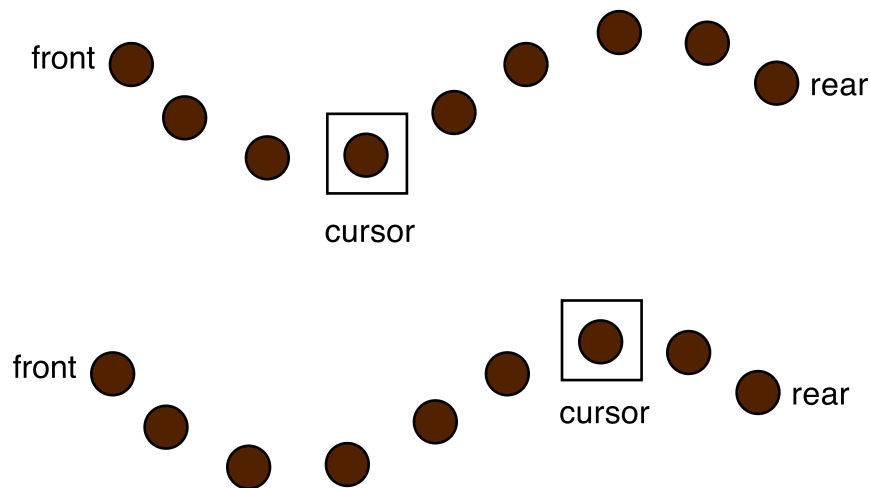
Positional List

The Concept

The **list** is a **generalization** of **both stacks and queues**. But we are very restricted as to **where** these insertions and deletions may occur. Generally, two positions are recognized: **front** and **rear**. If a list has at least one node, the front is the position occupied by this node; if a list is empty, the front coincides with the rear.



For flexibility, both access and mutation should happen anywhere in the list. For this, we need the concept of a **current position** or a **cursor**, which defines a *viewing point* or a *viewing window*. It is the place where the activities are directed. A **positional list** allows this. The use of such an ADT

- Allows a person to leave a line before reaching the front
- Allows word processing actions (insert or delete happen at the cursor)

Here's a possible informal specification for a positional list ADT

Accessor

```
boolean isEmpty() : Is list empty?
int lengthOf()    : yields # of nodes in list
Object getObj()   : contents at the cursor
boolean atFront() : Is cursor the front of list?"
boolean atRear()  : Is cursor the rear of list?"
```

Navigator

```
void toFront() : sets cursor to front of list
void toRear()  : sets cursor to rear of list
void toNext()  : moves cursor one place closer
to rear
void toPrev()  : moves cursor one place closer
to front
```

Mutators

void replace(x) : replaces current node's contents with x

void insert(x) : inserts new node containing x as the predecessor of the cursor

void insertFront(x) : inserts new node containing x in front

void insertRear(x) : inserts new node containing x at rear

void remove() : removes the node at the current position; the successor becomes the current position

Find the length of (i.e., the number of nodes in) a list.

```
public static int length(PositionalList list){
        int center = 0;
        list.toFront();
       // center = # of nodes before the cursor
       while ( !list.atRear() )  {
           center = center + 1;
           list.toNext();
        }
        return center;
    }
```

Priority Queue ADT

A regular queue is a FIFO object, but there are many real-life applications, where the FIFO is not adequate. Here are some examples.

- Air-traffic control
- Standby passenger queue for a flight
- Waiting list for a course at UI

In such cases, a **priority queue** is relevant. Each element (v) has a key (k) that defines its priority. *Lower key means higher priority*. So the object with the minimum key will be dequeued first.

Each **entry** is a **key-value** pair.

A Priority Queue ADT can support the following methods

Insert (k, v)	Creates value v with key k
removeMin ( )	Returns and removes the value with minimum key
Min( )	Returns the value with the minimum key
Size( )	Returns the number of entries
isEmpty( )	Returns true when the priority queue is empty

Priority can be a number, or any java object, as long as **they are comparable**, and help create a total order among the elements.

```
Compare(x, y): returns an integer i such that
• i < 0 if a < b,
• i = 0 if a = b
• i > 0 if a > b
An error occurs if a and b cannot be compared
```

If the ordering is not natural, then a comparator is provided at the queue construction time. Note that the notion of priority may vary from one user to another. For example, if there are two stocks P and Q to buy, one Alice may P to be more important than Q, but Bob may think differently.

One common use of a priority queue is the "event queue" in a simulation. Here

Key = time of the event

Value = Description of the event

Implementing a Priority Queue

Think of these

Use an array that stores elements **in arbitrary order**.

What is the *insert time*? What is the *removeMin* time?

4 — 5 — 2 — 1 — 9 — 3

Use an array with values **in sorted order**, low to high.

What is the **insert time**? What is the **removeMin** time?

1 — 2 — 3 — 4 — 5 — 9

Also, we'll need to keep track of the number of values currently in the queue, we'll need to decide how big to make the array initially, expand the array if it gets full

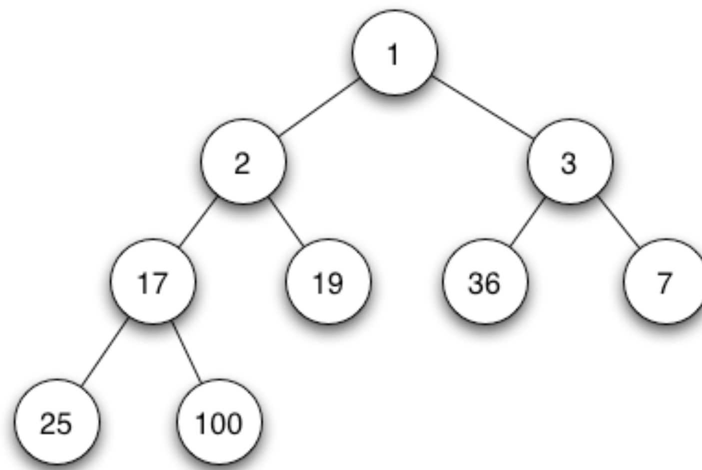
What if we use a linked list?

Binary Heap

A popular implementation of the priority queue ADT uses a **binary heap**. A binary heap is a **complete** binary tree, a tree in which every level is full, except possibly the bottom level, or bottom row, which is filled from left to right, and it satisfies the **heap-order property**

(For a min-heap) **"Key at a node ≥ Key at its parent"**

[For a max-heap, this order requirement is the reverse, i.e. "Key at a node ≥ Key at its parent"]



Source: Wikipedia

The array representation of the binary heap is as follows. Let us keep the key at index 0 blank or null.

	1	2	3	17	19	36	7	25	100
0	1	2	3	4	5	6	7	8	9

Thus, for node k, the left child and the right child are nodes 2k and (2k+1) respectively. Similarly, for a node i (that is not the root), the parent is node $\lfloor i/2 \rfloor$

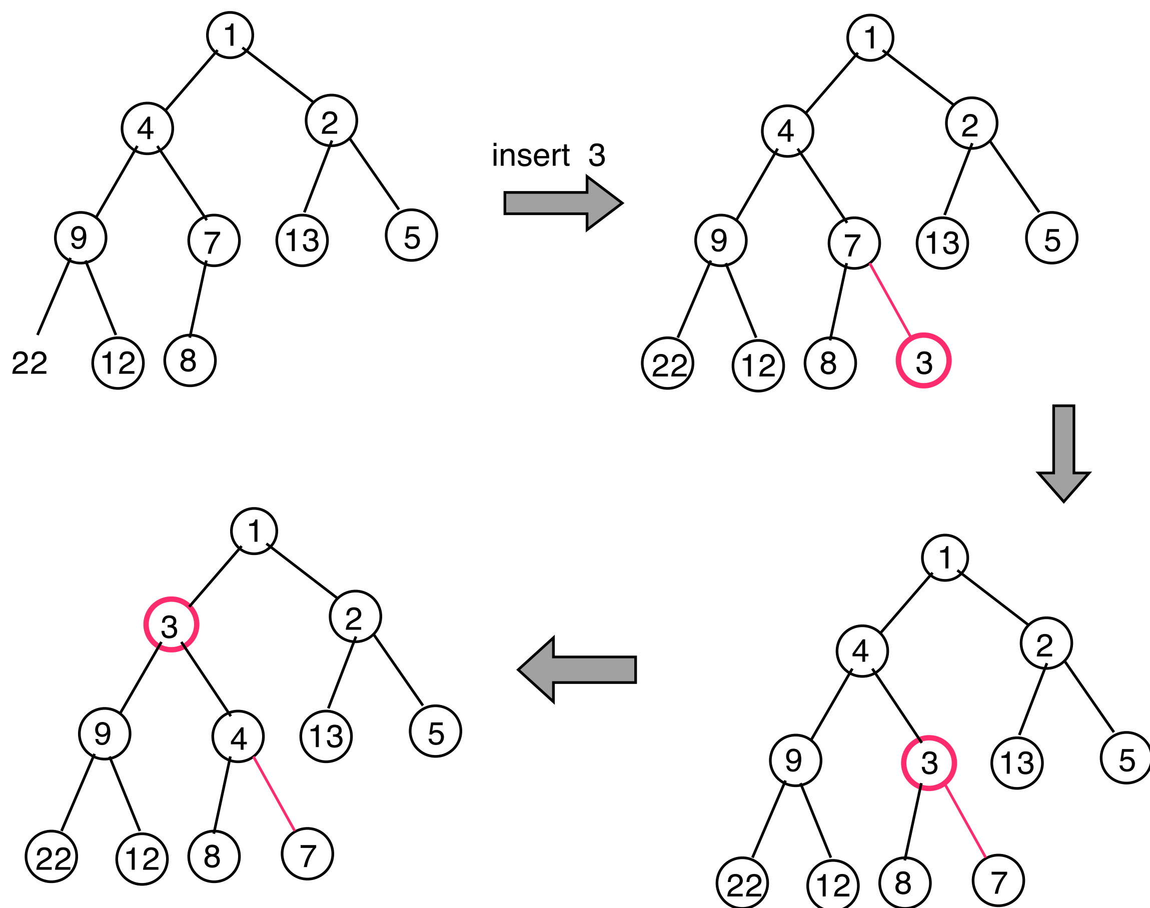
Priority queue operations with a binary heap

1. **Min ( )**

    Return the entry at the root node

2. **Insert x = (k, v)**

Add x to the bottom row, at the first "free" slot from the left. If the bottom level is full, then create a new level and make x the first element of the new level. If the heap-order property is violated, then let x bubble up the tree via repeated swap operations (between the keys at the parent and the child)

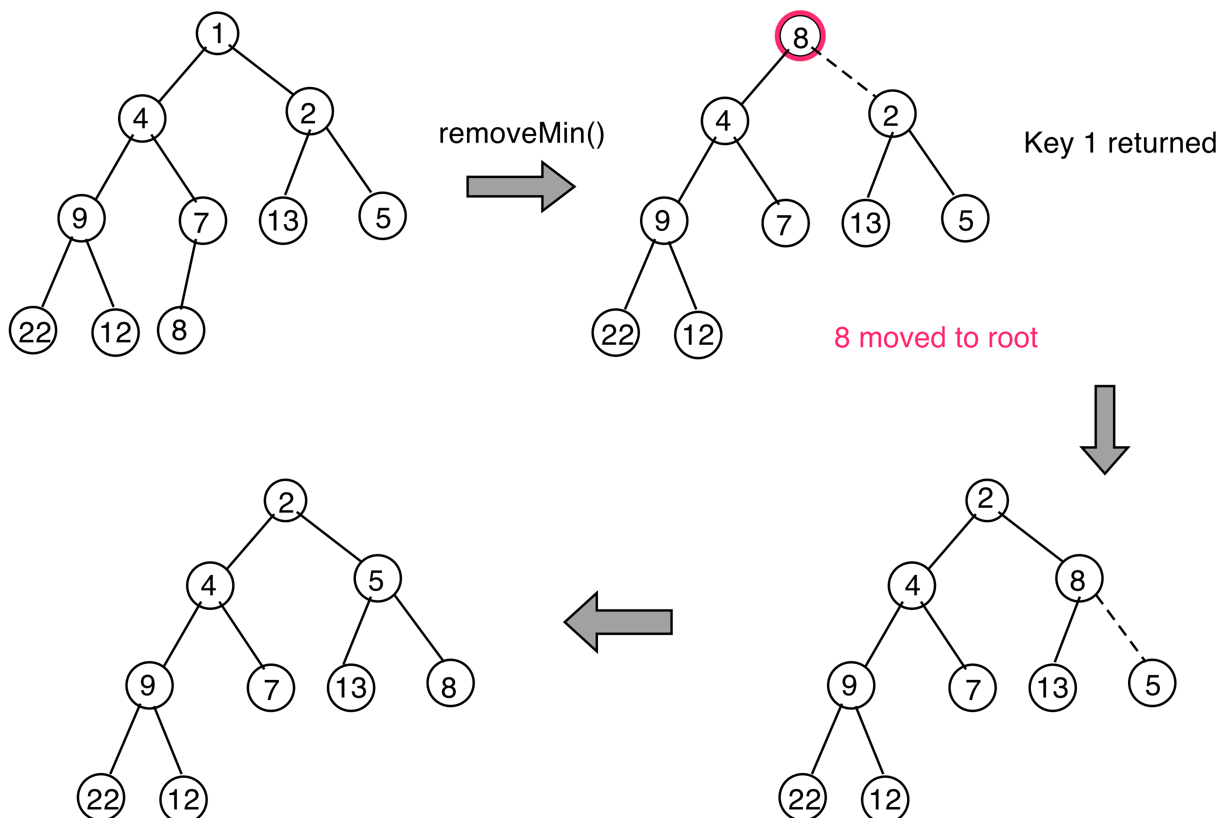


Argue that the heap-order property is satisfied at the end of the insertion.

3. **RemoveMin()**

If the heap is empty, then return null (or throw an exception). Otherwise, (1) return the root node, and (2) fill the hole with the last node in the array.

The new minimum must be one of the children of the former root. Since each subtree is a binary heap, the new root will bubble down (via repeated swap operations) until the heap property is restored.



Time complexity

	Unsorted array	Sorted array	Binary heap
min	O(n)	O(1)	O(1)
insert	O(1)*	O(n) O(log n)**	O(log n)
remove	O(n)	O(1)	O(log n)

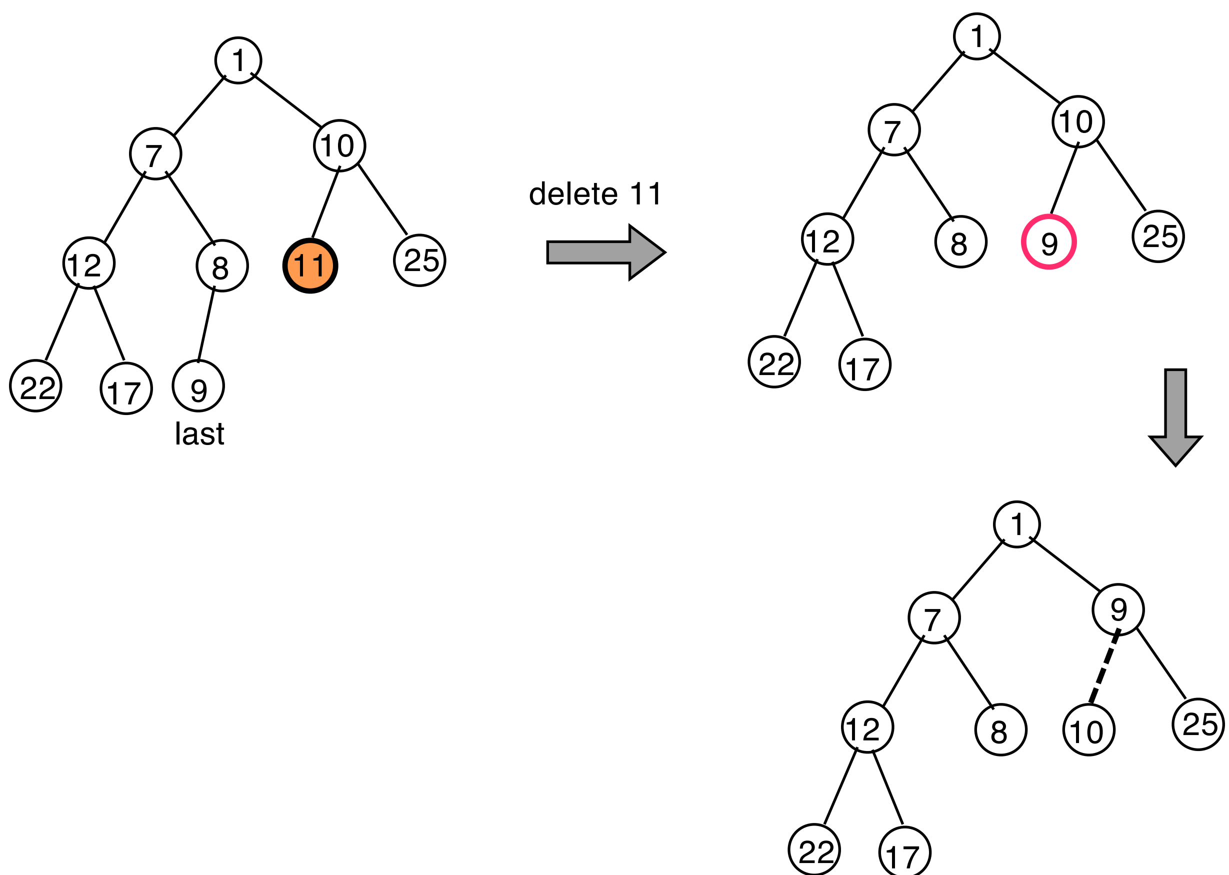
* It assumes that there is enough space to insert the entry. If not, then space has to be allocated and entries have to be copied to the new space, which will take O(n) time.

** You can use binary search for the position where the new entry will be inserted. But there is an overhead for making room in an array – the items have to be shifted to the right, and each such operation will take O(n) time.

Other operations on a Binary Heap

Delete Key.

It is useful when someone wants to leave the queue. The **last key** substitutes the deleted key. The last key bubbles up or down as needed to restore the heap order property.



Heapify

What is heapify? Given n elements, construct a binary heap out of them.

Bottom-up method.

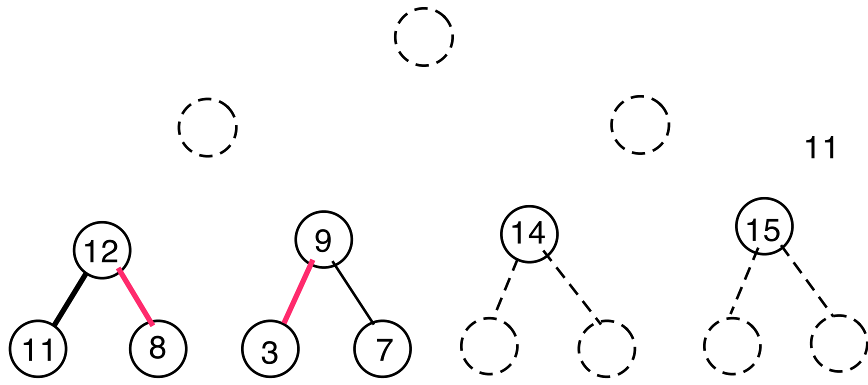
- First, decide how many levels will be there.
- Place a set of nodes at the bottom level. Each such node is a degenerate heap.
- Next, fill the next level by adding a parent to the nodes at the lower level. Bubble the added nodes, as necessary, to restore the heap order property.
- Continue this up to the topmost level.

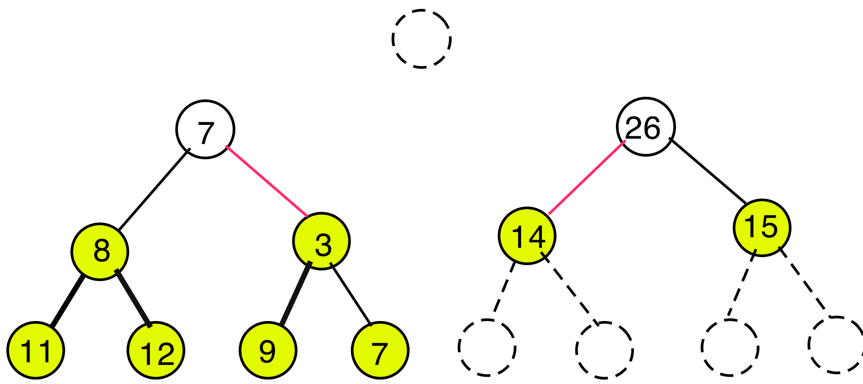
Can easily do in $O(n \log n)$ steps. Why?

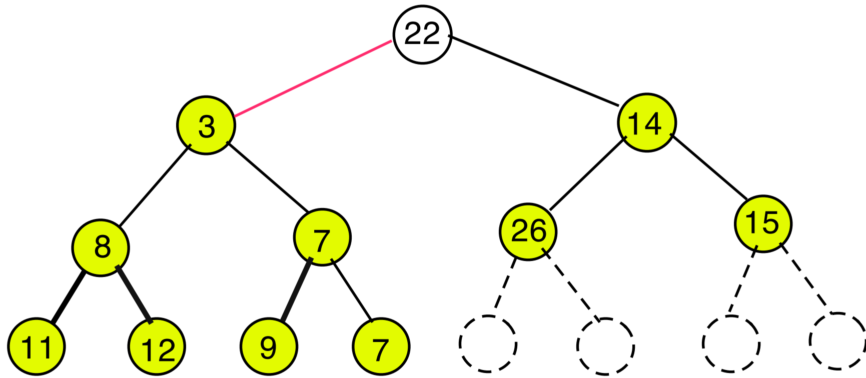
Example. Try to construct a heap from the elements

22	7	26	12	9	14	15	11	8	3	7

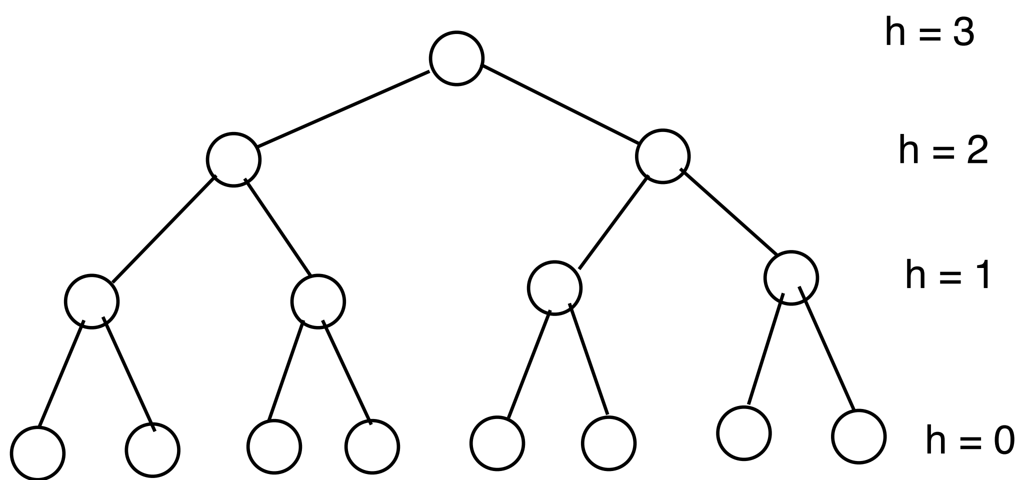
First, how many levels will be there?


11
12
9
14
15
11
8
3
7


7
26
8
3
14
15
11
12
9
7


22
3
14
8
7
26
15
11
12
9
7

Timewise, we can do better than O(n log n). We can show that heapification actually takes O(n) time. Why? Let us see.



Height = h	Number of nodes = n
0	8
1	4
2	2
3	1

So, the number of nodes n with height $h \le \left\lceil \frac{n}{2^{h+1}} \right\rceil$

Thus the total time for the n nodes to bubble down and form a binary heap =

$$\sum_{h=0}^{h=\lceil \log_2 n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil . h =$$

$$\le \sum_{h=0}^{h=\lceil \log_2 n \rceil} \frac{n}{2^h} . h \quad [\sum_{h=0}^{\infty} \frac{h}{2^h} = 2]$$

$$\le 2.n$$

Sorting with a priority queue

1. Insert elements into an empty priority queue
2. Remove the elements from the priority queue.

If we implement the PQ with a binary heap, then we can sort in O(n log n) time, since each insert or delete operation in a binary heap takes O(log n) time.

Phase 1. The kth insert takes O(log k) time. So this phase takes O(n log n) time.

Phase 2. The jth removeMin() operation takes O(log (n-j+1)) time. So this phase takes O(n log n) time.

This is the idea of **heap-sort**.