COMP122/20 - Data Structures and Algorithms

13 Priority Queues and Heaps

Instructor : Ke Wei 〖柯韋〗

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

March 13, 2020

Outline

Textbook §9.1 – 9.2, 9.3.1 – 9.3.2.

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:

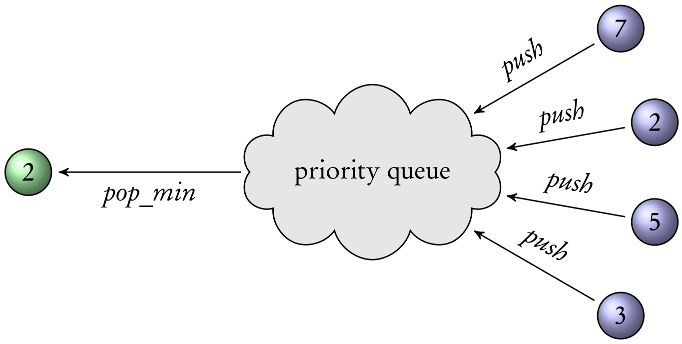
- *push*(*self*, *x*) — pushes a new item *x* into the priority queue;
- *pop_min*(*self*) — finds, returns and removes the minimum item from the priority queue;
- *get_min*(*self*) — finds, returns but does *not* remove the minimum item from the priority queue;
- *__bool__*(*self*) — returns `True`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

Some applications of priority queues:

- Printer queues.
- Task schedulers.
- Timers.

Priority Queues — Illustrated

Unlike stacks or queues, the outgoing order of the items in a priority queue does not depend on the incoming order, it is determined by the priorities.



Defining a Total Order (⩽) between Items

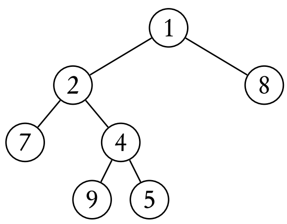
- A class must support at least the (⩽) comparison for its objects to be items of priority queues.
- In Python, this operation is defined by the *__le__*(*self*, *other*) special method.
- For example, we may define the lexicographical order between any two singly linked lists as follows.

```
class LnLs:
    def __le__(self, other):
        p, q = self.head, other.head
        while p is not None and q is not None:
            if p.elm != q.elm:
                return p.elm <= q.elm
            p, q = p.nxt, q.nxt
        return p is None
    ...
```


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.

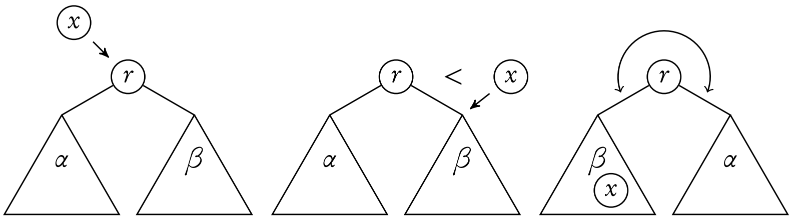
- The root is the minimum of all the tree nodes in a heap.
- The root can be accessed immediately.
- All the subtrees of a heap are also (sub)heaps. *The heap property is therefore recursive.*
- The nodes along any path in a heap are ordered.



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. Recursively insert the bigger one to the right subtree.
  4. Swap the two subtrees on the way back.



Balanced Heap Insertion

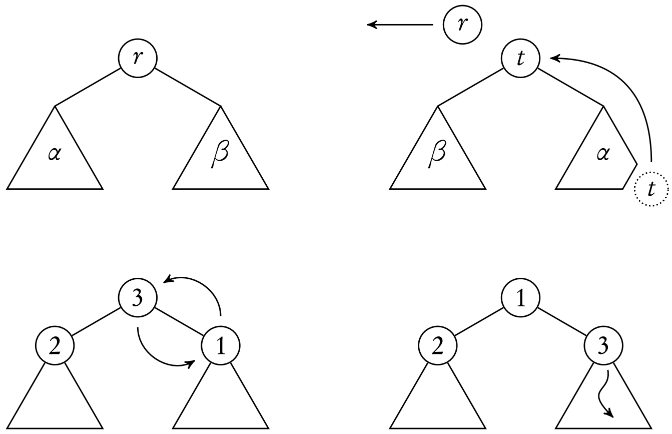
- The *insert_heap* function inserts a new element *x* into a perfectly balanced heap, and returns the new root.
- The only addition to the *insert_bal* function is the exchange of the root element with the new element when necessary.
- It shows that, besides the perfect balance property, the function also maintains the heap property.

```python
def insert_heap(root, x):
    if root is None:
        return Node(x)
    else:
        if not root.elm <= x:
            root.elm, x = x, root.elm
        root.left, root.right = insert_heap(root.right, x), root.left
        return root
```


Balanced Deletion of the Root

- When we have 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 to maintain the perfect balance.
- We put the taken node *t* to the root position, and *sift it down* to a proper location, to recover 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.
- To sifting a node down can be regarded as to merge 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.

```
def delete_leftmost(root):
    if root.left is None:
        return (None, root)
    else:
        root.left, root.right, leftmost = root.right, *delete_leftmost(root.left)
        return (root, leftmost)
```

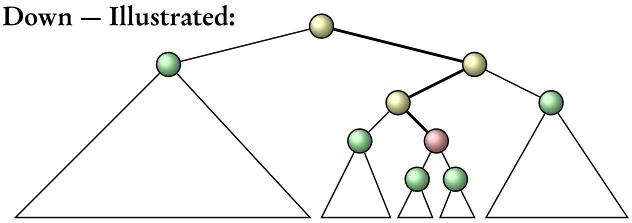

Sifting an Element Down

```
def sift_down(root):
    if root.left is not None:
        if root.right is None or root.left.elm <= root.right.elm:
            if not root.elm <= root.left.elm:
                root.elm, root.left.elm = root.left.elm, root.elm
                sift_down(root.left)
        else:
            if not root.elm <= root.right.elm:
                root.elm, root.right.elm = root.right.elm, root.elm
                sift_down(root.right)
```

Deleting the Root

```python
def delete_root(root):
    root, leftmost = delete_leftmost(root)
    if root is not None:
        root.elm = leftmost.elm
        sift_down(root)
    return root
```

Sifting Down — Illustrated:



Implementing Priority Queues Based on Heaps

```python
class BalHeap:
    def __init__(self):
        self.root = None
    def __bool__(self):
        return self.root is not None
    def push(self, x):
        self.root = insert_heap(self.root, x)
    def pop_min(self):
        x = self.get_min()
        self.root = delete_root(self.root)
        return x
    def get_min(self):
        if not self:
            raise IndexError
        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.