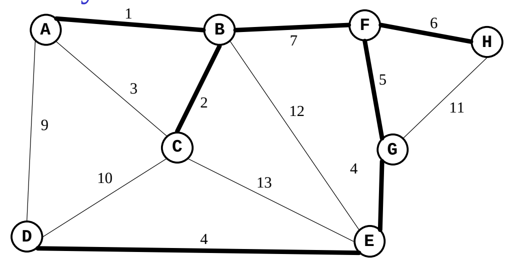
CSE 326: Data Structures
Topic 15: Random Random Algorithms

Luke McDowell
Summer Quarter 2003

Play at Home with Prim/Kruskal



2. Now find the MST using Kruskal's method.
3. Under what conditions will these methods give the same result?
4. What data structures should be used for Kruskal's? Running time?

Outline

- Review of probability
- Motivation for randomization
- Two randomized data structures
  - Treaps
  - Randomized Skip Lists
- One randomized algorithm
  - Primality checking

The Problem with Deterministic Data Structures

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

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

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

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

Average-case $T(n) = (\sum_I T(I)) / numPossInputs$

The Motivation for Randomization

Instead of randomizing the input (since we cannot!), consider randomizing the data structure
- No bad inputs, just unlucky random numbers
- Expected case good behavior on any input

Worst-case expected time

Definition:
- A *worst-case expected time* analysis is a *weighted sum* of all possible outcomes over some probability distribution

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

Expected $T(I) = \sum_S (\Pr(S) * T(I, S))$

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

Expected $T(n) = \max_I (\sum_S (\Pr(S) * T(I, S)))$

* Randomized data structure = = a data structure whose behaviour is dependant on a sequence of random numbers

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!*

Comparing different Analyses

Best-case = Average-Case = Amortized = Worst-case

"Worst-case expected time?"

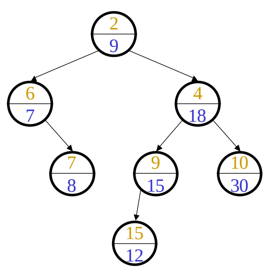
This topic: "Worst-case expected time" = "Expected time"

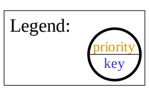
Treap Data Structure for the Dictionary ADT

Treaps:

heap in yellow; search tree in blue

- Have the binary tree *structure* property
- Have the BST *order* property
- Have the heap *order* property with randomly assigned priorities

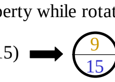


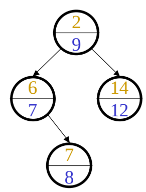


Treap Insert

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

insert(15)



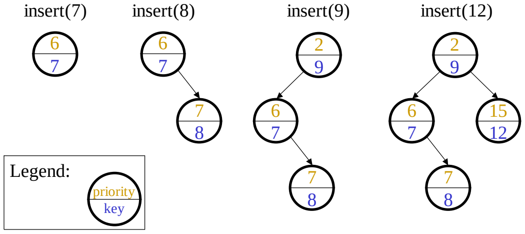


Runtime?

Tree + Heap… Why Bother?

Insert data in sorted order into a treap; what shape tree comes out?

insert(7) insert(8) insert(9) insert(12)



Treap Delete?

Treap Summary

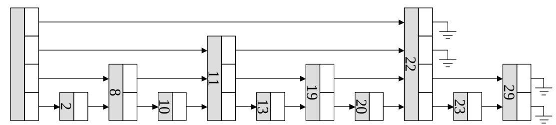
Implements Dictionary ADT
- Insert in *expected* O(log n) time
- Delete in *expected* O(log n) time
- Find in *expected* O(log n) time
- But *worst* case O(n)

Memory use
- O(1) per node
- About the cost of AVL trees

Very simple to implement, little overhead
- Less than AVL trees

Perfect Binary Skip List

- Sorted linked list
- # of links of a node is its *height*
- The height *i* link of each node (that has one) links to the next node of height *i* or greater
- There are *1/2 as many* height i+1 nodes as height i nodes



Find() in a Perfect Binary 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 next node along the *i* link is less than the target, traverse to the next node
  - Otherwise, decrease *i* by one

Runtime?

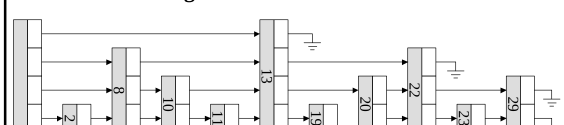
Insert() in a Perfect Binary Skip List

Randomized Skip List Intuition

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

Randomized Skip List

- Sorted linked list
- # of links of a node is its height
- The height i link of each node (that has one) links to the next node of height i or greater
- **There should be *about 1/2 as many* height i+1 nodes as height i nodes**



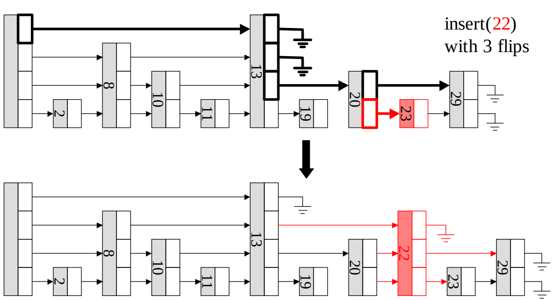
Find() in a RSL?

Runtime?

Insert() in a RSL

- Flip a coin until it comes up heads
  - This will take i flips. Make the new node's height i.
- Do a find, remembering nodes where we moved down one link
- Add the new node at the spot where the find ends
- Point all the nodes where we moved down (up to the new node's height) at the new node
- Point the new node's links where those redirected pointers were pointing

RSL Insert Example



Runtime?

Randomized Skip List
Summary

- Implements Dictionary ADT
  - Insert in *expected* O(log n)
  - Find in *expected* O(log n)
  - But *worst* case O(n)
- Memory use
  - O(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!

Primality Checking

- Given a number *P*, can we determine whether or not *P* is prime?

```
Date: Wed, 7 Aug 2002 11:00:43  -0700 (PDT)
Newsgroups: uw-cs.ugrads.openforum
Subject: Primes in P??

So, a paper published yesterday alleges they have found
  a deterministic polynomial algorithm to determine
  primality.

http://www.cse.iitk.ac.in/primality.pdf
```

Two Properties of Primes

If P is a prime **0 < A < P** and **0 < X < P**

Then:

1. $A^{P-1} = 1 \pmod{P}$
2. The only solutions to $X^2 = 1 \pmod{P}$ are: $X = 1$ and $X = P - 1$

Checking Primality

Systematic algorithm:

For all A such that 0 < A < P

Calculate A^{P-1} mod P using `pow()`

Check at each step of `pow()` and at end for two primality conditions

Problem?

Randomized algorithm:

Randomly pick an A and calculate A^{P-1} mod P using `pow()`.

Check primality conditions.

Problem?

Solution?

Evaluating Randomized Primality Testing

Your probability of being struck by lightning this year: 0.00004%

Your probability that a number that tests prime 11 times in a row is actually not prime: 0.00003%

Your probability of winning a lottery of 1 million people five times in a row: 1 in 2^{100}

Your probability that a number that tests prime 50 times in a row is actually not prime: 1 in 2^{100}

Other Real-World Applications

- Routing finding – computer networks, airline route planning
- VLSI layout – cell layout and channel routing
- Production planning – “just in time” optimization
- Protein sequence alignment
- Traveling Salesman
- Many other “NP-Hard” problems
  - A class of problems for which no exact polynomial time algorithms are known – so heuristic search is the best we can hope for