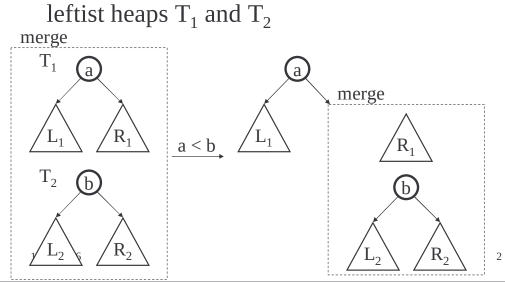
Priority Queues

(Today: Skew Heaps & Binomial Queues)

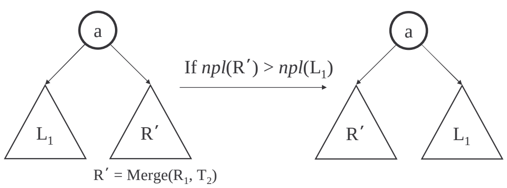
Chapter 6 in Weiss

Review Merging 2 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

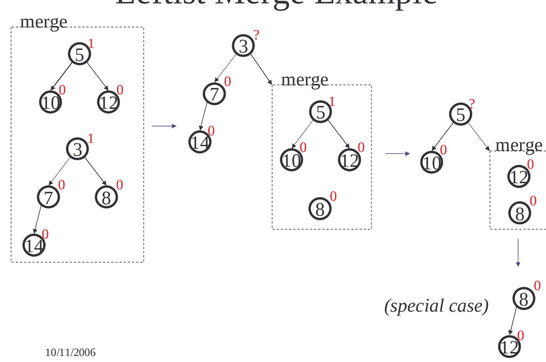


Leftist Merge Continued

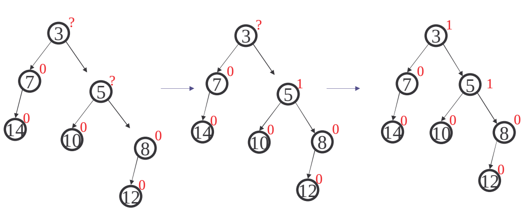


runtime:

Leftist Merge Example

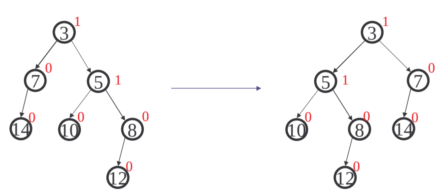


Sewing Up the Leftist Example



Done?

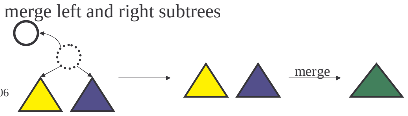
Finally…(Leftist)



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)

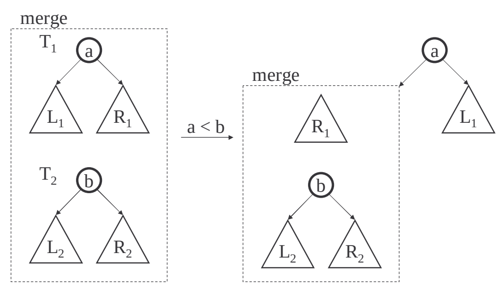
Difference from **average time:**

Skew Heaps

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

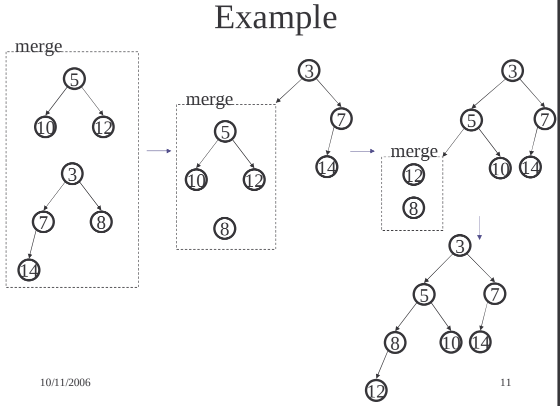
Solution: skew heaps
- "blindly" adjusting version of leftist heaps
- merge *always* switches children when fixing right path
- amortized time for: merge, insert, deleteMin = O(log *n*)
- however, worst case time for all three = O(*n*)

Merging Two Skew Heaps



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

Example



Skew Heap Code

```
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 =
- 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 =

Comparing Heaps

- Binary Heaps
- Leftist Heaps
- d-Heaps
- Skew Heaps

Still scope for improvement!

Yet Another Data Structure: Binomial Queues

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

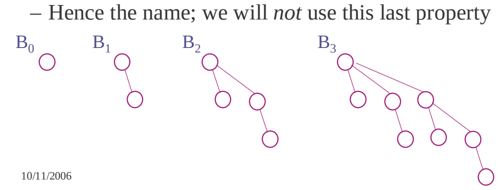
What's a forest?

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

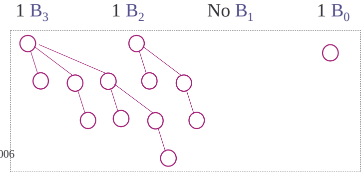


Binomial Queue with n elements

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 ?

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

Binomial Q with n nodes has height $\Theta(\log n)$

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 {

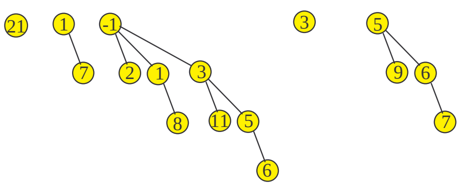
	# of 1's
a. $m \leftarrow$ total number of B_k's in the two BQs	
b. if m=0: continue;	0+0 = 0
c. if m=1: continue;	1+0 = 1
d. if m=2: combine the two B_k's to form a B_{k+1}	1+1 = 0+c
e. if m=3: retain one B_k and combine the other two to form a B_{k+1}	1+1+c = 1+c

}

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

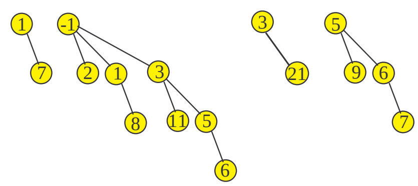
Example: Binomial Queue Merge

H1: H2:



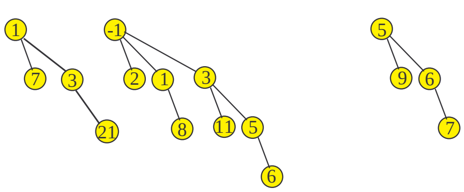
Example: Binomial Queue Merge

H1: H2:



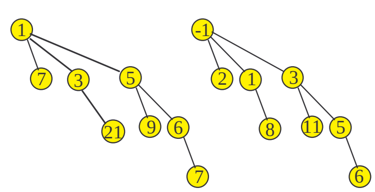
Example: Binomial Queue Merge

H1: H2:



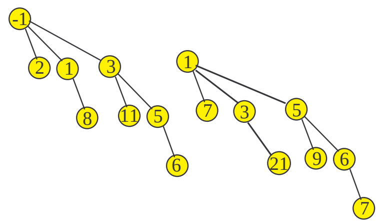
Example: Binomial Queue Merge

H1: H2:



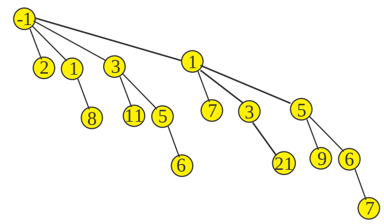
Example: Binomial Queue Merge

H1: H2:



Example: Binomial Queue Merge

H1: H2:



Merge Example

Complexity of Merge

Constant time for each height

Max height is: log *n*

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

Insert in a Binomial Queue

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

runtime

Worst case complexity: same as merge

O( )

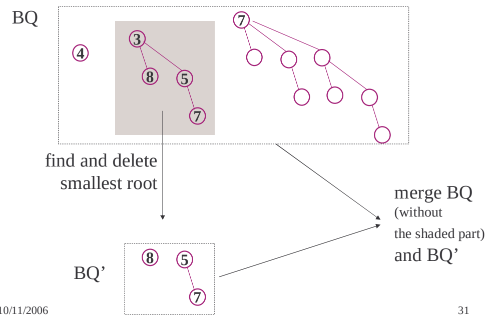
Average case complexity: O(1)

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

deleteMin in Binomial Queue

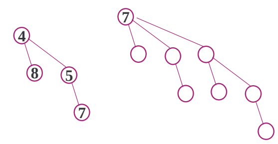
Similar to leftist and skew heaps….

deleteMin: Example



deleteMin: Example

Result:



runtime: