BINARY TREES AND HEAPS IN JAVA

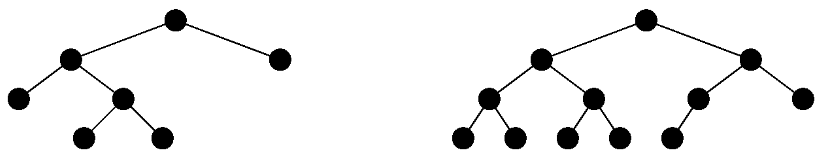
```
// DSutil.java
import java.util.*;

// A bunch of utility functions.
public class DSutil {

  // Swap two objects in an array
  public static void swap(Object[] array, int p1, int p2) {
    Object temp = array[p1];
       array[p1] = array[p2];
       array[p2] = temp;
  }

  // Randomly permute the Objects in an array
  static void permute(Object[] A) {
     for (int i = A.length; i > 0; i--)  // for each i
        swap(A, i-1, DSutil.random(i));  //   swap A[i-1] with
  }                                      //   a random element

  // Create a random number function to return values
  // uniformly distributed within the range 0 to n-1.
  static private Random value = new Random();// Random class object
    static int random(int n) { // My own function
          return Math.abs(value.nextInt()) % n;
    }
  }

// Elem.java
// Elem interface. This is just an Object with support for a key field.
interface Elem {             // Interface for generic element type
  public abstract int key(); // Key used for search and ordering
} // interface Elem

// IElem.java
// Sample implementation for Elem interface: a record w/ just an int field
public class IElem implements Elem {

  private int value;
  public IElem(int v) { value = v; }
  public IElem() {value = 0;}

  public int key() { return value; }
  public void setkey(int v) { value = v; }

  public String toString() { // Override Object.toString
    return Integer.toString(value);
  }
}
```

Binary Trees

A binary tree is made up of a finite set of nodes that is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root.

Full binary tree: Each node is either a leaf or internal node with exactly two non-empty children.

Complete binary tree: is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



(a) (b)

Traversals

Any process for visiting the nodes in some order is called a traversal.

Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes.

•Preorder traversal: Visit each node before visiting its children.
•Postorder traversal: Visit each node after visiting its children.
•Inorder traversal: Visit the left subtree, then the node, then the right subtree.

```java
// BinNode.java

interface BinNode { // ADT for binary tree nodes

        // Return and set the element value
        public Object element();
        public Object setElement(Object v);
        // Return and set the left child
        public BinNode left();
        public BinNode setLeft(BinNode p);

        // Return and set the right child
        public BinNode right();
        public BinNode setRight(BinNode p);

        // Return true if this is a leaf node
        public boolean isLeaf();

} // end interface BinNode


// BinNodePtr.java – we do not optimize the code using two different
// implementations of the BinNode interface for internal and leaf nodes
// (leafs do not need to waste space for storing any child pointer).
public class BinNodePtr implements BinNode {
    private Object elem;
    private BinNode left;
    private BinNode right;

    public BinNodePtr()                                    // Constructor 1
    { left = right = null; }
    public BinNodePtr(Object val)                          // Constructor 2
    { left = right = null; elem  = val; }
    public BinNodePtr(Object val, BinNode l, BinNode r) // Constructor 3
    { left = l; right = r; elem = val; }

    public Object element() { return elem; }
    public Object setElement(Object v) { return element = v; }

    public BinNode left() { return left; }
    public BinNode setLeft(BinNode p) { return left = p; }

    public BinNode right() { return right; }
    public BinNode setRight(BinNode p) { return right = p; }

    public boolean isLeaf() { return (left == null)&&(right == null); }

} // end class BinNodePtr
```

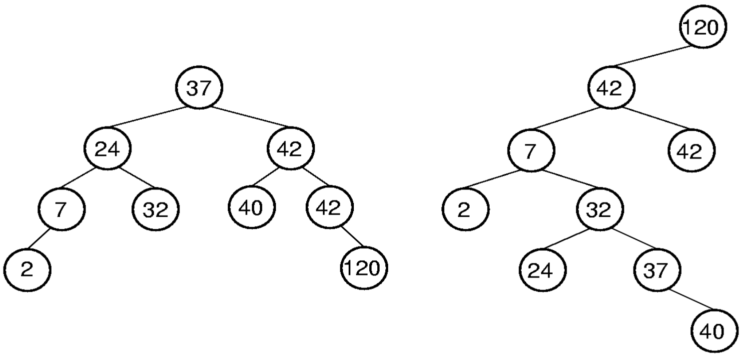
```java
// Main.java
public class Main {

    public static void main(String[] args) {
        // TODO code application logic here
        ……………
    }
    public static void visit(BinNode rt) {
       System.err.println(":"+rt.element());
    }
    public static void preorder(BinNode rt) // rt is root of subtree
    {
     if (rt == null) return; // Empty subtree
     visit(rt);
     preorder(rt.left());
     preorder(rt.right());
    }
    public static void postorder(BinNode rt) // rt is root of subtree
    {
     if (rt == null) return; // Empty subtree
     postorder(rt.left());
     postorder(rt.right());
     visit(rt);
    }

    public static void inorder(BinNode rt) // rt is root of subtree
    {
     if (rt == null) return; // Empty subtree
     inorder(rt.left());
     visit(rt);
     inorder(rt.right());
    }
} // end class Main
```

Binary Search Tree (BST)

Lists have a major problem:
either insert/delete, on the one hand, or search, on the other, must be O(*n*) time.
How can we make both update and search efficient ?
Answer: Use a new data structure...BST

BST Property

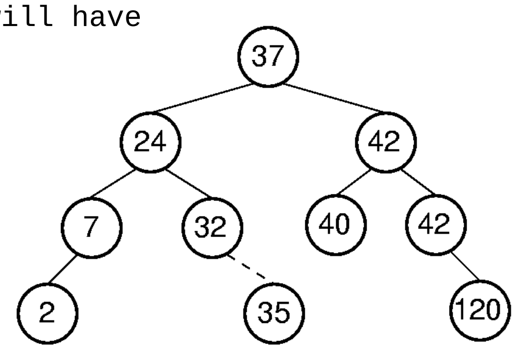
1. Key elements in <u>the **left subtree**</u> of a node with value *K* have key values < *K*
2. Key elements in <u>the **right subtree**</u> of a node with value *K* have key values >= *K*




```
public class BST { // BST implementation
        private BinNode root; // The root of the tree

        public BST() { root = null; }           // Initialize root

        public void clear() { root = null; }

        public boolean isEmpty() { return root == null; }

        public void print() {
           if (root == null) System.err.println("The BST is empty!");
           else {
                  printhelp(root, 0);
                  System.out.println();
           }
        }

        private void printhelp(BinNode rt, int level) {
           // In-Order visit
           if (rt == null) return;

            printhelp(rt.left(), level+1);

            for (int i = 0; i < level; i++)   // Indent based on level
                System.err.print("  ");
            System.err.println(rt.element()); // Print node value

            printhelp(rt.right(), level+1);
        }
```

// Source code examples based on "A Practical Introduction to Data Structures and Algorithm
// Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

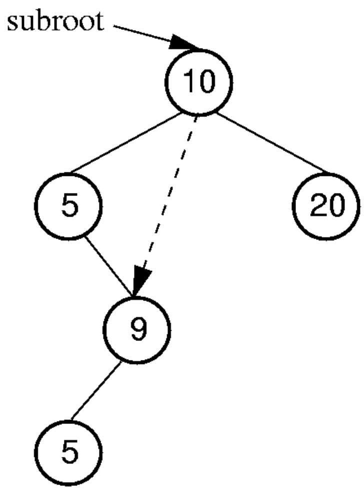
```
public Elem find(Elem key)
{ return findhelp(root, key); }


private Elem findhelp(BinNode rt, int key) {
        if (rt == null) return null;
        Elem it = (Elem)rt.element();
        if (key < it.key())
          return findhelp(rt.left(), key);
        else if (it.key() == key)
              return it;
             else
                 return findhelp(rt.right(), key);
}



public void insert(Elem val)
{ root = inserthelp(root, val); }


// The method returns a subtree identical to the old one except
// that it has been modified to contain the new node being inserted

// Convention: Insert duplicates in the right subtree.

// First find where the key "val" would have been if it were in
// the tree:
//            1) a leaf node or
//            2) an internal node with one child
// Then add a new node with key "val".

private BinNode inserthelp(BinNode rt, Elem val) {

      if (rt == null) return new BinNodePtr(val);
      Elem it = (Elem) rt.element();
      if (val.key() < it.key()) {
           BinNode toLeft = inserthelp(rt.left(), val);
           rt.setLeft(toLeft);
      }
      else {
           BinNode toRight = inserthelp(rt.right(), val);
           rt.setRight(toRight);
      }
      return rt;
 // Only the parent of the added node will have
 //  its child pointer modified
}
```



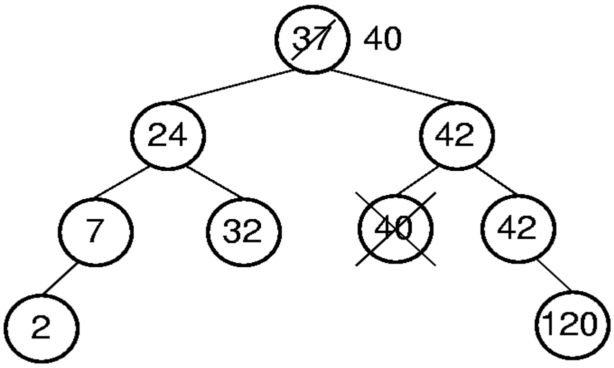
// Source code examples based on "A Practical Introduction to Data Structures and Algorithm
// Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

```
// Routines to get and remove the node with the smallest key.
// A node with the minimum key value will always be positioned as
// a left leaf of the BST, even in case of keys having duplicate
// values.
private Elem getmin(BinNode rt) {
      if (rt.left() == null)
        return (Elem)rt.element();
      else return getmin(rt.left());
}

// The method returns a subtree identical to the old
// one except that it has been modified deleting a
// node with the minimum key

// The parent of the node with the minimum key (S)
// has to change its left child to point to
// the right child of S.
private BinNode deletemin(BinNode rt) {
      if (rt.left() == null)
        return rt.right();
      else {
              BinNode toLeft = deletemin(rt.left());
              rt.setLeft(toLeft);
              return rt;
      }
}
```

Example with a duplicate minimum node:



```
// Removing an arbitrary node R from the BST requires that:
// (1) we first find R
// (2) we remove it from the tree taking care of the following cases:
// -- If R has no children then the pointer of Parent(R) is set to NULL
// -- If R has one child then the pointer of Parent(R) is set to R's child
// -- If R has two children:
//       Find a value in one of the two subtree that can replace R
//       preserving the BST property...that is substitute R with
//  the least value of the right subtree (which is the In-Order Successor)
// (in such a way we pick up a value that is less than others on the
// right and is also greater than all nodes on the left of the tree)
//  we'll make use of the previous method : BinNode deletemin(BinNode rt)
// (Preferred if the tree contains duplicates because of the convention
// about the insertion of duplicates)
//                                    OR
//the greatest value of the left subtree (In-Order Predecessor)
```

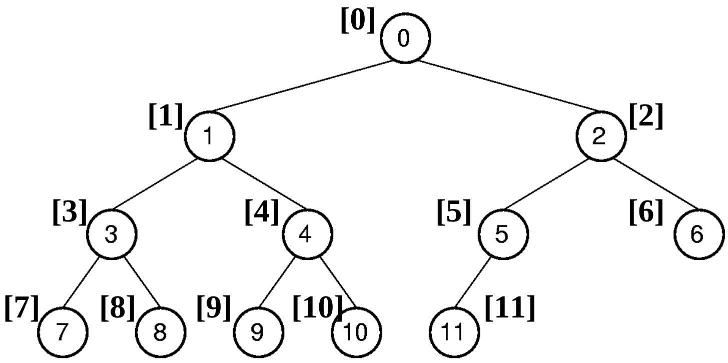



```
public void remove(int key) { root = removehelp(root, key); }

// The method returns a subtree identical to the old one except that it
// has been modified deleting a node with the minimum key

private BinNode removehelp(BinNode rt, int key) {
     if (rt == null) return null;
     Elem it = (Elem) rt.element();
     if (key < it.key())
       rt.setLeft(removehelp(rt.left(), key));
     else if (key > it.key())
            rt.setRight(removehelp(rt.right(), key));
          else { // now we have arrived to the node R
                 if (rt.left() == null)
                   rt = rt.right();
                 // Parent(R)'s link set to the other child of R
                 else if (rt.right() == null)
                        rt = rt.left();
                      else {
                             Elem temp = getmin(rt.right());
                             rt.setElement(temp);
                             rt.setRight(deletemin(rt.right()));
                      }
                 }
                 return rt;
          }
} // end class BST
```

Complete Binary Tree

A CBT is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Since a complete binary tree is so limited in its shape (there is only one possible shape for n nodes), it is reasonable to expect that space efficiency can be achieved with an array representation.



Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

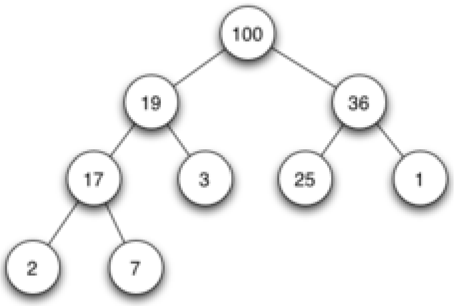
Parent(r) $= (r - 1)/2$ **if $0 < r < n$**
Leftchild(r) $= 2r + 1$ **if $2r + 1 < n$**
Rightchild(r) $= 2r + 2$ **if $2r + 2 < n$**
Leftsibling(r) $= r - 1$ **if r is even, $r > 0$, and $r < n$**
Rightsibling(r) $= r + 1$ **if r is odd, and $r + 1 < n$**

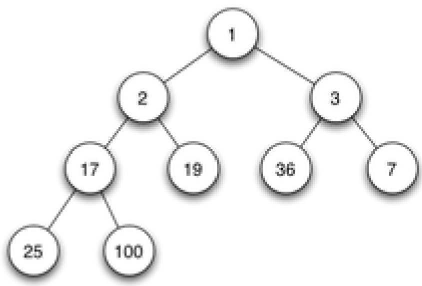
HEAP

Definition:
Complete binary tree with the **heap property**:

- **Max-heap**: every node store a value that is greater than or equal to the values of either of its children.

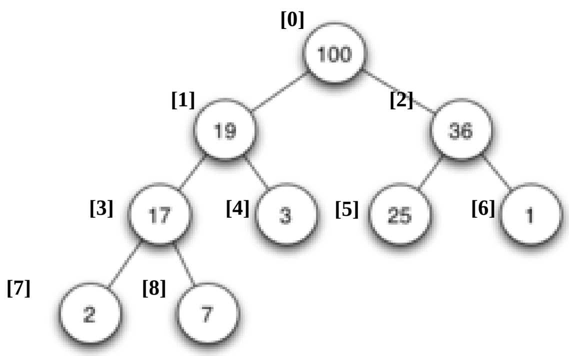
- **Min-heap**: every node store a value that is less than or equal to the values of either of its children.





The values are **partially** ordered.
An In-Order visit **does not** compute a sorted list of the key values, as well as any other type of traversal.

Heap representation:
normally, the array-based "complete binary tree" (CBT) representation.



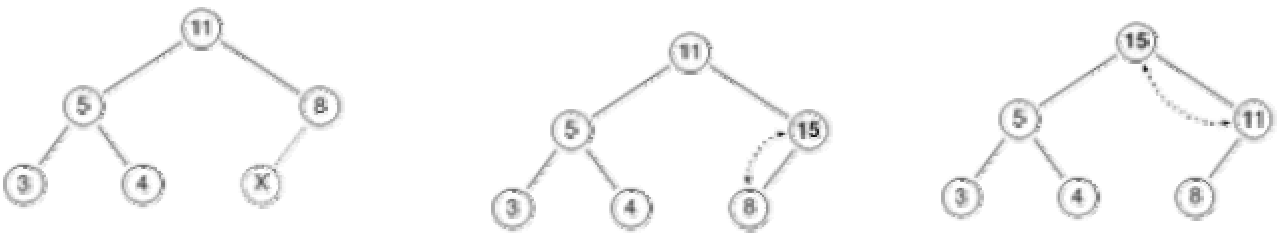
Obs.: In a CBT the leafs are positioned in the second half of the array.
In the above example with n = 9 the interval of the array positions storing a leaf value is [n/2, n-1] = [4,8].

// Source code examples based on "A Practical Introduction to Data Structures and Algorithm
// Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

```
public class MaxHeap {

        private Elem[] Heap;   // Pointer to heap array
        private int maxsize;   // Maximum size of the heap
        private int size;      // Number of elements in heap

        public MaxHeap(Elem[] h) // heapify an input array
        {
             Heap = h;
             size = maxsize = Heap.length;
             buildheap();    // see next
        }

        public int heapSize()          // Return size of heap
        { return size; }

        public int heapSizeLimit()     // Return size of heap
        { return maxsize; }

        public boolean isLeaf(int pos) // true if pos is leaf
        { return (pos >= size/2) && (pos < size); }

        public int parent(int pos) {   // Return pos for parent
               return (pos-1)/2;
        }

        // Return position for left child of pos
        public int leftchild(int pos) {
               return 2*pos + 1;
        }

        // Return position for right child of pos
        public int rightchild(int pos) {
               return 2*pos + 2;
        }


        // Continue on the next page...
```

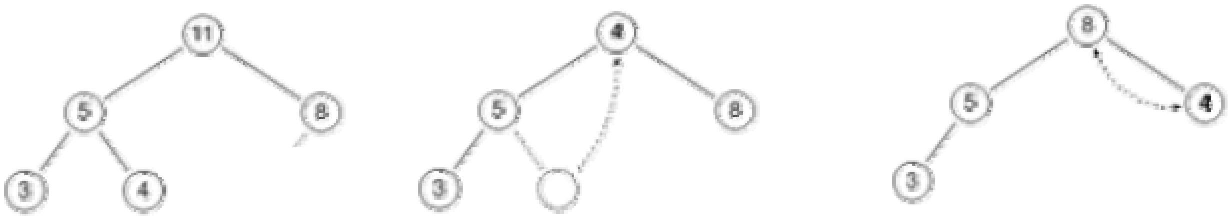
// Inserting a value in a Max-Heap

```
// If we have a heap, and we add an element, we can perform an operation
// known as sift-up in order to restore the heap property.
// We can do this in O(log n) time, using a binary heap, by following this
// algorithm:
//
//                (1) Add the element on the bottom level of the heap.
//                (2) Compare the added element with its parent;
//                        if they are in the correct order, stop.
//                (3) If not, swap the element with its parent and return to
//                    the previous step.
//
// We do this at maximum for each level in the tree — the height of the
// tree, which is O(log n). However, since approximately 50% of the
// elements are leaves and 75% are in the bottom two levels, it is likely
// that the new element to be inserted will only move a few levels upwards
// to maintain the heap. Thus, binary heaps support insertion in average
// constant time, O(1).

// X = 15
```




```
public void insert(Elem val) {
   if (size < maxsize) {

     int curr = size++;

     Heap[curr] = val;               // Start at the end of the Heap

     // Sift-up until curr’s parent’s key < curr’s key
      while (curr!=0 && Heap[curr].key() > Heap[parent(curr)].key()) {

             DSutil.swap(Heap, curr, parent(curr));
             curr = parent(curr);

      }
    }
}
```

```
// Removing the max value in a Max-Heap

// The procedure starts by swapping it with the last element on the last
// level. So, if we have the same max-heap as before, we remove the 11 and
// replace it with the 4
//
// Now the heap property is violated since 8 is greater than 4.
// The operation that restores the property is called sift-down.
// In this case, swapping the two elements 4 and 8, is enough to restore
// the heap property and we need not swap elements further.
//
// In general, the wrong node is swapped with its larger child in
// a max-heap (in a min-heap it would be swapped with its smaller child),
// until it satisfies the heap property in its new position.
//
// Note that the down-heap operation (without the preceding swap) can be
// used in general to modify the value in any position ...siftdown(pos).
```




```
public Elem removeMax() {
    if (size > 0) {
      DSutil.swap(Heap,0,--size); // Swap max with the last value
      if (size!=0)                // not the last element
        siftdown(0);    // put a new heap root value in the correct place
    }
}

private void siftdown(int pos) { // Put in place
      Assert.notFalse( pos >= 0 && pos < size, “Illegal heap Position!”);
       while (!isLeaf(pos)) {

           // Assign to an int j the position of the
           // child with the maximum key value.
           // The Heap is a CBT, thus an internal node always have a left
           // child, however such a child may not have a sibling...
           // see next..the condition j < size-1...

           int j = leftchild(pos); // rightchild(pos) == j+1 holds
           if ( j < size-1 && Heap[j].key() < Heap[j+1].key() )
             j++;

           if ( Heap[pos].key() >= Heap[j].key() )
             return;
           else {
                  DSutil.swap(Heap, pos, j);
                  pos = j;        // Move down
           }
       }
 }
```

// Source code examples based on "A Practical Introduction to Data Structures and Algorithm
// Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

// Building a Heap

```
// This procedure makes a heap out of an array.
// In other words, it rearranges elements of the array so the array
// satisfies the heap property.
// It works by heapifying the elements starting from the middle of the
// array toward the first element (non internal-nodes).
//
// The runtime of this algorithm is O(n) on an array-based heap
// implementation, where n is the number of nodes in the heap.


public void buildheap() // Heapify contents of Heap
{
    // it is assumed to be invoked only by the constructor;
    // the size has been initialized with the length of the array
    for (int i=size/2-1; i>=0; i--)
       siftdown(i);
}
```

// Remove an element at specified position

```
public Elem remove(int pos) {
    if (pos >=0 && pos < size) {
       DSutil.swap(Heap, pos, --size); // Swap with last value

       // sift up
       while (pos != 0 && Heap[pos].key() > Heap[parent(pos)].key()) {
            DSutil.swap(Heap, pos, parent(pos));
            pos = parent(pos);
       }

       if (size != 0) siftdown(pos); // push down
       return Heap[size];
    }
}


} // end class MaxHeap
```

HeapSort

- Strategy
  - Build an Heap with the array elements
  - Removes the heap's root (the largest element) by exchanging it with the heap’s last element
  - Transforms the resulting semi-heap back into a heap
- Efficiency
  - Compared to mergesort
    - Both heapsort and mergesort are O(n log n)
      in both the worst and average cases
    - Advantage over mergesort
      - Heapsort does not require a second array
  - Compared to quicksort
    - Quicksort performs better in the average case
      (empirically: for a constant factor)

```
// Java code to add at the file sortmain.java as reported on the lecture
// about Internal Sorting Algorithms.

static void siftDownForMaxHeap(Elem[] array, int pos, int hSize) {
   int leftChild;
   while (!((pos >= hSize/2) && (pos < hSize))) { // !isLeaf(pos)
     leftChild = 2*pos + 1;
     if ( leftChild < hSize-1 &&
          array[leftChild].key() < array[leftChild +1].key() )
       leftChild ++;

     if ( array[pos].key() >= array[leftChild].key() )  return;
     else {
            DSutil.swap(array, pos, leftChild);
            pos = leftChild;        // Move down
     }
   }
 }

static void HeapifyMaxHeap(Elem[] array, int hSize) {
    for (int i=hSize/2-1; i>=0; i--)
        siftDownForMaxHeap(array, i, hSize);
}

static void heapSort(Elem[] array) {
      int n = array.length;
      if (n > 1) {
           HeapifyMaxHeap(array, n);
           do {
             // move the current maximum to the end of the array
            // decrease the length of the heap
            // re-heapify the shorter heap (named also “semi-heap”)
            DSutil.swap(array,0,--n);
             if (n > 0) siftDownForMaxHeap(array, 0, n);

           } while (n > 1);
       }
}
```

// Source code examples based on "A Practical Introduction to Data Structures and Algorithm
// Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer