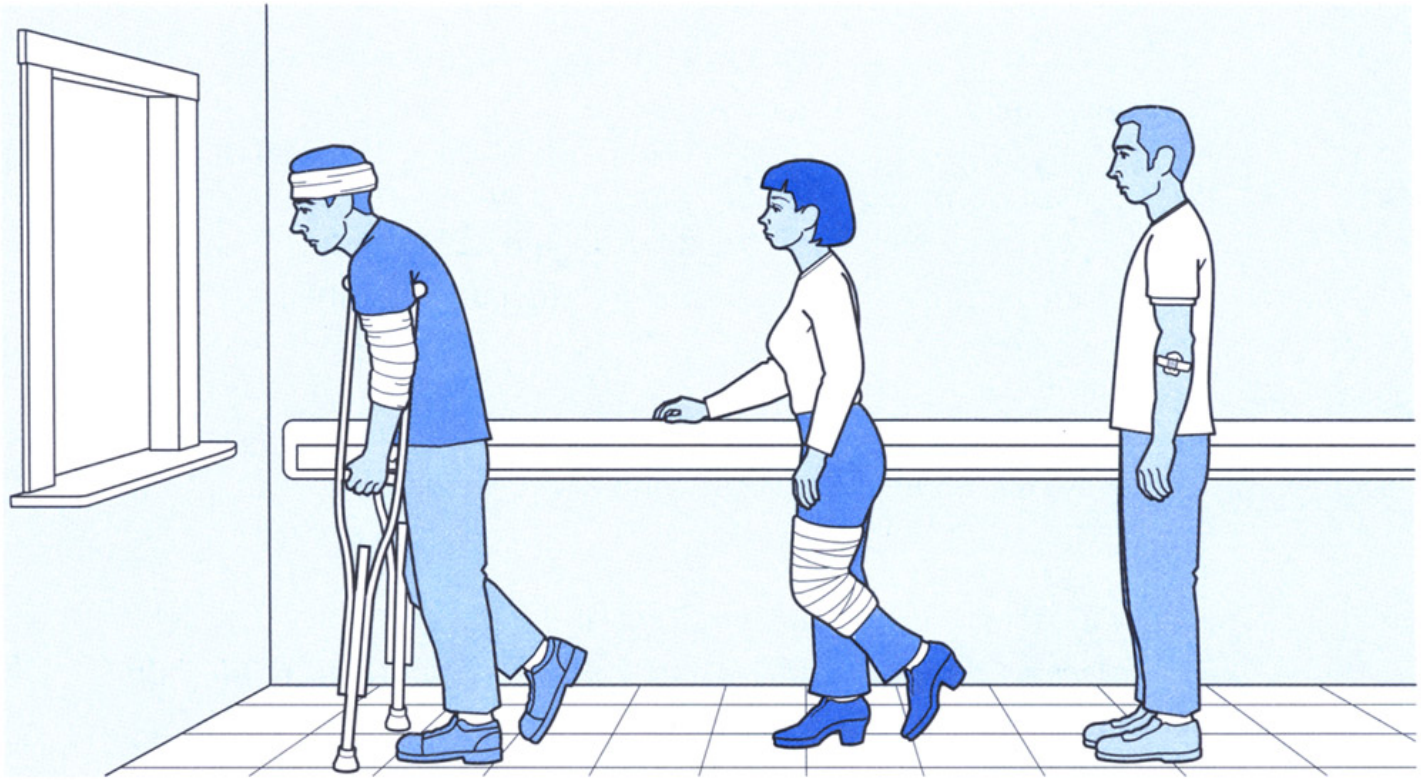
Algorithms Design and Analysis

Day 2
Tree-Based Data Structures

2015, AUT-CJLU

Part I: Priority Queues

Priority Queues



"FITPO": First-In-TopPriority-Out

Priority Queues

Priority Queues

A priority queue is a data structure that maintain a collection of objects taken from an ordered domain (`Comparable`) and supports the following operations

- Query operation:

  - getFirst(): Return the least value (with top priority).

- Update operations:

  - add(E v): Add a value to the queue.

  - remove(): Return and remove the least value.

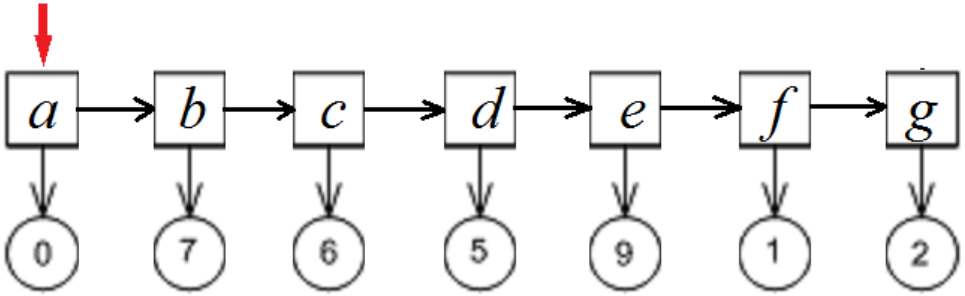
Priority Queues: Interface

The Priority Queue Interface

```
public interface PriorityQueue<E extends Comparable<E>>{
    public E getFirst();
    // pre:  priority queue not empty
    // post:  returns the minimum value
    public E remove();
    // pre:  priority queue not empty
    // post:  returns and removes the minimum value
    public void add(E value);
    // pre:  value is non-null
    // post:  value added to the priority queue
}
```

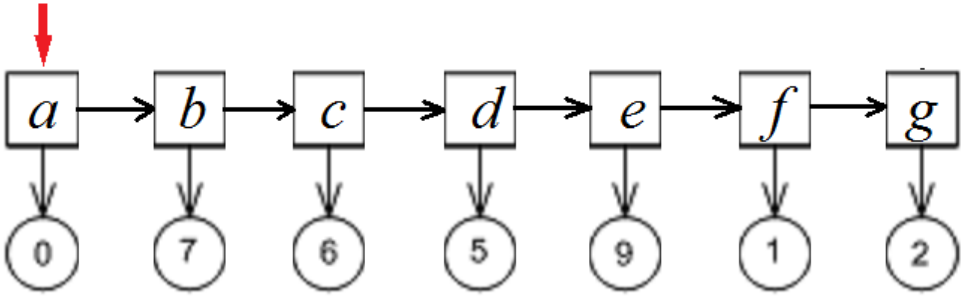
Implementation 1: List-Based Priority Queue

Idea: Use a list to store all elements.
The first element has the minimum value.



Implementation 1: List-Based Priority Queue

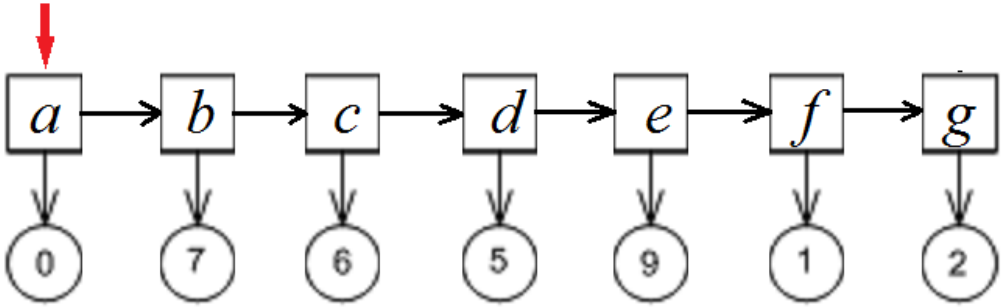
Idea: Use a list to store all elements.
The first element has the minimum value.




```
protected List<E> data;
public ListPriorityQueue(){
    data=new LinkedList<E>();
}
public E getFirst(){ return data.get(0); }
public void add(E v){
    if (v.compareTo(getFirst())<0) data.add(0,v);
    else data.add(1,v);
}
```

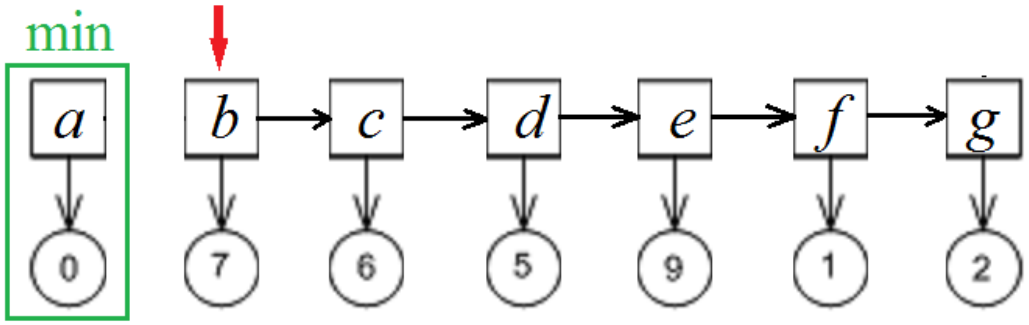
Implementation 1: List-Based Priority Queue

Remove()



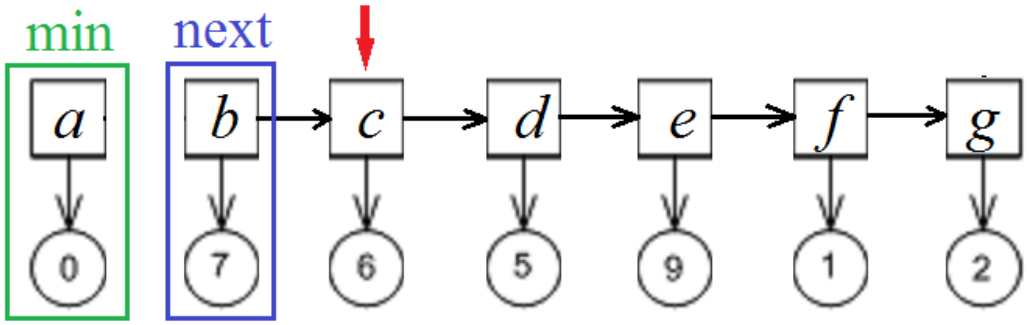
Implementation 1: List-Based Priority Queue

Remove()



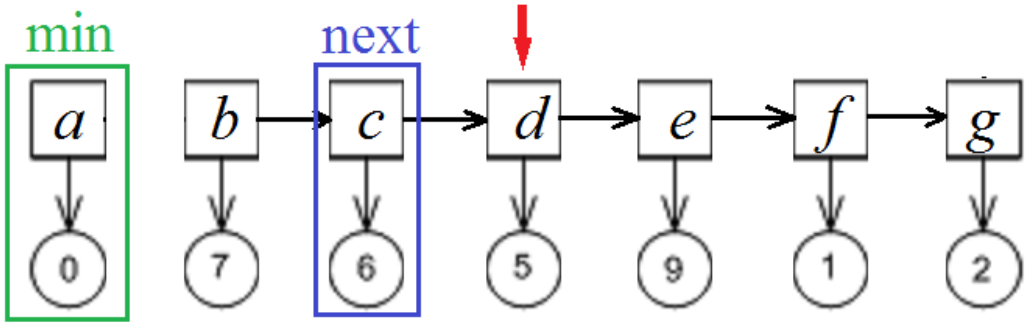
Implementation 1: List-Based Priority Queue

Remove()



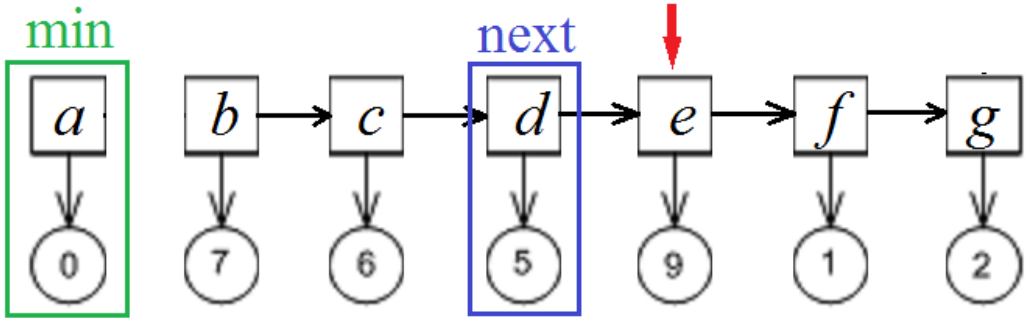
Implementation 1: List-Based Priority Queue

Remove()



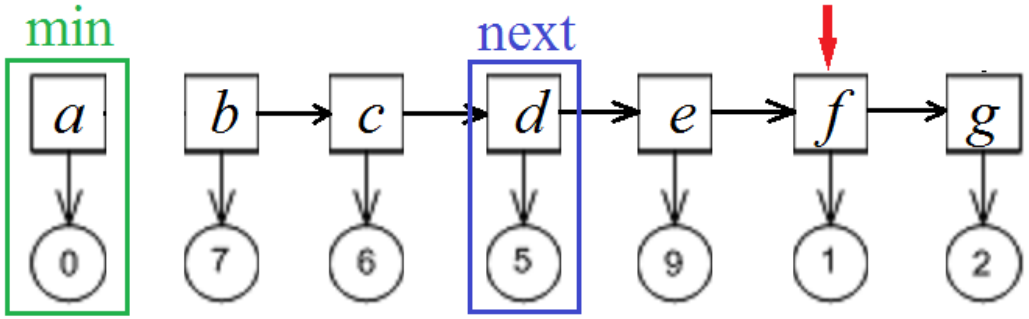
Implementation 1: List-Based Priority Queue

Remove()



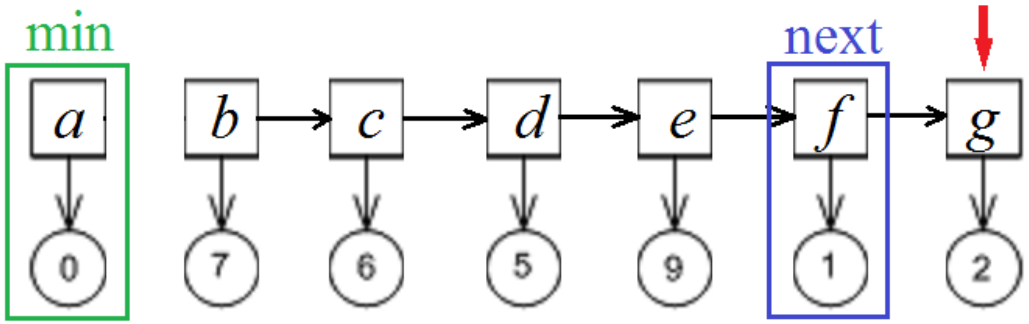
Implementation 1: List-Based Priority Queue

Remove()



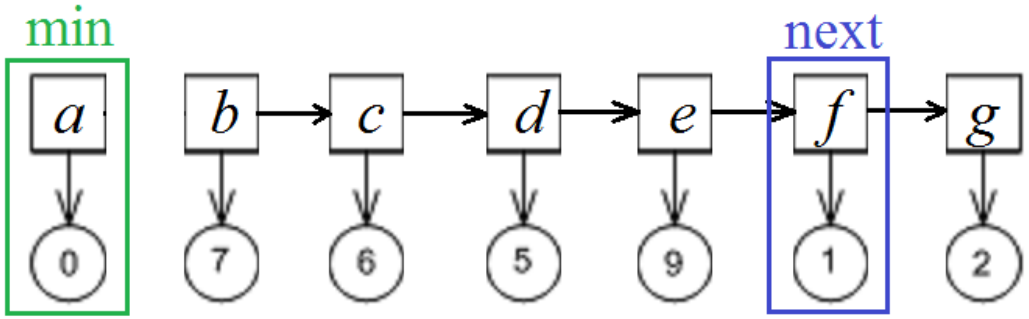
Implementation 1: List-Based Priority Queue

Remove()

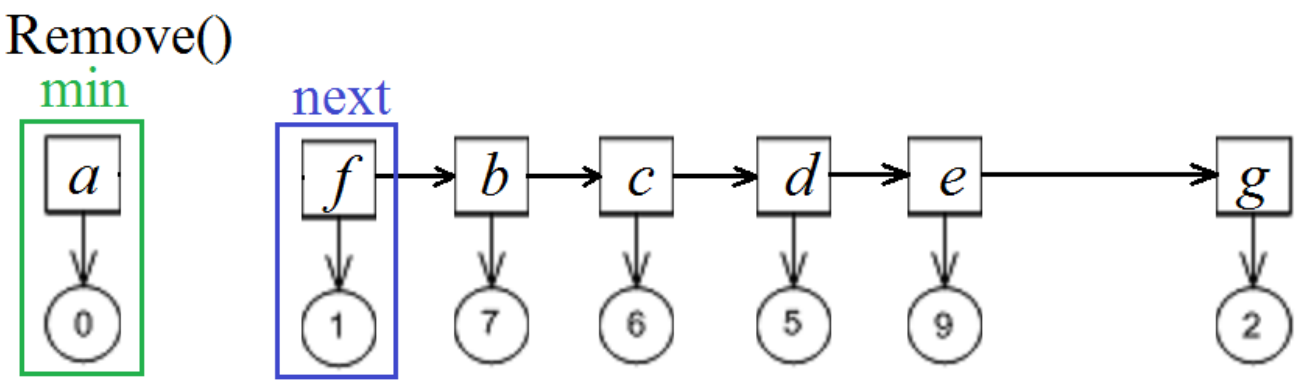


Implementation 1: List-Based Priority Queue

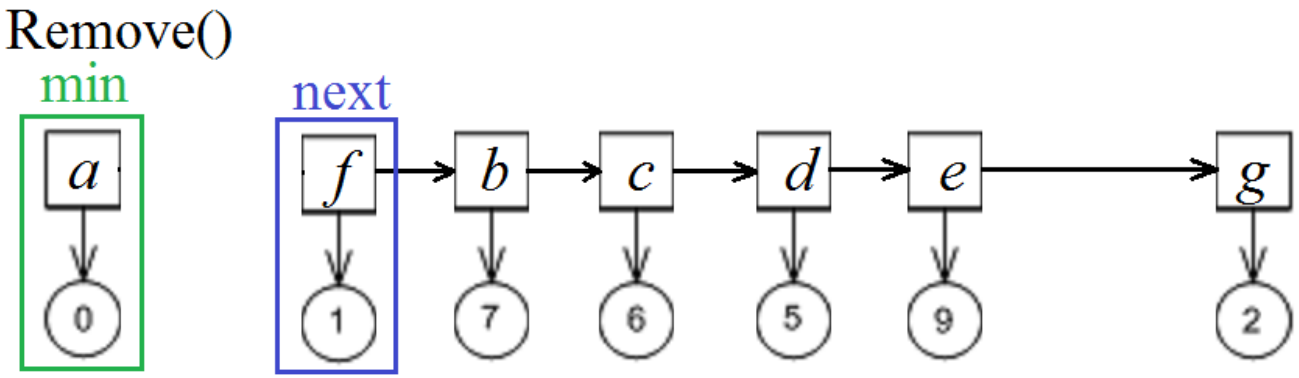
Remove()



Implementation 1: List-Based Priority Queue



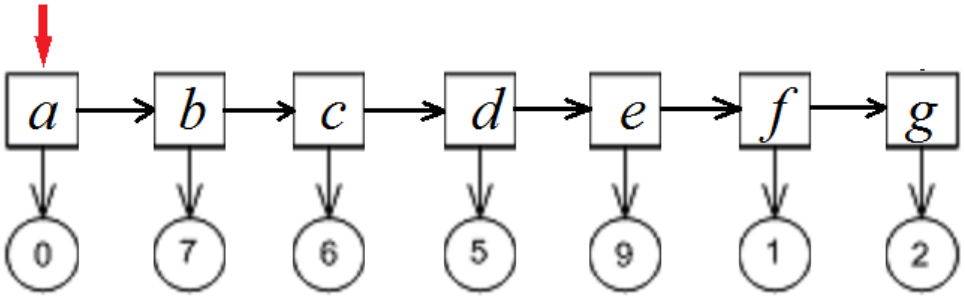
Implementation 1: List-Based Priority Queue




```
public E remove(){
    E min=data.remove(0);
    int current=0;
    for (int i=0; i<data.size(); i++){
        if(data.get(i).compareTo(data.get(current))<0)
            current=i;
    }
    E next=data.get(i);
    data.remove(i);
    data.addFirst(next);
    return min;
}
```

Implementation 1: List-Based Priority Queue

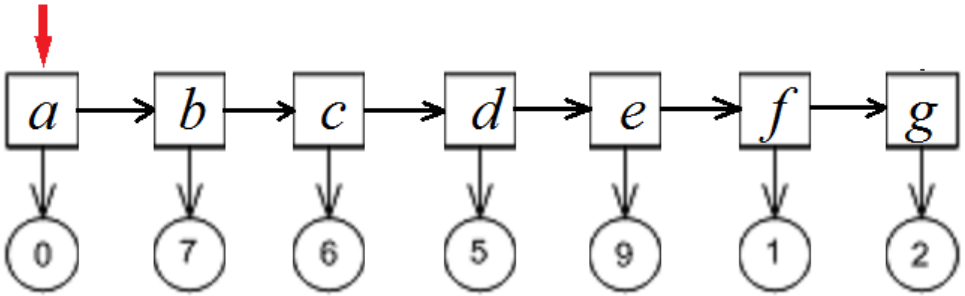
Question: What is the time complexity of each operation?



- getFirst():
- add():
- remove():

Implementation 1: List-Based Priority Queue

Question: What is the time complexity of each operation?

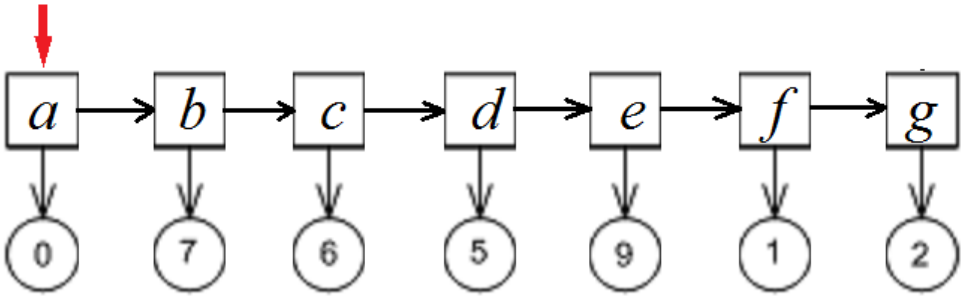


- getFirst(): $O(1)$
- add(): $O(1)$
- remove(): $O(n)$

  As we need to iterate over all elements in the list to search for the next smallest value.

Implementation 1: List-Based Priority Queue

Question: What is the time complexity of each operation?



- getFirst(): $O(1)$
- add(): $O(1)$
- remove(): $O(n)$

  As we need to iterate over all elements in the list to search for the next smallest value.

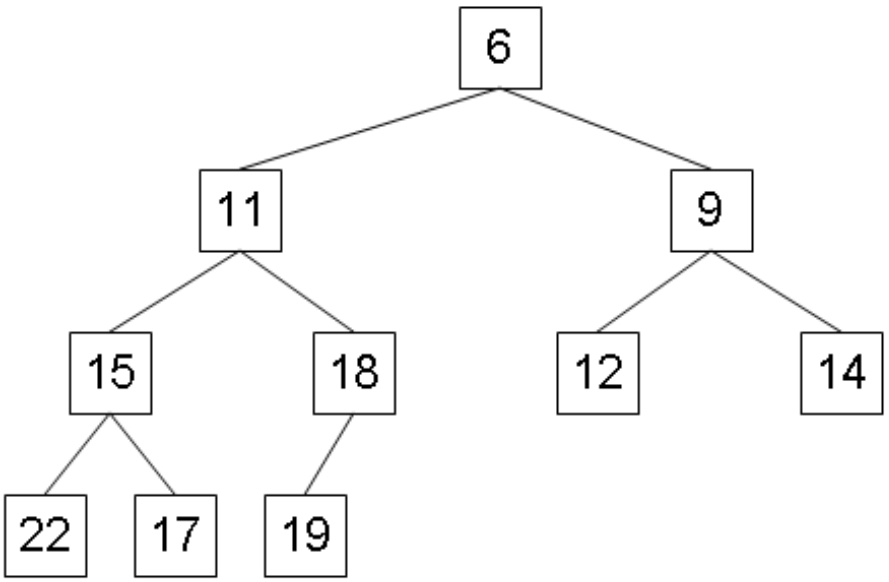
  Next Question. Is there a better way?

Implementation: Tree-Based Priority Queues

Idea: Put elements in a binary tree.
The root stores the least value.

Heap

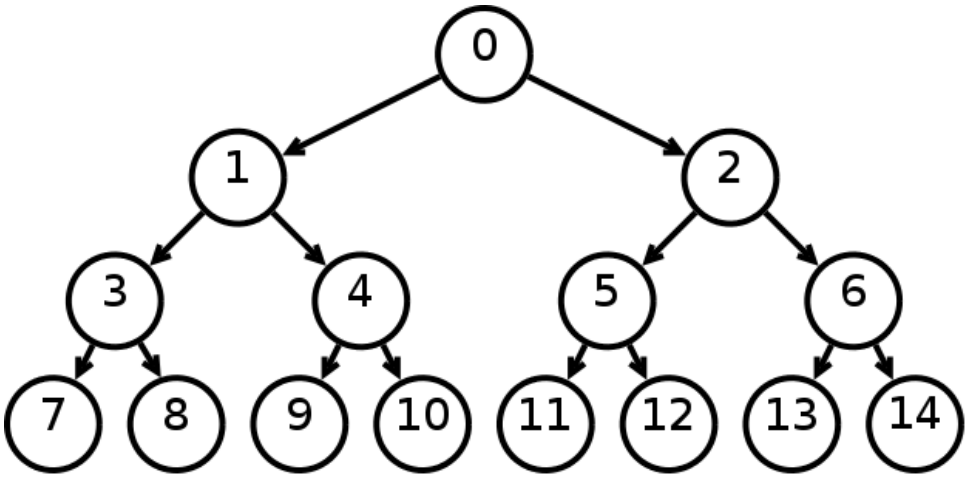
A heap is a tree data structure that implements the priority queue interface.



Complete Binary Trees

Complete Binary Trees

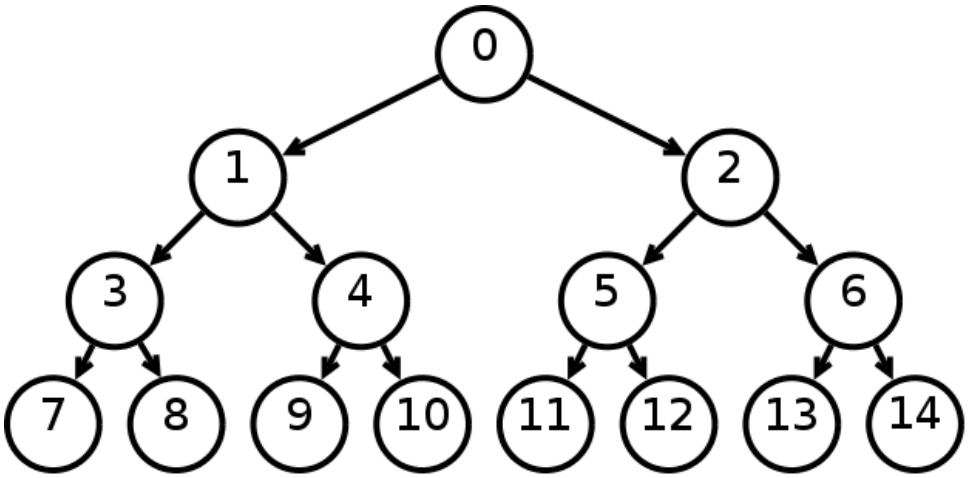
- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.



Complete Binary Trees

Complete Binary Trees

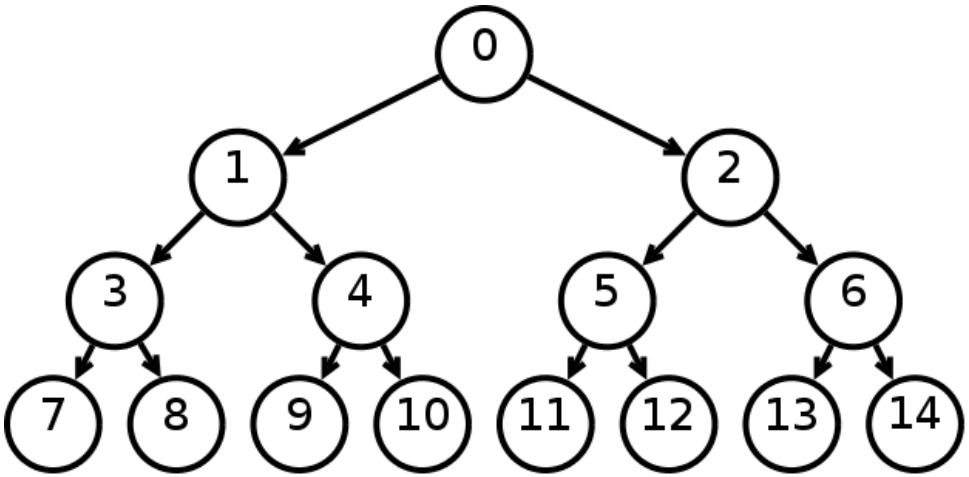
- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.



Complete Binary Trees

Complete Binary Trees

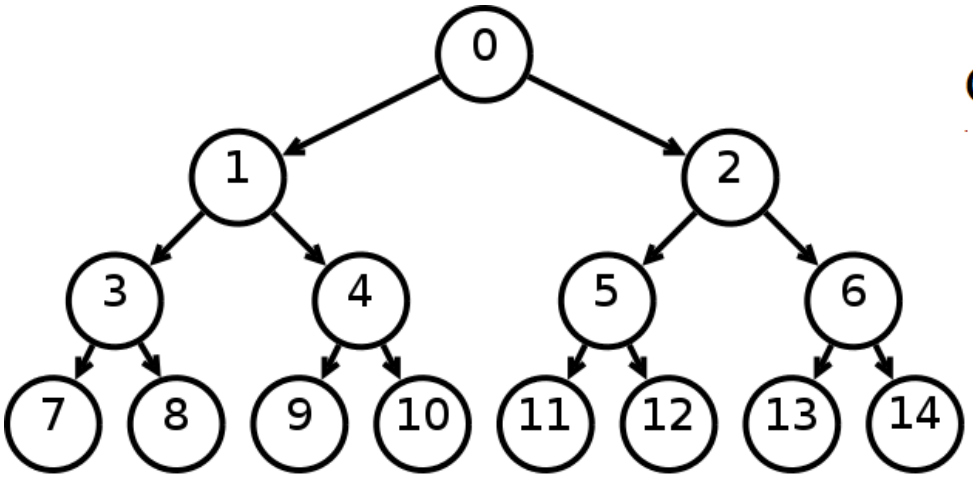
- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.



Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.

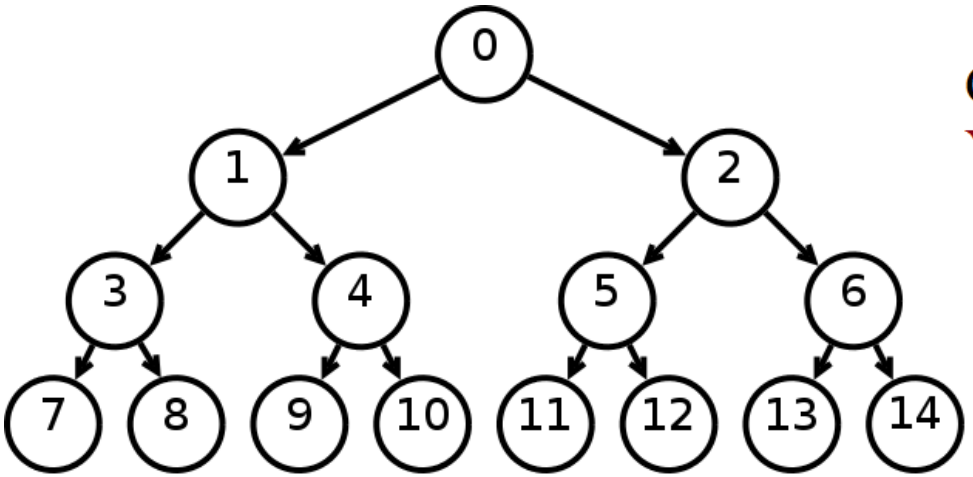


Complete?

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.

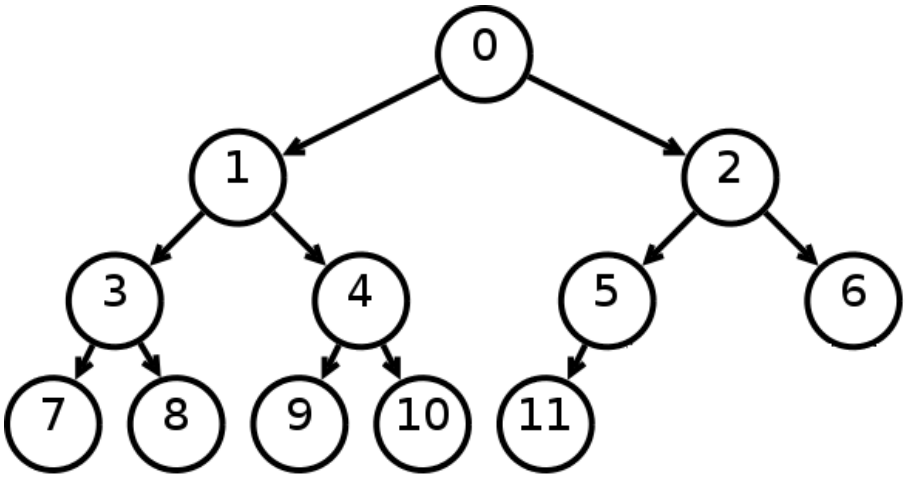


Complete?
Yes

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.

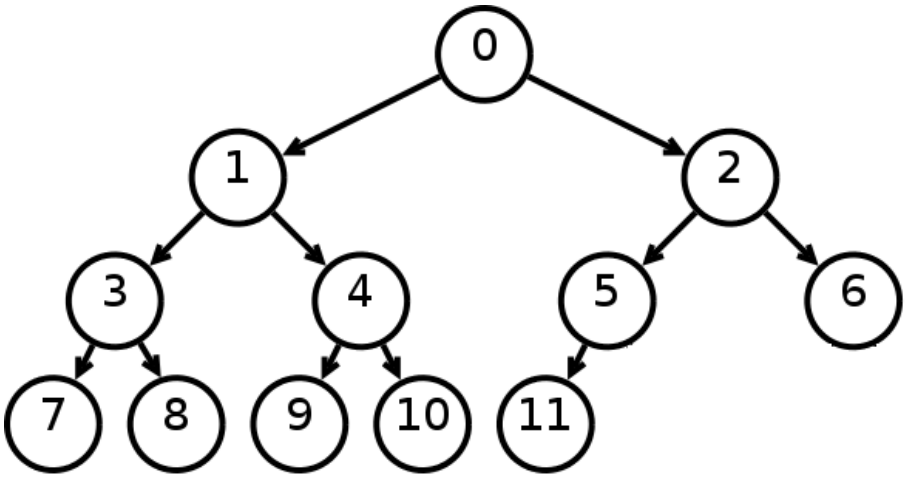


Complete?

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.



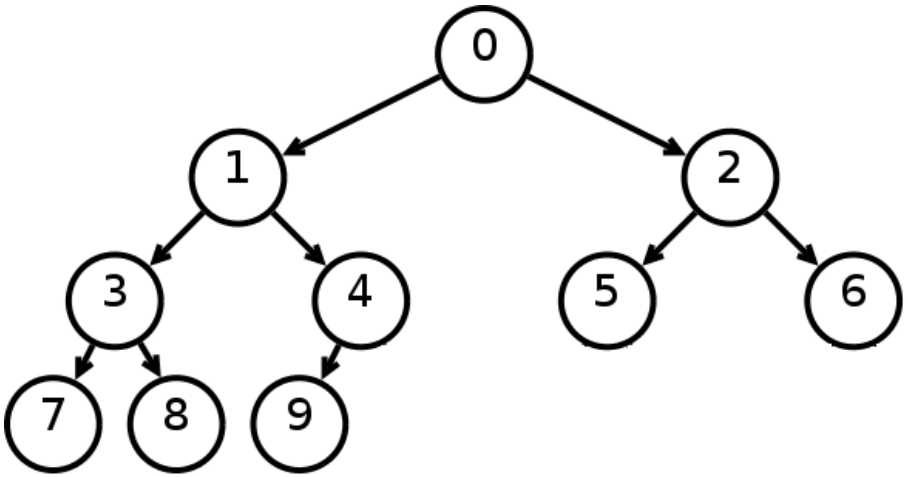
Complete?

Yes.

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.

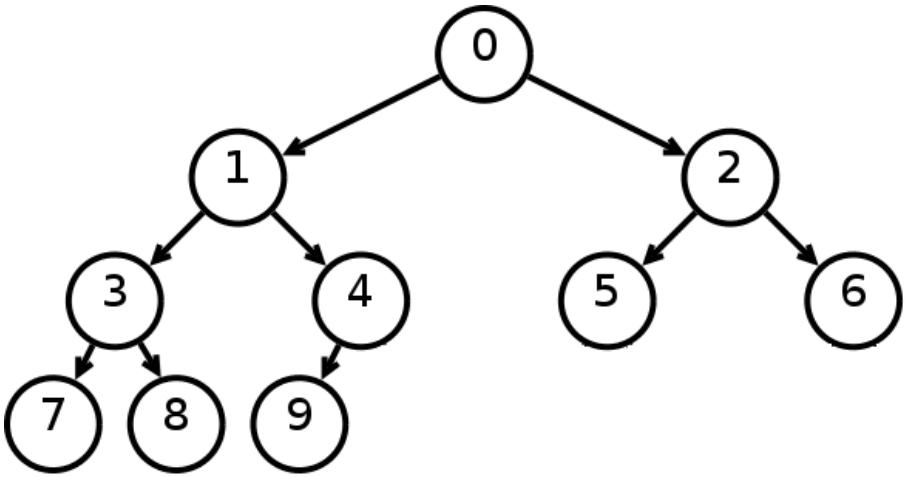


Complete?

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.



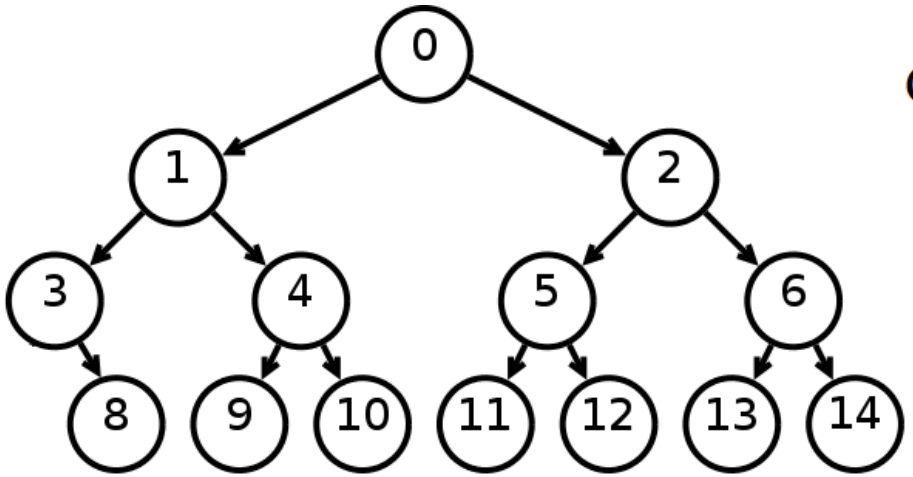
Complete?

Yes.

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.

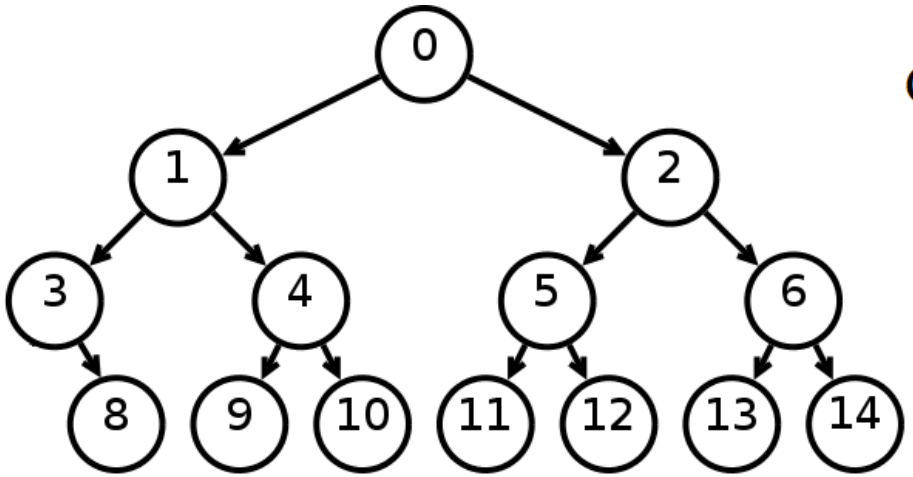


Complete?

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.



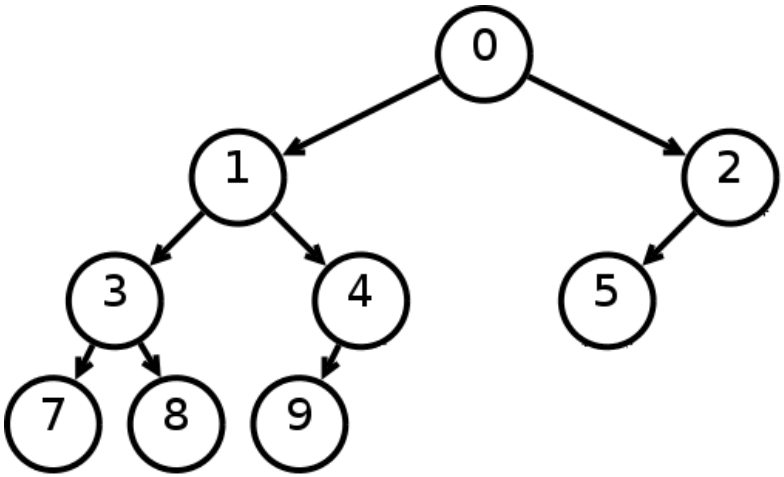
Complete?

No.

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.

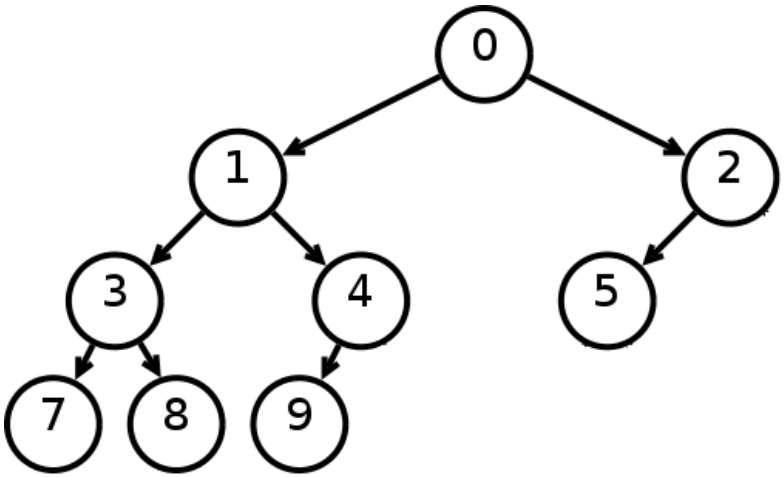


Complete?

Complete Binary Trees

Complete Binary Trees

- In a perfect binary tree of height h, the ith level ($i \leq h$) has exactly 2^i nodes.
- We say that the ith level of a binary tree has been filled if it contains 2^i nodes.
- A complete binary tree is a special type of binary tree such that
  - Every non-empty level is filled except possibly the last level.
  - All nodes in the last level are as far left as possible.



Complete?

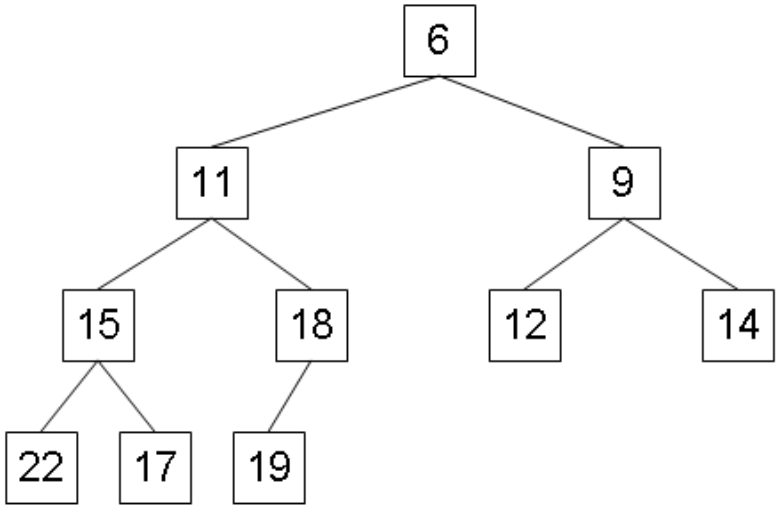
No.

Heaps

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.

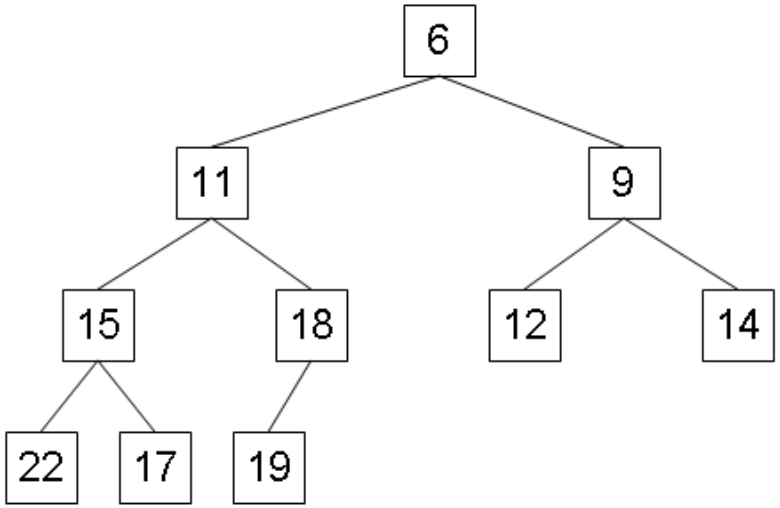


Heaps

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.

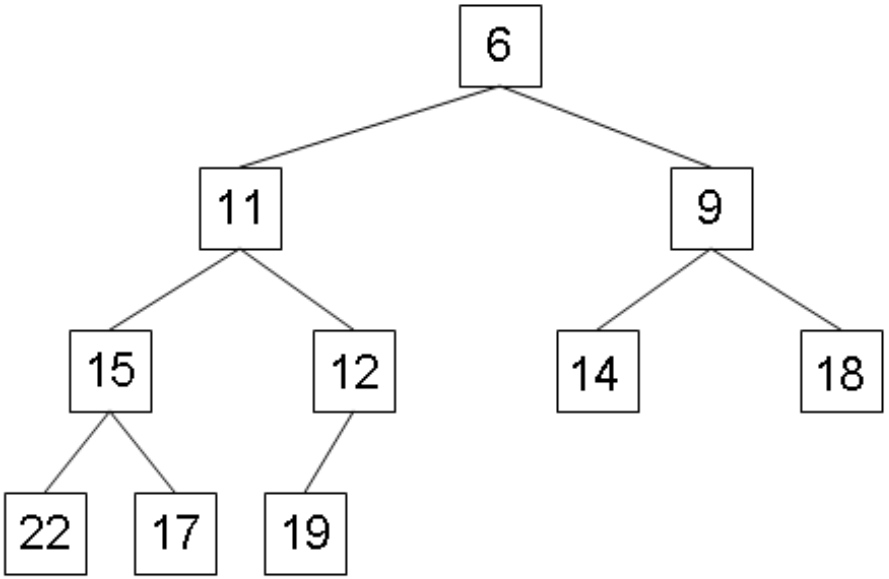


Note: Sometimes these heaps are called min heaps; in max heaps values of parents are bigger than values of children.

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.

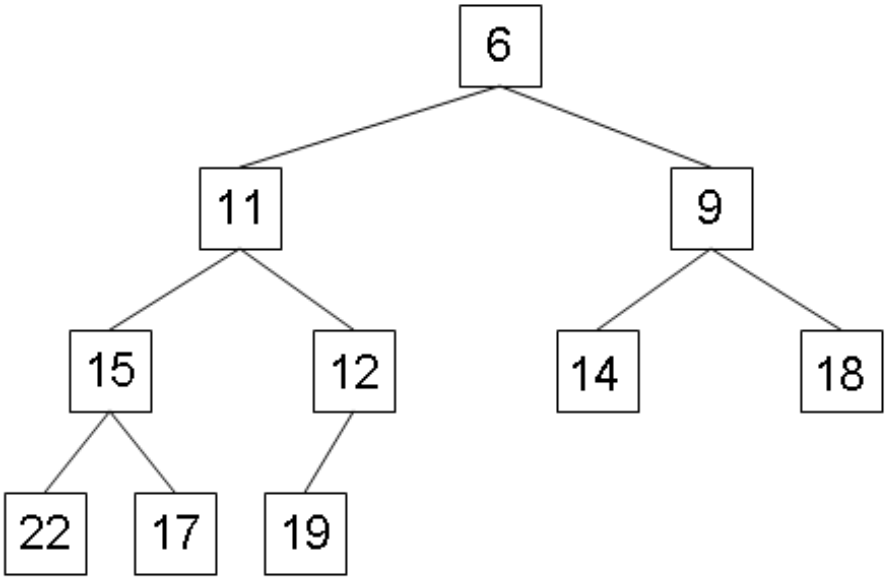


Heap?

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.



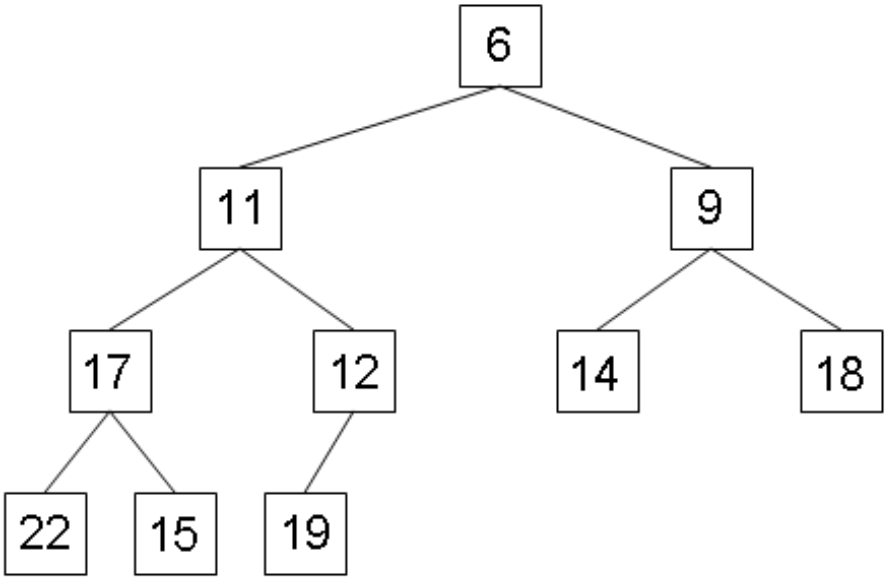
Heap?

Yes

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.

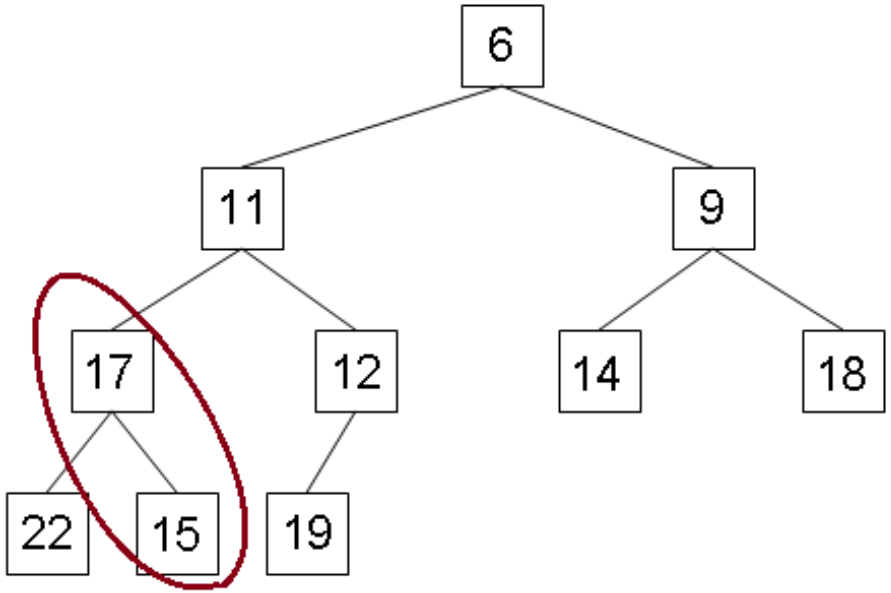


Heap?

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.



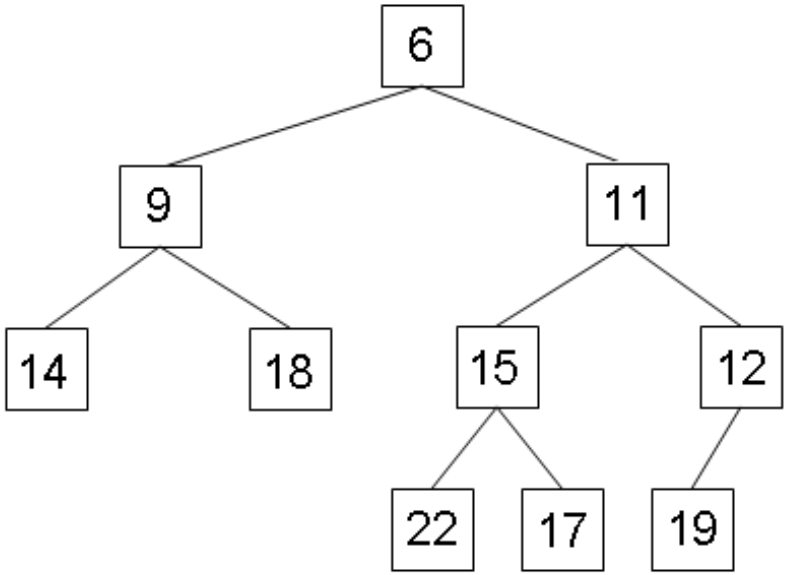
Heap?

No.

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.

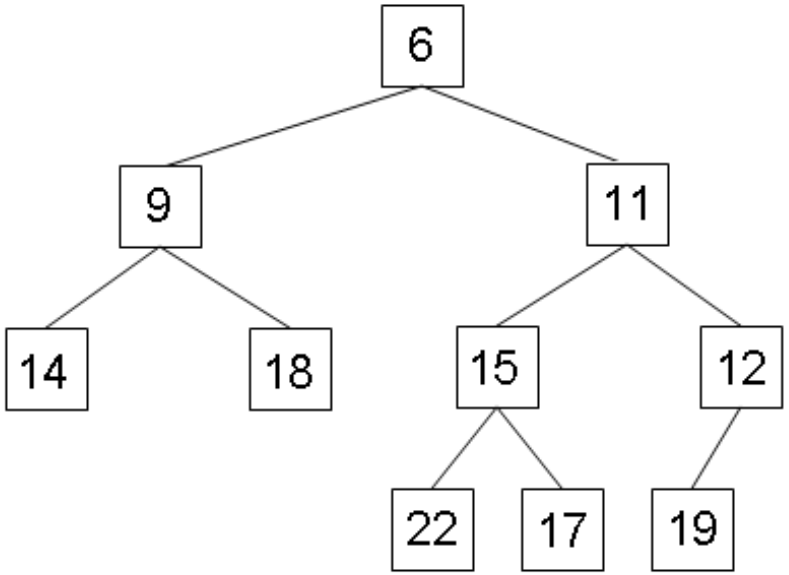


Heap?

Heaps

A heap is a complete binary tree such that

- every node stores a value taken from an ordered domain; and
- (Heap property) The value of every internal node is smaller than the values of its children.



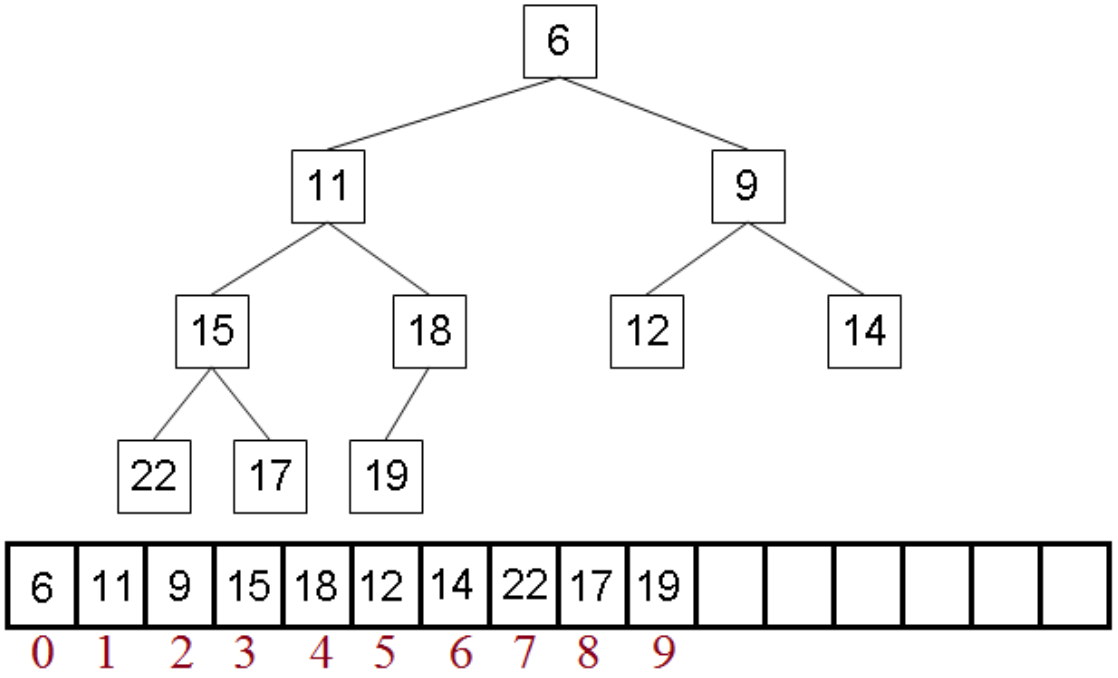
Heap?

No.

Not complete

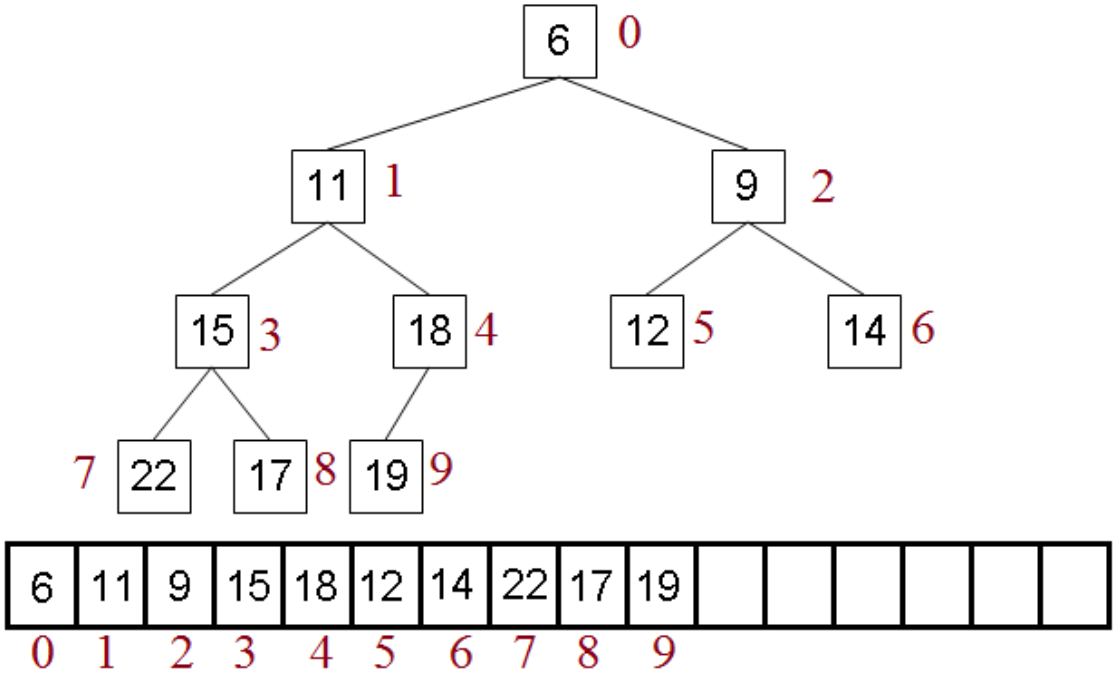
Implementation 2: Array-Based Heaps

Idea: Use an array to represent a complete binary tree.
Values are arranged in the array in level-order.



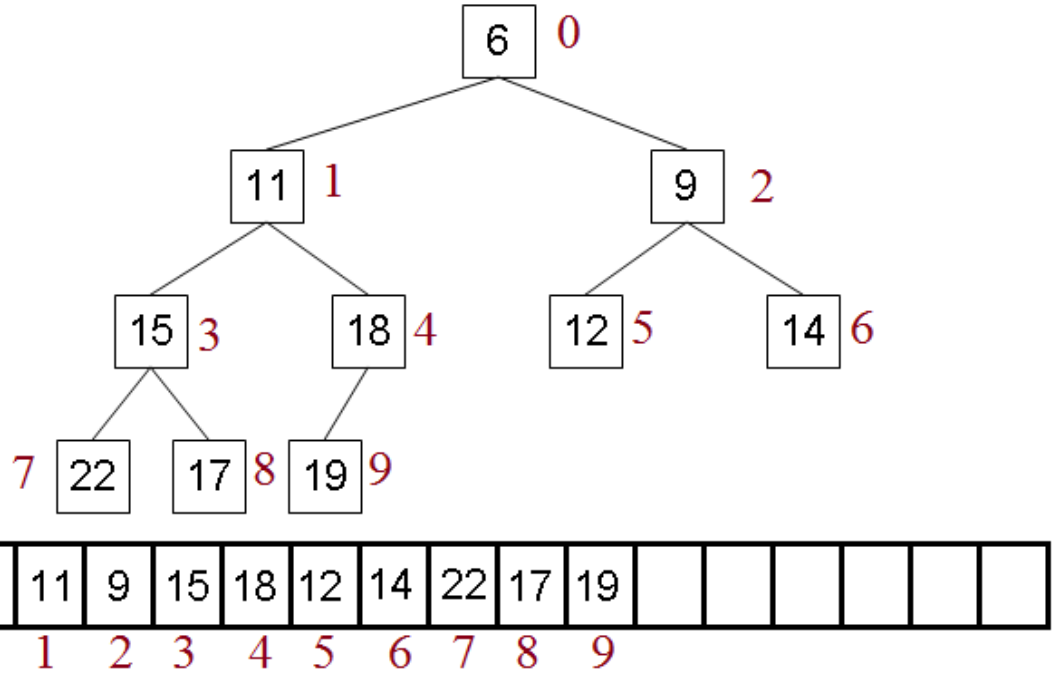
Implementation 2: Array-Based Heaps

Idea: Use an array to represent a complete binary tree.
Values are arranged in the array in level-order.



Implementation 2: Array-Based Heaps

Idea: Use an array to represent a complete binary tree.
Values are arranged in the array in level-order.

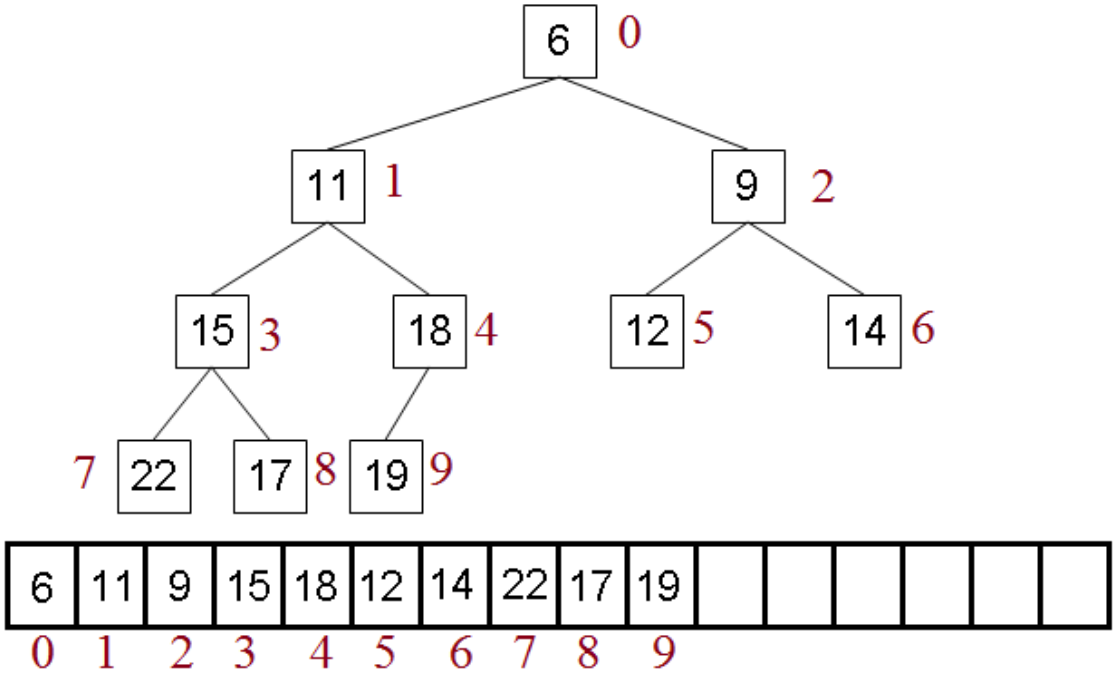


Observation:

- IndexOfLeftChild$= 2 \times$ IndexOfThis $+ 1$
- IndexOfRightChild$= 2 \times$ IndexOfThis $+ 2$
- IndexOfParent$= \left\lfloor \frac{\text{IndexOfThis}-1}{2} \right\rfloor$

Implementation 2: Array-Based Heaps

Idea: Use an array to represent a complete binary tree.
Values are arranged in the array in level-order.



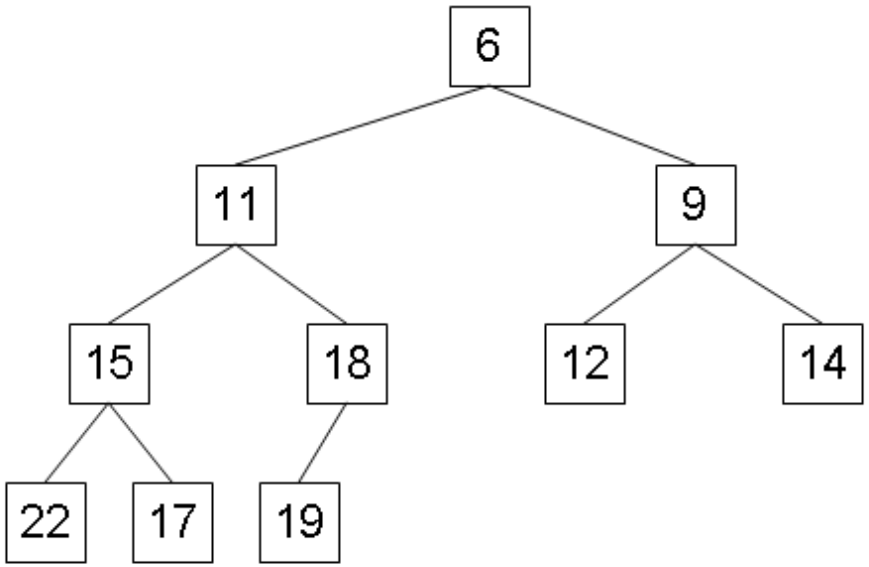
Observation:
In a complete binary tree, we can directly access the left child, right child, and parent of a node given its index in the array.

Implementation 2: Array-Based Heaps

```
protected E[] data;
public ArrayHeap(){
    data = (E)(new Object[INITIAL_CAPACITY]);
}
//compute location of parent given the location i
protected static int parent(int i){ return (i-1)/2;}
//compute location of left child given the location i
protected static int left(int i){ return 2*i+1;}
//compute location of right child given the location i
protected static in right(int i){ return 2*i+2;}
public E getFirst(){
    return data[0];
}
```

Implementation 2: Array-Based Heaps

The Add Operation

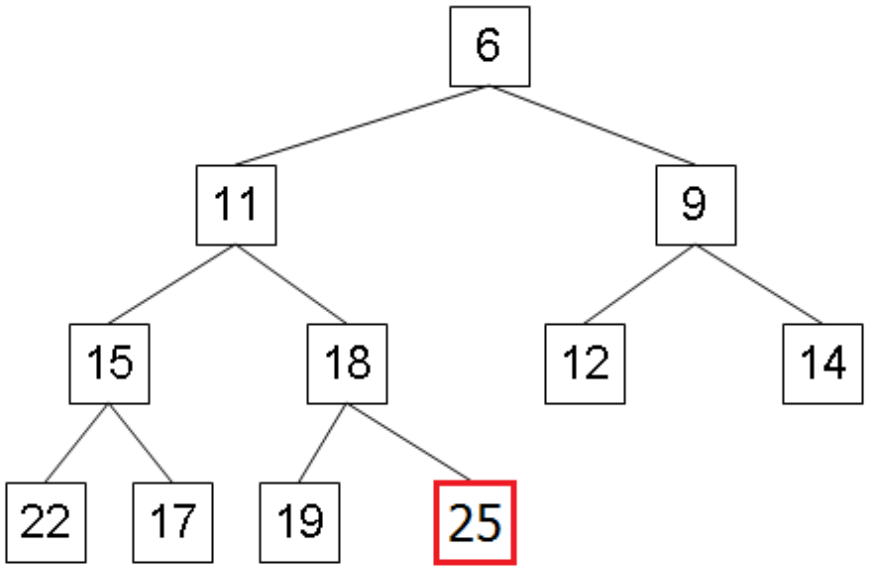


Add(25)

6	11	9	15	18	12	14	22	17	19						
0	1	2	3	4	5	6	7	8	9						

Implementation 2: Array-Based Heaps

The Add Operation



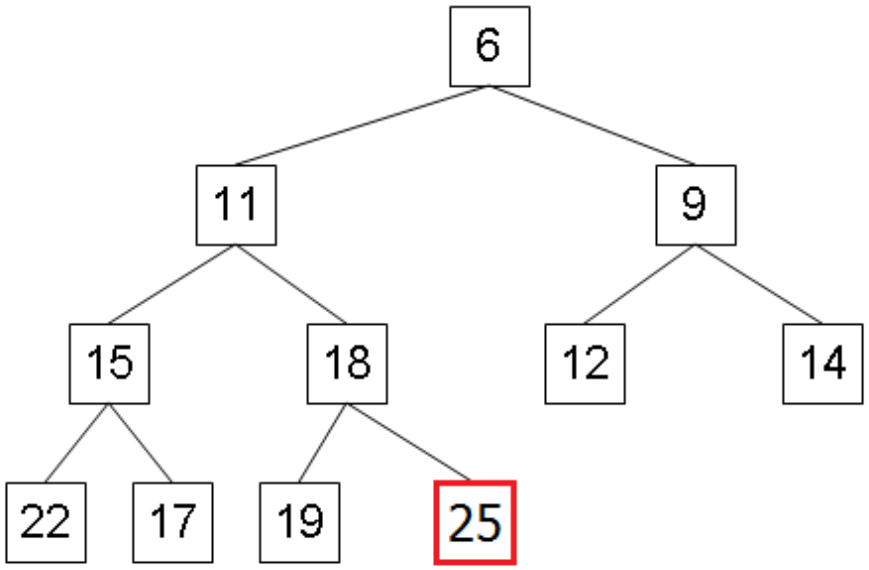
Add(25)

Add the new value at the last position

6	11	9	15	18	12	14	22	17	19	25					
0	1	2	3	4	5	6	7	8	9	10					

Implementation 2: Array-Based Heaps

The Add Operation



Add(25)

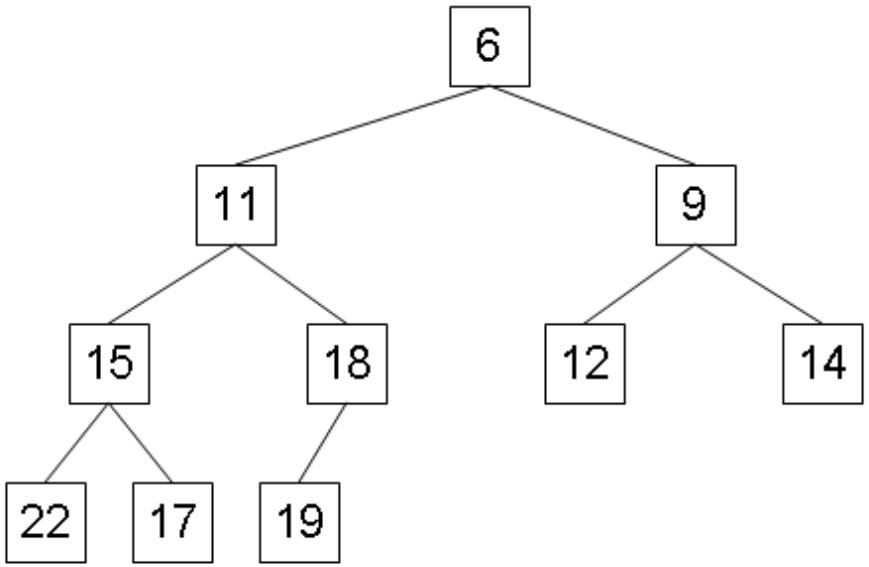
Add the new value at the last position

This works if the new value is bigger than the rest, but what about when it is small?

6	11	9	15	18	12	14	22	17	19	25					
0	1	2	3	4	5	6	7	8	9	10					

Implementation 2: Array-Based Heaps

The Add Operation

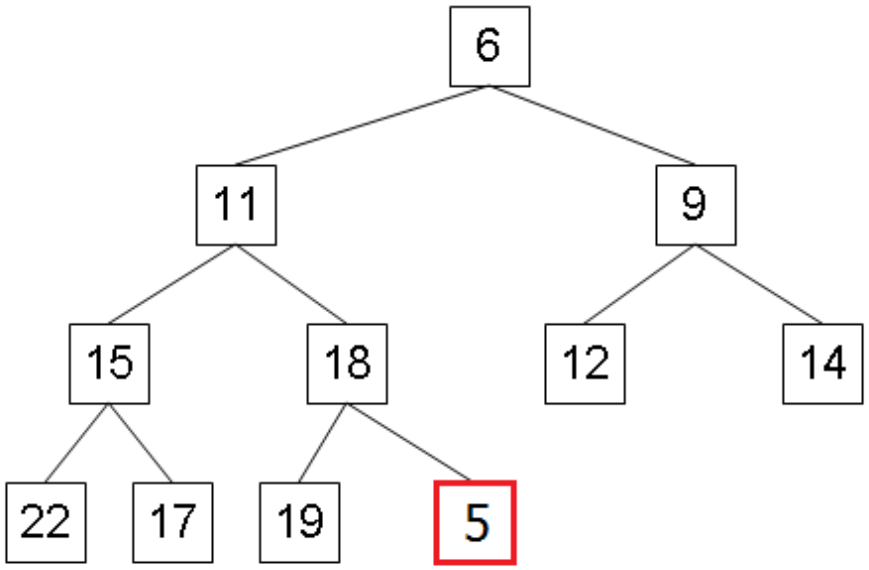


Add(5)

6	11	9	15	18	12	14	22	17	19						
0	1	2	3	4	5	6	7	8	9						

Implementation 2: Array-Based Heaps

The Add Operation



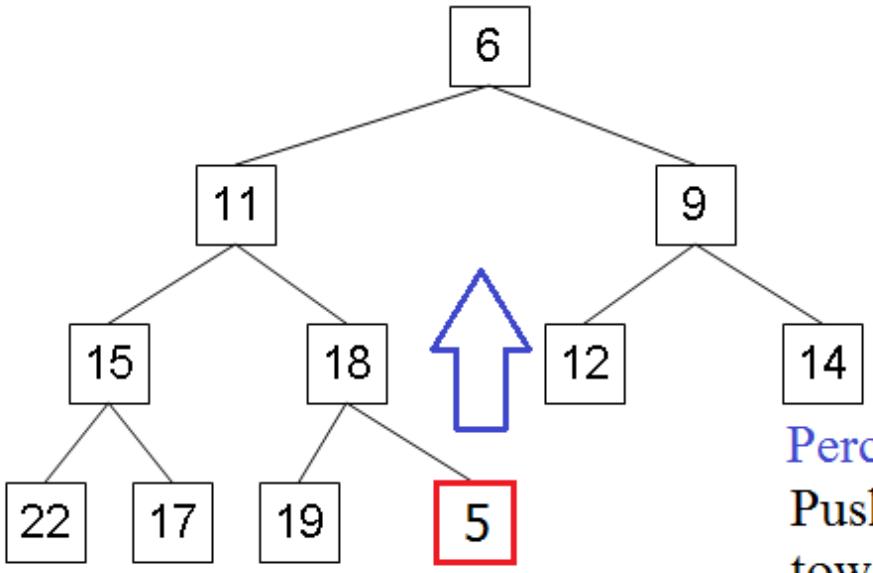
Add(5)

Add the new value at the last position

6	11	9	15	18	12	14	22	17	19	5					
0	1	2	3	4	5	6	7	8	9	10					

Implementation 2: Array-Based Heaps

The Add Operation



Add(5)

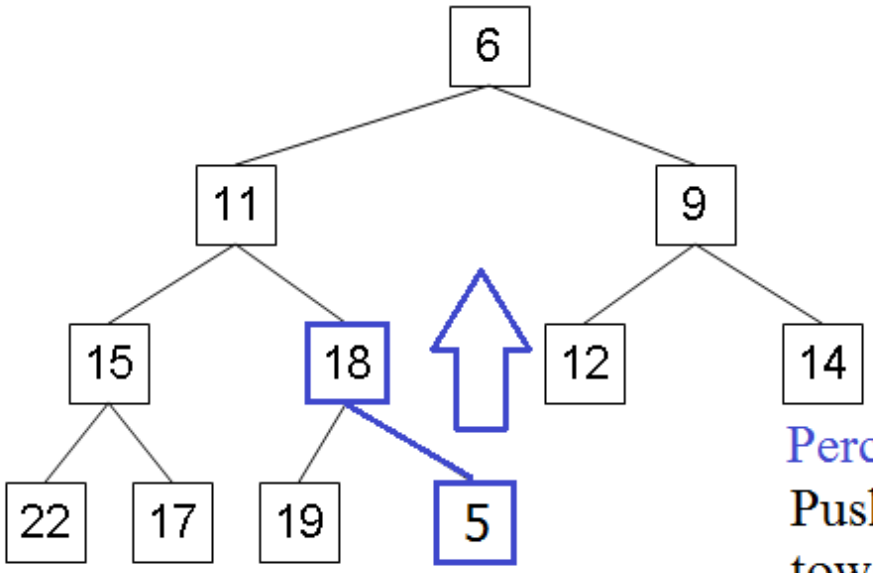
Add the new value at the last position

Percolate up the new value: Pushes the value upwards towards the root, until heap property is preserved.

6	11	9	15	18	12	14	22	17	19	5					
0	1	2	3	4	5	6	7	8	9	10					

Implementation 2: Array-Based Heaps

The Add Operation



Add(5)

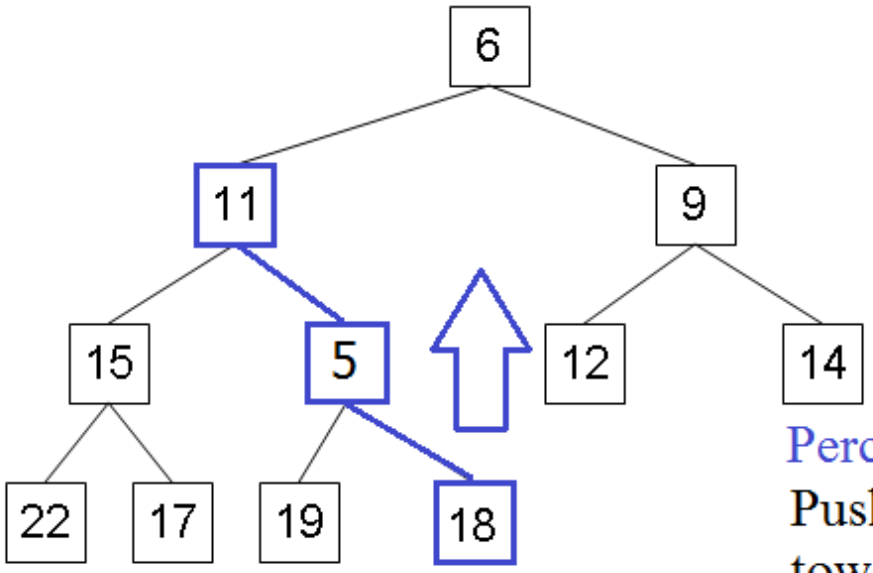
Add the new value at the last position

Percolate up the new value: Pushes the value upwards towards the root, until heap property is preserved.

0	1	2	3	4	5	6	7	8	9	10					
6	11	9	15	18	12	14	22	17	19	5					

Implementation 2: Array-Based Heaps

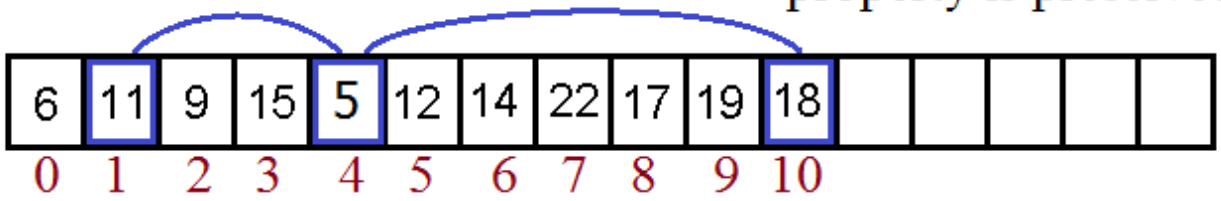
The Add Operation



Add(5)

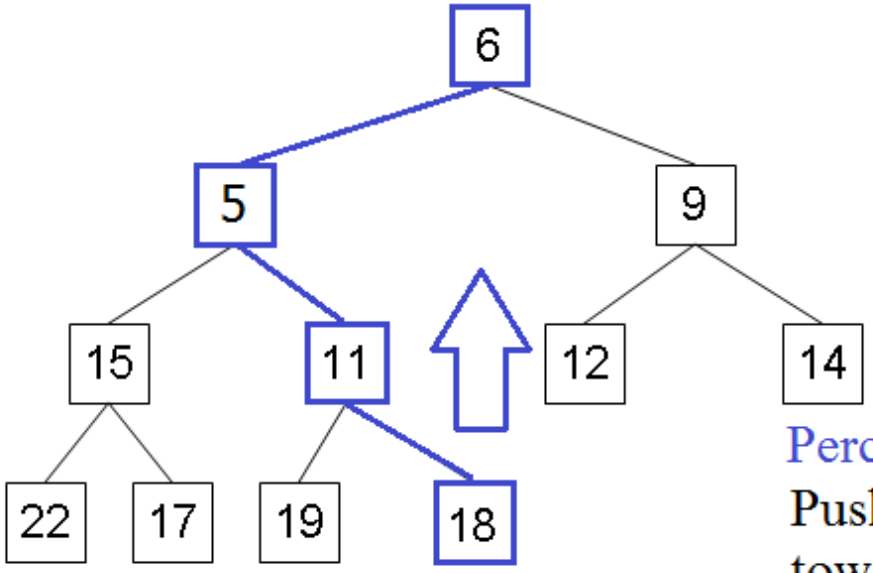
Add the new value at the last position

Percolate up the new value: Pushes the value upwards towards the root, until heap property is preserved.



Implementation 2: Array-Based Heaps

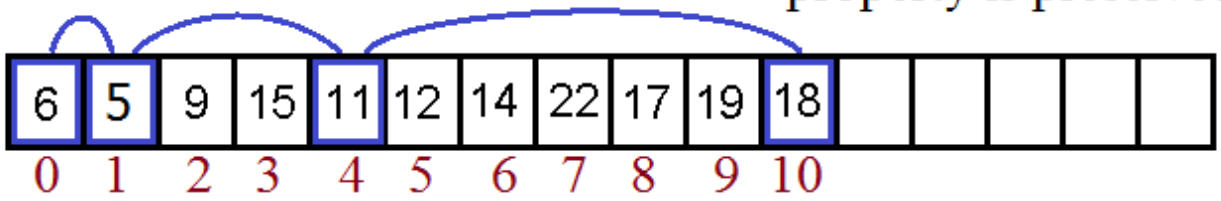
The Add Operation



Add(5)

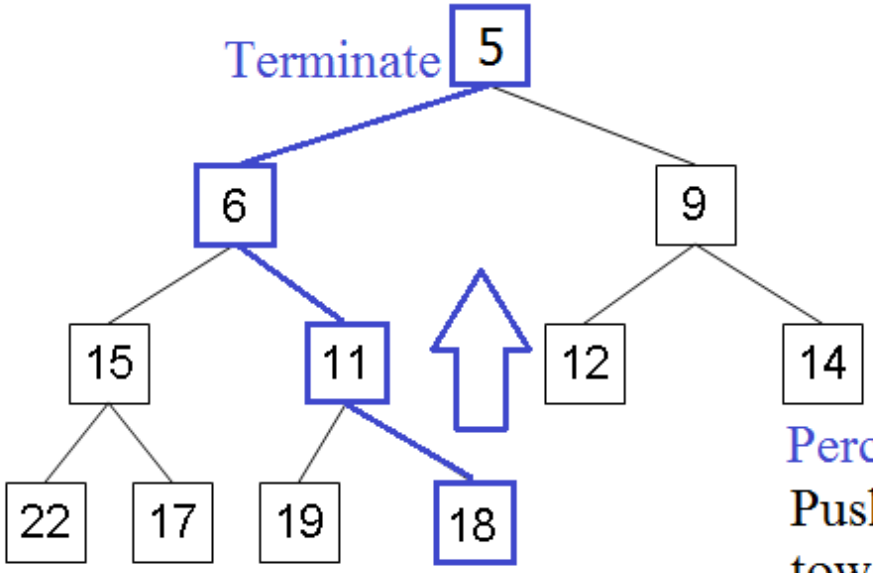
Add the new value at the last position

Percolate up the new value: Pushes the value upwards towards the root, until heap property is preserved.



Implementation 2: Array-Based Heaps

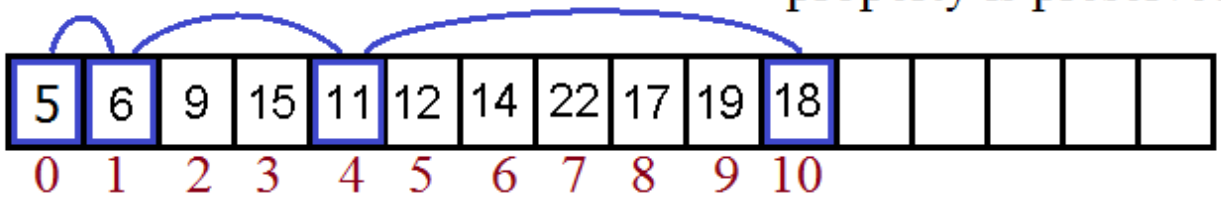
The Add Operation



Add(5)

Add the new value at the last position

Percolate up the new value: Pushes the value upwards towards the root, until heap property is preserved.



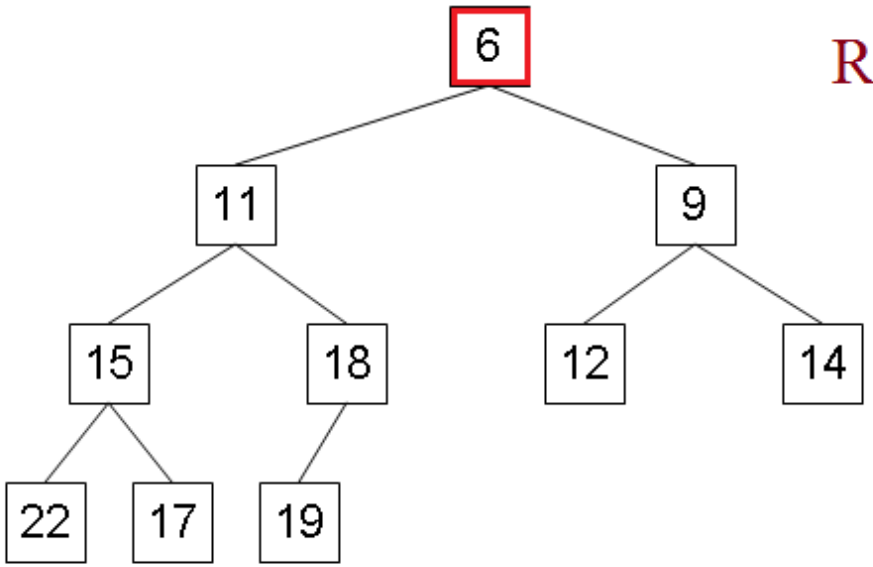
Implementation 2: Array-Based Heaps

The Add Operation

```
protected void percolateUp(int leaf){
// pre:  0<=leaf< size
// post:  moves node at index leaf up to the first
//        position that preserves heap property
   int parent = parent(leaf);
   E value = data[leaf];
   while(leaf>0 &&(value.compareTo(data[parent])<0)){
       data[leaf]=data[parent]);
       leaf=parent;
       parent=parent(leaf);
   }
   data[leaf]=value;
}
public void add(E value){
    ensureCapacity();//Ensure the array have a free position
    data[size]=value;
    percolateUp(size);
    size++;
}
```

Implementation 2: Array-Based Heaps

The Remove Operation

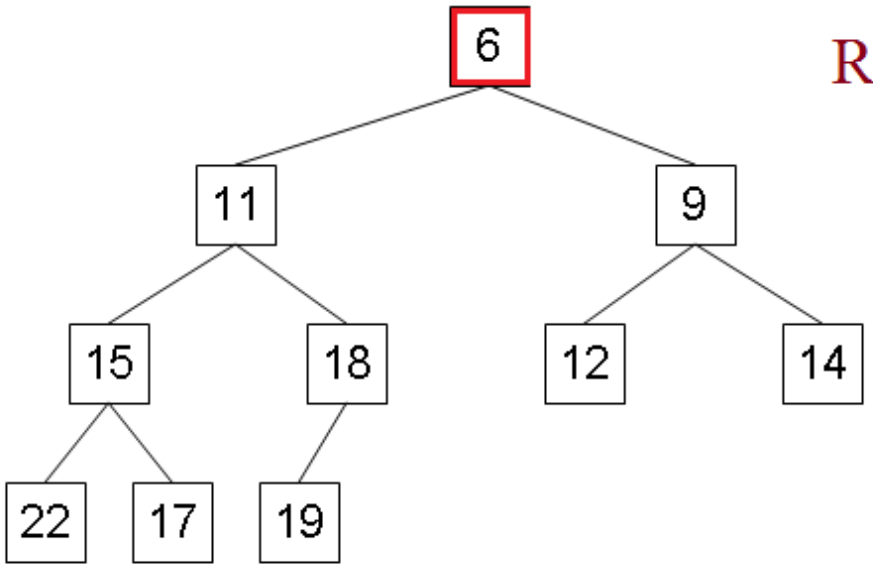


Remove()

6	11	9	15	18	12	14	22	17	19						
0	1	2	3	4	5	6	7	8	9						

Implementation 2: Array-Based Heaps

The Remove Operation



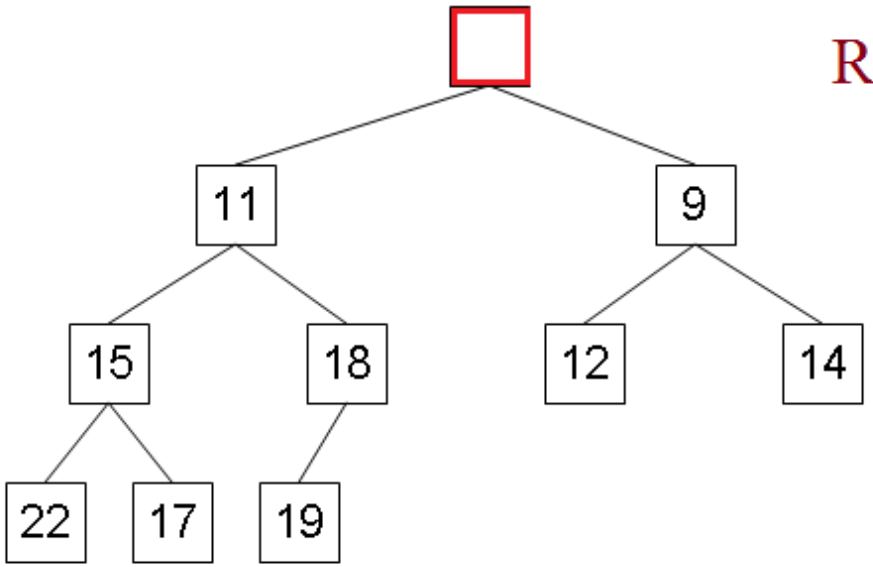
Remove()

Set the first position to null.

0	1	2	3	4	5	6	7	8	9						
6	11	9	15	18	12	14	22	17	19						

Implementation 2: Array-Based Heaps

The Remove Operation



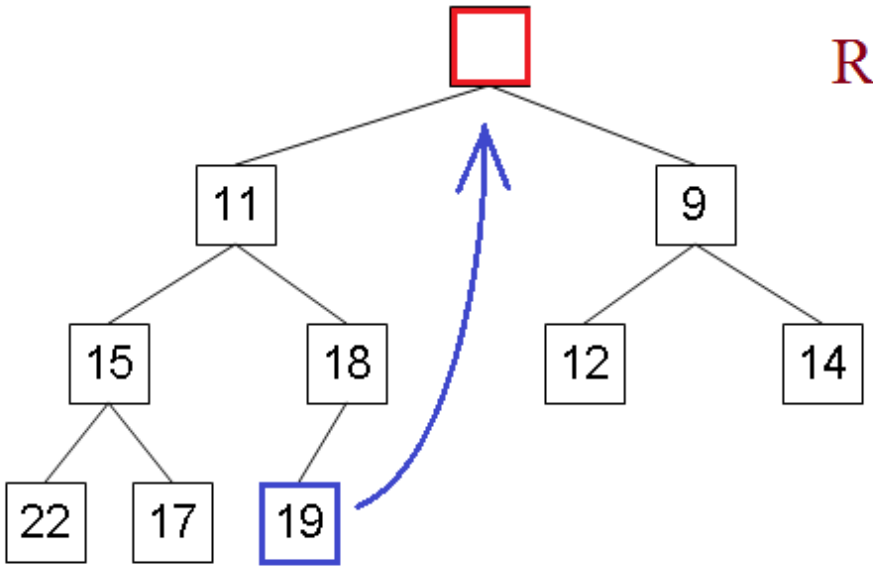
Remove()

Set the first position to null.

	11	9	15	18	12	14	22	17	19						
0	1	2	3	4	5	6	7	8	9						

Implementation 2: Array-Based Heaps

The Remove Operation



Remove()

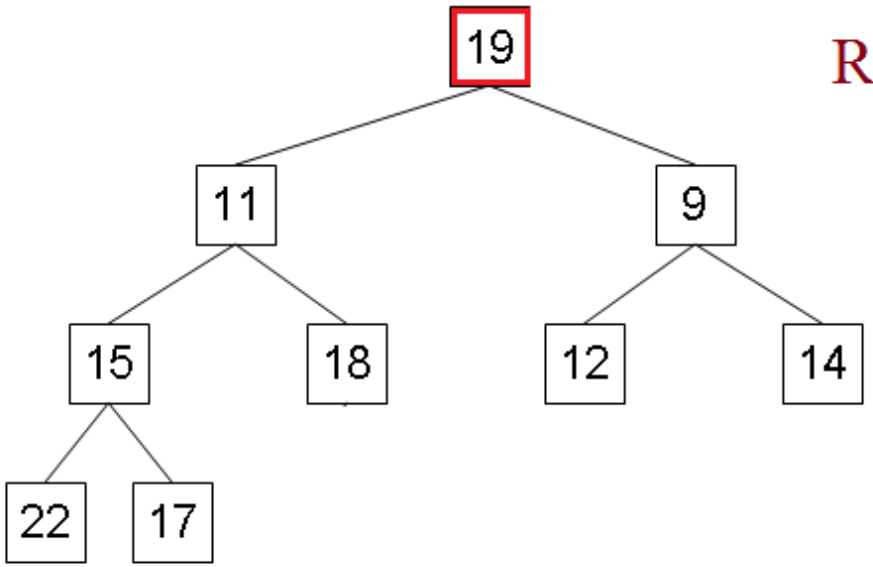
Set the first position to null.

Move the last value to the root.

	11	9	15	18	12	14	22	17	19
0	1	2	3	4	5	6	7	8	9

Implementation 2: Array-Based Heaps

The Remove Operation



Remove()

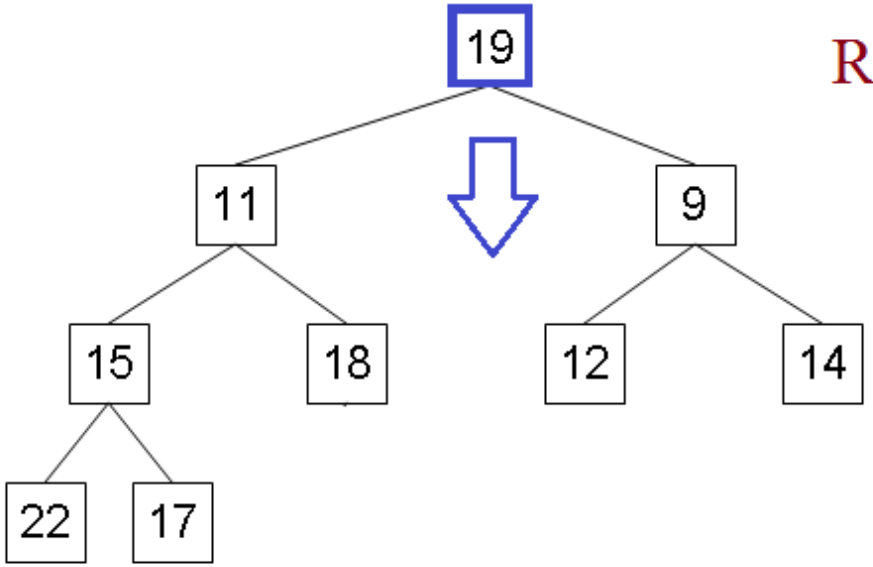
Set the first position to null.

Move the last value to the root.

0	1	2	3	4	5	6	7	8							
19	11	9	15	18	12	14	22	17							

Implementation 2: Array-Based Heaps

The Remove Operation



Remove()

Set the first position to null.

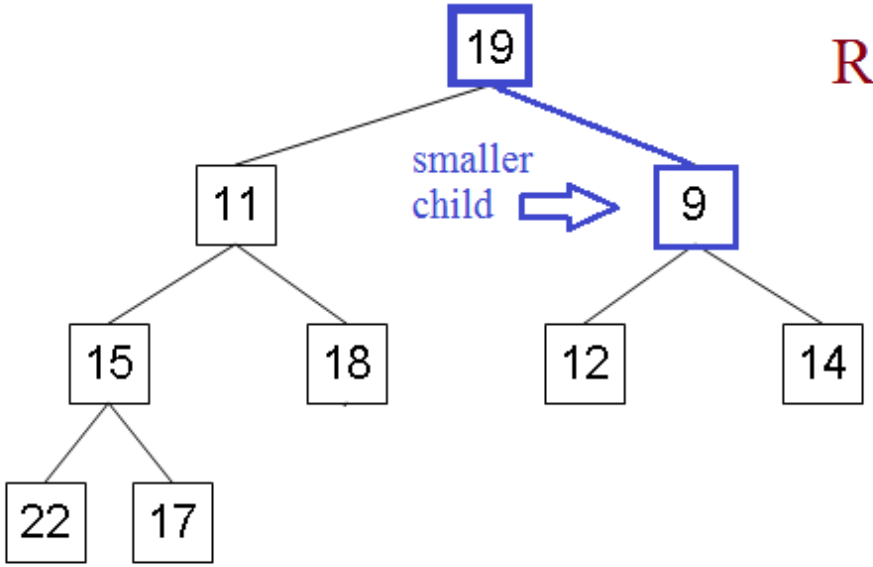
Move the last value to the root.

push down the root

0	1	2	3	4	5	6	7	8							
19	11	9	15	18	12	14	22	17							

Implementation 2: Array-Based Heaps

The Remove Operation

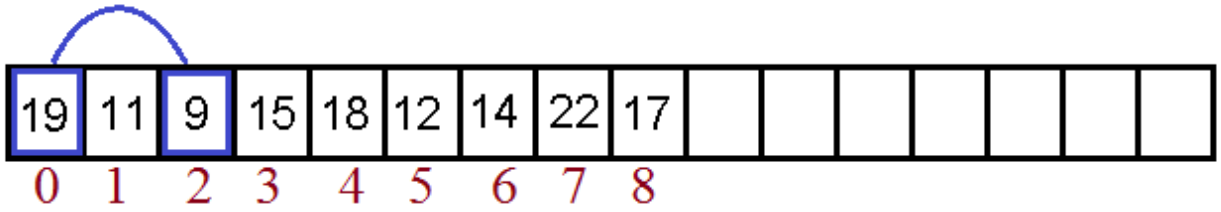


Remove()

Set the first position to null.

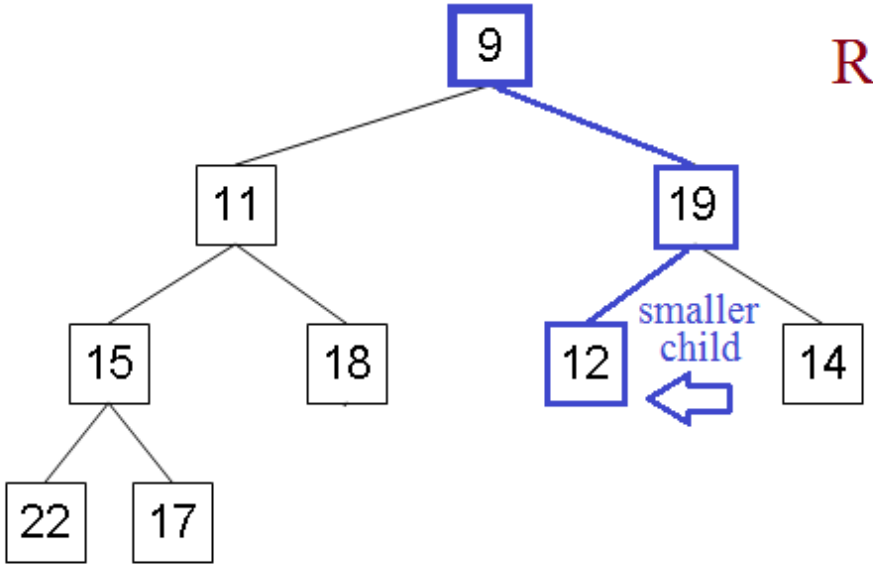
Move the last value to the root.

push down the root



Implementation 2: Array-Based Heaps

The Remove Operation

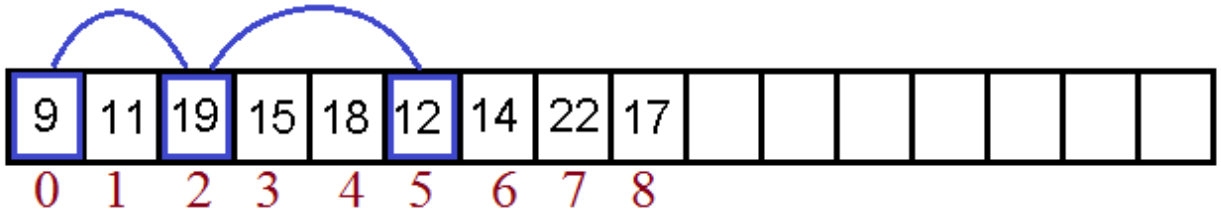


Remove()

Set the first position to null.

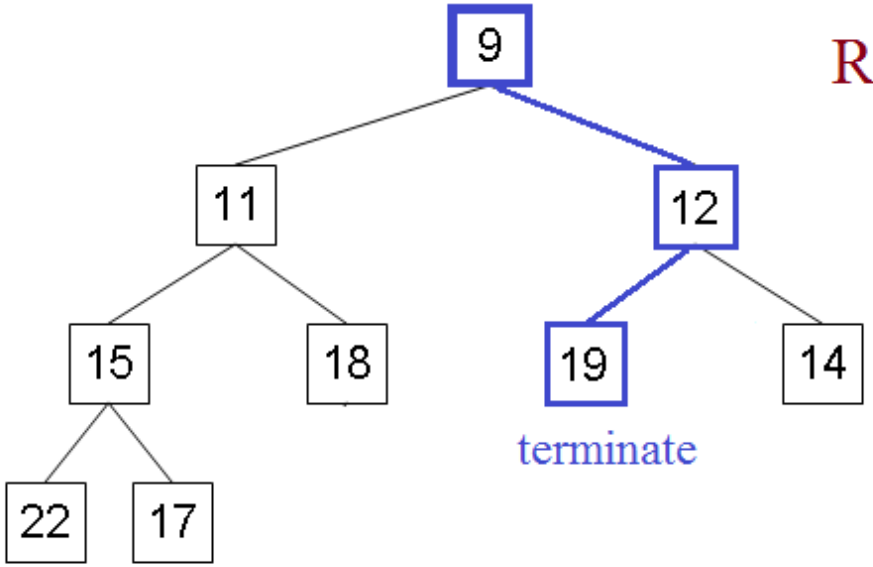
Move the last value to the root.

push down the root



Implementation 2: Array-Based Heaps

The Remove Operation

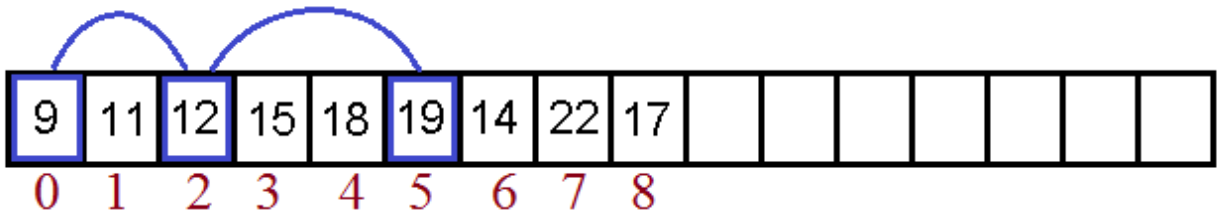


Remove()

Set the first position to null.

Move the last value to the root.

push down the root



Implementation 2: Array-Based Heaps

The Remove Operation

```
protected void pushDown(int root){
// pre:  0<=root<size
// post:  moves node at index root down to a leaf
    int left = left(root); int right = right(root);
    if(left(root)>=size) return;
    E value = data[root];
    int toSwap=root;
    if(right>=size) toSwap=left;
    else if(data[left].compareTo(data[right])<0)
         toSwap=left;
    else toSwap=right;
    if (value.compareTo(data[toSwap])<=0) return;
    data[root]=data[toSwap];
    data[toSwap]=value;
    pushDown(toSwap);
}
```

Implementation 2: Array-Based Heaps

The Remove Operation

```
protected E remove(){
    E minVal=getFirst();
    data[0]=data[size-1];
    size--;
    if (size>1) pushDown(0);
    return minVal;
}
```

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$
- `add(E v)`:
  `percolateUp(int leaf)` + Fixed Number of Statements

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$
- `add(E v)`:
  `percolateUp(int leaf)` + Fixed Number of Statements

  `percolateUp(int leaf)`:
  - Traverse the path from leaf up to the root.
  - The number of nodes is at most the height of the tree.

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$
- `add(E v)`:
  `percolateUp(int leaf)` + Fixed Number of Statements

  `percolateUp(int leaf)`:
  - Traverse the path from leaf up to the root.
  - The number of nodes is at most the height of the tree.
  $\Rightarrow$ Time complexity is linear on height h of the tree $O(h)$.

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$
- `add(E v)`:
  `percolateUp(int leaf)` + Fixed Number of Statements

  `percolateUp(int leaf)`:
  - Traverse the path from leaf up to the root.
  - The number of nodes is at most the height of the tree.
  $\Rightarrow$ Time complexity is linear on height h of the tree $O(h)$.
- `remove()`:
  `pushDown(int root)` + Fixed Number of Statements

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$
- `add(E v)`:
  `percolateUp(int leaf)` + Fixed Number of Statements

  `percolateUp(int leaf)`:
  - Traverse the path from leaf up to the root.
  - The number of nodes is at most the height of the tree.
  $\Rightarrow$ Time complexity is linear on height h of the tree $O(h)$.
- `remove()`:
  `pushDown(int root)` + Fixed Number of Statements

  `pushDown(int root)`:
  - Traverse one path from the root down to a leaf.
  - The number of nodes is at most the height of the tree.

Implementation 2: Array-Based Heaps

Time Complexity

- `getFirst()`: $O(1)$
- `add(E v)`:
  `percolateUp(int leaf)` + Fixed Number of Statements

  `percolateUp(int leaf)`:
  - Traverse the path from leaf up to the root.
  - The number of nodes is at most the height of the tree.
  $\Rightarrow$ Time complexity is linear on height h of the tree $O(h)$.
- `remove()`:
  `pushDown(int root)` + Fixed Number of Statements

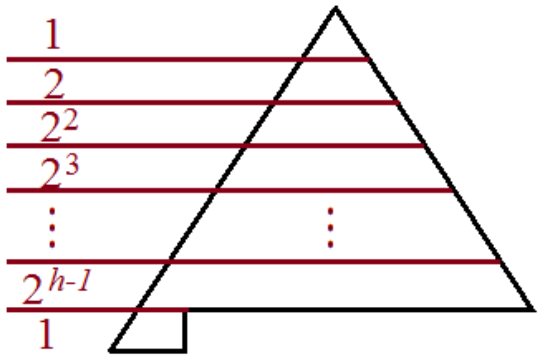
  `pushDown(int root)`:
  - Traverse one path from the root down to a leaf.
  - The number of nodes is at most the height of the tree.

  $\Rightarrow$ Time complexity is linear on height h of the tree $O(h)$.

Implementation 2: Array-Based Heaps

Time Complexity

Question: What is the height of the heap?



- Let n be the number of nodes in the tree.
- Let h be the height of the tree.
- The number of nodes in a complete binary tree

$$n \geq 1 + 2 + 2^2 + 2^3 + \cdots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$$

- Therefore $h \leqslant \log_2(n)$.

Implementation 2: Array-Based Heaps

Time Complexity

Implementing Priority Queue Using a Heap

	Time Complexity
getFirst	$O(1)$
add	$O(\log n)$
remove	$O(\log n)$

Another Sorting Algorithm: Heapsort

Idea: Using a heap data structure to assist sorting.

Heapsort: Given an unsorted array of integers `data`

1. Create an empty heap h
2. Add all elements in `data` to h
3. Remove all elements from h

Another Sorting Algorithm: Heapsort

Idea: Using a heap data structure to assist sorting.

```
public int[] heapsort(int[] data){
    ArrayHeap h = new ArrayHeap();
    for (int i=0; i<data.length;i++){
        h.add(data[i]);
    }
    for (int i=0; i<data.length;i++){
        data[i]=h.remove();
    }     return data;
}
```

Another Sorting Algorithm: Heapsort

Idea: Using a heap data structure to assist sorting.

```
public int[] heapsort(int[] data){
    ArrayHeap h = new ArrayHeap();
    for (int i=0; i<data.length;i++){
        h.add(data[i]);
    }
    for (int i=0; i<data.length;i++){
        data[i]=h.remove();
    }     return data;
}
```

Time complexity: $O(n \log n)$

Part II: Binary Search Trees

The Searching Problem

Searching.
The problem of identifying a given object from a collection of objects (usually taken from an ordered domain).

Binary Search Trees

Idea: Store a sorted list in a binary tree

Example:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	29	43	45	56	58	71	78	83	91	95	99	156	171	222	291

Binary Search Trees

Idea: Store a sorted list in a binary tree

Example:

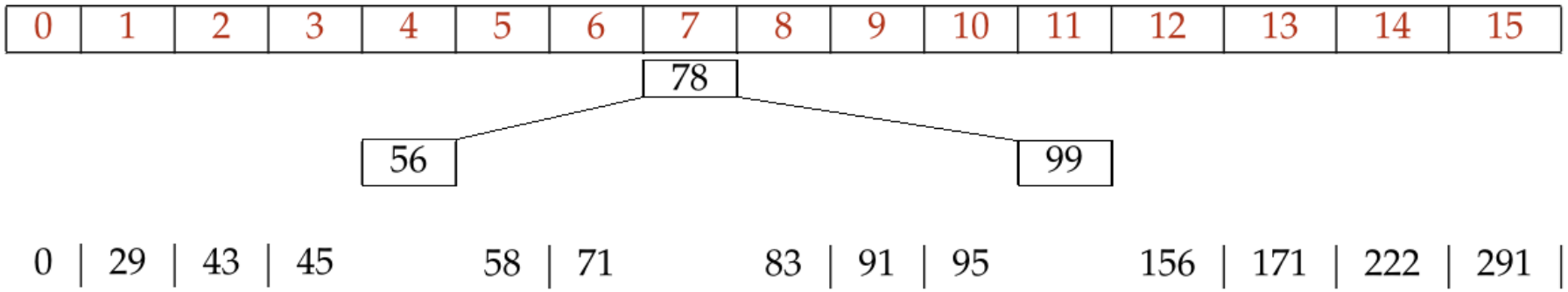
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
							78								

0 | 29 | 43 | 45 | 56 | 58 | 71 | 83 | 91 | 95 | 99 | 156 | 171 | 222 | 291 |

Binary Search Trees

Idea: Store a sorted list in a binary tree

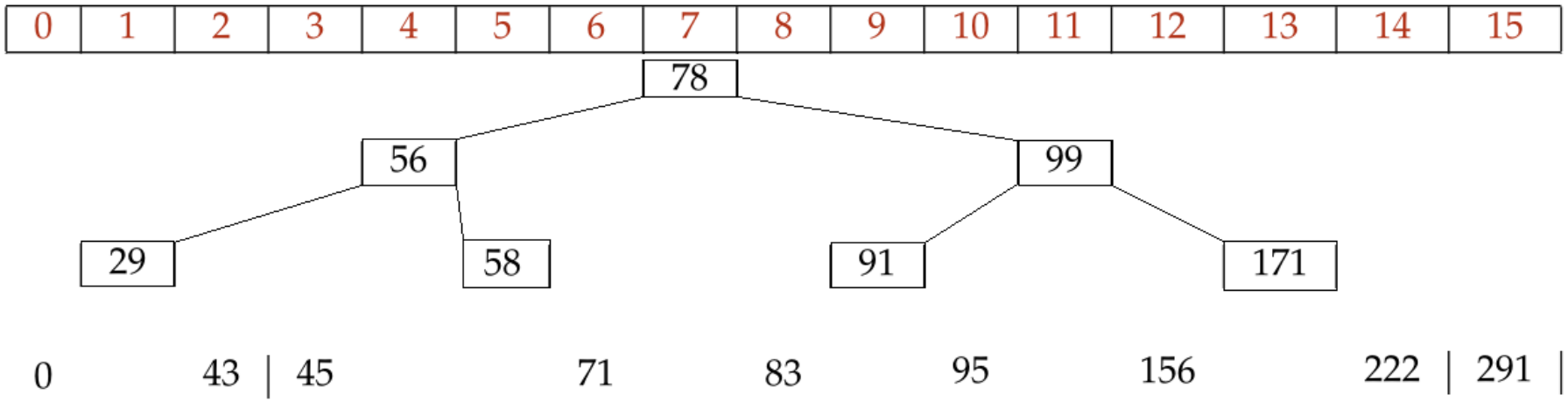
Example:



Binary Search Trees

Idea: Store a sorted list in a binary tree

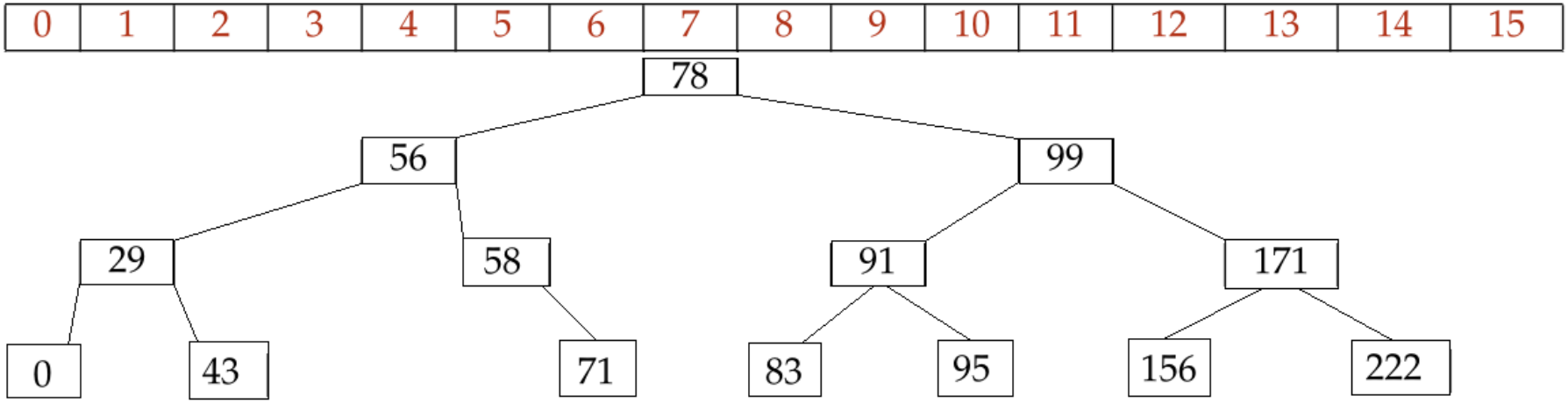
Example:



Binary Search Trees

Idea: Store a sorted list in a binary tree

Example:



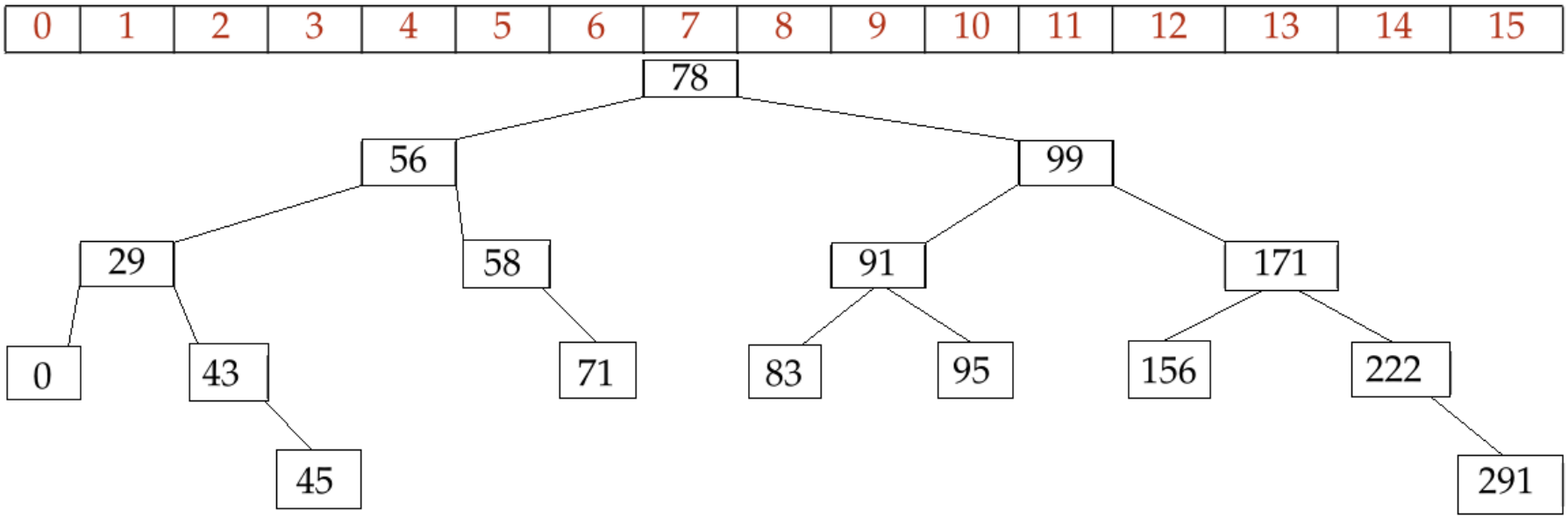
45

291

Binary Search Trees

Idea: Store a sorted list in a binary tree

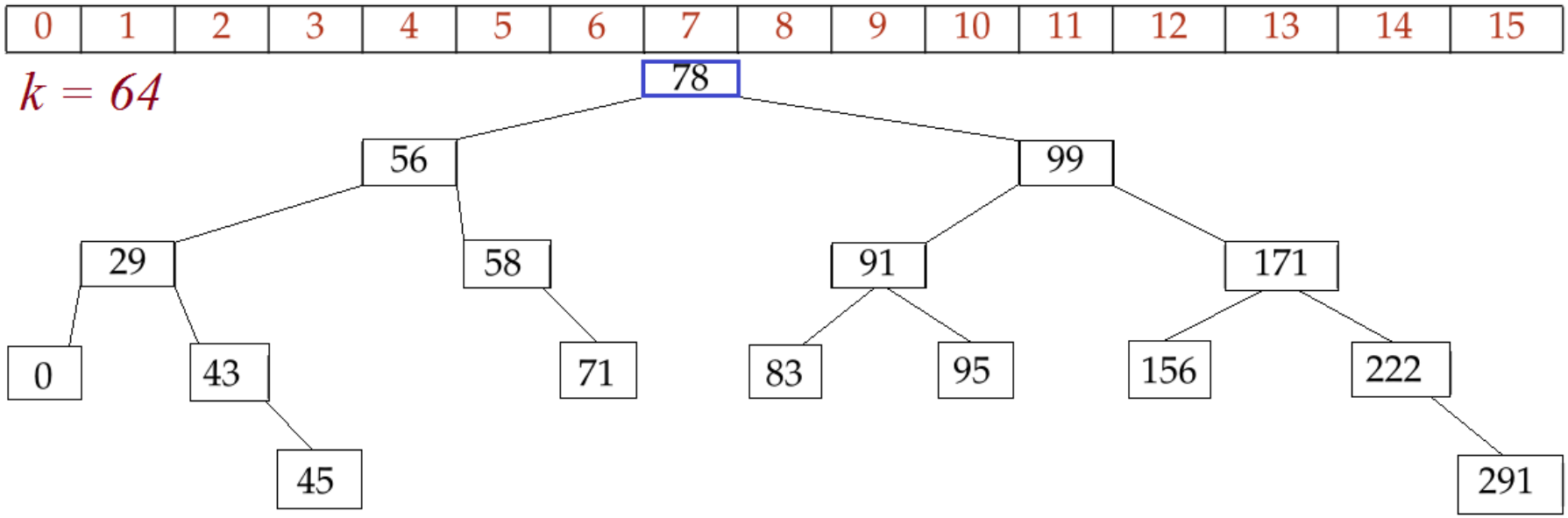
Example:



Binary Search Trees

Idea: Store a sorted list in a binary tree
To search for a key k, start from the root, and follow a path downwards.

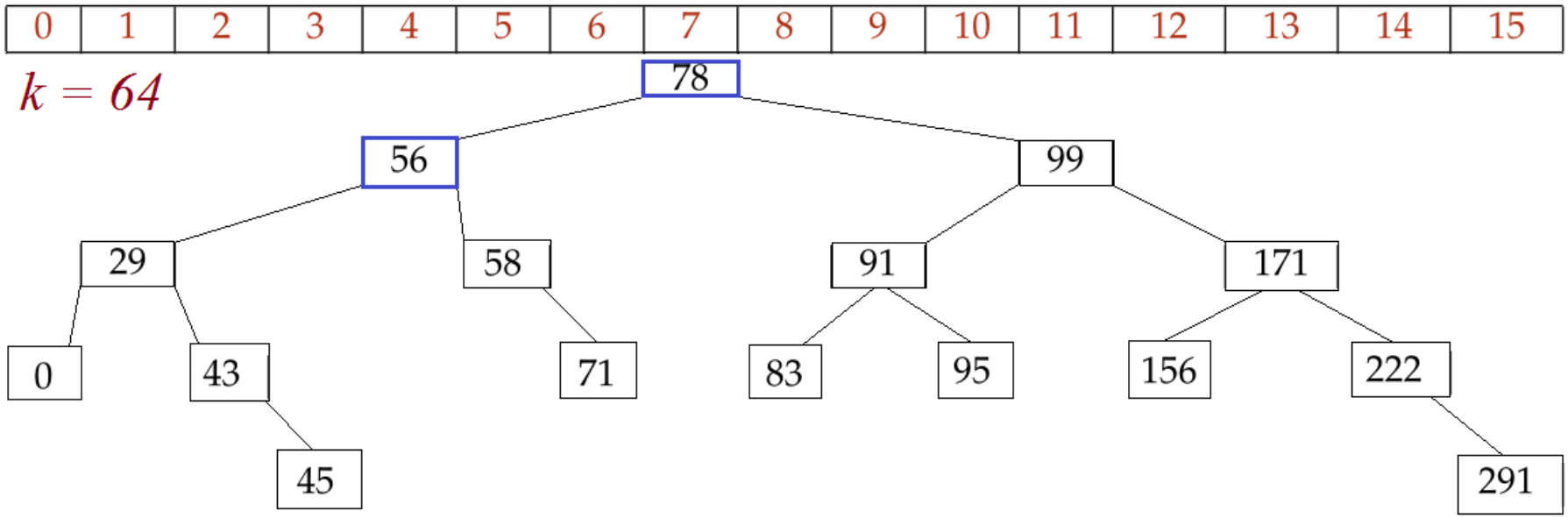
Example:



Binary Search Trees

Idea: Store a sorted list in a binary tree
To search for a key k, start from the root, and follow a path downwards.

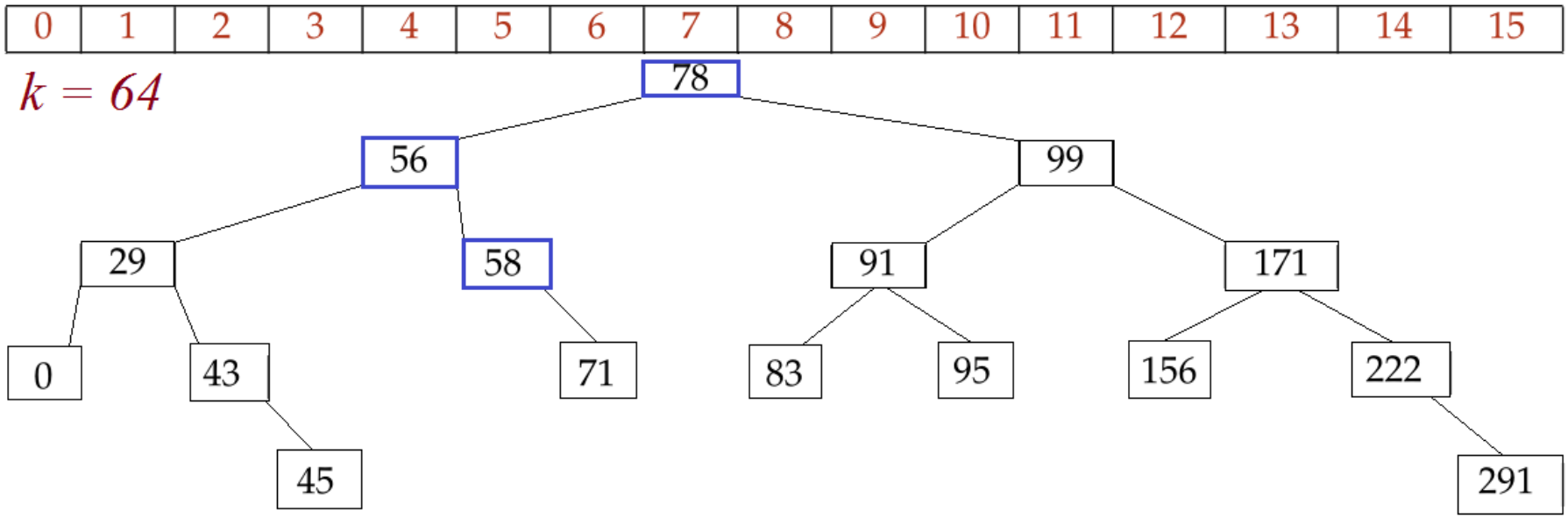
Example:



Binary Search Trees

Idea: Store a sorted list in a binary tree
To search for a key k, start from the root, and follow a path downwards.

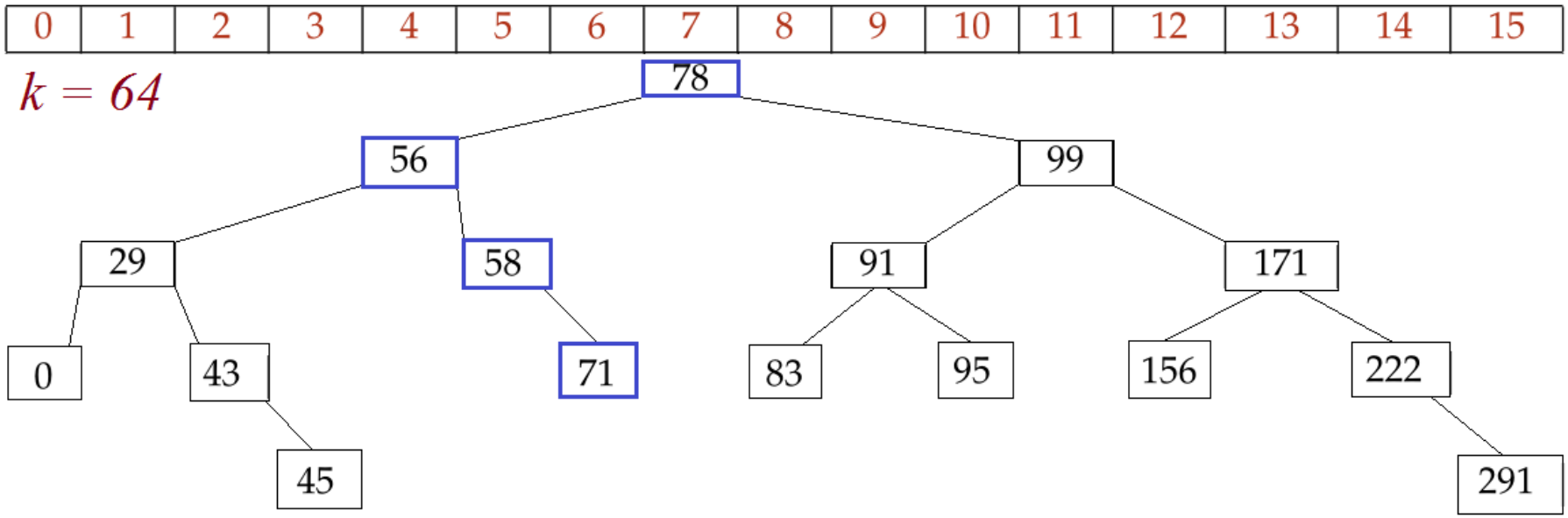
Example:



Binary Search Trees

Idea: Store a sorted list in a binary tree
To search for a key k, start from the root, and follow a path downwards.

Example:



Search Structures

Static v.s. Dynamic Searching

- Static Searching: Looking for a key from a fixed list.
  $\Rightarrow$ Solved using binary search algorithm

Search Structures

Static v.s. Dynamic Searching

- Static Searching: Looking for a key from a fixed list.
  $\Rightarrow$ Solved using binary search algorithm
- Dynamic Searching:
  - The list may change by update operations.
  - Maintain a data structure that allows efficient updates and queries.
  - Such a data structure is called a search structure.

Search Structures

Static v.s. Dynamic Searching

- Static Searching: Looking for a key from a fixed list.

  ⇒ Solved using binary search algorithm

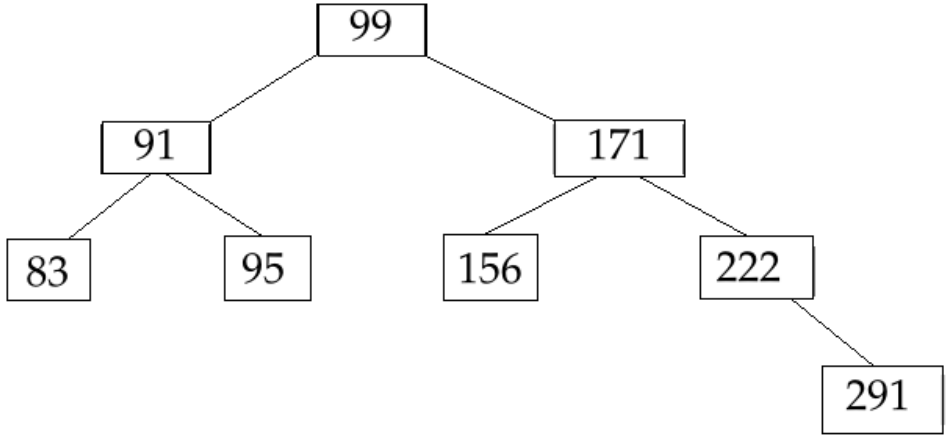
- Dynamic Searching:
  - The list may change by update operations.
  - Maintain a data structure that allows efficient updates and queries.
  - Such a data structure is called a search structure.

  ⇒ Example: binary search trees.

Binary Search Trees

Binary Search Trees

A binary search tree is a binary tree where every node associated with a value (from an ordered domain) and the following property is satisfied:
(Binary Search Property) For every node v, the value of its left child $\leq$ the value of $v \leq$ the value of its right child.



Binary Search Trees

Binary Search Trees

A binary search tree (BST) is a binary tree where every node associated with a value (from an ordered domain) and the following property is satisfied:
(Binary Search Property) For every node v, the value of its left child $\leq$ the value of $v \leq$ the value of its right child.

BST supports the following operations:

- The Query Operations:
  - contains(E value): returns true iff value is found
  - get(E value): returns object found in tree, or null
- The Update Operations:
  - add(E value): Add value to the BST
  - remove(E value): Remove a node with value

BST: Interface

```
public class BinarySearchTree<E extends Comparable<E>>{
    public BinarySearchTree()
    public void add(E value)
    // post:  value added to the tree
    public void contains(E value)
    // post:  returns true iff value is found
    public E get(E value)
    // post:  returns object found in tree, or null
    public E remove(E value)
    // post:  removes one instance of value, if found
}
```

Implementation 1: Simple Binary Search Trees

Idea: Use a binary tree to realise a BST

```
protected BinaryTree<E> root;

public BinarySearchTree(){
    root = new BinaryTree<E>();
    size=0;
}
```

Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

Pseudocode:

1. Compare `val` and `root.value()`
2. If `val` equals `root.value()`, return `root`
3. Otherwise check the appropriate child
4. If the child doesn't exist, return `root`
5. Otherwise recurse on the child.

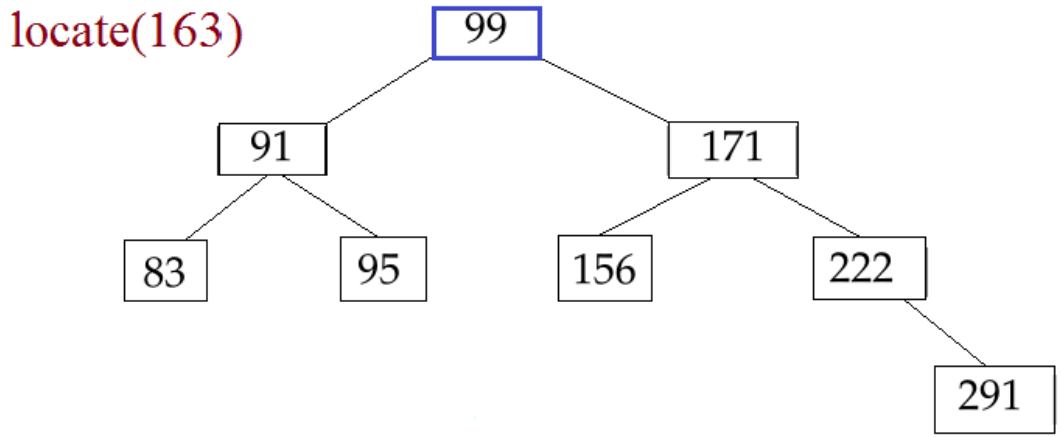
Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

Example:



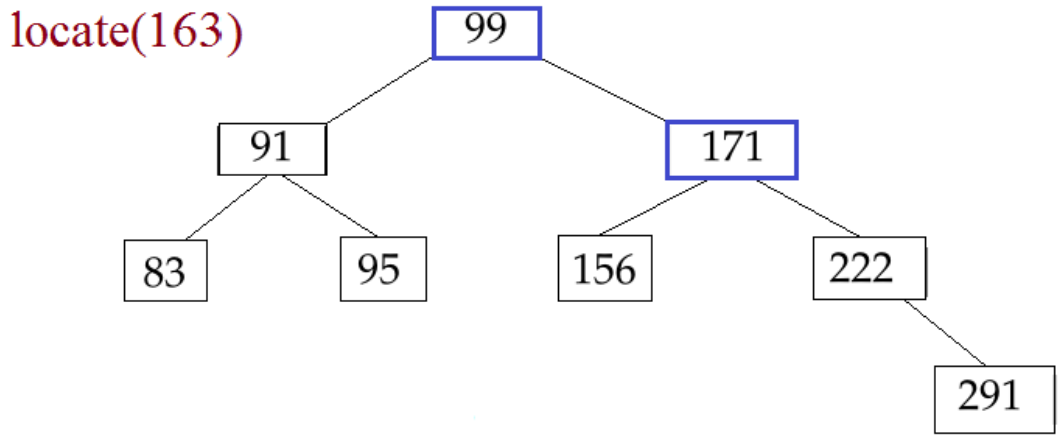
Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

Example:



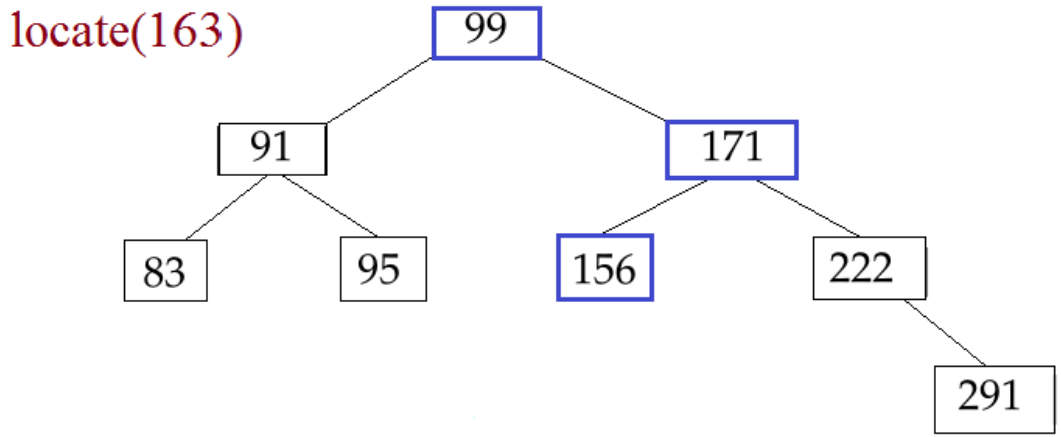
Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

Example:



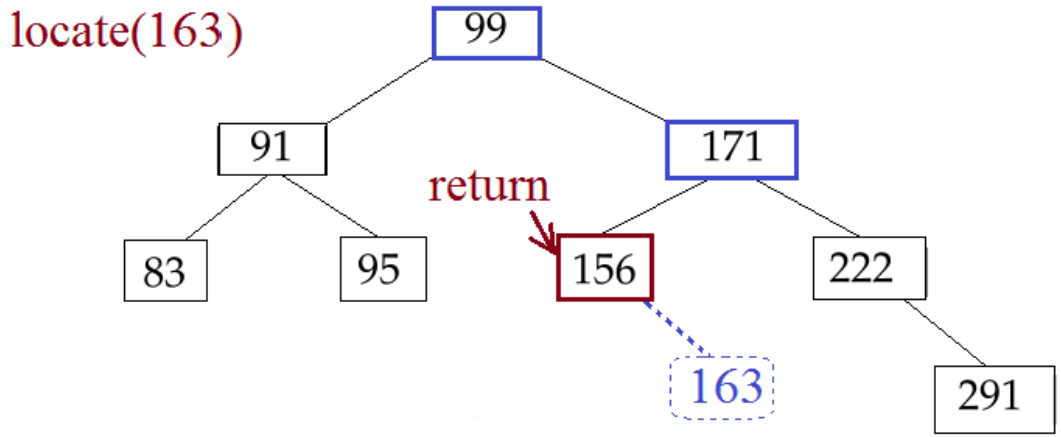
Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

Example:



Implementation 1: Simple Binary Search Trees

The Query Operations

Idea: First implement `locate(root, val)` that identifies the node where `val` is stored below `root`.

- If `val` exists in the BST, return the node storing it
- If `val` doesn't exist, return the nodes where it should be added as a child

```
protected BinaryTree<E> locate(BinaryTree<E> root, E val){
    E rootVal=root.value();
    BinaryTree<E> child;

    if(rootV.equals(val)) return root;
    if(val.compareTo(rootVal)<0) child=root.left();
    else child=root.right();

    if (child.isEmpty()) return root;
    else return locate(child, val);
}
```

Implementation 1: Simple Binary Search Trees

The Query Operations

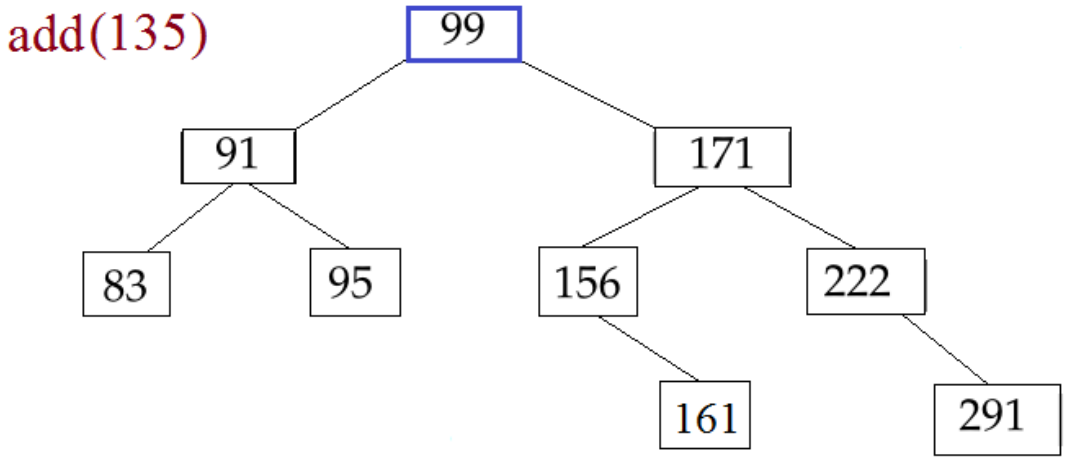
```
public boolean contains(E value){
    if (root.isEmpty()) return false;
    BinaryTree<E> loc = locate(root, value);
    return value.equals(loc.value());
}

public E get(E value){
    if (root.isEmpty()) return null;
    BinaryTree<E> loc = locate(root, value);
    if (!value.equals(loc.value())) return null;
    else return loc.value();
}
```

Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

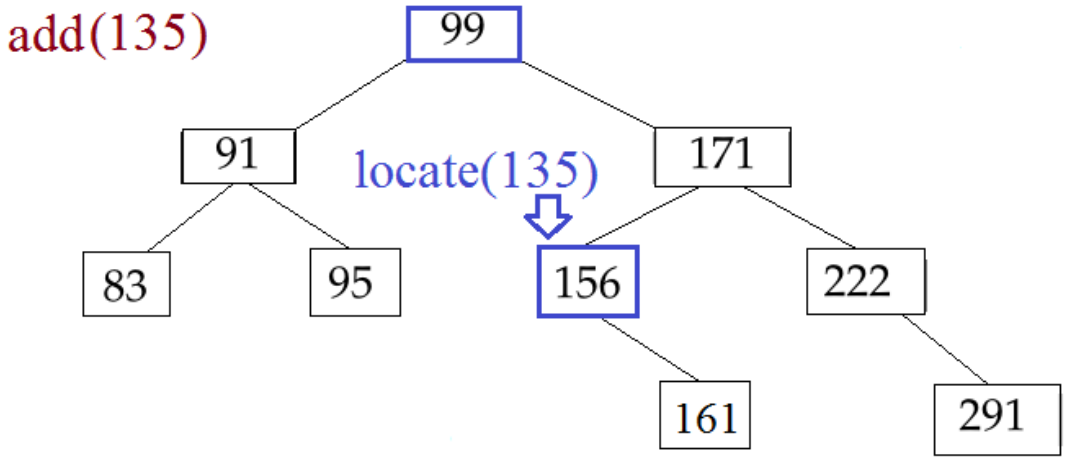


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

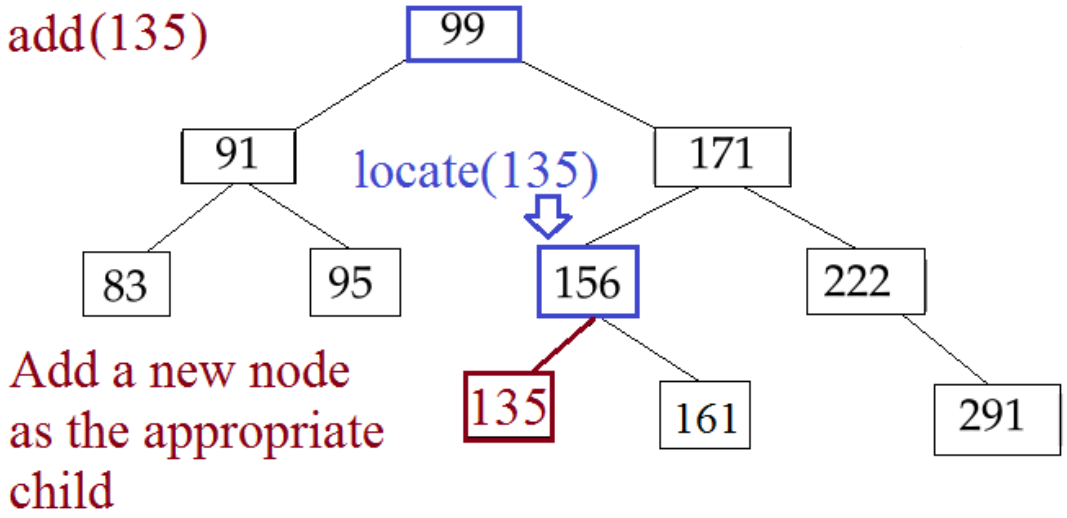


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

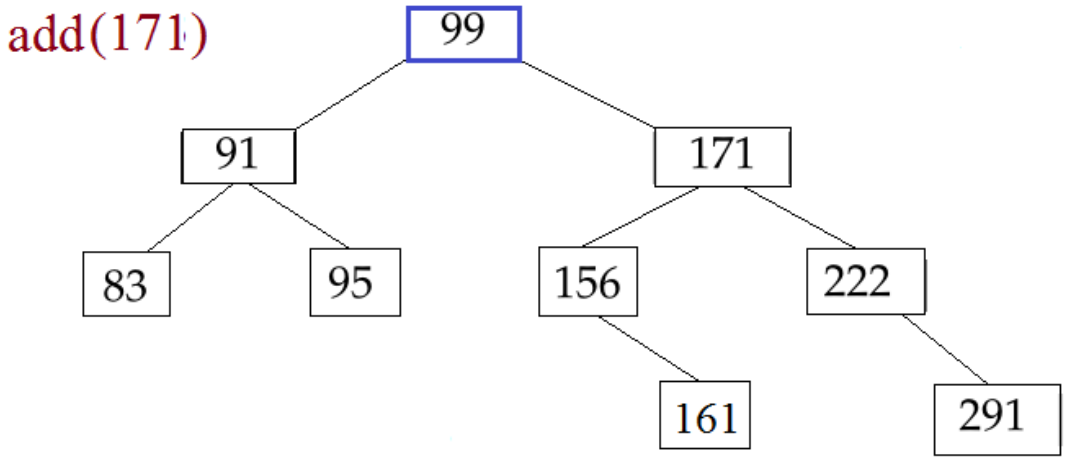


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

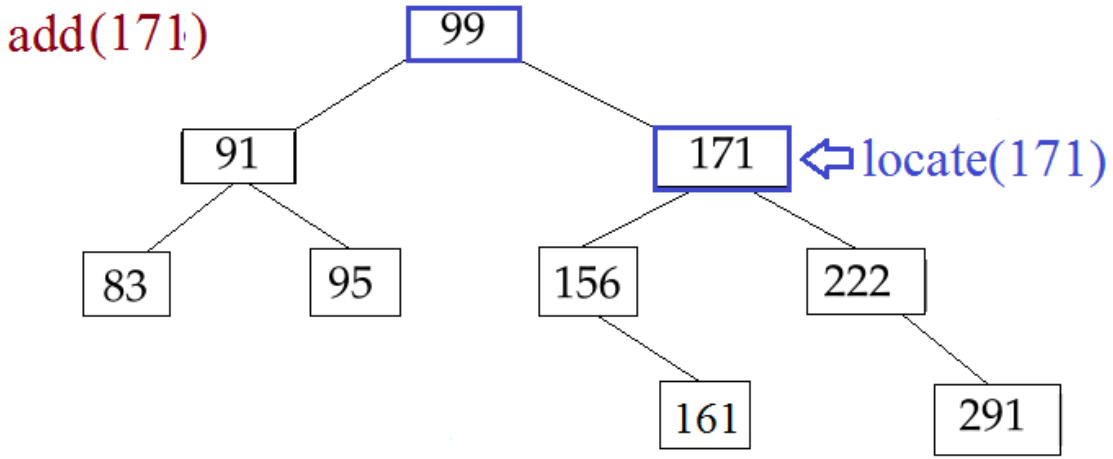


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

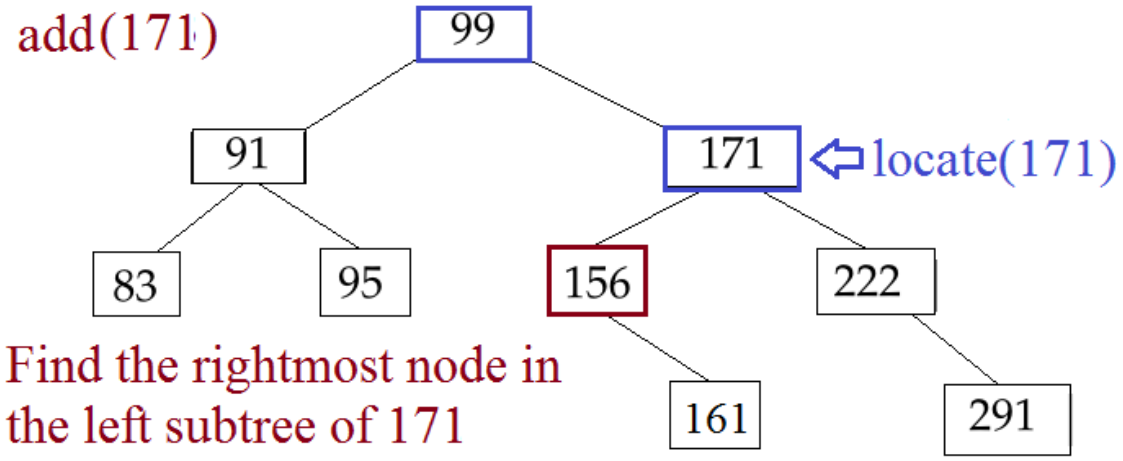


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

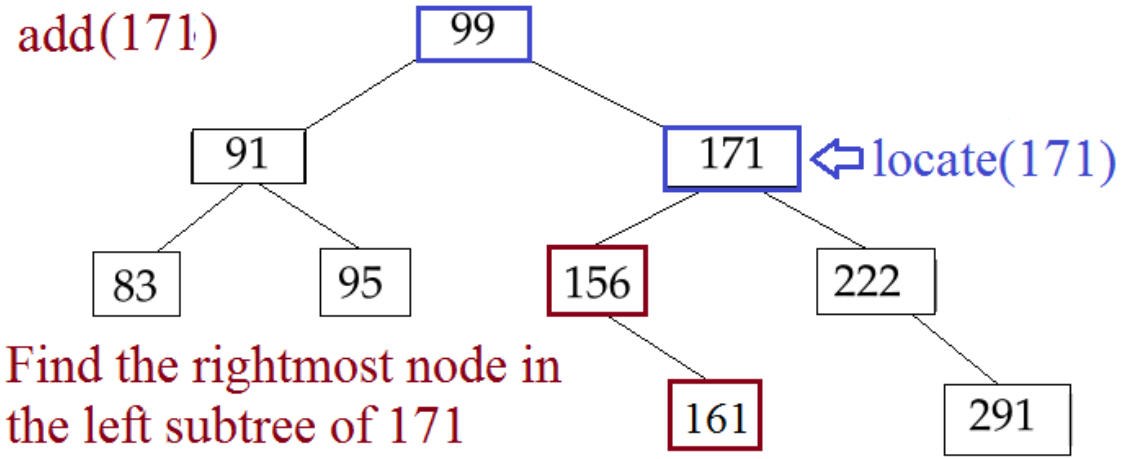


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.

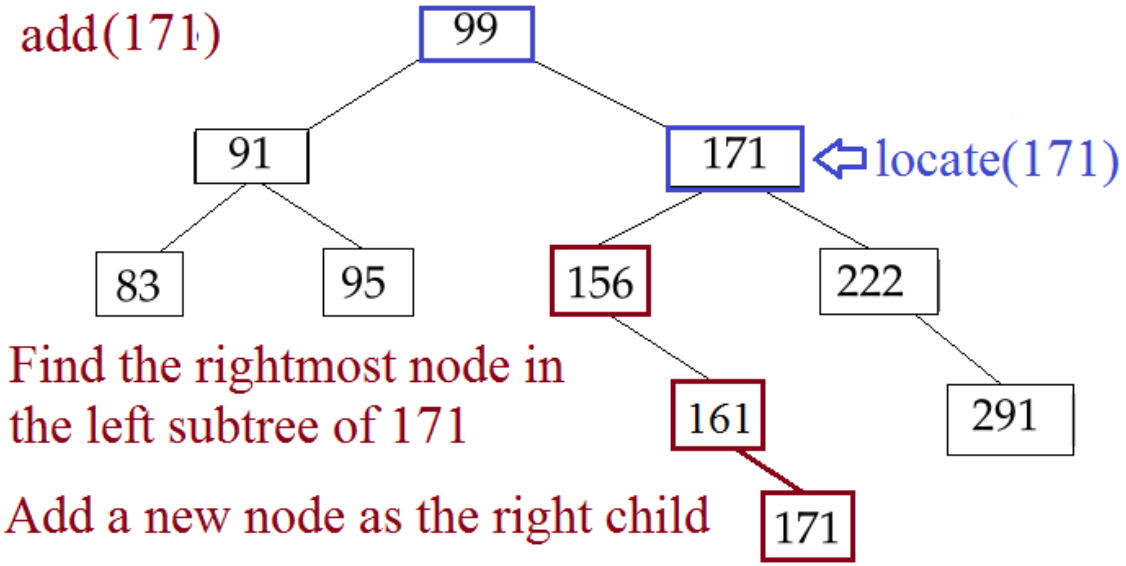


Implementation 1: Simple Binary Search Trees

The Add Operation

Idea: Use `locate(root, value)` to locate the node where value should be added.

- If value not found, add it as a child of located node
- Otherwise, add it as the rightmost descendent to the left of located node.



Implementation 1: Simple Binary Search Trees

The Add Operation

```
public BinaryTree<E> predecessor(BinaryTree<E> node){ ... }
// Finds the rightmost node of the left subtree of node

public void add(E value){
   BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
   //create a node with value, and empty children

   if (root.isEmpty()) root=newNode;
   else {
       BinaryTree<E> addLoc=locate(root,value);
       if (value.compareTo(addLoc.value())>0)
           addLoc.setRight(newNode) ;
       else{
           if (!addLoc.left.isEmpty())
               predecessor(addLoc).setRight()(newNode);
           else
               addLoc.setLeft(newNode);
       }
   }
}
```

Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

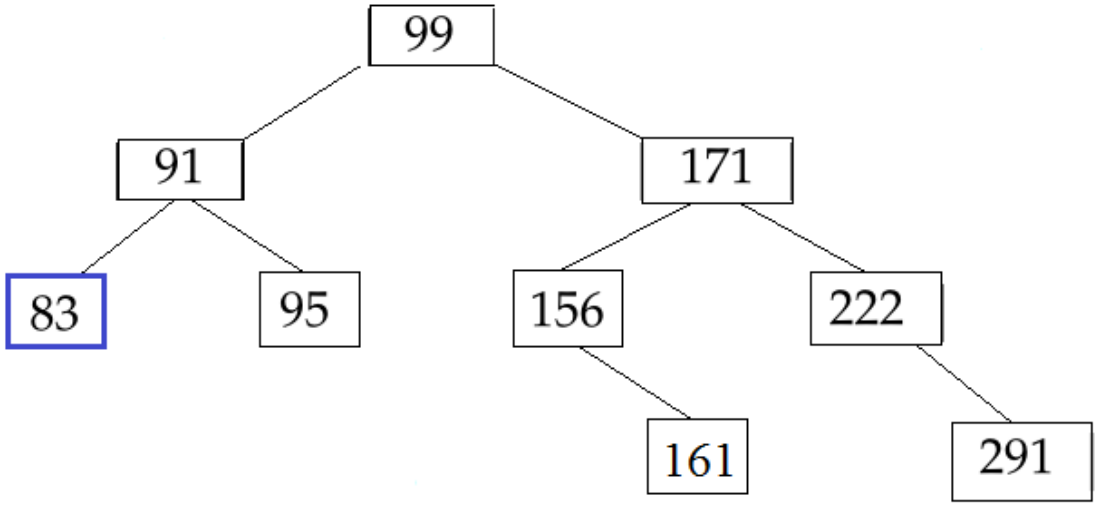
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(83)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

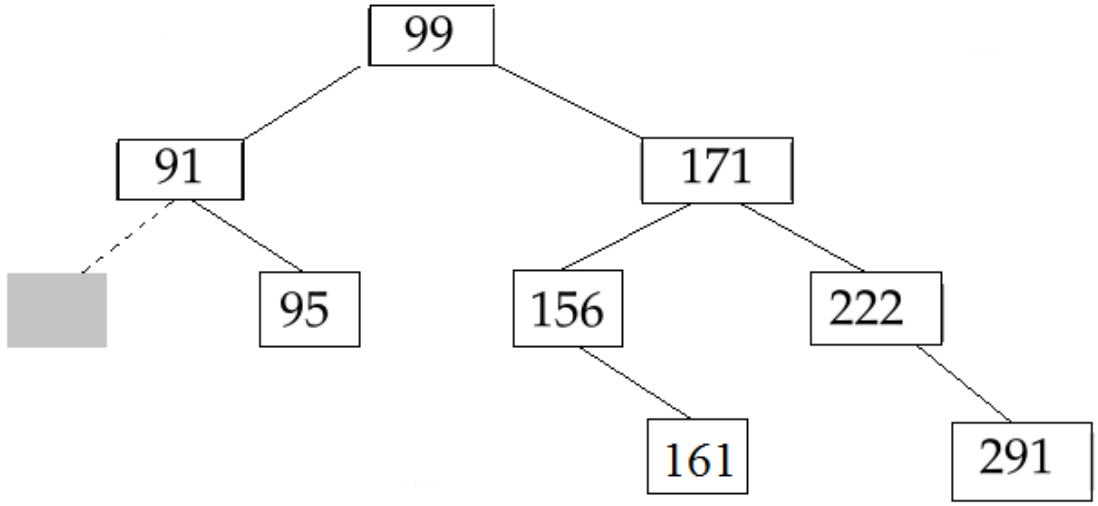
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(83)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

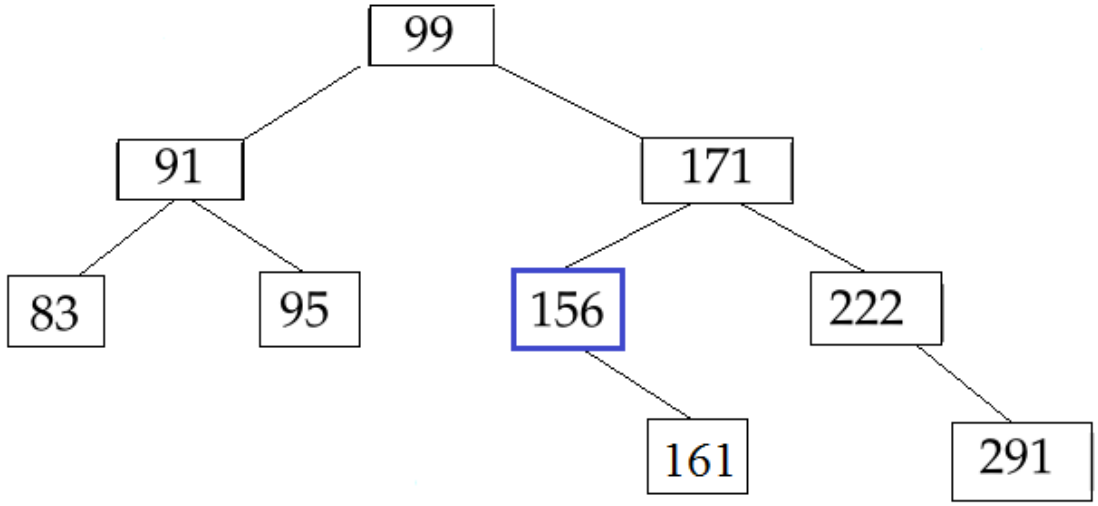
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(156)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

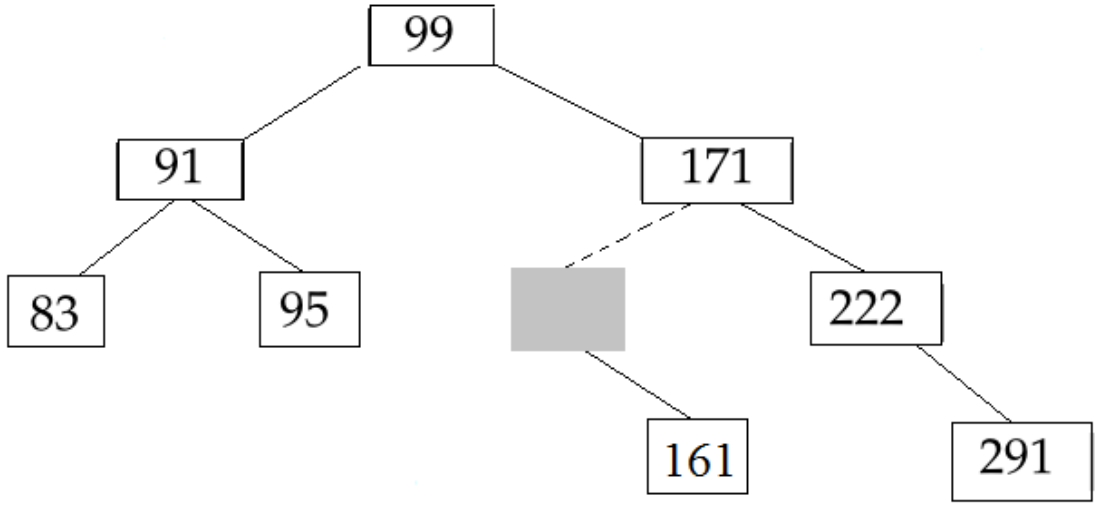
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(156)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

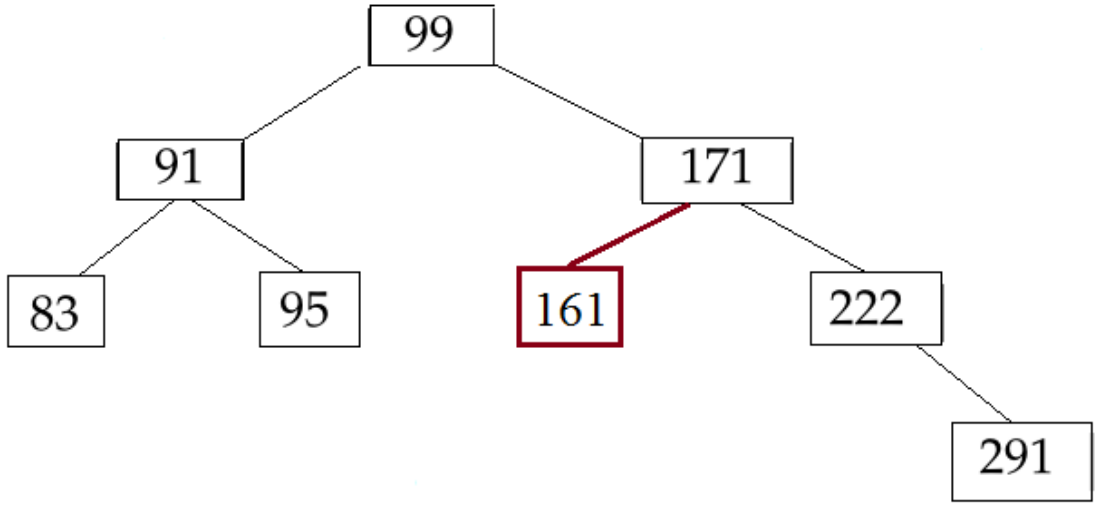
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(156)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

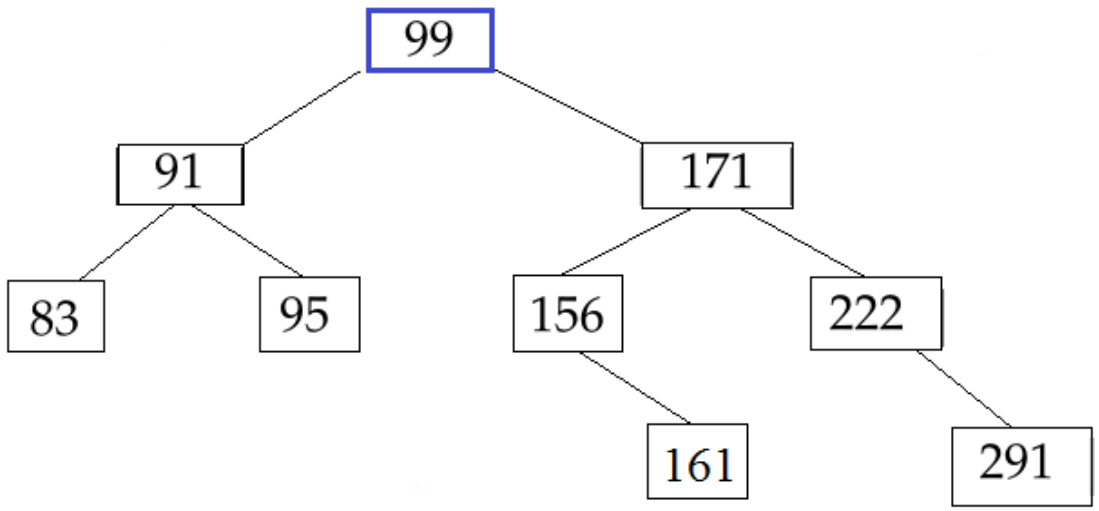
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(99)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

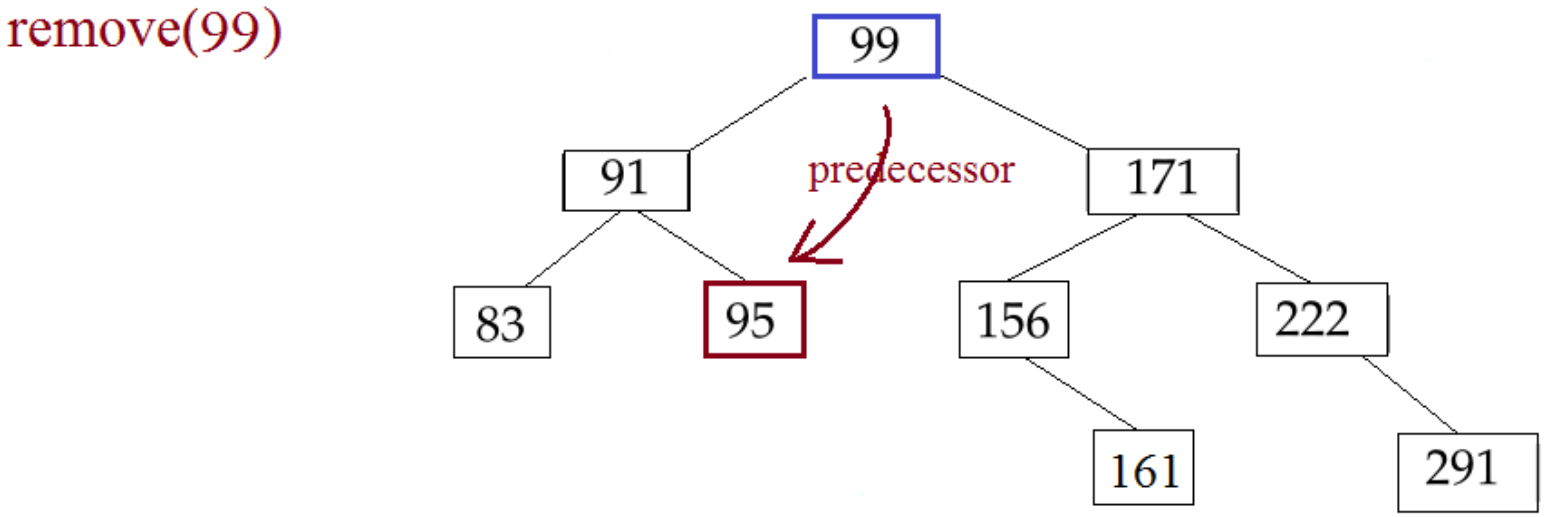
Remove it directly. No other changes needed.

Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

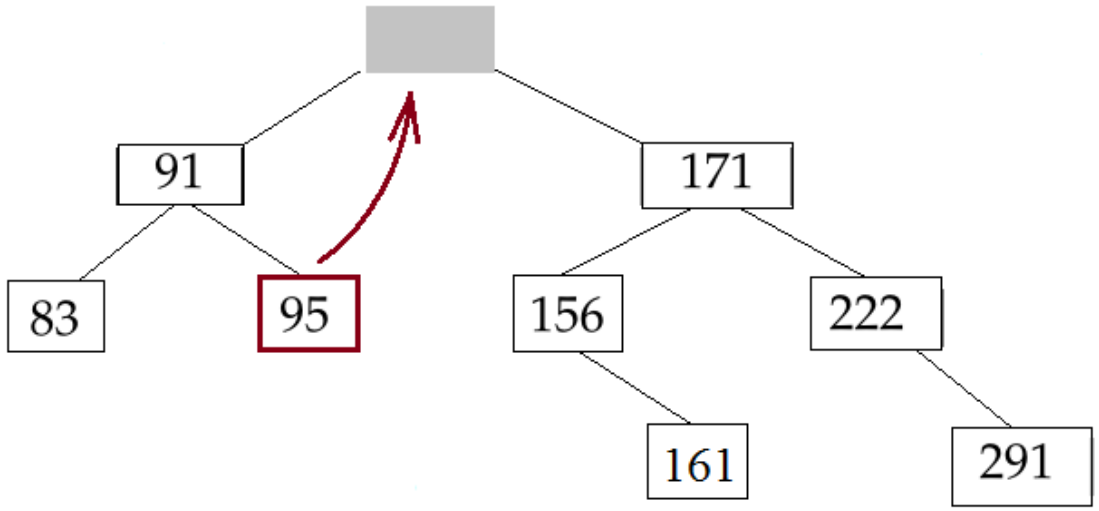
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(99)



Implementation 1: Simple Binary Search Trees

The Remove Operation

Idea: Suppose we want to remove node k

Case 1: k is a leaf.

Remove it directly. No other changes needed.

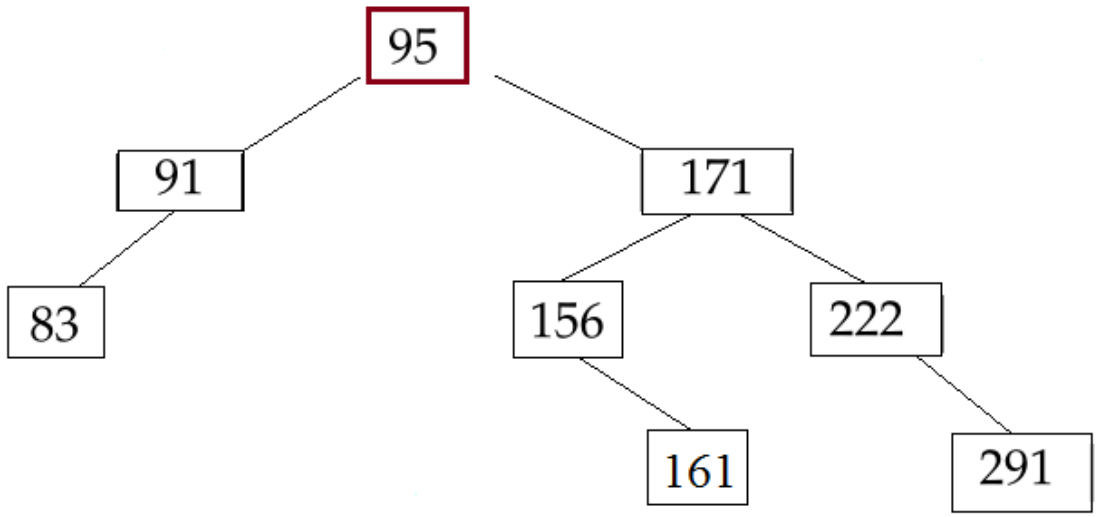
Case 2: k has only one child.

Let the child replace k.

Case 3: k has two children.

Let a's predecessor replace k.

remove(99)



Implementation 1: Simple Binary Search Trees

```
public BinaryTree<E> removeNode(BinaryTree<E> k){
// Return the resulting tree after k is removed
    if(k.left().isEmpty() && k.right().isEmpty())
        return EMPTY;
    else if(k.left().isEmpty())
        return k.right();
    else if(k.right().isEmpty())
        return k.left();
    else{
        BinaryTree<E> pre=predecessor(k);
        pre.parent().setRight(pre.left());
        BinaryTree<E> le=k.left();
        BinaryTree<E> ri=k.right();
        k.setLeft(EMPTY);
        k.setRight(EMPTY);
        pre.setLeft(le);
        pre.setRight(ri);
        return pre;
    }
}
```

Implementation 1: Simple Binary Search Trees

The Remove Operation

```
public void remove(E v){
//Remove the first occurrence of v
    if(!contains(v)) return;
    BinaryTree<E> locNode = locate(v);
    // Find the first occurrence of v
    if(locNode.equals(root))
        root=removeNode(locNode);
    else if(locNode.equals(locNode.parent().left())
        locNode.parent().setLeft(removeNode(locNode);
    else
        locNode.parent().setRight(removeNode(locNode);
}
```

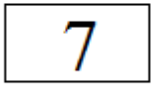
Implementation 1: Simple Binary Search Trees

A Run Through Example:

Implementation 1: Simple Binary Search Trees

A Run Through Example:

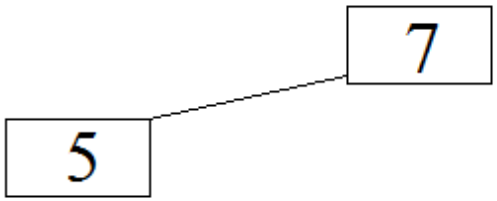
add(7)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

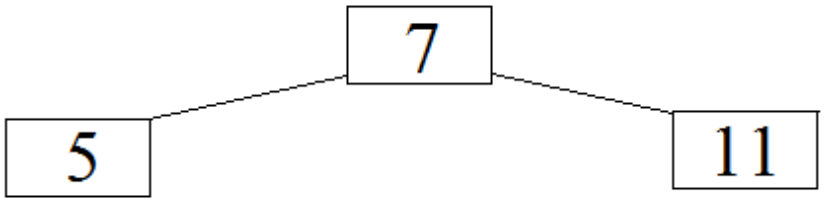
add(7) add(5)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

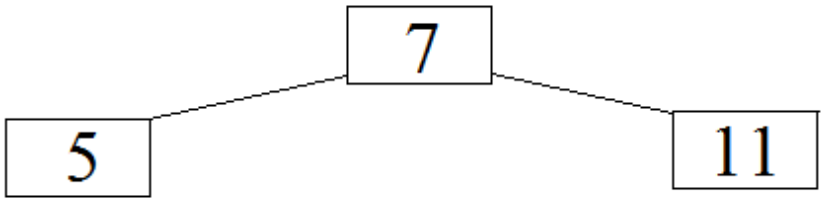
add(7) add(5) add(11)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

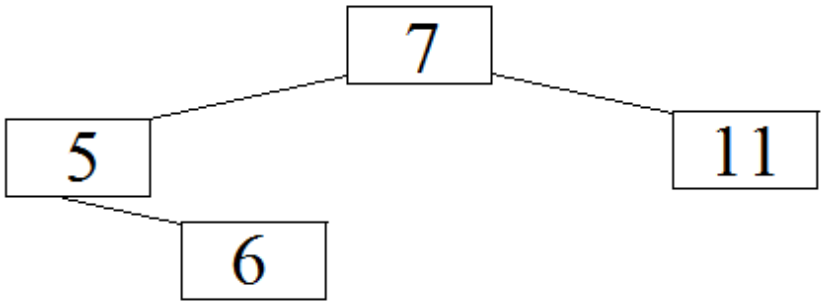
add(7) add(5) add(11) add(6)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

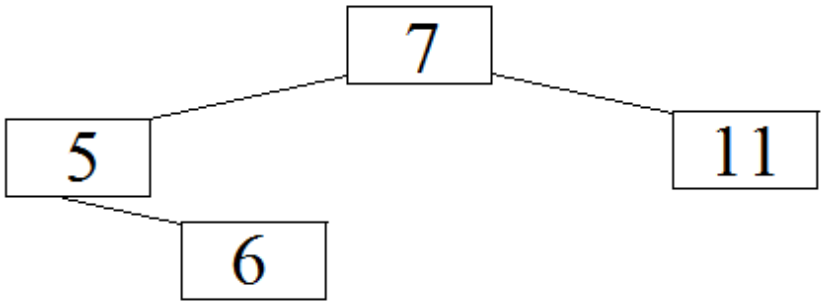
add(7) add(5) add(11) add(6)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

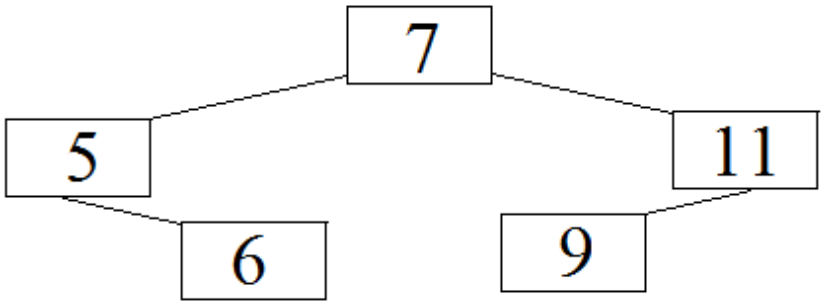
add(7) add(5) add(11) add(6) add(9)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

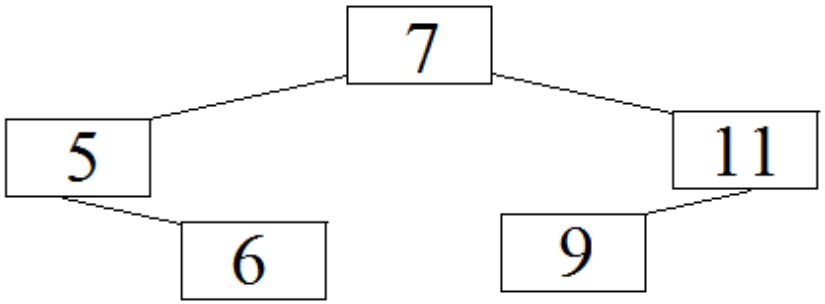
add(7) add(5) add(11) add(6) add(9)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

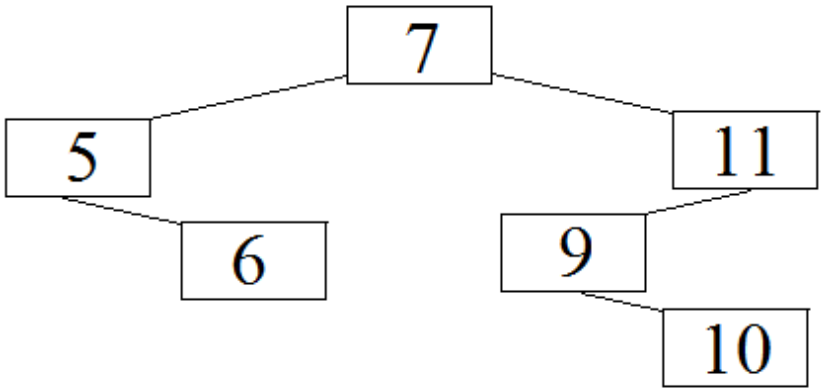
add(10)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

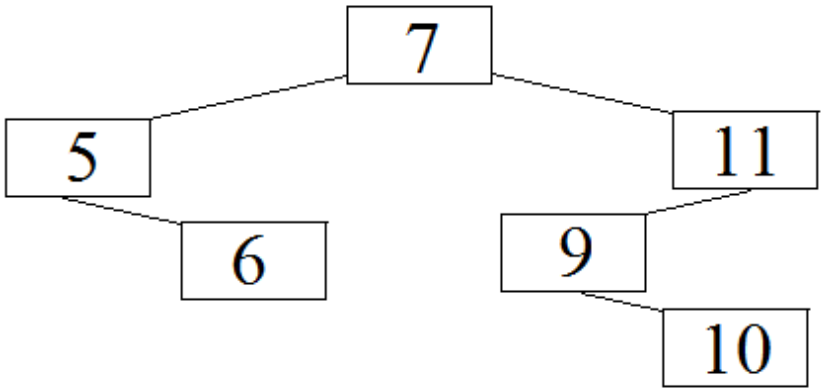
add(10)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

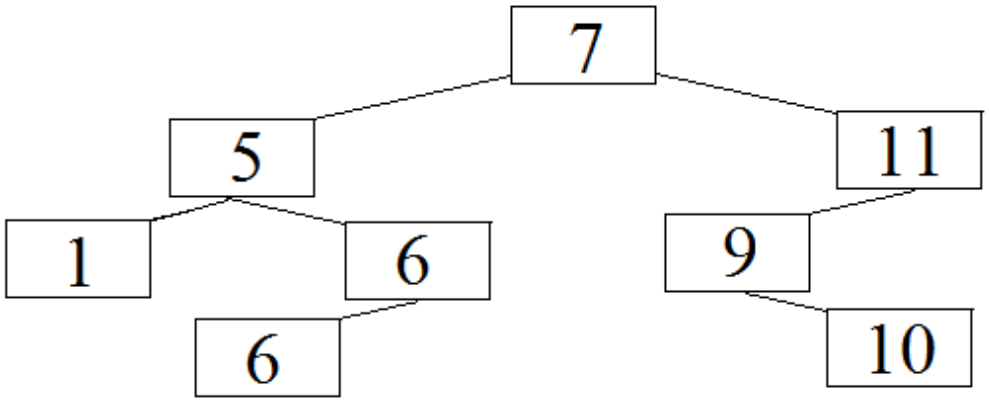
add(10) add(6)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

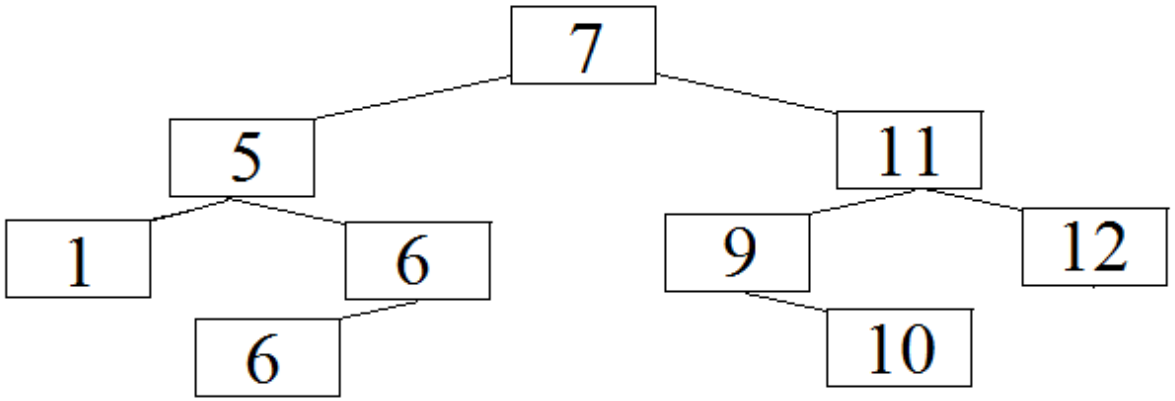
add(10) add(6) add(1)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

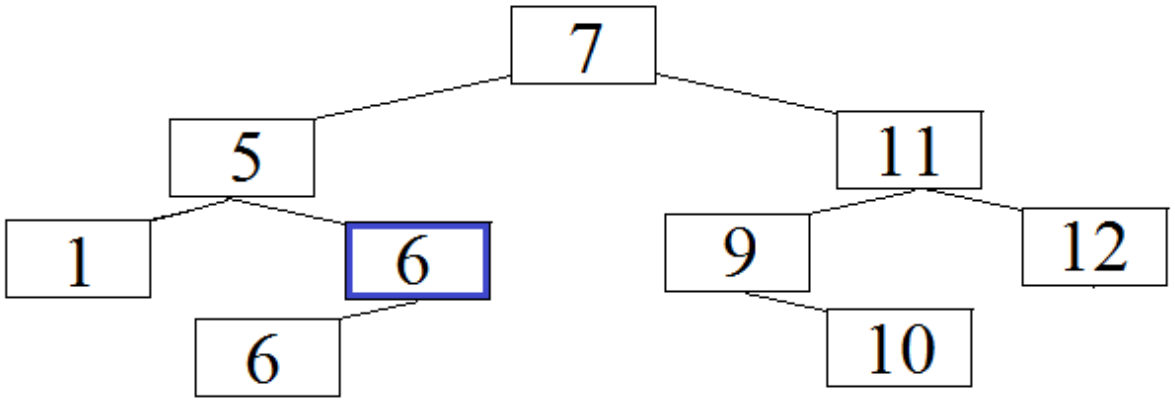
add(10) add(6) add(1) add(12)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

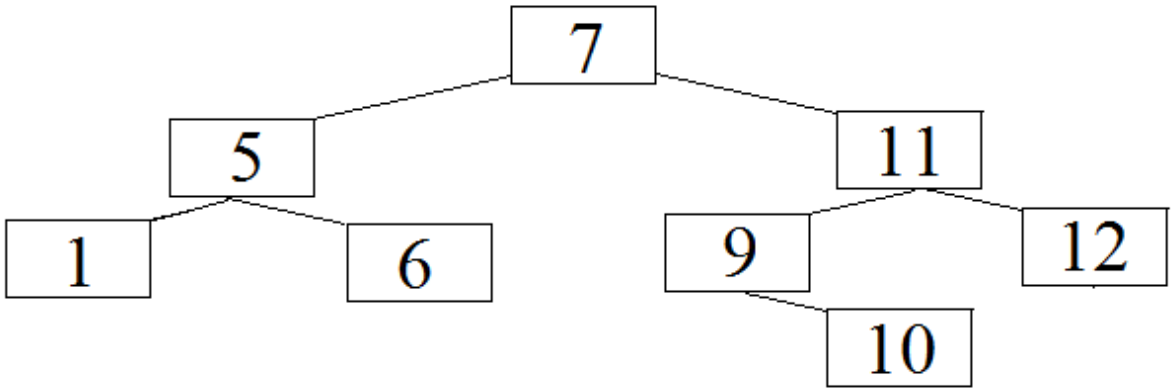
remove(6)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

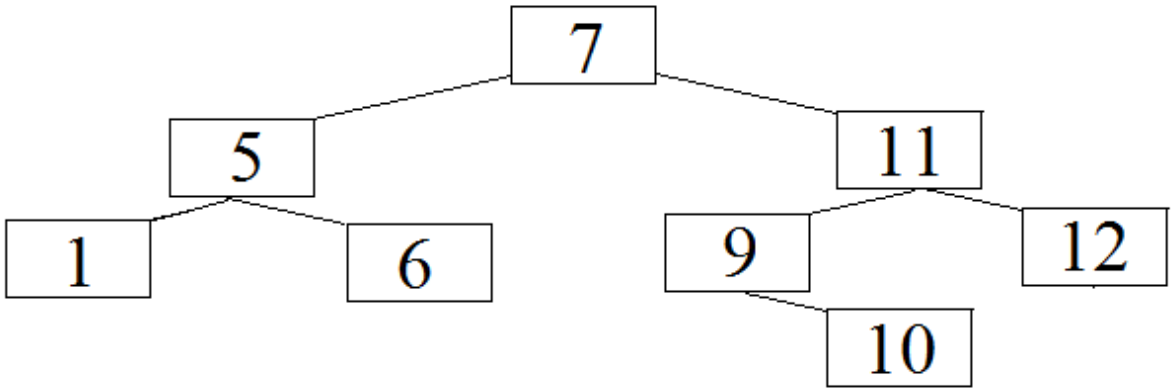
remove(6)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

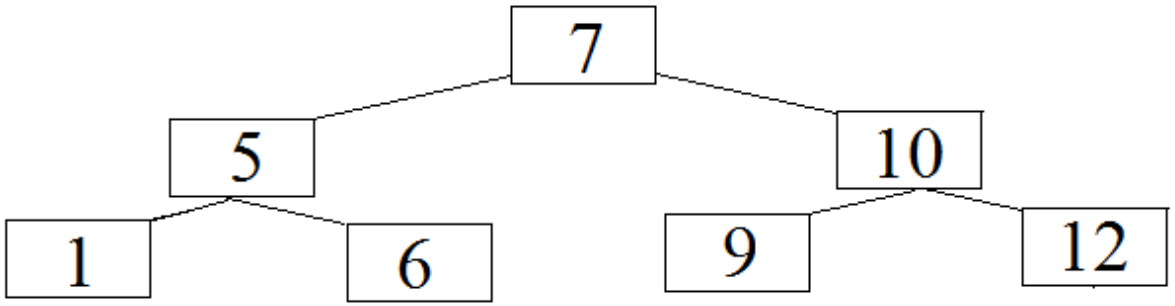
remove(6) remove(11)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

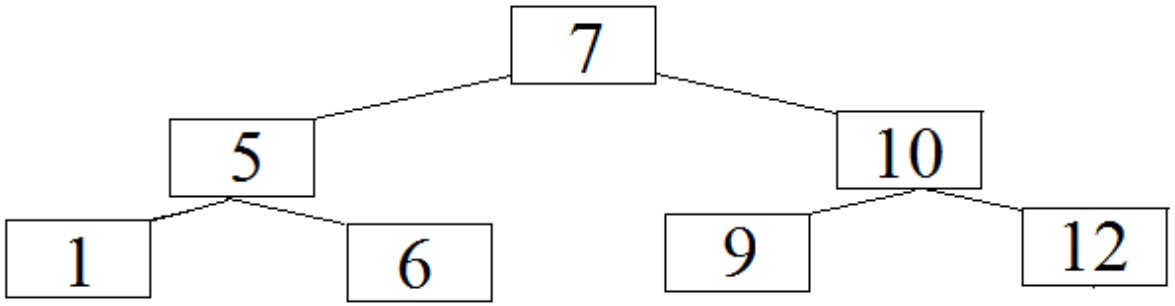
remove(6) remove(11)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

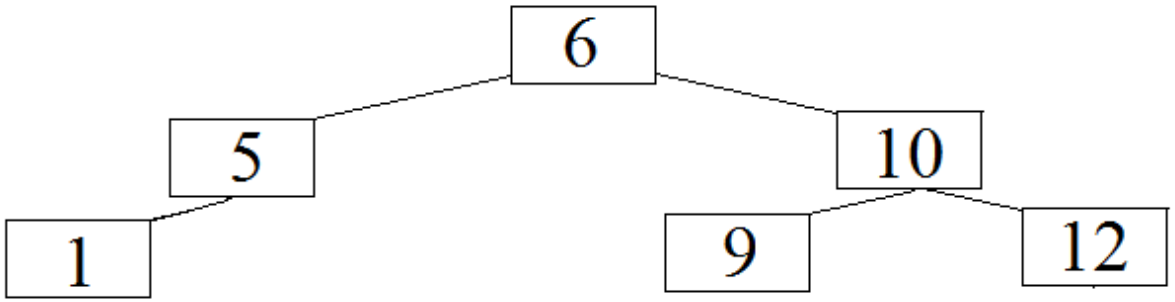
remove(6) remove(11) remove(7)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

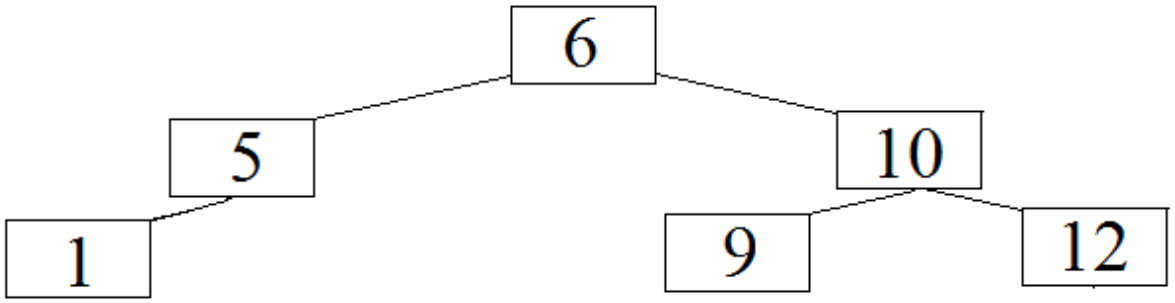
remove(6) remove(11) remove(7)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

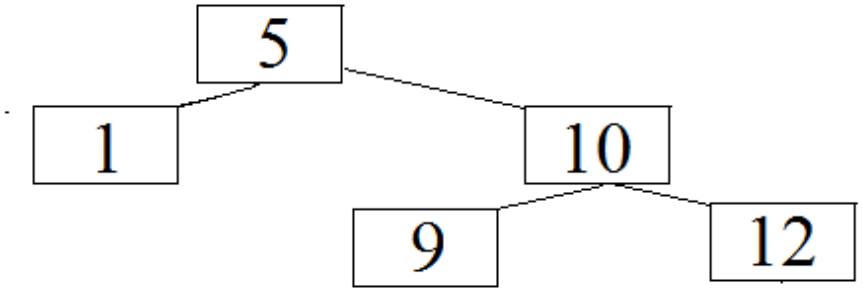
remove(6)



Implementation 1: Simple Binary Search Trees

A Run Through Example:

remove(6)



Implementation 1: Time Complexity

Observations:

- locate: Traverses down a path from the root (may reach a leaf)
- predecessor: Traverses down a path (may reach a leaf)
- add:
  - Calls locate
  - Calls predecessor
- remove:
  - Calls locate
  - Calls predecessor

Implementation 1: Time Complexity

Observations:

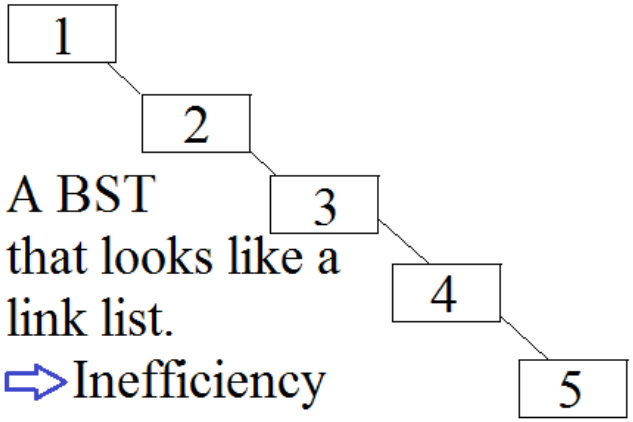
- locate: Traverses down a path from the root (may reach a leaf)
- predecessor: Traverses down a path (may reach a leaf)
- add:
  - Calls locate
  - Calls predecessor
- remove:
  - Calls locate
  - Calls predecessor

Therefore, all operations run in time $O(h)$ where h is the height of the BST.

Implementation 1: Time Complexity

Worst Case

All values are added in increasing (or decreasing) order:
`add(1), add(2), add(3), add(4), add(5),...`

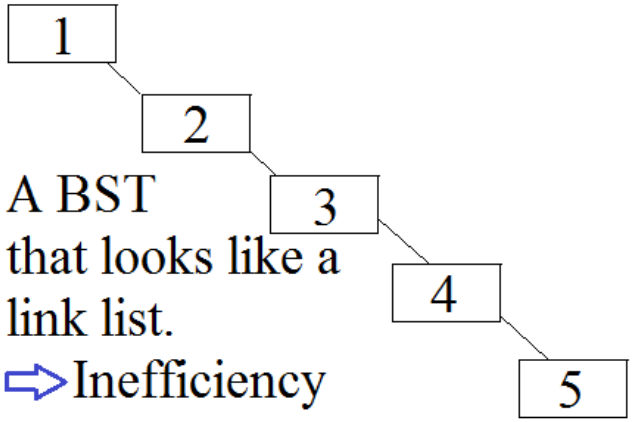


A BST that looks like a link list.
⇨ Inefficiency

Implementation 1: Time Complexity

Worst Case

All values are added in increasing (or decreasing) order:
`add(1), add(2), add(3), add(4), add(5),...`



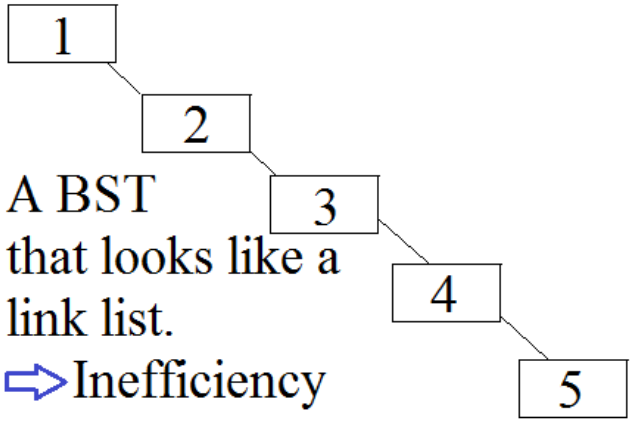
A BST that looks like a link list.
⇨ Inefficiency

The tree may become too thin, i.e., its height $h \approx n$

Implementation 1: Time Complexity

Worst Case

All values are added in increasing (or decreasing) order:
`add(1), add(2), add(3), add(4), add(5),...`



A BST that looks like a link list.
⇨ Inefficiency

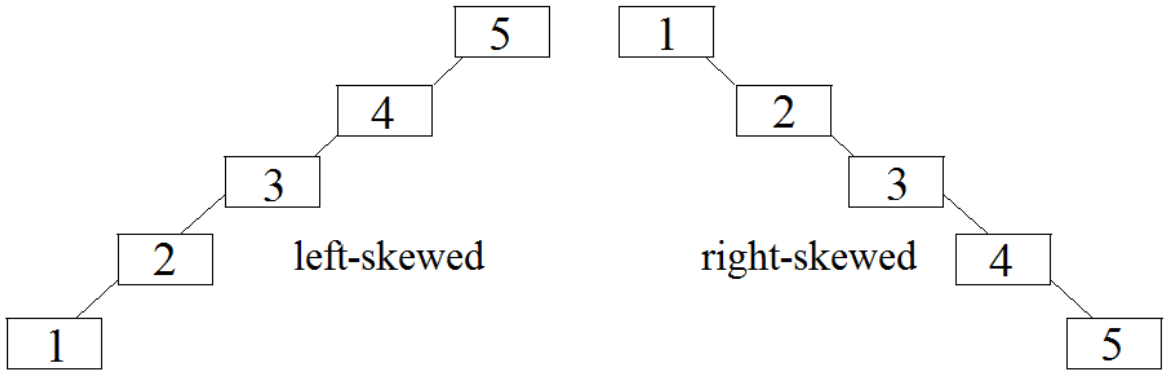
The tree may become too thin, i.e., its height $h \approx n$
Worst Case Time Complexity: $O(n)$

Part III: Splay Trees

Implementation 2: Splay Trees

Observation.

- The worst case of a BST occurs when the tree is unbalanced.



- The query operation `contains(E v)` does not modify the BST.
- To improve efficiency, we can allow operations to modify the structure of the BST so that it becomes balanced.

Self-Balancing BST

Goal: Design a BST data structure that can automatically re-balance itself.
Such BST are called self-balancing BST.

Self-Balancing BST

Goal: Design a BST data structure that can automatically re-balance itself.
Such BST are called self-balancing BST.

Splay Trees

A splay tree is a special type of BST that is self-balancing. The data structure uses a method called splay to re-balance itself during every query and update.

Characteristics:

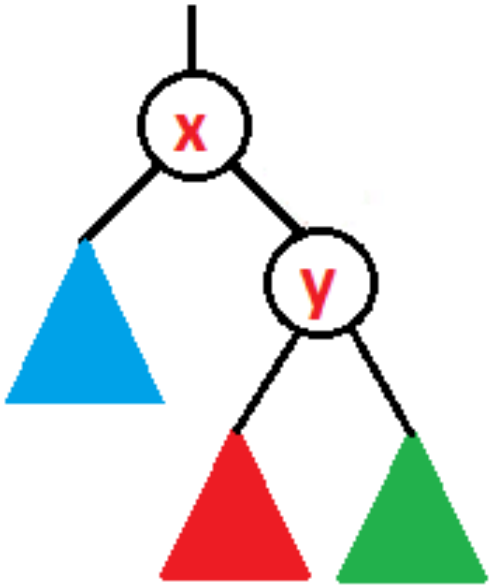
- Worst Case: Still bad $O(n)$, but rarely occur
- Average Case: Good $O(\log n)$.

Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

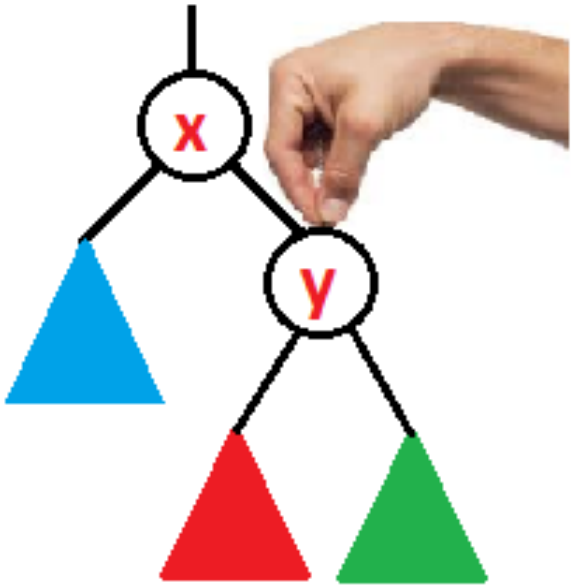


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

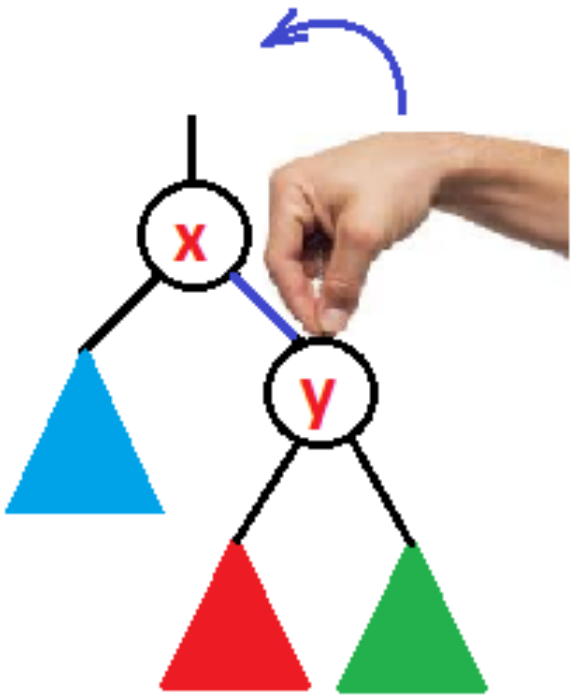


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

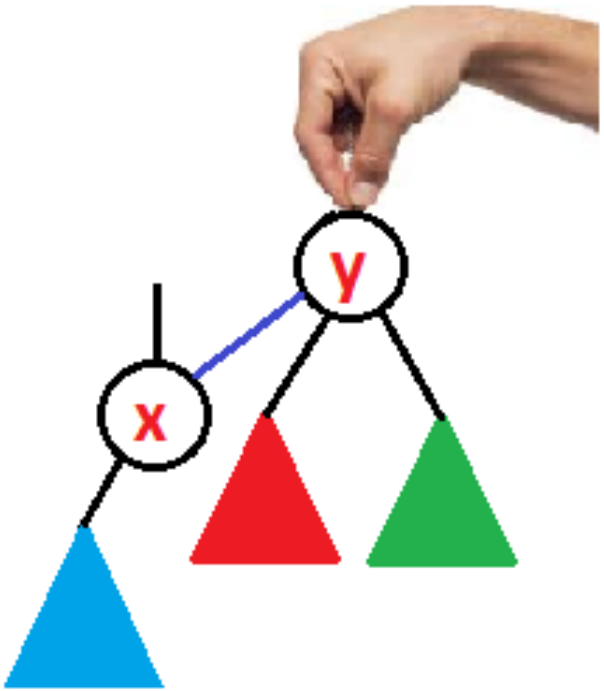


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

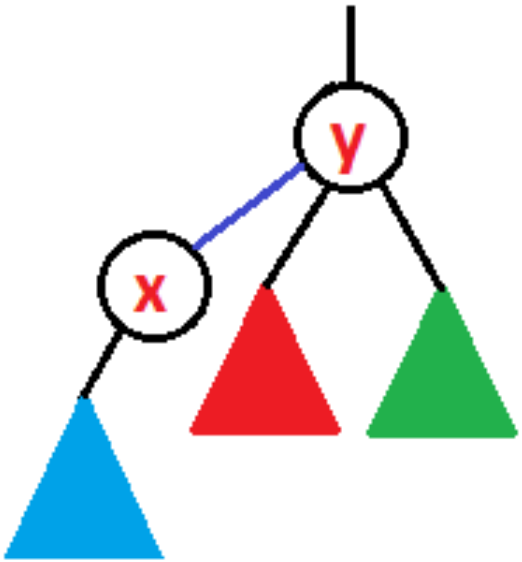


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

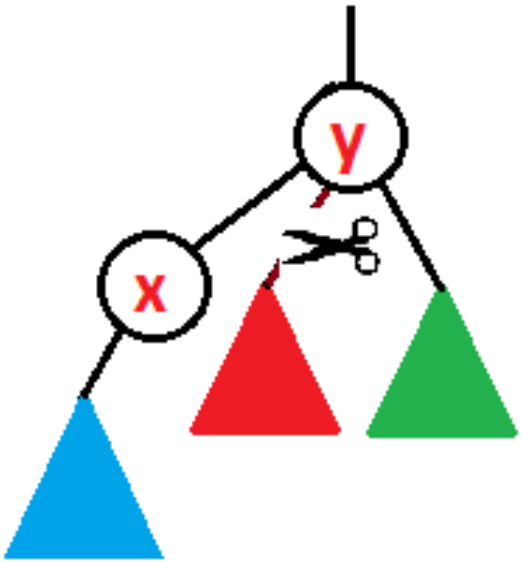


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

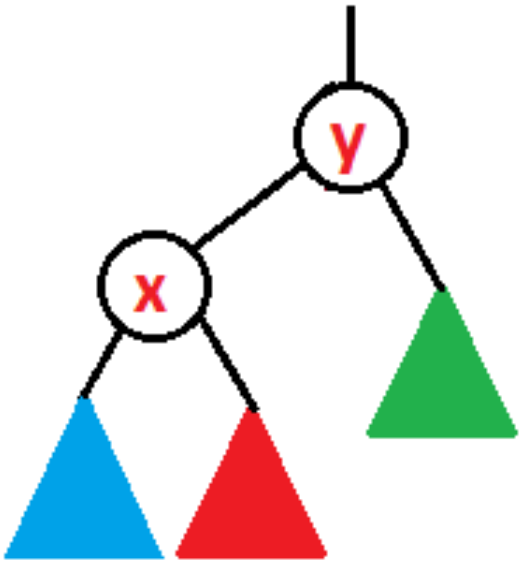


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():

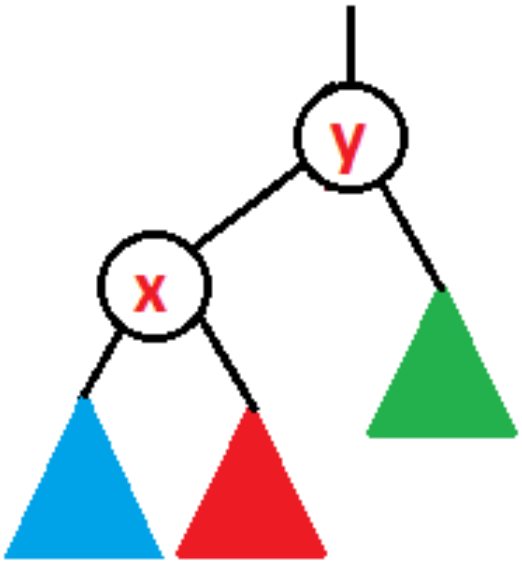


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate():



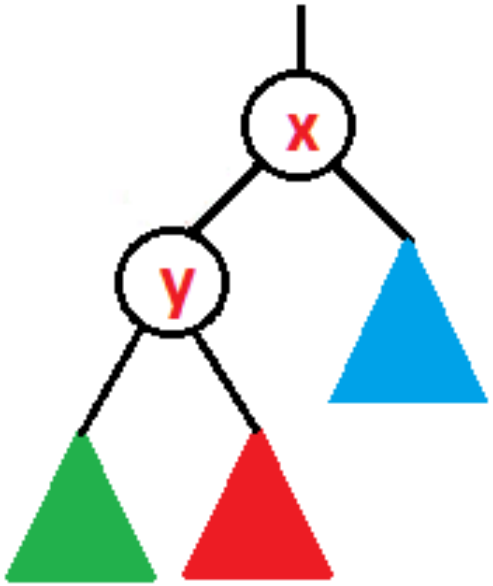
- Make y the parent of x
- Make the old left child of y the new right child of x.

Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

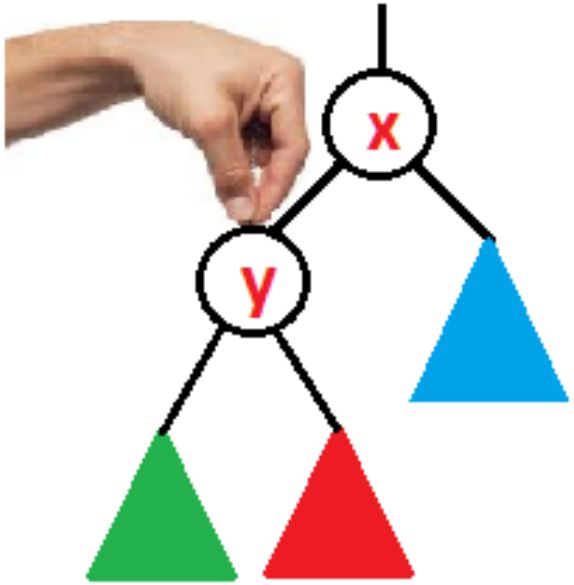


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

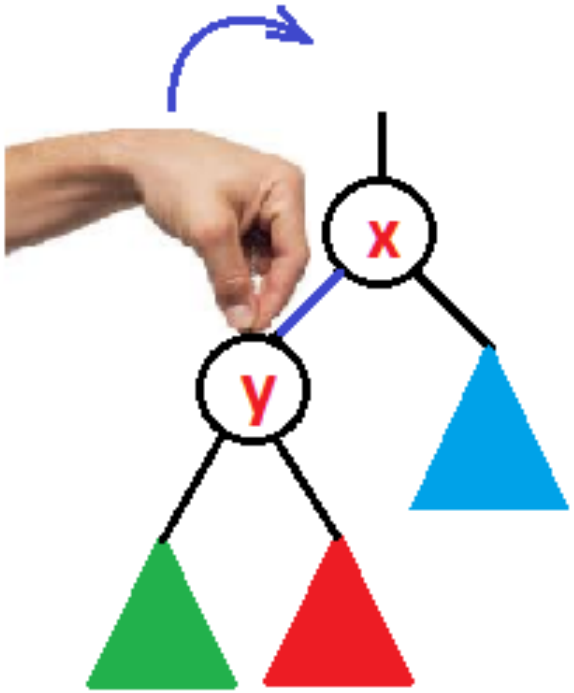


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

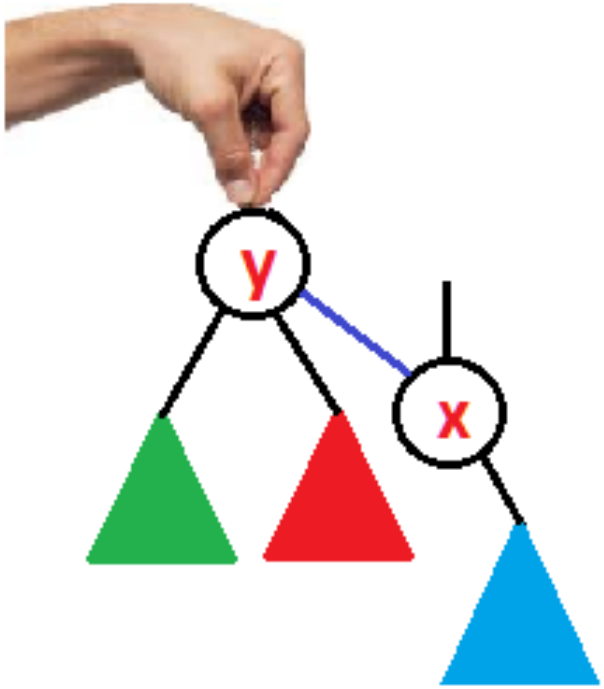


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

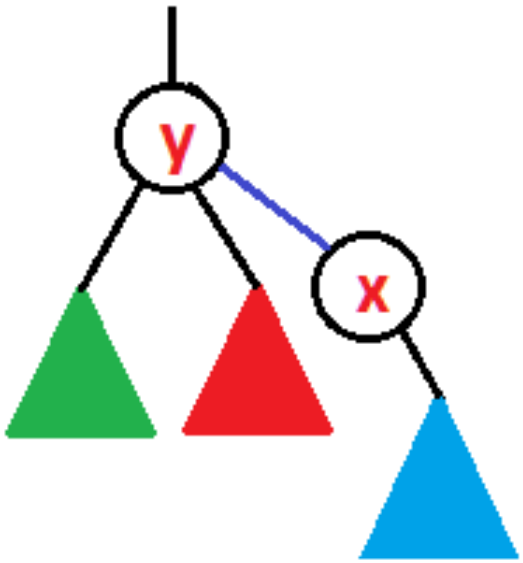


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

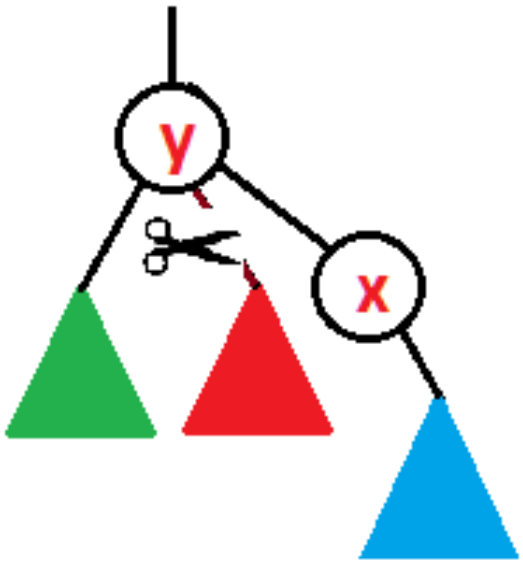


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

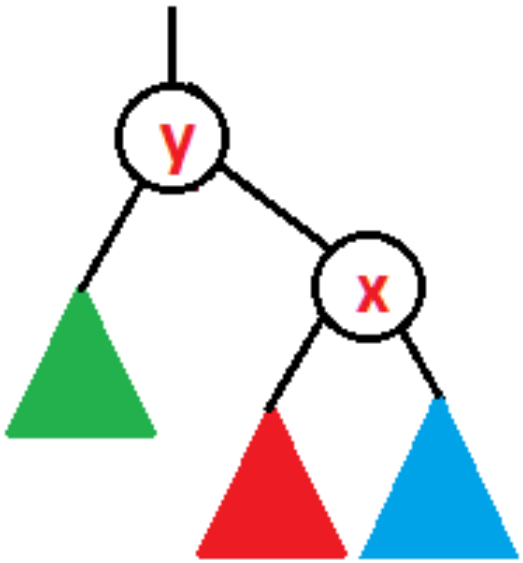


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():



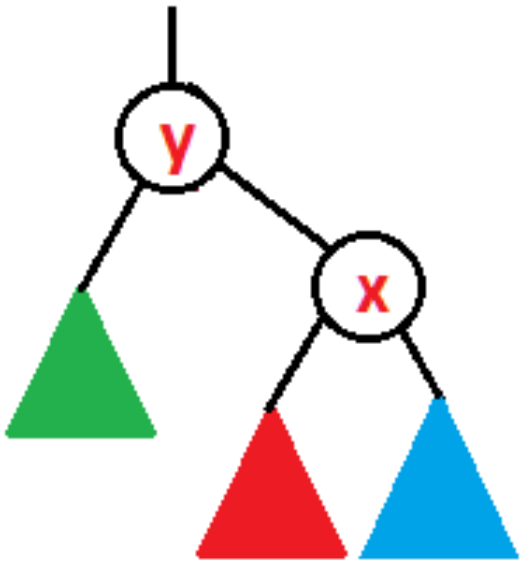
Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.RightRotate():

- Make y the parent of x
- Make the old right child of y the new left child of x.

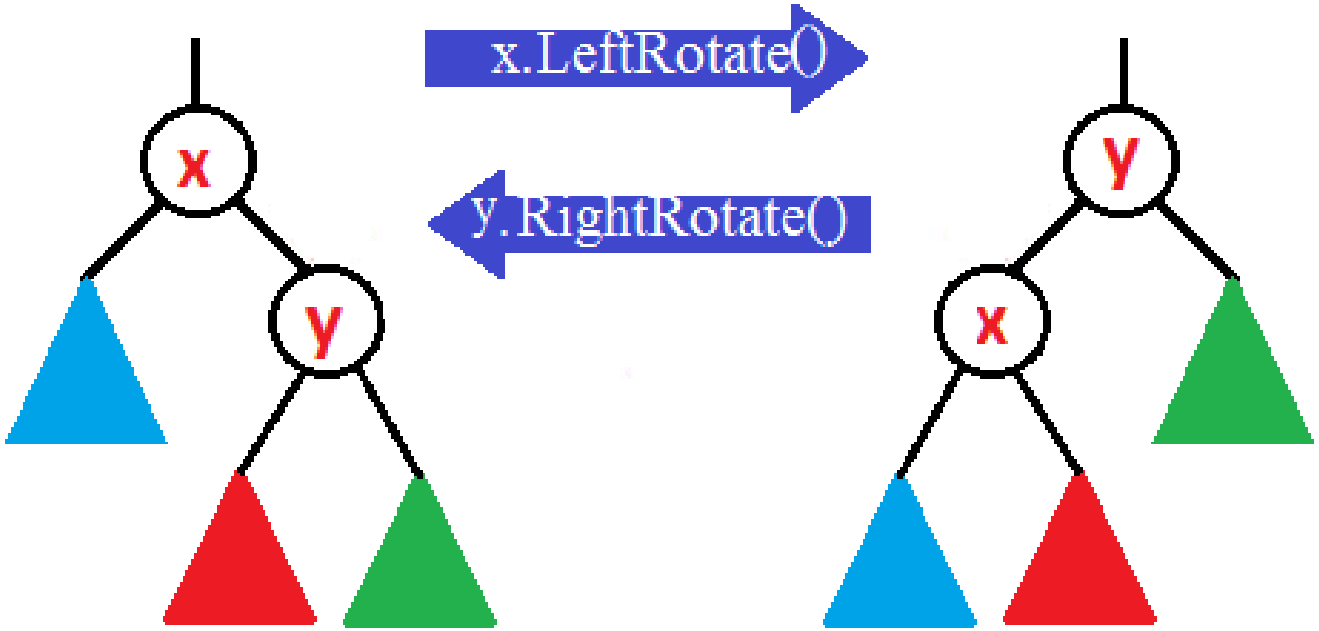


Implementation 2: Splay Trees

Tree Rotations

Idea. Change the parent/children relation without affecting the BST property.

x.LeftRotate() and y.RightRotate():



We often say "rotate up" a node, rather than "left/right rotate" its parent.

Implementation 2: Splay Trees

Tree Rotations

```
protected void rotateRight(){
// pre:  this node has a left subtree
// post:  rotates left child up

    BinaryTree<E> parent= parent();
    BinaryTree<E> newRoot=left();
    BinaryTree<E> oldRight=newRoot.right();
    boolean wasChild  = parent!=null;
    boolean wasLeftChild = isLeftChild();

    newRoot.setRight(this);
    setLeft(oldRight);
    if (wasChild){
        if(wasLeftChild) parent.setLeft(newRoot);
        else parent.setRight(newRoot);
    }
}
```

Implementation 2: Splay Trees

The Splay Operation

Idea: Rotate a node x upwards until it becomes the root.
- Make the tree become more balanced.

Implementation 2: Splay Trees

The Splay Operation

Idea: Rotate a node x upwards until it becomes the root.
- Make the tree become more balanced.

Pseudocode: splay(x)

Repeat the following until x is the root:
- if x has no grandparent, rotate x up
- else there are two cases:
  - Case 1: x and its parent are both left children or both right children.

    First rotate parent up; then rotate x up

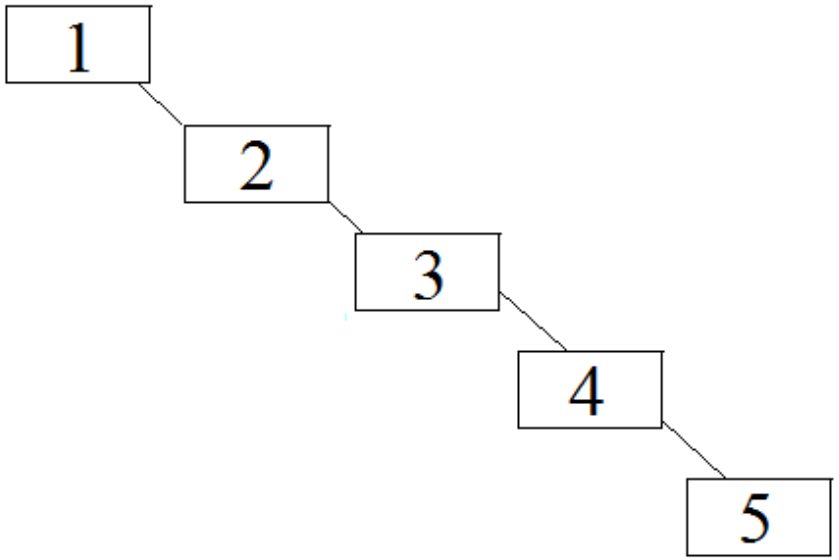
  - Case 2: one of x and its parent is a left child, the other is right child.

    First rotate x up; then rotate x up again

Implementation 2: Splay Trees

The Splay Operation

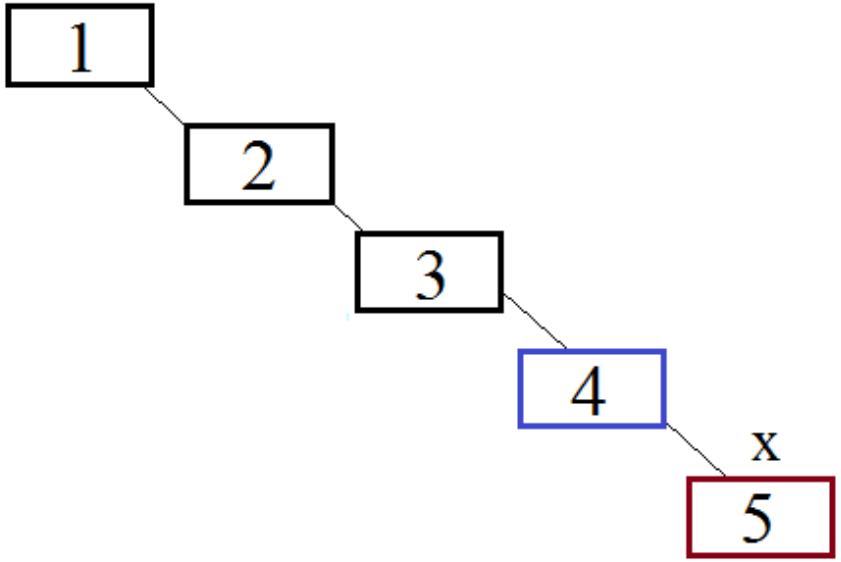
Example: Splay(5)



Implementation 2: Splay Trees

The Splay Operation

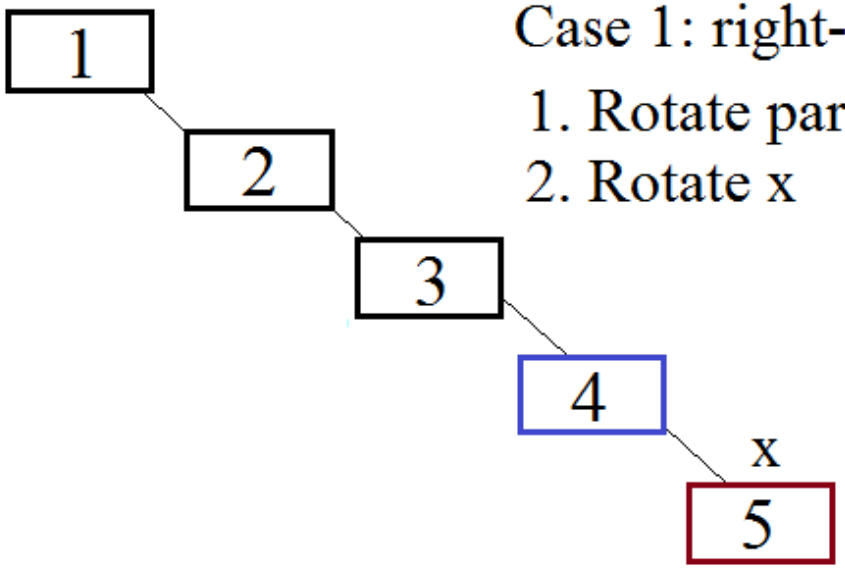
Example: Splay(5)



Implementation 2: Splay Trees

The Splay Operation

Example: Splay(5)



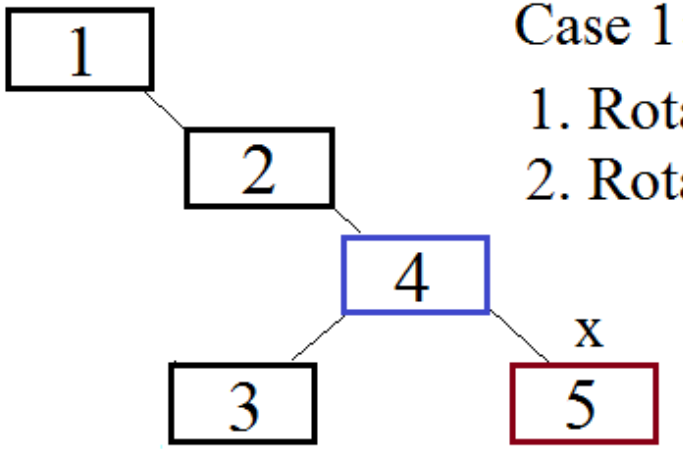
Case 1: right-right

1. Rotate parent
2. Rotate x

Implementation 2: Splay Trees

The Splay Operation

Example: Splay(5)



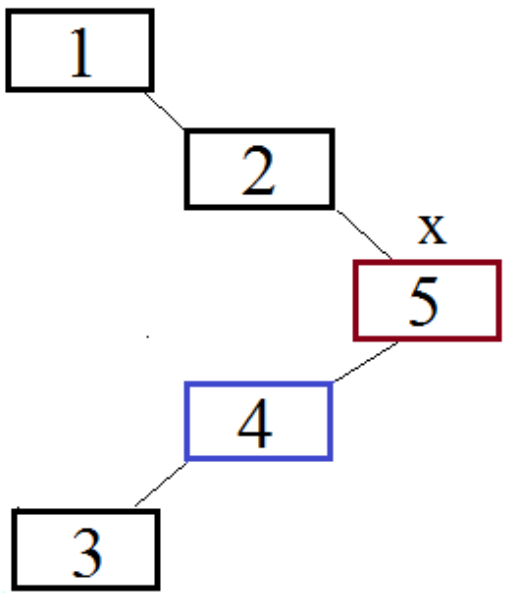
Case 1: right-right

1. Rotate parent
2. Rotate x

Implementation 2: Splay Trees

The Splay Operation

Example: Splay(5)



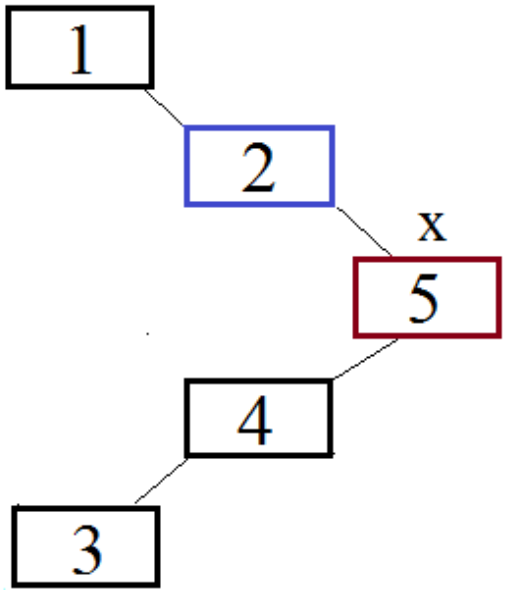
Case 1: right-right

1. Rotate parent
2. Rotate x

Implementation 2: Splay Trees

The Splay Operation

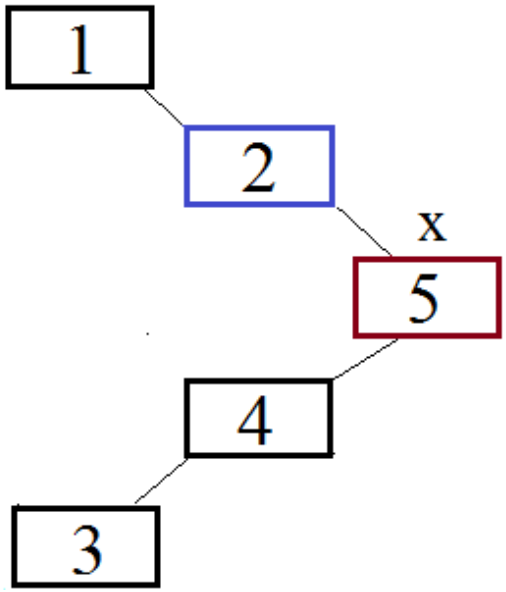
Example: Splay(5)



Implementation 2: Splay Trees

The Splay Operation

Example: Splay(5)



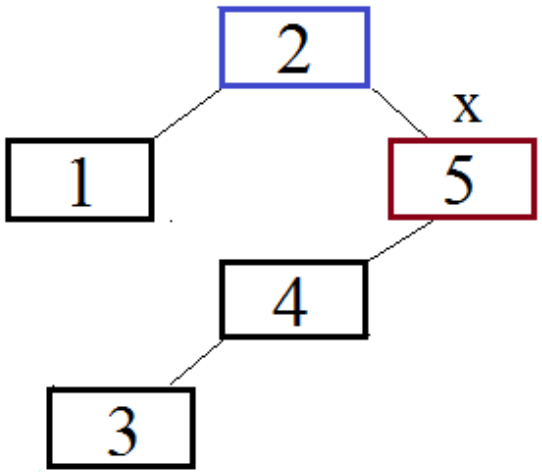
Case 1: right-right

1. Rotate parent
2. Rotate x

Implementation 2: Splay Trees

The Splay Operation

Example: Splay(5)



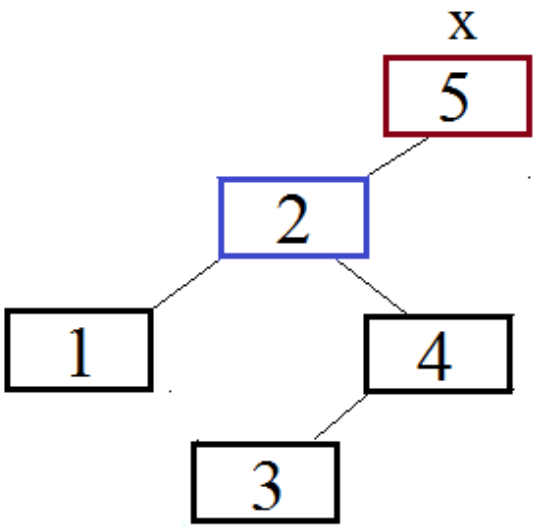
Case 1: right-right

1. Rotate parent
2. Rotate x

Implementation 2: Splay Trees

The Splay Operation

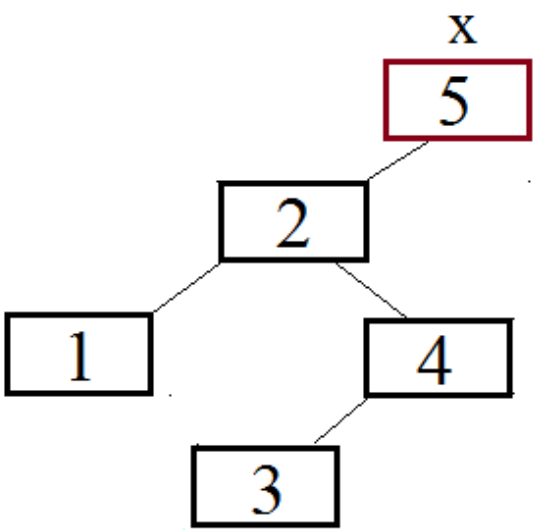
Example: Splay(5)



Implementation 2: Splay Trees

The Splay Operation

Example: Splay(5)

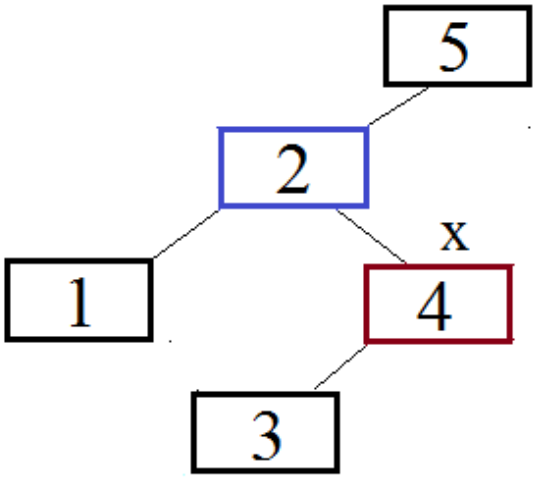


Terminate

Implementation 2: Splay Trees

The Splay Operation

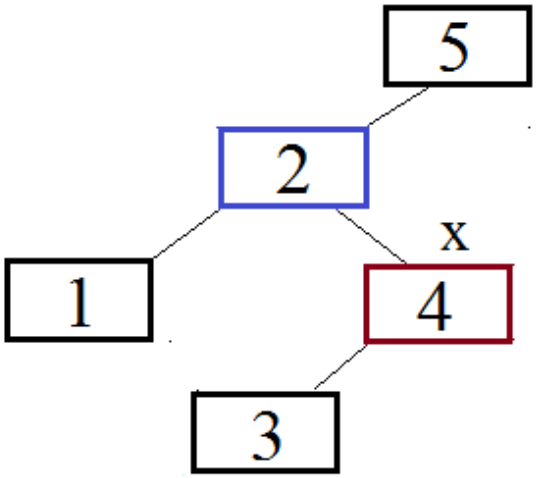
Example: Splay(4)



Implementation 2: Splay Trees

The Splay Operation

Example: Splay(4)



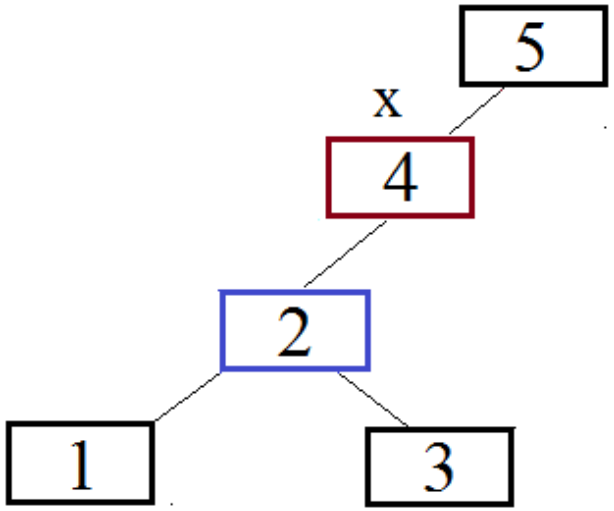
Case 2: right-left

1. Rotate x
2. Rotate x again

Implementation 2: Splay Trees

The Splay Operation

Example: Splay(4)



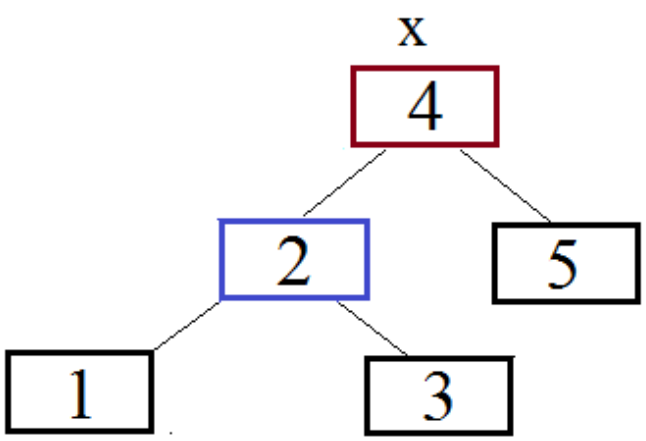
Case 2: right-left

1. Rotate x
2. Rotate x again

Implementation 2: Splay Trees

The Splay Operation

Example: Splay(4)



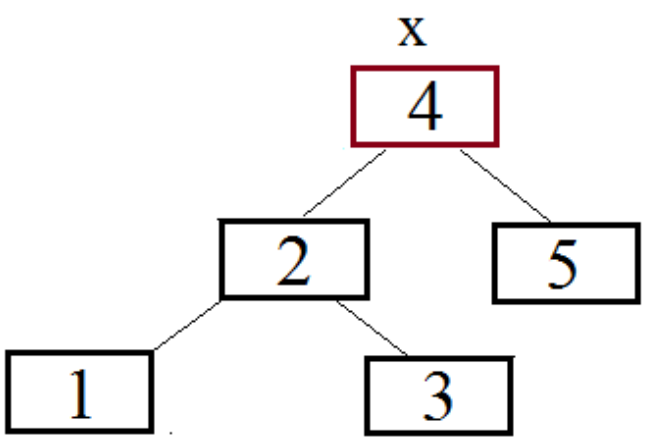
Case 2: right-left

1. Rotate x
2. Rotate x again

Implementation 2: Splay Trees

The Splay Operation

Example: Splay(4)

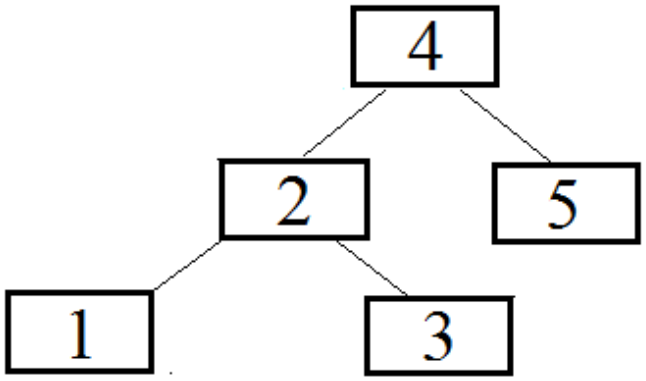


Terminate

Implementation 2: Splay Trees

The Splay Operation

Example:



Note: After splay(x)
- x becomes the root
- the tree becomes more balanced

Implementation 2: Splay Trees

Splay Trees: Implementation

```
public class SplayTree<E extends Comparable<E>>
    extends BinarySearchTree<E> {
    public SplayTree()
    public void add(E val)
    public boolean contains(E val)
    public E get(E val)
    public E remove(E val)
    protected void splay(BinaryTree<E> x)
    public Iterator<E> iterator()
}
```

Implementation 2: Splay Trees

Splay Trees: Implementation

```
public boolean contains(E val){
    if (root.isEmpty()) return false;
    BinaryTree<E> loc=locate(root, val);
    if(val.equals(loc.value())){
        splay(loc);
        root=loc;
    } else return false;
}
```

Implementation 2: Splay Trees

Splay Trees: Implementation

- add(v):
  - First add v as in a simple binary search tree.
  - Then splay v to the root
- remove(v):
  - First splay v to the root
  - Then delete v as in a simple binary search tree

Note: Splay is performed during every add/remove/contains/get.

Implementation 2: Splay Trees

Splay Trees: Time Complexity

- Disadvantage: Splay trees do not improve the worst case time complexity:

  If inserting values in increasing order, the BST still will look like a linked list.

  The tree still may be very unbalanced.

Implementation 2: Splay Trees

Splay Trees: Time Complexity

- Disadvantage: Splay trees do not improve the worst case time complexity:

  If inserting values in increasing order, the BST still will look like a linked list.

  The tree still may be very unbalanced.

- Advantage:

  - Last added value can be quickly accessed

  - In a sequence of operations, the worst case cost can be averaged out; so the total time for any sequence of operations is small.

Implementation 2: Splay Trees

Splay Trees: Time Complexity

- Disadvantage: Splay trees do not improve the worst case time complexity:

  If inserting values in increasing order, the BST still will look like a linked list.

  The tree still may be very unbalanced.

- Advantage:

  - Last added value can be quickly accessed

  - In a sequence of operations, the worst case cost can be averaged out; so the total time for any sequence of operations is small.

  $\Rightarrow$ For a sequence of m add/remove/contains/get operations, a splay tree takes time $O(m \log n)$.

Implementation 2: Splay Trees

Splay Trees: Time Complexity

- Disadvantage: Splay trees do not improve the worst case time complexity:

  If inserting values in increasing order, the BST still will look like a linked list.

  The tree still may be very unbalanced.

- Advantage:

  - Last added value can be quickly accessed

  - In a sequence of operations, the worst case cost can be averaged out; so the total time for any sequence of operations is small.

  $\Rightarrow$ For a sequence of m add/remove/contains/get operations, a splay tree takes time $O(m \log n)$.

  $\Rightarrow$ The average time for each individual operation is $O(\log n)$.

Part IV: Red-Black Trees

Implementation 3: Red-Black Trees

Red-Black Tree

A red-black tree is a self-balancing binary search tree data structure for storing elements that are taken from an ordered domain and facilitates search, insert and delete operations.

Node

A node x in the tree contains the following fields:

$$color(x),\ key(x),\ left(x),\ right(x), p(x)$$

There is a special type of node called sentinel, whose every field is *null*; the sentinel is denoted by *NIL*.

Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

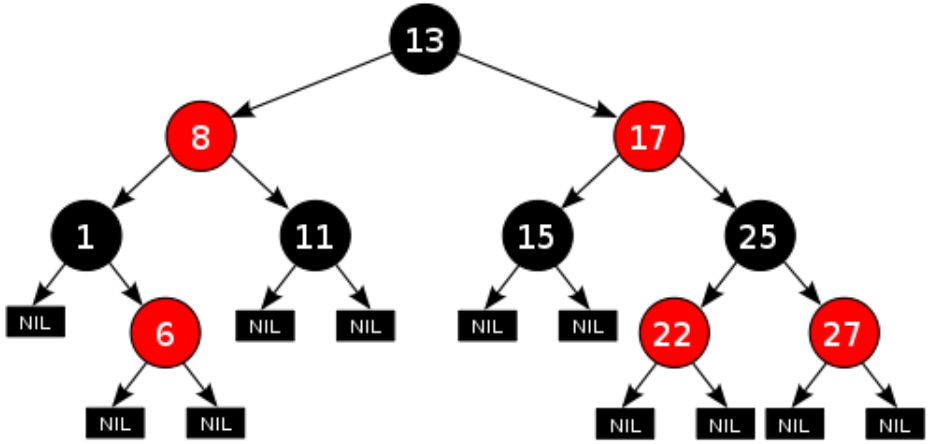
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree?



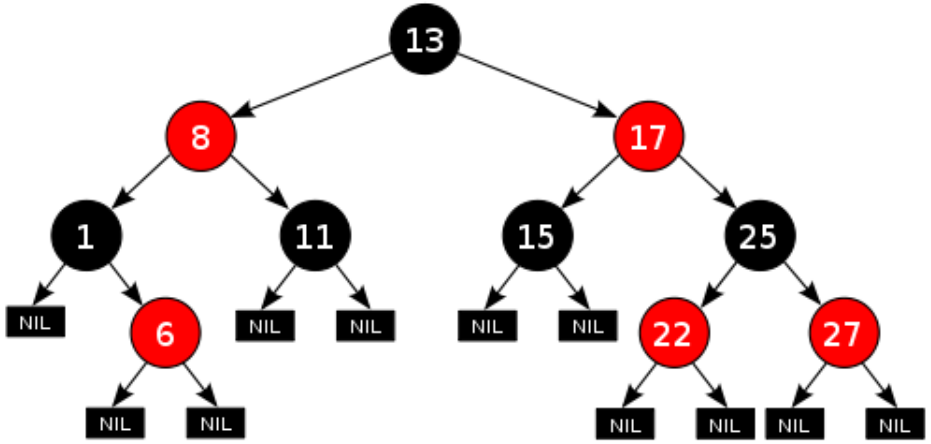
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree? Yes!



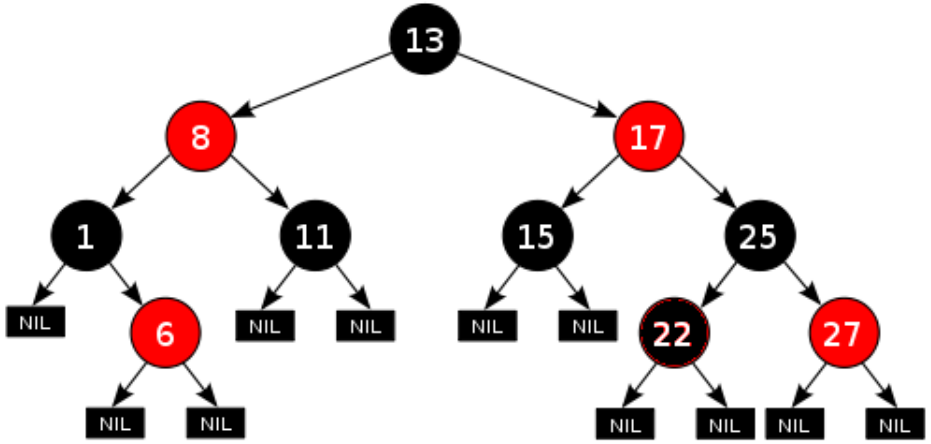
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree?



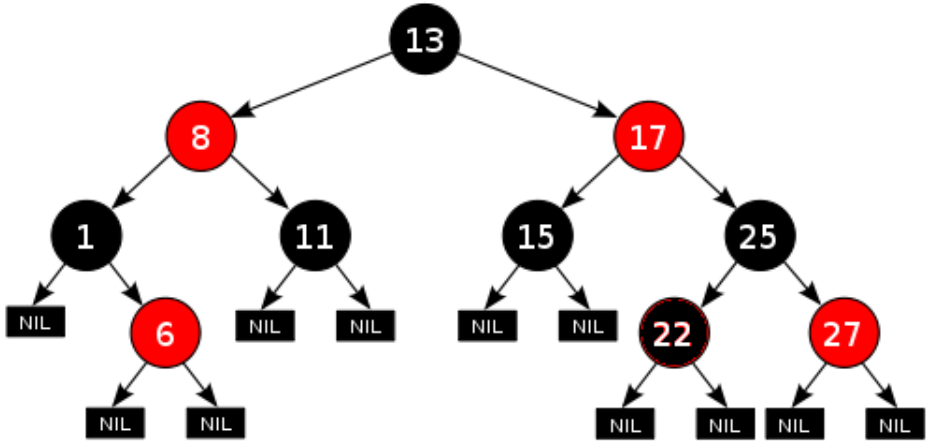
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree? No!



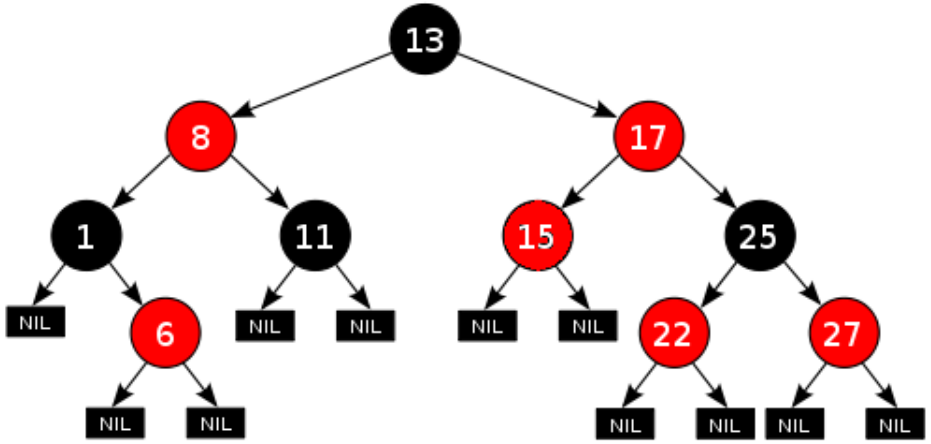
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree?



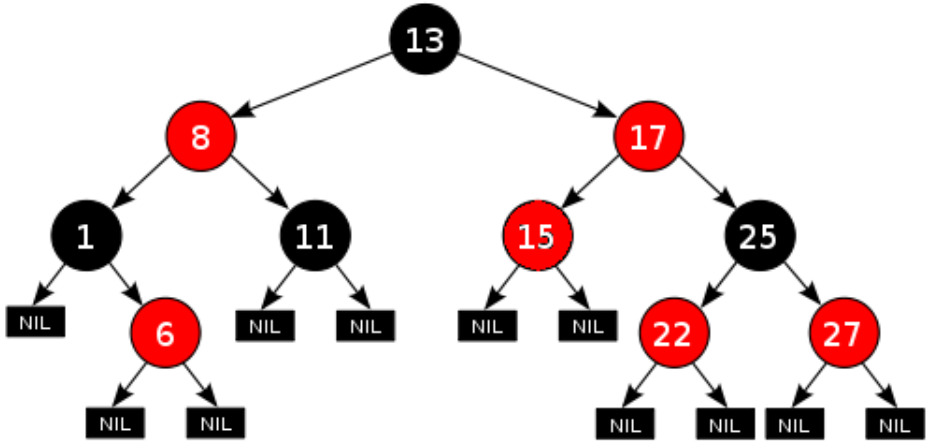
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree? No!



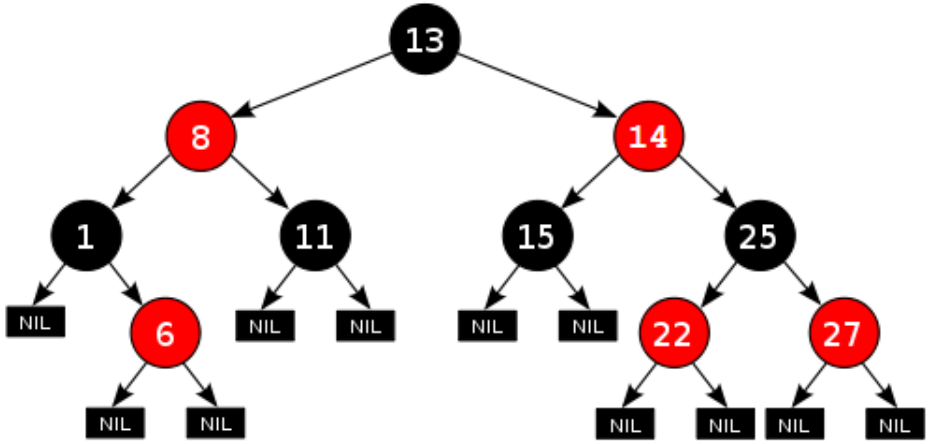
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree?



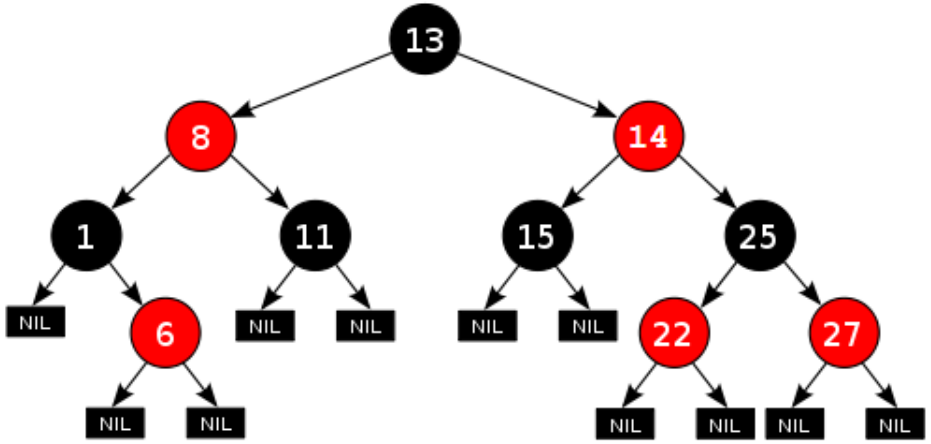
Implementation 3: Red-Black Trees

Definition

A red-black tree is a full binary search tree where every node is either red or black, and:

- (Black root) The root is black
- (Black leaves) Every leaf is *NIL* and is black
- (Red node property) If a node is red, then both of its children are black
- (Balance property) For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Is this a red-black tree? No!

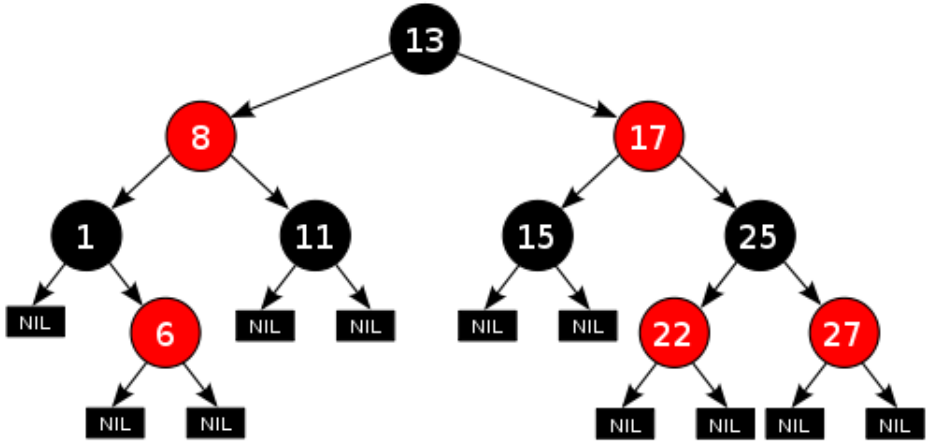


Implementation 3: Red-Black Trees

Black Height

- For a node x in a red-black tree, we call the number of black nodes on any path from (exclusively) x to a leaf the black-height of x, denoted by $bh(x)$.
- The black-height of the tree is the black height of the root.

Example:



The black height of the above tree is 2.

Implementation 3: Red-Black Trees

Claim. The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Implementation 3: Red-Black Trees

Claim. The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof. We prove the claim by induction on the height of x.

Implementation 3: Red-Black Trees

Claim. The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof. We prove the claim by induction on the height of x.
Base case: When the height of x is 1, x is a leaf.
Then $bh(x) = 0$, and the subtree rooted at x contains $2^0 - 1 = 0$ internal nodes.

Implementation 3: Red-Black Trees

Claim. The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof. We prove the claim by induction on the height of x.
Base case: When the height of x is 1, x is a leaf.
Then $bh(x) = 0$, and the subtree rooted at x contains $2^0 - 1 = 0$ internal nodes.
Inductive step:
Suppose the statement holds for all height-k nodes.
Let x be a height-$(k+1)$ node.
We want to prove the subtree at x contains $\geq 2^{bh(x)} - 1$ internal nodes.
Since the tree is full, x must have two children, say y, z.
By the balance property, $bh(y) \geq bh(x) - 1$ and $bh(z) \geq bh(x) - 1$.
By the ind.hyp., subtrees of y, z contain $\geq 2^{bh(x)-1} - 1$ internal nodes each.
The subtree of x must contain $\geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes.
This proves the claim, and hence the lemma. □

Implementation 3: Red-Black Trees

Lemma

A red-black tree with n internal (non-NIL) nodes has height at most $2\log(n+1)$.

Implementation 3: Red-Black Trees

Lemma

A red-black tree with n internal (non-NIL) nodes has height at most $2\log(n+1)$.

Let h be the height of the tree.
By Red-node property, $bh(root) \geq h/2$.
Then we have

$$
\begin{aligned}
n &\geq 2^{bh(root)} - 1 && \text{by the claim} \\
n &\geq 2^{h/2} - 1 && \text{by the inequality above} \\
n+1 &\geq 2^{h/2} && \text{rearrange} \\
\log(n+1) &\geq h/2 && \text{both sides take log} \\
2\log(n+1) &\geq h && \text{rearrange}
\end{aligned}
$$

Implementation 3: Red-Black Trees

Consequence of the above lemma

The Search(x) operations can be performed in time $\Theta(\log n)$ on a red-black tree.

Implementation 3: Red-Black Trees

Consequence of the above lemma

The Search(x) operations can be performed in time $\Theta(\log n)$ on a red-black tree.

Question

What about *Insert* and *Remove*?

Implementation 3: Red-Black Trees

Consequence of the above lemma

The Search(x) operations can be performed in time $\Theta(\log n)$ on a red-black tree.

Question

What about *Insert* and *Remove*?

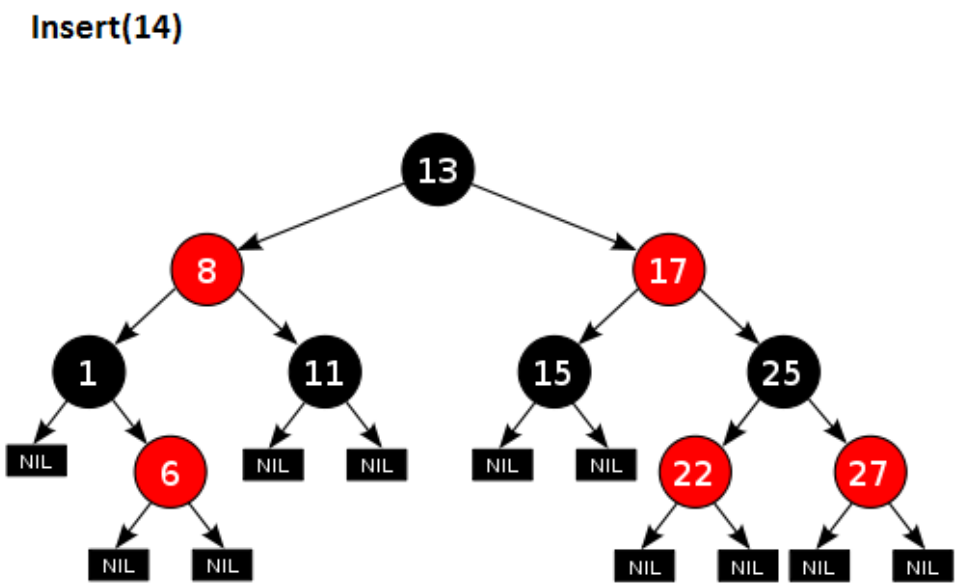
Challenge

The original binary search tree operation may destroy the red-black tree properties.
Therefore, we need to design new algorithms for handling insert and remove.

Implementation 3: Red-Black Trees

Insert(x)

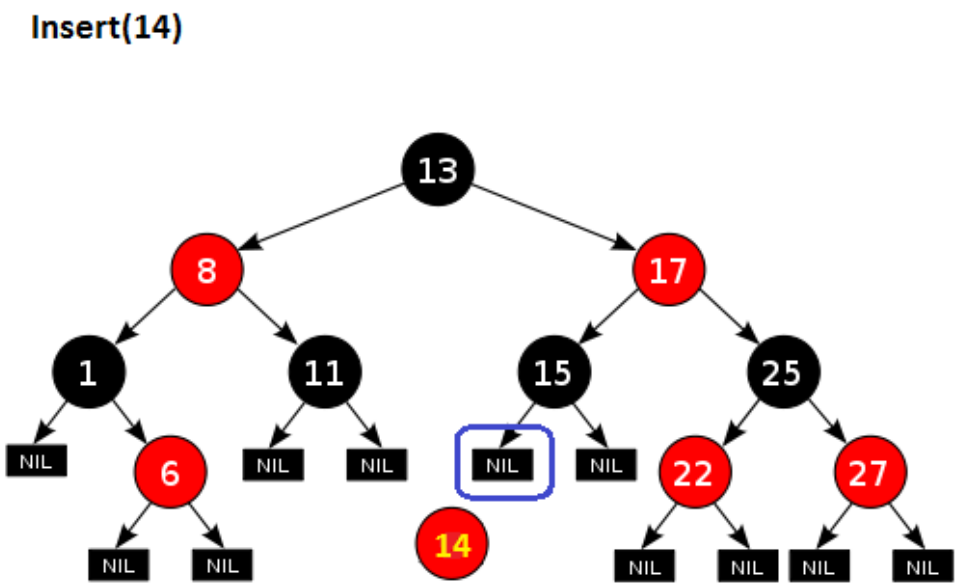
Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.



Implementation 3: Red-Black Trees

Insert(x)

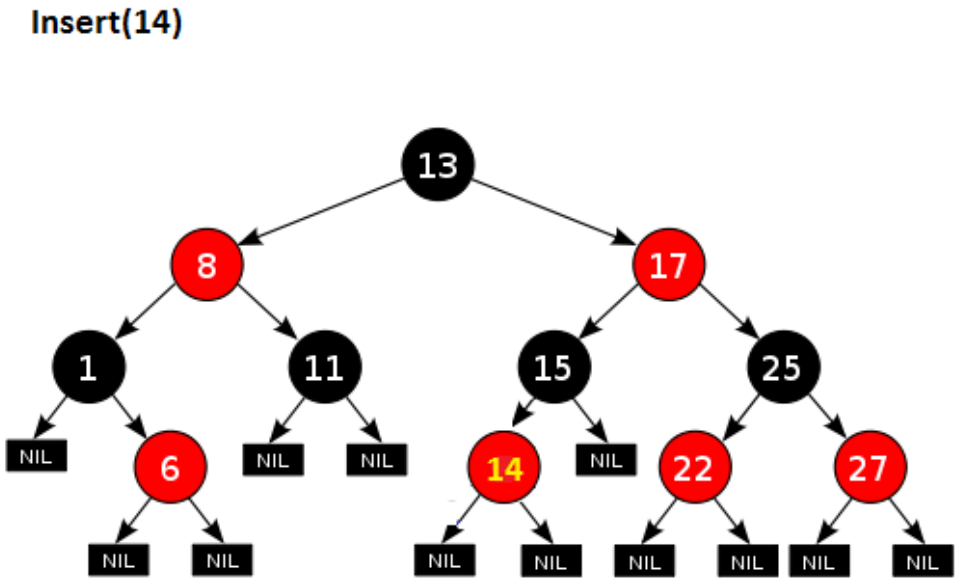
Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.



Implementation 3: Red-Black Trees

Insert(x)

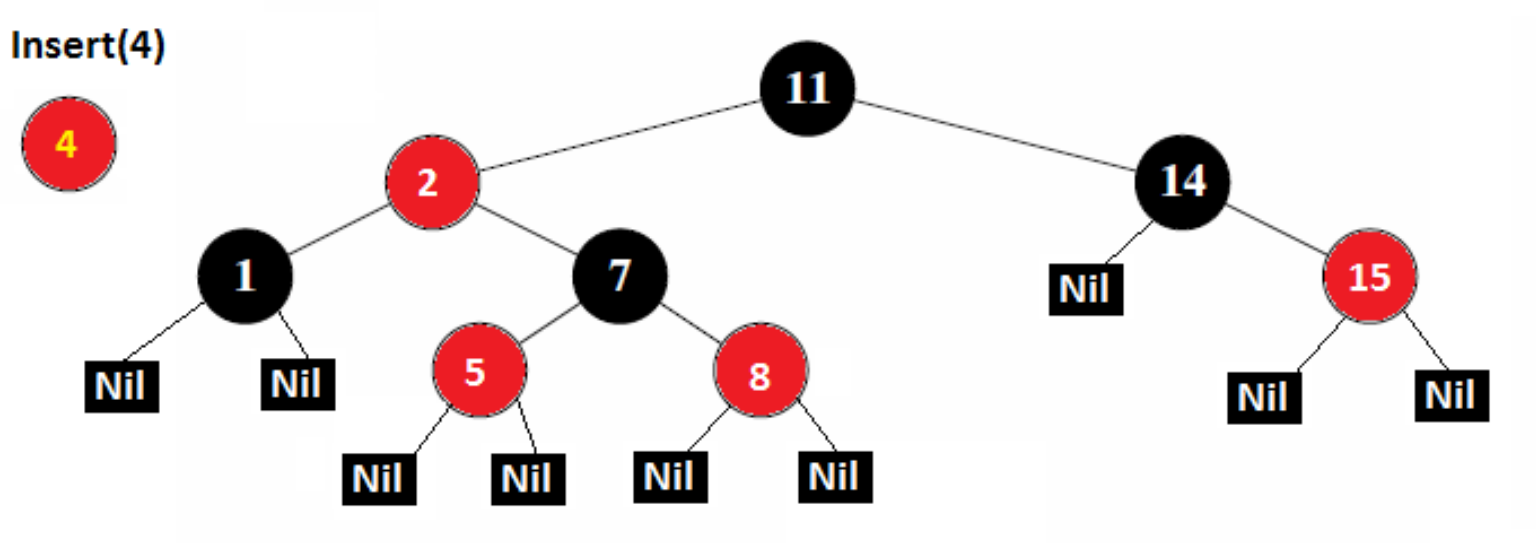
Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.



Implementation 3: Red-Black Trees

Insert(x)

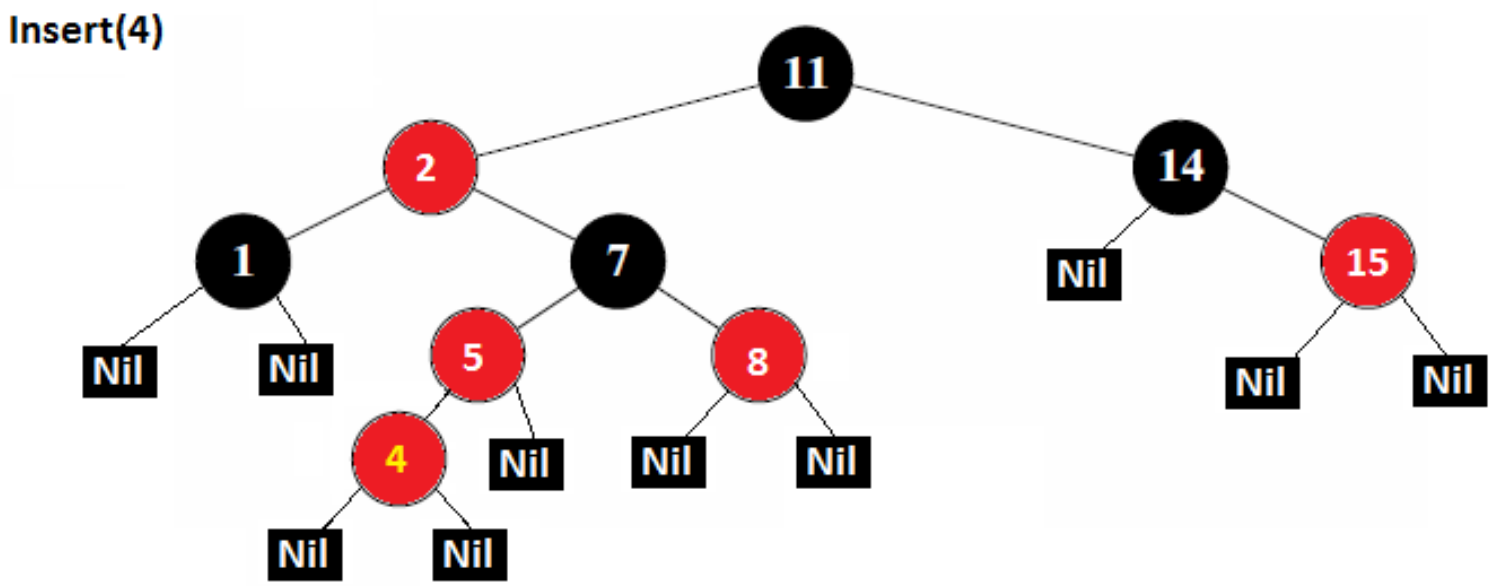
Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.



Implementation 3: Red-Black Trees

Insert(x)

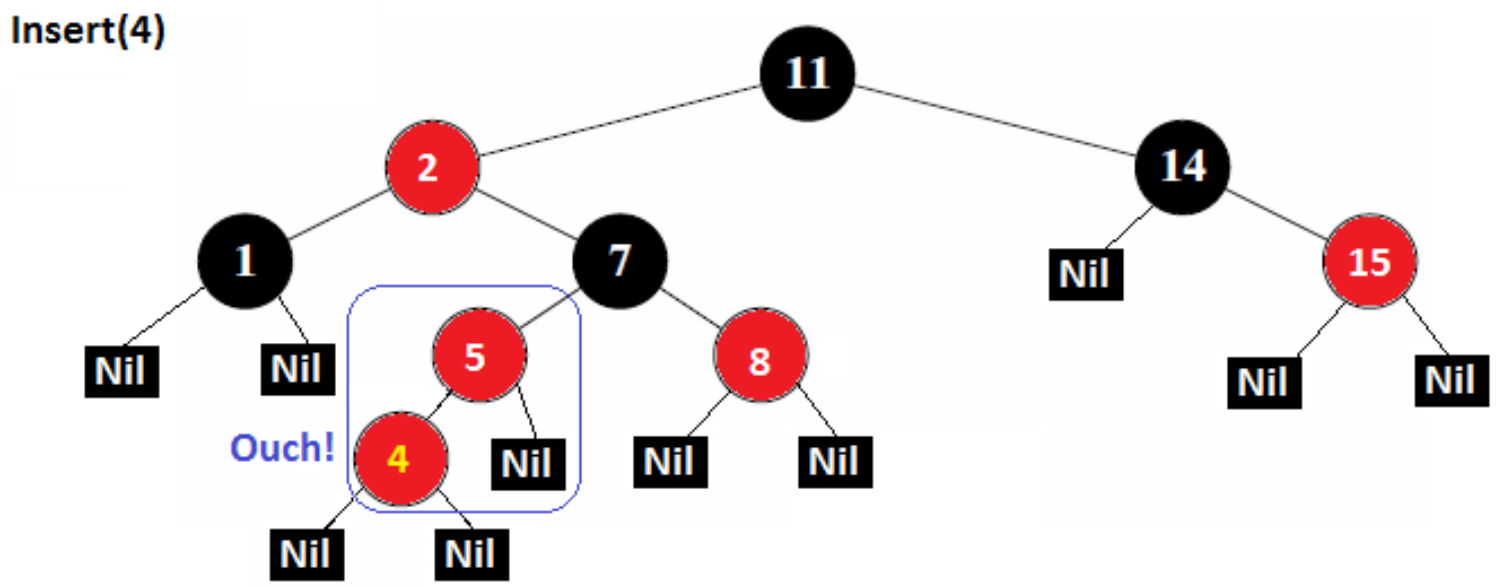
Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.



Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

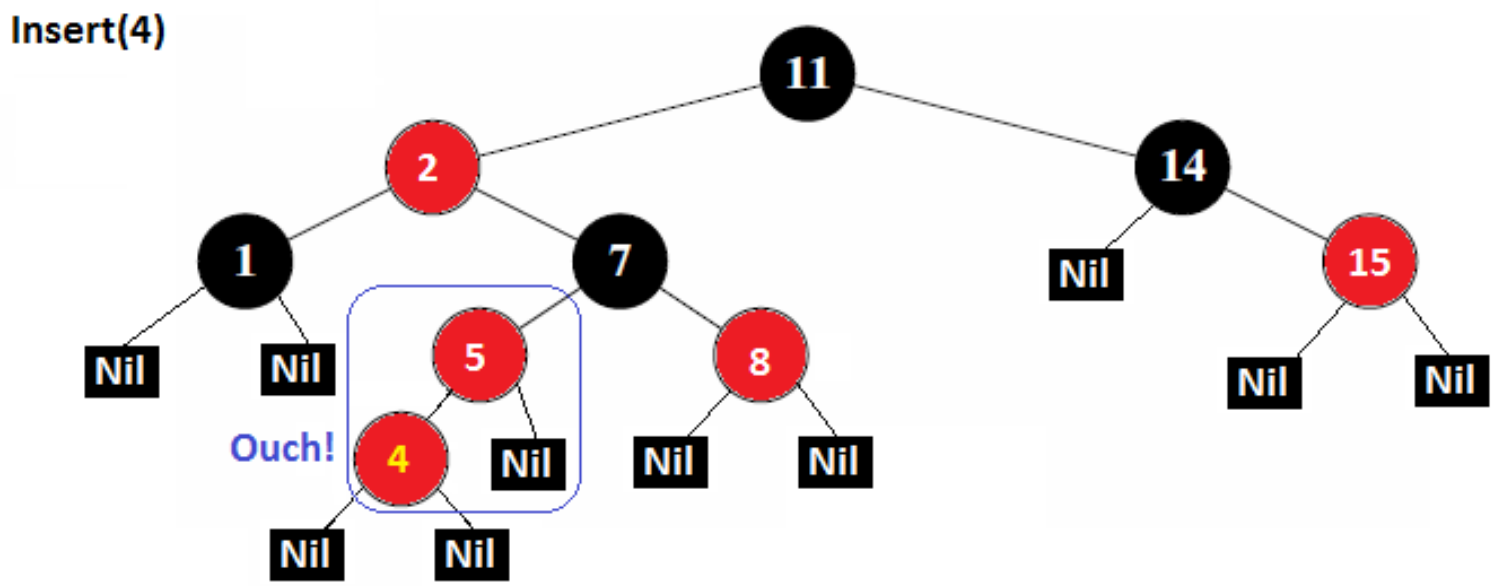


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

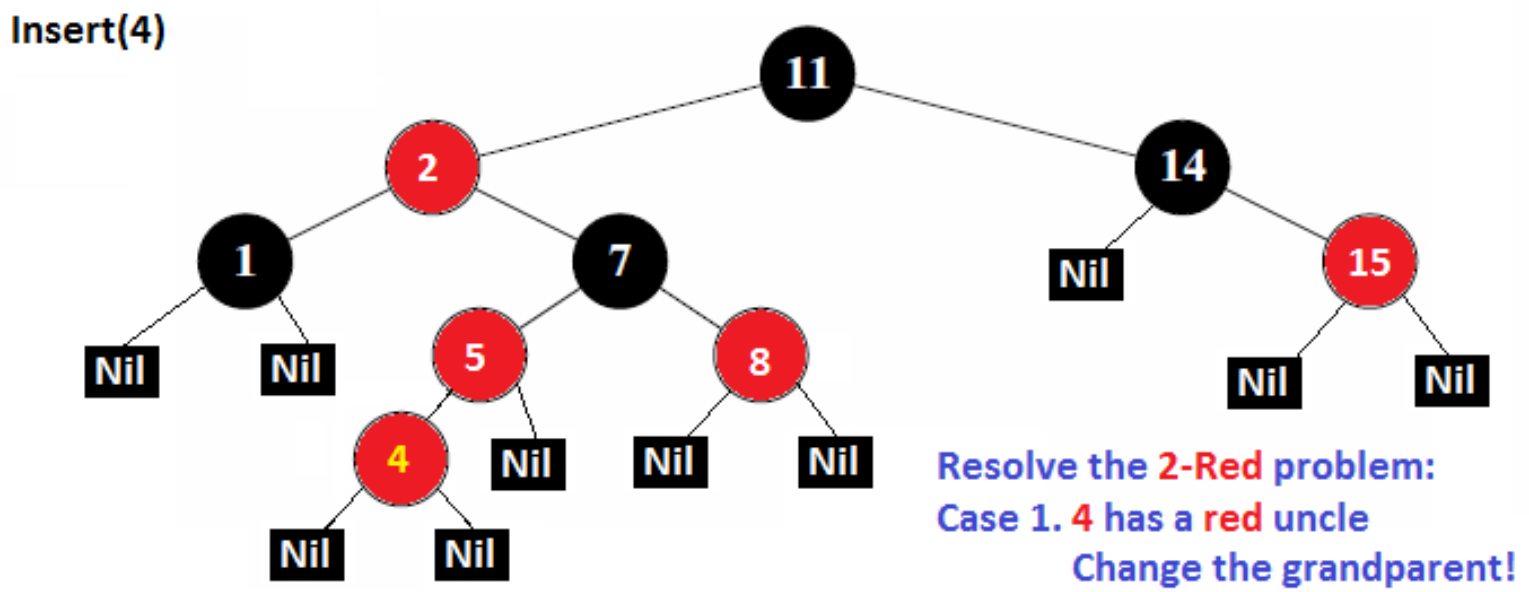


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

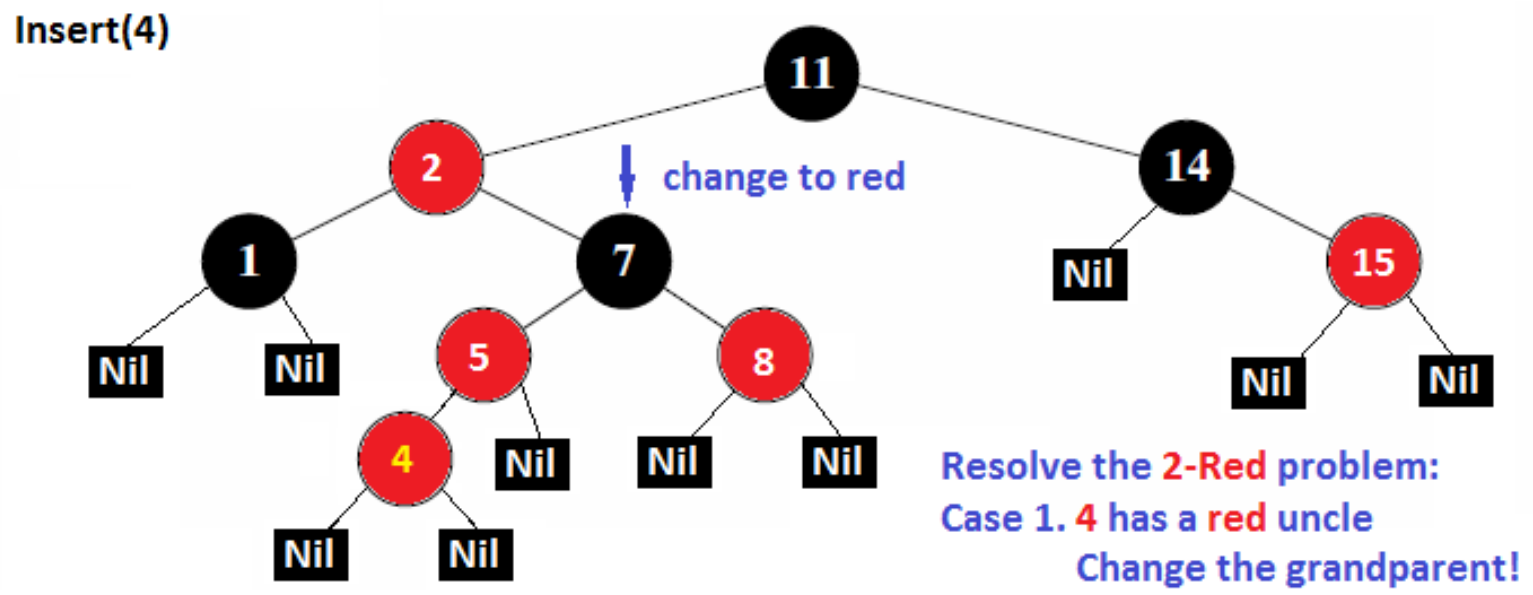


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

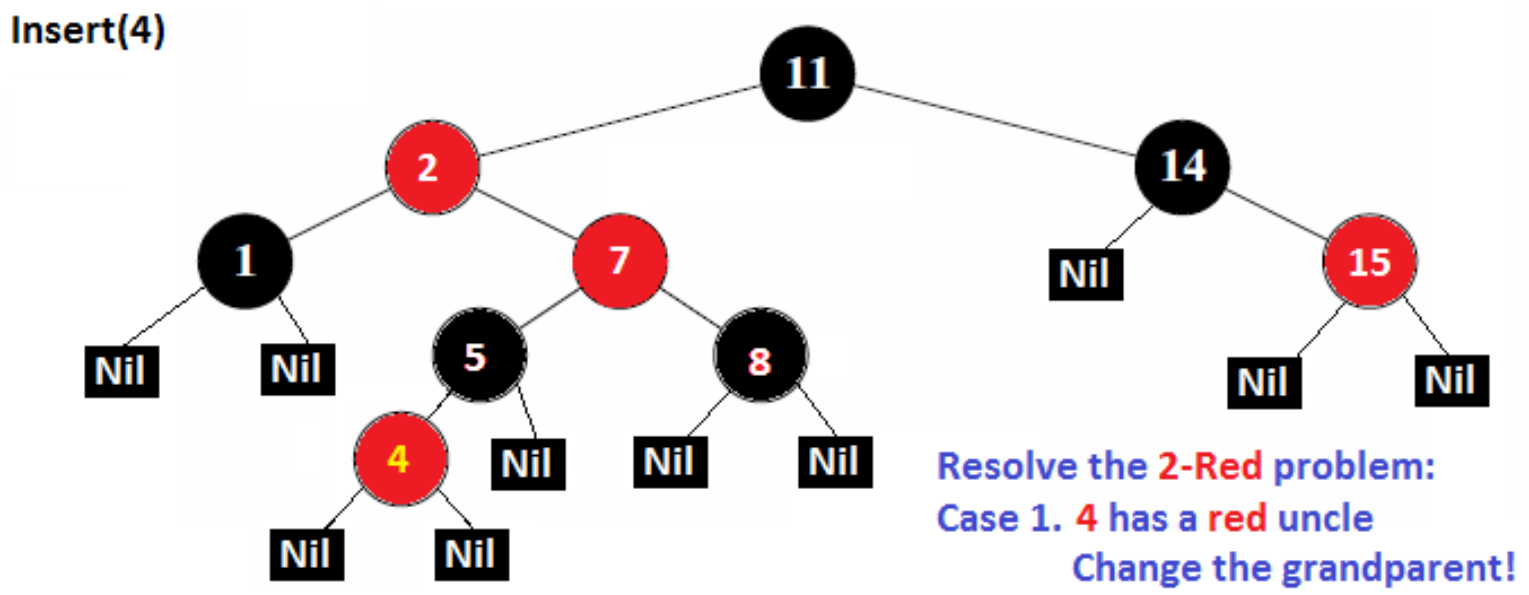


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

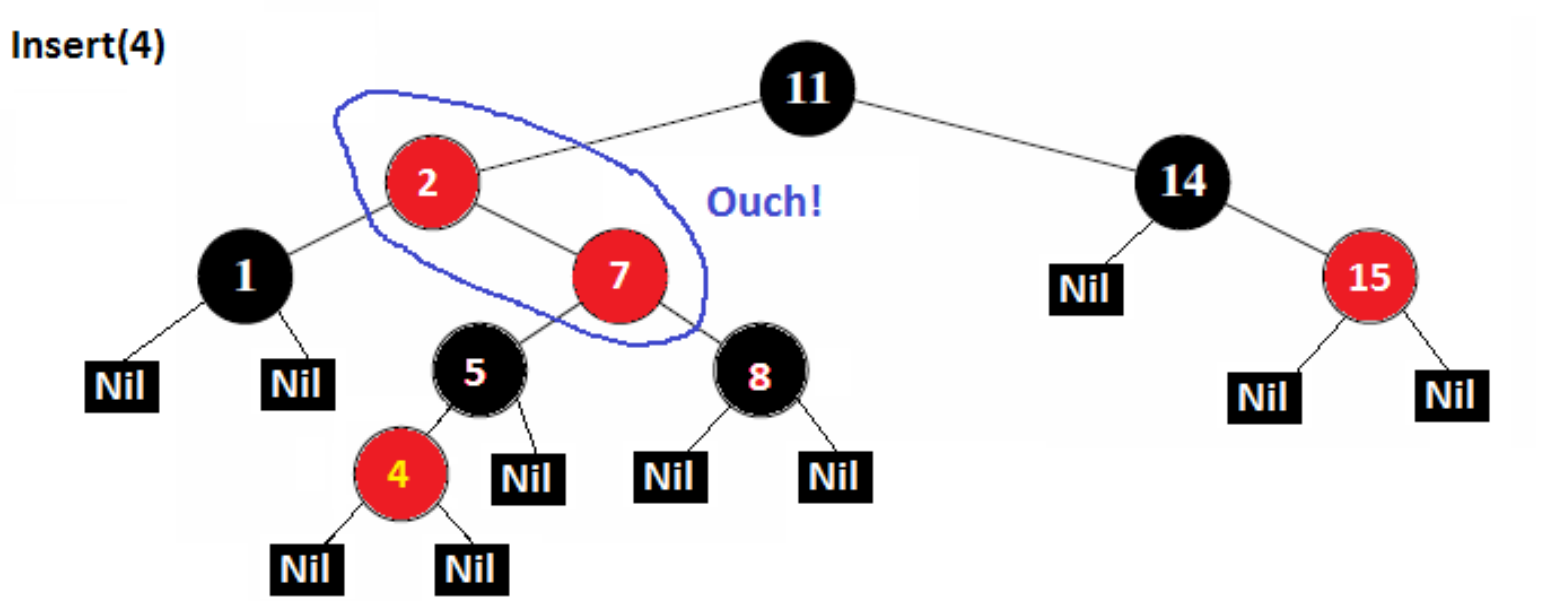


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

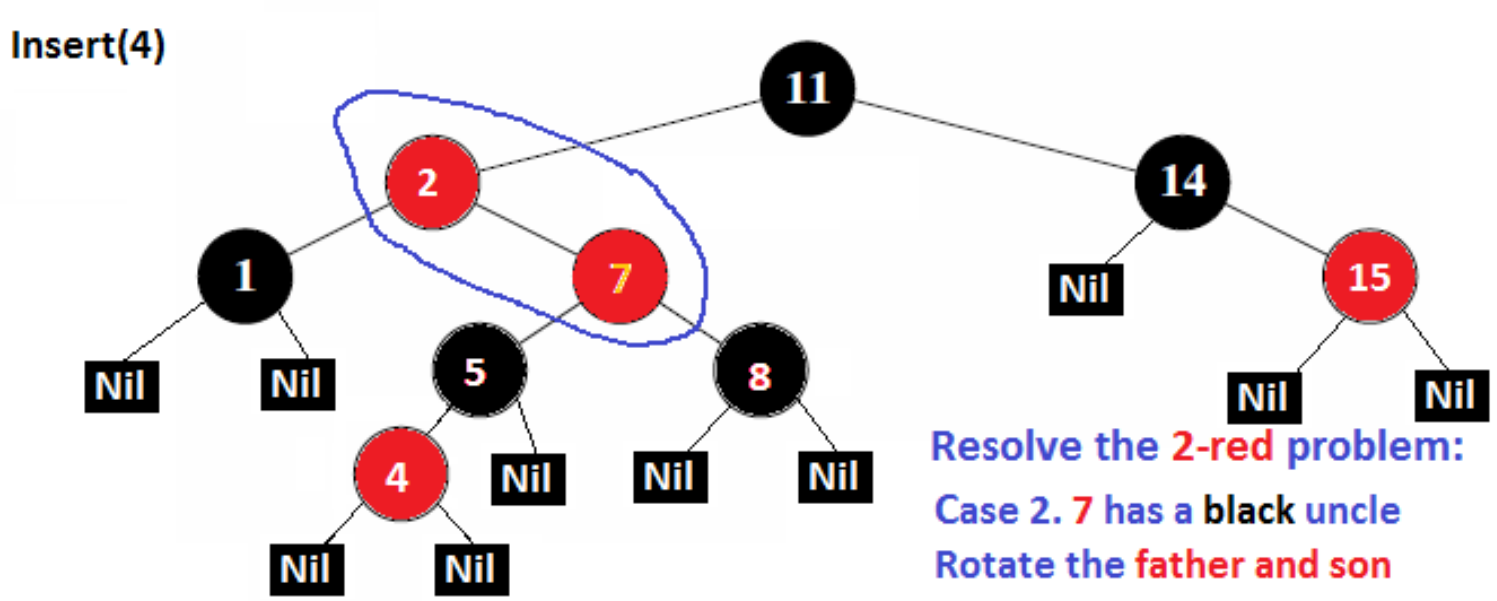


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

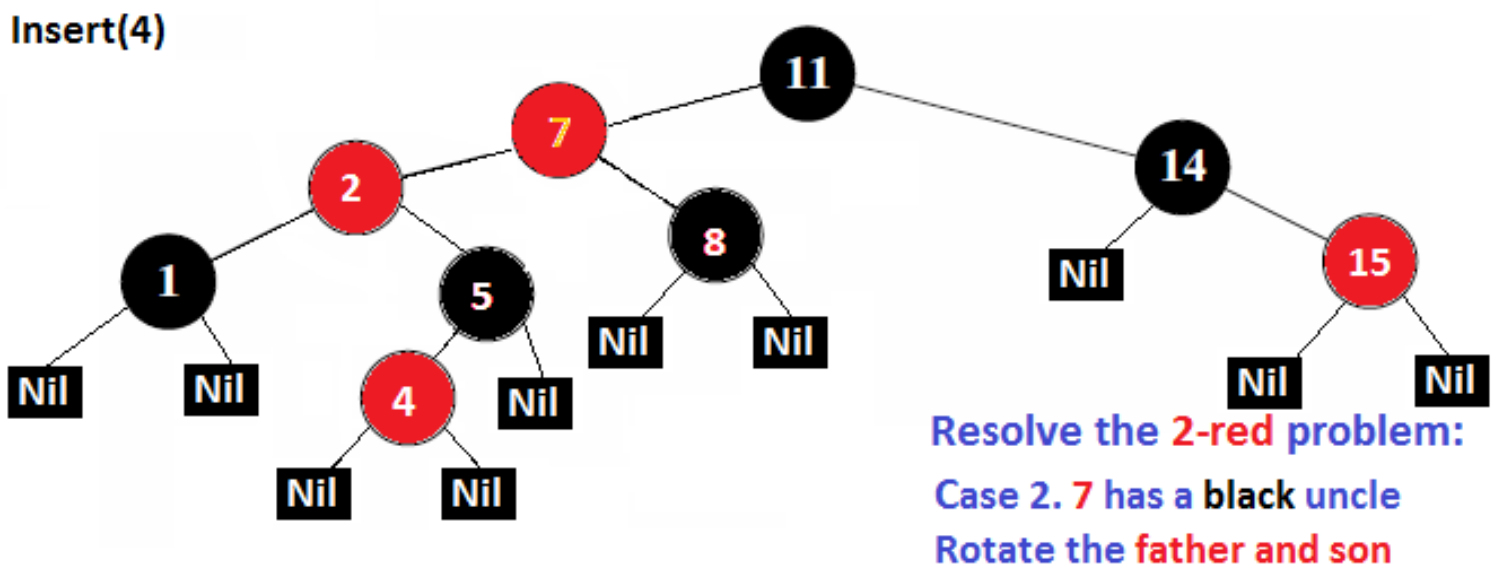


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

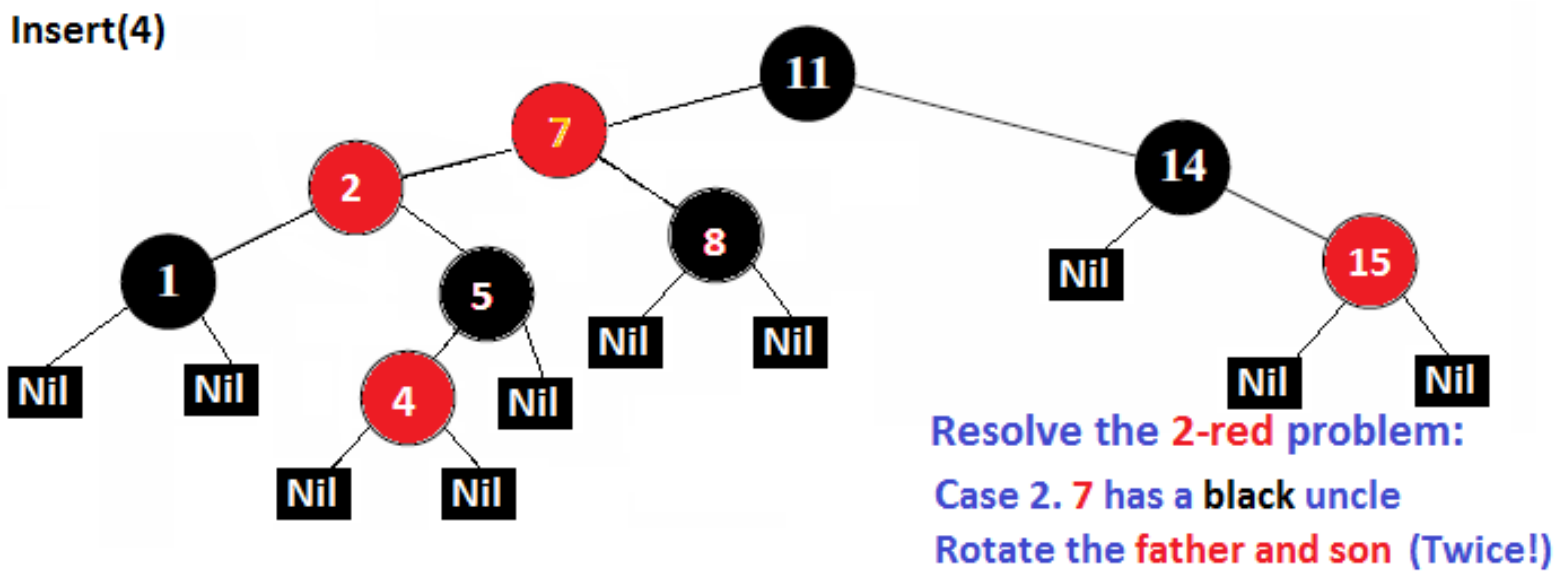


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

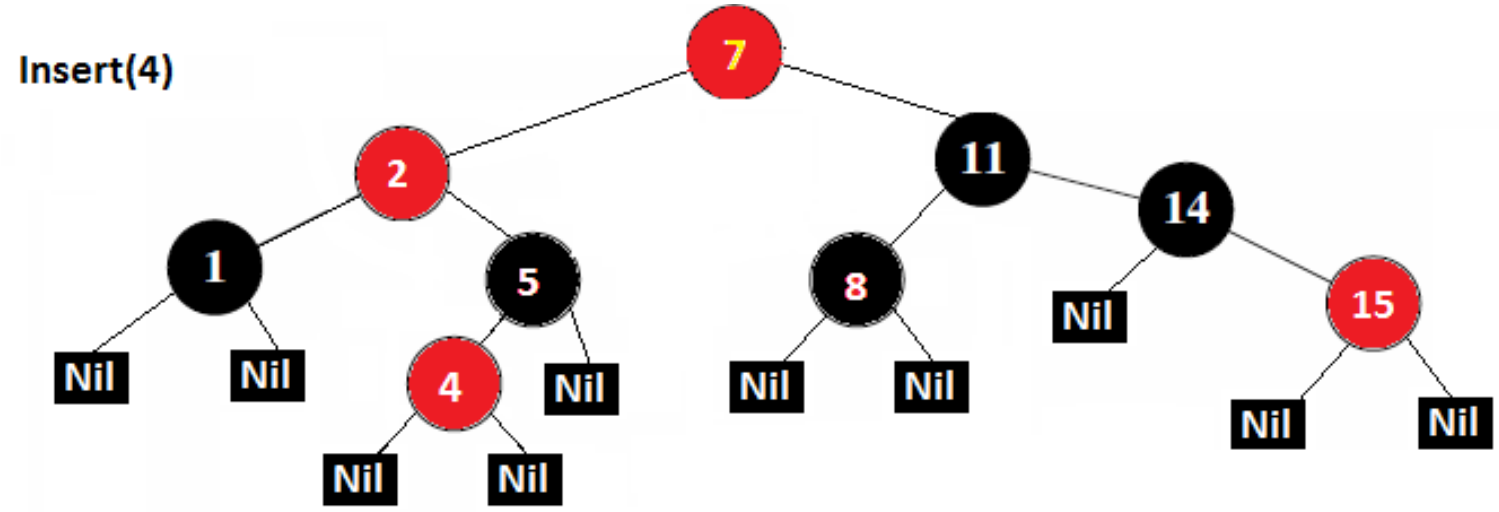


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!

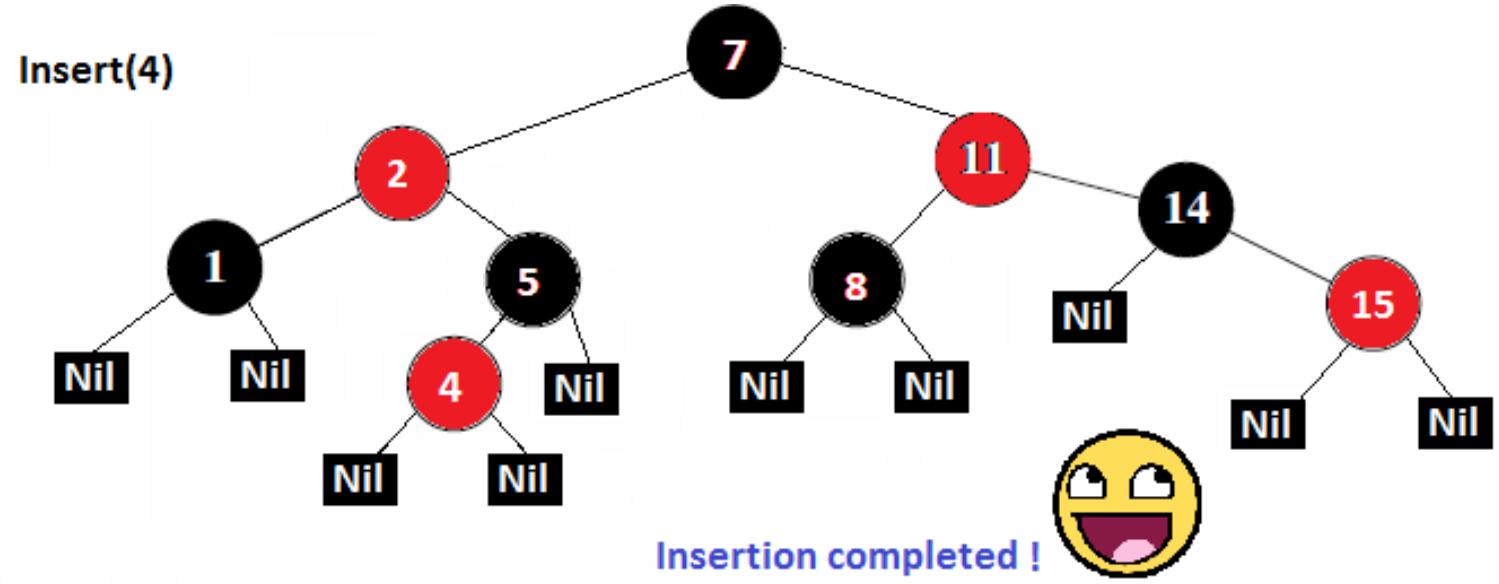


Implementation 3: Red-Black Trees

Insert(x)

Step 1: Do almost the same as in a binary search tree.
Always set the new node to red.
Don't forget to take care of the *NIL*s.

Step 2: Fix-up!



Implementation 3: Red-Black Trees

Question

Unless it is the only node, the inserted node is always first set to red. Why is that?

Implementation 3: Red-Black Trees

Question

Unless it is the only node, the inserted node is always first set to red. Why is that?

Answer: To maintain the balance property.

Implementation 3: Red-Black Trees

Question

Unless it is the only node, the inserted node is always first set to red. Why is that?

Answer: To maintain the balance property.

2-red

We call the case when two red nodes linked together a 2-red.

Implementation 3: Red-Black Trees

Question

Unless it is the only node, the inserted node is always first set to red. Why is that?

Answer: To maintain the balance property.

2-red

We call the case when two red nodes linked together a 2-red.

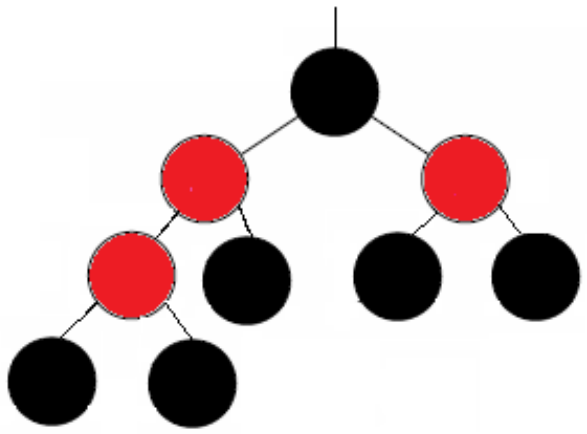
Bad news: Inserting a node may create a 2-red.
Good news: This is the only 2-red in the tree.

Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 1: The uncle of the lower node in the 2-red is red.
1. Re-color the grandparent red.
2. Re-color both the parent and the uncle **black**.



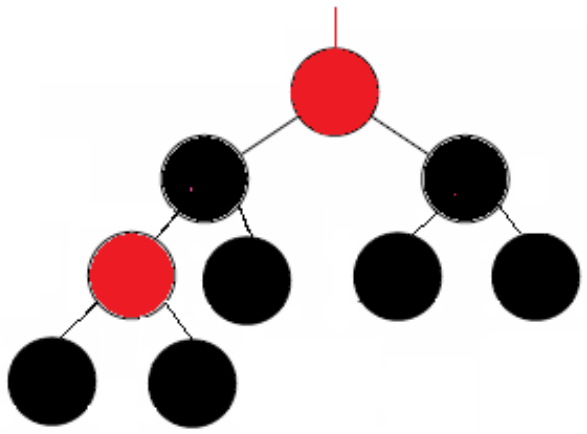
This would not break the balance property.
Also there would be at most one 2-red after re-coloring.

Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 1: The uncle of the lower node in the 2-red is red.
1. Re-color the grandparent red.
2. Re-color both the parent and the uncle **black**.



This would not break the balance property.
Also there would be at most one 2-red after re-coloring.

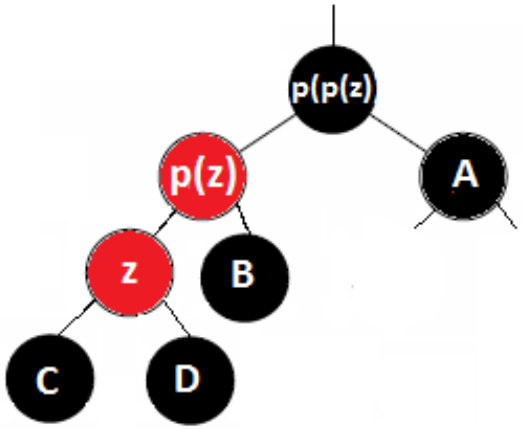
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 1: The 2-red and their parent form a straight line.

1. Rotate the higher node in the 2-red up
2. Switch the color.



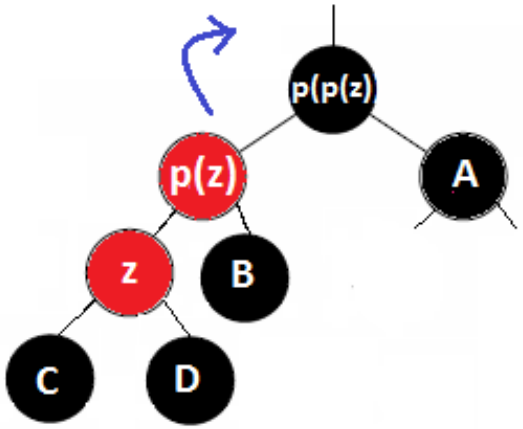
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 1: The 2-red and their parent form a straight line.

1. Rotate the higher node in the 2-red up
2. Switch the color.



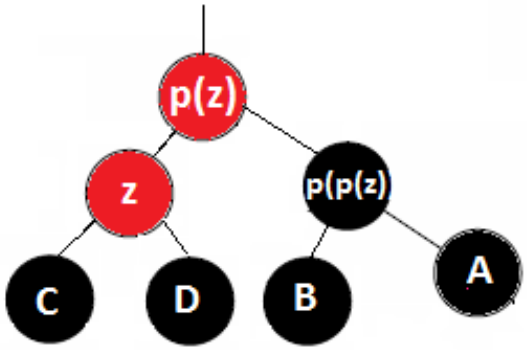
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 1: The 2-red and their parent form a straight line.

1. Rotate the higher node in the 2-red up
2. Switch the color.



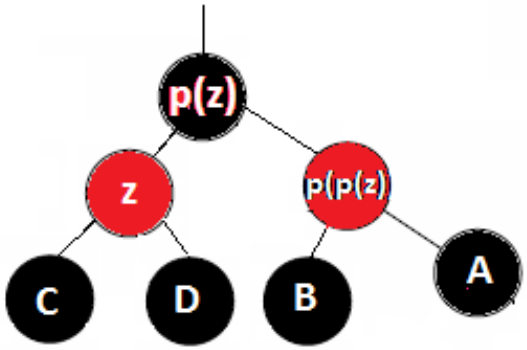
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 1: The 2-red and their parent form a straight line.

1. Rotate the higher node in the 2-red up
2. Switch the color.



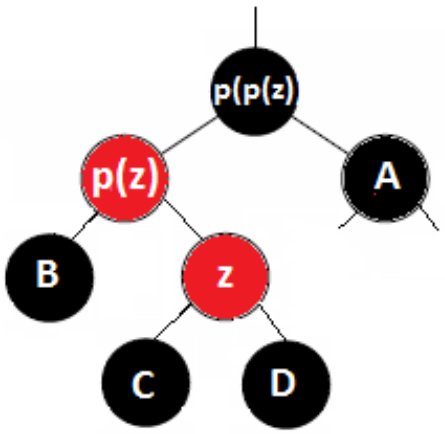
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 2: The 2-red and their parent don't form a straight line.

1. Rotate the lower node in the 2-red up,
   so that the 2-red and their parent become in a straight line
2. Apply Subcase 1



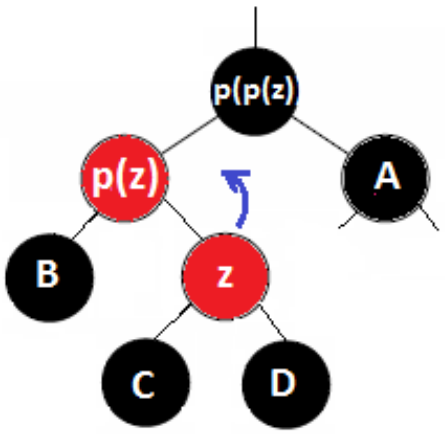
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 2: The 2-red and their parent don't form a straight line.

1. Rotate the lower node in the 2-red up,
   so that the 2-red and their parent become in a straight line
2. Apply Subcase 1



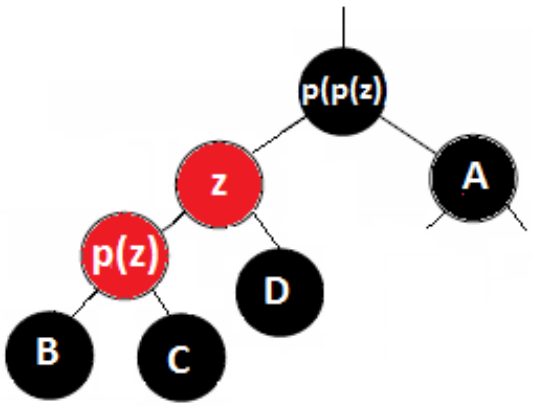
Implementation 3: Red-Black Trees

Suppose we have a 2-red in our tree; and this is the only 2-red.

There are two cases

Case 2: The uncle of the lower node in the 2-red is **black**.
Subcase 2: The 2-red and their parent don't form a straight line.

1. Rotate the lower node in the 2-red up,
   so that the 2-red and their parent become in a straight line
2. Apply Subcase 1



Implementation 3: Red-Black Trees

Insert(x)

1. Insert x as in a binary search tree, but additionally do
   - If x is the only node, change x to **black**.
   - Otherwise, set x red.
   - Appropriately change the NIL nodes.
   - If 2-red, run fixed-up
2. Fixed-up:
   - Let $z \leftarrow x$.
   - While 2-red and z's uncle is red do // $\Rightarrow$ Case 1
     - Set the grandparent $p(p(z))$ of z red.
     - Set both z's parent $p(z)$ and uncle **black**.
     - $z \leftarrow p(p(z))$
   - If still 2-red, then do the following: // $\Rightarrow$ Case 2
     - If $z, p(z), p(p(z))$ are on a straight line then // $\Rightarrow$ Subcase 1
       Set $p(p(z))$ to red.
       Rotate $p(z)$ up
       Change $p(z)$ to **black**
     - else // $\Rightarrow$ Subcase 2
       Rotate z up.
       Set $p(z)$ to red.
       Rotate z up again.
       Change z to **black**.

Implementation 3: Red-Black Trees

Analysis of Insert(x)

- Each iteration of while-loop takes constant time.
- Every time we repeat the while-loop, the node z goes up by one level.
- Hence the number of iterations of while-loop is $\Theta(h)$, where h is $\Theta(\log n)$.
- Therefore Insert(x) takes $\Theta(\log n)$.