Priority Queues (Binary Heaps)

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

Priority Queue

A priority queue stores a set S of n integers and supports the following operations:

- `Insert(`e`)`: Adds a new integer to S.
- `Delete-min`: Removes the smallest integer in S, and returns it.

Example

Suppose that the following integers are inserted into an initially empty priority queue: 93, 39, 1, 26, 8, 23, 79, 54.

If next we perform a `Delete-Min`, the operation returns 1, after which $S = \{93, 39, 26, 8, 23, 79, 54\}$.

The next `Delete-Min` returns 8, and leaves $S = \{93, 39, 26, 23, 79, 54\}$.

Unlike an ordinary queue (which obeys FIFO), a priority queue guarantees that the elements always leave in ascending order, regardless of the order by which they are inserted.

Next we will implement a priority queue using a data structure called the binary heap to achieve the following guarantees:

- $O(n)$ space consumption
- $O(\log n)$ insertion time
- $O(\log n)$ delete-min time.

Binary Heap

Let S be a set of n integers. A binary heap on S is a binary tree T satisfying:

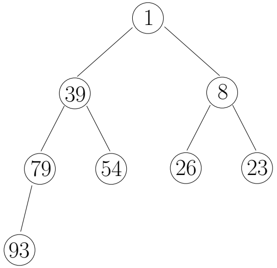
1. T is complete.
2. Every node u in T corresponds to a distinct integer in S—the integer is called the key of u (and is stored at u).
3. If u is an internal node, the key of u is smaller than those of its child nodes.

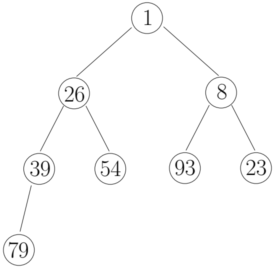
Note:

- Condition 2 implies that T has n nodes.
- Condition 3 implies that the key of u is the smallest in the subtree of u.

Example

Two possible binary heaps on $S = \{93, 39, 1, 26, 8, 23, 79, 54\}$:





The smallest integer of S must be the key of the root.

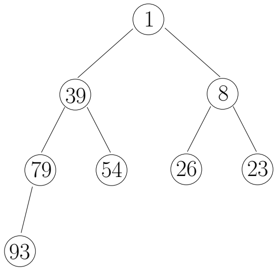
Insertion

We perform `insert`(e) on a binary heap T as follows:

1. Create a leaf node z with key e, while ensuring that T is a complete binary tree—notice that there is only one place where z can be added.
2. Set $u \leftarrow z$.
3. If u is the root, return.
4. If the key of $u >$ the key of its parent p, return.
5. Otherwise, swap the keys of u and p. Set $u \leftarrow p$, and repeat from Step 3.

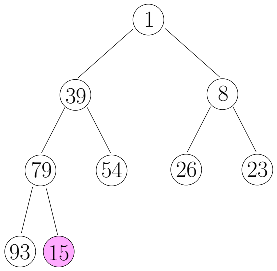
Example

Assume that we want to insert 15 into the binary heap below:



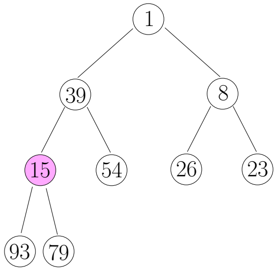
Example

First, add 15 as a new leaf, making sure that we still have a complete binary tree.



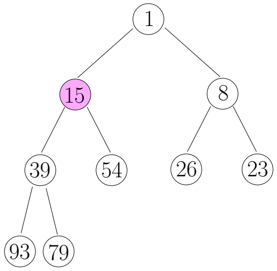
15 causes a violation by being smaller than its parent. This is fixed by a swap with its parent; see next.

Example



15 still causes a violation, necessitating another swap, as shown next.

Example



No more violation. Insertion complete.

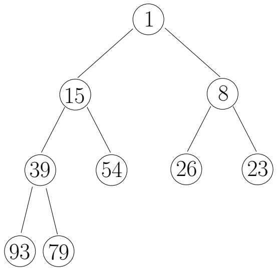
Delete-Min

We perform a delete-min on a binary heap T as follows:

1. Report the key of the root.
2. Identify the rightmost leaf z at the bottom level of T.
3. Delete z, and store the key of z at the root.
4. Set $u \leftarrow$ the root.
5. If u is a leaf, return.
6. If the key of $u <$ the keys of the children of u, return.
7. Otherwise, let v be the child of u with a smaller key. Swap the keys of u and v. Set $u \leftarrow v$, and repeat from Step 5.

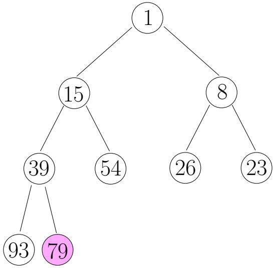
Example

Assume that we perform a delete-min from the binary heap below:



Example

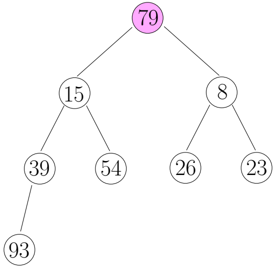
First, find the rightmost leaf at the bottom level, namely, 79.



Notice that the tree is still a complete binary tree after removing this leaf.

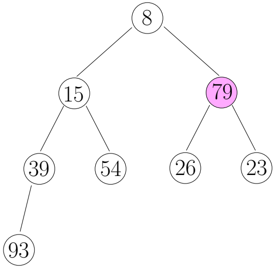
Example

Remove the leaf, but place the value 79 in the root.



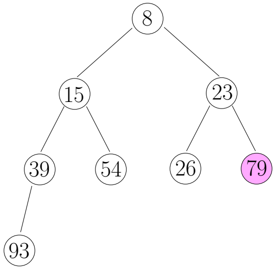
79 causes a violation by being greater than its children. This is fixed by swapping it with node 8, which is the child of the root with a smaller key. See the next slide.

Example



Node 79 still has a violation, causing another swap as shown next.

Example



The final tree after the delete-min.

How to Find the Rightmost Leaf at the Bottom Level

Before analyzing the running time of `insert` and `delete-min`, let us first consider a sub-problem:

> Given a complete binary tree T with n nodes, how to identify quickly the rightmost leaf node at the bottom level of T.

Our aforementioned algorithms depend on a fast solution to the above.

How to Find the Rightmost Leaf at the Bottom Level

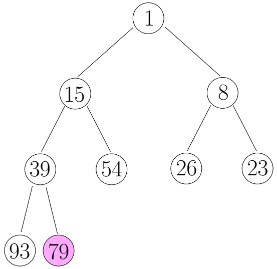
Next, we give a clever algorithm for solving the sub-problem in $O(\log n)$ time.

First, write the value of n in binary form. Think: How to do this in $O(\log n)$ time using only the atomic operations we are allowed?

Skip the most significant bit. We will scan the remaining bits from left to right, and descend as instructed by the next bit:

- If the next bit is 0, we go to the left child of the current node.
- Otherwise, go to the right child.

Example



Here $n = 9$, which is 1001 in binary. Skipping the first bit 1, we scan the remaining bits and descend accordingly:

- The 2nd leftmost bit is 0; so we turn left, and go to node 15.
- The 3rd leftmost bit is 0; so we turn left, and go to node 39.
- The 4th leftmost bit is 1; so we turn right, and go to node 79 (done).

Analysis of Insertion and Delete-Min

We are now ready to prove that our `insertion` and `delete-Min` algorithms finish in $O(\log n)$ time.

It suffices to point out the key facts:

- Step 1 of the insertion algorithm (Slide 7) and Step 2 of the delete-min algorithm (Slide 12) can be performed in $O(\log n)$ time, using our solution to the previous sub-problem.
- The rest of insertion ascends a root-to-leaf path, while that of delete-min descends a root-to-leaf path. The time is $O(\log n)$ in both cases.

Now officially we have reached the following conclusion. We can maintain a priority queue on a set S of elements such that:

- At any moment, the space consumption is $O(n)$, where $n = |S|$.
- An insertion can be processed in $O(\log n)$ time.
- A delete-min can be processed in $O(\log n)$ time.