COMP122/20 - Data Structures and Algorithms

14 Array-Based 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 16, 2020

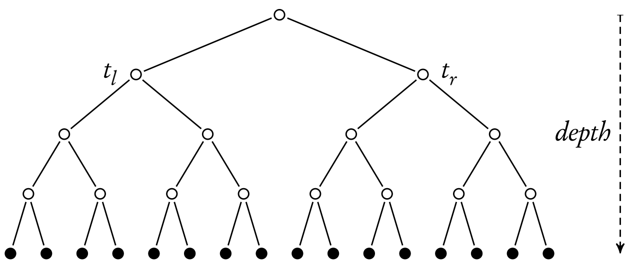
Outline

Textbook §8.3.2, 9.3.3 – 9.3.7.

Full Binary Trees

A full binary tree is

- either *empty*, or
- a binary tree whose two subtrees — t_l, t_r — are also full binary trees of the same size.

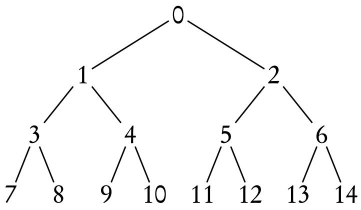


A full binary tree of depth d has size $2^{d+1} - 1$.

Numbering Nodes in a Full Binary Tree

We number the nodes in a full binary tree from top to bottom, left to right.

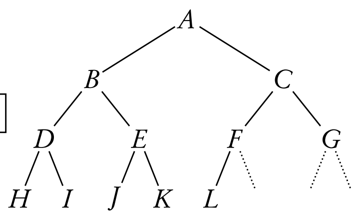
- Given that the root is numbered 0, the left most node of depth d is numbered $2^d - 1$.
- The left child and right child of a node numbered i are numbered $2i+1$ and $2i+2$.
- The parent of a node numbered i is numbered $\left\lfloor \frac{i-1}{2} \right\rfloor$.



Array-Based Complete Binary Trees

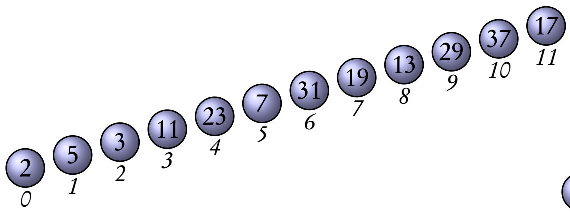
- We may store the nodes of a full binary tree in an array-based list, each taking a position according to their numbers, that is, the node numbered i is stored as element $a[i]$ in list a.
- On the other hand, an array of elements can be structured as a full binary tree, if its size is $2^{d+1} - 1$. For an array of arbitrary size n, by removing those nodes with numbers greater than $n-1$ from the full binary tree, we have a *complete binary tree*.

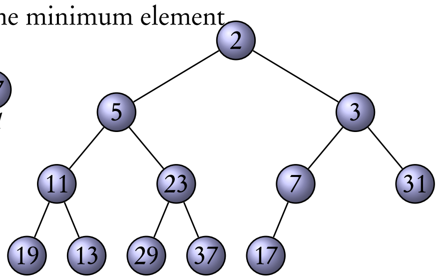
A	B	C	D	E	F	G	H	I	J	K	L



Array-Based Heaps

If a complete binary tree also has the heap property, then such a heap can be stored in an array-based list a. Obviously, the root $a[0]$ contains the minimum element.





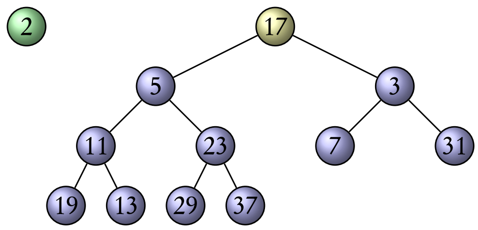

```
class ArrayHeap:
    def __init__(self):
        self.a = []
```

```
    def __bool__(self):
        return bool(self.a)
```

Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array-based list.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Sifting Down

The function *sift_down* takes a starting vacant position i and the element x to sift down, finds the sifting path and moves the elements along the path, finally puts x at the end position.

```
def sift_down(a, i, x):
    n = len(a)
    j = 2*i+1   # index of left child
    if j < n:   # at least a child exists
        if j+1 < n and not a[j] <= a[j+1]: # right child exists and is smaller
            j += 1 # index of right child
        if not x <= a[j]:   # x must be put down further
            a[i] = a[j]
            return sift_down(a, j, x)
    a[i] = x
```


The *pop_min* Method

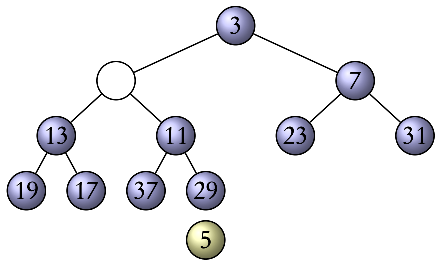
We use *sift_down* to help recover the heap property after the deletion of the root in the *pop_min* method.

```
    def pop_min(self):
        x, last = self.a[0], self.a[-1]

        del self.a[-1]
        if self.a:
            sift_down(self.a, 0, last)

        return x
```

The *push* Method and Sifting Up

We can only append an element x to the end of an array-based list — the bottom-right most of a heap — efficiently, we need to relocate it to recover the heap property.

- Such an append leaves no hole in the heap, maintaining the complete binary tree.
- We *sift up* x to a proper location where the parent is no greater, maintaining the heap property.



The *sift_up* Function and the *push* Method — Code

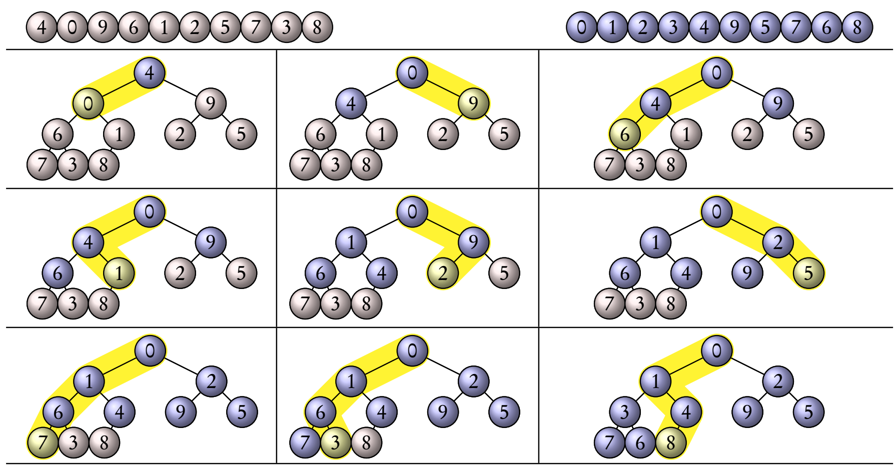
The *sift_up* method rotates x with the ancestors greater than x. We don't need to check with the size of the heap, for the sifting-up goes towards the root, whose index is `0`.

```
def sift_up(a, i, x):
    if i > 0:
        j = (i-1)//2   # index of parent
        if not a[j] <= x:
            a[i] = a[j]
            return sift_up(a, j, x)
    a[i] = x
```

```
def push(self, x):
    self.a.append(None)
    sift_up(self.a, len(self.a)-1, x)
```

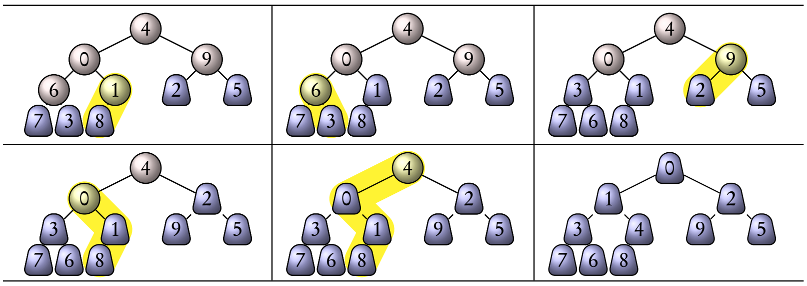
We insert an element by appending it to the heap and sifting it up.

Building a Heap — Heapify-ing — by Insertion



Heapifying by Merging

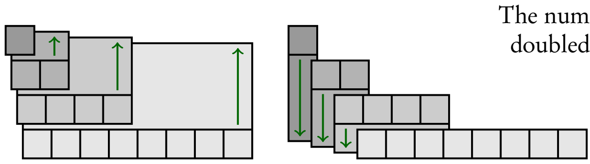
We may even build the heap by sifting down, starting from the bottom up to the top. Sifting a node down can be regarded as merging the node with its two sub-heaps into a bigger heap.




```
def heapify(a):
    for i in range((len(a)-2)//2, -1, -1):
        sift_down(a, i, a[i])
```


Which Is Better — Up or Down?

- When heapifying by sifting-up, the deeper levels get the larger multipliers;
- When heapifying by sifting-down, the deeper levels get the smaller multipliers.



The number of nodes is doubled at each level.

heapifying by sifting-up   heapifying by sifting-down (T_n)

$$T_n = \frac{n}{4} + \frac{2n}{8} + \frac{3n}{16} + \frac{4n}{32} + \cdots$$
$$= \left(\frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} + \cdots\right) + \left(\frac{n}{8} + \frac{2n}{16} + \frac{3n}{32} + \cdots\right) = \frac{n}{2} + \frac{T_n}{2}.$$

Analysis

For a heap of n elements, we only need a fix amount of auxiliary space for sifting up and sifting down.

- $\mathcal{O}(1)$ auxiliary space.

We count the number of element comparisons.

- A sifting up moves along a path from bottom to top, in each step, there is only one comparison with the parent.
- A sifting down moves along a path from top to bottom, in each step, there are two comparisons, one between the children, one with the selected child.

Since the maximum depth of a complete binary tree is d when the number of nodes is between 2^d and $2^{d+1}-1$, the *push* and *pop_min* of a heap of size n all take only $\mathcal{O}(\log n)$ time. For sifting-down heapification, the number of moves is at most

$$\frac{n}{4} + \frac{2n}{8} + \frac{3n}{16} + \cdots = n \in \mathcal{O}(n).$$