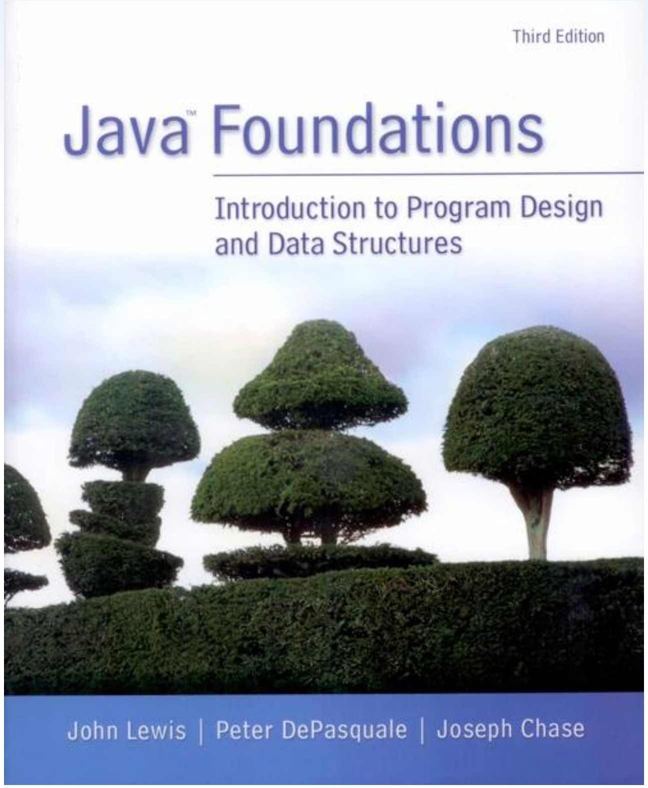


Chapter 21

Heaps and Priority Queues

Chapter Scope

- Heaps, conceptually
- Using heaps to solve problems
- Heap implementations
- Using heaps to implement priority queues

Heaps

- A *heap* is a complete binary tree in which each element is less than or equal to both of its children
- So a heap has both structural and ordering constraints
- As with binary search trees, there are many possible heap configurations for a given set of elements
- Our definition above is really a *minheap*
- A similar definition could be made for a *maxheap*

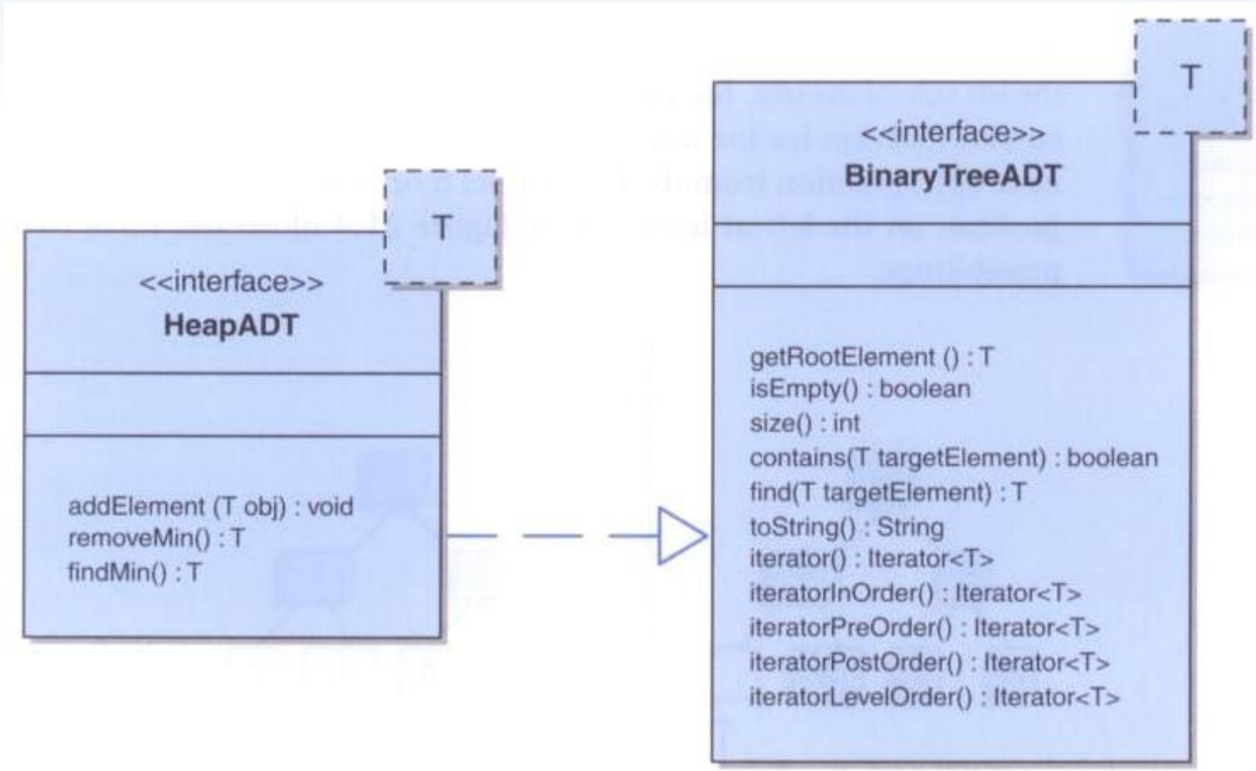
Heaps

- Operations on a heap:

Operation	Description
addElement	Adds the given element to the heap.
removeMin	Removes the minimum element in the heap.
findMin	Returns a reference to the minimum element in the heap.

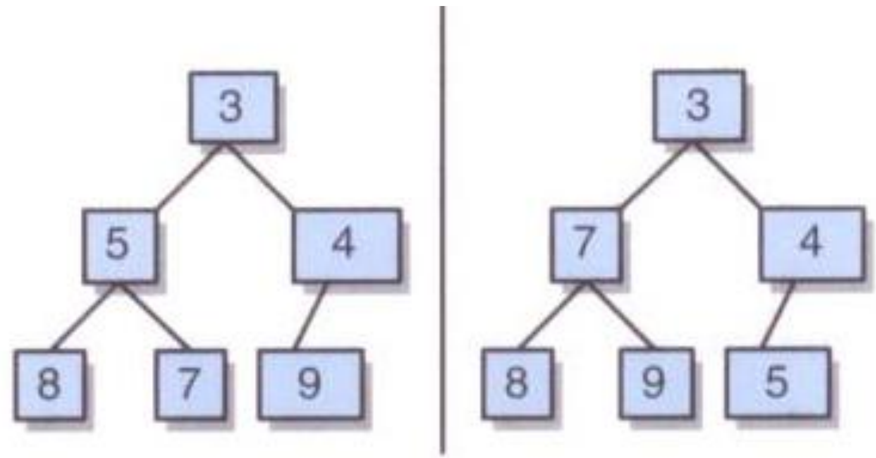
```java
package jsjf;

/**
 * HeapADT defines the interface to a Heap.
 *
 * @author Java Foundations
 * @version 4.0
 */
public interface HeapADT<T> extends BinaryTreeADT<T>
{
    /**
     * Adds the specified object to this heap.
     *
     * @param obj the element to be added to the heap
     */
    public void addElement(T obj);

    /**
     * Removes element with the lowest value from this heap.
     *
     * @return the element with the lowest value from the heap
     */
    public T removeMin();

    /**
     * Returns a reference to the element with the lowest value in
     * this heap.
     *
     * @return a reference to the element with the lowest value in the heap
     */
    public T findMin();
}
```

Heaps


T
<<interface>>
HeapADT
addElement (T obj) : void
removeMin() : T
findMin() : T
T
<<interface>>
BinaryTreeADT
getRootElement () : T
isEmpty() : boolean
size() : int
contains(T targetElement) : boolean
find(T targetElement) : T
toString() : String
iterator() : Iterator<T>
iteratorInOrder() : Iterator<T>
iteratorPreOrder() : Iterator<T>
iteratorPostOrder() : Iterator<T>
iteratorLevelOrder() : Iterator<T>

Heaps

- Two minheaps containing the same data:

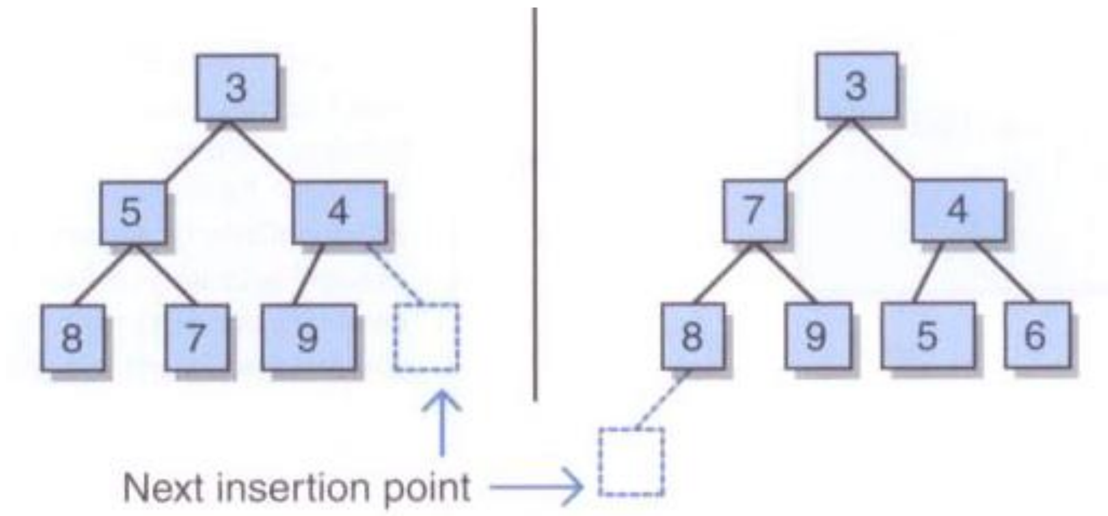


Adding a New Element

- To add an element to the heap, add the element as a leaf, keeping the tree complete
- Then, move the element up toward the root, exchanging positions with its parent, until the relationship among the elements is appropriate
- This will guarantee that the resulting tree will conform to the heap criteria

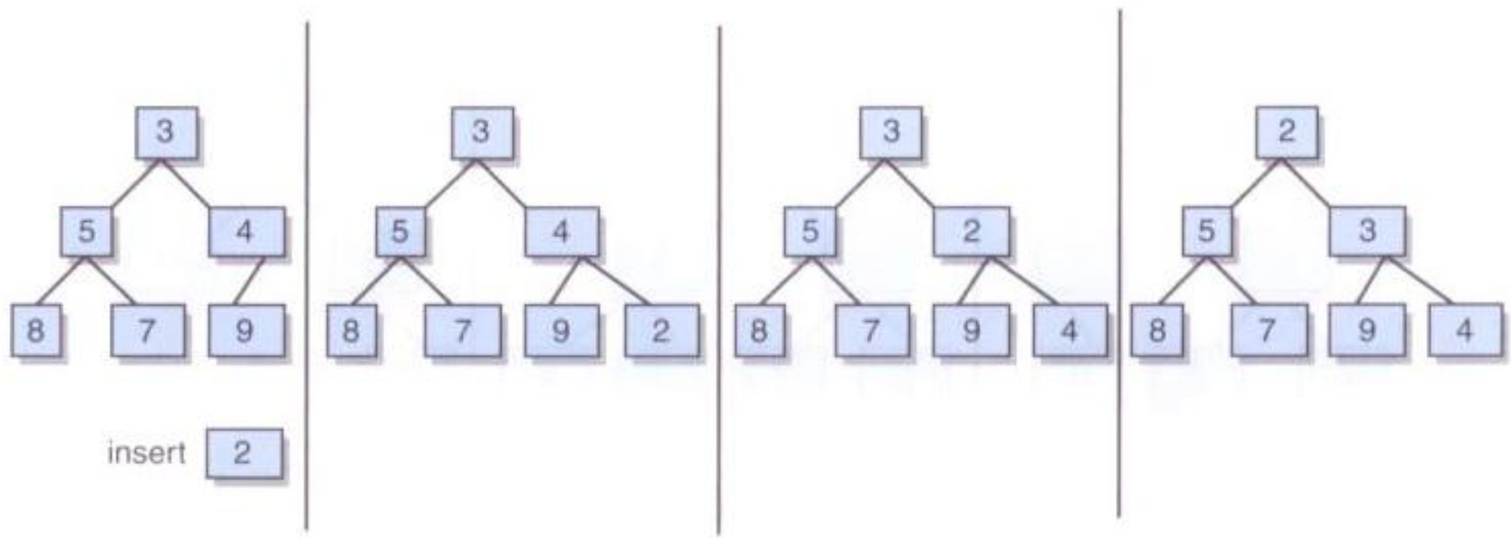
Adding a New Element

- The initial insertion point for a new element in a heap:



Adding a New Element

- Inserting an element and moving it up the tree as far as appropriate:

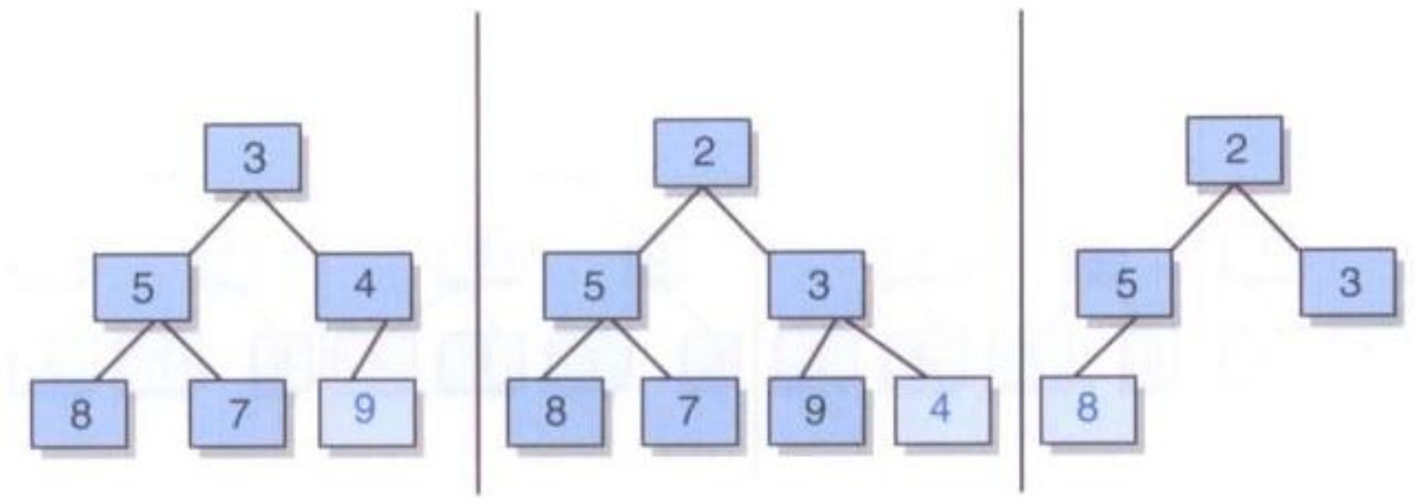


Removing the Min Element

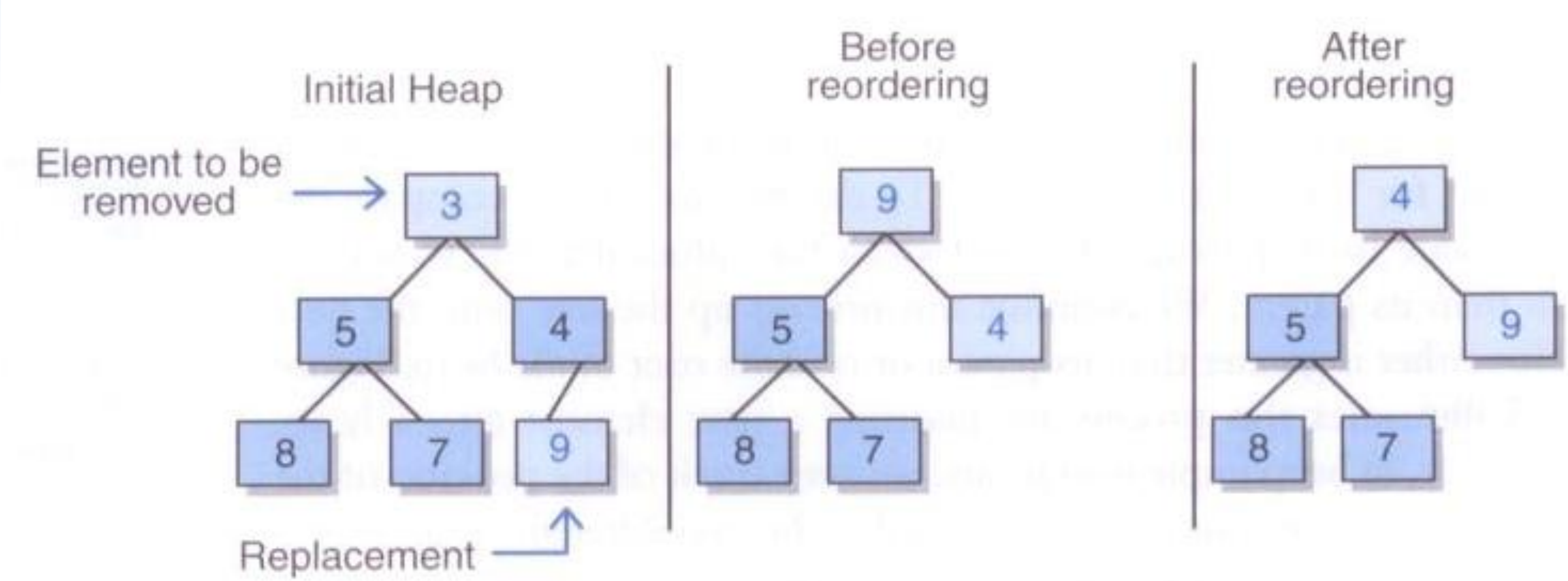
- Remove the root (min) and reconstruct the heap
- First, move the last leaf of the tree to be the new root of the tree
- Then, move it down the tree as needed until the relationships among the elements is appropriate
- In particular, compare the element to the smaller of its children and swap them if the child is smaller

Removing the Min Element

- The element to replace the root is the "last leaf" in the tree:



Removing the Min Element



Priority Qeueus

- Recall that a FIFO queue removes elements in the order in which they were added
- A *priority queue* removes elements in priority order, independent of the order in which they were added
- Priority queues are helpful in many scheduling situations
- A heap is a classic mechanism for implementing priority queues

```java
/**
 * PrioritizedObject represents a node in a priority queue containing a
 * comparable object, arrival order, and a priority value.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class PrioritizedObject<T> implements Comparable<PrioritizedObject>
{
    private static int nextOrder = 0;
    private int priority;
    private int arrivalOrder;
    private T element;

    /**
     * Creates a new PrioritizedObject with the specified data.
     *
     * @param element the element of the new priority queue node
     * @param priority the priority of the new queue node
     */
    public PrioritizedObject(T element, int priority)
    {
        this.element = element;
        this.priority = priority;
        arrivalOrder = nextOrder;
        nextOrder++;
    }
```

```java
/**
 * Returns the element in this node.
 *
 * @return the element contained within the node
 */
public T getElement()
{
    return element;
}

/**
 * Returns the priority value for this node.
 *
 * @return the integer priority for this node
 */
public int getPriority()
{
    return priority;
}

/**
 * Returns the arrival order for this node.
 *
 * @return the integer arrival order for this node
 */
public int getArrivalOrder()
{
    return arrivalOrder;
}
```

```java
    /**
     * Returns a string representation for this node.
     *
     */
    public String toString()
    {
        return (element + "   " + priority + "   " + arrivalOrder);
    }

    /**
     * Returns 1 if the this object has higher priority than
     * the given object and -1 otherwise.
     *
     * @param obj the object to compare to this node
     * @return the result of the comparison of the given object and
     *         this one
     */
    public int compareTo(PrioritizedObject obj)
    {
      int result;

      if (priority > obj.getPriority())
          result = 1;
      else if (priority < obj.getPriority())
          result = -1;
      else if (arrivalOrder > obj.getArrivalOrder())
          result = 1;
      else
          result = -1;

      return result;
    }
}
```

```java
import jsjf.*;

/**
 * PriorityQueue implements a priority queue using a heap.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class PriorityQueue<T> extends ArrayHeap<PrioritizedObject<T>>
{
    /**
     * Creates an empty priority queue.
     */
    public PriorityQueue()
    {
        super();
    }

    /**
     * Adds the given element to this PriorityQueue.
     *
     * @param object the element to be added to the priority queue
     * @param priority the integer priority of the element to be added
     */
    public void addElement(T object, int priority)
    {
        PrioritizedObject<T> obj = new PrioritizedObject<T>(object, priority);
        super.addElement(obj);
    }
```

```java
    /**
     * Removes the next highest priority element from this priority
     * queue and returns a reference to it.
     *
     * @return a reference to the next highest priority element in this queue
     */
    public T removeNext()
    {
        PrioritizedObject<T> obj = (PrioritizedObject<T>)super.removeMin();
        return obj.getElement();
    }
}
```

Implementing Heaps with Links

- The operations on a heap require moving up the heap as well as down
- So we'll add a parent pointer to the `HeapNode` class, which is itself based on the node for a binary tree
- In the heap itself, we'll keep track of a pointer so that we always know where the last leaf is

```java
package jsjf;

/**
 * HeapNode represents a binary tree node with a parent pointer for use
 * in heaps.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class HeapNode<T> extends BinaryTreeNode<T>
{
    protected HeapNode<T> parent;

    /**
     * Creates a new heap node with the specified data.
     *
     * @param obj the data to be contained within the new heap node
     */
    public HeapNode(T obj)
    {
        super(obj);
        parent = null;
    }
```

```java
    /**
     * Return the parent of this node.
     *
     * @return the parent of the node
     */
    public HeapNode<T> getParent()
    {
        return parent;
    }

    /**
     * Sets the element stored at this node.
     *
     * @param the element to be stored
     */
    public void setElement(T obj)
    {
        element = obj;
    }

    /**
     * Sets the parent of this node.
     *
     * @param node the parent of the node
     */
    public void setParent(HeapNode<T> node)
    {
        parent = node;
    }
}
```

```java
package jsjf;

import jsjf.exceptions.*;

/**
 * LinkedHeap implements a heap.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class LinkedHeap<T> extends LinkedBinaryTree<T> implements HeapADT<T>
{
    public HeapNode<T> lastNode;

    public LinkedHeap()
    {
        super();
    }
```

```java
/**
 * Adds the specified element to this heap in the appropriate
 * position according to its key value.
 *
 * @param obj the element to be added to the heap
 */
public void addElement(T obj)
{
    HeapNode<T> node = new HeapNode<T>(obj);

    if (root == null)
        root=node;
    else
    {
        HeapNode<T> nextParent = getNextParentAdd();
        if (nextParent.getLeft() == null)
            nextParent.setLeft(node);
        else
            nextParent.setRight(node);

        node.setParent(nextParent);
    }
    lastNode = node;
    modCount++;

    if (size() > 1)
        heapifyAdd();
}
```

```java
/**
 * Returns the node that will be the parent of the new node
 *
 * @return the node that will be the parent of the new node
 */
private HeapNode<T> getNextParentAdd()
{
    HeapNode<T> result = lastNode;

    while ((result != root) && (result.getParent().getLeft() != result))
        result = result.getParent();

    if (result != root)
        if (result.getParent().getRight() == null)
            result = result.getParent();
        else
        {
            result = (HeapNode<T>)result.getParent().getRight();
            while (result.getLeft() != null)
                result = (HeapNode<T>)result.getLeft();
        }
    else
        while (result.getLeft() != null)
            result = (HeapNode<T>)result.getLeft();

    return result;
}
```

```java
/**
 * Reorders this heap after adding a node.
 */
private void heapifyAdd()
{
    T temp;
    HeapNode<T> next = lastNode;

    temp = next.getElement();

    while ((next != root) &&
      (((Comparable)temp).compareTo(next.getParent().getElement()) < 0))
    {
        next.setElement(next.getParent().getElement());
        next = next.parent;
    }
    next.setElement(temp);
}
```

```java
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it. Throws an EmptyCollectionException
 * if the heap is empty.
 *
 * @return the element with the lowest value in this heap
 * @throws EmptyCollectionException if the heap is empty
 */
public T removeMin() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("LinkedHeap");

    T minElement =  root.getElement();

    if (size() == 1)
    {
        root = null;
        lastNode = null;
    }
```

```java
    else
    {
        HeapNode<T> nextLast = getNewLastNode();
        if (lastNode.getParent().getLeft() == lastNode)
            lastNode.getParent().setLeft(null);
        else
            lastNode.getParent().setRight(null);

        ((HeapNode<T>)root).setElement(lastNode.getElement());
        lastNode = nextLast;
        heapifyRemove();
    }

    modCount++;

    return minElement;
}
```

```java
/**
 * Returns the node that will be the new last node after a remove.
 *
 * @return the node that willbe the new last node after a remove
 */
private HeapNode<T> getNewLastNode()
{
    HeapNode<T> result = lastNode;

    while ((result != root) && (result.getParent().getLeft() == result))
        result = result.getParent();

    if (result != root)
        result = (HeapNode<T>)result.getParent().getLeft();

    while (result.getRight() != null)
        result = (HeapNode<T>)result.getRight();

    return result;
}
```

```java
/**
 * Reorders this heap after removing the root element.
 */
private void heapifyRemove()
{
    T temp;
    HeapNode<T> node = (HeapNode<T>)root;
    HeapNode<T> left = (HeapNode<T>)node.getLeft();
    HeapNode<T> right = (HeapNode<T>)node.getRight();
    HeapNode<T> next;

    if ((left == null) && (right == null))
        next = null;
    else if (right == null)
        next = left;
    else if (((Comparable)left.getElement()).compareTo(right.getElement()) < 0)
        next = left;
    else
        next = right;

    temp = node.getElement();
```

```java
        while ((next != null) &&
          (((Comparable)next.getElement()).compareTo(temp) < 0))
        {
            node.setElement(next.getElement());
            node = next;
            left = (HeapNode<T>)node.getLeft();
            right = (HeapNode<T>)node.getRight();

            if ((left == null) && (right == null))
                next = null;
            else if (right == null)
                next = left;
            else if (((Comparable)left.getElement()).compareTo(right.getElement()) < 0)
                next = left;
            else
                next = right;
        }
        node.setElement(temp);
    }
```

Implementing Heaps with Arrays

- Since a heap is a complete tree, an array-based implementation is reasonable
- As previously discussed, a parent element at index n will have children stored at index 2n+1 and 2n+2 of the array
- Conversely, for any node other than the root, the parent of the node is found at index (n-1)/2

```java
package jsjf;

import java.util.*;
import jsjf.exceptions.*;

/**
 * ArrayBinaryTree implements the BinaryTreeADT interface using an array
 *
 * @author Java Foundations
 * @version 4.0
 */
public class ArrayBinaryTree<T> implements BinaryTreeADT<T>, Iterable<T>
{
    private static final int DEFAULT_CAPACITY = 50;

    protected int count;
    protected T[] tree;
    protected int modCount;

    /**
     * Creates an empty binary tree.
     */
    public ArrayBinaryTree()
    {
        count = 0;
        tree = (T[]) new Object[DEFAULT_CAPACITY];
    }
```

```java
/**
 * Creates a binary tree with the specified element as its root.
 *
 * @param element the element which will become the root of the new tree
 */
public ArrayBinaryTree(T element)
{
    count = 1;
    tree = (T[]) new Object[DEFAULT_CAPACITY];
    tree[0] = element;
}
```

```java
package jsjf;

import jsjf.exceptions.*;

/**
 * ArrayHeap provides an array implementation of a minheap.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class ArrayHeap<T> extends ArrayBinaryTree<T> implements HeapADT<T>
{
    /**
     * Creates an empty heap.
     */
    public ArrayHeap()
    {
        super();
    }
```

```java
/**
 * Adds the specified element to this heap in the appropriate
 * position according to its key value.
 *
 * @param obj the element to be added to the heap
 */
public void addElement(T obj)
{
    if (count == tree.length)
        expandCapacity();

    tree[count] = obj;
    count++;
    modCount++;

    if (count > 1)
        heapifyAdd();
}
```

```java
/**
 * Reorders this heap to maintain the ordering property after
 * adding a node.
 */
private void heapifyAdd()
{
    T temp;
    int next = count - 1;

    temp = tree[next];

    while ((next != 0) &&
      (((Comparable)temp).compareTo(tree[(next-1)/2]) < 0))
    {

        tree[next] = tree[(next-1)/2];
        next = (next-1)/2;
    }

    tree[next] = temp;
}
```

```java
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it. Throws an EmptyCollectionException if
 * the heap is empty.
 *
 * @return a reference to the element with the lowest value in this heap
 * @throws EmptyCollectionException if the heap is empty
 */
public T removeMin() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("ArrayHeap");

    T minElement = tree[0];
    tree[0] = tree[count-1];
    heapifyRemove();
    count--;
    modCount--;

    return minElement;
}
```

```java
/**
 * Reorders this heap to maintain the ordering property
 * after the minimum element has been removed.
 */
private void heapifyRemove()
{
    T temp;
    int node = 0;
    int left = 1;
    int right = 2;
    int next;

    if ((tree[left] == null) && (tree[right] == null))
        next = count;
    else if (tree[right] == null)
        next = left;
    else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
        next = left;
    else
        next = right;
    temp = tree[node];
```

```
    while ((next < count) &&
      (((Comparable)tree[next]).compareTo(temp) < 0))
    {
        tree[node] = tree[next];
        node = next;
        left = 2 * node + 1;
        right = 2 * (node + 1);
        if ((tree[left] == null) && (tree[right] == null))
            next = count;
        else if (tree[right] == null)
            next = left;
        else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
            next = left;
        else
            next = right;
    }
    tree[node] = temp;
}
```

Heap Sort

- Given the ordering property of a heap, it is natural to think of using a heap to sort a list of numbers
- A *heap sort* sorts a set of elements by adding each one to a heap, then removing them one at a time
- The smallest element comes off the heap first, so the sequence will be in ascending order

```java
package jsjf;

/**
 * HeapSort sorts a given array of Comparable objects using a heap.
 *
 * @author Java Foundations
 * @version 4.0
 */
public class HeapSort<T>
{
    /**
     * Sorts the specified array using a Heap
     *
     * @param data the data to be added to the heapsort
     */
    public void HeapSort(T[] data)
    {
        ArrayHeap<T> temp = new ArrayHeap<T>();

        // copy the array into a heap
        for (int i = 0; i < data.length; i++)
            temp.addElement(data[i]);

        // place the sorted elements back into the array
        int count = 0;
        while (!(temp.isEmpty()))
        {
            data[count] = temp.removeMin();
            count++;
        }
    }
}
```