Priority Queues

CSE 373
Data Structures & Algorithms
Ruth Anderson

Today's Outline

- **Announcements**
  - Assignment #3 due Thurs, May 7th.
- **Today's Topics:**
  - **Priority Queues**
    - **Binary Min Heap - buildheap**
    - **D-Heaps**
    - **Leftist Heaps**

Facts about Binary Min Heaps

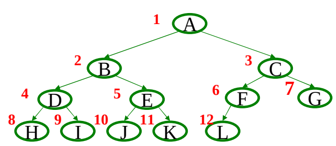
Observations:

- 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

Representing Complete Binary Trees in an Array

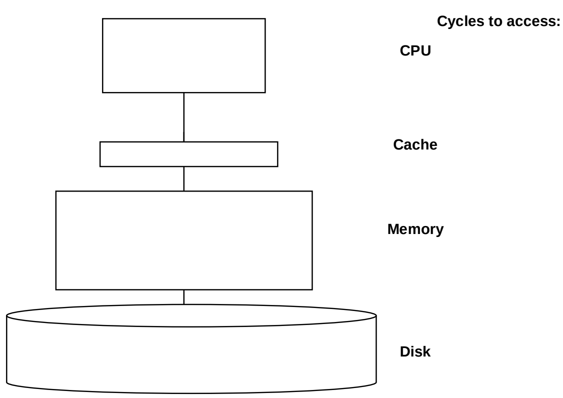


From node **i**:

left child:
right child:
parent:

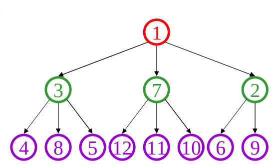
implicit (array) implementation:

	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13



A Solution: *d*-Heaps

- Each node has *d* children
- Still representable 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

Operations on *d*-Heap

- Insert : runtime =
- deleteMin: runtime =

5/01/2009 7

Priority Queues

(Leftist Heaps)

5/01/2009 8

One More Operation

- Merge two heaps. Ideas?

5/01/2009 9

New Operation: Merge

Given two heaps, merge them into one heap

- first attempt: insert each element of the smaller heap into the larger.
  runtime:
- second attempt: concatenate binary heaps' arrays and run buildHeap.
  runtime:

5/01/2009 10

Leftist Heaps

Idea:

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

Leftist heaps:

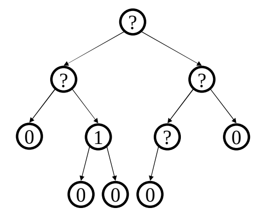
1. Most nodes are on the left
2. All the merging work is done on the right

5/01/2009 11

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, aka zero children) = 0
- *npl*(node with one child) = 0



Equivalent definitions:

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

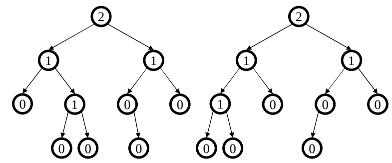
5/01/2009 12

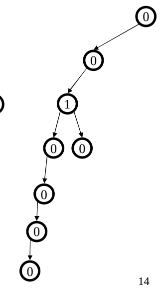
Leftist Heap Properties

- 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?

Are These Leftist?





Every subtree of a leftist tree is leftist!

Right Path in a Leftist Tree is Short (#1)

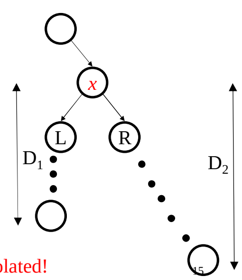
<u>Claim</u>: The right path is as short as *any* in the tree.

<u>Proof</u>: (By contradiction)

Pick a shorter path: $D_1 < D_2$
Say it diverges from right path at *x*

npl(L) ≤ D_1-1 because of the path of length D_1-1 to null

npl(R) ≥ D_2-1 because every node on right path is leftist



Leftist property at *x* violated!

Right Path in a Leftist Tree is Short (#2)

<u>Claim</u>: If the right path has **r** nodes, then the tree has at least
2

2^r-1 nodes.

<u>Proof</u>: (By induction)

Base case : **r=1**. Tree has at least 2^1-1 = 1 node

Inductive step : assume true for **r' < r**. Prove for tree with right path at least **r**.

1. Right subtree: right path of **r-1** nodes
   ⇒ 2^{r-1}-1 right subtree nodes (by induction)
2. Left subtree: also right path of length at least **r-1** (by previous slide) ⇒ 2^{r-1}-1 left subtree nodes (by induction)

Total tree size: $(2^{r-1}-1) + (2^{r-1}-1) + 1 = 2^r-1$

Why do we have the leftist property?

Because it guarantees that:

- the *right path is really short* compared to the number of nodes in the tree
- A leftist tree of N nodes, has a right path of at most **log (N+1)** nodes

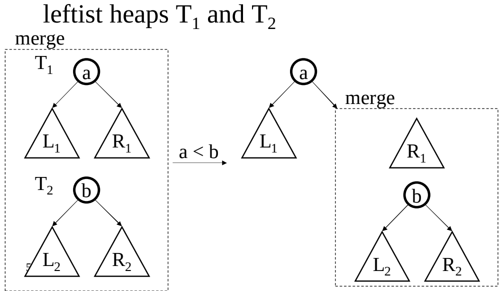
Idea – perform all work on the right path

Merge two heaps (basic idea)

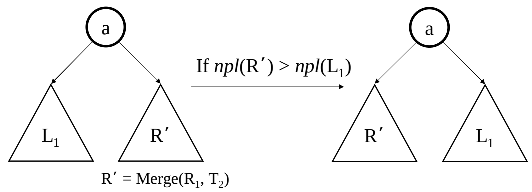
- Put the smaller root as the new root,
- Hang its left subtree on the left.
- <u>Recursively</u> merge its right subtree and the other tree.

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

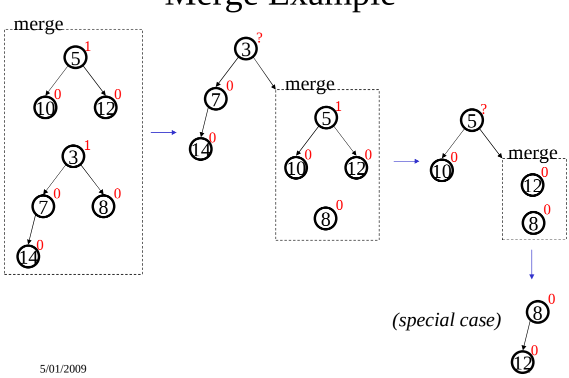


Merge Continued

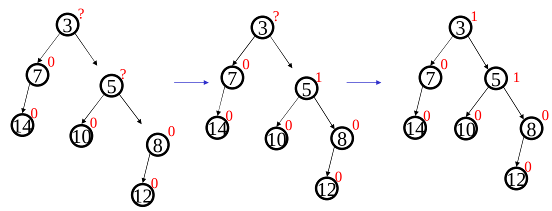


runtime:

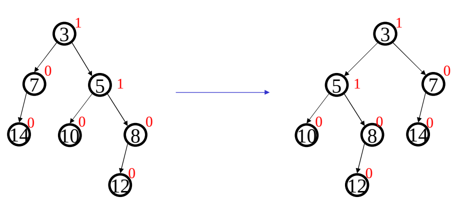
Merge Example



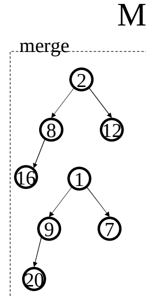
Sewing Up the Example



Finally…



Merge Two Leftist Heaps



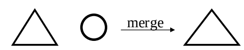
Student Activity

Other Heap Operations

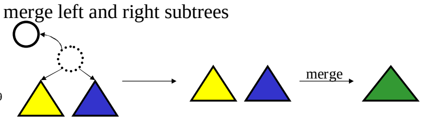
- insert ?
- deleteMin ?

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



Leftist Heaps: Summary

Good

-
-

Bad

-
-

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)

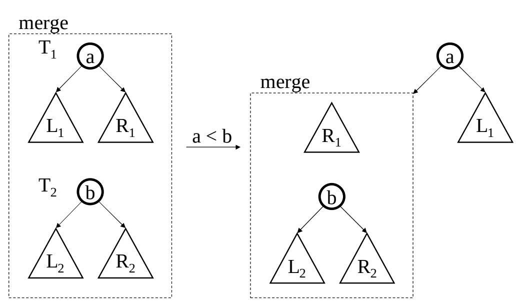
Difference from **average time:**

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

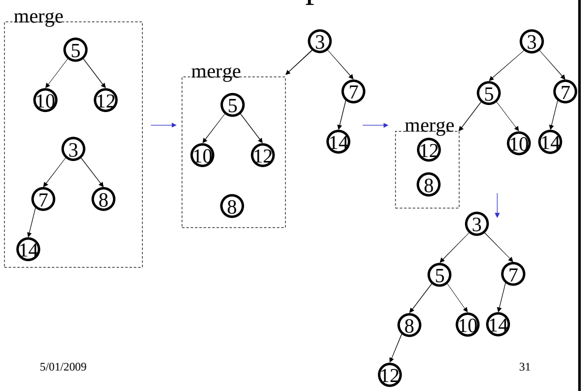
Solution: skew heaps
- "blindly" adjusting version of leftist heaps
- merge *always* switches children when fixing right path
- amortized time for: merge, insert, deleteMin = O(log *n*)
- however, worst case time for all three = O(*n*)

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 =
- Amortized Analysis (Chapter 11)
- Result: *M* merges take time *M* log *n*

  ⇒ amortized complexity of all ops =

Comparing Priority Queues

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

Student Activity