Looking for Efficiency

• Dynamic (supports inserts and deletes)
• Efficient ==> guaranteed logarithmic time for execution
  – Treaps (bound on a randomly constructed BST)
  – BSTs
  – Red-Black Trees
  – B-Trees (2-3 trees, 3-4 trees, etc.)
• Find the key -or- you find its predecessor/successor
Looking for simplicity…

• Arrays
• Linked Lists
• Treaps
• Trees –
  – Red-Black Trees
  – Balanced Binary Search Trees
  – Rotating Trees (Splay Trees)
• How long does it take to implement these?
  – Can you do it closed-book?
  – How easy to debug/prove correctly implemented?
Solution: Skip Lists

• Efficient performance
  – Expected $O(\log n)$
  – Not guaranteed, but with high probability
  – Likelihood it is $>> O(\log n)$ is only $1-1/n^a$ (1 minus 1 over some polynomial in n; only tiny epsilon of prob that bigger than $\log n$)

• Dynamic insert, delete, search

• Randomized structure (like treaps)

• Simple implementation
Starting from Scratch

• Need to make a searchable structure
  – Can’t remember exact detail of trees, treaps, etc…
  – Has to be dynamic (size not sure), so arrays are not practical
  – Guess it will have to be a ---?
  – Linked List (doh)

• What problems do we have with a linked list
  – How efficient is it (how long to find something)?
  – Can we improve the efficiency?
  – How simple can we make the structure? “closed book” implementation

• Linked List = order n efficiency (theta-n)

• How to make a linked list better/faster?
  – More links: add “skip ahead” links? – too sophisticated
  – Build a tree on top of the structure? – too sophisticated
  – Add another list and link to it: simple, effective
Doubly-linked list

• Real example: 7th avenue line in New York City*
  – Subway is a kind of real-life skip list implementation
  – 4 sets of tracks make this possible

• Express lines, local lines
  – 14, 34, 42, 72, 96 (express)
  – 14, 23, 34, 42, 50, 59, 66, 72, 79, 86, 96, 103, 110, 116, 125 (local)
  – Common stops have links between them, so can quickly skip ahead using the express lines to go to closest stop and then switch over to the local for the destination stop.
  – That is, links between equal keys (from a linked-list perspective)

• We have roughly sqrt of express stops – could be better

*Thanks to Erik Demaine for this example/idea
Skip List: Search (x)

• Search process:
  – Go right in top list L (level 1) until going right would go too far;
  – Walk down to level 2 list
  – Walk right in level 2 until find (x)

• Best thing in the lists is to spread the keys out uniformly in the “extra” lists
  – Not so for NYC: theirs is based on popular stops
  – Spreading uniformly gives search cost = |L1| + |L2| / |L1|

• Let L2 = n, all stops and L1 can be whatever we want
  – Then want to minimize value of |L1| + n/|L1|
  – So up to constant factors, I can let |L1|=n/|L1|
  – Then by simplification, == L1^2 = n so L1 = sqrt(n)
  – And search cost ~ 2*sqrt n. (*2 because 2 lists)
Setting Up the Skip Lists

- Local line will have $\sqrt{n}$ values between each of the express stops (since distributing uniformly, and goal for express line is $\sqrt{n}$ stations at this stage…
- So search at most $\sqrt{n}$ stops in the Express line
- Followed by at most $\sqrt{n}$ stops in the local line
- $= 2 \times \text{Sqrt}(n)$ worst case cost
Improving Performance

• How can we improve our efficiency?
  – I want to get home ASAP
• Add another line, only this one will be a super-express
• If we had 2*sqrt(n) for 2 lines, then what would the cost be for 3 lines?
  – 3*cube-root(n)…. Etc!
• The idea is that the multiplicative sum of the ratios = n
  – Thus, for 2 lines: sqrt(n)*sqrt(n) = n ???
  – For 3 lines, cube-root(n)*sqrt(n)*sqrt(n) ????
  – So for “k” lines, k*k^th root(n) is our efficiency operationally
• Target is lg(n) efficiency – so if we set k=lg(n) then:
  – lg(n)*lg(nth)Root(n) = n^(1/lg(n)) => a^b = 2^(b lg a) => 2^(lg n/lg n)
  – => 2^1 = 2 => 2lg(n) is efficiency & our goal
Creating the Skip List

• Starting with \( \log(n) \) lists
  – Figure out which new elements get promoted to other, shorter lists
  – Maintain ratio of elements so that overall \( \log(n) \) ratio – ratio is 2:1 at each row all the way down…
  – \( R^{\log(n)} = n \Rightarrow \) (think: \( r-2 = 0 \), so \( R=2 \); see previous slide)
  – That is, ratio is \( \log(n) \) between each successive row:
    – All \( n \), \( \frac{1}{2} n \), \( \frac{1}{4} n \), \( \frac{1}{8} n \), etc. all the way through…
    – Shortest list has two elements, 1\(^{st}\) and last
  – Note how similar it is to a binary tree (except for repeated elements):
    – At depth “\( i \)” we have \( 2^i \) nodes (thus at level \( n \), have \( 2^n \) nodes)

• A special “far left” value is –infinity
  – Always promoted up to next highest list
  – Allows left-hand insertions since list traversal always starts at top left
Searching the Skip List

• Ex: search for 57 (x)
  – Start at 1\textsuperscript{st} level in 1\textsuperscript{st} node; 14\textless{}57;
  – Go right: 79 > 57 so (back at 14) go down to next level’s node 14
  – Go right: 50 < 57 so go right again; 79 > 57 so (back to 50): go down
  – 50 < 57 so go right; 66 > 57 so back to 50 & go down again...
  – 50 < 57 so go right; 57 = 57; Found (x)
Insertions into Skip List

- Need to maintain structure and ratios
  - to guarantee $WHP \ O(lg \ n)$

- Need method to resolve question of which insertions move into which of the “express” lines
  - Remember that all items are inserted into the lowest-level list (n)
  - This is called maintaining the invariant

- Insert(x) issues
  - Search(x) : example x=75
  - If found, notify and quit
  - If not found, insert (x) into bottom list (“maintain the invariant”)
  - But now unbalanced: and if I insert “k” elements, becomes very unbalanced at that next highest level since may have “k” items to walk between them
  - So how to decide when a new inserted item x gets moved to next highest level?
Maintaining List Ratios in Skip Lists

- Idea is that each level has $\frac{1}{2}$ the number of elements of the next level down
  - So to decide if new item X is to go into one level up, we need a 50-50 random choice, i.e., coin-flip equivalent

- Algorithm then is
  - Flip a coin (presumed fair coin)
  - If Heads, promote up and flip again (REPEAT)
  - If Tails, do nothing else at that level

- Probability over “k” insertions that the new element is promoted to the next level up is $\frac{1}{2}$

- Examples using inserts of:
  - 44 (tails; insert into bottom only)
  - 09 (heads; insert into bottom; insert into next level up; tails, stop)
  - $1/n$ probability of going up n levels in the list....
Skip List Deletion

- **Straightforward:**
  - Search \((X)\) in each of the lists;
  - Delete \((X)\) where found
  - No “rebalancing” is done…

- **Insertions are the tricky, randomized action**
  - Coin flip to promote or not
  - Averages out over many events
  - Special “negative infinity” value maintains leftmost entry point
Probability in Algorithms

• Many algorithms use probability to analyze the data structure’s efficiency

• Skip Lists: “with high probability”
  – Actual phrase with real mathematical meaning:
  – Extremely small likelihood it will not meet the $O(lg n)$ performance
  – Approximately $1/n^\alpha$ probability of being worse
  – As $\alpha \rightarrow \infty$ the probability becomes infinitesimally small
  – See proof of the probability in the Pugh’s Skip List PDF

• We will see basic probability used to determine expected performance of certain algorithms and/or data structures
Basic Probability Review

• Probability is based on likelihood an event will occur
  – Set $S$ of all possible outcomes of an experiment;
    • Example: Dice $S=\{1,2,3,4,5,6\}$ or Coin $S=\{\text{Heads}, \text{Tails}\}$
  – $E$ is a subset of $S$ describing a particular event or outcome
    • Example: Dice roll is even $E=\{2,4,6\}$

• Probability of an event is proportional to all possible events
  – $P(\text{even dice roll})=\{2,4,6\}/\{1,2,3,4,5,6\}=\{3 \text{ events}\}/\{6 \text{ events}\}=50\%$
  – Assumes random events and fair dice – each event is equally probable
  – Probabilities range from 0 to 1, i.e., $0 \leq P(E) \leq 1$
  – If the probability of an event is $P(E)$, the probability it will NOT occur is $1-P(E)$

\[
P(E) = \frac{\# \text{ times event } E \text{ occurs in } S}{\text{All of } S}
\]