UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 10
Priority Queue Data Structure

Yoan Pinzón

Table of Content Session 10

- **Priority Queues Data Structure**
  - ▷ Max Priority Queues
    - ⋄ Heaps
    - ⋄ An Application (Heap Sort)

Priority Queue Data Structure

- **Priority Queues:** FIFO structure where elements are deleted in increasing (decreasing) order of priority rather than in the order in which they arrived in the queue.

- **Max Priority Queues**: The Find/Delete operations apply to the element of maximum priority.

- **Min Priority Queues**: The Find/Delete operations apply to the element of minimum priority.

The ADT MaxPriorityQueue

AbstractDataType MaxPriorityQueue
{
instances: finite collection of elements, each has a priority
operations:

`isEmpty()`: return `true` iff the queue is empty

`size()`: return number of elements in the queue

`getMax()`: return element with maximum priority

`put(x)`: inserts the element `x` into the queue

`removeMax()`: remove the element with maximum
priority and return this element;

}

Interface Definition of MaxPriorityQueue

```java
package unal.datastructures;

interface MaxPriorityQueue<T extends Comparable<? super T>>
{
   boolean isEmpty( );
   int size( );
   T getMax( );
   void put( T theObject );
   T removeMax( );
}
```

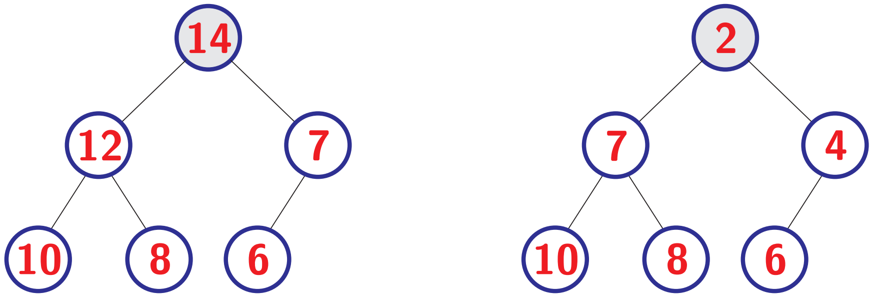
Representation of a MaxPriorityQueue

- As a Linear list
- As a Heap

Heaps

- **Max tree (min tree):** is a tree in which the value in each node is greater (less) than or equal to those in its children.

- **Max heap (min heap):** is a max (min) tree that is also a complete binary tree.



(a) MaxHeap *(b) MinHeap*

Representation of a Heap

- Since a heap is a complete binary tree, a heap can be efficiently represented as an *array*

- We can make use of property P5 to move from one node in the heap to its parent or to one of its children

- A heap with n elements has height $\lceil \log_2(n+1) \rceil$

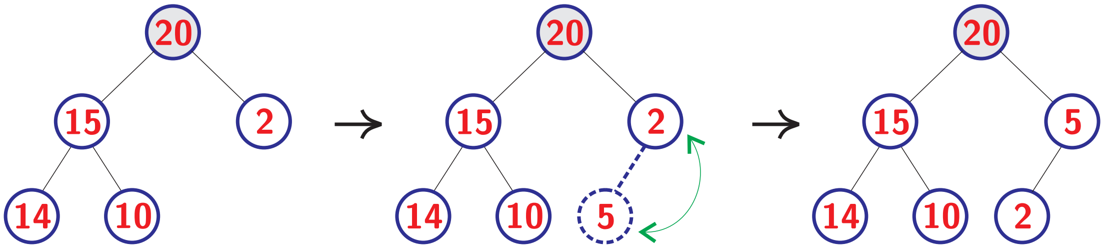
Insertion into a MaxHeap

1) Insert a new element as a leaf of the heap

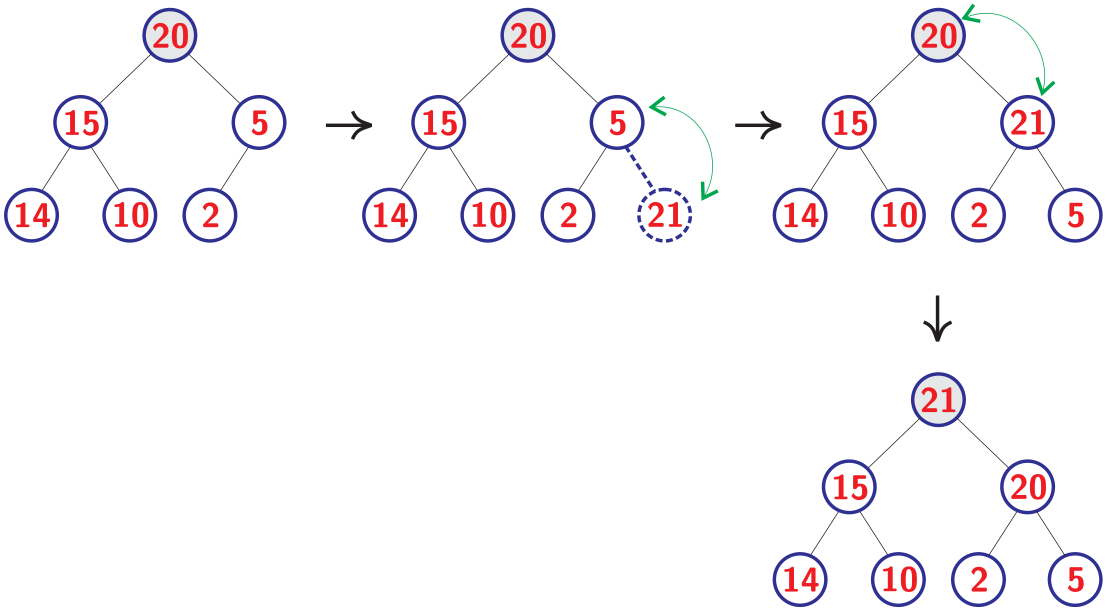
2) Walk up to the root to restore the heap properties

Time Complexity: $O(\log n)$

Example: Insert 5 into heap



Example: Insert 21 into heap



Deletion from a MaxHeap

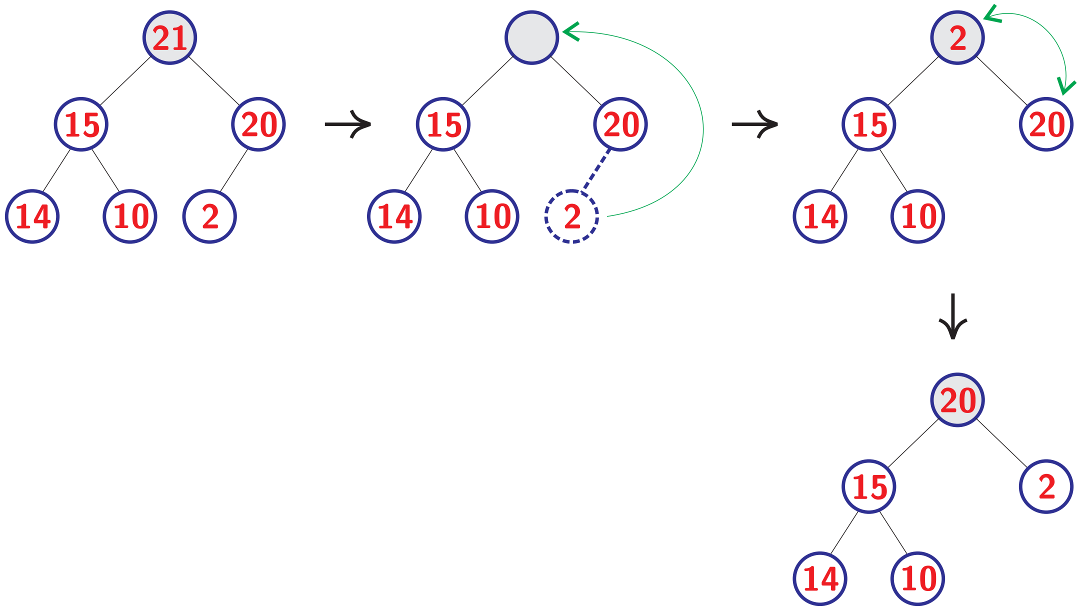
1) Delete the root element

2) Delete the rightmost leaf at the highest level and put it in the root

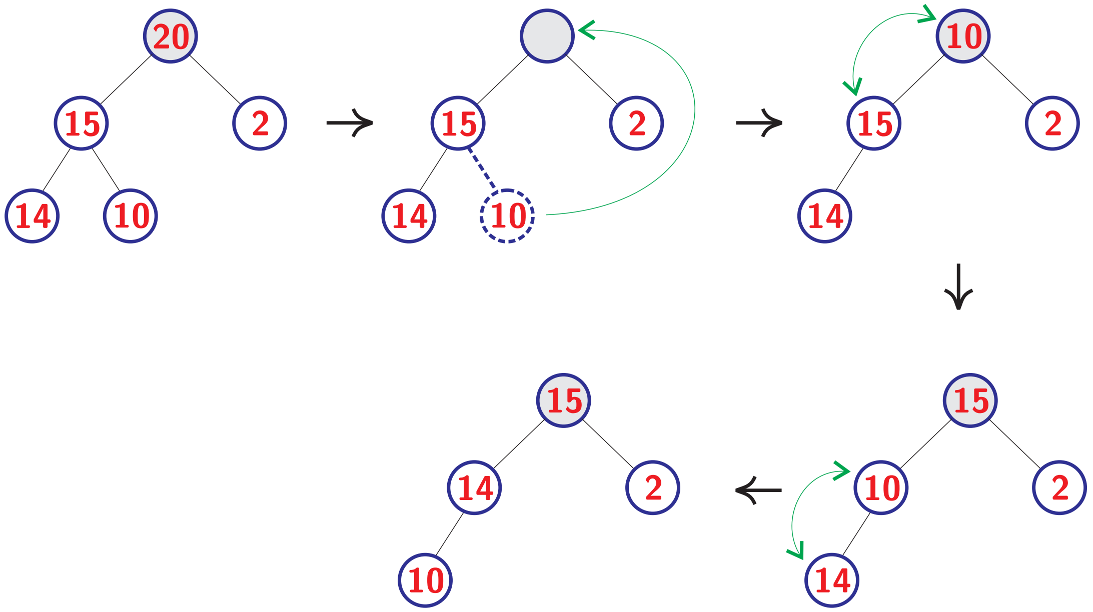
3) Restore the heap properties by walking down from root to a leaf by following the path determined by the child having the largest value

Time Complexity: $O(\log n)$

Example: Remove Max element from heap



Example: Delete 20.

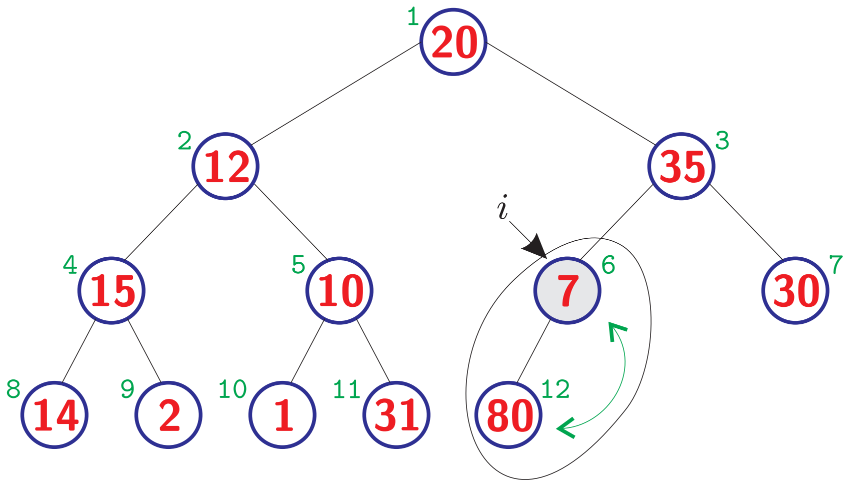


Initializing a MaxHead

- n insertions. Time $O(n \log n)$.
- Playing a tournament. Time $O(n)$.

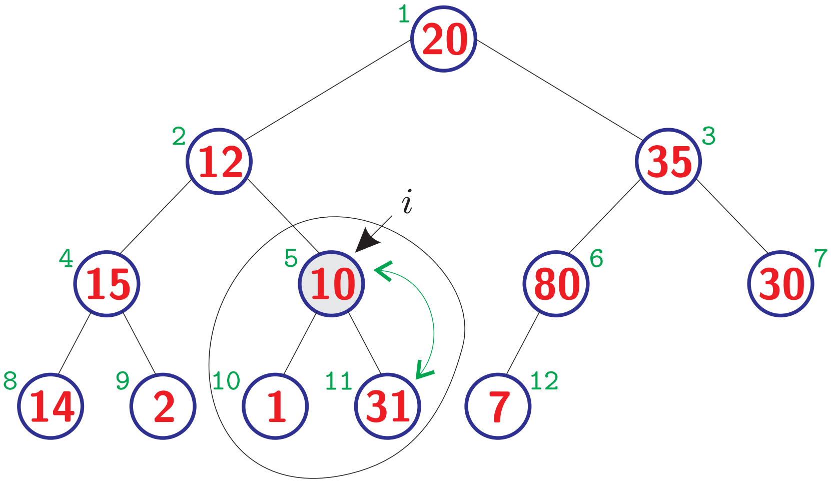
Example: Initialize a heap with `a[1:12] = [20, 12, 35, 15, 10, 7, 30, 14, 2, 1, 31, 80]`

Array `a` may be interpreted as a complete binary tree as follows

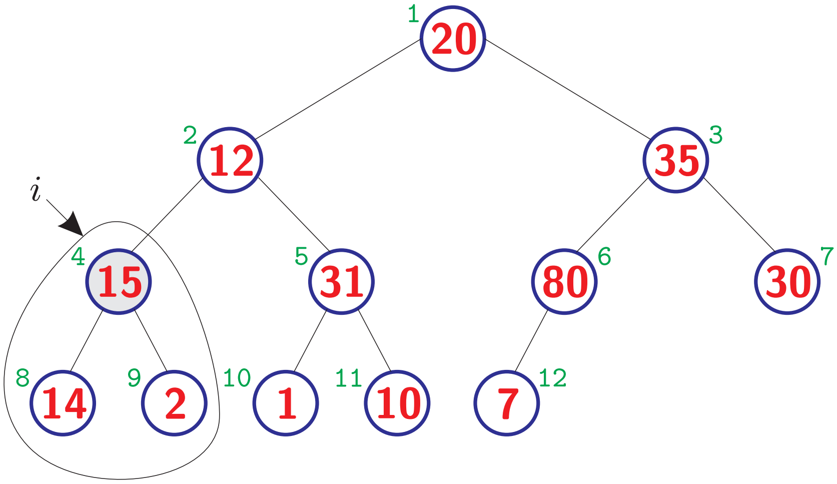


To *heapify* (i.e. make into a max heap) we begin with the last element that has a child. This element must be at position $i = \lfloor n/2 \rfloor = 6$

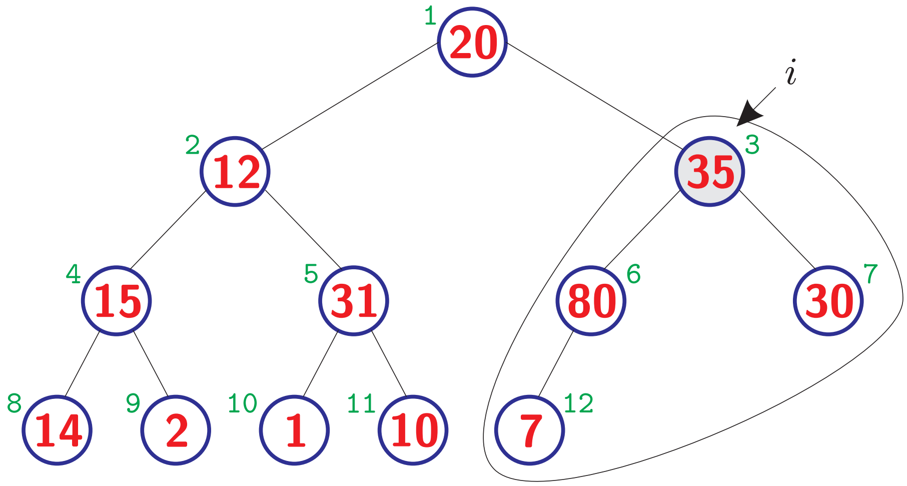
Restore the heap properties for the subtree rooted at node 6 now examine the heap properties for the subtree rooted at node 5



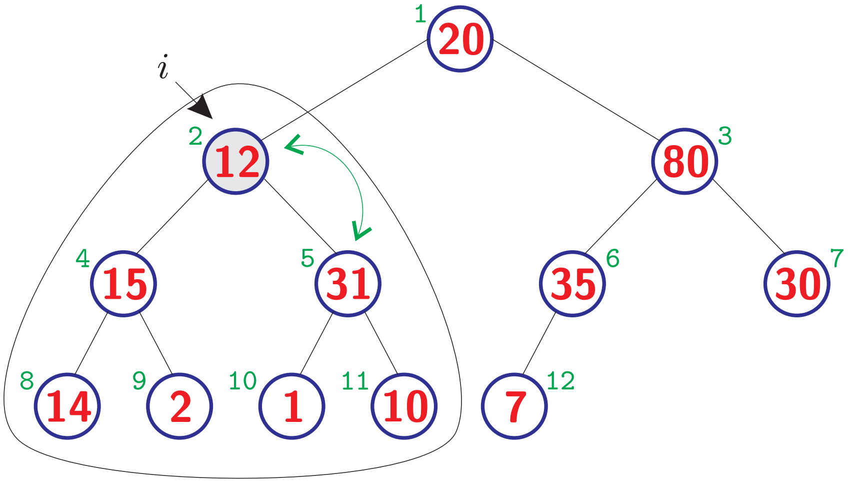
now for the subtree rooted at node 4



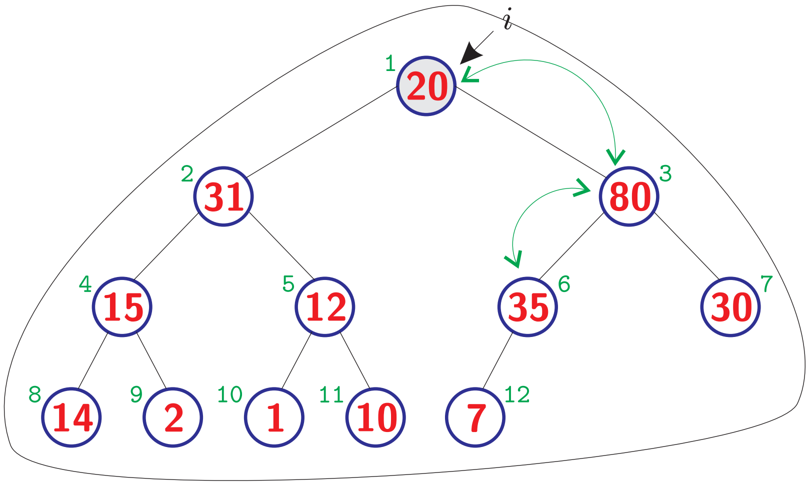
now for the subtree rooted at node 3



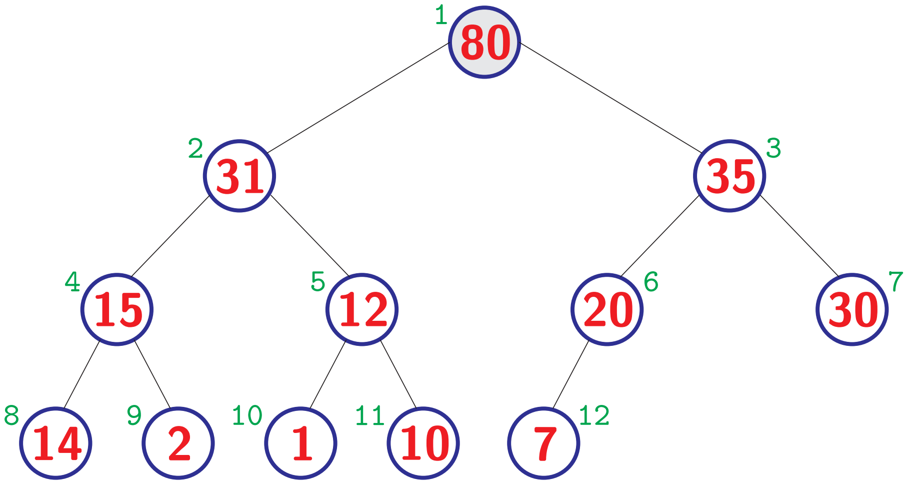
now for the subtree rooted at node 2



now for the subtree rooted at node 1



The resulting max heap is as follows



Class Definition of MaxHeap

```
package unal.datastructures;

import java.util.*;

public class MaxHeap<T extends Comparable<? super T>> ↙
  ↳ implements MaxPriorityQueue<T>
{
   // fields
   T[] heap;  // array for complete binary tree
   int size;  // number of elements in heap

   // constructors
   public MaxHeap( int initialCapacity ) { /* ... */ }
   public MaxHeap( ) { /* ... */ }

   // methods
```

```
   public boolean isEmpty( ) { /* ... */ }
   public int size( ) { /* ... */ }
   public T getMax( ) { /* ... */ }
   public void put( T theElement ) { /* ... */ }
   public T removeMax( ) { /* ... */ }
   public void initialize( T[] theHeap ) { /* ... */ }
   public String toString( ) { /* ... */ }
   public static void main( String[] args ) { /* ... */ }
}
```

constructor

```java
/** create a heap with the given initial capacity
 * @throws IllegalArgumentException when
 * initialCapacity < 1 */
@SuppressWarnings( "unchecked" )
public MaxHeap( int initialCapacity )
{
   if( initialCapacity < 1 )
      throw new IllegalArgumentException
                 ( "initialCapacity␣must␣be␣>=␣1" );
   heap = ( T[] ) new Comparable[ initialCapacity + 1 ];
   size = 0;
}

/** create a heap with initial capacity 10 */
public MaxHeap( )
{
   this( 10 );
}
```

isEmpty

```java
/** @return true iff tree is empty */
public boolean isEmpty( )
{
   return size == 0;
}
```

size

```
/** @return number of elements in the heap */
public int size( )
{
   return size;
}
```

getMax

```
/** @return maximum element
 * @return null if the heap is empty */
public T getMax( )
{
   return ( size == 0 ) ? null : heap[ 1 ];
}
```

put

```java
/** put theElement into the heap */
@SuppressWarnings( "unchecked" )
public void put( T theElement )
{
   // increase array size if necessary
   if( size == heap.length - 1 )
   {
      T[] old = heap;
      heap = ( T[] ) new Comparable[ 2 * heap.length ];
      for( int i = 0; i < old.length; i++ )
         heap[ i ] = old[ i ];
   }

   // find place for theElement
   // currentNode starts at new leaf and moves up tree
   int currentNode = ++size;
   while( currentNode != 1 &&
         heap[ currentNode / 2 ].compareTo( theElement ) < 0 )
   {
```

```java
      // cannot put theElement in heap[ currentNode ]
      heap[ currentNode ] = heap[ currentNode / 2 ]; // move element down
      currentNode /= 2;                      // move to parent
   }

   heap[ currentNode ] = theElement;
}
```

removeMax

```
/** remove max element and return it */
public T removeMax( )
{
   // if heap is empty return null
   if( size == 0 ) return null;    // heap empty

   T maxElement = heap[ 1 ]; // max element

   // reheapify
   T lastElement = heap[ size-- ];

   // find place for lastElement starting at root
   int currentNode = 1,
       child = 2;    // child of currentNode
   while( child <= size )
   {
      // heap[ child ] should be larger child of currentNode
      if( child < size &&
         heap[ child ].compareTo( heap[ child + 1 ] ) < 0 ) ↙
```

```
              ↪ child++;

      // can we put lastElement in heap[ currentNode ]?
      if( lastElement.compareTo( heap[ child ] ) >= 0 )
         break;  // yes

      // no
      heap[ currentNode ] = heap[ child ]; // move child up
      currentNode = child;          // move down a level
      child *= 2;
   }
   heap[ currentNode ] = lastElement;

   return maxElement;
}
```

initialize

```
/** initialize max heap to element array theHeap */
@SuppressWarnings("unchecked")
public void initialize( T[] theHeap )
{
   int theSize = theHeap.length;
   heap = ( T[] ) new Comparable[ theSize + 1 ];
   for( int i = 1; i< heap.length; i++ )
      heap[ i ] = theHeap[ i - 1 ];
   size = theSize;
   // heapify
   for( int root = size / 2; root >= 1; root-- )
   {
      T rootElement = heap[ root ];

      // find place to put rootElement
      int child = 2 * root; // parent of child is target
                          // location for rootElement
      while( child <= size )
      {
```

```
         // heap[ child ] should be larger sibling
         if( child < size &&
            heap[ child ].compareTo( heap[ child + 1 ] ) < 0 ) ↙
              ↪ child++;

         // can we put rootElement in heap[ child / 2 ]?
         if( rootElement.compareTo( heap[ child ] ) >= 0 )
            break; // yes

         // no
         heap[ child / 2 ] = heap[ child ]; // move child up
         child *= 2;                        // move down a level
      }
      heap[ child / 2 ] = rootElement;
   }
}
```

toString

```
@Override
public String toString( )
{
   StringBuilder s = new StringBuilder( );
   s.append( "The␣" + size + "␣elements␣are␣[␣" );
   if( size > 0 )
   {  // nonempty heap
      // do first element
      s.append( Objects.toString( heap[ 1 ] ) );
      // do remaining elements
      for( int i = 2; i <= size; i++ )
         s.append( ",␣" + Objects.toString( heap[ i ] ) );
   }
   s.append( "␣]" );

   return new String( s );
}
```

main

```
/** test program */
public static void main( String[] args )
{
   // test constructor and put
   MaxHeap<Integer> h = new MaxHeap<>( 4 );
   h.put( new Integer( 10 ) );
   h.put( new Integer( 20 ) );
   h.put( new Integer( 5 ) );

   // test toString
   System.out.println( "Elements␣in␣array␣order␣are" );
   System.out.println( h );
   System.out.println( );

   h.put( new Integer( 15 ) );
   h.put( new Integer( 30 ) );

   System.out.println( "Elements␣in␣array␣order␣are" );
   System.out.println( h );
```

```java
      System.out.println( );

      // test remove max
      System.out.println( "The␣max␣element␣is␣" + h.getMax( ) );
      System.out.println( "Deleted␣max␣element␣" + h.removeMax( ) );
      System.out.println( "Deleted␣max␣element␣" + h.removeMax( ) );
      System.out.println( "Elements␣in␣array␣order␣are" );
      System.out.println( h );
      System.out.println( );

      // test initialize
      Integer[] z = new Integer[ 10 ];
      for( int i = 0; i < 10; i++ )
         z[ i ] = new Integer( i );
      h.initialize( z );
      System.out.println( "Elements␣in␣array␣order␣are" );
      System.out.println( h );
   }
```

Compiling `MaxHeap.java`

```
C:\2016699\code> javac unal\datastructures\MaxHeap.java ↵
C:\2016699\code> java unal.datastructures.MaxHeap ↵
Elements in array order are
The 3 elements are [ 20, 10, 5 ]

Elements in array order are
The 5 elements are [ 30, 20, 5, 10, 15 ]

The max element is 30
Deleted max element 30
Deleted max element 20
Elements in array order are
The 3 elements are [ 15, 10, 5 ]

Elements in array order are
The 10 elements are [ 9, 8, 6, 7, 4, 5, 2, 0, 3, 1 ]
```

Heap Application

Heap Sort

A heap can be used to sort n elements in $O(n \log n)$ time

1) Initialize a max heap with n elements (time $O(n)$)

2) Extract (i.e. delete) elements from the heap one at a time. Each deletion takes $O(\log n)$ time, so the total time is $O(n \log n)$

File HeapSort.java

```java
package unal.applications;

import unal.datastructures.*;

public class HeapSort
{
   /** sort the elements a[0 : a.length - 1] using
    * the heap sort method */
   public static <T extends Comparable<? super T>> void heapSort( T[] a )
   {
      // create a max heap of the elements
      MaxHeap<T> h = new MaxHeap<>( );
      h.initialize( a );

      // extract one by one from the max heap
      for( int i = a.length - 1; i >= 0; i-- )
         a[ i ] = h.removeMax( );
   }
   /** test program */
   public static void main( String [ ] args )
   {
      Integer[] a = { new Integer( 3 ),
                    new Integer( 2 ),
                    new Integer( 4 ),
                    new Integer( 1 ),
                    new Integer( 6 ),
                    new Integer( 9 ),
                    new Integer( 8 ),
                    new Integer( 7 ),
                    new Integer( 5 ),
                    new Integer( 0 )};

      // output elements to be sorted
      System.out.println( "The␣elements␣are" );
      for( int i = 0; i < a.length; i++ )
         System.out.print( a[ i ] + "␣" );
      System.out.println( );

      // sort the elements
```

```
      heapSort( a );

      // output in sorted order
      System.out.println( "The␣sorted␣order␣is" );
      for( int i = 0; i < a.length; i++ )
         System.out.print( a[ i ] + "␣" );
      System.out.println( );
   }
}
```

Compiling `HeapSort.java`

```
C:\2016699\code> javac unal\applications\HeapSort.java ↵
C:\2016699\code> java unal.applications.HeapSort ↵
The elements are
3 2 4 1 6 9 8 7 5 0
The sorted order is
0 1 2 3 4 5 6 7 8 9
```