Binary Search Trees (BST)

Subhabrata Samajder

IIIT, Delhi

Winter Semester,

5th April, 2023

Tree Algorithms

Counting Nodes of a Binary Tree

Counting Nodes of a Binary Tree

- One needs to visit all the nodes.
- Count the current node, and
- recursively visit the left sub-tree and the right sub-tree.

```
int count (Node *Root) {
  if (pRoot==null)
    return 0;
  else
    return 1 + count(pRoot->pLeft) + count(pRoot->pRight);
}
```

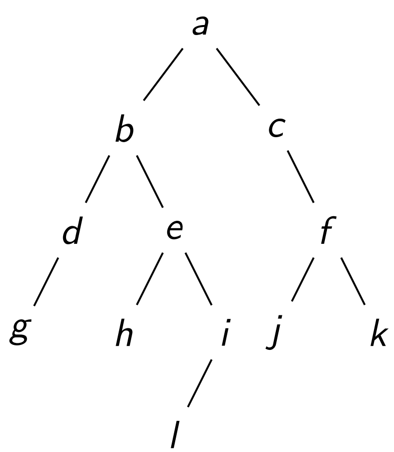
Counting Nodes of a Binary Tree

- One needs to visit all the nodes.
- Count the current node, and
- recursively visit the left sub-tree and the right sub-tree.

```
int count (Node *Root) {
  if (pRoot==null)
    return 0;
  else
    return 1 + count(pRoot->pLeft) + count(pRoot->pRight);
}
```

Exercise: Write an equivalent iterative code for counting nodes.

Height of a Tree



- The height of a binary tree is the number of levels of tree.
  - Tree height: 5.
- Height of left sub-tree: 4.
- Height of right sub-tree: 3.

Height of a Binary Tree

1. Get the height of left sub tree, say LeftHeight.
2. Get the height of right sub tree, say RightHeight.
3. Take $\max\{LeftHeight, RightHeight\}$ and add 1 for the root
4. Call recursively.

Height of a Binary Tree: C Code

```
/* height of binary tree */
int height (Node pRoot) {
  if (pRoot==null)
    return 0;
  else
    return 1 + max(height(root->pLeft), height (root->pRight));
}
```

Copying a Binary Tree

- Copy the current node.
- Recursively call the routine for left sub-tree and right sub-tree.

```
BTNode *Copy (BTNode pRoot) {
  if (pRoot == null) {
    return null;
  }

  {BTNode *copy = NULL;
  copy = (BTNode *)malloc(sizeof(BTNode));
  copy->nData = pRoot->nData; }

  copy->pLeft = Copy(pRoot->pLeft);
  copy->pRight = Copy(pRoot->pRight);

  return copy;
}
```

Binary Search Tree (BST)

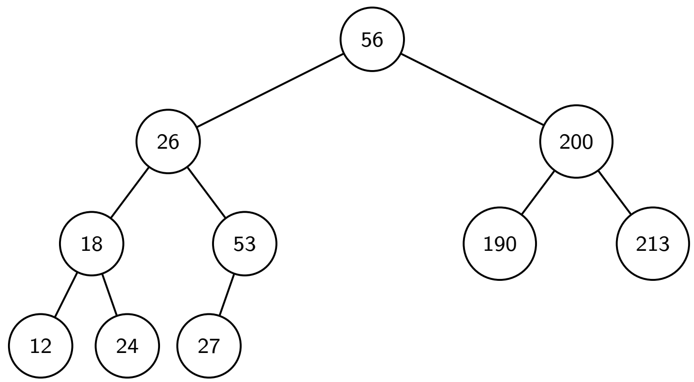
Binary Search Tree (BST)

- The name itself suggests the purpose of the tree.
- We can easily carry out binary search on such a tree.
- The process is similar to Binary Search method.
- We create a binary search tree where the elements are stored in a sorted manner.

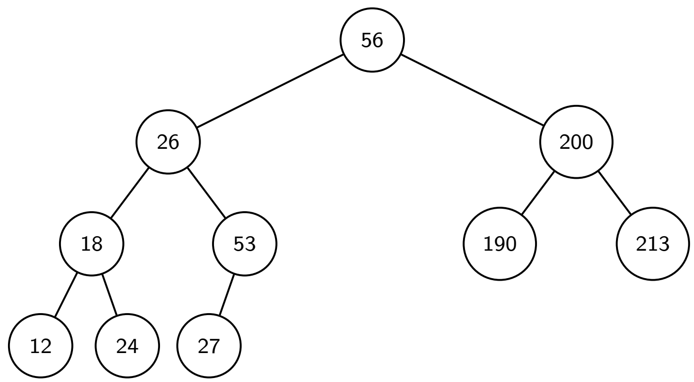
Binary Search Tree (BST): Definition

- A BST is a special type of rooted Binary tree.
- The value stored at the root is
  - *more* than any value in its left sub-tree and
  - *less* than any value in its *right sub-tree*.
  - This is called the binary search property.
- A null tree is a BST.
- The binary search property must be satisfied for all the nodes on the tree and their sub-trees.

Maximum and Minimum



Maximum and Minimum



- Minimum element is the leftmost node.
- Maximum element is the rightmost node.

Tree-Maximum and Tree-Minimum: Pseudocodes

Tree-Maximum(x)

I/P: The root x of a BST T.
O/P: Maximum element of T.

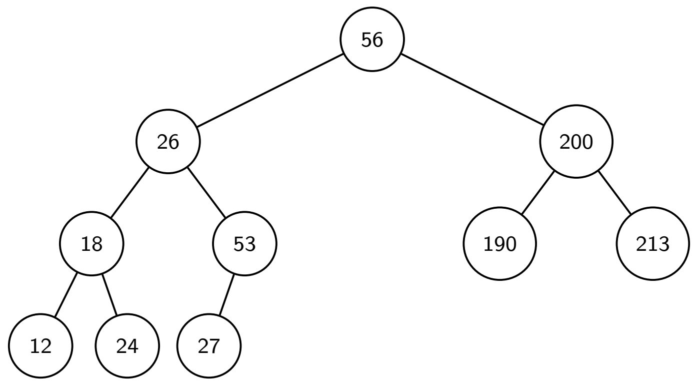
```
Begin
   while (right[x] ≠ nil) {
      x ← right[x];
   }

   return;
End
```

Tree-Minimum(x)

I/P: The root x of a BST T.
O/P: Minimum element of T.

```
Begin
   while (left[x] ≠ nil) {
      x ← left[x];
   }

   return;
End
```

Search in a BST



Search in a BST: Pseudocode

$\textsc{Tree-Search}(x, k)$

I/P: The root x of a BST T and a key k.
O/P: Returns **True** if $k \in T$, otherwise returns **False**.

```
Begin
  if (x = nil)
    return False;
  if (k = key[x])
    return True;
  if (k ≤ key[x])
    return TREE-SEARCH(left[x], k);
  else
    return TREE-SEARCH(right[x], k);
End
```

Iterative BST Search Returning True/False

```
int searchBST (BTNode *pRoot, int target) {
   BTNode *current = NULL;

   current = pRoot;
   while (current != NULL) {
      if (current->nData == target)
         return 1;
      else if (current->data > target)
         current = current->pLeft;
      else
         current = current->pRight;
   }

   return 0;
}
```

Iterative BST Search Returning Node Reference

```
int search (BTNode *pRoot, int target) {
  BTNode *current = NULL;

  current = pRoot;
  while (current != NULL) {
    if (current->nData == target)
      return current;
    else if (current->data > target)
      current = current->pLeft;
    else
      current = current->pRight;
  }

  return NULL;
}
```

Complexity of Search in a BST

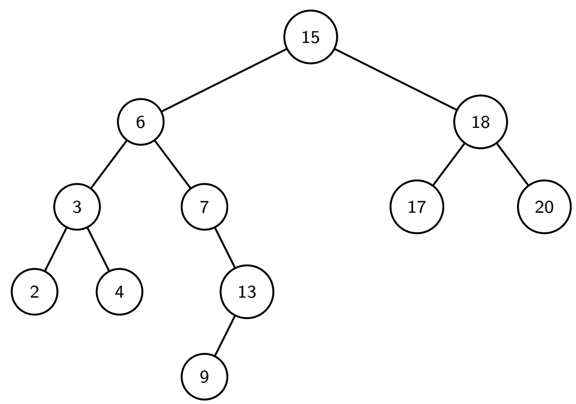
- **Worst-case:** Starts from the root and ends at the leaves.
- $\therefore$ search path corresponds to height of the tree which is $\mathcal{O}(\log n)$ if the tree is complete.
- **The recurrence relation for almost full BST:**

$$T(n) = T(n/2) + 1 \quad \Rightarrow \quad T(n) = \mathcal{O}(\log n).$$

- **Note:** If the tree is skewed, then its height is very nearly n.
    - **Worst-case complexity:** $\mathcal{O}(n)$.

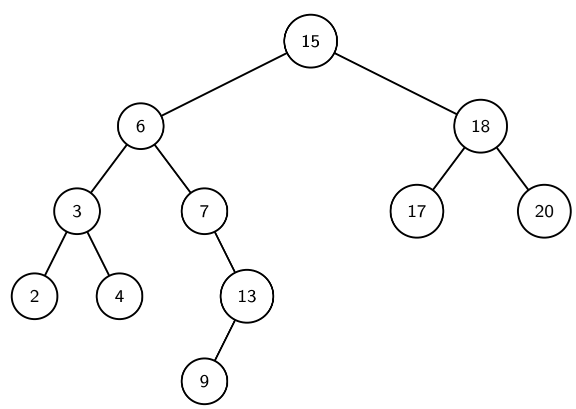
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!



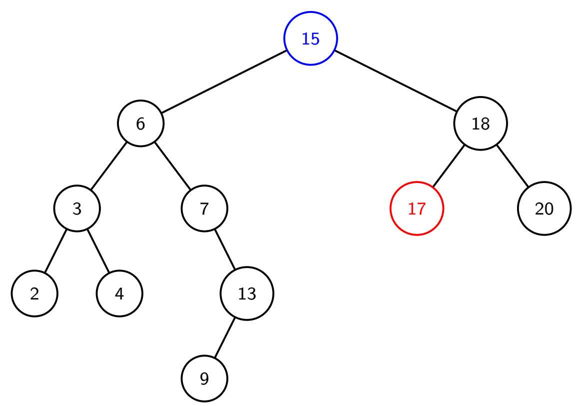
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 1:** The right sub-tree of a node x is non-empty.
    - **Successor of x:** The leftmost node in the right sub-tree.
    - $\therefore$ call $\textsc{Tree-Minimum}(\textit{right}[x])$.



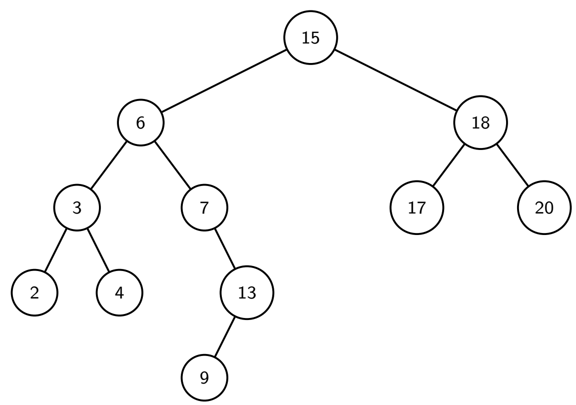
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 1:** The right sub-tree of a node x is non-empty.
    - **Successor of x:** The leftmost node in the right sub-tree.
    - $\therefore$ call $\textsc{Tree-Minimum}(\mathit{right}[x])$.
    - **Example:** Successor of 15 is 17.



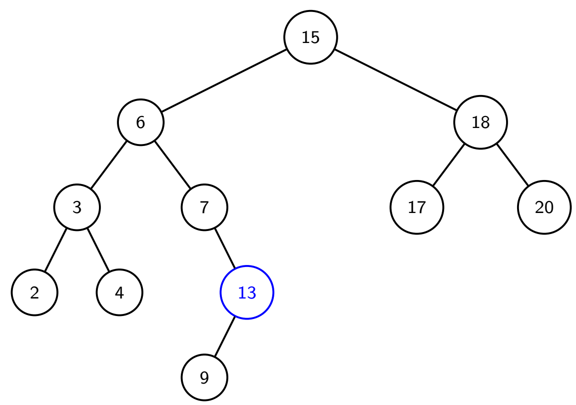
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The right sub-tree of a node x is empty.
    - **Successor of x:** The lowest ancestor of x whose left child is also an ancestor of x.



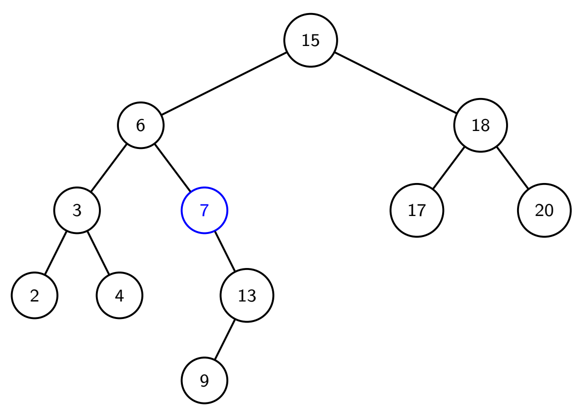
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The right sub-tree of a node x is empty.
    - **Successor of x:** The lowest ancestor of x whose left child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the left child of its parent.
    - **Example:** Consider the node 13.



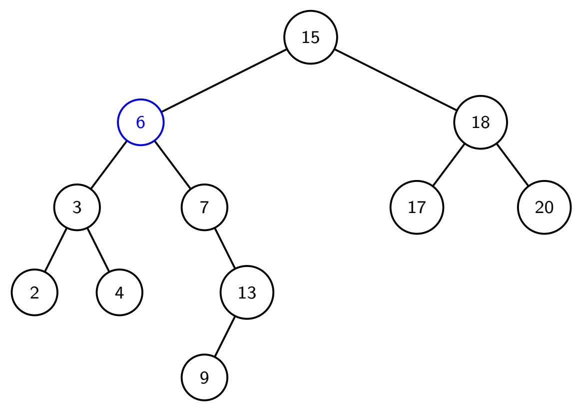
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The right sub-tree of a node x is empty.
    - **Successor of x:** The lowest ancestor of x whose left child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the left child of its parent.
    - **Example:** Consider the node 13.



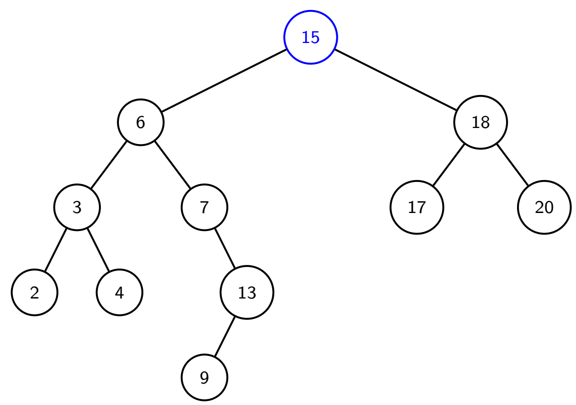
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The right sub-tree of a node x is empty.
    - **Successor of x:** The lowest ancestor of x whose left child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the left child of its parent.
    - **Example:** Consider the node 13.



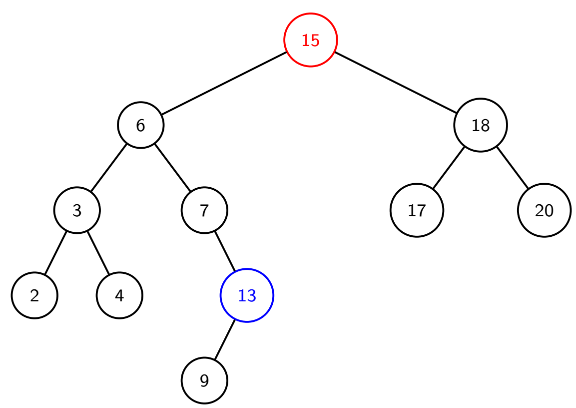
Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The right sub-tree of a node x is empty.
    - **Successor of x:** The lowest ancestor of x whose left child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the left child of its parent.
    - **Example:** Consider the node 13.



Successor

- The structure of BST allows us to determine the successor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The right sub-tree of a node x is empty.
    - **Successor of x:** The lowest ancestor of x whose left child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the left child of its parent.
    - Then this parent is the successor.
    - **Example:** The successor of 13 is 15.



Tree-Successor

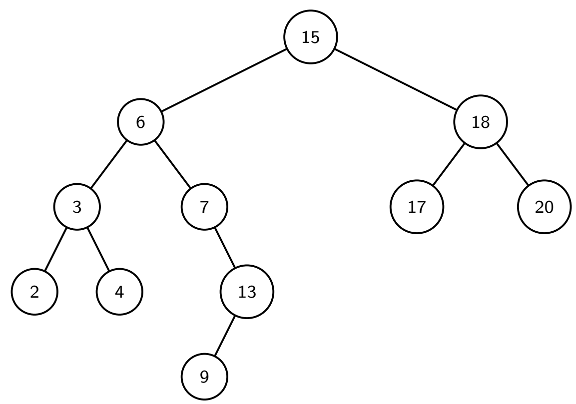
Tree-Successor(x)

I/P: A node x whose successor we need to find.
O/P: The successor of x.

```
Begin
  if (right[x] ≠ nil)
    return Tree-Minimum(right[x]);
  y ← parent[x];
  while (y ≠ nil) and (x = right[y])
    x ← y;
    y ← parent[y];
  return y;
End
```

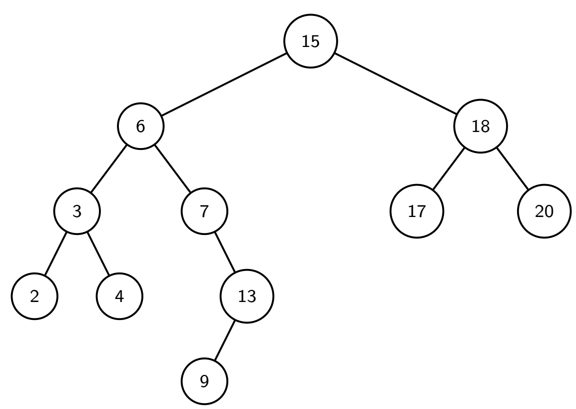
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!



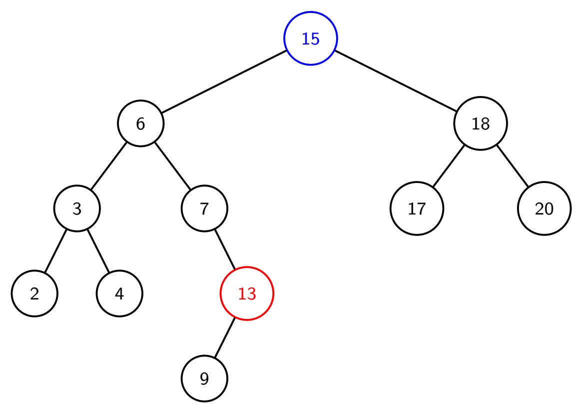
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 1:** The left sub-tree of a node x is non-empty.
    - **Predecessor of x:** The rightmost node in the left sub-tree.
    - $\therefore$ call $\textsc{Tree-Maximum}(\mathit{left}[x])$.



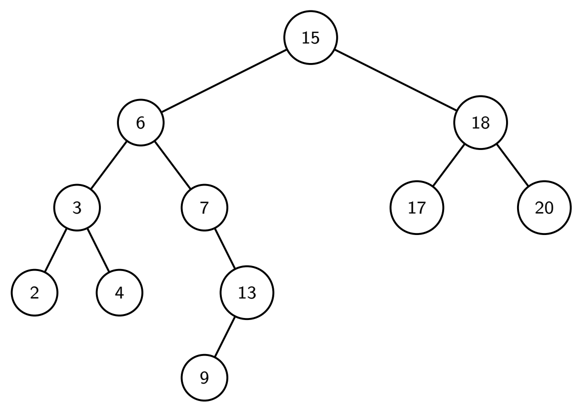
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 1:** The left sub-tree of a node x is non-empty.
    - **Predecessor of x:** The rightmost node in the left sub-tree.
    - $\therefore$ call $\textsc{Tree-Maximum}(\mathit{left}[x])$.
    - **Example:** Predecessor of 15 is 13.



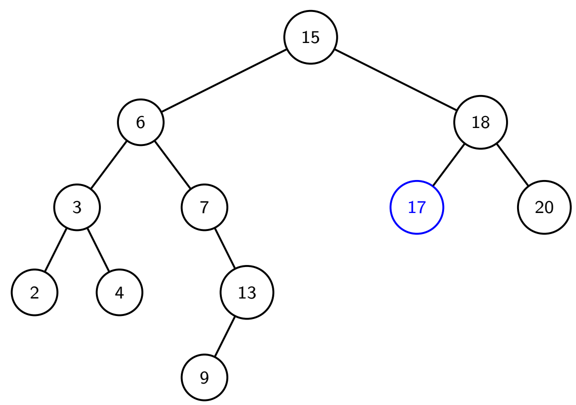
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The left sub-tree of a node x is empty.
    - **Predecessor of x:** The lowest ancestor of x whose right child is also an ancestor of x.



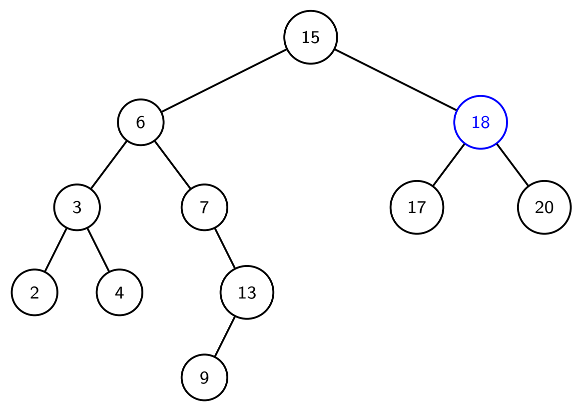
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The left sub-tree of a node x is empty.
    - **Predecessor of x:** The lowest ancestor of x whose right child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the right child of its parent.
    - **Example:** Consider the node 17.



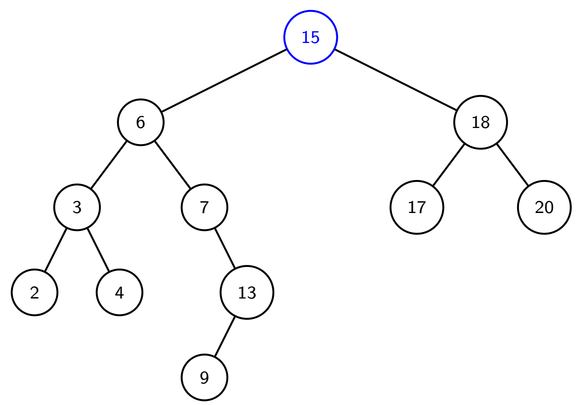
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The left sub-tree of a node x is empty.
    - **Predecessor of x:** The lowest ancestor of x whose right child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the right child of its parent.
    - **Example:** Consider the node 17.



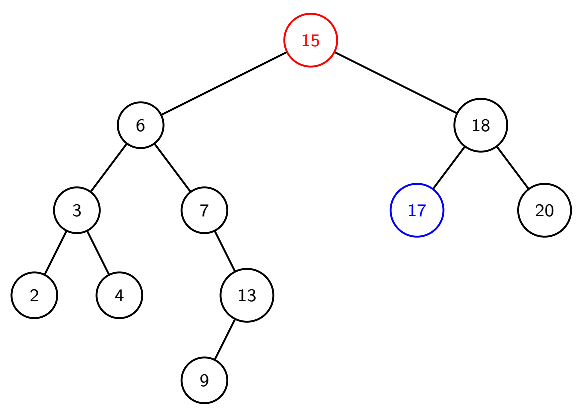
Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The left sub-tree of a node x is empty.
    - **Predecessor of x:** The lowest ancestor of x whose right child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the right child of its parent.
    - **Example:** Consider the node 17.



Predecessor

- The structure of BST allows us to determine the Predecessor of a node without ever comparing keys!
- Two cases may arise:
  - **Case 2:** The left sub-tree of a node x is empty.
    - **Predecessor of x:** The lowest ancestor of x whose right child is also an ancestor of x.
    - Go up the tree from x until we encounter a node that is the right child of its parent.
    - Then this parent is the predecessor.
    - **Example:** The Predecessor of 17 is 15.



Tree-Predecessor

Tree-Predecessor(x)

I/P: A node x whose predecessor we need to find.
O/P: The predecessor of x.

```
Begin
   if (left[x] ≠ nil)
      return Tree-Maximum(left[x]);
   y ← parent[x];
   while (y ≠ nil) and (x = left[y])
      x ← y;
      y ← parent[y];
   return y;
End
```

A Theorem

Theorem

The dynamic set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR *and* PREDECESSOR *can be made to run in* $\mathcal{O}(h)$ *time in a BST of height* h.

Checking BST property

Checking BST property

```
int BST(BTNode *pRoot, int min, int max) {
  if (pRoot == null)
    return true;

  return ((root->nData > min) && (root->data < max) &&
         BST(root->pLeft, min, pRoot->nData) &&
         BST(pRoot->pRight, pRoot->data, max));
}

// or do an inorder traversal and keep checking
```

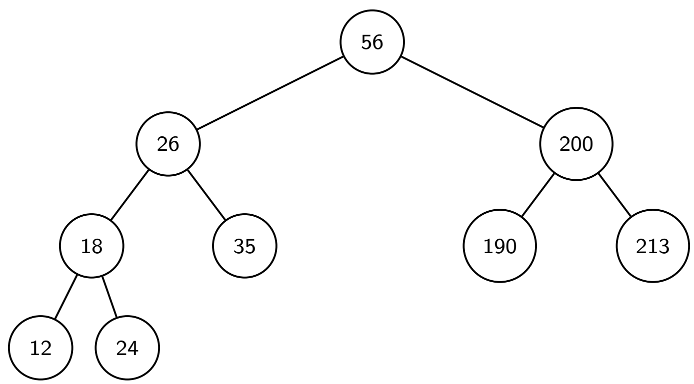
Inserting a node in a BST

Inserting a Node in a BST

- First find the *right place* to insert it.
- Follow the path from the root to the "*appropriate node*".
- That is the node which will be the parent of the new node.
- The new node is then connected as its *left* or *right child*, depending on whether the new node's key is *less* or *greater* than that of the parent.

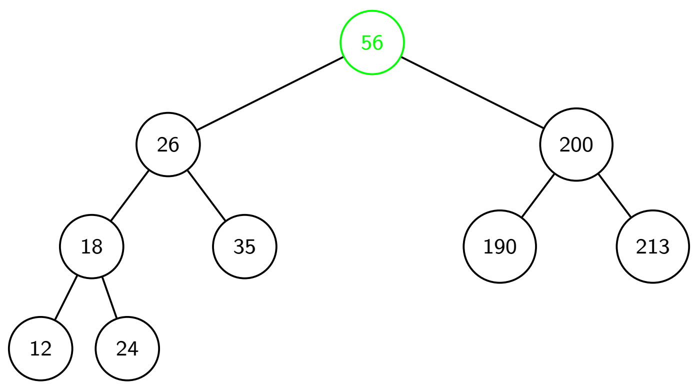
Insertion on a BST

- Insert 30 in the given BST.



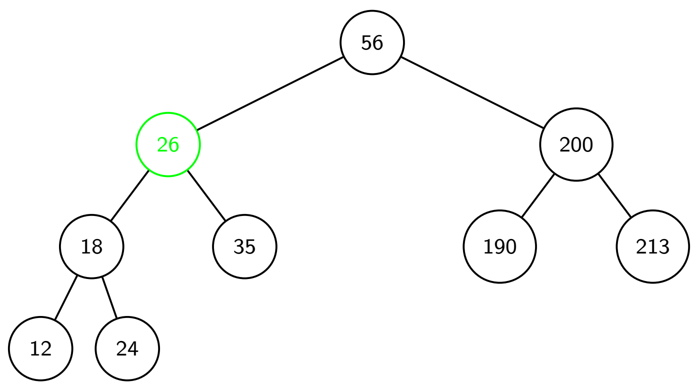
Insertion on a BST

- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.



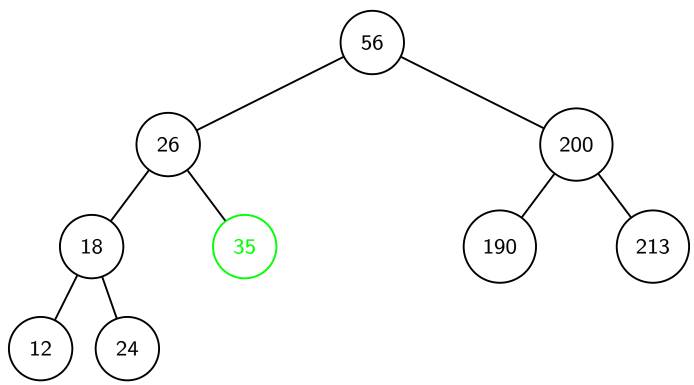
Insertion on a BST

- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.



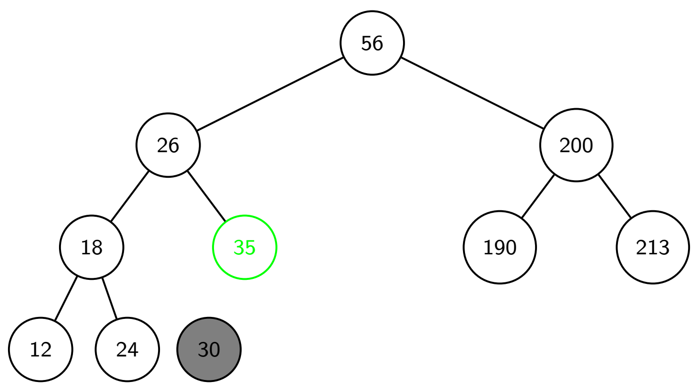
Insertion on a BST

- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.



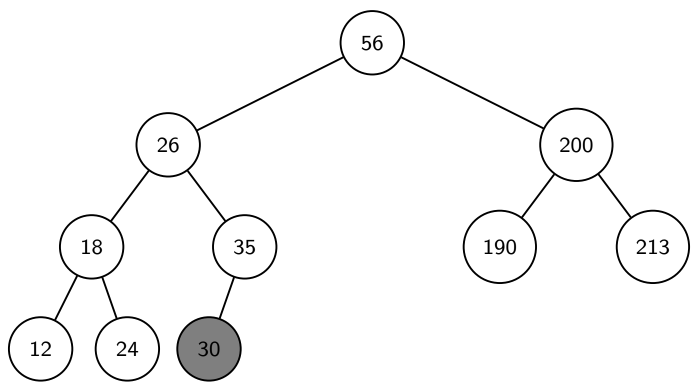
Insertion on a BST

- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.
- Create a new node with value 30.



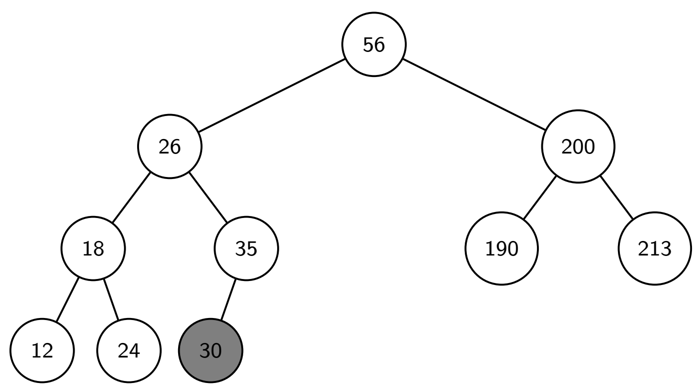
Insertion on a BST

- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.
- Create a new node with value 30.
- If 30 has to be on the tree, then only place is left child of 35.
- Attach the node 30 as left child of 35.



Insertion on a BST

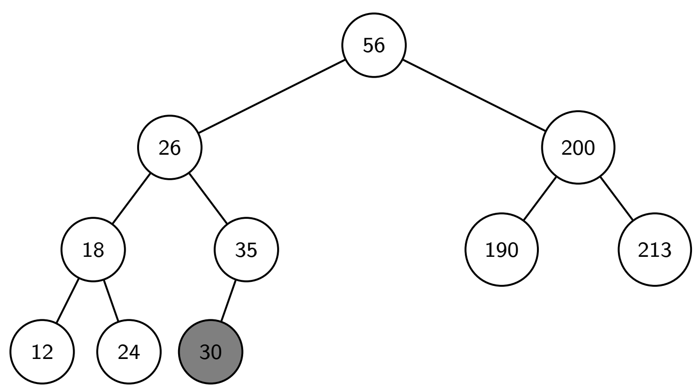
- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.
- Create a new node with value 30.
- If 30 has to be on the tree, then only place is left child of 35.
- Attach the node 30 as left child of 35.



Complexity?

Insertion on a BST

- Insert 30 in the given BST.
- Try to locate appropriate location for 30 on the tree.
- Create a new node with value 30.
- If 30 has to be on the tree, then only place is left child of 35.
- Attach the node 30 as left child of 35.



Complexity: $\mathcal{O}(h)$.

Inserting a Node in a BST: C Code

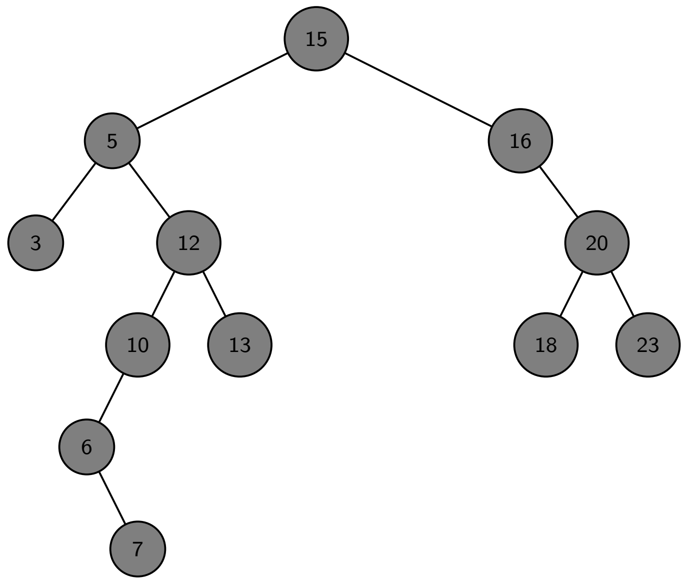
```
BTNode *insert (BTNode pRoot, int value) {
  if (pRoot == null) {
    pRoot = (BTNode *)malloc(sizeof(BTNode));
    pRoot->nData = value;
    pRoot->pLeft = pRoot->pRight = NULL;
  }
  else {
    if (value ≤ pRoot->nData)
      pRoot->pLeft = insert (pRoot->pLeft, value);
    else
      root->pRight = insert (root->pRight, value);
  }

  return pRoot;
}
```

Deletion in Binary Search Tree

Deleting in a BST

- Delete a specified item from the BST and adjust the tree.
- Use the binary search property to locate the target item:
  - **Starting at the root** probe down the tree till either the target node is reached or a leaf node reached (i.e., target node is not in the tree)
- Removal of a node must not leave a “gap” in the tree,

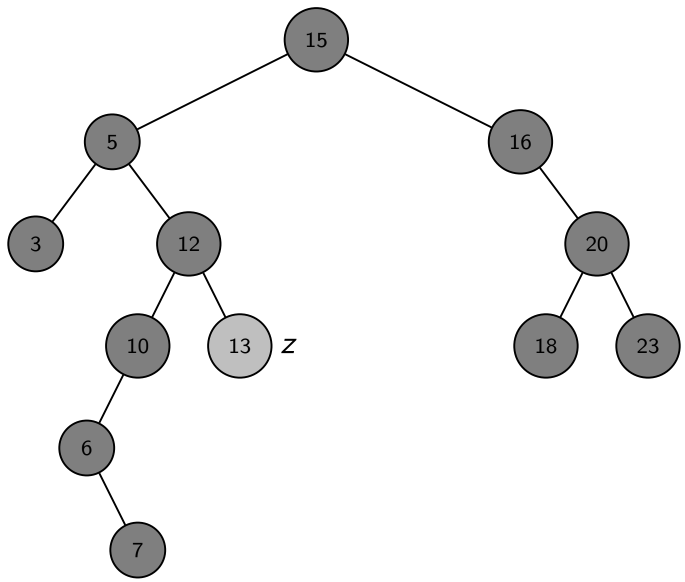
Deleting a Node z from a BST



Deleting a Node z from a BST

Three cases may arise:

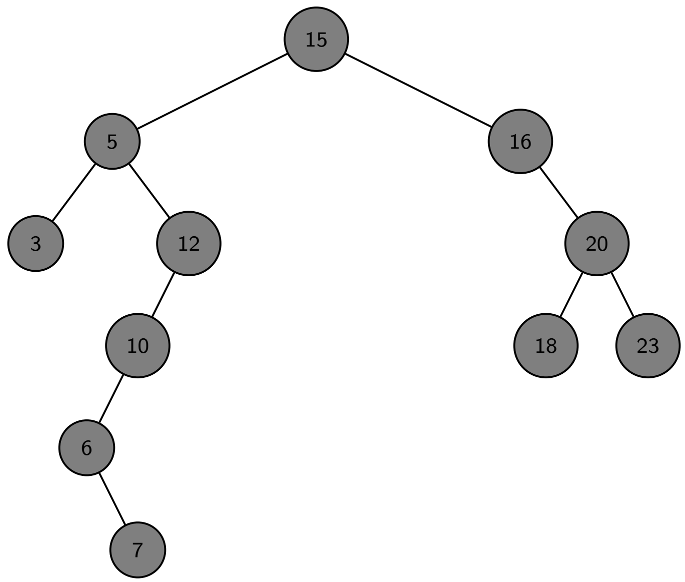
- z **has no chilren:** Consider $z = 13$.



Deleting a Node z from a BST

Three cases may arise:

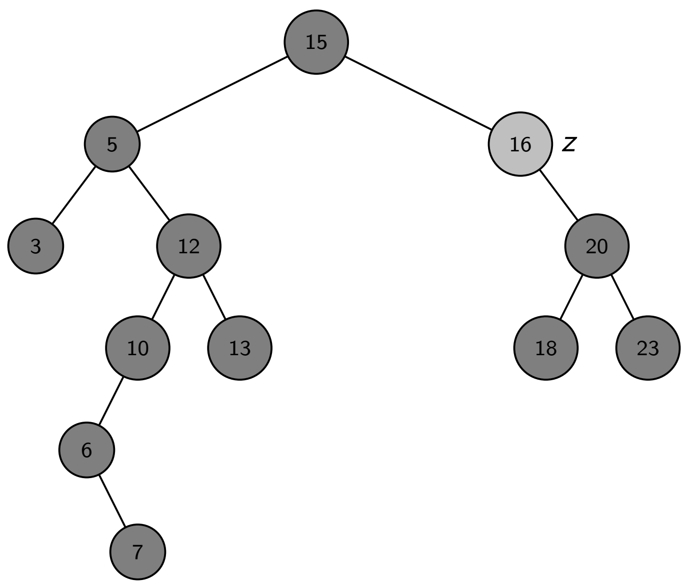
- z **has no chilren:** Consider $z = 13$.
  - Just remove it!



Deleting a Node z from a BST

Three cases may arise:

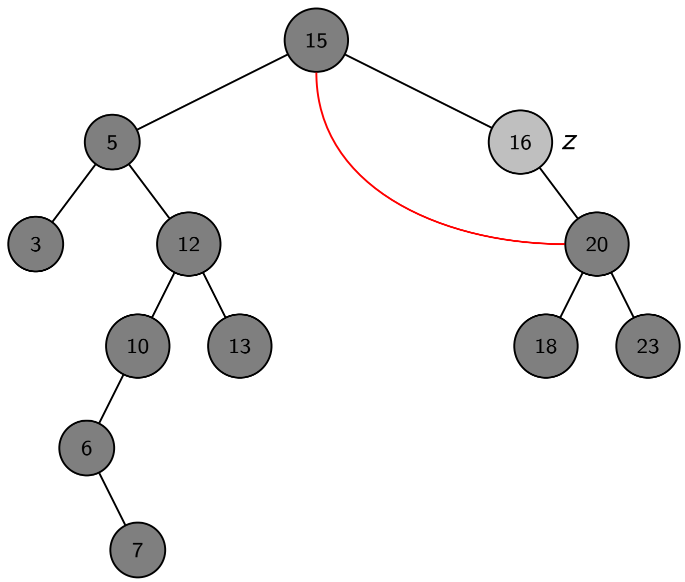
- z **has only one chilren:** Consider $z = 16$.



Deleting a Node z from a BST

Three cases may arise:

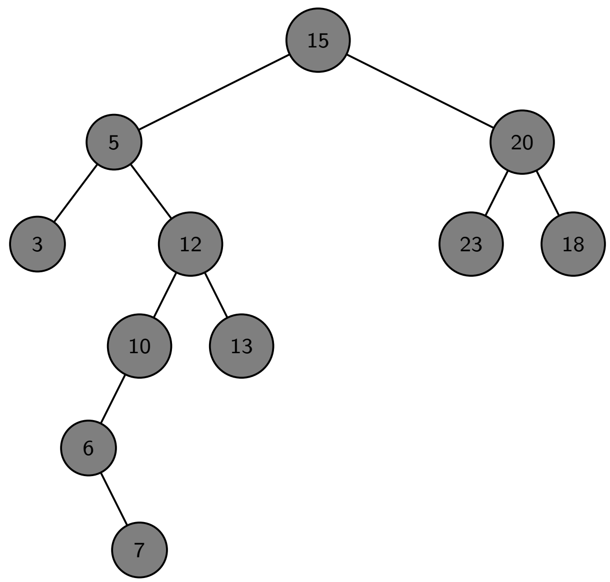
- z **has only one chilren:** Consider $z = 16$.
  - Splice out z.



Deleting a Node z from a BST

Three cases may arise:

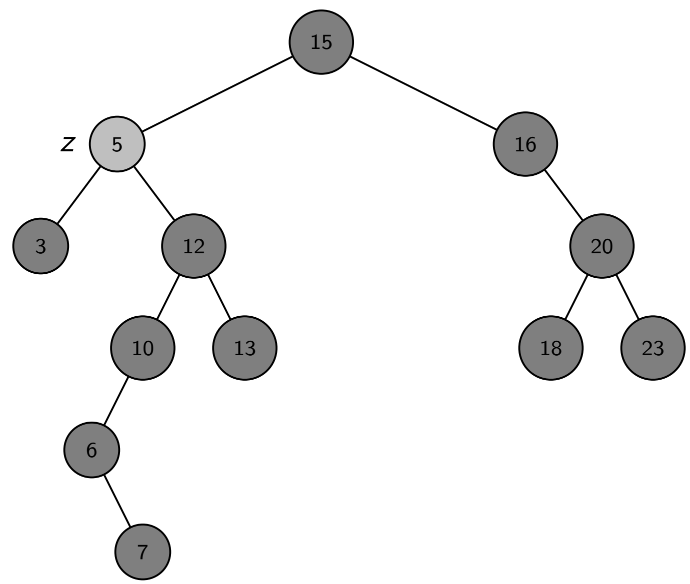
- z **has only one chilren:** Consider $z = 16$.
  - Splice out z.



Deleting a Node z from a BST

Three cases may arise:

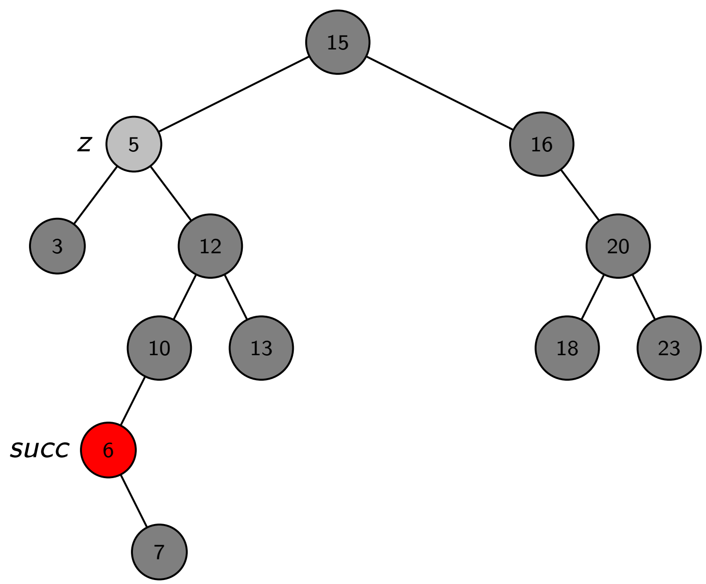
- z **has only two chilren:** Consider $z = 5$.



Deleting a Node z from a BST

Three cases may arise:

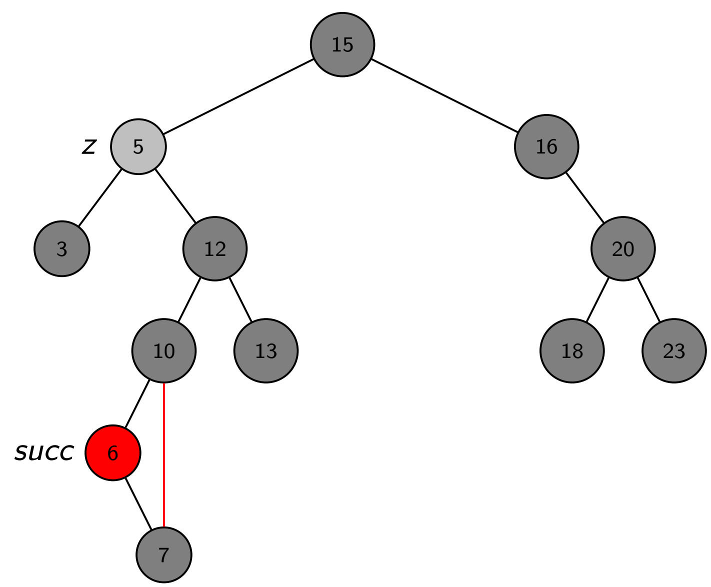
- z **has only two chilren:** Consider $z = 5$.
  - Find the successor of $z = 5$.



Deleting a Node z from a BST

Three cases may arise:

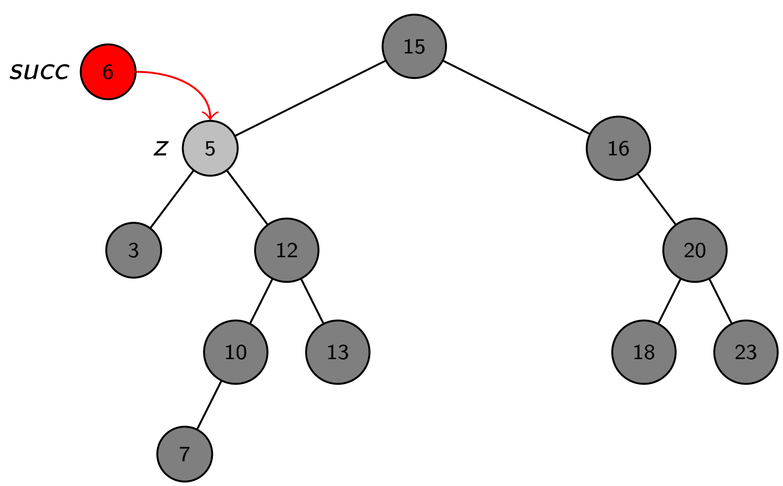
- z **has only two chilren:** Consider $z = 5$.
  - Find the successor of $z = 5$.
  - Splice out or Delete the successor of z depending on whether *succ* has one child or no child, respectively.
  - In our case we splice out the node $succ = 6$.



Deleting a Node z from a BST

Three cases may arise:

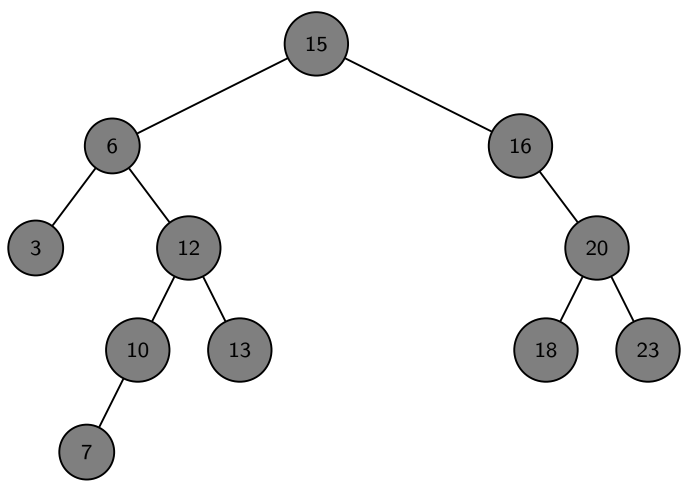
- z **has only two chilren:** Consider $z = 5$.
  - Find the successor of $z = 5$.
  - Splice out or Delete the successor of z depending on whether *succ* has one child or no child, respectively.
  - In our case we splice out the node $succ = 6$.
  - Copy 6 to the node 5.



Deleting a Node z from a BST

Three cases may arise:

- z **has only two chilren:** Consider $z = 5$.
  - Find the successor of $z = 5$.
  - Splice out or Delete the successor of z depending on whether *succ* has one child or no child, respectively.
  - In our case we splice out the node $succ = 6$.
  - Copy 6 to the node 5.



TREE-DELETE(T, z)

I/P: A tree T and a pointer to the node z to be deleted.
O/P: The updated tree T with it's node z deleted.

```
Begin
  if (left[z] = nil or right[z] = nil)
    y ← z;
  else
    y ← TREE-SUCCESSOR(z);

  if (left[y] ≠ nil)
    x ← left[y];
  else
    x ← right[y];

  if (x ≠ nil)
    p[x] ← p[y];

  if (p[y] = nil)
    root[T] ← x;
  else if (y = left[p[y]])
    left[p[y]] ← x;
  else
    right[p[y]] ← x;

  if (y ≠ z) {
    key[z] ← key[y];
    copy y's satellite data into z;
  }

  return y;
End
```

Theorem

Theorem

The dynamic-set operations INSERT *and* DELETE *can be made to run in* $\mathcal{O}(h)$ *time on a binary tree of height* h*.*

Thank You for your kind attention!

Books Consulted

1. Chapter 4.3.3 of *Introduction to Algorithms: A Creative Approach* by Udi Manber.

2. Chapter 12 of *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Questions!!