Priority Queues and Heaps

In this lecture, we will discuss another important ADT called *PriorityQueue*. Like stacks and queues, priority queues store arbitrary collections of elements. Recall that stacks have a Last-In First-Out (LIFO) access policy and queues have a First-In First-Out (FIFO) policy. Priority queues have a more complicated policy: each element stored in a priority queue has a *priority*, and the next element to be removed is the one with highest priority.

Priority queues have numerous applications. For example, they can be used to schedule jobs on a shared computer. Some jobs will be more urgent than others and thus get higher priorities. A priority queue keeps track of the jobs to be performed and their priorities. If a job is finished or interrupted, then one with highest priority will be performed next.

6.1 The *PriorityQueue* ADT

A *PriorityQueue* stores a collection of *elements*. Associated with each element is a *key*, which is taken from some linearly ordered set, such as the integers. Keys are just a way to represent the priorities of elements; larger keys mean higher priorities. In its most basic form, the ADT supports the following operations:

- insertItem(k, e): Insert element e with key k.
- maxElement$()$: Return an element with maximum key; an error occurs if the priority queue is empty.
- removeMax$()$: Return and remove an element with maximum key; an error occurs if the priority queue is empty.
- isEmpty$()$: Return TRUE if the priority queue is empty and FALSE otherwise.

Note that the keys have quite a different function here than they have in *Dictionary*. Keys in priority queues are used "internally" to determine the position of an element in the queue (they don't really mean anything when they stand-alone), whereas in dictionaries keys are the main means of accessing elements. Observe that in the *PriorityQueue* ADT, we cannot access elements directly using their keys, we can only take an element that currently has a maximum key (which is something that depends on relative order).

In many textbooks, you'll find priority queues in which a *smaller* key means higher priority and which consequently support methods minElement$()$ and removeMin$()$ instead of maxElement$()$ and removeMax$()$. This is a superficial distinction: by reversing all comparisons, it is easy to turn one version into the other. More formally, if $\leq$ is the order on our keys then we define the new order $\leq_{\text{rev}}$ by

$$k_1 \leq_{\text{rev}} k_2 \Leftrightarrow k_2 \leq k1.$$

It is a simple exercise to show that $\leq_{\text{rev}}$ is a total order if and only if $\leq$ is.

6.2 The search tree implementation

A straightforward implementation of the *PriorityQueue* ADT is based on binary search trees. If we store the elements according to the value of their keys, with the largest key stored in the rightmost internal vertex of the tree (remember that we have the habit of keeping all the leaf vertices empty), we can easily implement all methods of *PriorityQueue* such that their running time is $\Theta(h)$, where h is the height of the tree (except isEmpty$()$, which only requires time $\Theta(1)$). If we use AVL trees, this amounts to a running time of $\Theta(\lg(n))$ for insertItem(k, e), maxElement$()$ and removeMax$()$.

6.3 Abstract Heaps

A *heap* is a data structure that is particularly well-suited for implementing the *PriorityQueue* ADT. Although also based on binary trees, heaps are quite different from binary search trees, and they are usually implemented using arrays, which makes them quite efficient in practice. Before we explain the heap data structure, we introduce an abstract model of a heap that will be quite helpful for understanding the priority queue algorithms that we develop here.

We need some more terminology for binary trees: for $i \geq 0$, the ith *level* of a tree consists of all vertices that have distance i from the root. Thus the 0th level consists just of the root, the first level consists of the children of the root, the second level of the grandchildren of the root, etc. Recall that a *complete* Binary Search Tree of height h has $2^{h+1} - 1$ vertices in total (counting both the internal vertices and the leaves). We say that a binary tree of height h (where we have the usual notion of *left* and *right* children) is *almost complete* if levels $0, \ldots, h-1$ have the maximum possible number of vertices (i.e., level i has 2^i vertices), and on level h we have *some of* the possible vertices for that level, such that the vertices (leaves) of level h that *do* belong to the tree at level h appear in a contiguous group to the left of the vertices-positions that *do not* belong to the tree. Figure 6.1 shows an almost complete tree and two trees that are not almost complete.

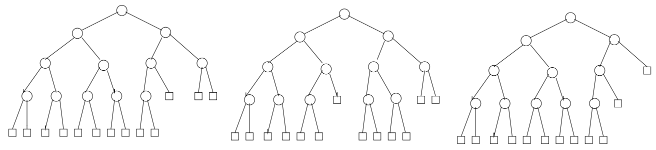

Figure 6.1. The first tree is almost complete; the second and third are not.

Now we have a theorem (similar to proof that AVL trees have $O(\lg n)$ height):

Theorem 6.2. *An almost complete binary tree with n internal vertices has height $\lfloor \lg(n) \rfloor + 1$.*

PROOF We first recall that a complete binary tree (a binary tree where all levels have the maximum possible number of vertices) of height h has $2^{h+1} - 1$ vertices in total (2^h leaves and $2^h - 1$ internal vertices). This can be proved by induction on h.

The number of internal vertices of an almost complete tree of height h is greater than the number of internal vertices of a complete tree of height $h-1$. Moreover it is at most the number of internal vertices of a complete tree of height h. Thus for the number n of vertices of an almost complete tree of height h we have

$$2^{h-1} \leq n \leq 2^h - 1.$$

For all n with $2^{h-1} \leq n \leq 2^h - 1$ we have $\lfloor \lg(n) \rfloor = h - 1$; this implies the theorem. $\square$

In our abstract model, a heap is an almost complete binary tree whose internal vertices store items such that the following *heap condition* is satisfied:

(H) For every vertex v except the root, the key stored at v is smaller than or equal to the key stored at the parent of v.

The *last vertex* of a heap of height h is the rightmost internal vertex in the hth level.

Insertion

To insert an item into the heap, we create a new last vertex and insert the item there. It may happen that the key of the new item is larger than the key of its parent, its grandparent, etc. To repair this, we bubble the new item up the heap until it has found its position.

```
Algorithm insertItem($k, e$)
1. Create new last vertex $v$.
2. while $v$ is not the root and $k > v.parent.key$ do
3.     store the item stored at $v.parent$ at $v$
4.     $v \leftarrow v.parent$
5. store $(k, e)$ at $v$
```

Algorithm 6.3

Consider the insertion algorithm 6.3. The correctness of the algorithm should be obvious (we could prove it by induction). To analyse the running time, let us assume that each line of code can be executed in time $\Theta(1)$. This needs to be justified, as it depends on the concrete way we store the data. We will do this in the next section. The loop in lines 2–4 is iterated at most h times, where h is the height of the tree. By Theorem 6.2 we have $h = \Theta(\lg(n))$. Thus the running time of insertItem is $\Theta(\lg(n))$.

Finding the Maximum

Property (H) implies that the maximum key is always stored at the root of a heap. Thus the method maxElement just has to return the element stored at the root of the heap, which can be done in time $\Theta(1)$ (with any reasonable implementation of a heap).

Removing the Maximum

The item with the maximum key is stored at the root, but we cannot easily remove the root of a tree. So what we do is replace the item at the root by the item stored at the last vertex, remove the last vertex, and then repair the heap. The repairing is done in a separate procedure called heapify (Algorithm 6.4). Applied to a vertex v of a tree such that the subtrees rooted at the children of v already satisfy the heap property (H), it turns the subtree rooted at v into a tree satisfying (H), without changing the shape of the tree.

```
Algorithm heapify($v$)
1. if $v.left$ is an internal vertex and $v.left.key > v.key$ then
2.     $s \leftarrow v.left$
3. else
4.     $s \leftarrow v$
5. if $v.right$ is an internal vertex and $v.right.key > s.key$ then
6.     $s \leftarrow v.right$
7. if $s \neq v$ then
8.     exchange the items of $v$ and $s$
9.     heapify($s$)
```

Algorithm 6.4

The algorithm for heapify works as follows: In lines 1–6, it defines s to be the largest of the keys of v and its children. Since the children of v may be leaves not storing items, some care is required. If $s = v$, the key of v is larger than or equal to the keys of its children. Thus the heap property (H) is satisfied at v. Since the subtrees rooted at the children are already heaps, (H) is satisfied at every vertex in these subtrees, so it is satisfied in the subtree rooted at v, and no further action is required. If $s \neq v$, then s is the child of v with the larger key. In this case, the items of s and v are exchanged, which means that now (H) is satisfied at v. (H) is still satisfied by the subtree of v that is not rooted by s, because this subtree has not changed. It may be, however, that (H) is no longer satisfied at s, because s has a smaller key now. Therefore, we have to apply heapify recursively to s. Eventually, we will reach a point where we call heapify(v) for a v that has no internal vertices as children, and the algorithm terminates.

The running time of heapify can be easily expressed in terms of the height h of v. Again assuming that each line of code requires time $\Theta(1)$ and observing that the recursive call is made to a vertex of height $h-1$, we obtain

$$T_{\text{heapify}}(h) = \Theta(1) + T_{\text{heapify}}(h-1).$$

Moreover, the recursive call is only made if the height of v is greater than 1, thus heapify applied to a vertex of height 1 only requires constant time. To solve the recurrence, we use the following lemma:

Lemma 6.5. *Let* $f : \mathbb{N} \to \mathbb{R}$ *such that for all* $n \geq 1$

$$f(n) = \Theta(1) + f(n-1).$$

Then $f(n)$ *is* $\Theta(n)$.

PROOF We first prove that $f(n)$ is $O(n)$. Let $n_0 \in \mathbb{N}$ and $c > 0$ such that $f(n) \leq c + f(n-1)$ for all $n \geq n_0$. Let $d = f(n_0)$. We claim that

$$f(n) \leq d + c\,(n - n_0) \tag{6.1}$$

for all $n \geq n_0$.

We prove (6.1) by induction on $n \geq n_0$. As the induction basis, we note that (6.1) holds for $n = n_0$.

For the induction step, suppose that $n > n_0$ and that (6.1) holds for $n-1$. Then

$$f(n) \leq c + f(n-1) \leq c + d + c\,(n - 1 - n_0) = d + c\,(n - n_0).$$

This proves (6.1).

To complete the proof that $f(n)$ is $O(n)$, we just note that

$$d + c\,(n - n_0) \leq d + c\,n = O(n).$$

The proof that $f(n)$ is $\Omega(n)$ can be done completely analogously and is left as an exercise. □

The Lemma should be intuitively obvious: we have $f(n) = \Theta(1) + f(n-1)$. Remember that $\Theta(1)$ means essentially a non-zero constant[1]. So, unwinding the recurrence, we have

$$f(n) = \underbrace{\Theta(1) + \Theta(1) + \cdots + \Theta(1)}_{n \text{ times}} + f(0).$$

Of course $f(0)$ is just a constant so it is clear that the expression on the r.h.s. is $\Theta(n)$. An experienced theoretician would not bother with the proof by induction, not because of laziness but because of genuine understanding. (The acid test for this is to be able to supply the proof if necessary.)

Applying Lemma 6.5 to the function $T_{\text{heapify}}(h)$ yields

$$T_{\text{heapify}}(h) = \Theta(h). \tag{6.2}$$

Finally, Algorithm 6.3 shows how to use heapify for removing the maximum.

Since the height of the root of the heap, which is just the height of the heap, is $\Theta(\lg(n))$ by Theorem 6.2, the running time of removeMax is $\Theta(\lg(n))$.

6.4 An array based implementation

The last section gives algorithms for a tree-based heap data structure. However, one of the reasons that heaps are so efficient is that they can be stored in arrays in a straightforward way. The items of the heap are stored in an array H, and the internal vertices of the tree are associated with the indices of the array by numbering them level-wise from the left to the right. Thus the root is associated with 0, the left child of the root with 1, the right child of the root with 2, the leftmost grandchild of the root with 3, etc. (see Figure 6.7). For every vertex v the left child

[1]Strictly speaking a function such as $1 + 1/(n+1)$ is $\Theta(1)$ but of course is not constant. However this function is bounded form below by 1 and from above by 2 and it does no harm to think of it as a constant in our context. The same applies to any $\Theta(1)$ function.

Algorithm removeMax()

```
1. $e \leftarrow root.element$
2. $root.item \leftarrow last.item$
3. delete $last$
4. heapify($root$)
5. **return** e;
```

Algorithm 6.6

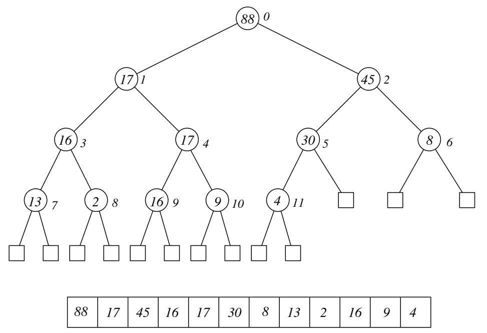

Figure 6.7. Storing a heap in an array. The small numbers next to the vertices are the indices associated with them.

of v is $2v+1$ and the right child of v is $2v+2$ (if they are internal vertices). The root is 0, and for every vertex $v > 0$ the parent of v is $\lfloor \frac{v-1}{2} \rfloor$. We store the number of items the heap contains in a variable $size$. Thus the last vertex is $size - 1$.

Observe that we only have to insert or remove the last element of the heap. Using dynamic arrays, this can be done in amortised time $\Theta(1)$. Therefore, using the algorithms described in the previous section, the methods of *PriorityQueue* can be implemented with the running times shown in Table 6.8. In many important priority queue applications we know the size of the queue in advance, so we can fix the array size when we set up the priority queue (no need for a dynamic array). In this situation, the running times $\Theta(\lg(n))$ for insertItem and removeMax are not amortised, but simple worst case running times.

Algorithm 6.9 gives the pseudocode to take an array H storing items and turn it into a heap. This method is very useful in applications:

To understand buildHeap, observe that $\lfloor \frac{n-2}{2} \rfloor$ is the maximum v such that $2v + 1 \leq n - 1$, i.e., the last vertex of the heap that has at least one child. Recall that if heapifyv is applied to a vertex v of a tree such that the subtrees rooted at the children of v already satisfy the heap property (H), this turns the subtree rooted at v into a tree satisfying (H). So the algorithm turns all subtrees of the tree into a heap in a bottom up fashion. Since subtrees of height 1 are automatically heaps,

method	running time
insertItem	$\Theta(\lg(n))$ amortised
maxElement	$\Theta(1)$
removeMax	$\Theta(\lg(n))$ amortised
isEmpty	$4\Theta(1)$

Table 6.8.

Algorithm buildHeap(H)

```
1. $n \leftarrow H.length$
2. for $v \leftarrow \lfloor \frac{n-2}{2} \rfloor$ downto $0$ do
3.     heapify($v$)
```

Algorithm 6.9

the first vertex where something needs to be done is $\lfloor \frac{n-2}{2} \rfloor$, the last vertex of height at least 2.

Straightforward reasoning shows that the running time of heapify() is

$$O(n \lg(n)),$$

because each call of heapify requires time $O(\lg(n))$. Surprisingly, a more refined argument shows that we can do better:

Theorem 6.10. *The running time of buildHeap is $\Theta(n)$, where n is the length of the array H.*

PROOF Let $h(v)$ denote the height of a vertex v and $m = \lfloor \frac{n-2}{2} \rfloor$. By 6.2, we obtain

$$\begin{aligned} T_{\text{buildHeap}}(n) &= \sum_{v=0}^{m} \Theta(1) + T_{\text{heapify}}(h(v)) \\ &= \Theta(m) + \sum_{v=0}^{m} \Theta\big(h(v)\big) \\ &= \Theta(m) + \Theta\Big(\sum_{v=0}^{m} h(v)\Big). \end{aligned}$$

Since $\Theta(m) = \Theta(n)$, all that remains to prove is that $\sum_{v=0}^{m} h(v) = O(n)$. Observe that

$$\begin{aligned} \sum_{v=0}^{m} h(v) &= \sum_{i=1}^{h} i \cdot \big(\text{number of vertices of height } i\big) \\ &= \sum_{i=0}^{h-1} (h-i) \cdot \big(\text{number of vertices on level } i\big) \end{aligned}$$

$$\le \sum_{i=0}^{h-1} (h-i) \cdot 2^i,$$

where $h = \lfloor \lg(n) \rfloor + 1$. The rest is just calculation. We use the fact that

$$\sum_{i=0}^{\infty} \frac{i+1}{2^i} = 4. \tag{6.3}$$

Then

$$\begin{aligned} \sum_{i=0}^{h-1} (h-i)\, 2^i &= n \frac{\sum_{i=0}^{h-1} (h-i)\, 2^i}{n} = n \sum_{i=0}^{h-1} (h-i) \frac{2^i}{n} \\ &\le n \sum_{i=0}^{h-1} (h-i)\, 2^{i-(h-1)} \qquad (\text{since } n \ge 2^{h-1}) \\ &= n \sum_{i=0}^{h-1} (i+1)\, 2^{-i} \\ &\le n \sum_{i=0}^{\infty} \frac{i+1}{2^i} \\ &= 4n. \end{aligned}$$

Thus $\sum_{v=0}^{m} h(v) \le 4n = O(n)$, which completes our proof. □

6.5 Further reading

If you have [GT], there is an entire chapter on “Priority Queues” (with details about Heaps).

If you have [CLRS], look at the chapter titled “Heapsort” (ignore the sorting for now).

Exercises

1. Where may an item with minimum key be stored in a heap?
2. Suppose the following operations are performed on an initially empty heap:
   - Insertion of elements with keys $20, 21, 3, 12, 18, 5, 24, 30$.
   - Two removeMax operations.

   Show how the data structure evolves, both as an abstract tree and as an array.
3. Show that for any n there is a sequence of n insertions into an initially empty heap that requires time $\Omega(n \lg n)$ to process.

   Remark: Compare this to the method buildHeap with its running time of $\Theta(n)$.
4. Let $S = \sum_{i=0}^{\infty} \frac{i+1}{2^i}$, we assume without proof that the series does indeed converge (this is easy to show). Prove that $S = 1 + \frac{1}{2}(S + \sum_{i=0}^{\infty} \frac{1}{2^i}) = 2 + \frac{1}{2}S$. Deduce that $S = 4$.