Heap Merge Operation

• Useful operation for priority queues
  – Combining two sets of priorities
    (perhaps to balance them or because of a failure)

• Also useful to simplify heap implementation
  – Implement Insert, DeleteMin in terms of Merge
    • Insert – Create one-element heap, merge with existing
    • DeleteMin – Remove root, merge children
Heap Merge Operation

- first attempt:
  insert each element from smaller heap into larger
  
  \textit{runtime: } O(n \log n) \text{ worst, } O(n) \text{ average}

- second attempt:
  concatenate heap arrays and run \text{buildHeap}
  
  \textit{runtime: } O(n) \text{ worst}

- another approach:
  retain the existing information in the heaps
  
  \textit{array shifting keeps us at } O(n)
Heap Properties

• Recall our two heap properties
  – Structure and Ordering

• It is structure that has been holding us back
  – Have been throwing out existing information in the ordering so we can maintain structure

• Why do we have the structure property?

• Let’s see what happens if we abandon it
Heap Merge Basics

• Use a pointer-based binary heap
  – Node ➔ Left, Node ➔ Right

• Merge two heaps by:
  – Choose one root as the merged root
    • (choose the smaller to preserve the order property)
  – Retain one of its subtrees
  – Merge its other subtree with the other root
Heap Merge Basics

\[
\text{merge} \\
T_1 \quad a \\
\quad L_1 \quad R_1 \\
T_2 \quad b \\
\quad L_2 \quad R_2 \\
\text{merge} \\
\quad L_1 \\
\quad R_1 \\
\quad L_2 \quad R_2 \\
\text{a < b}
\]
Heap Merge Basics

• How expensive is the merge?
  – Constant cost per level recursively visited
  – Worst case
    • Deep path in the tree, you recurse down it
    • Whoops: $O(n)$

• Ok, so maybe completely ignoring structure is not the best approach to this
Leftist Heaps

Idea:
Avoid recursing down a deep path in the heap

Strategy:
Focus all heap maintenance in a shallow portion 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

$npl(x) = \text{min distance to a descendant with 0 or 1 children}$

- $npl(\text{null}) = -1$
- $npl(\text{leaf}) = 0$
- $npl(\text{single-child node}) = 0$

 Equivalent definitions:
1. $npl(x)$ is the height of largest complete subtree rooted at $x$
2. $npl(x) = 1 + \min\{npl(\text{left}(x)), npl(\text{right}(x))\}$
Leftist Heap Properties

• Heap-order property
  – parent value is ≤ to child values (same as before)
  – result: minimum element is at the root

• Leftist property
  – For every node \( x \), \( npl(\text{left}(x)) \geq npl(\text{right}(x)) \)
  – result: tree is at least as “heavy” on left as right

Are leftist trees…
complete?
balanced?
Are These Leftist?
Are These Leftist?

Every subtree of a leftist tree is leftist!
Why the leftist property?

• Because it guarantees that:
  – The *right path is really short* compared to the number of nodes in the tree
  – For a leftist tree of $N$ nodes, the right path has at most $\log(N+1)$ nodes

• See the two proofs we’re about to skip

• **Win** – Perform all work on the right path, thus obtaining $O(\log N)$ merge
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 $x$

$npl(L) \leq D_1 - 1$ because of the path of length $D_1 - 1$ to null

$npl(R) \geq D_2 - 1$ because every node on right path is leftist

Leftist property at $x$ violated!
Right Path in a Leftist Tree is Short (#2)

Claim: If right path has \( r \) nodes, then the leftist 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 \( r - 1 \) (prior slide)
   \[ \Rightarrow 2^{r-1} - 1 \text{ left subtree nodes (by induction)} \]

Total tree size: \( (2^{r-1} - 1) + (2^{r-1} - 1) + 1 = 2^r - 1 \)
Leftist Heap Merge

• Merge two heaps by:
  – Choose smaller root as the merged root
  – Retain its left subtree (don’t touch it)
  – Merge its right subtree with the other root

• Done?
  – Need to preserve leftist property
If $npl(R') > npl(L_1)$

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

runtime: $O(\log n)$
Leftest Merge Example

(special case)
Sewing Up the Example

Done?
Finally…

Left Tree:
- Root: 3
- Left child: 7
  - Left child: 14
  - Right child: 10
    - Right child: 8
      - Right child: 12

Right Tree:
- Root: 3
- Left child: 5
  - Left child: 10
    - Right child: 8
    - Right child: 12
- Right child: 7
  - Right child: 14
Seems like a lot of work

• Tradeoff of the leftist heap
  – Guaranteed $O(\log n)$ merge
  – Comes at cost:
    • Added storage for caching subtree NPL values
    • Added complexity/logic to maintain and check NPL
  – And reality is:
    • Right side is “often” heavy and requires a switch

• Any wacky ideas in the room?
Skew Heaps

• “Blindly” adjusting version of leftist heaps

• Always switch left and right children of the root selected to be the merged root

• Don’t keep track of anything, don’t check anything, just always switch them
Skew Heap Merge

Only one step per iteration, with children \textit{always} switched
Skew Heap Merge Example

merge

merge

merge

merge
Amortized Complexity

Suppose you run $M$ times and average the running times

Amortized complexity:

$max$ total # steps algorithm takes, in the worst case, for $M$ consecutive operations on inputs of size $N$, divided by $M$ (i.e., divide the max total by $M$).

If $M$ operations take total $O(M \log N)$ time in the worst case, *amortized* time per operation is $O(\log N)$.

Skew heaps have amortized complexity $O(\log N)$. 