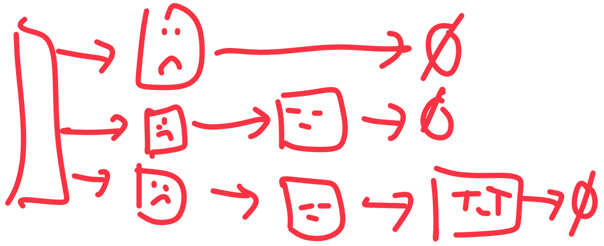
Data Structures and Algorithms
Skip List

CS 225
Brad Solomon

December 6, 2024

The last new content lecture



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Capstone probability lectures with a literature example — the Skip List!

Review fundamentals of probabilistic data structures with the skip list

Conceptualize Skip List ADT functions

Analyze efficiency of skip list while reviewing fundamentals of probability

The skip list is not on the final exam!

Where it all began… A faulty list

Imagine you have a list ADT implementation ***except***…

Every time you called **insert**, it would fail 50% of the time.

It turns out this system is also useful as an alternative linked list! **How?**

An alternative linked list

Goal: Visit nodes in my linked list in, **on average**, $log\ n$ steps

Big Picture: I need a way to access nodes **X** positions past the head

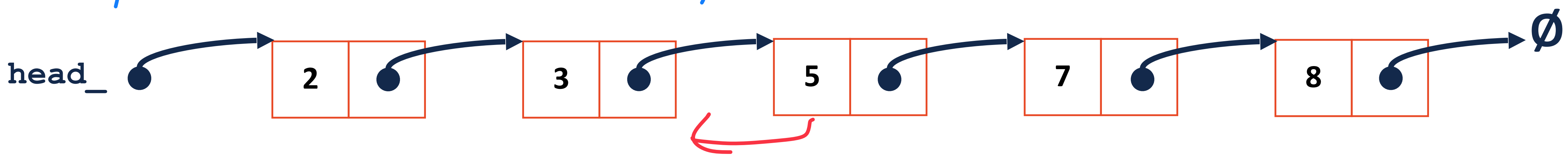
n/2

Pointer to X

X

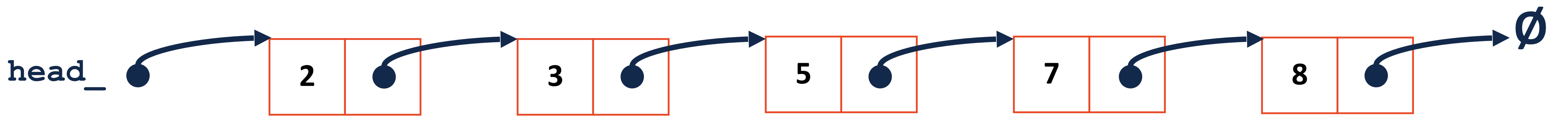
midpoint pointer

Max search is n/2



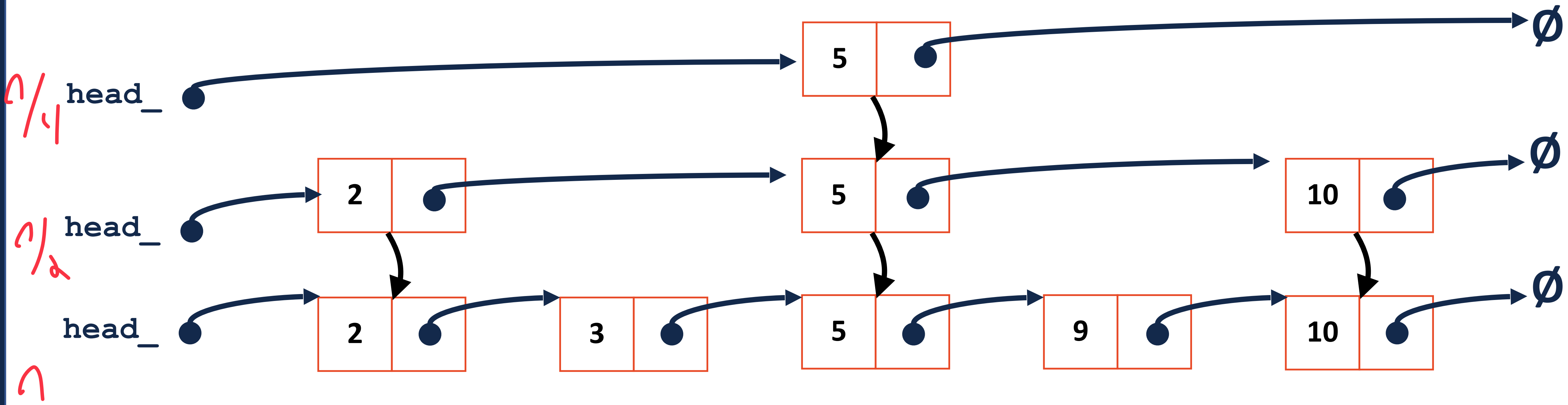
Linked List with ‘Checkpoints’

With some small overhead costs, we can store **checkpoints**.



Linked List with Perfect Checkpoints

For optimal checkpoints, we want half the number of items at each level.

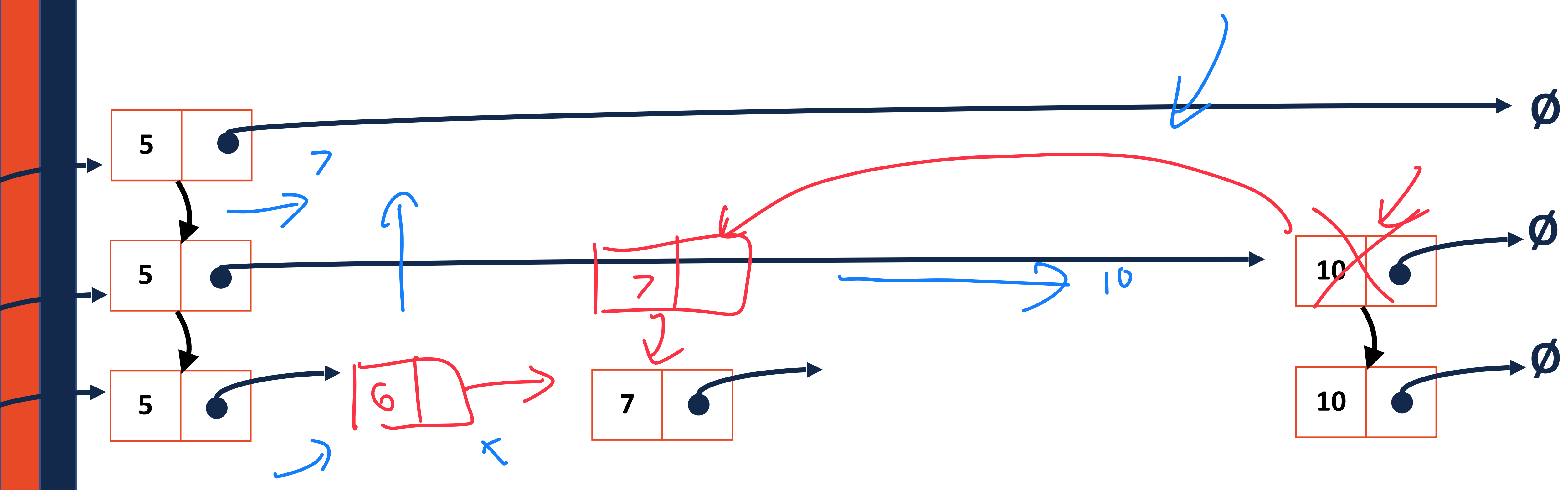


Linked List with Perfect Checkpoints

For optimal checkpoints, we want half the number of items at each level.

Maintaining this while inserting and deleting is too costly!

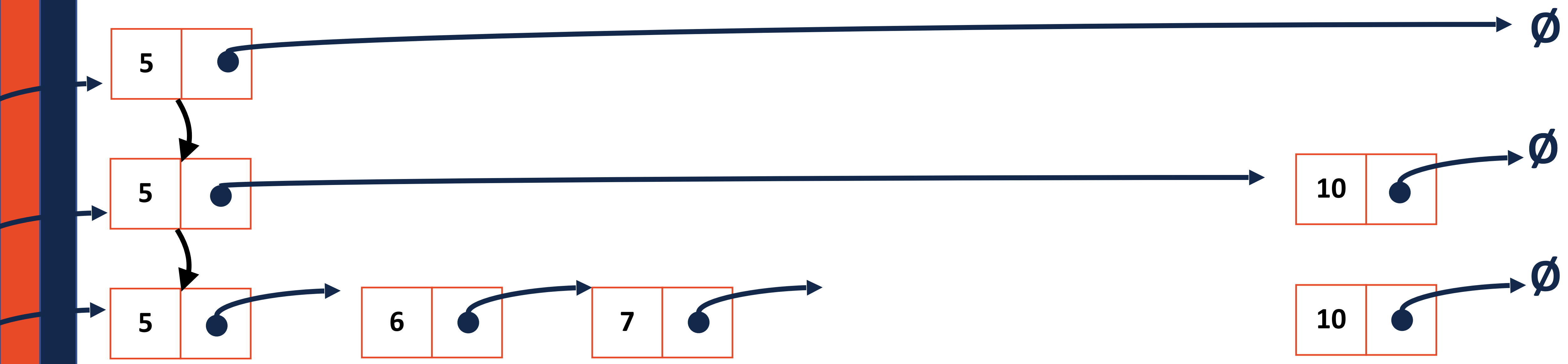
slow!



Linked List with Perfect Checkpoints

For optimal checkpoints, we want half the number of items at each level.

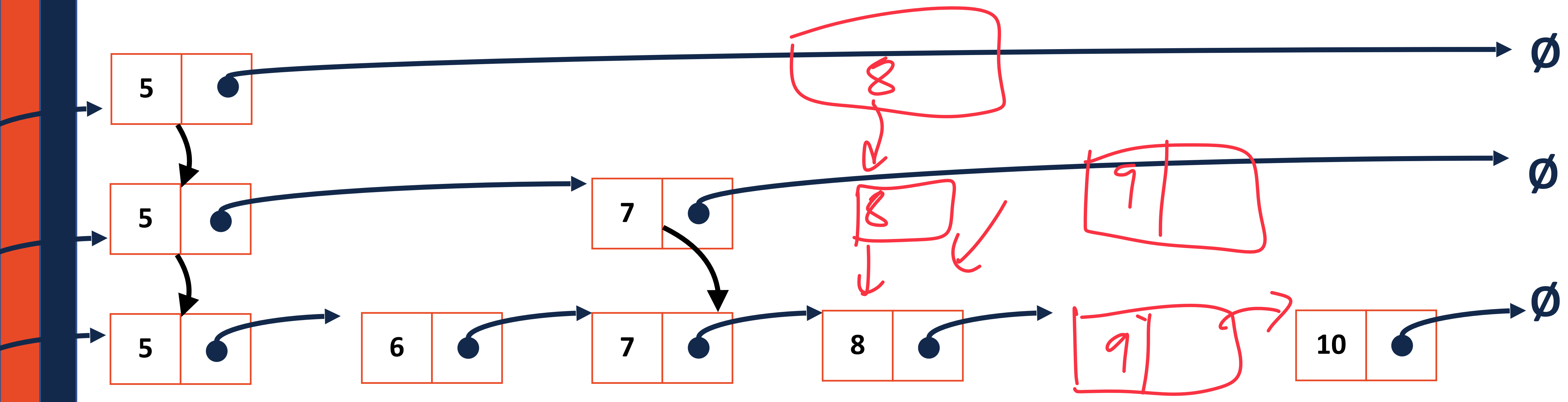
Maintaining this while inserting and deleting is too costly!



Linked List with Perfect Checkpoints

For optimal checkpoints, we want half the number of items at each level.

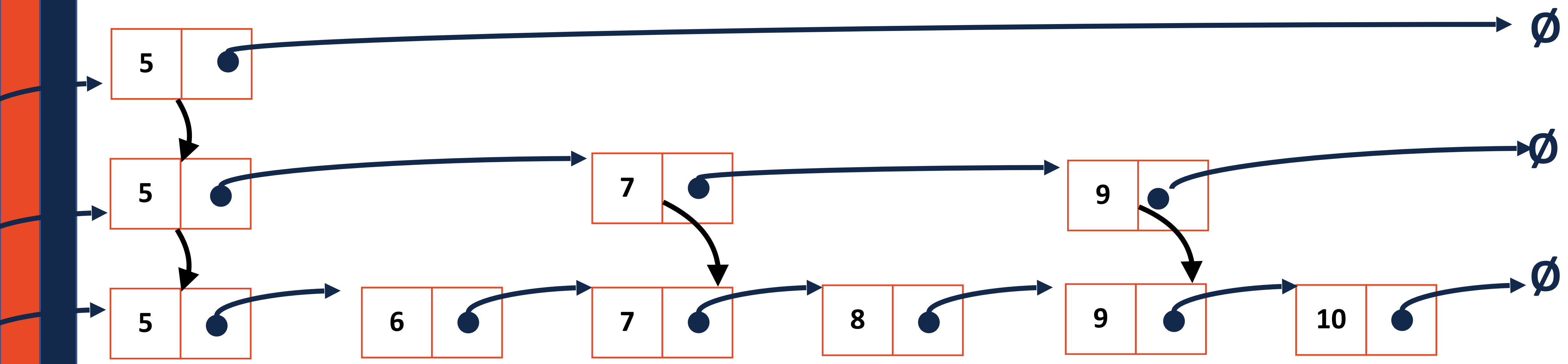
Maintaining this while inserting and deleting is too costly!



Linked List with Perfect Checkpoints

For optimal checkpoints, we want half the number of items at each level.

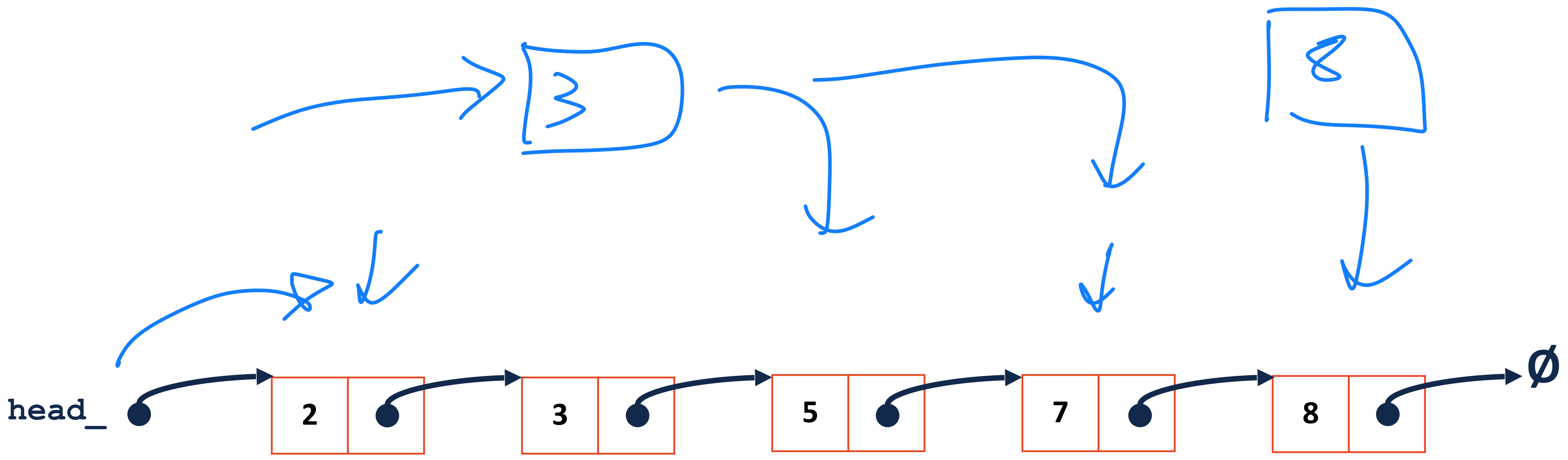
Maintaining this while inserting and deleting is too costly!



Linked List with Random Checkpoints

Problem: Having an optimal set of checkpoints is costly to maintain

Solution: Randomly choose checkpoints

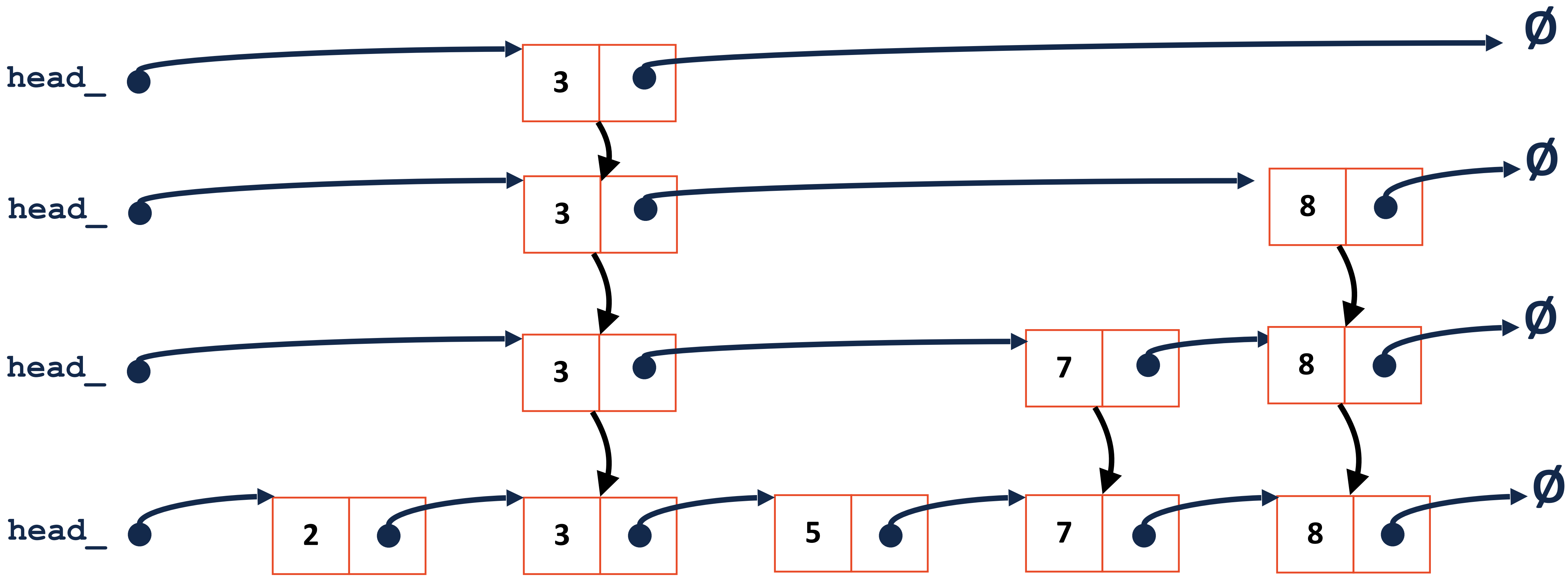


Linked List with Random Checkpoints

Instead of having **exactly** half each level, let's have **approximately** half!

To analyze runtimes we use: Expectation → Avg Case

try   Then use



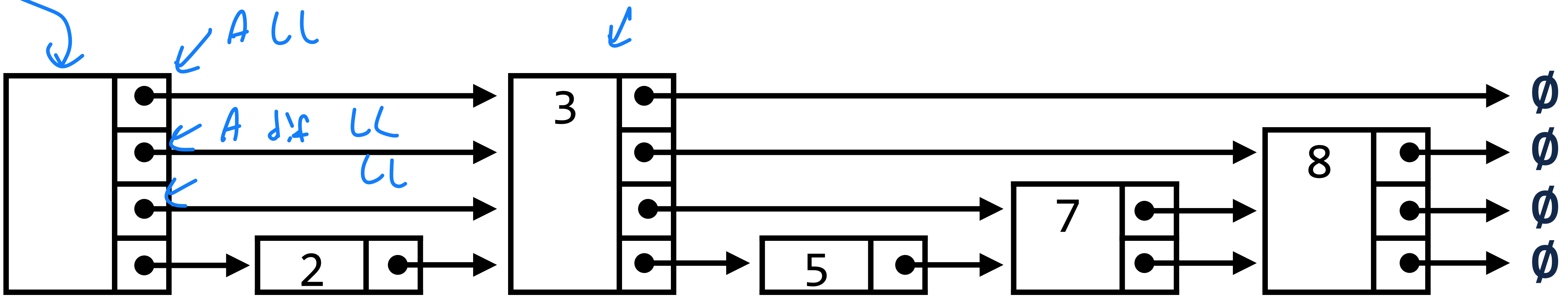
The Skip List

An ordered linked list where each node has variable size

Each node has at most one key but an arbitrary number of pointers

The decision for height is **randomized**

Claim: The **average** time to find, insert, or remove is $log\ n$



The Skip List

What would a SkipNode class look like?

↳ Key (T)

↳ An array or vector of pointers (a different next)

How about the SkipList class?

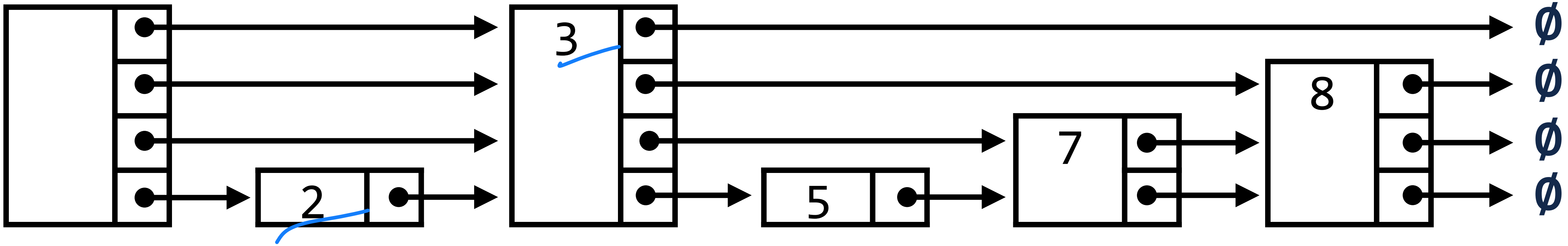
SkipNode * head

imp dependent:

int max (max height)

float C (coin flip constant)

↳ Decides random height



Skip List

```
template <class T>
class SkipList{
  public:
    class SkipNode{
      public:
        SkipNode(){
          next.push_back(nullptr);
        }

        SkipNode(int h, T & d){
          data = d;
          for(int i = 0; i <= h; i++){
            next.push_back(nullptr);
          }
        }
        T data;
        std::vector<SkipNode*> next;
    };

    int max; // max height
    float c; //update constant
    SkipNode* head;
    ...

```

head → ∅

SkipNode

SkipList

Skip List ADT

Find

Insert

Remove

Constructor

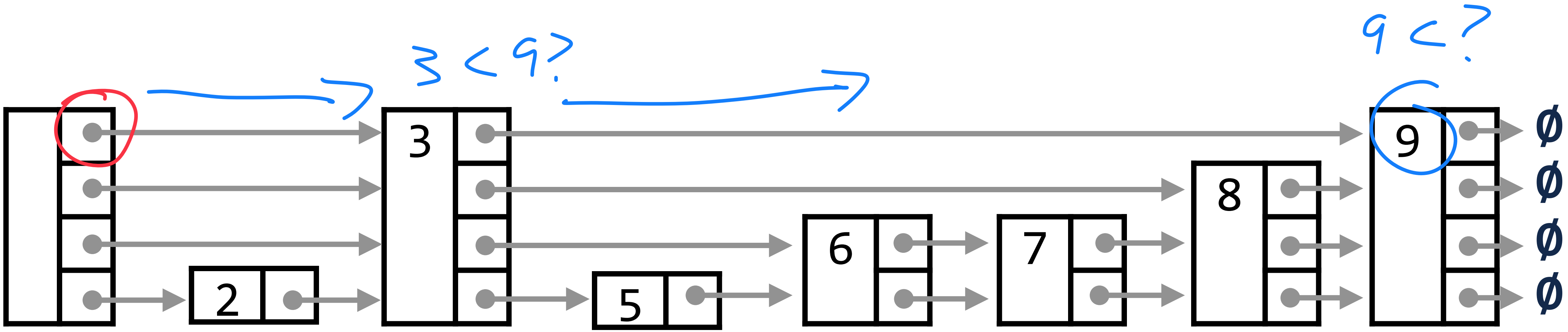
Skip List Find

Find(9)

1) Start at head & top level

↳ Has fewest nodes

2) At each level, it's LL find()* (ordered)



Skip List Find

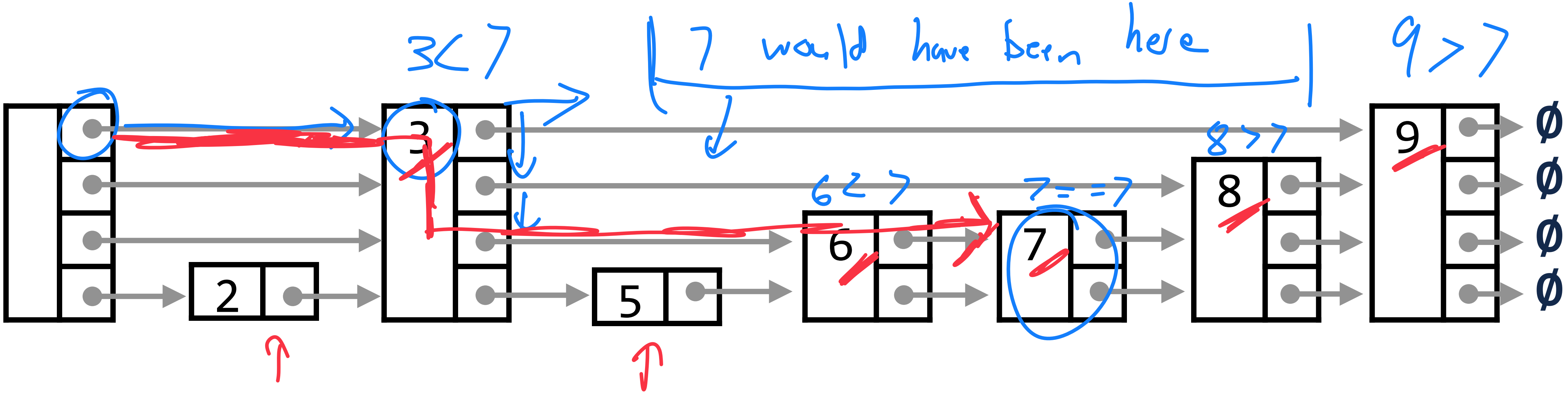
`Find(7)`

Starting at top level… if next node's key matches, done!

If key smaller than next node's key, **move down a level**

If key larger than next node's key, **go to next node at current level**

Q: Is this any more efficient? 8 & 9 looked up but skip 2 & 5



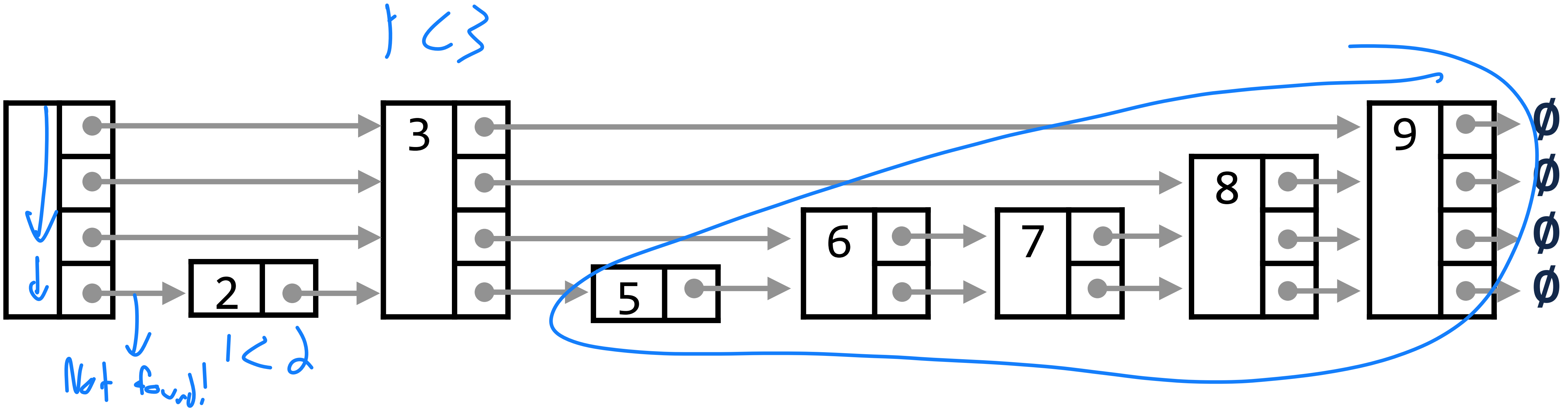
Skip List Find

Find(1)

Starting at top level… if next node's key matches, done!

If key smaller than next node's key, **move down a level**

If key larger than next node's key, **go to next node at current level**

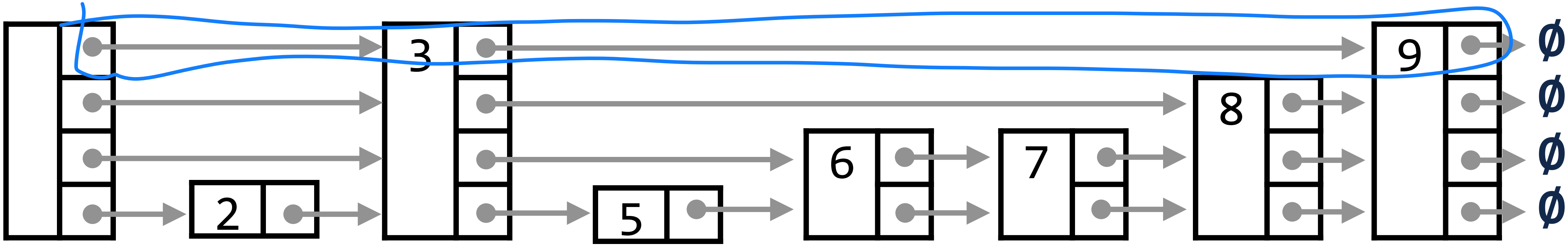


Skip List Find

Could you code up Skip List Find?

1) Linked list find on ordered data

2) Track current level & walk down it



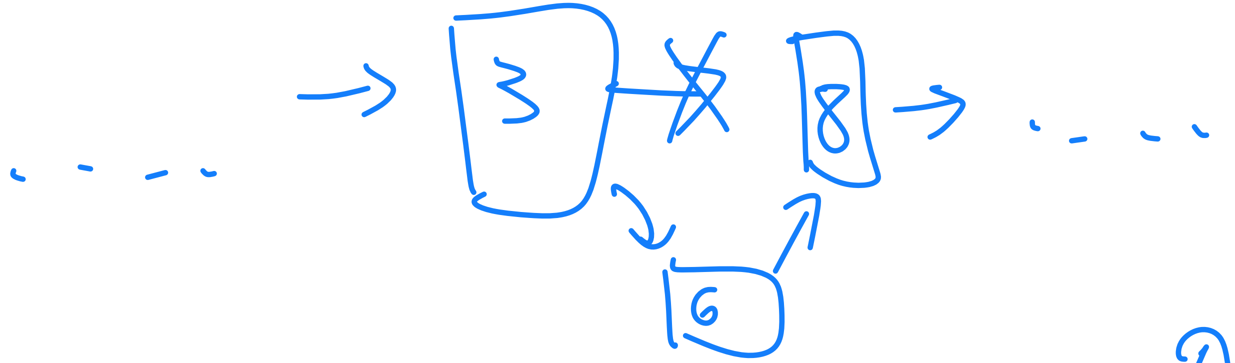
Skip List Insert

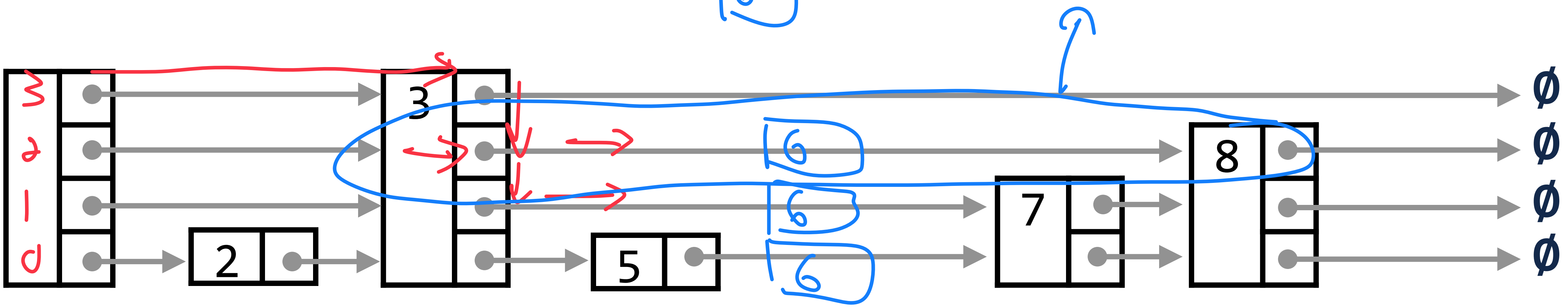
Insert(6)

↳ Random height is 3

0) Randomly determine height

1) Start @ top level and find item





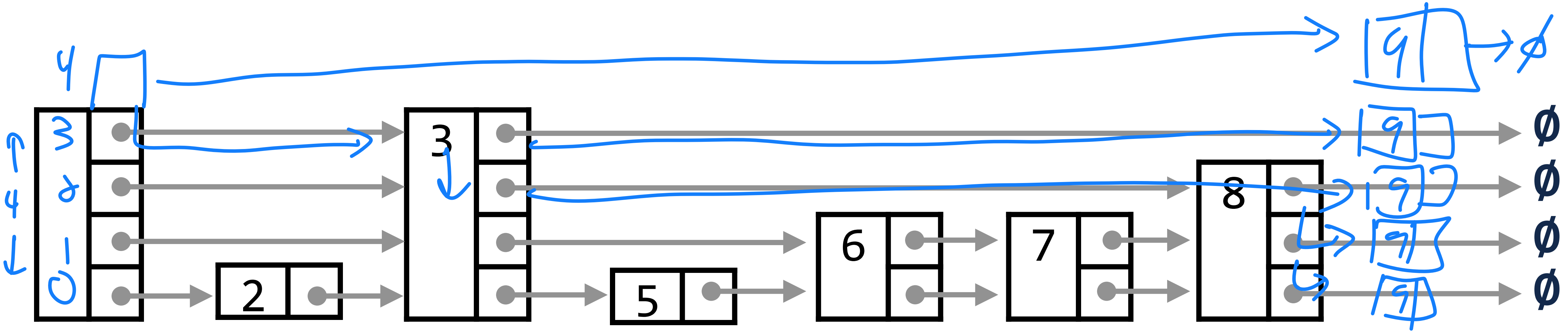
Skip List Insert

Insert(9)

↳ height = 5

Randomly generate height for insert

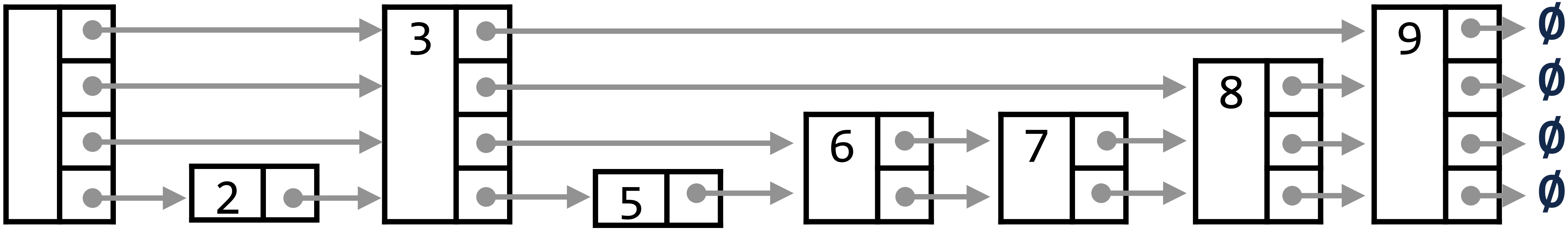
Use Find() logic but insert at every list with height >= random



Skip List Insert

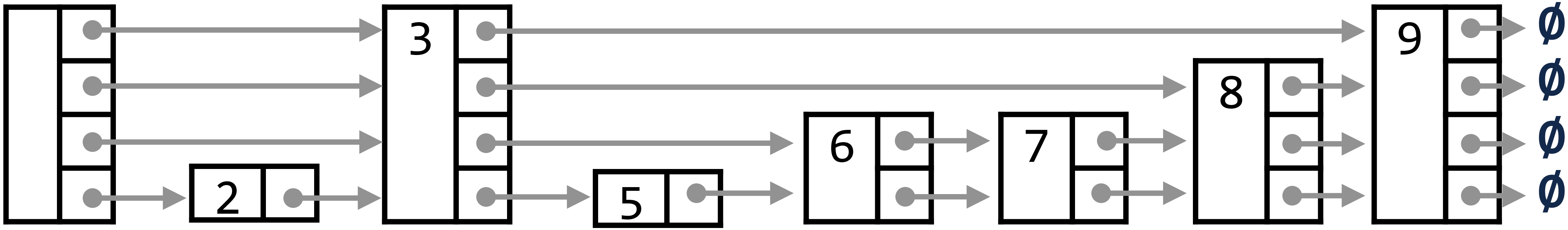
Randomly generate height for insert

Use Find() logic but insert at every list with height >= random



Skip List Insert

Could you code up Skip List Insert?



Skip List Remove

Remove (9)

Use find logic but remove!

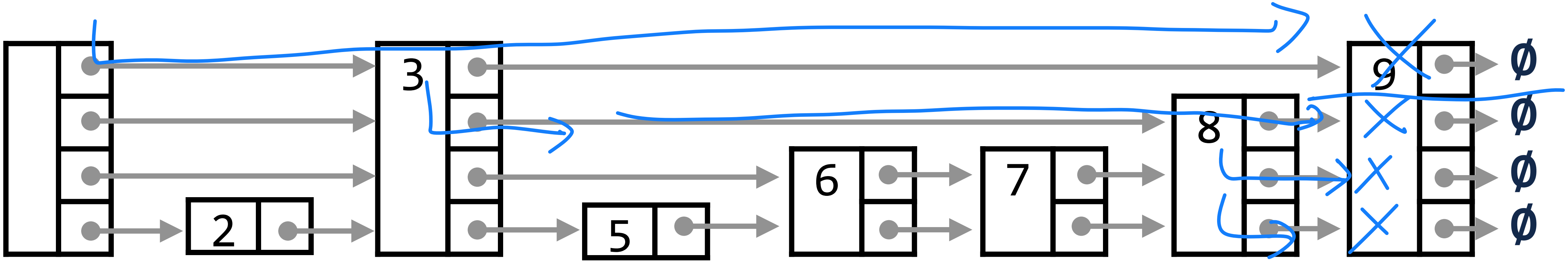
D → 3 → 9 (remove pointer

3 → 8 → 9 (remove pointer)

8 → 9 ↓

8 → 9 ↓

↳ at level 0, delete node before remove pointer

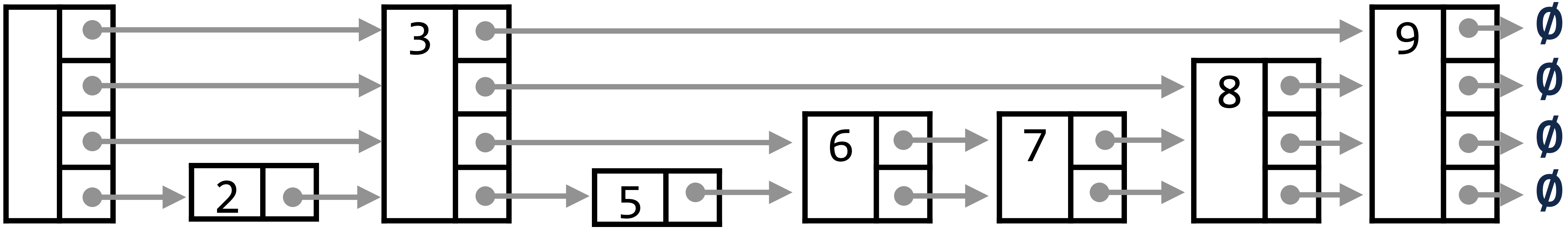


Skip List Remove

`Remove(3)`

Use Find() logic but remove before descending **the previous node**

The remove is a standard Linked List Remove (but at each level)

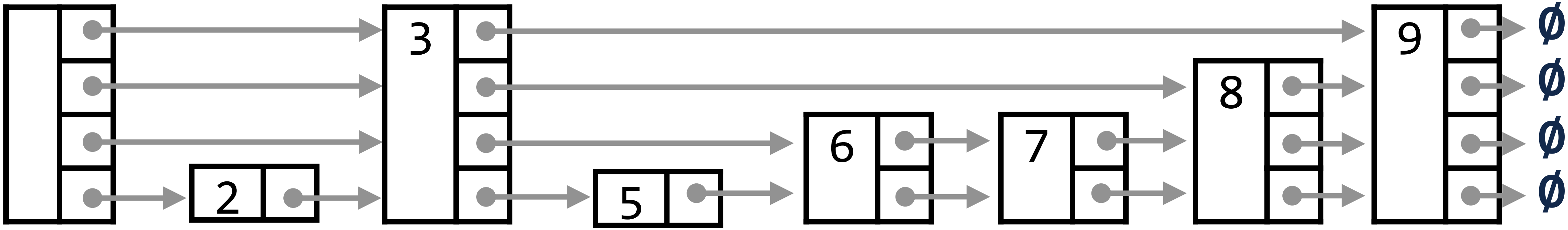


Skip List Remove

Remove(5)

Use Find() logic but remove before descending **the previous node**

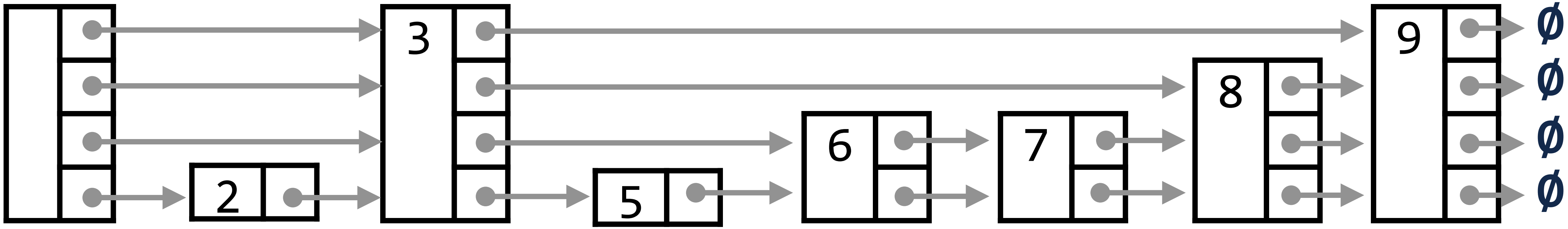
The remove is a standard Linked List Remove (but at each level)



Skip List Efficiency

We've seen the full ADT but haven't explored the runtime

What is the Big O for Find()?

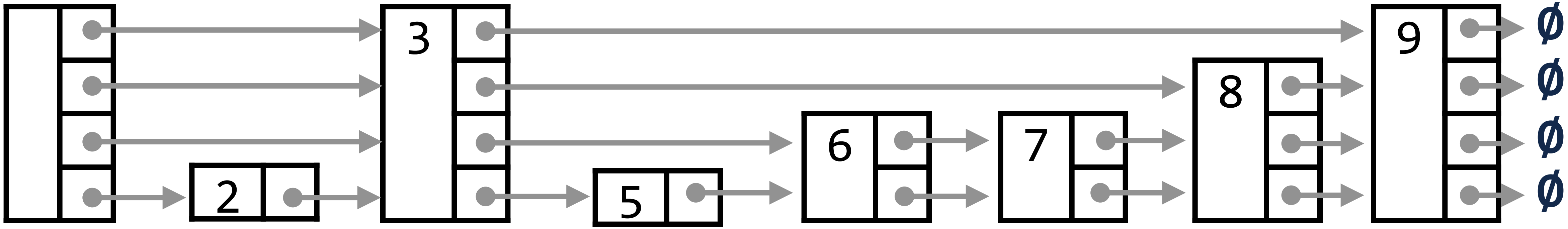


Skip List Efficiency

We've seen the full ADT but haven't explored the runtime

What is the Big O for Find()? $O(n)$ for n nodes (keys)

Using probability, how can we show skip list is better than $O(n)$?



Skip List Efficiency

We've seen the full ADT but haven't explored the runtime

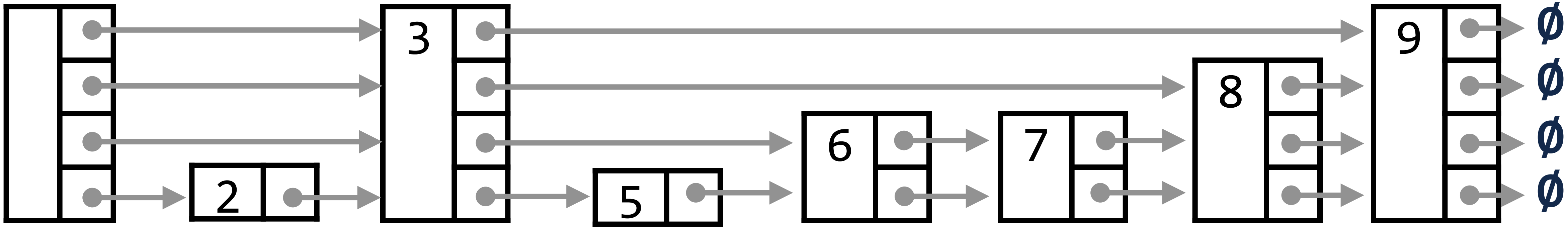
What is the Big O for Find()? $O(n)$ for n nodes (keys)

Using probability, how can we show skip list is better than $O(n)$?

1) Formalize the probability of SkipList reaching height $h > log\ n$

2) Define a recurrence relationship for search path

3) Use (1) and (2) to show that our **average** search time is $log\ n$



Skip List Random Height

We want half the number of items (approximately) at each level.

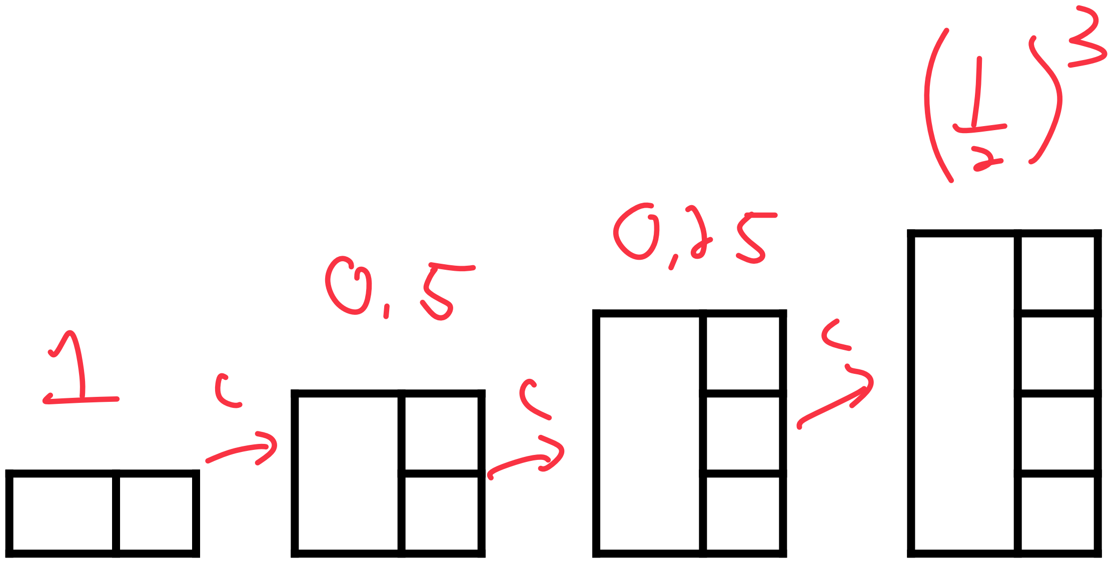
How can we do this?

Skip List Random Height

By definition, each increase in height occurs with probability c.

If $c = 0.5$ (a coin flip), to reach level l, we must flip l heads in a row

By definition the probability a node reaching level l is c^l




```
template <class T>
int SkipList<T>::randHeight(){
  float frac = rand()/(float)RAND_MAX;
  int h = 0;
  while(frac < c){
    h++;
    frac = rand()/(float)RAND_MAX;
  }
  return h;
}
```

Skip List Expectation

unbounded height

if flip inf coins, eventually get string of heads

$\sim \cancel{n}$

$\log n$

We want to bound the height of a SkipList of n nodes but this is deceptively hard to prove **in expectation:**

$n/2^l$

$$E[h] = \sum_{l=0}^{\infty} E[I_l] \qquad I_l = \begin{cases} 1 & \text{if } l\text{th level contains a node} \\ 0 & \text{if } l\text{th level contains no nodes} \end{cases}$$

$1 + .95 + .9 + \ldots$

$$E[h] = \sum_{l=0}^{\lceil log\ n \rceil} E[I_l] + \sum_{l=\lceil log\ n \rceil + 1}^{\infty} E[I_l] \approx \sum_{l=0}^{\lceil log\ n \rceil} 1 + \sum_{l=\lceil log\ n \rceil + 1}^{\infty} \frac{n}{2^l}$$

$n/2^{\log_2 n} = 1$

$\log(n)$

$\frac{n}{2^l} \to 0$ as $l \to \infty$

Skip List Average-Case Performance

Instead we will define an equation for the likelihood of SkipList of n nodes having a height larger than $log\ n$ and claim that the probability is small.

With a probability c of increasing a node's height by 1:

Probability a single node reaches level l:

Skip List Average-Case Performance

Instead we will define an equation for the likelihood of SkipList of n nodes having a height larger than $log\ n$ and claim that the probability is small.

With a probability c of increasing a node's height by 1:

Probability a single node reaches level l: c^l

Probability a single node does not reach level l:

Skip List Average-Case Performance

Instead we will define an equation for the likelihood of SkipList of n nodes having a height larger than $log\ n$ and claim that the probability is small.

With a probability c of increasing a node's height by 1:

Probability a single node reaches level l: c^l

Probability a single node does not reach level l: $1 - c^l$

Probability n nodes do not reach level l:

Skip List Average-Case Performance

Instead we will define an equation for the likelihood of SkipList of n nodes having a height larger than $log\ n$ and claim that the probability is small.

With a probability c of increasing a node's height by 1:

Probability a single node reaches level l: c^l

Probability a single node does not reach level l: $1 - c^l$

Probability n nodes do not reach level l: $(1 - c^l)^n$

Skip List Average-Case Performance

Instead we will define an equation for the likelihood of SkipList of n nodes having a height larger than $log\ n$ and claim that the probability is small.

Probability n nodes do not reach level l: $\left(1 - c^l\right)^n$

Probability at least one node reaches level l:

Skip List Average-Case Performance

Instead we will define an equation for the likelihood of SkipList of n nodes having a height larger than $log\ n$ and claim that the probability is small.

Probability n nodes do not reach level l: $(1 - c^l)^n$

Probability at least one node reaches level l: $1 - (1 - c^l)^n$

Using this equation, the probability of exceeding height h is: nc^h

$\frac{n}{2^h}$

Skip List height is unbounded, but we control probability!

Skip List Average-Case Performance

To quote the original 1990 skipList paper:

> "*Don't worry, be happy*. Simply start a search at the highest level present in the list. As we will see in our analysis, the probability that the maximum level in a list of n elements is significantly larger than L(n) is very small."

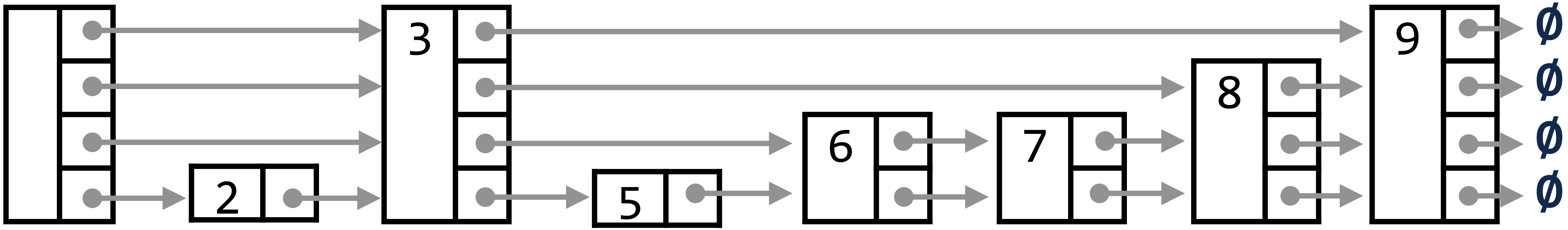
The authors use this logic to state $L(n) = log_{1/c}\ n$ as the optimal (or expected) max height.

William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33, 6 (June 1990), 668–676.

Skip List Expectation

Claim: Expected length of search of skip list is the height $\approx (log\ n)$

Proof: Direct with recurrence equation working backwards



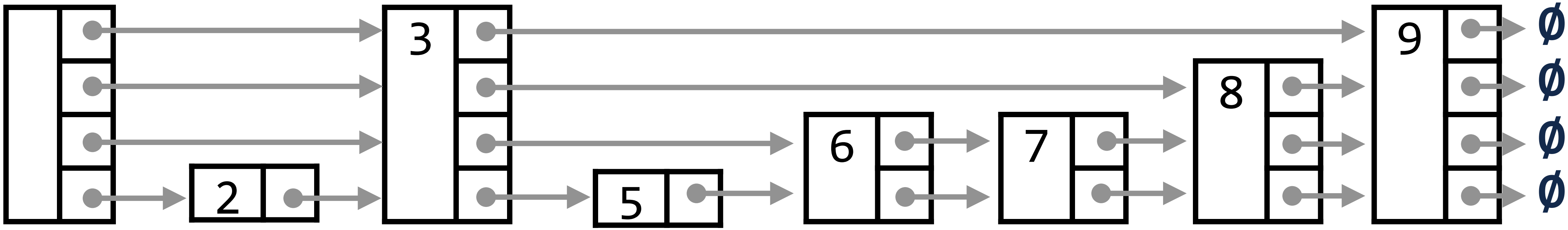
Skip List Expectation

Claim: Expected length of search of skip list is the height $\approx (log\ n)$

Proof: Direct with recurrence equation working backwards

Let $H(k)$ be the expected cost to search a path of k levels

Then $H(k) = 1 + (1 - c) * H(k) + c * H(k - 1)$



Skip List Expectation

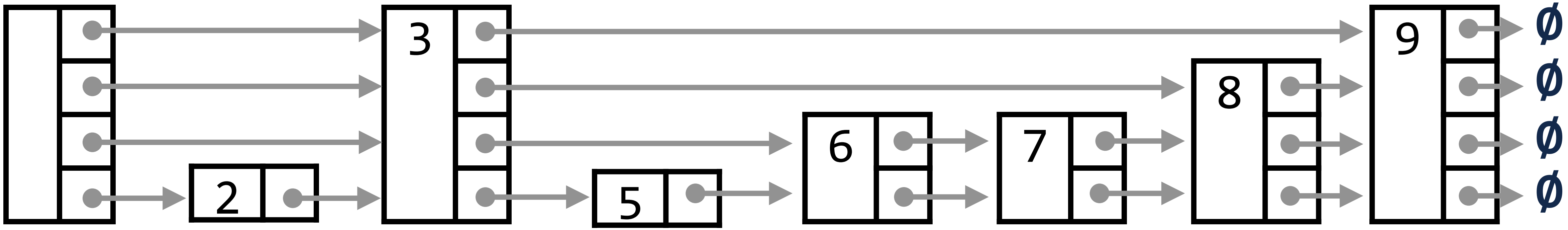
Claim: Expected length of search of skip list is the height $\approx (log\ n)$

Proof: Direct with recurrence equation working backwards

Let $H(k)$ be the expected cost to search a path of k levels

Then $H(k) = 1 + (1 - c) * H(k) + c * H(k - 1)$

Rewrite: $H(k) - (1 - c) * H(k) = 1 + c * H(k - 1)$



Skip List Expectation

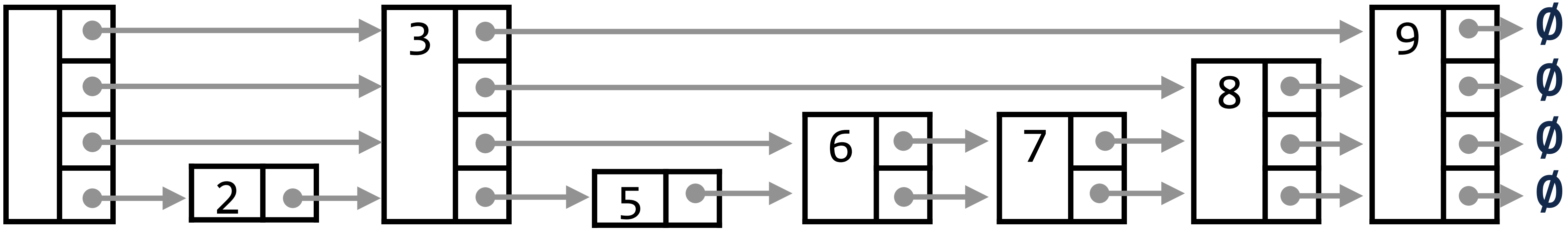
Claim: Expected length of search of skip list is the height $\approx (log\ n)$

Proof: Direct with recurrence equation working backwards

Let $H(k)$ be the expected cost to search a path of k levels

Rewrite: $H(k) - (1 - c) * H(k) = 1 + c * H(k - 1)$

Rewrite:



Skip List Expectation

Claim: Expected length of search of skip list is the height $\approx (log\ n)$

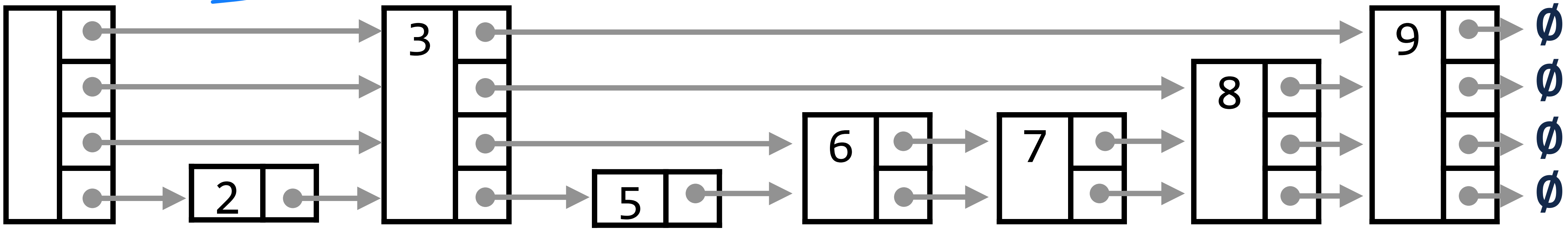
Proof: Direct with recurrence equation working backwards

Let $H(k)$ be the expected cost to search a path of k levels

Rewrite: $H(k) - (1 - c) * H(k) = 1 + c * H(k - 1)$

Rewrite: $c * H(k) = 1 + c * H(k - 1) = H(k) = 1/c + H(k - 1)$

Trivial Soln: k/c



Skip List Efficiency

We've seen the full ADT but haven't explored the runtime

What is the Big O for Find()? $O(n)$ for n nodes (keys)

Skip List mimics behavior of AVL Tree, despite being linked list

1) Our height is (on average) $log\ n$

2) The expected cost to traverse is height bounded!

3) So our average search time is $log\ n$

TABLE II. Timings of Implementations of Different Algorithms

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)

In Conclusion

If interested, read the original publication!

William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33, 6 (June 1990), 668–676.

https://doi.org/10.1145/78973.78977

If not, hopefully you learned a few things about probability in CS!