SE2205: Algorithms and Data Structures for Object-Oriented Design

Trees

Instructor: Pirathayini Srikantha

Western University

February 22, 2019

Readings/References

- Goodrich (8, 9, 11.1-11.3)

Table of Contents

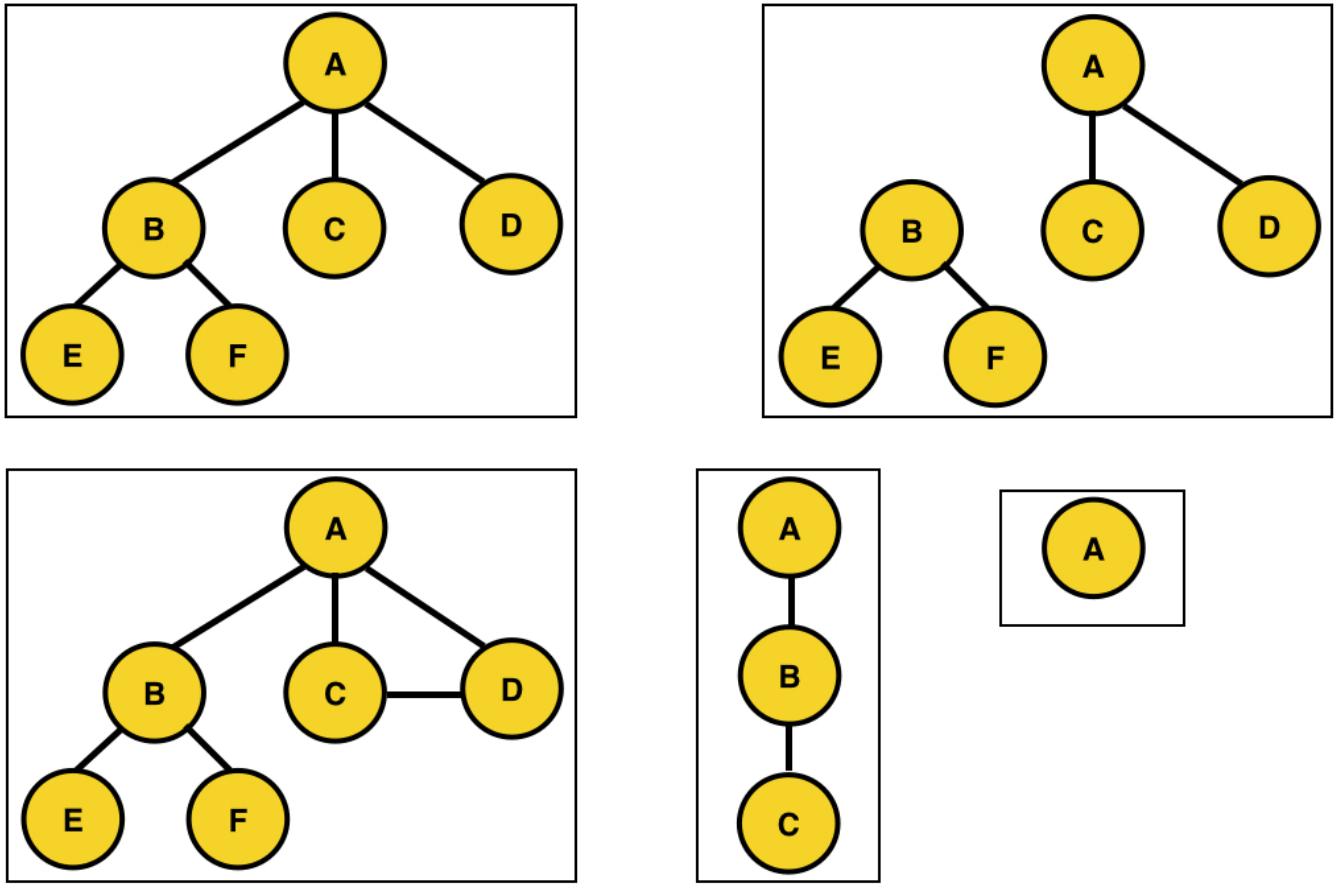
1. Introduction
2. Binary Trees
   - Heaps
   - Binary Search Trees
   - AVL Trees
   - Red-Black Tree
3. 2-3 Trees
4. B-Trees
5. Tries

Table of Contents

1. Introduction
2. Binary Trees
   - Heaps
   - Binary Search Trees
   - AVL Trees
   - Red-Black Tree
3. 2-3 Trees
4. B-Trees
5. Tries

Graphs

Express **relationships** amongst entities (e.g. connections in social media, coupling amongst systems, constraints amongst components, etc.)

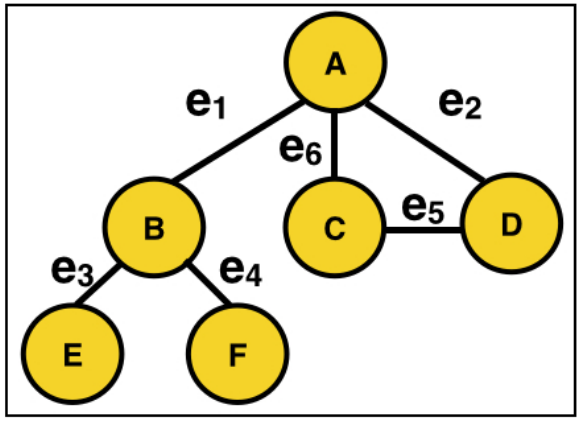


Some Formal Definitions

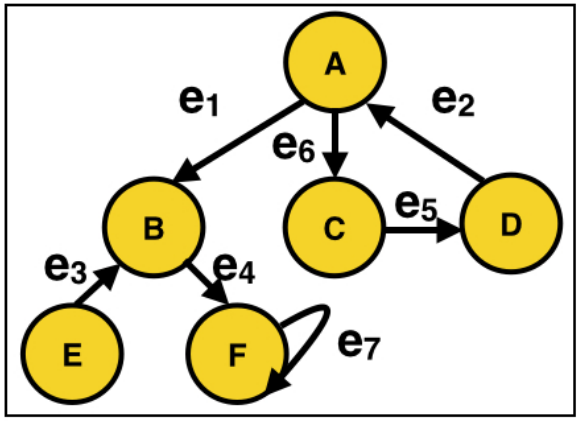
- **Graph** is represented by $G = (V, E)$
- V consists of all the **nodes** or vertices in the graph
- E consists of all the **edges** in the graph
- Each edge $e_k \in E$ is a pair of vertices $e_k = (v_i, v_j)$ where $v_i, v_j \in V$ are connected by a link
- Edges can be ordered (**directed**) or unordered (**undirected**)
- A **path** from v_i, v_j of length n consists of a sequence of edges $P = e_1, \ldots, e_n$ that connect these vertices (for directed graphs, these edges must be in the same direction)
- **Cycle** is a path that starts and ends at the same vertex while visiting remaining vertices in the path only once
- **Connected** graph: No vertices are disjoint
- **Unconnected** graph: Vertices are disjoint

Illustration of Formal Definitions

Undirected



Directed



$V=\{A, B, C, D, E, F\}$

$E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$

$e_1=(A,B)$, $e_1=(B,A)$

$P_{AE}=\{e_1,e_3\}$

$P_{AF}=\{e_1,e_4\}$

Cycle: $P_{AA}=\{e_6,e_5,e_2\}$

$E=\{e_1,e_2,e_3,e_4,e_5,e_7\}$

$e_1=(A,B)$, $e_1\neq(B,A)$

$P_{AE}=NA$

$P_{AF}=\{e_1,e_4\}$

Cycle: $P_{AA}=\{e_6,e_5,e_2\}$

Cycle: $P_{FF}=\{e_7\}$

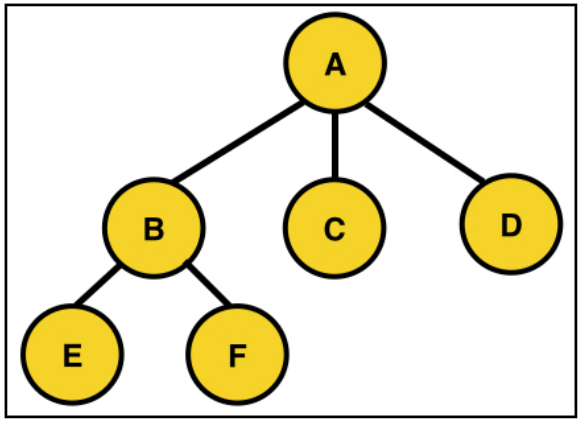
Connected

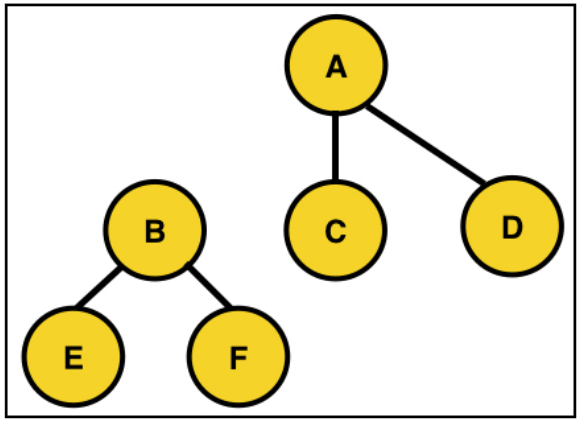
What are Trees?

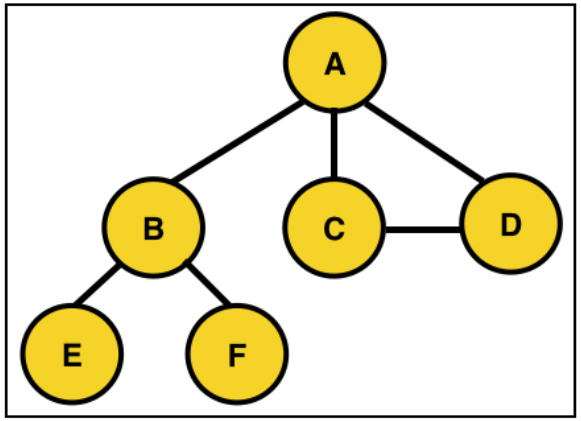
- A tree is composed of nodes/vertices and links/edges
- All nodes in a tree must be **connected**
- These nodes must **not** form cycles
- A tree is a **special** type of a graph
- A tree **must** consist of a single path from the root to the each descendant in the tree
- We will consider **undirected** trees

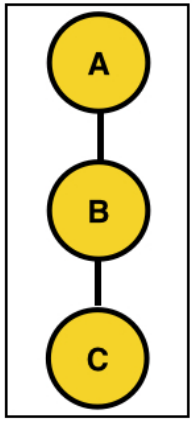
Examples of Trees

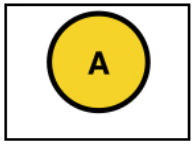
Which one of the following are trees?





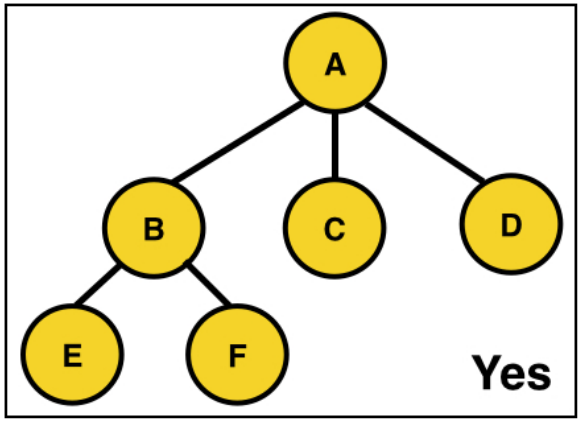


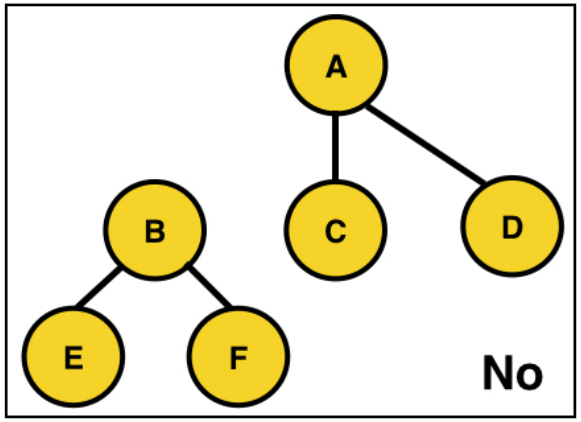


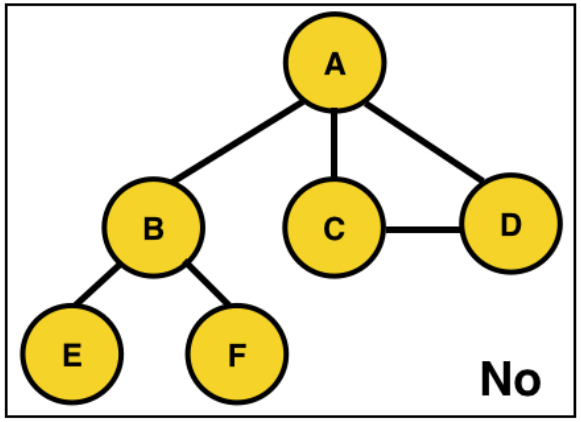


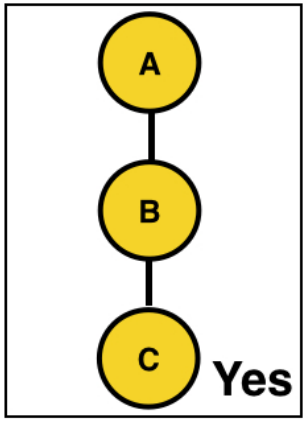
Examples of Trees

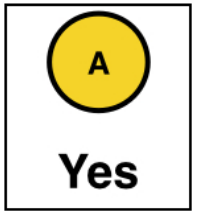
Which one of the following are trees?











Why are Trees Useful?

- Provides **natural representation** of many kinds of data (e.g. hierarchical, etc.)
- Applied to implement a large number of algorithms **efficiently**
- There exists many types of trees:
  - Static (shape is fixed)
  - Dynamic (shape can change with insertion/deletion of nodes or shape modifying changes)
  - Stores sub-solutions of problems (game tree)
  - Search trees (efficient organization of data for searching)
  - Efficient implementation of ADT/algorithmic operations (heaps/priority queues)

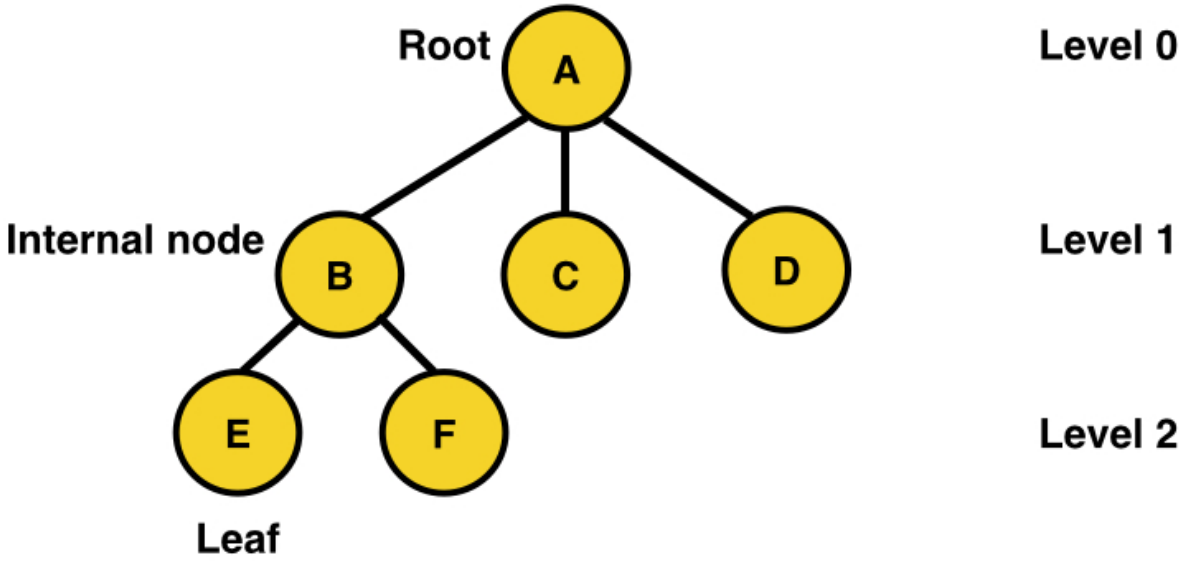
Tree Nomenclature

- Various components in trees can be referred to in three different ways (family, geometric, biological)
- **Family** relationship (parents/children, ancestors/descendants)
- **Geometric** relationship (left/right node)
- **Biological** (roots/leaves)

Tree Nomenclature

- **Levels**: If the number of edges required to reach nodes are the same, these nodes are at the same level
- **Root**: Top-most node
- **Leaf**: Nodes without any children
- **Internal node**: Nodes with children and is not a root
- **Child** of a node: Connected by a single edge and resides at a level below the node
- **Parent** of a node: Connected by a single edge and resides at a level above the node
- **Siblings**: All nodes having the same parent
- **Ancestors** of a node: All nodes encountered when traversing up from the node (towards the root)
- **Descendants** of a node: All nodes encountered when traversing down the node (away from the root)

Example: Tree Nomenclature



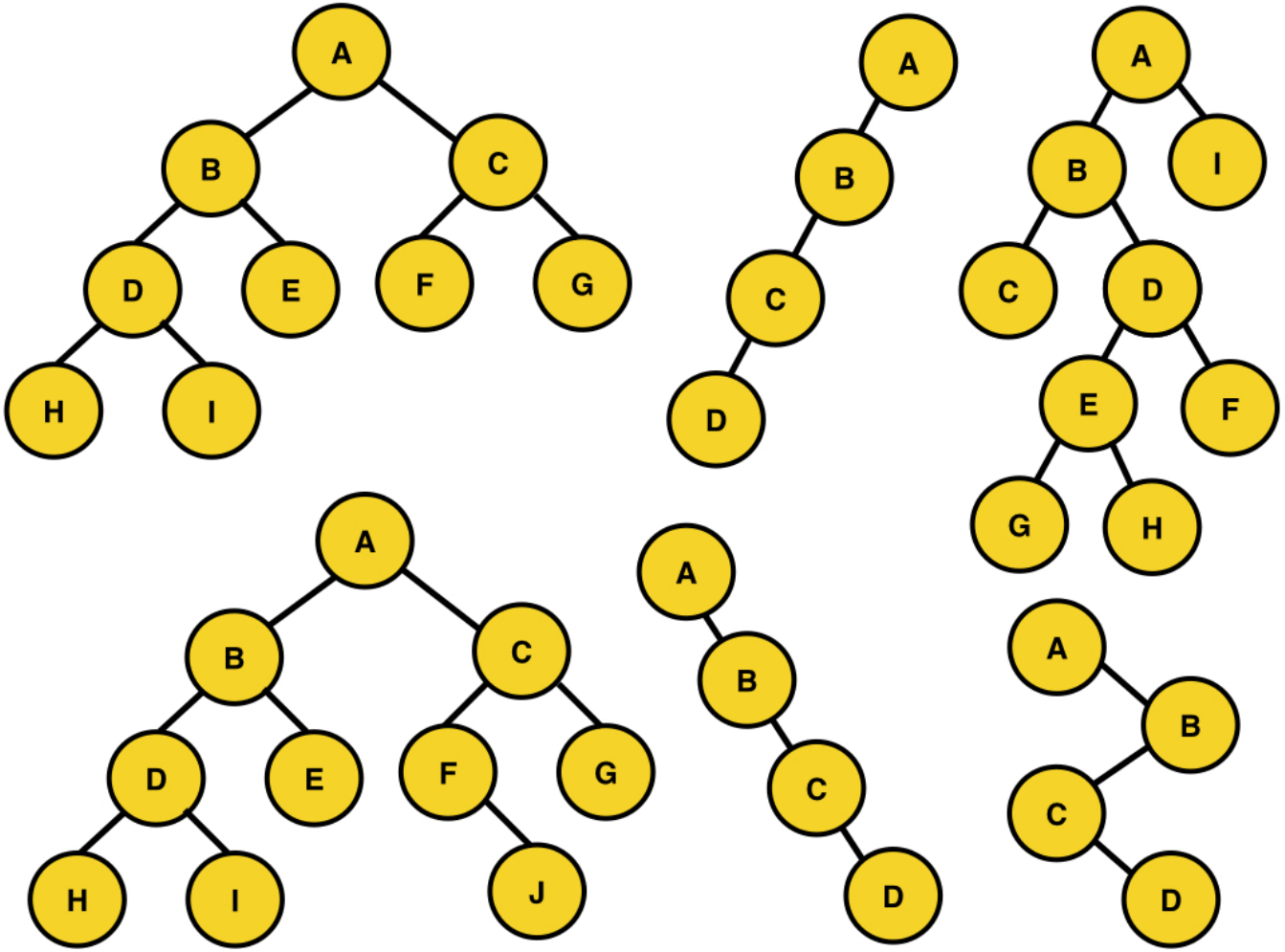
B: child of A and parent of E
Descendants of B: E, F
Ancestors of E: B, A
B is left child of A
C is middle child of A
D is right child of A

Tree Metrics

- **Depth**: Number of edges from root to node
- **Height**: Edges between node to its deepest leaf descendant
- **Degree**: Number of edges incident on a node
- **Maximum Children per Node *M***: Trees can be classified based on this (i.e. a binary tree has a maximum of 2 children)
- **Full**: Each node has exactly M children or zero children
- **Complete**: completely filled tree except for the bottom level (fill from left to right) (i.e. leafs reside on the bottom two adjacent levels)
- Other types of trees: **Left linear tree**, **Zig-zag tree**, **Right linear tree**

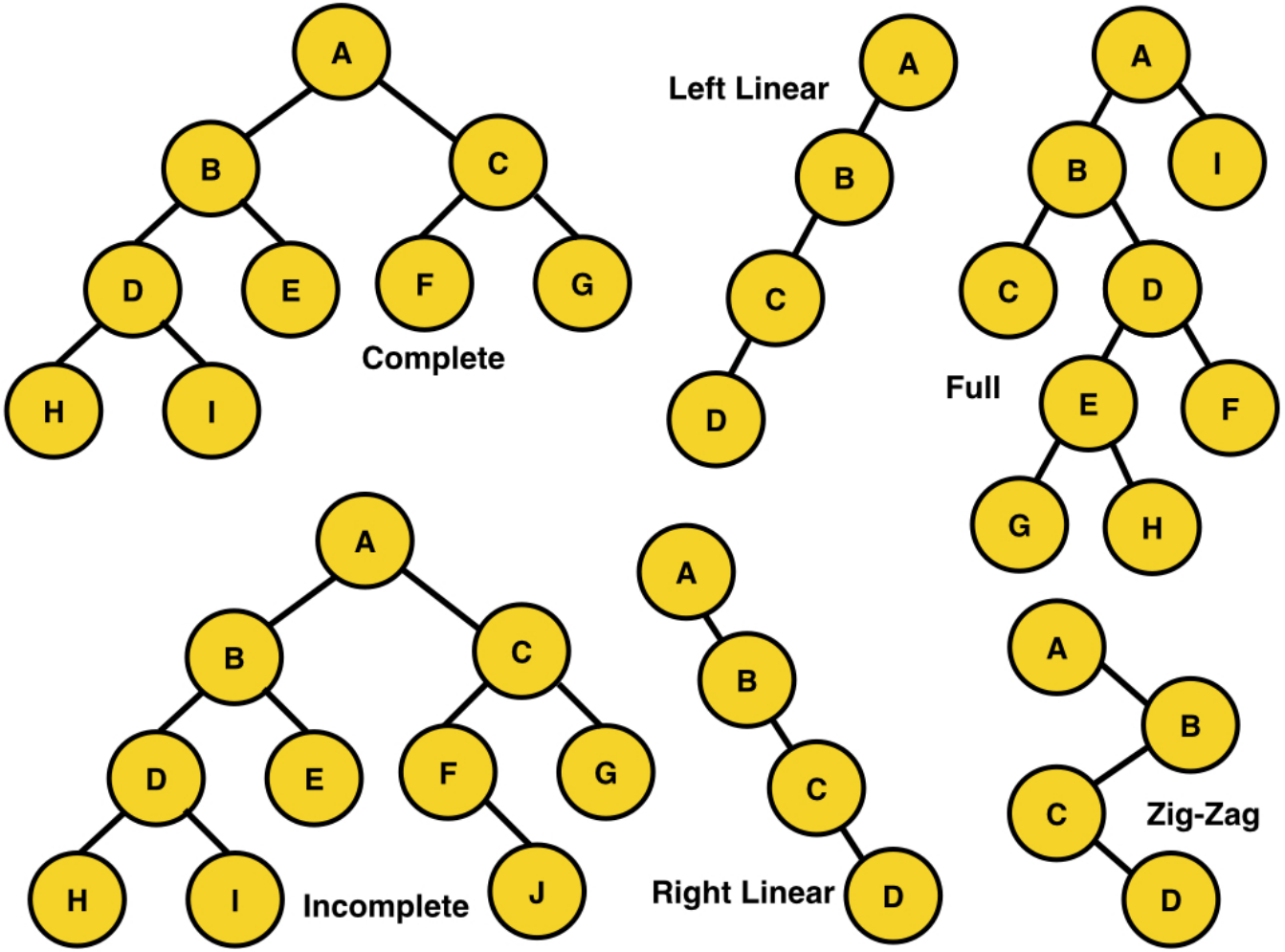
Examples of Trees

What Kind of Binary Trees are these?



Examples of Trees

What Kind of Binary Trees are these?

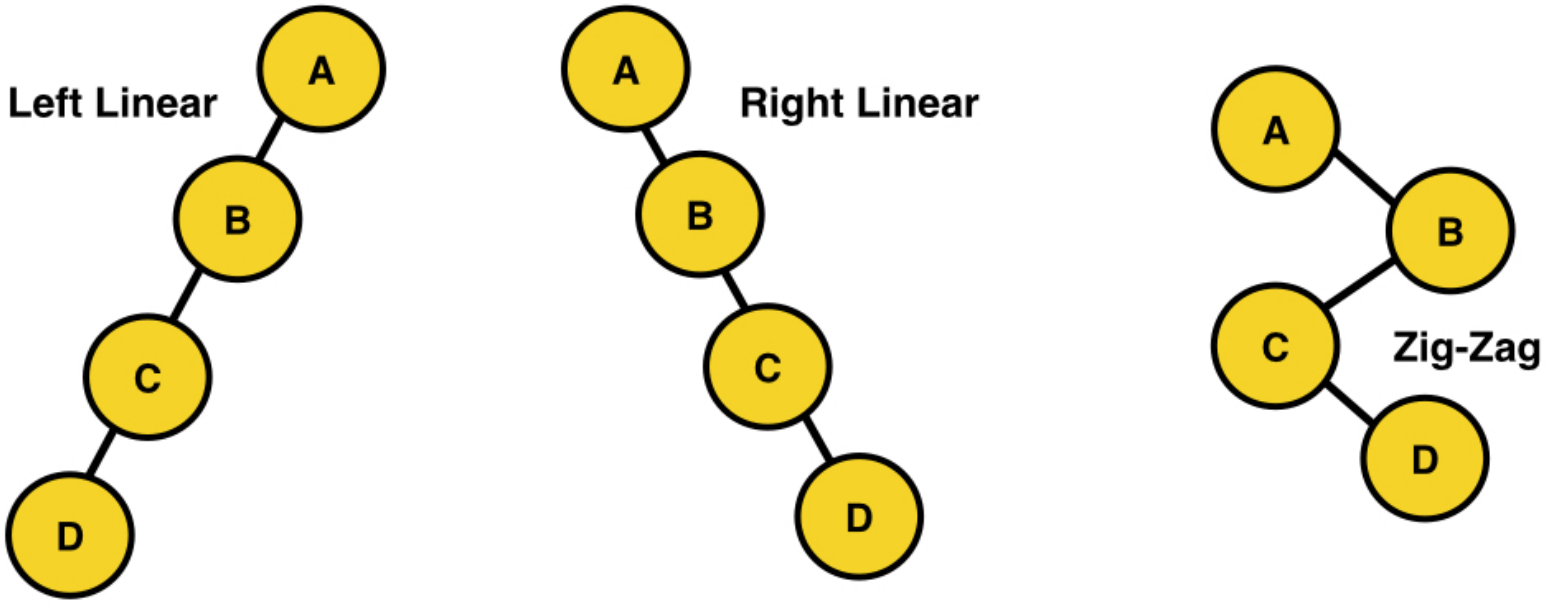


Operations on Trees

- Possible operations on a tree are:
  - Insertion
  - Traversal
  - Deletion
  - Rotation
  - Merge
- The manner in which these operations are performed depends on the **purpose** of the tree
- The asymptotic performance of these operations are dictated by the structure of the tree (tree height)
- In the worst case, number of node traversals is proportional to the longest path from the root to one of the leafs in the tree

Generalized Asymptotic Analysis

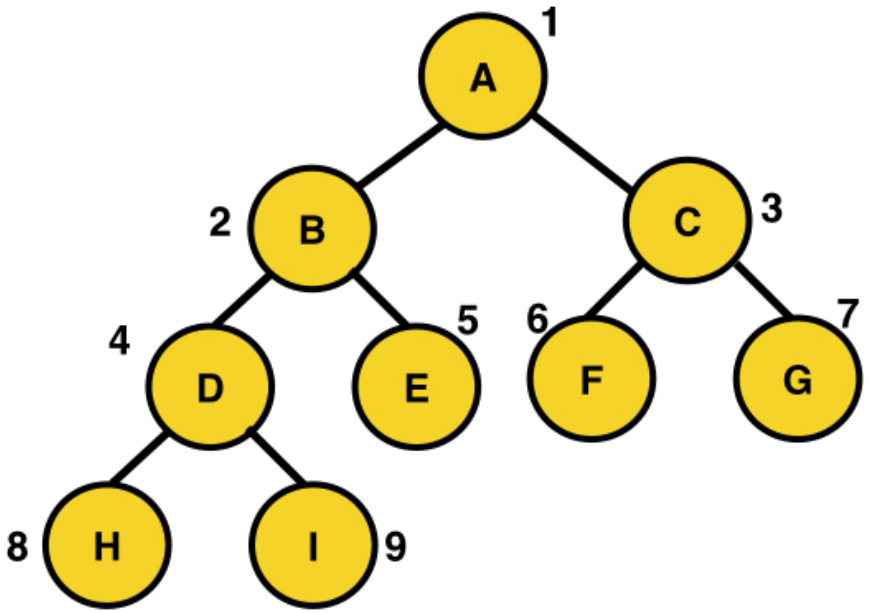
Left/Right Linear, Zig-Zag Tree



- Longest path length: `n-1`
- Asymptotic complexity of node traversal: O(n)

Generalized Asymptotic Analysis

Complete Trees



- Number each node in the tree from left to right
- The path length of each node i from the root is $\lfloor \log_2 i \rfloor$
- The largest path length is $\lfloor \log_2 n \rfloor$
- Asymptotic complexity node traversal: $O(\log n)$

Generalized Asymptotic Analysis

Lessons Learnt:

- Maintaining a **specific structure** in a tree can result in efficient retrieval of data
- Not keeping track of the tree structure can result in **skewed shapes** such as left/right linear or zig-zag
- No point in using a tree in these cases

Tree ADT Interface

General methods associated with the Tree ADT:

- `root()`: Returns the reference root of the tree
- `parent(p)`: Returns the parent of node `p`
- `children(p)`: Returns the collection of nodes that are immediate descendants of node `p`
- `numChildren(P)`: Return the count of the total number of children of node `p`
- `isInternal(p)`: Checks if node `p` is an internal node (not leaf)
- `isExternal(p)`: Checks if node `p` is an external node (no children)
- `isRoot(p)`: Checks if node `p` has no parent
- Helper functions: `size()`, `isEmpty()`, `iterator()`, `positions()`

Tree ADT Interface

```
public interface Position<E>{
  E getElement() throws IllegalStateException;
}
```

```
public interface Tree<E> extends Iterable<E>{
  Position<E> root();
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size();
  boolean isEmpty();
  Iterator<E> iterator();
  Iterable<Position<E>> positons();
}
```

Abstract Tree Class

- Some methods can be implemented without being aware of the underlying implementation of the tree
- Define an abstract class as only some of the methods specified in the `Tree` interface can be implemented
- How can the following methods be implemented based on future definitions of `root(p)`, `parent(p)`, `children(p)`, `numChildren(p)`?

```
public abstract class AbstractTree<E> implements Tree<E>{
  boolean isInternal(Position<E> p){...}
  boolean isExternal(Position<E> p){...}
  boolean isRoot(Position<E> p){...}
  boolean isEmpty(...)
}
```

Auxiliary `Tree` Methods

- **Depth of a node** `p`: `depth(p)`
  - Recursive implementation where at each recursive call, the parent of the current node is passed as the argument until the root node is reached
  - A counter of each recursive call is maintained
  - `depth(p) = depth(parent(p))+1` with base-case `isRoot(p)==True` where `depth(p)=0`)
- **Height of a subtree rooted at node** `p`: `height(p)`
  - Recursively compute the maximum height of the children and account for the edge connect the node `p` to the children
  - `height(p)=max[height(children(p))]+1` where the base-case is `isExternal(p)==True` where `height(p)=0`

Table of Contents

1. Introduction
2. Binary Trees
   - Heaps
   - Binary Search Trees
   - AVL Trees
   - Red-Black Tree
3. 2-3 Trees
4. B-Trees
5. Tries

What is a Binary Tree?

- **Simple** definition: Each node has a maximum of two children
- **Formal** definition: Each node has exactly two children where these children can be *empty*. Nodes with two empty children are leafs.
- **Recursive** definition: *Empty tree* or a node with *left* and *right* subtrees that are binary trees
- **Extended** binary trees: Explicitly show empty children with a $\square$

Binary Tree ADT

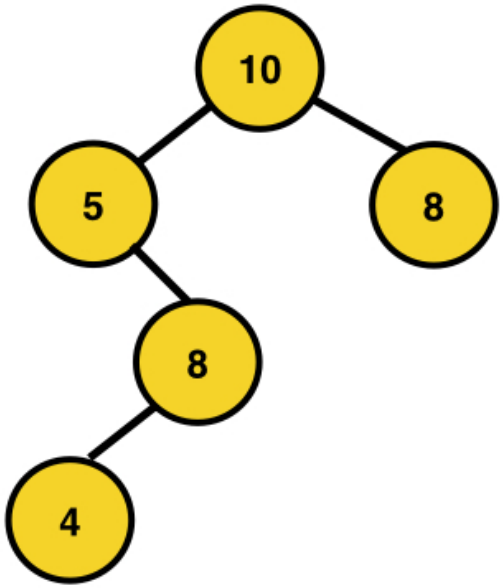
- For the binary tree, additional methods such as `left(p)`, `right(p)`, `sibling(p)` which return the appropriate reference are included
- Thus, the `Tree` interface can be extended to `BinaryTree` interface
- Certain methods defined in the `BinaryTree` interface do not depend on the underlying implementation of the tree
  - e.g. `sibling`, `numChildren`, `children`
- These are implemented in the `AbstractBinaryTree` class

```
public interface BinaryTree<E> extends Tree<E>{
  Position<E> left(Position<E> p) throws IllegalArgumentException;
  Position<E> right(Position<E> p) throws IllegalArgumentException;
  Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```

Abstract Binary Tree Class

How will the following functions be implemented using methods defined in the `BinaryTree` interface such as `left()`, `right()`, `parent()`, etc.?

```
public abstract class AbstractBinaryTree<E>
      extends AbstractTree<E> implements BinaryTree<E>{

  public Position<E> sibling(Position<E> p) {
    ...
  }

  public int numChildren(Position<E> p){
    ...
  }

  public Iterable<E> children(Position<E> p){
    ...
  }

}
```



Underlying Representation

- **Linked**: Each node in a tree will have two pointers (one pointing to the left child and the other pointing to the right child). Need to traverse through all nodes in a path from root to the destination node
- **Sequential**: All nodes are stored in a linear array, can directly access the child or parent node via indices obtained through simple computations

Underlying Representation

Linked Representation

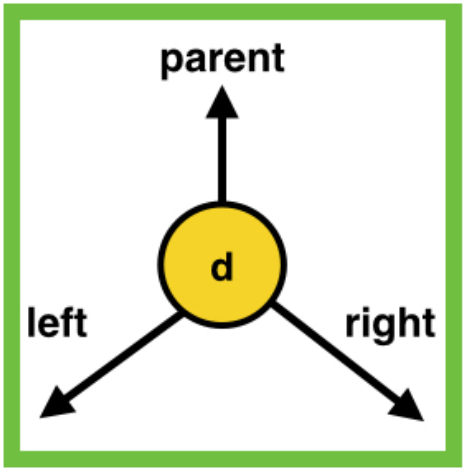
Linked Representation

```
protected static class Node<E> implements Position<E>{
  private E element;
  private Node<E> parent;
  private Node<E> left;
  private Node<E> right;

  public Node(E e, Node<E> p, Node<E> l, Node<E> r){
    element=e;
    parent=p;
    left=l;
    right=r;
  }

  public E getElement(){return element;}
  public Node<E> getParent(){return parent;}
  public Node<E> getLeft(){return left;}
  public Node<E> getRight(){return right;}

  public void setElement(E e){element=e;}
  public void setParent(Node<E> p){parent=p;}
  public void setLeft(Node<E> l){left=l;}
  public void setRight(Node<E> r){right=r;}
}
...
```

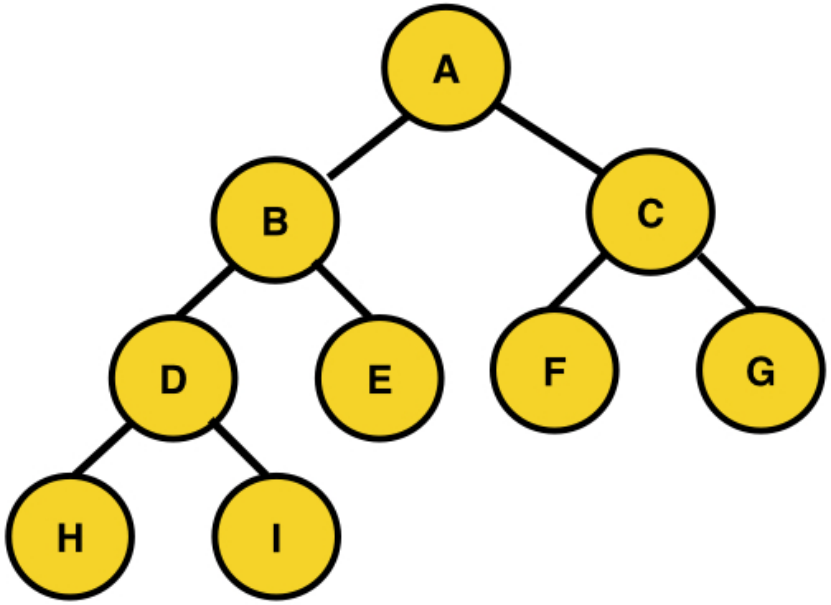
Inner reference variables

Accessor functions

Setting functions



Linked Representation



- Root node has no parent
- Can traverse the tree through the `left`, `right`, `parent` references
- Linked vs Sequential: **space** vs **accessibility**

Construction of a Linked Binary Tree

Operations that can be applied to a binary tree are:

- `addRoot(e)`: Create the first node (i.e. root) in an empty tree
- `addLeft(p,e)`: Given that `p` has no left child, a new node containing element `e` is appended as a left child for `p`
- `addRight(p,e)`: Given that `p` has no right child, a new node containing element `e` is appended as a right child for `p`
- `set(p,e)`: Replace the element stored in node `p` if it is not null
- `attach(p,t1,t2)`: Attaching two subtrees to node `p` if it is a leaf node where `t1` is the left subtree and `t2` is the right subtree
- `remove(p)`: Remove node at position `p` and attach the children of `p` to the parent of `p`
  - Only possible if the parent of `p` has one child

Need to check the validity of arguments and states!

Linked Binary Tree: Node Validity

```java
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{

  protected static class Node<E> implements Position<E>{
      ...
  }
  protected Node<E> validate(Position<E> p) throws IllegalArgumentException{
    if(!(p instanceof Node))
      throw new IllegalArgumentException("Not valid position type");
    Node<E> node = (Node<E>)p;
    if(node.getParent()==node)
      throw new IllegalArgumentException("p is no longer in the tree");
    return node;
  }

  protected Node<E> createNode(E e, Node<E> p, Node<E> l, Node<E> r){
    return new Node<E>(e, p, l, r);
  }

  protected Node<E> root = null;
  private int size=0;
  ...
}
```

Check if p is a valid and throw an exception otherwise:

1. **Check if p is an instance of Node**
2. **Cast p to node type**
3. **Check if node is an active component of the tree**
   - **If the node is no longer in use, its parent is set to itself**

Linked Binary Tree: Factory Method

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{

  protected static class Node<E> implements Position<E>{
      ...
  }
  protected Node<E> validate(Position<E> p) throws IllegalArgumentException{
    if(!(p instanceof Node))
      throw new IllegalArgumentException("Not valid position type");
    Node<E> node = (Node<E>)p;
    if(node.getParent()==node)
      throw new IllegalArgumentException("p is no longer in the tree");
    return node;
  }

  protected Node<E> createNode(E e, Node<E> p, Node<E> l, Node<E> r){
    return new Node<E>(e, p, l, r);
  }

  protected Node<E> root = null;
  private int size=0;
  ...
}
```

Factory design pattern that creates new instances of node

Linked Binary Tree: Internal Tree Fields

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{

  protected static class Node<E> implements Position<E>{
      ...
  }
  protected Node<E> validate(Position<E> p) throws IllegalArgumentException{
    if(!(p instanceof Node))
      throw new IllegalArgumentException("Not valid position type");
    Node<E> node = (Node<E>)p;
    if(node.getParent()==node)
      throw new IllegalArgumentException("p is no longer in the tree");
    return node;
  }

  protected Node<E> createNode(E e, Node<E> p, Node<E> l, Node<E> r){
    return new Node<E>(e, p, l, r);
  }

  protected Node<E> root = null;
  private int size=0;
  ...
}
```

Internal fields associated with a tree

Linked Binary Tree: Tree Structure

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{

  ...

  public LinkedBinaryTree(){}

  public int size(){return size;}
  public Position<E> root(){return root;}

  public Position<E> parent(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    return node.getParent();
  }
  public Position<E> left(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    return node.getLeft();
  }
  public Position<E> right(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    return node.getRight();
  }

  ...

}
```

Constructor for the Linked Binary Tree

Linked Binary Tree: Adding Nodes

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{

  ...

  public LinkedBinaryTree(){}

  public int size(){return size;}
  public Position<E> root(){return root;}

  public Position<E> parent(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    return node.getParent();
  }
  public Position<E> left(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    return node.getLeft();
  }
  public Position<E> right(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    return node.getRight();
  }

  ...

}
```

Accessor methods that throw exception if p is invalid

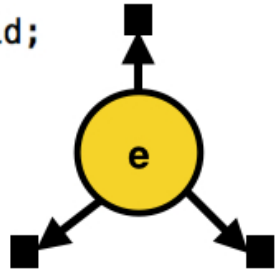
Linked Binary Tree: Adding Nodes

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public Position<E> addRoot(E e) throws IllegalStateException{
    if(!isEmpty()) throw new IllegalStateException("Tree is not empty");
    root=createNode(e, null, null, null);
    size=1;
    return root;
  }
  public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getLeft()!=null)
      throw new IllegalArgumentException("p already has a left child");
    Node<E> leftChild=createNode(e, node, null, null);
    node.setLeft(leftChild);
    size++;
    return leftChild;
  }
  public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getRight()!=null)
      throw new IllegalArgumentException("p already has a right child");
    Node<E> rightChild=createNode(e, node, null, null);
    node.setRight(rightChild);
    size++;
    return rightChild;
  }
  ...
}
```

Modifier methods that add nodes to the tree (size counter is incremented)

Linked Binary Tree: Adding Nodes

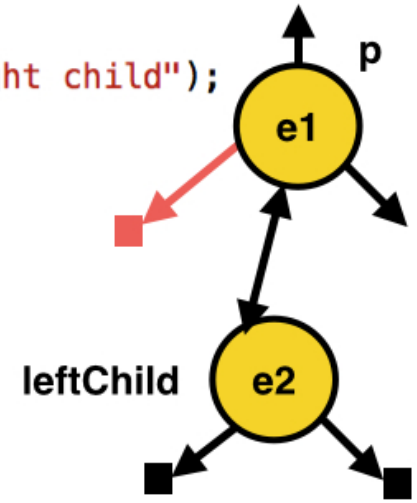
```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public Position<E> addRoot(E e) throws IllegalStateException{
    if(!isEmpty()) throw new IllegalStateException("Tree is not empty");
    root=createNode(e, null, null, null);
    size=1;
    return root;
  }
  public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getLeft()!=null)
      throw new IllegalArgumentException("p already has a left child");
    Node<E> leftChild=createNode(e, node, null, null);
    node.setLeft(leftChild);
    size++;
    return leftChild;
  }
  public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getRight()!=null)
      throw new IllegalArgumentException("p already has a right child");
    Node<E> rightChild=createNode(e, node, null, null);
    node.setRight(rightChild);
    size++;
    return rightChild;
  }
  ...
}
```



Creates a root node if the tree is empty

Linked Binary Tree: Adding Nodes

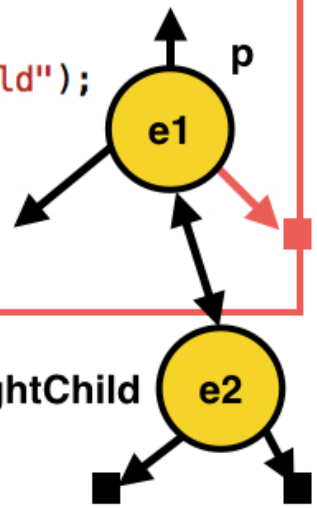
```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public Position<E> addRoot(E e) throws IllegalStateException{
    if(!isEmpty()) throw new IllegalStateException("Tree is not empty");
    root=createNode(e, null, null, null);
    size=1;
    return root;
  }
  public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getLeft()!=null)
      throw new IllegalArgumentException("p already has a left child");
    Node<E> leftChild=createNode(e, node, null, null);
    node.setLeft(leftChild);
    size++;
    return leftChild;
  }
  public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getRight()!=null)
      throw new IllegalArgumentException("p already has a right child");
    Node<E> rightChild=createNode(e, node, null, null);
    node.setRight(rightChild);
    size++;
    return rightChild;
  }
  ...
}
```



Add a left child if p has no left child already

Linked Binary Tree: Adding Nodes

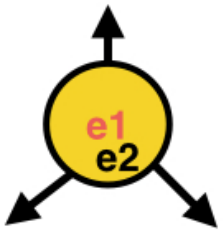
```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public Position<E> addRoot(E e) throws IllegalStateException{
    if(!isEmpty()) throw new IllegalStateException("Tree is not empty");
    root=createNode(e, null, null, null);
    size=1;
    return root;
  }
  public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getLeft()!=null)
      throw new IllegalArgumentException("p already has a left child");
    Node<E> leftChild=createNode(e, node, null, null);
    node.setLeft(leftChild);
    size++;
    return leftChild;
  }
  public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(node.getRight()!=null)
      throw new IllegalArgumentException("p already has a right child");
    Node<E> rightChild=createNode(e, node, null, null);
    node.setRight(rightChild);
    size++;
    return rightChild;
  }
  ...
}
```



Add a right child if p has no right child already

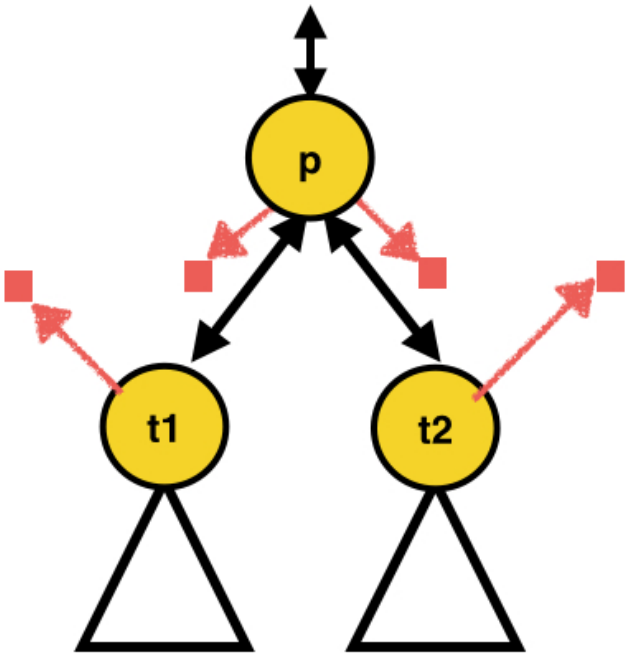
Linked Binary Tree: Setting Value

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public E set(Position<E> p, E e) throws IllegalArgumentException{
    Node<E> node = validate(p);
    E tempE = node.getElement();
    node.setElement(e);
    return tempE;
  }
  ...
}
```



Linked Binary Tree: Appending Sub-trees

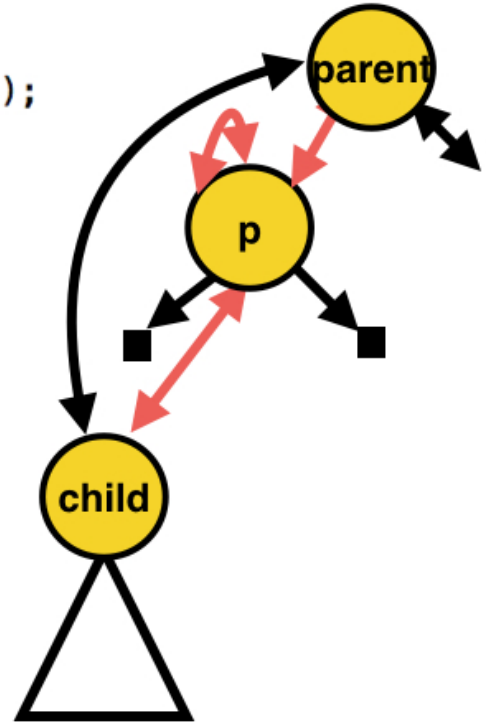
```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public void attach(Position<E> p, LinkedBinaryTree<E> t1,
                     LinkedBinaryTree<E> t2) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(isInternal(p)) throw new IllegalArgumentException("p is not a leaf node");
    size=size+t1.size()+t2.size();
    if(!t1.isEmpty()){
      t1.root.setParent(node);
      node.setLeft(t1.root);
      t1.root=null;
      t1.size=0;
    }
    if(!t2.isEmpty()){
      t2.root.setParent(node);
      node.setRight(t2.root);
      t2.root=null;
      t2.size=0;
    }
  }
  ...
}
```



Append two subtrees to p if it is a leaf node

Linked Binary Tree: Removing a Node

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>{
  ...
  public E remove(Position<E> p) throws IllegalArgumentException{
    Node<E> node = validate(p);
    if(numChildren(p)==2) throw new IllegalArgumentException("p has two children");
    Node<E> child = (node.getLeft()!=null ? node.getLeft():node.getRight());
    if(child!=null)
      child.setParent(node.getParent());
    if(node==root)
      root=child;
    else{
      Node<E> parent = validate(node.getParent());
      if(parent.getRight()!=null)
        parent.setLeft(child);
      else
        parent.setRight(child);
    }
    size--;
    E temp = node.getElement();
    node.setElement(null);
    node.setLeft(null);
    node.setRight(null);
    node.setParent(node);
    return temp;
  }
  ...
}
```

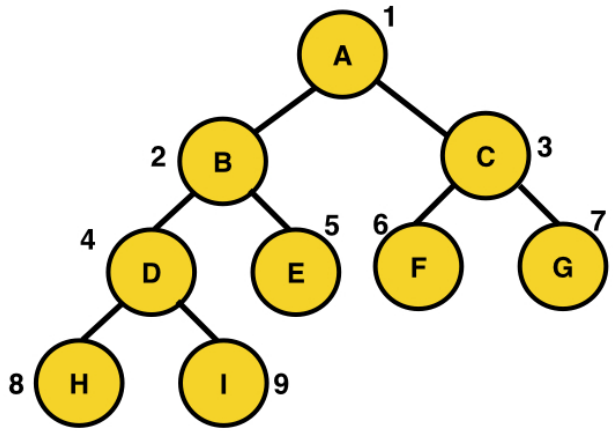


If p has only one child, set the child of p to be the child of the parent of p

Underlying Representation

Sequential Representation

Sequential Binary Tree

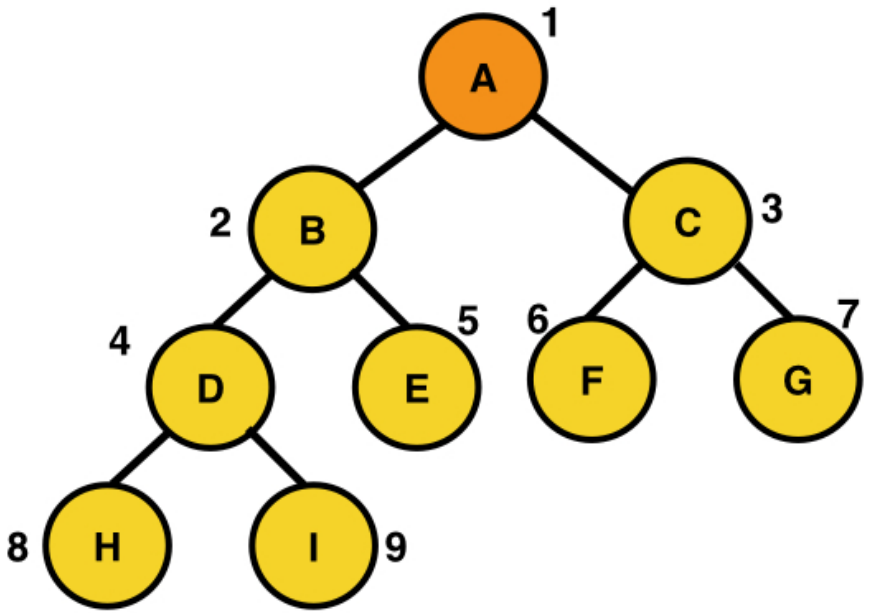


A:		A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

- Label each node from left to right level by level
- Arrange these nodes in the array at index mapped to the label
- Can directly compute the indices at which the **parent**, **left child**, or **right child** of a node resides
- Can also directly determine whether nodes are **roots** or **indices**

Sequential Binary Tree

Root Node

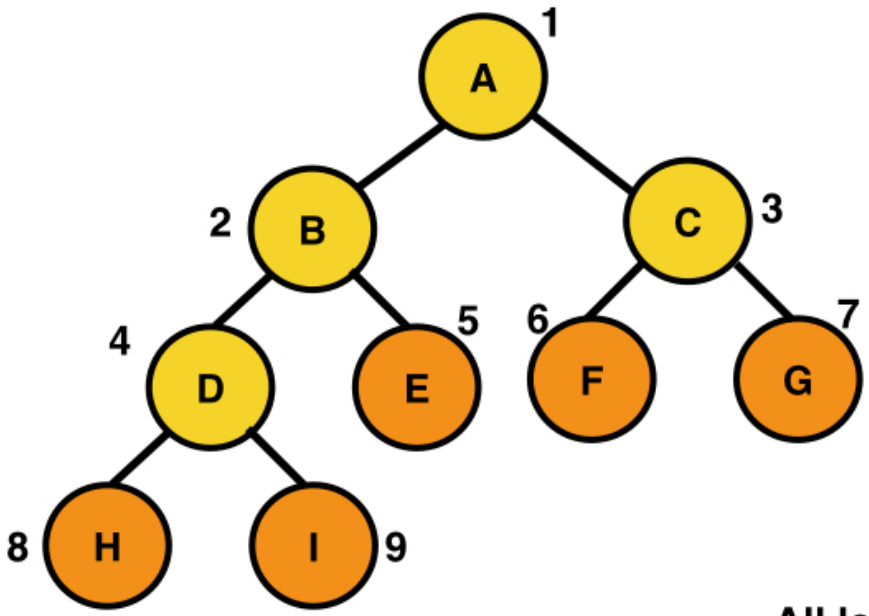


Root is at index 1

A:		A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

Sequential Binary Tree

Leaf Nodes

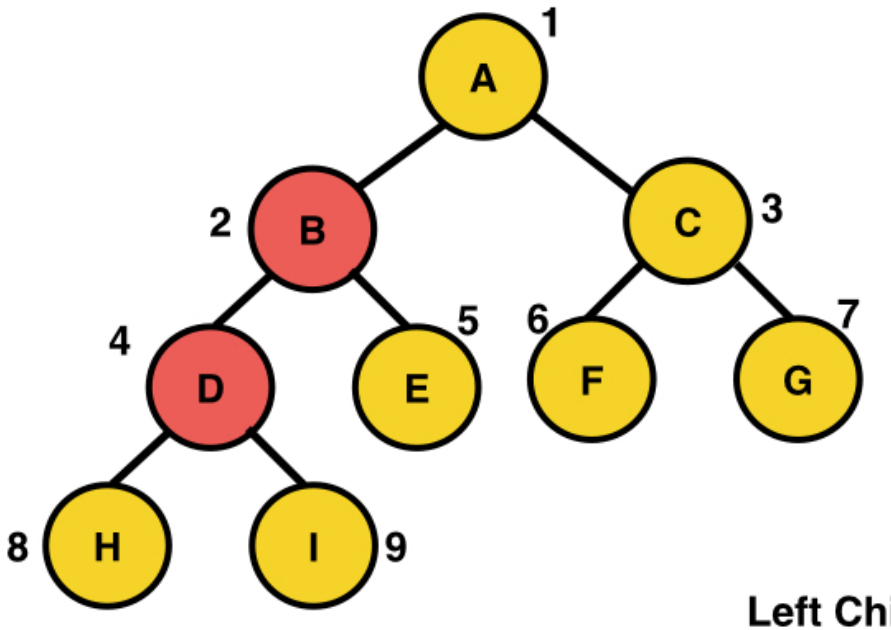


All leaf nodes reside at is at indices 5,6,7,8,9 (index*2>n)

A:		A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

Sequential Binary Tree

Left Child



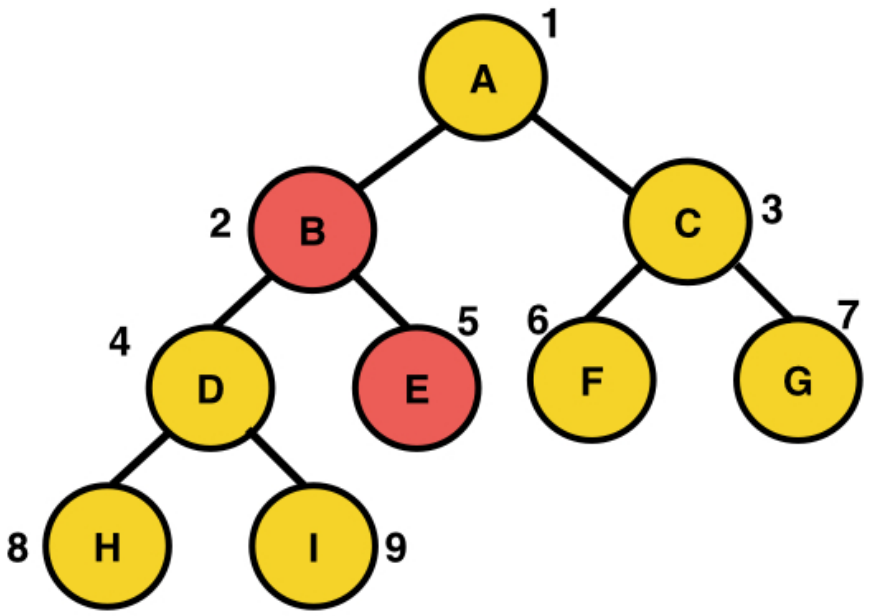
Left Child of Node i: 2i

Left Child of B(2): 4

A:		A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

Sequential Binary Tree

Right Child



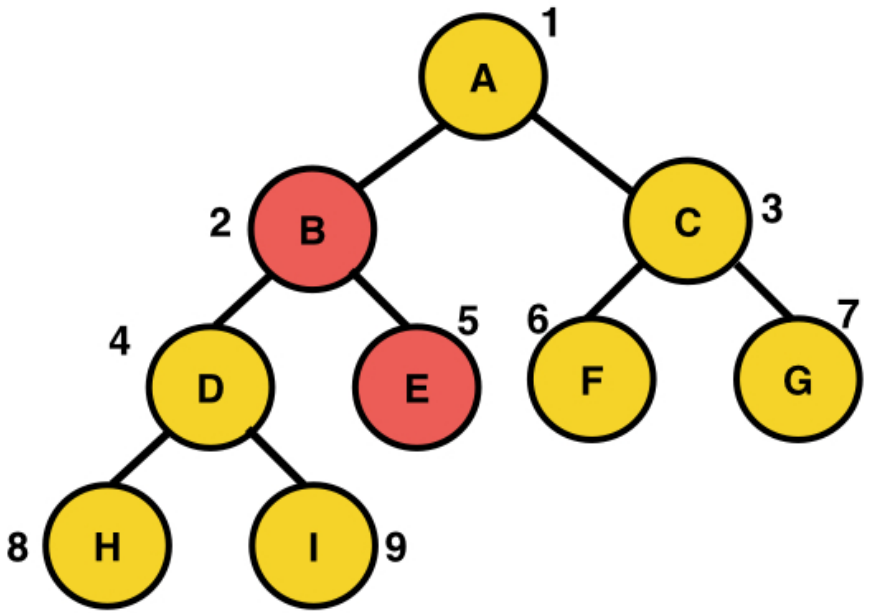
Right Child of Node i: 2i+1

Right Child of B(2): 5

A:		A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

Sequential Binary Tree

Parent of a Node



Parent of Node i: $\lfloor i/2 \rfloor$

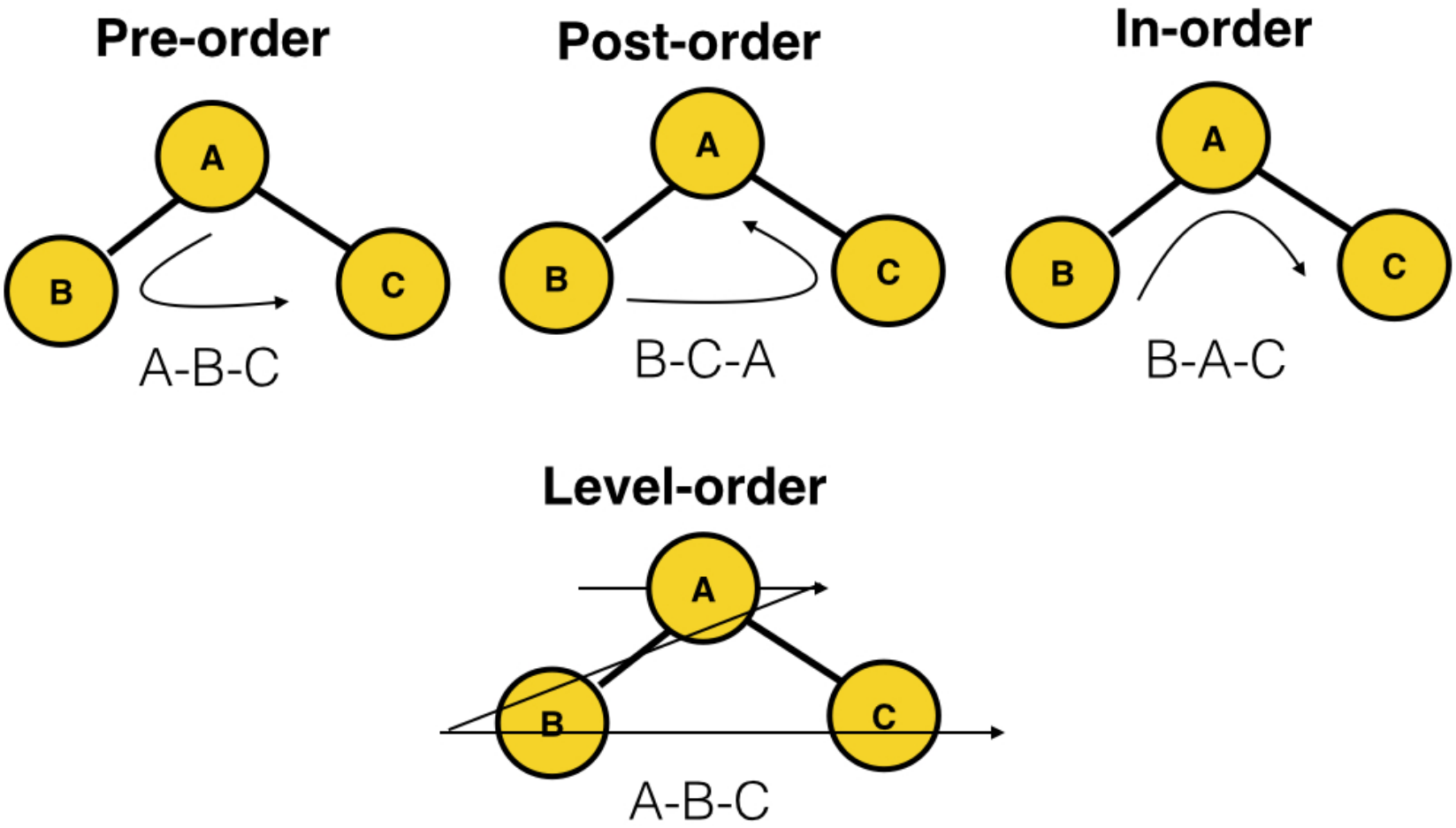
Parent of B(2): 2

A:		A	B	C	D	E	F	G	H	I
	0	1	2	3	4	5	6	7	8	9

Tree Traversal

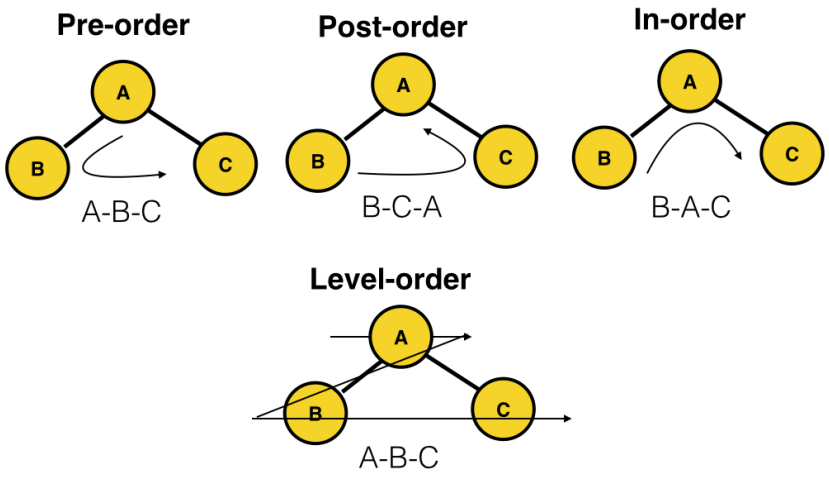
Tree Traversal

Suppose that you will have to visit each node in a linked tree only once. How will you go about doing that?



Tree Traversal

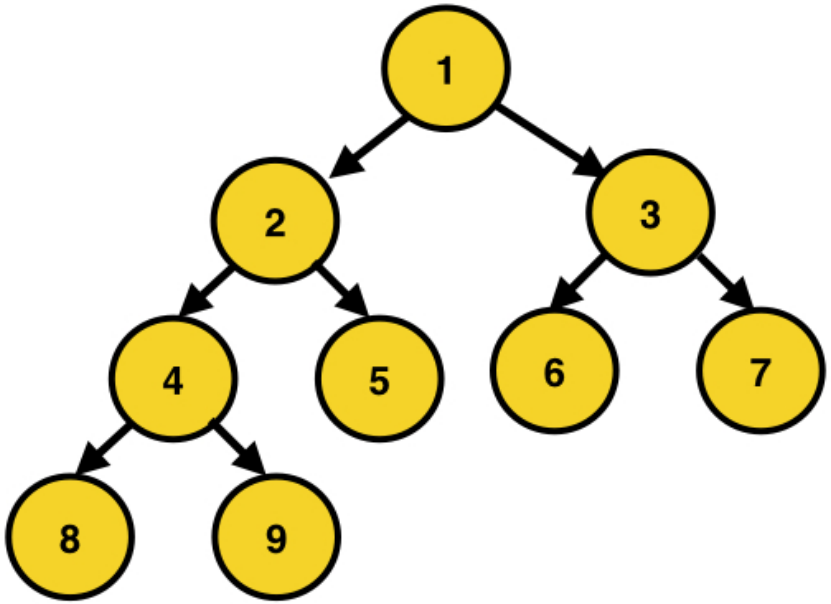
Generalized definitions of each traversal



- **Pre-order**: Visit root, then visit left subtree and then finally the right subtree (General Tree: Visit root and then children)
- **In-order**: Visit left subtree, then root and then finally the right subtree (Only applicable for binary tree)
- **Post-order**: Visit left subtree, then right subtree and finally the root (General Tree: Visit children from left to right and then root)
- **Level-order**: Visit nodes at each level from left to right

Tree Traversal

Example of Traversal



Pre-order: 1 2 4 8 5 9 3 6 7
In-order: 8 4 9 2 5 1 6 3 7
Post-order: 8 9 4 5 2 6 7 3 1
Level-order: 1 2 3 4 5 6 7 8 9

Tree Traversal

- Implementation composed of two main methods: `iterator()` and `positions()`
- `iterator()`: Returns an iterator to access each element in the tree
- `positions()`: Returns a collection of all positions in the tree (references to individual elements in the tree)

Tree Traversal: Helper Functions

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  private class ElementIterator implements Iterator<E>{
    Iterator<Position<E>> posIterator = positions().iterator();
    public boolean hasNext(){return posIterator.hasNext();}
    public E next(){return posIterator.next().getElement();}
    public void remove(){posIterator.remove();}
  }

  public Iterator<E> iterator(){return new ElementIterator();}

  public Iterable<Position<E>> positions() {...}
  ...
}
```

Helper functions for tree traversal

Tree Traversal: Helper Functions

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  private class ElementIterator implements Iterator<E>{
    Iterator<Position<E>> posIterator = positions().iterator();
    public boolean hasNext(){return posIterator.hasNext();}
    public E next(){return posIterator.next().getElement();}
    public void remove(){posIterator.remove();}
  }

  public Iterator<E> iterator(){return new ElementIterator();}

  public Iterable<Position<E>> positions() {...}
  ...
}
```

Private inner class that implements the Iterator<E> class

- **Internal field posIterator converts *Iterable* collection to an iterator**
- **Supports two accessor methods: *hasNext* and *next***
- ***remove* function eliminates the current element in the iterator**

Tree Traversal: Helper Functions

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  private class ElementIterator implements Iterator<E>{
    Iterator<Position<E>> posIterator = positions().iterator();
    public boolean hasNext(){return posIterator.hasNext();}
    public E next(){return posIterator.next().getElement();}
    public void remove(){posIterator.remove();}
  }

  public Iterator<E> iterator(){return new ElementIterator();}

  public Iterable<Position<E>> positions() {...}
  ...
}
```

Overrides the *iterator* constructor to return a new iterator instance defined by the ElementIterator class

Tree Traversal: Helper Functions

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  private class ElementIterator implements Iterator<E>{
    Iterator<Position<E>> posIterator = positions().iterator();
    public boolean hasNext(){return posIterator.hasNext();}
    public E next(){return posIterator.next().getElement();}
    public void remove(){posIterator.remove();}
  }

  public Iterator<E> iterator(){return new ElementIterator();}

  public Iterable<Position<E>> positions() {...}
  ...
}
```

Returns a collection composed of references to various elements in the tree traversed using one of the tree traversal methods: *preOrder()*, *postOrder()*, *inOrder()* or *levelOrder()*

Pre-order Tree Traversal Implementation

Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Calls the appropriate tree traversal function

Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Creates a collection of references that can be iterated

Tree Traversal: Pre-order

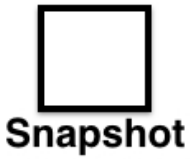
```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

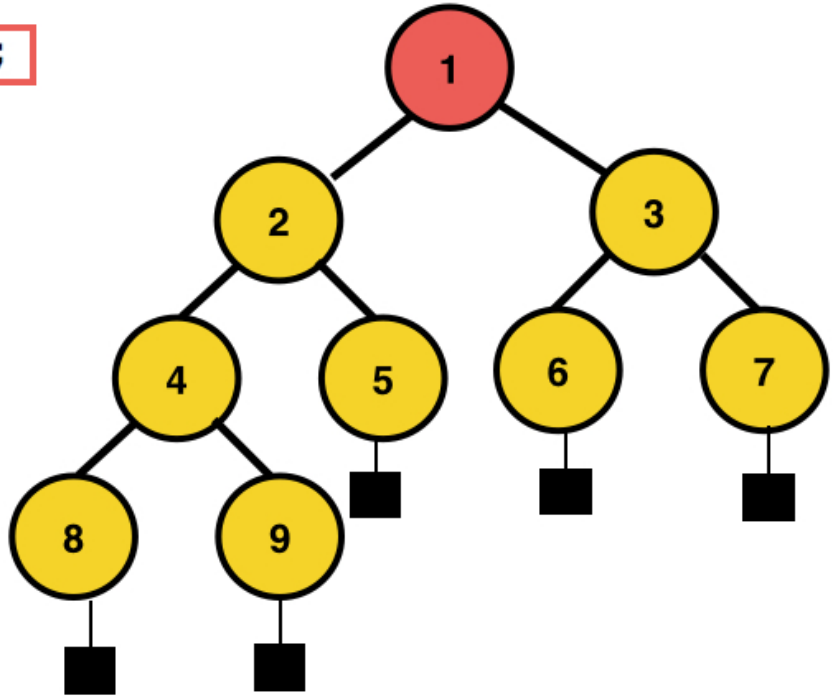
Recursive function that:

1) **First adds the root of the current subtree to the collection**
2) **Recursively call the function on each child**

Traversal of Linked Trees: Post-Order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```



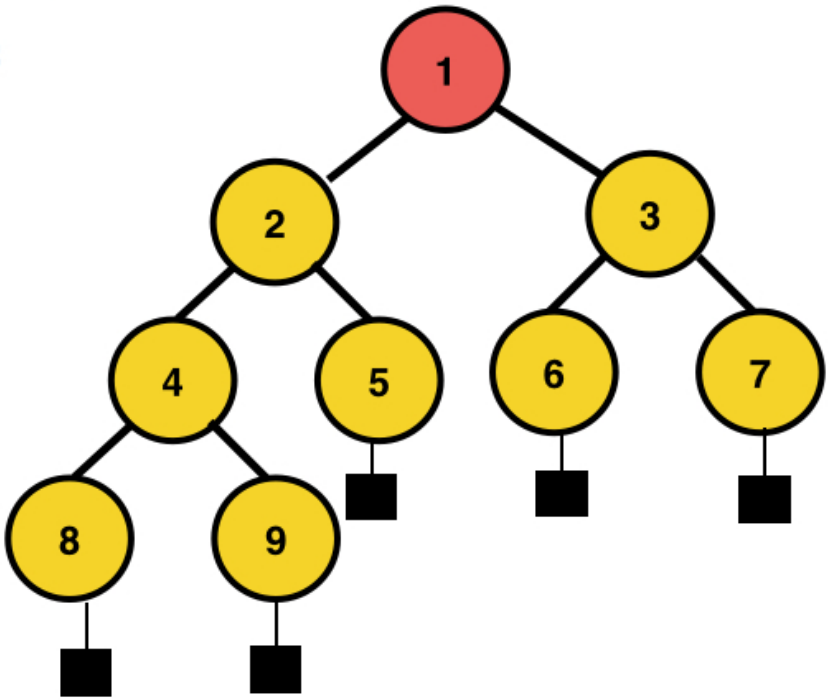


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Pos1

Snapshot

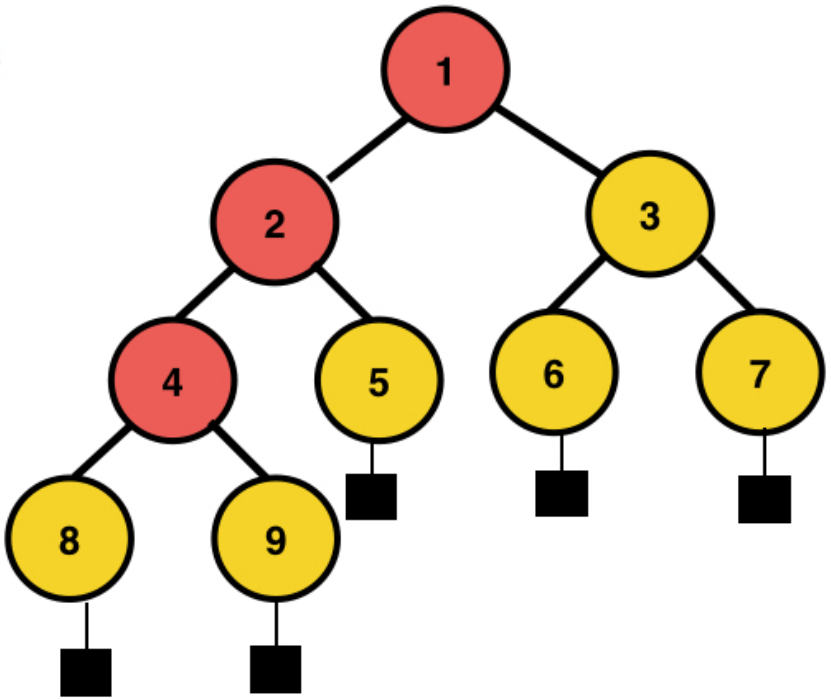


Traversal of Linked Trees: Pre-Order (Recursive)

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Pos1	Pos2	Pos4

Snapshot

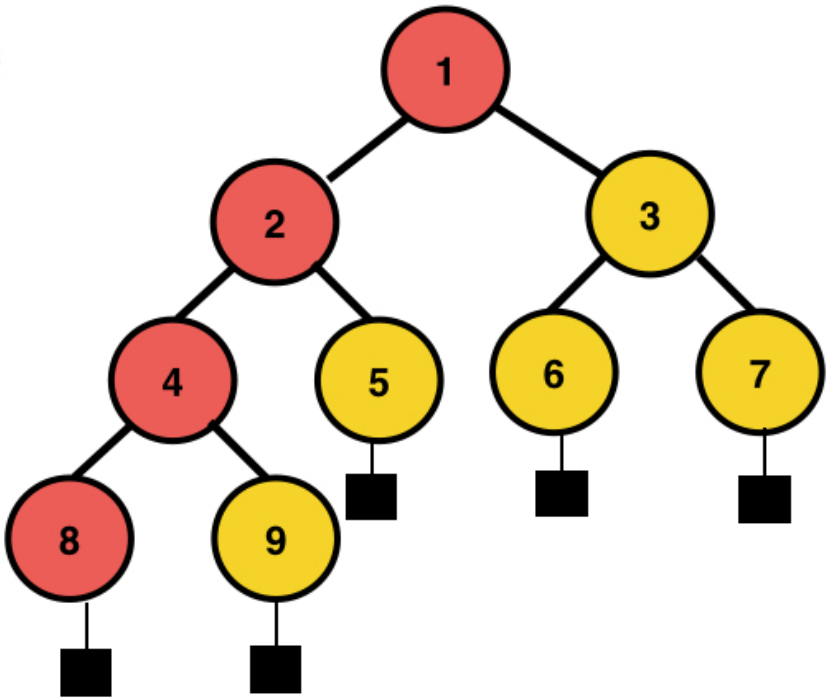


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Pos1	Pos2	Pos4	Pos8

Snapshot

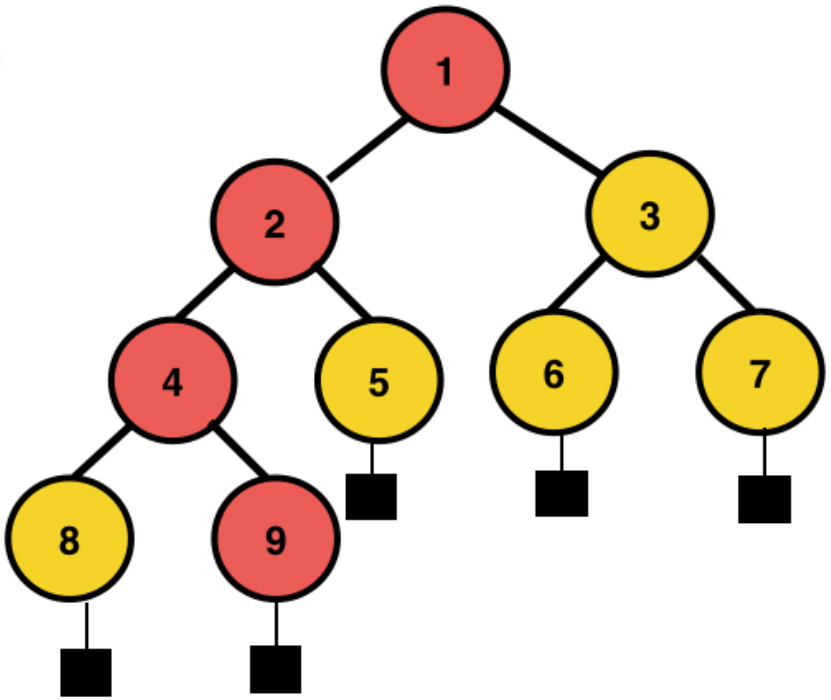


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Pos1	Pos2	Pos4	Pos8	Pos9

Snapshot

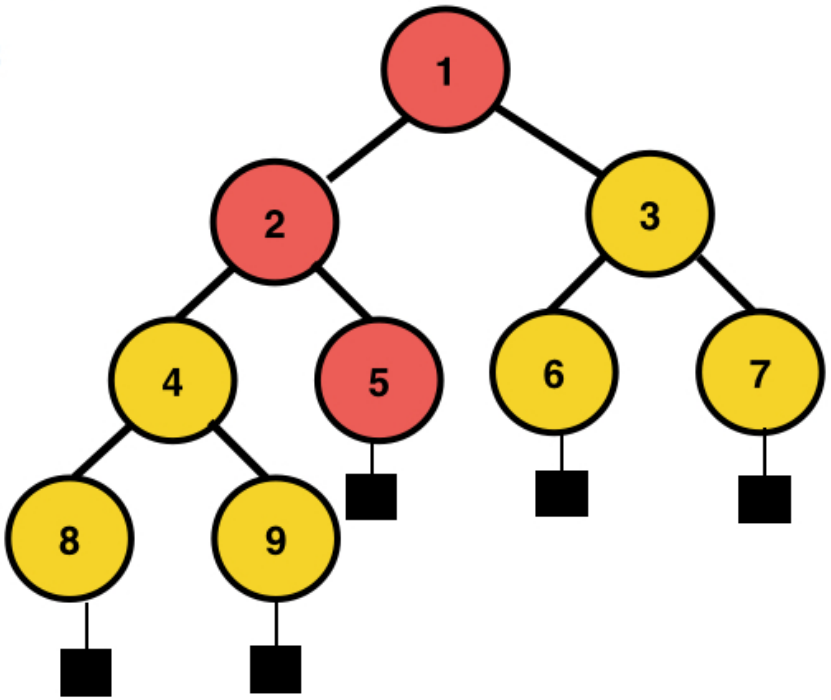


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Pos1	Pos2	Pos4	Pos8	Pos9	Pos5

Snapshot

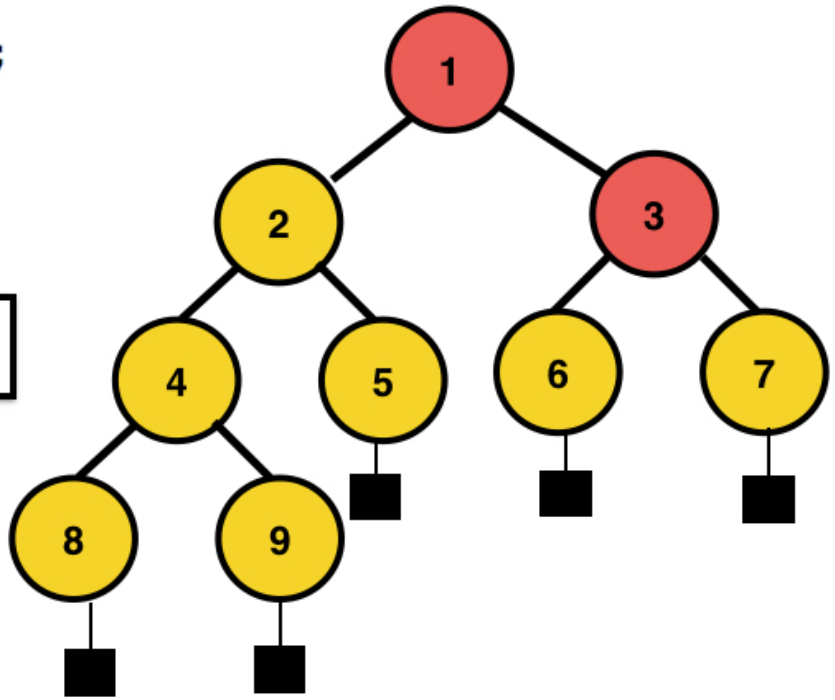


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Pos1	Pos2	Pos4	Pos8	Pos9	Pos5	Pos3

Snapshot

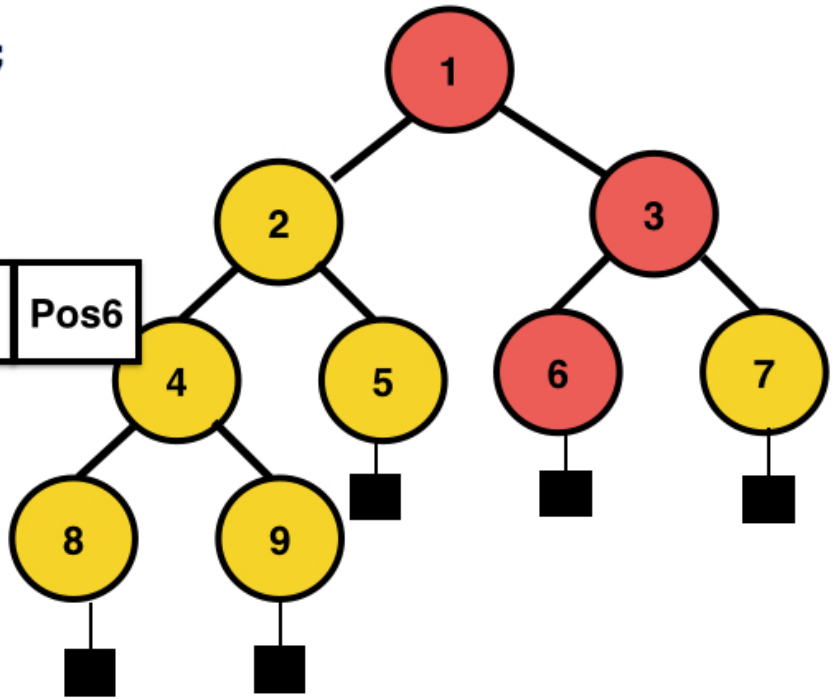


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

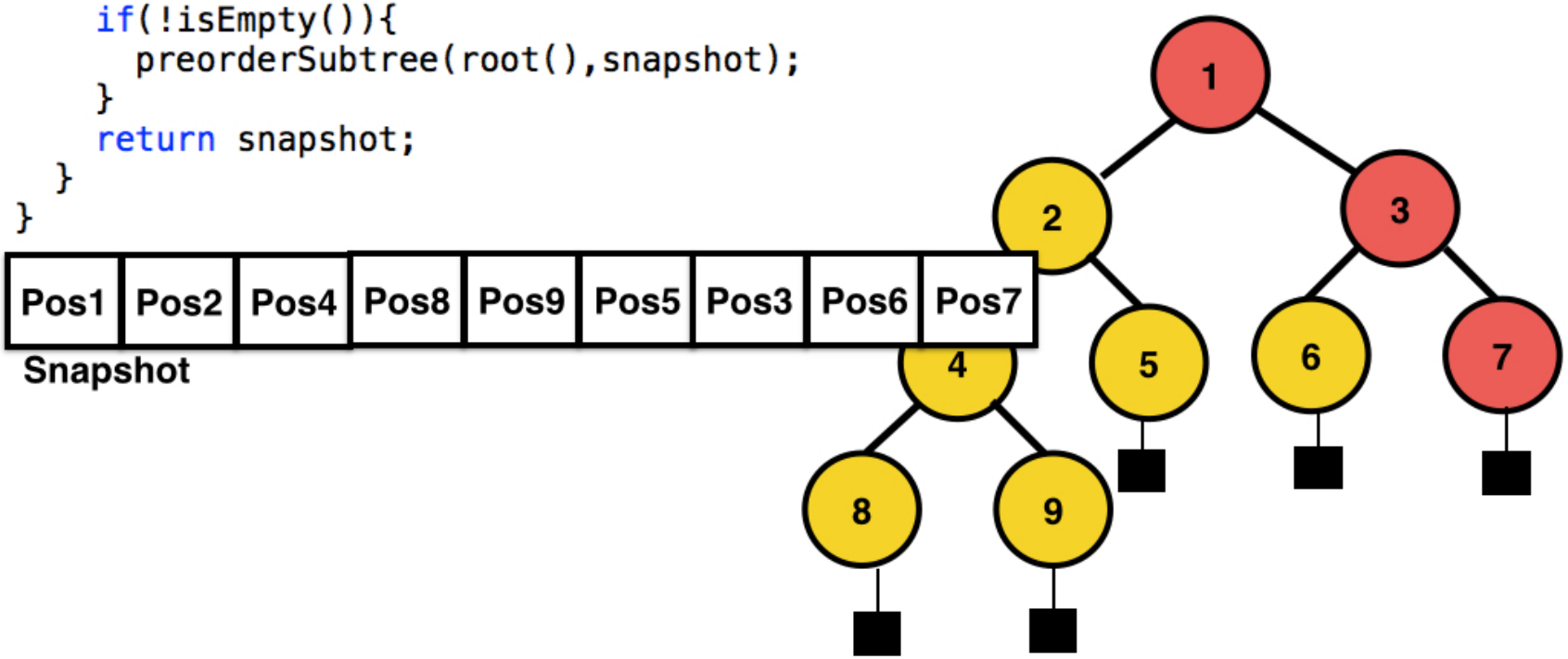
Pos1	Pos2	Pos4	Pos8	Pos9	Pos5	Pos3	Pos6

Snapshot



Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

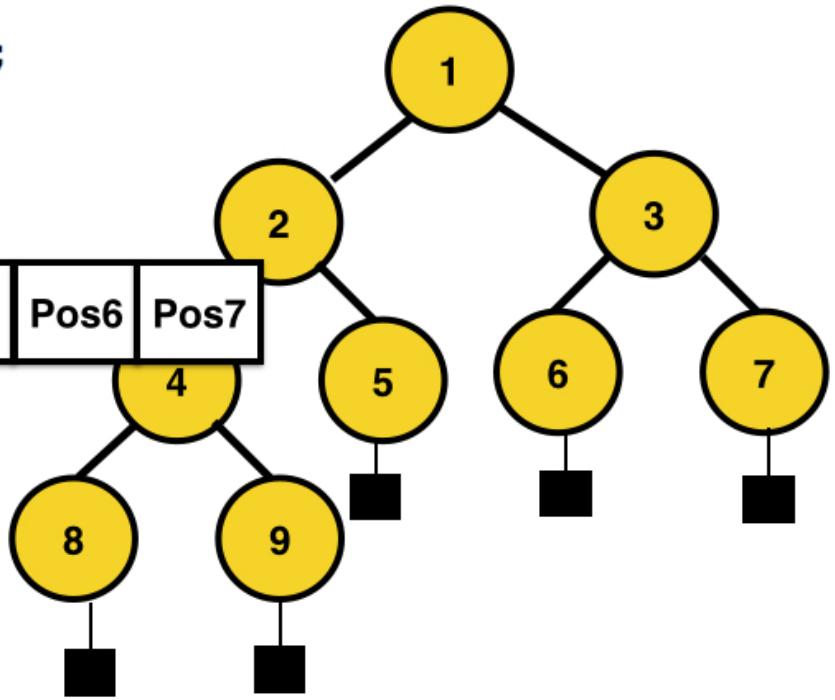


Tree Traversal: Pre-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return preorder();}

  private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){
    snapshot.add(p);
    for(Position<E> c : children(p))
      preorderSubtree(c,snapshot);
  }

  public Iterable<Position<E>> preorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      preorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

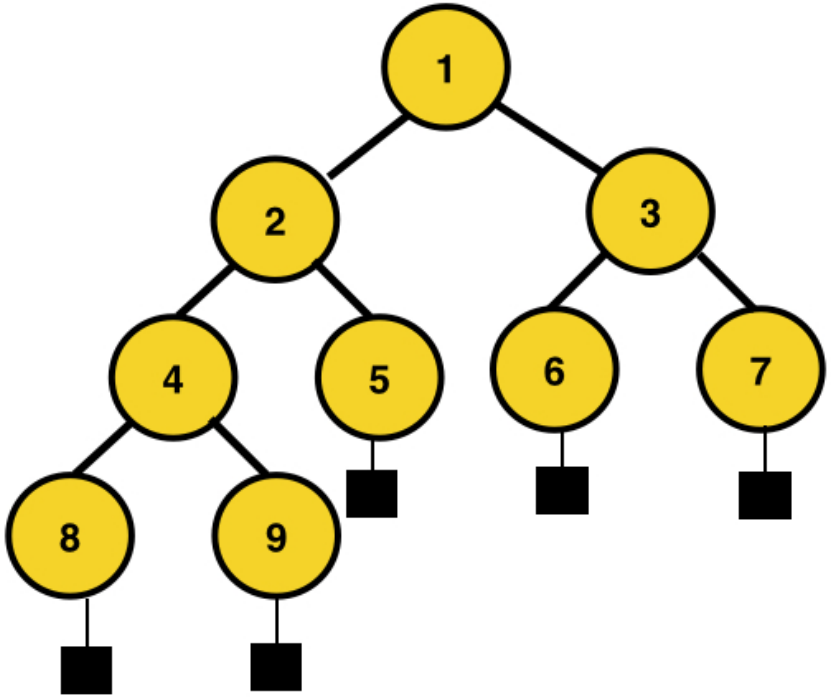
Pos1	Pos2	Pos4	Pos8	Pos9	Pos5	Pos3	Pos6	Pos7

Snapshot



Traversal of Linked Trees: Pre-Order

- Recursion allows for concise implementation of the pre-order traversal function
- How will you implement pre-order traversal *without* using recursion?



Tree Traversal: Post-order

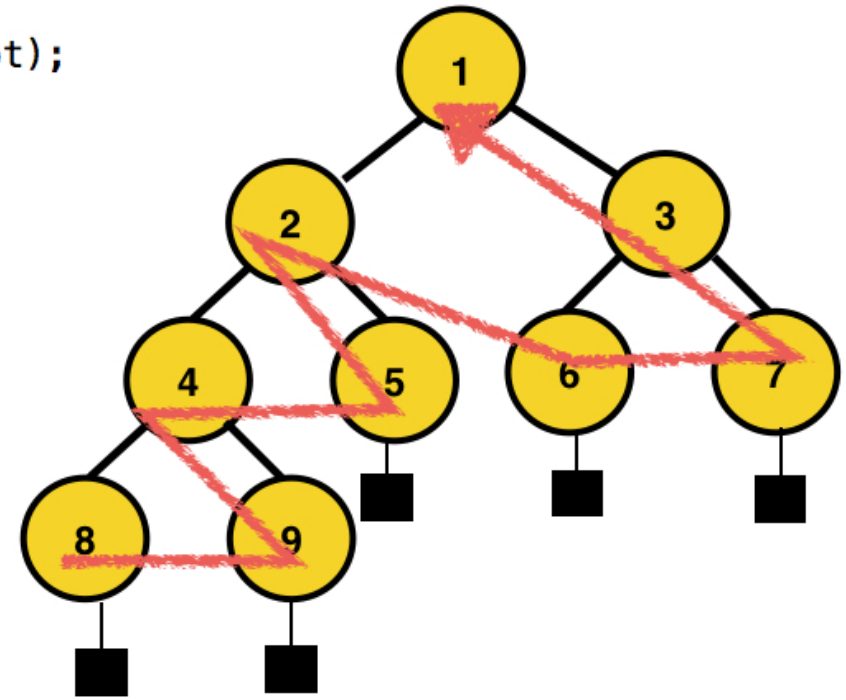
Post-order Traversal

Tree Traversal: Post-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return postorder();}

  private void postorderSubtree(Position<E> p, List<Position<E>> snapshot){
    for(Position<E> c : children(p))
      postorderSubtree(c,snapshot);
    snapshot.add(p);
  }

  public Iterable<Position<E>> postorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      postorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Recursive function that:

1) **Recursively call the function on each child**
2) **Add the root to the collection**



Tree Traversal: In-order

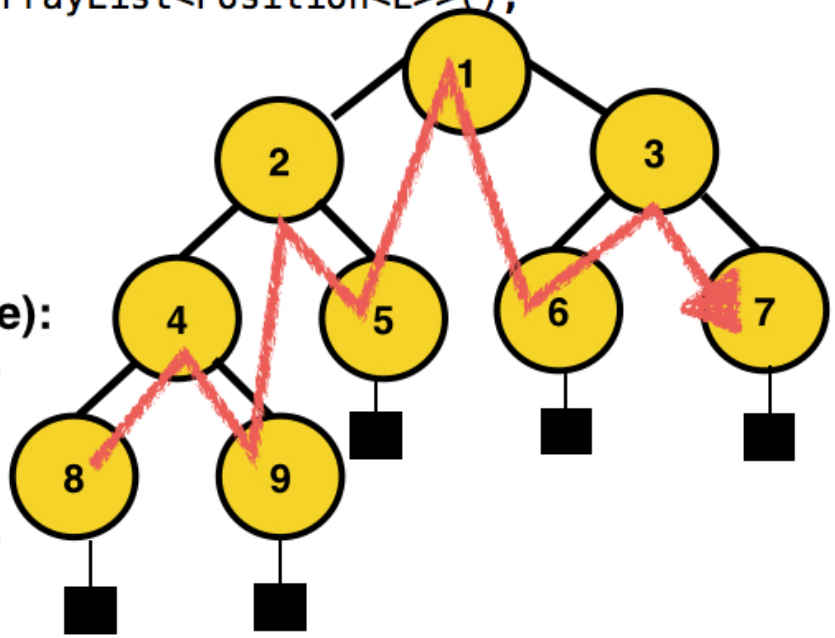
In-order Traversal

Tree Traversal: In-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return inorder();}

  private void inorderSubtree(Position<E> p, List<Position<E>> snapshot){
    Iterator<Position<E>> cList = children(p).iterator();
    if(cList.hasNext())
      inorderSubtree(cList.next(),snapshot);
    snapshot.add(p);
    if(cList.hasNext())
      inorderSubtree(cList.next(),snapshot);
  }

  public Iterable<Position<E>> inorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    if(!isEmpty()){
      inorderSubtree(root(),snapshot);
    }
    return snapshot;
  }
}
```

Recursive function (only for binary tree):

1) **Recursively call the function on the left child**
2) **Add the root to the collection**
3) **Recursively call the function on the right child**

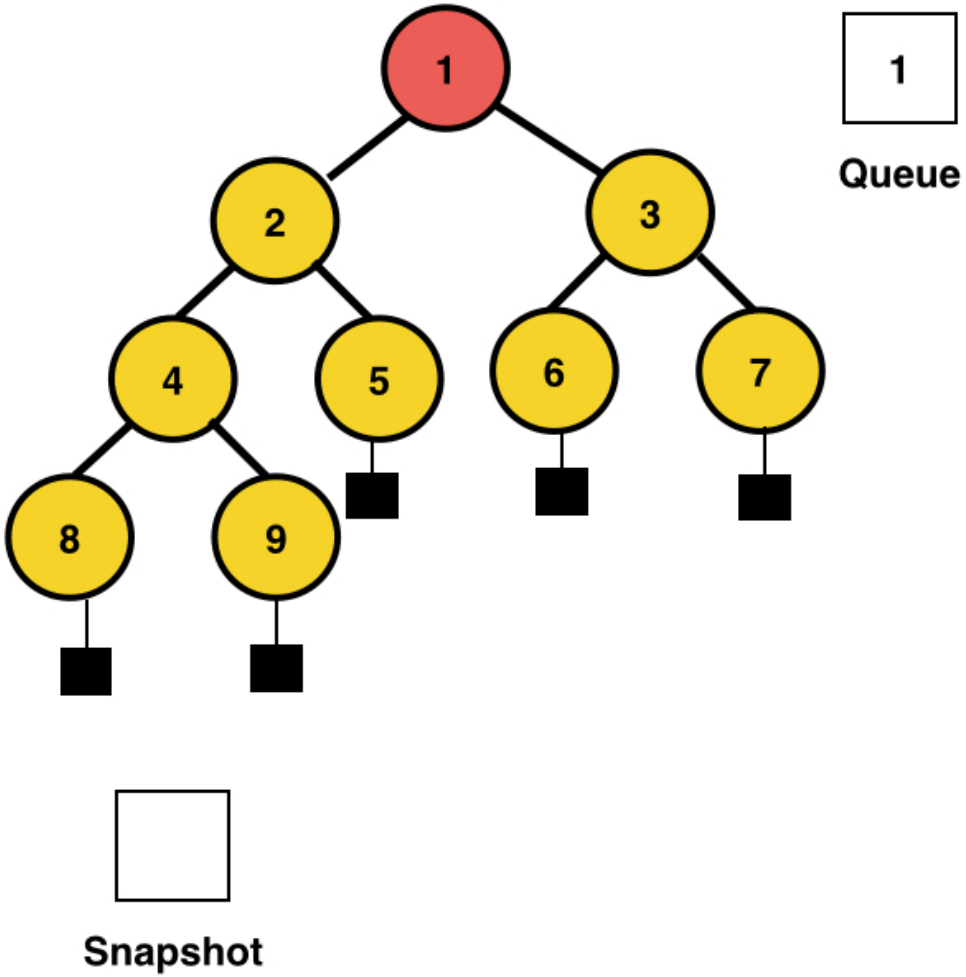


Traversal of Linked Trees

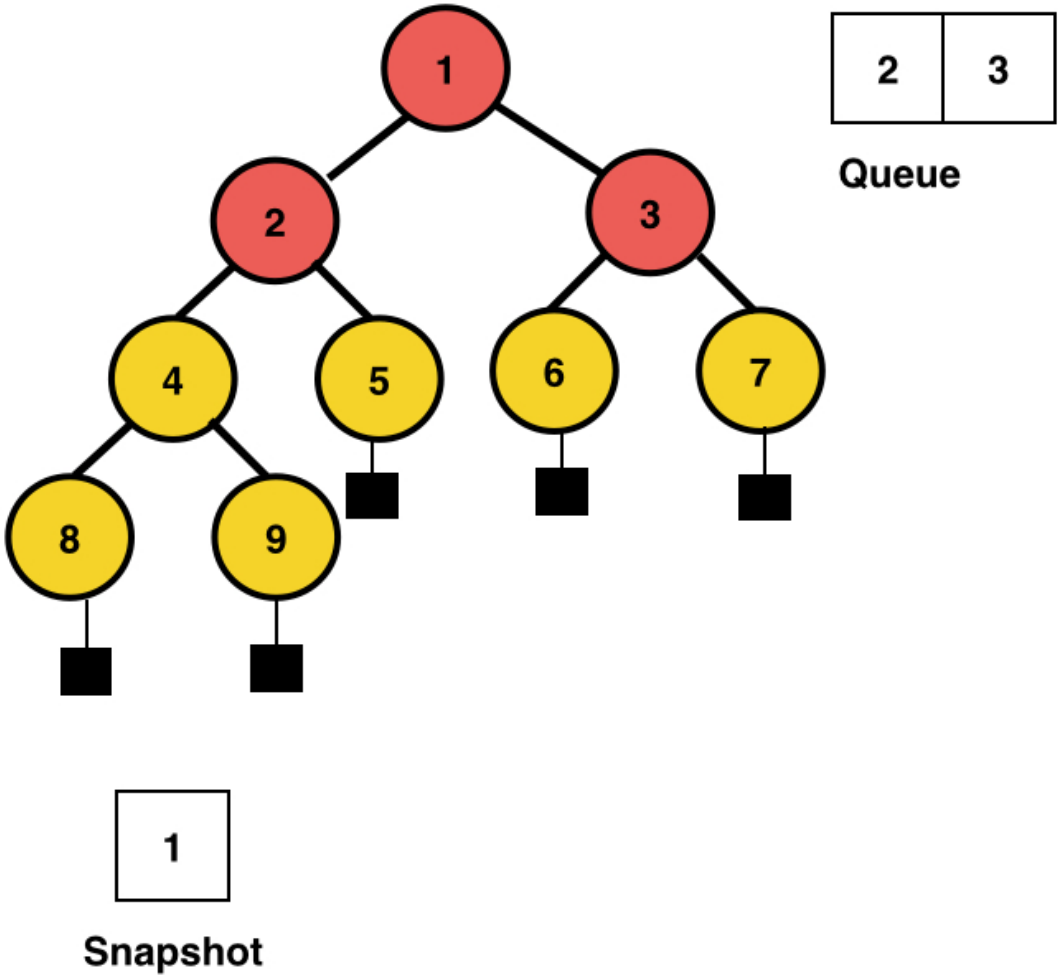
Level-OrderTraversal

- Need to traverse from the top level to the next level from left to right in each level
- Is there a recursive strategy for this?
- Can we use a queue?

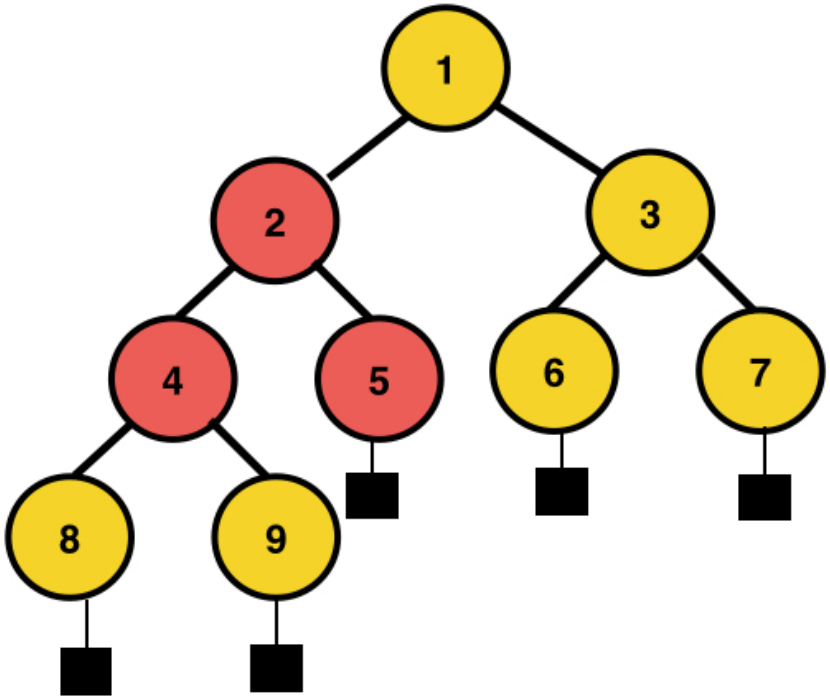
Tree Traversal: Level-order



Tree Traversal: Level-order



Tree Traversal: Level-order



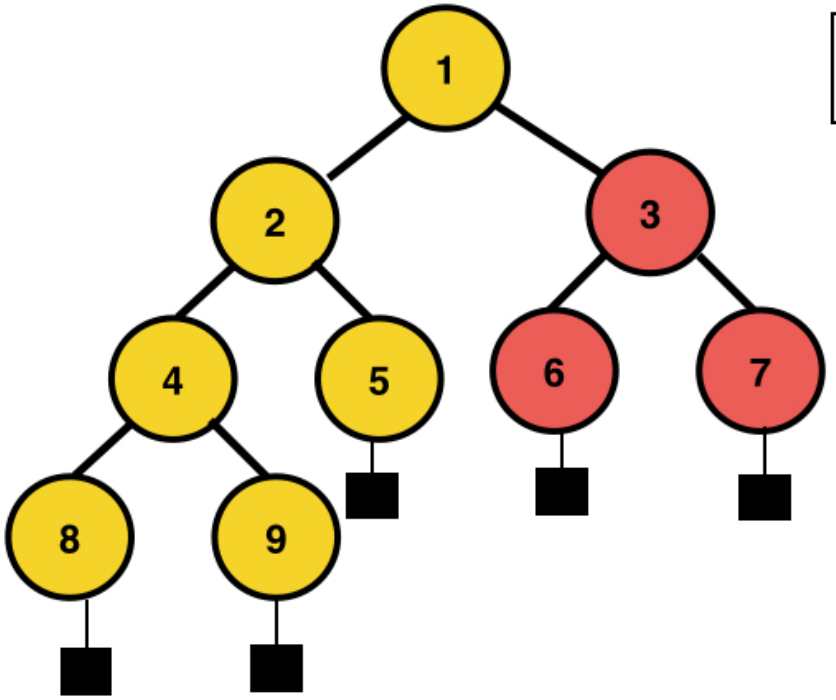
3	4	5

Queue

1	2

Snapshot

Tree Traversal: Level-order



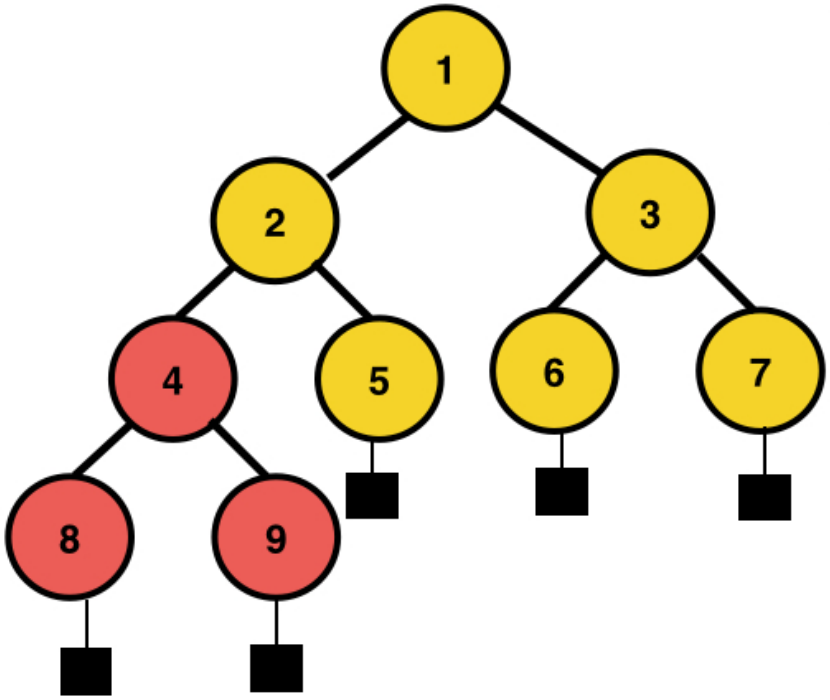
4	5	6	7

Queue

1	2	3

Snapshot

Tree Traversal: Level-order



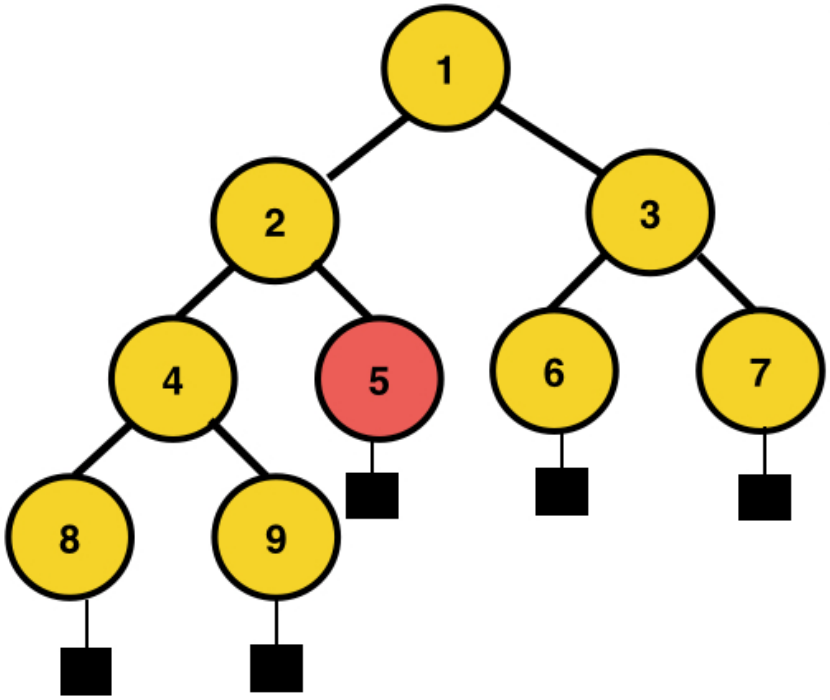
5	6	7	8	9

Queue

1	2	3	4

Snapshot

Tree Traversal: Level-order



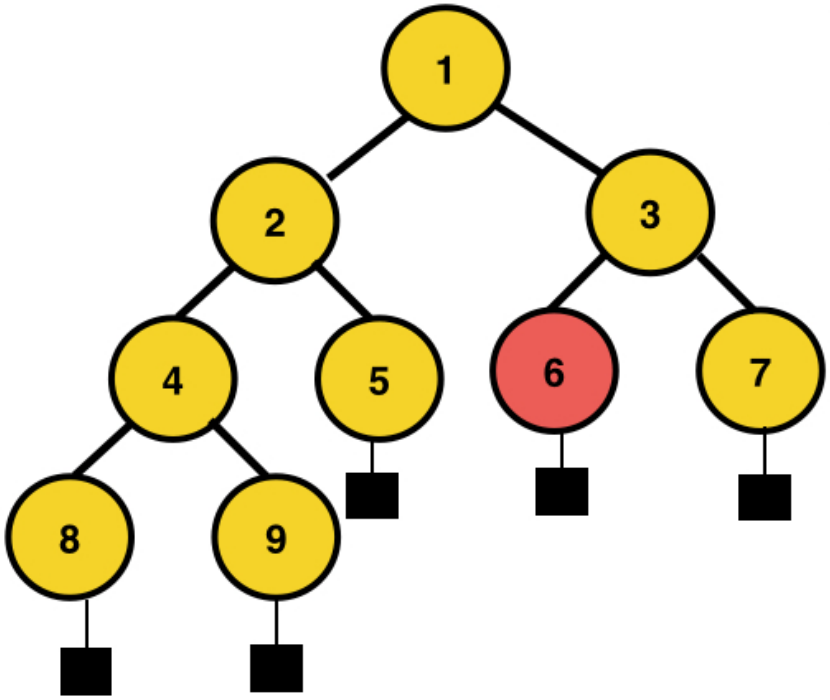
6	7	8	9

Queue

1	2	3	4	5

Snapshot

Tree Traversal: Level-order



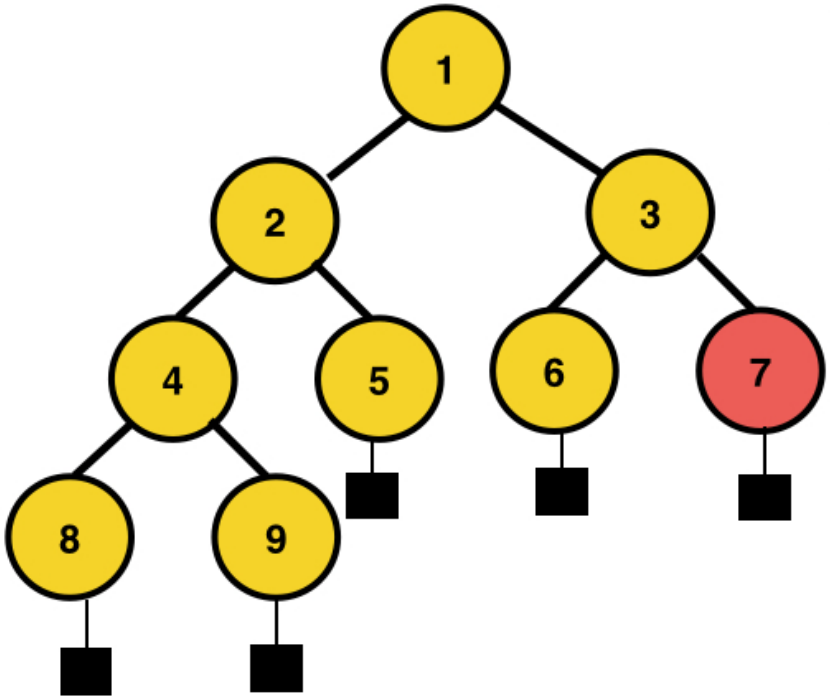
7	8	9

Queue

1	2	3	4	5	6

Snapshot

Tree Traversal: Level-order



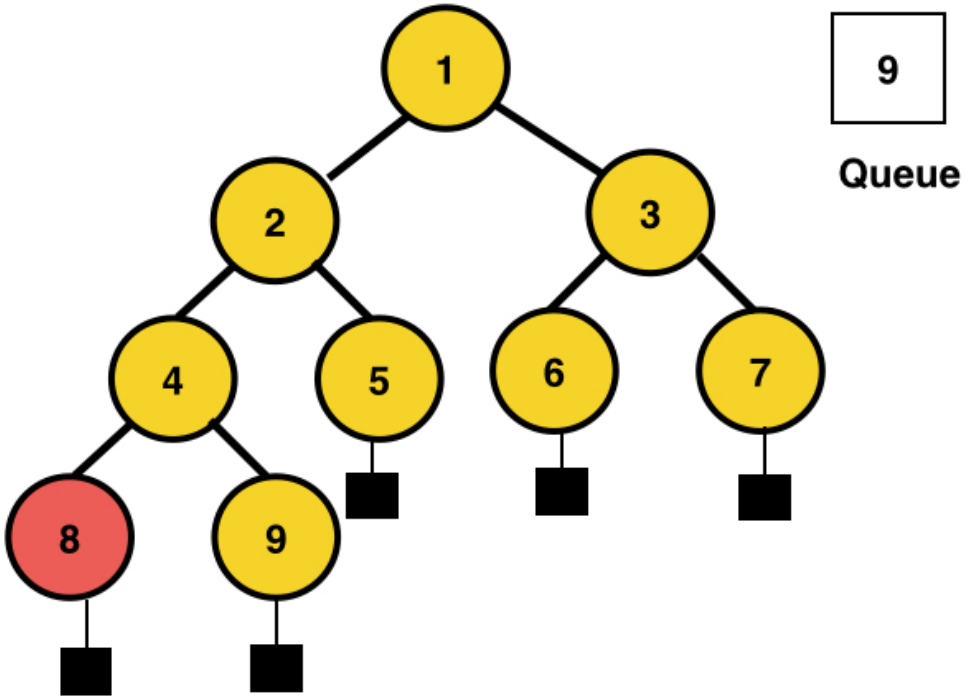
8	9

Queue

1	2	3	4	5	6	7

Snapshot

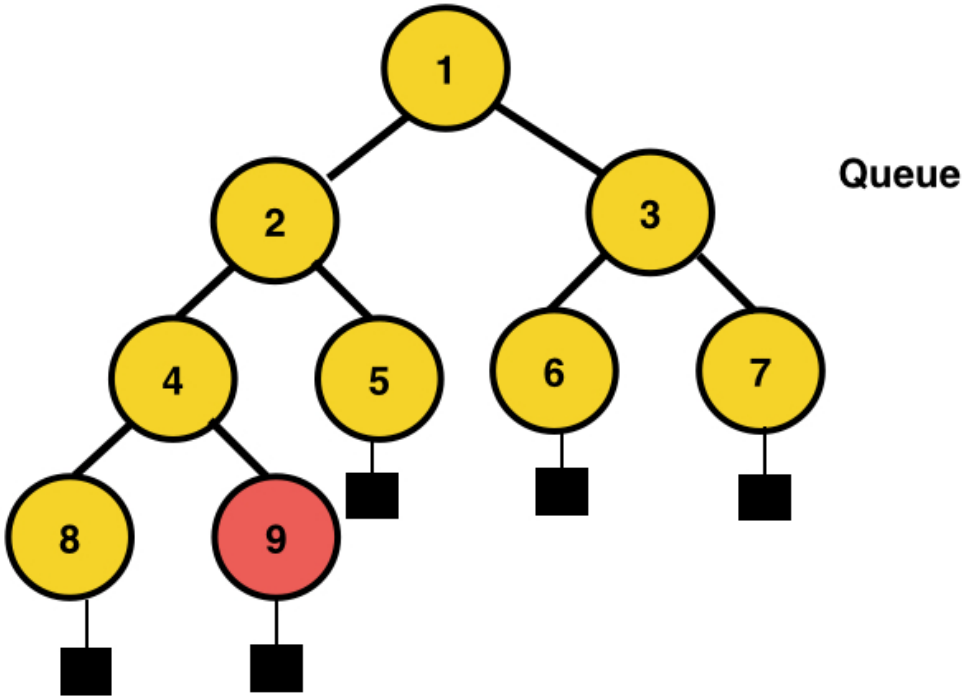
Tree Traversal: Level-order



1	2	3	4	5	6	7	8

Snapshot

Tree Traversal: Level-order



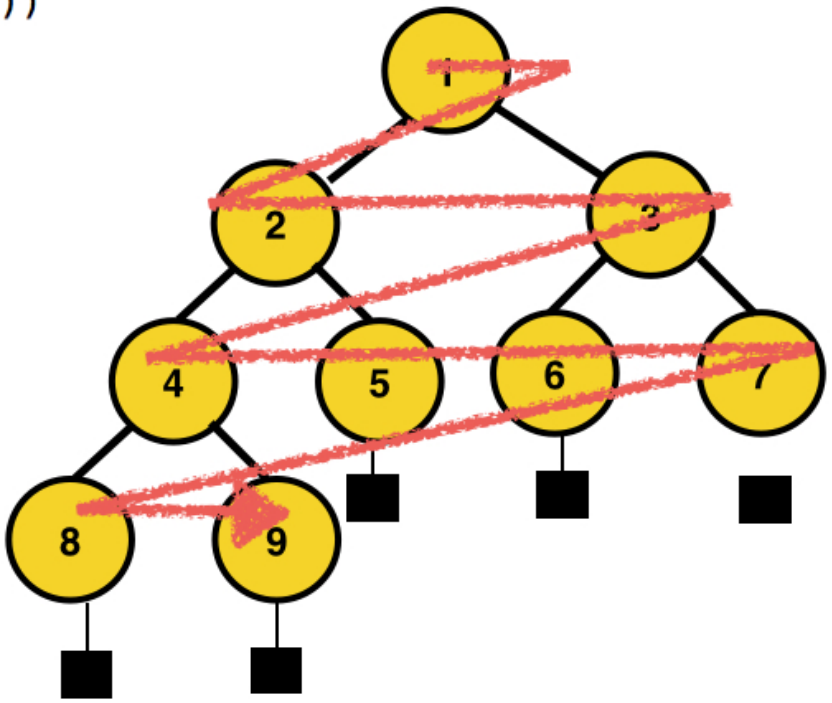
1	2	3	4	5	6	7	8	9

Snapshot

Tree Traversal: Level-order

```
public abstract class AbstractTree<E> implements Tree<E>{
  ...
  public Iterable<Position<E>> positions() {return inorder();}

  public Iterable<Position<E>> inorder(){
    List<Position<E>> snapshot = new ArrayList<Position<E>>();
    Position<E> p;
    if(!isEmpty()){
      LinkedListQueue<Position<E>> llQ = new LinkedListQueue();
      llQ.enqueue(root());
      while(!llQ.isEmpty()){
        p=llQ.dequeue();
        for(Position<E> c : children(p))
          llQ.enqueue(c);
        snapshot.add(p);
      }
    }
    return snapshot;
  }
}
```

1) If the tree is not empty, root is enqueued into the queue first.
2) Then, the queue is dequeued. This node is added into the snapshot.
3) The children of this node are enqueued.
4) Repeat until the queue is empty



Insertion and Deletion of Elements from Binary Tree

- So far we have investigated how nodes can be traversed or accessed for the sequential and linked implementations
- How are nodes **inserted** or **deleted** from binary trees?
- This depends on the **application** and the **type** of tree!
- We will look at four types of trees: **heaps**, **binary search tree**, **AVL tree**, **red-black tree**

Modifying Operations on Heaps

Heaps

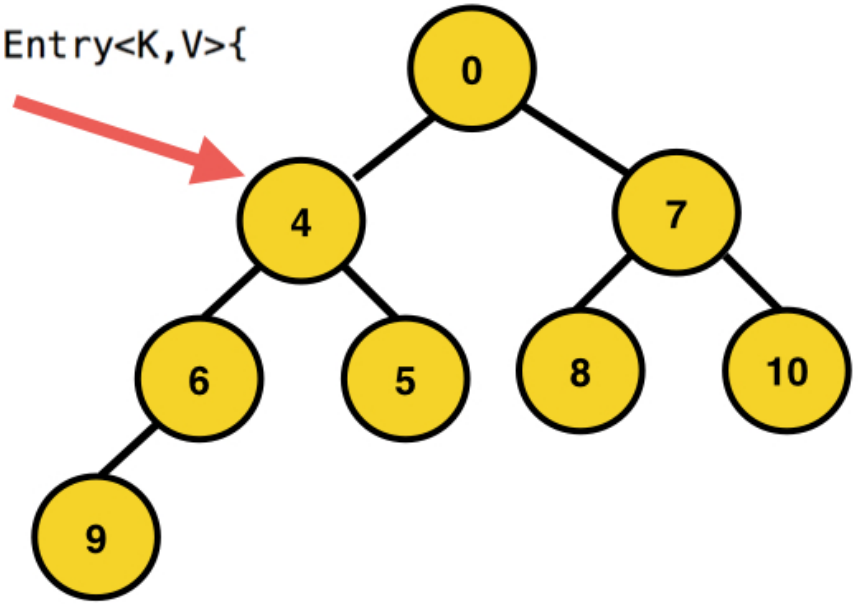
Modifying Operations on Heaps

- **Heaps** are a special type of binary tree
- Items are inserted and deleted from the tree so that a **particular property** is always heeded by key elements in each node of the tree
- The key element in the parent node is always **smaller** than that of its descendants (e.g. `int`, `char`)
- A delete operation on the heap removes the node containing the smallest element (i.e. the root)
- Items are inserted/deleted into a heap so that it is always **complete**
- A sequential implementation is used for illustration purposes

Modifying Operations on Heaps

An Example of a Heap

```
public interface Entry<K,V>{
  K getKey();
  V getValue();
}
```




```
public interface PriorityQueue<K,V>{
  int size();
  boolean isEmpty();
  Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
  Entry<K,V> min();
  Entry<K,V> removeMin();
}
```

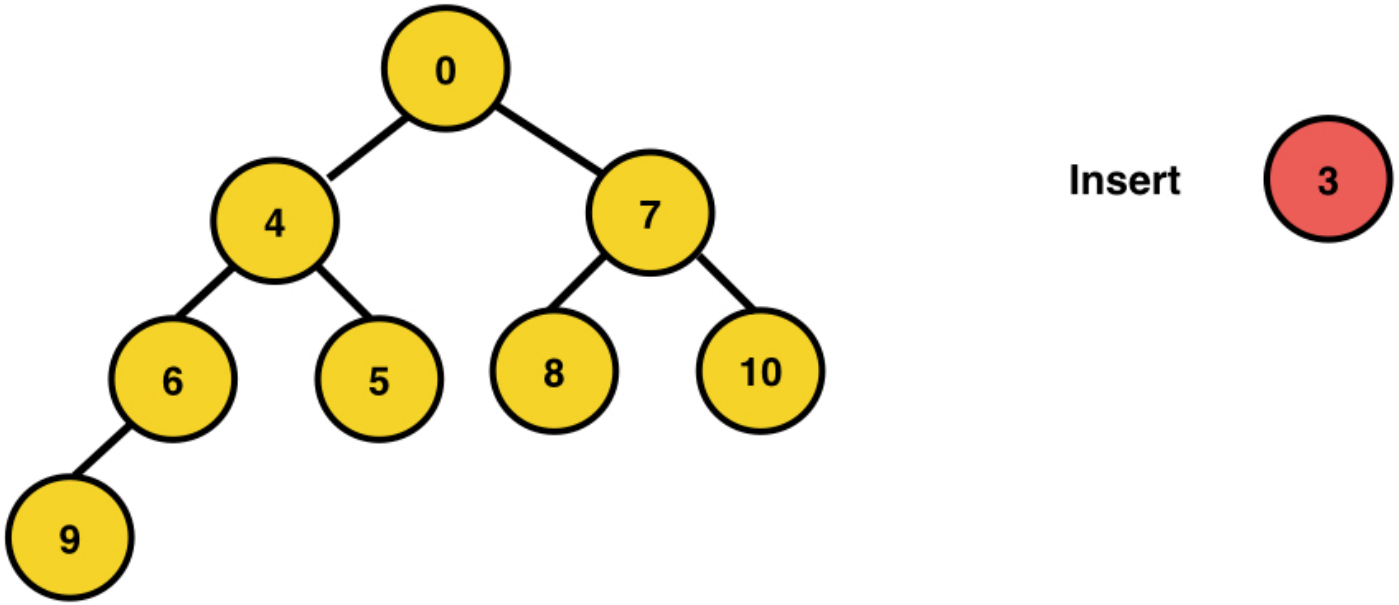
Modifying Operations on Heaps

Insertion into a Heap

Modifying Operations on Heaps: Insertion

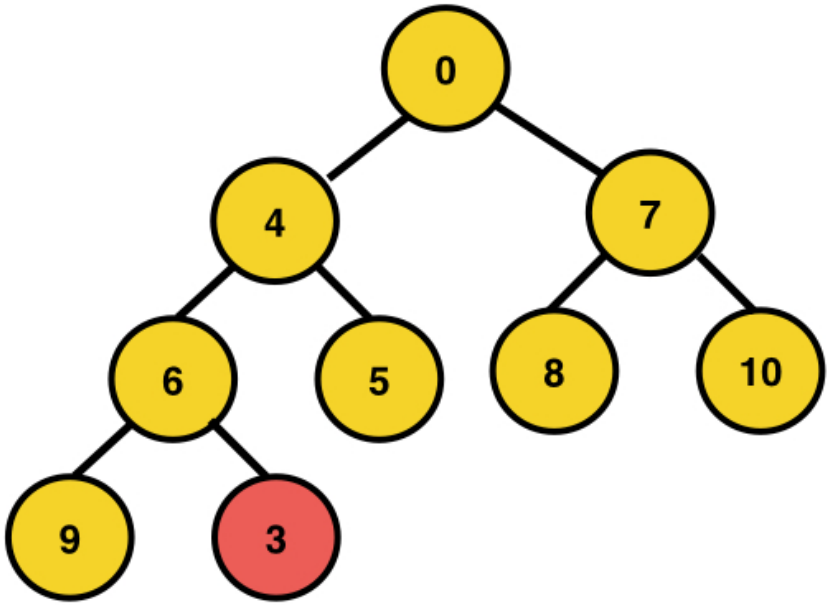
- New node to be inserted is placed at the next available spot on the left in the last level of the heap
- This node is **bubbled up** and exchanged with parent nodes until no more exchanges are possible

Modifying Operations on Heaps: Insertion



1) Need to maintain the heap property
2) Need to maintain the completeness property

Modifying Operations on Heaps: Insertion

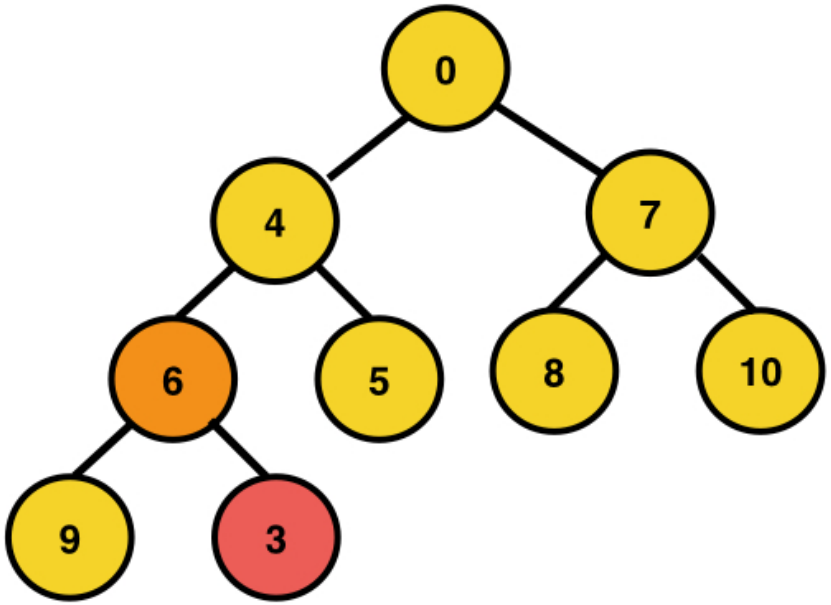


Inserting at the end of the array will maintain the completeness property but not necessarily the heap property

heap:

0	4	7	6	5	8	10	9	11	3
0	1	2	3	4	5	6	7	8	9

Modifying Operations on Heaps: Insertion

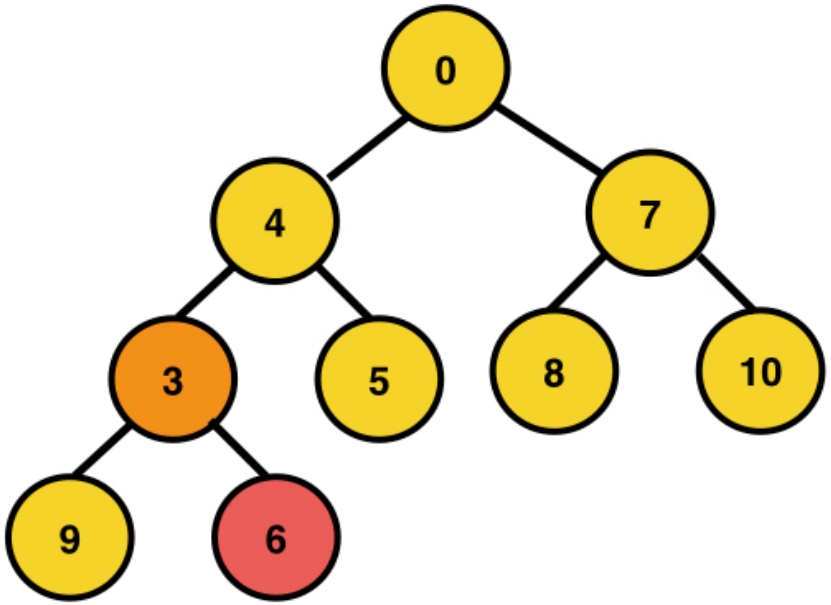


- Need to apply "upheap" to restore the heap property
- Compare the parent (j-1)/2 and see if this node meets the heap property

heap:

0	4	7	6	5	8	10	9	3
0	1	2	3	4	5	6	7	8

Modifying Operations on Heaps: Insertion

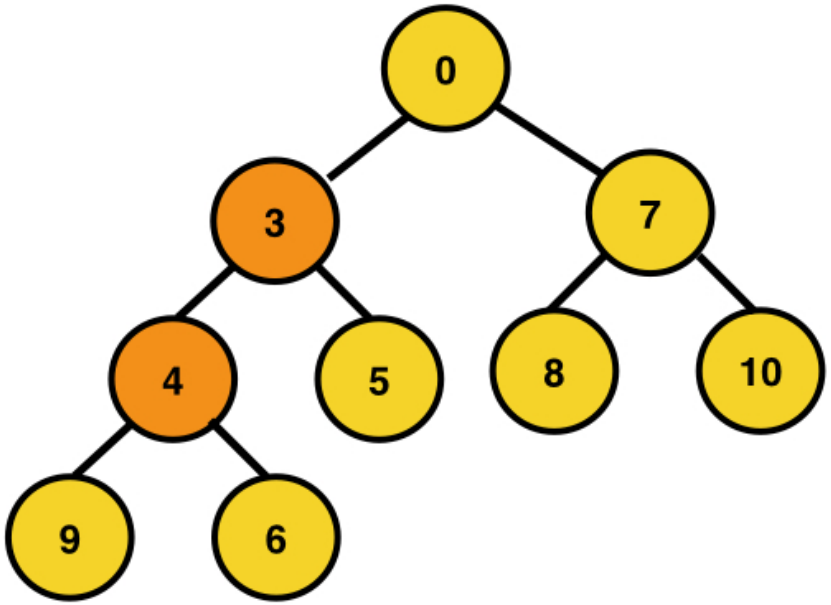


- If heap property doesn't hold then, swap the parent and child

heap:

0	4	7	3	5	8	10	9	6
0	1	2	3	4	5	6	7	8

Modifying Operations on Heaps: Insertion

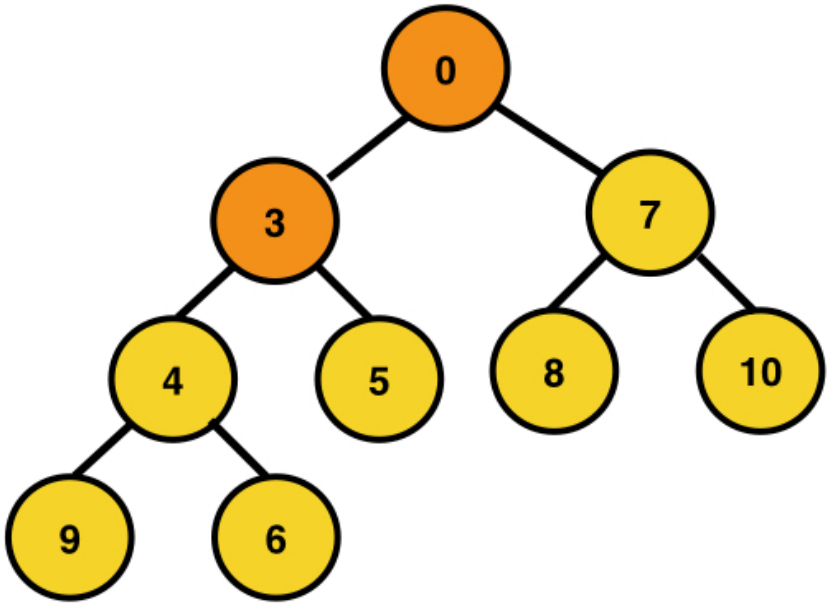


- Repeat the same up the tree, until the root is reached

heap:

0	1	2	3	4	5	6	7	8
0	3	7	4	5	8	10	9	6

Modifying Operations on Heaps: Insertion



- Repeat the same up the tree, until the root is reached

heap:

0	1	2	3	4	5	6	7	8
0	3	7	4	5	8	10	9	6

Modifying Operations on Heaps: Insertion

```
public class DefaultComparator<E> implements Comparator<E>{
  public int compare(E a, E b) throws ClassCastException{
    return ((Comparable<E>) a).compareTo(b);
  }
}
```

In the heap class:

```
private Comparator<K> comp=new DefaultComparator<K>();
protected int compare(Entry<K,V> a, Entry<K,V> b){
  return comp.compare(a.getKey(), b.getKey());
}
protected boolean checkKey(K key) throws IllegalArgumentException{
  try{
    return(comp.compare(key,key)==0);
  }catch(ClassCastException e){
    throw new IllegalArgumentException("Incompatible Key.");
  }
}
```

- Comparator interface that allows for the comparison of two keys

```
protected void upheap(int j){
  while(j>0){
    int p = parent(j);
    if(compare(heap.get(j), heap.get(p))>=0) break;
    swap(j,p);
    j=p;
  }
}
```

Modifying Operations on Heaps: Insertion

- What special properties of the heap have been maintained after insertion?
- The tree is **complete** and every node contains **smallest** data element in comparison to descendants
- What is the complexity of the insertion operation?
- Since a node is always inserted into a complete tree, the complexity is O(logn)

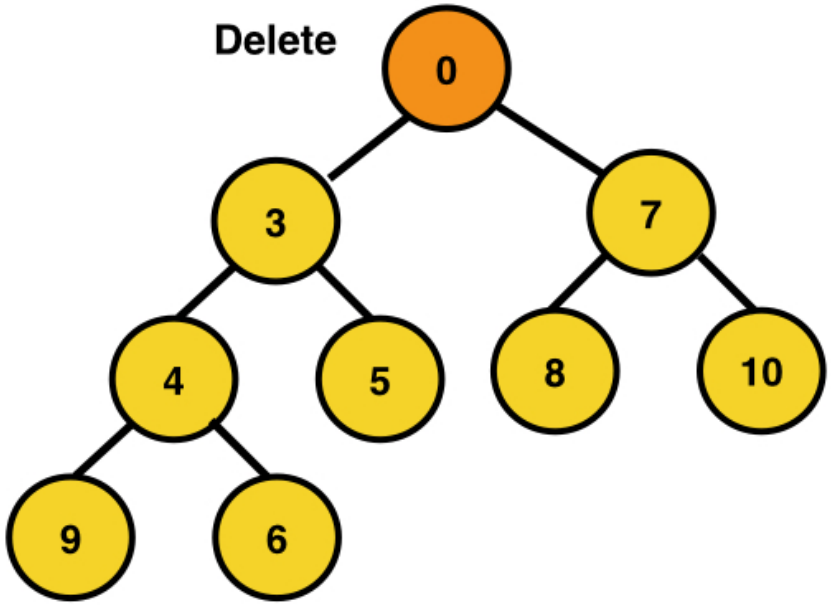
Modifying Operations on Heaps

Deletion from a Heap

Modifying Operations on Heaps: RemoveMin

- Node at the root is removed and replaced with the left-most leaf at the bottom-most level of the heap
- This node is then **bubbled down** and exchanged with child nodes until no more exchanges are possible

Modifying Operations on Heaps: RemoveMin

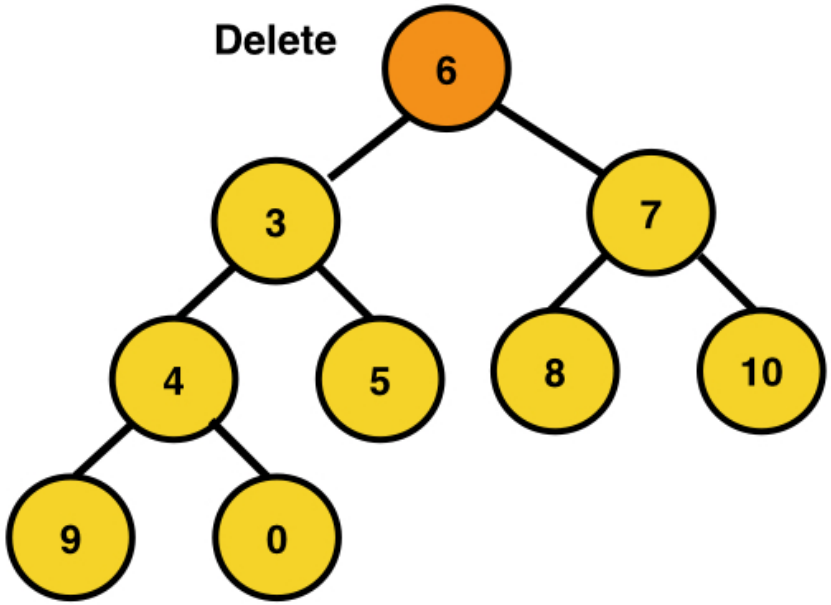


1) Remove the root which is the minimum key
2) Need to maintain the heap property
3) Need to maintain the completeness property

heap:

0	1	2	3	4	5	6	7	8
0	3	7	4	5	8	10	9	6

Modifying Operations on Heaps: RemoveMin

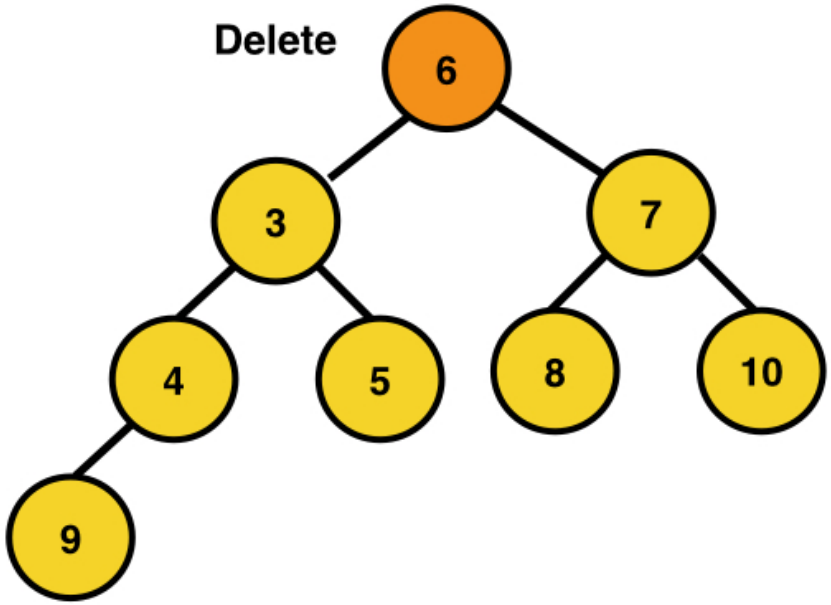


1) Remove the root which is the minimum key
2) Replace root with last element in the heap

heap:

0	1	2	3	4	5	6	7	8
6	3	7	4	5	8	10	9	0

Modifying Operations on Heaps: RemoveMin

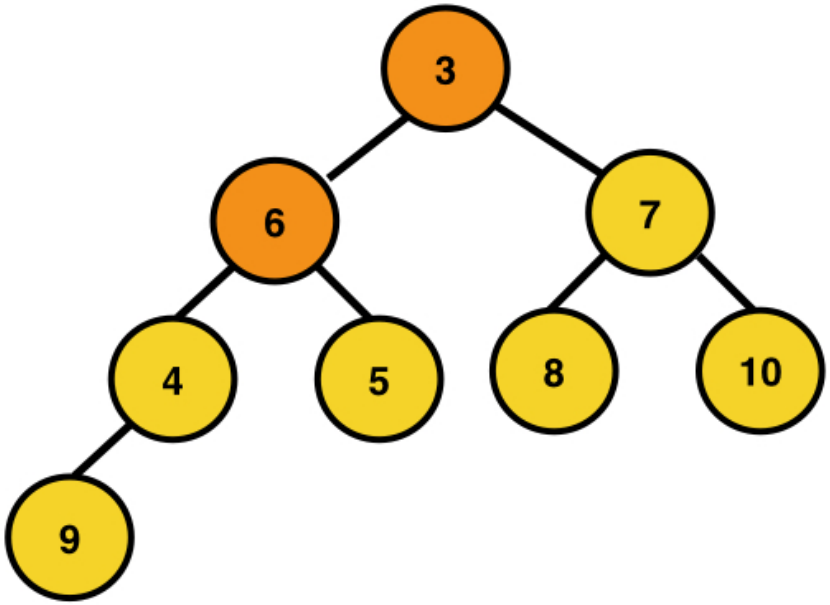


1) Remove the last element
2) To restore heap property, apply "downheap"
3) Swap the parent with the smaller of the two children

heap:

0	1	2	3	4	5	6	7
6	3	7	4	5	8	10	9

Modifying Operations on Heaps: RemoveMin

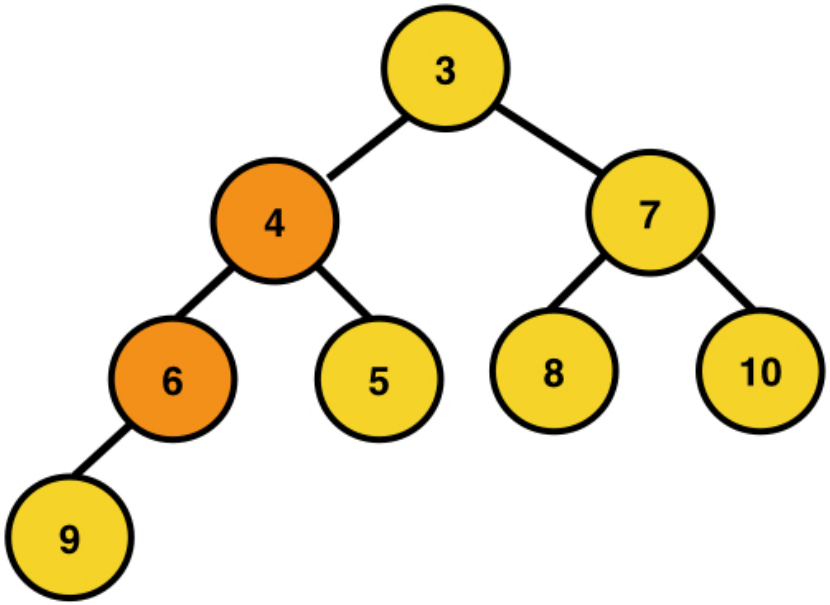


1) Remove the last element
2) To restore heap property, apply "downheap"
3) Swap the parent with the smaller of the two children

heap:

0	1	2	3	4	5	6	7
3	6	7	4	5	8	10	9

Modifying Operations on Heaps: RemoveMin

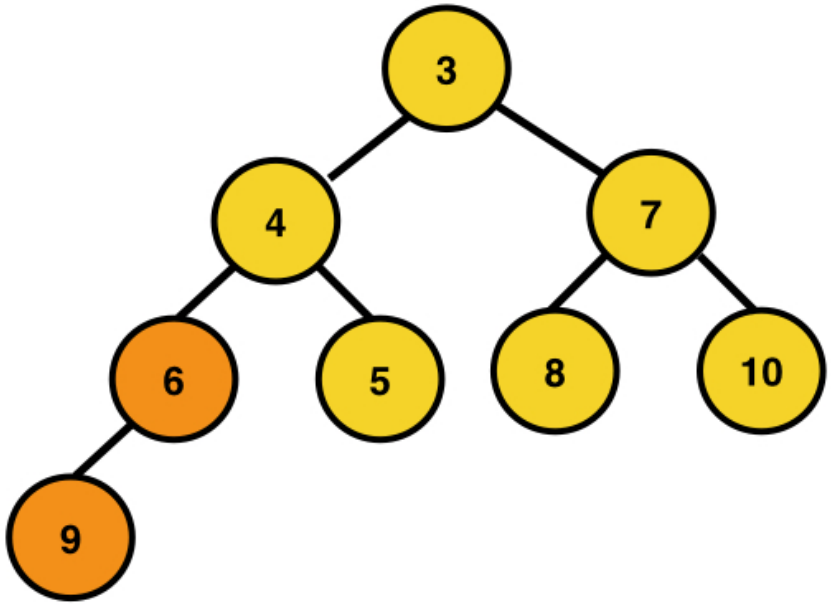


1) Remove the last element
2) To restore heap property, apply "downheap"
3) Swap the parent with the smaller of the two children

heap:

0	1	2	3	4	5	6	7
3	4	7	6	5	8	10	9

Modifying Operations on Heaps: RemoveMin



1) Remove the last element
2) To restore heap property, apply "downheap"
3) Swap the parent with the smaller of the two children

heap:

0	1	2	3	4	5	6	7
3	6	7	4	5	8	10	9

Modifying Operations on Heaps: RemoveMin

```
protected void downheap(int j){
  while(hasLeft(j)){
    int leftIndex=left(j);
    int smallChildIndex=leftIndex;
    if(hasRight(j)){
      int rightIndex = right(j);
      if(compare(heap.get(leftIndex),heap.get(rightIndex))>0)
        smallChildIndex=rightIndex;
    }
    if(compare(heap.get(smallChildIndex),heap.get(j))>=0)
      break;
    swap(j,smallChildIndex);
    j=smallChildIndex;
  }
}
```

Modifying Operations on Heaps: RemoveMin

- Deletion also maintains two special properties of heaps
- The tree is still complete and parent nodes contain smallest value of all descendant nodes
- Complexity of deletion is the number of levels in the heap (bubbling down operation): O(log n)

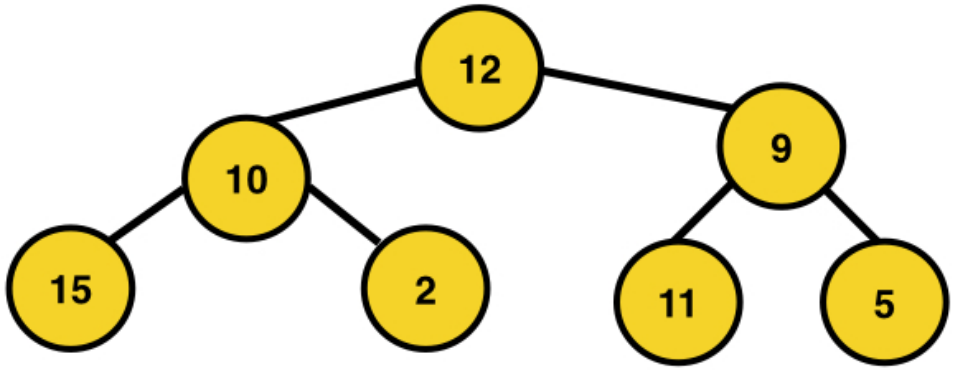
Modifying Operations on Heaps: Heapifying

Heapifying a Complete Tree

Modifying Operations on Heaps: Heapifying

- Suppose that you are provided with a **complete** tree
- However, this tree contains elements with data that do not follow the **heap** property (i.e. elements in the tree have no order)
- Is it possible to reorganize this tree so that it is a heap?
- Yes! Can use the `downheap` function for this!

Modifying Operations on Heaps: Heapifying

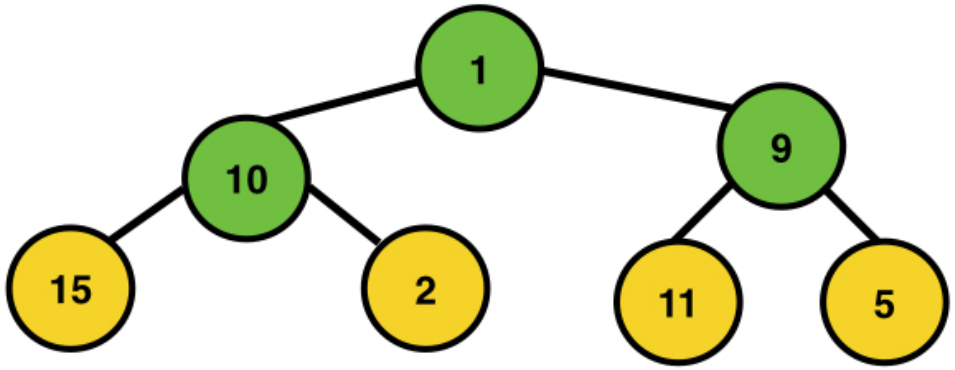


Converting a complete binary tree into a heap
1) Find out which nodes are not leaves and start from leftmost node
2) Apply "downheap" on that node and repeat until root is reached

heap:

1	10	9	15	2	11	5
0	1	2	3	4	5	6

Modifying Operations on Heaps: Heapifying

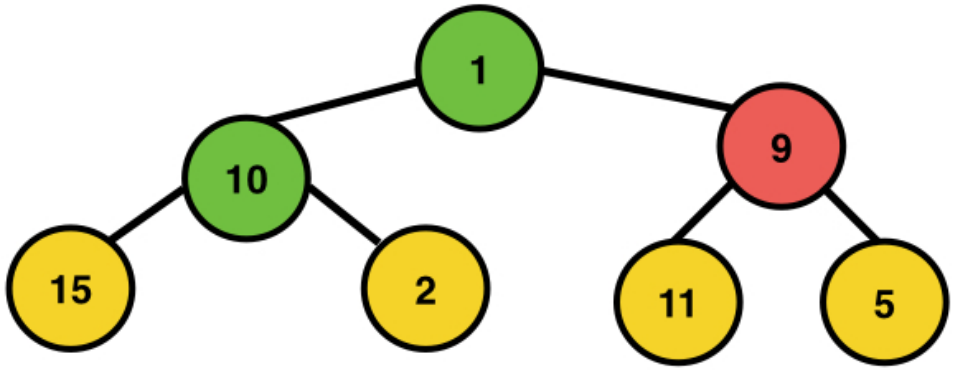


Converting a complete binary tree into a heap
1) Find out which nodes are not leaves and start from leftmost node
2) Apply "downheap" on that node and repeat until root is reached

heap:

0	1	2	3	4	5	6
1	10	9	15	2	11	5

Modifying Operations on Heaps: Heapifying

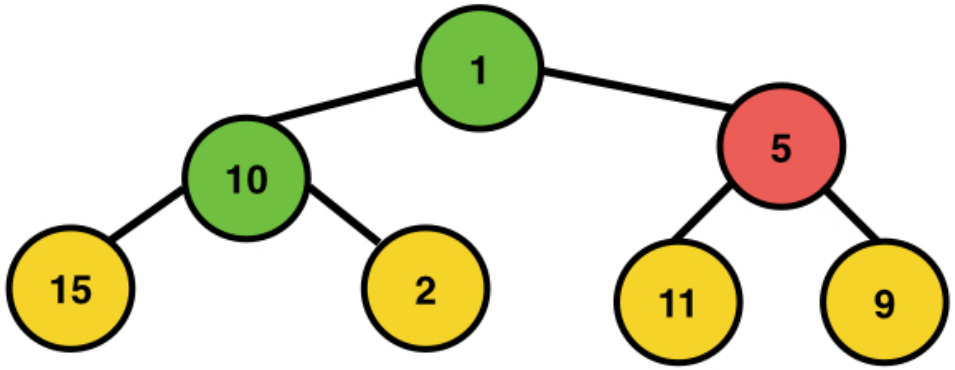


Converting a complete binary tree into a heap
1) Find out which nodes are not leaves and start from leftmost node
2) Apply "downheap" on that node and repeat until root is reached

heap:

0	1	2	3	4	5	6
1	10	9	15	2	11	5

Modifying Operations on Heaps: Heapifying

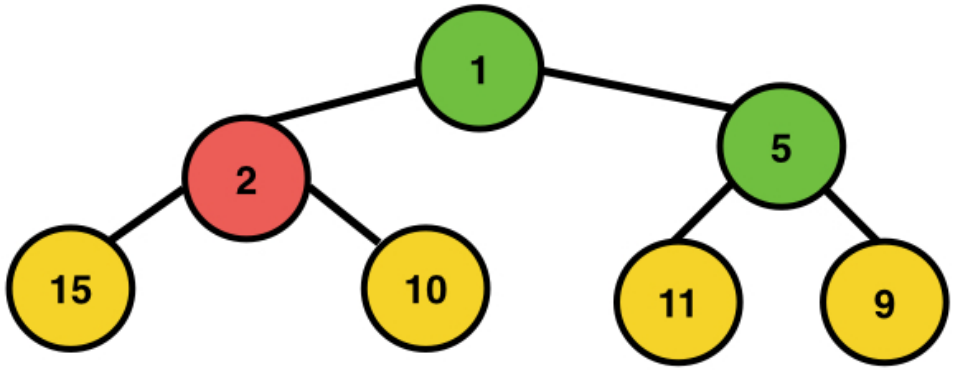


Converting a complete binary tree into a heap
1) Find out which nodes are not leaves and start from leftmost node
2) Apply "downheap" on that node and repeat until root is reached

heap:

0	1	2	3	4	5	6
1	10	5	15	2	11	9

Modifying Operations on Heaps: Heapifying

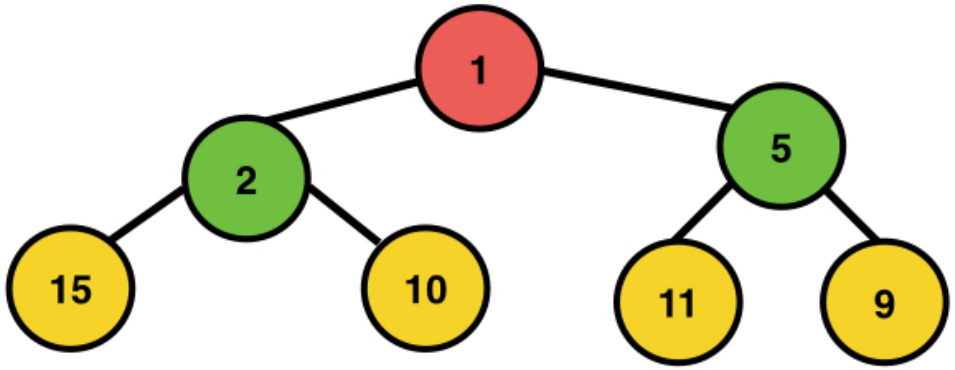


Converting a complete binary tree into a heap
1) Find out which nodes are not leaves and start from leftmost node
2) Apply "downheap" on that node and repeat on the next left node until root is reached

heap:

0	1	2	3	4	5	6
1	2	5	15	10	11	9

Modifying Operations on Heaps: Heapifying

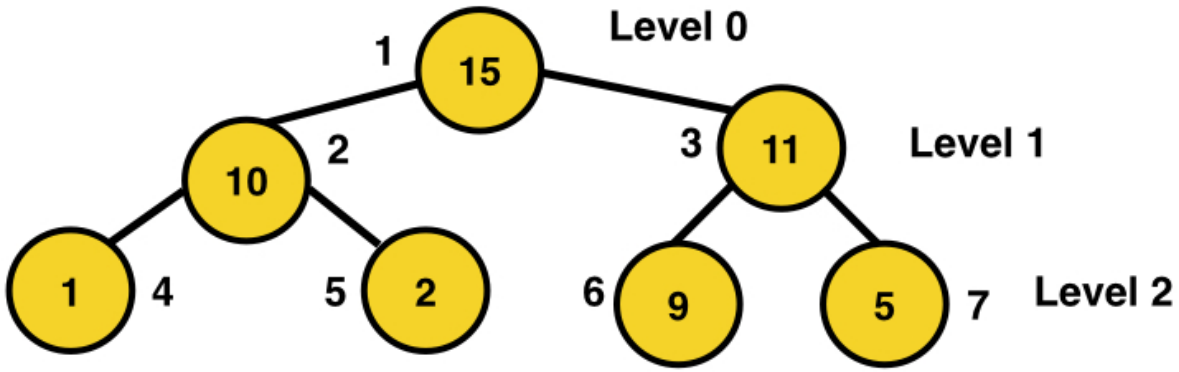


Converting a complete binary tree into a heap
1) Find out which nodes are not leaves and start from leftmost node
2) Apply "downheap" on that node and repeat on the next left node until root is reached

heap:

0	1	2	3	4	5	6
1	2	5	15	10	11	9

Modifying Operations on Heaps: Heapifying



- What is the time complexity of the `heapify` operation?
- Assume that the number of levels in the tree is l
- Number of exchanges in `reHeapify` starting at node at level i is $l - i$ in the worst case
- Number of nodes at level i is 2^i
- `reHeapify` operation is performed on all non-leaf nodes (i.e. worst case all nodes in $l - 1$ levels)
- Cost is $S = \sum_{i=0}^{l-1}(l - i)2^i$

Modifying Operations on Heaps: Heapifying

Time Complexity for Heapifying a Complete Tree

$$S = \sum_{i=0}^{l-1} (l-i)2^i \text{: Let } j = l - i$$

$$S = \sum_{j=1}^{l} j2^{(l-j)} = 2^l \sum_{j=1}^{l} j2^{-j} = 2^l \sum_{j=1}^{l} \frac{j}{2^j}$$

Can show that $\sum_{j=1}^{l} \frac{j}{2^j} < 2$

$$S < 2^{(l+1)}$$

It can be verified that $2^l \leq n \leq 2^{l+1} - 1$. Since $2^l < n$, $2^{(l+1)} < 2n$. This implies that $S < 2n$. Hence O(S) is O(n)!

Modifying Operations on Heaps: Heapifying

Comparison with other Sorting Algorithms

- Complexity of sorting elements with bubble sort: O(n^2)
- Complexity of heapifying a tree: O(n)
- Complexity of retrieving/removing the largest value from array sorted by bubble sort: O(1)
- Complexity of retrieving/removing a key from a heapified tree: O(logn)
- Complexity of inserting an element into an array sorted by bubble sort: O(n)
- Complexity of inserting a key into a heapified tree: O(logn)

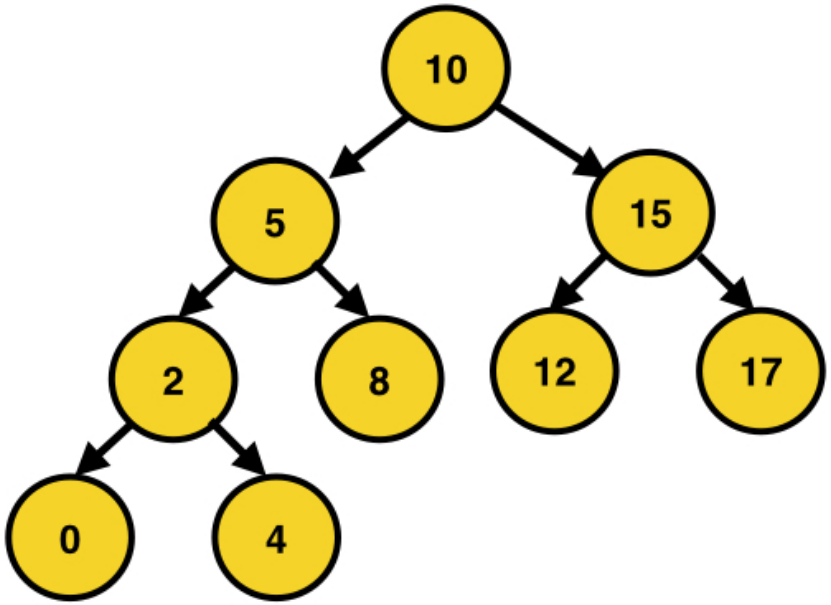
Modifying Operations on BST

Binary Search Tree

What are Binary Search Trees?

- A Binary Search Tree (BST) is a special type of tree that has several **ordering** constraints on all nodes in the tree
- The **left subtree** of a node in a BST will have data values that are **strictly lesser** than the node's key
- The **right subtree** of a node in a BST will have data values that are **strictly greater** than the node's key
- **No duplicate** nodes will be present in a BST
- No restriction on the structure of the BST
- Typically, BSTs have a **linked** implementation
- The purpose of a BST is to **search** for data and the typical operations on it are `search` or `insert`

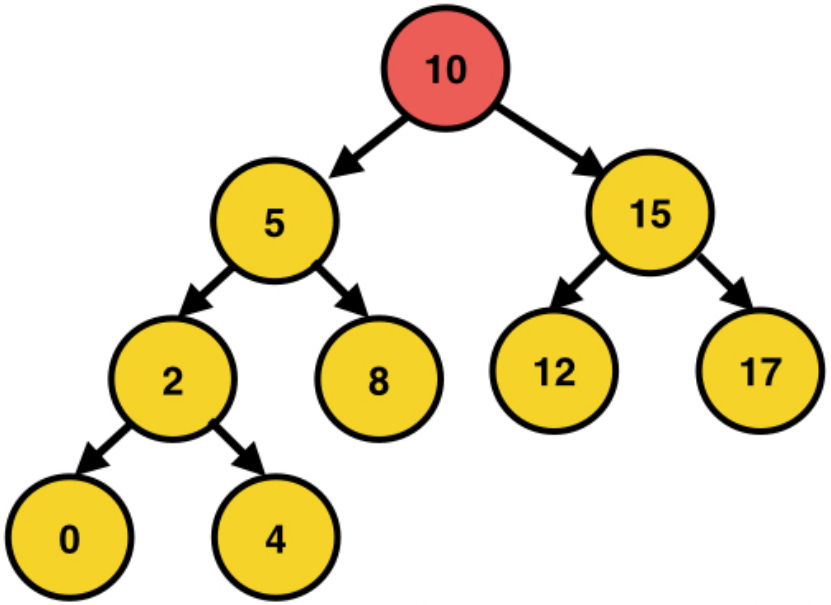
Example of a BST



Operations on a BST

Search a BST

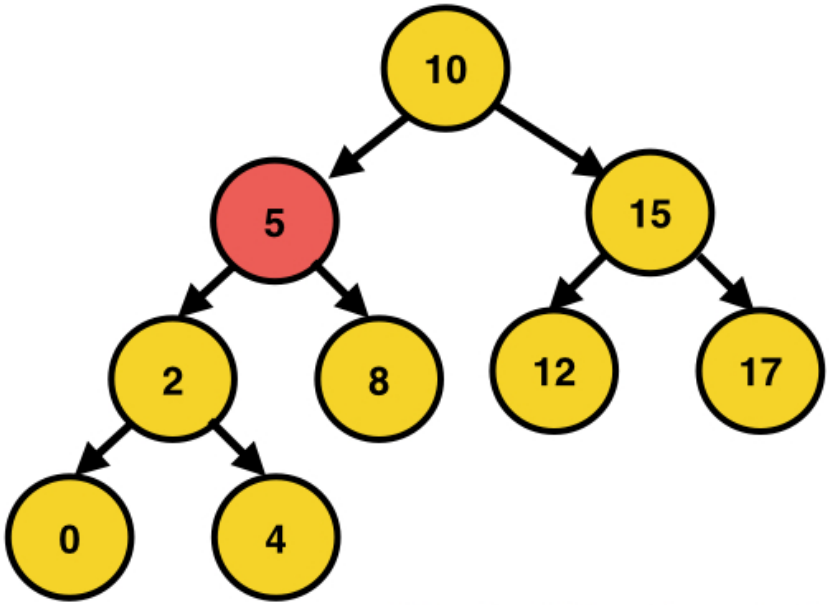
Operations on a BST: Search



Search for 4

```
public Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key)
{
  if(isExternal(p))
    return p;
  int comp = compare(key, p.getKey());
  if(comp==0){
    return p;
  }
  else if(comp<0){
    return treeSearch(left(p),key);
  }
  else{
    return treeSearch(right(p),key);
  }
}
```

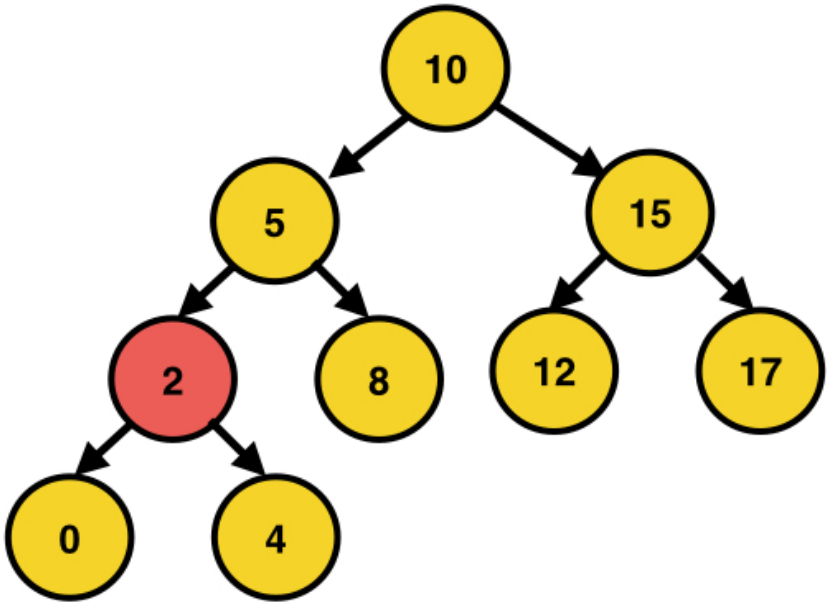
Operations on a BST: Search



Search for 4

```
public Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key)
{
  if(isExternal(p))
    return p;
  int comp = compare(key, p.getKey());
  if(comp==0){
    return p;
  }
  else if(comp<0){
    return treeSearch(left(p),key);
  }
  else{
    return treeSearch(right(p),key);
  }
}
```

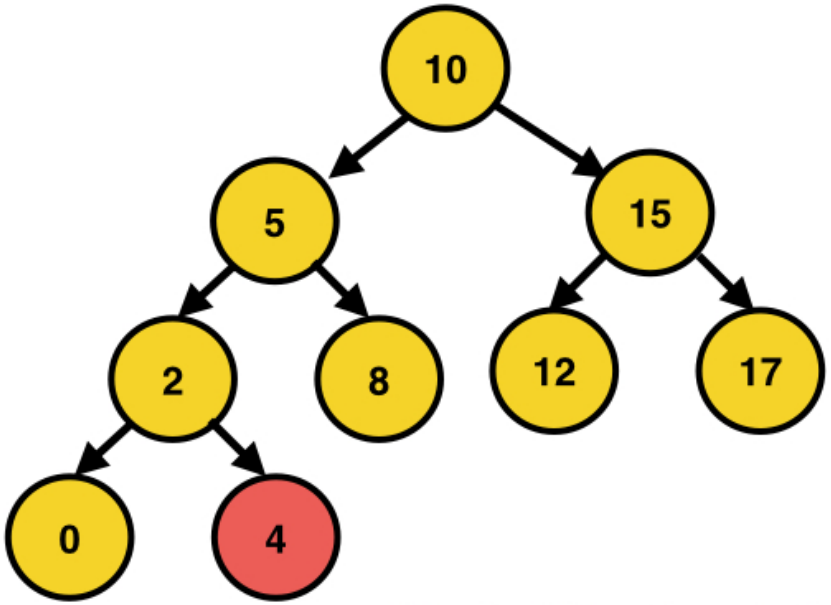
Operations on a BST: Search



Search for 4

```
public Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key)
{
  if(isExternal(p))
    return p;
  int comp = compare(key, p.getKey());
  if(comp==0){
    return p;
  }
  else if(comp<0){
    return treeSearch(left(p),key);
  }
  else{
    return treeSearch(right(p),key);
  }
}
```

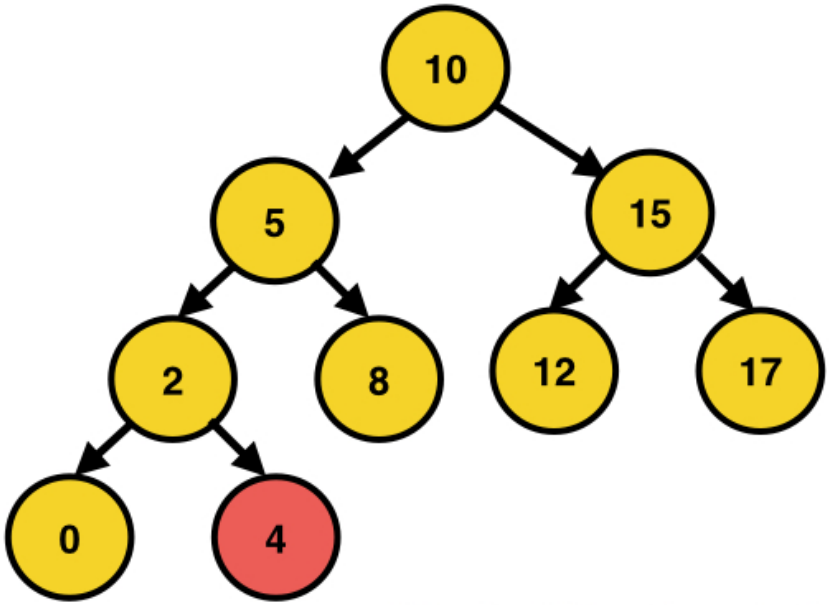
Operations on a BST: Search



Search for 4

```
public Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key)
{
  if(isExternal(p))
    return p;
  int comp = compare(key, p.getKey());
  if(comp==0){
    return p;
  }
  else if(comp<0){
    return treeSearch(left(p),key);
  }
  else{
    return treeSearch(right(p),key);
  }
}
```

Operations on a BST: Search



Search for 4

```
public Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key)
{
  if(isExternal(p))
    return p;
  int comp = compare(key, p.getKey());
  if(comp==0){
    return p;
  }
  else if(comp<0){
    return treeSearch(left(p),key);
  }
  else{
    return treeSearch(right(p),key);
  }
}
```

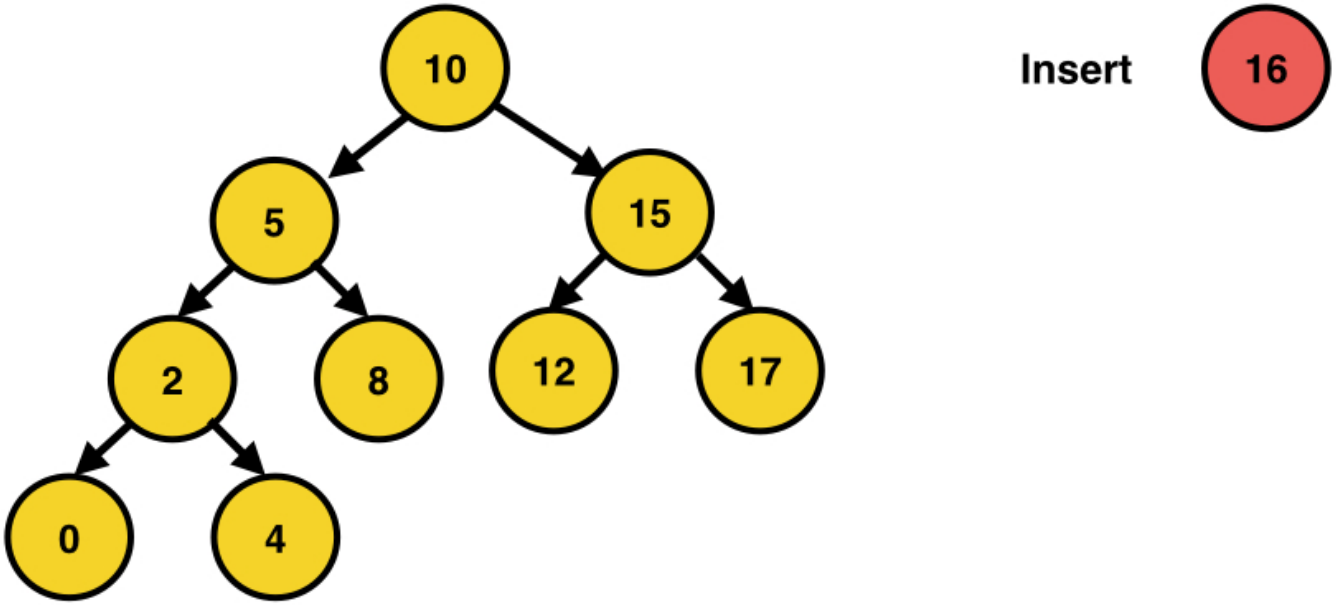
Operations on a BST: Search

- Complexity of search in a BST is the longest path in the tree
- Can we establish a firm statement about the longest path in the tree?

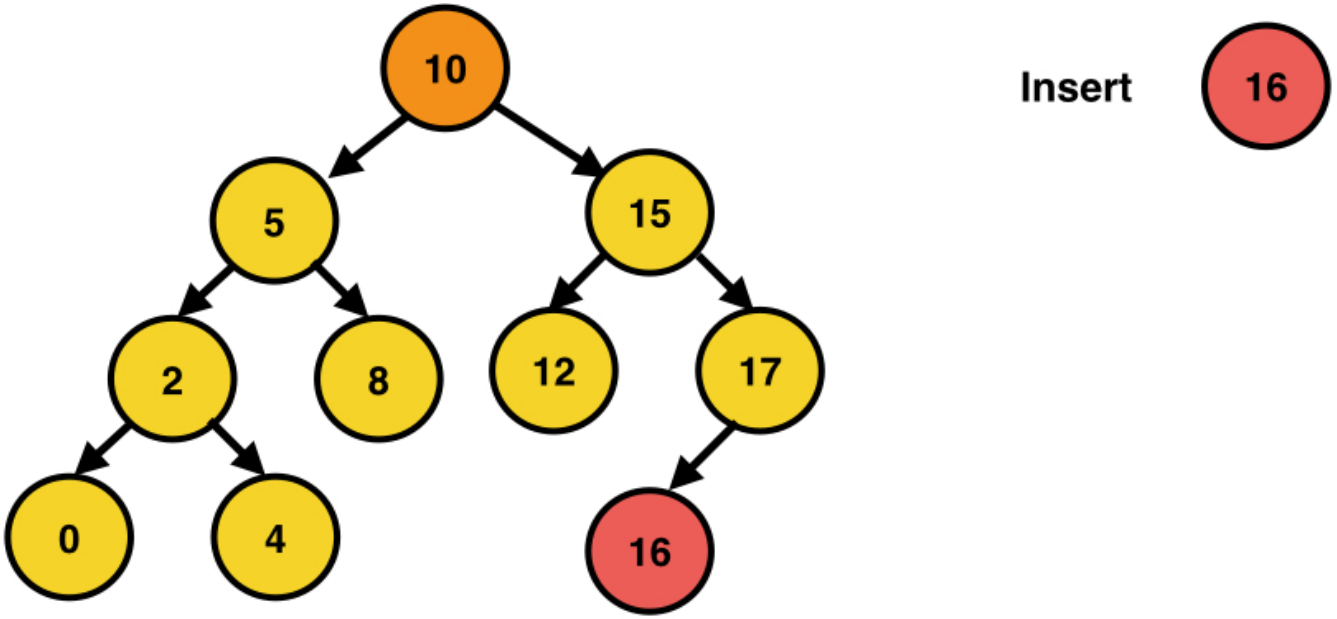
Operations on a BST

Insertion into a BST

Operations on a BST: Insertion



Operations on a BST: Insertion

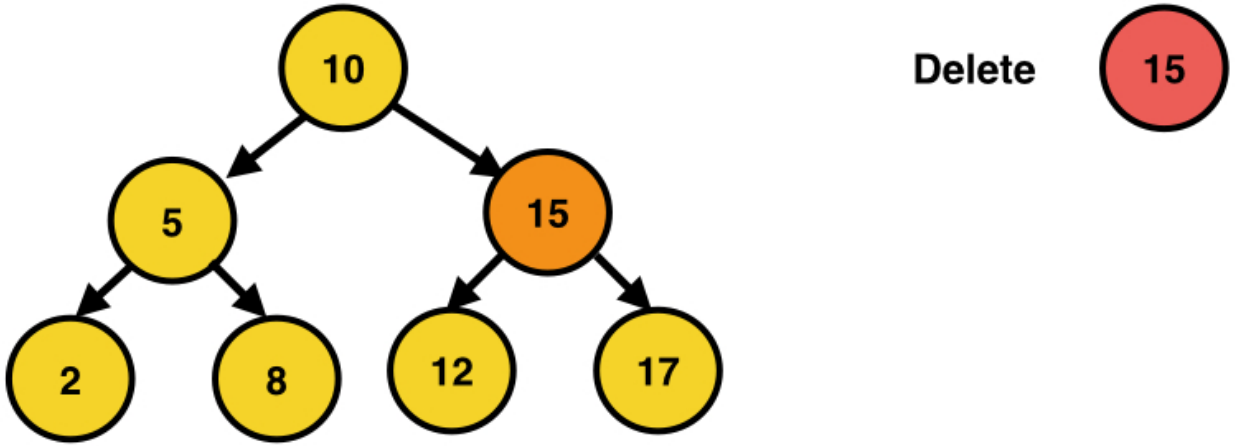



```
public V put(K key, V value) throws IllegalArgumentException{
  checkKey(key);
  Entry<K,V> newEntry = new Entry(key, value);
  Position<Entry<K,V>> newEntry = treeSearch(root(), key);
  V tempVal;
  if(isExternal(p)){
    tree.set(p,entry);
    tree.addLeft(p,null);
    tree.addRight(p,null);
    tempVal=null;
  }
  else{
    tempVal = p.getValue();
    tree.set(p, newEntry);
  }
  return tempVal;
}
```

Operations on a BST

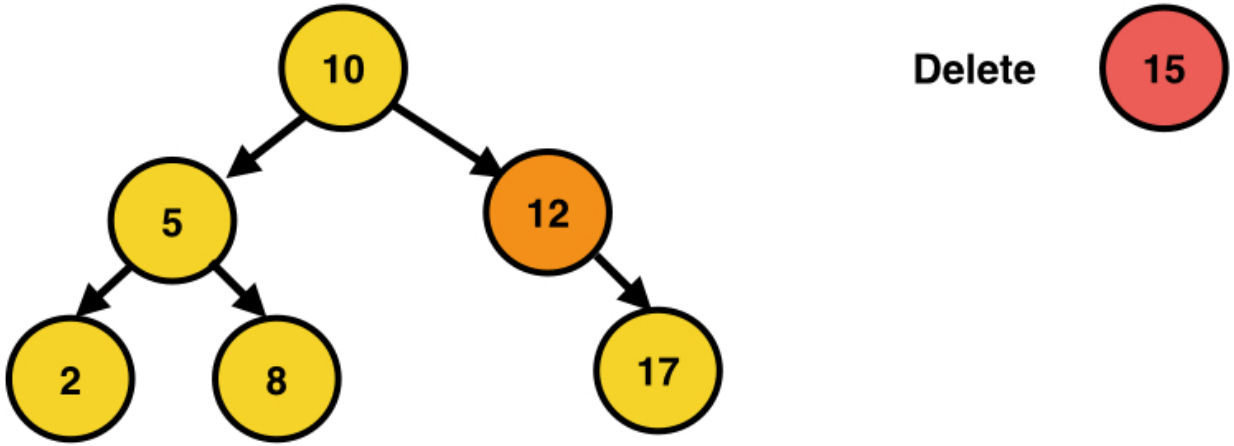
Deletion

Operations on a BST: Deletion



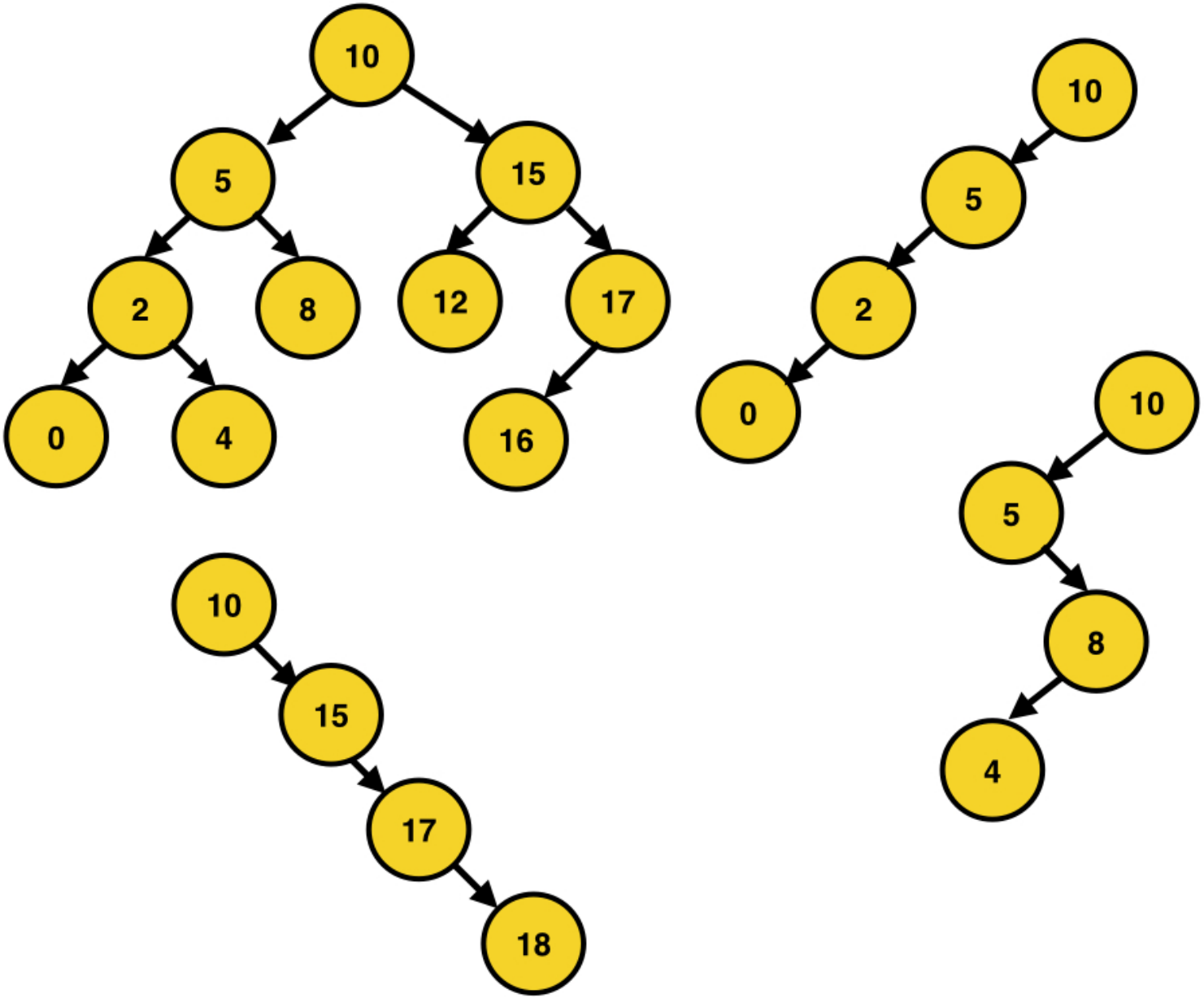

```
public V remove(K key) throws IllegalArgumentException{
  checkKey(key);
  Position<Entry<K,V>> p = treeSearch(root(),key);
  if(isExternal(p))
    return null;
  else{
    V old = p.getValue();
    if(isInternal(left(p)) && isInternal(right(p))){
      Position<Entry<K,V>> replacement = treeMax(left(p));
      set(p, replacement.getElement());
      p = replacement;
    }
    Position<Entry<K,V>> leaf = (isExternal(left(p))?left(p):right(p));
    remove(leaf);
    remove(p);
  }
}
```

Operations on a BST: Deletion




```
public V remove(K key) throws IllegalArgumentException{
  checkKey(key);
  Position<Entry<K,V>> p = treeSearch(root(),key);
  if(isExternal(p))
    return null;
  else{
    V old = p.getValue();
    if(isInternal(left(p)) && isInternal(right(p))){
      Position<Entry<K,V>> replacement = treeMax(left(p));
      set(p, replacement.getElement());
      p = replacement;
    }
    Position<Entry<K,V>> leaf = (isExternal(left(p))?left(p):right(p));
    remove(leaf);
    remove(p);
  }
}
```

Operations on a BST: Resulting BSTs



Operations on a BST

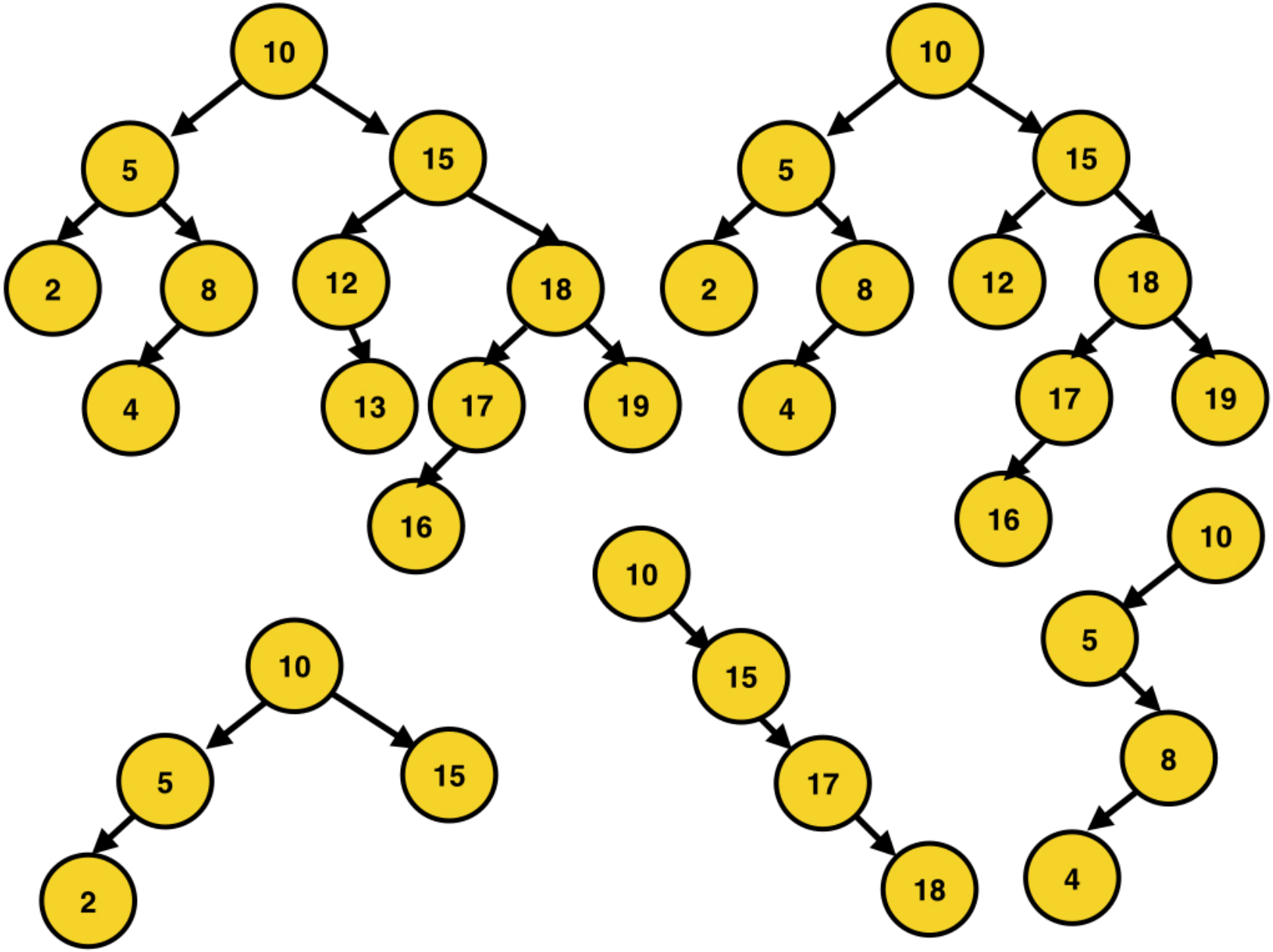
- Tree **structure** influences the performance of a BST
- Best case: Tree is **balanced** (i.e. height of subtrees of all nodes differ by only 1) and performance is O(logn)
- Worst case: Tree is left **linear** or right **linear** (performance degrades to O(n))
- Unlike heaps, there is **no** restriction on maintaining a balance in the structure of the BST

AVL Tree

What are AVL Trees?

- AVL stands for **A**delson-**V**elskii **L**andis
- This is a class of binary search trees that are **almost balanced**
- A tree satisfies this **AVL property** if the difference between heights of the left sub-tree and right sub-tree of all nodes is at most 1 (shortcut: compare the longest path of both subtrees)
- In an AVL tree, every insertion or deletion operation ensures that this AVL property is maintained
- It can be shown that the operations on an AVL tree is O(logn)

AVL Trees Examples

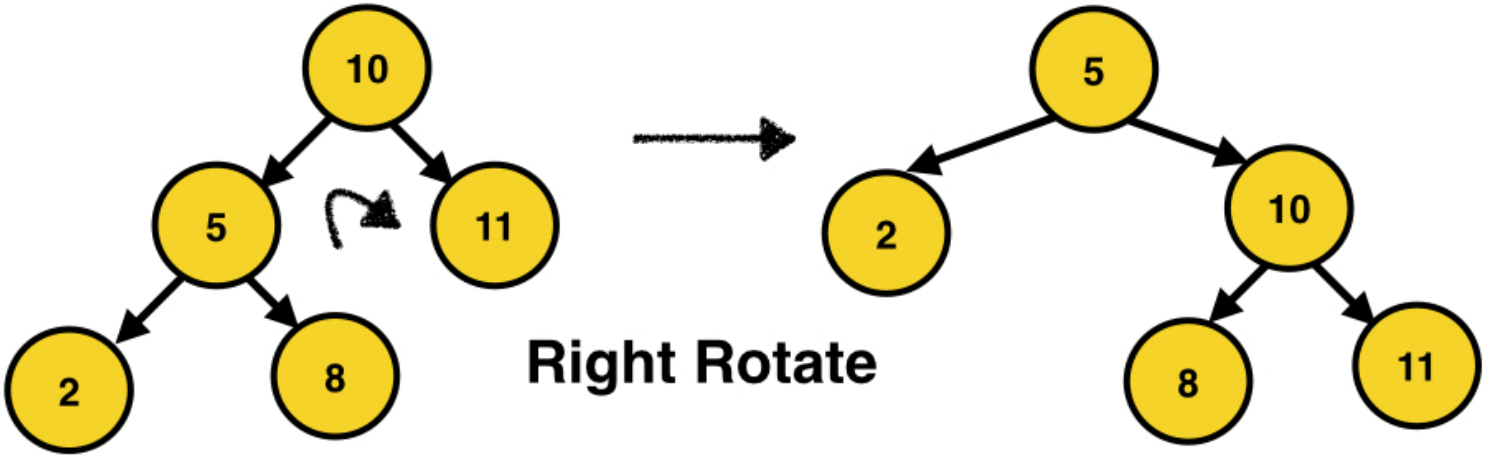


Operations on an AVL Tree

- Insertion or deletion operations may cause an AVL tree to lose its AVL property
- Operations to restore the AVL property include: **right rotation**, **left rotation**
- Four different cases that deviate the AVL property: **Left Left**, **Left Right**, **Right Right**, **Right Left**
- **Balance factor BF** (height of left subtree - height of right subtree) can be used to determine whether a tree heeds the AVL property or not

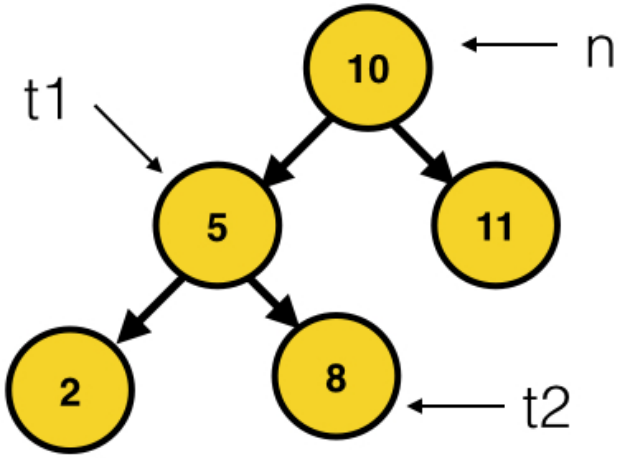
Right Rotation

AVL Trees: Right Rotate



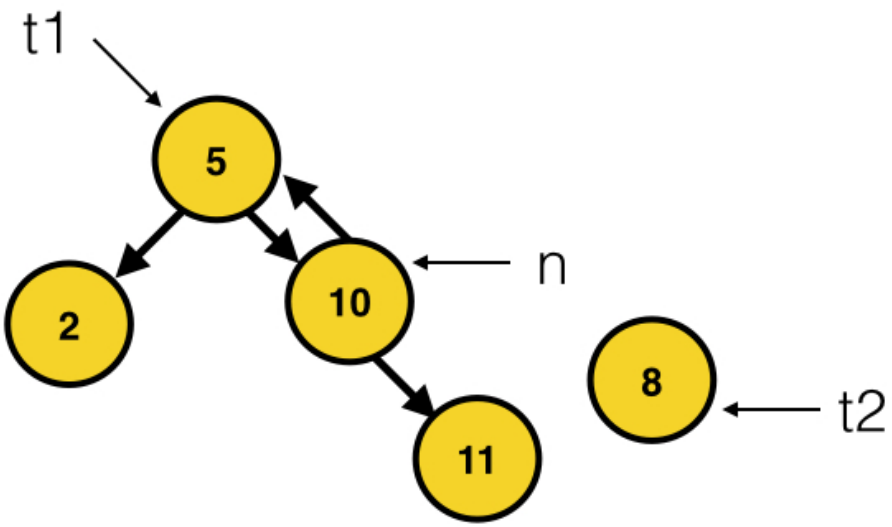

```
public Position<Entry<V,K>> rightRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = left(n);
  Position<Entry<V,K>> t2 = right(t1);
  t1.setRight(n);
  n.setLeft(t2);
  return t1;
}
```

AVL Trees: Right Rotate



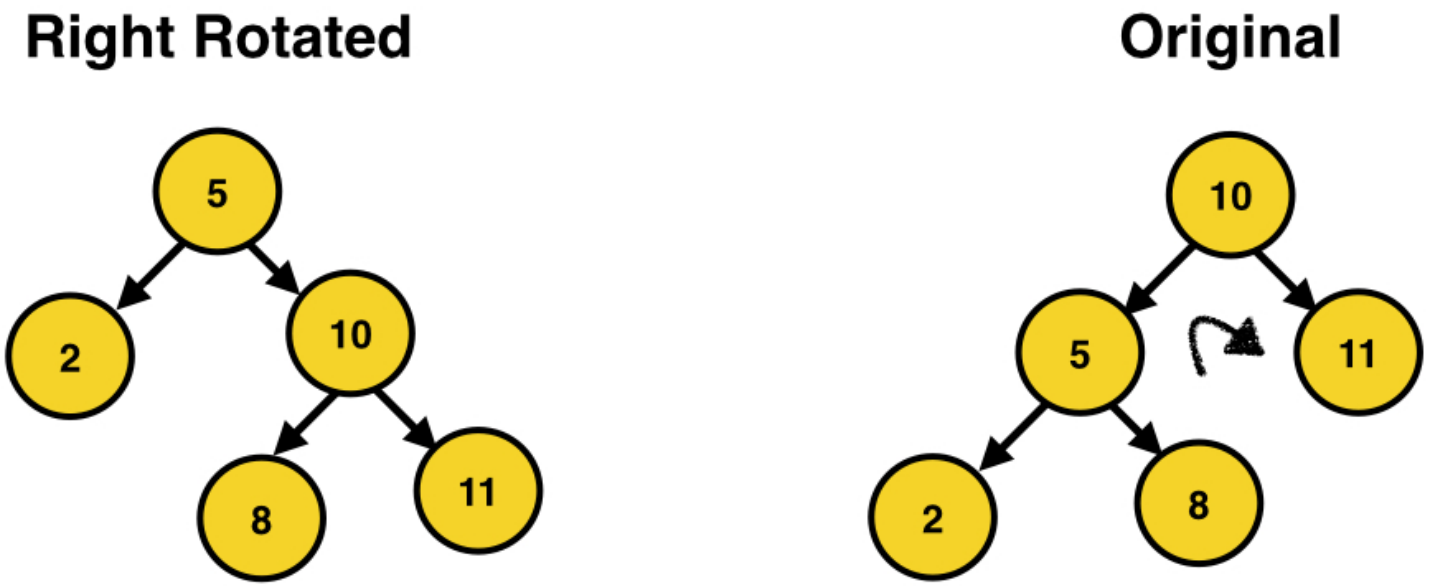

```
public Position<Entry<V,K>> rightRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = left(n);
  Position<Entry<V,K>> t2 = right(t1);
  t1.setRight(n);
  n.setLeft(t2);
  return t1;
}
```

AVL Trees: Right Rotate



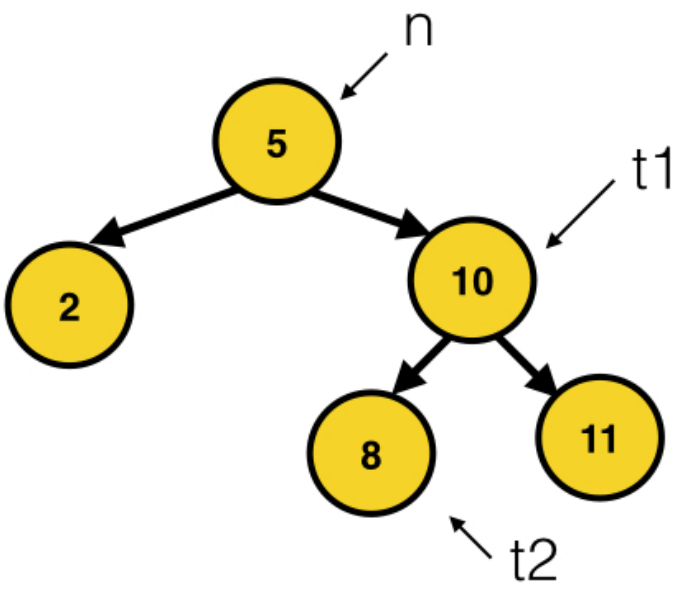

```
public Position<Entry<V,K>> rightRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = left(n);
  Position<Entry<V,K>> t2 = right(t1);
  t1.setRight(n);
  n.setLeft(t2);
  return t1;
}
```

AVL Trees: Right Rotate




```
public Position<Entry<V,K>> rightRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = left(n);
  Position<Entry<V,K>> t2 = right(t1);
  t1.setRight(n);
  n.setLeft(t2);
  return t1;
}
```

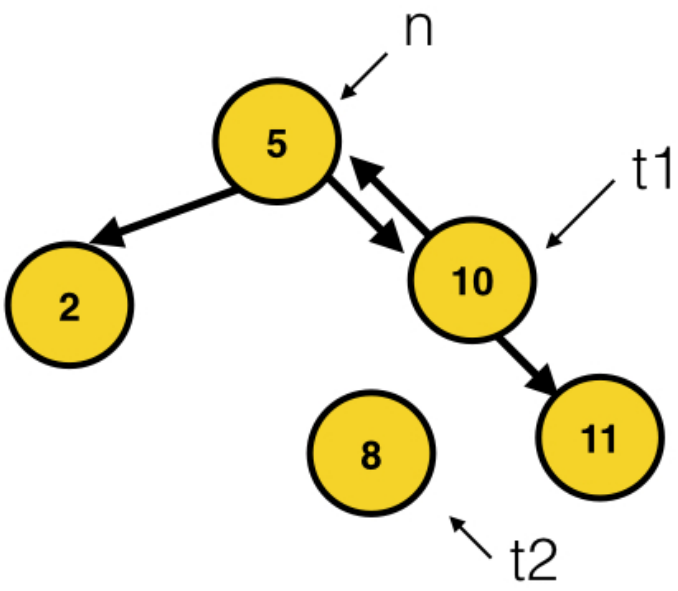
AVL Trees: Right Rotate




```
public Position<Entry<V,K>> leftRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = right(n);
  Position<Entry<V,K>> t2 = left(t1);
  t1.setLeft(n);
  n.setRight(t2);
  return t1;
}
```

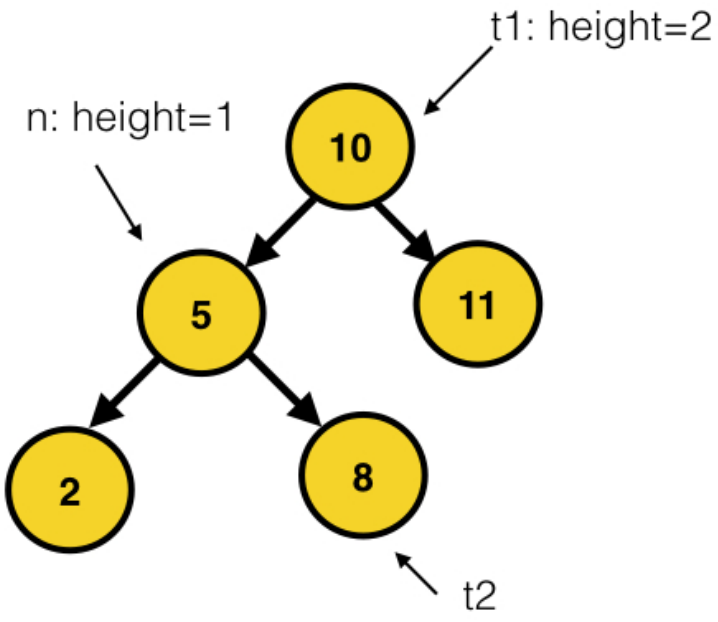
Left Rotation

AVL Trees: Left Rotate



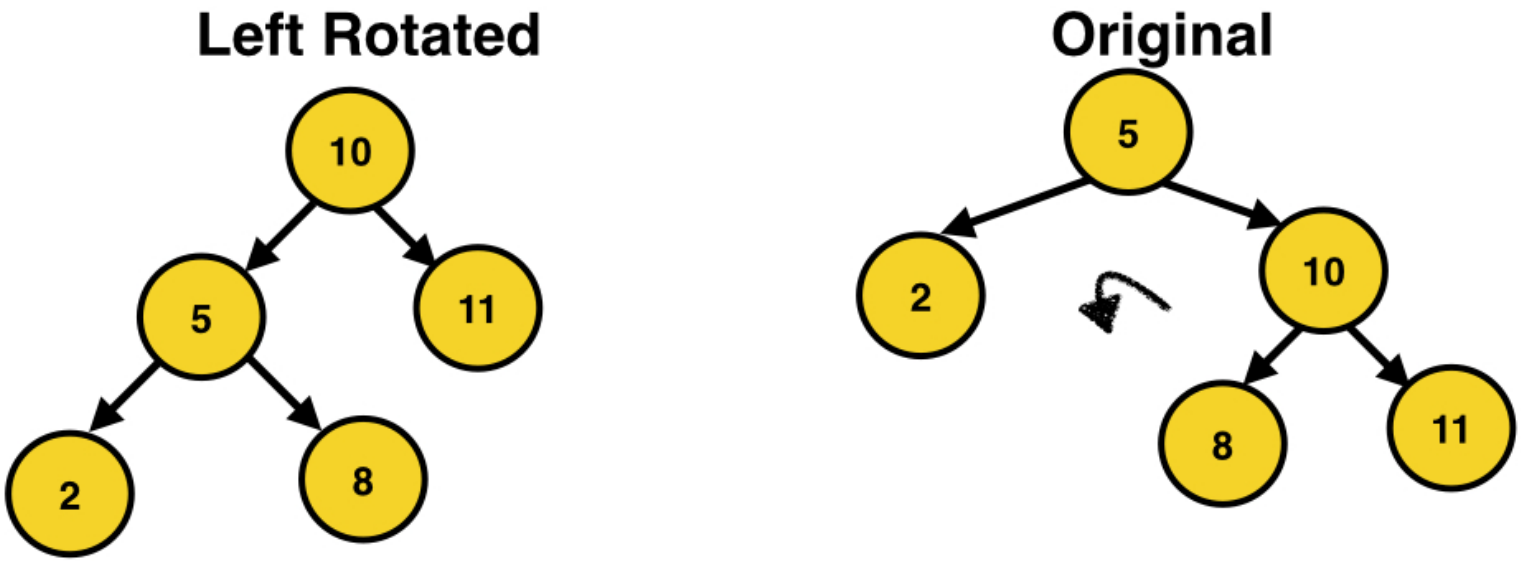

```
public Position<Entry<V,K>> leftRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = right(n);
  Position<Entry<V,K>> t2 = left(t1);
  t1.setLeft(n);
  n.setRight(t2);
  return t1;
}
```

AVL Trees: Left Rotate



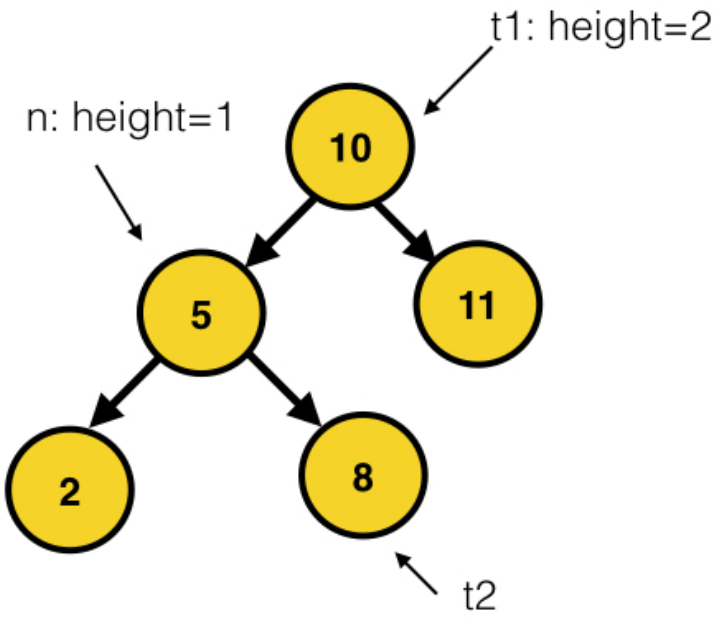

```
public Position<Entry<V,K>> leftRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = right(n);
  Position<Entry<V,K>> t2 = left(t1);
  t1.setLeft(n);
  n.setRight(t2);
  return t1;
}
```

AVL Trees: Left Rotate



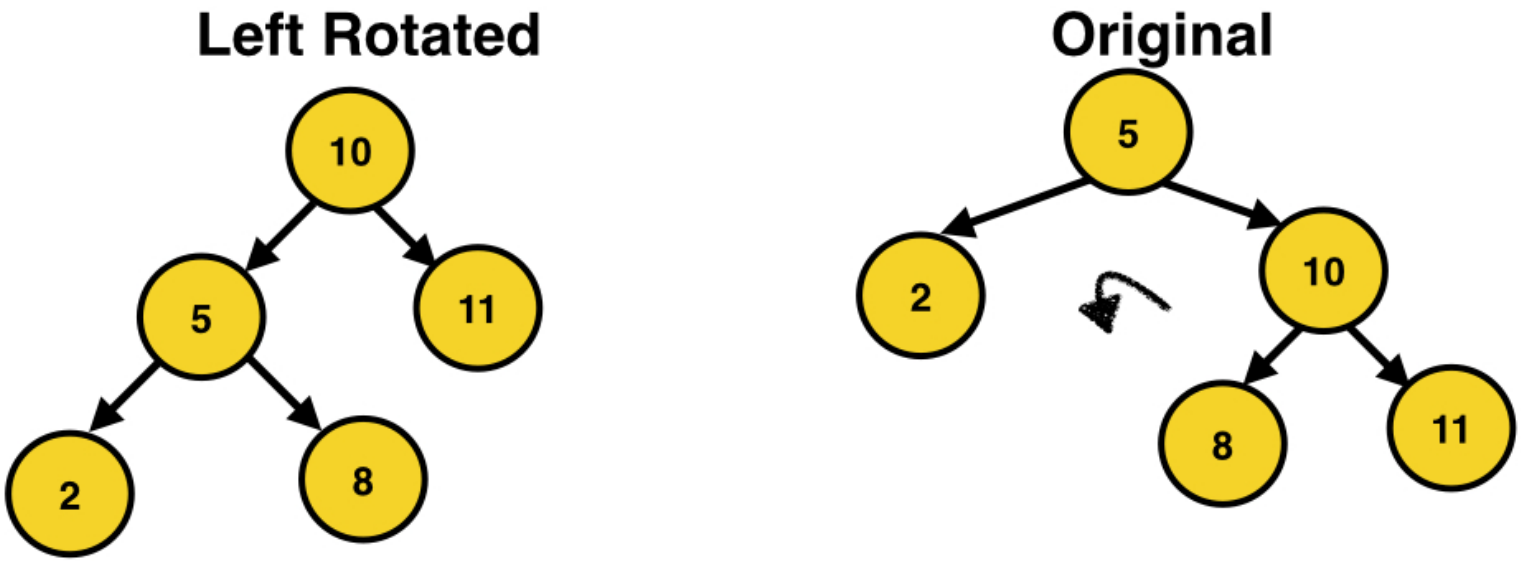

```
public Position<Entry<V,K>> leftRotate(Position<Entry<V,K>> n){
  Position<Entry<V,K>> t1 = right(n);
  Position<Entry<V,K>> t2 = left(t1);
  t1.setLeft(n);
  n.setRight(t2);
  return t1;
}
```

AVL Trees: Left Rotate




```
struct Node * leftRotate(struct Node * n)
{
    struct Node * t1=n->rChild;
    struct Node * t2=t1->lChild;
    t1->lChild=n;
    n->rChild=t2;
    n->height=max(height(n->lChild), height(n->rChild))+1;
    t1->height=max(height(t1->lChild), height(t1->rChild))+1;
    return t1;
}
```

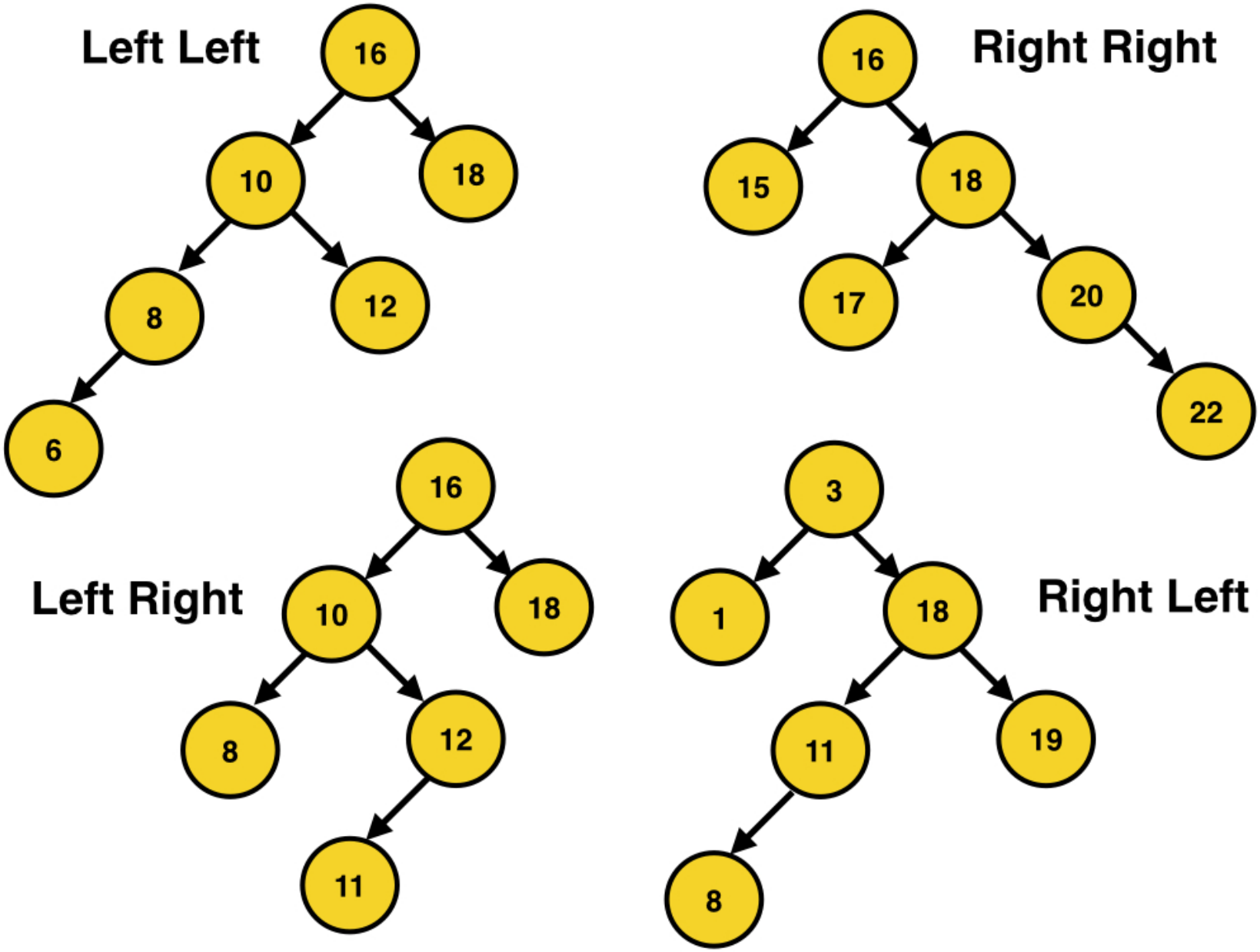
AVL Trees: Left Rotate




```
struct Node * leftRotate(struct Node * n)
{
    struct Node * t1=n->rChild;
    struct Node * t2=t1->lChild;
    t1->lChild=n;
    n->rChild=t2;
    n->height=max(height(n->lChild), height(n->rChild))+1;
    t1->height=max(height(t1->lChild), height(t1->rChild))+1;
    return t1;
}
```

Four Common Instances with non-AVL Property

AVL Trees: Four Instances with non-AVL Property

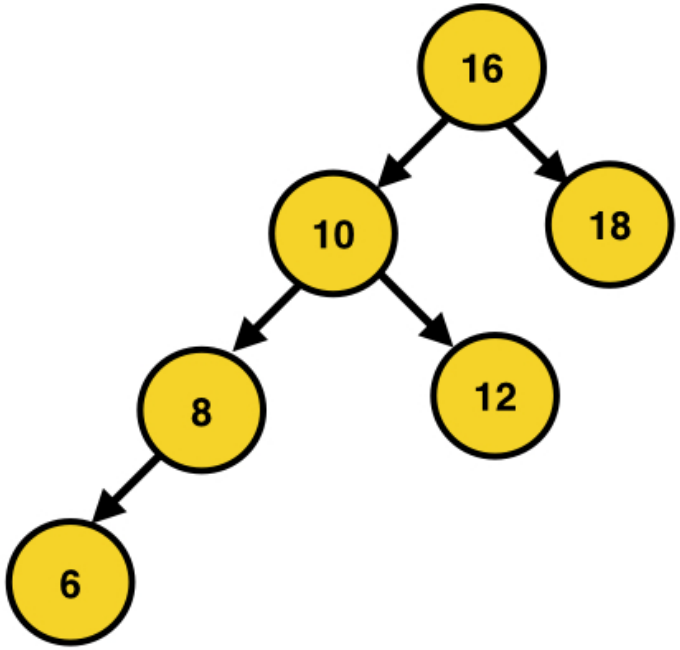


AVL Trees: Identifying the type of imbalance introduced

- After **insertion**, check every node to see if it is balanced or not (with BF)
- If the BF is **greater** than 1, then the left subtree of the node is deeper than the right subtree.
  - If the node **inserted** is lesser than the key in the left child of the unbalanced node, then this is a **left left** tree
  - If the node **inserted** is greater than the key in the left child of the unbalanced node, then this is a **left right** tree
- If the BF is **lesser** than -1, then the right subtree of the node is deeper than the left subtree.
  - If the node inserted is **lesser** than the key in the right child of the unbalanced node, then this is a **right left** tree
  - If the node inserted is **greater** than the key in the right child of the unbalanced node, then this is a **right right** tree
- Similar steps can be taken to detect imbalance in a tree after **deletion** operations

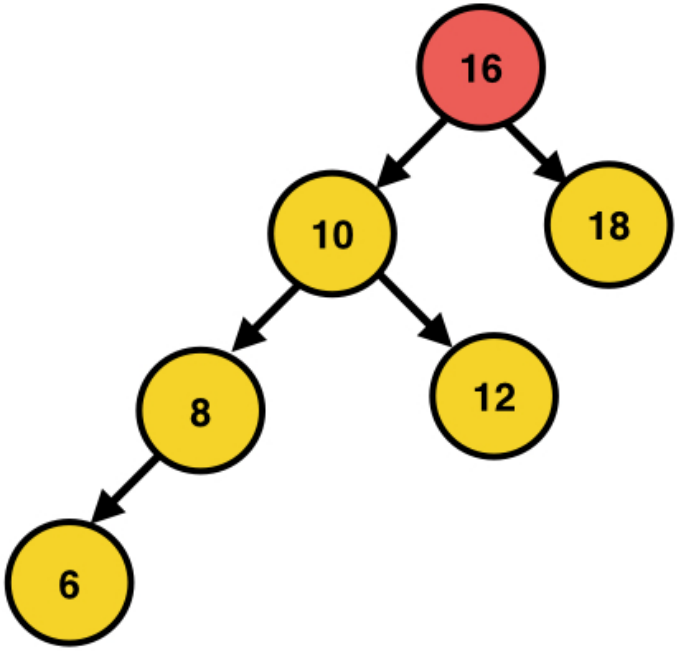
Balancing a Left-Left Tree

AVL Trees: Balancing a Left Left Tree



- How can a Left Left tree be identified?
- Check the balance factor

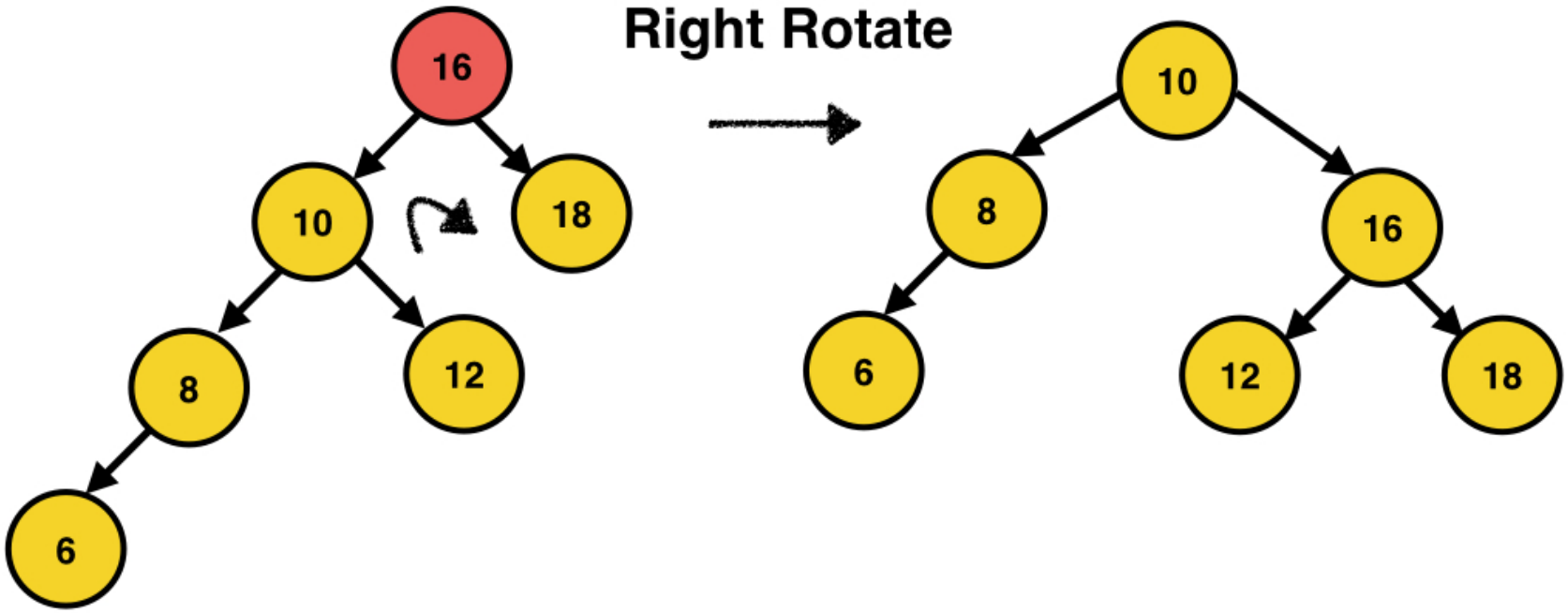
AVL Trees: Balancing a Left Left Tree



BF>1
Data inserted < left child of unbalanced node

Balance factors:
Node 16: 2
Node 10: 1
Node 8: 0
Node 6: 0
Node 18: 0
Node 12: 0

AVL Trees: Balancing a Left Left Tree

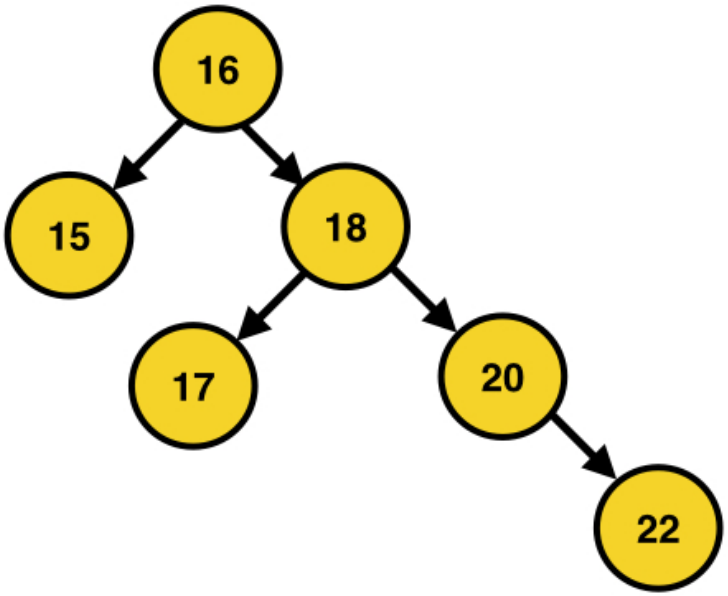


Balance factors:
Node 16: 2
Node 10: 1
Node 8: 1
Node 6: 0
Node 18: 0
Node 12: 0

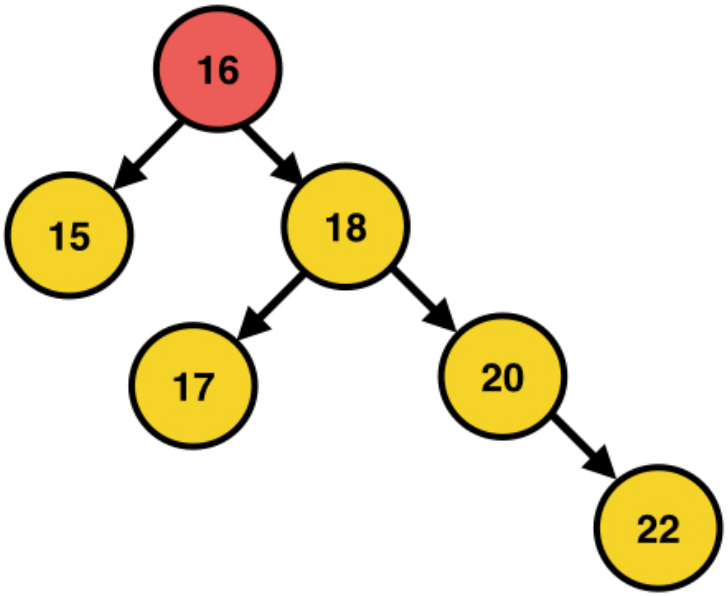
Balance factors:
Node 16: 0
Node 10: 0
Node 8: 0
Node 6: 0
Node 18: 0
Node 12: 0

Balancing a Right-Right Tree

AVL Trees: Balancing a Right Right Tree



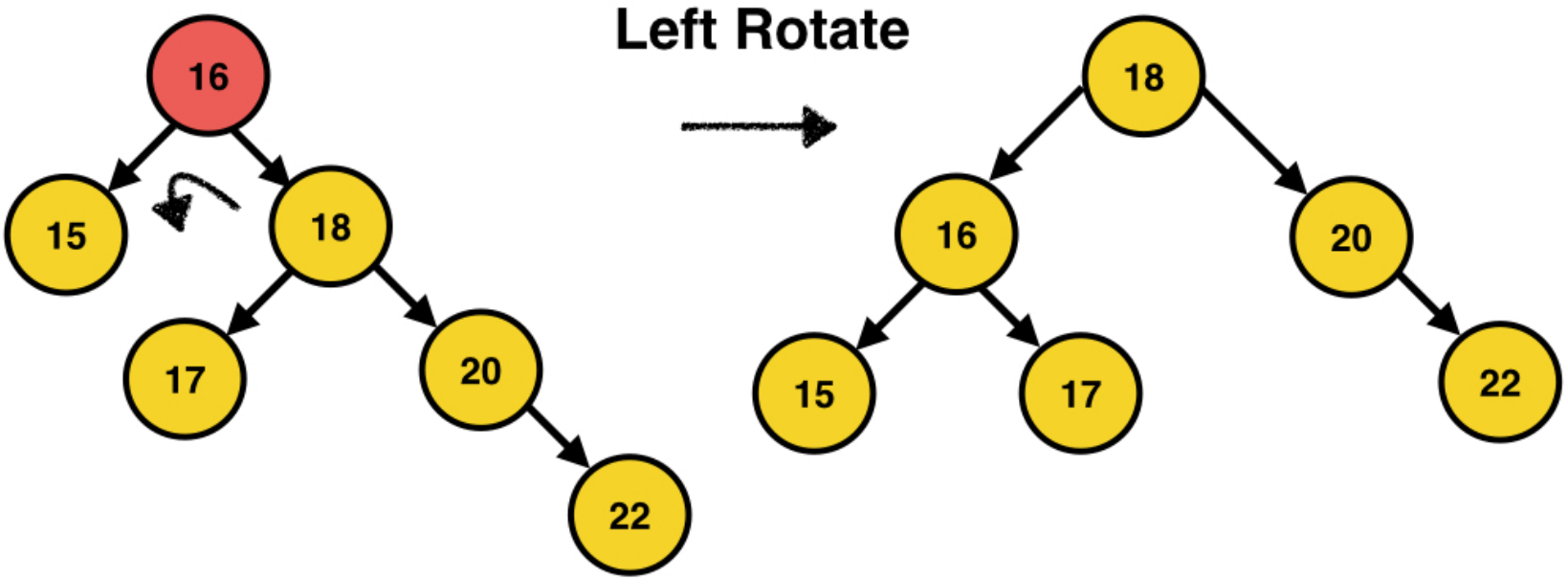
AVL Trees: Balancing a Right Right Tree



BF<-1
Data inserted > right child of unbalanced node

Balance factors:
Node 16: -2
Node 18: -1
Node 20: 0
Node 22: 0
Node 15: 0
Node 17: 0

AVL Trees: Balancing a Right Right Tree

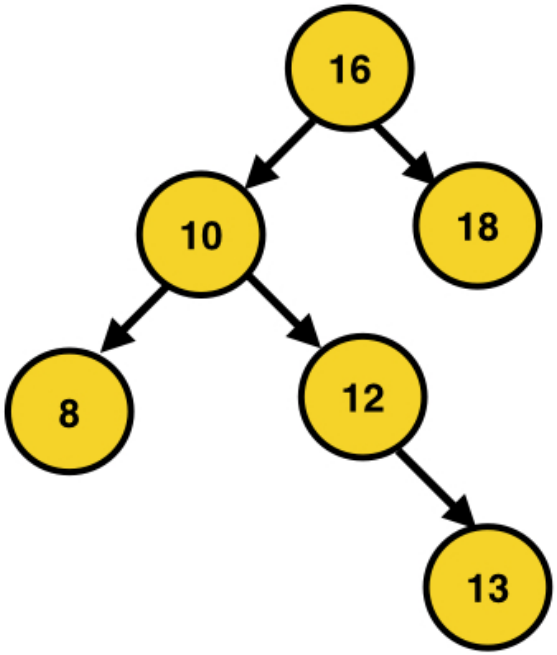


Balance factors:
Node 16: -2
Node 18: -1
Node 20: 0
Node 22: 0
Node 15: 0
Node 17: 0

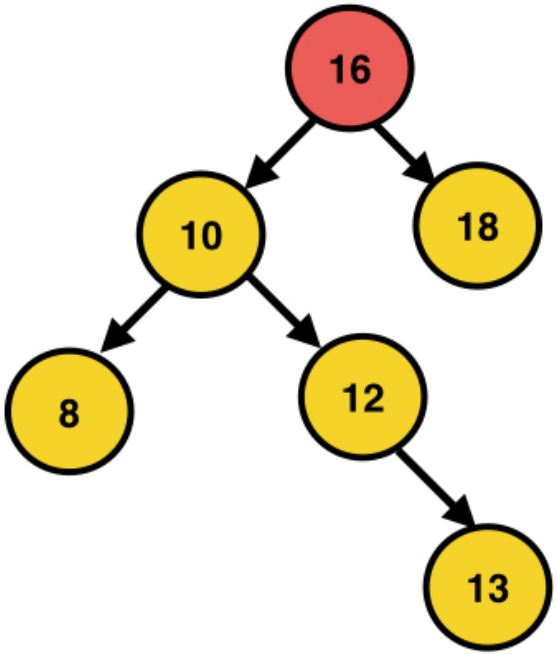
Balance factors:
Node 16: 0
Node 18: 0
Node 20: 0
Node 22: 0
Node 15: 0
Node 17: 0

Balancing a Left-Right Tree

AVL Trees: Balancing a Left Right Tree



AVL Trees: Balancing a Left Right Tree

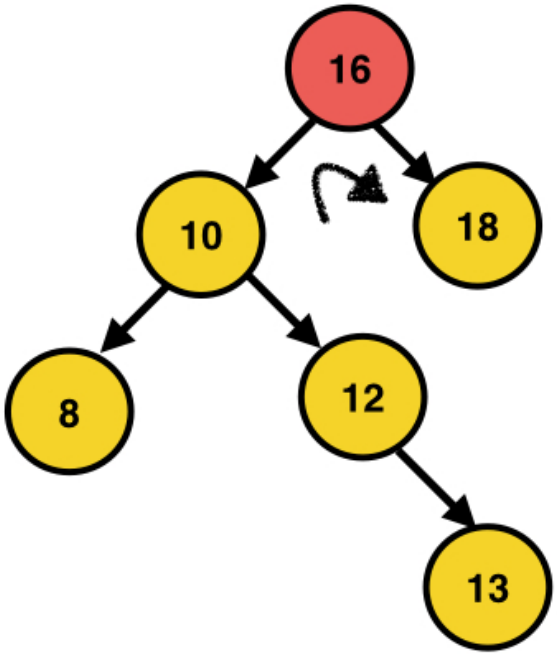


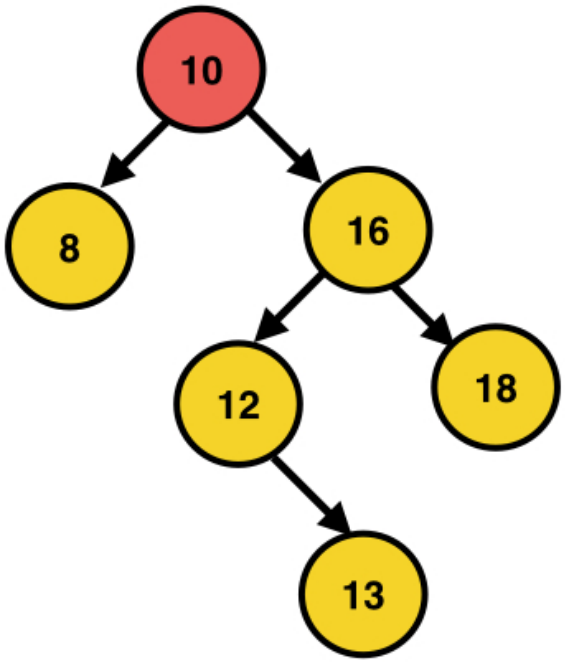
BF>1
Data inserted > left child of unbalanced node

Balance factors:
Node 16: 2
Node 18: 0
Node 10: -1
Node 8: 0
Node 12: 0
Node 13: 0

AVL Trees: Balancing a Left Right Tree

Can we fix this with a right rotate?



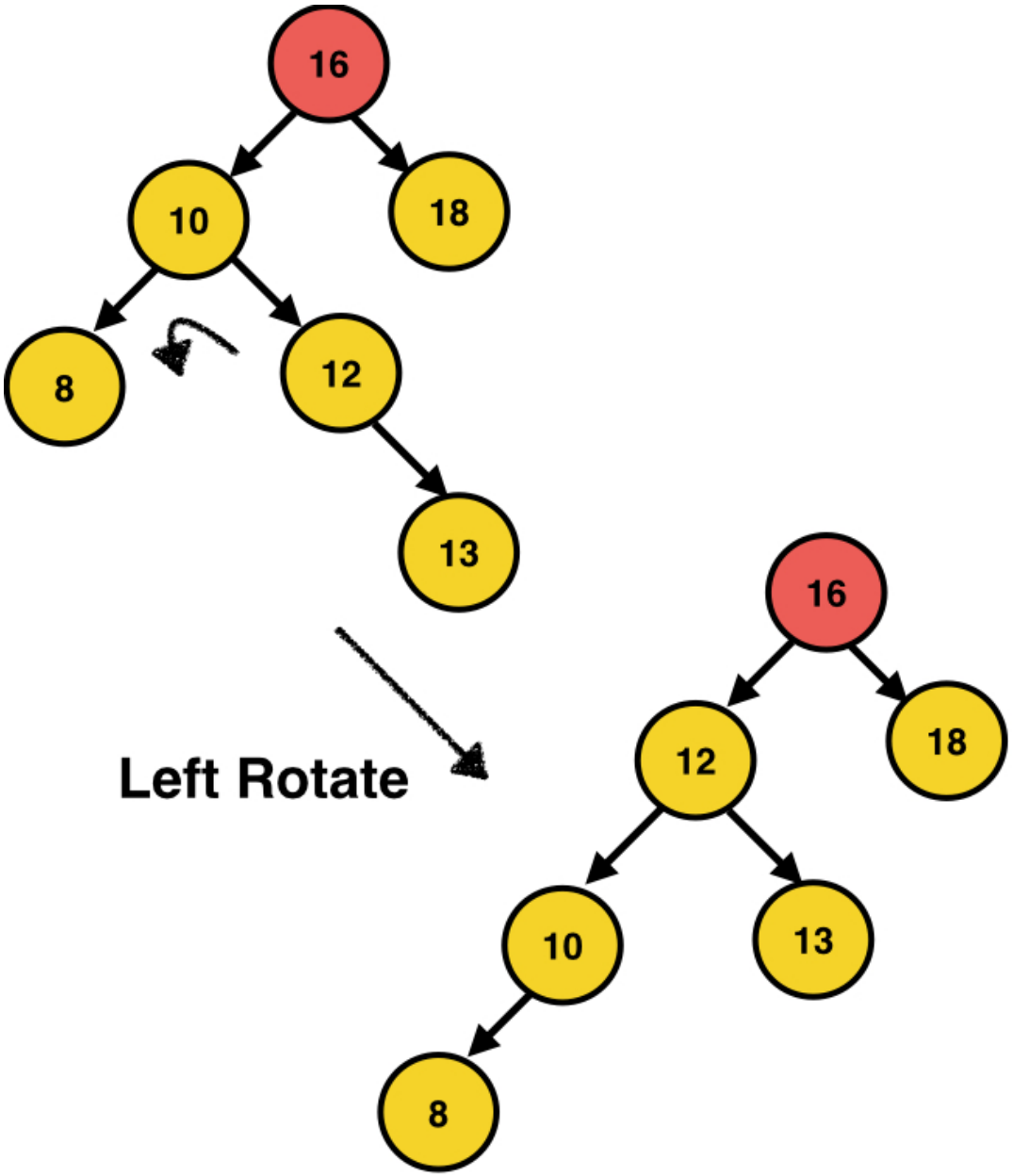


Balance factors:
Node 16: 2
Node 18: 0
Node 10: -1
Node 8: 0
Node 12: 0
Node 13: 0

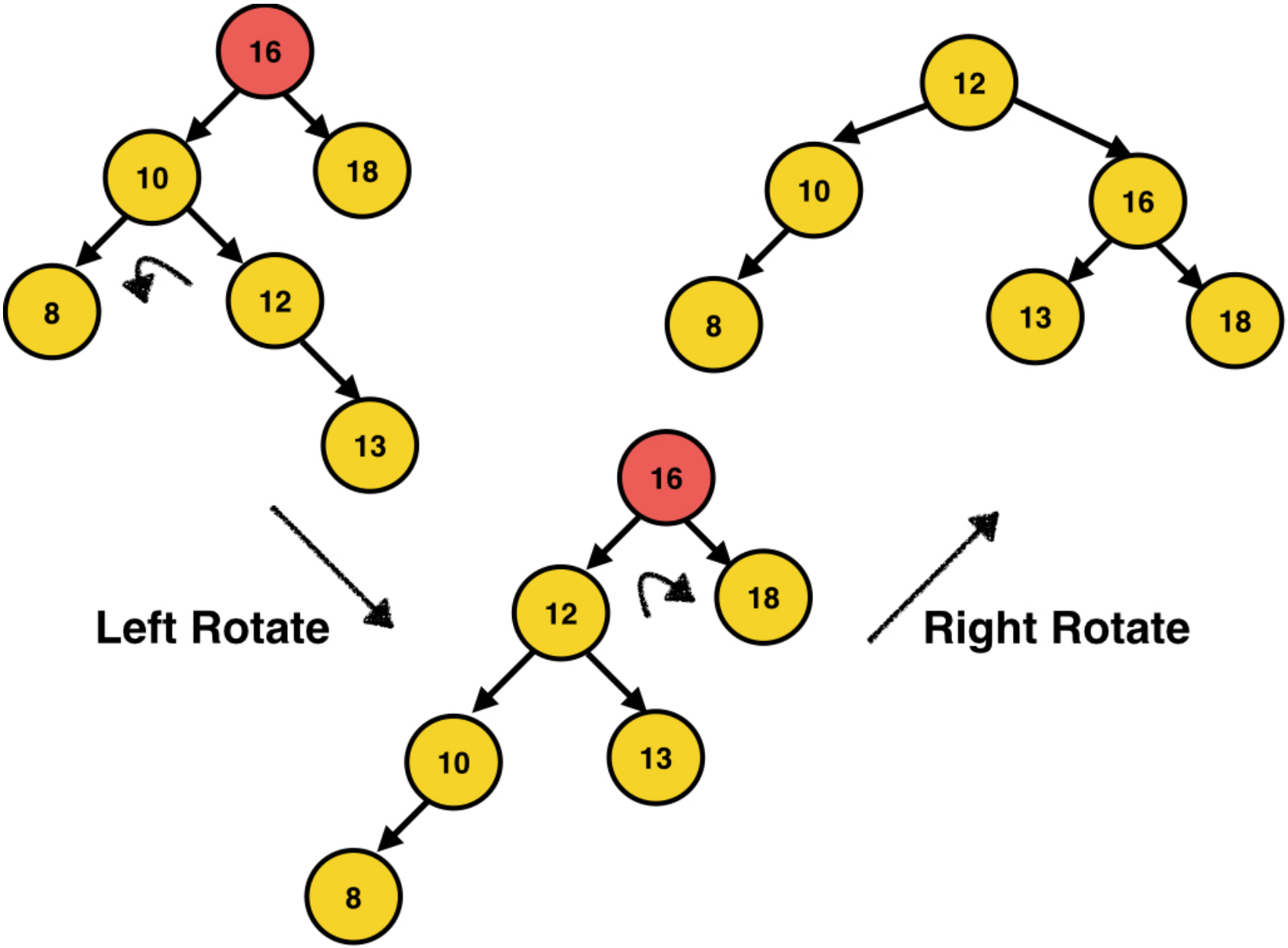
No!! Same problem!

Balance factors:
Node 16: 1
Node 18: 0
Node 10: -2
Node 8: 0
Node 12: 0
Node 13: 0

AVL Trees: Balancing a Left Right Tree

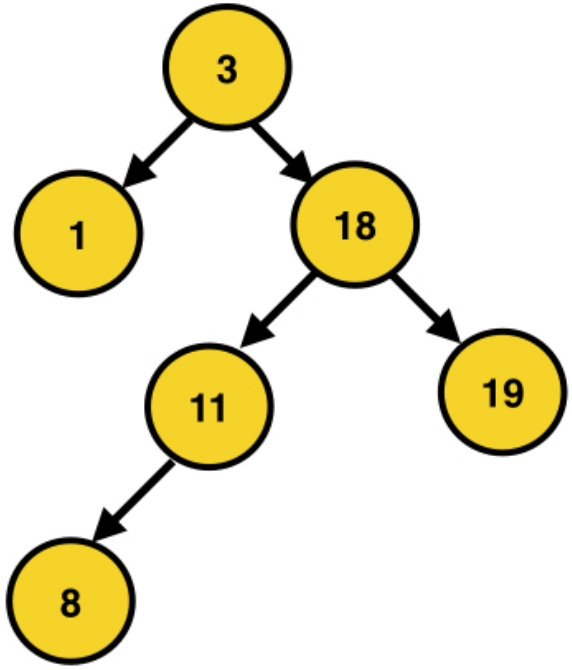


AVL Trees: Balancing a Left Right Tree

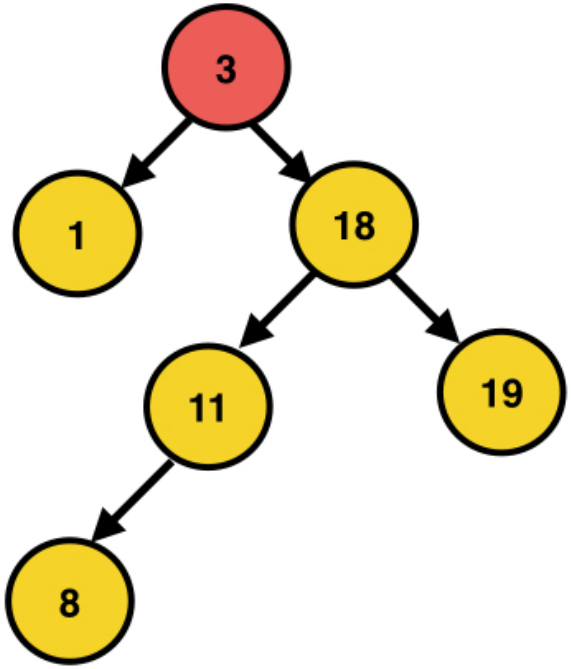


Balancing a Right-Left Tree

AVL Trees: Balancing a Right Left Tree



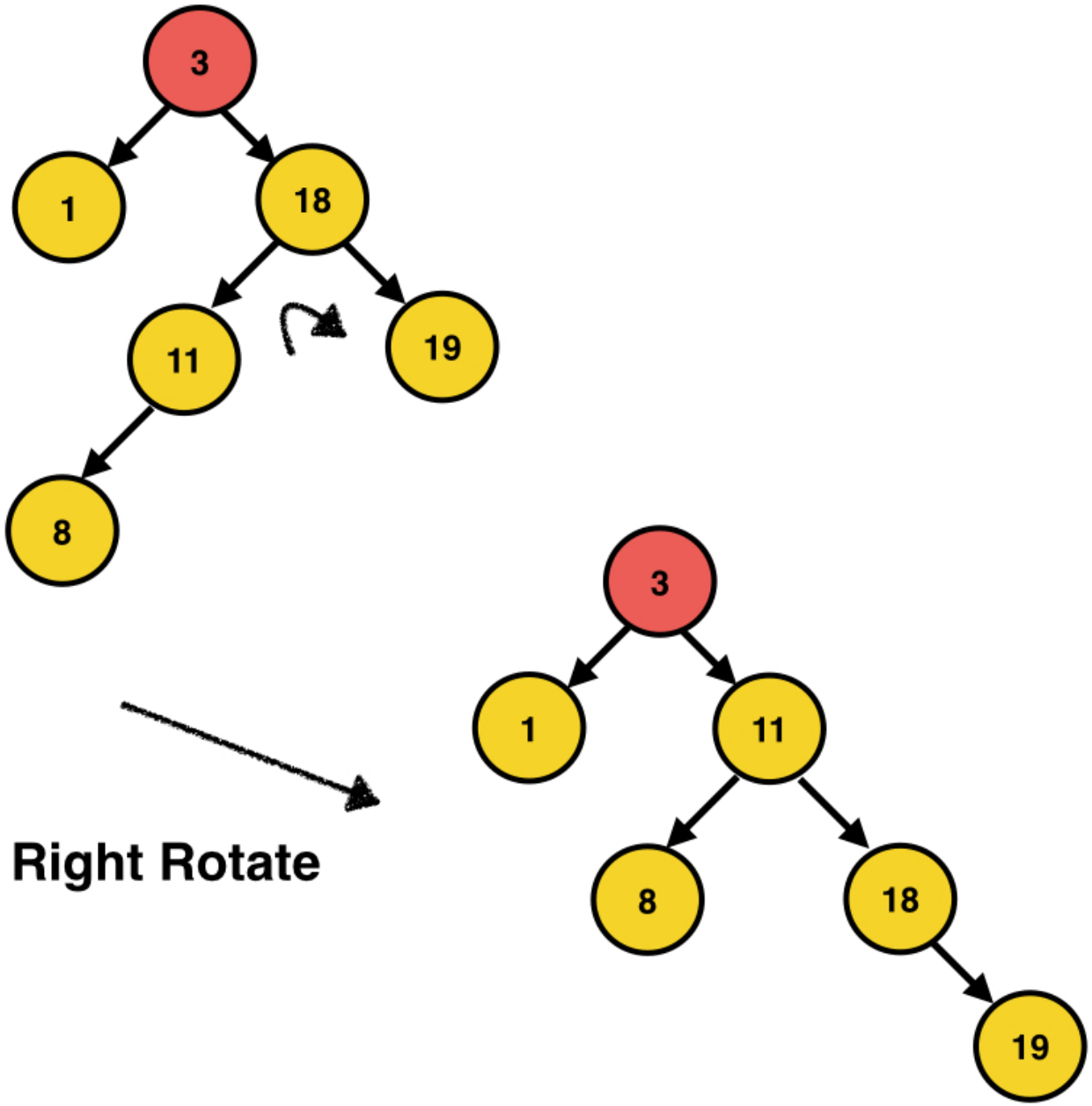
AVL Trees: Balancing a Right Left Tree



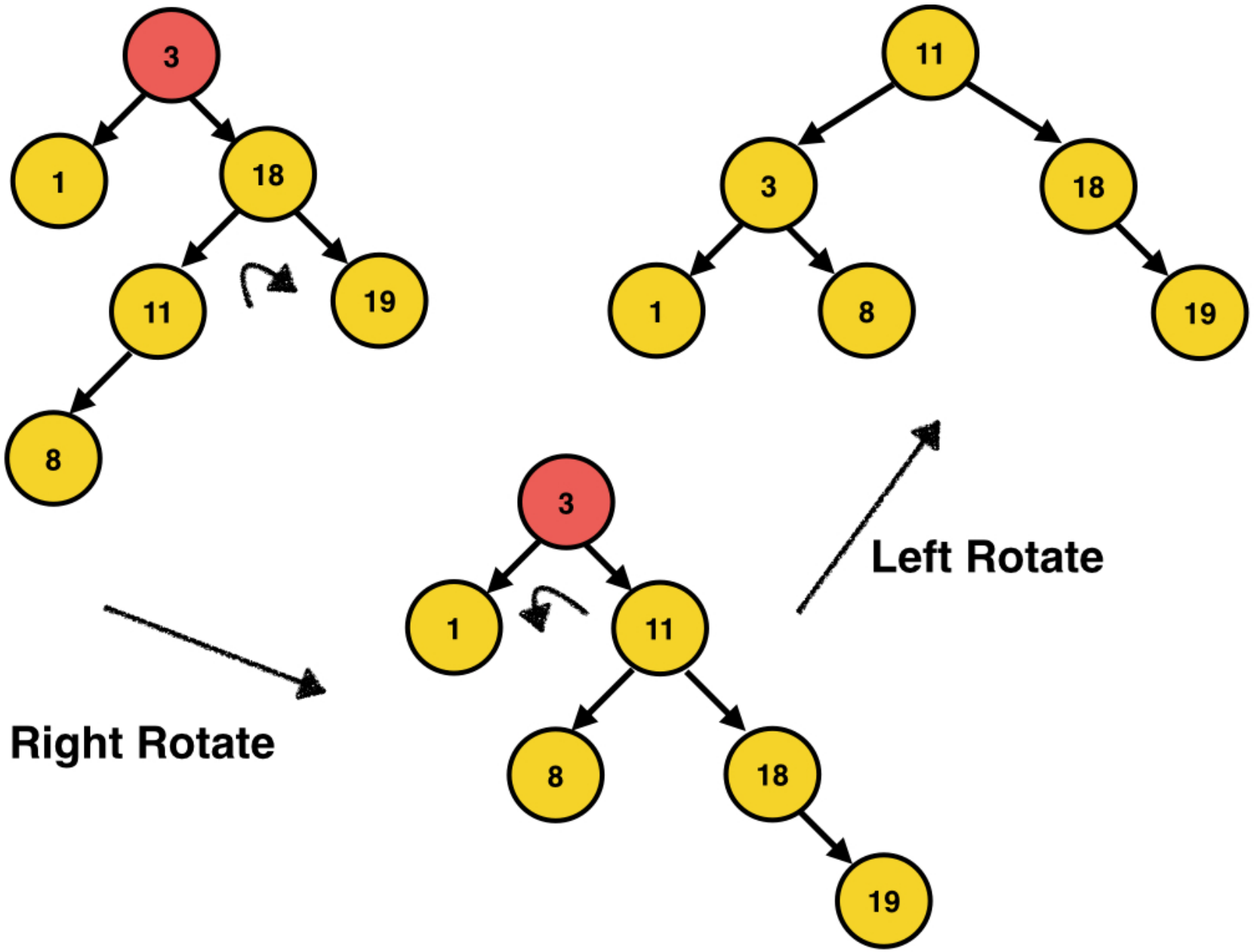
BF<-1
Data inserted < right child of unbalanced node

Balance factors:
Node 3: -2
Node 1: 0
Node 18: 1
Node 11: 0
Node 19: 0
Node 8: 0

AVL Trees: Balancing a Right Left Tree



AVL Trees: Balancing a Right Left Tree



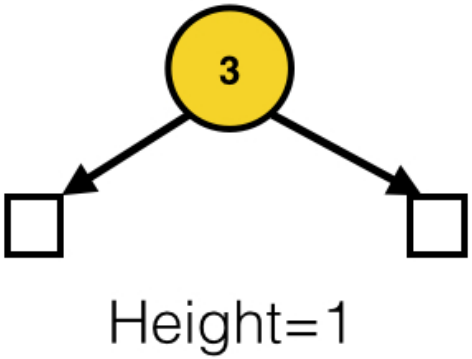
Efficiency of an AVL Tree

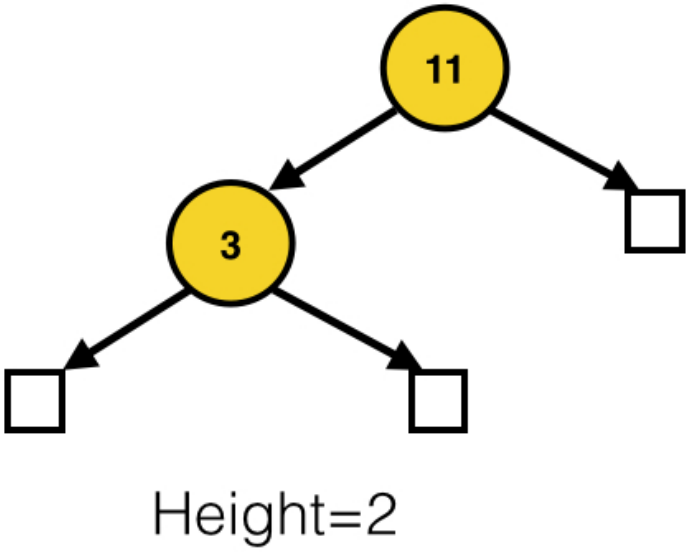
AVL Trees: Efficiency

- **Modifying** or **search** operations on an AVL tree is O(h) where h is the height of the AVL tree
- Nodes in an AVL tree are typically placed on **two adjacent levels**
- The **minimum** number of nodes on an AVL tree can be computed with the following **recursive** definition:
- An AVL tree consists of:
  - Root node
  - One subtree with height being at least $h - 1$
  - Another subtree with height being at least $h - 2$
- Assuming that the AVL tree under consideration is an extended binary tree (empty children are also accounted for) then the above is a recurrence relation:

$$N_1 = 1$$
$$N_2 = 2$$
$$N_h = N_{h-1} + N_{h-2} + 1$$

AVL Trees: Efficiency





Assuming that the AVL tree under consideration is represented as an extended binary tree (empty children are also accounted for) then the recursive definition can be expressed as a recurrence relation:

$$N_1 = 1$$
$$N_2 = 2$$
$$N_h = N_{h-1} + N_{h-2} + 1$$

AVL Trees: Efficiency

This is a Fibonacci sequence!

$$N_1 = 1$$
$$N_2 = 2$$
$$N_h = N_{h-1} + N_{h-2} + 1$$

h	f_h	N_h
1	1	1
2	2	2
3	3	4
4	5	7
5	8	12
6	13	20

$$N_h = f_{h+1} - 1$$
$$2^{\lfloor (h+1)/2 \rfloor} \leq f_h \leq 2^h \rightarrow 2^{(h+1)/2} - 1 \leq N_h \leq 2^h - 1$$

AVL Trees: Efficiency

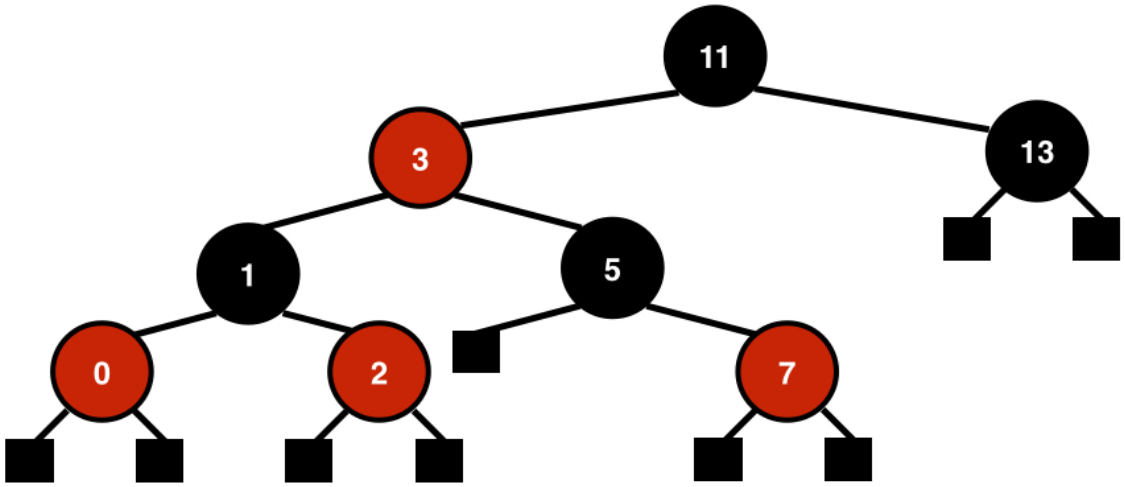
- The height is **bounded** by $log(n)$ and therefore the complexity of searching an AVL tree is O(logn)!
- During an insertion, the tree may need to be **re-balanced**
- Since balancing operations require **constant** time, complexity of insertion and deletion is also O(logn)

Red-Black Tree

Introduction to Red-Black Tree

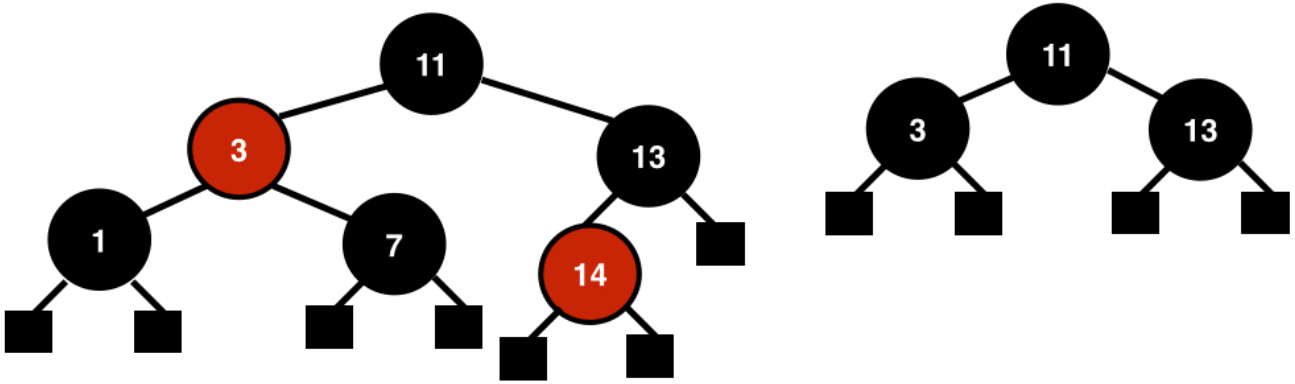
- Insertion/deletion operations in an AVL tree maintain **balance** in the structure of the tree
- If the number of insertions and deletions is frequent, then there will be a large number of rotations
- **Red-black** tree is a **binary search tree** that uses node colouring properties to maintain certain degree of balance in the tree without requiring too many rotations
- **AVL** tree is suited for **frequent search operations** and red-black trees are suited for **frequent insertions/deletions**

Properties of a Red-Black Tree



- Red-black trees are **BSTs** and are expressed in **extended form**
- (P1) Nodes including NULL nodes are **either red or black** (need an additional colour field to indicate the colour of the node)
- (P2) Root and leafs must be all **black**
- (P3) **Parent of a red** node must be **black**
- (P4) All paths from a node to its descendent leafs must have the **same number of black nodes**

Height of a Red-Black Tree



- **Black height of a node x (BH_x)**: Number of black nodes from that node to all possible descendent leafs
- Number of black nodes n_b in a tree rooted at r with BH_r is at least $2^{BH_r} - 1$
- Since the red nodes can **alternate** with every black node along the path from the root to the leaf in the **worst case** (P3), the height of the tree h is $h \leq 2BH_r$
- Then, $n_b \geq 2^{\frac{h}{2}} - 1 \Leftrightarrow n \geq 2^{\frac{h}{2}} - 1$ as $n \geq n_b$ resulting in $2log_2(n+1) \geq h$. Hence O(h) is $O(logn)$

Operations on a Red-Black Tree

- Complexity of a **search** operation is **bounded by the height** of the tree (i.e. path from the root to the deepest leaf in the tree)
- As the height of the red-black tree is bounded from above by $2log_2(n+1)$, the complexity of search is O(logn)
- Insertion/deletion operations can result in **violations** to one or more of red-black tree properties

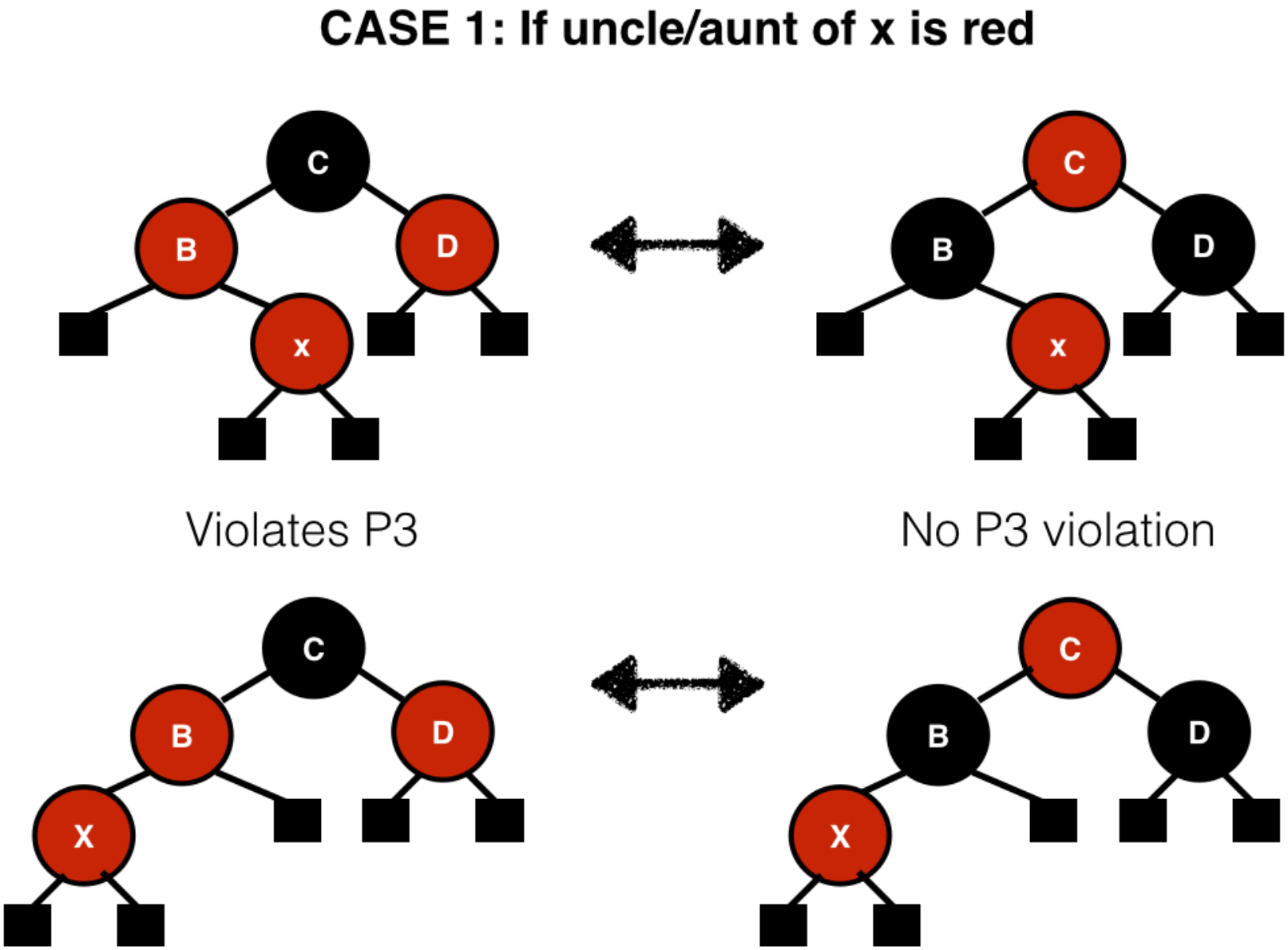
Insertion into a Red-Black Tree

- A new node is first set to red and then inserted into the tree in a manner similar to **BST insertion**
- This insertion may **violate** P3 (parent might be red) but will not violate P4
- In order to restore these properties, the tree is **traversed** upwards from the newly inserted node via recoloring until the violation can be fixed with a rotation
- Each node in the BST has a **parent pointer** as well to facilitate moving up the tree

Insertion into a Red-Black Tree

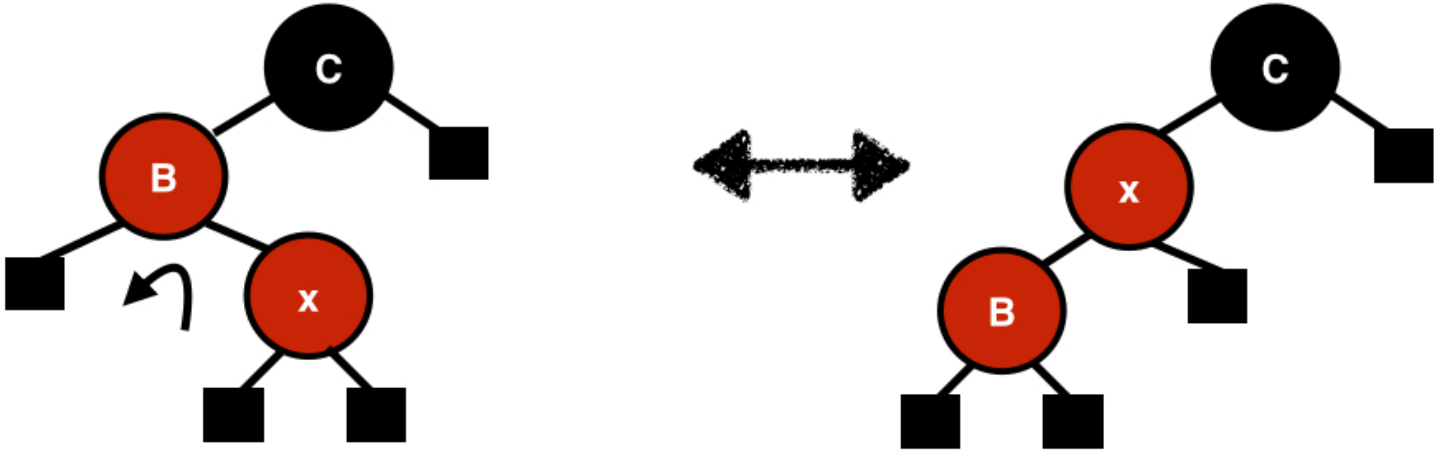
- Insertion process can be **generalized** into cases that involve the parent, grandparent and siblings of parent:
  - C1: If the new node's parent is the left child of the new node's grandparent, then there exists three cases
  - C2: Similarly, there are three cases if the new node's parent is the right child of the grandparent
- Since for both cases operations are **symmetric**, let us consider the three cases for C1

Insertion into a Red-Black Tree: Case 1



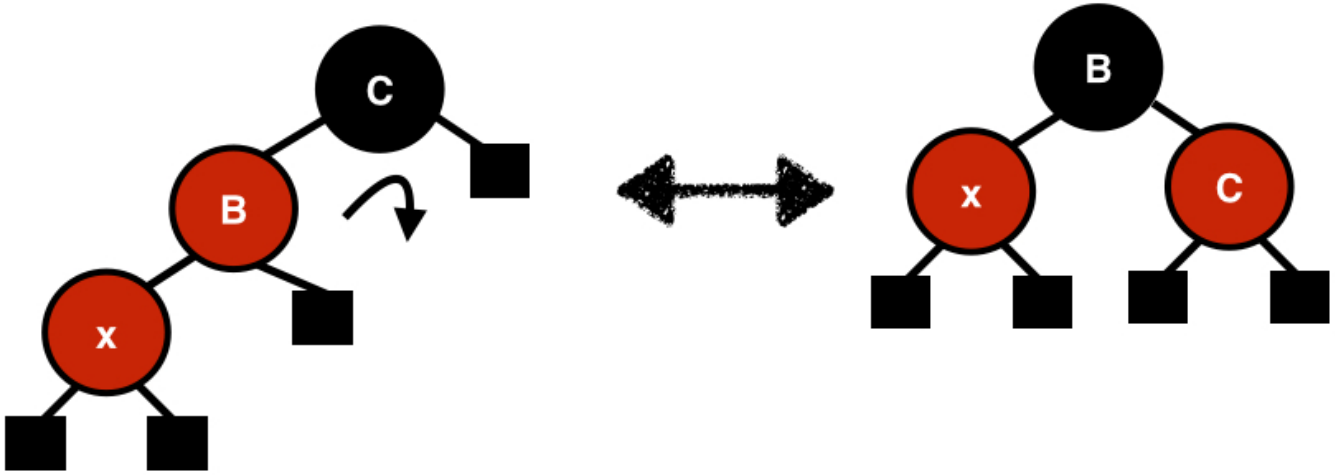
Insertion into a Red-Black Tree: Case 2

CASE 2: If x is the right child and uncle/aunt is not red



Insertion into a Red-Black Tree: Case 3

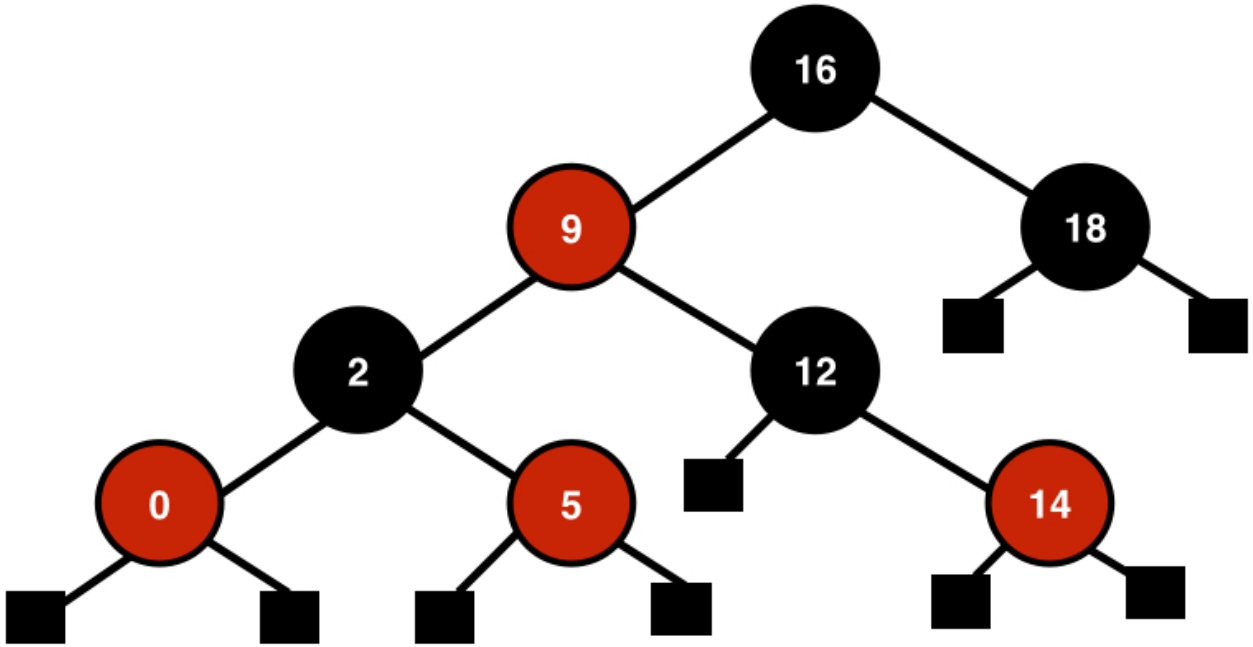
CASE 3: If x is the left child and uncle/aunt is not red



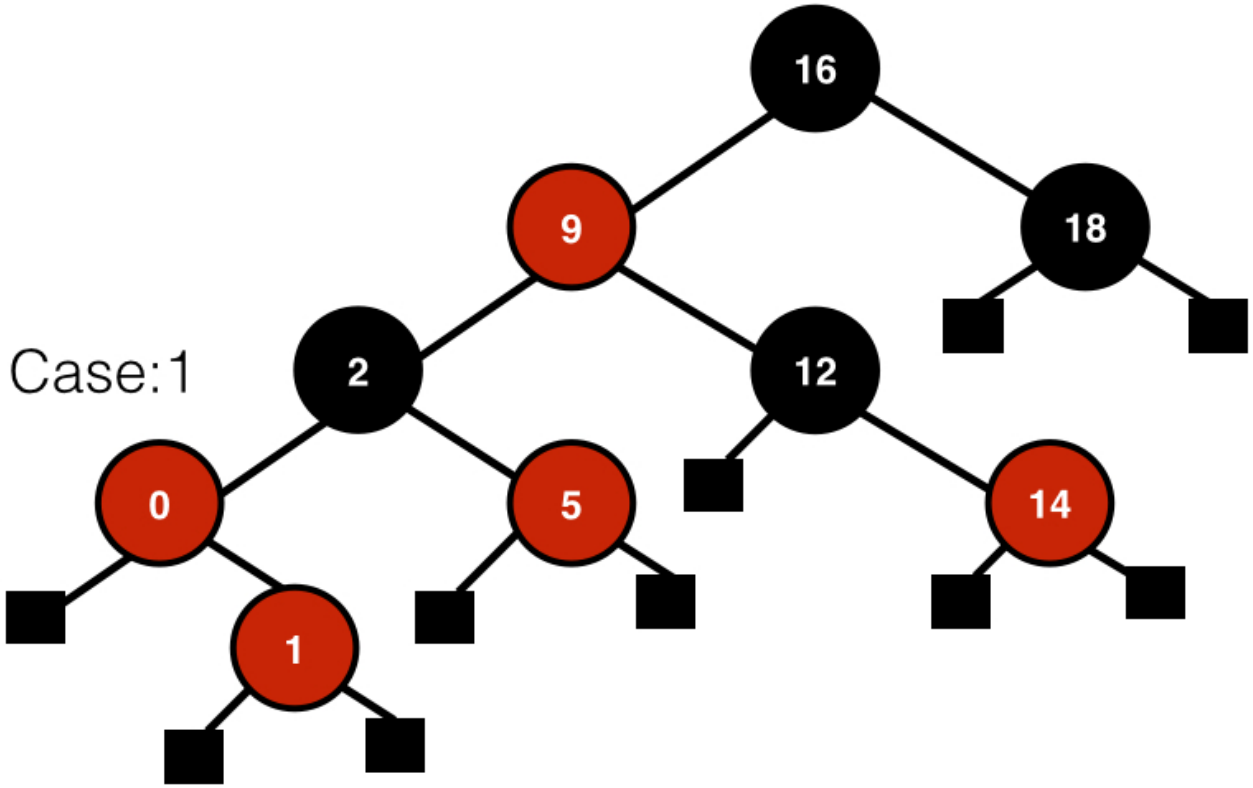
Insertion into a Red-Black Tree

Inserting into a Red-Black Tree

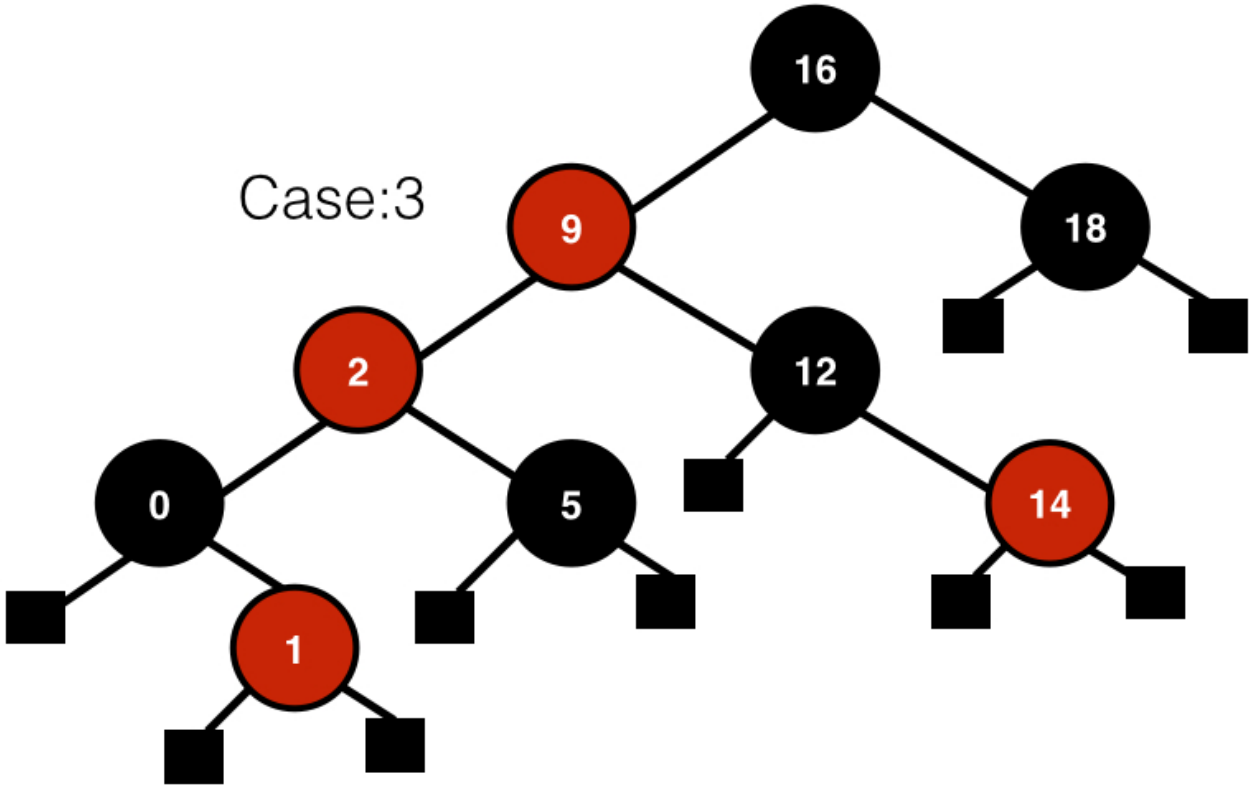
Insertion into a Red-Black Tree



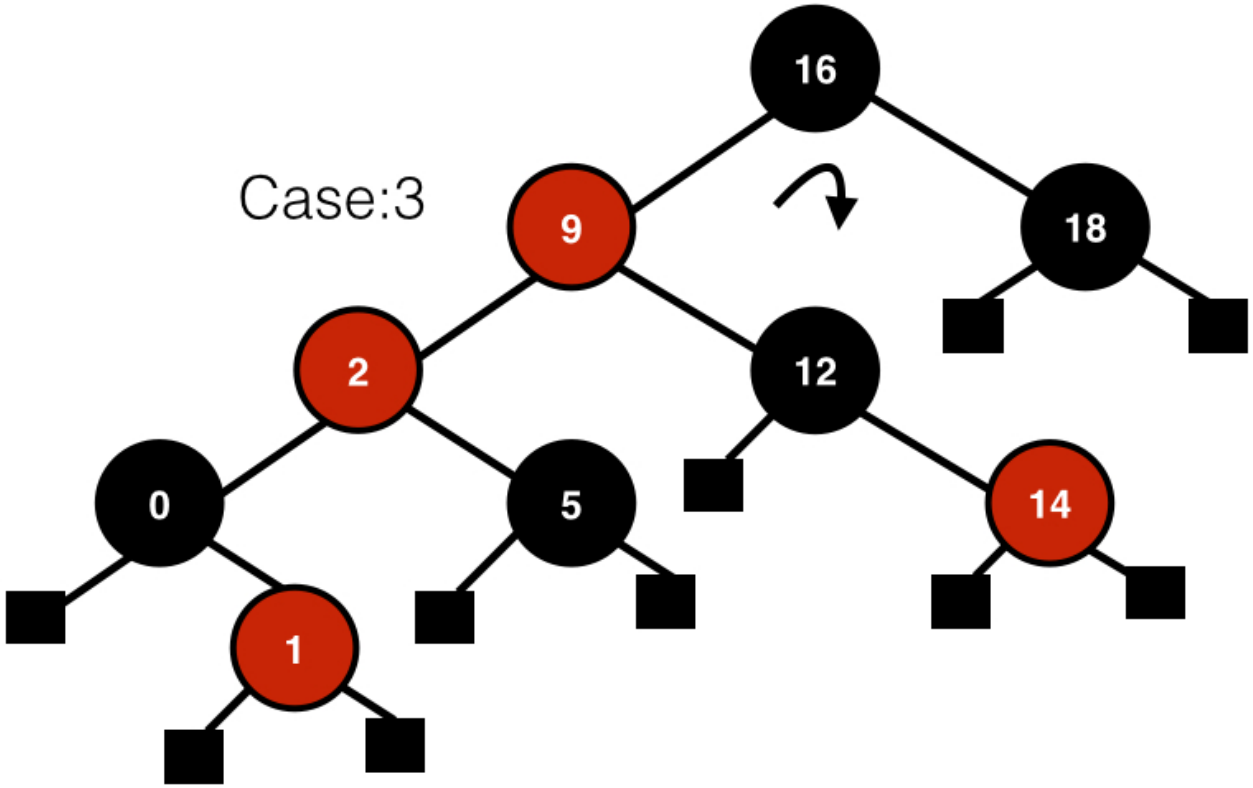
Insertion into a Red-Black Tree



Insertion into a Red-Black Tree



Insertion into a Red-Black Tree



Insertion into a Red-Black Tree

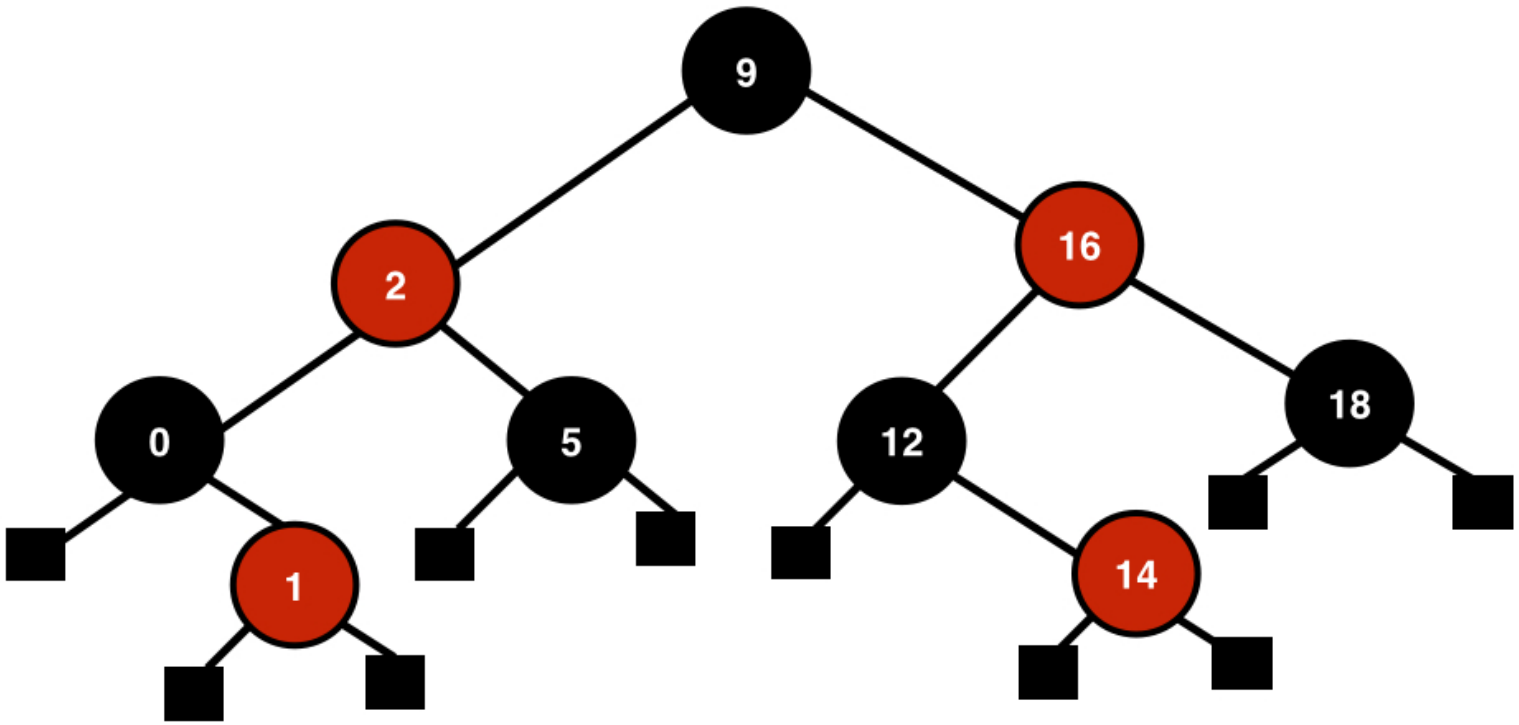


Table of Contents

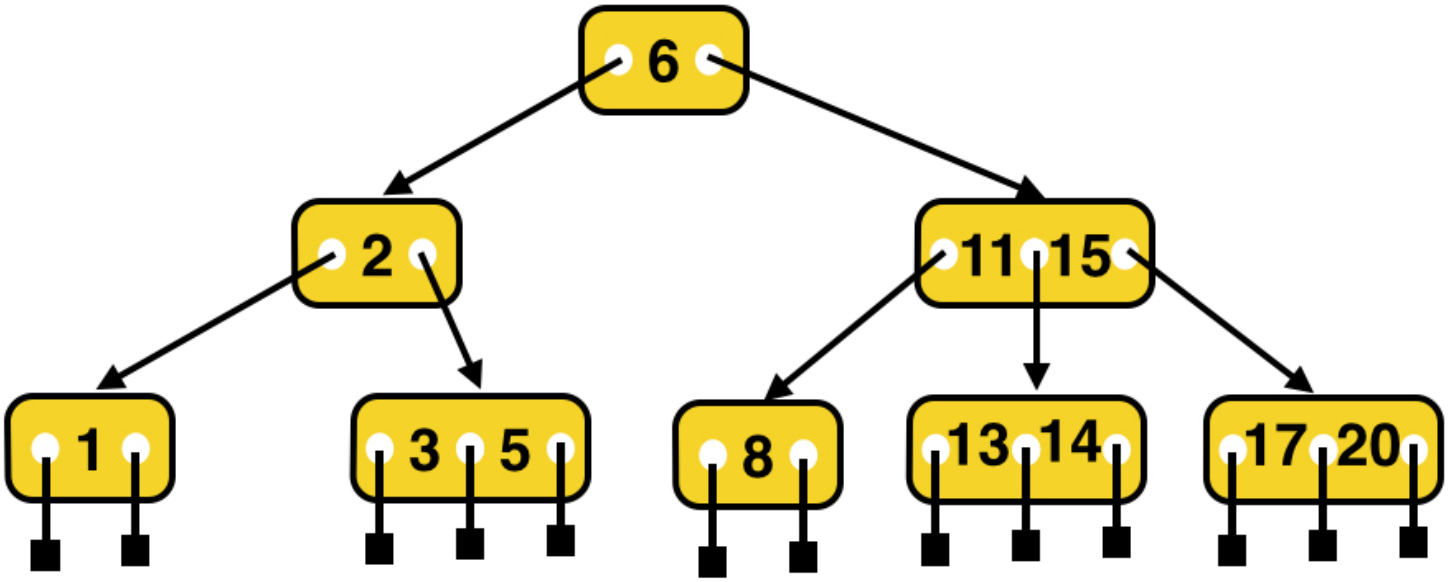
1. Introduction
2. Binary Trees
   - Heaps
   - Binary Search Trees
   - AVL Trees
   - Red-Black Tree
3. 2-3 Trees
4. B-Trees
5. Tries

2-3 Trees

Introduction to 2-3 Trees

- We have looked at **binary trees** so far with various properties
- Heaps, AVL trees and red-black trees impose a **particular tree structure** so that **certain efficiencies** can be obtained
- Particularly, each node is permitted one key and up to two children and the tree height is **almost balanced**
- How about **relaxing** the restriction placed on the number of keys and children each node can have and **enforce perfect balance in height** of all subtrees?
- **2-3 trees** are one such variant of trees which are allowed to have one or two keys in each node and two to three children but all leafs of the tree must lie on the **same level** and there exists an ordering constraint on all keys in the tree

2-3 Trees: An Example



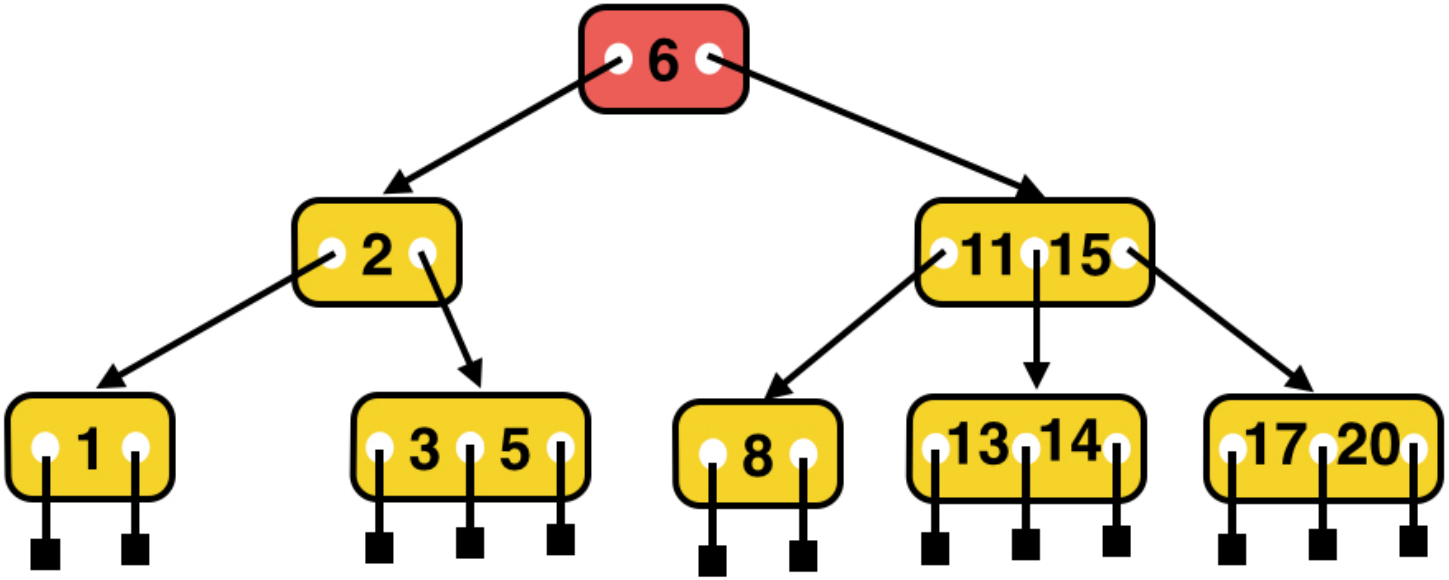
- Each node can have 1 to 2 keys and the number of pointers to children can be 2 or 3

Searching for a Key

2-3 Trees: Searching

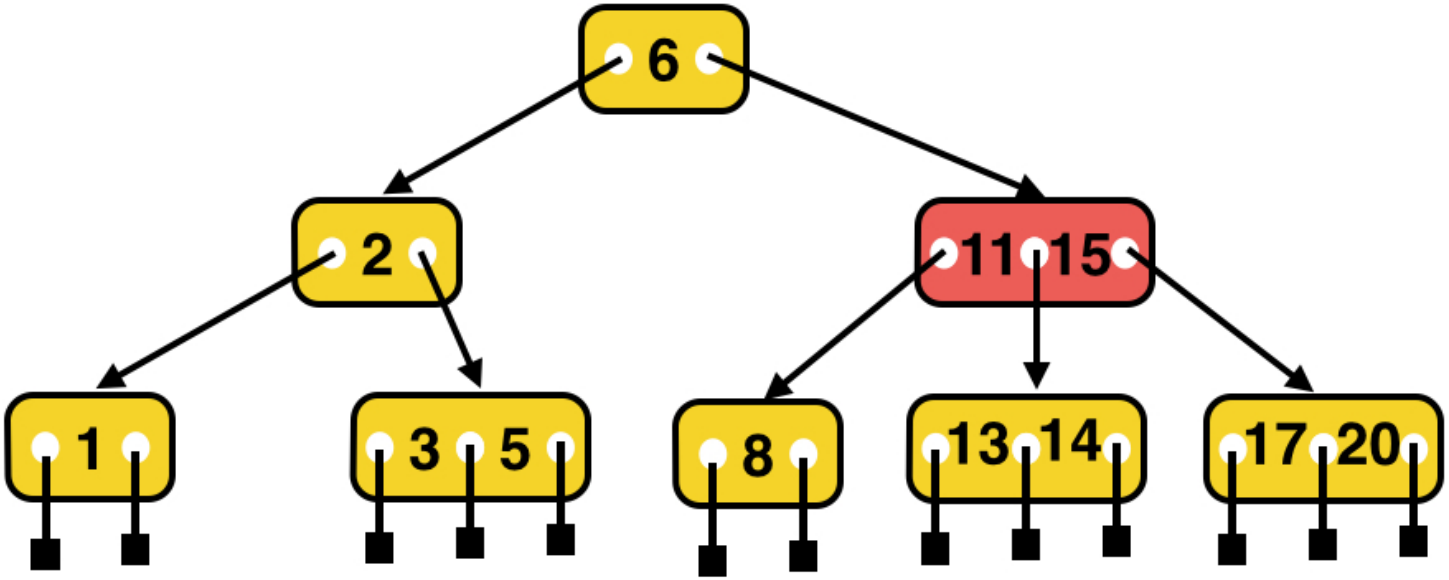
- Starting from the root node, compare the keys in the node with the key being searched
- If there are two keys in the node, check if the key being searched is
  - Less than the left key (follow the left pointer to the next node)
  - In between the left and the right key (follow the middle pointer to the next node)
  - Greater than the right key (follow the right pointer to the next node)
  - Equal to one of the keys in the node (return 1)
  - Reached a NULL node (return 0)

2-3 Trees: Searching



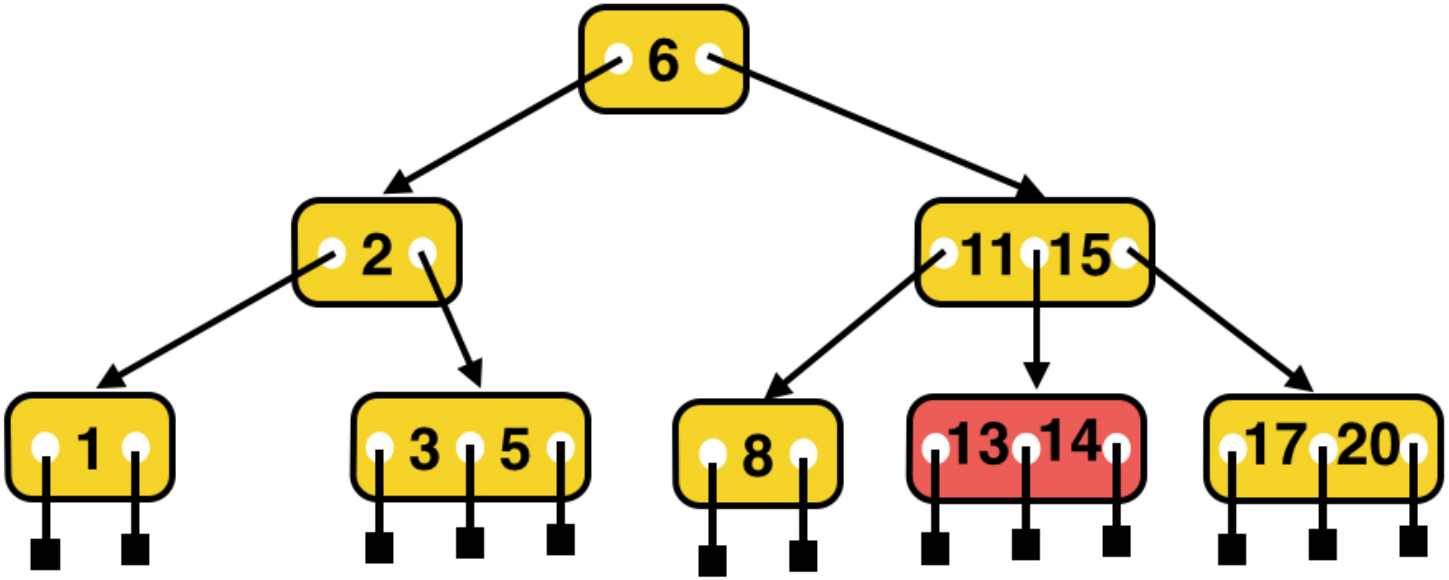
Searching for key 13

2-3 Trees: Searching



Searching for key 13

2-3 Trees: Searching



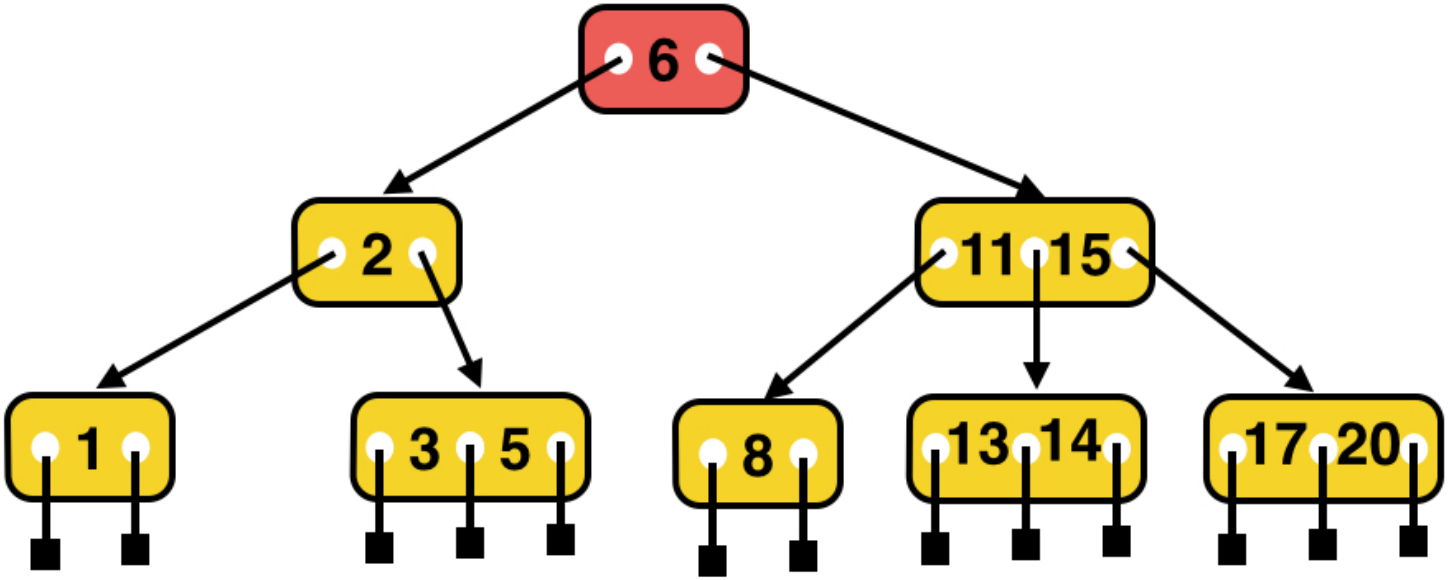
Searching for key 13

Inserting a Key

2-3 Trees

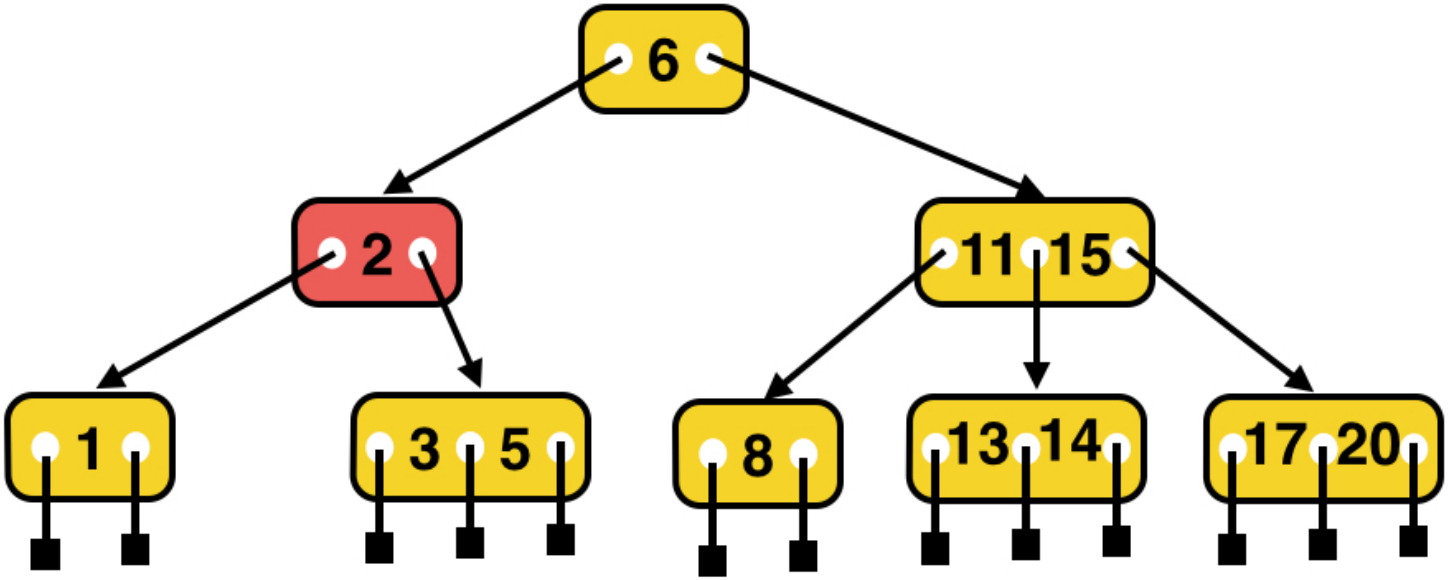
- Starting from the root node, compare the keys in the node with the key being inserted in a manner similar to the **search** operation
- Once a leaf has been reached, the key can be inserted into the leaf node if there is only one key
- Complications arise when the node is full (i.e. the number of keys in the node is 2)
- Nodes are not permitted to overflow and therefore full nodes are split and some keys result in moving up the tree

2-3 Trees: Simple Example



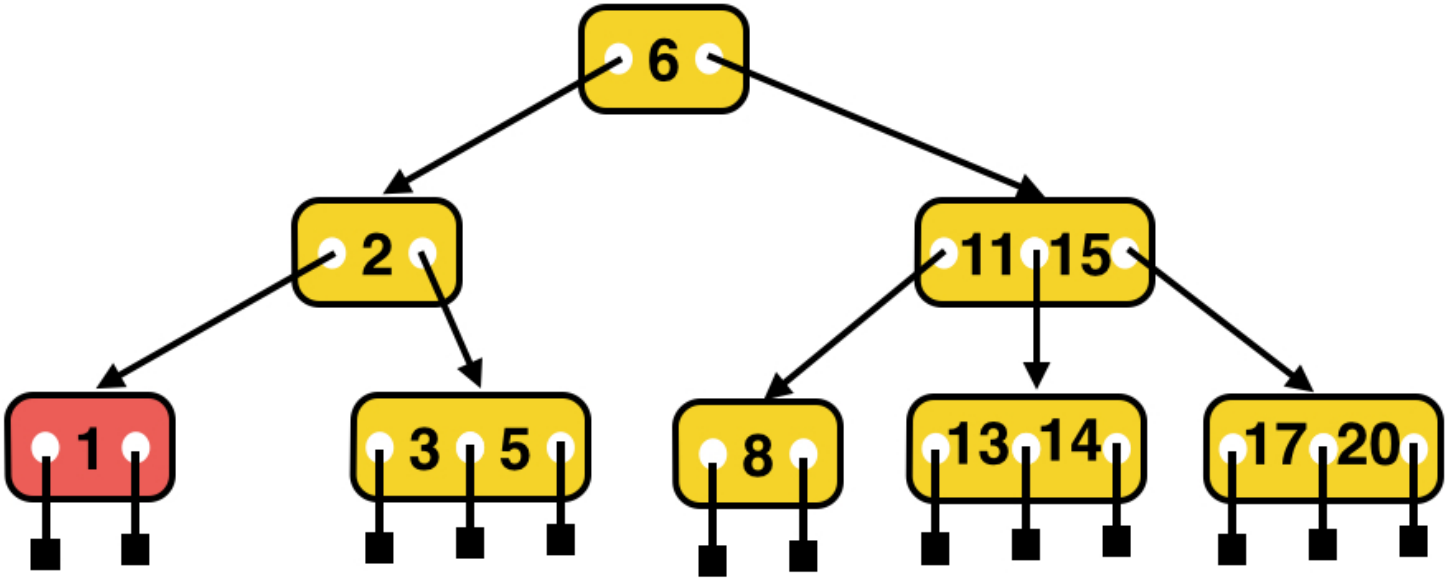
Inserting key 0

2-3 Trees: Simple Example



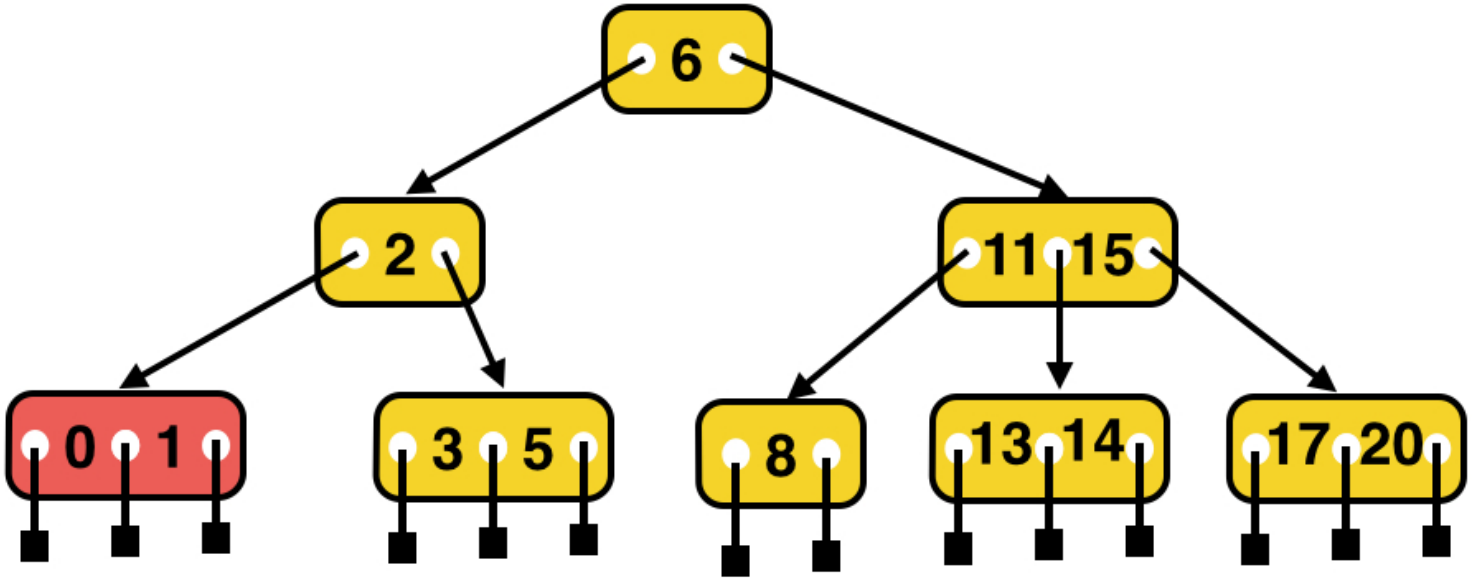
Inserting key 0

2-3 Trees: Simple Example



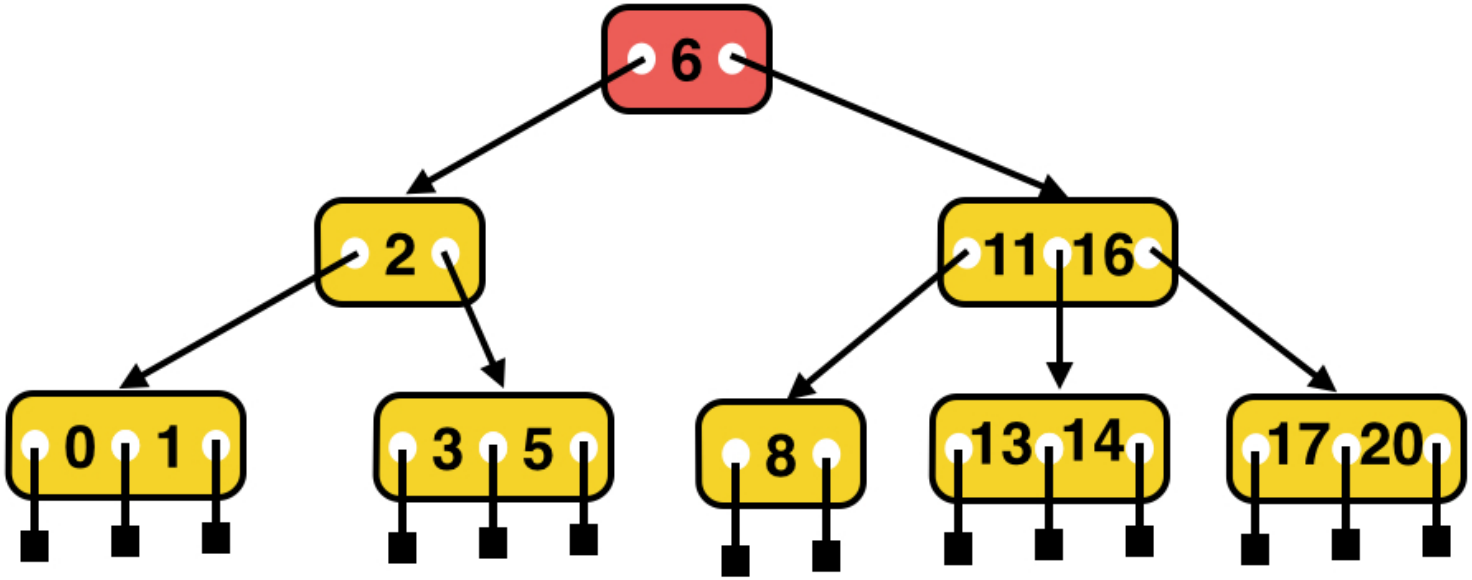
Inserting key 0

2-3 Trees: Simple Example



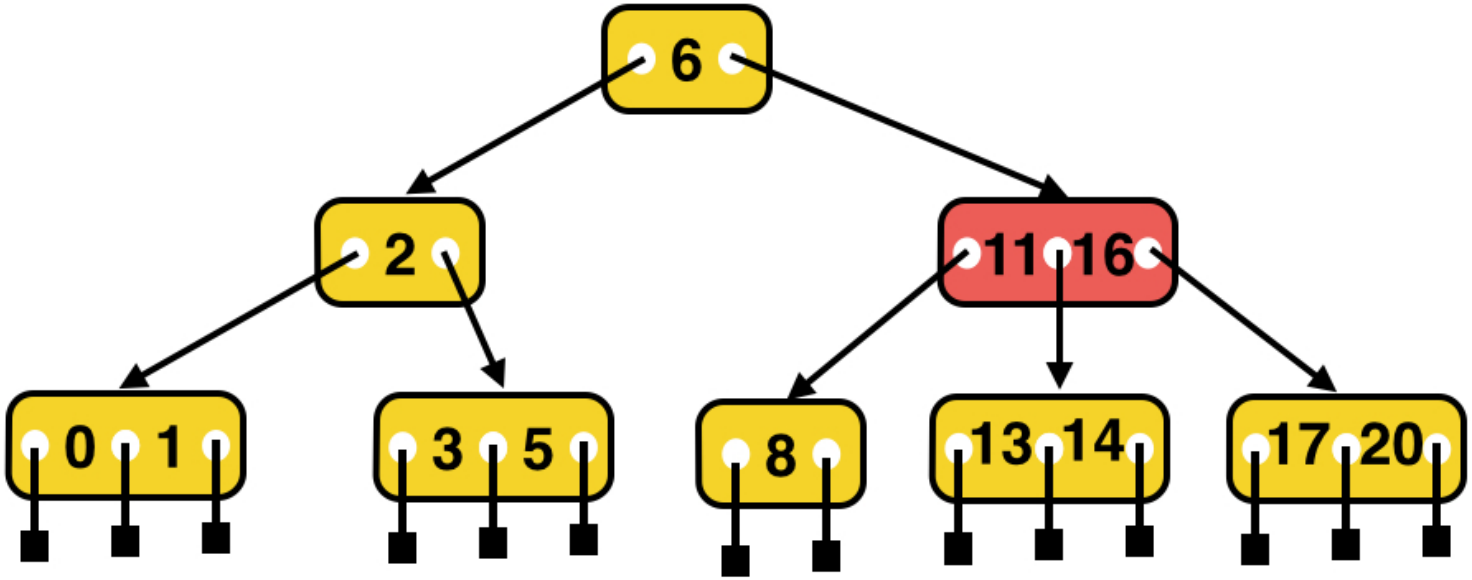
Inserting key 0

2-3 Trees: Example with Overflows



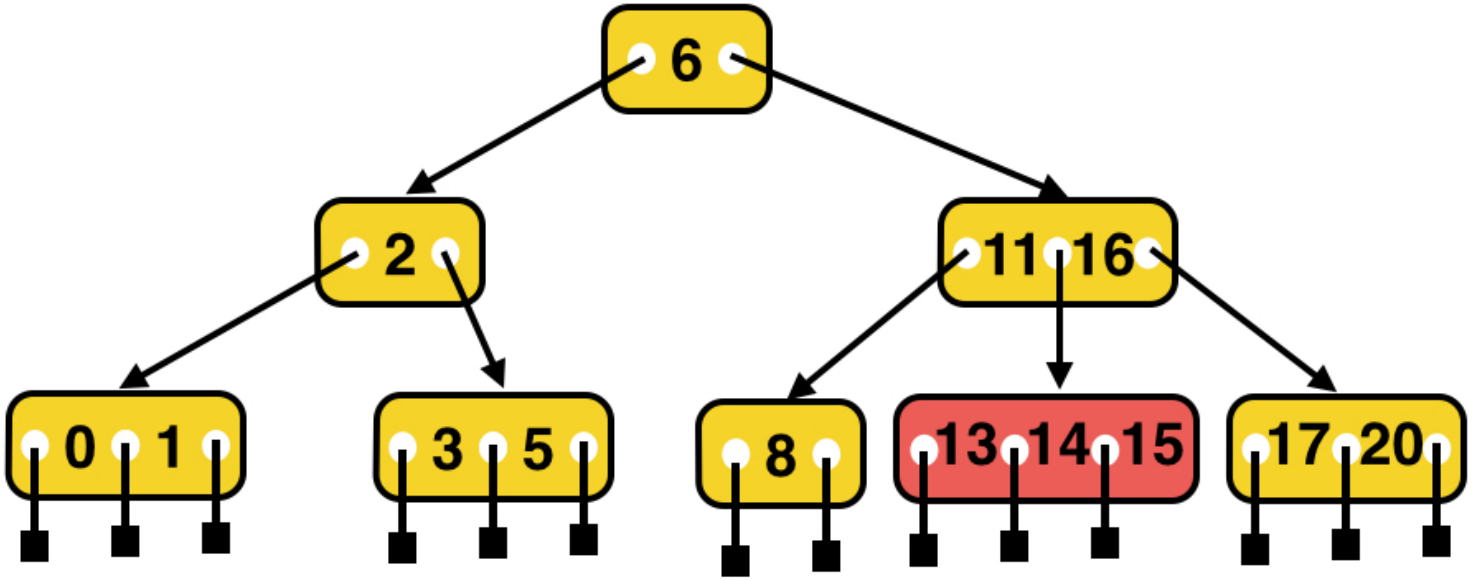
Inserting key 15

2-3 Trees: Example with Overflows



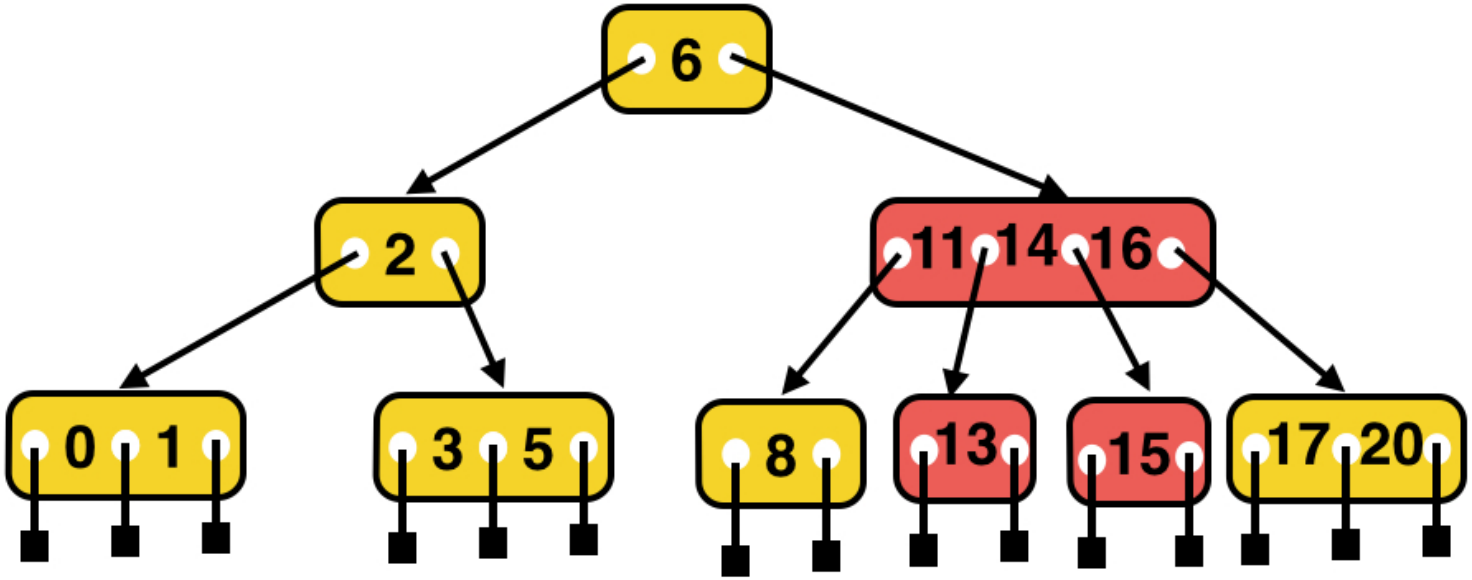
Inserting key 15

2-3 Trees: Example with Overflows



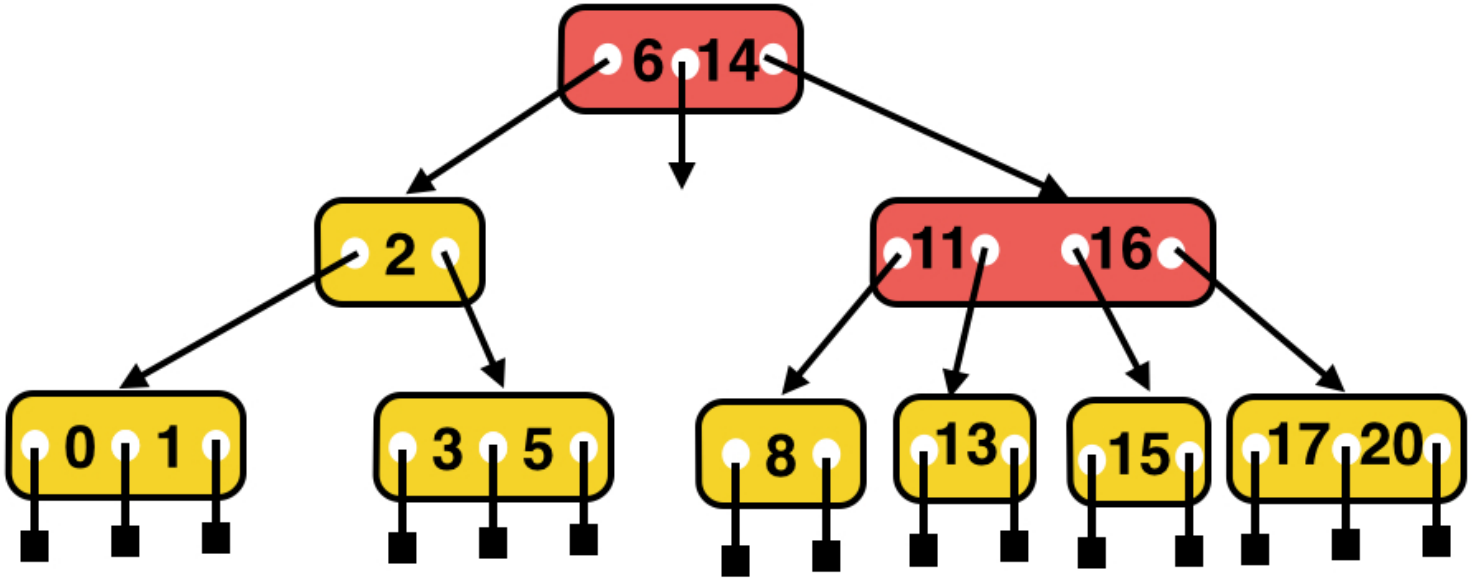
Inserting key 15

2-3 Trees: Example with Overflows



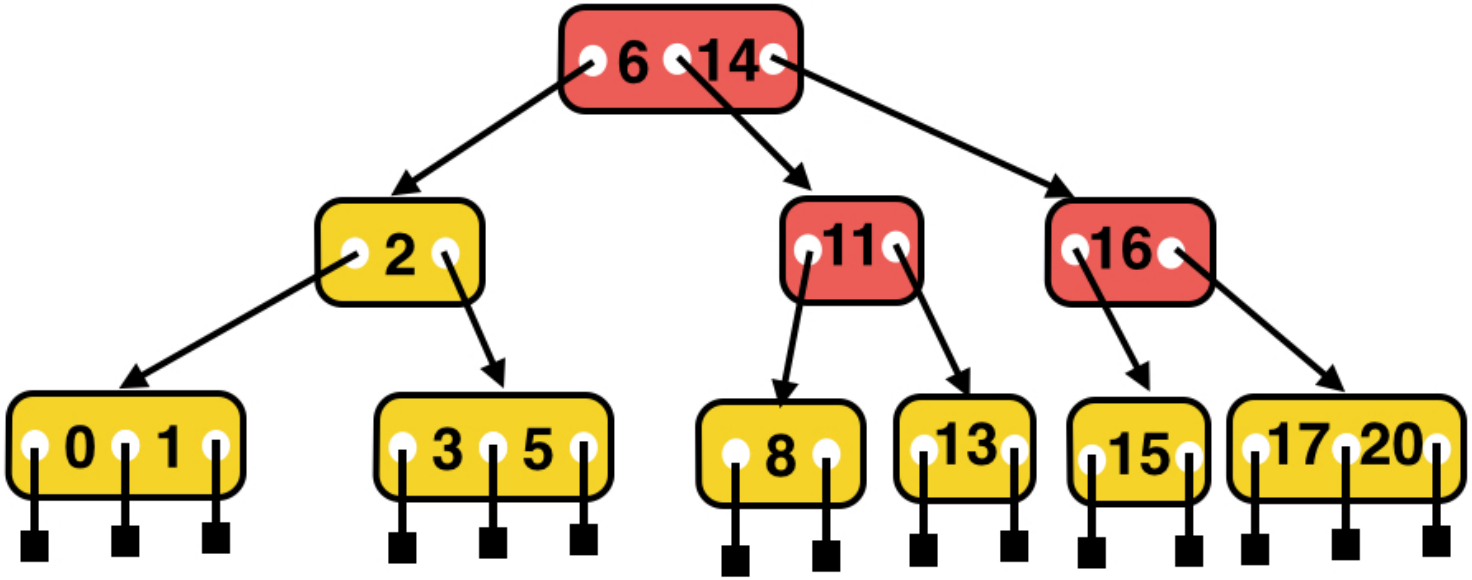
Inserting key 15

2-3 Trees: Example with Overflows



Inserting key 15

2-3 Trees: Example with Overflows



Inserting key 15

2-3 Trees: Performance

- The asymptotic complexity of an operation on a tree is **dependent on the height** h in the tree
- In the worst case, the 2-3 tree is a full and complete binary tree (i.e. $n = 2^{h+1} - 1$) and therefore $h = log(n+1) - 1$
- Hence, the search operation is O(logn)
- For insertion operations, suppose a node overflows at level k of the tree
- In the worst case, number of splits that can occur is $k + 1$ and again the complexity is dependent on the height of the tree which is O(logn)
- Similar analysis can be made for deletion

Table of Contents

1. Introduction
2. Binary Trees
   - Heaps
   - Binary Search Trees
   - AVL Trees
   - Red-Black Tree
3. 2-3 Trees
4. B-Trees
5. Tries

B-Trees

B-Trees: Introduction

- B-trees are a **generalization** of the 2-3 trees
- Each node in the tree is **assigned** an order m
- A node can have at most $m - 1$ keys and between $\lceil m/2 \rceil$ to m children (m is also called the **branching factor**)
- A root node is permitted to have 2 to m children
- All leaf nodes reside at **same level**

B-Trees: Complexity

- Suppose a B-tree is full (i.e. every node has m children)
- If there are p nodes and n keys in the tree, then $p = n/(m-1)$
- Given that the height of the B-tree is h, then $p = 1 + m + m^2 + \ldots + m^h$
- This is a geometric progression and therefore $p = (m^{h+1} - 1)/(m-1)$
- Combining $p = (m^{h+1} - 1)/(m-1)$ and $p = n/(m-1)$, you can obtain $h = log_m(n+1) - 1$

B-Trees: Complexity

- Suppose $m = 256$ and $n = 16777215$, then $h = log_{256}(16777215) - 1 = 2$ (in the worst case, three nodes may need to be accessed to search for a key)
- B-Trees are useful for dealing with memory
- Accessing memory from the hard drive is costly in terms of time and suppose that there isn't sufficient memory to store a large number of records in RAM
- With a B-tree with order 256, only three hard disk swaps are necessary for the provided value of n
- In RAM, binary trees can be used to process each node to locate the key being searched

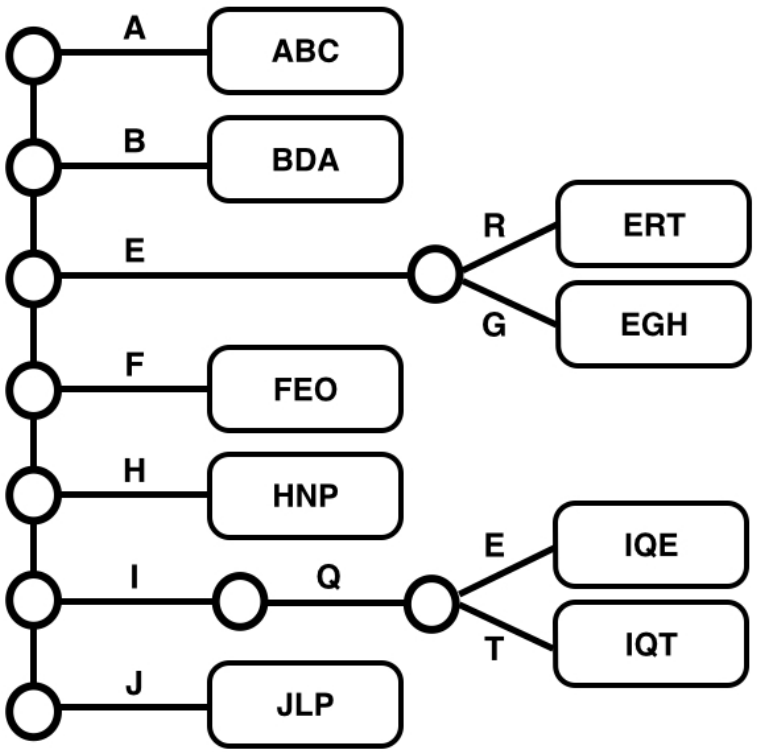
Table of Contents

1. Introduction
2. Binary Trees
   - Heaps
   - Binary Search Trees
   - AVL Trees
   - Red-Black Tree
3. 2-3 Trees
4. B-Trees
5. Tries

Tries: Introduction

- The data structure trie stems from the word re**trie**val
- A trie is used to typically store and efficiently retrieve words
- A trie consists of a **discrimination net**
- The discrimination net consists of one node for each first letter of the words stored
- All words having the same first letter are grouped together and these nodes are separated based on the second letter and so on

Tries: Example



- If the keys stored are random, then the number of characters examined is around $log_m N$