CIS112

Heaps, HeapSort, Priority Queue

BBBF
Yeditepe University

v2025-05-10

Content

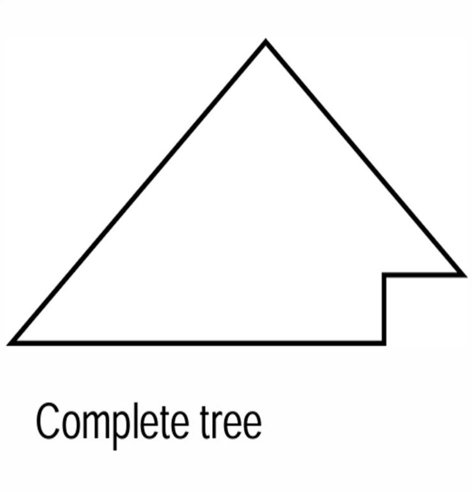
- **Introduction to Heaps**
  - **Removal**
  - **Insertion**
- **Heapsort**
  - **Using the Same Array**
- **Priority Queue**
- **References**

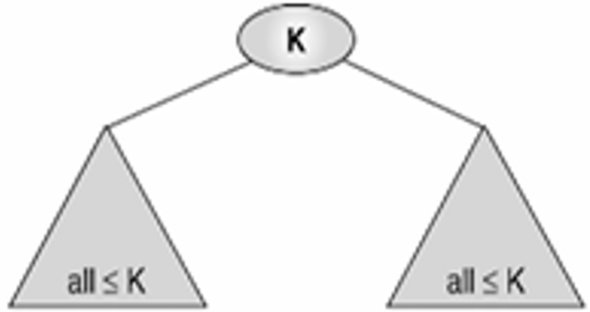
Introduction to Heaps

Introduction to Heaps

A heap is a binary tree with these characteristics:

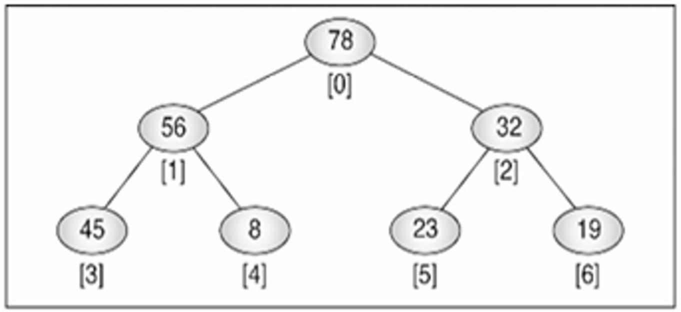
• A complete binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children



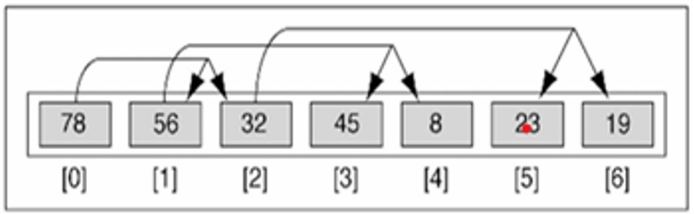


Introduction to Heaps

- It's (usually) implemented as an array.
- Each node in a heap satisfies the *heap condition*, which states that every node's key is larger than (or equal to) the keys of its children.

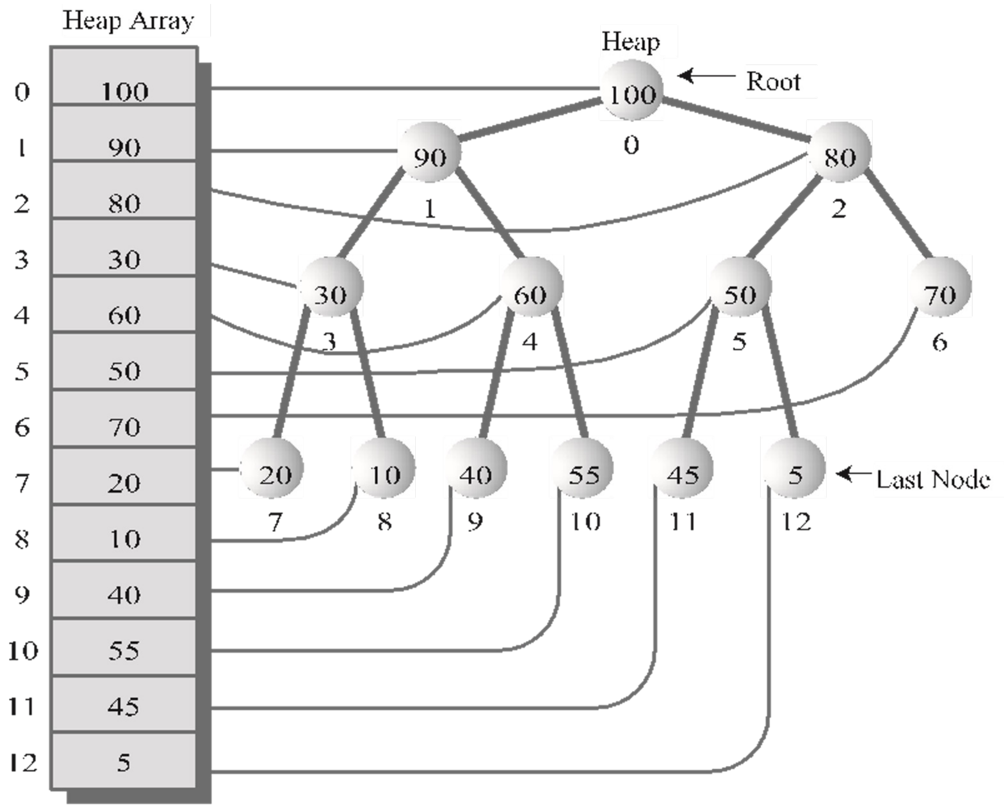


(a) Heap in its tree form



(b) Heap in its array form

Introduction to Heaps
A heap and its underlying array



For a node at index x in the array:

- Its parent is (x-1) / 2
- Its left child is 2*x + 1
- Its right child is 2*x + 2

Introduction to Heaps
Insertion

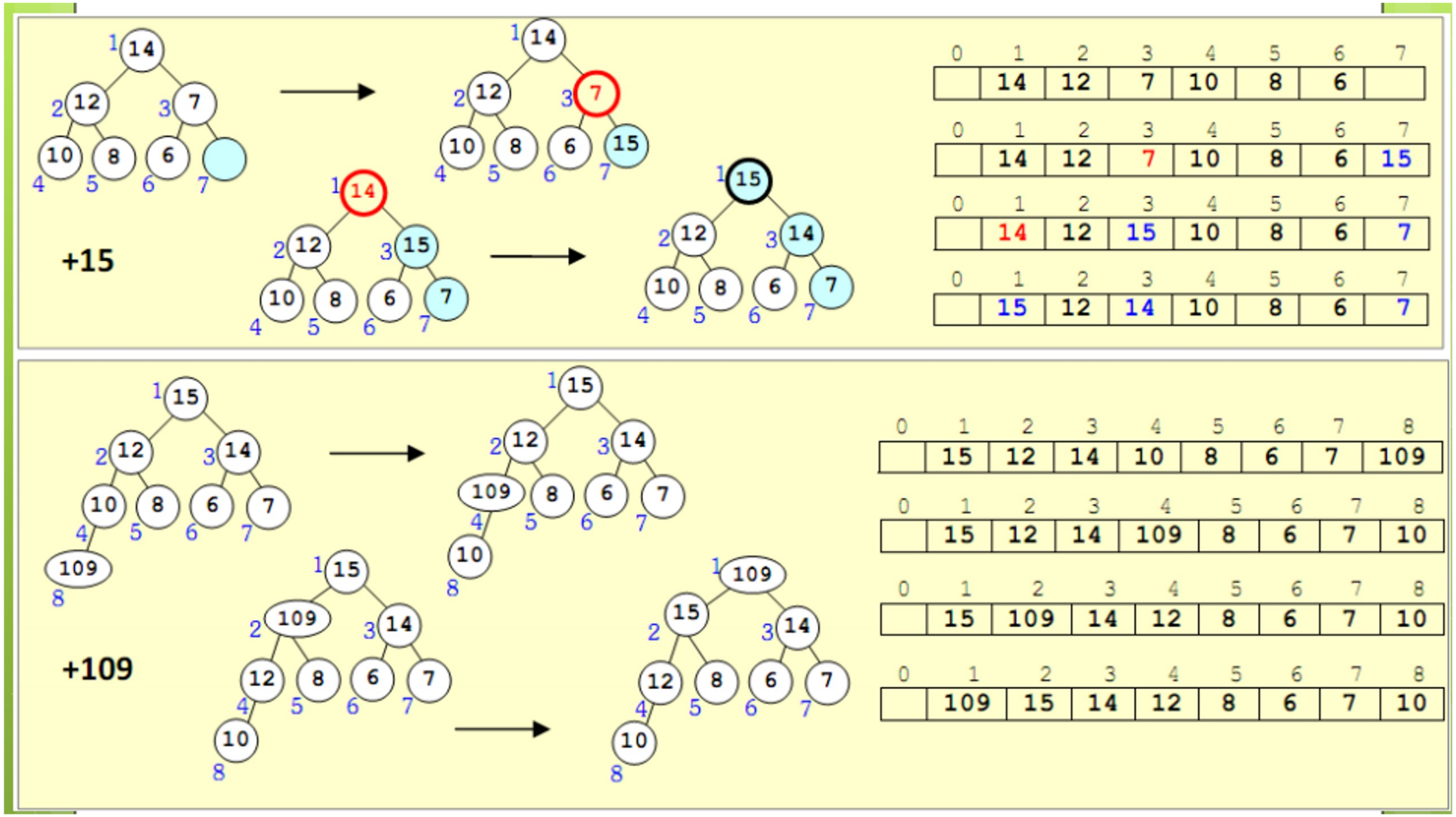
The node to be inserted is placed in the first open position at the end of the array, increasing the array size by one:

heapArray[N] = newNode;

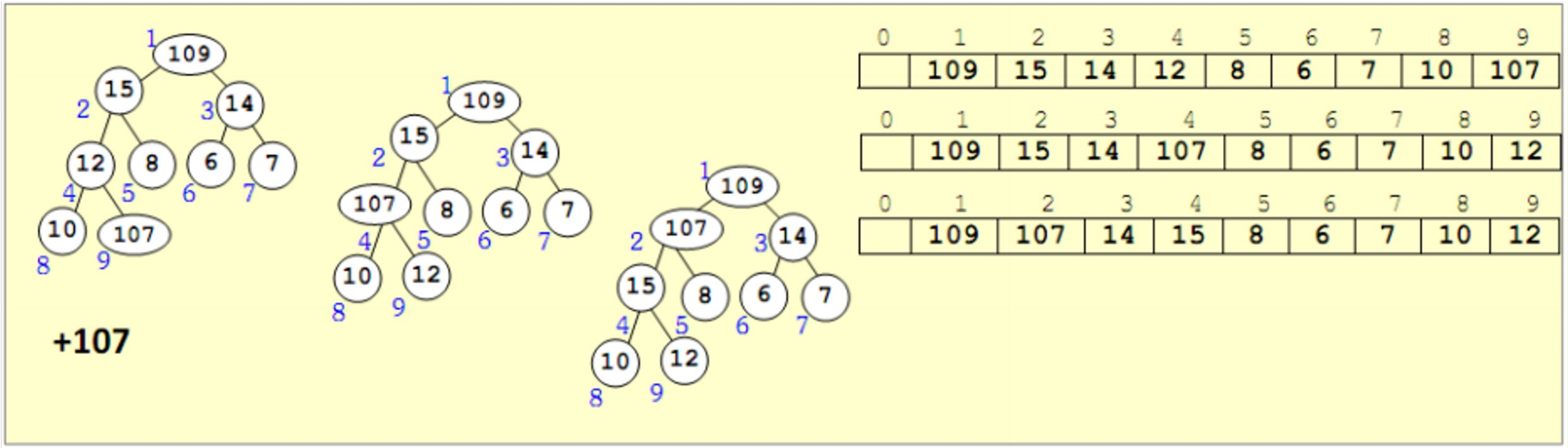
N++;

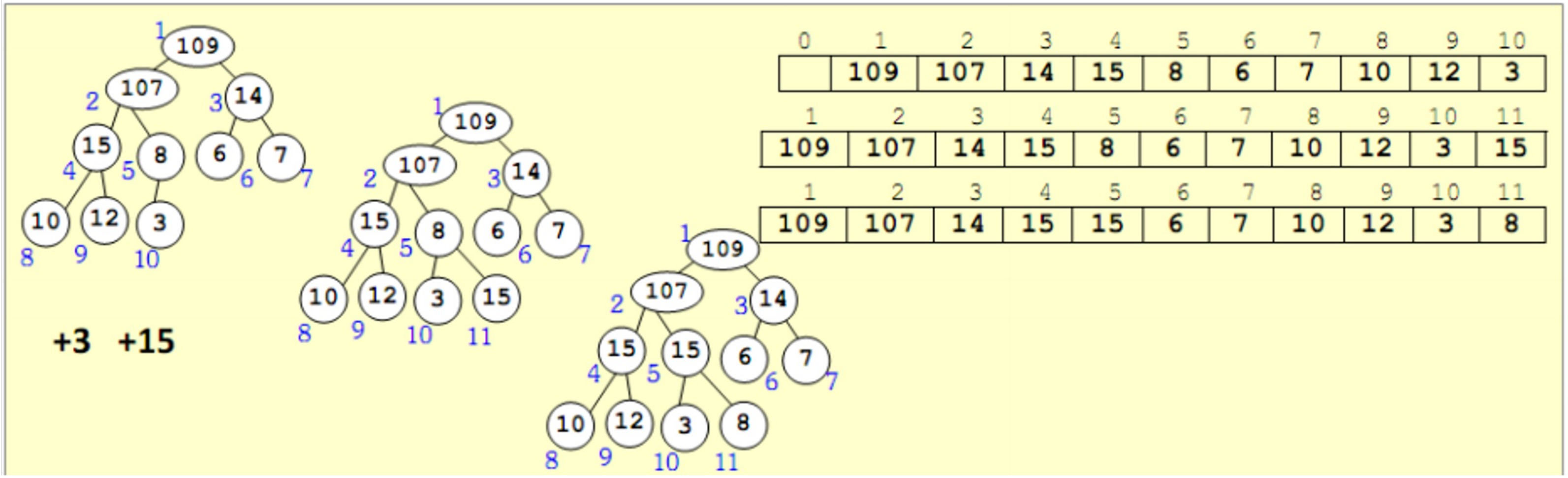
The new node will usually need to be trickled upward until it's below a node with a larger key and above a node with a smaller key

Insertion



Insertion


+107
0 1 2 3 4 5 6 7 8 9
109 15 14 12 8 6 7 10 107
0 1 2 3 4 5 6 7 8 9
109 15 14 107 8 6 7 10 12
0 1 2 3 4 5 6 7 8 9
109 107 14 15 8 6 7 10 12


+3 +15
0 1 2 3 4 5 6 7 8 9 10
109 107 14 15 8 6 7 10 12 3
1 2 3 4 5 6 7 8 9 10 11
109 107 14 15 8 6 7 10 12 3 15
1 2 3 4 5 6 7 8 9 10 11
109 107 14 15 15 6 7 10 12 3 8

Insertion: Java Implementation

```
public boolean insert(int key){
    if(currentSize==maxSize) // if array is full,
        return false; // failure
    Node newNode = new Node(key); // make a new node
    heapArray[currentSize] = newNode; // put it at the end
    trickleUp(currentSize++); // trickle it up
    return true; // success
} // end insert()
```

Insertion: Java Implementation

```
public void trickleUp(int index){
    int parent = (index-1) / 2;
    Node bottom = heapArray[index];
    while( index > 0 && heapArray[parent].getKey() < bottom.getKey() )
    {
        heapArray[index] = heapArray[parent]; // move node down
        index = parent; // move index up
        parent = (parent-1) / 2; // parent <- its parent
    } // end while
    heapArray[index] = bottom;
} // end trickleUp()
```

Introduction to Heaps
Removal

Removal means removing the node with the maximum key. This node is always the root, so removing it is easy. The root is always at index 0 of the heap array:

```
maxNode = heapArray[0];
```

Introduction to Heaps
Removal

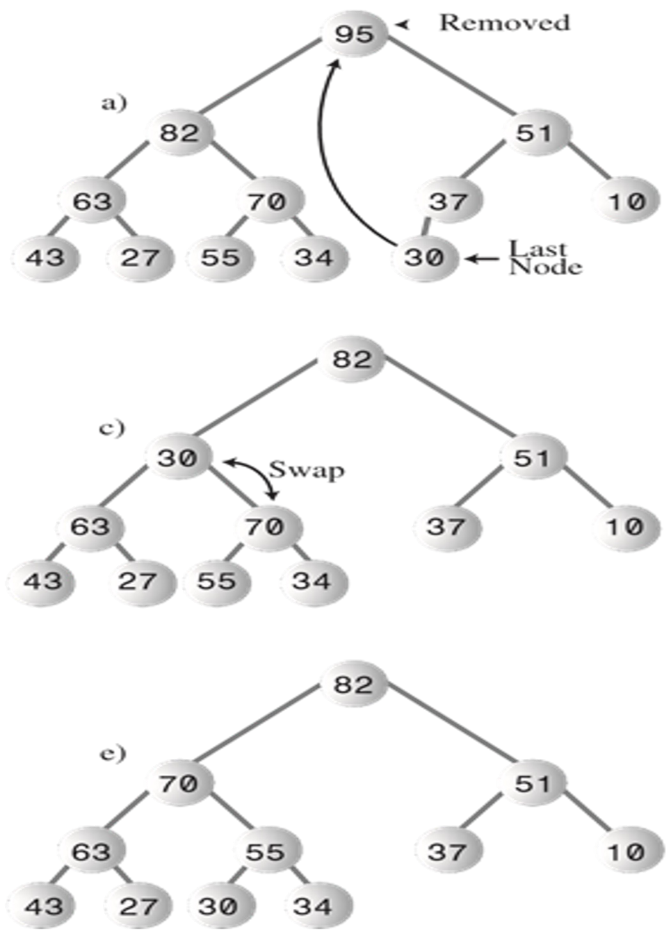
Here are the steps for removing the maximum node:

1. Remove the root.

2. Move the last node into the root.

3. Trickle the last node down until it's below a larger node and above a smaller one.

Introduction to Heaps
Removal

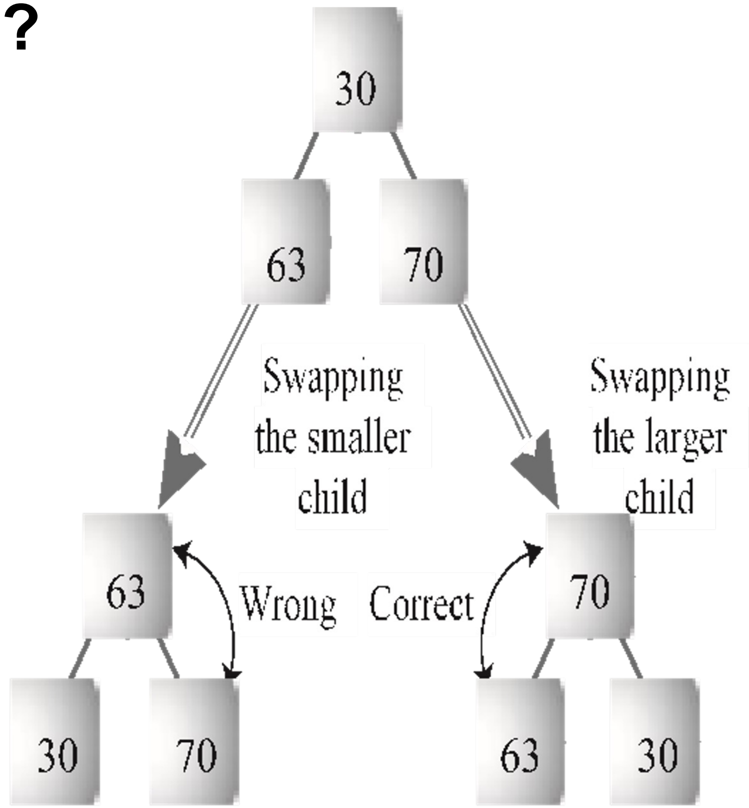


Removal: Java Implementation

```
public Node remove() // delete item with max key
{ // (assumes non-empty list)
        Node root = heapArray[0]; // save the root
        heapArray[0] = heapArray[--currentSize]; // root <- last
        trickleDown(0); // trickle down the root
        return root; // return removed node
} // end remove()
```

Introduction to Heaps Removal

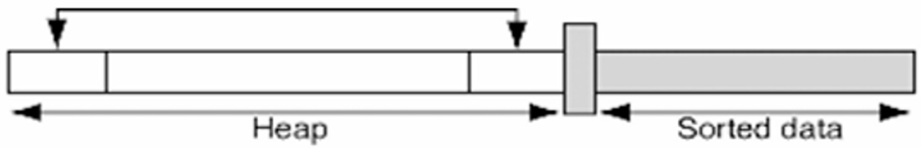
Which child to swap?



Heapsort

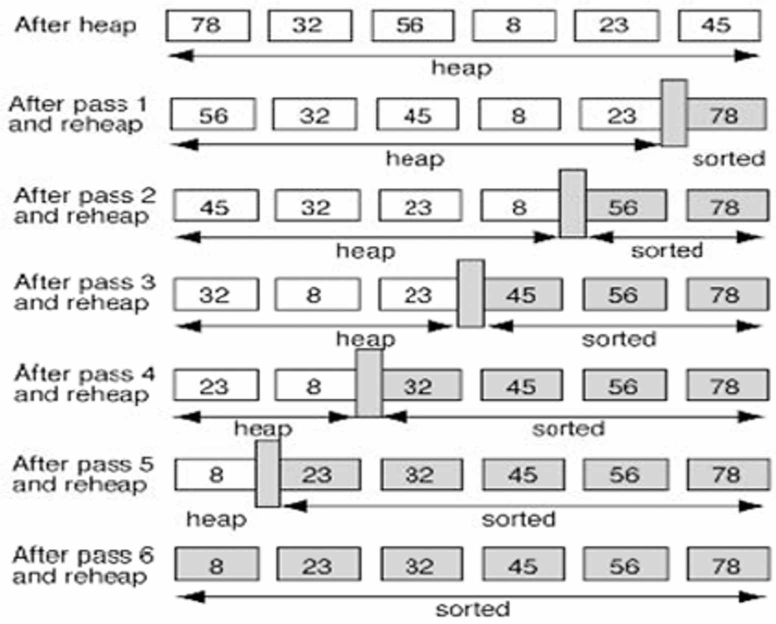
Heapsort Introduction

The concept



1. Use a wall separating between the heap and the sorted list
2. In the heap exchange the first with the last
3. Advance the wall
4. Reconstruct the heap (*reheap down process*)

```
Algorithm heapSort (heap, last)
1  set walker to 1
2  loop (heap built)
   1  reheapUp (heap, walker)
   2  increment walker
3  end loop
   Heap created. Now sort it.
4  set sorted to last
5  loop (until all data sorted)
   1  exchange (heap, 0, sorted)
   2  decrement sorted
   3  reheapDown (heap, 0, sorted)
6  end loop
end heapSort
```



Heapsort Introduction

The basic idea is to insert all the unordered items into a heap using the normal **insert()** routine.

Repeated application of the **remove()** routine will then remove the items in sorted order.

Heapsort runs in **O(N*logN)** time no matter how the data is distributed.

Heapsort
Java Implementation

```
for(int j=0; j<size; j++)
    theHeap.insert( anArray[j] );
    // from unsorted array

for(int j=0; j<size; j++)
    anArray[j] = theHeap.remove();

    // to sorted array
```

Heapsort
Make an Unordered Array into Heap: heapify

The following code fragment applies **trickleDown()** to all nodes, except those on the bottom row, starting at N/2-1 and working back to the root:

```
for(int j=N/2-1; j >=0; j--)
        theHeap.trickleDown(j);
```

Heapsort
Make an unordered array into heap: heapify

A recursive approach can be used to form a heap from an array

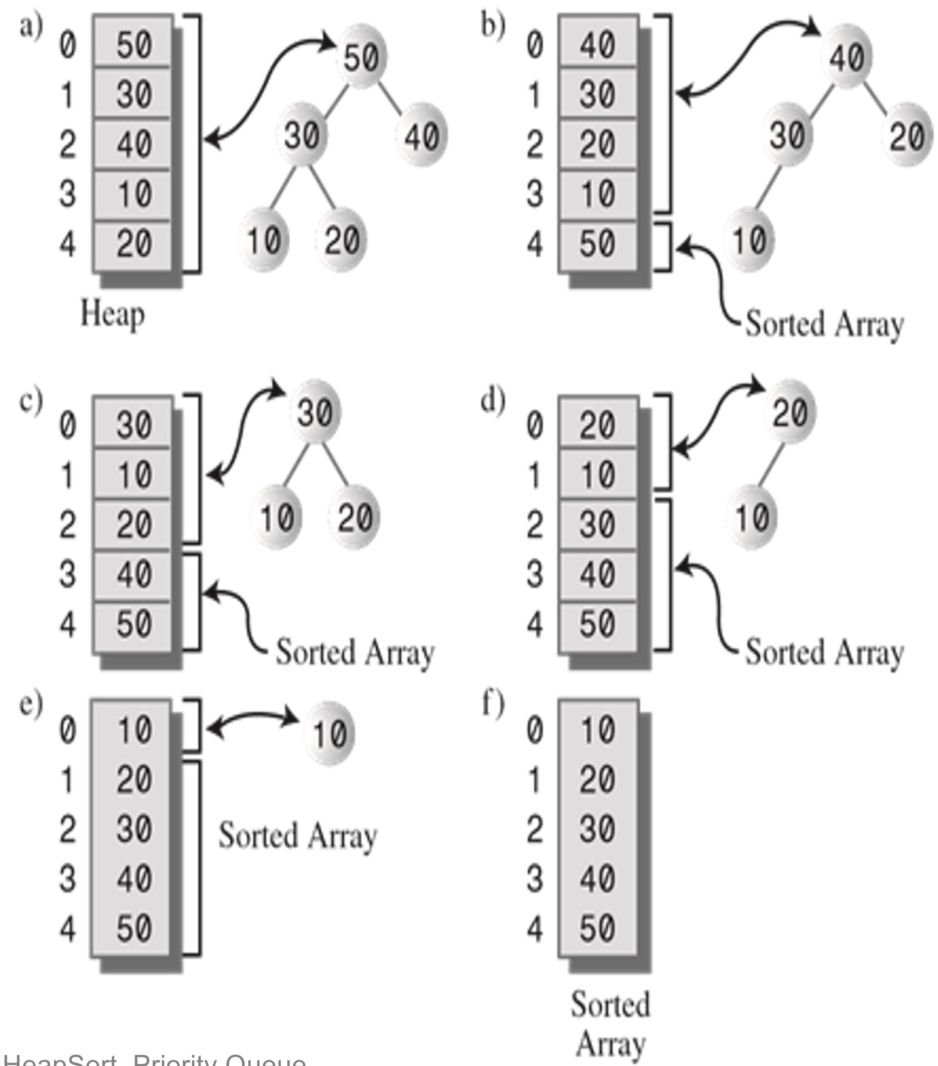
```
heapify(int index) // transform array into heap
{
    if(index > N/2-1) // if node has no children,
    return; // return
    heapify(index*2+2); // turn right subtree into heap
    heapify(index*2+1); // turn left subtree into heap
    trickleDown(index); // apply trickle-down to node
}
```

Heapsort
Using the Same Array

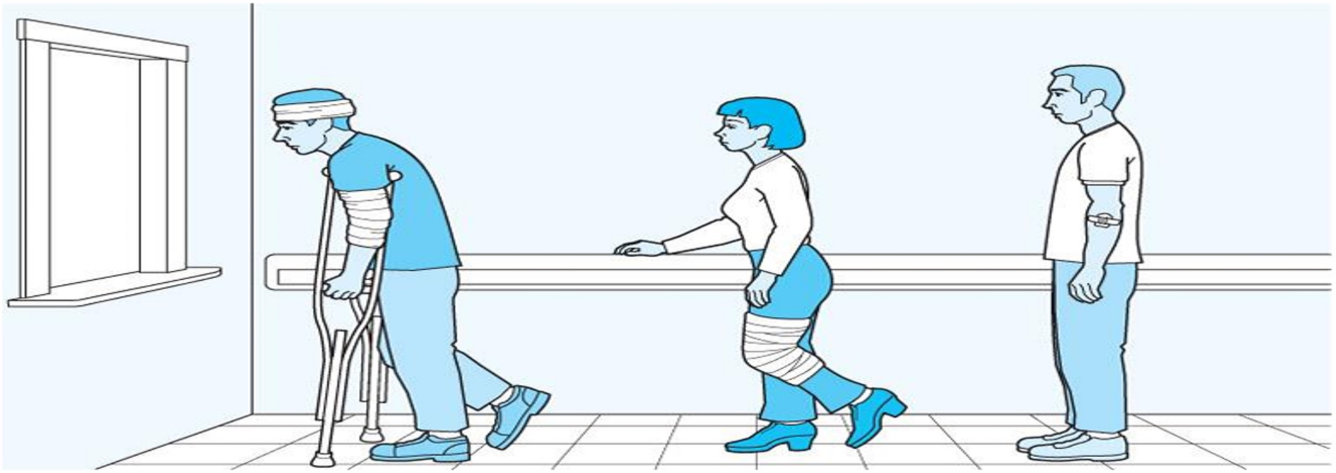
The same array can be used both for the heap and for the initial array. This cuts in half the amount of memory needed for heapsort; no memory beyond the initial array is necessary.

However, the situation becomes more complicated when we apply **remove()** repeatedly to the heap. Where are we going to put the items that are removed?

Heapsort
Using the Same Array



Priority Queue



Priority Queue
An ADT in which only the item with the highest priority can be accessed

There’s a very close relationship between a priority queue and the heap used to implement it. We can replace the usage of a key value of each node in the heapsort array to a priority value of each item in the queue.

Priority Queue Introduction

```
class Heap
{
    private Node heapArray[];
    public void insert(Node nd){ }
    public Node remove(){ }

}
```

```
class priorityQueue
{
    private Heap theHeap;
    public void enqueue(Node nd)
    {
        theHeap.insert(nd);
    }
    public Node dequeue()
    {
         return theHeap.remove()
    }
}
```

References

- [1] Robert Lafore, Data Structures & Algorithms in Java, Second Edition, Copyright © 2003 by Sams Publishing.