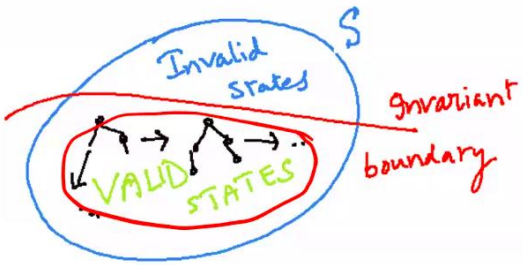
Binary Search Tree

- Consider a problem to store and search a modifiable collection

	Insert	Delete	Search
Array	O(1)	O(n)	O(n)
Linked list	O(1)	O(n)	O(n)
Sorted array	O(n)	O(n)	O(log n)
BST	O(log n)	O(log n)	O(log n)

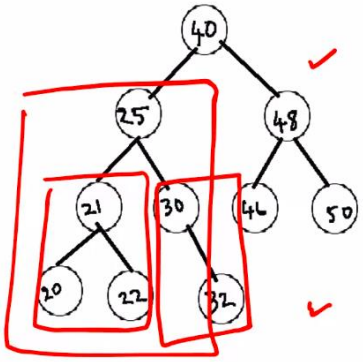
Data Structure Design via Invariants

An ***invariant*** is just a set of conditions that will hold before and after every "step" of your program/algorithm.

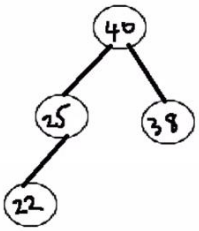


https://fgiesen.wordpress.com/2010/09/27/data-structures-and-invariants/

Binary Search Tree



A binary search tree

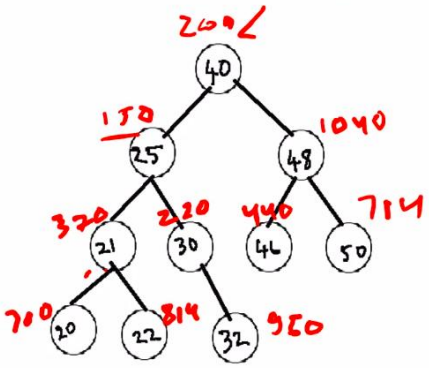


Not a binary search tree

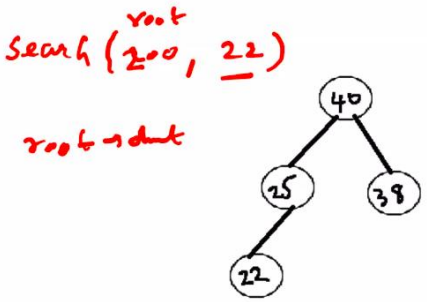
Operations

- 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.

Binary Search Tree

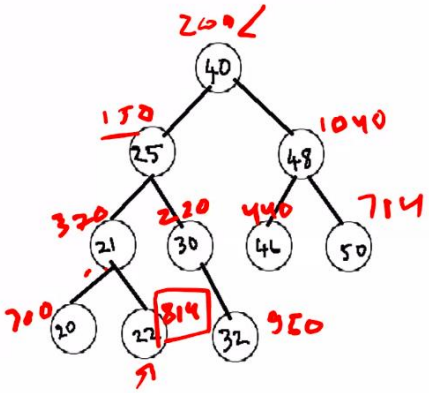


A binary search tree

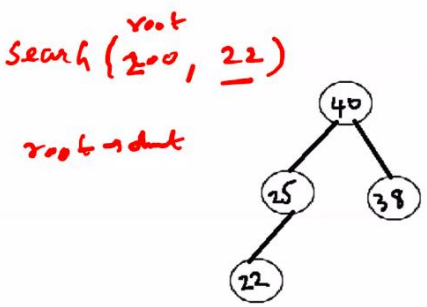


Not a binary search tree

Binary Search Tree

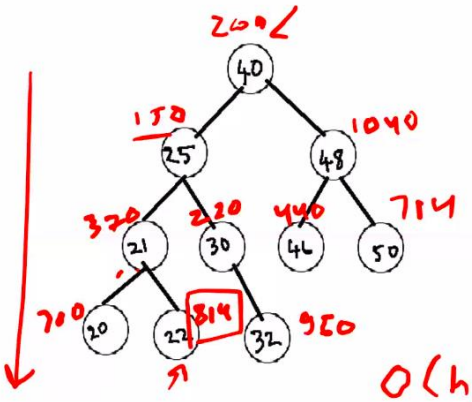


A binary search tree

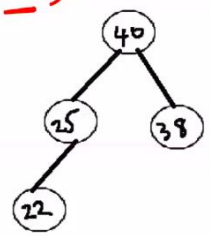


Not a binary search tree

Binary Search Tree



A binary search tree

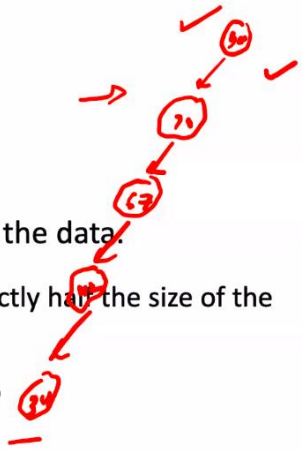


Not a binary search tree

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 no further exploration possible (no child in the required side)

Find(x)

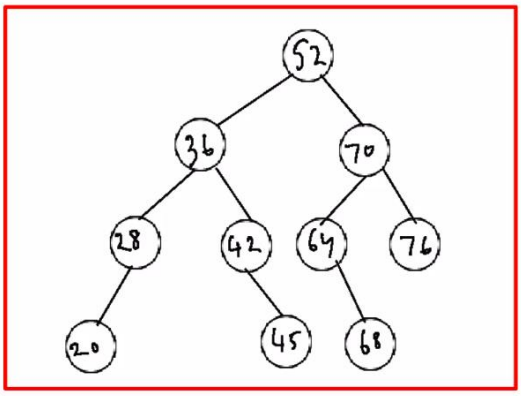


- Notice the similarity to binary search.
- In both cases, we continue search in a subset of the data.
  - In the case of binary search the subset size is exactly half the size of the current set.
  - Is that so in the case of a binary search tree also?
  - May not always be true.

Find(x)

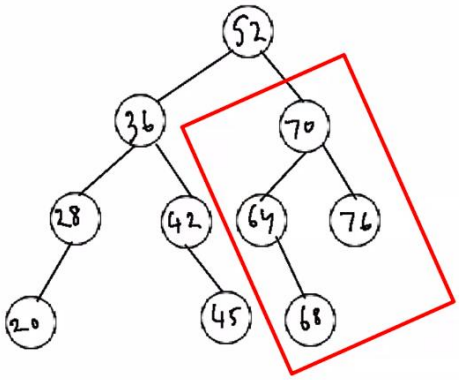
- How to analyze the runtime?
- Number of comparisons is a good metric.
- Notice that for a successful or an unsuccessful search, the worst case number of comparisons is equal to the height of the tree.
- What is the height of a binary search tree?
  - We'll postpone this question for now.

Find(x)



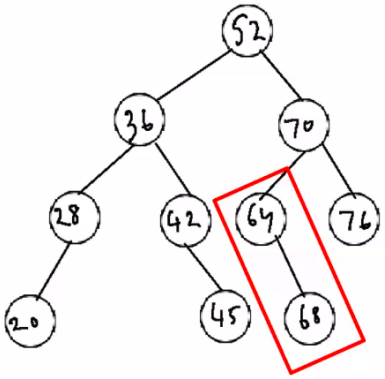
- Search for 68.
- Since 52 < 68, we search in the right subtree.

Find(x)



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

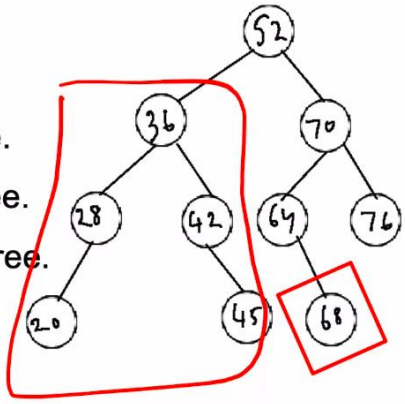
Find(x)



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

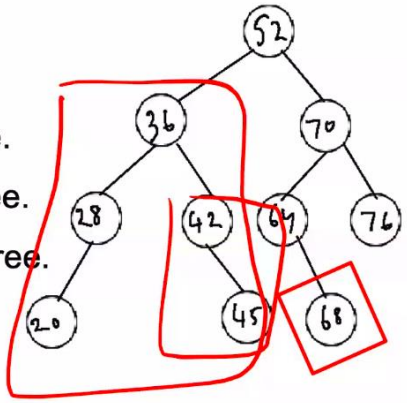
Find(x)

- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.
- Since 68 < 65, again search in the right subtree.
- Finally, find 68 as a leaf node.
- Now, try Find(48)



Find(x)

- Search for 68.
- Since 52 < 68, we search in the right subtree.
- Since 68 < 70, again search in the left subtree.
- Since 68 < 65, again search in the right subtree.
- Finally, find 68 as a leaf node.
- Now, try Find(48)



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
```

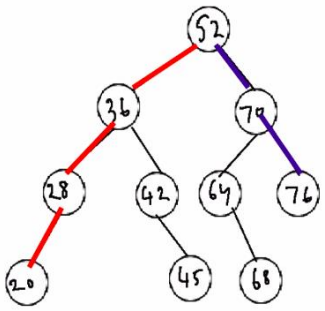
Observation on Find(x)

- 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.

Operation FindMin and FindMax

```
struct BST {
    int data;
    BST* left;
    BST* right;
};
```

```
int FindMin (BST* root){
  if(root == NULL) {
    cout << "Tree is empty\n";
      return -1;
  }
  while (root->left != NULL) {
      root = root->left;
  }
  return root->data;
}
```

Operation FindMin and FindMax

```
struct BST {
    int data;
    BST* left;
    BST* right;
};
```

```
int FindMin (BST* root){
  if(root == NULL) {
    cout << "Tree is empty\n";
      return -1;
  }
  while (root->left != NULL) {
      root = root->left;
  }
  return root->data;
}
```

Operation FindMin and FindMax

```
struct BST {
    int data;
    BST* left;
    BST* right;
};
```

```
int FindMin (BST* root) {
    if(root == NULL) {
        cout << "Tree is empty\n";
        return -1;
    }
    while (root->left != NULL) {
        root = root->left;
    }
    return root->data;
}
```

```
else if (root->left == NULL) {
    return root->data;
}
return FindMin(root->left);
```

Operation FindMin and FindMax


```
struct BST {
    int data;
    BST* left;
    BST* right;
};
```

```
int FindMin (BST* root) {
    if(root == NULL) {
        cout << "Tree is empty\n";
        return -1;
    }
    while (root->left != NULL) {
        root = root->left;
    }
    return root->data;
}
```

```
else if (root->left == NULL) {
    return root->data;
}
return FindMin(root->left);
```

Binary Search Tree: Insert(x)

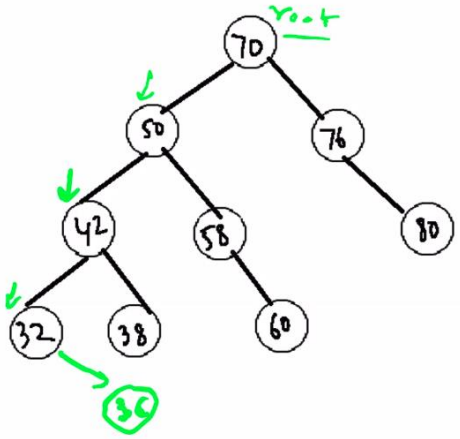
- Where should x be inserted?
- Should satisfy the search invariant.
  - So, if x is larger than the root, insert in the right subtree
  - if x is smaller than the root, insert in the left subtree.
- Repeat the above till we reach a deficient node.
- Can always add a new child to a deficient node.
-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

Insert(x)

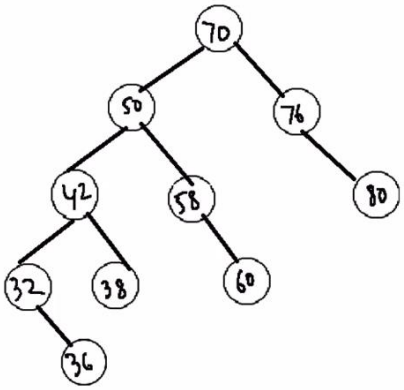
- Notice the analogy to Find(x)
- If x is not in the tree, Find(x) stops at a deficient node.
- Now, we are inserting x as a child of the deficient node last visited by Find(x).
- If the tree is presently empty, then x will be the new root.
- Let us consider a few examples.

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)

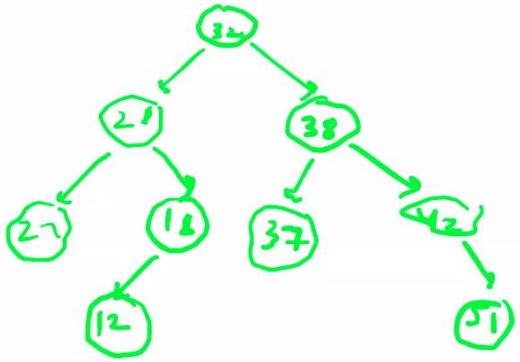


- Now, 36 > 32. So 36 is inserted as a right child of 32.

Insert(x)

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

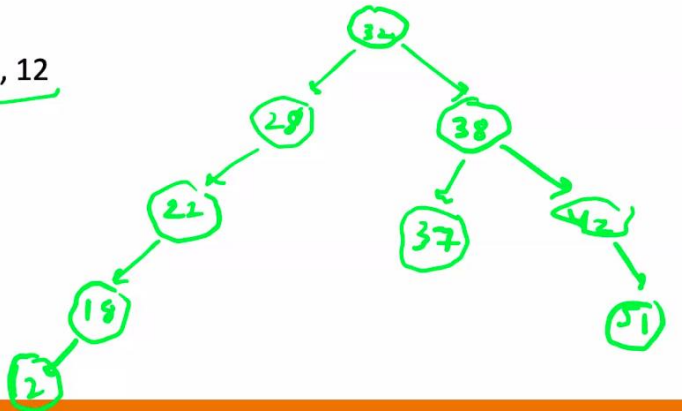
32, 28, 22, 38, 42, 51, 18, 37, 12



Insert(x)

- 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

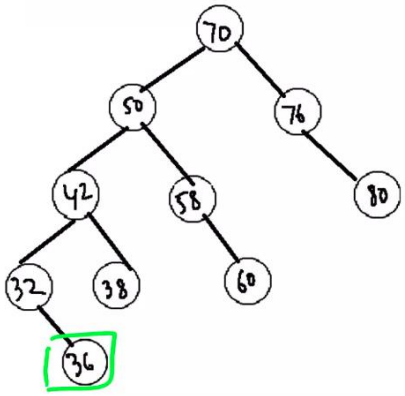


Insert()

```
BST* Insert ( Bst *root, int data)
    if (root == NULL) {
       root = CreateNode (data);
       return root;
    }
else if ( data <= root->data) {
    root-> left = Insert (root->left, data);
    }
else {
    root -> right = Insert (root->right , data);
    }
    return root ;
    }
```

```
BST* CreateNode (int data){
    Bst* newNode = new Bst();
    newNode -> data = data ;
newNode -> left = newNode -> right = NULL;
    return newNode ;
    }
```

Insert(x)



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

Insert()

```
BST* Insert ( Bst *root, int data)
  if (root == NULL) {
     root = CreateNode (data);
     return root;
  }
else if ( data <= root->data) {
   root->left = Insert (root->left, data);
  }
else {
   root->right = Insert (root->right , data);
  }
  return root;
}
```

```
BST* CreateNode (int data){
   Bst* newNode = new Bst();
   newNode -> data = data ;
   newNode-> left = newNode->right = NULL;
   return newNode;
}
```

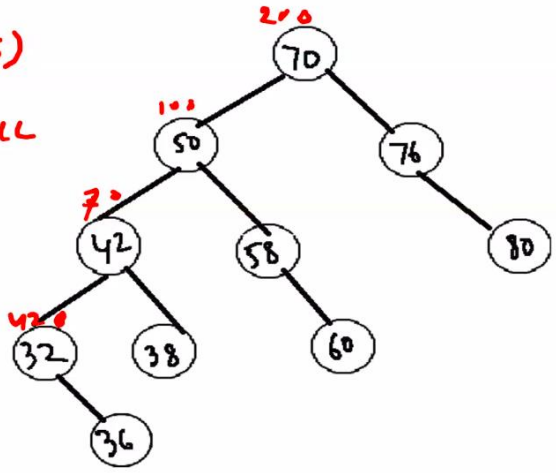
Insert(x)

- 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)

case 1 →
remove (208, 36)

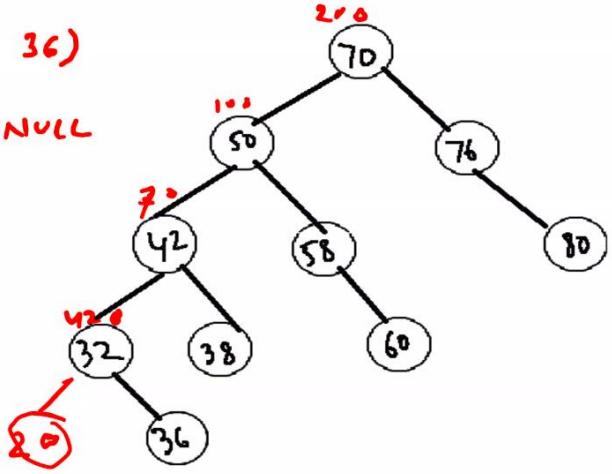
420 → right = NULL



Binary Search Tree: Remove(x)

case 1 →
remove (208, 36)

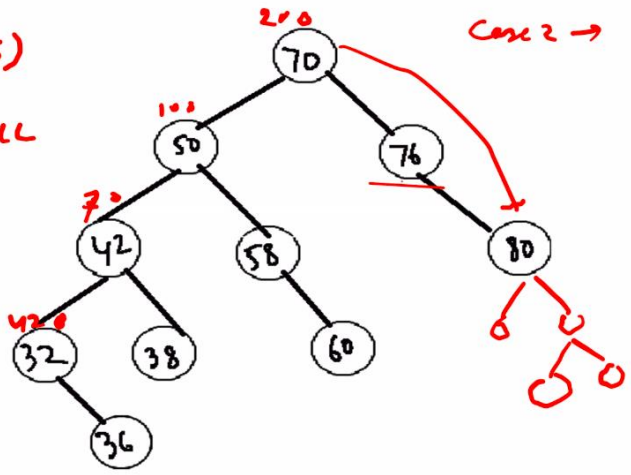
420 → right = NULL



Binary Search Tree: Remove(x)

case 1 → remove (208, 36)

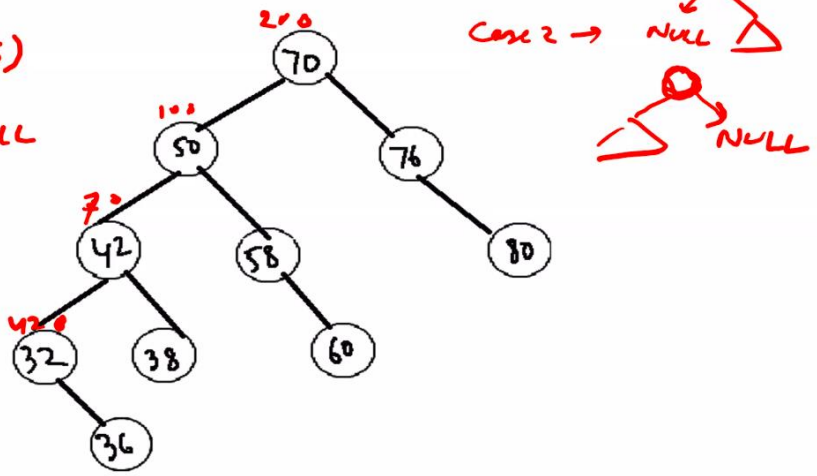
420 → right = NULL



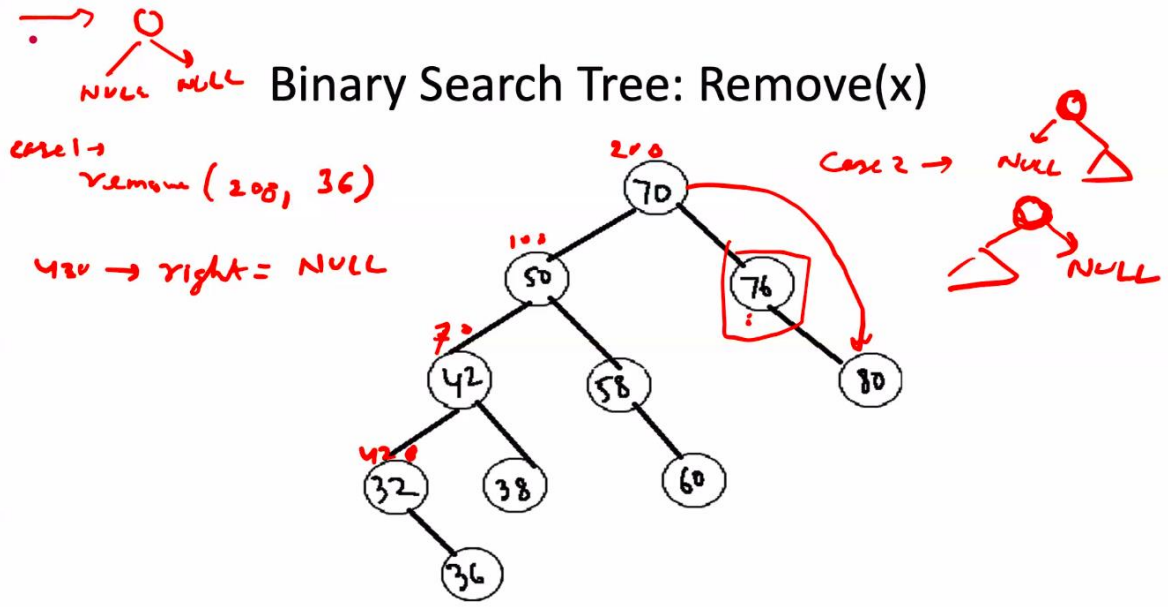
Binary Search Tree: Remove(x)

case 1 → remove (208, 36)

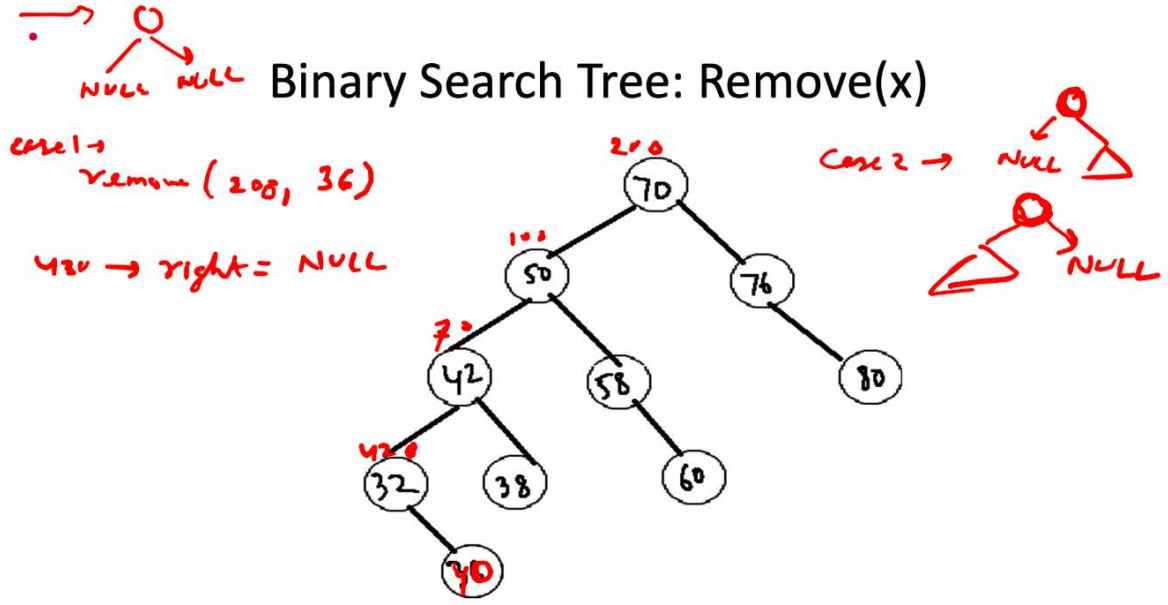
420 → right = NULL



Binary Search Tree: Remove(x)



Binary Search Tree: Remove(x)

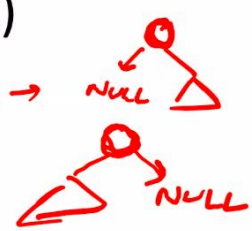


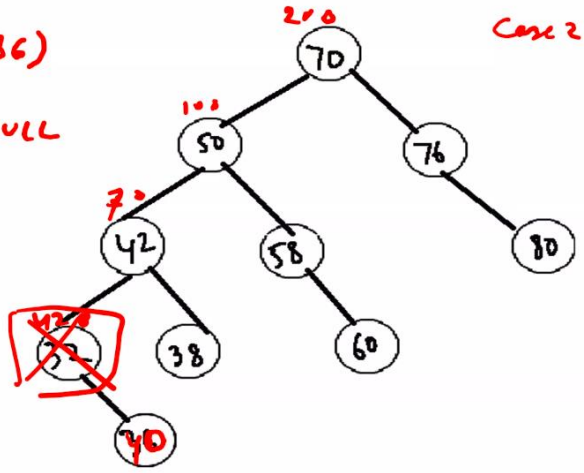
Binary Search Tree: Remove(x)

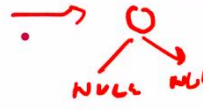
case 1 →
remove (208, 36)

42 → right = NULL

Case 2 →





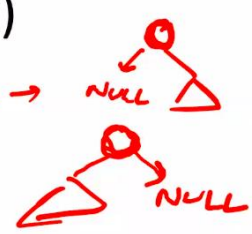


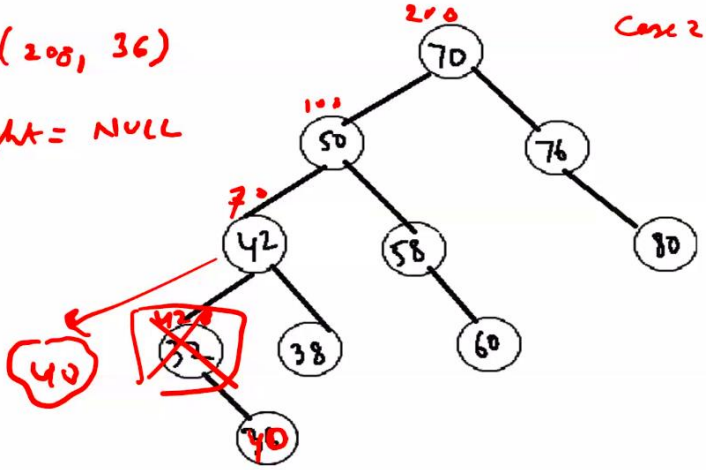
Binary Search Tree: Remove(x)

case 1 →
remove (208, 36)

42 → right = NULL

Case 2 →

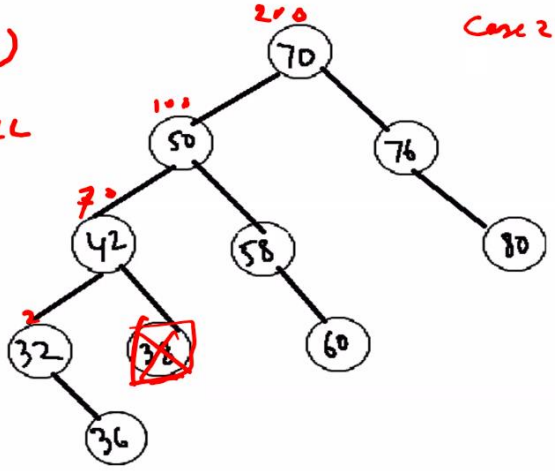




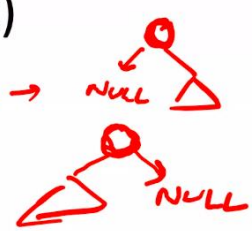
Binary Search Tree: Remove(x)

case 1 →
remove (208, 36)

42 → right = NULL



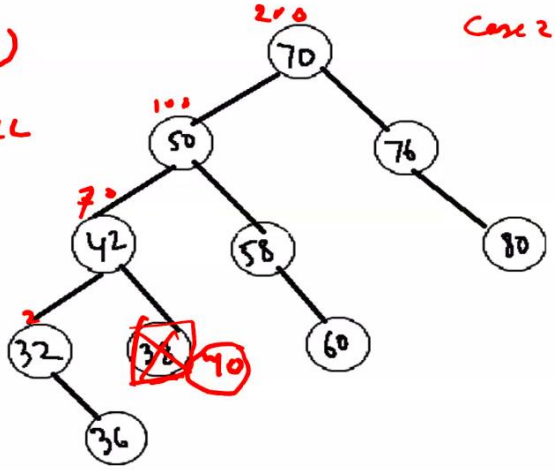
Case 2 →



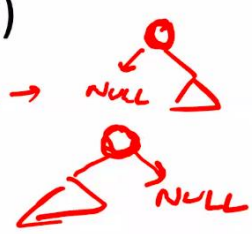
Binary Search Tree: Remove(x)

case 1 →
remove (208, 36)

42 → right = NULL



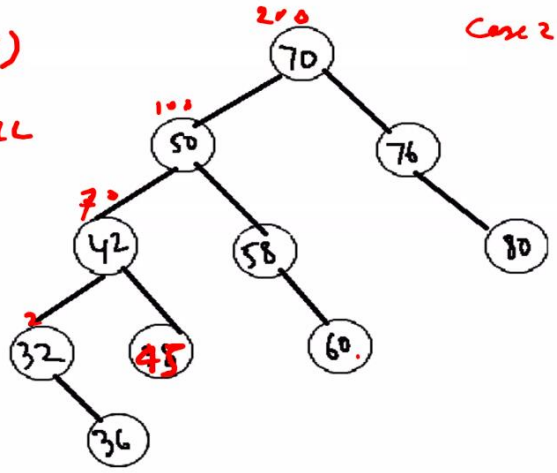
Case 2 →



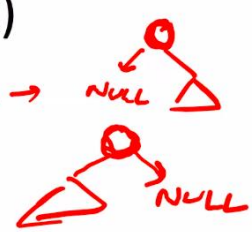
Binary Search Tree: Remove(x)

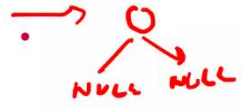
case 1 → remove (208, 36)

208 → right = NULL



Case 2 →

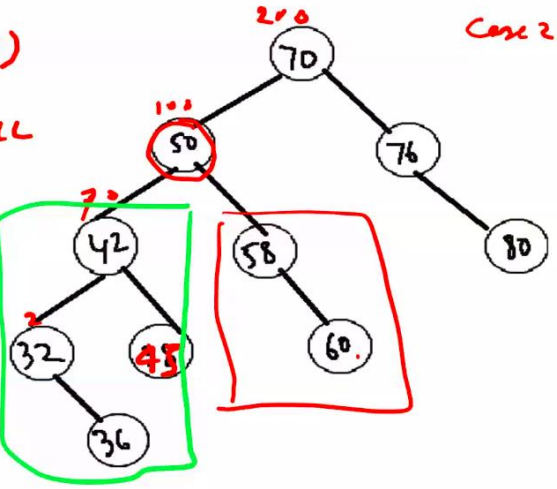




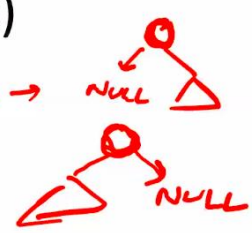
Binary Search Tree: Remove(x)

case 1 → remove (208, 36)

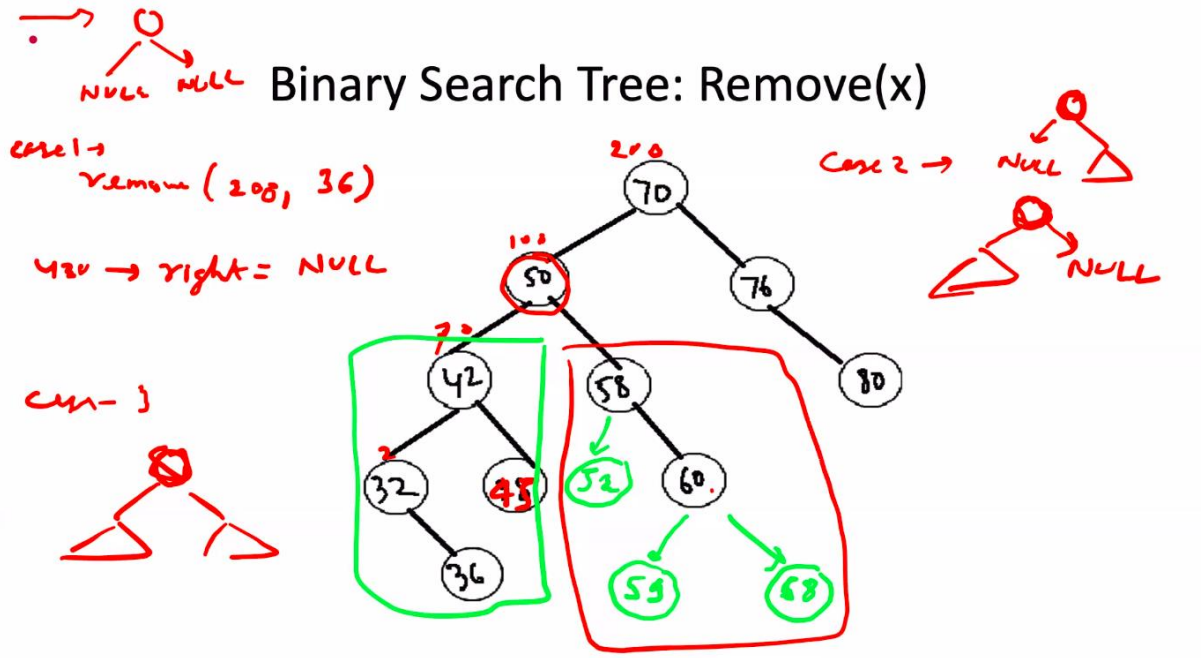
208 → right = NULL

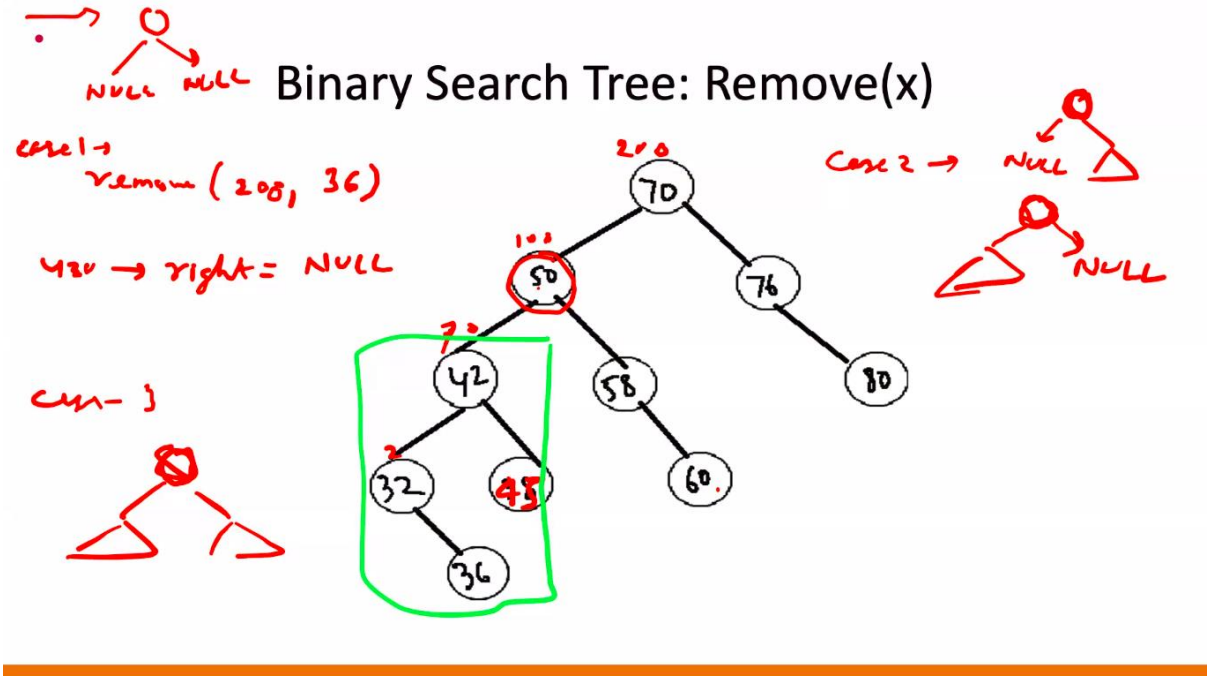


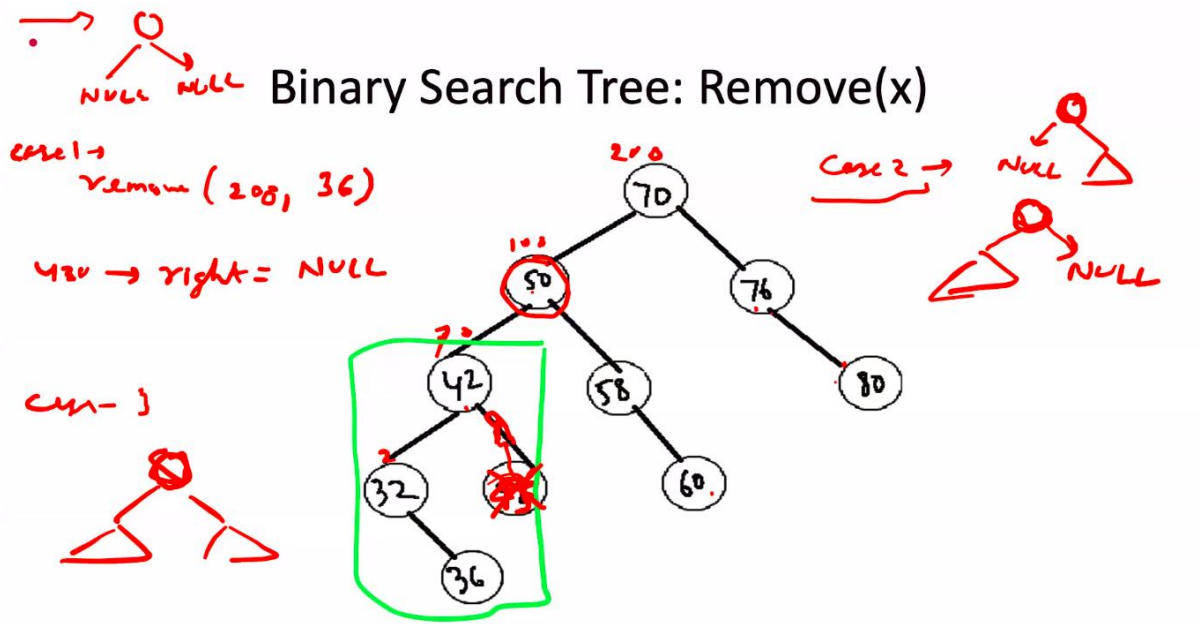
Case 2 →



Case 3

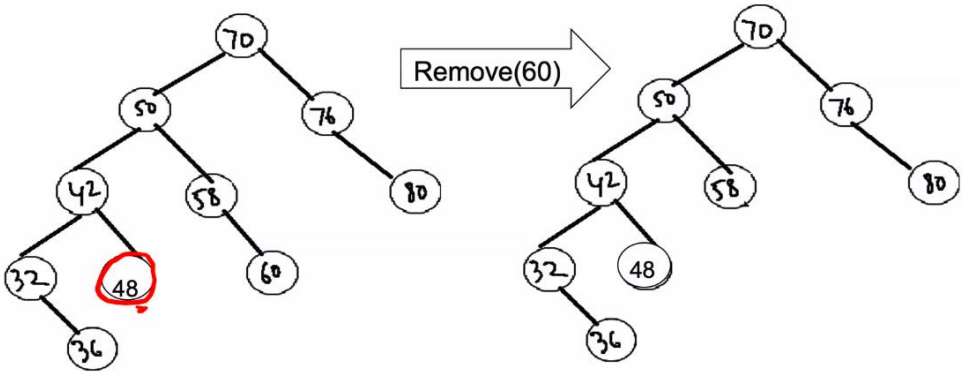

NULL
NULL
Binary Search Tree: Remove(x)
case 1 →
remove (208, 36)
208 → right = NULL
case - 1
208
70
100
50
76
72
42
58
80
32
45
52
60
36
59
68
Case 2 →
NULL
NULL


NULL
NULL
Binary Search Tree: Remove(x)
case 1 →
remove (208, 36)
208 → right = NULL
case - 1
208
70
100
50
76
72
42
58
80
32
45
60
36
Case 2 →
NULL
NULL



Remove(x)

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



Binary Search Tree: Remove(x)