CSE 326: Data Structures

Topic #17:

Randomized Data Structures:
Simpler Alternatives to Balanced BSTs

Ashish Sabharwal
Autumn, 2003

Today’s Outline

- Admin
  - **Final** on Monday, Dec 15, 2:30-4:20 pm, in class
  - **Guest lectures**
- Motivation for randomization
  [Section 10.4, Introduction]
- Two randomized data structures
  - **Treaps** [Section 12.5]
  - **Randomized Skip Lists** [Section 10.4.2]

Before we begin…

- Syllabus for the final:
  - Everything from day 1, including guest lectures!
  - More emphasis on material *after* midterm
- Friday guest lecture:
  - Our very own Ethan-Phelps Goodman
  - Information Retrieval, the Google way
- Monday guest lecture:
  - William Penteney, CSE grad student
  - Zero-knowledge Data Structures

The Problem with Deterministic Data Structures

We’ve seen many data structures with good average case performance on random inputs, but bad behavior on specific inputs

We define the *worst case* runtime over all possible inputs *I* of size *n* as:

$$\text{Worst-case } T(n) = \max_I T(I)$$

We define the *average case* runtime over all possible inputs *I* of size *n* as:

$$\text{Average-case } T(n) = \left(\sum_I T(I) \right) / \mathit{numPossInputs}$$

Something in-between: Randomization

Instead of randomizing the input (since we cannot!), randomize the data structure

- No bad inputs, just unlucky random numbers
- Expected good behavior on every input

Randomized data structure: a data structure whose behavior is dependant on a sequence *S* of random numbers

- Runtime of operation on input *I* is $T(I,S)$

Worst-case expected time

Definition:

- *Worst-case expected time* is the *weighted sum* of all possible runtimes on input *I* over some probability distribution on *S*

Thus, for *some particular* input *I*, we expect the runtime to be

$$\text{Expected } T(I) = \sum_S \left(\Pr(S) * T(I, S) \right)$$

And the *worst-case expected* runtime of a *randomized* data structure is:

$$\text{Expected } T(n) = \max_I \left(\sum_S (\Pr(S) * T(I, S)) \right)$$

Note: compare this with definitions of worst-case $T(n)$ and average-case $T(n)$

What's the Difference?

- Randomized with good expected time
  - Once in a while you will have an expensive operation, but no inputs can make this happen all the time
- Deterministic with good average time
  - If your application happens to always use the "bad" case, you are in big trouble!
- *Expected time is kind of like an insurance policy for your algorithm!*

Allstate. You're in good hands.

7

Comparing Different Upper Bound Analyses

Best-case $\leq$ Average-Case $\leq$ Amortized $\leq$ Worst-case

"Worst-case expected time?"

This lecture: "Worst-case expected time" = "Expected time"

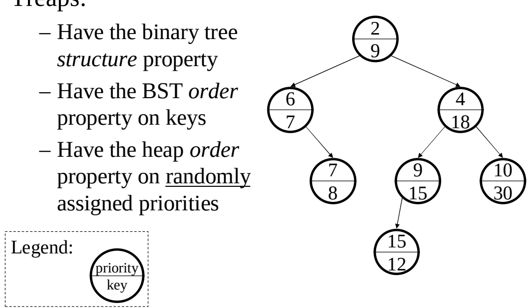
8

#1: Treap Data Structure for the Dictionary ADT

Treaps:

- Have the binary tree *structure* property
- Have the BST *order* property on keys
- Have the heap *order* property on <u>randomly</u> assigned priorities

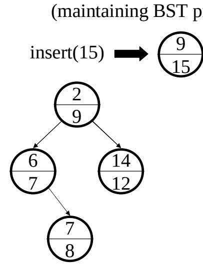
heap in yellow; search tree in blue



9

Treap Insert

- Choose a random priority
- Insert as in normal BST
- Rotate up using single rotations until heap order is restored (maintaining BST property while rotating)



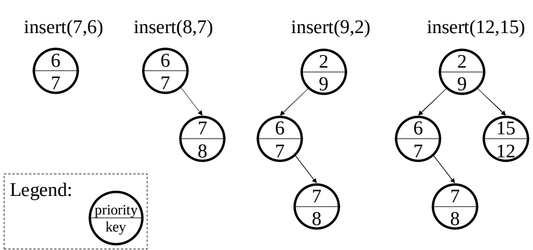
Runtime?

10

Tree + Heap… Why Bother?

Insert data in sorted order into a treap: 7, 8, 9, 12

What shape tree comes out?



11

Treap Shape

- Fix *n* distinct keys
  - What shape can a BST with these keys take? What does it depend on?
  - How about a *balanced* BST?
  - How about a Treap when *n* distinct priorities are chosen?

12

Treap Delete?

13

Treap Summary

Implements Dictionary ADT

- Insert in *expected* $\Theta(\log n)$ time
- Delete in *expected* $\Theta(\log n)$ time
- Find in *expected* $\Theta(\log n)$ time
- But *worst* case $\Theta(n)$

Memory use

- $\Theta(1)$ per node
- About the cost of AVL trees

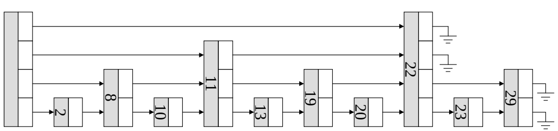
Very simple to implement, little overhead

- Less than AVL trees; only single rotations; no *npl* stuff

14

#2a: Perfect Skip List

- Sorted linked list
- # of links of a node is its *height*
- The height i link of a node (if it exists) links to the next node of height i or greater, at distance 2^{i-1}
- Result: There are *1/2 as many* height i+1 nodes as height i nodes



15

Find() in a Perfect Skip List

- Start i at the maximum height
- Until the node is found, or i =1 and the next node is too large:
  - If the key in the next node along the i link is less than the target, traverse to the next node
  - Otherwise, decrease i by one

Runtime?

16

Insert() in a Perfect Skip List

17

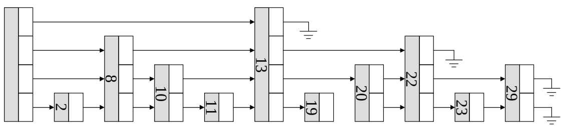
Let's Simplify Life: *Randomized* Skip List

- It's *far* too hard to insert into a perfect skip list
- But is perfection necessary?
- What matters in a skip list?

18

#2b: Randomized Skip List

- Sorted linked list
- # of links of a node is its height
- The height *i* link of a node (if it exists) links to the next node of height *i* or greater, *at whatever distance*
- **Need: There should be *about* 1/2 as many height *i*+1 nodes as height *i* nodes**



19

Find() in a RSL?

Runtime?

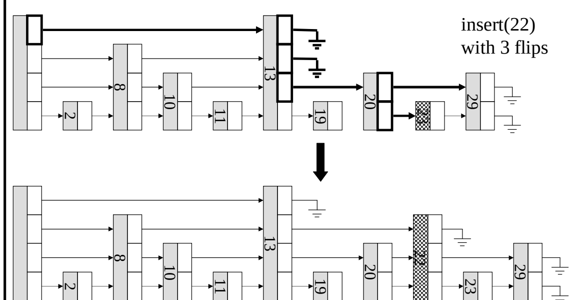
20

Insert() in a RSL

1. Flip a coin until it comes up heads
   - This will take *i* flips. Make the new node's height *I*
     $\Rightarrow$ Pr[height is *i*] = $1/2^i$
     $\Rightarrow$ Expected # nodes of height *i*+1 = ½ # nodes of height *i*
2. Do a find, remembering nodes where we moved down one link
3. Add the new node at the spot where the find ends
4. Point all the nodes where we moved down (up to the new node's height) at the new node
5. Point the new node's links where those redirected pointers were pointing

21

RSL Insert Example



Runtime?

Randomized Skip List: Summary

- Implements Dictionary ADT
  - Insert in *expected* $\Theta(\log n)$
  - Find in *expected* $\Theta(\log n)$
  - But *worst* case $\Theta(n)$
- Memory use
  - $\Theta(1)$ memory per node
  - About double a linked list
- About as efficient as balanced search trees
  (even better for some operations)
  But **much** easier to implement!

23

To Do

- Homework #3 due Friday!
- Read section 10.4 (introduction and 10.4.2)
- Read section 12.5

24