## CMP-5014Y Data Structures and Algorithms

Geoff McKeown

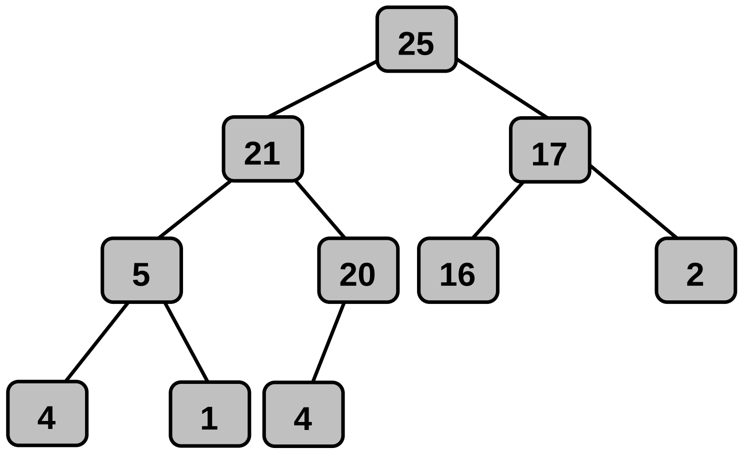
# *Heaps*

## *A Structure for Implementing Priority Queues*

### Lecture Objectives

- ◇ To introduce a Heap abstract data type.
- ◇ To discuss the ideas underlying an array implementation.
- ◇ To discuss complexity issues.
- ◇ To present a Java implementation.
- ◇ To illustrate with diagrams the basic operations of `siftUp` and `siftDown`.
- ◇ To introduce HeapSort and to discuss its complexity.

***Example***




Informally:

- a heap is a ***complete binary tree***:
  - all levels, except possibly the last, have a full complement of nodes;
  - nodes on the final level are filled in from left to right;
- value at each node in a (***max***) heap is at *least as large as the values at its children nodes*.

**Definition** A non-empty binary tree is said to be ***complete*** if it satisfies the following property: the $n \geq 1$ nodes can be numbered such that

- node 1 is the root node,
- parent of node $i =$ node $\lfloor i/2 \rfloor, i = 2, 3, \ldots, n$.

**Definition** (***max heap***) HEAP($X$)
Let $X$ be a totally ordered set. A heap on $X$ is either empty, $\emptyset$, or it is a complete binary tree, $t$, comprising $n_t \geq 1$ nodes to each node of which a value of $X$ is assigned such that:
value of node $i \leq$ value of parent of node $i$, $i = 2, 3, \ldots, n_t$.

The ***size*** of a heap is the number of nodes in the tree. A heap is empty if and only if its size is 0.

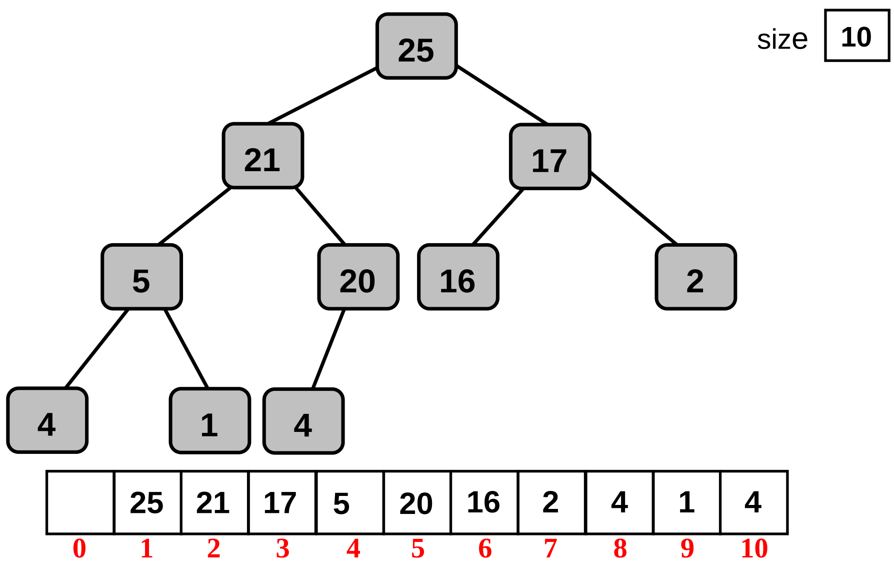
## Java Interface

```
package binarytree;

public interface ADT_HEAP<Comparable>
{
    public boolean isEmptyHeap( );

    public int getSize( );

    public Heap copyHeap( );

    public void deleteMax( );

    public Comparable findMax( );

    public void insert( Comparable item );
}
```

## Array implementation

A heap can be stored in an array, `heapArray`, as follows:

- the root is stored in `heapArray[1]`;
- if the value of a node, $s$, is stored in `heapArray[i]`, then the value of the leftchild of $s$ is stored in `heapArray[2*i]` and the value of the rightchild of $s$ is stored in `heapArray[2*i+1]`.



## Implementation of ADT_Heap<Comparable>

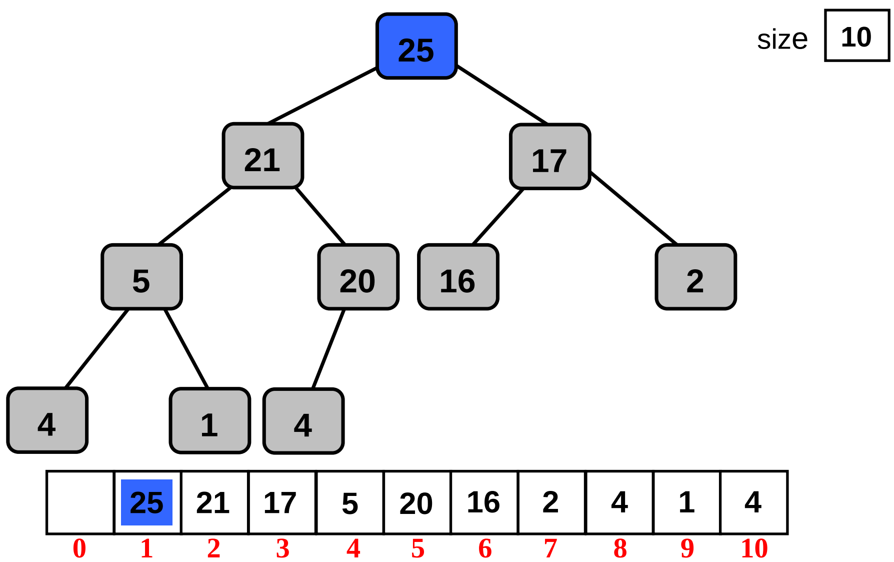
```
package Heaps;
public class Heap implements ADT_Heap<Comparable>
{
// Array implementation
    static final int MAX_SIZE = 24;
        // default maximium size of heap
    private int size;
    private int maxSize;
    private Comparable [] heapArray;
    // Constructor which creates a heap of a given initial
    // maximum capacity
    public Heap( int sz )
    {
        size = 0;
        maxSize = sz;
        heapArray = new Comparable [ maxSize + 1 ];
            // Location 0 of heapArray is not used
    }
    // Default constructor
    public Heap( )
        { this( MAX_SIZE ); }
```

```java
// Construct a heap from a given array
// of Comparable items

public Heap( Comparable [] a )
{
    this( a.length );

    size = a.length;

    for ( int i = 0; i < size; i++ )

        heapArray[ i+1 ] = a[ i ];

    // simple but inefficient heapifier:

    for ( int i = 2; i <= size; i++ )

        siftUp( heapArray, i, i );
}
```

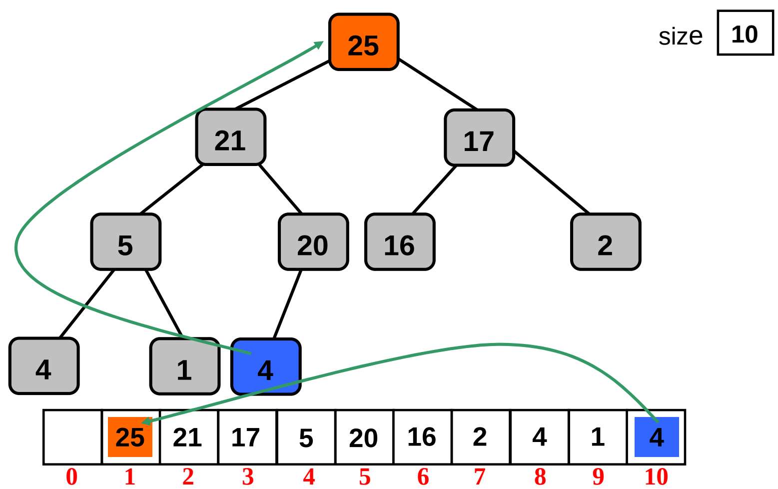
```java
public boolean isEmptyHeap( )
    { return size == 0; }

public int getSize( )
    { return size; }
```

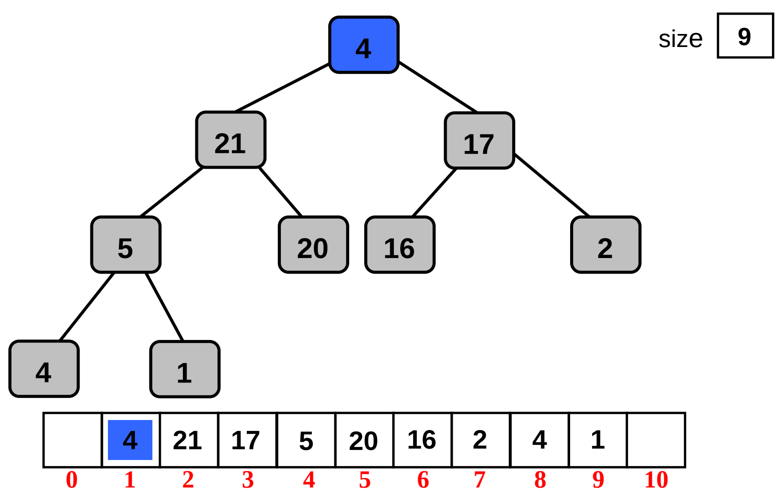
```java
// To return a copy of this heap
public Heap copyHeap()
{
    Heap h = new Heap( this.maxSize );
    h.size = this.size;
    h.heapArray = new Comparable [maxSize + 1];
    for ( int i =1; i <= size; i++ )
        h.heapArray[i] = this.heapArray[i];
}
```

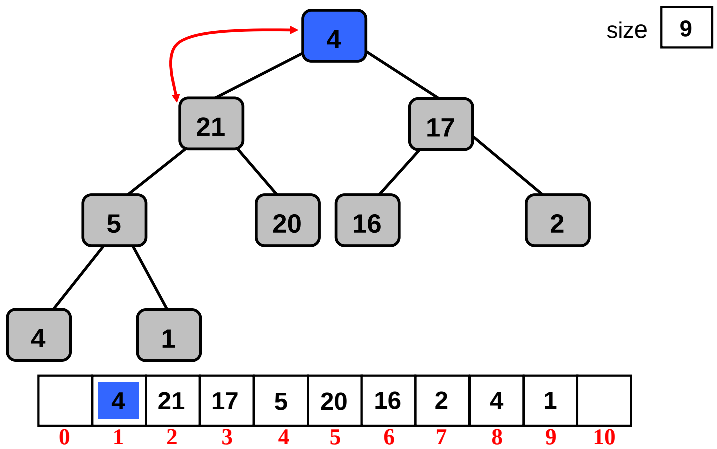
```java
public Comparable findMax( )
    { return heapArray[1]; }
```

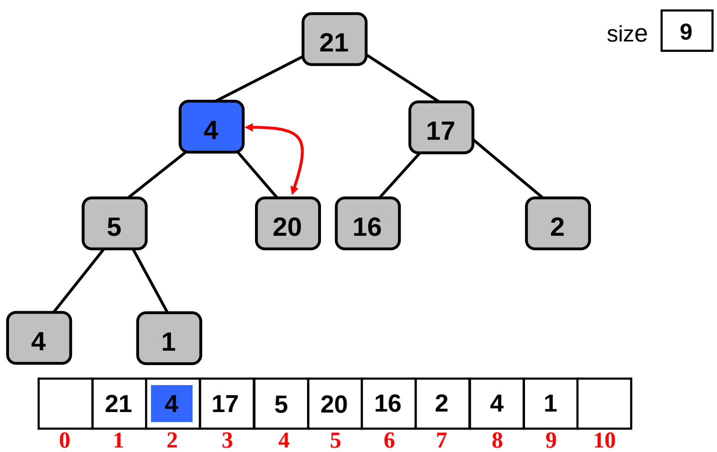


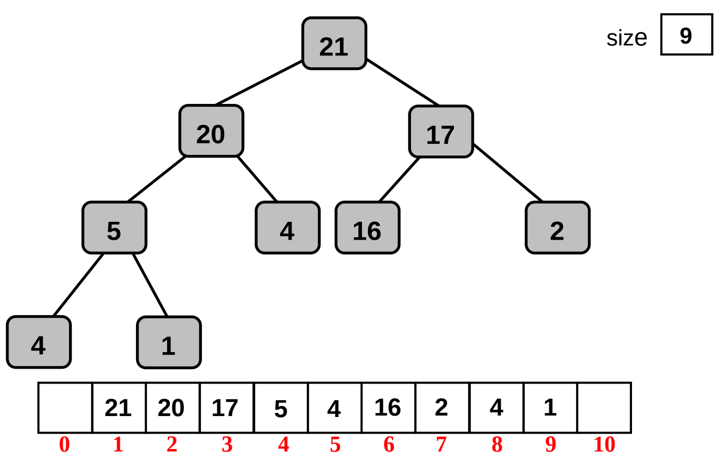
```
public void deleteMax( )
{
    heapArray[1] = heapArray[ size-- ];
    siftDown( heapArray, 1, size );
}
```



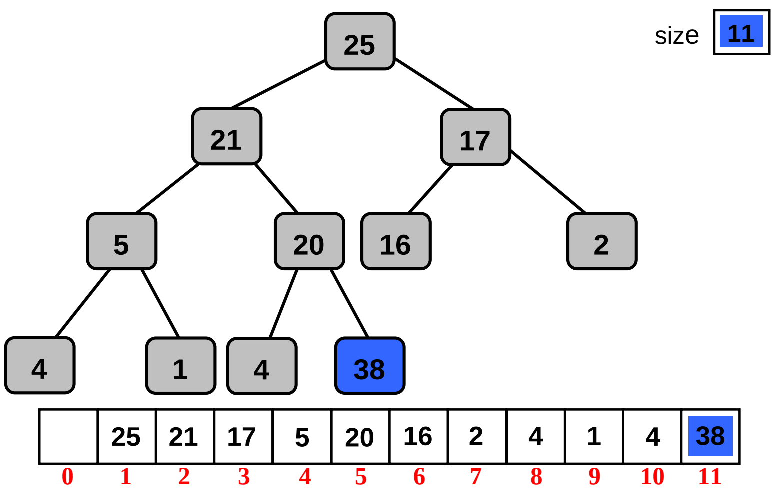



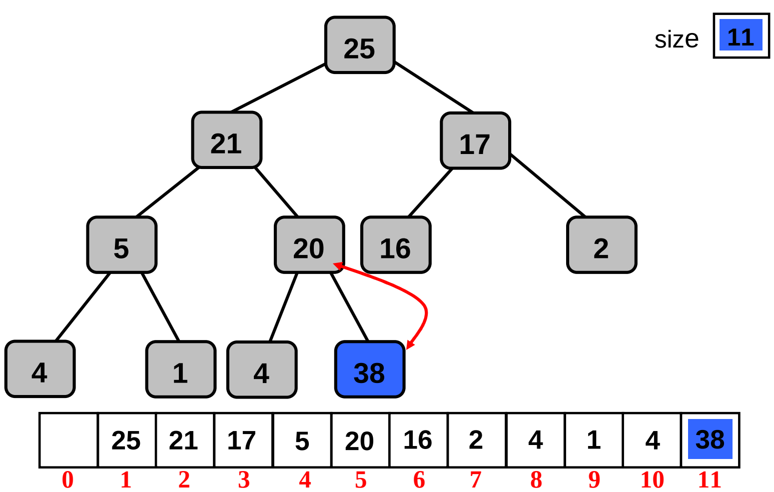

size
9
4
21
17
5
20
16
2
4
1
4 21 17 5 20 16 2 4 1
0 1 2 3 4 5 6 7 8 9 10

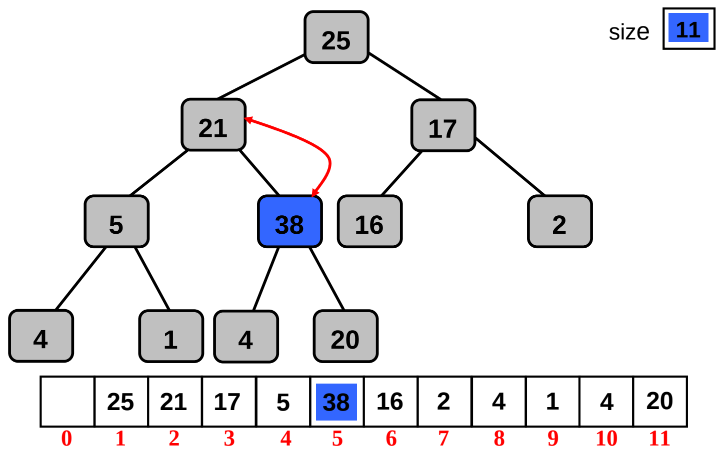

size
9
21
4
17
5
20
16
2
4
1
21 4 17 5 20 16 2 4 1
0 1 2 3 4 5 6 7 8 9 10

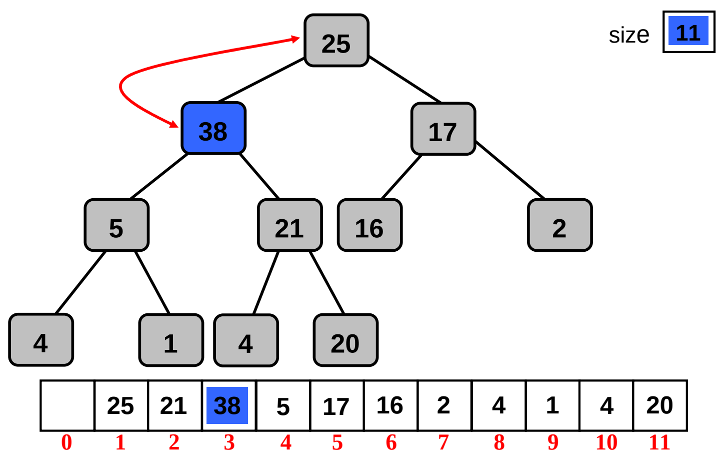

size
9
21
20
17
5
4
16
2
4
1
21 20 17 5 4 16 2 4 1
0 1 2 3 4 5 6 7 8 9 10

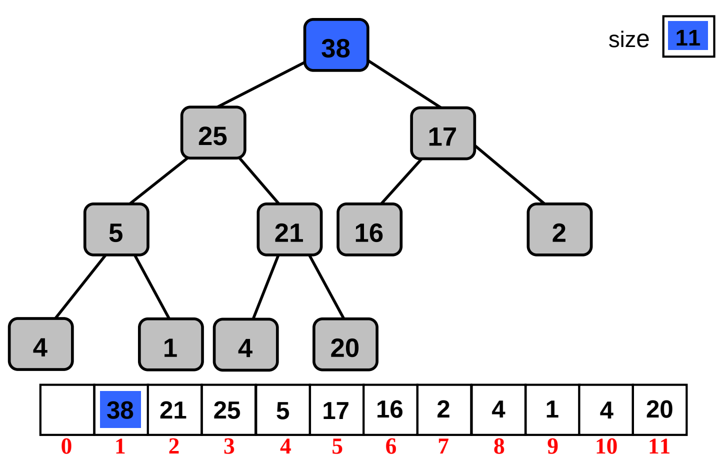
```java
public void insert( Comparable item )
{
    if ( size == maxSize )
        ;// Exercise:  write code to handle array overflow:
    heapArray[ ++size ] = item;
    siftUp( heapArray, size, size );
}
```




size 11
25
21
17
5
38
16
2
4
1
4
20
25 21 17 5 38 16 2 4 1 4 20
0 1 2 3 4 5 6 7 8 9 10 11


size 11
25
38
17
5
21
16
2
4
1
4
20
25 21 38 5 17 16 2 4 1 4 20
0 1 2 3 4 5 6 7 8 9 10 11


size 11
38
25
17
5
21
16
2
4
1
4
20
38 21 25 5 17 16 2 4 1 4 20
0 1 2 3 4 5 6 7 8 9 10 11

```java
public String toString()
{
    String str = "";
    for ( int i = 1; i < size; i++ )
        str += heapArray[i].toString( ) + ", ";
    if ( size != 0 ) str += heapArray[size].toString( );
    return( str );
}
```

***Exercise***

Determine the run-time complexity of the above method. How can we make this method more efficient?

```java
private static void siftUp
        ( Comparable [] a, int k, int m )
{
// pre-condition:
// 1 <= k <= m and the binary tree corresponding
// to a[1] ...  a[m] is a heap except that the value
// in a[k] may be greater than its parent.
    Comparable v = a[k];
    int j = k;
    int i = j/2;
    while( i > 0 && a[i].compareTo(v) < 0 )
    {
        a[j] = a[i];
        j = i;
        i = j/2;
    }
    a[j] = v;
}
```

```
private static void siftDown
        ( Comparable [] a, int k, int m )
{
// pre-condition:
// 1 <= k <= m and the binary tree corresponding
// to a[1] ...  a[m] is a heap except that the value
// in a[k] may be less than one or both of its children.
    Comparable v = a[k];
    int max_index;
    Comparable maxCh;
    int i = k;
    boolean more = m >= 2*k;
            // a[k] must have at least one child
    while( more )
    {
        max_index = maxChild( a, i, m );
        maxCh = a[ max_index ];
        if( v.compareTo( maxCh ) < 0)
        {
            a[i] = maxCh;
            i = max_index;
            more = m >= 2*i;
        }
        else
            more = false;
    }
        a[i] = v;
}
```

```
    private static int maxChild( Comparable [] a, int j, int m )
    {
    // pre-condition:  2*j <= m
    //
        int left_ch = 2*j;
        if ( left_ch == m)
            return left_ch;
        int right_ch = left_ch + 1;
        if ( a[ left_ch ].compareTo( a[ right_ch ] ) < 0 )
            return right_ch;
        else
            return left_ch;
    }
}
```

## *A More Efficient Method of Constructing a Heap from an Array of Elements*

◇ Run-time complexity of the heapifier in the heap constructor,

`Heap( Comparable [] a ),`

is $O(n \log n)$.

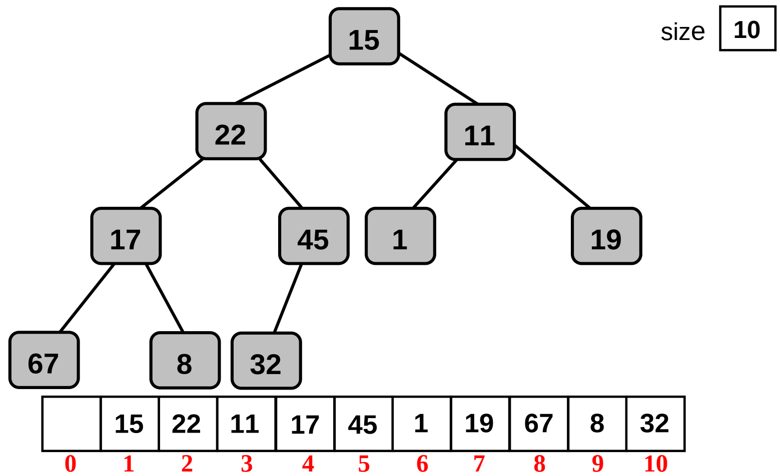
◇ *Idea for an improved method:*

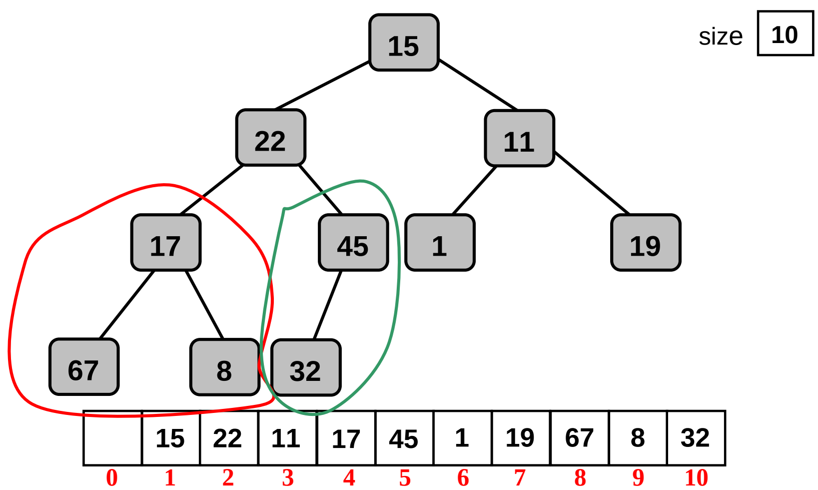
▷ let `heapArray` initially contain the bag of elements to be *heapified*, and let the height of the underlying complete binary tree be $l$ ;

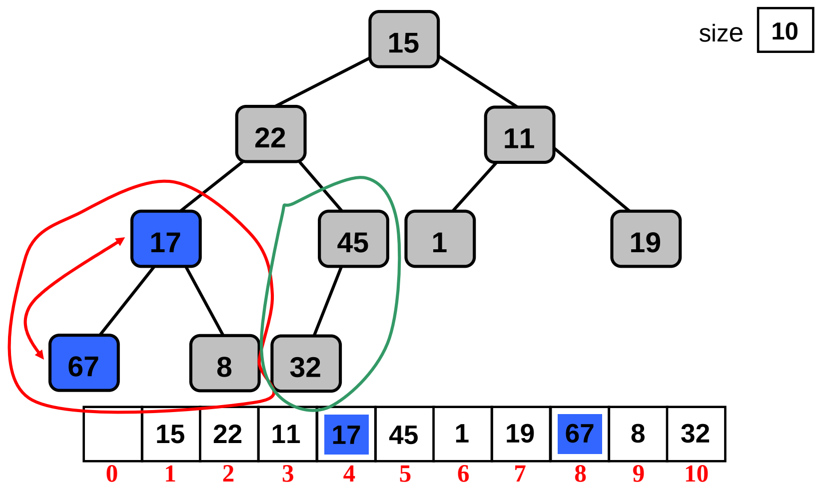
▷ make each of the subtrees whose roots are at level $l - 1$ into a heap, using `siftDown`;

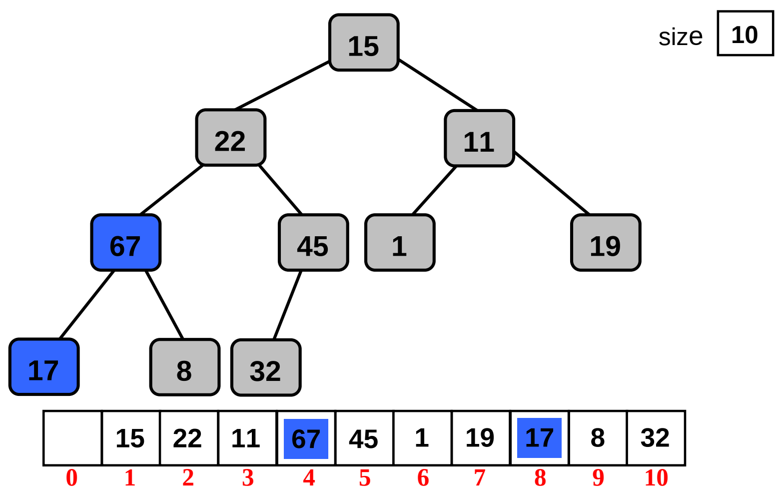
▷ repeat this process for every subtree rooted at each previous level down to and including level 0.

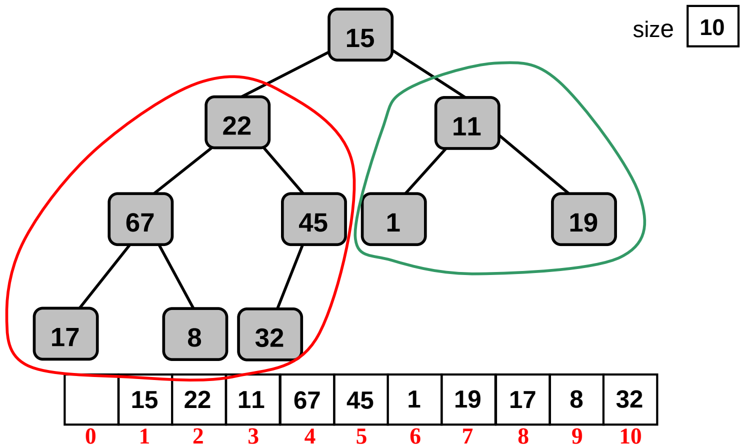
*Example*:

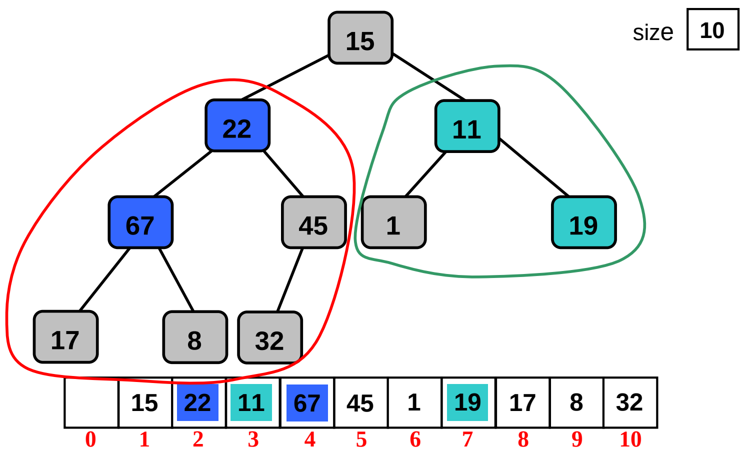

size
10
15
22
11
17
45
1
19
67
8
32
15 22 11 17 45 1 19 67 8 32
0 1 2 3 4 5 6 7 8 9 10

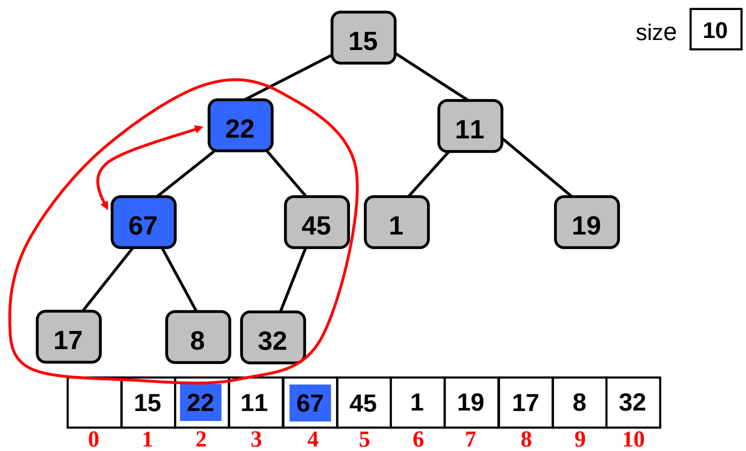

size
10
15
22
11
17
45
1
19
67
8
32
15 22 11 17 45 1 19 67 8 32
0 1 2 3 4 5 6 7 8 9 10

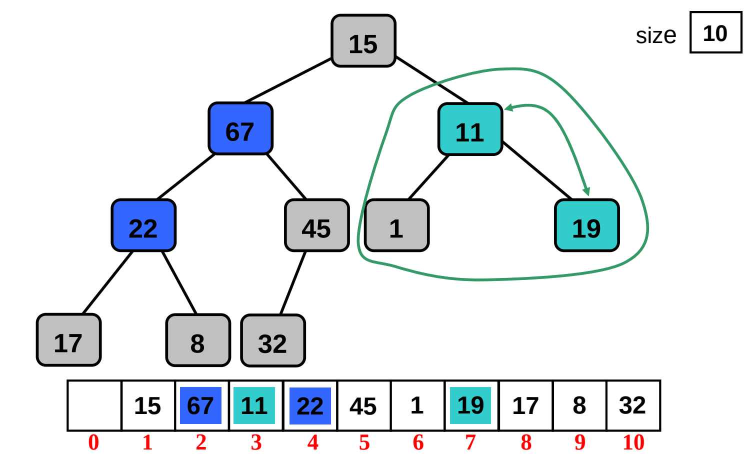

size
10
15
22
11
17
45
1
19
67
8
32
0 1 2 3 4 5 6 7 8 9 10

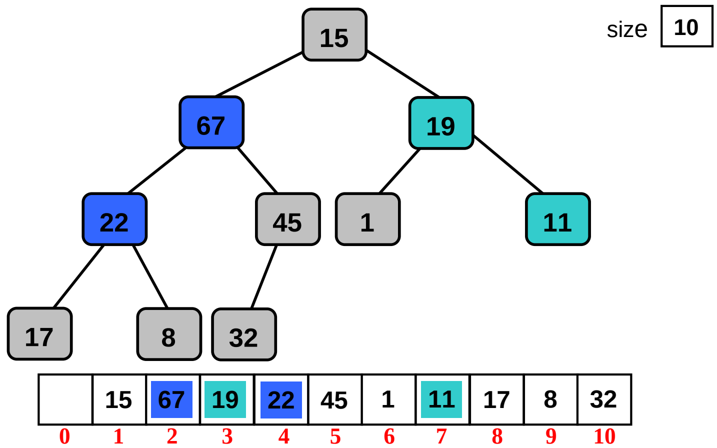

size
10
15
22
11
67
45
1
19
17
8
32
0 1 2 3 4 5 6 7 8 9 10

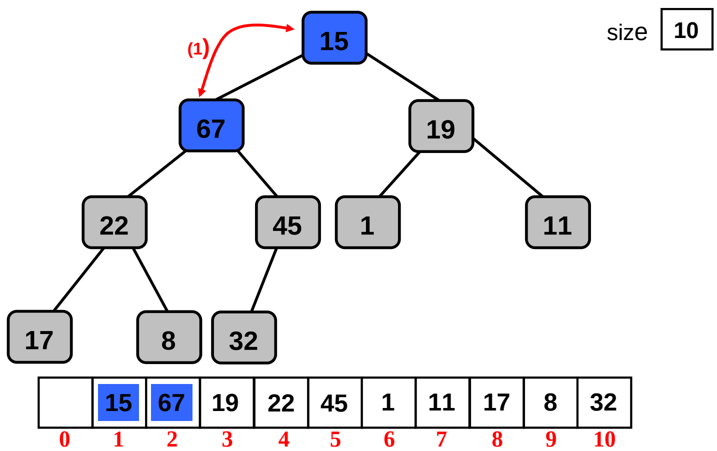

size
10
15
22
11
67
45
1
19
17
8
32
15 22 11 67 45 1 19 17 8 32
0 1 2 3 4 5 6 7 8 9 10

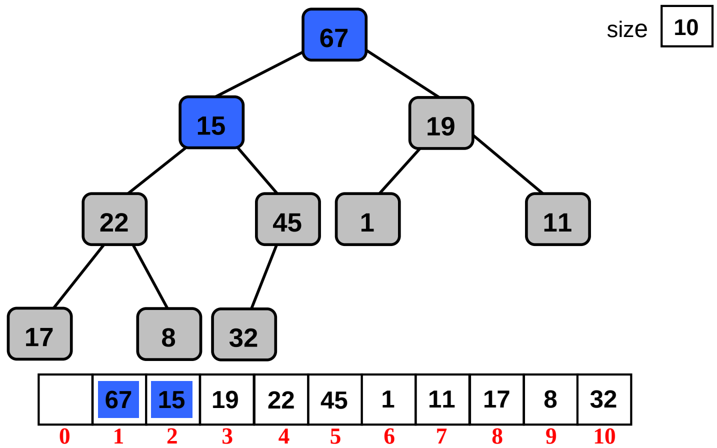

size
10
15
22
11
67
45
1
19
17
8
32
15 22 11 67 45 1 19 17 8 32
0 1 2 3 4 5 6 7 8 9 10

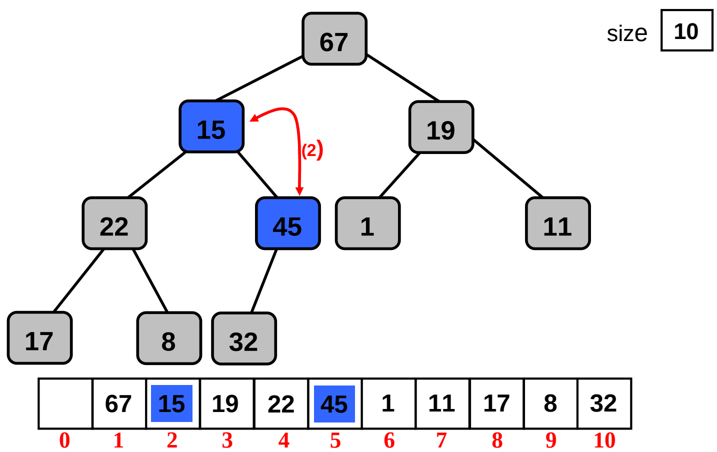

size
10
15
22
11
67
45
1
19
17
8
32
15 | 22 | 11 | 67 | 45 | 1 | 19 | 17 | 8 | 32
0 1 2 3 4 5 6 7 8 9 10

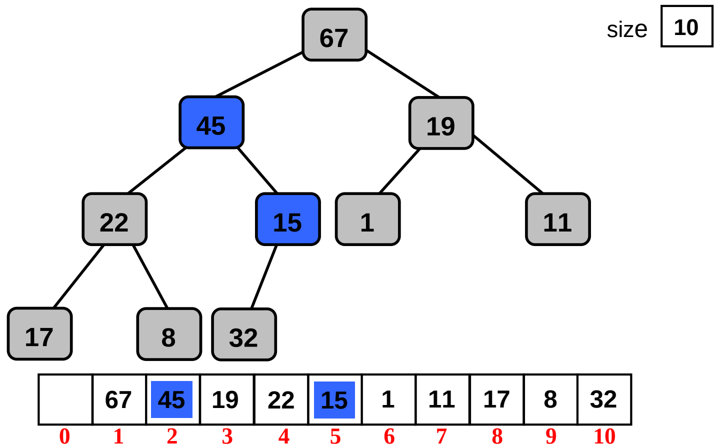

size
10
15
67
11
22
45
1
19
17
8
32
15 | 67 | 11 | 22 | 45 | 1 | 19 | 17 | 8 | 32
0 1 2 3 4 5 6 7 8 9 10

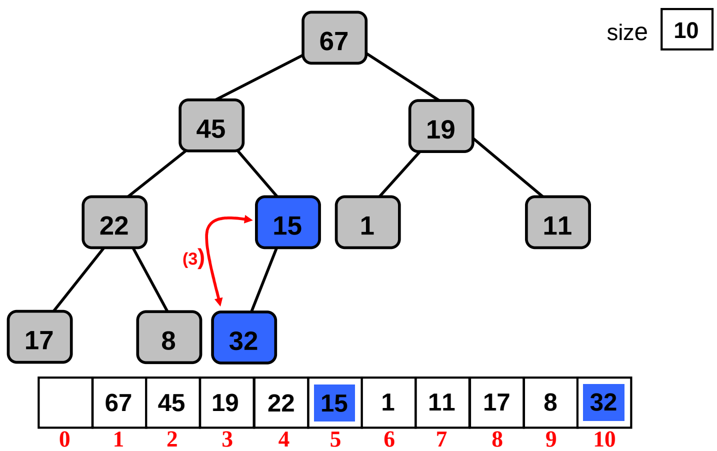

size 10
15
67 19
22 45 1 11
17 8 32
15 67 19 22 45 1 11 17 8 32
0 1 2 3 4 5 6 7 8 9 10

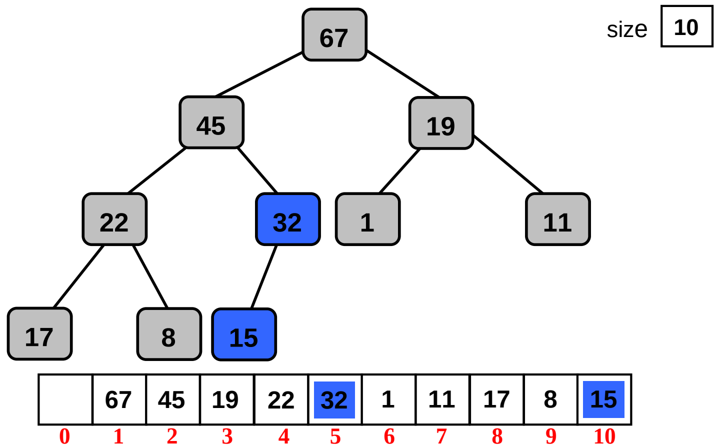

size 10
(1)
15
67 19
22 45 1 11
17 8 32
15 67 19 22 45 1 11 17 8 32
0 1 2 3 4 5 6 7 8 9 10


size
10
67
15
19
22
45
1
11
17
8
32
0
1
2
3
4
5
6
7
8
9
10


size
10
67
15
19
(2)
22
45
1
11
17
8
32
0
1
2
3
4
5
6
7
8
9
10


size
10
67
45
19
22
15
1
11
17
8
32
67 45 19 22 15 1 11 17 8 32
0 1 2 3 4 5 6 7 8 9 10


size
10
67
45
19
22
15
1
11
(3)
17
8
32
67 45 19 22 15 1 11 17 8 32
0 1 2 3 4 5 6 7 8 9 10


size
10
67
45
19
22
32
1
11
17
8
15
67
45
19
22
32
1
11
17
8
15
0
1
2
3
4
5
6
7
8
9
10

## *Exercises*

(i) Replace the inefficient heapifier in the heap constructor,

```
Heap( Comparable [] a ),
```

by code based on the above method;

(ii) Deduce that the run-time complexity of the resulting constructor is $O(n)$, where $n$ is the size of the constructed heap.

## HeapSort

To sort $n$ values using ***Heapsort***

(i) first construct a heap from the $n$ values using the efficient constructor;

(ii) largest value is now at the top of the heap

- ◇ output this value, then do `deleteMax`;

(iii) repeat step 2 until all values have been output.

Run-time complexity is

$$c_1 n + \sum_{i=1}^{n} (c_2 + c_3 \log i)\,,$$

which is $O(n \log n)$.