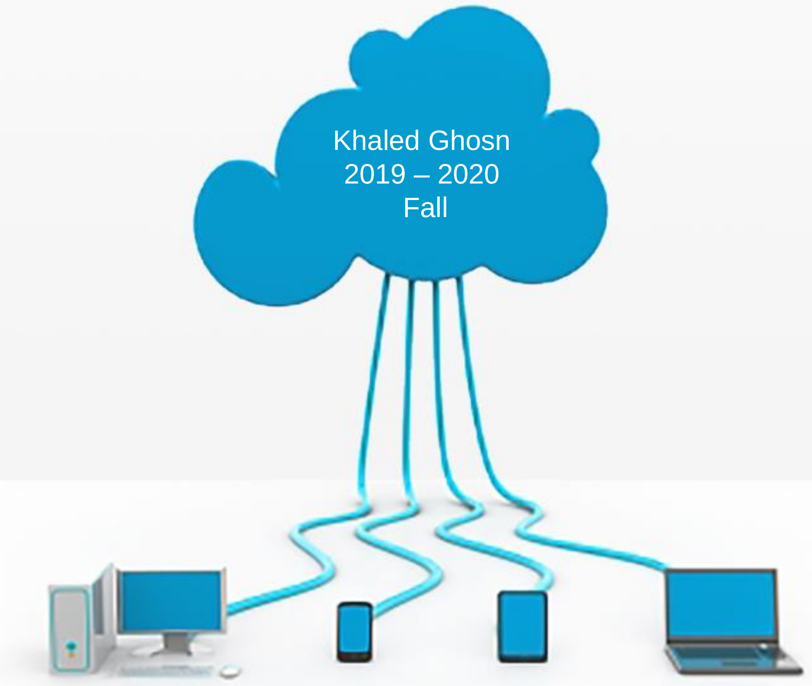
ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON

AUL

Binary Search Trees

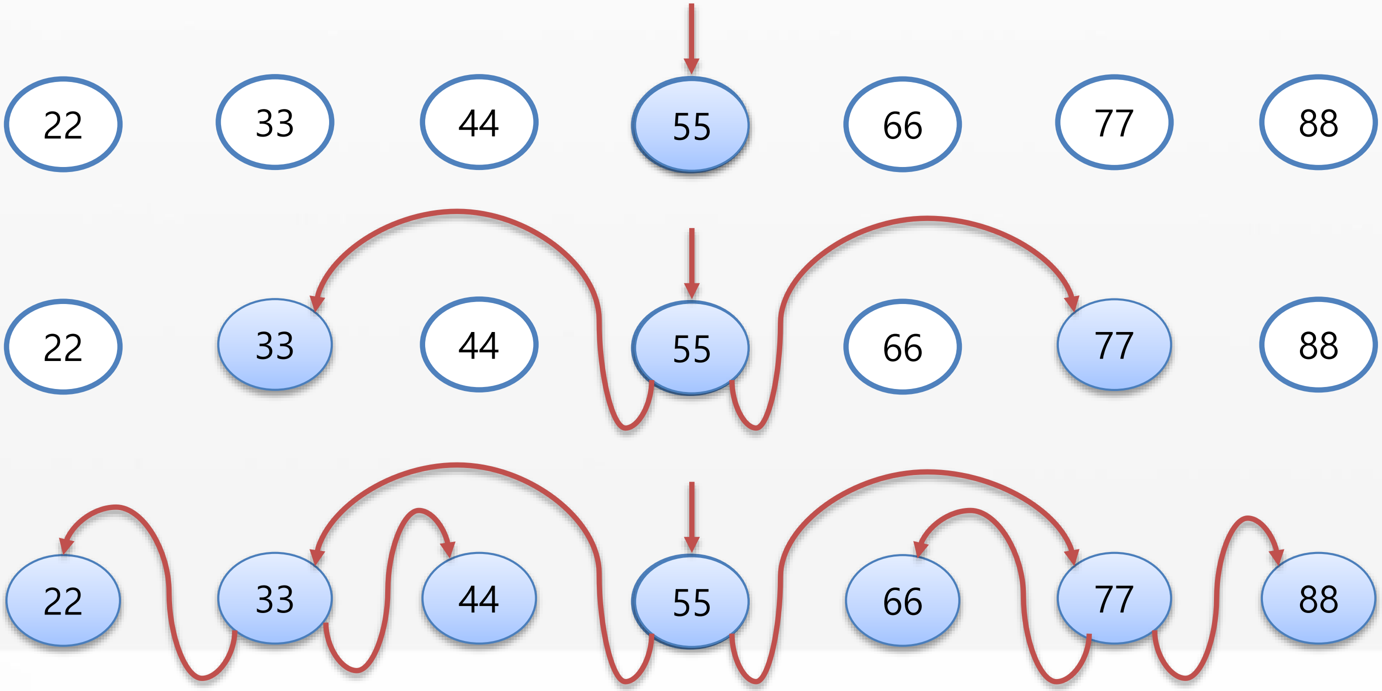
Introduction & Operations



Khaled Ghosn
2019 – 2020
Fall

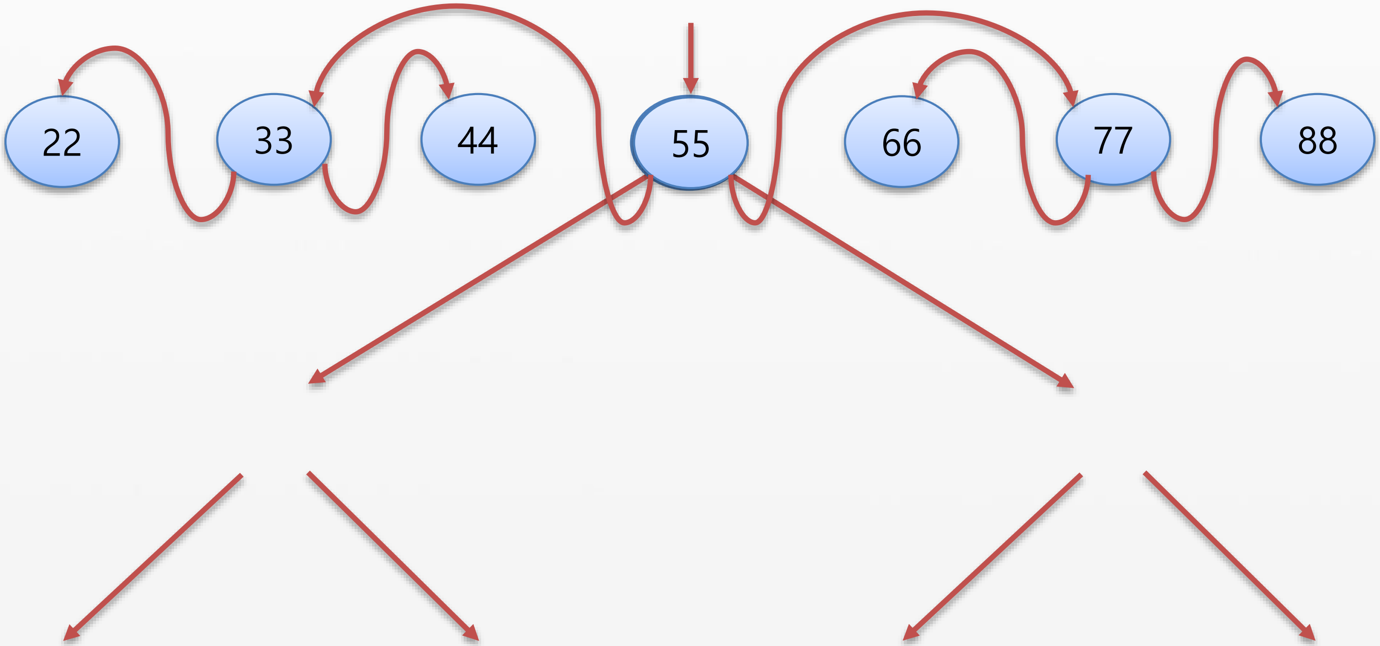
Binary Search Tree

Binary search

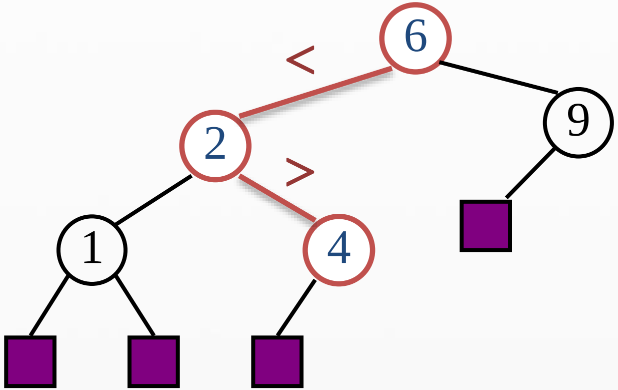


Binary Search Tree

Binary search

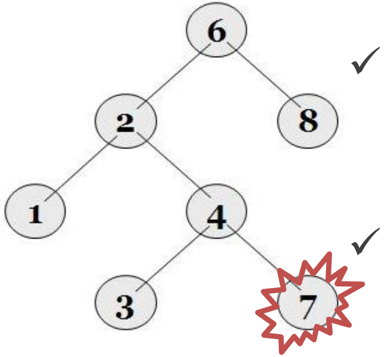


Binary Search Tree



Definition

- ✓ Also called: Ordered or Sorted Tree
- ✓ Is a binary tree
- ✓ A total order is defined on these values
  - every two values can be compared with each other
- ✓ The left sub-tree (if any) is a binary search tree and all elements are less than the root element
  - left sub-tree of a node contains only values lesser, than the node's value
- ✓ The right sub-tree (if any) is a binary search tree and all elements are greater than the root element
  - right sub-tree of a node contains only values greater, than the node's value



Not a *binary search tree*, but a *binary tree*

Binary Search Tree

Properties

- ✓ Nodes stores only the *keys*
  - keys may be any (homogenous) comparable

- ✓ A binary search tree (**BST**) is based on binary tree, but with the following additional properties:
  - The left sub-tree of a node contains only nodes with keys less than the node's key
  - The right sub-tree of a node contains only nodes with keys greater than the node's key

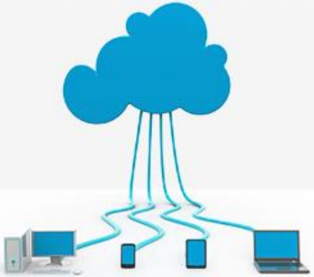
- ✓ Both the left and right sub-trees must also be binary search trees

- ✓ No duplicate values

Binary Search Tree

What for binary search trees are used?

- ✓ Binary search tree is used to construct map data structure
- ✓ In practice, data can be often associated with some unique key
  - For instance, in the phone book such a key is a telephone number. Storing such a data in binary search tree allows to look up for the record by key faster, than if it was stored in unordered list
- ✓ Also, BST can be utilized to construct set data structure, which allows to store an unordered collection of unique values and make operations with such collections



Binary Search Tree

Used *Everywhere*

Anywhere we need to ***find*** things fast based on a ***key***

- Arrays
- Sets
- Dictionaries
- Router tables
- Page tables
- Representing expressions
- Symbol tables
- C++ structures
- …

Invented in 1960 by Windley, Booth, Colin, and Hibbard

Binary Search Tree

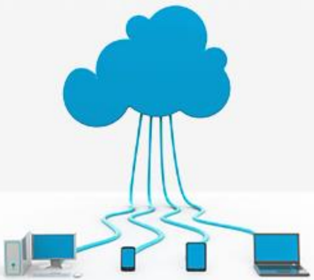
Performance

	Average	Worst Case
Space	O (n)	O (n)
Search (time)	O (log n)	O (n)
Insert (time)	O (log n)	O (n)
Delete (time)	O (log n)	O (n)

Binary Search Tree

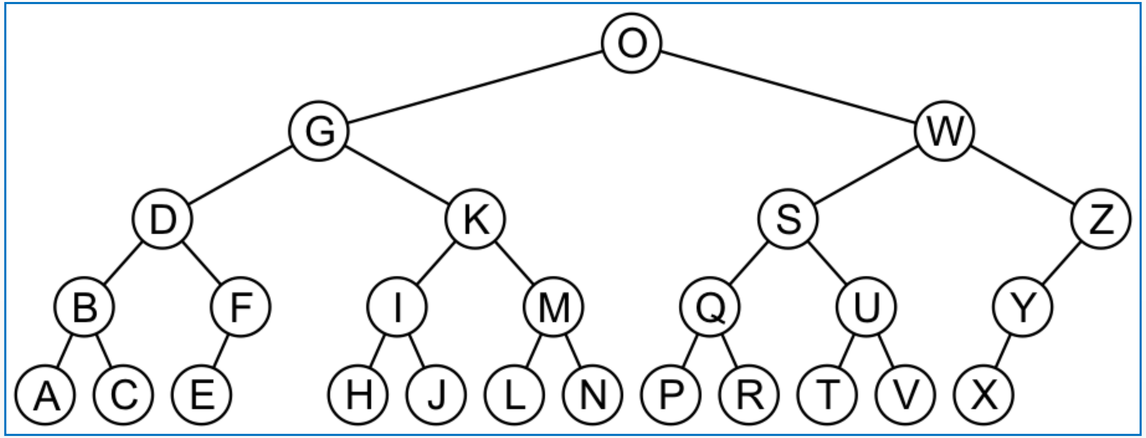
Some types of BST

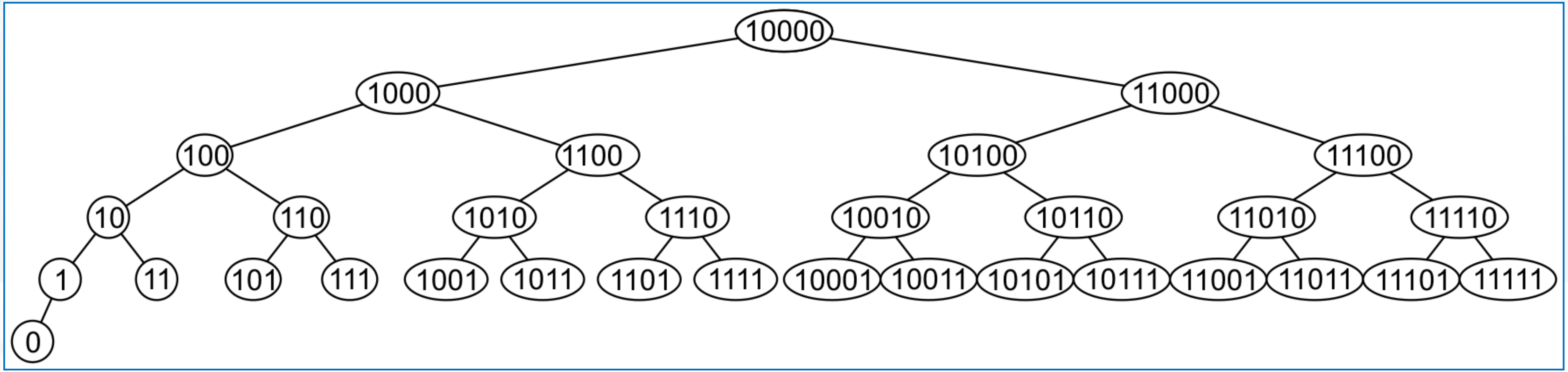
- ✓ Balanced Search Trees: AVL Trees & Red-Black Trees, Splay Trees
- ✓ Splay Trees: exploit most-recently-used (mru) info, automatically frequently accessed elements nearer to the root
- ✓ Heap (Treap) Trees: where nodes holds priority
- ✓ Tango Trees: optimized for fast search
- ✓ B-Trees: Reduce disk access by increasing branching factor



Binary Search Tree

Examples

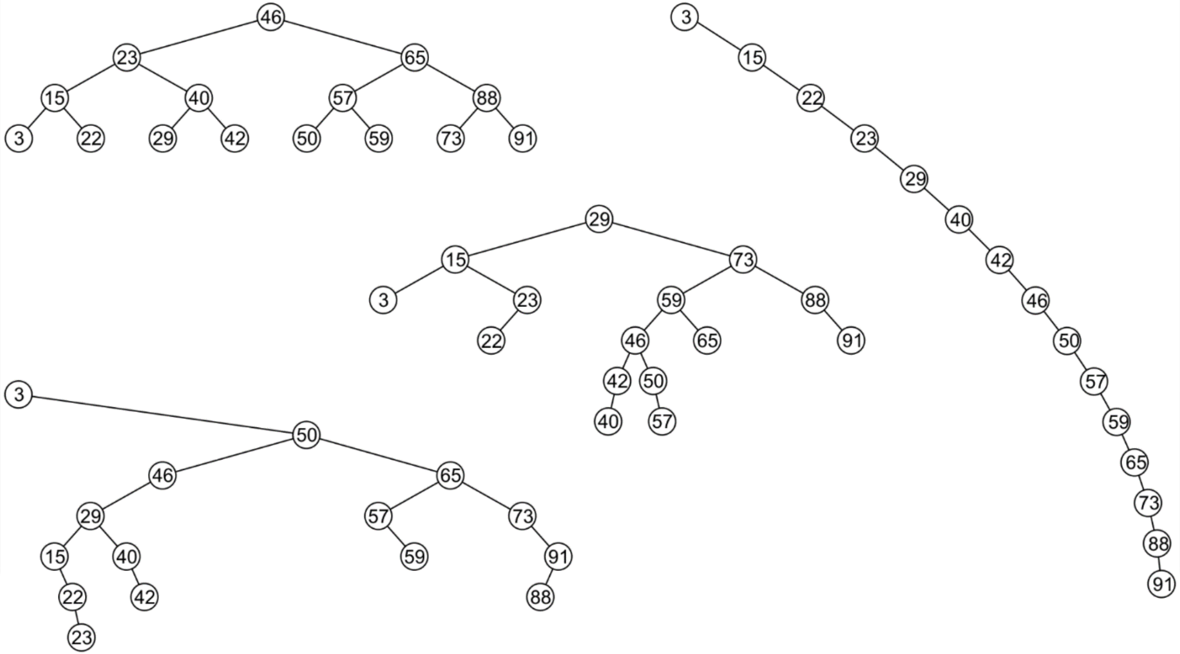




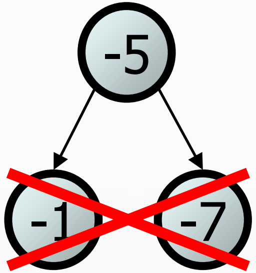
Binary Search Tree

Examples

- ✓ All these binary search trees store the same data

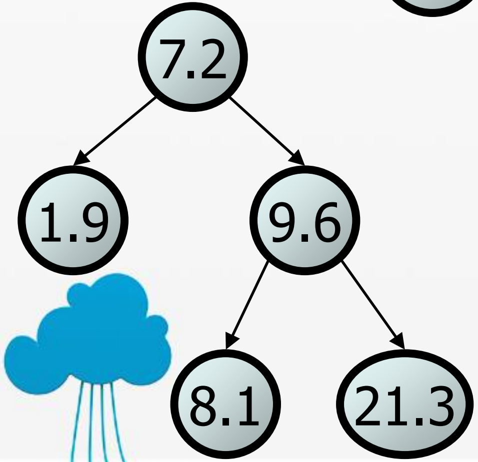


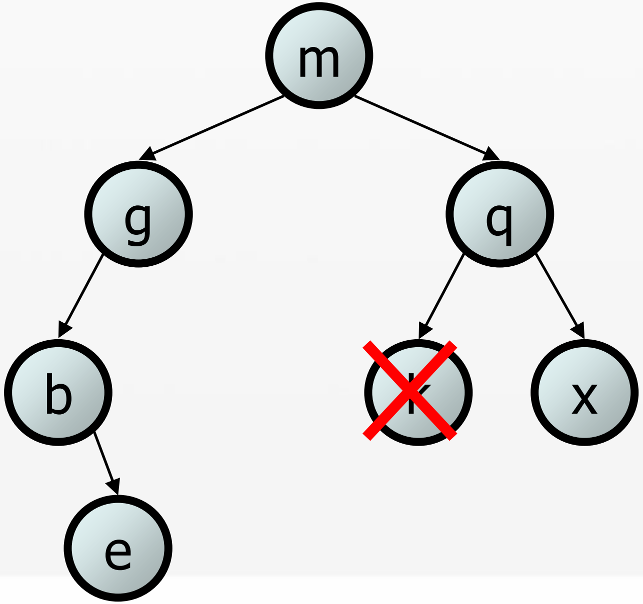
Binary Search Tree

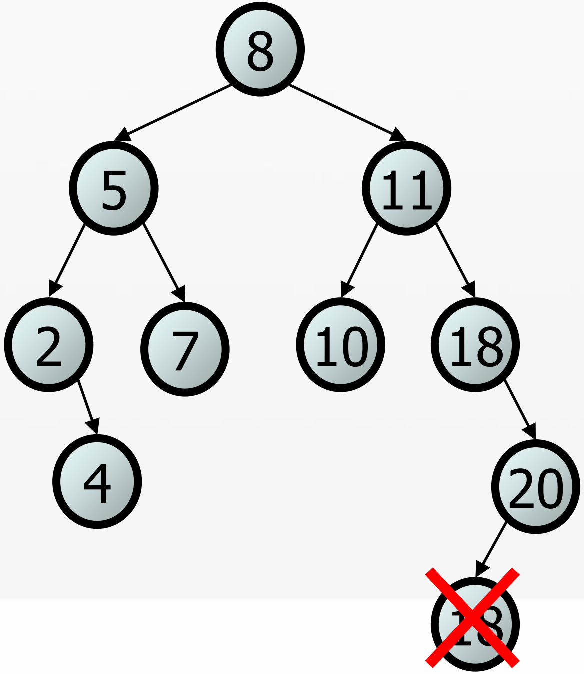


Examples

- ✓ Which of the trees shown are legal binary search trees?

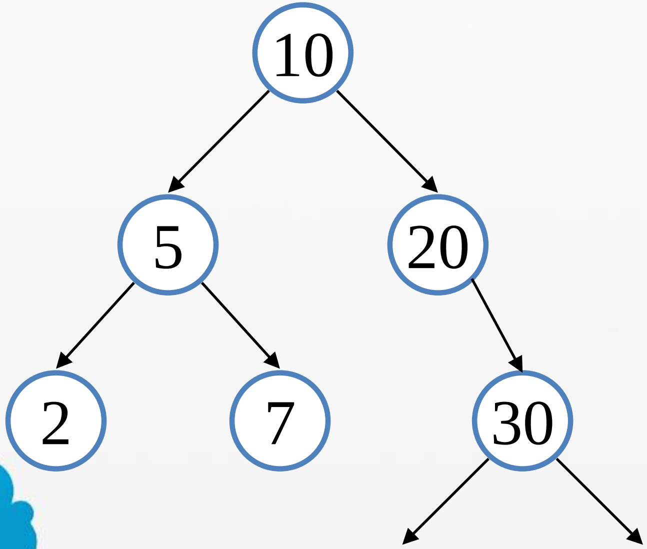


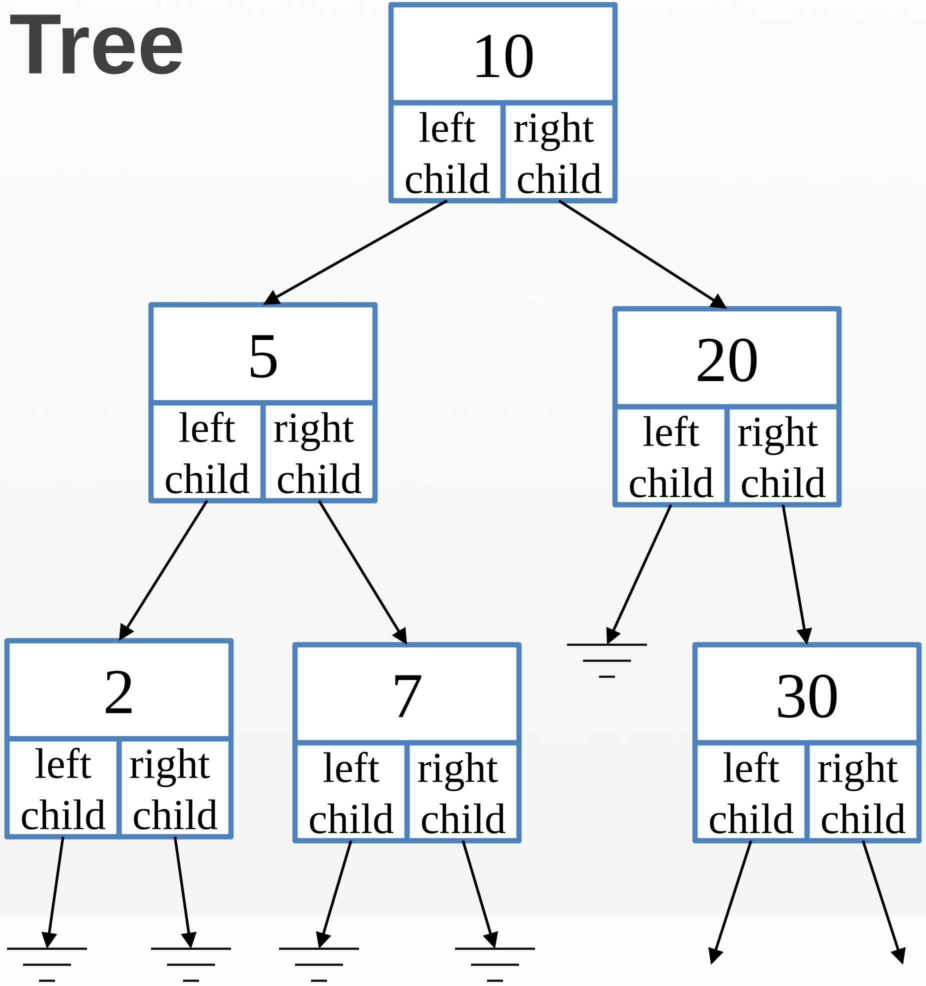


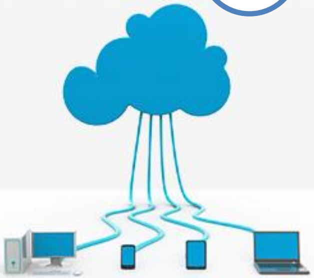


Binary Search Tree

Representation







Binary Search Tree

Operations

- ✓ Creation (a new BST)
- ✓ Insertion (add a new value)
- ✓ Deletion (remove a value)
- ✓ Searching (retrieving values; Lookup operation)
- ✓ Get Values (from BST in order)
- ✓ FindMin
- ✓ FindMax

Binary Search Tree

Adding a value

Adding a value to BST can be divided into two stages:

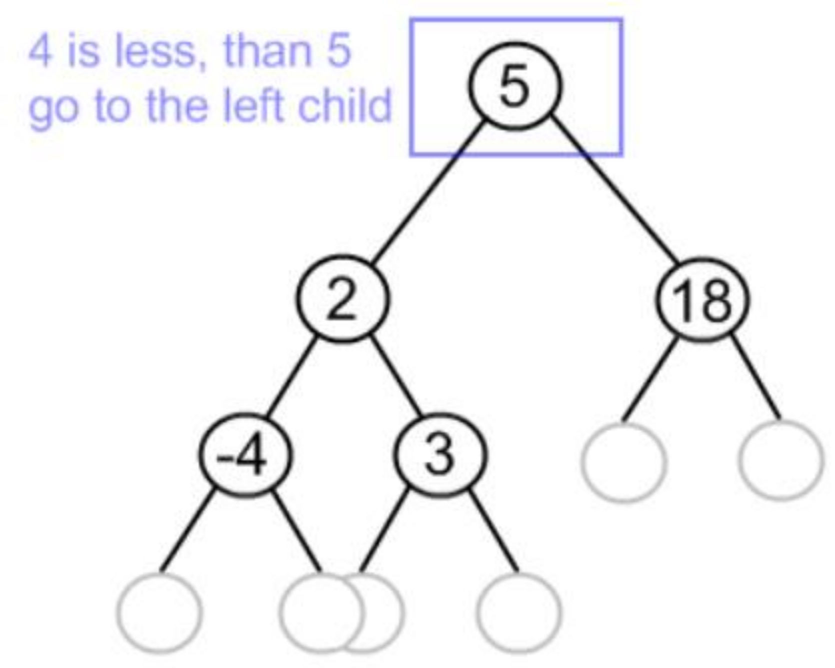
A. Search for a place to put a new element;
    1. check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,
    2. if a new value is less than the node's value:
        - if a current node has no left child, place for insertion has been found;
        - otherwise, handle the left child with the same algorithm
    3. if a new value is greater, than the node's value:
        - if a current node has no right child, place for insertion has been found;
        - otherwise, handle the right child with the same algorithm

B. Insert the new element to this place

Binary Search Tree

Adding a value

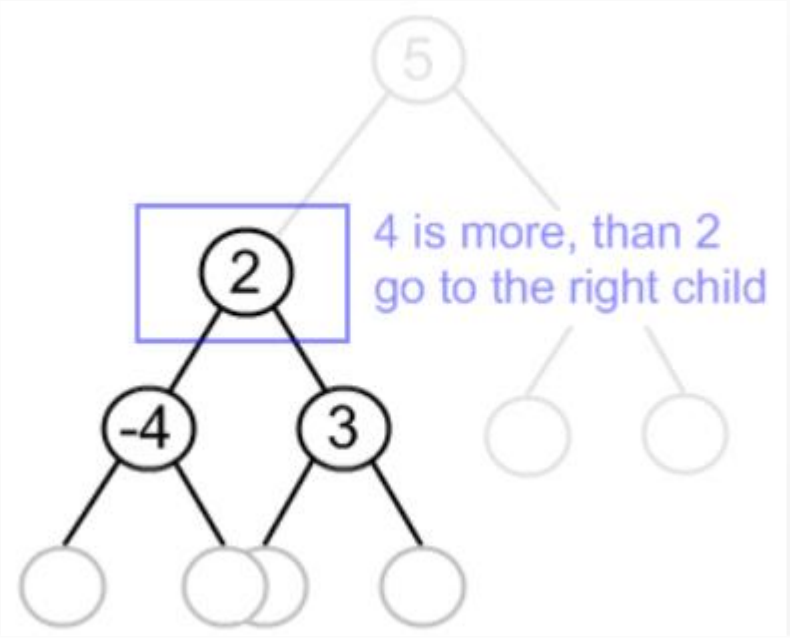
Insert 4 to the tree



Binary Search Tree

Adding a value

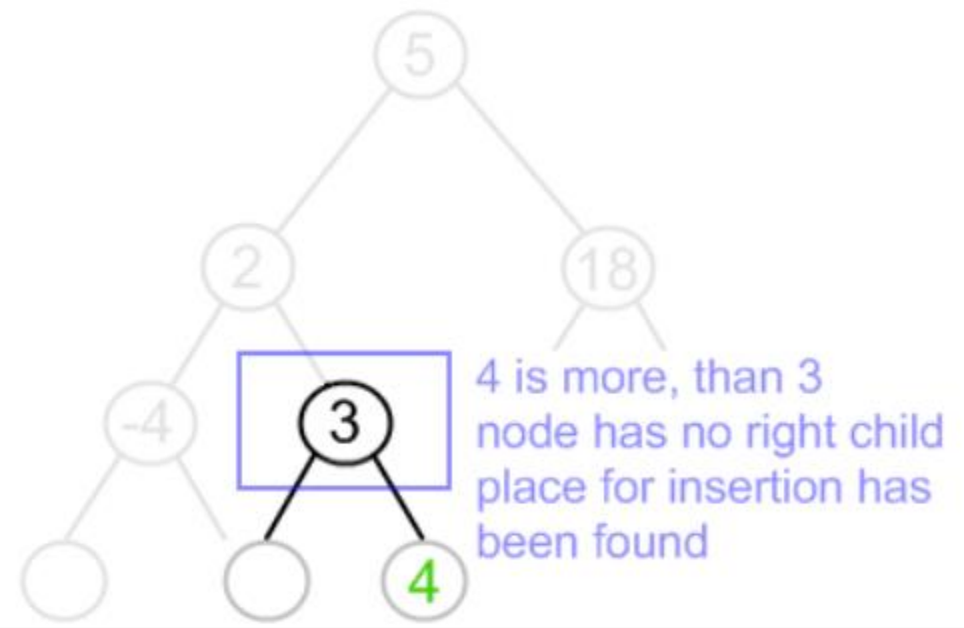
Insert 4 to the tree



Binary Search Tree

Adding a value

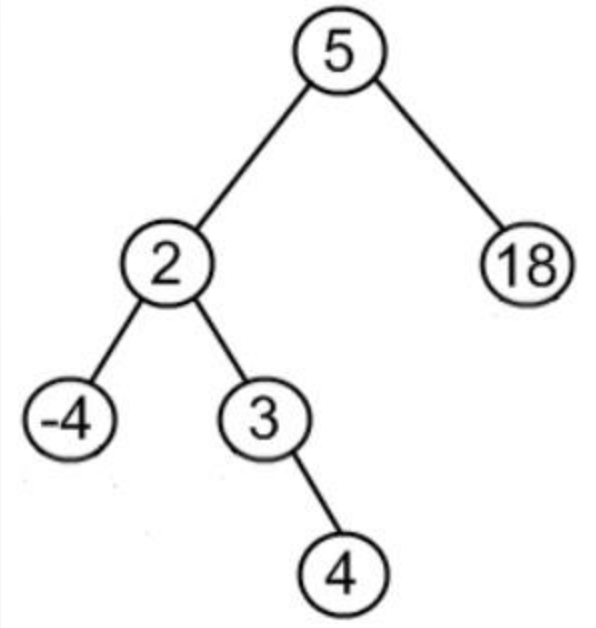
Insert 4 to the tree



Binary Search Tree

Adding a value

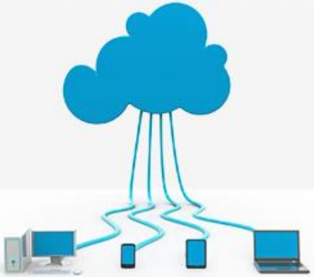
Insert 4 to the tree



Binary Search Tree

Adding a value

```
public class BST {
    // ...
    public boolean Add(int value)
    {
        if (Root == null) {
            Root = new BSTNode(value);
            return true;
        }
        return Root.Add(value);
    }
}
```



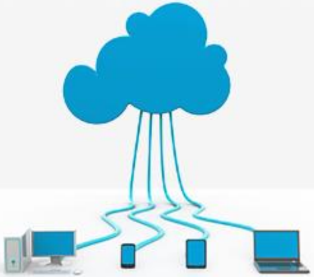
Binary Search Tree

Adding a value

```
public class BSTNode {
    // ...
    public boolean Add(int value) {
        if (value == this.Element) // Duplicate value
            return false;

        else if (value < this.Element) {
            if (Left == null) {
                    Left = new BSTNode(value);  return true;   }
            else    return Left.Add(value);
        }

        else {                              // if (value > this.Element)
            if (Right == null) {
                    Right = new BSTNode(value);  return true; }
            else    return Right.Add(value);
        }
}
```



Binary Search Tree

Lookup operation

- ✓ Searching for a value in a BST is very similar to **Add** operation.

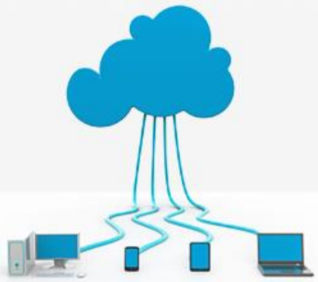
- ✓ Search algorithm traverses the tree "**in-depth**", choosing appropriate way to go, following binary search tree property and compares value of each visited node with the one, we are looking for. Algorithm stops in two cases:
  a) a node with necessary value is found;
  b) algorithm has no way to go.

Binary Search Tree

Lookup operation

Like an add operation (and almost every operation on BST) search algorithm utilizes recursion. Starting from the root:

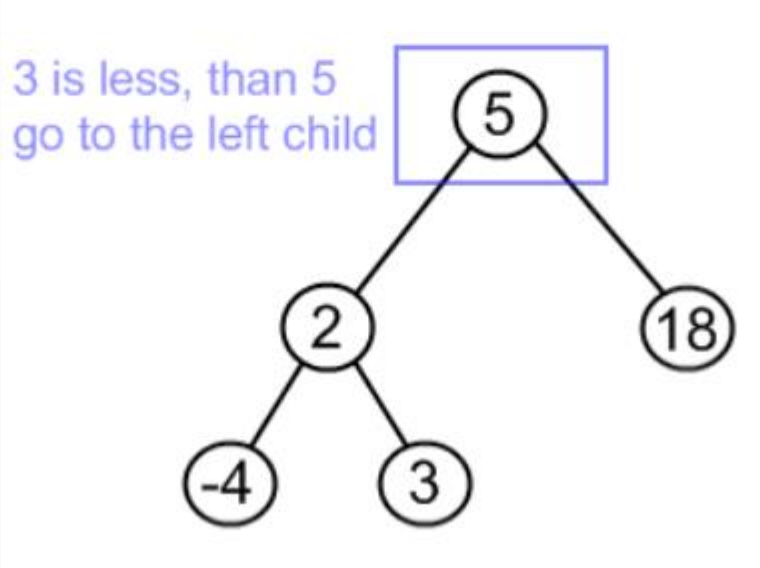
1. check, whether value in current node and searched value are equal. If so, value is found. Otherwise,
2. if searched value is less, than the node's value:
   - if current node has no left child, searched value doesn't exist in the BST;
   - otherwise, handle the left child with the same algorithm
3. if a new value is greater, than the node's value:
   - if a current node has no right child, searched value doesn't exist in the BST;
   - otherwise, handle the right child with the same algorithm



Binary Search Tree

Lookup operation

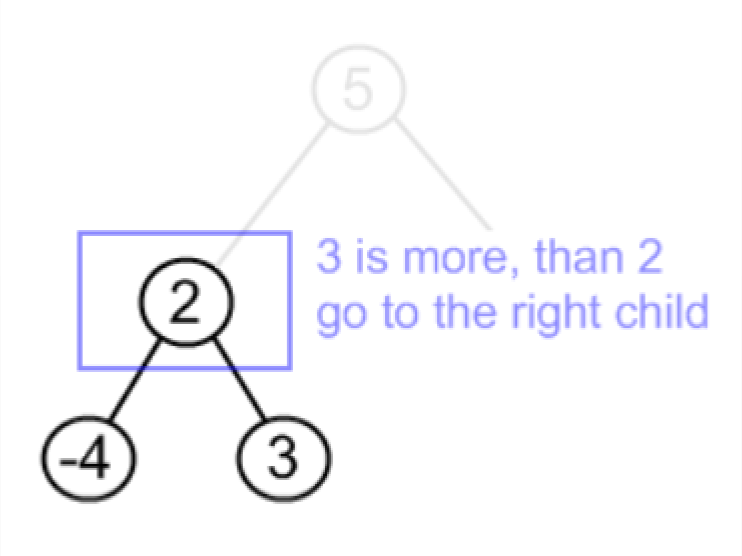
Search for 3 in the tree



Binary Search Tree

Lookup operation

Search for 3 in the tree



Binary Search Tree

Lookup operation

Search for 3 in the tree

Binary Search Tree

Lookup operation

```
public class BST {
    // ...
    public boolean Search ( int value )
    {
        if ( Root == null )
            return false;

        return Root.Search( value );
    }
}
```

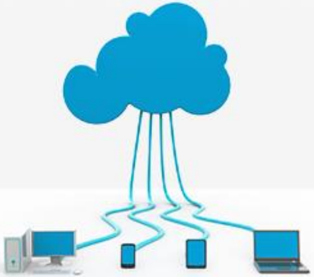
Binary Search Tree

Lookup operation

```
public class BSTNode {
    // ...
    public boolean Search ( int value ) {
        if ( value == this.Element ) // value found
            return true;

        else if (value < this.Element) {
            if (Left == null) {
                    return false;   }
            else    return Left.Search( value );
        }

        else {                              // if (value > this.Element)
            if (Right == null) {
                    return false; }
            else    return Right.Search( value );
        }
}
```



Binary Search Tree

Remove operation

- ✓ Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:
  1. search for a node to remove;
  2. if the node is found, run remove algorithm
     - ➢ 3 cases:
       a) Node to be removed has no children
       b) Node to be removed has one child
       c) Node to be removed has two children

Binary Search Tree

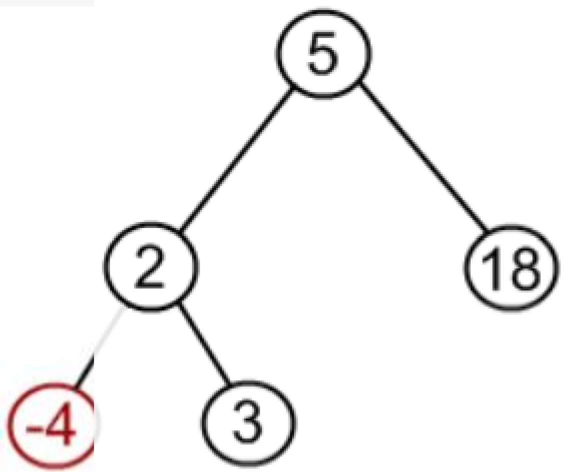
Remove operation

a) <u>Node to be removed has no children</u>

This case is quite simple

Algorithm sets corresponding link of the parent to NULL and disposes the node

Example. Remove -4 from a BST



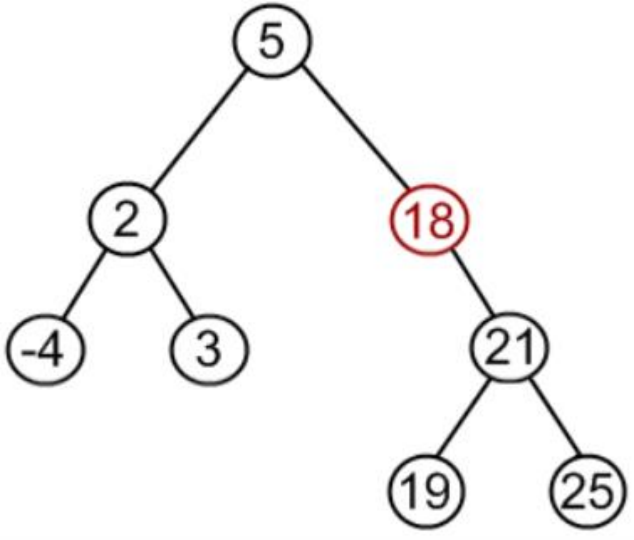
Binary Search Tree

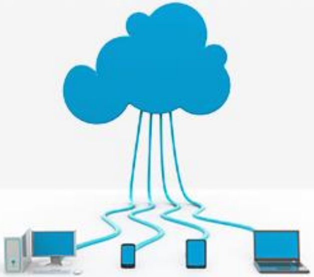
Remove operation

b) <u>Node to be removed has one child</u>

It this case, node is cut from the tree and algorithm links single child (with it's sub-tree) directly to the parent of the removed node

Example. Remove 18 from a BST

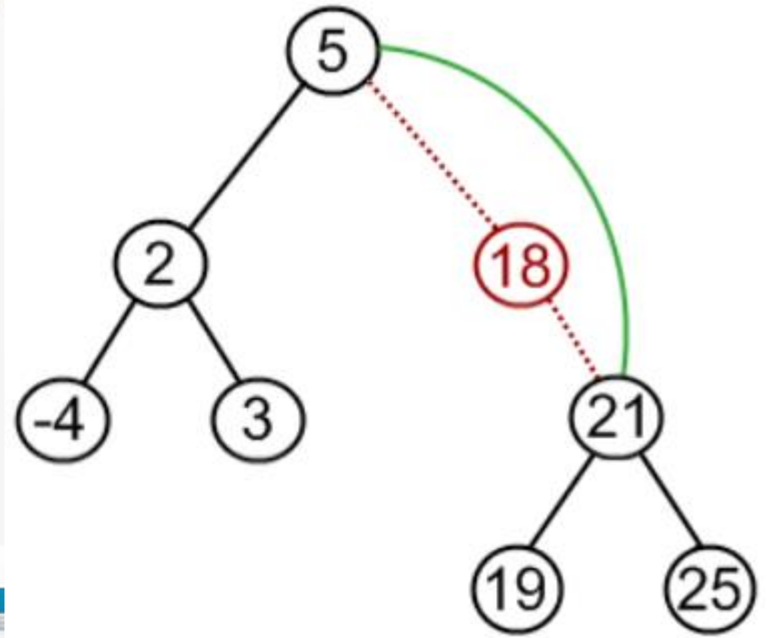


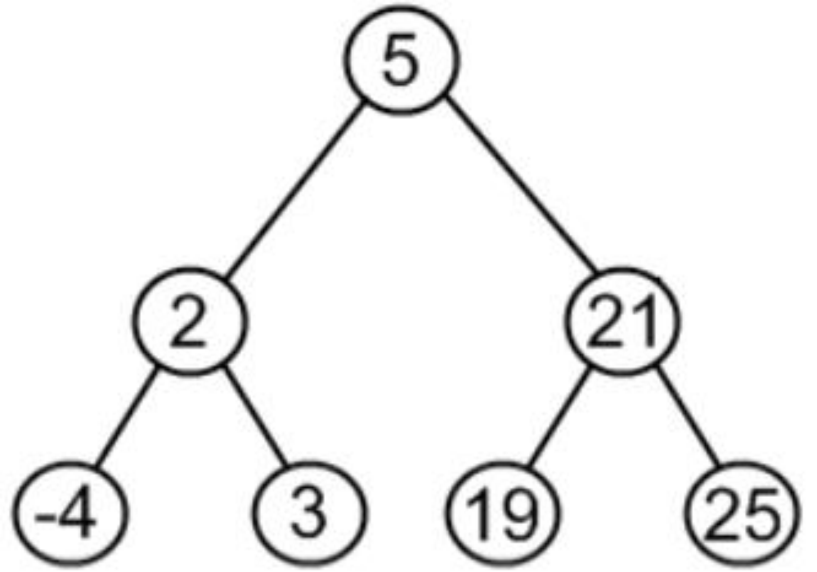


Binary Search Tree

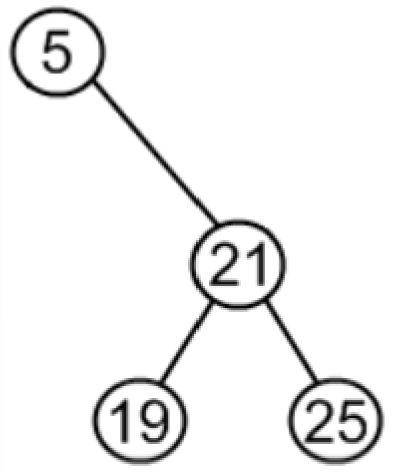
Remove operation

b) Node to be removed has one child





Binary Search Tree



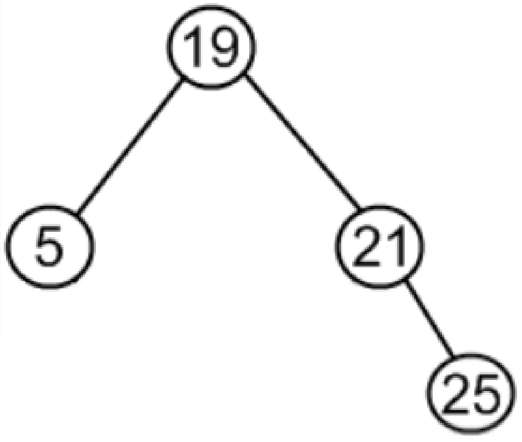
Remove operation

c) <u>Node to be removed has two children</u>

This is the most complex case

To solve it, let us see one useful BST property first

We are going to use the idea, that the same set of values may be represented as different binary-search trees



Binary Search Tree

Remove operation

c) Node to be removed has two children

1) find a minimum value in the right sub-tree;
2) replace value of the node (to be removed) with found minimum
   - now, right sub-tree contains a duplicate!
3) apply remove to the right sub-tree to remove a duplicate

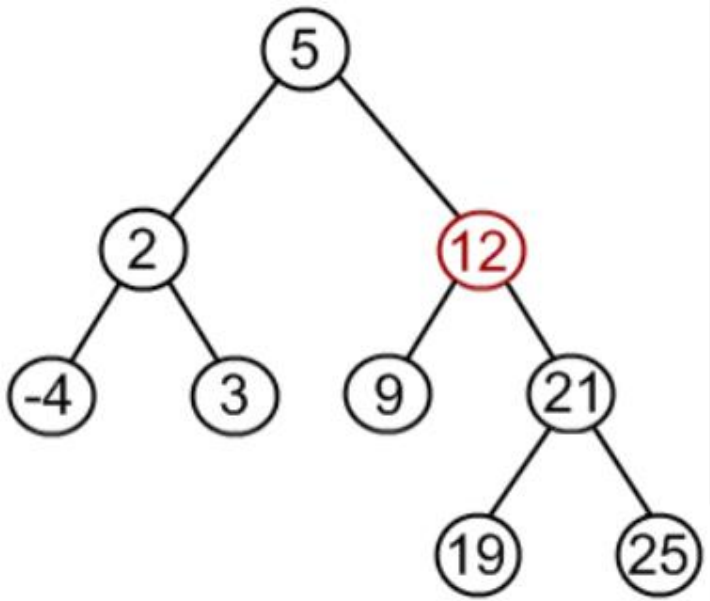
Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only

Binary Search Tree

Remove operation

c) Node to be removed has two children

Example. Remove 12 from a BST

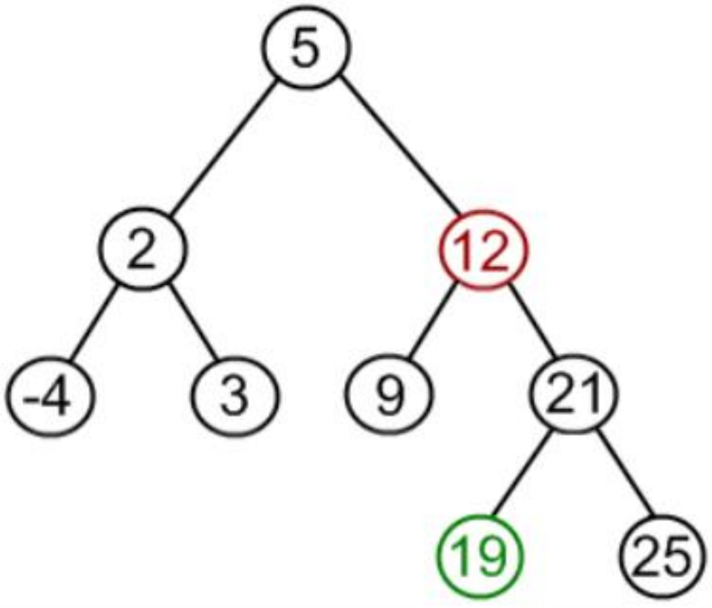


Binary Search Tree

Remove operation

c) <u>Node to be removed has two children</u>

1) Find minimum element in the right sub-tree of the node to be removed
   - In current example it is 19



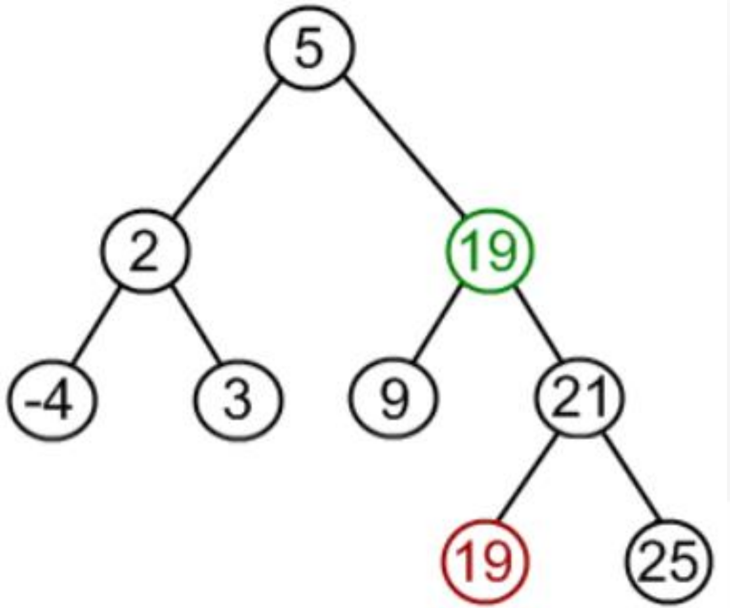
Binary Search Tree

Remove operation

c) Node to be removed has two children

2) Replace 12 with 19. Notice, that only values are replaced, not nodes

- Now we have two nodes with the same value

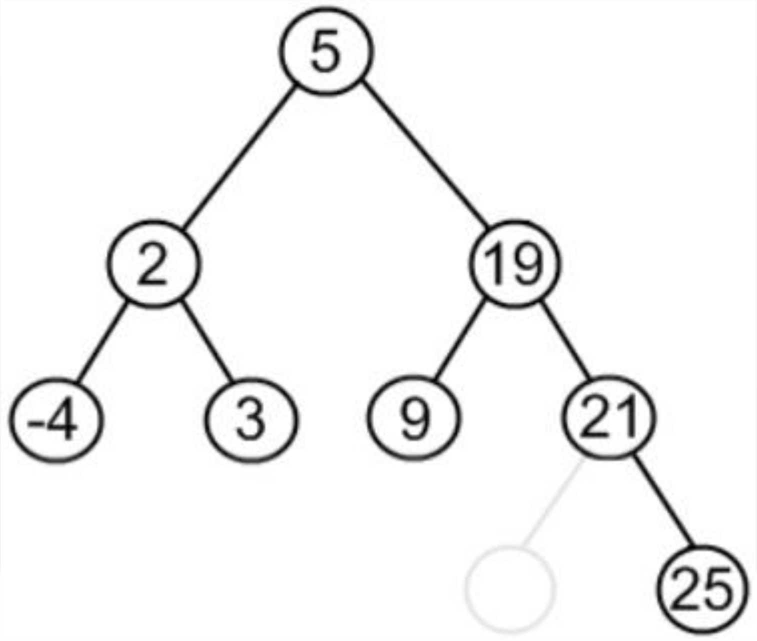


Binary Search Tree

Remove operation

c) Node to be removed has two children

   3) Remove 19 from the left sub-tree

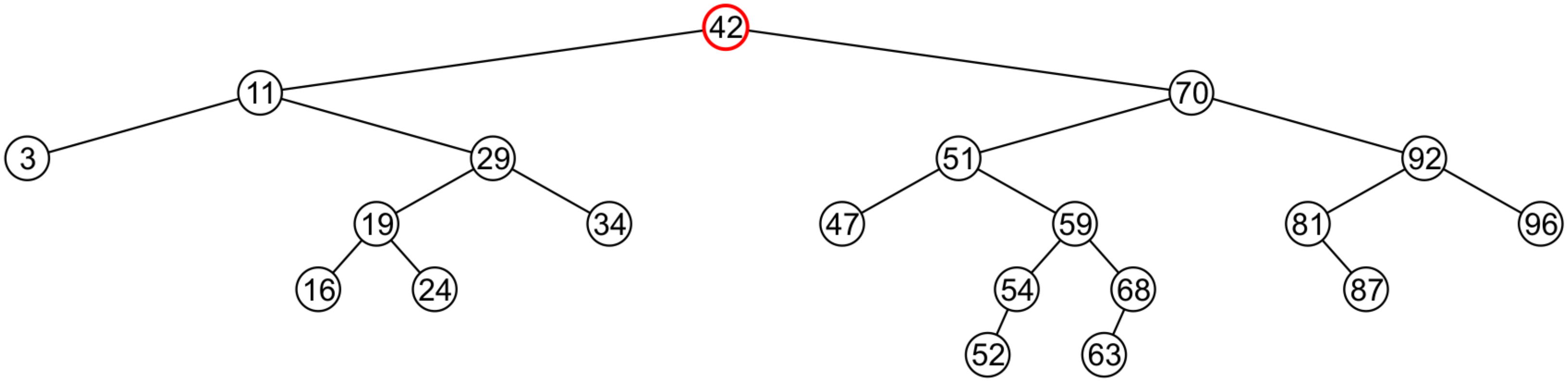


Binary Search Tree

Remove operation

c) <u>Node to be removed has two children</u>

Example. Remove 42 from a BST



Binary Search Tree

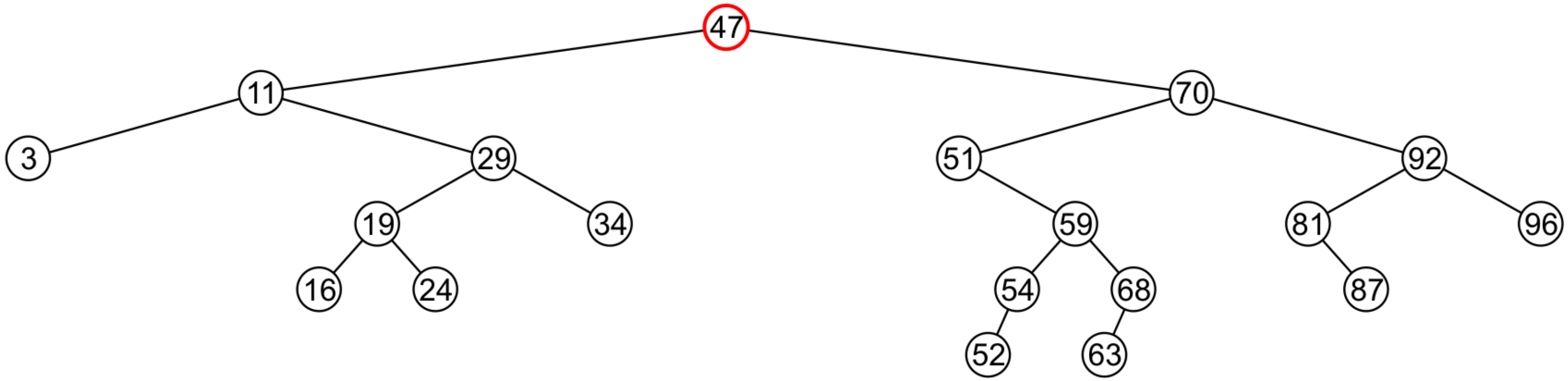
Remove operation

c) Node to be removed has two children

Another Example. Remove 47 from a BST

We will perform two operations:

1) Replace 47 with the minimum object in the right sub-tree
2) Erase that object from the right sub-tree



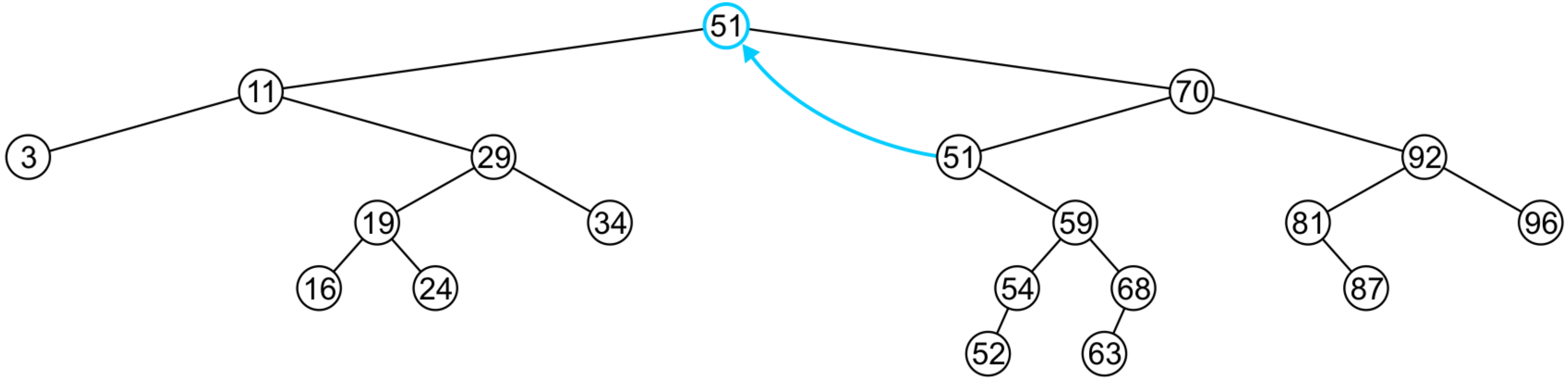
Binary Search Tree

Remove operation

c) <u>Node to be removed has two children</u>

Another Example. Remove 47 from a BST

We copy 51 from the right sub-tree



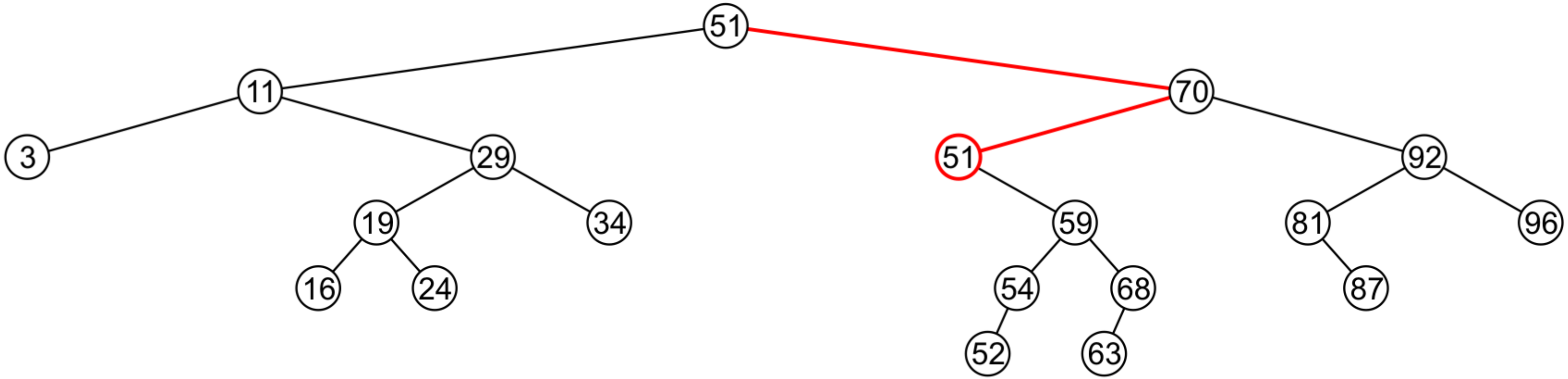
Binary Search Tree

Remove operation

c) Node to be removed has two children

Another Example. Remove 47 from a BST

We must proceed by delete 51 from the right sub-tree



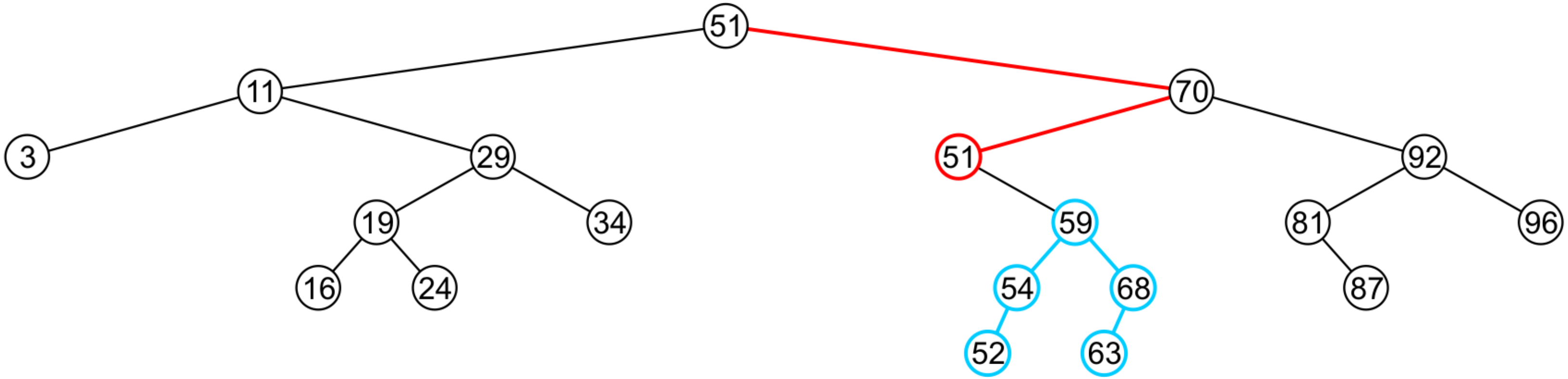
Binary Search Tree

Remove operation

c) Node to be removed has two children

Another Example. Remove 47 from a BST

In this case, the node storing 51 has just a single child



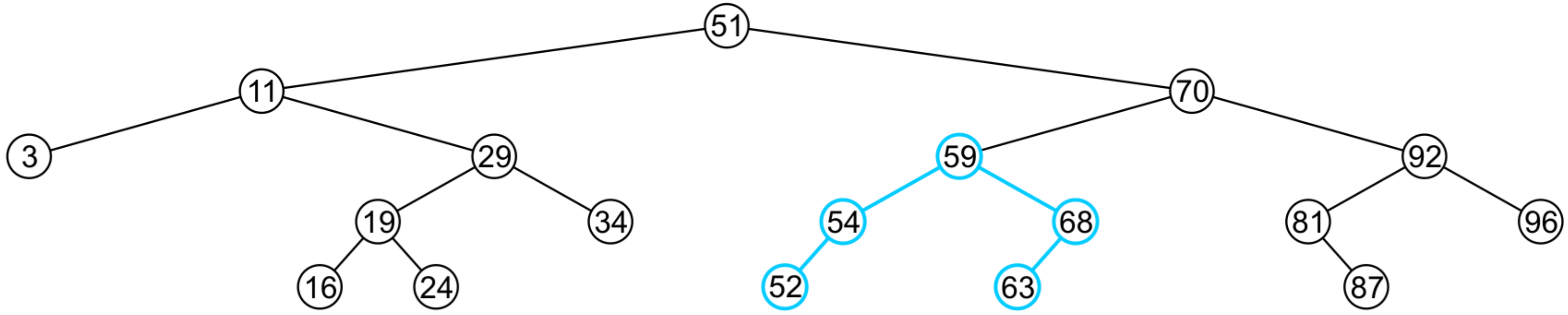
Binary Search Tree

Remove operation

c) <u>Node to be removed has two children</u>

Another Example. Remove 47 from a BST

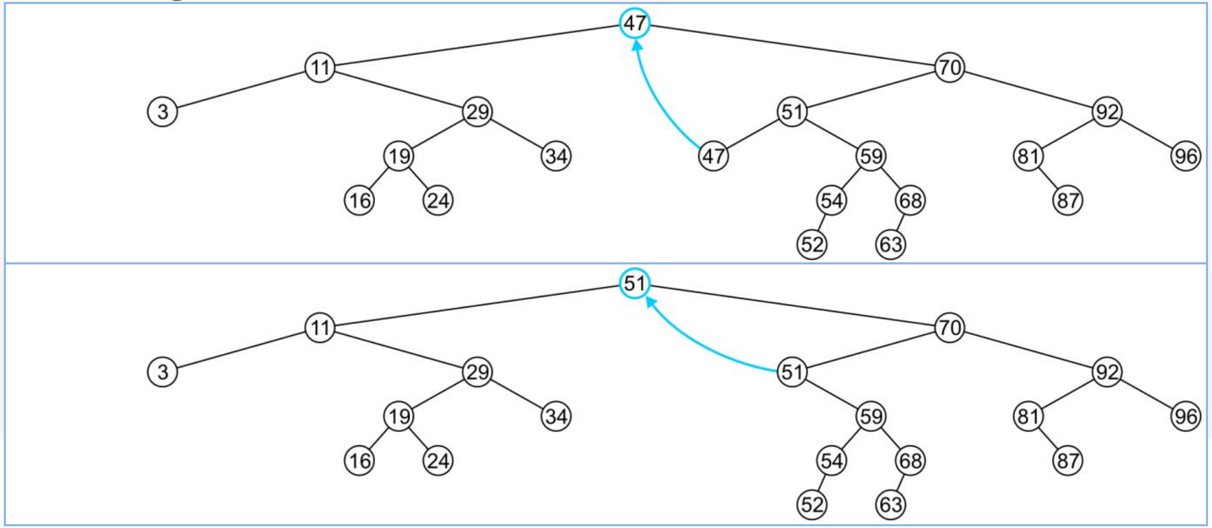
We delete the node containing 51 and assign the member variable left_tree of 70 to point to 59



Binary Search Tree

Remove operation

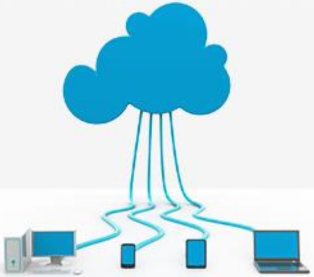
c) Node to be removed has two children
   In the two examples of removing a full node, we promoted:
   - A node with no children
   - A node with right child



Binary Search Tree

Remove operation

```
public class BST {
    // ...
    public boolean Remove( int value ) {
        if (Root == null)
            return false;
        else {
            if (Root.getElement() == value) {
                BSTNode auxRoot = new BSTNode(0);
                auxRoot.setLeft(Root);
                boolean result = Root.Remove(value, auxRoot);
                Root = auxRoot.getLeft();
                return result;
            }
            else  return Root.Remove(value, null);
        }
}
```



Binary Search Tree

Remove operation

```
public class BSTNode { // ...
    public boolean Remove(int value, BSTNode parent) {
        if (value < this.Element) {
            if (Left != null)     return Left.Remove(value, this);
            else                  return false;
        }
        else if (value > this.Element) {
            if (Right != null)    return Right.Remove(value, this);
            else                  return false;
        }
        else {
            if (Left != null && Right != null) {
                this.Element = Right.FindMin();
                Right.Remove(this.Element, this);
            }
            else if (parent.Left == this)
                parent.Left = (Left != null) ? Left : Right;
            else if (parent.Right == this)
                parent.Right = (Left != null) ? Left : Right;
            return true;
    } }
```