Lecture 28: Heaps (as an implementation for priority queues)

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

March 17, 2014

In this lecture we talk about a different type of binary tree called a **heap**. Heaps are an efficient way of implementing priority queues. We will first review priority queues, then present the main properties of heaps, discuss their implementation using arrays and show the main operations available on the heap ADT.

1 Priority queues

A very useful data structure for many applications is the **priority queue**. A priority queue contains pairs of objects and "priorities" (or keys) associated with them. For example, a computing server may get jobs that it has to run, and each job may have an integer number associated with it, indicating how urgent the job is. In this case, the processing of the jobs is not done on a first-come first-served basis, but instead it is done in the order of their priorities, from the most urgent to the least urgent job. In the further description we assume without loss of generality that lower keys indicate more priority.

The main operations that need to be implemented in a priority queue are:
Object findMin() - returns the object that has the most priority (the lowest key)
void insert (**Object** o, **int** priority) - inserts in the queue the specified object with the specified priority
Object removeMin() - removes from the queue the object with most priority

So far, we have discussed implementations of queues using linear data structures, such as linked lists or arrays. Priority queues can be implemented using such structures as well. What are the options?

1. **Sorted array**
   If we implement the queue as a sorted array, the minimum is always the first element (assuming increasing order) so retrieving it is trivial and takes $O(1)$. Removing the minimum element is also $O(1)$, if we allow the head of the queue to be represented as an index which can loop around in the array as needed. In this case, removing the min is achieved by just incrementing this index. Inserting a new element in the queue is more complicated, because we

must first do a binary search to find the appropriate place for the element, and then we must shift the other array elements to make room for the new entry. This takes $O(\log n) + O(n)$ in the worst case (where n is the number of elements in the queue), which results in $O(n)$ complexity.

2. **Linked List**
If we implement the queue as a linked list, the element with most priority will be the first element of the list, so retrieving the content as well as removing this element are both $O(1)$ operations. However, inserting a new object in its right position requires traversing the list element by element, which is an $O(n)$ operation.

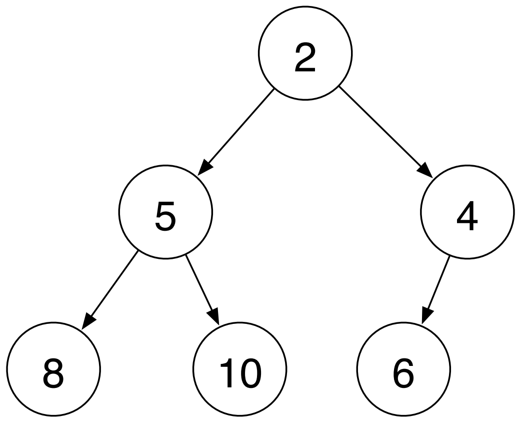
Hence, neither the array nor the linked list implementation work well if a lot of insertion operations are required by the application using the priority queue. The heap data structure allows insertions to be performed more efficiently, while making removal a bit more expensive.

2 Heap definition

A **heap** is a binary tree with two important properties:

- For any node n other than the root, $n.key \geq n.parent.key$. In other words, the parent always has more priority than its children.
- If the heap has height h, the first $h - 1$ levels are full, and on the last level the nodes are all packed to the left.

An example of a heap, which respects these properties, is presented below. For simplicity, only the keys are shown. Note that, unlike in the case of binary search trees, in a heap there is no special relationship between siblings. The height of the heap in this example is 2. The ordering between the keys of parents and children is present *at every level*.



Suppose that we have a heap of height h. The maximum number of nodes for this height is achieved if the last level is complete. In this case, we have $1 + 2 + \cdots + 2^h = 2^{h+1} - 1$ total nodes in the tree. What is the minimum number of nodes in the heap? To see this, consider the fact that the heap must be complete at every internal level i from 0 to $h - 1$. Hence, if the height is h, we must have $1 + 2 + \ldots + 2^{h-1} = 2^h - 1$ nodes on the first $h - 1$ levels and we need 1 node on the last level, for a total of $2^h - 1 + 1 = 2^h$ nodes. Hence, the number of nodes n in a heap of size h is between 2^h and $2 \cdot 2^h - 1$. Inverting this relationship, we get that the height h is $O(\log n)$. This suggests that heap operations should work in $O(\log n)$ (where n is the number of elements in the heap).

3 Implementing heaps using arrays

Because the elements in a heap are "packed" together, it is easy to implement a heap using an array. Assume that we start indexing the array at 1, and consider the example heap above. The array representation will be as follows:

Content	2	5	4	8	10	6
Indices	1	2	3	4	5	6

Computing the last available location for a node becomes easy: you just need to know the last available spot in the heap. The children of a node i are in the array at indices $2i$ (left child) and $2i + 1$ (right child). Hence, finding the parent of a node of index i is also easy: we compute the result of the *integer division* $i/2$. Note that this is integer division, *not* floating point division. An example implementation of a heap using arrays is provided in an associated Java file.

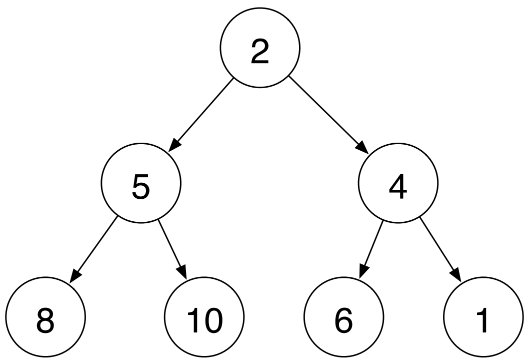
4 Heap algorithms

In this section we present pseudocode and discuss the complexity of heap operations; we also show some examples of these operations in action. The pseudocode is given assuming that these would be methods in a heap class.

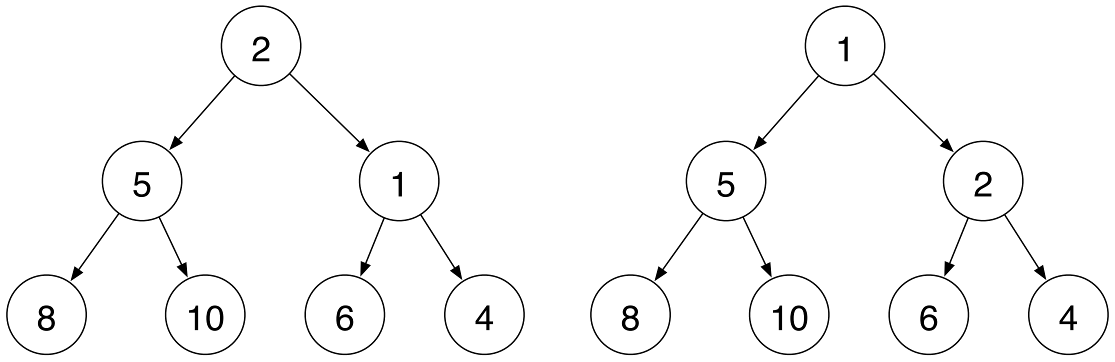
```
Algorithm findMin()
return root
```

Obviously, this is $O(1)$, like in the case of the other priority queue implementations.

Inserting an element in a heap is a bit more complex, because we want to maintain the heap properties. In order to do this, we will insert the element in the first available spot on the frontier (making sure that all nodes are still packed to the left). Then, we will re-organize the heap to restore the parent-child ordering property. As an example, consider inserting an element with key 1 in the heap above. We first put the element in the leftmost spot available on the last layer.



Now, the heap property is violated because 1 is smaller than its parent. So we will let 1 "bubble up" on the heap, by swapping it with its parent, until either it becomes the root, or the heap property is satisfied. The two successive states of the heap during this operation are depicted below.

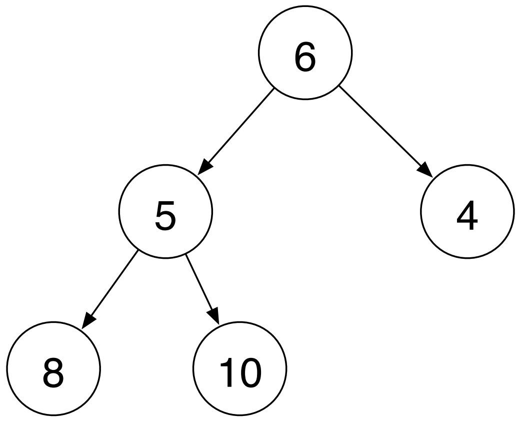


The pseudocode of the algorithm is as follows:

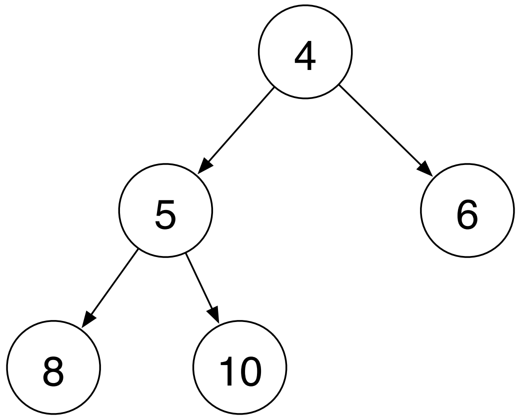
```
**Algorithm** insert (**Object** o, **int** priority)
**Input:** An object and the corresponding priority
**Output:** The object is inserted in the heap with the corresponding priority

lastNode ← getLast() //get the position at which to insert
lastNode.setKey(priority)
lastnode.setContent(o)
n← lastNode
while n.getParent()! = null and n.getParent().getKey() > priority
    swap(n,n.getParent())
```

The getLast() method is supposed to find the place at which a node can be inserted on the last level. If the heap is implemented as an array, this will just create an element at the last available position in the array, so this will be an $O(1)$ operation. Hence, in this case, the complexity is determined by the while loop, in which the "bubbling up" is implemented. How far can a node move? At most it can move all the way to the root, and the length of this path is $O(\log n)$ - the same as the height of the tree.

Removing a node similarly has to ensure that the two heap properties are retained. Hence, when removing, the first step is to replace the root of the tree with the rightmost element on the last level. However, this will likely break the heap property, which needs to be restored by "bubbling down" this element back towards the leaves. This can be done by repeatedly swapping the node with its child that has the *smallest key*, until both children have bigger keys than the node (in which case the heap property is now correct). For example, consider doing removeMin() on our example heap. First, element 2 will be removed, and 6 will be promoted to the root, resulting in the following tree:



Then, 6 will be swapped with 4 (its smallest child), resulting in the following tree:



At this point, the heap property has been restored, so no more changes are made.

The pseudocode of the algorithm is as follows:

Algorithm removeMin()
Output: The object associated with the minimum key is removed from the heap and returned

```
lastNode ← getLast()
value ← root.getContent()
swap(lastNode, root)
update(lastNode) //changes the index of the last available location in the array
n ← root
```

```
while n.getKey() > min(n.getLeft().getKey(), n.getRight().getKey())
    if n.getLeft().getKey() < n.getRight().getKey() then swap(n, n.getLeft())
    else swap(n, n.getRight())
return value
```

Note that this is pseudocode, and as such we have omitted some checks (e.g. if the children are null). Again, an element can only bubble down for a number of steps equal to the height of the tree so the complexity is $O(\log n)$.

To summarize, if we implement a priority queue using a heap, both insertion and removals will be $O(\log n)$ where n is the size of the queue. So, compared to sorted array and linked list implementations, we have gained efficiency for insertions (which were $O(n)$) and lost somme efficiency for removal operations (which were $O(1)$). However, if the application requires a lot of insertions, and/or if we think as the number of insertions and removal as roughly matched, overall this is a win: an insertion removal pair in a linear data structure costs $O(1) + O(n) = O(n)$, and now the cost is $O(\log n) + O(\log n) = O(\log n)$.

5 Using heaps for sorting

We can easily use the idea of a heap to sort an array of n integers. The approach is to insert all array elements in the heap, then repeatedly remove the minimum element. The pseudocode of the algorithm is as follows:

```
Algorithm HeapSort ($a$)
Input: An array $a$ of numbers
Output: The array is sorted

Heap h = new Heap($a$) // we want the heap to use $a$ as its internal memory
for $i = 1$ to $n$ do
    h.insert($a[i]$)
for $i = n$ to $1$ do
    $a[i] \leftarrow$ h.removeMin() //this orders the array in descending order
for $i = 1$ to $n/2$ do
    swap($a[i]$, $a[n - i]$) //this swaps elements so we can get increasing order again
```

The complexity of the 3 loops is $O(n \log n)$, $O(n \log n)$ and $O(n)$ respectively, so the sorting runs in $O(n \log n)$. If the heap works directly on the array a, the algorithm is also in place.