 > 48, we search in the left subtree.
- Since 36 < 48, search in the right subtree.
- Since 42 < 48, search in the right subtree.
- finally, 45 < 48, but no right subtree. So declare NOT FOUND.

Find(x) Pseudocode

```
procedure Find(x, T)
begin
    if T == NULL return NO;
    if T->data == x return YES;
    else if T->data < x
        return Find(x, T->right);
    else
        return Find(x, T->left);
end
```

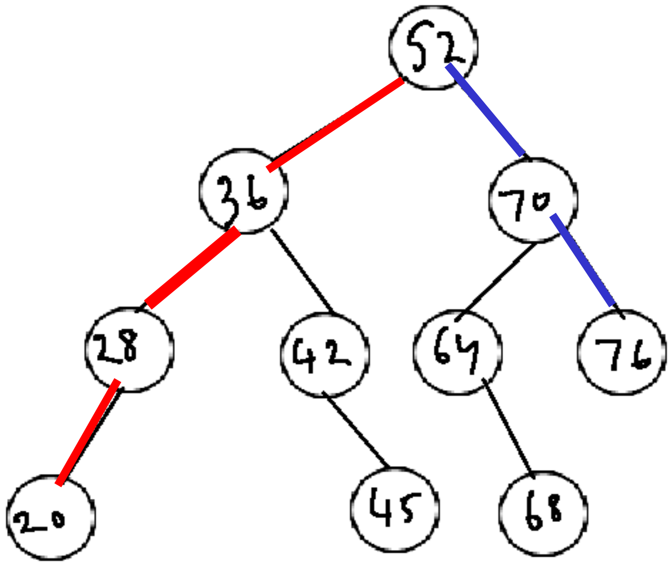
Observation on Find(x)

- Travel along only one path of the tree starting from the root.
- Hence, important to minimize the length of the longest path.
  - This is the depth/height of the tree.

Operation FindMin and FindMax

- Consider FindMin.
- Where is the smallest element in a binary search tree?
- Recall that values in the left subtree are smaller than the root, at every node.
- So, we should travel leftward.
  - stop when we reach a leaf or
  - a node with no left child.
  - Essentially, a deficient node missing a left child.
- FindMax is similar. How should we travel?

Operation FindMin and FindMax



- On the above tree, findMin will travese the path shown in red.
- FindMax will travel the path shown in blue.

Operation FindMin and FindMax

```
procedure FindMin(T)
begin
    if T = NULL return null;
    if T−> left = NULL return T;
    return FindMin(T−>left);
end
```

- Both these operations also traverse one path of the tree.
- Hence, the time taken is proportional to the depth of the tree.
- Notice how the depth of the tree is important to these operations also.

Insert(x)

- Let us now study how to insert an element into an existing binary tree.
- Assume for simplicity that no duplicate values are inserted.

Insert(x)

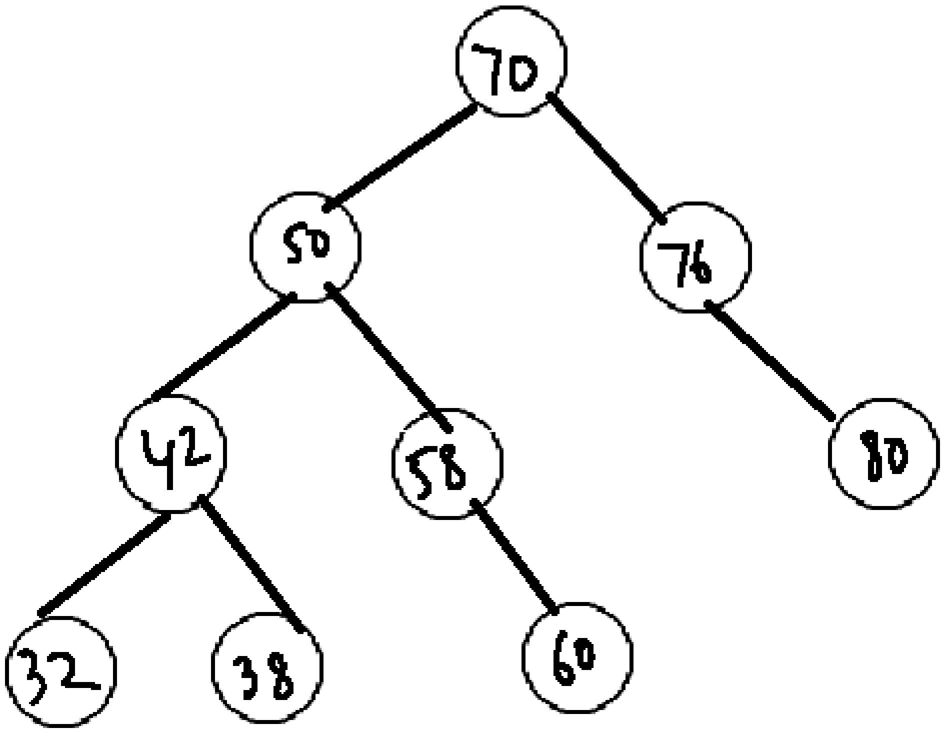
- Where should x be inserted?
- Should satisfy the search invariant.
  - So, if x is larger than the root, insert in the right subtree
  - if x is smaller than the root, insert in the left subtree.
- Repeat the above till we reach a deficient node.
- Can always add a new child to a deficient node.
-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

Insert(x)

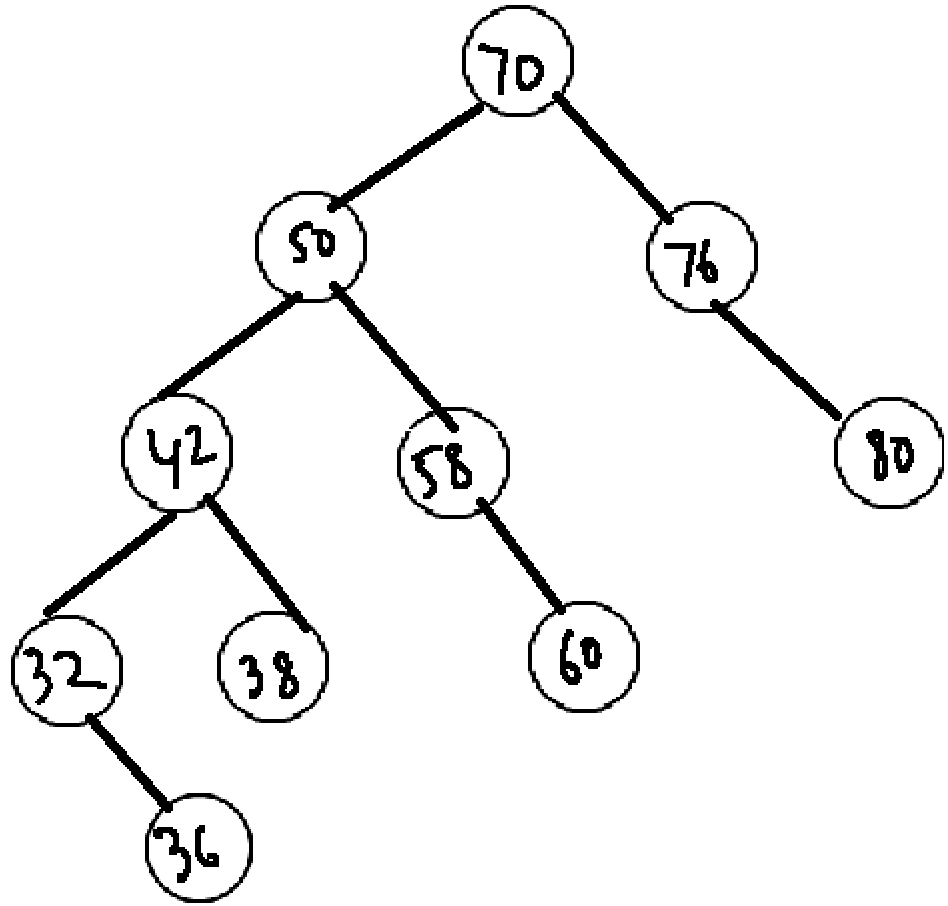
- Notice the analogy to Find(x)
- If x is not in the tree, Find(x) stops at a deficient node.
- Now, we are inserting x as a child of the deficient node last visited by Find(x).
- If the tree is presently empty, then x will be the new root.
- Let us consider a few examples.

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.
- The resulting tree is shown in the picture.

Practice Problem

- Show the binary search tree obtained by inserting

  32, 47, 51, 29, 22, 42, 64, 17, 45, 40

  in that order into an initially empty binary searh tree.

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.

Duplicates?

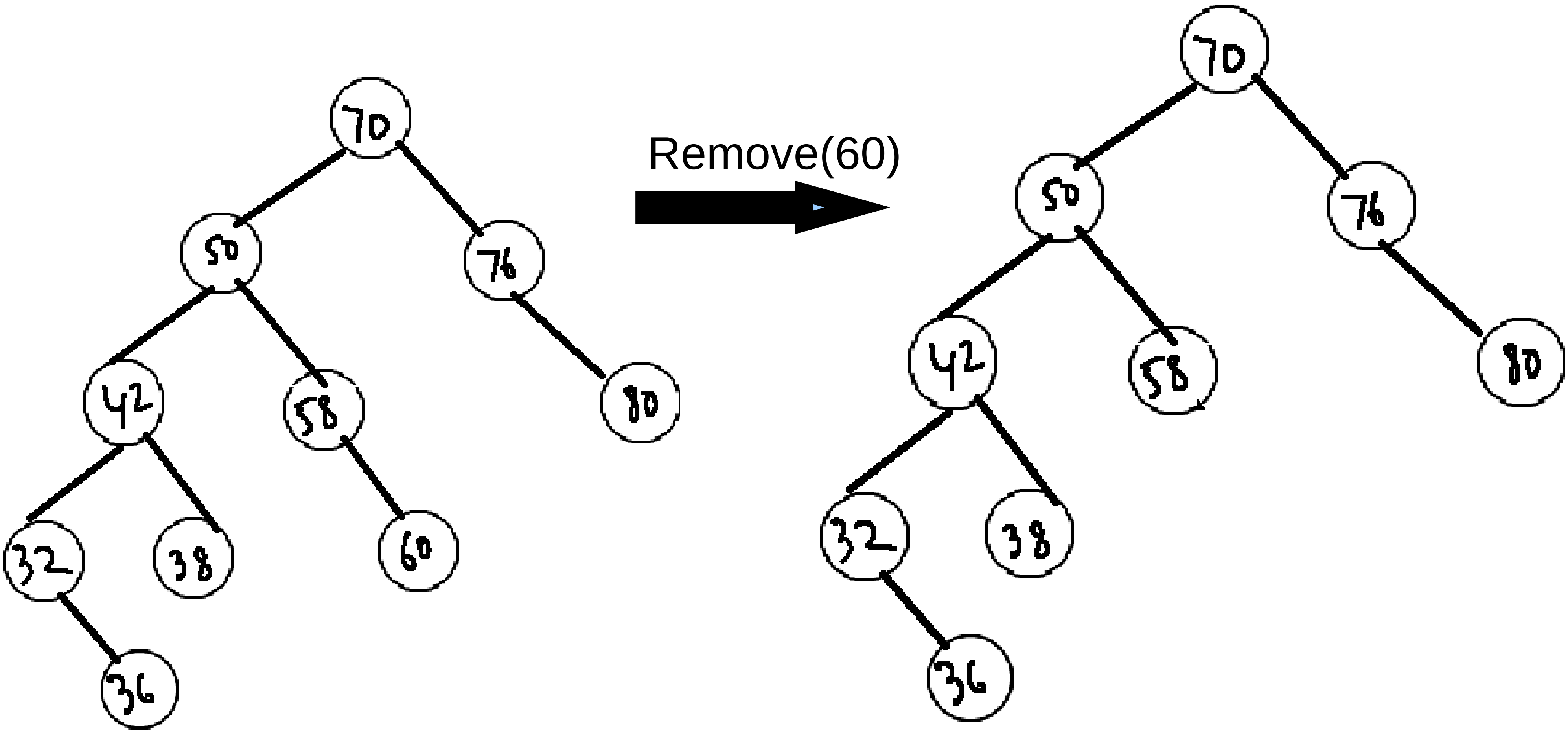
- To handle duplicates, two options
  - report an error message
  - to keep track of the number of elements with the same value

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

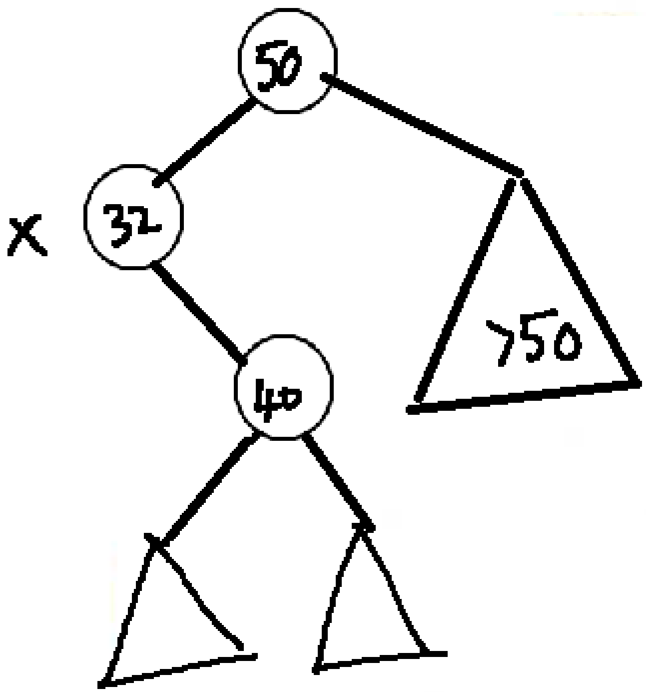
Remove(x)

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

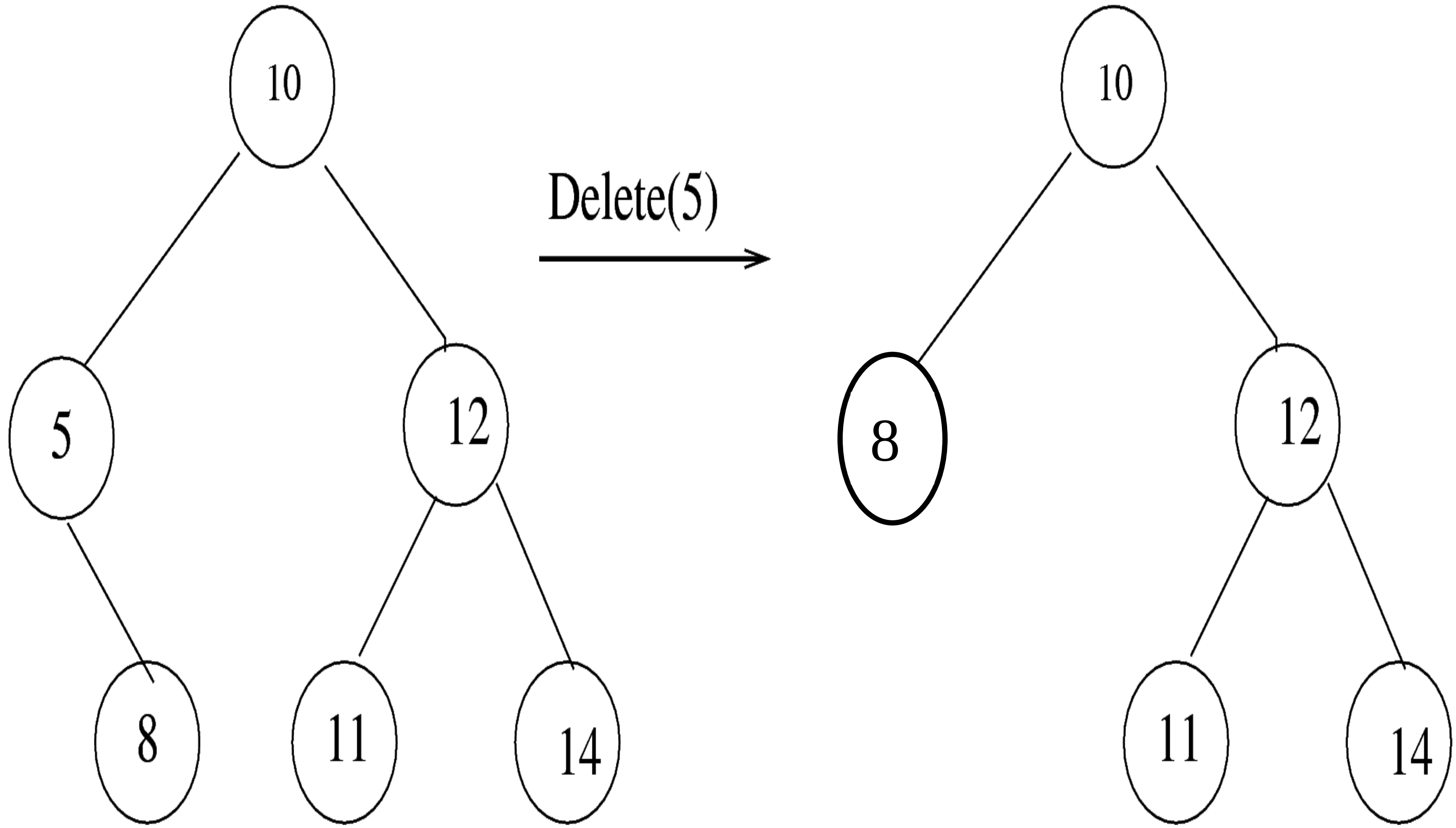


Remove(x)

- Suppose x has only one child, say right child.
- Say, x is a left child of its parent.
- Notice that 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)



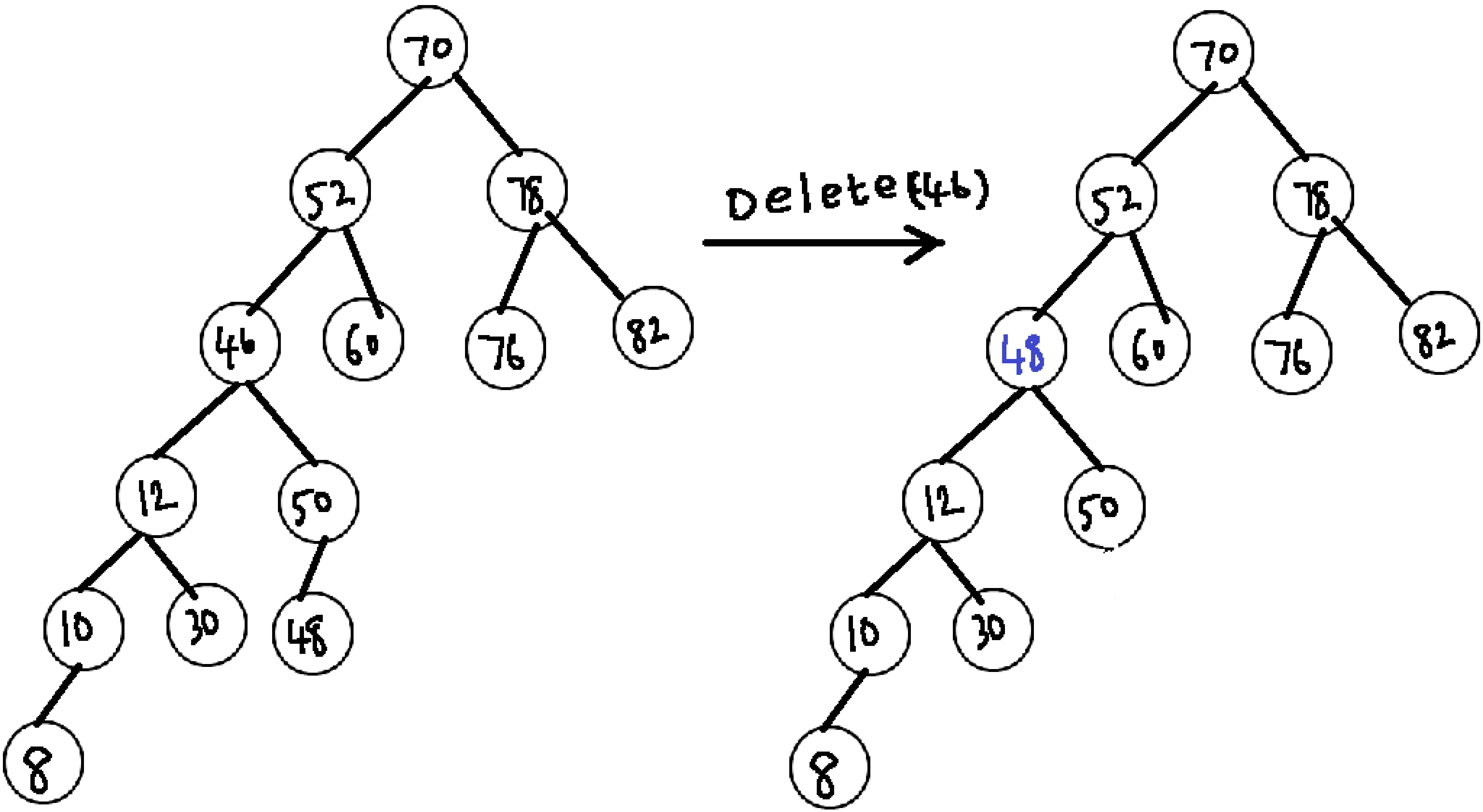
Remove(x) – The Difficult Case

- x has both children.
- Cannot promote any one child of x to be child of parent(x).
- But, what is a good value to replace x?
- Notice that, the replacement should satisfy the search invariant.
- So, the replacement node should have a value more than all the left subtree nodes and smaller than all right subtree nodes.

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 leaf node or 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

Practice Problem

- From the tree of the previous problem, delete nodes 47, 22, and 42 in that order.``

Remove(x)

```
Procedure Delete(x, T)
begin
    if T = NULL then return NULL;
    T′ = Find(x);

    if T′ has at most 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

- What are some bounds on the depth of a binary search tree of n nodes?
- A depth of n is also possible.

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.

Depth of a Binary Search Tree

- 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).

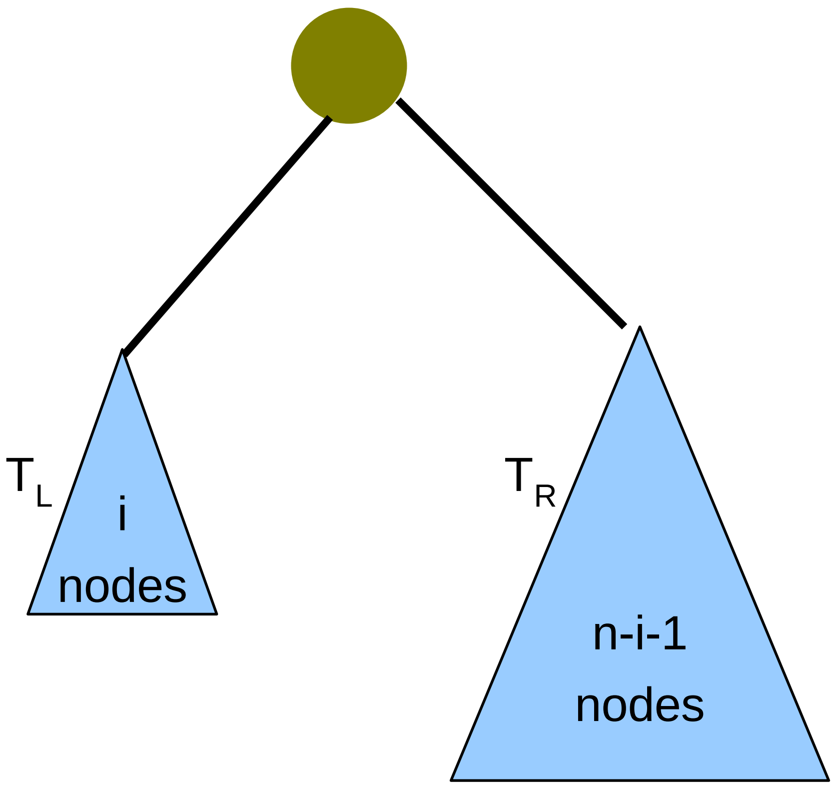
Depth of a Binary Search Tree

- 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.
  - $\Sigma_{i=1}^{n}$ d(i), where d(i) is the depth of node i.
- Note that D(1) = 0.

Depth of a Binary Search Tree

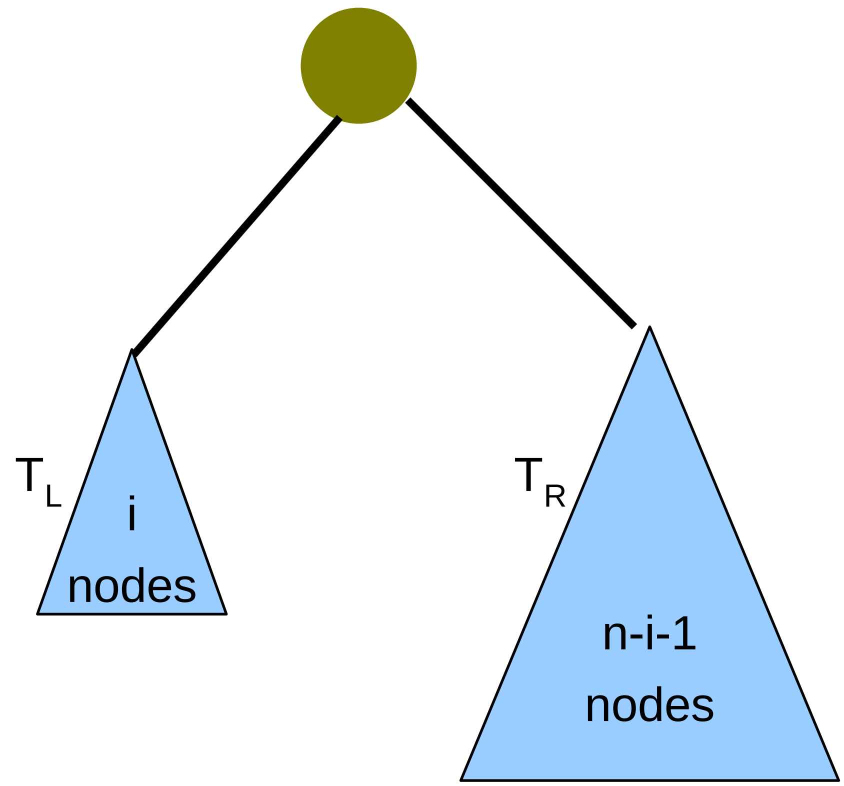
- In a tree with N nodes, there is one root node and a left subtree of i nodes and a right subtree of n−i−1 nodes.
- Using our notation, D(i) is the internal path length of the left subtree.
- D(n-i-1) is the internal path length of the right subtree.

Depth of a Binary Search Tree



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

Depth of a Binary Search Tree



- 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 $D(i) = (1/n) \Sigma_{j=0}^{n-1} D(j)$
- Similar is the case with the right subtree.
- The recurrence relation simplifies to

$$D(n) = (2/n) (\Sigma_{j=0}^{n-1} D(j)) + N - 1$$

- Can be solved using known techniques.
  - Left as homework.

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.