Announcements

• Start PA2 early!!
• It is due next week.
Random number generation

- Random numbers are useful in many applications: cryptography, software testing, games, probabilistic data structures, etc.
- A truly random number would be truly unpredictable
- However, ordinary programs are deterministic, and therefore their outputs are predictable, if you know the inputs, and the algorithm!
- So, you can’t write an ordinary program to generate truly random numbers; you generate pseudorandom numbers
  - (you could connect your computer to a nonalgorithmic device and collect truly random bits from it; see /dev/random on Unix systems, or various web services)
  - A typical pseudorandom number generator (RNG) is initialized with a seed; if the seed is truly random, the RNG will have as much randomness as there is in the seed
- Though the generated numbers aren’t truly random, they should appear quite random... what does that mean?
Tests for randomness

• A good pseudorandom number generator (RNG) will generate a sequence of numbers that will pass various statistical tests of randomness... the numbers will appear to be random, and can be used in many applications as if they were truly random

• Some good properties of a random sequence of numbers:
  • Each bit in a number should have a 50% probability of changing (1->0 or 0->1) when going from one number to the next in the sequence
  • The sequence of numbers should not repeat at all, or if the numbers do cycle, the cycle should be very (very!) long

• How to write a program that generates good pseudorandom numbers? One common approach is the \textit{linear congruential} RNG
Linear congruential RNG’s

• A typical pseudorandom number generator works like this:
  • The RNG is initialized by setting the value of a “seed”
    • The sequence of numbers generated is determined by the value of the seed; if you want
      the same sequence again, start with the same seed
    • If the seed has 64 bits, and those bits are picked truly randomly (e.g. by flipping a fair coin)
      then the RNG sequence has only 64 bits of “true randomness”
  • To generate the next pseudorandom number, the RNG updates the value of the seed, and returns it (or some part of it):
    \[
    \text{seed} = F(\text{seed}) \\
    \text{return partOf(\text{seed});}
    \]
  • The function \( F \) can take various forms. If it has the form
    \[
    F(x) \{ \text{return } (a \times x + c) \mod m; \}
    \]
  • ... for some constants \( a, c, m \), then this is a “linear congruential” RNG: it
    computes a linear (or affine) function of the seed, with the result taken
    congruent with respect to (i.e., modulo) \( m \)

    (Interesting to compare this to some hash functions used for strings...)
Linear congruential generators, cont’d

• How to choose the constants \(a, c, m\) for the generator?
• Not so easy to do! Many values for these constants produce RNG’s whose sequences aren’t very random at all
• The following values work well, and are used in some C/C++ standard library implementations:
  \[
  \begin{align*}
  a &= 1103515245 \\
  c &= 12345 \\
  m &= 1<<31
  \end{align*}
  \]
• The generator in the java.util.Random class uses the following values. This RNG’s low-order bits are not very random; they have a short cycle. So, computation is done with 64-bit longs, and the low order 16 bits are not exposed to the user:
  \[
  \begin{align*}
  a &= 25214903917 \\
  c &= 11 \\
  m &= 1<<48
  \end{align*}
  \]
• Other fancier approaches are needed for highly-secure cryptography, etc.
Pseudorandom numbers in C++

- The `rand()` and `srand(int)` functions are defined in the C standard library, for use in C and C++ programs.
- The `rand()` function returns pseudorandom numbers in the range $[0, \text{RAND\_MAX}]$.
  - `#include <cstdlib>` to define the `const int RAND\_MAX`
- `srand(x)` sets the seed for `rand()`’s generator.
- If `rand()` is called in your program before any call to `srand(int)`, it first calls `srand(1)`.
- Call `srand(int)` at any time to reset the seed.
  - `call srand(1)` to restart the default sequence.
  - `#include <ctime>`, and call `srand(time(0))` to get a seed that depends on the current system clock time, but only to 1 second resolution.
Skip Lists: Motivation

Which data structure is faster in the worst case for finding elements, assuming the elements are sorted?

A. An array
B. A linked list
C. They can both be made equally fast
Skip Lists: Motivation

- Which data structure is faster in the worst case for *inserting* elements, assuming the elements are sorted?
  A. An array
  B. A linked list
  C. They can both be equally fast (slow)
Toward Skip lists

- Adding forward pointers in a list can vastly reduce search time, from \(O(N)\) to \(O(\log N)\) in the worst case.
- However, what is the main problem with this approach?
  A. Pointers have to stay appropriately spaced to maintain the worst case bound
  B. The pointers take the space requirements for the list from \(O(N)\) to \(O(N^2)\)
  C. Pointers only point forward in the list, so you can get better running time bounds by including backward pointers as well.
Toward Skip lists

- Adding forward pointers in a list can vastly reduce search time, from $O(N)$ to $O(\log N)$ in the worst case.
- However, what is the main problem with this approach?
  A. Pointers have to stay appropriately spaced to maintain the worst case bound.

Deterministically adjusting pointer location is costly. Skip lists fix this problem by using randomness to randomly determine where pointers go.
Is the following a correct implementation of a set using a SkipList? (As described in your reading?)

A. Yes
B. No, because the values in the nodes are not in sorted order
C. No, because the nodes at different levels are not properly spaced
D. No, because there are no level 4 nodes
E. More than one of these
Our definition of SkipLists: Always sorted

The book is a bit vague about whether SkipLists have to be sorted.

When we talk about a SkipList in this class, the elements in the SkipList must be sorted. The structure above is not a valid SkipList.
Creating a Skip List

Draw the Skip List that results from inserting the following elements with the following levels:

Elements: 42, 3, 12, 20, 9, 5
Levels: 3, 1, 1, 2, 1, 2
Creating a Skip List

Draw the Skip List that results from inserting the following elements with the following levels

Elements: 42, 3, 12, 20, 9, 5
Levels: 3, 1, 1, 2, 1, 2
SkipList find, pseudocode (slight difference from the reading)

- To find key $k$:
  1. Let $x$ be list header (root). Let $i$ be the highest non-null index pointer in $x$
  2. While pointer $x[i]$ is not null and points to a key smaller than $k$, let $x = x[i]$ (follow the pointer)
  3. If the pointer $x[i]$ points to a key equal to $k$ return true
  4. If the pointer $x[i]$ points to a key larger than $k$, decrement $i$ (drop down a level in $x$)
  5. If $i < 0$ return false. Else go to 2.

Assumes index pointers are 1 less than level
Find in a Skip List

Which of the following pointers are checked and/or followed in a find for the element 35?

A. Red only
B. Red and blue only
C. Red, blue and purple only
D. Red, blue, purple and black
E. Some other combination of pointers
Find in a Skip List

Highlight the pointers that are checked/followed in a find for the element 12. Annotate the order in which they are checked.
Why Skip Lists?

Why use a skip list?
- Simple implementation
- Simple in-order traversal
- Fast (comparable to Balanced Binary Tree in the average case)
- Amenable to concurrent modification (changes are quite local)

Insert 22 with level 2: highlight everything that needs to change in the list. Try to figure out how you would keep track of all this information.
SkipList insert: slow motion

- prev[i] should point to the node whose level i pointer needs to point to the new node.
- curr[i] should point to the same node that new node’s level i pointer must point to after its inserted.
SkipList insert: slow motion

prev

root

curr

lvl:

3 21 25 6

26
SkipList insert: slow motion

Because prev[3] is null, start at root again

lvl:

2
SkipList insert: slow motion

root

prev

lvl:

2

curr

26
SkipList insert: slow motion
SkipList insert: slow motion
SkipList insert: slow motion
SkipList insert: slow motion

prev

root

newNode

lvl: -1

curr

25
SkipList insert: slow motion
SkipList insert: slow motion
SkipList insert: slow motion
Properties of skip lists

- It is possible for a skip list to be badly “unbalanced”, so that searching in it is like searching in a linked list
  - for example, this will happen if the levels of the nodes in the list are nonincreasing (or nondecreasing) as you move from the beginning of the list to the end
- So, it is possible that the time cost of operations in a skip list will be $O(N)$
- However, Pugh shows that the expected number of comparisons to do a find in a skip list with $N$ nodes using probability $p$ for generating levels is
  $$E[\text{comparisons}] \leq \frac{\log_{1/p} N}{p} + \frac{1}{1 - p}$$
- (For $p = 1/2$, this implies that the average number of comparisons should be no more than about $2 \log_2 N$, which can be compared to about $1.386 \log_2 N$ for a randomized search tree)
- And so the expected time costs for insert and delete are also $O(\log N)$
Properties of skip lists, cont’d

• The expected time costs are like average time costs, averaged over many constructions of a skip list with the same N keys, but different randomly chosen node levels.

• An interesting question is: For a single skip list with N keys, how likely is it that its `height’ is much greater than log N?

• Pugh analyzed this question and, though he did not publish a formula to compute it, he shows graphs which indicate that the answer is:
  • it is possible, but it can be considered extremely unlikely

• Overall, skip lists are comparable to randomized search trees in:
  • time costs of the basic operations
  • memory cost
  • implementation complexity