At this lab section, we will cover implementation of the priority queue and heap data structures in Java.

Priority Queue
Max-Min Heap

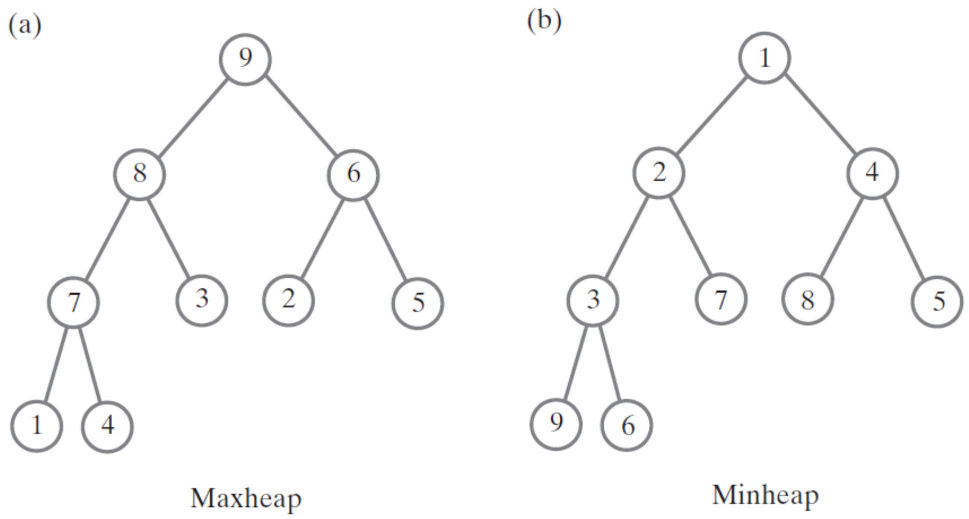
Asst. Prof. Dr. Feriştah DALKILIÇ
Res. Asst. Fatih DİCLE

PART 1 – Heap

A **heap** is a complete binary tree whose nodes contain Comparable objects and are organized as

- In a **maxheap**, the object in a node is **greater** than or **equal** to its descendant objects.
- In a **minheap**, the object in a node is **less** than or **equal** to its descendant objects.

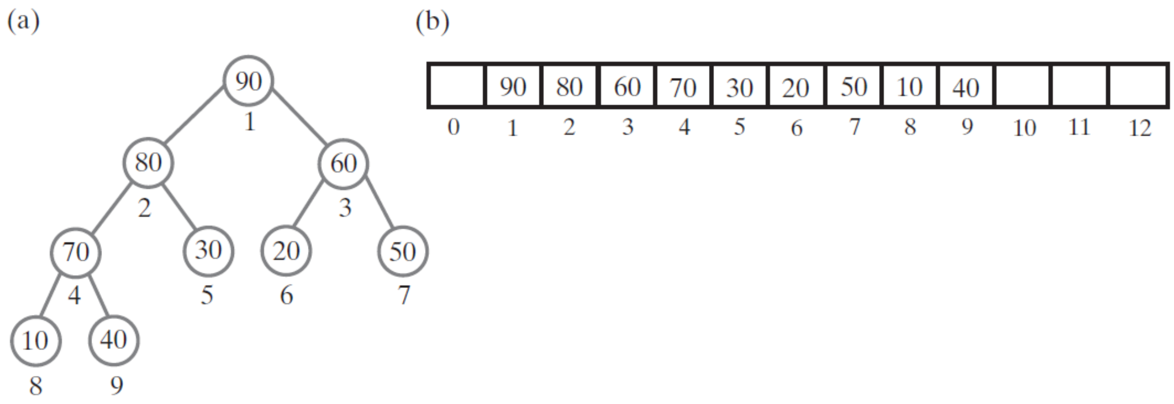
(a) A maxheap and (b) a minheap that contain the same values



When a binary tree is complete, using an array instead of linked nodes is desirable. You can use a level-order traversal to store the tree's data into consecutive locations of an array. This representation enables you to quickly locate the data in a node's parent or children. If you begin storing the tree at index 1 of the array, the node at array index i

- has a parent at index $i/2$, unless the node is the root (i is 1)
- has any children at indices $2i$ and $2i + 1$

(a) A complete binary tree with its nodes numbered in level order;
(b) its representation as an array



	90	80	60	70	30	20	50	10	40			
0	1	2	3	4	5	6	7	8	9	10	11	12

Exercise – 1

At this section, you are given MaxHeapInterface.java and MaxHeap.java to experiment the max heap data structure.

Step – 1

Create a new Java Project. Add the java files MaxHeapInterface.java and MaxHeap.java.

Step – 2

Add a new class with the name of “Test.java” and paste the following code. This code adds the numbers in an Integer array into the max heap data structure one by one and then removes all entries until the heap is not empty.

Test.java

```java
public class Test {

	public static void main(String[] args) {


		Integer[] A = {14, 20, 2, 15, 10, 21};

		MaxHeapInterface<Integer> maxHeap = new MaxHeap<>();
		for (int i = 0; i < A.length; i++) {
			maxHeap.add(A[i]);
		}

		if (maxHeap.isEmpty())
			System.out.println("The heap is empty - INCORRECT");
		else
			System.out.println("The heap is not empty; it contains " +
					maxHeap.getSize() + " entries.");

		System.out.println("The largest entry is " + maxHeap.getMax());

		System.out.println("\n\nRemoving entries in descending order:");
		while (!maxHeap.isEmpty())
			System.out.println("Removing " + maxHeap.removeMax());
	}

}
```

Step – 3

Paste the output of the Test.java.

```
The heap is not empty; it contains 6 entries.
The largest entry is 21



Removing entries in descending order:
Removing 21
Removing 20
Removing 15
Removing 14
Removing 10
Removing 2
```

Exercise – 2

We could create a heap from a collection of objects by using the add method to add each object to an initially empty heap as we experiment in Exercise-1. Since add is an O(log n) operation, creating the heap in this manner would be O(n log n).

We can create a heap more efficiently by using the method **reheap** instead of the method add. Building a heap in this manner is O(n). In applying reheap, we begin at the first nonleaf closest to the end of the array. This nonleaf is at index lastIndex/2, since it is the parent of the last leaf in the tree. We then work toward heap [1].

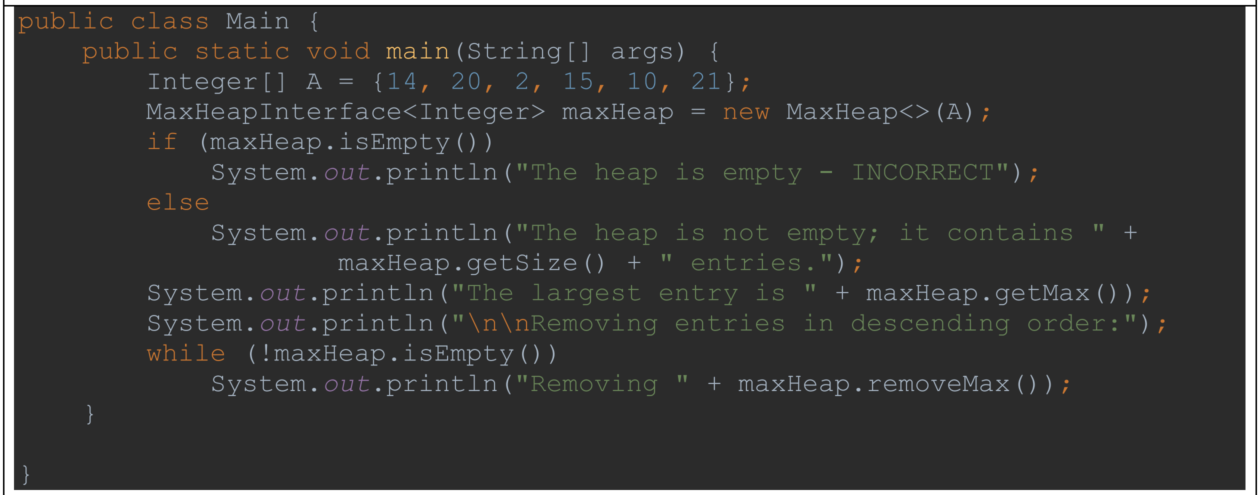
The following Java statements transform the array heap into a heap:

```
for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
      reheap(rootIndex);
```

Step – 1

Modify Test.java to construct a heap without adding entries one by one. Paste your code and output.
Hint: use the constructor that takes a collection as a parameter.

Your Code



```
public class Main {
    public static void main(String[] args) {
        Integer[] A = {14, 20, 2, 15, 10, 21};
        MaxHeapInterface<Integer> maxHeap = new MaxHeap<>(A);
        if (maxHeap.isEmpty())
            System.out.println("The heap is empty - INCORRECT");
        else
            System.out.println("The heap is not empty; it contains " +
                    maxHeap.getSize() + " entries.");
        System.out.println("The largest entry is " + maxHeap.getMax());
        System.out.println("\n\nRemoving entries in descending order:");
        while (!maxHeap.isEmpty())
            System.out.println("Removing " + maxHeap.removeMax());
    }

}
```

Your Output

```
The heap is not empty; it contains 6 entries.
The largest entry is 21


Removing entries in descending order:
Removing 21
Removing 20
Removing 15
Removing 14
Removing 10
Removing 2
```

Exercise – 3

In this section you are expected to modify Exercise-1 to obtain a min heap data structure.

Step – 1

Create the classes MinHeapInterface.java and MinHeap.java by making necessary changes on some method names and content of `add` and `reheap` methods in MaxHeapInterface.java and MaxHeap.java.
Hint: Change comparison operators in `add` and `reheap` methods.

Step – 2

Modify Test.java as follows.

Test.java

```
public class Test {

        public static void main(String[] args) {


                Integer[] A = {14, 20, 2, 15, 10, 21};

                MinHeapInterface<Integer> minHeap = new MinHeap<>();
                for (int i = 0; i < A.length; i++) {
                        minHeap.add(A[i]);
                }

                if (minHeap.isEmpty())
                        System.out.println("The heap is empty - INCORRECT");
                else
                        System.out.println("The heap is not empty; it contains " +
                                        minHeap.getSize() + " entries.");

                System.out.println("The smallest entry is " + minHeap.getMin());

                System.out.println("\n\nRemoving entries in ascending order:");
                while (!minHeap.isEmpty())
                        System.out.println("Removing " + minHeap.removeMin());
        }
}
```

Step – 3

Paste the output of the Test.java.

```
The heap is not empty; it contains 6 entries.
The smallest entry is 2


Removing entries in ascending order:
Removing 2
Removing 10
Removing 14
Removing 15
Removing 20
Removing 21
```

PART 2 –Priority Queue

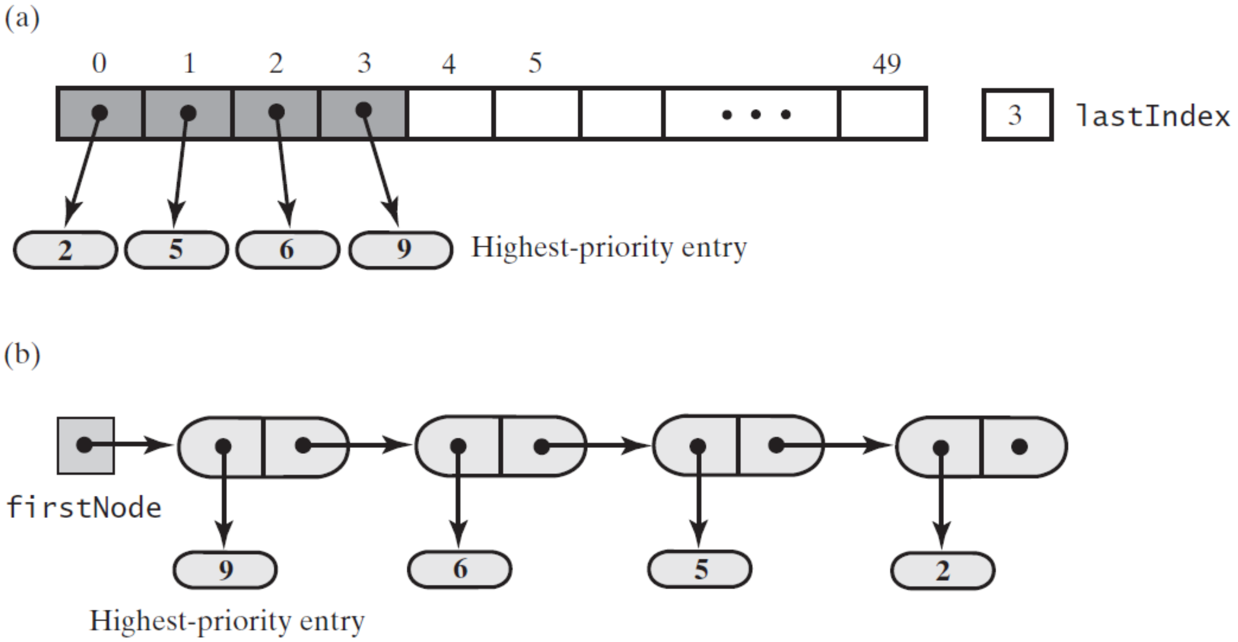
Priority Queue is a more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. We assign priority to an item based on its key value. Lower the value, higher the priority.

Applications of Priority Queue:

- CPU Scheduling
- Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
- All queue applications where priority is involved.

A priority queue can be implemented by using a linked list, an array, or a heap data structure.

Two possible implementations of a priority queue using (a) an array; (b) a chain of linked nodes



Exercise - 4

In this section, we will use a heap to implement the ADT priority queue.

In addition to typical ADT operations such as add, isEmpty, getSize, and clear, a heap has operations that retrieve and remove the object in its root. This object is either the largest or the smallest object in the heap, depending on whether we have a maxheap or a minheap. This characteristic enables us to use a heap to implement the ADT priority queue.

Step – 1

Create PriorityQueueInterface.java that contains the following interface definition.

```
public interface PriorityQueueInterface<T extends Comparable<? super T>>
{
   /** Adds a new entry to this priority queue.
       @param newEntry  An object to be added. */
   public void add(T newEntry);

   /** Removes and returns the entry having the highest priority.
       @return  Either the object having the highest priority or, if the
                priority queue is empty before the operation, null. */
   public T remove();
   /** Retrieves the entry having the highest priority.
       @return  Either the object having the highest priority or, if the
                priority queue is empty, null. */
   public T peek();

   /** Detects whether this priority queue is empty.
       @return  True if the priority queue is empty, or false otherwise. */
   public boolean isEmpty();

   /** Gets the size of this priority queue.
       @return  The number of entries currently in the priority queue. */
   public int getSize();

   /** Removes all entries from this priority queue. */
   public void clear();
} // end PriorityQueueInterface
```

Step – 2

Create HeapPriorityQueue.java class that implements PriorityQueueInterface. Paste the following code and fill in the method bodies.

```
public final class HeapPriorityQueue<T extends Comparable<? super T>>
                   implements PriorityQueueInterface<T>
{
      private MaxHeapInterface<T> pq;

      public HeapPriorityQueue()
      {
            //to be filled
      } // end default constructor

      public void add(T newEntry)
      {
            //to be filled
      } // end add

      public T remove()
      {
            //to be filled
      } // end remove

      public T peek()
      {
            //to be filled
      } // end peek

      public boolean isEmpty()
      {
            //to be filled
      } // end isEmpty

      public int getSize()
      {
            //to be filled
      } // end getSize

      public void clear()
      {
            //to be filled
      } // end clear
} // end HeapPriorityQueue
```

Completed HeapPriorityQueue.java

```
public final class HeapPriorityQueue<T extends Comparable<? super T>>
        implements PriorityQueueInterface<T>
{
    private MaxHeapInterface<T> pq;
    public HeapPriorityQueue()
    {
        pq = new MaxHeap<T>();
    } // end constructor

    public void add(T newEntry)
    {
        pq.add(newEntry);
    } // end add

    public T remove()
    {
        return pq.removeMax();
    } // end remove

    public T peek()
    {
        return pq.getMax();
    } // end peek

    public boolean isEmpty()
    {
        return pq.isEmpty();
    } // end isEmpty

    public int getSize()
    {
        return pq.getSize();
    } // end getSize

    public void clear()
    {
        pq.clear();
    } // end clear
} // end HeapPriorityQueue
```

Step – 3

Add Customer.java that contains the following code.

```
public class Customer implements Comparable<Customer>{
      private String name;
      private Integer priority;

      public Customer(String name, Integer priority)
      {
            this.name = name;
            this.priority = priority;
      }

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }
```

```
	public Integer getPriority() {
		return priority;
	}

	public void setPriority(Integer priority) {
		this.priority = priority;
	}

	@Override
	public int compareTo(Customer other) {
		if(this.priority > other.priority) return 1;
		else if(this.priority < other.priority) return -1;
		else return 0;
	}
}
```

Step – 4

Modify the Test.java to simulate a priority bank queue with customer names and priorities. Add all customers in to the HeapPriorityQueue. Retrieve the customer with highest priority. Print all queue elements. At the end of the process the queue can be empty.

Customer Name	Priority
Berker	5
Kemal	20
Elif	70
Fatma	80
Murat	60
Sevgi	100
Mustafa	10
Merve	80

Your Test.java

```
public class Main {
    public static void main(String[] args) {
        PriorityQueueInterface<Customer> cs = new
HeapPriorityQueue<Customer>();
        cs.add(new Customer("Berker",5) );
        cs.add(new Customer("Kemal",20) );
        cs.add(new Customer("Elif",70) );
        cs.add(new Customer("Fatma",80) );
        cs.add(new Customer("Murat",60) );
        cs.add(new Customer("Sevgi",100) );
        cs.add(new Customer("Mustafa",10) );
        cs.add(new Customer("Merve",80) );

        while (!cs.isEmpty()){
            System.out.print(cs.peek().getName()+"     ");
            System.out.println(cs.remove().getPriority());
        }
    }
}
```

Your Output

```
Sevgi     100
Merve     80
Fatma     80
Elif     70
Murat     60
Kemal     20
```

```
Mustafa    10
Berker    5
```