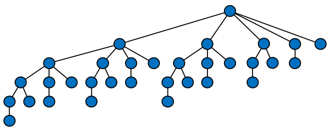
Lecture 4. Amortized Analysis (Continued)

CpSc 8400: Algorithms and Data Structures
Brian C. Dean



School of Computing
Clemson University
Spring, 2021

Amortized Analysis

- In general, an operation runs in O(f(n)) amortized time if any sequence of k such operations runs in O(k f(n)) time.
  - Example: Any sequence of k operations takes O(k) worst-case time, so we say that each operation takes O(1) **amortized** time.
- Amortized analysis is worst-case analysis, just averaged over a *sequence* of operations.

Amortization of Costs

Actual Cost	Amortized Cost
Jan: \$10	Jan: \$20
Feb: \$10	Feb: \$20
…	…
Nov: \$10	Nov: \$20
Dec: \$130	Dec: \$20

- So our cost is "\$20/month, amortized".
- This is a simpler, more accurate description of our cost structure.
- Compare with actually paying \$20/month…

3

3

Methods for Amortized Analysis

- Aggregate (based on definition): find worst-case running time of sequence of k arbitrary operations and divide by k.
- Accounting method: keep track of "credits" on elements or parts of a data structure that reflect "pre-charged" work.
- Potential function: lump all pre-charged work into one global "potential" function Φ
  - Nonnegative, starts at zero
  - Amortized cost = actual cost + $\Delta\Phi$

4

4

Example: Priority Queues and Skew Heaps

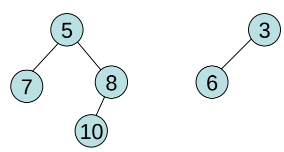
- A priority queue maintains a dynamic collection of elements, with the most important being the next to exit.
- Fundamental operations:
  – *insert*
  – *remove-min* (or *remove-max*).
- Very common type of data structure, with many applications.
- Many ways to implement priority queues…

5

5

Example: Priority Queues and Skew Heaps

- Suppose we store our priority queue in a “heap-ordered” binary tree.
  – Heap property: parent ≤ child.
  – Each node maintains a pointer to its left child and right child.
  – The tree is not necessarily “balanced”. It could conceivably be nothing more than a single sorted path.



6

6

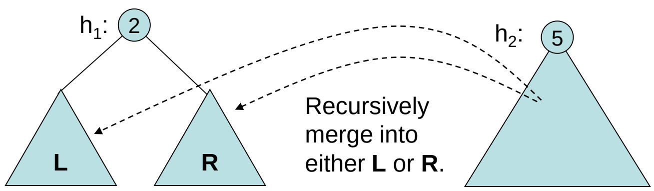
All You Need is Merge…

- Suppose we can *merge* two heap-ordered trees in O(log n) time.
- All priority queue operations now easy to implement in O(log n) time!
  - *insert*: merge with a new 1-element tree.
  - *remove-min*: remove root, merge left & right subtrees.

7

Merging Two Heap-Ordered Trees

- Take two heap-ordered trees h_1 and h_2, where h_1 has the smaller root.
- Clearly, h_1's root must become the root of the merged tree.
- To complete the merge, recursively merge h_2 into either the left or right subtree of h_1:

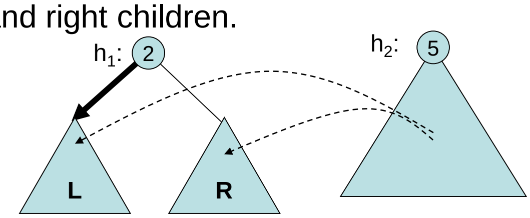


- As a base case, the process ends when we merge a heap h_1 with an empty heap, the result being just h_1.

8

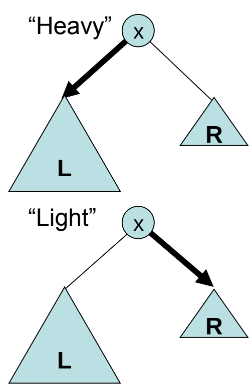
Skew Heaps

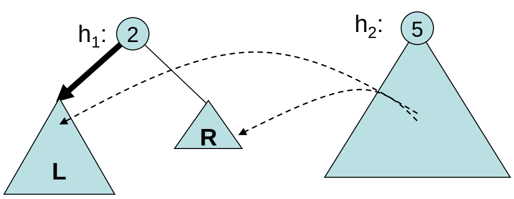
- Merge towards h_1's "preferred child", then toggle preferred child pointer (so alternate merging into left and right subtrees).
  - Equivalently: Always merge into R, but afterwards just swap h_1's children.
  - Equivalently (but more confusing…): Merge two heaps along their "right spines", then walk back up the right spine of the result, and for each element except the lowest, swap left and right children.
- Remarkably, this makes merge run in just O(log n) amortized time!



Skew Heaps: Amortized Analysis

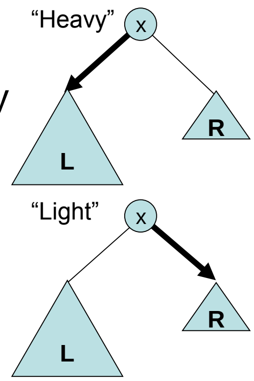
- Call nodes as "heavy" or "light"…

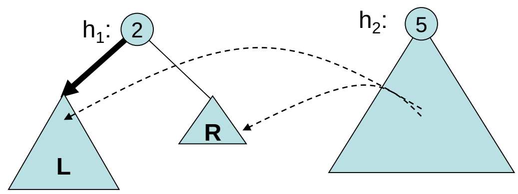




Skew Heaps: Amortized Analysis

- Call nodes as “heavy” or “light”…
- Let H and L be the number of heavy and light “steps” along a merge.

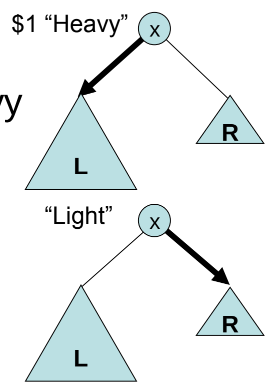


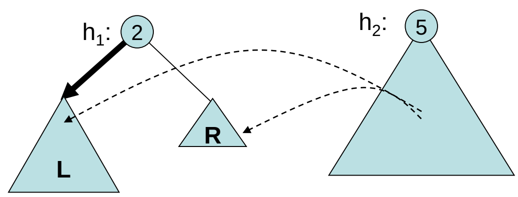


11

Skew Heaps: Amortized Analysis

- Call nodes as “heavy” or “light”…
- Let H and L be the number of heavy and light “steps” along a merge.
- L ≤ 2 log n, but H could be large.

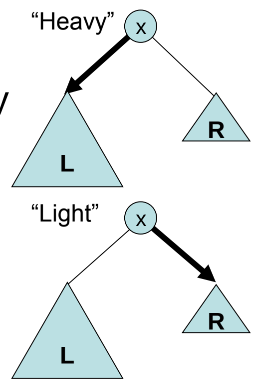


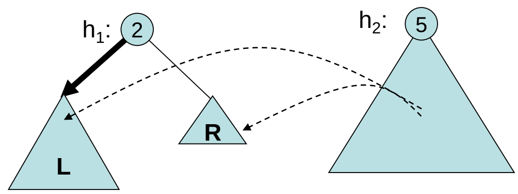


12

Skew Heaps: Amortized Analysis

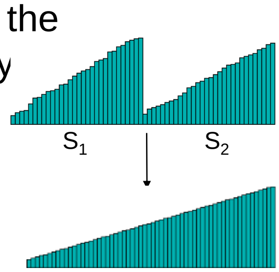
- Call nodes as “heavy” or “light”…
- Let H and L be the number of heavy and light “steps” along a merge.
- L ≤ 2 log n, but H could be large.
- Insight: pay for H using potential!
  ϕ = # of heavy nodes
- Amortized cost of merge?





Analogy: Merging Two Sorted Sequences S_1 and S_2

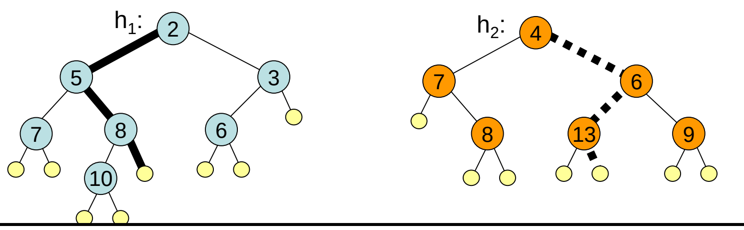
- **Iterative Outlook:** Scan two pointers p_1 and p_2 monotonically through S_1 and S_2, always selecting the min($S_1[p_1]$, $S_2[p_2]$) and advancing the corresponding pointer.
- **Recursive Outlook:** Select the minimum of $S_1[1]$ and $S_2[1]$ to be the first element of the merged sequence, then recursively solve the left-over problem.



- Both approaches take O(n) time, and are essentially equivalent.

Merging Two Heap-Ordered Trees (Null Path Merging Viewpoint)

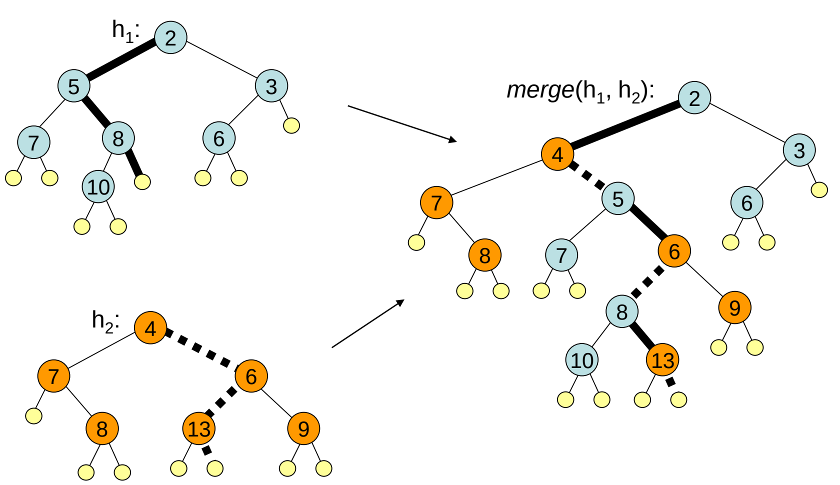
- A **null path** is a path from the root of a tree down to an "empty space" at the bottom of the tree.
- Given specific null paths in h_1 and h_2, it's easy to merge h_1 and h_2 along these paths.
  - The keys along a null path are a sorted sequence.
  - Merging along null paths is like merging two sorted sequences.
  - This process is also equivalent to the recursive merging process from the previous slide.



15

15

Merging Two Heap-Ordered Trees (Null Path Merging Viewpoint)



16

16

Running Time Analysis

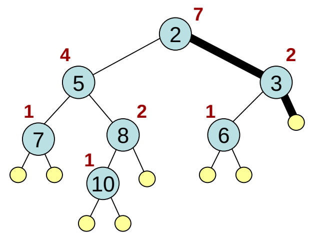
- The time required to merge two heaps along null paths is proportional to the combined lengths of these paths.
- So all we need is a method to find "short" null paths and we will have an efficient merging algorithm.
- Note that every n-node binary tree has a null path of length O(log n).
- There are many ways to find short null paths, each of which leads us to a different mergeable heap data structure…

17

17

Size-Augmented Mergeable Heaps

- One way to find short null paths…
  - Augment each element with the size of its subtree.
  - Now we can find a null path of length O(log n) by repeatedly stepping to whichever child has smaller size.
  - Each step reduces the size of our current subtree by at least a factor of 2.

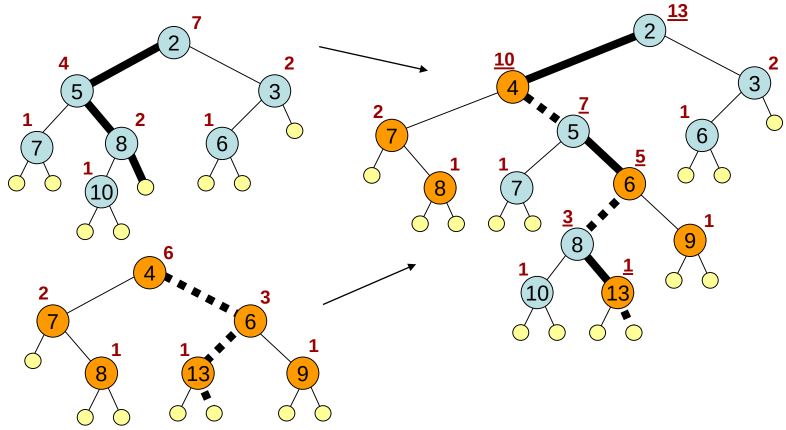


18

18

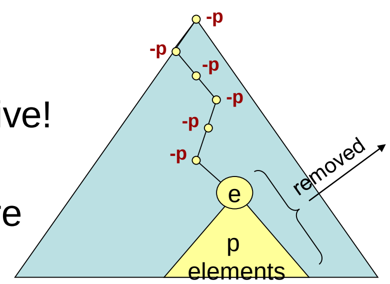
Updating Augmented Information After Merging

- When we merge two heaps, we walk back up the merge path and update augmented information.



Trouble with Decrease-Key…

- Recall: *decrease-key(e)* removes the subtree rooted at element e, decreases e's key, and then merges the result back into the main tree.
- When we remove e's subtree, we need to update the augmented size information along the path from the root down to e.
- However, if e is deep in the tree, this can be too expensive!
- This also affects *delete* and *increase-key,* since these are built using *decrease-key*.



What About Delete / Decrease-Key In a Skew Heap?

- In a skew heap, *insert* and *remove-min* are based on merging, so they run in O(log n) amortized time.
- For *decrease-key* (and *increase-key*), simply delete an element and re-insert it with a new key.
- How do we implement *delete* efficiently? (so that it doesn't interfere with the amortized analysis of other operations…)

21