Lecture 9
Skip Lists
• Data structure
• Randomized insertion
• Expected Time Bound
• Distributed Hash Tables and the CHORD system
Skip lists

• Simple randomized dynamic search structure
  – Invented by William Pugh in 1989
  – Easy to implement
• Maintains a dynamic set of $n$ elements in $O(\lg n)$ time per operation in expectation
One linked list

Start from simplest data structure: (sorted) linked list

• Searches take $\Theta(n)$ time in worst case
• How can we speed up searches?

14 → 23 → 34 → 42 → 50 → 59 → 66 → 72 → 79
Two linked lists

Suppose we had *two* sorted linked lists (on subsets of the elements)

- Each element can appear in one or both lists
- How can we speed up searches?
Two linked lists as a subway

Idea: Express and local subway lines
    (à la New York City 7th Avenue Line)
    • Express line connects a few of the stations
    • Local line connects all stations
    • Links between lines at common stations
Searching in two linked lists

SEARCH(x):

- Walk right in top linked list ($L_1$) until going right would go too far
- Walk down to bottom linked list ($L_2$)
- Walk right in $L_2$ until element found (or not)
Searching in two linked lists

**Example:** Search(59)

Too far: 59 < 72
**Design of two linked lists**

**Question:** Which nodes should be in $L_1$?

- In a subway, the “popular stations”
- Here we care about worst-case performance
- **Best approach:** Evenly space the nodes in $L_1$
- But how many nodes should be in $L_1$?
Analysis of two linked lists

**Analysis:**

- Search cost is roughly $|L_1| + \frac{|L_2|}{|L_1|}$
- Minimized (up to constant factors) when terms are equal

\[ |L_1|^2 = |L_2| = n \Rightarrow |L_1| = \sqrt{n} \]
Analysis of two linked lists

**Analysis:**

- $|L_1| = \sqrt{n}$, $|L_2| = n$
- Search cost is roughly

$$|L_1| + \left|\frac{L_2}{|L_1|}\right| = \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$$
More linked lists

What if we had more sorted linked lists?

- 2 sorted lists $\Rightarrow 2 \cdot \sqrt{n}$
- 3 sorted lists $\Rightarrow 3 \cdot \frac{3}{2} \sqrt{n}$
- $k$ sorted lists $\Rightarrow k \cdot \sqrt{n}$
- $\lg n$ sorted lists $\Rightarrow \lg n \cdot \sqrt{n} = 2 \lg n$
**lg \(n\) linked lists**

\(lg \ n\) sorted linked lists are like a binary tree  
(in fact, level-linked B\(^+\)-tree)
Searching in $\lg n$ linked lists

**EXAMPLE:** SEARCH(72)
Skip lists

*Ideal skip list* is this $\lg n$ linked list structure

*Skip list data structure* maintains roughly this structure subject to updates (insert/delete)
**INSERT(x)**

To insert an element $x$ into a skip list:

- **SEARCH($x$)** to see where $x$ fits in bottom list
- Always insert into bottom list

**INVARIANT:** Bottom list contains all elements

- Insert into some of the lists above…

**QUESTION:** To which other lists should we add $x$?
**INSERT**(x)

**QUESTION:** To which other lists should we add x?

**Idea:** Flip a (fair) coin; if HEADS, promote x to next level up and flip again

- Probability of promotion to next level = \( p = \frac{1}{2} \)
- On average:
  - 1/2 of the elements promoted 0 levels
  - 1/4 of the elements promoted 1 level
  - 1/8 of the elements promoted 2 levels
  - etc.

**Approx. balanced?**
Example of skip list

**Exercise:** Try building a skip list from scratch by repeated insertion using a real coin

**Small change:**
- Add special $-\infty$ value to every list
  $\Rightarrow$ can search with the same algorithm
Skip lists

A skip list is the result of insertions (and deletions) from an initially empty structure (containing just $-\infty$)

- **INSERT**(x) uses random coin flips to decide promotion level
- **DELETE**(x) removes x from all lists containing it

How many levels? $O(\log_{1/p} n)$ on average
Expected Time for SEARCH

- Search for target begins with head element in top list
- Proceed horizontally until current element greater than or equal to target
- If the current element is equal to the target, it has been found. If the current element is greater than the target, go back to the previous element and drop down vertically to the next lower list and repeat the procedure.
- The expected number of steps in each linked list is seen to be $1/p$, by tracing the search path backwards from the target until reaching an element that appears in the next higher list.
- The total expected cost of a search is $O(\log_{1/p} n) \cdot (1/p)$ which is $O(\lg n)$ when $p$ is a constant
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Robert Morris
Ion Stoica, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

MIT and Berkeley
A peer-to-peer storage problem

- 1,000,000 scattered music enthusiasts
- Willing to store and serve replicas
- How do you find the data?
The lookup problem

Client
Lookup("title")

Internet

N1
N2
N3
N4
N5
N6

Publisher

Key="title"
Value=MP3 data...
Centralized lookup (Napster)

SetLoc("title", N4)
Publisher@N4
Key="title"
Value=MP3 data...

N1 N2 N3

DB

N4 N5 N6 N7 N8

Client

Lookup("title")

Simple, but $O(N)$ state and a single point of failure
Routed queries (Chord)

Publisher
Key="title"
Value=MP3 data...

Client
Lookup("title")

L9.24

Copyright © 2001-8 by Leiserson et al
Chord properties

- Efficient: $O(\log(N))$ messages per lookup
  - $N$ is the total number of servers
- Scalable: $O(\log(N))$ state per node
- Robust: survives massive failures
Chord overview

• Provides peer-to-peer hash lookup:
  – Lookup(key) → IP address
  – Chord does not store the data

• How does Chord route lookups?

• How does Chord maintain routing tables?
Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space

- How to map key IDs to node IDs?
Consistent hashing [Karger 97]

A key is stored at its successor: node with next higher ID
Basic lookup

"Where is key 80?"

"N90 has K80"
Simple lookup algorithm

Lookup(my-id, key-id)

n = my successor
if my-id < n < key-id
   call Lookup(id) on node n    // next hop
else
   return my successor    // done

• Correctness depends only on successors
“Finger table” allows \( \log(N) \)-time lookups

A Skip-List!
Finger $i$ points to successor of $n + 2^i$
Lookup with fingers

Lookup(my-id, key-id)
look in local finger table for
  highest node n s.t. my-id < n < key-id
if n exists
  call Lookup(id) on node n        // next hop
else
  return my successor           // done
Lookups take $O(\log(N))$ hops
Joining: linked list insert

1. Lookup(36)
Join (2)

2. N36 sets its own successor pointer
Join (3)

3. Copy keys 26..36 from N40 to N36
Join (4)

4. Set N25’s successor pointer

Update finger pointers in the background
Correct successors produce correct lookups