COMP122/20 - Data Structures and Algorithms

Appendix.1 Leftist 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

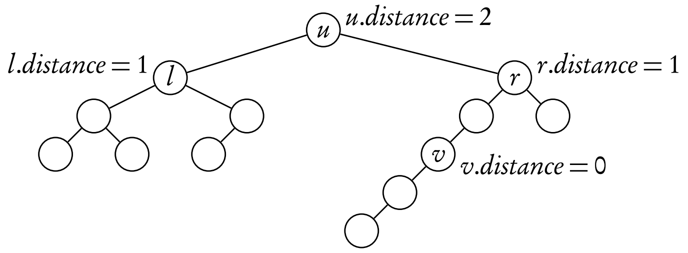
April 7, 2020

Outline

1. Leftist Heaps
2. Mergeable Heaps
3. Insertion and Removal
4. Implementation
5. Analysis

Distance of a Node

- For a binary tree, we define the *distance* of a node to be the length of the path from the node to its nearest descendant which has at least one empty subtree.
- The distance of the tree is the distance of its root.
- The distance of an empty tree is defined as −1.



- By the notion of distance, we want to know how quick we can reduce to an empty subtree.

Leftist Heaps

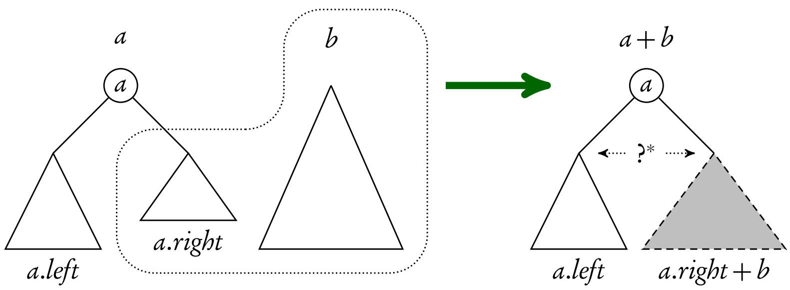
- To process the trees easier, just like in the perfectly balanced trees, we strengthen the condition so that we know which way to go for the nearest empty subtree.
- For every node, if the distance of its left subtree is greater than or equal to that of its right subtree, then the tree is called a *leftist tree*.
- Since we are walking to the right, the distance is now the length of the path to the *right-most* node.
- If a leftist tree also admits the heap property, then it is a leftist heap.

Mergeable Heaps

- We can prove that a leftist tree of size n has a distance less than $\log n$.
- The main operation that a leftist heap provides efficiently is $merge(a, b)$, where a and b are two leftist heaps — to merge two leftist heaps into one leftist heap.
  1. Obviously, if one of the heaps is empty, the result is just the other heap.
  2. Otherwise, we split the heap whose root has the smaller element (suppose it is heap a) into three parts — the root node a, the subheaps $a.left$ and $a.right$.
  3. We make node a the new root, and $a.left$ one of the subheap; we then recursively merge $a.right$ with the other heap b, and make the result the other subheap.
  4. We swap the subheaps when necessary so that the right subheap has the smaller distance.
- At each recursive call, we reduce to the right subheap of either a or b, so the number of steps can't be more than $a.distance + b.distance$, thus in logarithmic time.

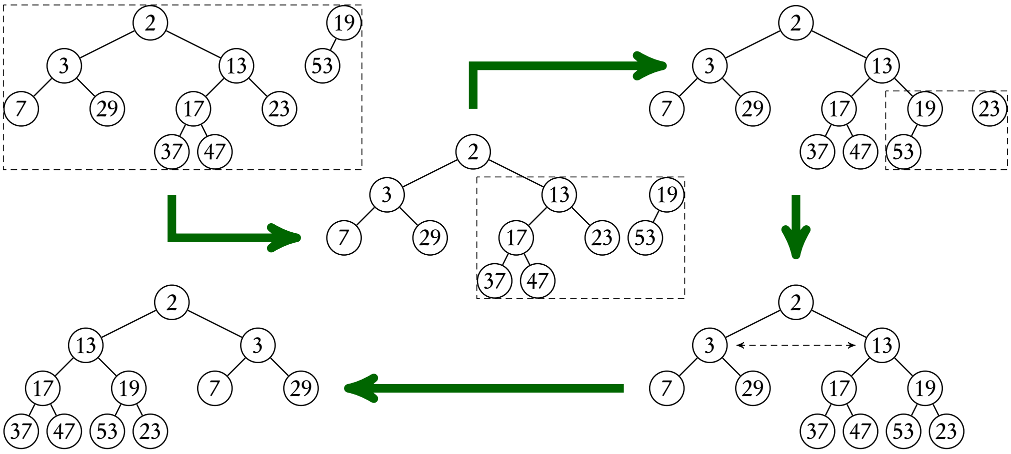
Merging Two Leftist Heaps — Illustrated

Suppose $a.elm \leqslant b.elm$.



* We swap the subheaps of the new heap when the distance of $a.left$ is less than the distance of $a.right + b$, to recover the leftist tree property.

Merging Two Leftist Heaps — Example



Insertion and Removal of Leftist Heaps

Insertion of a new node:

- make the new node a singleton heap — a heap having only the root node,
- merge it with the old heap.

Removal of the root node:

- merge the left and right (sub)heaps,
- return the old root as the removed node.

The *Node* Class and the Preorder Traversal

```python
class Node:
    def __init__(self, elm):
        self.elm = elm
        self.distance = 0
        self.left = self.right = None
```

```python
def distance(r):
    return r.distance if r else -1
```

```python
def preorder_leftist(r):
    while r is not None:
        yield r.elm
        yield from preorder_leftist(r.right)
        r = r.left
```

- Since we define the distance of an empty tree as −1, and we cannot store this in `None`, we define a function to uniformly return the distance of a tree, whether empty or not.
- The traversal of a leftist tree must not go recursively to the left, because it can be very deep. We loop along the leftmost path and recursively traverse the right subtrees.

Merging Two Leftist Heaps

```
def merge_leftists(a, b):
    if a is None:
        return b
    elif b is None:
        return a
    else:
        if not a.elm <= b.elm:
            a, b = b, a
        a.right = merge_leftists(a.right, b)
        if distance(a.left) < distance(a.right):
            a.left, a.right = a.right, a.left
        a.distance = distance(a.right)+1
        return a
```


Implementing Priority Queues Based on Leftist Heaps

```
class Leftist:
    def __init__(self):
        self.root = None
    def __bool__(self):
        return self.root is not None
    def push(self, x):
        self.root = merge_leftists(self.root, Node(x))
    def pop_min(self):
        x = self.get_min()
        self.root = merge_leftists(self.root.left, self.root.right)
        return x
    def get_min(self):
        if not self:
            raise IndexError
        return self.root.elm
```


The Relation between Distance and Size

Let s be the distance of a leftist tree that has n nodes. We prove by induction on n that

$$n \geqslant 2^{s+1} - 1.$$

- Base case: for $n = 0$, the empty leftist tree has distance $s = -1$, therefore $n \geqslant 0 = 2^{-1+1} - 1$.
- Induction step: for $n \geqslant 1$, let s_l and n_l be the distance and size of the left subtree, and s_r and n_r the distance and size of the right subtree, respectively. We have

$$\begin{aligned}
n &= n_l + n_r + 1 && \text{[by binary tree]}\\
&\geqslant (2^{s_l+1} - 1) + (2^{s_r+1} - 1) + 1 && \text{[by induction hypothesis]}\\
&= 2^{s_l+1} + 2^{s_r+1} - 1 && \text{[by arithmetic]}\\
&\geqslant 2^{s_r+1} + 2^{s_r+1} - 1 = 2^{s_r+1+1} - 1 && \text{[by leftist tree } (s_l \geqslant s_r)\text{, arithmetic]}\\
&= 2^{s+1} - 1. && \text{[by distance of leftist tree } (s = s_r + 1)\text{]}
\end{aligned}$$