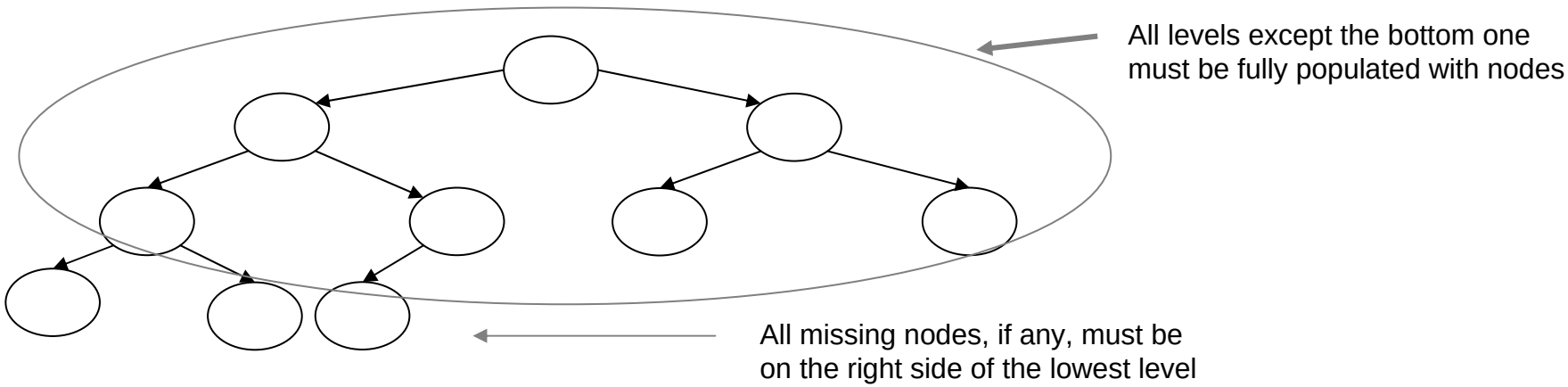
Binary Heaps

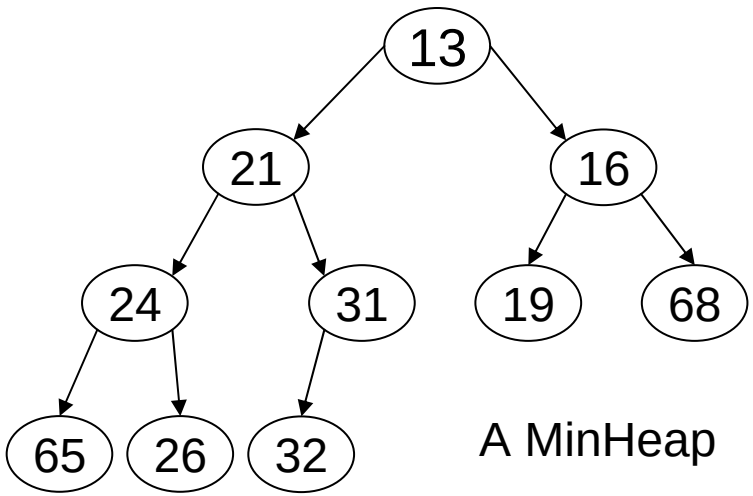
- What is a Binary Heap?
- Array representation of a Binary Heap
- MinHeap implementation
- Operations on Binary Heaps:
    - enqueue
    - dequeue
    - deleting an arbitrary key
    - changing the priority of a key
- Building a binary heap
    - top down approach
    - bottom up approach
- Heap Applications:
    - Heap Sort
    - Heap as a priority queue

What is a Binary Heap?

- A **binary heap** is a *complete binary tree* with one (or both) of the following heap order properties:
  - **MinHeap property:** Each node must have a key that is less or equal to the key of each of its children.
  - **MaxHeap property:** Each node must have a key that is greater or equal to the key of each of its children.
- A binary heap satisfying the MinHeap property is called a MinHeap.
- A binary heap satisfying the MaxHeap property is called a MaxHeap.
- A binary heap with all keys equal is both a MinHeap and a MaxHeap.
- Recall: A complete binary tree may have missing nodes only on the right side of the lowest level.

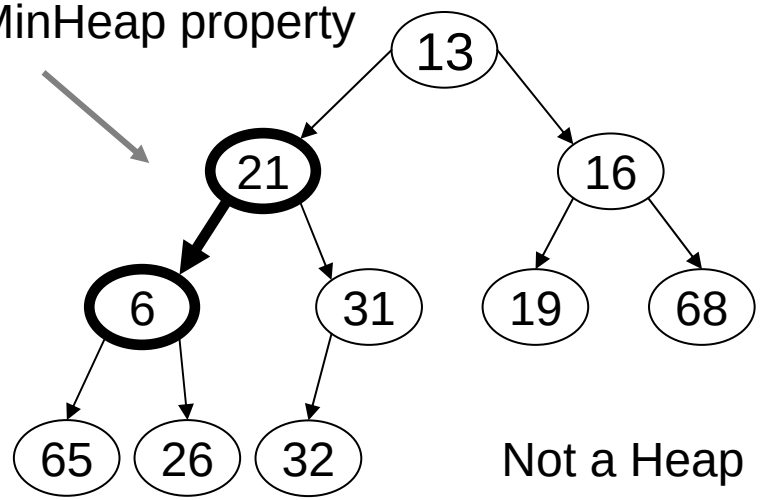


MinHeap and non-MinHeap examples

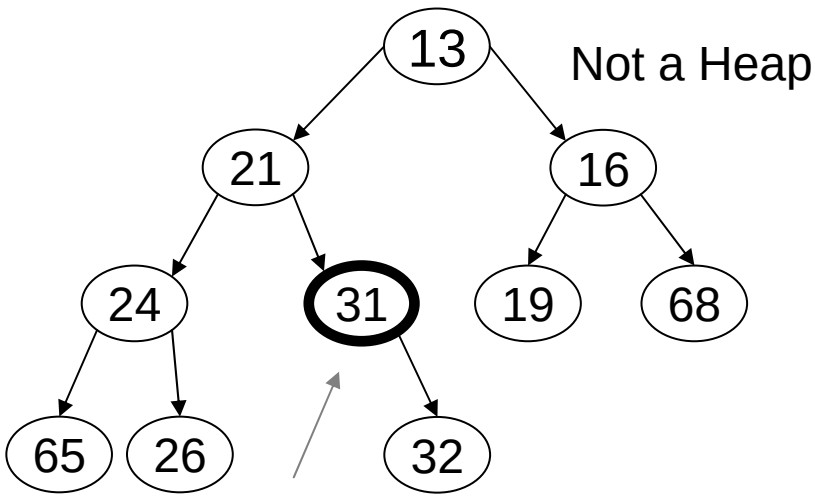


A MinHeap

Violates MinHeap property
21>6

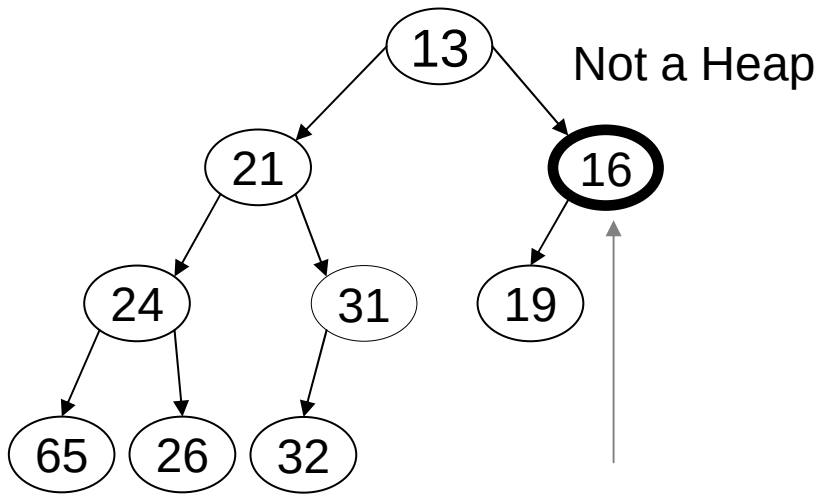


Not a Heap



Not a Heap

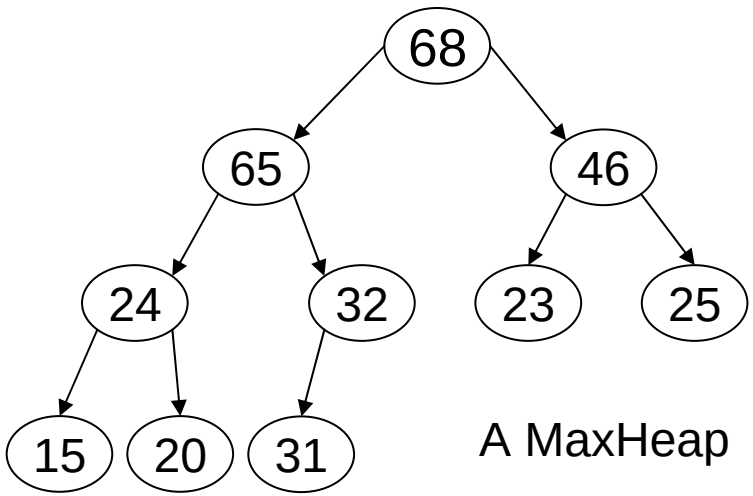
Violates heap structural property



Not a Heap

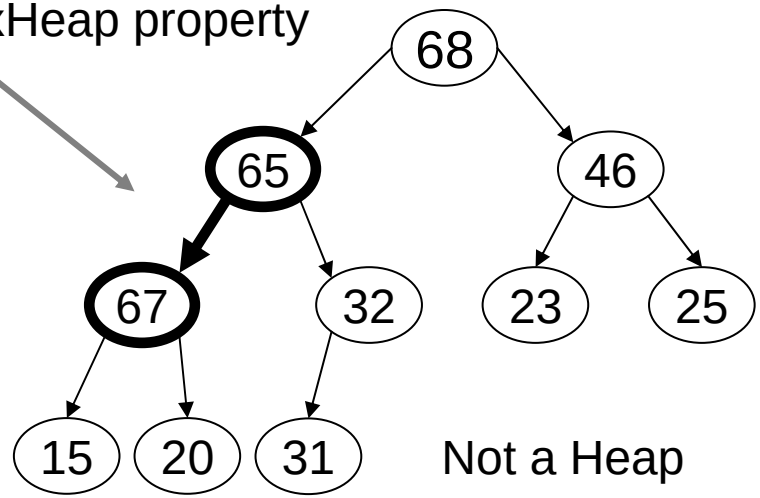
Violates heap structural property

MaxHeap and non-MaxHeap examples

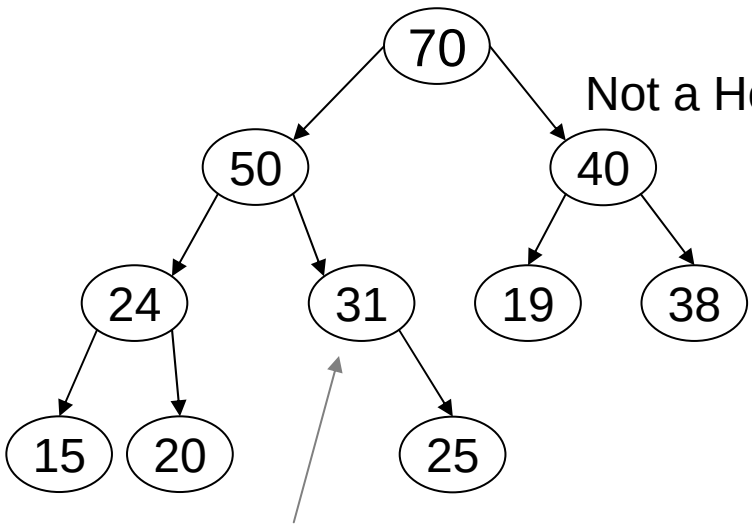


A MaxHeap

Violates MaxHeap property
65 < 67



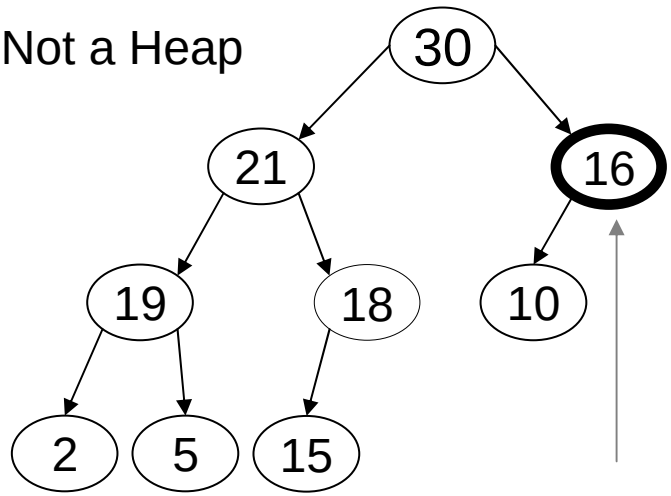
Not a Heap



Not a Heap

Violates heap structural property

Not a Heap

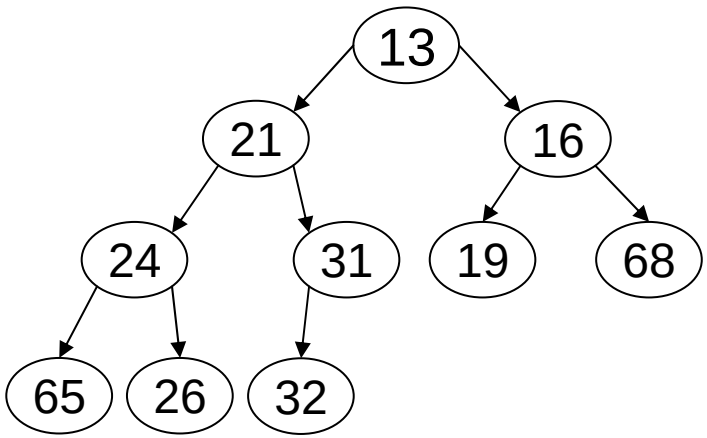


Violates heap structural property

Array Representation of a Binary Heap

- A heap is a dynamic data structure that is represented and manipulated more efficiently using an array.
- Since a heap is a complete binary tree, its node values can be stored in an array, without any gaps, in a breadth-first order, where:

Value(node $_{i+1}$) ⟶ array[i], for i ≥ 0



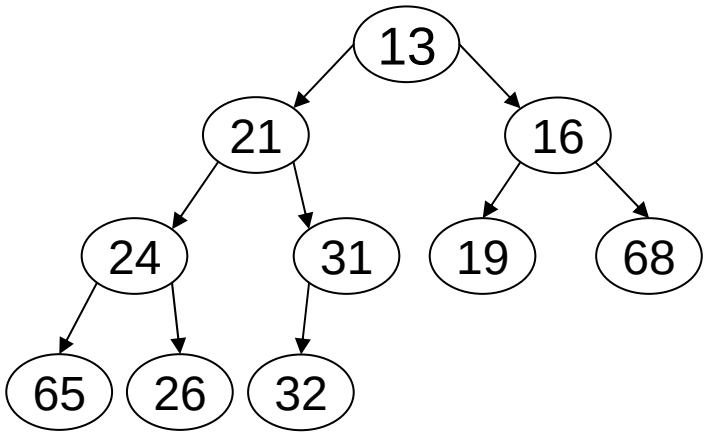
13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9

- The root is array[0]
- The parent of array[i] is array[(i – 1)/2], where i > 0
- The left child, if any, of array[i] is array[2i+1].
- The right child, if any, of array[i] is array[2i+2].

Array Representation of a Binary Heap (contd.)

- We shall use an implementation in which the heap elements are stored in an array starting at index 1.

Value(node $_i$) ⟶ array[i] , for i $\geq$ 1



	13	21	16	24	31	19	68	65	26	32
0	1	2	3	4	5	6	7	8	9	10

- The root is array[1].
- The parent of array[i] is array[i/2], where i > 1
- The left child, if any, of array[i] is array[2i].
- The right child, if any, of array[i] is array[2i+1].

MinHeap Implementation

- A binary heap can serve as a priority queue
- Our MinHeap class will implement the following PriorityQueue interface

```
public interface PriorityQueue extends Container{

    public abstract void enqueue(Comparable comparable);

    public abstract Comparable findMin();

    public abstract Comparable dequeueMin();
}
```

MinHeap Implementation (contd.)

```
public class BinaryHeap extends AbstractContainer
                        implements PriorityQueue {

   protected Comparable array[];

   public BinaryHeap(int i){
        array = new Comparable[i + 1];
    }

   public BinaryHeap(Comparable[] comparable) {
      this(comparable.length);
      for(int i = 0; i < comparable.length; i++)
         array[i + 1] = comparable[i];
      count = comparable.length;

      buildHeapBottomUp();
   }
```

MinHeap enqueue

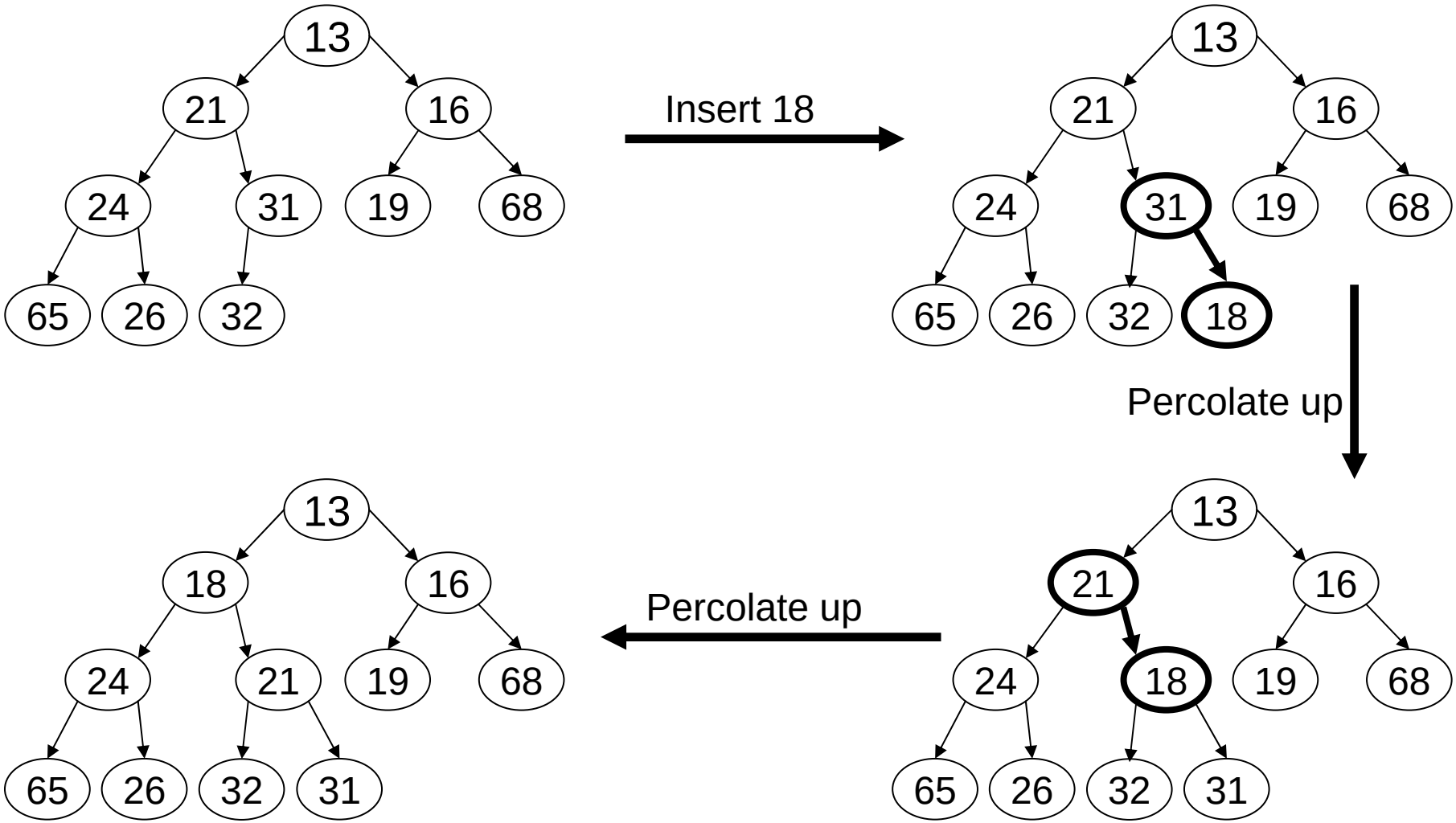
- The pseudo code algorithm for enqueing a key in a MinHeap is:

```
1  enqueue(e1)
2  {
3       if(the heap is full) throw an exception ;
4       insert e1 at the end of the heap ;
5       while(e1 is not in the root node and e1 < parent(e1))
6                 swap(e1 , parent(e1)) ;
7  }
```

- The process of swapping an element with its parent, in order to restore the heap order property is called percolate up, sift up, or reheapification upward.

- Thus, the steps for enqueue are:
  1. Enqueue the key at the end of the heap.
  2. As long as the heap order property is violated, percolate up.

MinHeap Insertion Example



MinHeap enqueue implementation

- To have better efficiency, we avoid repeated swapping
- We find a place (hole) for the new key, move the hole upward when needed, and at the end, put the key into the hole

```
public void enqueue(Comparable comparable){
   if(isFull()) throw new ContainerFullException();

   int hole = ++count;

   // percolate up via a hole
   while(hole > 1 &&
                  array[hole / 2].compareTo(comparable)>0){
      array[hole] = array[hole / 2];
      hole = hole / 2 ;
   }
   array[hole] = comparable;
}

public boolean isFull(){
   return count == array.length - 1;
}
```

MinHeap dequeue

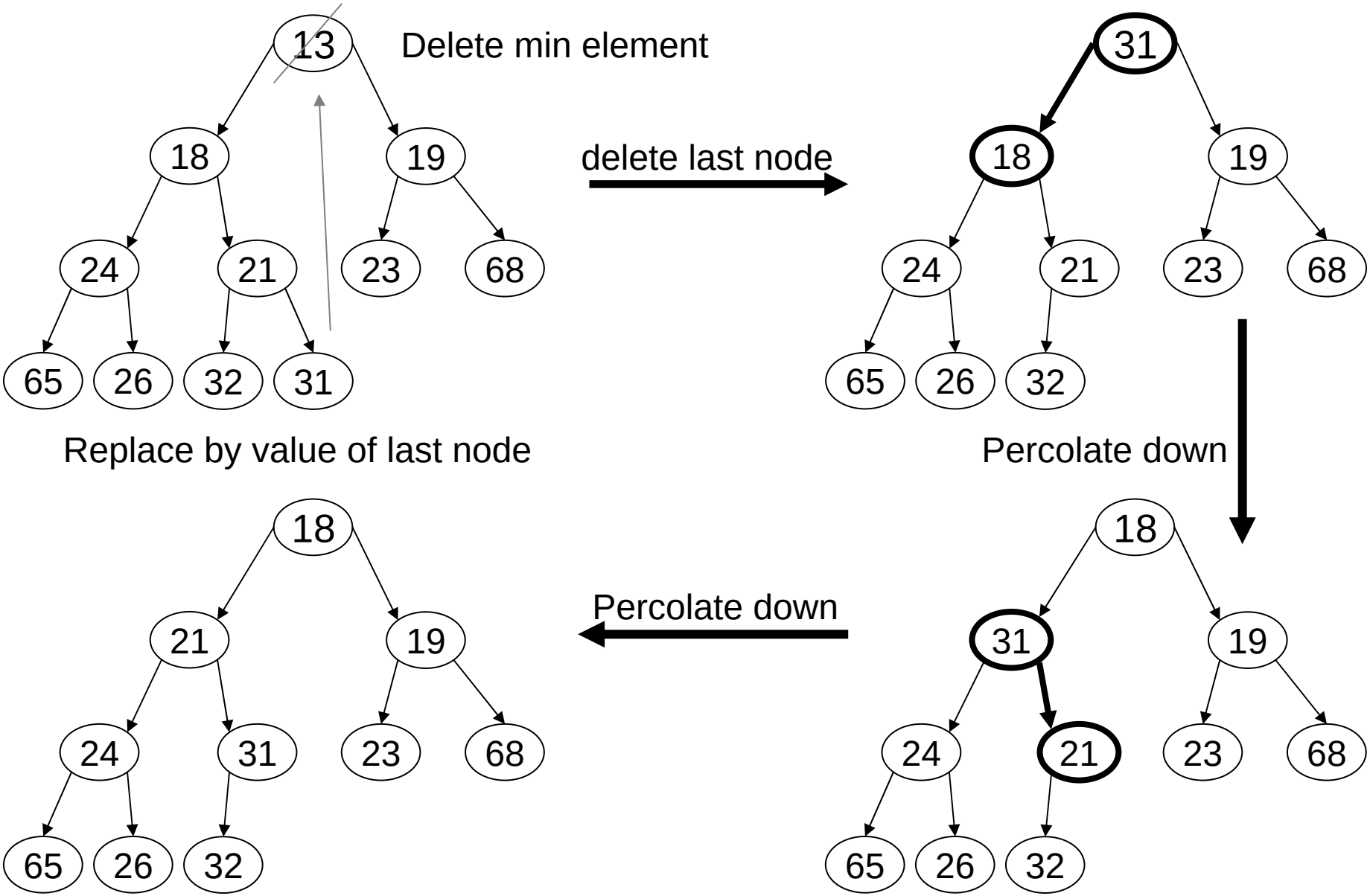
- The pseudo code algorithm for deleting the root key in a MinHeap is:

```
1     dequeueMin(){
2          if(Heap is empty) throw an exception ;
3          extract the element from the root ;
4          if(root is a leaf node){ delete root ; return; }
5          copy the element from the last leaf to the root ;
6          delete last leaf ;
7          p = root ;
8          while(p is not a leaf node and p > any of its children)
9                 swap p with the smaller child ;
10        return ;
11    }
```

- The process of swapping an element with its child, in order to restore the heap order property is called percolate down, sift down, or reheapification downward.

- Thus, the steps for deletion are:
  1. Replace the key at the root by the key of the last leaf node.
  2. Delete the last leaf node.
  3. As long as the heap order property is violated, percolate down.

MinHeap Dequeue Example



MinHeap dequeue Implementation

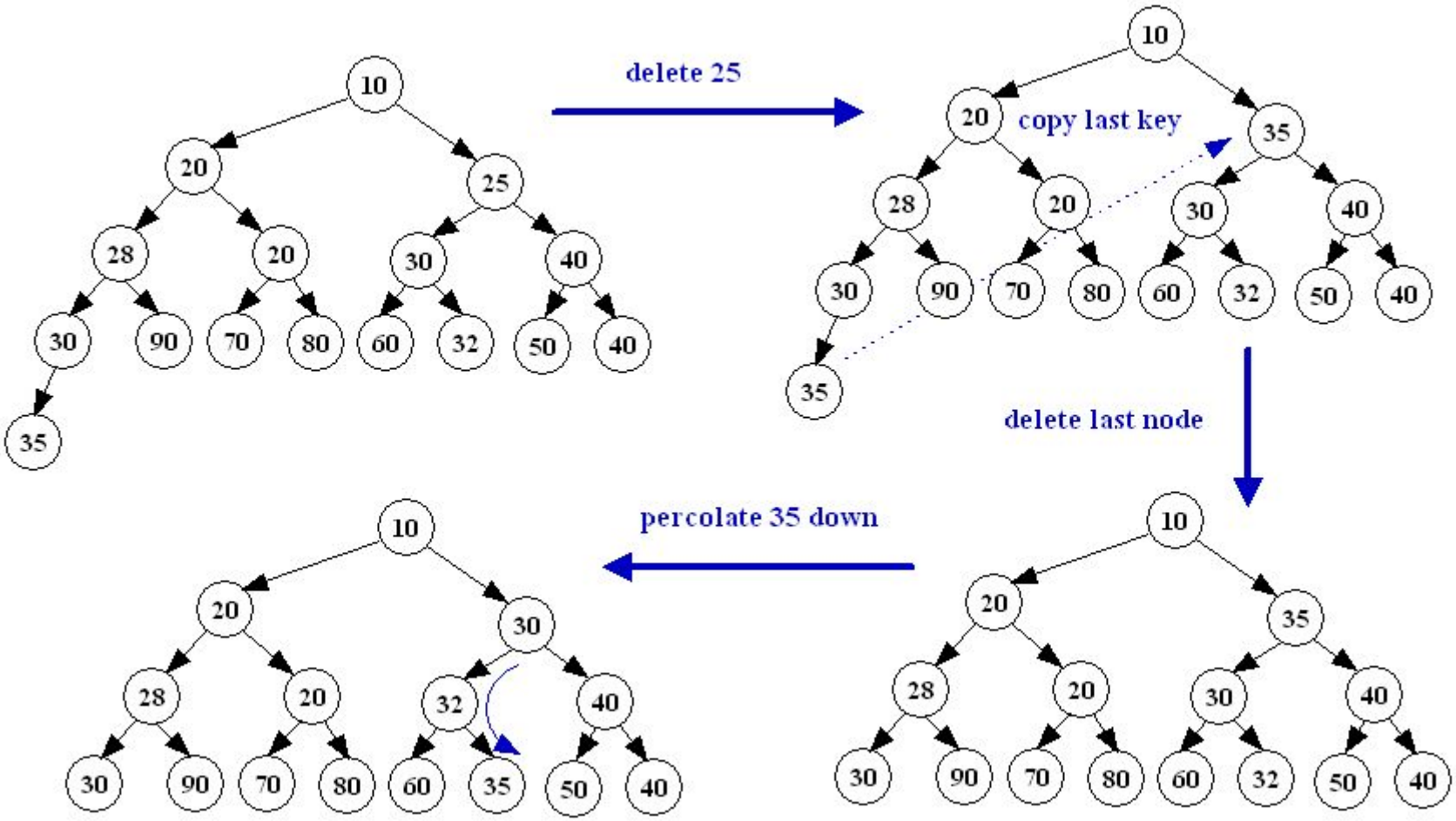
```
public Comparable dequeueMin(){
   if(isEmpty()) throw new ContainerEmptyException();
   Comparable minItem = array[1];
   array[1] = array[count];
   count--;
   percolateDown(1);
   return minItem;
}

private void percolateDown(int hole){
   int minChildIndex;
   Comparable temp = array[hole];
   while(hole * 2 <= count){
      minChildIndex = hole * 2;
      if(minChildIndex + 1 <= count && array[minChildIndex + 1].
                                 compareTo(array[minChildIndex])<0)
         minChildIndex++;
      if(array[minChildIndex].compareTo(temp)<0){
         array[hole] = array[minChildIndex];
         hole = minChildIndex;
      } else
         break;
   }
   array[hole] = temp;
}
```

Deleting an arbitrary key

The algorithm of deleting an arbitrary key from a heap is:

- Copy the key **x** of the last node to the node containing the deleted key.
- Delete the last node.
- Percolate **x** down until the heap property is restored.

Example:

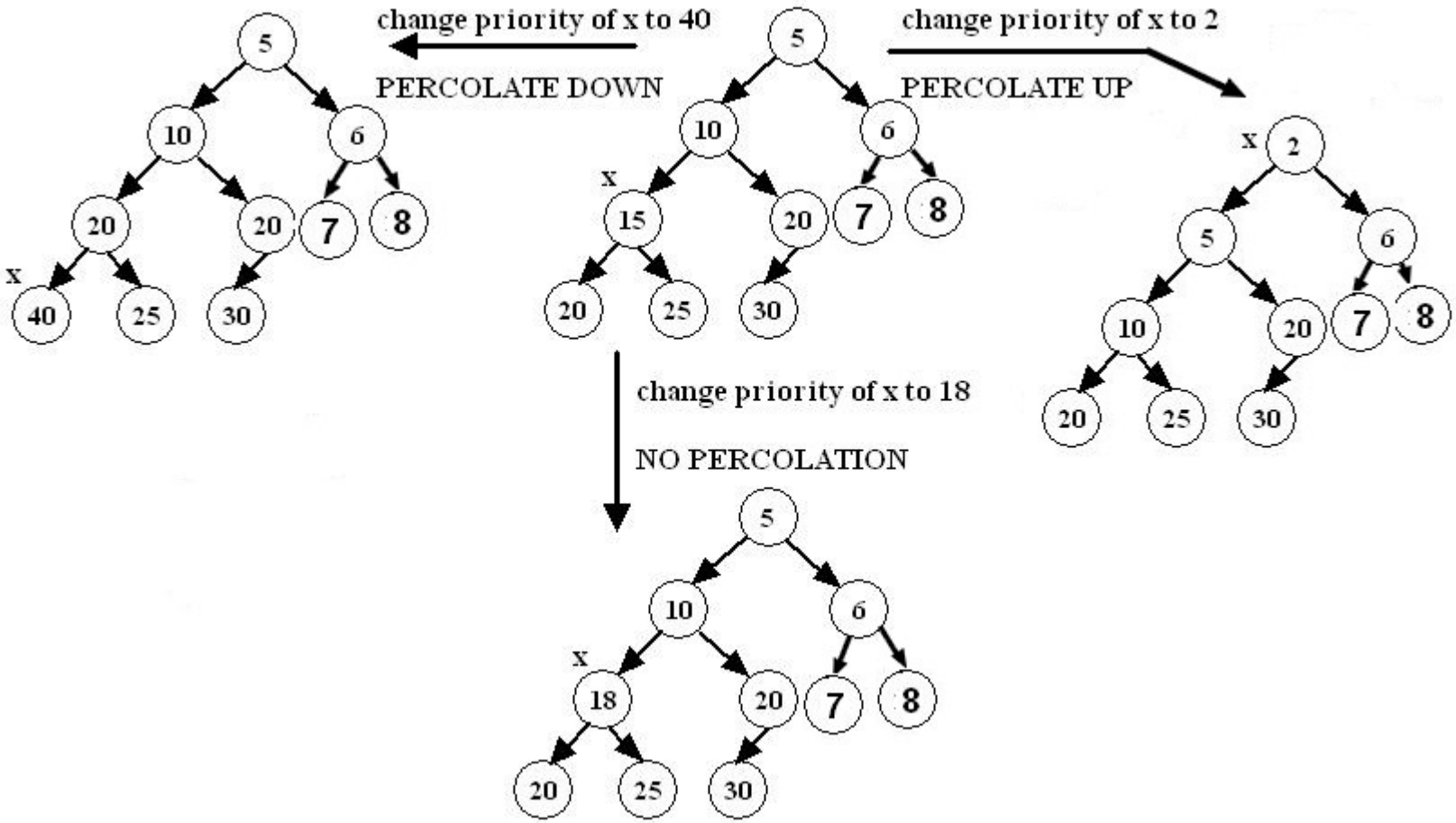


Changing the priority of a key

There are three possibilities when the priority of a key **x** is changed:

1. The heap property is not violated.
2. The heap property is violated and **x** has to be percolated up to restore the heap property.
3. The heap property is violated and **x** has to be percolated down to restore the heap property.

Example:



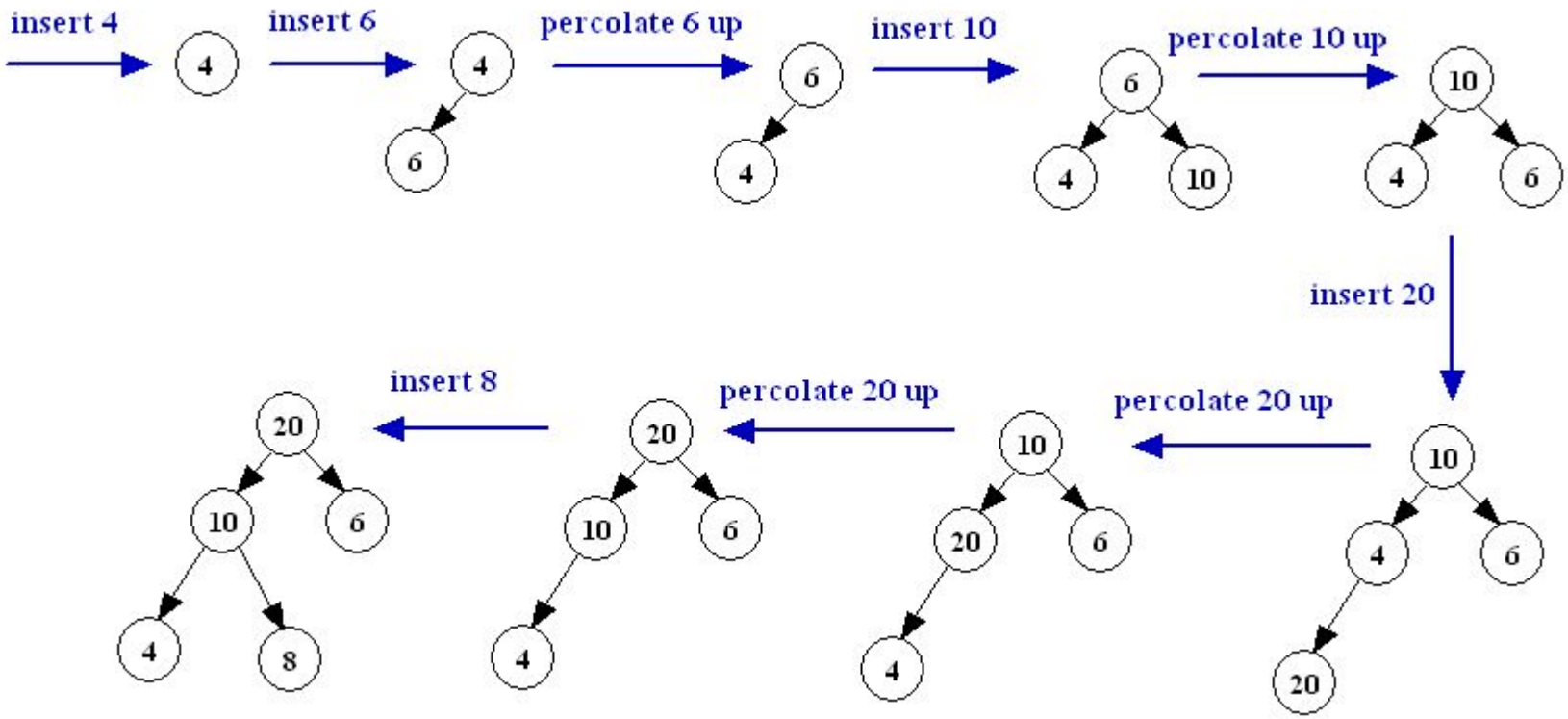
Building a heap (top down)

- A heap is built top-down by inserting one key at a time in an initially empty heap.
- After each key insertion, if the heap property is violated, it is restored by percolating the inserted key upward.

The algorithm is:

```
for(int i=1; i <= heapSize; i++){
   read  key;
   binaryHeap.enqueue(key);
}
```

Example: Insert the keys 4, 6, 10, 20, and 8 in this order in an originally empty max-heap



Converting an array into a Binary heap (Building a heap bottom-up)

- The algorithm to convert an array into a binary heap is:

1. Start at the level containing the last non-leaf node (i.e., array[n/2], where n is the array size).
2. Make the subtree rooted at the last non-leaf node into a heap by invoking percolateDown.
3. Move in the current level from right to left, making each subtree, rooted at each encountered node, into a heap by invoking percolateDown.
4. If the levels are not finished, move to a lower level then go to step 3.

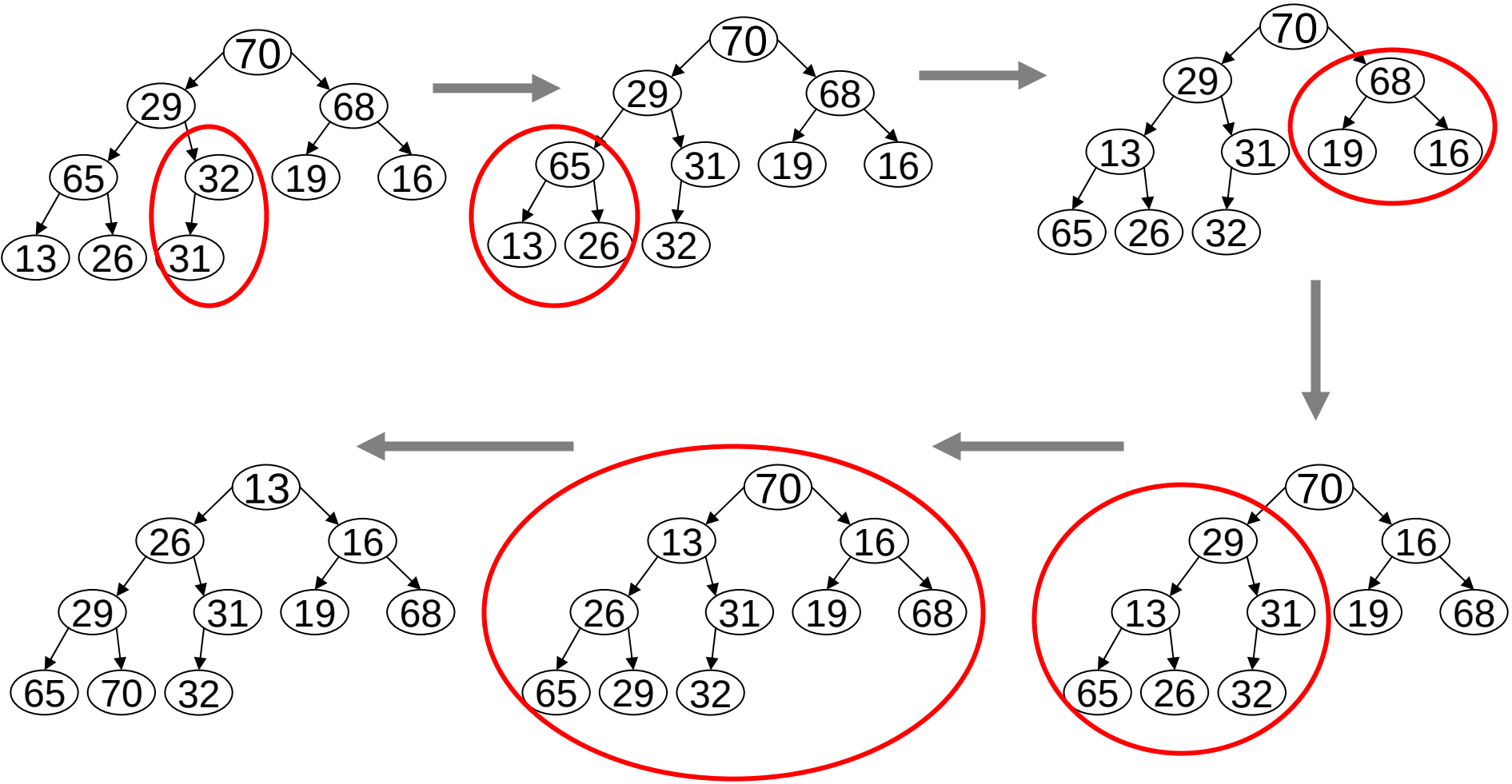
- The above algorithm can be refined to the following method of the BinaryHeap class:

```
private void buildHeapBottomUp()
{
 for(int i = count / 2; i >= 1; i--)
            percolateDown(i);
}
```

Converting an array into a MinHeap (Example)

70	29	68	65	32	19	16	13	26	31

At each stage convert the highlighted tree into a MinHeap by percolating down starting at the root of the highlighted tree.



Heap Application: Heap Sort

- A MinHeap or a MaxHeap can be used to implement an efficient sorting algorithm called Heap Sort.
- The following algorithm uses a MinHeap:

```
public  static void  heapSort(Comparable[] array){
   BinaryHeap  heap  =  new BinaryHeap(array) ;
   for(int i = 0; i < array.length; i++)
      array[i] = heap.dequeueMin() ;
}
```

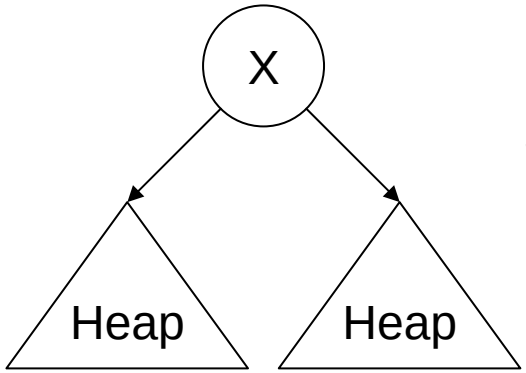
- Because the dequeueMin algorithm is O(log n), heapSort is an O(n log n) algorithm.
- Apart from needing the extra storage for the heap, heapSort is among efficient sorting algorithms.

Heap Applications: Priority Queue

- A heap can be used as the underlying implementation of a priority queue.
- A priority queue is a data structure in which the items to be inserted have associated priorities.
- Items are withdrawn from a priority queue in order of their priorities, starting with the highest priority item first.
- Priority queues are often used in resource management, simulations, and in the implementation of some algorithms (e.g., some graph algorithms, some backtracking algorithms).
- Several data structures can be used to implement priority queues. Below is a comparison of some:

Data structure	Enqueue	Find Min	Dequeue Min
Unsorted List	O(1)	O(n)	O(n)
Sorted List	O(n)	O(1)	O(1)
AVL Tree	O(log n)	O(log n)	O(log n)
MinHeap	O(log n)	O(1)	O(log n)

Priority Queue (Contd.)



X is the element with highest priority

```
1  priorityQueueEnque(e1)
2  {
3       if(priorityQueue is full) throw an exception;
4       insert e1 at the end of the priorityQueue;
5       while(e1 is not in the root node and e1 < parent(e1))
6                swap(e1 , parent(e1));
7   }
```

```
1     priorityQueueDequeue(){
2         if(priorityQueue is empty) throw an exception;
3         extract the highest priority element from the root;
4         if(root is a leaf node){ delete root ; return; }
5         copy the element from the last leaf to the root;
6         delete last leaf;
7         p = root;
8         while(p is not a leaf node and p > any of its children)
9              swap p with the smaller child;
10        return;
11    }
```