Heaps

Heaps

A **Heap** is a complete binary tree with the following (recursive!) definition:

A heap is either empty or whose root:

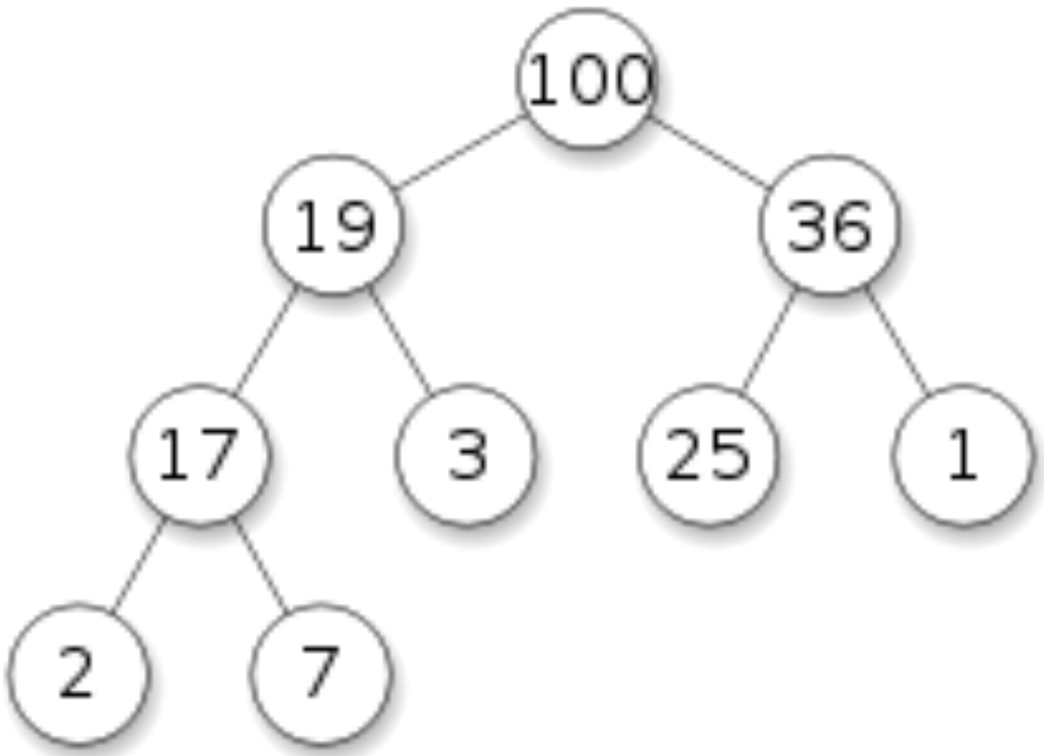
- contains a value greater or equal to that of each of its children.
- has heaps as subtrees.

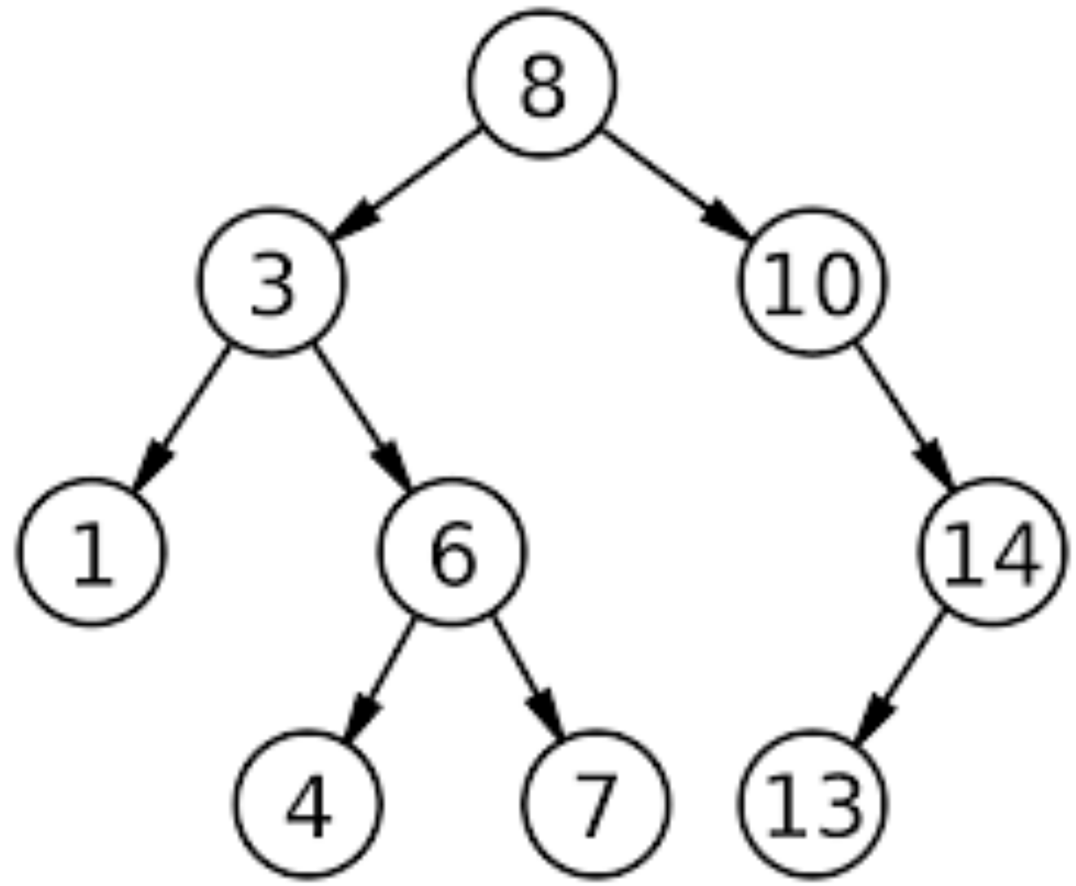
The heap resembles a search tree, but a search tree is really sorted, while the heap is ordered in a weaker sense.
Moreover, a heap is always **complete**.

In a **Maxheap**, the root contains the largest value, in a **Minheap**, the root contains the smallest value.

Example

Which tree is a heap? Which one is a sorted binary tree?





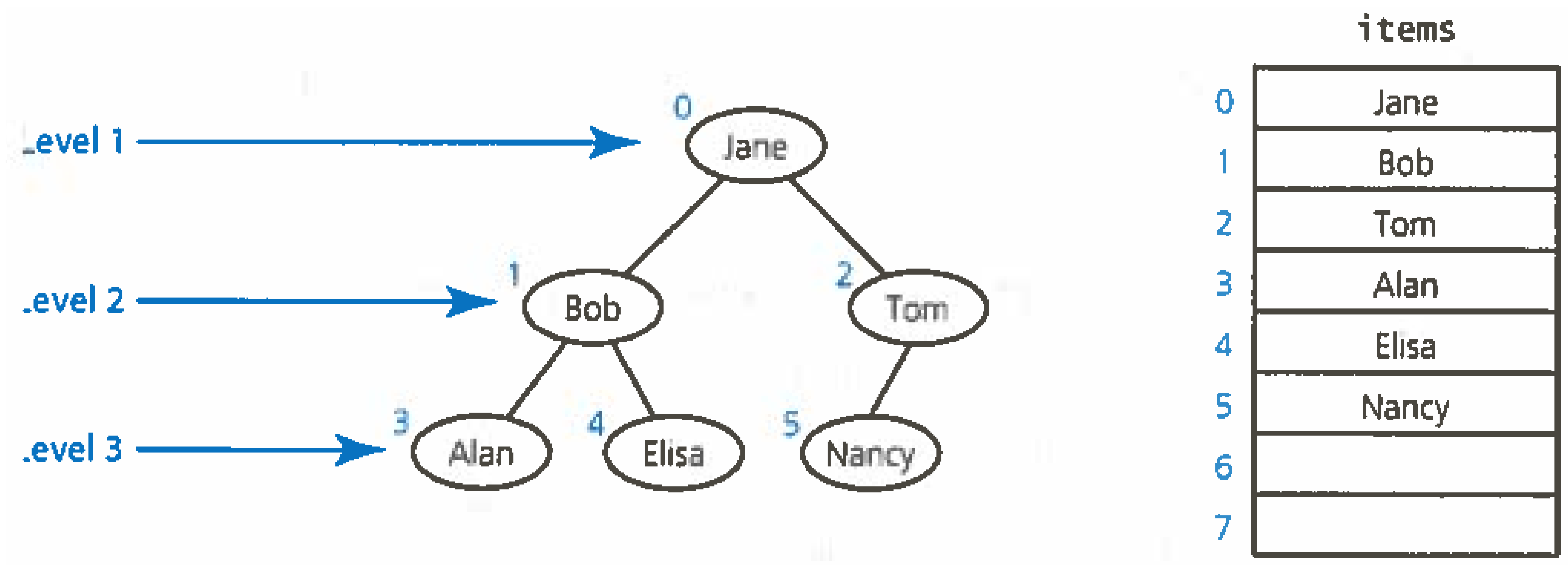
A Heap Interface

```
template<class ItemType>
class HeapInterface
{
public:
   virtual bool isEmpty() const = 0;

   //Returns the number of nodes in the Heap
   virtual int getNumberOfNodes() const = 0;

   // Height of the heap
   virtual int getHeight() const = 0;

   //Returns the data in the root
   virtual ItemType peekTop() const = 0;

   //Adds a new data to the Heap
   virtual bool add(const ItemType& newData) = 0;

   //Remove the root node
   virtual bool remove() = 0;

   //Remove all the nodes
   virtual void clear() = 0;
};
```

Heap Implementation

Since a Heap is a complete binary tree, an array implementation is most of the times the most convenient (at least if you know the maximum number of nodes you'll need).
<u>Question</u>: Why?

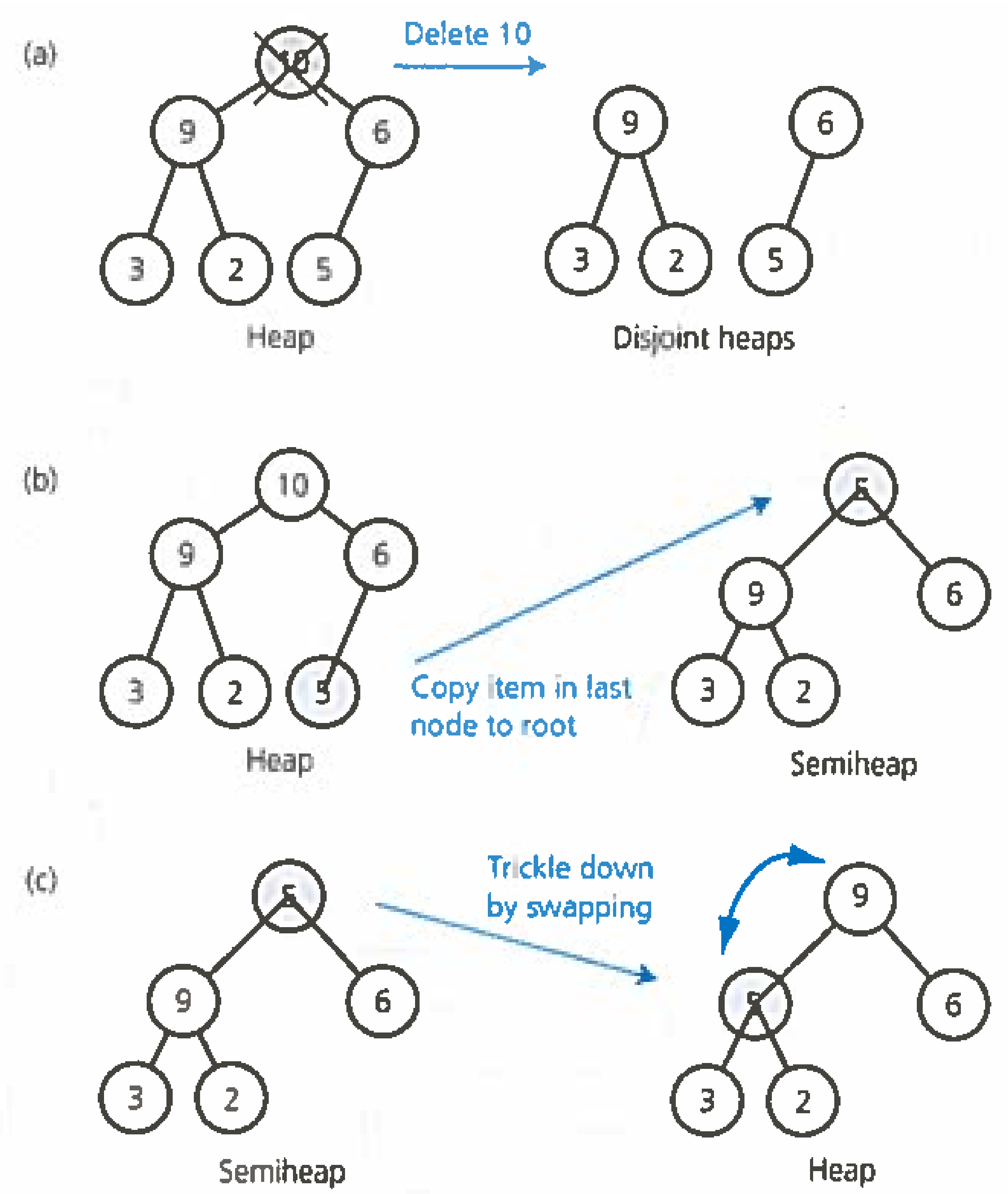
Completeness allows us an efficient implementation:



Consider the element `items[i]`.

The left child is `items[2*i+1]`
The right child is `items[2*i+2]`
The parent is `items[(i-1)/2]`

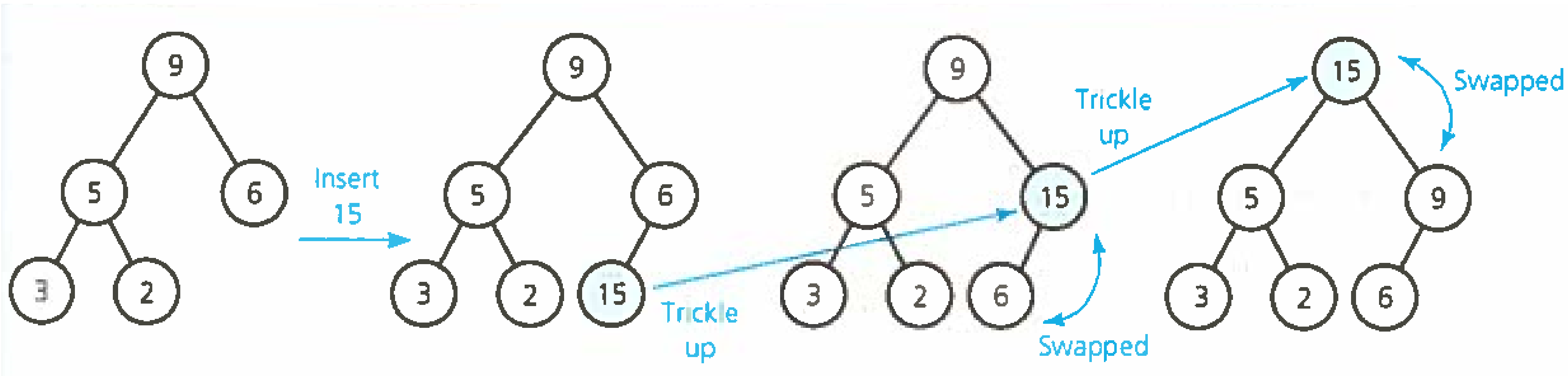
Node removal



Recursive "Trickle down" procedure (for Node Removal)

```
//This is pseudo-code
heapRebuild(int rootIndex , T items, int itemCount){
   if (root is not a leaf){

      largerChildIndex = 2*rootIndex + 1 //left child always exists
      if (right child exists){
         rightChildIndex = 2*rootIndex + 2;
         if (items[rightChildIndex]>items[largerChildIndex])
            largerChildIndex = rightChildIndex;
      }

      if (items[rootIndex]<items[largerChildIndex]){

         swap items[rootIndex] with items[largerChildIndex];

         heapRebuild(largerChildIndex,items,itemCount);
      }
   }
}
```

Node insertion



The idea is the opposite of the removal. This time you insert the new node as a new leaf and then you "trickle up" the node in the correct position.

```
//Pseudo-code
items[itemCount]=newData; //insert at the end of the tree
newDataIndex = itemCount;
while (newDataIndex>=0) && !inPlace{
    parentIndex = (newDataIndex-1)/2;
    if (items[newDataIndex])<=items[parentIndex] inPlace= true;
    else {
        swap items[newDataIndex] with items[parentIndex]
        newDataIndex = parentIndex;
    }
}
itemCount++;
```

Heap Implementation

```
template<class ItemType>
class ArrayMaxHeap : public HeapInterface<ItemType>
{
private:
   static const int ROOT_INDEX = 0;        // Helps with readability
   static const int DEFAULT_CAPACITY = 21; // Small capacity to test for a full heap
   ItemType* items;                        // Array of heap items
   int itemCount;                          // Current count of heap items
   int maxItems;                           // Maximum capacity of the heap

   // Returns the array index of the left child (if it exists).
   int getLeftChildIndex(const int nodeIndex) const;

   // Returns the array index of the right child (if it exists).
   int getRightChildIndex(int nodeIndex) const;

   // Returns the array index of the parent node.
   int getParentIndex(int nodeIndex) const;

   // Tests whether this node is a leaf.
   bool isLeaf(int nodeIndex) const;

   // Converts a semiheap to a heap.
   void heapRebuild(int subTreeRootIndex);

   // Creates a heap from an unordered array.
   void heapCreate();

public:
   ArrayMaxHeap();
   ArrayMaxHeap(const ItemType someArray[], const int arraySize);
   virtual ~ArrayMaxHeap();

   // HeapInterface Public Methods:
   bool isEmpty() const;
   int getNumberOfNodes() const;
   int getHeight() const;
   ItemType peekTop() const throw(PrecondViolatedExcep);
   bool add(const ItemType& newData);
   bool remove();
   void clear();
}; // end ArrayMaxHeap
```

Heap as Priority Queue

We have already seen the implementation of a PQ as a sorted list or array.

It is easy to realize that <u>the structure of a Heap is exactly the same as the one of a priority queue</u>. The root of the heap acts as the front of the queue.

A PQ can be therefore realized inheriting the heap implementation we discussed.

Advantages of the Heap implementation:

- The heap is balanced by definition (it is a complete tree)
- A sorted tree can be used instead of an heap, but it can become unbalanced,
  degrading performance.
- Heap operations are easier than the ones required by a self-balancing tree.

Heap Sort

Problem: sort an array.

Idea: convert the array into a heap and then keep peeking the root content and the remove the root itself.
The heapRebuild routine (see before) will then restructure the heap before the next removal.

Second idea:
Use the same array to store the heap and the sorted array.

Complexity:
$O(N \log_2 N)$ in both average and worst cases!
Remember that MergeSort has the same properties but it needs an additional array.
Quicksort has a $O(N^2)$ in the worst case.

Questions:
1) What is the worst case for QuickSort?
2) Can you see why it is not the case for HeapSort?