Skip Lists

Skip lists are yet another way to implement a set/dictionary.

A skip list $S$ is a sequence of $h + 1$ lists $S_0, \ldots, S_h$:

- $S_0$ has all keys, and $-\infty$ and $+\infty$, in non-decreasing order
- for each $i$ from 1 to $h - 1$, $S_i$ has a subset of keys from $S_{i-1}$, and $-\infty$ and $+\infty$, in non-decreasing order
- $S_h$ stores $-\infty$ and $+\infty$, in non-decreasing order

Randomization: a key appearing in $S_{i-1}$ has $1/2$ probability of appearing again in $S_i$.

Average-case times of find, insert, remove are $\Theta(\log(n))$.

Finding A Key

Starting from the top-left corner:

How would you find 80?

How would you discover that 85 is absent?

Basic Helper: SkipSearch

This basic helper underlies find, insert, and remove:
Find where the largest key less than or equal to $k$ appears.

SkipSearch($k$):

1. $p :=$ the 1st node in $S_h$ (it has $-\infty$)
2. while $p$.below $\neq$ null
   - $p := p$.below
3. while $p$.after.key $\leq k$
   - $p := p$.after
4. return $p$

find

find($k$):

1. $p :=$ SkipSearch($k$)
2. if $p$.key $= k$ then
   - return $p$.value
3. else
   - throw NotFound

Horizontal links in $S_i$ so you can go left or right in $\Theta(1)$ time.

Vertical links when a key in $S_i$ reappears in $S_{i-1}$ so you can go up or down in $\Theta(1)$ time.
To remove 40:
SkipSearch(40) lands at its place in \( S_0 \).
Take out the whole column, from bottom to top. Fix up pointers.

If this causes some lists to become just \((-\infty, +\infty)\), discard them and decrease \( h \) accordingly.

To insert 35:
SkipSearch(35) lands at 30 in \( S_0 \).
Insert after that. Toss coin to add to more levels.
To level up, may have to go left first.

May have to create a new level and increase \( h \) by 1.

Policy on The Number of Levels
Usually two choices:
- \( h = 3 \cdot \lceil \log_2(n) \rceil \)
  When inserting, stop if you hit \( S_h \).
  Create new levels only when \( n \) is large enough to increase \( h \).
- Unlimited: create new levels as long as the coin says tail.
  We will see why you likely get just \( O(\log(n)) \) levels anyway.

Average-Case Time of SkipSearch
Time is proportional to the number of levels + forward steps.
**Probability on The Number of Levels**

Want: \( \Pr(\text{some key in } S_{\lfloor \log_2(n) \rfloor}) \)

\[
\begin{align*}
\Pr(\text{some key in } S_k) &= \Pr(1\text{st key in } S_k \lor \cdots \lor \text{n\textsuperscript{th} key in } S_k) \\
&\leq \Pr(1\text{st key in } S_k) + \cdots + \Pr(\text{n\textsuperscript{th} key in } S_k) \\
\Pr(\text{ith key in } S_k) &= \Pr(k \text{ coin tosses are all tails}) \\
&= \frac{1}{2^k} \\
\Pr(\text{some key in } S_{\lfloor \log_2(n) \rfloor}) &\leq \frac{n}{2^{\lfloor \log_2(n) \rfloor}} \\
&\leq \frac{n}{2^{\lfloor \log_2(n) \rfloor}} \\
&= \frac{n}{n^3} \\
&= \frac{1}{n^2}
\end{align*}
\]

**Expected Value of The Number of Forward Steps**

When scanning forward in \( S_i \):

- Each forward step goes to a key not seen when you were in \( S_{i+1} \).
- If you have to take \( k \) forward steps, you see \( k \) keys all failing to make it to \( S_{i+1} \). Probability is \( 1/2^k \).

\[
E(\text{number of forward steps}) = \sum_{k=0}^{\infty} k \cdot \frac{1}{2^k} = 2
\]

(Similar to expected number of tosses until head.)

Expected number of forward steps in one level is 2.

**Average-Case Time of Skip List Operations**

Number of levels: very likely \( 3 \cdot \lfloor \log_2(n) \rfloor \), or you can enforce it.

Number of forward steps per level: average 2.

Average-case times:

- \text{SkipSearch}: \( \Theta(\log(n)) \)
- \text{find}: \text{SkipSearch}, then \( \Theta(1) \) for one key comparison
- \text{remove}: \text{SkipSearch}, then \( \Theta(\log(n)) \) for removing from levels
- \text{insert}: \text{SkipSearch}, then \( \Theta(\log(n)) \) for inserting into levels

All end as \( \Theta(\log(n)) \).

**Next Topic**

We have seen two ways of implementing dictionaries: binary search trees, skip lists.

They all assume that keys can be compared by \( \leq, =, \geq \).

Another way of implementing dictionaries makes a different assumption.

**Hash Tables**

Assume you can map keys to natural numbers from 0 to \( N - 1 \).

Call that function \( h \).

Use an array \( A \) of length \( N \).

Put key \( k \) in \( A[h(k)] \). Except . . .

What if \( h \) maps several different keys to the same number?

Solution:

\( A \) is an array of lists. Put key \( k \) in the list at \( A[h(k)] \).

(Later we will see another solution.)

If there are \( n \) keys, and \( h \) distributes them randomly and uniformly:

Average-case time of insert is \( \Theta(1) \).

Average-case times of find and remove are \( \Theta(n/N) \).

(We’re also assuming that \( h(k) \) takes \( O(1) \) to compute.)
Hash Functions

We like to factor $h$ into two stages

- hash code/value: map keys to ints
  This stage does not depend on $N$.
  This stage just depends on the kind of keys.
  Design this stage to try to avoid collisions.

- compression map: map ints to $[0, N - 1]$
  This stage does not depend on the kind of keys.
  This stage just depends on $N$.
  Design this stage to try to be uniform.

Hash Codes

Hash code design depends on the kind of keys. Examples:

- int: self.
- Array of ints: sum them.
- String: Each character is internally a number. Sum them.

This idea can be extended to data that are internally sequences of machine words.

Try to make sure that every bit contributes.

Polynomial Hash Codes

Polynomial hash codes make sure that every bit contributes, and also their positions contribute.

So that for example “stop” and “pots” get different hash codes.

Pick constant $a$, not zero, not one.

If your key is internally the machine words $x_0, \ldots, x_{k-1}$, then the hash code is

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

Compute by:

$\begin{align*}
c &:= 0 \\
&\text{for } i \text{ from } 0 \text{ to } k - 1 \\
c &:= c \cdot a + x_i
\end{align*}$

Compression Maps

Suppose the hash code is $c$.

Division method: $c \mod N$

Simple idea, but susceptible to regular patterns in keys (more collisions).

Multiply-add-divide method: $(a \cdot c + b) \mod N$

Pick random $a$, random $b$.

Less susceptible to regular patterns in keys.

Either way, choose prime number for $N$ to further fight regular patterns. This also determines array length.

Next time: More tricks on hash tables!