Section 3.5: Heaps

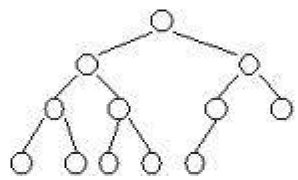
Priority Queue – abstract data type that processes items according to a specific priority

Operations

- insert
- findMin (or findMax)
- deleteMin (or deleteMax)

Examples of data structures that implement a priority queue: **heap**, array, binary search tree

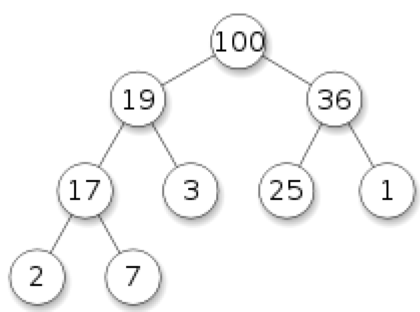
Recall: A **complete binary tree** is a binary tree in which every internal node has at most two children, and all levels are filled except possibly the last, which is filled from left to right.



A **heap** (or **binary maxheap**) is an array (conceptualized as a complete binary tree).

- the root is the element a[1], i.e., the first element of the array
- the data in the array satisfies a special property, called the **heap property**:
  for every node i other than the root, the data stored at the parent of i is greater than or equal to the data stored at node i

`heap property:   a[parent(i)] ≥ a[i]`



Data or key value	100	19	36	17	3	25	1	2	7
Index of node	1	2	3	4	5	6	7	8	9

The heap also has an attribute called the **heapSize**, which is the number of elements in the heap. Clearly, the heapSize must always be less than or equal to the number of elements in the array.

`a.heapSize ≤ a.length`

Which of the following are heaps?

array `a =`

100	19	36	19	3	25	1	2	7
1	2	3	4	5	6	7	8	9

```
a.heapSize = 9
a.length = 9
```

array `a =`

100	19	36	17	3	40	1	2	7	8
1	2	3	4	5	6	7	8	9	10

```
a.heapSize = 7
a.length = 10
```

array `a =`

100	19	36	17	3	40	1	2	7	8
1	2	3	4	5	6	7	8	9	10

```
a.heapSize = 5
a.length = 10
```

For an array that does represent a heap with heapSize equal to the length of the array, where can you find the maximum value of the array?

What is the height of a heap with n nodes?

Section 3.5: Heap Operations
(binary maxheap)

For the operations that follow:

- v is the array (the heap)
- n is the heapSize of v
- i is an index in the array v

left(i)	**right(i)**	**parent(i)**
return 2*i	return 2*i+1	return i/2

heap_largest(v)
```
return v[1]
```

siftdown(v,i,n)   // other texts call this operation heapify
// Assumptions: subtree rooted at left child of i is a heap,
// subtree rooted at right child of i is a heap
// only node i violates the heap property
// n is the heapSize
```
while (left(i) <= n)      // i.e., while i has a left child
{
  child = left(i)
  if  (left(i) < n && v[right(i)] > v[left(i)])
    child = right(i)      // child contains index of larger
child
  if (v[child] > v[i])
  {
    temp = v[i]           // swap v[i] and v[child]
    v[i] = v[child]
    v[child] = temp
  }
  else
    break                 // exit while loop
  i = child
}
```

heap_delete(v,n)   // extracts the max, i.e., deletes the root
// n is the heapSize
```
v[1] = v[n]
n = n - 1
siftdown(v,1,n)
```

heapify(v,n)   // other texts call this operation buildHeap
```
for  i = n/2 downto 1
   siftdown(v,i,n)
```

```
heapSort(v,n)
// takes an array v, builds a heap and sorts it
heapify(v,n)
for  i = n  downto 2
{
   swap(v[1], v[i])
   siftdown(v,1,i-1)
}
```

```
heap_insert(val,v,n)
// n  is the heapSize
// val is the value to be inserted
n = n+1
i = n
while (i > 1 && val > v[parent(i))
{
  v[i] = v[parent(i)]
  i = parent(i)
}
v[i] = val
```

```
updateKey(v,i,n)
// n  is the heapSize
// used to maintain the heap property after the key at index i
// has been changed; only v[i] violates the heap property;
// trace a path from i up through the heap (following parents),
// swapping along the way, until the heap property is restored
// see if you can write an algorithm for this operation!
```

```
recursive_siftdown(v,i,n)
// n  is the heapSize
if (left(i) <= n)
{
  child = left(i)
  if  (left(i) < n && v[right(i)] > v[left(i)])
    child = right(i)      // child contains index of larger
child
  if (v[child] > v[i])
  {
    swap(v[i],v[child])
    recursive_siftdown(v,child,n)
  }
}
```