The Binary Heap

A *binary heap* is a data structure that implements the abstract data type *priority queue*. That is, a binary heap stores a set of elements with a total order (that means that every two elements can be compared), and supports the following operations on this set:

- `findmin()` returns the smallest element;
- `deleteMin()` removes the smallest element from the set and returns it;
- `insert(el)` inserts a new element into the set;
- `len` returns the current size (number of elements) of the set.

A *complete binary tree* is a binary tree in which every level is full, except possibly the bottom level, which is filled from left to right as in Fig. 1. For a given number of elements, the shape of the complete binary tree

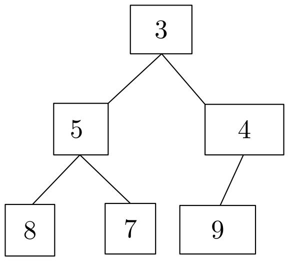

Figure 1: A complete binary tree.

is prescribed completely by this description.

A *binary heap* is a complete binary tree whose elements satisfy the *heap-order property*: no node contains an element less than its parent's element. Every subtree of a binary heap is also a binary heap, because every subtree is complete and satisfies the heap-order property.

Findmin. There are many different heaps possible for the same data, as there are many different ways to satisfy the heap property. However, all possible heaps have one feature in common: A smallest element must be in the root node. This implies that the `findmin` operation is very easy—it just returns the element from the root node.

Insert. To insert an element into the binary heap, we first have to add a node to the tree. Because we need to maintain that the tree is complete, there is only one possible place where we can add the node: If the lowest level is not yet full, we need to add the next open position on this level. If the lowest level is full, we need to add a new node at the leftmost position on the next level, see Fig. 2 for both cases.

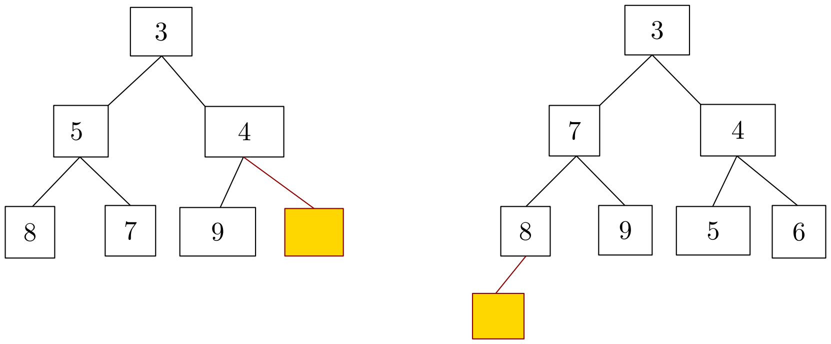

Figure 2: Two cases of adding a node.

Of course, the new element may violate the heap-order property. We correct this by having the element bubble up the tree until the heap-order property is satisfied. More precisely, we compare the new element x with its parent; if x is less, we exchange x with its parent, then repeat the procedure with x and its new parent.

Fig. 3 shows the four stages during the insertion of the new element 2.

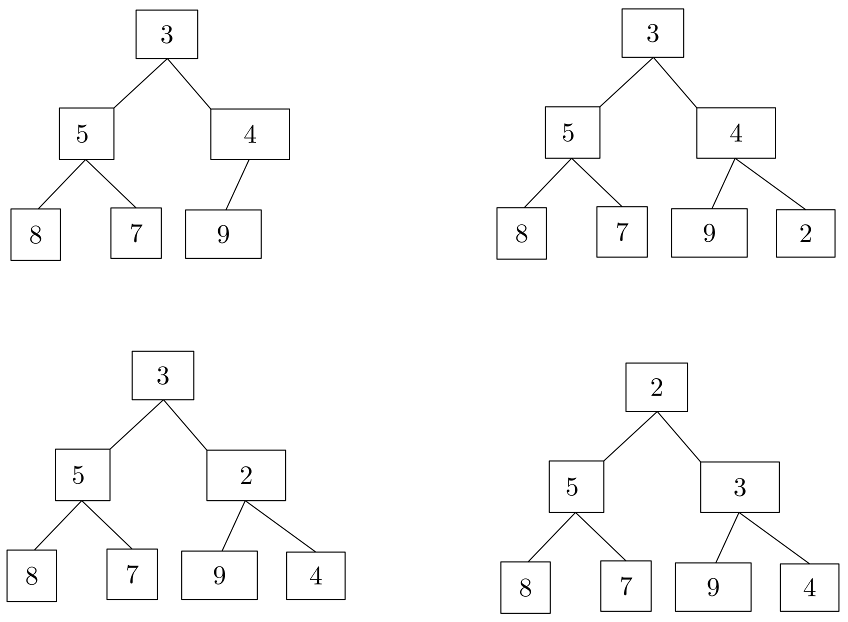

Figure 3: Adding element 2.

When we finish, is the heap-order property satisfied? Yes, if the heap-order property was satisfied before the insertion. Let's look at a typical exchange of x with a parent p during the insertion operation. Since the heap-order property was satisfied before the insertion, we know that $p \leqslant s$ (where s is x's sibling, see Fig. 4), $p \leqslant l$, and $p \leqslant r$ (where l and r are x's children). We only swap if $x < p$, which implies that $x < s$; after the swap, x is the parent of s. After the swap, p is the parent of l and r. All other relationships in the subtree rooted at x are maintained, so after the swap, the tree rooted at x has the heap-order property.

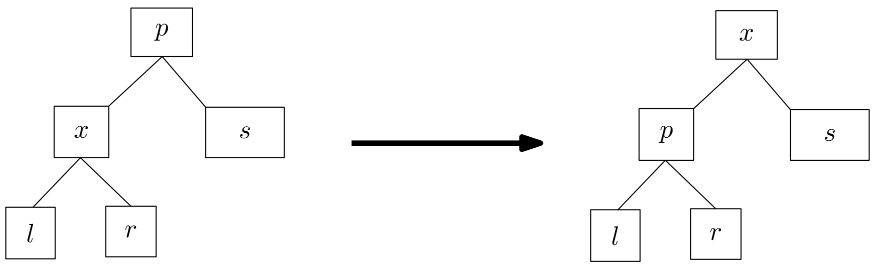

Figure 4: Maintaining the heap property during bubble-up.

Deletemin. The `deletemin()` operation starts by removing the entry at the root node and saving it for the return value. This leaves a gaping hole at the root. We fill the hole with the last entry in the tree (which we call x), so that the tree is still complete.

It is unlikely that x is the minimum element. Fortunately, both subtrees rooted at the root's children are heaps, and thus the new mimimum element is one of these two children. We bubble x down the heap as follows: if x has a child that is smaller, swap x with the child having the smaller element. Next, compare x with its new children; if x still violates the heap-order property, again swap x with the child with the smaller element. Continue until x is less than or equal to its children, or reaches a leaf.

Fig. 5 shows the stages in this process. Note that if in the original heap we replace 5 by 10, then 6 would not bubble down all the way to a leaf, but would remain on a higher level.

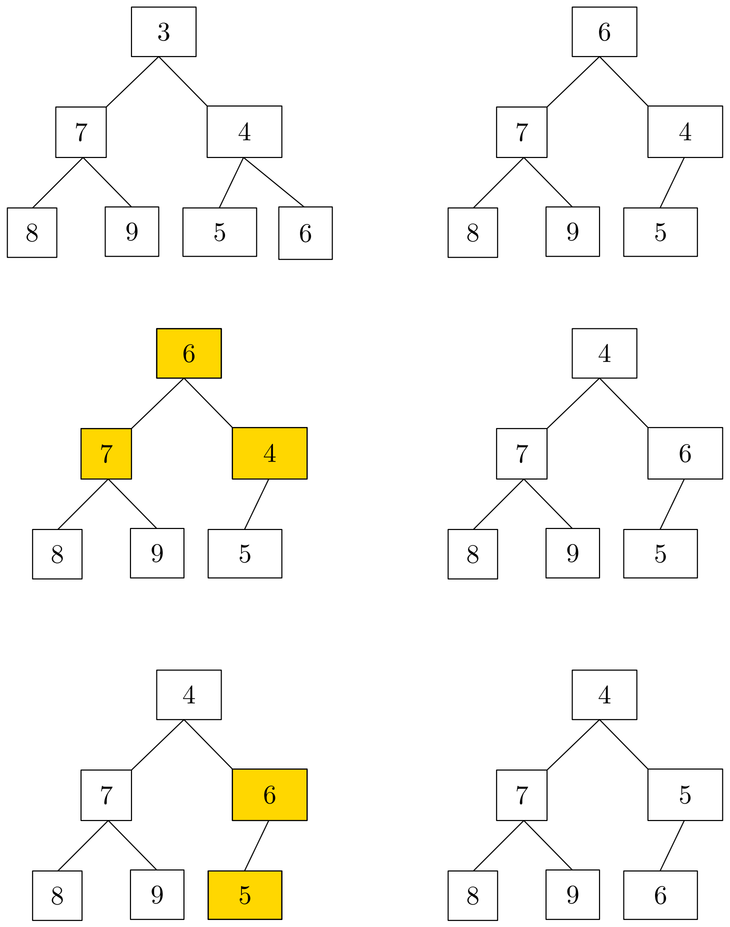

Figure 5: Removing the minimum element.

Array-based implementation. Of course a complete binary tree can be implemented as a linked data structure, with `Node` objects that store references to their parent and children. However, because the tree is complete, we can actually store it very compactly in an array or Python list. To do so, simply number the nodes of the tree from top to bottom, left to right, starting with one for the root, so that the nodes are numbered from 1 to n. We observe that the left child of node i has index $2i$ and the right child has index $2i+1$. The parent of node i has index $\lfloor i/2 \rfloor$. We will simply store node i in slot i of an array, and we can move from a node to its parent or its children by doing simple arithmetic on the index.

(We started numbering from 1 for convenience—the numbers are nicer. Of course this means that we waste slot 0 of the array—not a big problem, but it could be fixed by changing the numbering scheme slightly.)

Implementation. We now give a complete implementation in Python. Our `PriorityQueue` class can be constructed either by providing a Python list to be used, or by passing `None` (and the `PriorityQueue` class will create a Python list itself). We use the doubling-technique for growing the list. Initially, we create a list of some default capacity, and set `_size` to zero.

The methods __len__ and findmin() are very easy:

```
def __len__(self):
  return self._size

def findmin(self):
  if self._size == 0:
    raise ValueError("empty priority queue")
  return self._data[1]
```

The insert method first ensures there is enough space to add a node to the tree. For efficiency, we do not actually place x in the new node, but keep it empty (a "hole"). Then we bubble the hole up in the tree until x is no longer smaller than the parent of the hole, and finally place x in the hole.

```
def insert(self, x):
  if self._size + 1 == len(self._data):
    # double size of the array storing the data
    self._data.extend( [ None ] * len(self._data) )
  self._size += 1
  hole = self._size
  self._data[0] = x
  # Percolate up
  while x < self._data[hole // 2]:
    self._data[hole] = self._data[hole // 2]
    hole //= 2
  self._data[hole] = x
```

Note that we use a little trick here: We put x into `_data[0]`. This allows us to skip a test for `hole == 1` in the `while`-loop. Since the root has no parent, the loop should end there. However, since `hole//2 == 0` for `hole == 1`, in the root we will compare x to `_data[0]`, that is to itself, so the comparison will return `False` and the loop will end automatically.

Finally, the hardest operation is deletemin. It makes use of two internal methods. First, the method `_smaller_child(hole, in_hole)` checks whether one of the children of the node with index `hole` is smaller than the element `in_hole`. If so, it returns the index of the smaller element, otherwise it returns zero:

```
def _smaller_child(self, hole, in_hole):
  if 2 * hole <= self._size:
    child = 2 * hole
    if child != self._size and self._data[child + 1] < self._data[child]:
      child += 1
    if self._data[child] < in_hole:
      return child
  return 0
```

The internal method `_percolate_down(i)` bubbles down the element in node i:

```
def _percolate_down(self, i):
  in_hole = self._data[i]
  hole = i
  child = self._smaller_child(hole, in_hole)
  while child != 0:
    self._data[hole] = self._data[child]
    hole = child
    child = self._smaller_child(hole, in_hole)
  self._data[hole] = in_hole
```

With these internal methods,`deleteMin()` is rather easy:

```
def deletemin(self):
  min_item = self.findmin()         # raises error if empty
  self._data[1] = self._data[self._size]
  self._size -= 1
  self._percolate_down(1)
  return min_item
```

Analysis. Since the heap is always a complete binary tree, its height is $O(\log n)$ at any time, where n is the current size of the set. It follows that the operations `deletemin` and `insert` take time $O(\log n)$.

The operations `len` and `findmin` take constant time.

Now imagine we have n elements and we want to insert them into a binary heap. Calling `insert` n times would take total time $O(n \log n)$. Perhaps surprisingly, we can actually do better than this by creating an array for the n elements, throwing the elements into this array in some arbitrary order, and then running the following method:

```
def build_heap(self):
  for i in range(self._size // 2, 0, -1):
    self._percolate_down(i)
```

The method works backward from the last internal node (non-leaf node) to the root node, in reverse order in the array or the level-order traversal. When we visit a node this way, we bubble its entry down the heap using `_percolate_down` as in `deletemin`.

We are making use of the fact that if the two subtrees of a node i are heaps, then after calling `_percolate_down(i)` the subtree rooted at node i is a heap. By induction, this implies that when `_build_heap` has completed, the entire tree is a binary heap.

The running time of `_build_heap` is somewhat tricky to analyze. Let h_i denote the height of node i, that is, the length of a longest path from node i to a leaf. The running time of `_percolate_hown(i)` is clearly $O(h_i)$, and so the running time of `_build_heap` is $O(\sum_{i=1}^{\lfloor n/2 \rfloor} h_i)$.

How can we bound this sum? Here is a simple and elegant argument. For simplicity, let's assume that $n = 2^{h+1} - 1$, so the tree has height h and is perfect—all leaves are on level h. Clearly adding nodes on the bottom level can only increase the value of the sum.

Now, for every node i there is a path P_i of length h_i from i to a leaf as follows: From i, go down to its right child. From there, always go down to the left child until you reach a leaf. It is easy to see that for two different nodes i and j, the paths P_i and P_j do not share any edge. It follows that the total number of edges of all the paths P_i, for $1 \leqslant i \leqslant n$, is at most the number of edges of the tree, which is $n - 1$. So we have $\sum_{i=1}^{n} h_i \leqslant n - 1$, and so the running time of `_build_heap` is $O(n)$.

Heap Sort

Any priority queue can be used for sorting:

```
def pqsort(a):
  pq = PriorityQueue()
  for e in a:
    pq.insert(e)
  for i in range(len(a)):
    a[i] = pq.deletemin()
```

The binary heap is particularly well suited to implement a sorting algorithm, because we can use the array containing the input data to implement the binary heap. We therefore obtain an *in-place* sorting algorithm, which sorts the data inside the array.

Initially, we run `_build_heap` to put the objects inside the array into heap order. We then remove them one by one using `deletemin`. Every time an element is removed from the heap, the heap needs one less array slot. We can use this slot to store the element that we just removed from the heap, and obtain the following algorithm:

```
def heapsort(a):
  heap = PriorityQueue(a)
  for i in range(len(a)-1, 1, -1):
    a[i] = heap.deletemin()
```

Here is an example run:

```
>>> a = [0, 13, 2, 9, 55, 5, 17, 1, 89, 45]
>>> heapsort(a)
>>> a
[0, 89, 55, 45, 17, 13, 9, 5, 2, 1]
```

The element in slot 0 of the array was not sorted, since our `PriorityQueue` implementation doesn't use this slot. This could be fixed by changing the indexing of the array.

Also, the array is sorted backwards, because we have to fill it starting at the last index. This can be fixed by using a *max-heap* instead of a *min-heap*.