1 Why Skip Lists?

- At this point, an algorithms course traditionally talks about balanced binary trees
- (e.g. red-black trees, AVL trees)
- Idea: dynamically rebalance tree to keep height $\Theta(\log n)$ at all times
- Unfortunately, balancing trees efficiently is rather complex!
- In 1987, Bill Pugh came up with a new randomized data structure with same expected performance as balanced trees.
- Much simpler to describe, so we’ll do this first and come back to balanced trees later.

2 Skip List Definition

A skip list is like an ordered doubly linked list, but it has extra pointers allowing us to jump across several elements in the list at a time.
Better start with an example:

- Each node of the list has both key and “pillar” of some height $t$ ($t$ varies among pillars)
• pillar: an array of next and prev pointers
• bottom of pillar is level 0, runs up to level \( t - 1 \)
• all pillars of height at least \( \ell + 1 \) are linked as a list by pointers stored at level \( \ell \) (original list is at level 0)
• notice that no pointer at any level jumps over more nodes than the pointer above it
• head and tail pillars at ends with values \(-\infty\) and \(+\infty\) form ends of lists at every level

Why is this randomized? Height of each pillar is chosen at random.

3 Simple Operations

• min() is head.next[0].key \((+\infty\) if list is empty) 
• max() is tail.prev[0].key \((-\infty\) if list is empty) 
• If we keep pointers to head and tail around, cost is \( \Theta(1) \).

How about successor?

• Assuming we’re holding a record \( x \), succ\((x)\) is just next node in lowest-level list. Return \( x$.next[0].key.
• Similarly easy for pred\((x)\).
• Both are \( \Theta(1) \).

How about deletion?

\[
\text{REMOVE}(x) \\
\text{for } \ell \text{ in } 0 \ldots x$.height – 1 \\
\text{splice } x \text{ out of linked list at level } \ell
\]

Cost is \( \Theta(t) \) for a pillar of height \( t \).

4 Searching for a Key

• idea: like search in ordered list, but…
• can use lists at higher levels to skip to middle of list quickly

\[
\text{FIND}(k) \\
\ell \leftarrow \text{head.height} – 1 \\
x \leftarrow \text{head} \\
\text{while } \ell \geq 0 \text{ do} \\
\quad y \leftarrow x$.next[\ell] \\
\quad \text{if } y$.key = k \\
\quad \quad \text{return } y \\
\quad \text{else if } y$.key < k
\]
5 Inserting a Key

Insertion is a lot like search. For simplicity, assume that keys are all unique.

- must create pillar for new node
- will choose height of new pillar at random
- height distribution is geometric, not uniform
- \( \Pr[\text{height} = t] = \left(\frac{1}{2}\right)^t, t \geq 1 \)
- if we flip a fair coin, when does it first come up heads?
- call generator \( \text{RandomHeight()} \)

A small problem – what if \( t \) comes out higher than height of head and tail pillars? Easy answer: double their heights, perhaps repeatedly, to make the head and tail at least \( t \) high each time this happens (like resizing a hash table), and you won’t do it too often.

\[
\text{INSERT}(z)
\]

\[
t \leftarrow \text{RandomHeight}
\]

allocate a pillar of height \( t \) for \( z \)

\[
\ell \leftarrow \text{head.height} - 1
\]

\[
x \leftarrow \text{head}
\]

\[
\text{while } \ell \geq 0 \text{ do}
\]

\[
y \leftarrow x.\text{next}[\ell]
\]

\[
\text{if } y.\text{key} < \text{key}
\]

\[
x \leftarrow y
\]

\[
\text{else}
\]

\[
\text{if } \ell < t
\]

\[
\text{link } z \text{ into list at level } \ell \text{ between } x \text{ and } y
\]

\[
\ell \leftarrow -
\]

**Example:** insert 4 into list, suppose new pillar has height 3.

- Intuition: could place \( z \) separately by traversing list at every level starting at head.
• If new node $z$ belongs between $x$ and $x.next[\ell]$... at level $\ell$,
• then it surely belongs after $x$ at every level below $\ell$
• (but maybe not immediately after, so keep going)

6 Cost?

What good is a skip list, anyway?

• **insert, find, remove** seem hard to analyze
• will analyze *expected* performance over random choices of pillar heights
• will show that for skip list of $n$ elements, these three ops run in expected time $O(\log n)$
• just as good as worst-case performance of balanced trees!