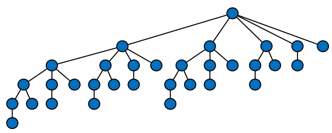
Lecture 12. Randomized Data Structures

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing

Clemson University

Spring, 2016

Warm-Up: Randomized Reduction Practice

- Take an array of distinct numbers and randomly permute it.
- If you scan through the array keeping a running maximum, how many times will this maximum be reset?

Recap: The Randomized Reduction Lemma

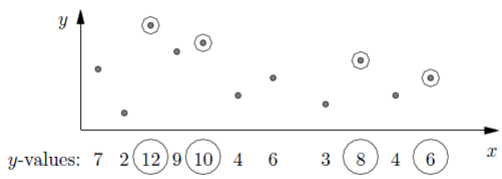
- Suppose we have an algorithm for which:
  - We start with a problem of size n.
  - In every iteration, the effective size of the problem is reduced to a constant fraction of its original size **with some constant probability.**
- Then, our algorithm performs only O(log n) iterations **with high probability**.

Warm-Up: Randomized Reduction Practice

- Take an array of distinct numbers and randomly permute it.
- If you scan through the array keeping a running maximum, how many times will this maximum be reset?
- How many non-dominated points do we expect to see in a collection of n random points in 2D?

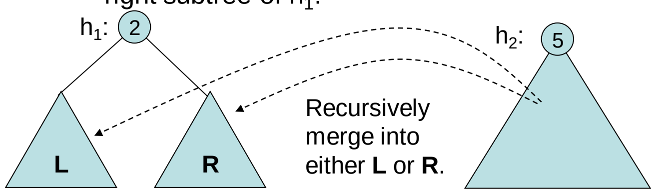
Warm-Up: Randomized Reduction Practice

- Take an array of distinct numbers and randomly permute it.
- If you scan through the array keeping a running maximum, how many times will this maximum be reset?
- How many non-dominated points do we expect to see in a collection of n random points in 2D?



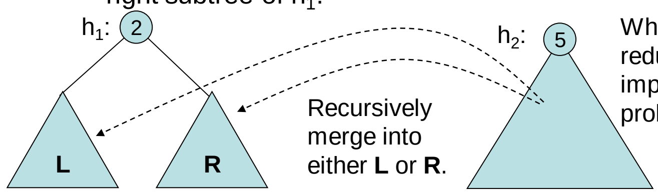
Simple Application: Mergeable Heaps

- Suppose we store elements in heap-ordered binary trees.
- All priority queue operations easy once we have *merge*:
  - Insert: merge with a new 1-element heap
  - Remove-min: remove root, merge two child subtrees back together
- A simple way to merge is using randomization!
  - Take two heap-ordered trees h_1 and h_2, h_1 having 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:



Simple Application: Mergeable Heaps

- Suppose we store elements in heap-ordered binary trees.
- All priority queue operations easy once we have *merge*:
  - Insert: merge with a new 1-element heap
  - Remove-min: remove root, merge two child subtrees back together
- A simple way to merge is using randomization!
  - Take two heap-ordered trees h_1 and h_2, h_1 having 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:

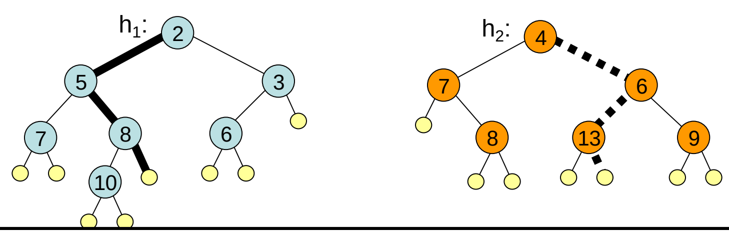


Why does the randomized reduction lemma easily imply that with high probability, this takes only O(log n) time?

7

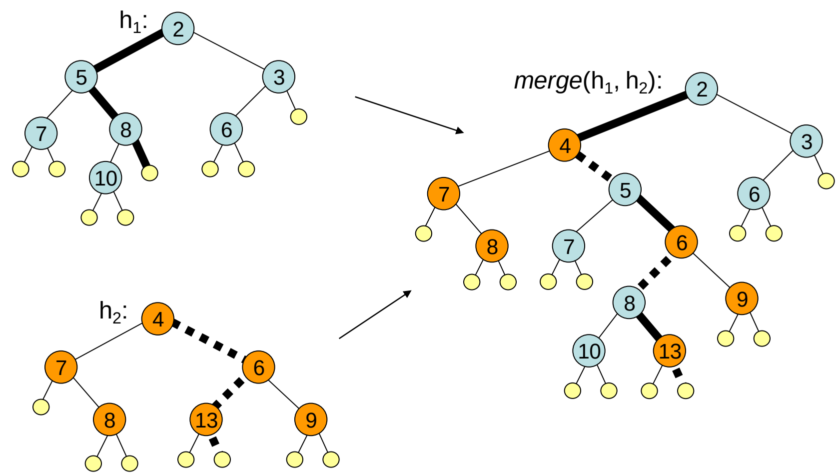
Merging Two Heap-Ordered Trees (Null Path Merging Viewpoint)

- 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.



8

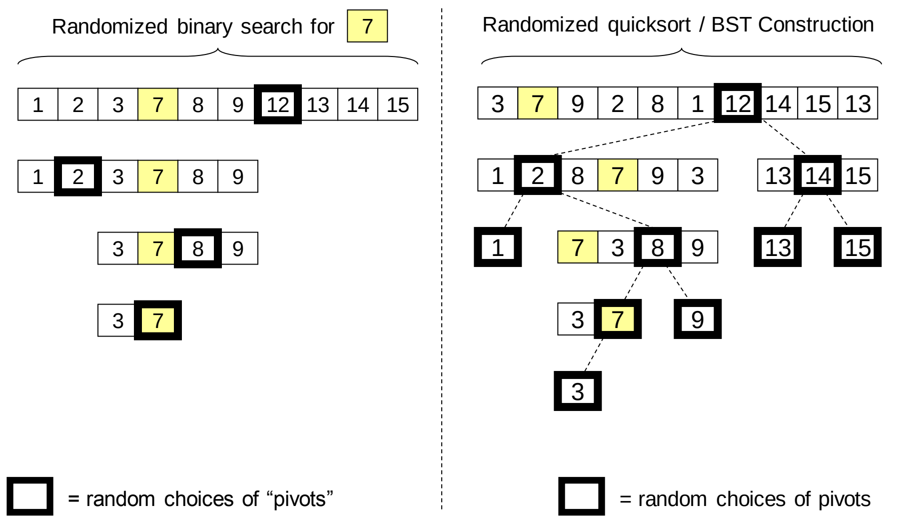
Merging Two Heap-Ordered Trees (Null Path Merging Viewpoint)



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 (e.g., leftist heaps, skew heaps).
- Our preceding approach is perhaps the simplest – just choose null paths by “walking down the tree randomly”!

Recap: Randomized Quicksort, Binary Search, and BST Construction

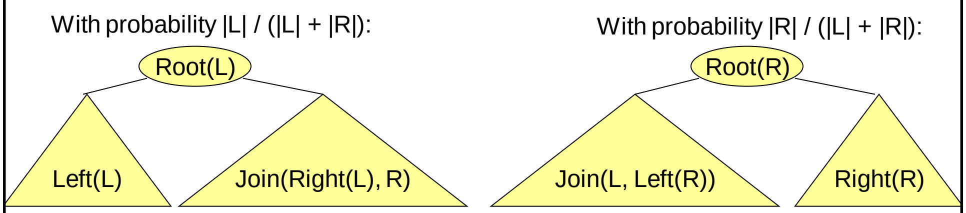


Recap: Randomly-Built BSTs

- **Theorem:** if we build a BST on n elements by inserting them in random order, then with high probability each call to insert will take O(log n) time.
- Equivalently, with high probability:
  - Each element will have depth O(log n).
  - The entire tree will have depth O(log n).
  - The entire tree will take O(n log n) time to build.
- **Corollary:** randomized quicksort runs in O(n log n) time with high probability!

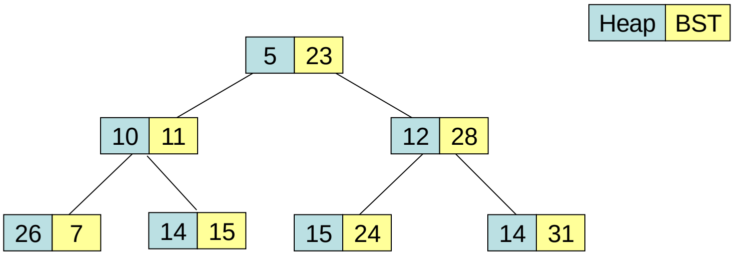
Recap: Randomly-Balanced BSTs

- To insert an element e into an (n – 1)-element tree:
  - With probability 1/n, insert e at the root (insert as usual, then rotate up to root).
  - Otherwise (with probability 1 – 1/n), recursively insert into the left or right subtree of the root,
- To delete an element e, replace e with the a **randomized join** of e's two subtrees L and R:



13

Treaps



- A treap is a binary tree in which each node contains two keys, a "**heap key**" and a "**BST key**".
- It satisfies the heap property with respect to the heap keys, and the BST property with respect to the BST keys.
- The BST keys will store the elements of our BST; we'll choose heap keys as necessary to assist in balancing.

14

Treaps

- If heap keys are all distinct, then there is only one valid "shape" for a treap. (why?)
- If we choose heap keys at random, then our treap will be randomly-structured!

  What about *insert* and *delete*?
  - *insert* : Insert new element as leaf using the standard BST insertion procedure (so BST property satisfied). Assign new element a random heap key. Then restore heap property (while preserving BST property) using *sift-up* implemented with rotations.
  - *delete* : Give element a heap key of +∞, sift it down (again using rotations, to preserve BST property) to a leaf, then delete it. Or, use *join*…

15

Several Data Structures Satisfy a Combination of BST + Heap Properties

- **Treaps.** Elements stored in BST keys. Heap keys chosen randomly to help with balance.
- **Rank-Sensitive Priority Queues** [Dean, Jones '09]. Elements stored in heap keys. BST keys randomly to help with "balance".
- **Priority Search Trees.** Holds a collection of points in the 2D plane; useful for 2D range searching. X coordinates form "BST" part, and y coordinates form "heap" part.
- **Cartesian Trees.** Represents a sequence in the BST part; the same elements also satisfy the heap property. Useful for a wide range of applications like range minimum queries, suffix tree construction, lowest common ancestors, and more.

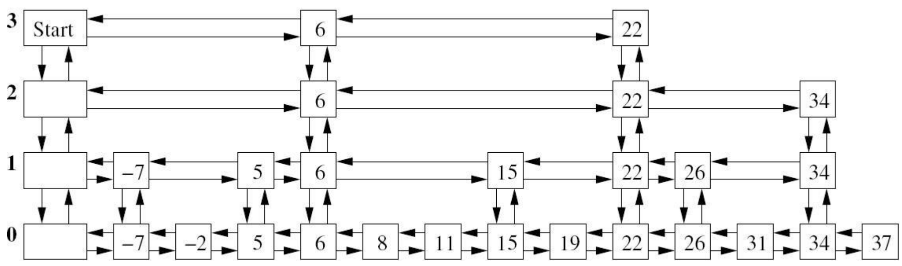
16

Skip Lists

- A skip list is a simple randomized dictionary data structure that provides O(log n) w.h.p. performance guarantees (more or less equivalent to a randomly-balanced BST).
- Very simple to implement and analyze.
- Based on linked lists rather than BSTs (often billed as an alternative to balanced BSTs…)
- Suppose we store a dictionary as a sorted linked list. Recall that scanning down the list is the bottleneck operation (taking O(n) time in the worst case).
  - How might we speed this up?

17

Example



- We insert a dummy "start" element (with effective key value -∞) that is present on all levels.
- Define L as the maximum level in the skip list.

18

Fundamental Operations

- To *find* an element, repeatedly scan right until the next step would take us too far, then step down.
- To *insert* a new element,
  - Insert into the level-0 list.
  - Flip a fair coin. If heads, also insert into the level-1 list, then flip another fair coin, and if heads again, insert into level-2 list, etc.
- To *delete*, simply remove an element from every level on which it exists.
- Other operations like *pred*, *succ*, *min*, *max*, *rank*, and *select*, are easy to implement.

19

Analysis

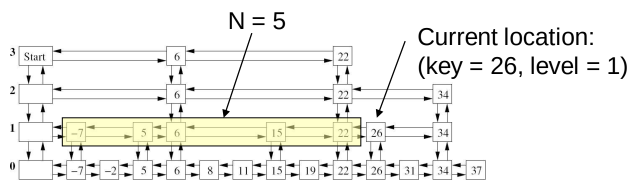
- The running time of each operation is dominated by the running time of finding an element, so let's focus on analyzing the running time of *find*.
- **Clever idea:** work backwards!
  - Starting from some element e in the level-0 list, retrace the *find* path in reverse.
  - Step up whenever possible, otherwise step to the left.
  - **Claim:** This runs in O(log n) time w.h.p.
  - **Proof:** We'd like to use the randomized reduction lemma. But how…
  - Using the union bound, we can extend this to an O(log n) w.h.p. for all elements in the skip list.

    (and this also shows that L = O(log n) w.h.p.)

20

Analysis

- Let N denote the number of elements in current level to the left of our current location during the backward scan.



- **Claim**: In each step, N is reduced to half of its current value with probability at least ¼.

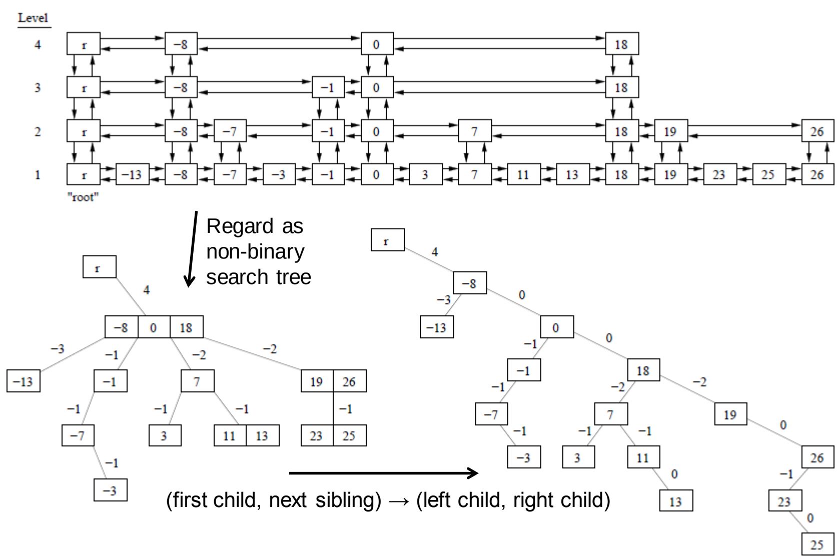
Analysis

- **Claim:** In each step, N is reduced to half of its current value with probability at least ¼.
- **Proof:** For N to be reduced to ≤ ½N two events must occur:
  - **A:** Our next step moves up, since we'd flipped heads at the current (element, level).
  - **B:** At most half the N elements to our left in the current level also flipped heads (and hence also exist on the next level).
- Pr[A ∩ B] = Pr[A]Pr[B] since A & B are independent.
- Pr[A] = ½ (unbiased coin flip)
- Pr[B] = Pr[at most N/2 heads in N coin flips] ≥ ½.
- Hence, Pr[N reduced to ≤ N/2] = Pr[A ∩ B] ≥ ¼.

Skip Lists Versus BSTs

- Skip lists were initially introduced as a “simpler alternative” to balanced BSTs.
  - Recall they provide more or less exactly the same functionality, but are based on linked lists, and do not involve any complicated tree-balancing mechanism.
- [Dean, Jones ‘07]: However, the skip list can also be converted into a fairly simple randomized BST balancing mechanism.
  - And one can even map the other way, converting any “BST” balancing mechanism to an equivalent “skip list” implementation. So if you want, say, static optimality in a skip list, use a “splay skip list”!

Skip Lists Versus BSTs



Operations

- Rebalancing after insert or delete is accomplished by a simple sequence of rotations, mimicking what happens with the skip list.

```
INSERT(T, x):
Insert x as a leaf in T (standard BST insert)
Set w(x, parent(x)) = H − h(parent(x))
while Random(0,1) = 0
    Promote(x)
If w(x, parent(x)) = 0 and x is a left child
    Rotate x with parent(x)

PROMOTE(x):
while w(x, parent(x)) = 0
    Rotate x with parent(x)
Increment w(x, parent(x))
Decrement w(x, lchild(x)) and w(x, rchild(x))
```

```
DELETE(x):
If x has no children, simply delete x
while x has two children
    Demote(x)
Let c be the single child of x
w(x, parent(x)) = w(x, parent(x)) + w(x, c)
Replace x with c.

DEMOTE(x):
if w(x, rchild(x)) = 0
    Rotate x with rchild(x)
Decrement w(x, parent(x))
Increment w(x, lchild(x)) and w(x, rchild(x))
while w(x, lchild(x)) = 0
    Rotate x with lchild(x)
```