CS 240 – Data Structures and Data Management

Module 5: Other Dictionary Implementations

T. Biedl E. Schost O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2021

Outline

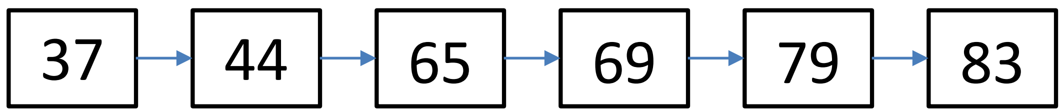
- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Re-ordering items

Outline

- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Re-ordering items

Dictionary ADT: Implementations thus far

- A *dictionary* is a collection of *key-value pairs* (KVPs)
    - *search*, *insert*, and *delete*
- Realizations
    - **Balanced search trees** (AVL trees)
        - $\Theta(\log n)$ search, insert, and delete
        - complex code and not necessarily the fastest running time in practice
    - **Binary search trees**
        - $\Theta(height)$ search, insert and delete
        - simpler than AVL tree
        - randomization helps efficiency
    - **Ordered array**
        - simple implementation
        - $\Theta(\log n)$ search
        - $\Theta(n)$ insert and delete
    - **Ordered linked list**



        - simple implementation
        - $\Theta(n)$ search, insert and delete
        - search is the bottleneck, insert and delete would be $\Theta(1)$ if do search first and account for its running time separately
        - efficient search (like binary search) in ordered linked list?

Outline

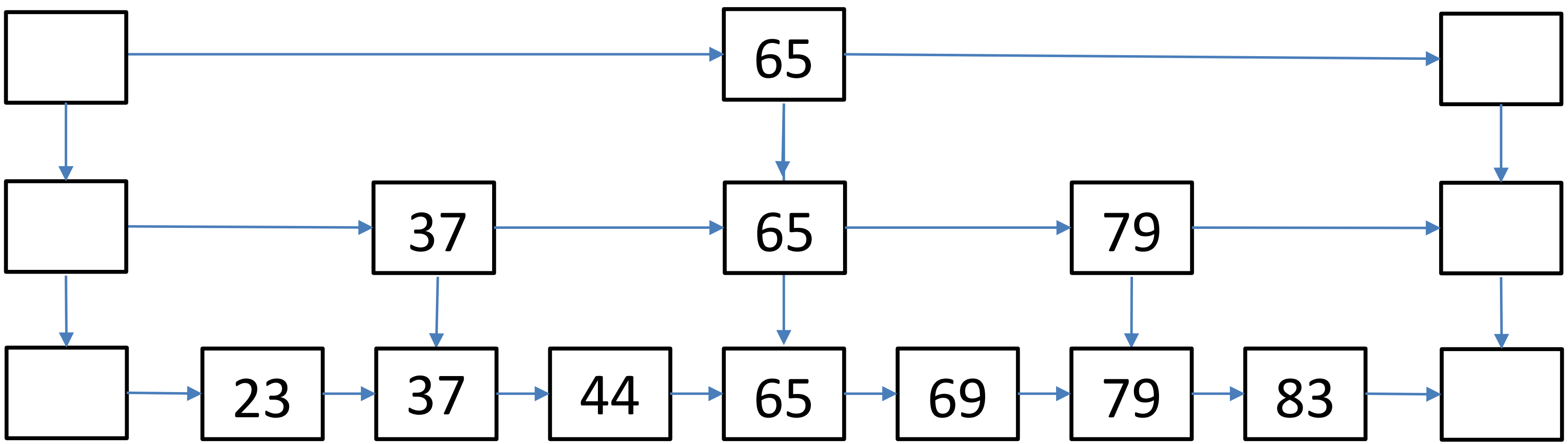
- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Re-ordering items

Skip Lists: Motivation

- Ordered array has efficient binary search

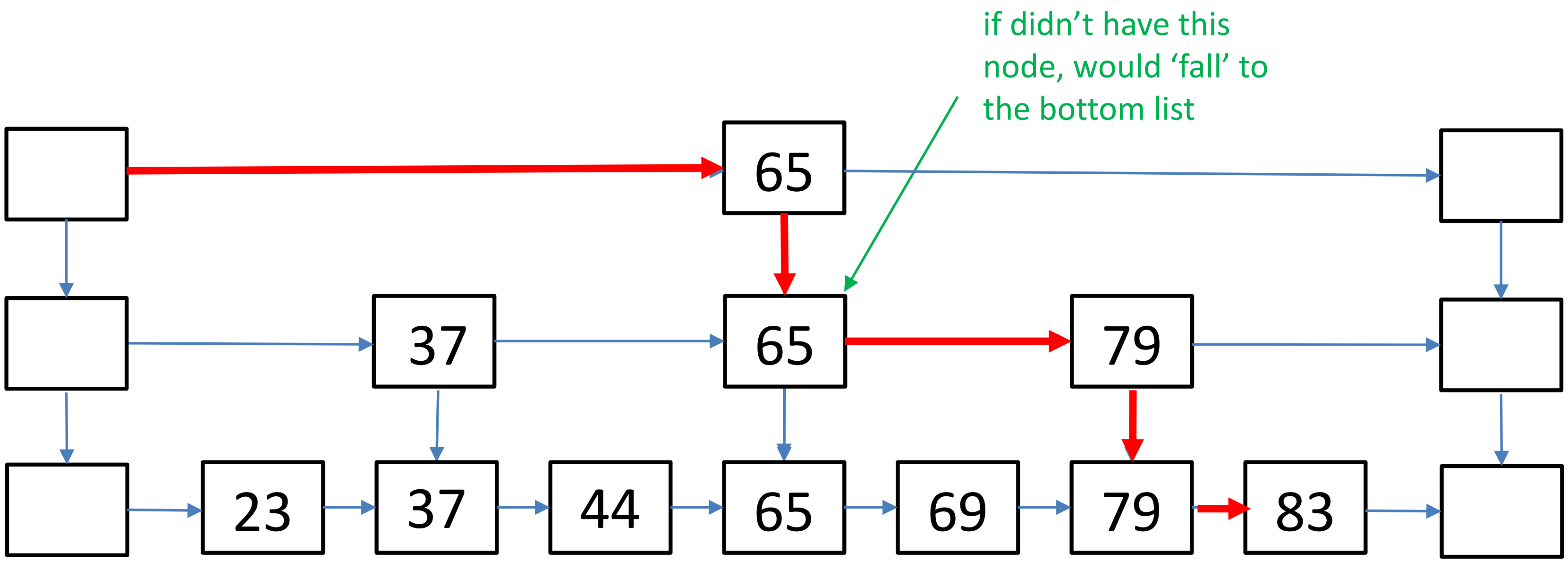
0	1	2	3	4	5	6
23	37	44	65	69	79	83

- Can we imitate binary search in an ordered linked list?



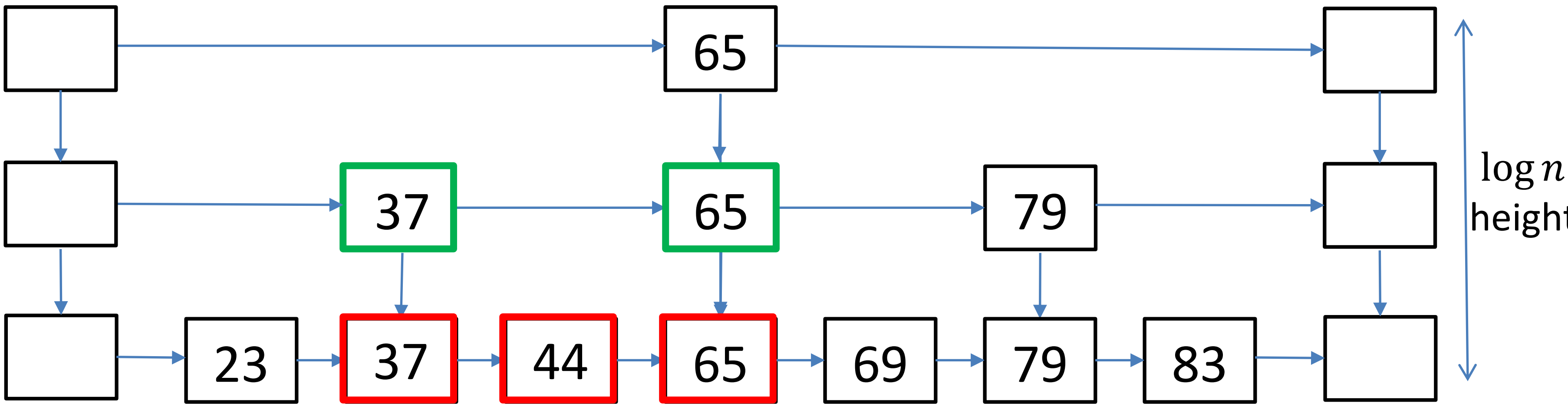
Skip Lists: Motivation

- Search(83)



Skip Lists: Motivation

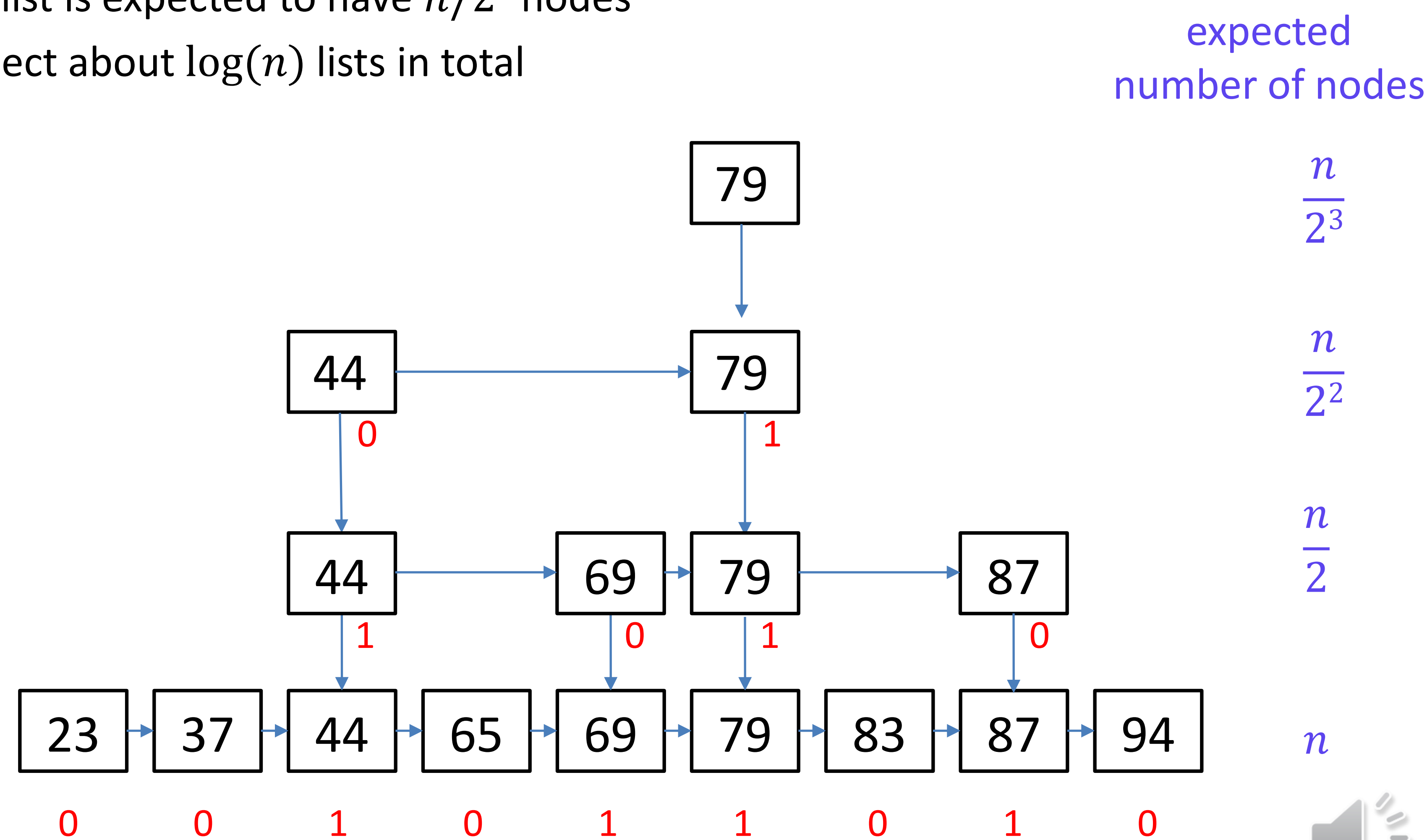
- Imitating binary search with a *hierarchy* of linked lists
  - build from bottom to top, each higher up list has 1/2 of previous list items
  - $\log n$ height (total number of linked lists needed)



- When searching, go through the highest level possible
  - thus visit at most two items at each level
- Easy to implement if data structure is *static*
  - know all items beforehand, no need to insert or delete, but in static case an ordered array will work, and is more efficient (no links)
- To enable insert and delete, use *randomization*

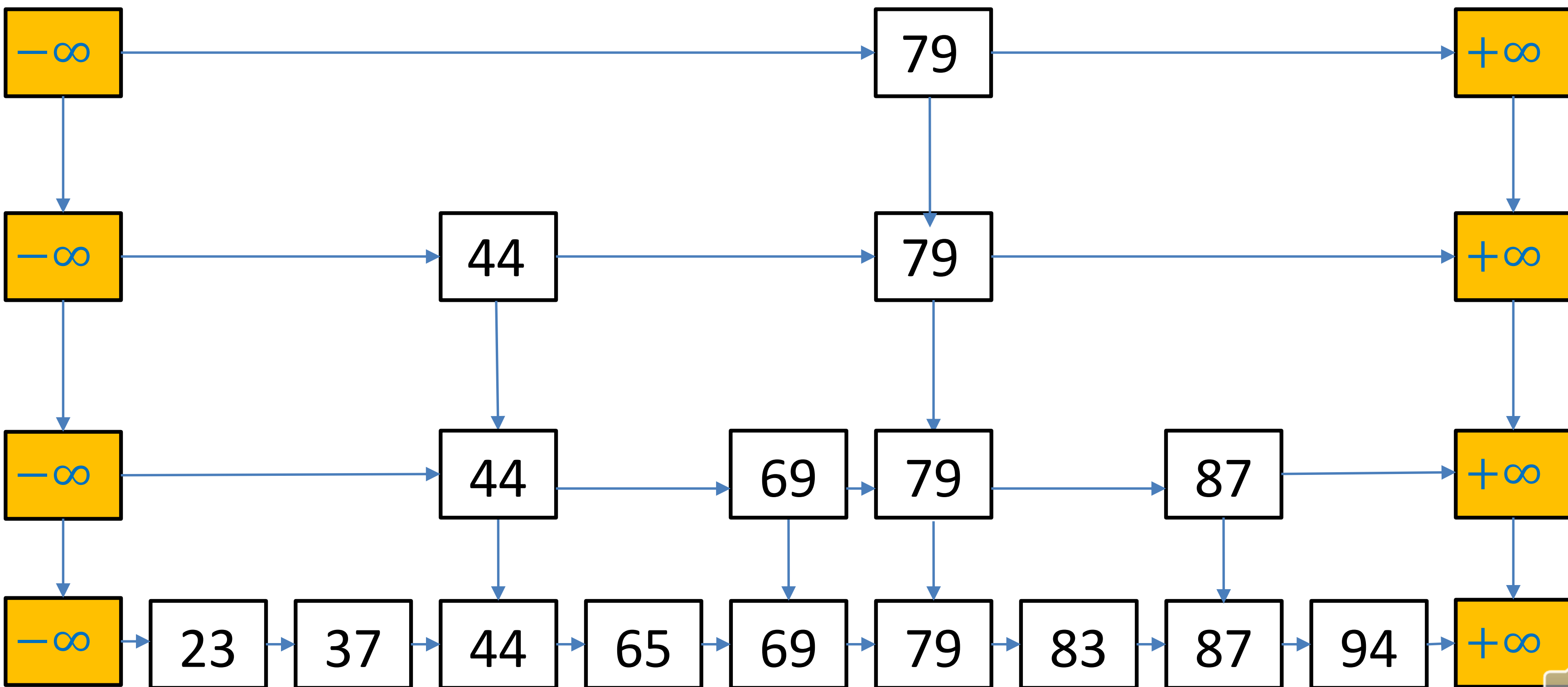
Skip Lists: Motivation

- For next level, choose each item from previous level with probability ½ (coin toss)
- ith list is expected to have $n/2^i$ nodes
- Expect about $\log(n)$ lists in total



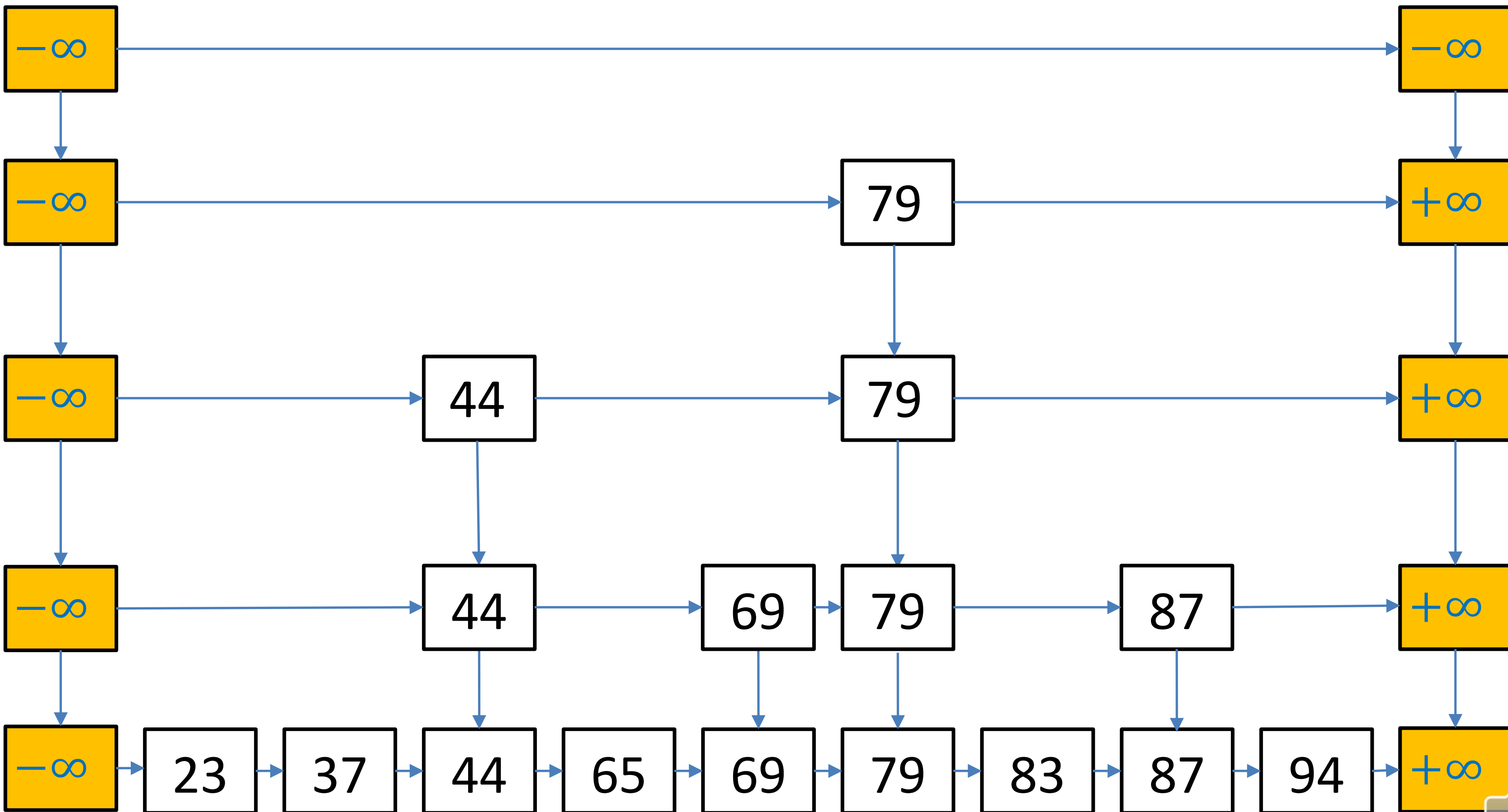
Skip Lists: Motivation

- Insert 'boundary' nodes with special sentinel symbols $-\infty$ and $+\infty$
  - to simplify code for searching



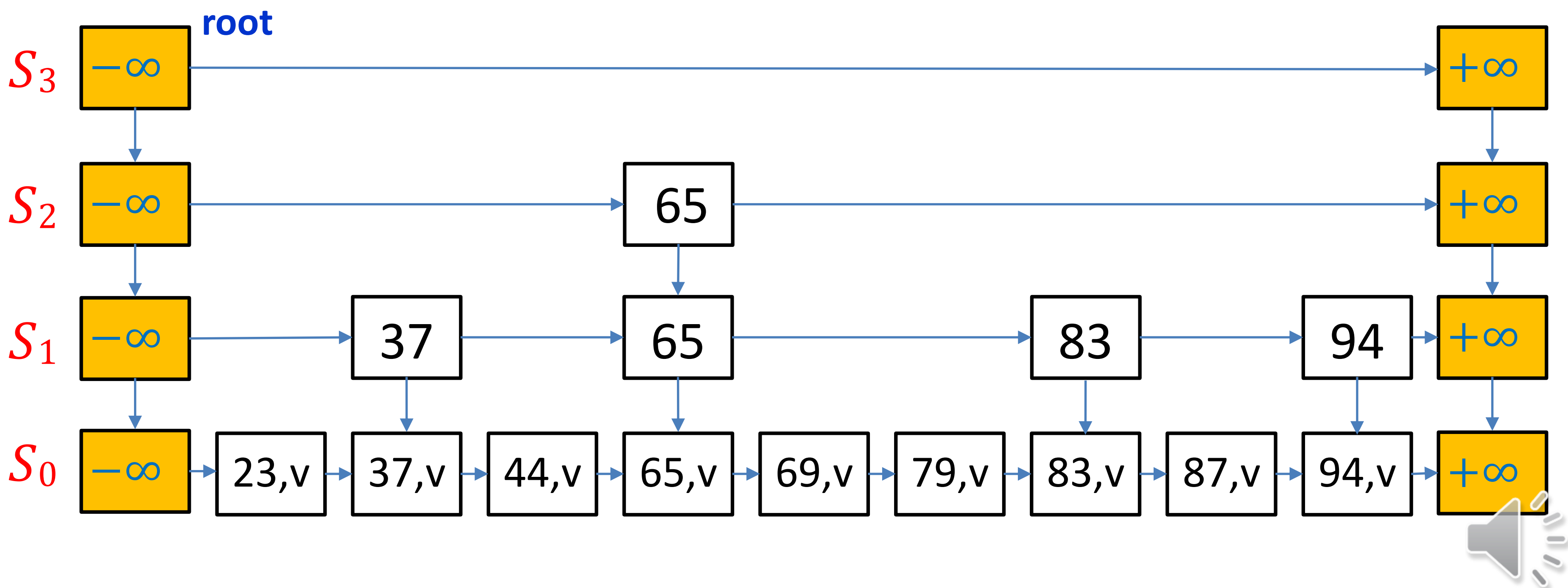
Skip Lists: Motivation

- Insert sentinel only level, with only $-\infty$ and $+\infty$
  - to simplify code for searching



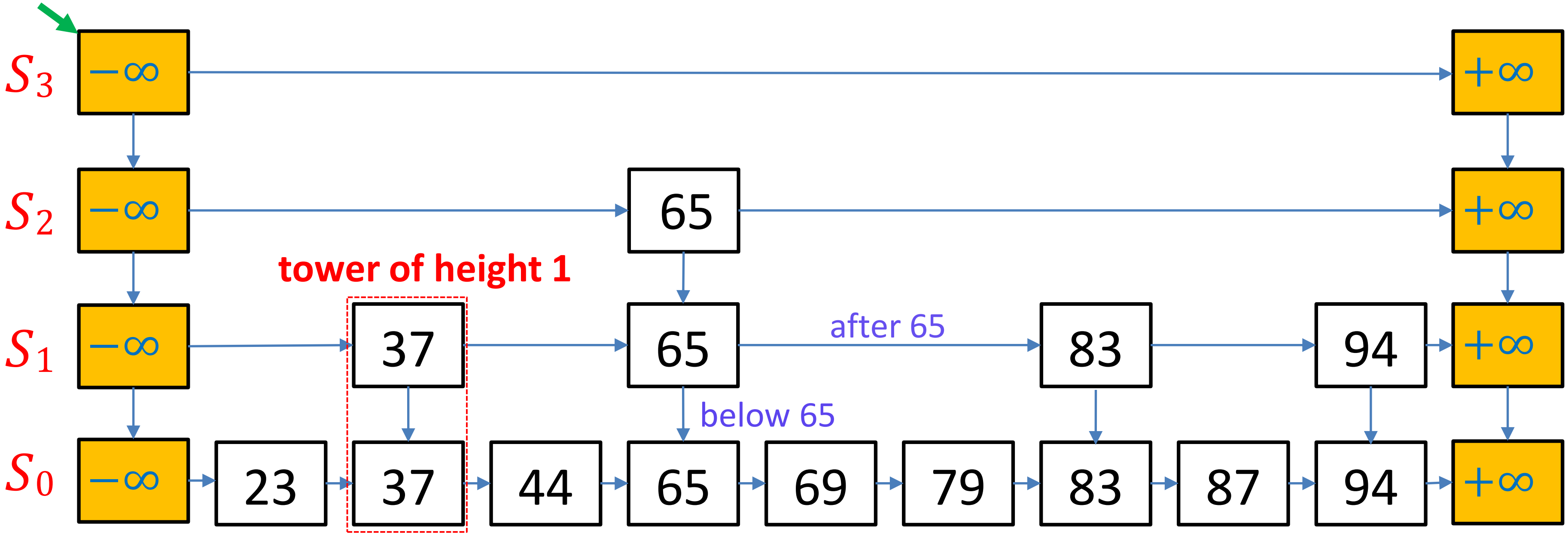
Skip Lists [Pugh'1989]

- A hierarchy S of ordered linked lists (*levels*) $S_0, S_1, \dots, S_h$
  - S_0 contains the KVPs of S in non-decreasing order
  - other lists store only keys, or links to nodes in S_0
  - each S_i contains special keys (sentinels) $-\infty$ and $+\infty$
  - each S_i is randomly generated subsequence of S_{i-1} i.e., $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$
  - S_h contains only sentinels, the left sentinel is the root



Skip Lists [Pugh'1989]

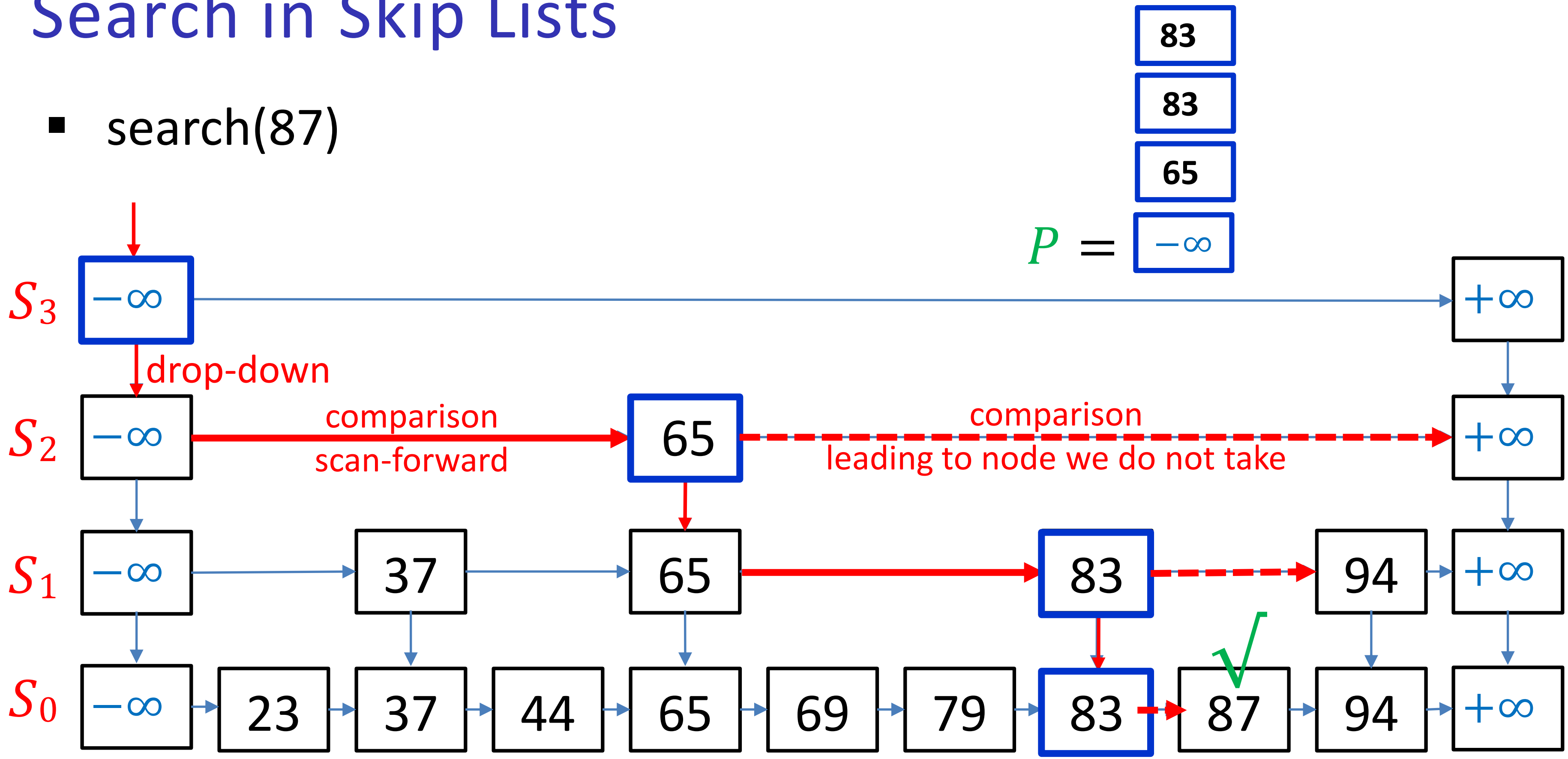
- Will show only keys from now on



- Each KVP belongs to a *tower* of nodes
- Height of the skip list is the maximum height of any tower
- Each node p has references to after(p) and below(p)
- There are (usually) more nodes than keys

Search in Skip Lists

- search(87)



- For each level, **predecessor** of key k is the node before node with key k, or, if key k is not present at that level, the node before where k would be
- P collects predecessors of key k at level S_0, S_1, …
  - these are needed for insert/delete
- k is in skip list if and only if $P.top().after$ has key k

Search in Skip Lists

getPredecessors(k)

$p \leftarrow$ *root*

$P \leftarrow$ stack of nodes, initially containing p

while $p.below \neq NIL$ **do** // keep dropping down until reach S_0

$p \leftarrow p.\text{below}$

while $p.after.key < k$ **do**

$p \leftarrow p.after$ //move to the right

$P.push(p)$ // this is next predecessor

return P

skipList::search(k)

$P \leftarrow$ *getPredecessors*(k)

$top \leftarrow P.top()$ //predecessor of k in S_0

if $top.after.key = k$ **return** $top.after$

else return 'not found, but would be after top'

Insert in Skip Lists

$S_3 \leftarrow$ if in S_2, then insert new item with probability ½

$S_2 \leftarrow$ if in S_1, then insert new item with probability ½

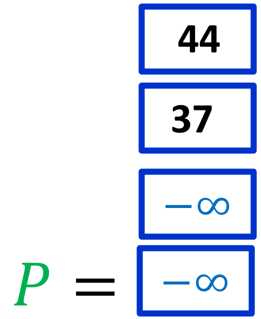
$S_1 \leftarrow$ insert new item with probability ½

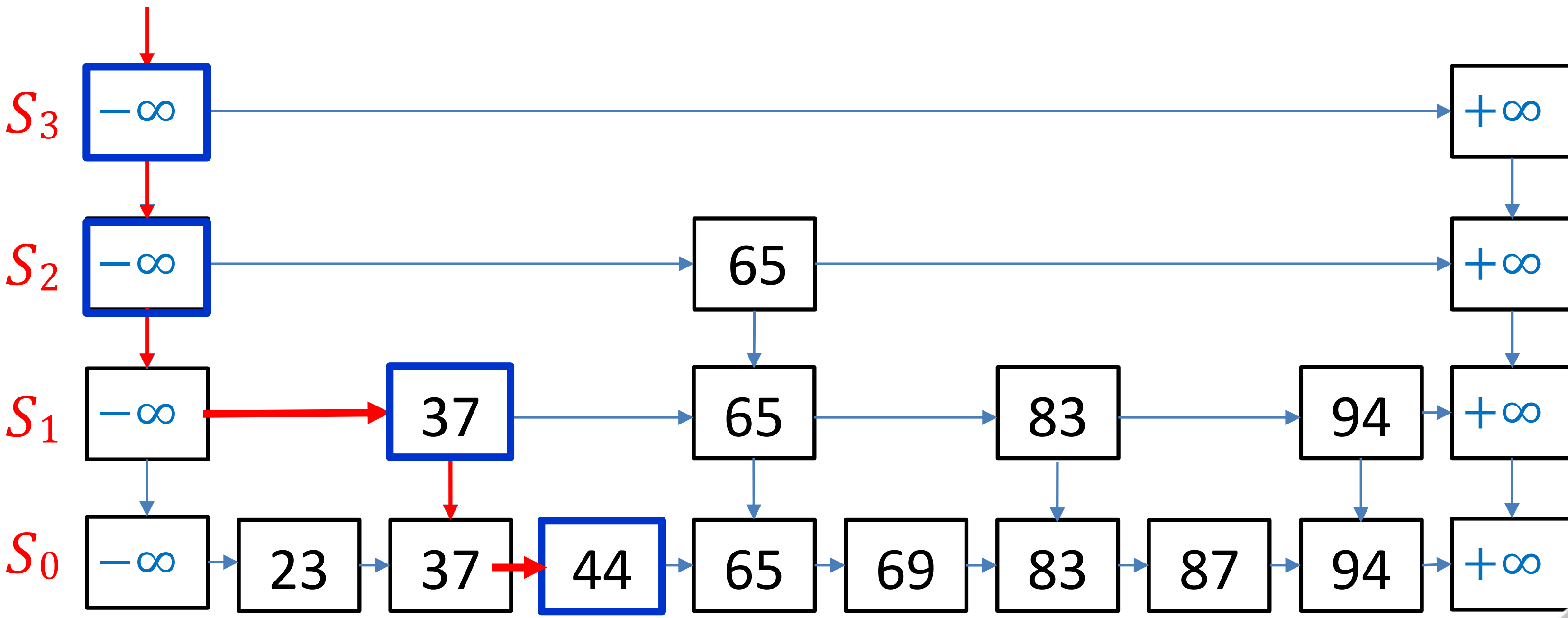
$S_0 \leftarrow$ insert new item

- Keep "tossing a coin" until T appears
- Insert into S_0 and as many other S_i as there are heads
- Examples
  - H, H, T (insert into S_0, S_1, S_2) $\Rightarrow$ will say $i = 2$
  - H, T (insert into S_0, S_1) $\Rightarrow$ will say $i = 1$
  - T (insert into S_0) $\Rightarrow$ will say $i = 0$

Insert in Skip Lists: Example 1

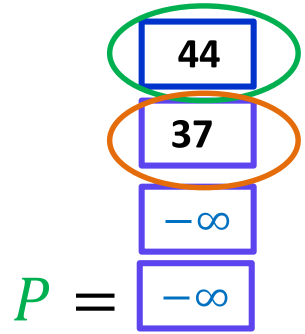
- *skipList::insert*$(52, v)$
- coin tosses: $H, T \Rightarrow i = 1$
- *getPredecessors*(52)

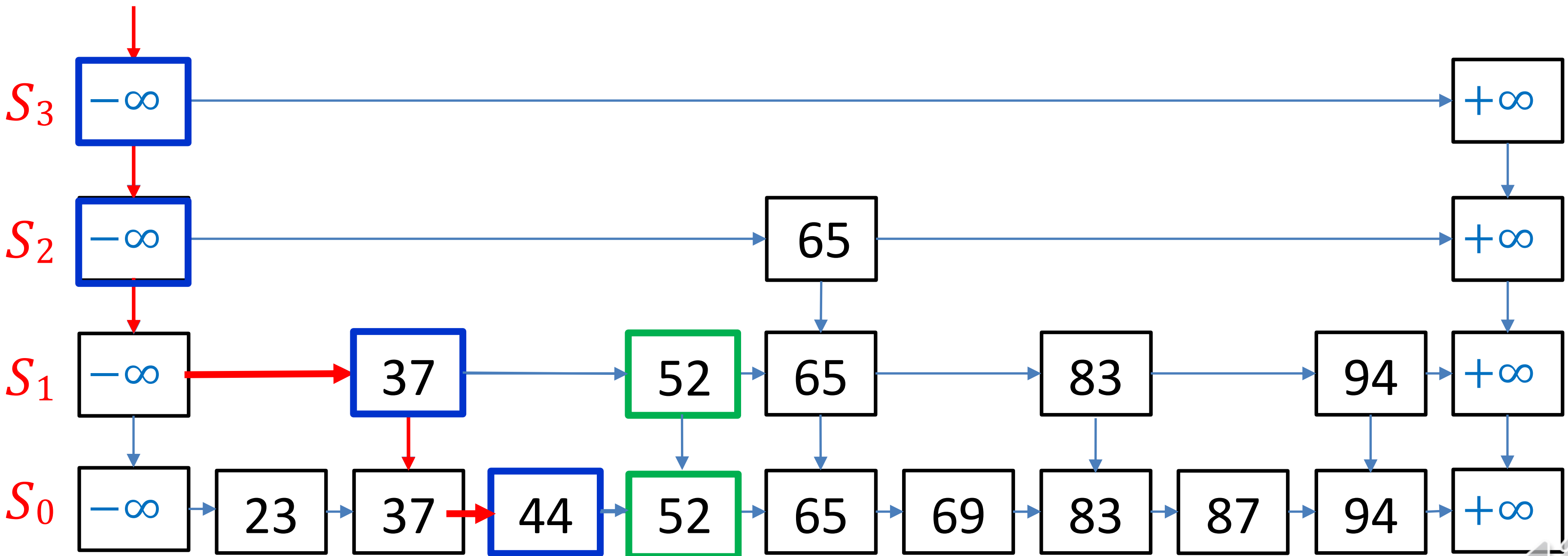




Insert in Skip Lists Example 1

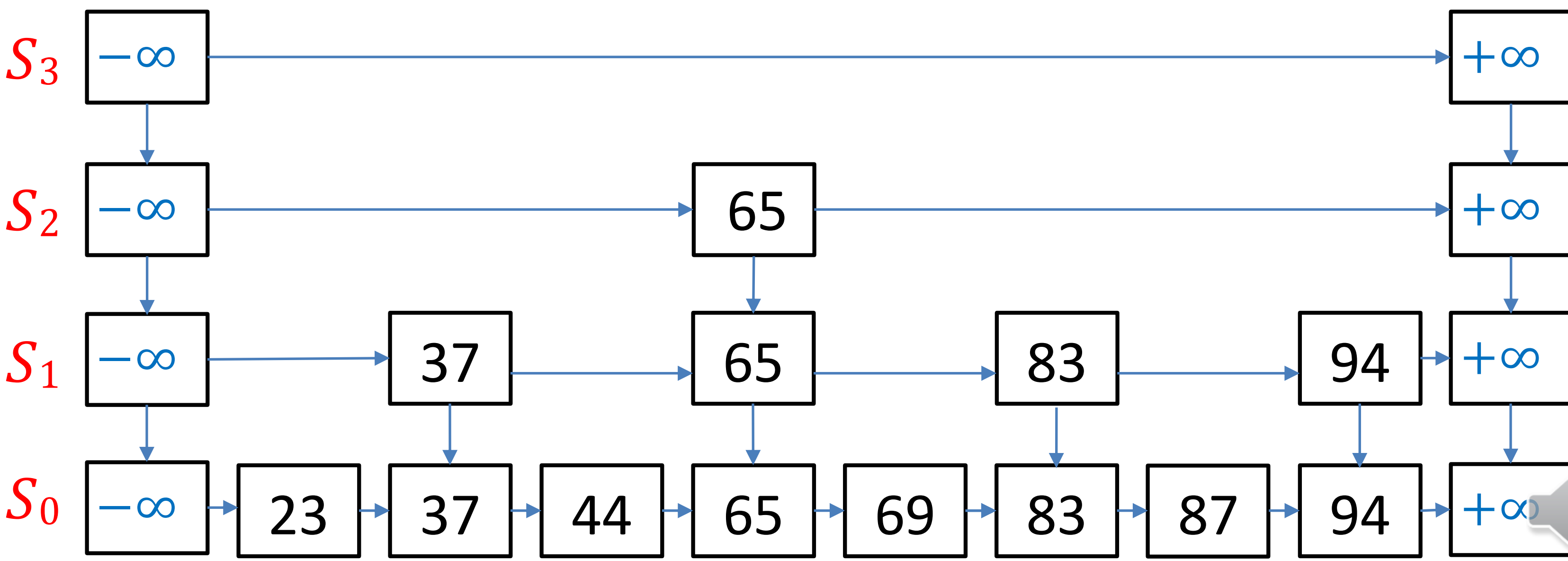
- *skipList::insert*$(52, v)$
- coin tosses: $H, T \Rightarrow i = 1$
- *getPredecessors*(52)
- now insert into S_0 and S_1





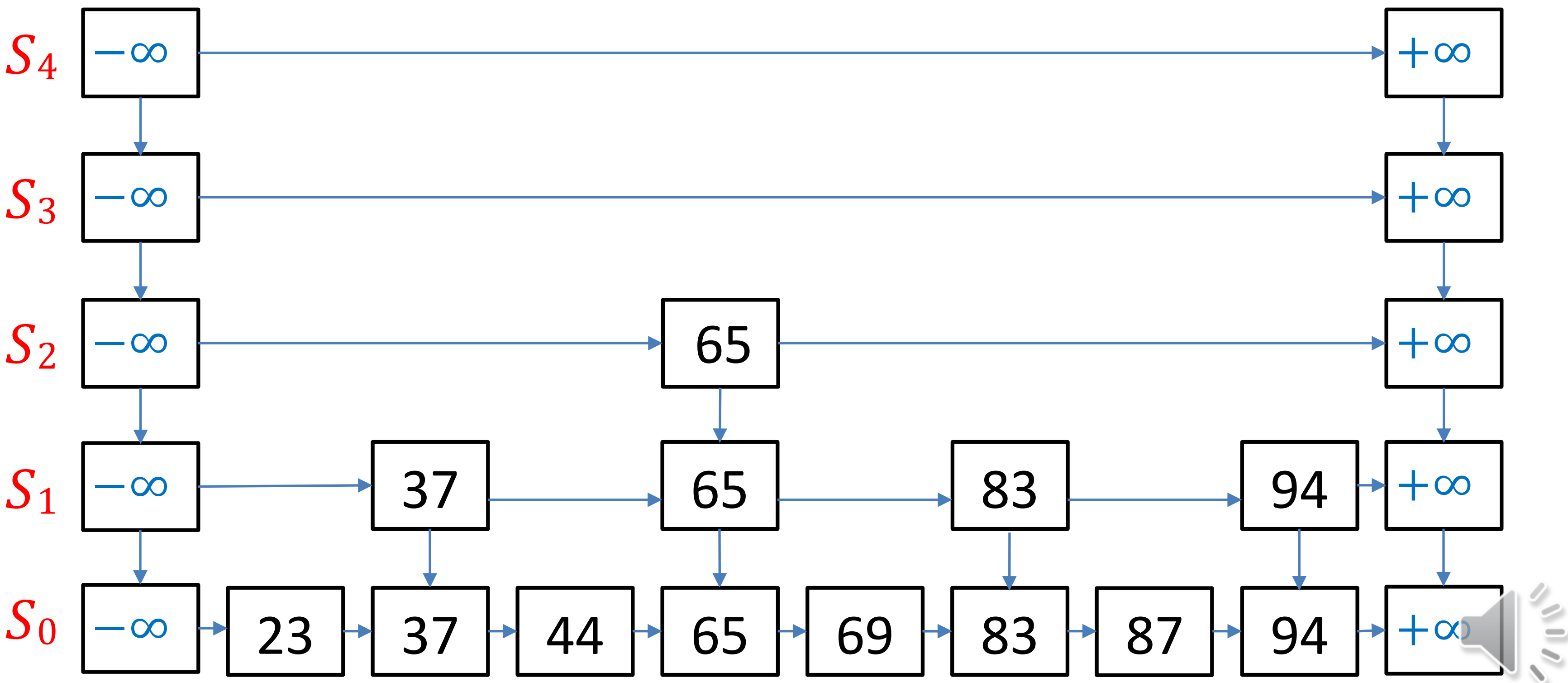
Insert in Skip Lists: Example 2

- *skipList::insert*$(100, v)$
- coin tosses: $H, H, H, T \Rightarrow i = 3$
- first increase height



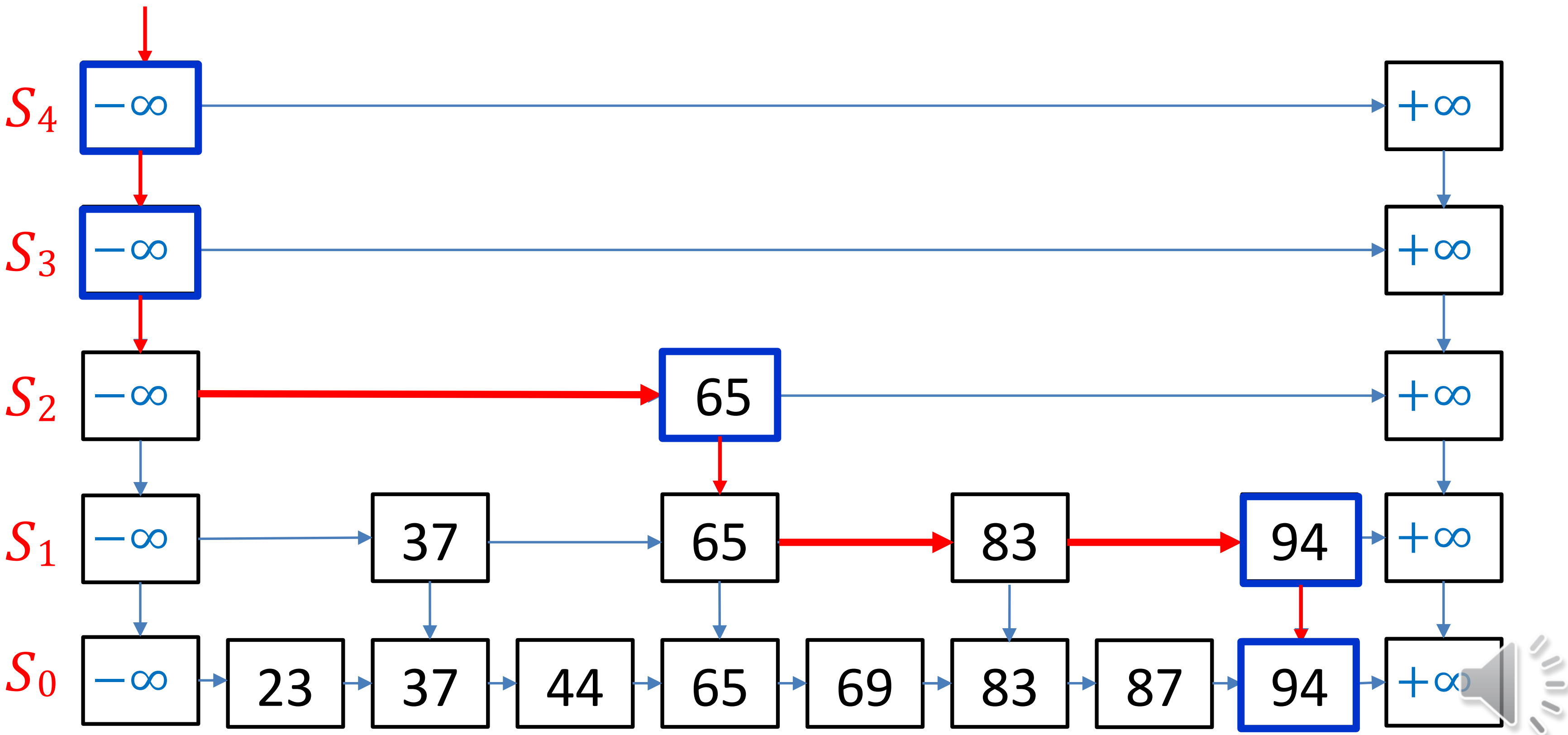
Insert in Skip Lists: Example 2

- *skipList::insert*$(100, v)$
- coin tosses: $H, H, H, T \Rightarrow i = 3$
- first increase height
- next *getPredecessors* (100)



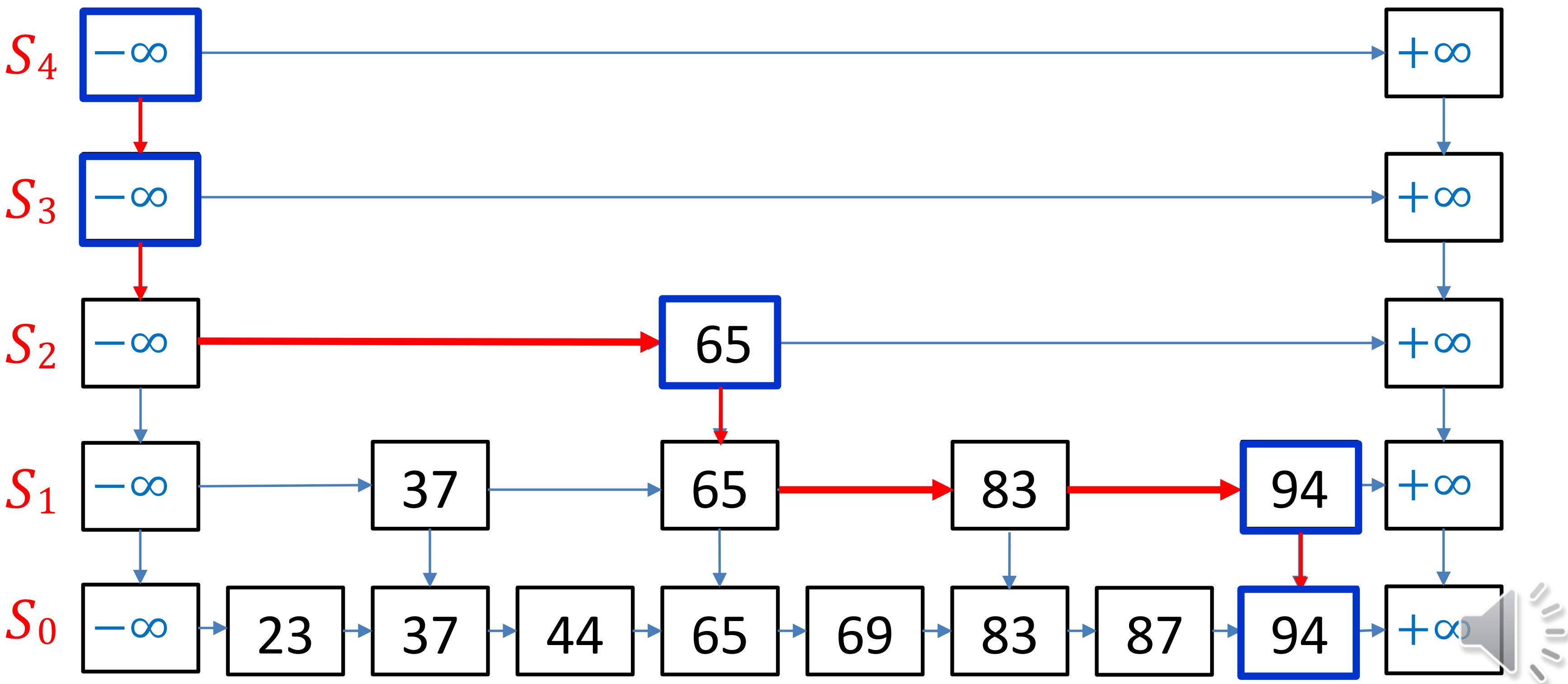
Insert in Skip Lists: Example 2

- *skipList::insert*$(100, v)$
- coin tosses: $H, H, H, T \Rightarrow i = 3$
- first increase height
- next *getPredecessors* (100)



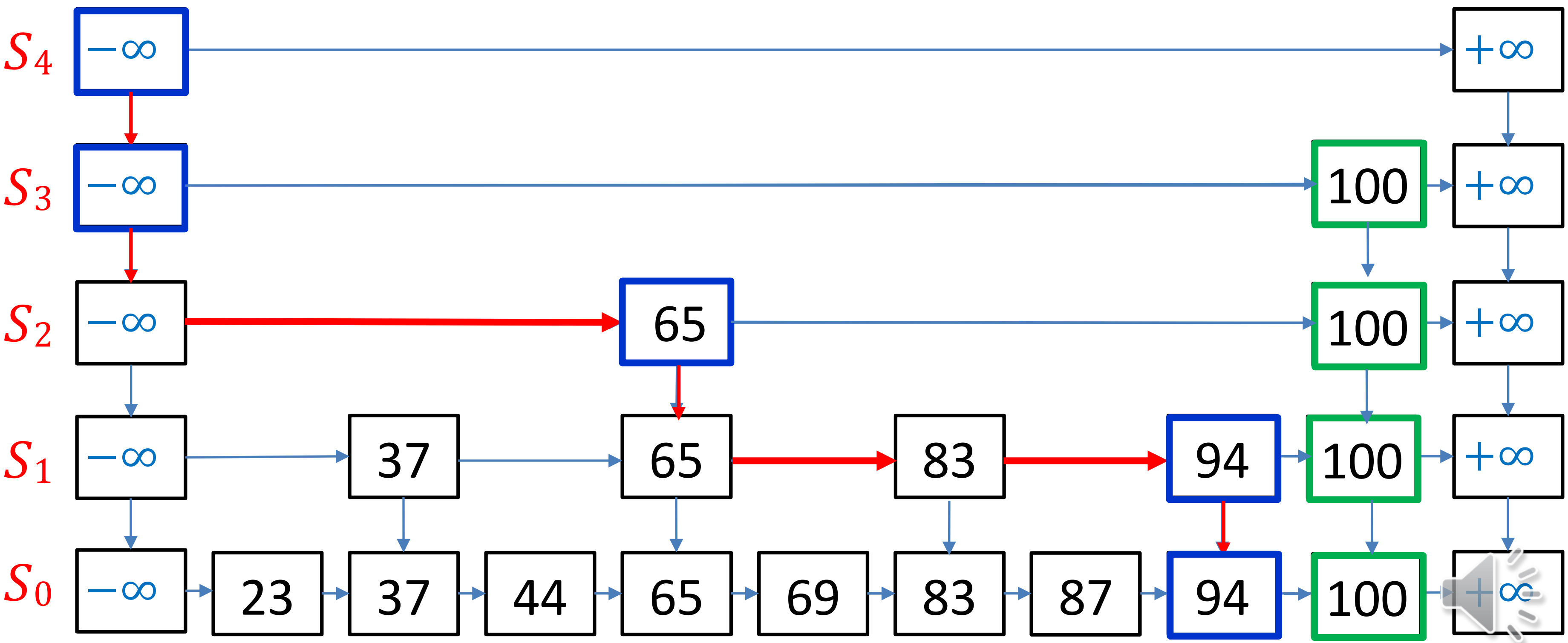
Insert in Skip Lists: Example 2

- *skipList::insert*$(100, v)$
- coin tosses: $H, H, H, T \Rightarrow i = 3$
- first increase height
- next *getPredecessors* (100)
- insert new key



Insert in Skip Lists: Example 2

- *skipList::insert*$(100, v\ )$
- coin tosses: $H, H, H, T \ \Rightarrow \ i \ = 3$
- first increase height
- next *getPredecessors* (100)
- insert new key

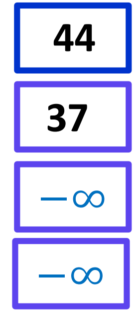


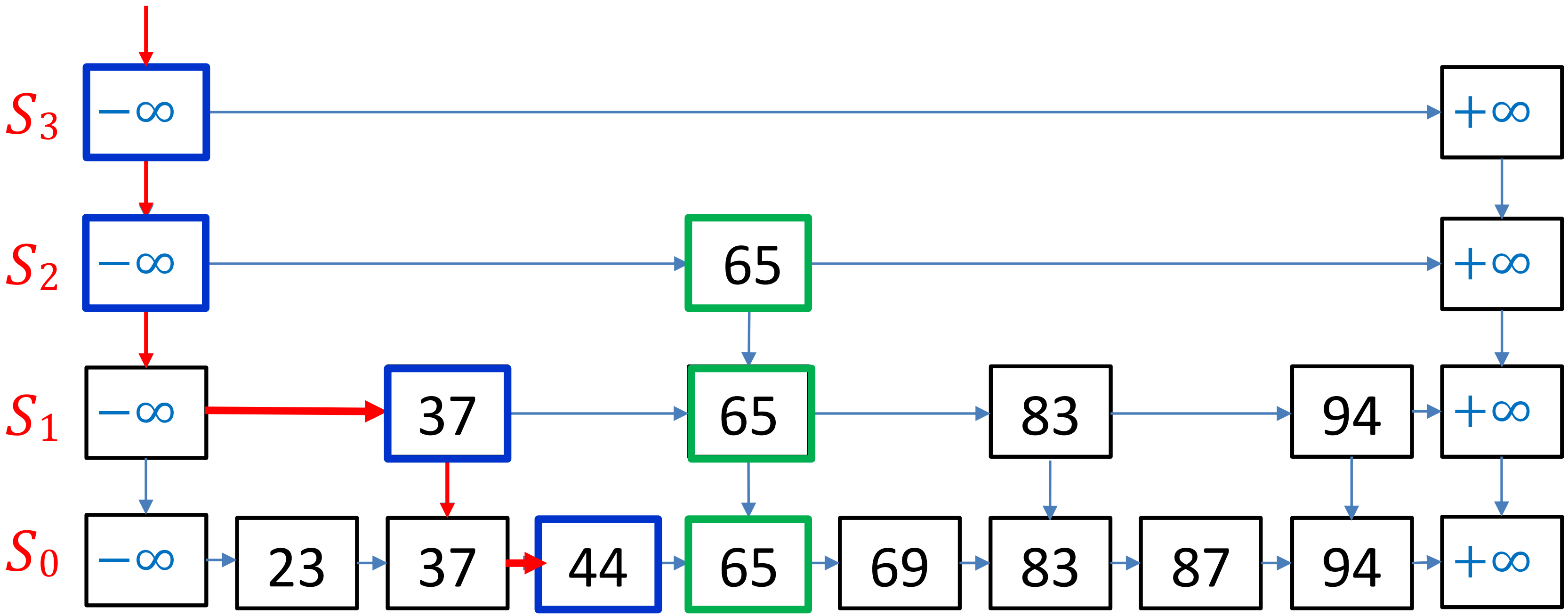
Insert in Skip Lists

```
skipList::insert(k, v)
        P ← getPredecessors(k)
        for (i ← 0; random(2) = 1; i ← i + 1) {}   // random tower height
        while i ≥ P.size()                          // increase skip-list height?
            root ← new sentinel-only list linked in appropriately
            P.append(left sentinel of root)
        p ← P.pop()                                 // insert (k, v) in S_0
        zBellow ← new node with (k, v) inserted after p
        while i > 0                                 // insert k in S_1 S_2,..., S_i
            p ← P.pop()
            z ← new node with k added after p
            z.below ← zBellow
            zBellow ← z
            i ← i − 1
```

Example: Delete in Skip Lists

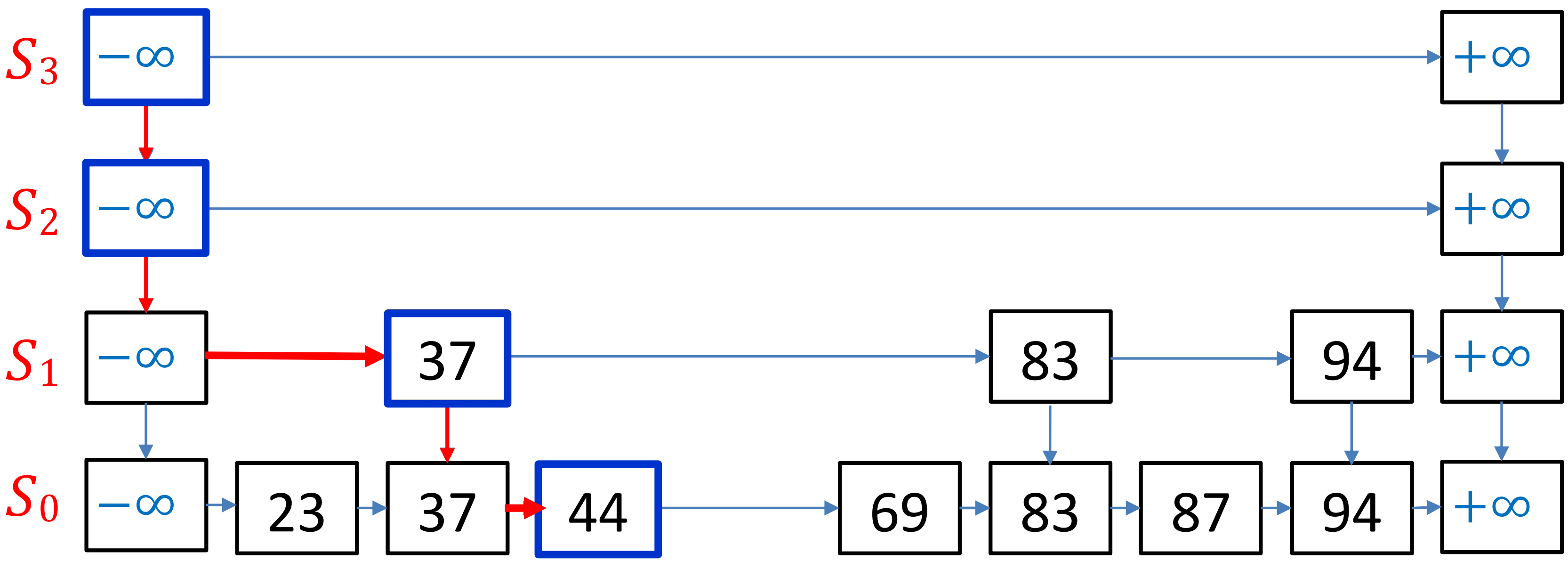
- *skipList::delete*(65)
  - first *getPredecessors*$(S, 65)$
  - then delete key 65 from all S_i
    - P has predecessor of each node to be deleted

$P =$ 



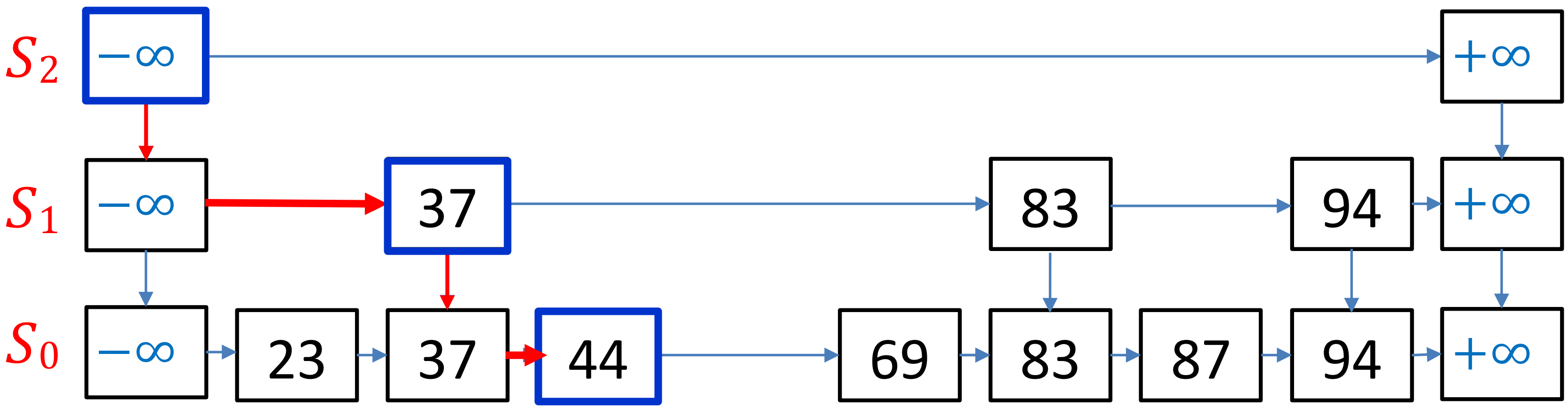
Example: Delete in Skip Lists

- *skipList::delete*(65)
  - first *getPredecessors*$(S, 65)$
  - then delete key 65 from all S_i
    - P has predecessor of each node to be deleted
  - height decrease: delete all unnecessary S_i, if any



Example: Delete in Skip Lists

- *skipList::delete*(65)
  - first *getPredecessors*$(S, 65)$
  - then delete key 65 from all S_i
    - P has predecessor of each node to be deleted
  - height decrease: delete all unnecessary S_i, if any



Delete in Skip Lists

skipList::delete(k)

$P \leftarrow$ *getPredecessors*(k)

while P is non-empty

$p \leftarrow P.pop()$ // predecessor of k in some layer

if $p.after.key = k$

$p.after \leftarrow p.after.after$

else break // no more copies of k

$p \leftarrow$ left sentinel of the root-list

while $p.below.after$ is the ∞ sentinel

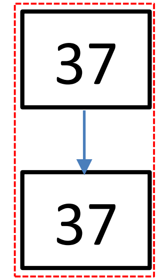
// the two top lists are both only sentinels, remove one

$p.below \leftarrow p.below.below$ // removes the second empty list

$p.after.below \leftarrow p.after.below.below$

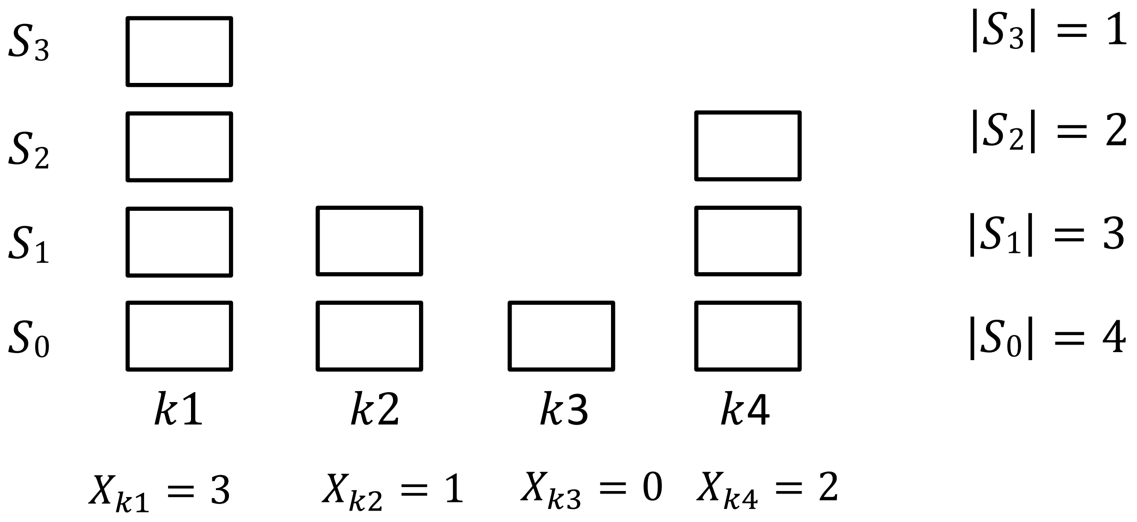
Skip List Analysis

tower of height 1



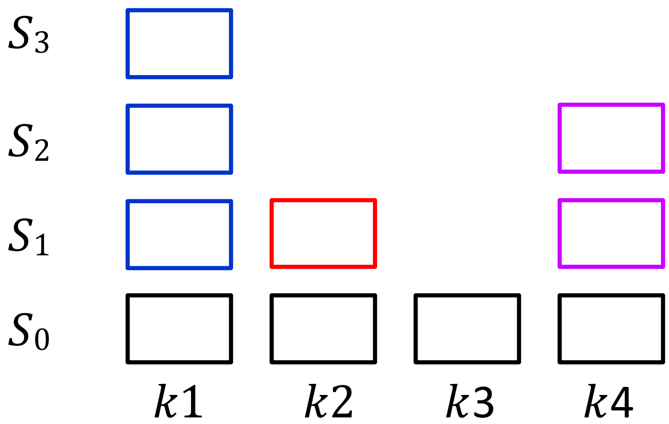
- Let X_k be the height of tower for key k
  - $P(X_k \geq 1) = \frac{1}{2}$, $P(X_k \geq 2) = \frac{1}{2} \cdot \frac{1}{2}$, $P(X_k \geq 3) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2}$
- In general $P(X_k \geq i) = P(\underbrace{H\ H\ \dots\ H}_{i \text{ times}}) = \left(\frac{1}{2}\right)^i$
- In the worst case, the height of a tower could be arbitrary large
  - no bound on height in terms of n
- Therefore operations could be arbitrarily slow, and space requirements arbitrarily large
- But this is exceedingly unlikely
- Therefore we analyse *expected* run-time and space-usage

Skip List Analysis



- Let X_k be the height of tower for key k, we know $P(X_k \geq i) = \frac{1}{2^i}$
- If $X_k \geq i$ then list S_i includes key k
- Let $|S_i|$ be the number of keys in list S_i
  - sentinels do not count towards the length
  - S_0 always contains all n keys

Skip List Analysis



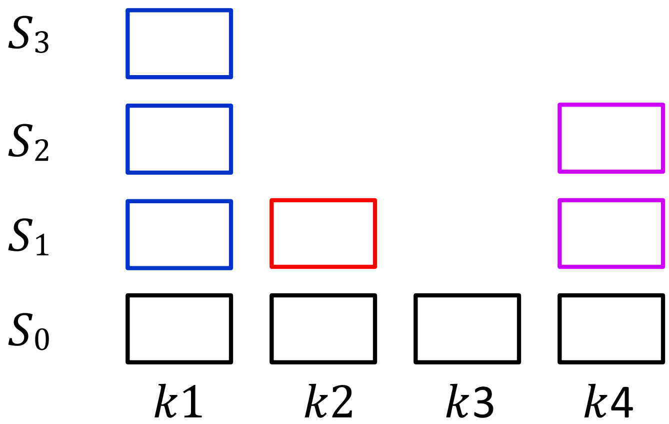
$I_{3,\,k1} = 1$ $\quad I_{3,\,k2} = 0$ $\quad I_{3,\,k3} = 0$ $\quad I_{3,\,k4} = 0$

$I_{2,\,k1} = 1$ $\quad I_{2,\,k2} = 0$ $\quad I_{2,\,k3} = 0$ $\quad I_{2,\,k4} = 1$

$I_{1,\,k1} = 1$ $\quad I_{1,\,k2} = 1$ $\quad I_{1,\,k3} = 0$ $\quad I_{1,\,k4} = 1$

- Let X_k be the height of tower for key k, we know $P(X_k \geq i) = \frac{1}{2^i}$
- If $X_k \geq i$ then list S_i includes key k
- Let $|S_i|$ be the number of keys in list S_i
  - sentinels do not count towards the length
- Let $I_{i,\,k} = \begin{cases} 0 & \text{if } X_k < i \\ 1 & \text{if } X_k \geq i \end{cases}$
- $|S_i| = \sum_{key\ k} I_{i,\,k}$

Skip List Analysis



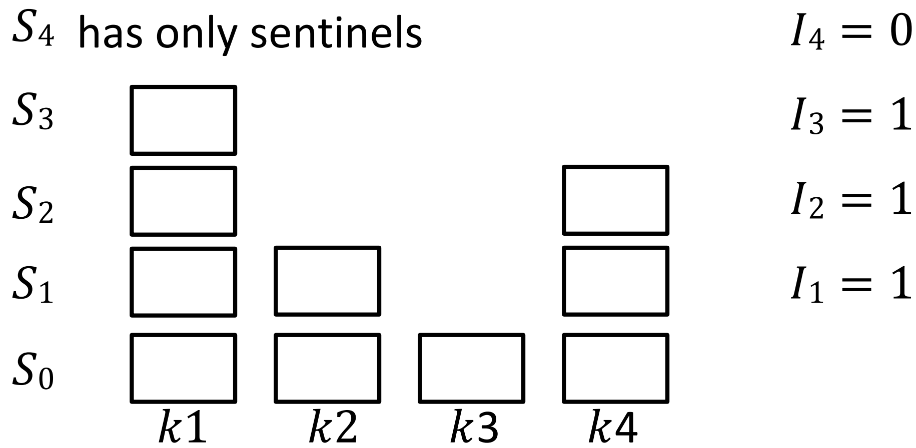
$I_{3,\,k1} = 1$ $\quad I_{3,\,k2} = 0$ $\quad I_{3,\,k3} = 0$ $\quad I_{3,\,k4} = 0$

$I_{2,\,k1} = 1$ $\quad I_{2,\,k2} = 0$ $\quad I_{2,\,k3} = 0$ $\quad I_{2,\,k4} = 1$

$I_{1,\,k1} = 1$ $\quad I_{1,\,k2} = 1$ $\quad I_{1,\,k3} = 0$ $\quad I_{1,\,k4} = 1$

- Let X_k be the height of tower for key k, we know $P(X_k \geq i) = \frac{1}{2^i}$
- Let $|S_i|$ be the number of keys in list S_i
- Let $I_{i,\,k} = \begin{cases} 0 & \text{if } \; X_k < i \\ 1 & \text{if } \; X_k \geq i \end{cases}$
- $|S_i| = \sum_{key\ k} I_{i,\,k}$
- $E[|S_i|] = E\left[\sum_{key\ k} I_{i,\,k}\right] = \sum_{key\ k} E[I_{i,\,k}] = \sum_{key\ k} P(I_{i,\,k} = 1) = \sum_{key\ k} P\,(X_k \geq i) = \frac{n}{2^i}$
- The expected length of list S_i is $\frac{n}{2^i}$

Skip List Analysis



- $|S_i|$ is number of keys in list S_i
  - $E[|S_i|] = \frac{n}{2^i}$
- Let $I_i = \begin{cases} 0 & \text{if } |S_i| = 0 \\ 1 & \text{if } |S_i| \geq 1 \end{cases}$
- $h = 1 + \sum_{i \geq 1} I_i$ (here +1 is for the sentinel-only level)
- Since $I_i \leq 1$ we have that $E[I_i] \leq 1$
- Since $I_i \leq |S_i|$ we have that $E[I_i] \leq E[|S_i|] = \frac{n}{2^i}$
- For ease of derivation, assume n is a power of 2
- $$E[h] = E\left[1 + \sum_{i \geq 1} I_i\right] = 1 + \sum_{i \geq 1} E[I_i] = 1 + \sum_{i=1}^{\log n} E[I_i] + \sum_{i=1+\log n}^{\infty} E[I_i]$$
  $$\leq 1 + \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i}$$
  $$\leq 1 + \log n + \sum_{i=0}^{\infty} \frac{n}{2^{i+1+\log n}}$$

Skip List Analysis

S_4 has only sentinels $\qquad I_4 = 0$

$S_3 \qquad I_3 = 1$

- $|S_i|$ is number of keys in list S_i
  - $E[|S_i|] = \frac{n}{2^i}$
- Let $I_i = \begin{cases} 0 & \text{if } |S_i| = 0 \\ 1 & \text{if } |S_i| \geq \end{cases}$
- $h = 1 + \sum_{i \geq 1} I_i$ (here +
- Since $I_i \leq 1$ we have tha
- Since $I_i \leq |S_i|$ we have t
- For ease of derivation, a
- $E[h] = E\left[1 + \sum_{i \geq 1} I_i\right] =$

$$\sum_{i=0}^{\infty} \frac{n}{2^{i+1+\log n}} = \frac{1}{2}\sum_{i=0}^{\infty} \frac{n}{2^i 2^{\log n}}$$

$$= \frac{1}{2}\sum_{i=0}^{\infty} \frac{n}{2^i n}$$

$$= \frac{1}{2}\sum_{i=0}^{\infty} \frac{1}{2^i} = 1$$

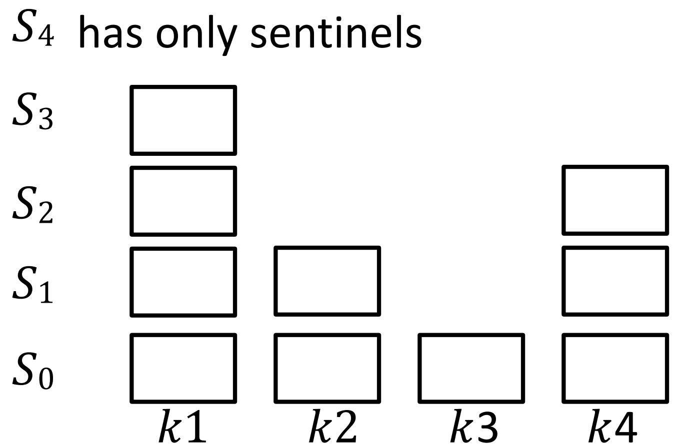
$$S = \sum_{i=0}^{\infty} \frac{1}{2^i} \qquad 2S = \sum_{i=0}^{\infty} \frac{1}{2^{i-1}} = 2 + \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$S = 2S - S = 2$$

$$\ldots{+}\log n \; 2^i$$

$$+ \sum_{i=0}^{\infty} \frac{n}{2^{i+1+\log n}}$$

Skip List Analysis



$I_4 = 0$

$I_3 = 1$

$I_2 = 1$

$I_1 = 1$

- $|S_i|$ is number of keys in list S_i
  - $E[|S_i|] = \frac{n}{2^i}$
- Let $I_i = \begin{cases} 0 & \text{if } |S_i| = 0 \\ 1 & \text{if } |S_i| \geq 1 \end{cases}$
- $h = 1 + \sum_{i\geq 1} I_i$ (here +1 is for the sentinel-only level)
- Since $I_i \leq 1$ we have that $E[I_i] \leq 1$
- Since $I_i \leq |S_i|$ we have that $E[I_i] \leq E[|S_i|] = \frac{n}{2^i}$
- For ease of derivation, assume n is a power of 2
- $E[h] = E\left[1 + \sum_{i\geq 1} I_i\right] = 1 + \sum_{i\geq 1} E[I_i] = 1 + \sum_{i=1}^{\log n} E[I_i] + \sum_{i=1+\log n}^{\infty} E[I_i]$

$$\leq 1 + \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i}$$

$$\leq 1 + \log n + 1$$

- Expected height of skip list is at most $2 + \log n$

Skip List Analysis: Expected Space

- We need space for nodes storing sentinels and nodes storing keys

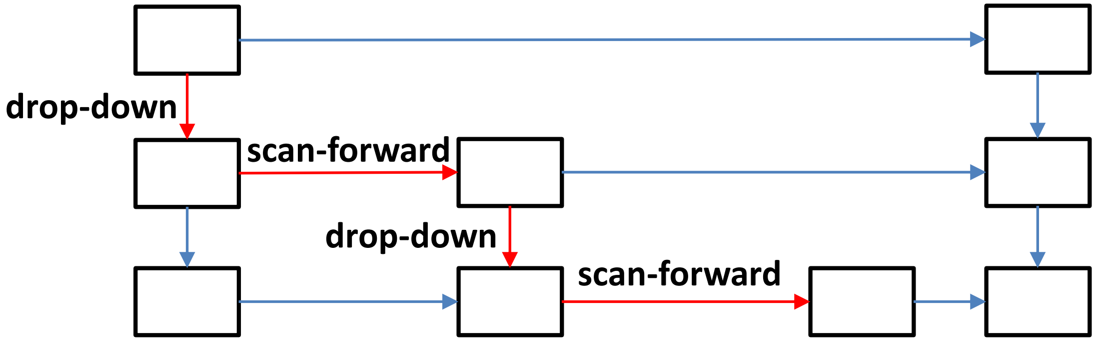
1. Space for nodes storing sentinels
   - there are $2h + 2$ sentinels, where h be the skip list height
   - $E[h] \leq 2 + \log n$
   - expected space for sentinels is at most

$$E[2h + 2] = 2E[h] + 2 \leq 6 + 2\log n$$

2. Space for nodes storing keys
   - Let $|S_i|$ be the number of keys in list S_i
     - $E[|S_i|] = \frac{n}{2^i}$
   - expected space for keys is $E\left[\sum_{i \geq 0} |S_i|\right] = \sum_{i \geq 0} \frac{n}{2^i} = 2n$

- Total expected space is $\Theta(n)$

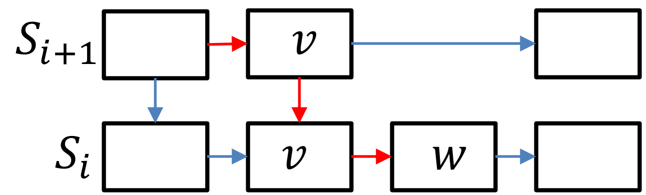
Skip List Analysis: Expected Running Time



- *search*, *insert*, and *delete* are dominated by the running time of *getPredecessors*
- So let us analyze the expected time of *getPredecessors*
- In *getPredecessors*, running time is proportional to the number of 'drop-down' and 'scan-forward'
- We can 'drop-down' at most h times, where h is skip list height
    - expected height h is $O(\log n)$
    - total expected time spent on 'drop-down' operations is $O(\log n)$
- Will show on the next slide that the expected number of 'scan-forward' is also $O(\log n)$
- So the expected running time is $O(\log n)$

Skip List Analysis: Expected Running Time

- What about ‘scan-forward’?
  - assume $i < h$ (if $i = h$, then we are at the top list and do not scan forward at all)
  - let v be leftmost key in S_i we visit during search
    - we v reached by dropping down from S_{i+1}
  - let w be the key right after v
    - height of tower of w in this case is at least i



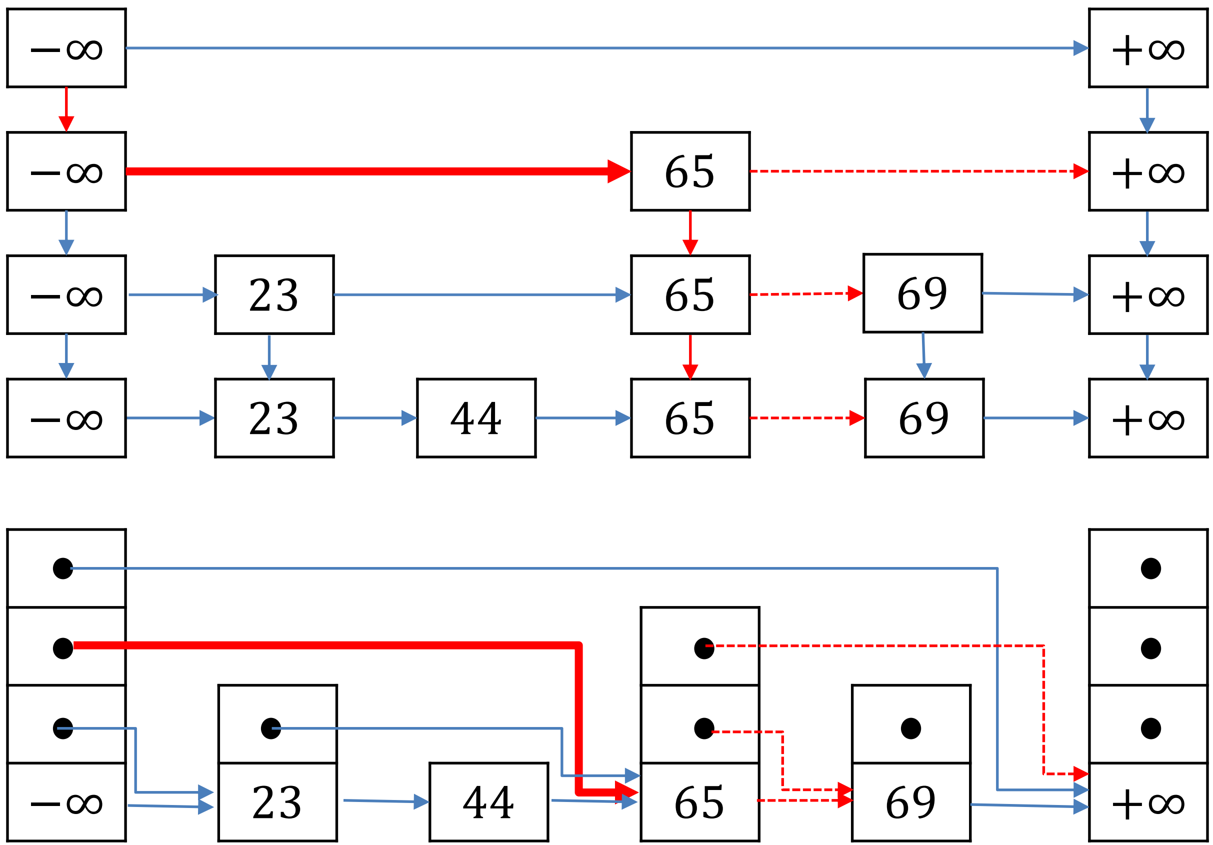
  - What is the probability of scanning from v to w?
  - If we do scan forward from v to w, then w did not exist in S_{i+1}
    - otherwise, we would scan forward from v to w in S_{i+1}
    - in other words, we always enter the tower of any node ‘at the top’
  - Thus if we do scan forward from v to w, then the tower of w has height i
    - P(tower of w has height i | tower of w has height at least i) $=$ ½
    - we scan forward from v to w with probability at most ½
      - ‘at most’ because we could scan-down down if $key < w$
    - repeating the argument, the probability of scan-forward l times is at most $(1/2)^l$

$$E[\text{ number of scans}] = \sum_{l\geq 1} l \cdot \mathrm{P}(\text{scans} = l) = \sum_{l\geq 1} \mathrm{P}(\text{scans} \geq l) \leq \sum_{l\geq 1} \frac{1}{2^l} = 1$$

- Expected number of scan-forwards at any level is 1, over all levels h, which is $\mathrm{O}(\log n)$

Arrays Instead of Linked Lists

- As described now, they are no faster than randomized binary search trees
- Can save links by implementing each tower as an array
    - this not only saves space, but gives better running time in practice
    - when ‘scan-forward’, we know the correct array location to look at (level i)
- Search(67)



Summary of Skip Lists

- For a skip list with n items
  - expected space usage is $O(n)$
  - expected running time for search, insert, delete is $O(\log n)$
- Two efficiency improvements
  - use arrays for key towers for more efficient implementation
  - can show: a biased coin-flip to determine tower-height gives smaller expected run-times
  - with arrays and biased coin-flip skip lists are fast in practice and easy to implement

Outline

- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Re-ordering items

Re-ordering Items

- Unordered arrays (or lists) are among simplest data structures to implement
- But for Dictionary ADT
    - *search*: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Lists/arrays are a very simple a popular implementation
- Can we make search in unordered arrays (or lists) more effective in practice?
    - No: if items are accessed equally likely
    - Yes: otherwise
        - intuition: frequently accessed items should be in the front
- Two cases
    - know the access distribution beforehand
    - do not know access distribution beforehand
- For short lists or extremely unbalanced distributions this may be faster than AVL trees or Skip Lists, and easier to implement

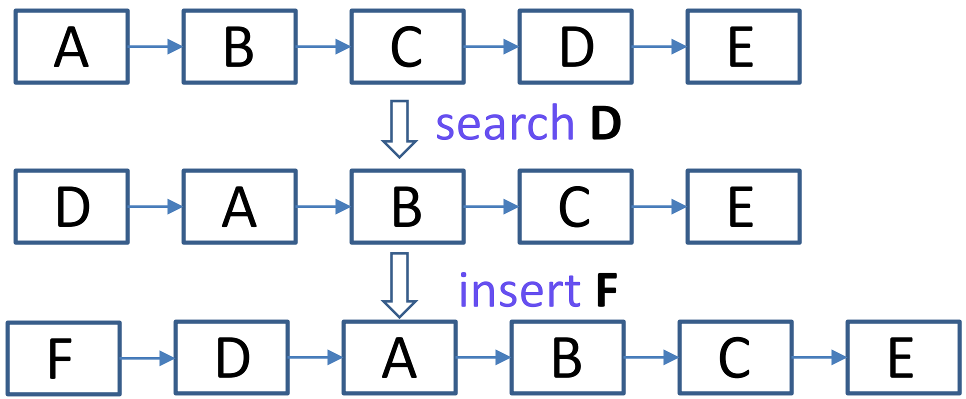
Optimal Static Ordering

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

- Order C A B D E has expected cost
  $$\frac{1}{26}\cdot 1+\frac{2}{26}\cdot 2+\frac{8}{26}\cdot 3+\frac{10}{26}\cdot 4+\frac{5}{26}\cdot 5 \approx 3.61$$
- Order D B E A C has expected cost
  $$\frac{10}{26}\cdot 1+\frac{8}{26}\cdot 2+\frac{5}{26}\cdot 3+\frac{2}{26}\cdot 4+\frac{1}{26}\cdot 5 \approx 2.54$$
- Claim: ordering items by non-increasing access-probability minimizes expected access cost, i.e. best *static* ordering
- Proof Idea: for any other ordering, exchanging two items that are out-of-order according to access probabilities makes total cost decrease

Dynamic Ordering

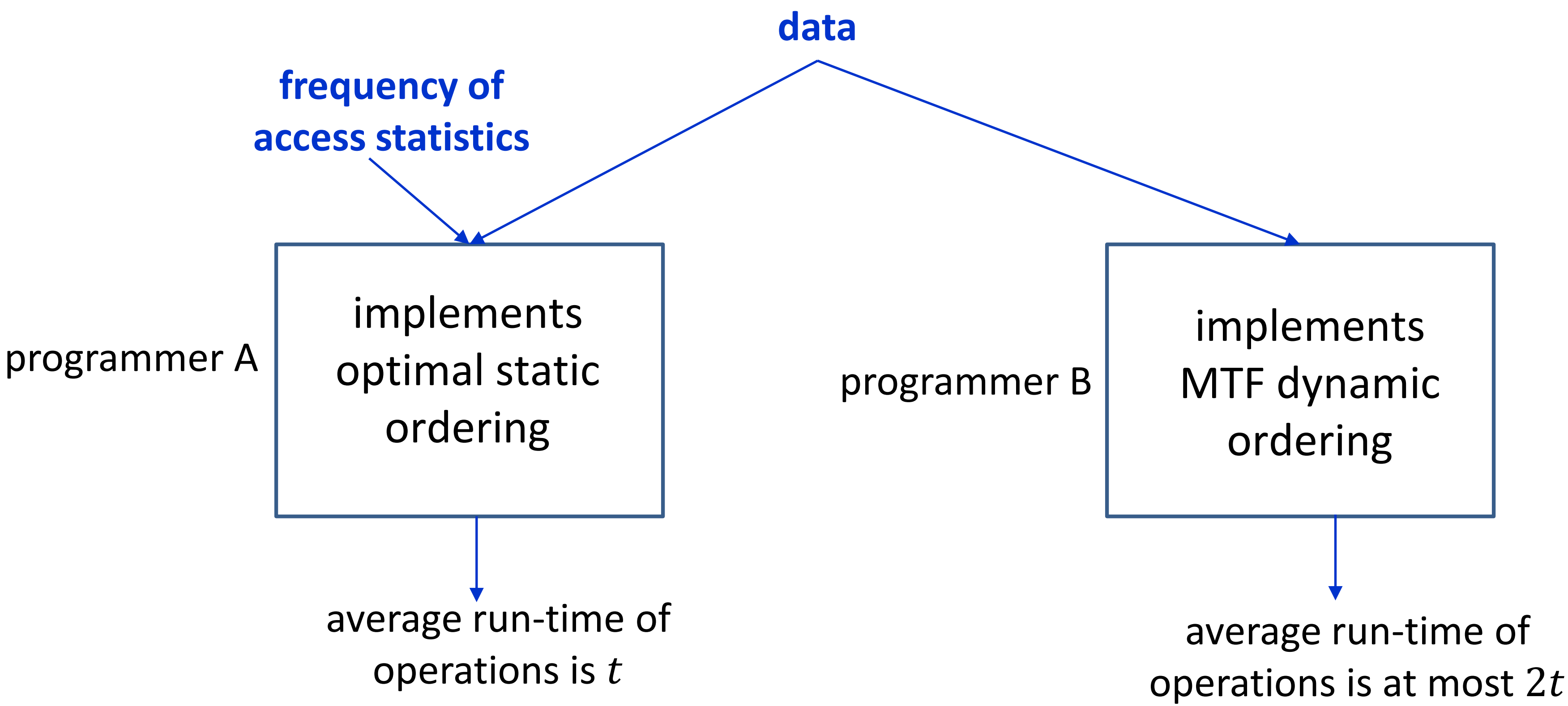
- What if we do *not know the access probabilities* ahead of time?
- Rule of thumb (*temporal locality*)
    - recently accessed item is likely to be accessed soon again
- In list: always insert at the front
- Move-To-Front heuristic (MTF): after search, move the accessed item to the front



- We can also do MTF on an array
    - but should then insert and search from the back so that we have room to grow

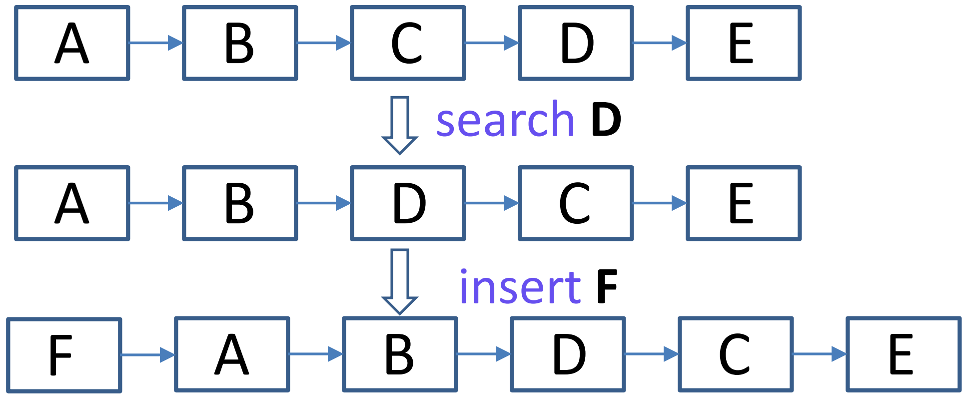
Dynamic Ordering: MTF

- Can show: MTF is "2-competitive"
  - no more than twice as bad as the optimal "offline" ordering



Dynamic Ordering: Transpose

- Transpose heuristic: Upon a successful search, swap accessed item with the immediately preceding item



- Avoids drastic changes MTF might do, while still adapting to access patterns
- Worst case is $\Theta(n)$ for both transpose and MTF