Priority Queues

An internet router receives data packets, and forwards them in the direction of their destination. When the line is busy, packets need to be queued.

Some data packets have higher priority than others, and need to be sent first. Queuing is not FIFO.

A Priority Queue is a data type where elements have different priorities. The priority queue provides three main methods:

findMin Return the most urgent element (highest priority, e.g. minimal event time).
deleteMin Remove the most urgent element from the priority queue.
insert Add a new element to the priority queue.

Scala Priority Queues

In `scala.collection.mutable.PriorityQueue[T]`, the methods are called `head`, `dequeue`, and `enqueue`.

```
scala> import scala.collection.mutable.PriorityQueue
scala> val p = new PriorityQueue[Double]
scala> p.enqueue(2.3, 5.9, 1.5, 9.8, 13.0)
scala> p.head
res1: Double = 13.0
scala> p.dequeue()
res2: Double = 13.0
scala> p.head
res3: Double = 9.8
scala> p.enqueue(11.2)
scala> p.head
res5: Double = 11.2
```


Binary Heaps

(an implementation of Priority Queues)

The abstract data type Priority Queue provides three main methods:

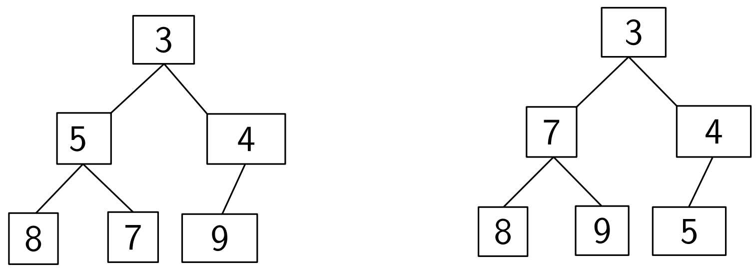
findMin Return the smallest element.
deleteMin Remove the smallest element.
insert Add a new element.

Implementation	`findMin`	`deleteMin`	`insert`
Unsorted list	$O(n)$	$O(n)$	$O(1)$
Sorted list	$O(1)$	$O(1)$	$O(n)$
Heap	$O(1)$	$O(\log n)$	$O(\log n)$

Heaps and Heap Order

We will use a Heap: a complete binary tree, with one element per node.

Heap-order: For every node x with parent p, the element in p is smaller than the element in x.



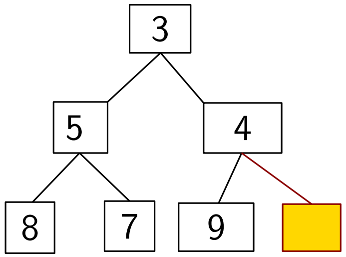
Many different heaps are possible for the same data.

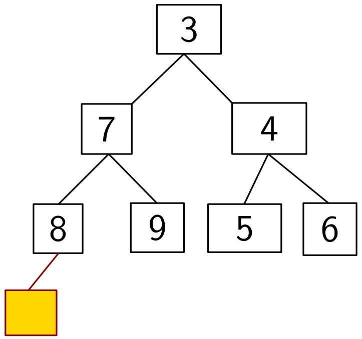
The Insert Operation

- The `insert` method adds a given element to the heap.
- We need to maintain heap order and completeness

There is only one correct node that can be added:

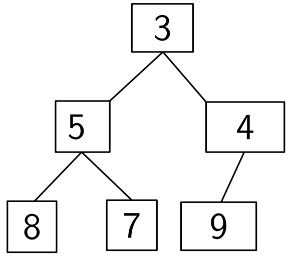
- Either the next open position from the left at level h
- Or the first position in level $h+1$ if level h is full





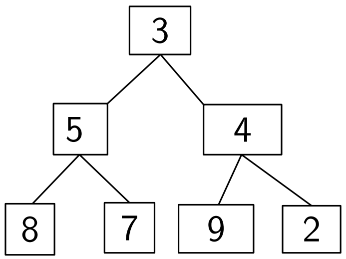
KAIST CS206

- Once we have placed the new node in the proper position, then we must account for the ordering property
- We simply compare the new node to its parent value and swap the values if necessary
- We continue this process up the tree until either the new value is greater than its parent or the new value becomes the root of the heap



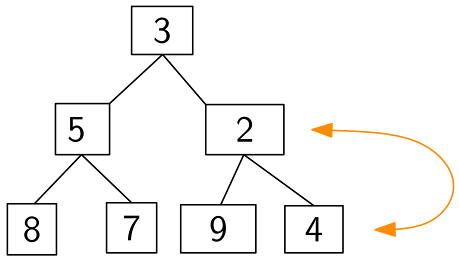
KAIST CS206

- Once we have placed the new node in the proper position, then we must account for the ordering property
- We simply compare the new node to its parent value and swap the values if necessary
- We continue this process up the tree until either the new value is greater than its parent or the new value becomes the root of the heap

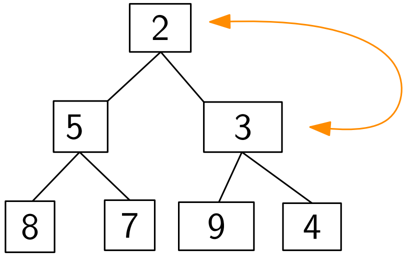


KAIST CS206

- Once we have placed the new node in the proper position, then we must account for the ordering property
- We simply compare the new node to its parent value and swap the values if necessary
- We continue this process up the tree until either the new value is greater than its parent or the new value becomes the root of the heap



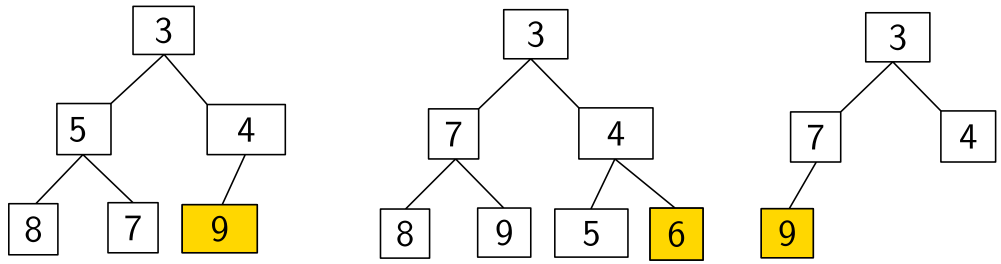
- Once we have placed the new node in the proper position, then we must account for the ordering property
- We simply compare the new node to its parent value and swap the values if necessary
- We continue this process up the tree until either the new value is greater than its parent or the new value becomes the root of the heap



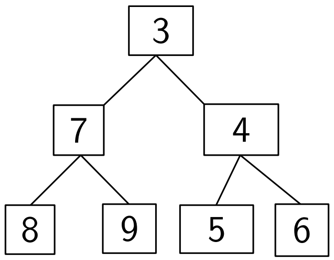
The DeleteMin Operation

- The `deleteMin` method removes the minimum element from the heap
- The minimum element is always stored at the root
- After removing the minimum, we have to fill the root with a replacement element.

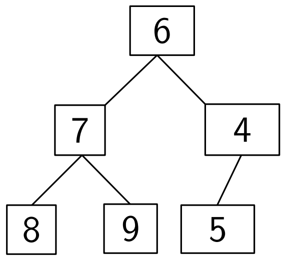
- The replacement element is always the last leaf
- The last leaf is always the last element at level h



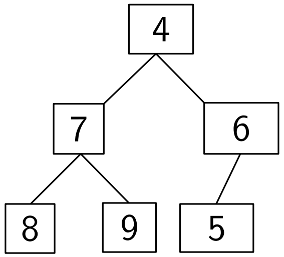
- Once the element stored in the last leaf has been moved to the root, the heap will have to reordered
- This is accomplished by comparing the new root element to the smaller of its children and swapping them if necessary
- This process is repeated down the tree until the element is either in a leaf or is less than both of its children



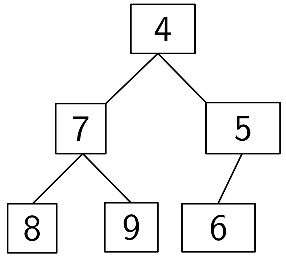
- Once the element stored in the last leaf has been moved to the root, the heap will have to reordered
- This is accomplished by comparing the new root element to the smaller of its children and swapping them if necessary
- This process is repeated down the tree until the element is either in a leaf or is less than both of its children



- Once the element stored in the last leaf has been moved to the root, the heap will have to reordered
- This is accomplished by comparing the new root element to the smaller of its children and swapping them if necessary
- This process is repeated down the tree until the element is either in a leaf or is less than both of its children



- Once the element stored in the last leaf has been moved to the root, the heap will have to reordered
- This is accomplished by comparing the new root element to the smaller of its children and swapping them if necessary
- This process is repeated down the tree until the element is either in a leaf or is less than both of its children



Because the tree has a fixed structure, we can implement it using an array.

The height of the tree is $\log n$, and so `insert` and `deleteMin` take $O(\log n)$ time.

Building a heap out of n given items takes time $O(n \log n)$ using n insert operations.

We can do better: Just throw all n items into the array, then run `buildHeap` to fix the heap-ordering in $O(n)$ time.

Idea: If we have a binary tree such that the two subtrees of the root are heaps, then we only need to let the root element percolate down into a correct position to ensure that the tree is a heap.

Priority-Queue Sort

We can easily sort using a priority queue:

```
def pqSort(A: Array[E]) {
  val Q = new PriorityQueue[E]
  for (el <- A)
    Q.enqueue(el)
  for (i <- 0 until A.length)
    A(i) = Q.dequeue()
}
```

What is the time-complexity?

- $n\times$ `insert`,
- $n\times$ `deleteMin`.

Priority-Queue Sort

Priority queue implemented as unsorted list:
$n \times O(1)$ for `insert`: $O(n)$
$n \times O(n)$ for `deleteMin`: $O(n^2)$

$\approx$ Selection-Sort

Priority queue implemented as sorted list:
$n \times O(n)$ for `insert`: $O(n^2)$
$n \times O(1)$ for `deleteMin`: $O(n)$

$\approx$ Insertion-Sort

Priority queue implemented using binary heap:
$n \times O(\log n)$ for `insert`: $O(n \log n)$
$n \times O(\log n)$ for `deleteMin`: $O(n \log n)$

$\Rightarrow$ Heap-Sort

Heap-Sort

- Instead of inserting the n elements one by one, we create a heap operating on `A`;
- Use `buildHeap` to ensure heap ordering on the heap in $O(n)$ time;
- Take out one element at a time using `deleteMin`.

We don't need extra storage!
We can put the items obtained by `deleteMin` into the array element that has become free because the size of the heap has decreased.

Improvements:

- Use max-heap so that items are sorted into increasing order
- Change indexing of array so that heap starts at index 0.