Skip Lists

Adversarial Behavior

Several data structures have potentially **very bad** runtime guarantees if the elements they store are inserted in "adversarial order."

- Given n elements, how could you make the most unbalanced Binary Search Tree out of those elements?
- Given a Hash Table with a particular collision resolution policy, how could you choose elements to insert to make the lookup as slow as possible?

Adversarial Behavior

Several data structures have potentially **very bad** runtime guarantees if the elements they store are inserted in "adversarial order."

- Given n elements, how could you make the most unbalanced Binary Search Tree out of those elements? **Insert in sorted order to build a linked list.**
- Given a Hash Table with a particular collision resolution policy, how could you choose elements to insert to make the lookup as slow as possible? **Insert a bunch of elements following the same probe sequence, maximizing expected probe length.**

Adversarial Behavior: A Problem?

We have a few different "defenses" against this adversarial behavior.

- Self-balancing BSTs keep tree height logarithmic with constant-time modifications.
- Assumptions that elements are inserted into Hash Tables somewhat randomly "smooths out" the probe sequences

Deterministic Data Structures

So far, all data structures are fully determined by the data that we insert and the order in which we insert it.

- Shape of a tree
- Slots occupied in Hash Tables and Bloom Filters
- Graphs/Lists/Arrays/Heaps

This can be perilous in the "worst case" cases

Randomized Data Structures

Data Structures where some element of the organization of the data is chosen **randomly** so that the structure is not readily predictable from the elements it contains.

- Sometimes we'll make bad random choises, yes, but this happens anyways with badly ordered inputs
- Sacrificing a *guaranteed optimal solution* for an *expected reasonable* one.

Doing Better Than a Linked List

Linked Lists are interesting academically (and work well as **Double-Ended Queues**) but have significant shortcomings.

- Linear time random access
- Binary search can't be performed in $O(\log n)$ time because of the pointer traversal

Could we come up with an idea that uses **less pointer overhead** than a BST but still allows for $O(\log n)$ searching through sorted data?

Idea: Finding Your Spot in a Book

Your bookmark has fallen out of your book! How do you find where you were?

- **Binary Search?** ❌ No! Too much possibility of spoiling something important.
- **Skip a bunch of pages?** ✅
  - At the start, skip forward a bunch
  - The closer you feel you are to the right spot, the narrower each "skip" gets
  - If you overshoot, you're not likely to overshoot by much.

Skip Lists: Main Idea

Take a Linked List, but enable jumps of different sizes by creating nodes of different heights.

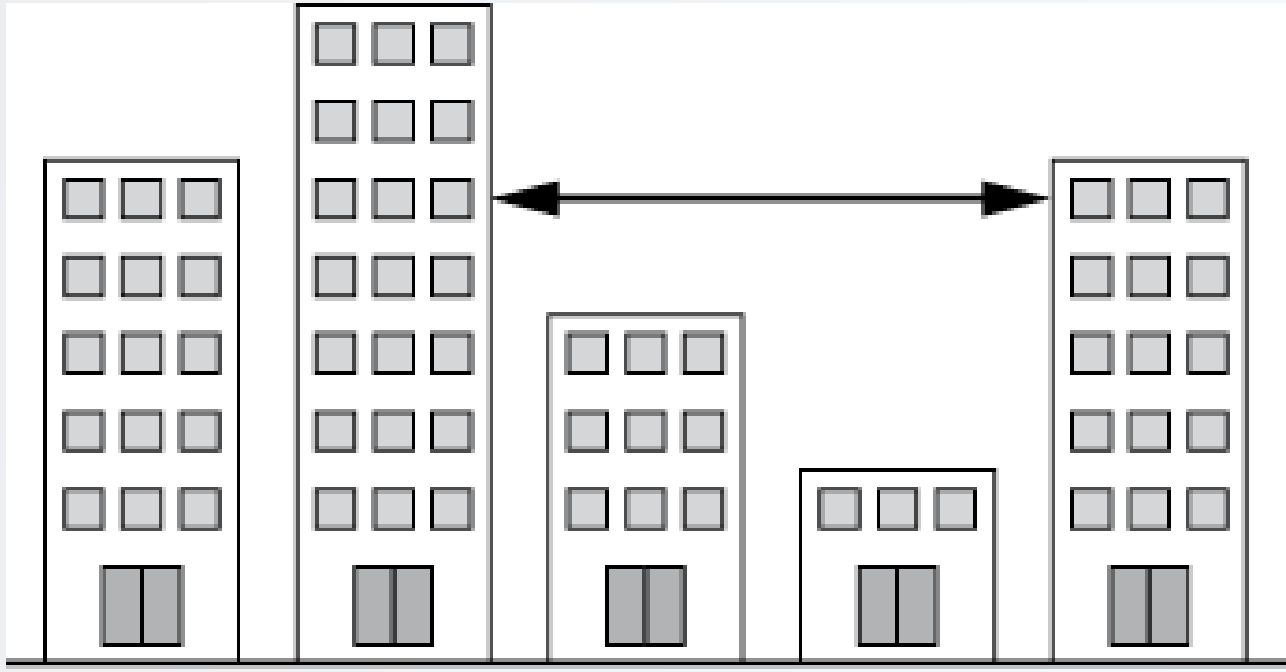
- Possible to jump from a node of height k to any node of height k or lower.
- Make tall nodes rare and short nodes common
- Going between tall nodes represent large "skips"; between short nodes are short "skips."

This describes a **Skip List.**

Sending Flashlight Signals Between Tall Buildings

How far you can send a signal depends on your current floor and the heights of buildings in the path.

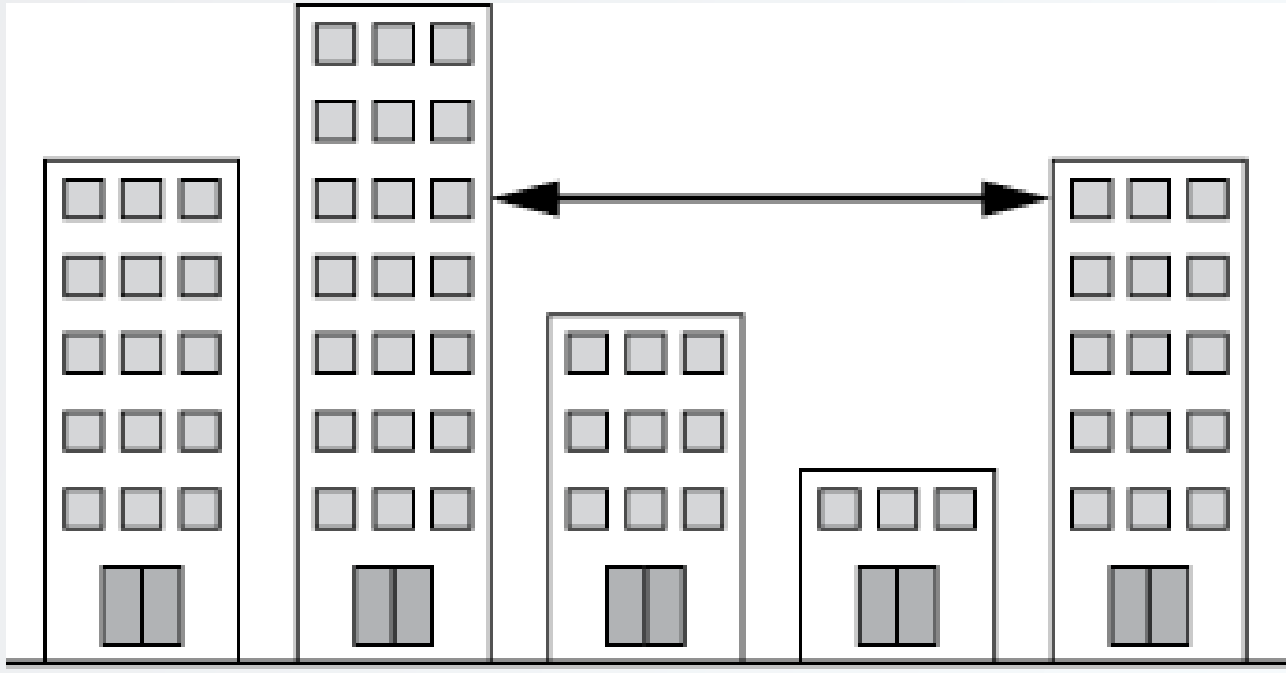
- First floor → only send to next-door neighbor.
- Fifth floor → send a message over top of smaller intermediate neighbors or move down to talk to smaller neighbors.



Sending Flashlight Signals Between Tall Buildings

Key ideas:

- Taller buildings are more "expensive" in terms of cost to build and space taken, but they are more powerful.
- Taller buildings are only useful if there are relatively large gaps between them.



Structure of a Skip List

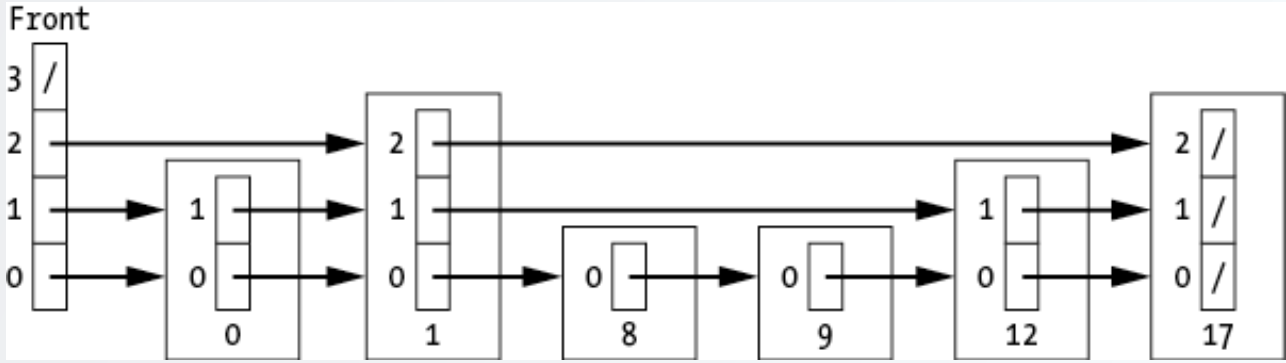
```java
public class SkipList {
    int topLevel;
    int maxLevel;
    SkipListNode front;
}
```

```java
public class SkipListNode<T> {
    T element;
    int height;
    SkipListNode<T>[] next;
}
```

- Nodes always store values in **increasing sorted order.**
- Use a sentinel `front` node to point to the first node of each height
- Each node stores an array of size `height + 1` for all next nodes at heights $0, 1, 2, \ldots$ `height`
- Crucially, the pointers in `next` will often point to different nodes!

Example Skip List

The full skip list contains six total values. This is equivalent to the list following all height 0 pointers.

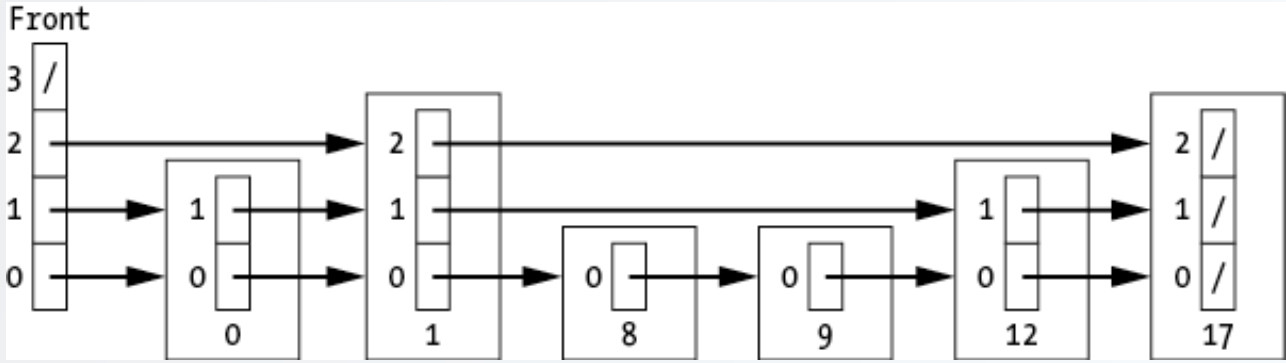


Example Skip List

The list of height 1 pointers reaches nodes 0, 1, 12, and 17.

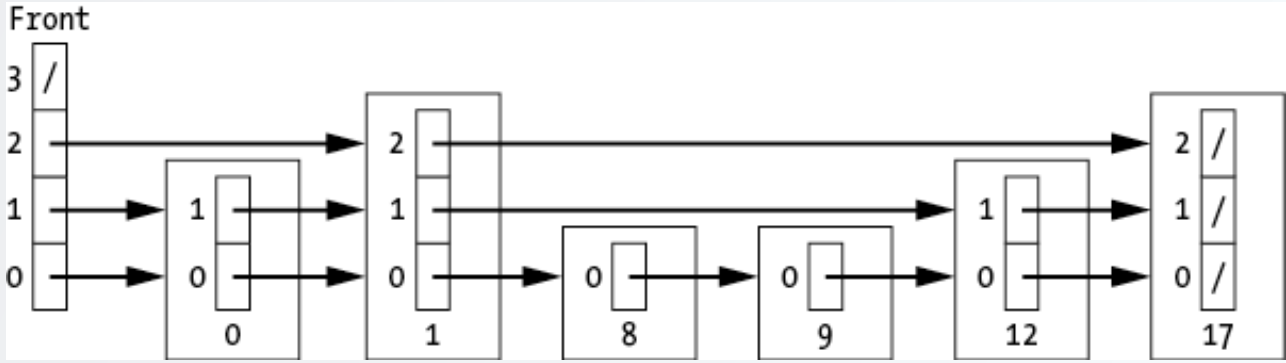
The list of height 2 pointers reaches nodes 1 and 17.

The list of height 3 is empty in this case.



Example Skip List

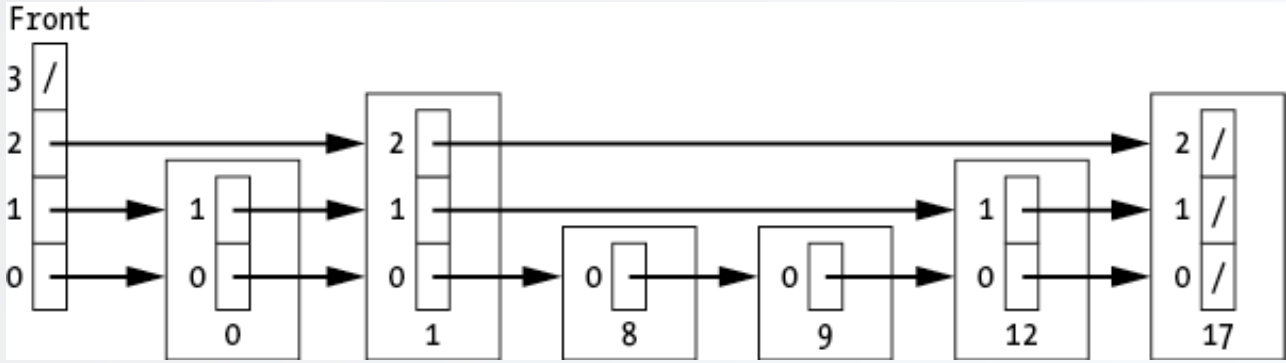
What's the smallest number of pointers we need to follow to reach node 12?



Example Skip List

What's the smallest number of pointers we need to follow to reach node 12?

Front to 1 using level 2, then 1 to 12 using level 1 (although we would need to observe that the next level 2 pointer would have taken us a step too far.)



Vital Ideas

Skip lists should have:

- Nodes of many different heights
- Decreasing frequency for increasing heights
- Maximal spacing between nodes of the same height

These guarantees can be obtained (in expectation) by choosing the heights of nodes randomly. More on this later.

Searching a Skip List

```
public SkipListNode search(T target) {
    int level = topLevel;
    SkipListNode current = front;

    while (level >= 0) {
        while (current.next[level] != null && current.next[level].key < target) {
            current = current.next[level];
        }
        level--;
    }

    SkipListNode result = current.next[0];
    if (result != null && result.element == target) {
        return result;
    }
    return null;
}
```

Searching a Skip List

```
int level = topLevel;
SkipListNode current = front;
...
```

Start at the front of the list with a maximum height for the longest jumps possible...

Searching a Skip List

```
...
while (level >= 0) {
    while (current.next[level] != null && current.next[level].key < target) {
        current = current.next[level];
    }
    level--;
}
...
```

- **Inner Loop:** proceed along current level as long as the next "skip" doesn't take you past the desired element.
- **Outer Loop:** repeat the inner loop at descending levels, taking shorter and shorter steps until we find what we're looking for.

Searching a Skip List

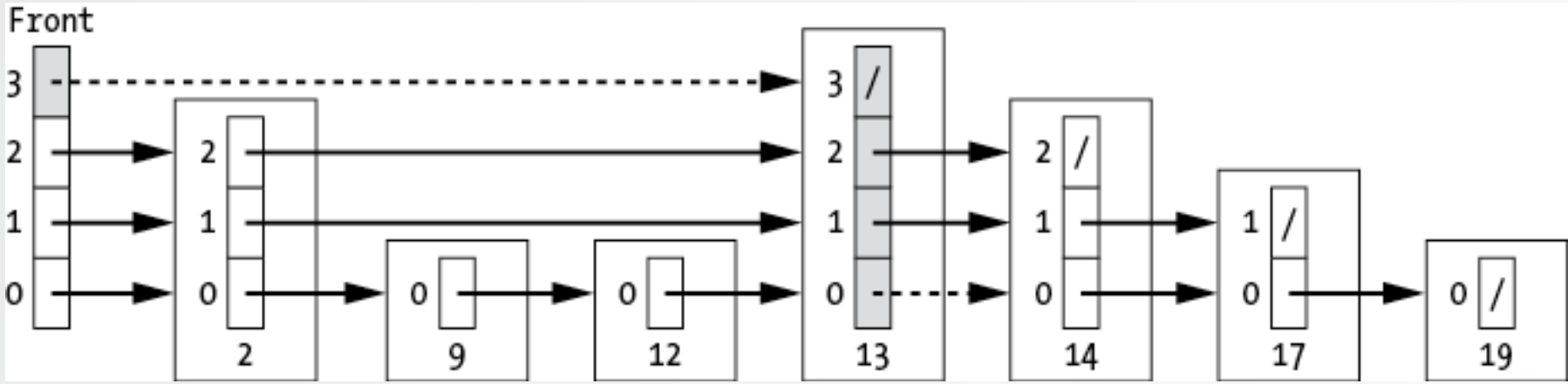
```
...
SkipListNode result = current.next[0];
if (result != null && result.element == target) {
    return result;
}
return null;
```

If the target element is there, it's in the node **after** the `current` node where we stopped.

- If that next node has it, return that next node.
- Otherwise, report `null` to indicate absence.

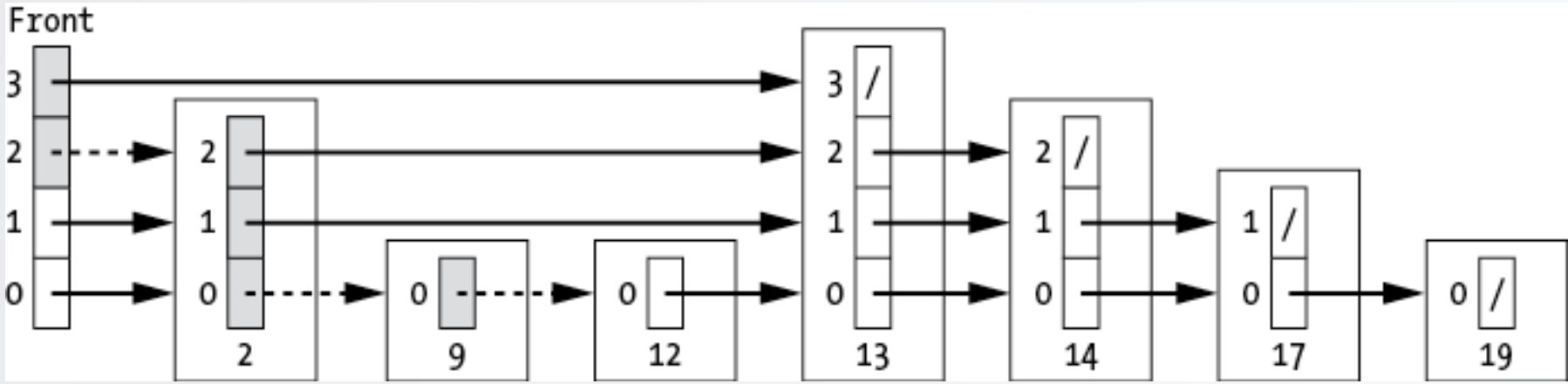
Example: Searching

Searching for the value `13` in our list.



Example: Searching

Searching for the value `12` (or `11`) in our list.



Search always starts from the top-left. It proceeds to the right and down.

Inserting a Node in a Linked List

This is for a regular linked list:

```
public void insertBefore(Node n, T element) {
    Node current = head;
    while (current != null && current.next != n) {
        current = current.next;
    }
    Node newNode = new Node(element);
    newNode.next = current.next;
    current.next = newNode;
}
```

"Get up to right before where you need to insert, then create a new node and stitch the pointers together."

Inserting a Node: Helper

Find the node AT EACH HEIGHT! that would be an immediate predecessor of the new element to insert.

```
private SkipListNode[] predecessors(T target) {
    int level = topLevel;
    SkipListNode current = front;
    SkipListNode[] predecessors = front.copy();

    while (level >= 0) {
        while (current.next[level] != null && current.next[level].key < target) {
            current = current.next[level];
        }
        predecessors[level] = current
        level--;
    }
    return predecessors;
}
```

Inserting a Node: The Meat

```
public void insert(T element) {
    SkipListNode<T>[] predecessors = predecessors(element);
    int newLevel = randomLevel(); // pick a random level
    if (newLevel > list.topLevel) {
        list.topLevel = newLevel;
    }
    SkipListNode<T> newNode = SkipListNode<>(key, value, newLevel);
    int j = 0;
    while (j <= newLevel) {
        newNode.next[j] = predecessors[j].next[j];
        predecessors[j].next[j] = newNode;
        j = j + 1;
    }
}
```

Inserting a Node: The Meat

```
public void insert(T element) {
    ...
    int newLevel = randomLevel(); // pick a random level
    ...
}
```

We'll **randomly** select the height of the new node that we're creating. More on this in a minute.

Inserting a Node: The Meat

```
public void insert(T element) {
    ...
    if (newLevel > list.topLevel) {
        list.topLevel = newLevel;
    }
    ...
}
```

If we have a new tallest node height, we'll want to update our state properly to avoid bugs later on.

Inserting a Node: The Meat

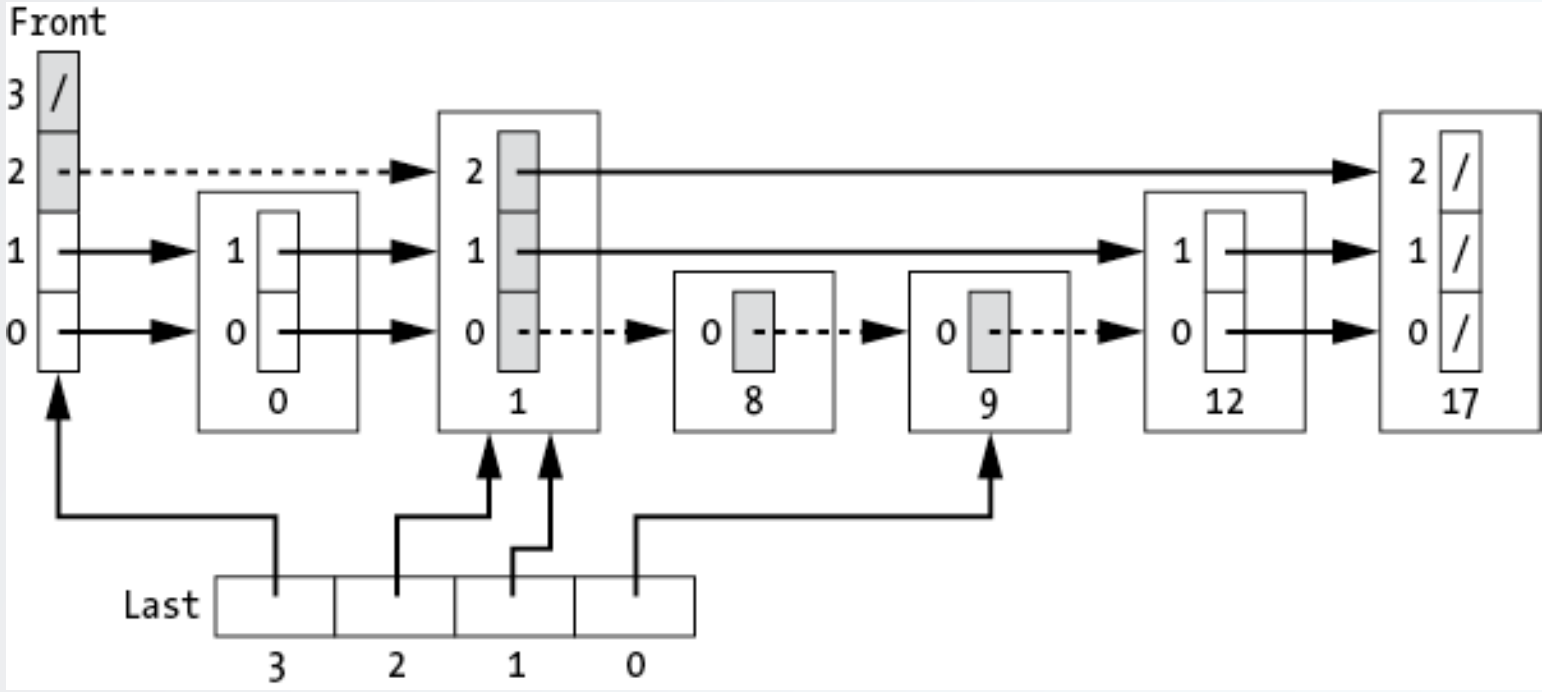
```
public void insert(T element) {
    ...
    while (j <= newLevel) {
        newNode.next[j] = predecessors[j].next[j];
        predecessors[j].next[j] = newNode;
        j = j + 1;
    }
    ...
}
```

Walk down the levels of the node, inserting this node in the typical "linked list" way. Note that this is only modifying

Inserting a Node: Demo

If we insert `10`, it might "block the view" of: `front`, node containing `1`, and node containing `9`.

The actual "blocking" depends on the height of the new node.



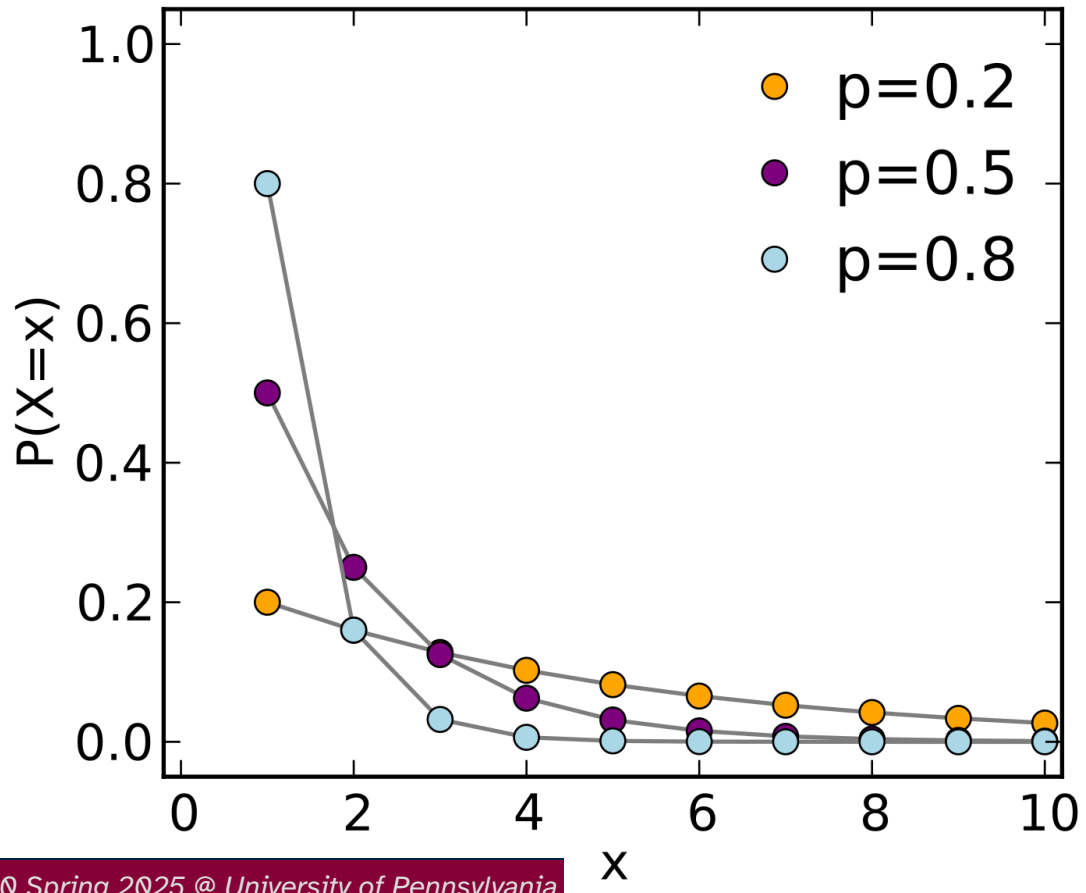
Picking the Height

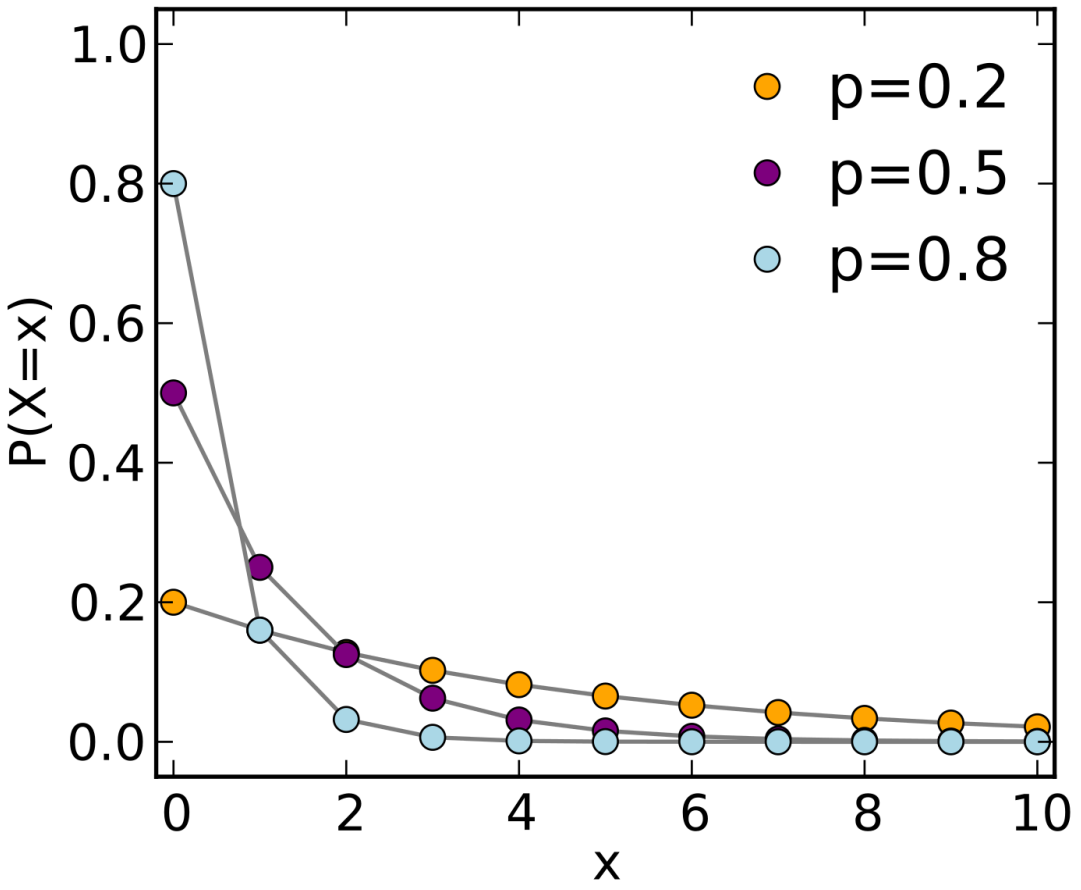
```
private int generateHeight(double p) {
    int height = 0;
    double draw = Math.random(); // [0, 1)
    while (draw < p) {
        height++;
        draw = Math.random();
    }
    return height;
}
```

"Flip a weighted coin until you stop getting heads. The number of successes is the height of the node. Higher p gives higher nodes."

Height Distribution

This is a **geometric distribution;** taller nodes are increasingly rare.





Height Distribution

This is a **geometric distribution;** the average height of a node will be $\frac{1}{1-p}$.

Probability	Average Height/# Pointers per Node
1/2	2
1/e	1.58
1/4	1.33
1/8	1.14
1/16	1.33

The expected search length of n elements is $\leq \frac{\log_{1/p} n}{p} + \frac{1}{1-p}$. Empirically speaking, $p = 1/4$ gives good speed, and that's better than the 1.5 pointers/node needed for BSTs.

Runtimes, Empirically

Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
recursive 2–3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self–adjusting trees:			
top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Table 2 - Timings of implementations of different algorithms

Stolen from https://www.cs.ucdavis.edu/~amenta/w04/skiplists.pdf, itself adapted from the original designer's analysis.

Summary

- **Linked Lists** do not provide fast random access and **BSTs** have lots of pointers plus complex implementations to maintain balance.
- **Randomized Data Structures** have structures that are not fully determined by the elements they contain, which can prevent against adversarial behavior (if not unlucky behavior.)
- **Skip Lists** enable $O(\log n)$ search, insertion, and deletion with fewer pointers and comparable/better runtime when compared to trees.