Data Structures and Algorithms

LECTURE 09: BINARY TREES, HEAPS, BINARY SEARCH TREES

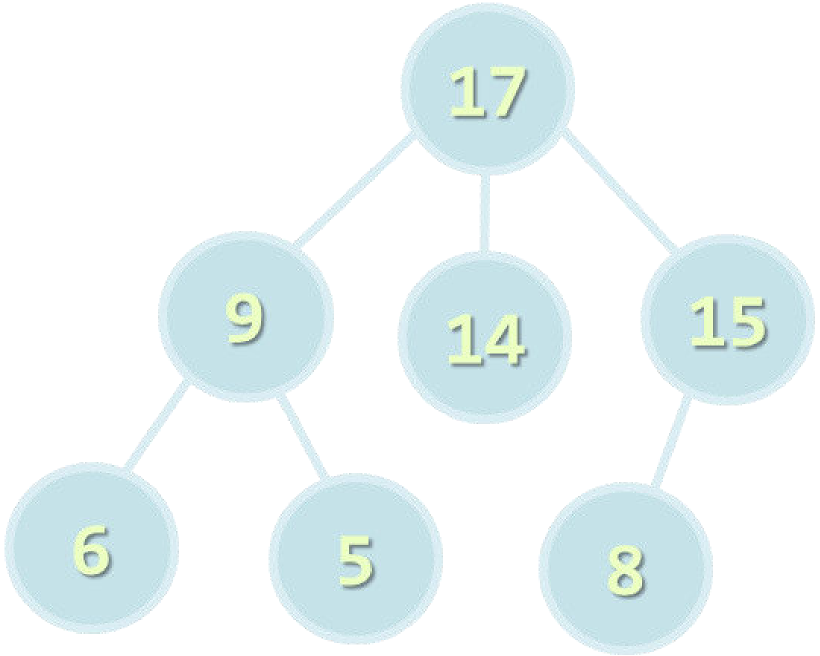
UNIVERSITY of GREENWICH
Alliance with FPT Education

Pearson
BTEC

Contents

- Binary Trees
  - Traversal algorithms
- Heaps
  - Binary heap, Min/Max heaps
- Binary Search Trees

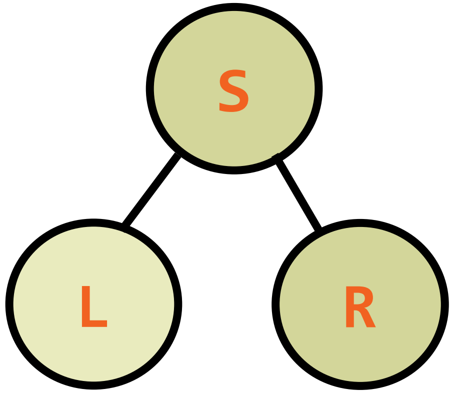
Binary Trees and BT Traversal



Preorder, In-Order, Post-Order

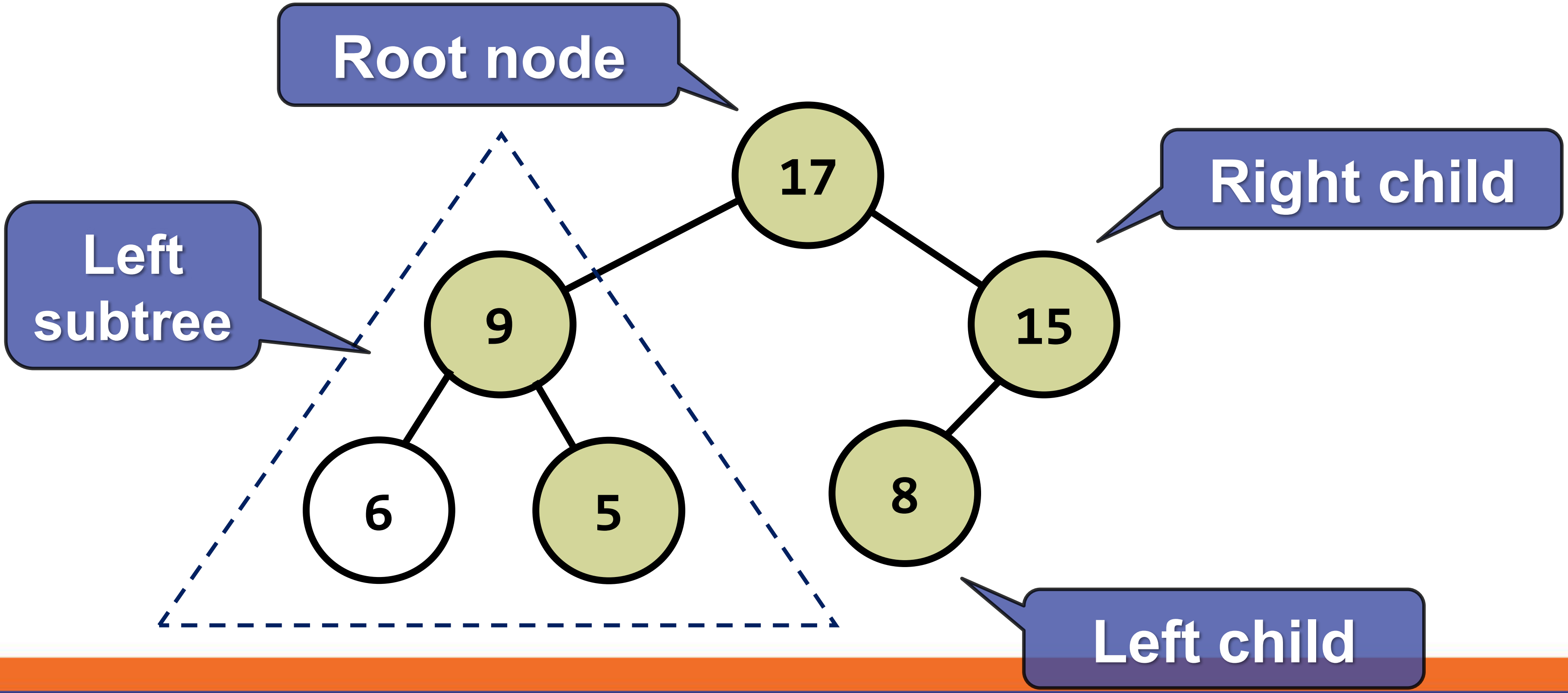
Binary Tree

- ADS representing tree like hierarchy
- Each node has **at most two** children
  - Children are called **left** and **right**
  - The **parent** is also called **source**



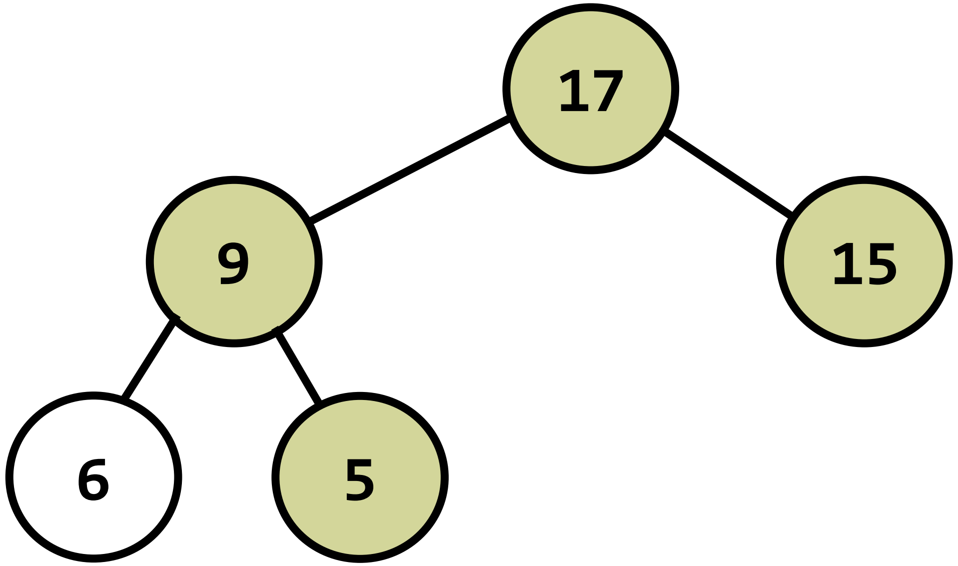
Binary Trees

- **Binary trees**: the most widespread form
  - Each node **has at most 2 children** (**left** and **right**)



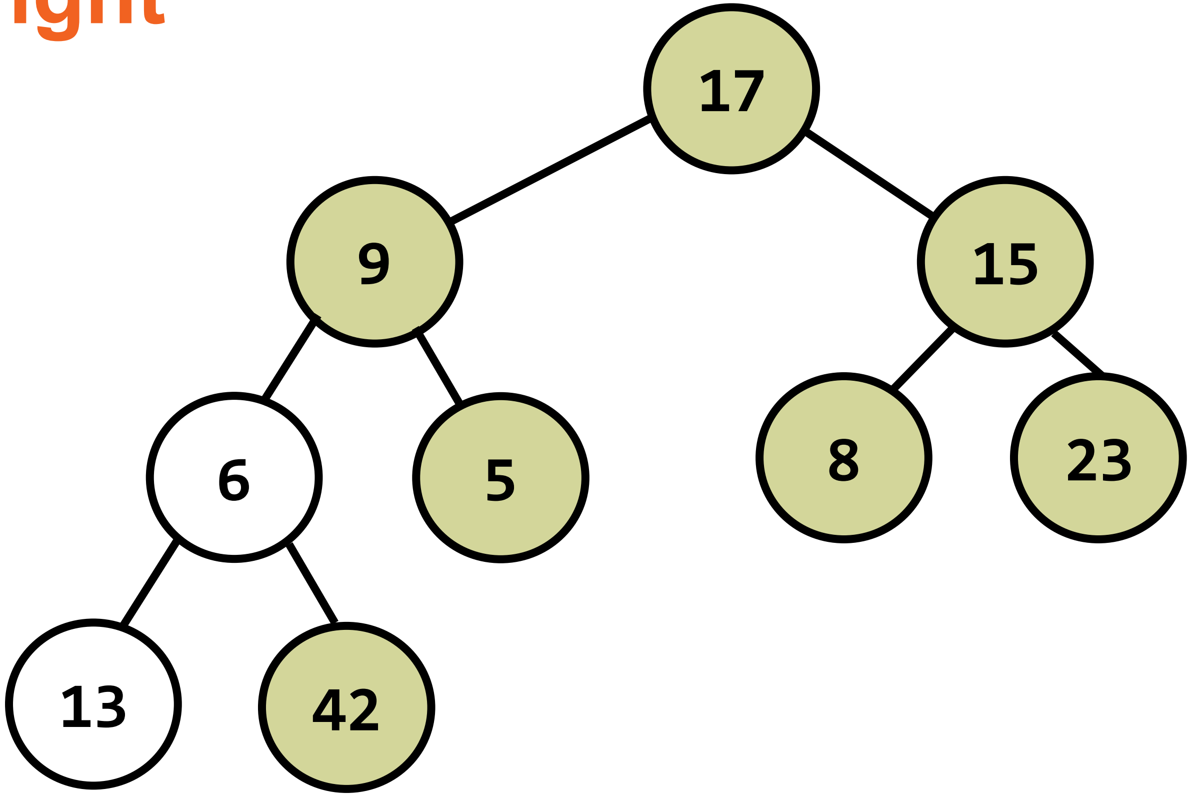
Types of Binary Trees

- **Full** – each node has **0** or **2** children



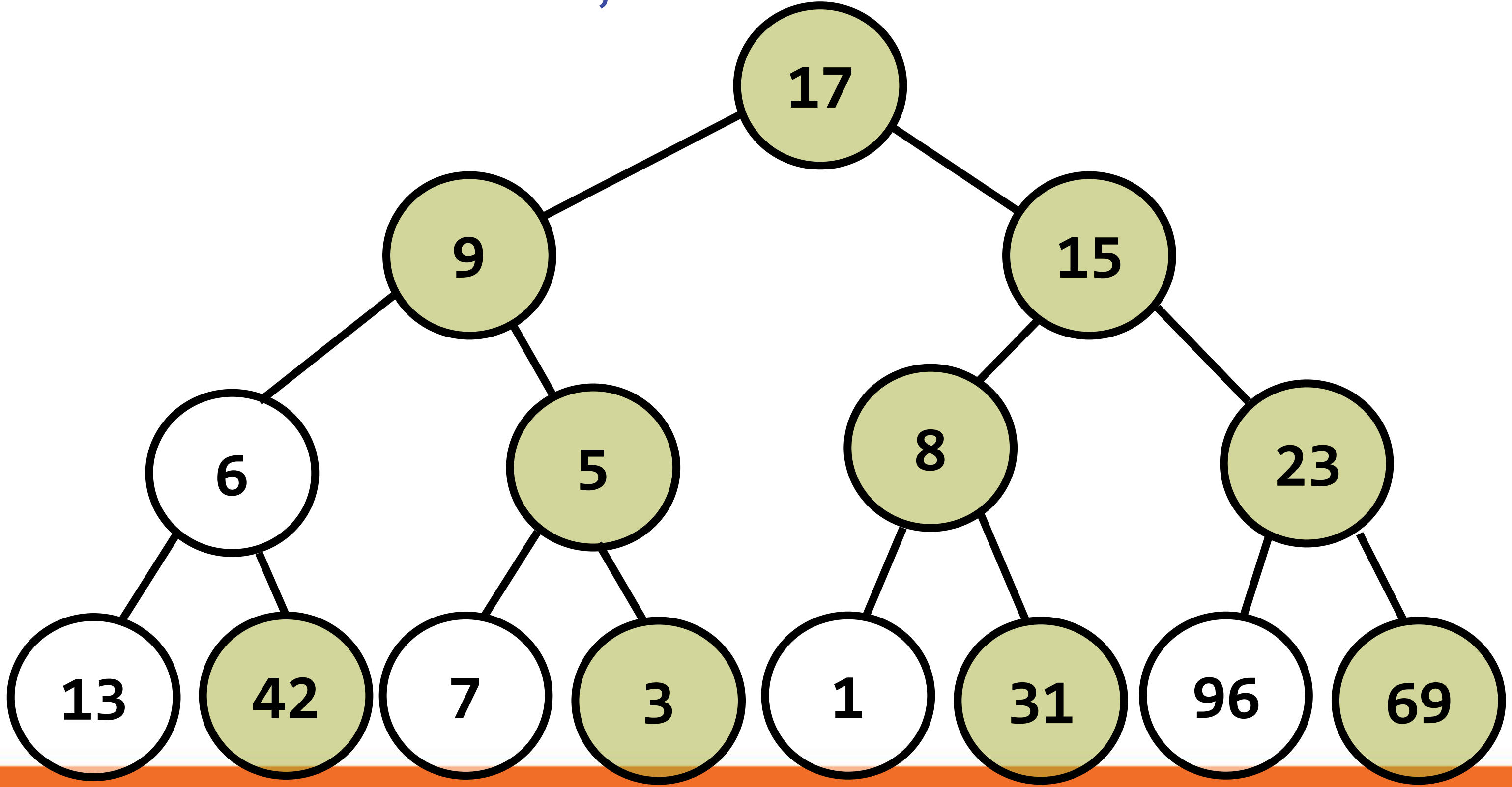
Types of Binary Trees

- **Complete** – nodes are filled **top** to **bottom** and **left** to **right**



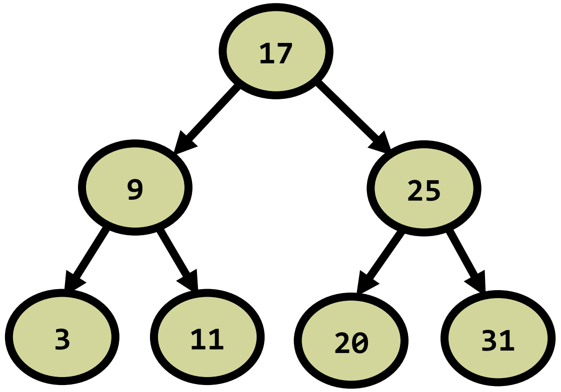
Types of Binary Trees

- **Perfect** – combines **complete** and **full**
  - leafs are at the **same level**, other nodes have **two** children



Problem: Binary Tree Traversals

- Inside the given skeleton
  - Implement **AbstractBinaryTree<E>**
  - Implement **asIndentedPreOrder**, each level indented +2




```
17
  9
    3
    11
  25
    20
    31
```

  - **preOrder**, **inOrder** and **postOrder**
    - Return the nodes as list List**<AbstractBinaryTree<E>>**

Solution: BT Traversals - Constructor

- Fields and constructor:

```
public class BinaryTree<E> implements AbstractBinaryTree<E> {
    private E key;
    private BinaryTree<E> left;
    private BinaryTree<E> right;

public BinaryTree(E key, BinaryTree<E> left, BinaryTree<E> right) {
    this.key = key;
    this.left = left;
    this.right = right;
}
```

Solution: BT Traversals - Print

```
public String asIndentedPreOrder(int indent) {
    String out = createPadding(indent) + getKey();
    if (getLeft() != null) {
        out +="\n" + getLeft().asIndentedPreOrder(indent + 2);
    }
    if (getRight() != null) {
        out +="\n" + getRight().asIndentedPreOrder(indent + 2);
    }
    return out;
}
```

Process Node

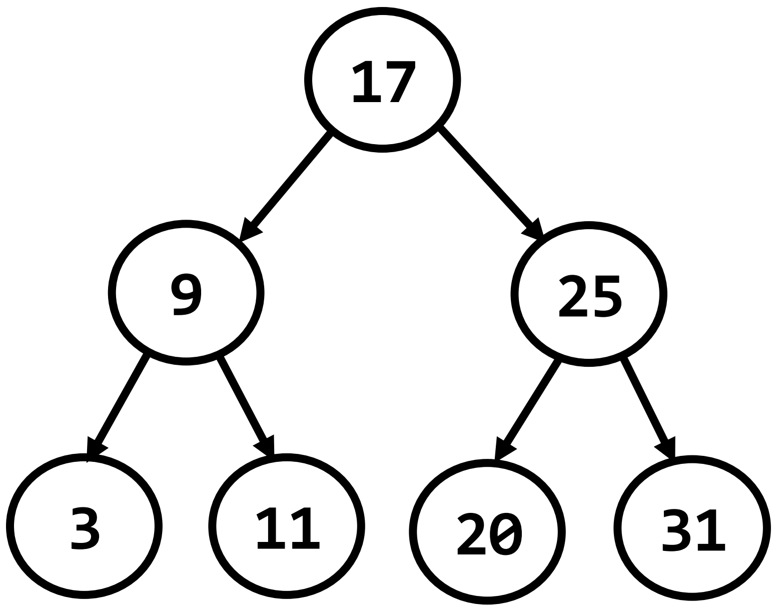
Traverse Left

Traverse Right

Binary Trees Traversal: Pre-order

- Root → Left → Right

```
preOrder (node) {
  if (node != null) {
    print node.key
    preOrder(node.left)
    preOrder(node.right)
  }
}
```

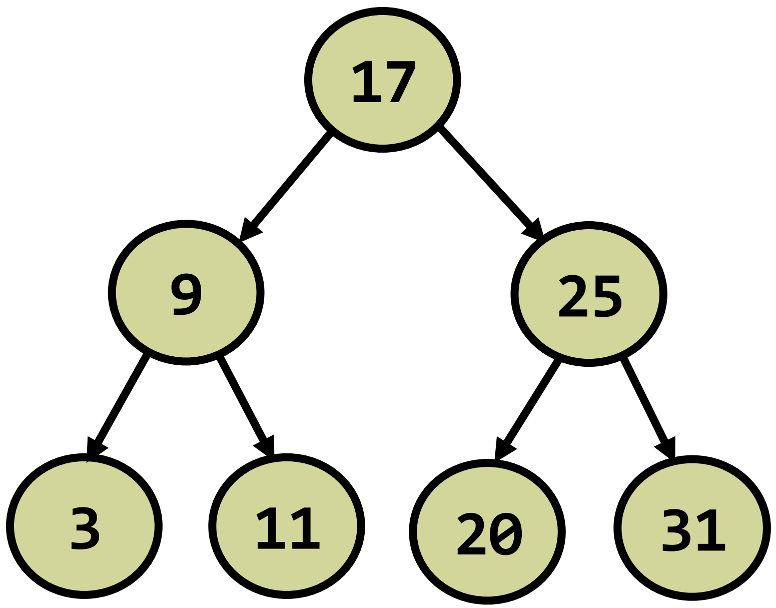


→ 17 9 3 11 25 20 31

Binary Trees Traversal: In-order

- Left → Root → Right

```
inOrder (node) {
  if (node != null) {
    inOrder(node.left)
    print node.key
    inOrder(node.right)
  }
}
```

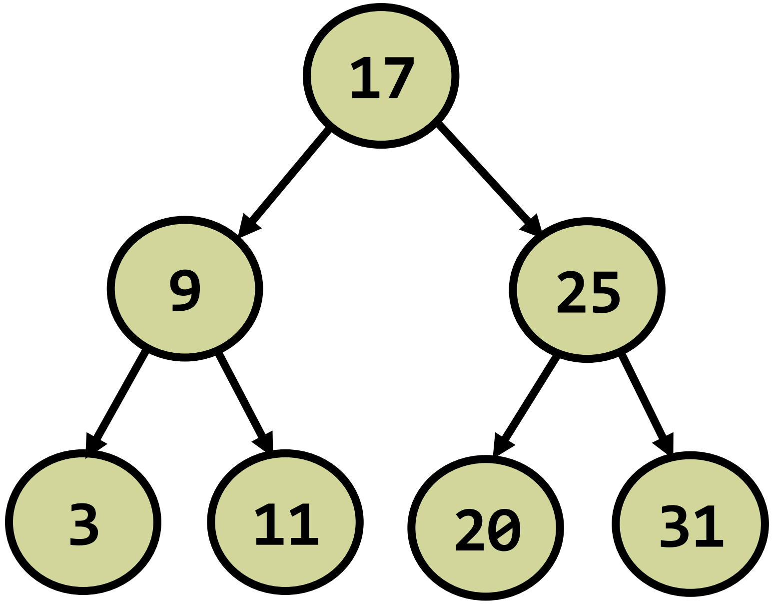


→ 3 9 11 17 20 25 31

Binary Trees Traversal: Post-order

- Left → Right → Root

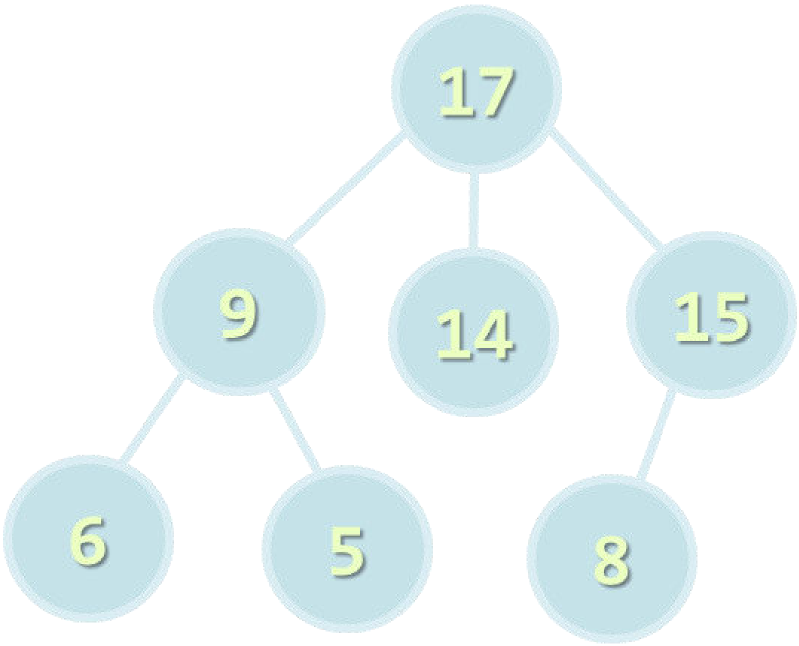
```
postOrder (node) {
  if (node != null) }
    postOrder(node.left)
    postOrder(node.right)
    print node.key
  }
}
```



→ 3 11 9 20 31 25 17

Solution: BT Traversals - forEachInOrder

```
public void forEachInOrder(Consumer<E> consumer) {
    if (this.getLeft() != null) {
        this.getLeft().forEachInOrder(consumer);
    }
    consumer.accept(this.getKey());
    if (this.getRight() != null) {
        this.getRight().forEachInOrder(consumer);
    }
}
```



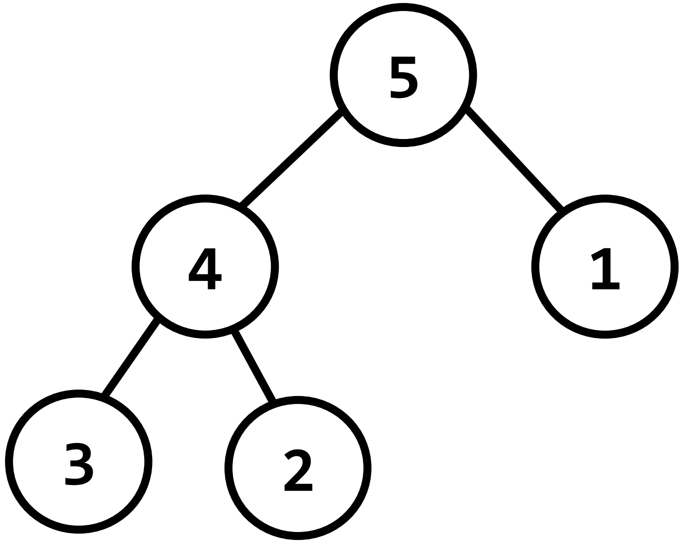
Heap, Binary Heap

What is Heap?

- **Heap**
  - Tree-based data structure
  - Stored in an array
- Heaps hold the **heap property** for each node:
  - **Min Heap**
    - parent ≤ children
  - **Max Heap**
    - parent ≥ children

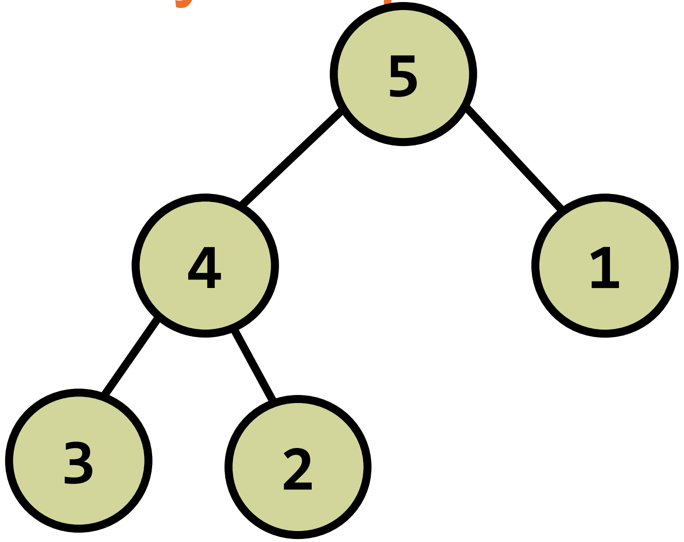
Binary Heap

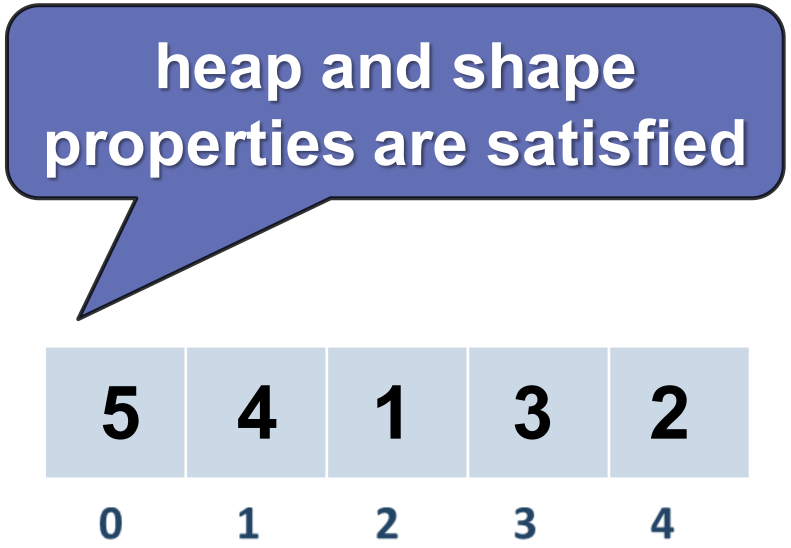
- **Binary heap**
  - Represents a Binary Tree
- **Shape property** - Binary heap is a **complete binary tree**:
  - Every level, except the last, is **completely filled**
  - Last is filled **from left to right**



Binary Heap – Array Implementation

- Binary heap can be efficiently stored in an array





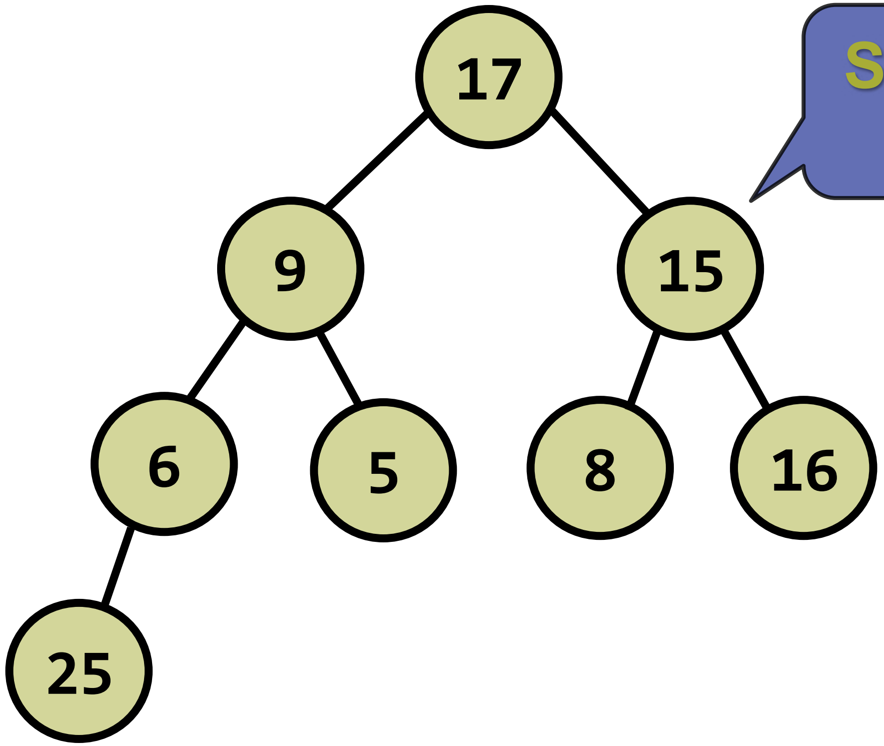
- **Parent(i)** = (i - 1) / 2
- **Left(i)** = 2 * i + 1; **Right(i)** = 2 * i + 2

Heap Insertion

- To preserve **heap properties**:
  - **Insert** at the end
  - **Heapify** element up

Promote while element > parent

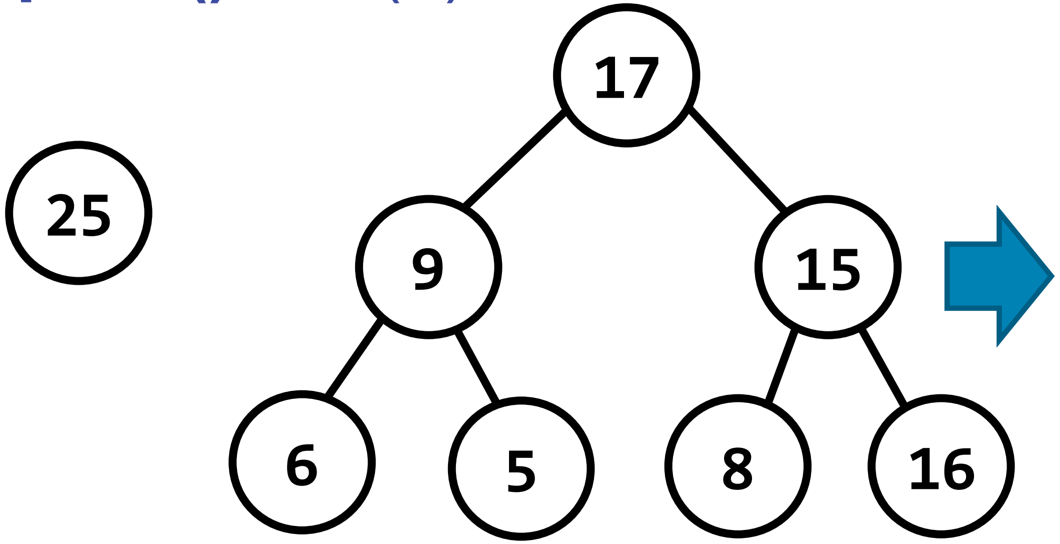
- Right: Max Heap
  - Insert 16
  - Insert 25

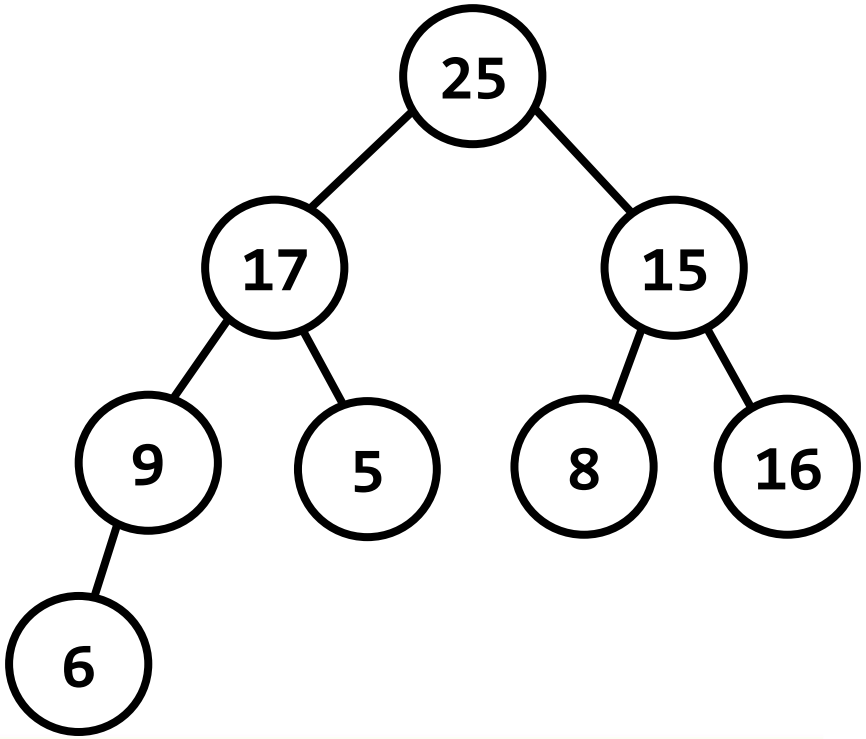


Satisfy heap property

Problem: Heap Add and Peek

- Implement a max **MaxHeap<E>** with:
  - **int size()**
  - **void add(E element)** – O(logN)
  - **E peek()** – O(1)





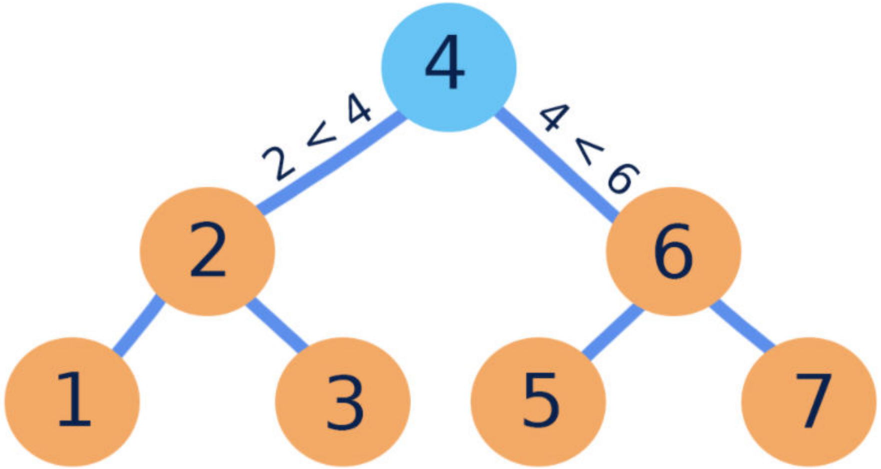
Solution: Heap Add and Peek (1)

```
public class MaxHeap<E extends Comparable<E>> implements
Heap<E> {
    // TODO: store the elements
    @Override
    public void add(E element) {
        this.elements.add(element);
        this.heapifyUp(this.size() - 1);
    }
}
```

Solution: Heap Add and Peek (2)

```
private void heapifyUp(int index) {
    while (index > 0 && less(parent(index), get(index))) {
        int parentAt = getParentAt(index);
        Collections.swap(this.elements, parentAt, index);
        index = parentAt;
    }
}
// TODO: Implement less(), parent() and getParentAt()
```

Binary Search Trees

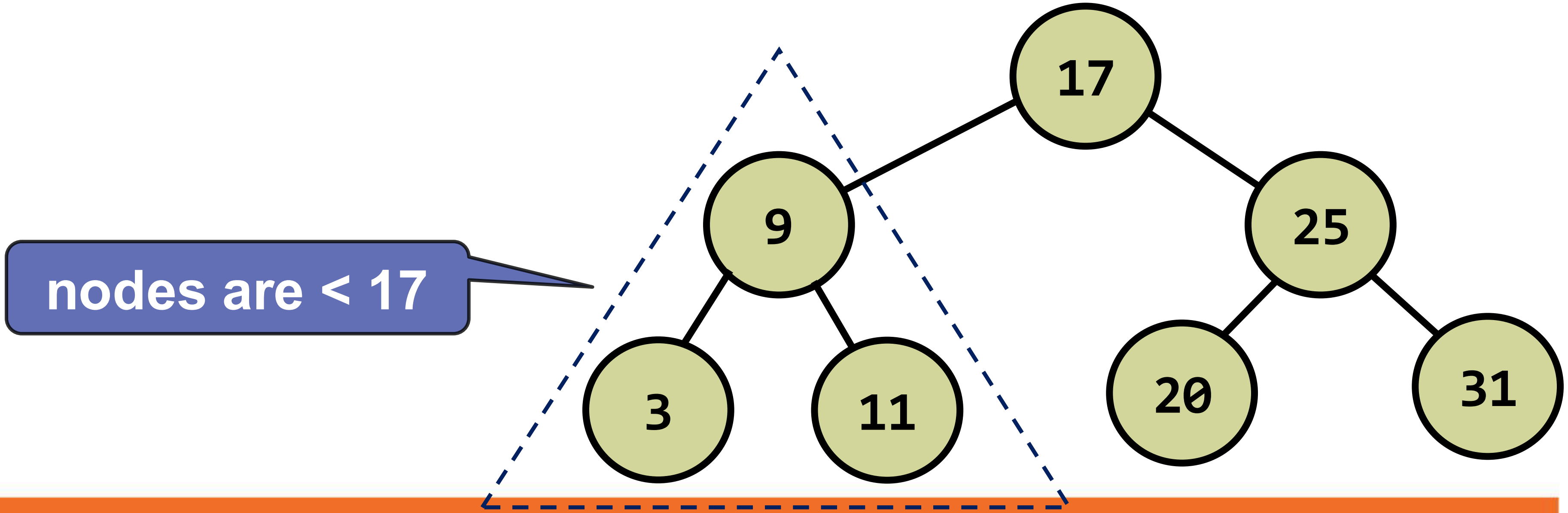


Two Children at Most

Binary Search Trees

- **Binary search trees** are **ordered**
  - For each node ***x***
    - Elements in left subtree of ***x*** are **<** ***x***
    - Elements in right subtree of ***x*** are **>** ***x***

what about ==

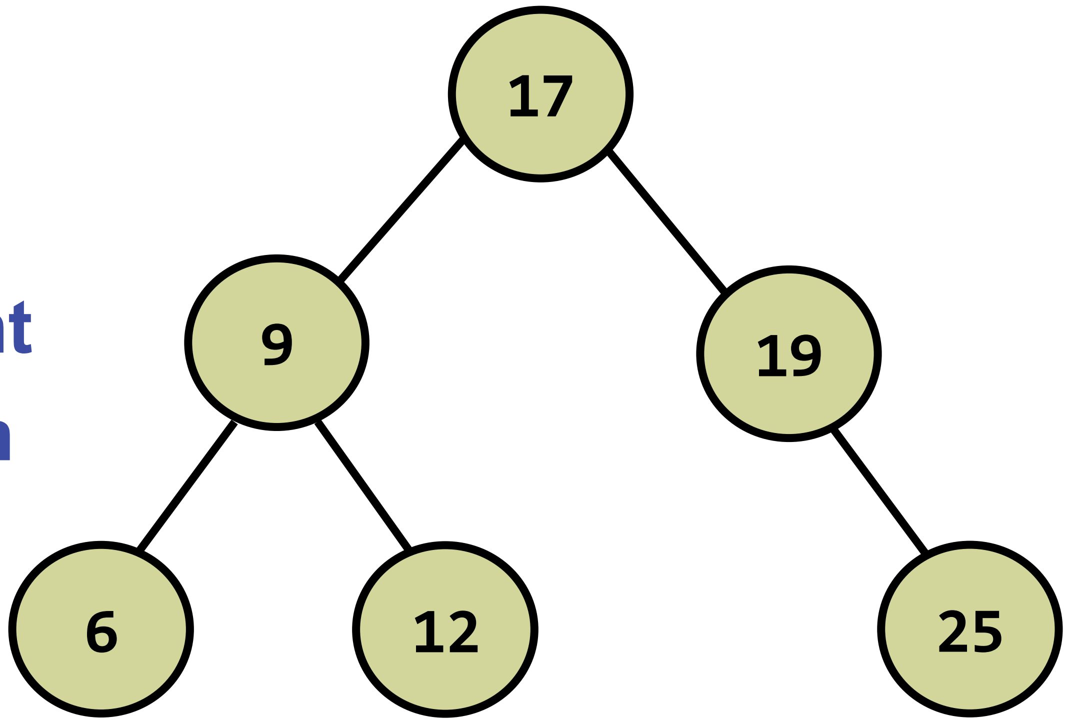


UNIVERSITY of GREENWICH
Alliance with FPT Education

Pearson BTEC

BST - Search

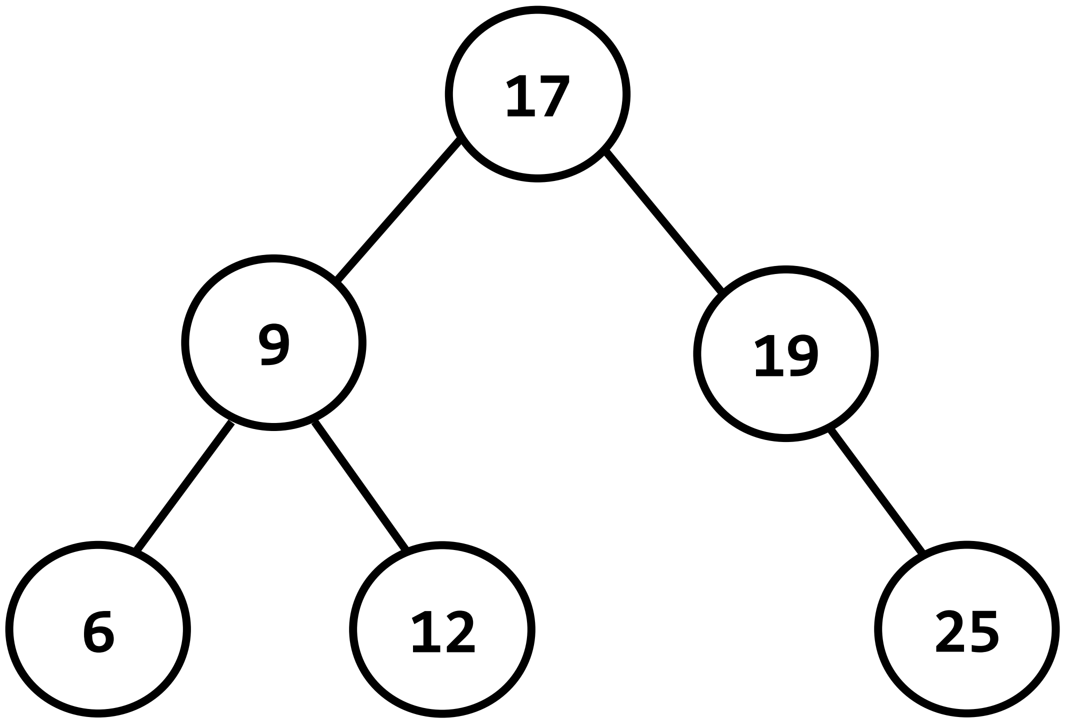
- Search for **x** in BST
  - if node is not null
    - if x **<** node.value → **go left**
    - else if x **>** node.value → **go right**
    - else if x **==** node.value → **return**




```
Search 12 → 17 9 12
Search 27 → 17 19 25 null
```

BST - Insert

- Insert **x** in BST
  - – if node is **null** → insert x
  - – else if x **<** node.value → **go left**
  - – else if x **>** node.value → **go right**
  - – else → node **exists**

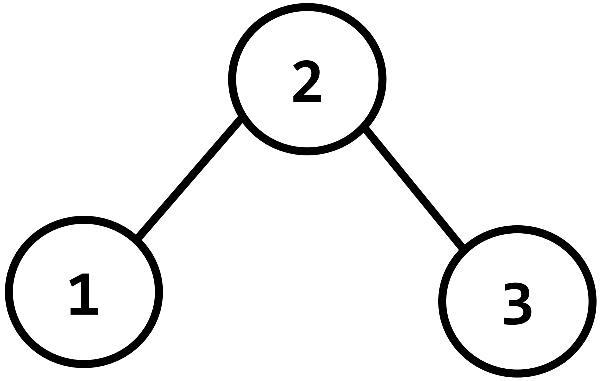


Insert 12 → 17 9 12 return

Insert 27 → 17 19 25 null(insert)

Problem: BST

- You are given a skeleton
  - Implement **AbstractBinarySearchTree<E>**
    - **bool contains(E element)**
    - **void insert(E element)**



Solution: BST Contains

```
public boolean contains(E element) {
    Node<E> current = this.root;
    while (current != null){
        if (element.compareTo(current.value) < 0){
            current = current.leftChild;
        } else if (element.compareTo(current.value) > 0){
            current = current.rightChild;
        } else {
            break;
        }
    }
    return current != null;
}
```

Solution: BST Insert

```
public void insert(E element) {
    if (this.root == null) {
        this.root = new Node<>(element);
    } else {
        // TODO: Find the place to insert
        if (parent.value.compareTo(element) > 0){
            parent.leftChild = new Node<>(element);
        } else {
            parent.rightChild = new Node<>(element);
        }
    }
}
```

Problem: BST Search

- Implement:
    - **BST<E> search(E value)**
- Make sure the method works for:
    - **empty tree**
    - tree with **one element**
    - tree with **two elements - root + left/right**
    - tree with **multiple elements**

Solution: BST Search

```java
public AbstractBinarySearchTree<E> search(E element) {
    Node<E> current = this.root;
 // TODO: Find the node with the element
    return new BinarySearchTree<>(current);
}
```

Solution: BST Search (2)

```
private BinarySearchTree(Node<E> root) {
  this.copy(root);
}

private void copy(Node<E> node) {
  if (node == null) return;

  this.insert(node.value);
  this.copy(node.leftChildre);
  this.copy(node.rightChildren);
}
```

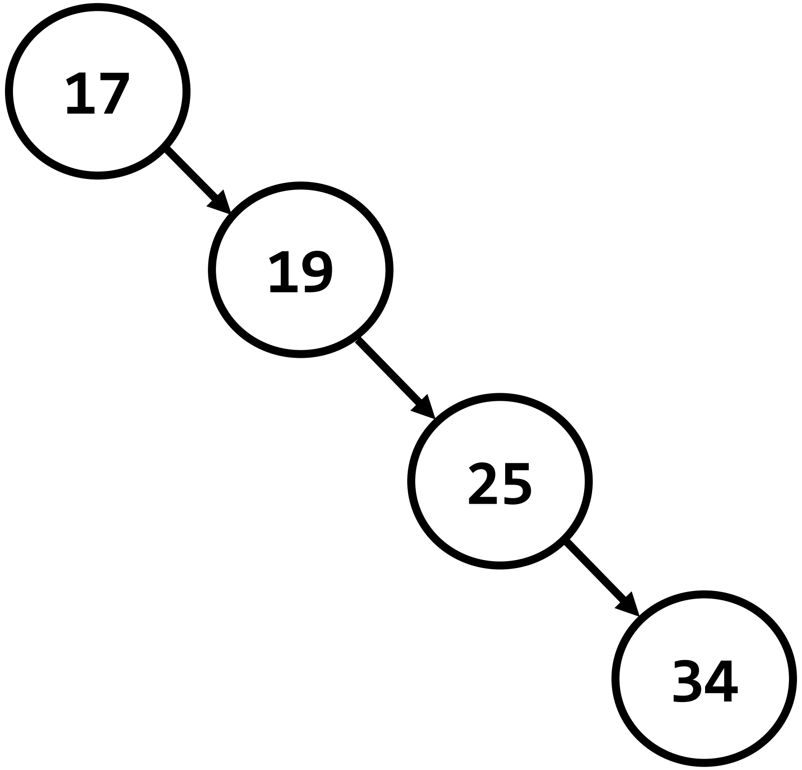
Pre-Order Traversal

BST - Search Operation Speed - Quiz

- What is the speed of the **search(E)** operation on BST?
  - O(n)
  - O(log(n))
  - O(1)

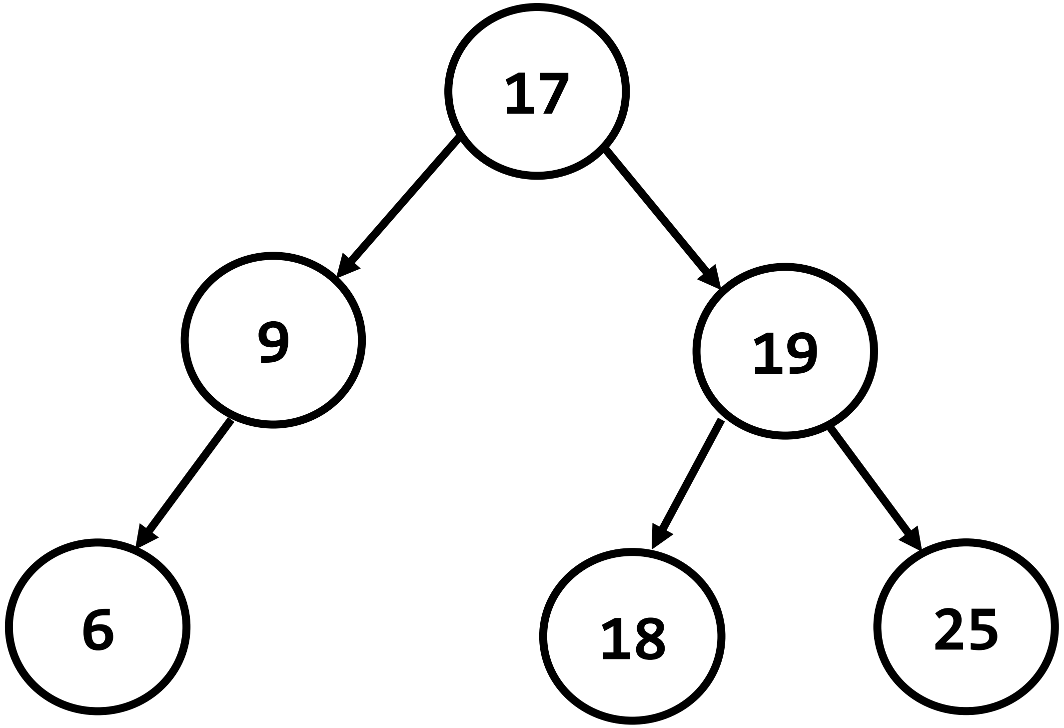
BST - Search Operation Speed - Answer

- What is the speed of the **search(E)** operation on BST?
  - O(n)
  - O(log(n))
  - O(1)



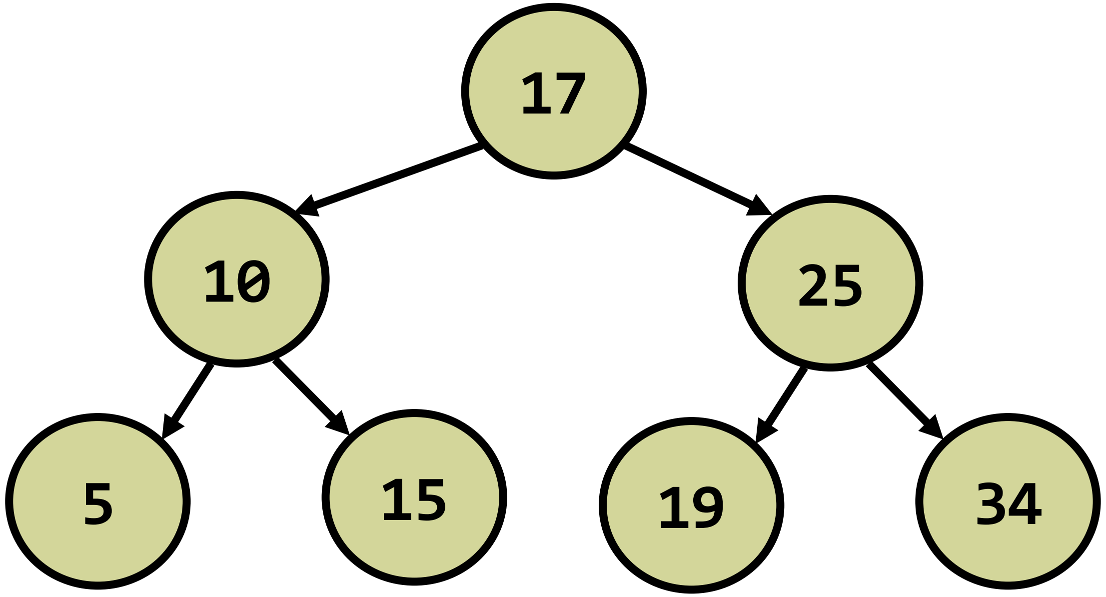
Binary Search Trees – Operation Speed

- Insert – **height** of tree
- Search – **height** of tree



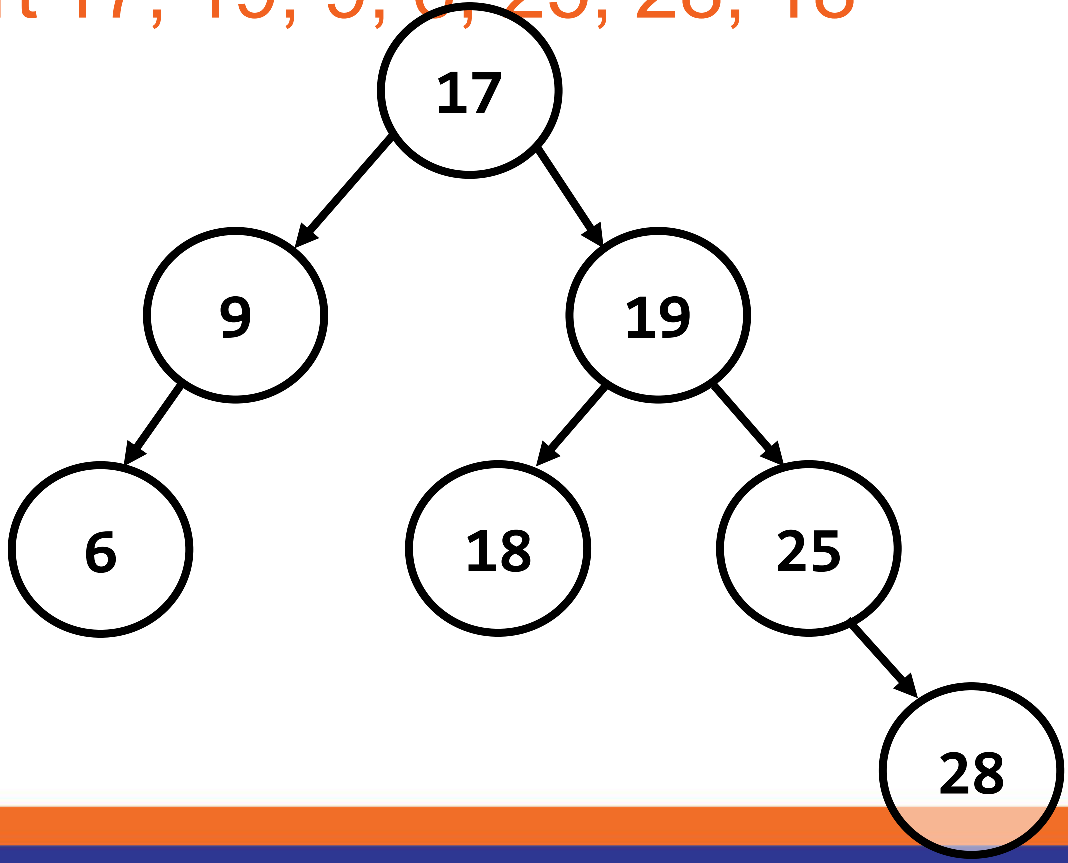
Binary Search Trees – Best Case

- Example: Insert 17, 10, 25, 5, 15, 19, 34



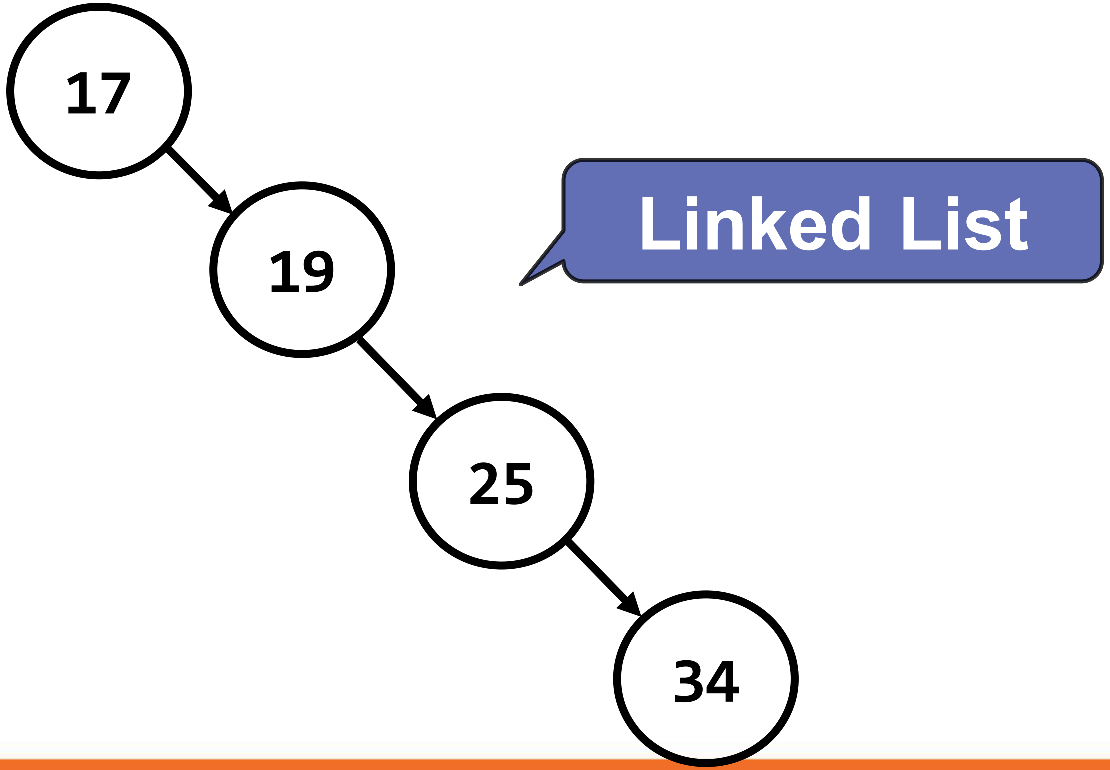
Binary Search Trees – Average Case

- You can insert values in ever **random** order
- Example: Insert 17, 19, 9, 6, 25, 28, 18



Binary Search Trees – Worst Case

- You can insert values in ever **increasing/decreasing** order
- Example: Insert 17, 19, 25, 34



Balanced Binary Search Trees

- Binary search trees can be **balanced**
  - Balanced trees have for each node
    - Nearly equal number of nodes in its subtrees
  - **Balanced trees** have **height of ~ log(n)**

Summary

- **Binary** trees have **0** or **2** children
- **Heaps** are used to **implement priority** queues
- Binary Heaps have tree-like structure
- **Efficient** operations
  - **Add**
  - **Find** min
  - **Remove** min