13 Priority Queues and Heaps

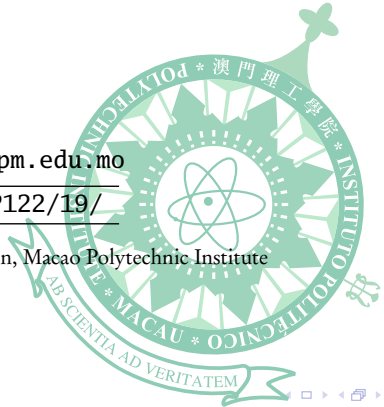
Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

March 11, 2019



Outline

Priority Queues

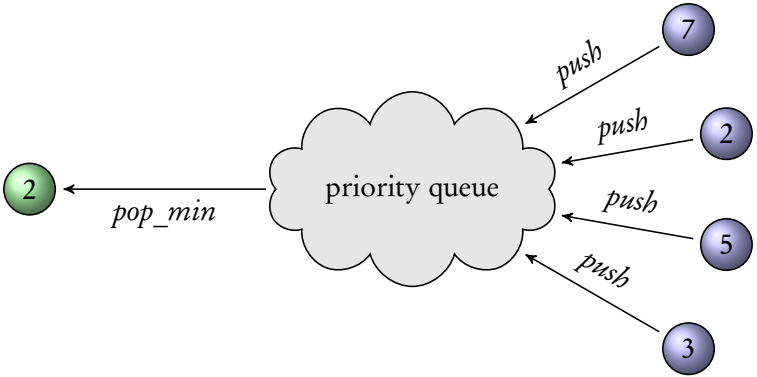
A priority queue is a collection of items with priorities, where the item with the highest priority is called the minimum item. It is an ADT that provides the following operations:

- *push*(*self*, *x*) — pushes a new item *x* into the priority queue;
- *pop_min*(*self*) — finds, returns and removes the minimum item from the priority queue;
- *get_min*(*self*) — finds, returns but does *not* remove the minimum item from the priority queue;
- *__bool__*(*self*) — returns `True` if the priority queue is not empty, otherwise `False`.

Some applications of priority queues:

- Printer queues.
- Task schedulers.
- Timers.

Priority Queues — Illustrated



Defining a Total Order ($\leqslant$) between Items

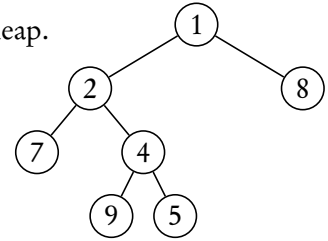
- A class must support at least the ($\leqslant$) comparison for its objects to be items of priority queues.
- In Python, this operation is defined by the *__le__*(*self*, *other*) special method.
- For example, we may define the lexicographical order between any two singly linked lists as follows.

```
1  class LnLs:
2      def __le__(self, other):
3          p, q = self.head, other.head
4          while p is not None and q is not None:
5              if p.elm != q.elm:
6                  return p.elm <= q.elm
7              p, q = p.nxt, q.nxt
8          return p is None
9      ...
```

The Heap Property of Binary Trees

For a binary tree, the heap property is that, for every node x in the tree, x is less than or equal to its children (if any). A binary tree with heap property is called a *heap*. Obviously, we have the following facts.

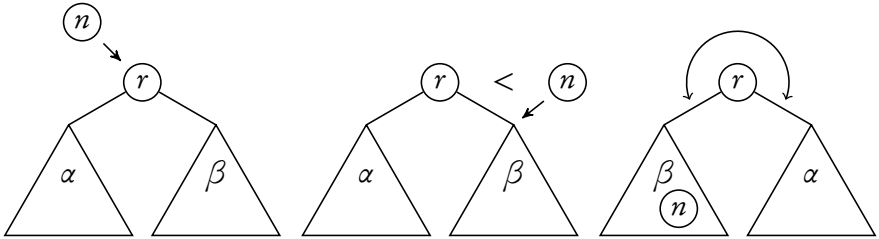
- The root is the minimum node of all the tree nodes in a heap.
- The root can be accessed immediately.
- All the subtrees of a heap are also (sub)heaps.
- The nodes along any path in a heap are ordered.



Because every node is less than or equal to its children, the heap is also called a *min-heap*. It is possible to use the reverse heap order, that is, every node is greater than or equal to its children, in this case, the heap is called a *max-heap*.

Perfectly Balanced Heaps (Insertion)

- We employ a perfectly balanced binary tree to achieve minimal tree depth.
- Recall that we always insert to one side and swap sides to keep the balance. We should also maintain the heap property while inserting.
  1. Compare the new item with the root.
  2. Choose the smaller one to be the new root.
  3. Insert the bigger one to the right subtree.
  4. Swap the two subtrees on the way back.



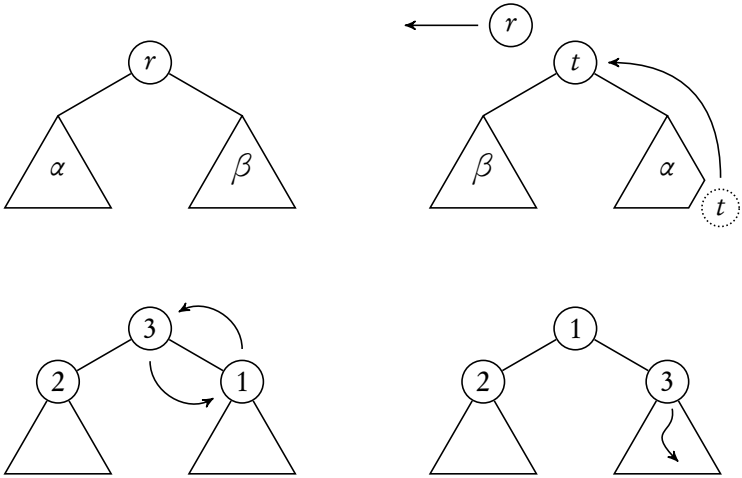
Balanced Heap Insertion

```
1  def insert_node(root, p):
2      if root is None:
3          p.left = p.right = None
4          return p
5      else:
6          lh, rh = root.left, root.right
7          if not root.elm <= p.elm:
8              root, p = p, root
9          root.left, root.right = insert_node(rh, p), lh
10         return root
```

Balanced Deletion of the Root

- Having removed the root, we need to relocate a node from one of the remaining subtrees to the root.
- We always take a node t from the left side and swap sides.
- We put the taken node to the root position, and *sift it down* to a proper location, maintaining the heap property.
  1. Compare t with the two children.
  2. If t is the smallest, then let it stay at the root and stop.
  3. Otherwise,
     - take the smaller child as the new root,
     - put t down to the root of the subtree originally containing the smaller child,
     - then recursively sift t down the subtree.
- Sifting a node down can be regarded as merging the node with two (sub)heaps altogether to form a new heap.

Balanced Deletion of Root — Illustrated



Deleting the Leftmost Leaf

- Since every left subtree is no less than the corresponding right subtree, we detach the leftmost leaf and swap subtrees along the left path to keep the tree balanced.
- If the heap has only one node, the root will change to `None` after the deletion. We return two nodes as a pair, one is the new root and the other is the deleted node.

```
1 def delete_leftmost(root):
2     if root.left is None:
3         return (None, root)
4     else:
5         sub, leftmost = delete_leftmost(root.left)
6         root.left, root.right = root.right, sub
7         return (root, leftmost)
```

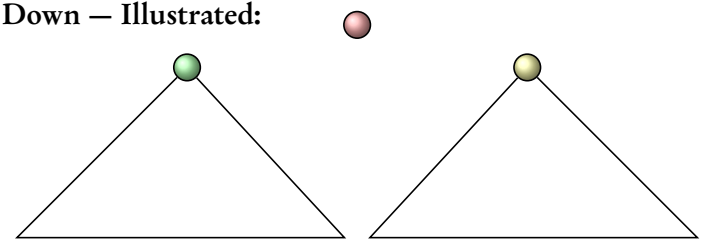
Sifting a Node Down

```
1  def sift_down(p, lh, rh):
2      if lh is None:
3          p.left = p.right = None
4          return p
5      else:
6          mh = lh if rh is None or lh.elm <= rh.elm else rh
7          if p.elm <= mh.elm:
8              p.left, p.right = lh, rh
9              return p
10         else:
11             h = sift_down(p, mh.left, mh.right)
12             mh.left, mh.right = (h, rh) if mh is lh else (lh, h)
13             return mh
```

Deleting the Root

```
1 def delete_root(root):
2     p, leftmost = delete_leftmost(root)
3     return None if p is None else sift_down(leftmost, p.left, p.right)
```

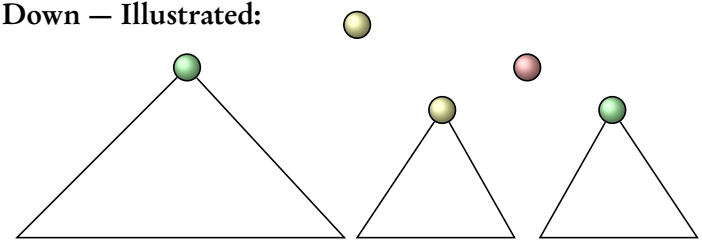
Sifting Down — Illustrated:



Deleting the Root

```
1  def delete_root(root):
2      p, leftmost = delete_leftmost(root)
3      return None if p is None else sift_down(leftmost, p.left, p.right)
```

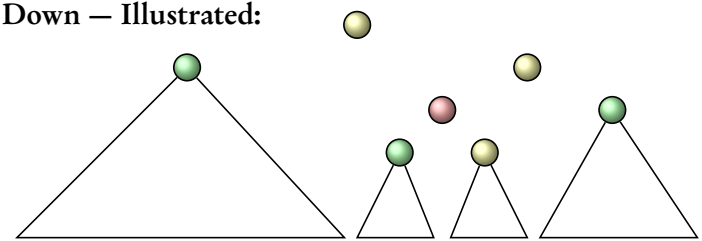
Sifting Down — Illustrated:



Deleting the Root

```
1 def delete_root(root):
2     p, leftmost = delete_leftmost(root)
3     return None if p is None else sift_down(leftmost, p.left, p.right)
```

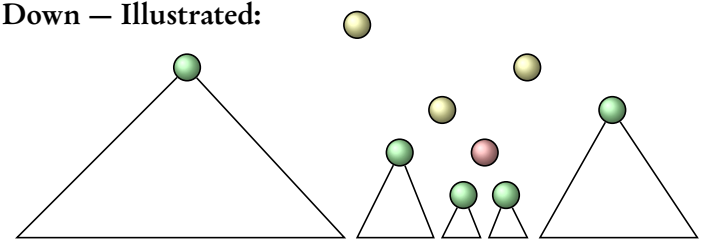
Sifting Down — Illustrated:



Deleting the Root

```
1 def delete_root(root):
2     p, leftmost = delete_leftmost(root)
3     return None if p is None else sift_down(leftmost, p.left, p.right)
```

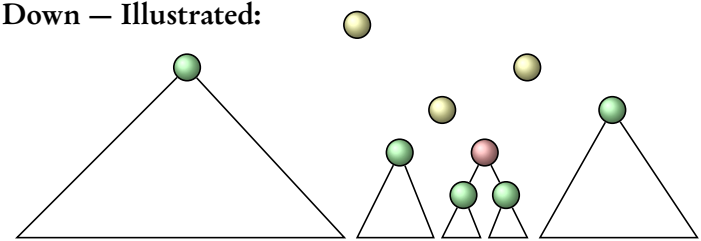
Sifting Down — Illustrated:



Deleting the Root

```
1 def delete_root(root):
2     p, leftmost = delete_leftmost(root)
3     return None if p is None else sift_down(leftmost, p.left, p.right)
```

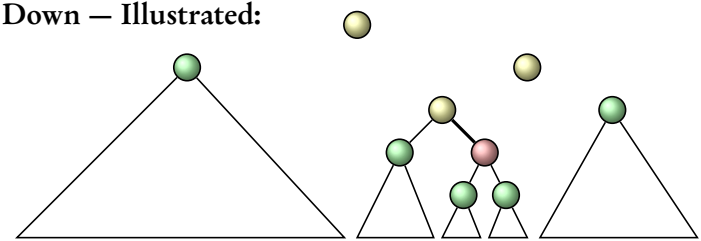
Sifting Down — Illustrated:



Deleting the Root

```
1  def delete_root(root):
2      p, leftmost = delete_leftmost(root)
3      return None if p is None else sift_down(leftmost, p.left, p.right)
```

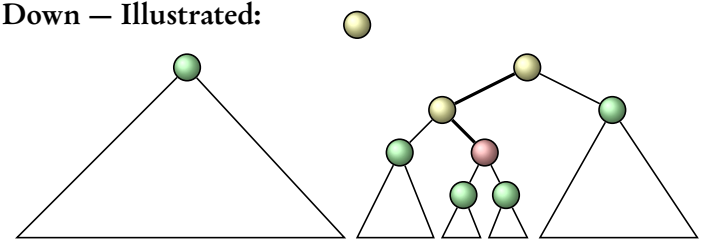
Sifting Down — Illustrated:



Deleting the Root

```
1  def delete_root(root):
2      p, leftmost = delete_leftmost(root)
3      return None if p is None else sift_down(leftmost, p.left, p.right)
```

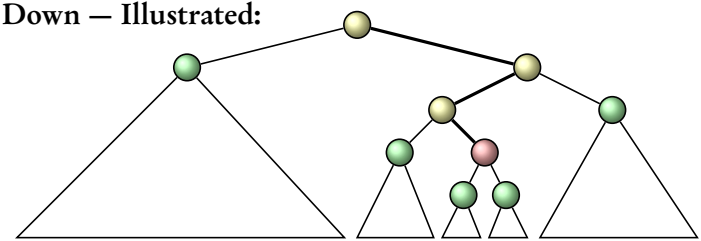
Sifting Down — Illustrated:



Deleting the Root

```
1  def delete_root(root):
2      p, leftmost = delete_leftmost(root)
3      return None if p is None else sift_down(leftmost, p.left, p.right)
```

Sifting Down — Illustrated:



Implementing Priority Queues Based on Heaps

```
1  class BalHeap:
2      def __init__(self):
3          self.root = None
4      def __bool__(self):
5          return self.root is not None
6      def push(self, x):
7          self.root = insert_node(self.root, Node(x))
8      def pop_min(self):
9          x = self.get_min()
10         self.root = delete_root(self.root)
11         return x
12     def get_min(self):
13         if not self:
14             raise IndexError
15         return self.root.elm
```

Analysis

For a heap of n items, we only need the amount of auxiliary space proportional to the *height* of the heap for the recursive calls of the insertion and sifting down.

- $\mathcal{O}(h)$ auxiliary space, where h is the height of the heap.

We count the number of item comparisons for the time complexity.

- An insertion at most compares the new node to all the nodes on a path from the root to some leaf.
- A sifting down also moves a node along a path from top to bottom, in each step, there are two comparisons, one between the two children, the other between the node and the smaller child.

Since the maximum height of a perfectly balanced binary tree is h when the size is between 2^h and $2^{h+1} - 1$, the *push* and *pop_min* of such a heap of size n all take only $\mathcal{O}(\log n)$ time and auxiliary space.