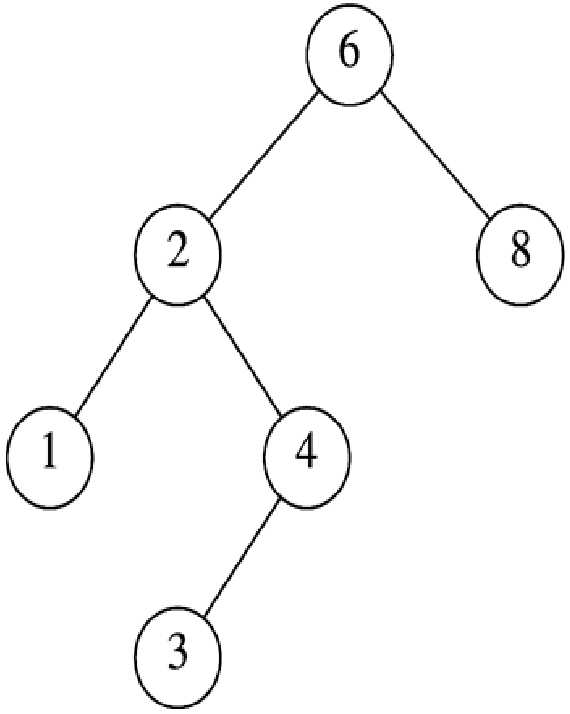
Binary Search Tree (Data Structure)

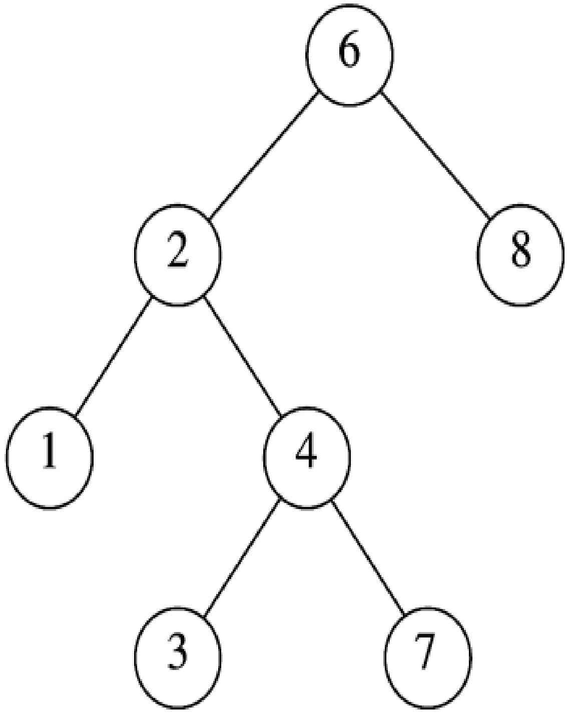
Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Binary Search Trees

- Binary search tree
  - Every element has a **unique key**, at most **two children** (left and right), and **a parent**.
  - The key in the **left child** is **smaller** than the key in the parent.
  - The key in a **right child** is **larger** than the key in the parent.
  - The left and right subtrees are also binary search trees.

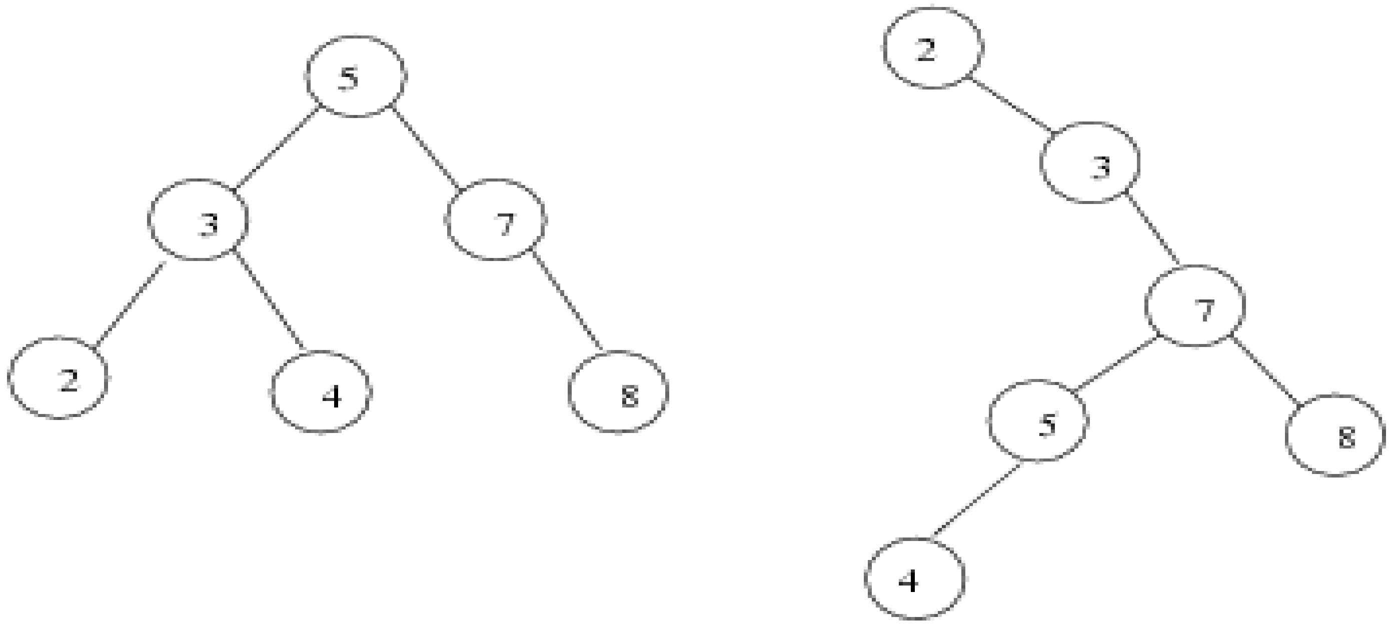
Binary Search Tree





Which one is NOT a BST?

Binary Search Tree

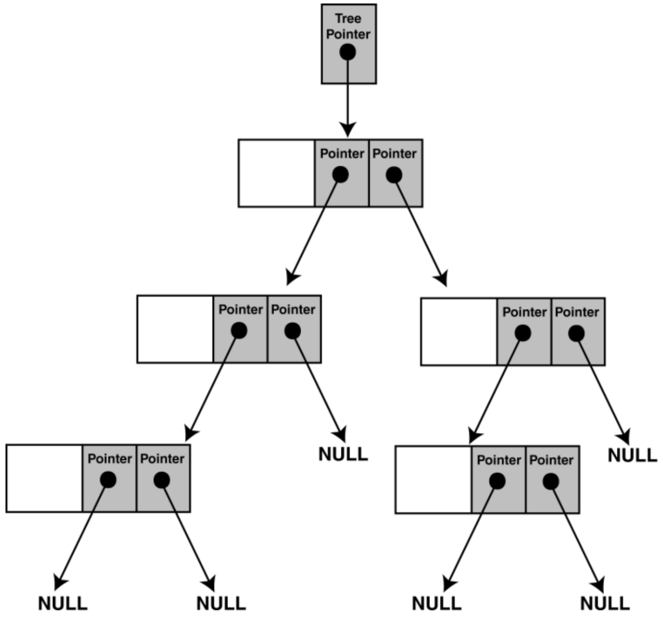


Two binary search trees representing the same key set

Binary Search Tree

- A binary tree is represented as a non-linear linked list where each node may point to two other nodes.

```
struct Node {
   int key;
   Node* left;
   Node* right;
};
```

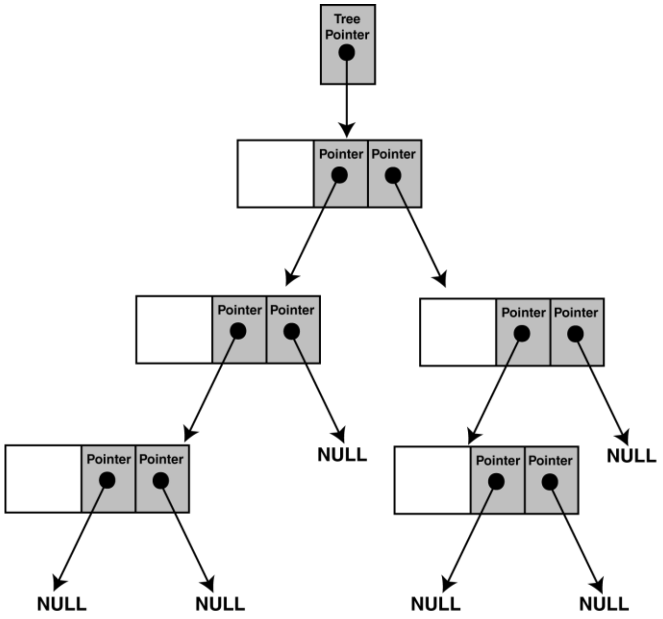


Binary Search Tree

- It is anchored at the top by a *tree pointer*, which is like the head pointer in a linked list.
- The first node in the list, which has no parent, is called the *root node*.
- The root node has pointers to two other nodes, which are called *children*, or *child nodes*.

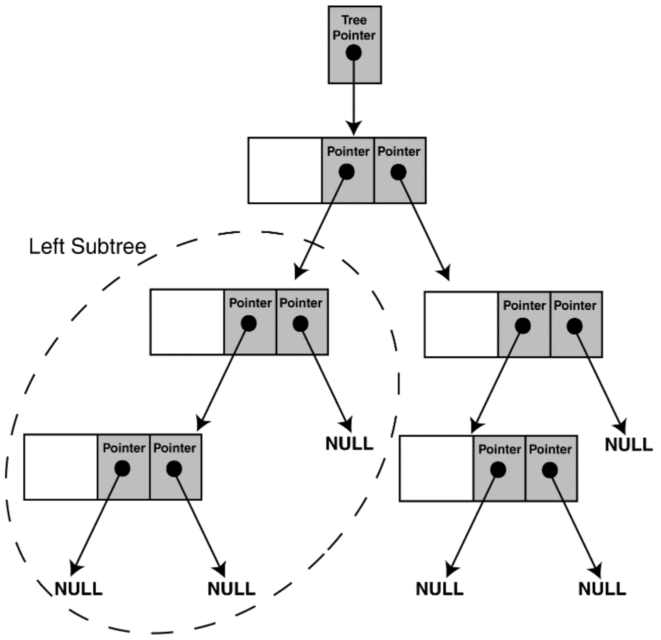
Binary Search Tree

- A node that has no children is called a *leaf node*.
- All pointers that do not point to a node are set to NULL.



Binary Search Tree

- Binary trees can be divided into *subtrees*. A subtree is an entire branch of the tree, from one particular node down.



Binary Search Tree

- Viewed as data structures that can support dynamic set of operations.
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - Dictionaries.
  - Priority Queues.
- Basic operations take time proportional to the height of the tree – $O(h)$.

Binary Search Tree

- Binary search trees are excellent data structures for searching large amounts of information. They are commonly used in database applications to organize key values that index database records.

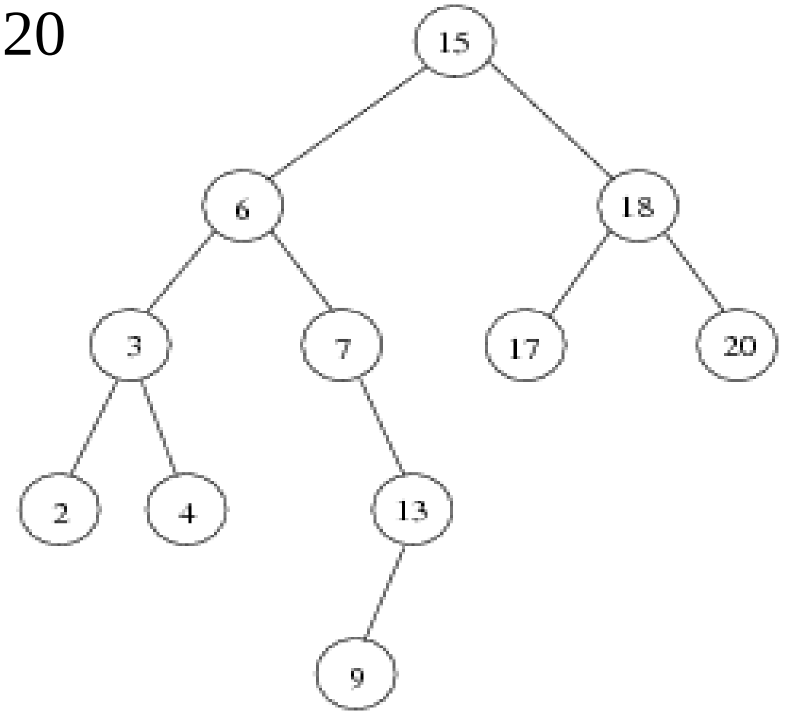
Traversing Binary Tree

- There are three common methods for traversing a binary tree and processing the value of each node:
  - *inorder*
  - *preorder*
  - *postorder*
- Each of these methods is best implemented as a recursive function.

Inorder Traversal

1. The node’s left subtree is traversed.
2. The node’s data is processed.
3. The node’s right subtree is traversed.

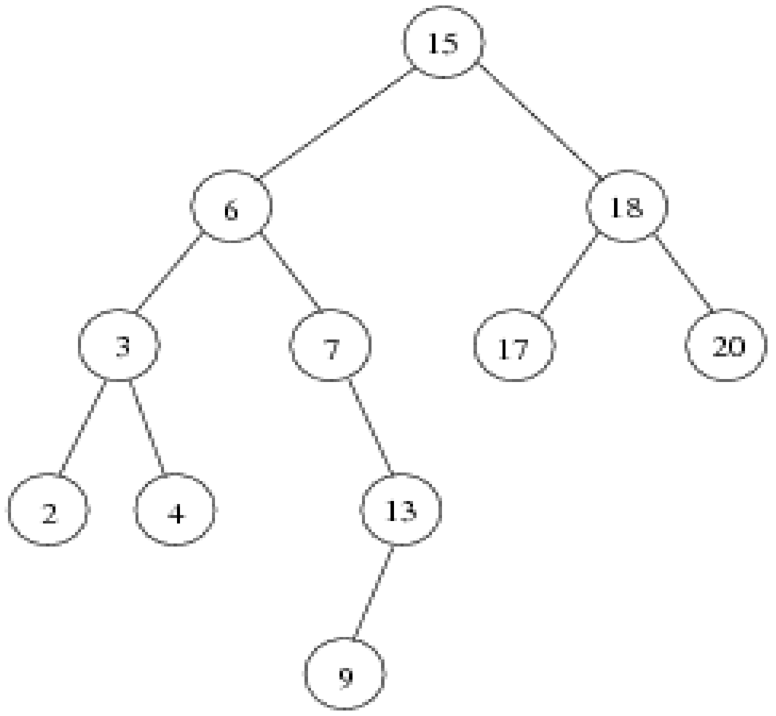
2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20



Preorder Traversal

1. The node's data is processed.
2. The node's left subtree is traversed.
3. The node's right subtree is traversed.

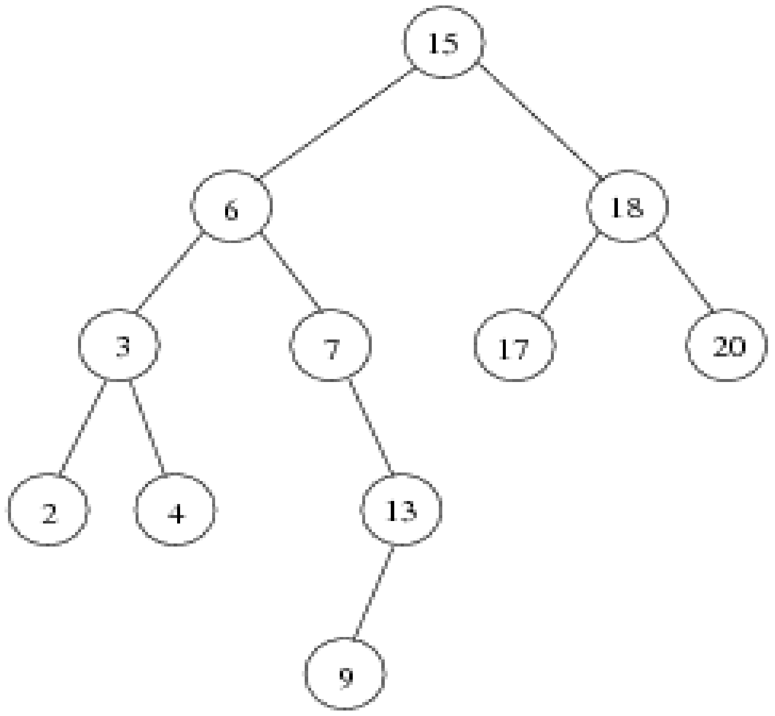
15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20



Postorder Traversal

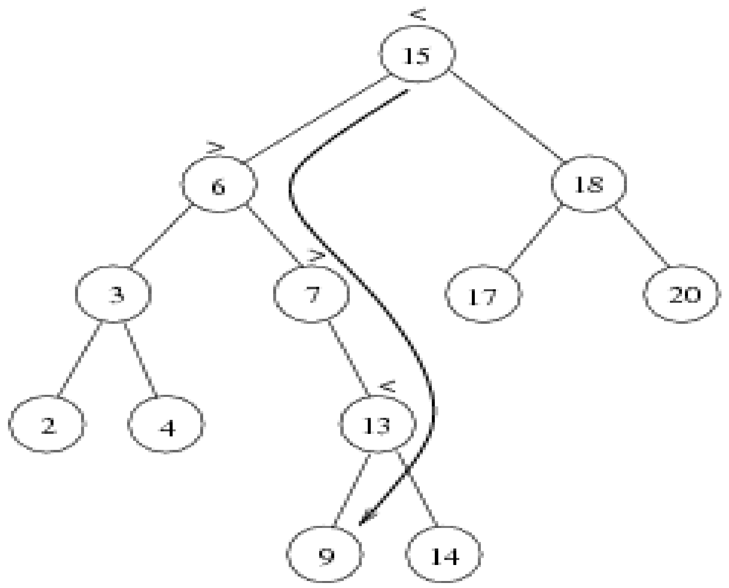
1. The node's left subtree is traversed.
2. The node's right subtree is traversed.
3. The node's data is processed.

2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15



Searching Binary Search Tree

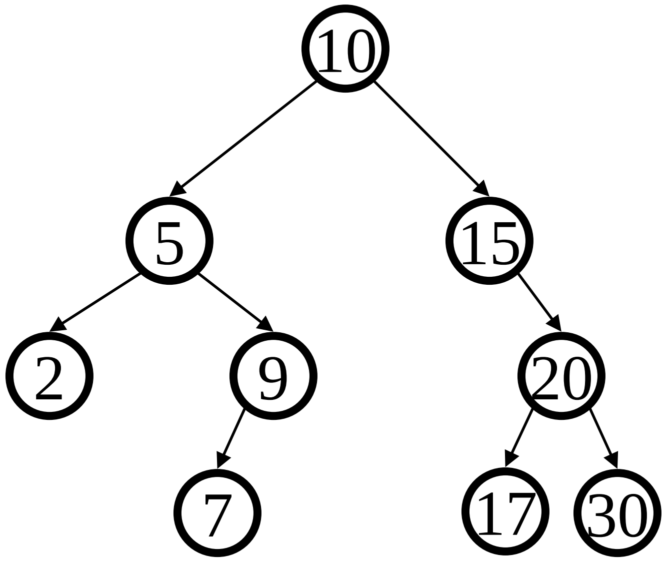
Example: Search for 9 ...



Search for 9:

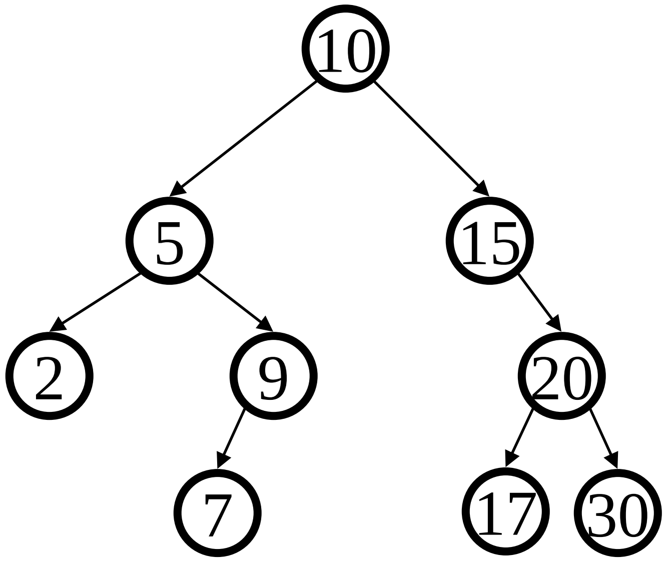
1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Recursive Search



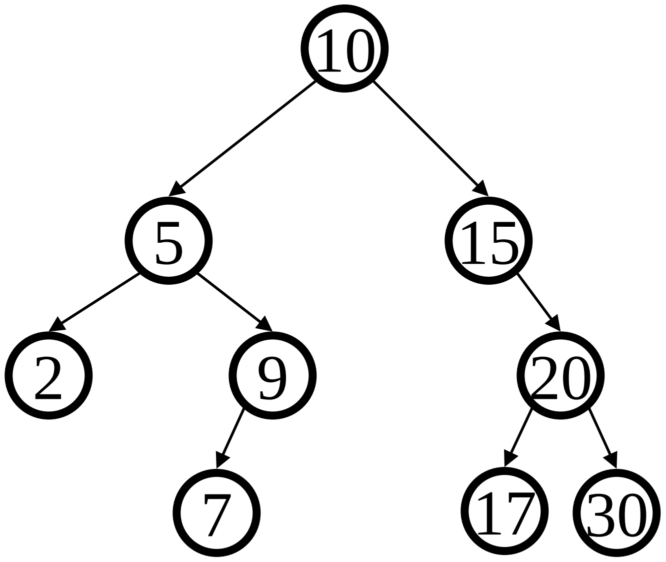
```
Node* search(int key, Node * tree)
{
   if (tree == NULL || tree->key == key)
     return tree;
   else if (key < tree->key)
     return search(key, tree->left);
   else if (key > tree->key)
     return search(key, t->right);
}
```

Iterative Search



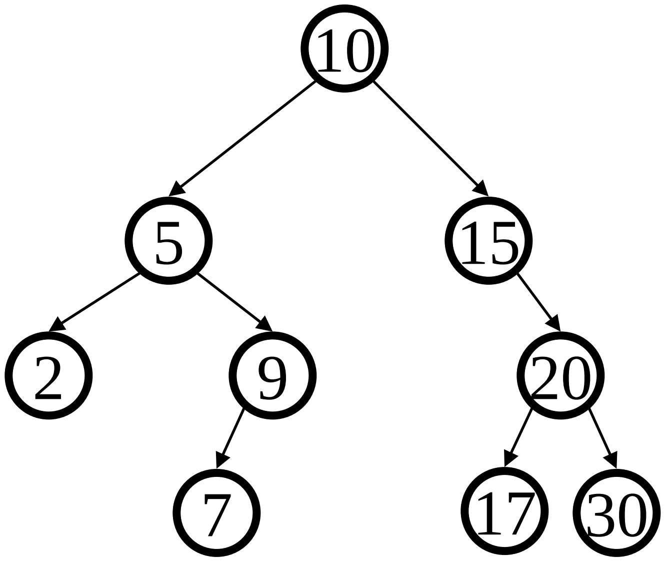
```
Node* search(int key, Node * tree)
{
  while (tree != NULL && tree->key != key)
  {
    if (key < tree->key)
      tree = tree->left;
    else
      tree = tree->right;
  }

  return tree;
}
```

Find Minimum



```
Node* findMin(Node * tree)
{
  if (tree == NULL || tree->left == NULL)
        return tree;
  else return findMin(tree->left);
}
```

Find Maximum



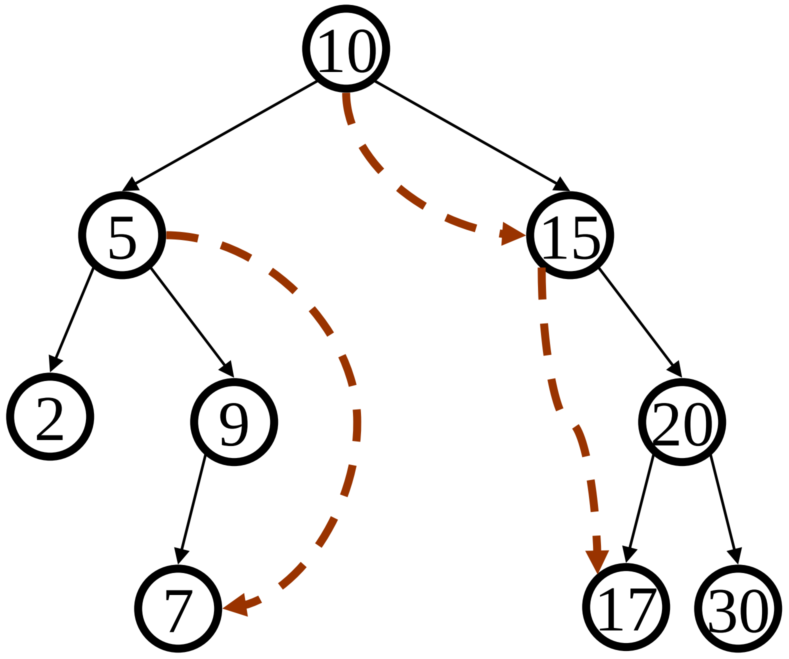
$$\begin{tikzcd}
 & & & 10 \arrow[dll] \arrow[dr] & & \\
 & 5 \arrow[dl] \arrow[dr] & & & 15 \arrow[d] & \\
2 & & 9 \arrow[dl] & & 20 \arrow[dl] \arrow[dr] & \\
 & 7 & & 17 & & 30
\end{tikzcd}$$

```
Node* findMax(Node * tree)
{
   if (tree == NULL || tree->right == NULL)
         return tree;
   else return findMax(tree->right);
}
```

Successor Node

Next larger node in this node's subtree

```
Node * succ(Node * tree) {
  if (tree->right == NULL)
    return NULL;
  else
    return findMin(tree->right);
}
```

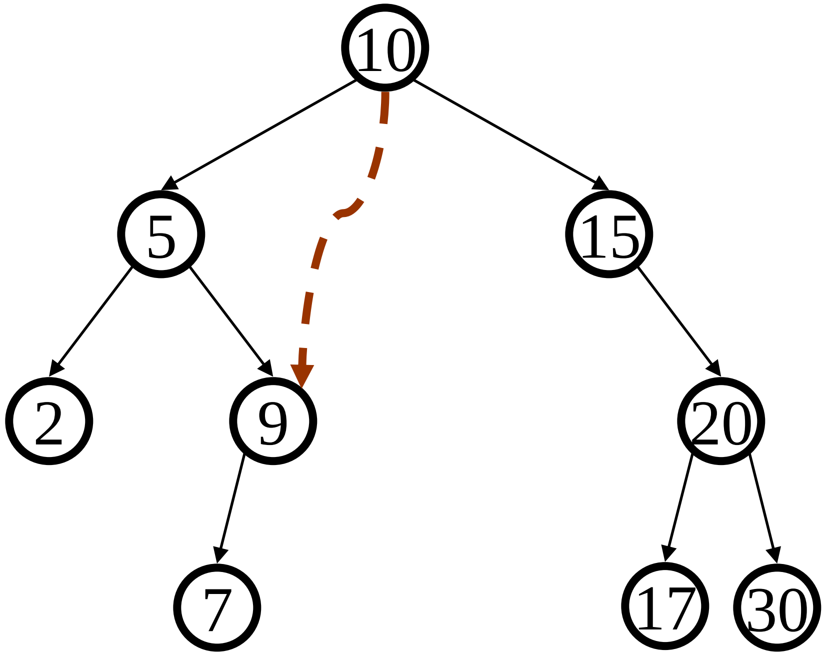


How many children can the successor of a node have?
What is the limitation of this algorithm?

Predecessor Node

Next smaller node in this node’s subtree

```
Node * pred(Node * tree) {
  if (tree->left == NULL)
    return NULL;
  else
    return findMax(tree->left);
}
```

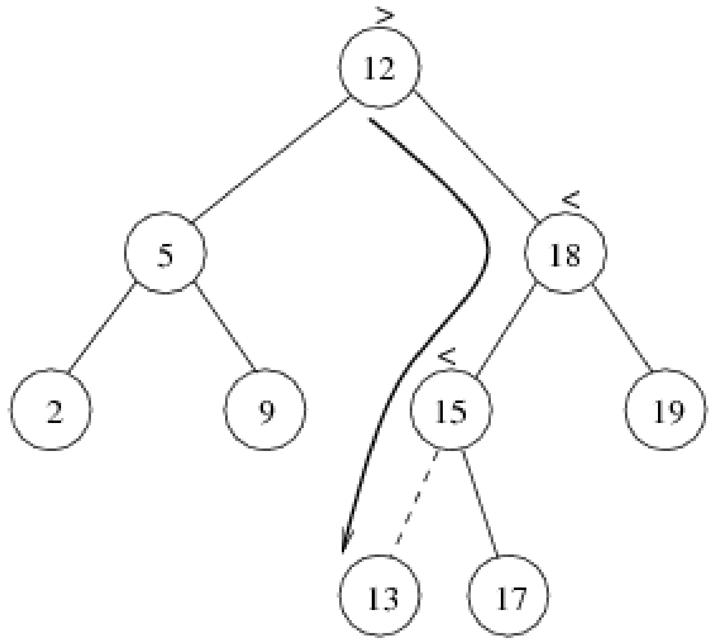


How many children can the predecessor of a node have?

What is the limitation of this algorithm?

Insert a Node

- Proceed down tree as in search
- If new key not found, then insert a new node at last spot traversed



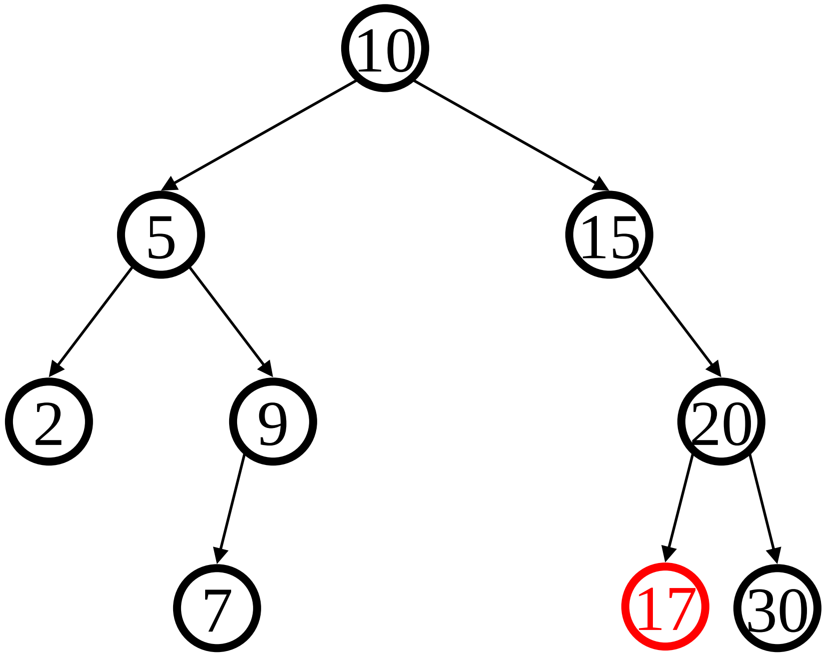
Inserting 13

```
void insert(int key, Node *& tree)
{
  if ( tree == NULL ) {
    tree = new Node(key);

  } else if (key < tree->key) {
    insert( key, tree->left );

  } else if (key > tree->key) {
    insert( key, tree->right );
  }
}
```

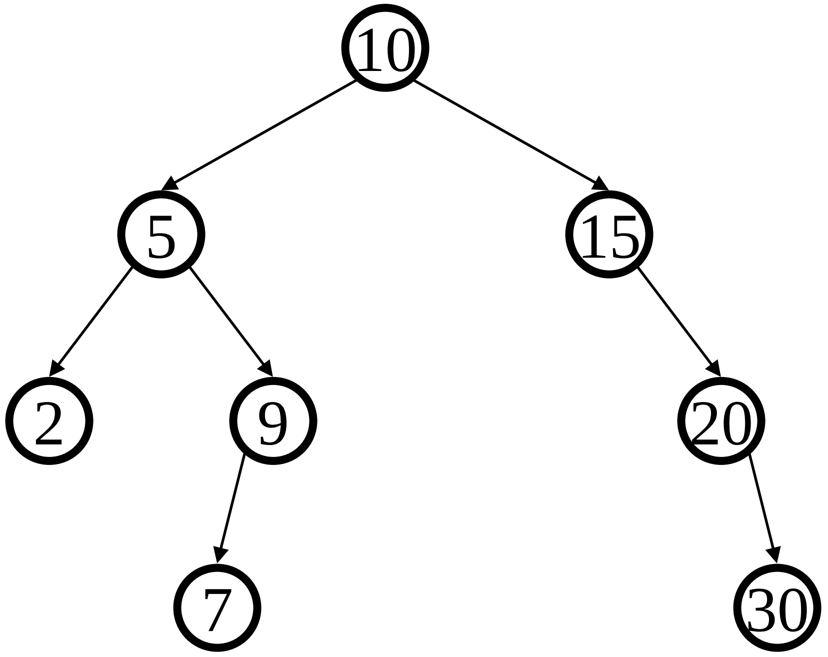
Deleting a Node

- There are three possible situations when we are deleting a node:
  - Leaf node has no child
  - Non-leaf node has only one child
  - Non-leaf node has two children.

Deletion - Leaf Node



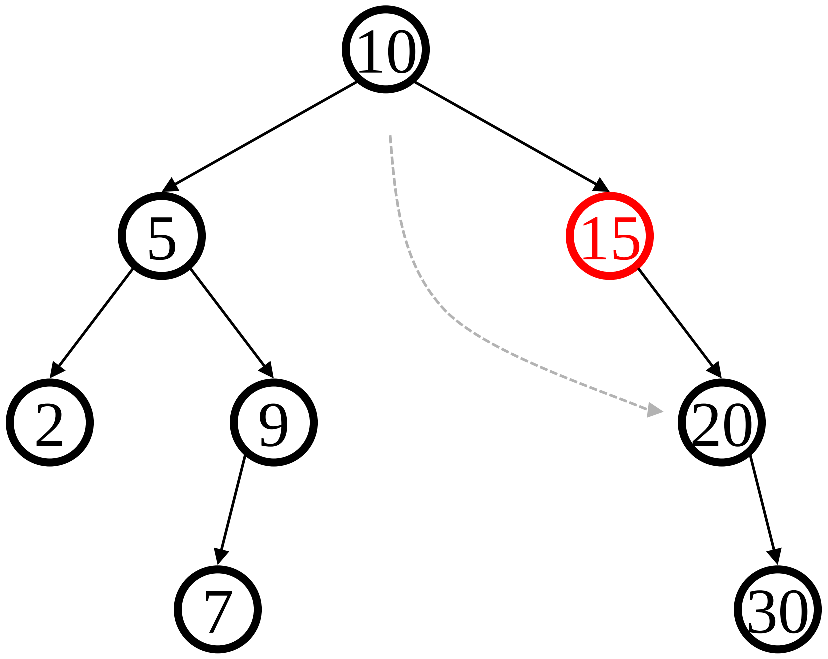
Before

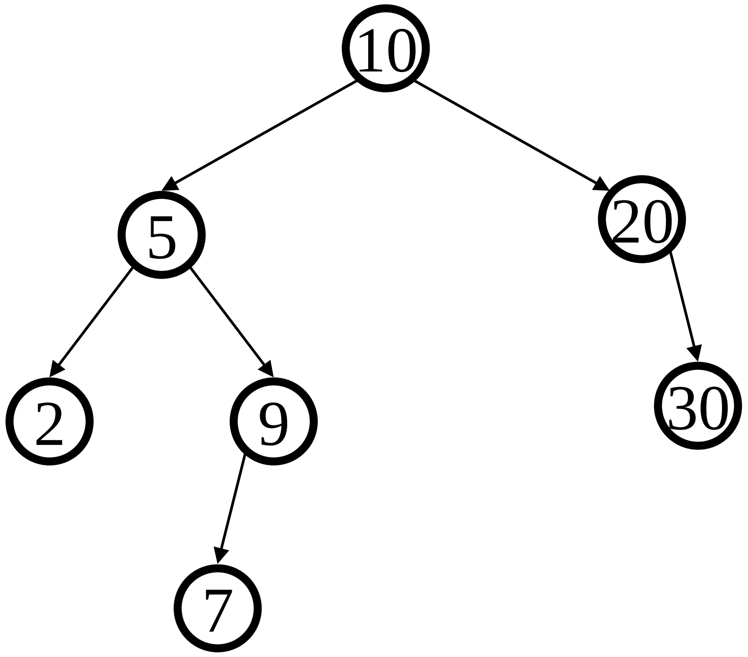


After

Deletion – Non-leaf One Child

Deletion Strategy: Bypass the node being deleted



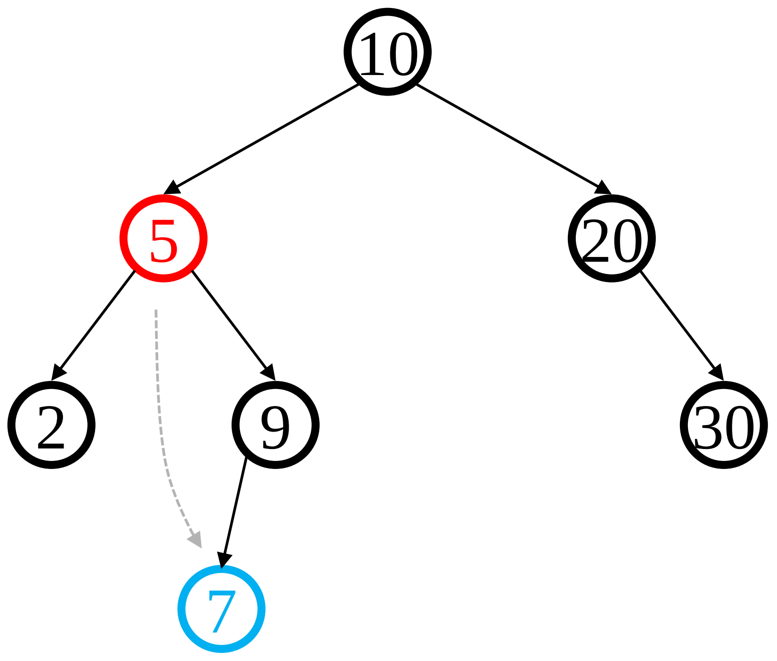


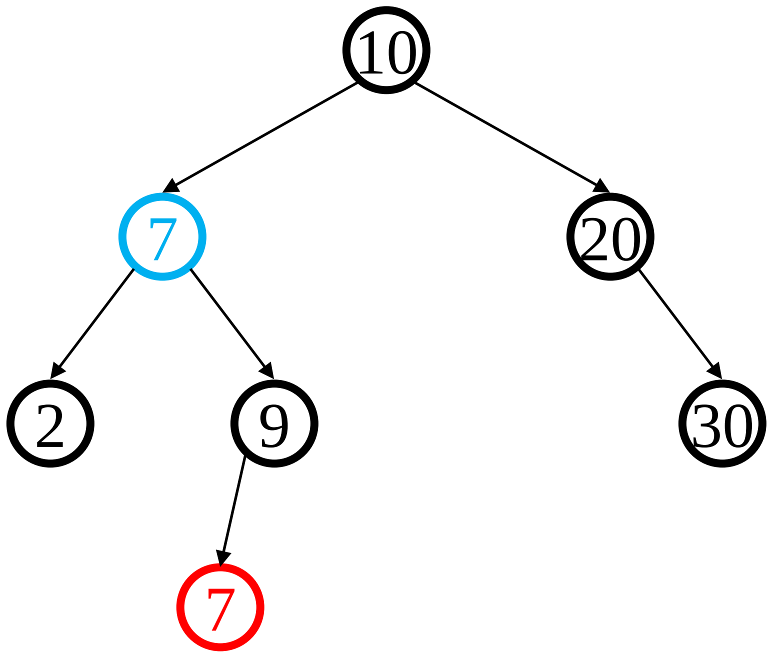
Before

After

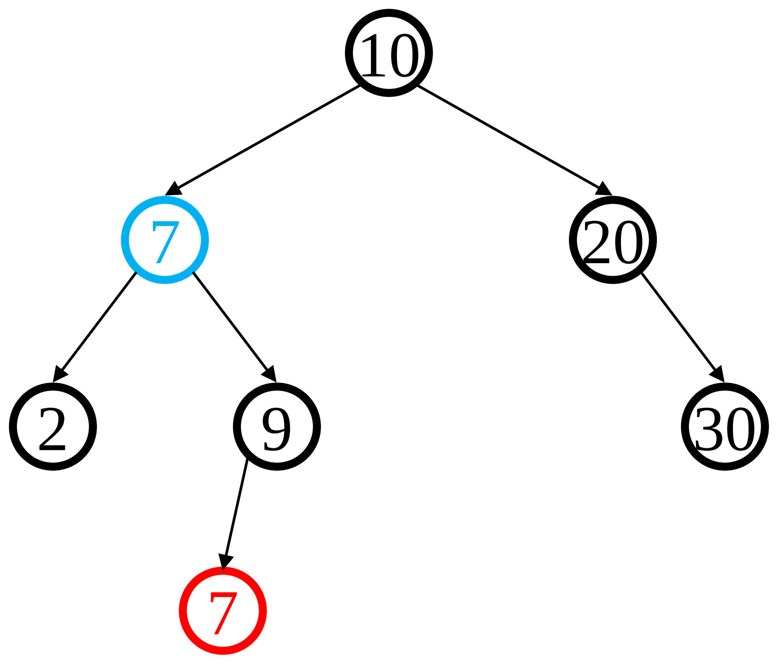
Deletion – Non-leaf Two Children

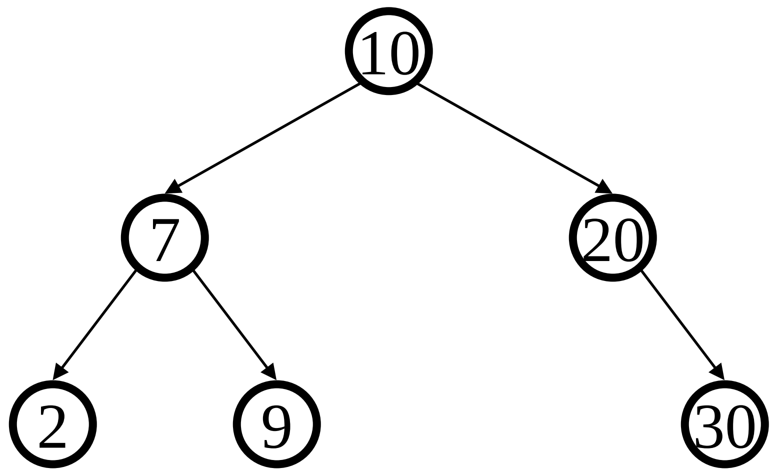
Deletion Strategy: Replace the node with the smallest node in the right subtree, i.e., the successor, whose value is guaranteed to be between left and right subtrees.





Deletion – Non-leaf Two Children





Could we have used predecessor instead?

Deletion Code

```
void delete(int key, Node *& tree) {
  if(tree == NULL) return;
  if(key < tree->key)
       delete(key, tree->left);
  else if(key > tree->key)
       delete(key, tree->right);
  else if (tree->left != NULL && tree->right != NULL) {
       tree->key = findMin(tree->right)->key;
       delete(tree->key, tree->right);
  }
  else {
       Node* nodeToDelete = tree;
       tree = (tree->left != NULL) ? tree->left : tree->right;
       delete nodeToDelete;
  }
}
```