CSE 326 Data Structures

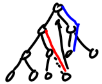
CSE 326 : Dave Bacon

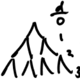
Priority Queues : Floyd's Algorithm, D heaps, Leftist heaps, ...

Homework 2 due friday

Binary Min Heaps (summary)

- **insert**: percolate up. O(log N) time.
- **deleteMin**: percolate down. O(log N) time.





depth = d

$\#$ nodes @ d $= 2^d$

$N = 2^{d+1} - 1$

$S = 1 + 2 + \cdots + 2^d$

$= 1 + 2(1 + 2 + \cdots + 2^d) - 2^{d+1}$

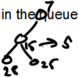
$S = 1 + 2S - 2^{d+1}$ $\quad S = 2^{d+1} - 1$

$d+1 = \log N+1$

$d = O(\log N)$

Other Priority Queue Operations

- **decreaseKey**
  - given a pointer to an object in the queue, reduce its priority value



Solution: change priority and percolate up

O(log N)

- **increaseKey**
  - given a pointer to an object in the queue, increase its priority value

15 → 29

Why do we need a *pointer*? Why not simply data value?

Hard to Find

Solution: change priority and percolate down

More Priority Queue Operations

- **Remove(objPtr)**
  - given a pointer to an object in the queue, remove it

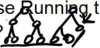
  Solution: set priority to negative infinity, percolate up to root and deleteMin

root

15 → -∞

Worst case Running time for all of these:

FindMax? O(N)



ExpandHeap – when heap fills, copy into new space. O(N)

More Priority Queue Operations

5, 15, 25, 16, 39   N

- **buildHeap**

  Naïve solution:

  But in 1 by 1

etc.

Running time:

insert O(log N)

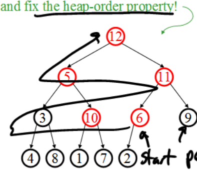
$\longrightarrow$ O(N log N)

Can we do better?

BuildHeap: Floyd's Method

12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!



start

leaf
leaves
do not
need
to be
perc – down

Buildheap pseudocode

```
private void buildHeap() {
  for ( int i = currentSize/2; i > 0; i-- )
     percolateDown( i );
}
```

O(log N)

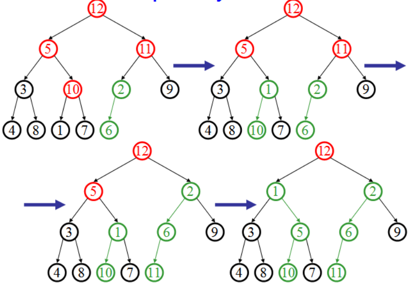
N

↓

runtime:

1st guess $\underline{O(N \log N)}$??

BuildHeap: Floyd's Method

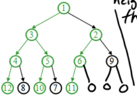


Finally… sum of the heights of the nodes.

height	#
0	2^h
1	2^{h-1}
$\vdots$	
h	1

$\sum_{i=0}^{h} i\,2^{h-i}$ ← # at height

height



height = h

nodes $2^{h+1}-1$

$$\sum_{i=0}^{h}\sum_{j=0}^{i} 2^{h-j}$$

$$\to 2^{h+1}-2-h$$

$O(N)$

vs

$\cancel{O(N\log N)}$

$2^{(h-1)}$

runtime:

$$S = 2^h\cdot 0 + 2^{h-1}\cdot 1 + 2^{h-2}\cdot 2 + 2^{h-3}\cdot 3 + \cdots + 1\cdot h$$

$$2S = 2^h\cdot 1 + 2^{h-1}\cdot 2 + 2^{n-2}\cdot 3 + \cdots + 2h$$

$$S = 2^h + 2^{h-1} + 2^{h-2} + \cdots + 2 - h$$

$$2^{h+1} - 1 - 1 \quad + h$$

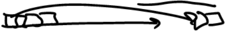
$$S = 2^{h+1} - 2 - h$$

Facts about Heaps

fast

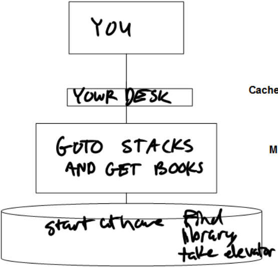
Observations:

- finding a child/parent index is a multiply/divide by two
- operations jump widely through the heap
- each percolate step looks at only two new nodes
- inserts are at least as common as deleteMins



Realities:

- division/multiplication by powers of two are equally fast
- looking at only two new pieces of data: bad for cache!
- with huge data sets, disk accesses dominate

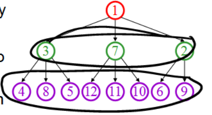


Cycles to access:

CPU	1
Cache (L1, L2)	1,2,3,4
Memory	200
Disk	10^6

A Solution: *d*-Heaps

- Each node has *d* children
- Still representible by array
- Good choices for *d*:
  - – (choose a power of two for efficiency)
  - – fit one set of children in a cache line
  - – fit one set of children on a memory page/disk block



12	1	3	7	2	4	8	5	12	11	10	6	9

$\log_d N$

Operations on *d*-Heap

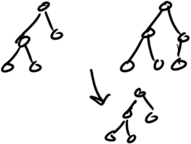
- Insert : runtime = $\underline{O(\log_d N)}$

- deleteMin: runtime = $O(\underset{\nearrow}{d} \log_d N)$

Does this help insert or deleteMin more?

One More Operation

- Merge two heaps. Ideas?



How to merge

Leftist Heaps → Friday

Leftist Heaps

Idea:

Focus all heap maintenance work in one small part of the heap

Leftist heaps:

1. Most nodes are on the left
2. All the merging work is done on the right

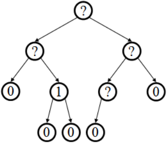
Definition: Null Path Length

null path length (npl) of a node x = the number of nodes between x and a null in its subtree

OR

npl(x) = min distance to a descendant with 0 or 1 children

- *npl*(null) = -1
- *npl*(leaf) = 0
- *npl*(single-child node) = 0



Equivalent definitions:

1. *npl(x)* is the height of largest complete subtree rooted at *x*
2. *npl(x)* = 1 + min{*npl*(left(x)), *npl*(right(x))}

Leftist Heap Size

- A leftist tree with r nodes on the right path must have at least 2^r-1 nodes
- Induction
- r=1
- Assume true for 1,..,r-1. Then leftist heap size r:

Leftist Heap Properties

- Heap-order property
  - parent's priority value is ≤ to childrens' priority values
  - <u>result</u>: minimum element is at the root

- Leftist property
  - For every node *x*, *npl*(left(*x*)) ≥ *npl*(right(*x*))
  - <u>result</u>: tree is at least as "heavy" on the left as the right

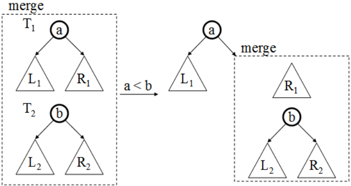
Are leftist trees…
complete?
balanced?

Merge two leftist heaps (basic idea)

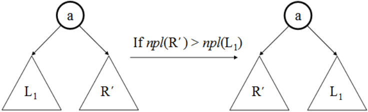
- Put the smaller root as the new root,
- Hang its left subtree on the left.
- Recursively merge its right subtree and the other tree.

Merging Two Leftist Heaps

- merge(T_1,T_2) returns one leftist heap containing all elements of the two (distinct) leftist heaps T_1 and T_2



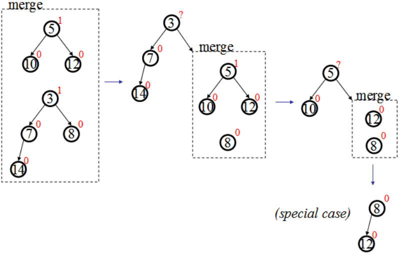
Leftist Merge Continued



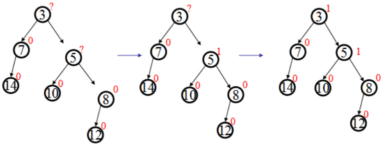
$R' = \text{Merge}(R_1, T_2)$

runtime:

Leftist Merge Example

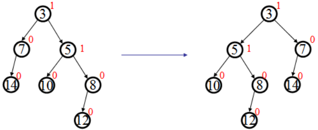


Sewing Up the Leftist Example



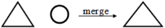
Done?

Finally…(Leftist)

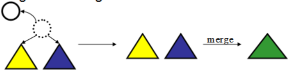


Operations on Leftist Heaps

- merge with two trees of total size n: O(log n)
- insert with heap size n: O(log n)
  - pretend node is a size 1 leftist heap
  - insert by merging original heap with one node heap



- deleteMin with heap size n: O(log n)
  - remove and return root
  - merge left and right subtrees



Random Definition: Amortized Time

am·or·tized time:

Running time limit resulting from "writing off" expensive runs of an algorithm over multiple cheap runs of the algorithm, usually resulting in a lower overall running time than indicated by the worst possible case.

If M operations take total O(M log N) time,
amortized time per operation is O(log N)

Difference from **average time:**