Skip Lists
CMSC 420
Linked Lists Benefits & Drawbacks

• Benefits:
  – Easy to insert & delete in $O(1)$ time
  – Don’t need to estimate total memory needed

• Drawbacks:
  – Hard to search in less than $O(n)$ time
    (binary search doesn’t work, eg.)
  – Hard to jump to the middle

• Skip Lists:
  – fix these drawbacks
  – good data structure for a dictionary ADT
Skip Lists

- Invented around 1990 by Bill Pugh
- Generalization of sorted linked lists – so simple to implement
- Expected search time is $O(\log n)$
- *Randomized* data structure:
  - use random coin flips to build the data structure
Perfect Skip Lists

header

2
2
2

10
15
16

31
31
71
96

sentinel
Perfect Skip Lists

- Keys in sorted order.
- $O(\log n)$ levels
- Each higher level contains $1/2$ the elements of the level below it.
- Header & sentinel nodes are in every level
Perfect Skip Lists, continued

• Nodes are of variable size:
  - contain between 1 and \(O(\log n)\) pointers

• Pointers point to the start of each node
  (picture draws pointers horizontally for visual clarity)

• Called *skip lists* because higher level lists let you skip over many items
Perfect Skip Lists, continued

Find 71

When search for k:
  If \( k = \text{key} \), done!
  If \( k < \text{next key} \), go down a level
  If \( k \geq \text{next key} \), go right
In other words,

- To find an item, we scan along the shortest list until we would “pass” the desired item.

- At that point, we drop down to a slightly more complete list at one level lower.

- Remember: sorted sequential searching...

```c
for(i = 0; i < n; i++)
    if(X[i] >= K) break;
if(X[i] != K) return FAIL;
```
Perfect Skip Lists, continued

Find 96

When search for k:
- If $k = \text{key}$, done!
- If $k < \text{next key}$, go down a level
- If $k \geq \text{next key}$, go right
Search Time:

- $O(\log n)$ levels --- because you cut the # of items in half at each level

- Will visit at most 2 nodes per level: If you visit more, then you could have done it on one level higher up.

- Therefore, search time is $O(\log n)$. 
Insert & Delete

• Insert & delete might need to rearrange the entire list

• Like Perfect Binary Search Trees, Perfect Skip Lists are too structured to support efficient updates.

• Idea:
  - Relax the requirement that each level have exactly half the items of the previous level
  - Instead: design structure so that we expect 1/2 the items to be carried up to the next level
  - Skip Lists are a randomized data structure: the same sequence of inserts / deletes may produce different structures depending on the outcome of random coin flips.
Randomization

- Allows for some imbalance (like the +1 -1 in AVL trees)
- Expected behavior (over the random choices) remains the same as with perfect skip lists.
- Idea: Each node is promoted to the next higher level with probability $1/2$
  - Expect $1/2$ the nodes at level 1
  - Expect $1/4$ the nodes at level 2
  - ... 
- Therefore, expect # of nodes at each level is the same as with perfect skip lists.
- Also: expect the promoted nodes will be well distributed across the list
Randomized Skip List:
Insertion:

Insert 87
**Insertion:**

Insert 87

Find $k$
Insert node in level 0

```javascript
let i = 1
while FLIP() == "heads":
    insert node into level $i$
    $i$++
```

Just insertion into a linked list after last visited node in level $i$
Deletion:

Delete 87
Deletion:

Delete 87
There are no “bad” sequences:

- We expect a randomized skip list to perform about as well as a perfect skip list.

- With some very small probability,
  - the skip list will just be a linked list, or
  - the skip list will have every node at every level
  - These degenerate skip lists are very unlikely!

- Level structure of a skip list is independent of the keys you insert.

- Therefore, there are no “bad” key sequences that will lead to degenerate skip lists
Skip List Analysis

• Expected number of levels = $O(\log n)$
  - $E[\# \text{ nodes at level } 1] = n/2$
  - $E[\# \text{ nodes at level } 2] = n/4$
  - ... 
  - $E[\# \text{ nodes at level } \log n] = 1$

• Still need to prove that # of steps at each level is small.
Consider the *reverse* of the path you took to find $k$:

Note that you *always* move up if you can. (because you always enter a node from its topmost level when doing a find)
Analysis, continued...

- What’s the probability that you can move up at a given step of the reverse walk?
  
  \[ P = 0.5 \]

- Steps to go up \( j \) levels =
  
  Make one step, then make either
  
  \( C(j-1) \) steps if this step went up [Prob = 0.5]
  
  \( C(j) \) steps if this step went left [Prob = 0.5]

- Expected # of steps to walk up \( j \) levels is:
  
  \[ C(j) = 1 + 0.5C(j-1) + 0.5C(j) \]
Analysis Continue, 2

- Expected # of steps to walk up $j$ levels is:
  \[ C(j) = 1 + 0.5C(j-1) + 0.5C(j) \]

So:

\[ 2C(j) = 2 + C(j-1) + C(j) \]

\[ C(j) = 2 + C(j-1) \]

Expected # of steps at each level = 2

- Expanding $C(j)$ above gives us: $C(j) = 2j$

- Since $O(\log n)$ levels, we have $O(\log n)$ steps, expected
Implementation Notes

• Node structures are of variable size

• But once a node is created, its size won’t change

• It’s often convenient to assume that you know the maximum number of levels in advance (but this is not a requirement).