CSE 326 Data Structures

CSE 326 : Dave Bacon

Priority Queues : Leftist Heaps,
Skew Heaps, Binomial Queues

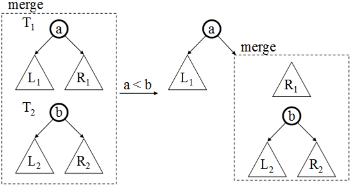
Logistics

- Updated Due Dates
  - Project 2, Phase A, due Friday, January 26
  - Homework 3, due Monday, January 29 in class 11:59

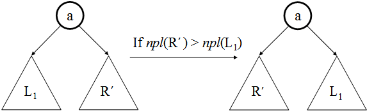
- Project 2A
  - Work in partners! Easier for you, good experience for “real” world. See webpage for instructions…don’t forget to email about your partnership (or, less desirably that you’re working alone.)

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



If $npl(R')$ > $npl(L_1)$

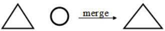
$R' = \text{Merge}(R_1, T_2)$

runtime: $O(\log n)$

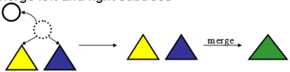
right path r $\geq 2^{r+1} - 1$ nodes

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:** ≠ each step is log N !

Average – still might be bad sequences

Skew Heaps

- simple to implement
- no npl

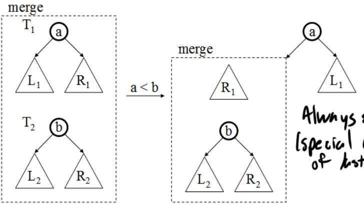
Problems with leftist heaps

- extra storage for npl
- extra complexity/logic to maintain and check npl
- right side is “often” heavy and requires a switch

Solution: skew heaps

- “blindly” adjusting version of leftist heaps
- merge *always* switches children when fixing right path
- amortized time for: merge, insert, deleteMin = O(log *n*)
- however, worst case time for all three = O(*n*)

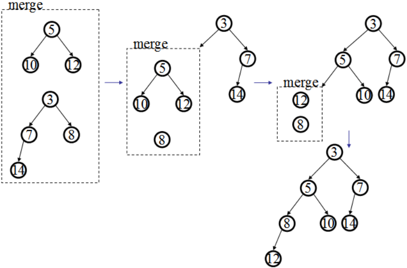
Merging Two Skew Heaps



Always swap
(special case
of last)

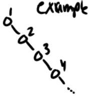
Only one step per iteration, with children *always* switched

Example


merge
5
10
12
3
7
8
14
merge
5
10
12
8
3
7
14
merge
12
8
3
5
7
10
14
3
5
7
8
10
14
12

Skew Heap Code

example



```
void merge(heap1, heap2) {
  case {
     heap1 == NULL: return heap2;
     heap2 == NULL: return heap1;
     heap1.findMin() < heap2.findMin():
          temp = heap1.right;
          heap1.right = heap1.left;
          heap1.left = merge(heap2, temp);
          return heap1;
     otherwise:
          return merge(heap2, heap1);
  }
}
```

Runtime Analysis: Worst-case and Amortized

- No worst case guarantee on right path length!
- All operations rely on merge

  ⇒ worst case complexity of all ops = $O(n)$

- Will do amortized analysis later in the course (see chapter 11 if curious)
- Result: *M* merges take time *M* log *n*

  ⇒ amortized complexity of all ops = $O(\log n)$

Comparing Heaps

- Binary Heaps

memory efficient (no pointers)

fast + simple

ins: $O(\log N)$ Avg $O(1)$

del: $O(\log N)$

merge: bad $O(N)$

- d-Heaps

fancy binary heaps

ins: $O(\log_d n)$

del: $O(d \log_d n)$

slower math

Still scope for improvement!

- Leftist Heaps

fast merge, ins, del

$O(\log n)$

complicated

memory cost (links, npl)

- Skew Heaps

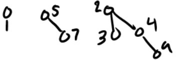
less storage

good amortized times

simple

Yet Another Data Structure: Binomial Queues

- Structural property
  - Forest of binomial trees with at most one tree of any height



What's a forest?

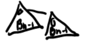
bunch of trees

What's a binomial tree?

- Order property
  - Each binomial tree has the heap-order property

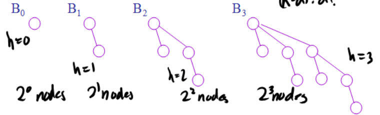
The Binomial Tree, B_h

- B_h has height *h* and exactly 2^h nodes
- B_h is formed by making B_{h-1} a child of another B_{h-1}



- Root has exactly *h* children
- Number of nodes at depth d is binomial coeff. $\binom{h}{d}$
  - Hence the name; we will *not* use this last property

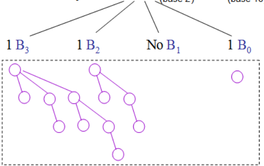
$\frac{n!}{(n-d)!\,d!}$



Binomial Queue with *n* elements

Binomial Q with *n* elements has a *unique* structural representation in terms of binomial trees!

Write *n* in binary: $n = 1101_{\text{(base 2)}} = 13_{\text{(base 10)}}$



1 B_3 1 B_2 No B_1 1 B_0

Properties of Binomial Queue

- At most one binomial tree of any height

- *n* nodes ⇒ binary representation is of size ?
  ⇒ deepest tree has height ?
  ⇒ number of trees is ?

All $O(\log n)$

Define: height(forest F) = $\max_{\text{tree T in F}}$ { height(T) }

Binomial Q with *n* nodes has height Θ(log *n*)

$\lfloor \log_2 n+1 \rfloor$

Operations on Binomial Queue

- Will again define *merge* as the base operation
  - insert, deleteMin, buildBinomialQ will use merge

- Can we do increaseKey efficiently? decreaseKey?

- What about findMin?

Merging Two Binomial Queues

Essentially like adding two binary numbers!

```
1.  Combine the two forests
2.  For $k$ from 1 to maxheight {
      a.  $m \leftarrow$ total number of $B_k$'s in the two BQs
      b.  if m=0:  continue;
      c.  if $m$=1:  continue;
      d.  if $m$=2:  combine the two $B_k$'s to form a $B_{k+1}$
      e.  if $m$=3:  retain one $B_k$ and combine the
                     other two to form a $B_{k+1}$
}
```

of 1's

$0+0 = 0$

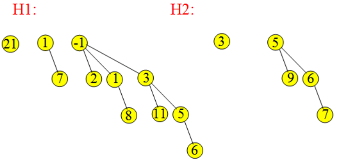
$1+0 = 1$

$1+1 = 0+c$

$1+1+c = 1+c$

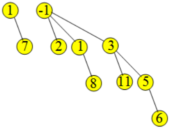
Claim: When this process ends, the forest has at most one tree of any height

Example: Binomial Queue Merge

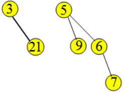


Example: Binomial Queue Merge

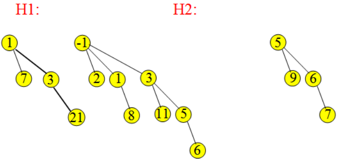
H1:



H2:



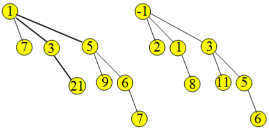
Example: Binomial Queue Merge



Example: Binomial Queue Merge

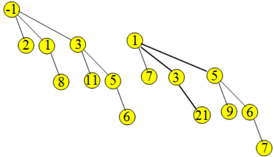
H1:

H2:



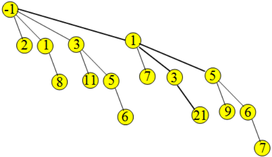
Example: Binomial Queue Merge

H1: H2:



Example: Binomial Queue Merge

H1: H2:



Complexity of Merge

Constant time for each height

Max height is: log *n*

$\Rightarrow$ worst case running time = $\Theta(\qquad)$

Insert in a Binomial Queue

Insert(*x*): Similar to leftist or skew heap

runtime

Worst case complexity: same as merge

O(   )

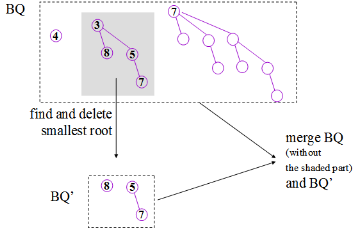
Average case complexity: O(1)

Why??   *Hint: Think of adding 1 to 1101*

deleteMin in Binomial Queue

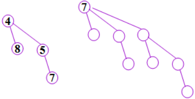
Similar to leftist and skew heaps….

deleteMin: Example



deleteMin: Example

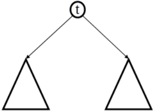
Result:



runtime:

Tree Calculations

Recall: height is max number of edges from root to a leaf

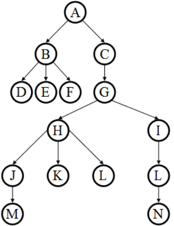


Find the height of the tree...

runtime:

Tree Calculations Example

How high is this tree?

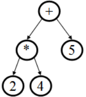


More Recursive Tree Calculations: Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

Three types:

- Pre-order: Root, left subtree, right subtree
- In-order: Left subtree, root, right subtree
- Post-order: Left subtree, right subtree, root



(an expression tree)

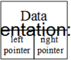
Traversals

```
void traverse(BNode t){
  if (t != NULL)
    traverse (t.left);
    print t.element;
    traverse (t.right);
  }
}
```

Binary Trees

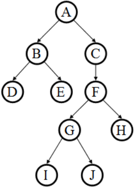
- Binary tree is
  - a root
  - left subtree *(maybe empty)*
  - right subtree *(maybe empty)*

- Representation:

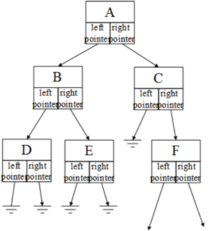

Data

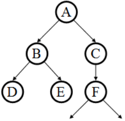
left pointer

right pointer

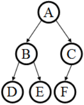


Binary Tree: Representation

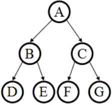




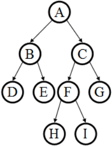
Binary Tree: Special Cases



Complete Tree



Perfect Tree



Full Tree