Priority Queues

CSE 373
Data Structures & Algorithms
Ruth Anderson
Spring 2008

Today’s Outline

• Admin:
  – HW #3 due this Thursday 5/1 at 11:59pm
  – Printouts due Friday in lecture.

• Priority Queues
  – Leftist Heaps
  – Skew Heaps

One More Operation

• Merge two heaps. Ideas?

Priority Queues

(Leftist Heaps)

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:

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

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

Leftist Heap Properties

- Heap-order property
  - parent’s priority value is ≤ to children’s priority values
  - result: minimum element is at the root
- Leftist property
  - For every node x, npl(left(x)) ≥ npl(right(x))
  - result: tree is at least as “heavy” on the left as the right

Are These Leftist?

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

Claim: The right path is as short as any in the tree.

Proof: (By contradiction)

Pick a shorter path: \( D_1 < D_2 \)
Say it diverges from right path at \( x \)

\[ npl(L) \leq D_1 - 1 \text{ because of the path of length } D_1 - 1 \text{ to null} \]
\[ npl(R) \geq D_2 - 1 \text{ because every node on right path is leftist} \]

Leftist property at \( x \) violated!

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

Claim: If the right path has \( r \) nodes, then the tree has at least \( 2^r - 1 \) nodes.

Proof: (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
   \[ \Rightarrow 2^{r-1} - 1 \text{ right subtree nodes (by induction)} \]
2. Left subtree: also right path of length at least \( r - 1 \) (by previous slide)
   \[ \Rightarrow 2^{r-1} - 1 \text{ left subtree nodes (by induction)} \]

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.
• Recursively merge its right subtree and the other tree.

Merging Two Leftist Heaps
• merge(T₁, T₂) returns one leftist heap containing all elements of the two (distinct) leftist heaps T₁ and T₂

Merge Continued
If np(R') > np(L₁)
R' = Merge(R₁, T₂)

Merge Example
(special case)

Sewing Up the Example

Finally…
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**

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 = \( O(n) \)

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:**
Merging Two Skew Heaps

Only one step per iteration, with children always switched

Skew Heap Code

```c
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