# CS 240 – Data Structures and Data Management

## Module 5: Other Dictionary Implementations

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2024

# Outline

- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Biased Search Requests

# Outline

- Dictionaries with Lists Revisited
    - Dictionary ADT
        - implementations so far
    - Skip Lists
    - Biased Search Requests

# 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**

$$\boxed{37} \to \boxed{44} \to \boxed{65} \to \boxed{69} \to \boxed{79} \to \boxed{83}$$

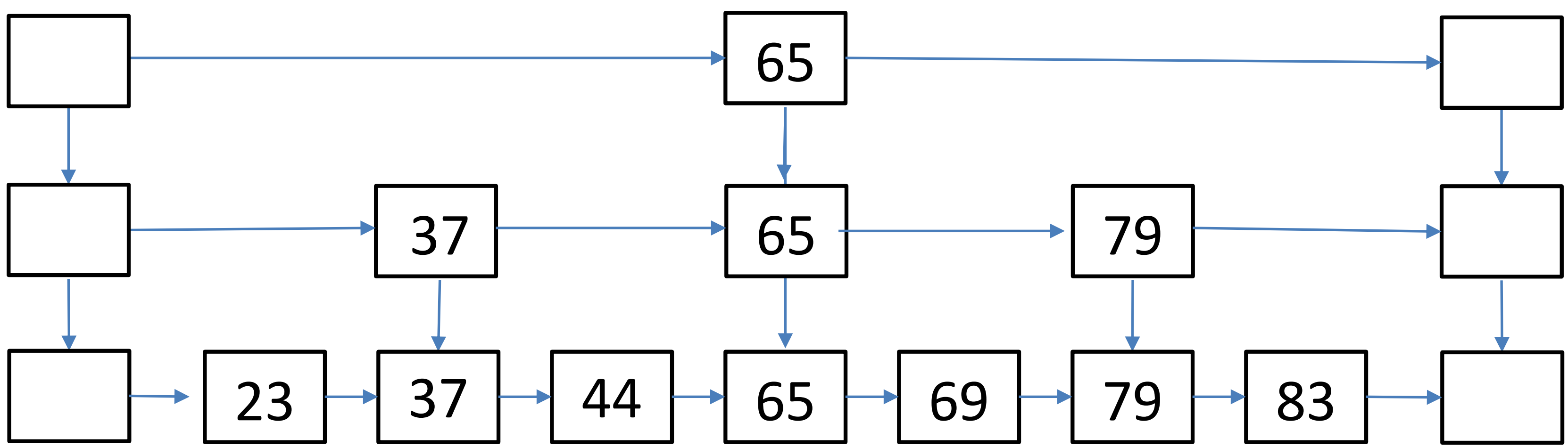
    - simple implementation
    - $\Theta(n)$ search, insert and delete
    - search is the bottleneck, insert and delete would be $\Theta(1)$ if do search first and account for its running time separately
    - efficient search (like binary search) in ordered linked list?

# Outline

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

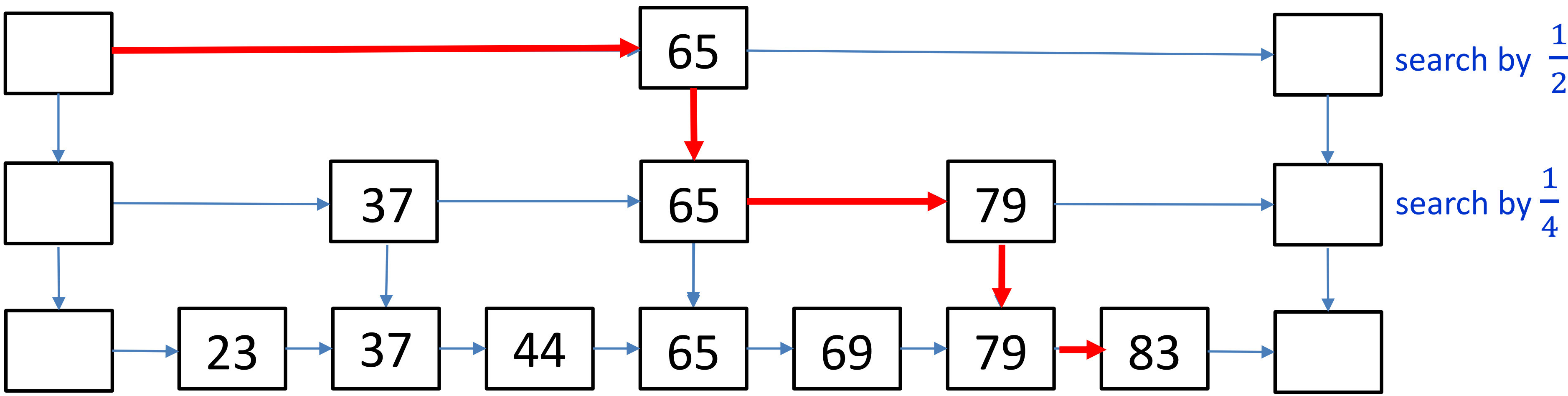
# Skip Lists: Motivation

- Build a *hierarchy* of linked lists to imitate binary search in ordered linked list wi
  - start from the bottom list and take every second item in the list above
  - downward links are needed to navigate from list above to the list below



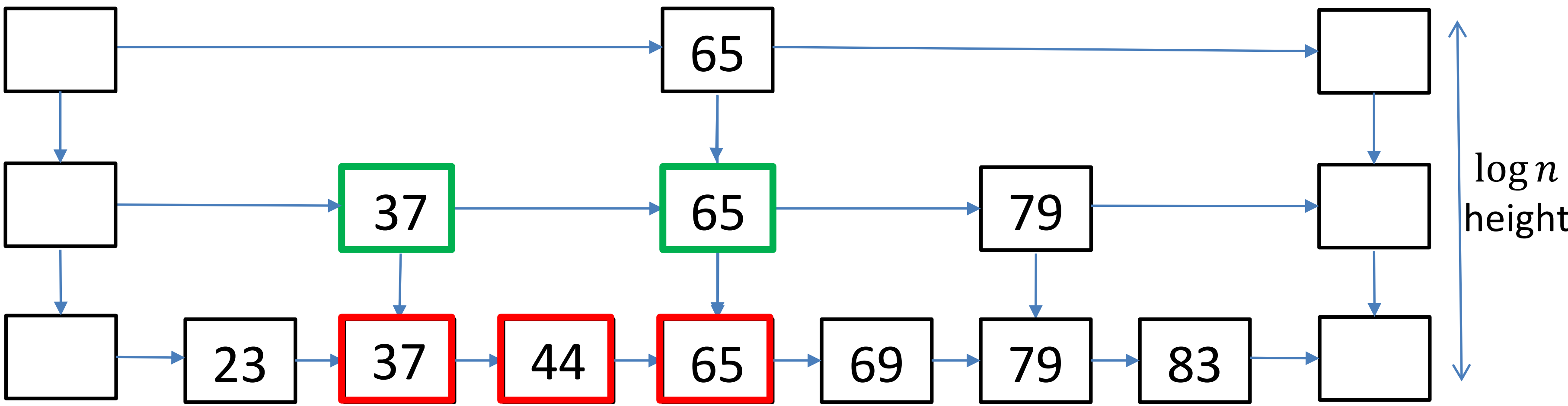
# Skip Lists: Motivation

- Search goes through the higher lists while possible, before dropping down to the list below
  - top list enables search by ½ of the list, next by ¼ of the list, and so on
- Search(83)



# Skip Lists: Motivation

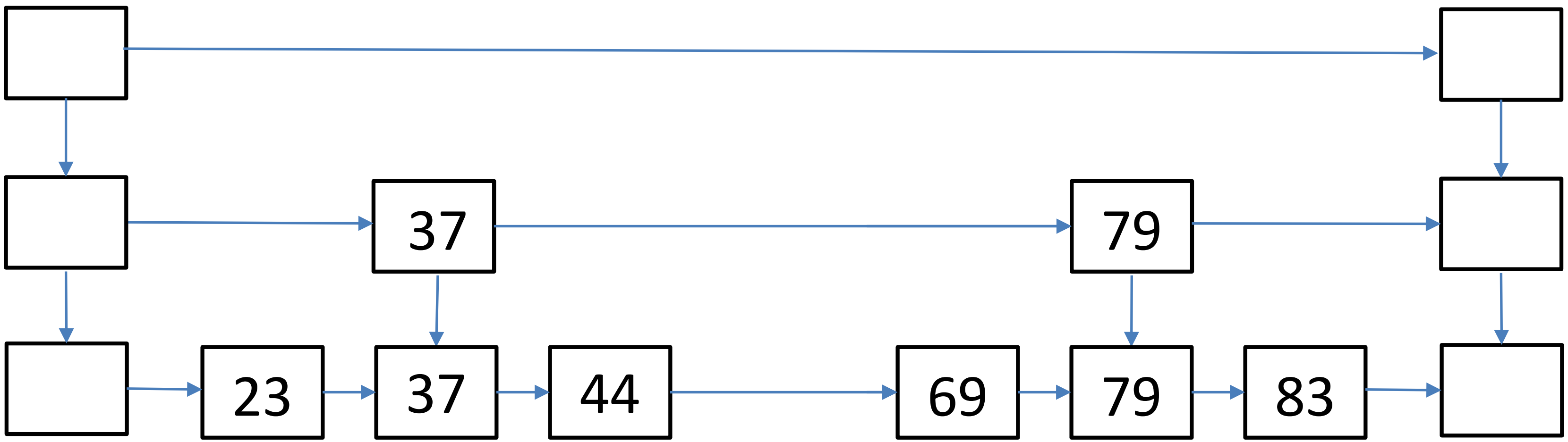
- Hierarchy of linked lists
  - each list has $1/2$ of items from the list below
  - total number of linked lists (height) is $\log n$
  - total number of nodes $\leq 2n$




- When searching, go through the highest level possible
  - thus visit at most two items at each level, and total time to search $\Theta(\log n)$

# Skip Lists: Motivation

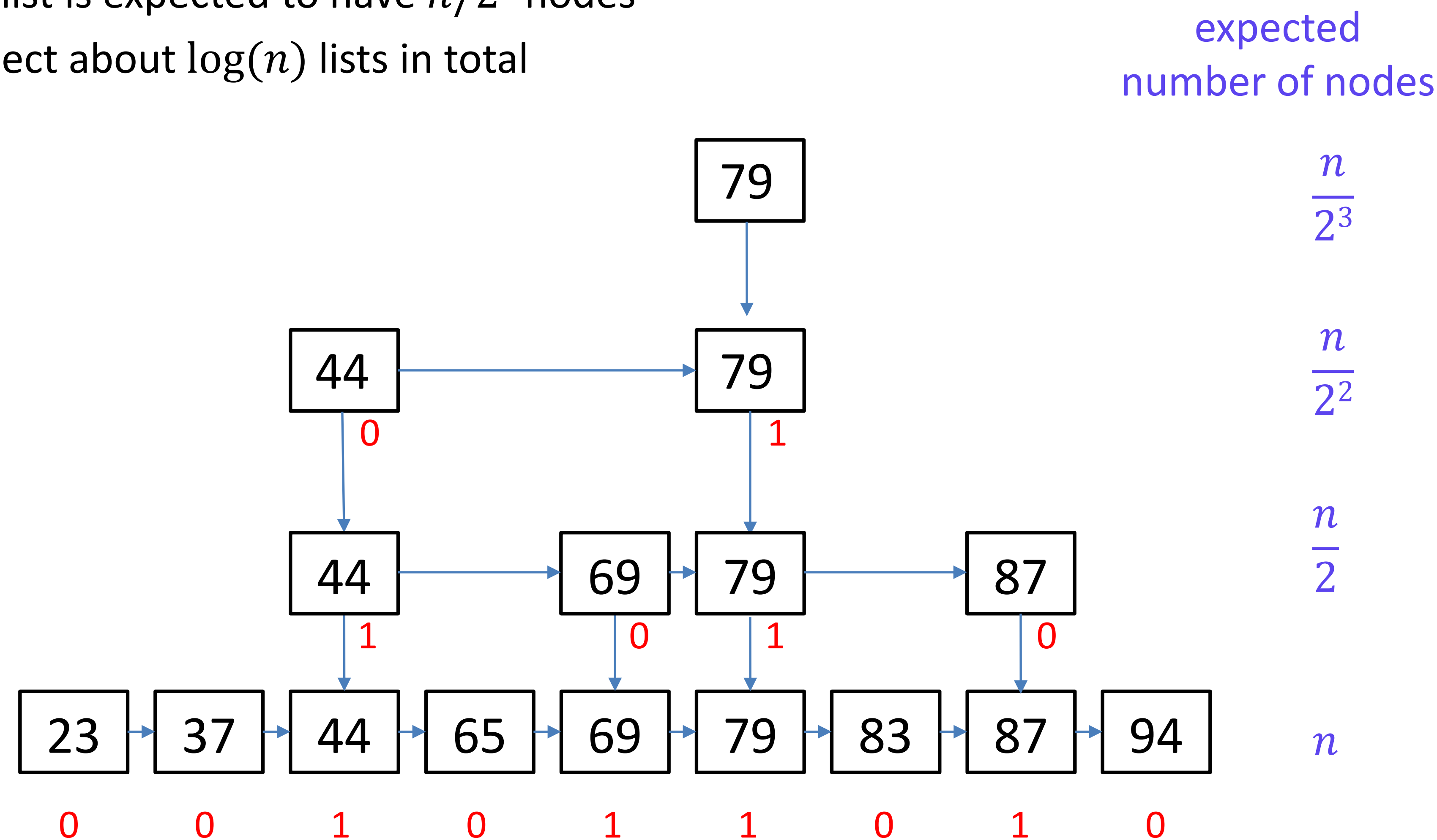
- Deleted 65, no longer every second item is in the list above




- **Big problem**: deletion or insertion of items ruins 'every second item is in the list above' property
    - crucial property for efficiency
- Thus the hierarchy of linked lists works only for *static* dictionary
    - know all items beforehand, and **do not** insert or delete
        - but in static case an ordered array is more efficient in practice (no links)
- *Randomization* enables hierarchical linked list **with** efficient insert and delete
    - instead of requiring a deterministic subset of items in list above, *randomly* chose a subset of the items in the list above

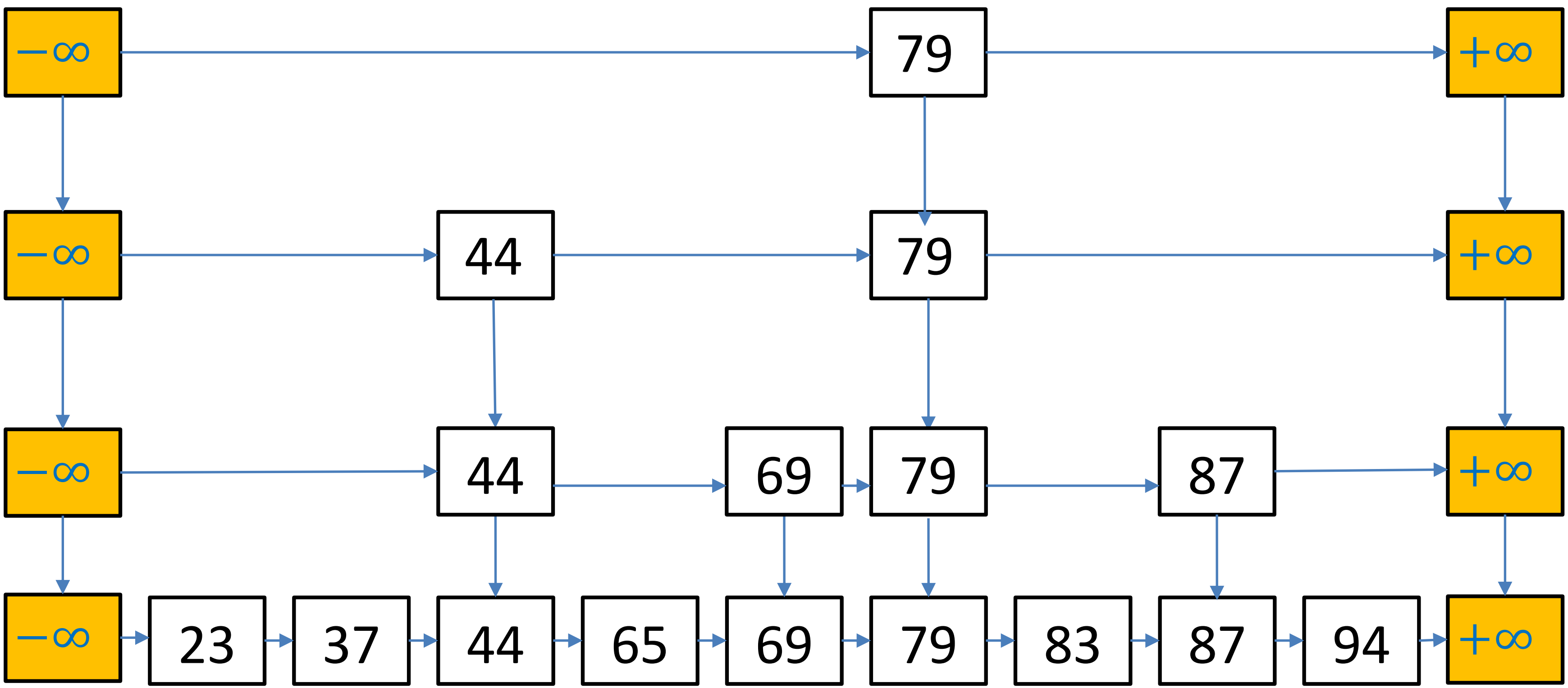
# Skip Lists: Motivation

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



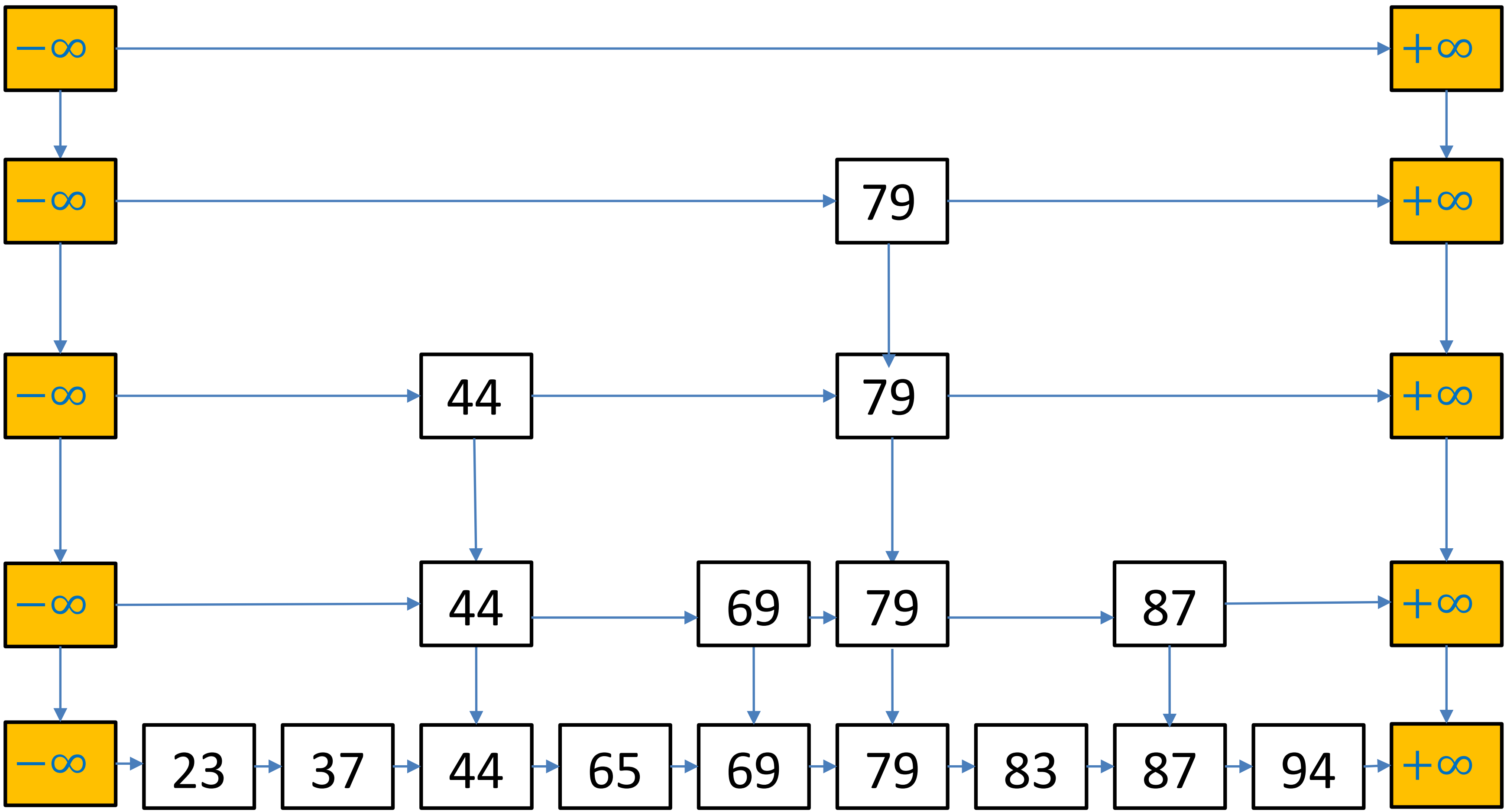
# Skip Lists: Motivation

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



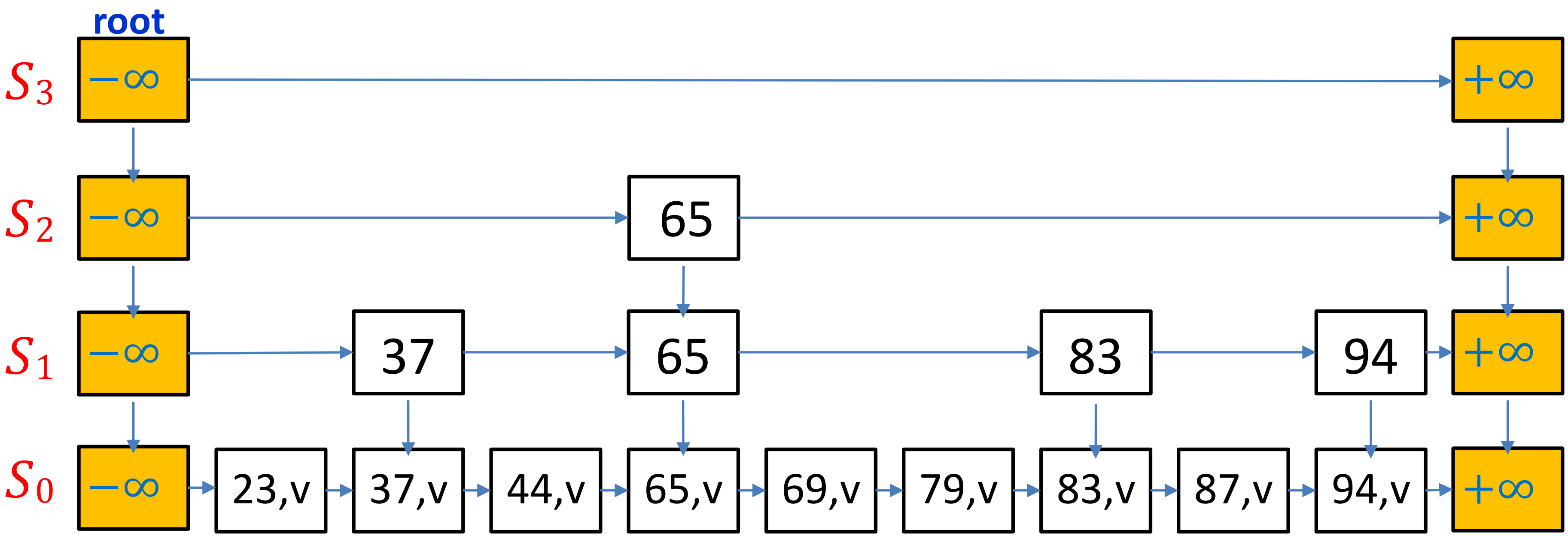
# Skip Lists: Motivation

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



# Skip Lists [Pugh'1989]

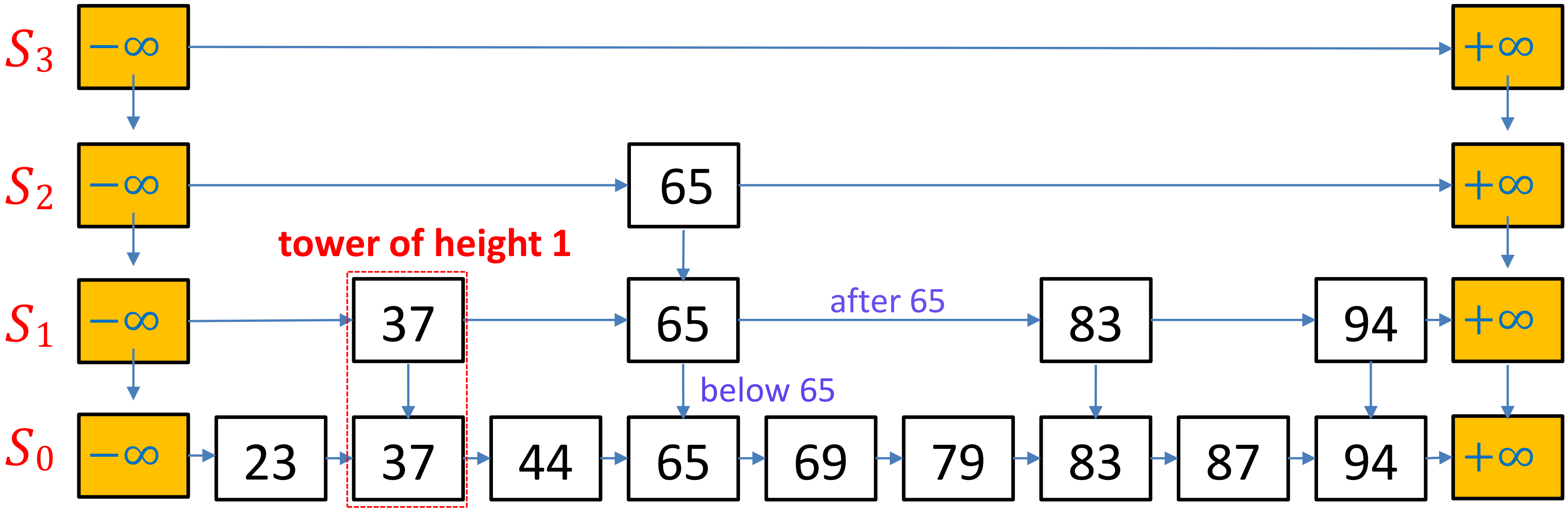
- A hierarchy $S$ of ordered linked lists (*levels*) $S_0, S_1, \dots, S_h$
  - $S_0$ contains the KVPs of $S$ in non-decreasing order
  - other lists store only keys
  - each $S_i$ contains special keys (sentinels) $-\infty$ and $+\infty$
  - each $S_i$ is randomly generated subsequence of $S_{i-1}$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




- $n$ is number of KVP (number of items stored); in this example, $n = 9$

# Skip Lists

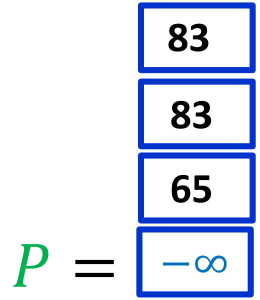
- Show only keys from now on

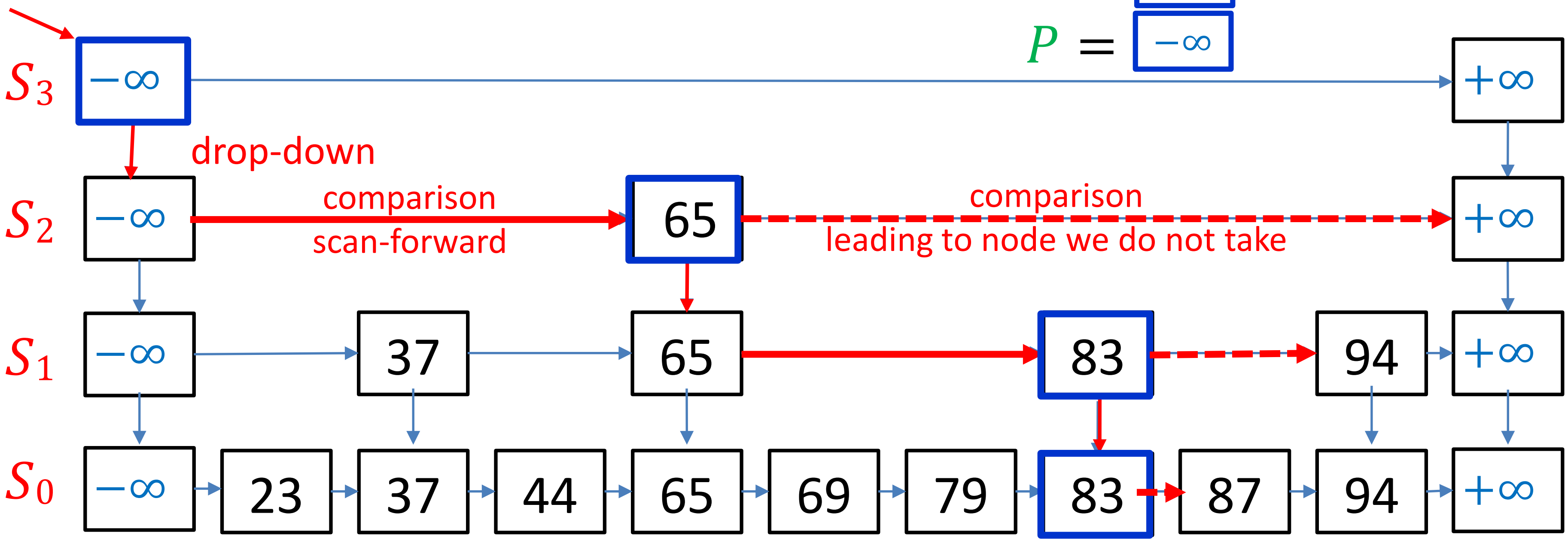



- Each KVP belongs to a *tower* of nodes
  - tower height is number of nodes $-1$
- Height of the skip list is the maximum height of any tower
  - height is 3 in this example
- Each node $p$ has references to after$(p)$ and below$(p)$
- There are (usually) more nodes than keys

# Search in Skip Lists

- search(87)




- For each level, **predecessor** of key $k$ is
  - If key $k$ is present at the level: node before node with key $k$
  - if key $k$ is not present at the level: node before node where $k$ would have been
- $P$ collects predecessors of key $k$ for all levels
  - nodes where we drop down and the rightmost node in $S_0$ with key $< \ k$
  - 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 $S_0$

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

**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*$(k)$

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

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

**else return** 'not found, but would be after $q$'

# Insert in Skip Lists

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

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

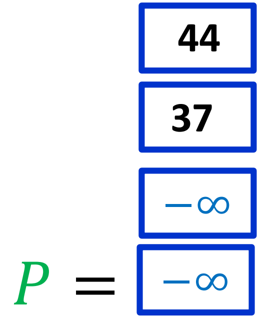
$S_1 \longleftarrow$ insert new item with probability ½

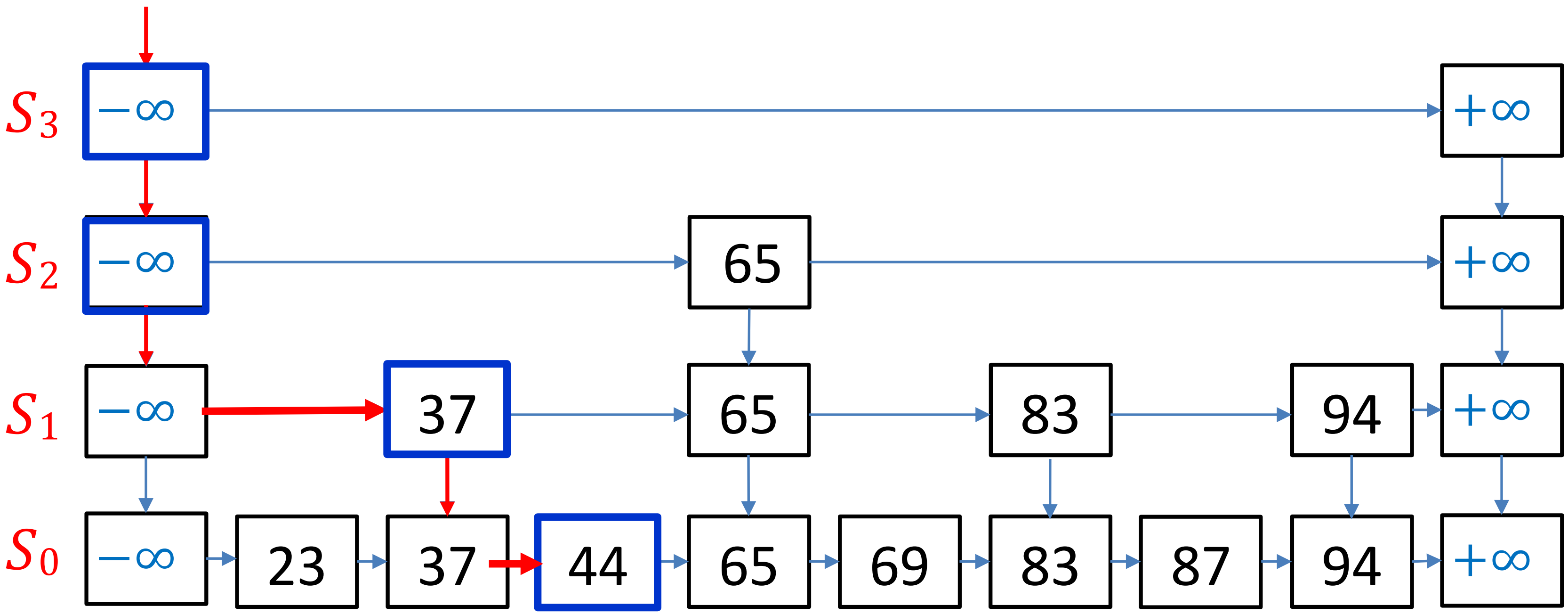
$S_0 \longleftarrow$ insert new item

- 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

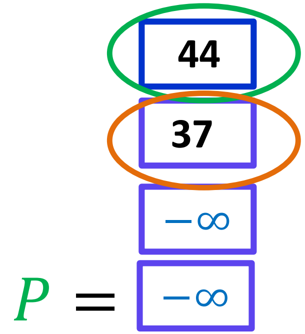
- *skipList::insert*$(52, v)$
- coin tosses: $H, T \Rightarrow i = 1$
- *getPredecessors*$(52)$

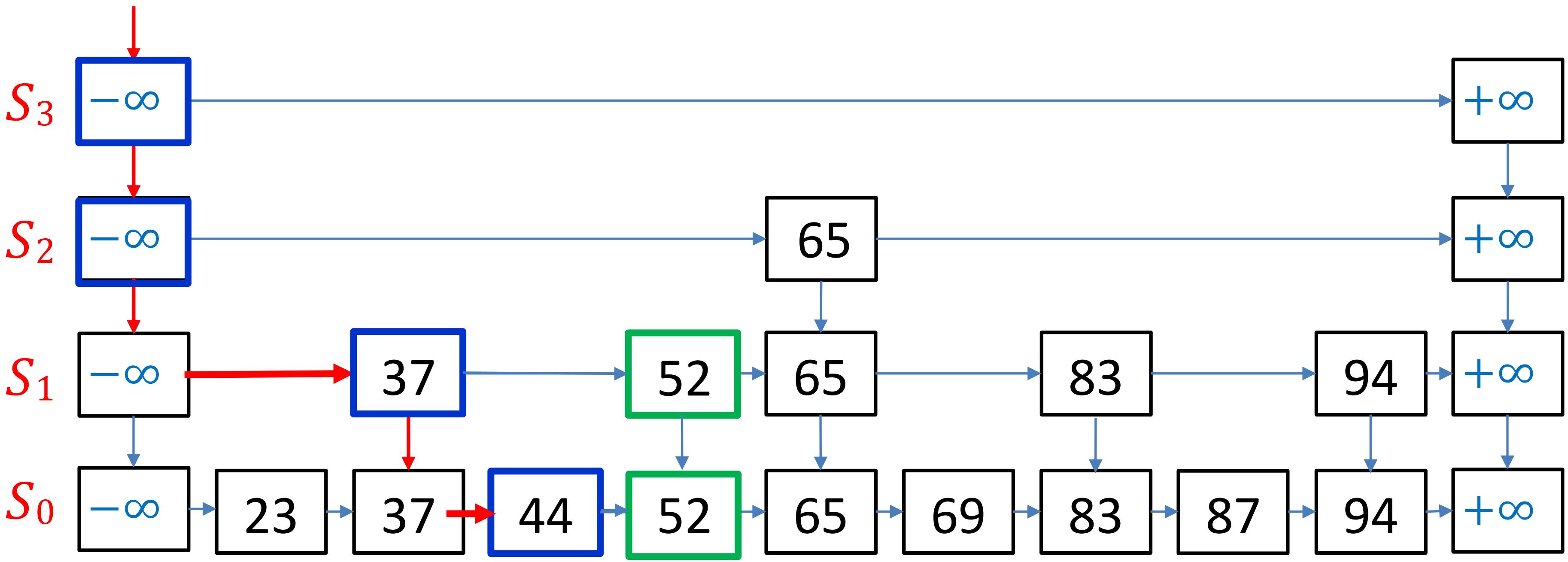


# Insert in Skip Lists: Example 1

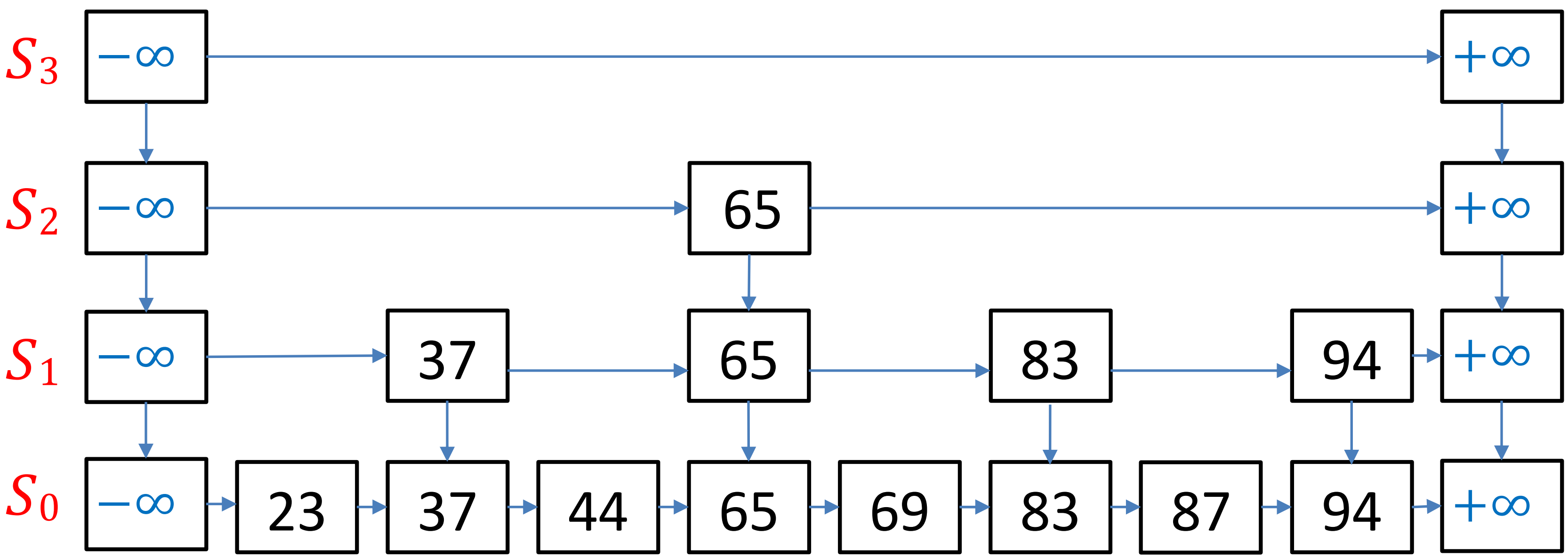
- *skipList::insert*$(52, v)$
- coin tosses: $H, T \Rightarrow i = 1$
- *getPredecessors*$(52)$
- now insert into $S_0$ and $S_1$



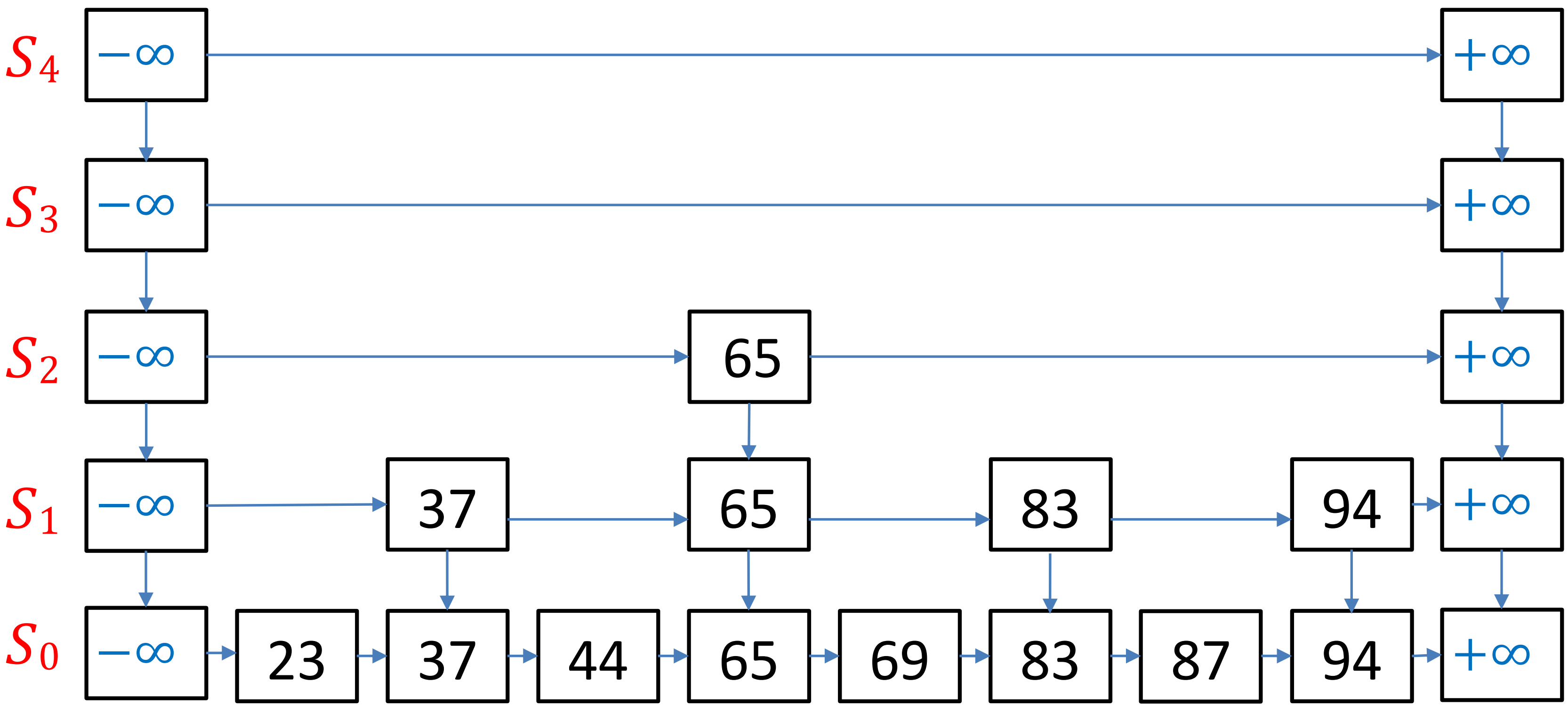
# Insert in Skip Lists: Example 2

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



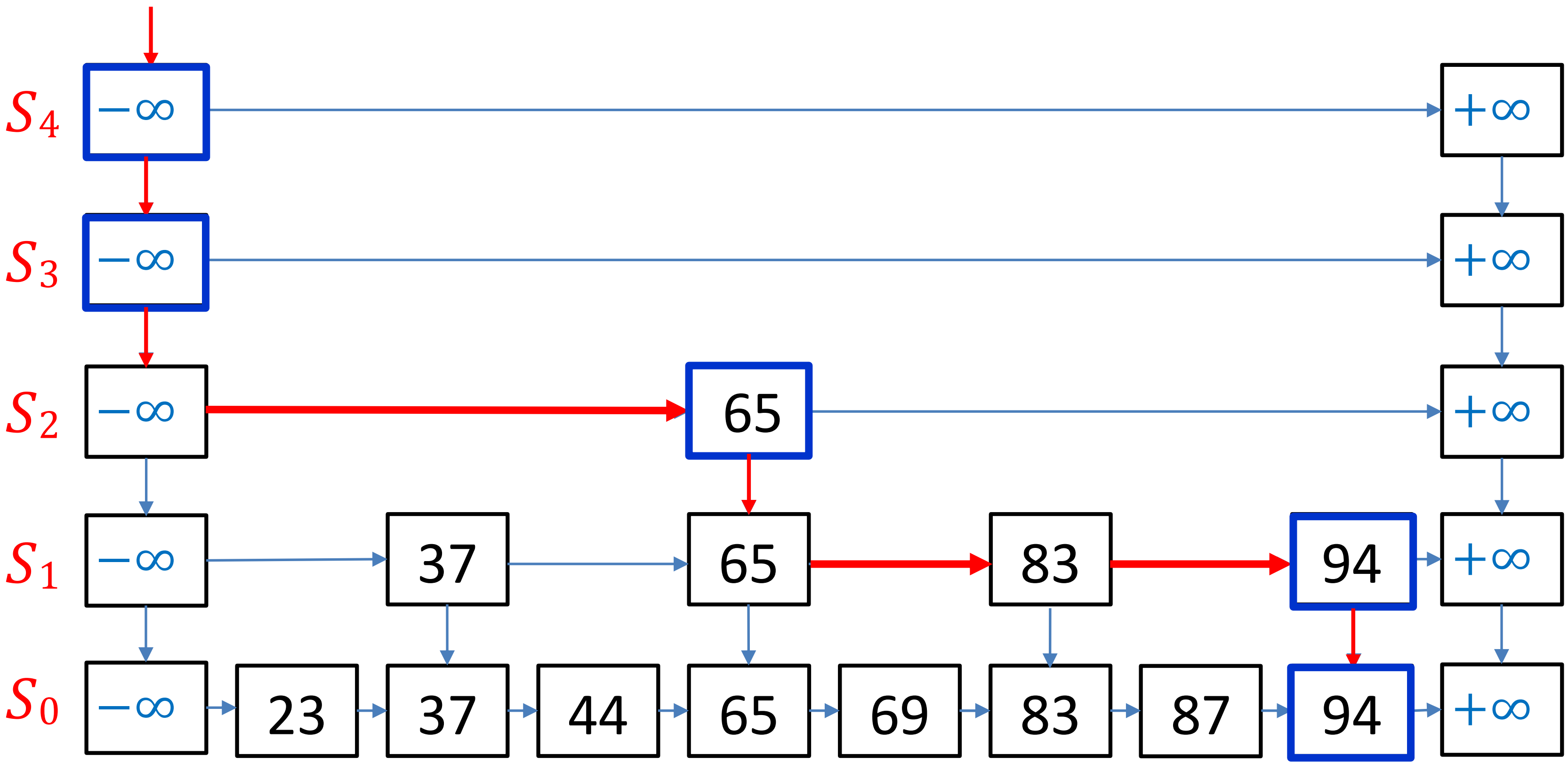
# Insert in Skip Lists: Example 2

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



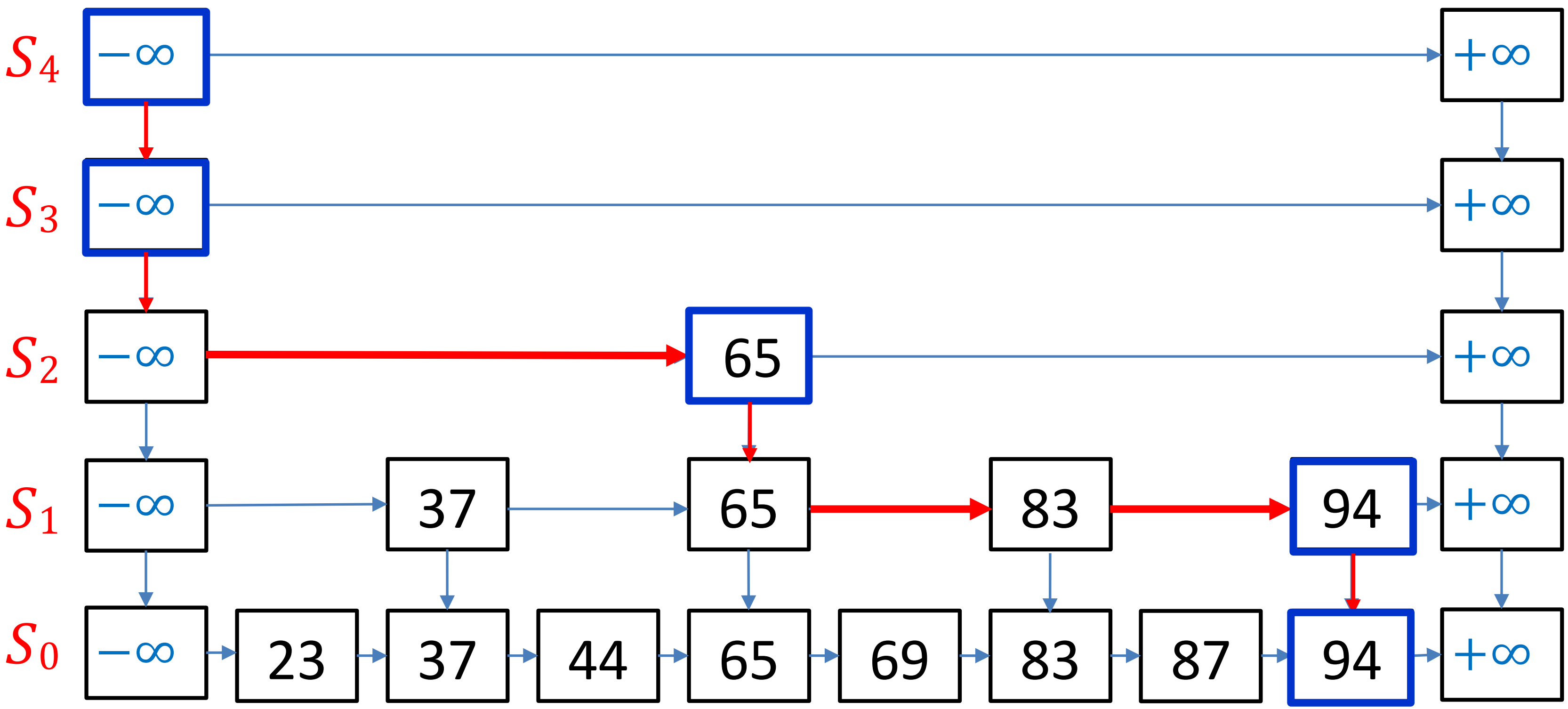
# Insert in Skip Lists: Example 2

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



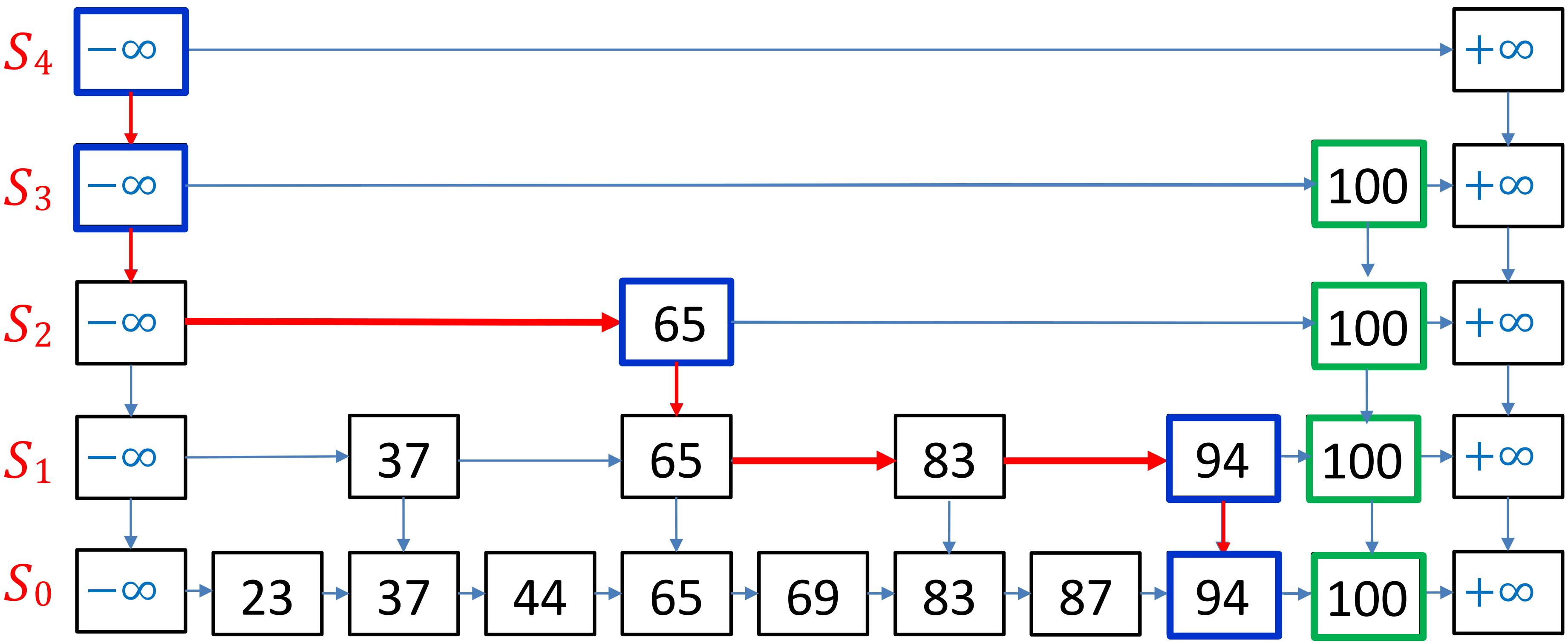
# Insert in Skip Lists: Example 2

- *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*$(k, v)$

**for** $(i \leftarrow 0;\ random(2) = 1; i \leftarrow i + 1)\ \{\}$ // random tower height

**for** $(h \leftarrow 0, p \leftarrow root.below; p \neq NILL;\ p \leftarrow p.bellow)$ **do** $h{+}{+}$

**while** $i \geq h$ // increase skip-list height if needed

$root$ $\leftarrow$ new sentinel-only list linked in appropriately

$h{+}{+}$

$P \leftarrow$ *getPredecessors*$(k)$

$p \leftarrow P.pop()$

$zBellow \leftarrow$ new node with $(k, v)$ inserted after $p$ // insert $(k, v)$ in $S_0$

**while** $i > 0$ // insert $k$ in $S_1\ S_2, \ldots, S_i$

$p \leftarrow P.pop()$

$z \leftarrow$ new node with $k$ added after $p$

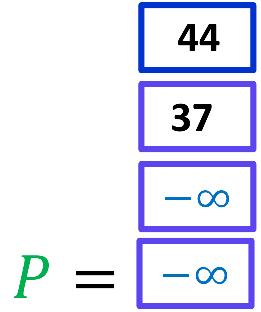
$z.below \leftarrow zBellow$

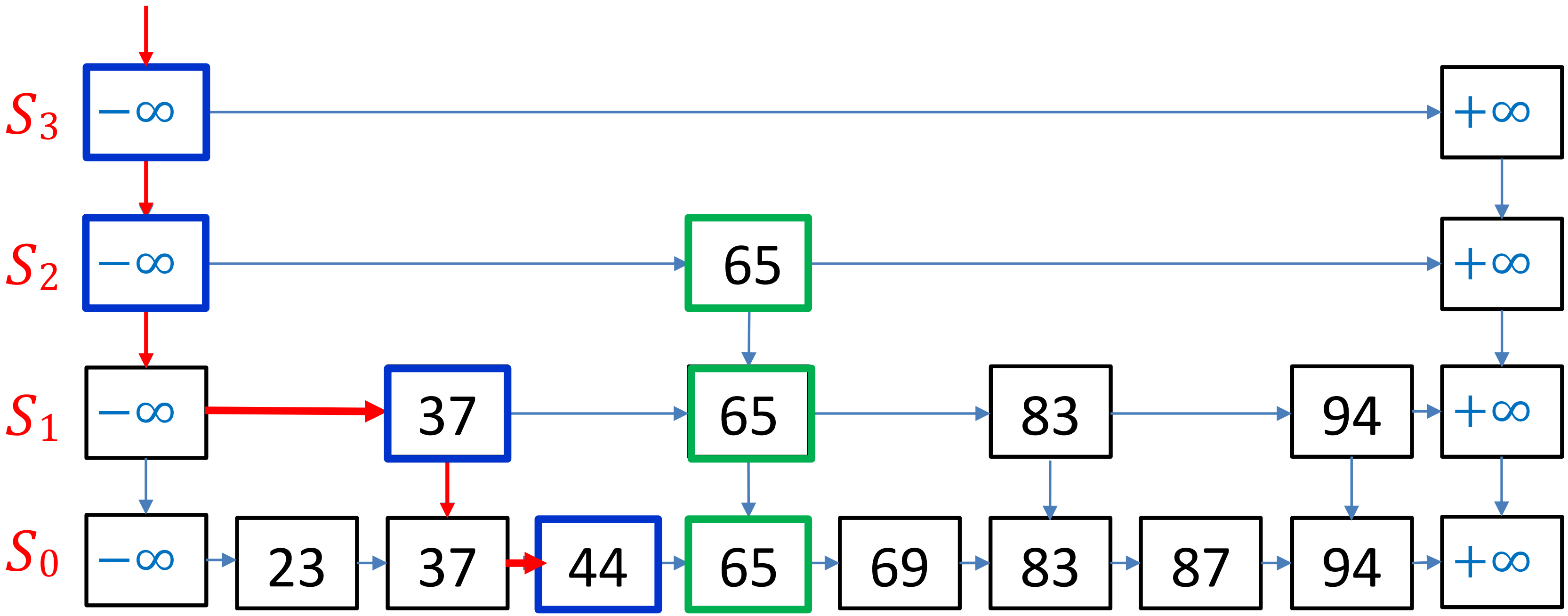
$zBellow \leftarrow z$

$i \leftarrow i - 1$

# Example: Delete in Skip Lists

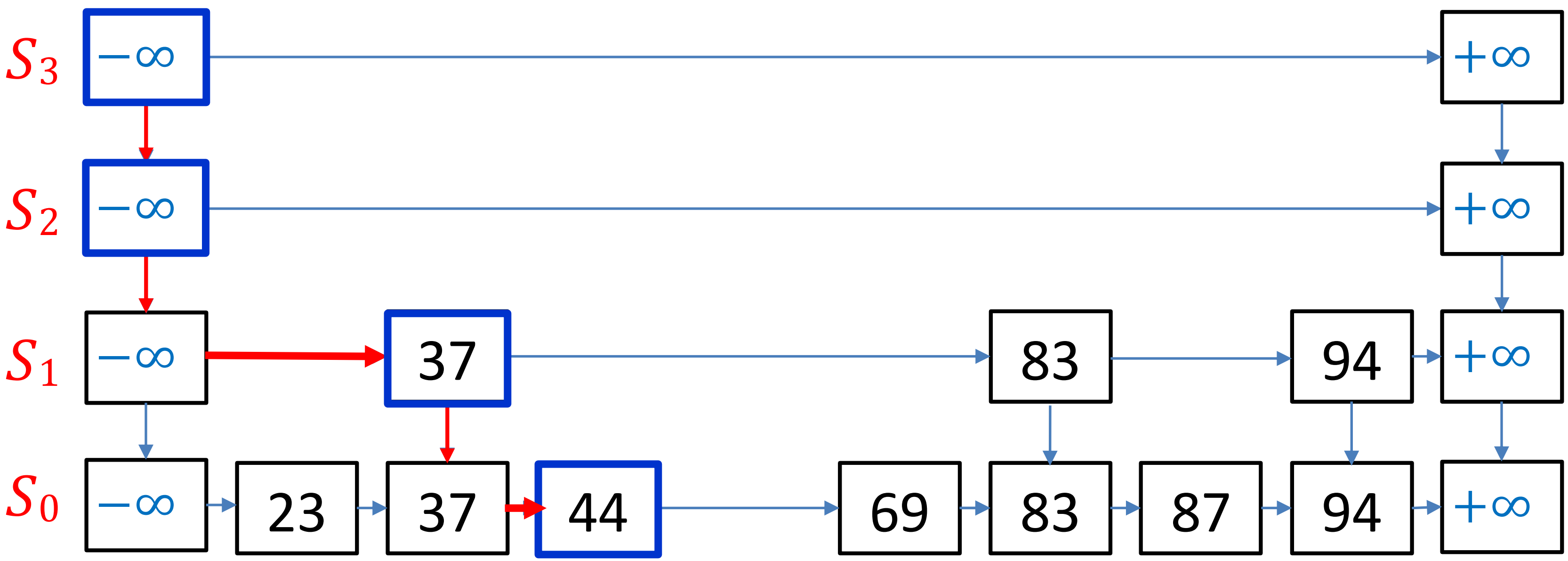
- *skipList::delete*$(65)$
  - first *getPredecessors*$(S, 65)$
  - then delete key $65$ from all $S_i$
    - $P$ has predecessor of each node to be deleted



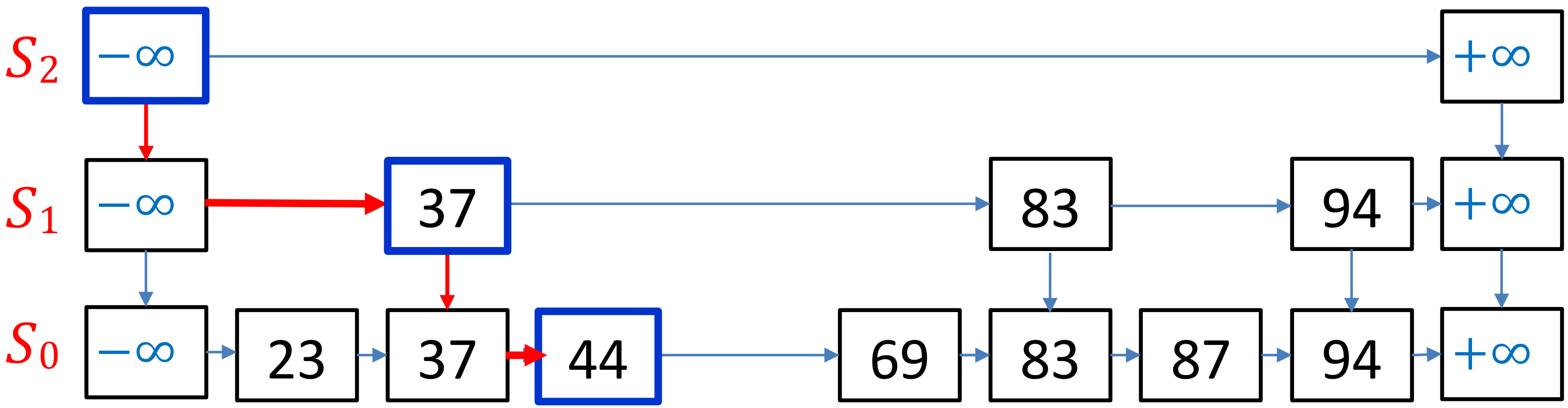
# Example: Delete in Skip Lists

- *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

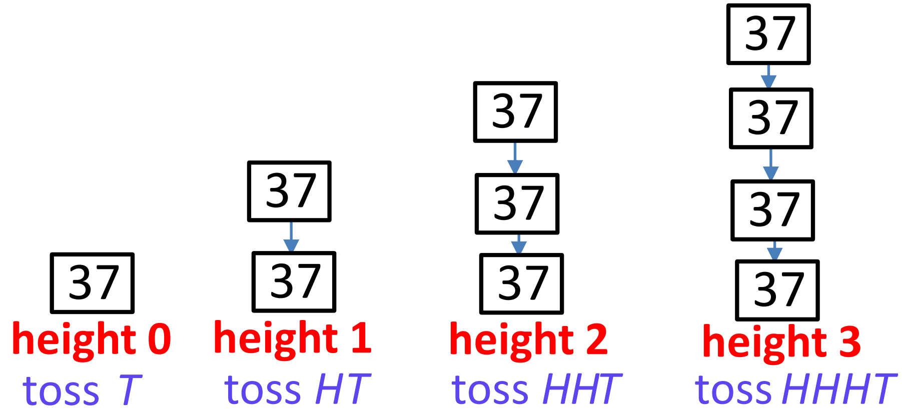
- *skipList::delete*$(65)$
  - first *getPredecessors*$(S, 65)$
  - then delete key $65$ from all $S_i$
    - $P$ has predecessor of each node to be deleted
  - height decrease: delete all unnecessary $S_i$, if any



# Delete in Skip Lists

```
skipList::delete(k)
        P ← getPredecessors(k)
        while P is non-empty
                p ← P.pop()                          // predecessor of k in some layer
                if p.after.key = k
                        p.after ← p.after.after
                else break                           // no more copies of k
        p ← 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 ← p.below.below                    // removes the second empty list
          p.after.below ← p.after.below.below
```

# Skip List Analysis




- Let $X_k$ be the height of tower for key $k$

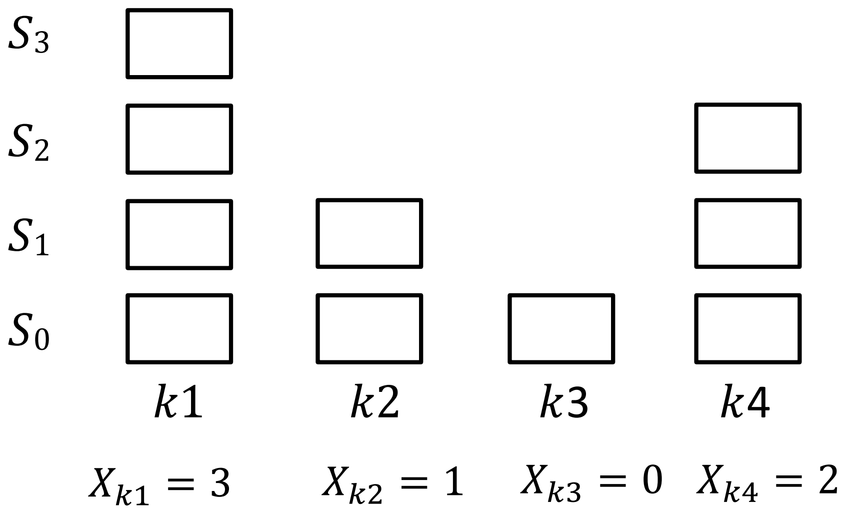
$$P(X_k \geq 1) = \frac{1}{2} \qquad P(X_k \geq 2) = \frac{1}{2} \cdot \frac{1}{2} \qquad P(X_k \geq 3) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2}$$

toss *H....* toss *HH....* toss *HHH....*

- In general $$P(X_k \geq i) = P(\underbrace{H\ H\ \ldots\ H}_{i \text{ times}}) = \left(\frac{1}{2}\right)^i$$

- In the worst case, the height of a tower could be arbitrary large
  - no bound on height in terms of $n$
- Operations could be arbitrarily slow, and space requirements arbitrarily large
  - but this is exceedingly unlikely
- Let us analyse *expected* run-time and space-usage (randomized data structure)

# Skip List Analysis




$|S_3| = 1$

$|S_2| = 2$

$|S_1| = 3$

$|S_0| = 4$

- 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

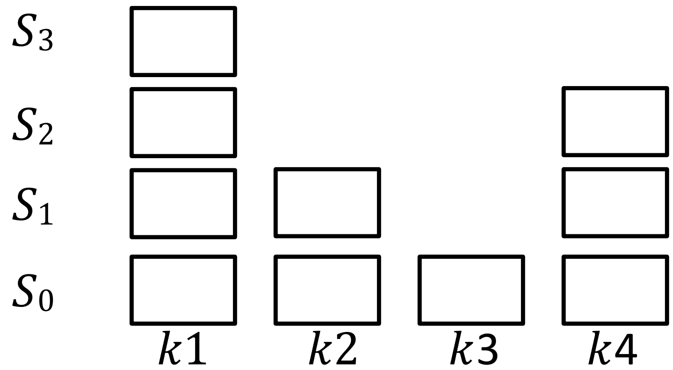
# Skip List Analysis

| | | | | | |
|---|---|---|---|---|---|
| $S_3$ | $\boxed{I_{3,\,k1} = 1}$ | $I_{3,\,k2} = 0$ | $I_{3,\,k3} = 0$ | $I_{3,\,k4} = 0$ | $\lvert S_3 \rvert = 1$ |
| $S_2$ | $\boxed{I_{2,\,k1} = 1}$ | $I_{2,\,k2} = 0$ | $I_{2,\,k3} = 0$ | $\boxed{I_{2,\,k4} = 1}$ | $\lvert S_2 \rvert = 2$ |
| $S_1$ | $\boxed{I_{1,\,k1} = 1}$ | $\boxed{I_{1,\,k2} = 1}$ | $I_{1,\,k3} = 0$ | $\boxed{I_{1,\,k4} = 1}$ | $\lvert S_1 \rvert = 3$ |
| $S_0$ | | | | | |
| | $X_{k1} = 3$ | $X_{k2} = 1$ | $X_{k3} = 0$ | $X_{k4} = 2$ | |

- 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 $S_i$
- Let $I_{i,\,k} = \begin{cases} 0 & \text{if } X_k < i \\ 1 & \text{if } X_k \geq i \end{cases} = \begin{cases} 0 & \text{if list } S_i \text{ does not include key } k \\ 1 & \text{if list } S_i \text{ includes key } k \end{cases}$
- $|S_i| = \sum_{key\ k} I_{i,\,k}$
- $E[|S_i|] = E\left[\sum_{key\ k} I_{i,\,k}\right] = \sum_{key\, k} E[I_{i,\,k}] = \sum_{key\, k} P(I_{i,\,k} = 1) = \sum_{key\, k} P(X_k \geq i) = \sum_{key\, k} \frac{1}{2^i} = \frac{n}{2^i}$
- The expected length of list $S_i$ is $\frac{n}{2^i}$

# Skip List Analysis

$S_4$ has only sentinels $\qquad I_4 = 0$




$I_3 = 1$

$I_2 = 1$

$I_1 = 1$

- $|S_i|$ 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
- $E[h] = E\left[1 + \sum_{i\geq 1} I_i\right] = 1 + \sum_{i\geq 1} E[I_i] = 1 + \sum_{i=1}^{\log n} E[I_i] + \sum_{i=1+\log n}^{\infty} E[I_i]$

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

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

# 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
- $E[h] = E\left[1 + \sum_{i \geq 1} I_i\right] =$

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

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

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

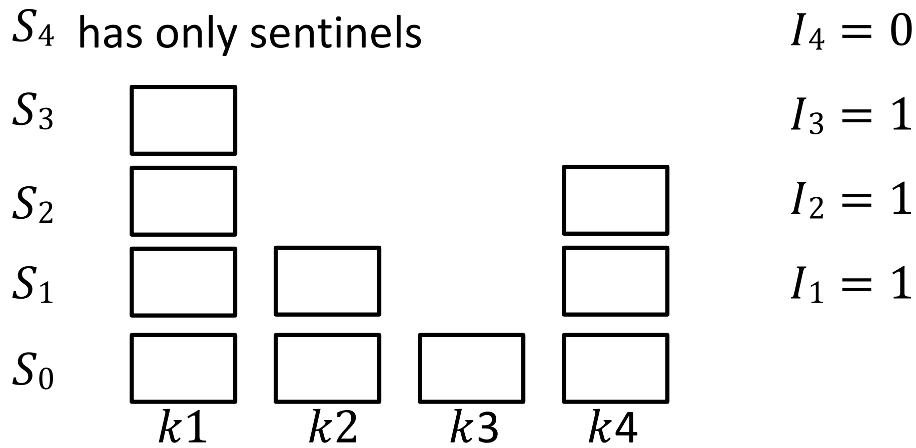
$$S = \sum_{i=0}^{\infty} \frac{1}{2^i} = \cancel{1} + \cancel{\frac{1}{2}} + \cancel{\frac{1}{2^2}} + \cancel{\frac{1}{2^3}} + \cdots$$

$$2S = \sum_{i=0}^{\infty} \frac{2}{2^i} = 2 + \cancel{1} + \cancel{\frac{1}{2}} + \cancel{\frac{1}{2^2}} + \cdots$$

$$2S - S = 2$$

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

# 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
- $$E[h] = E\left[1 + \sum_{i\geq 1} I_i\right] = 1 + \sum_{i\geq 1} E[I_i] = 1 + \sum_{i=1}^{\log n} E[I_i] + \sum_{i=1+\log n}^{\infty} E[I_i]$$
  $$\leq 1 + \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i}$$
  $$\leq 1 + \log n + \sum_{i=0}^{\infty} \frac{n}{2^{i+1+\log n}}$$
  $$= 1 + \log n + 1$$
- Expected skip list height $\leq 2 + \log n$

# Skip List Analysis: Expected Space

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

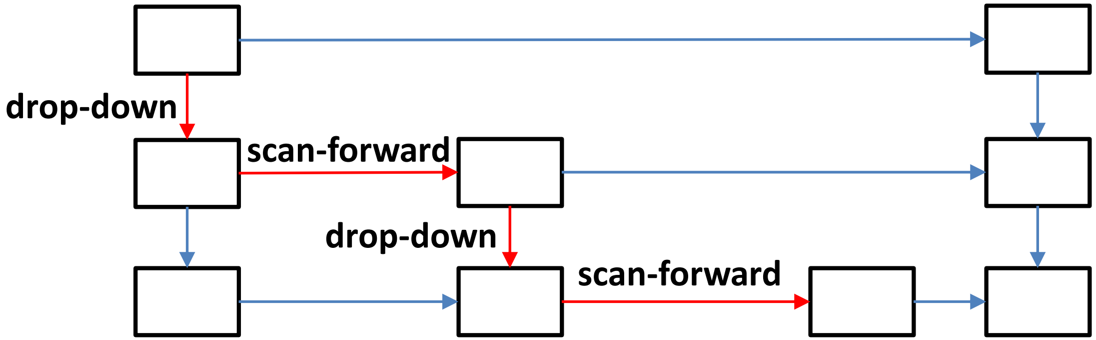
1. Space for nodes storing sentinels
   - there are $2h + 2$ sentinels, where $h$ be the skip list height
   - $E[h] \leq 2 + \log n$
   - expected space for sentinels is at most

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

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 $E\left[\sum_{i \geq 0} |S_i|\right] = \sum_{i \geq 0} E[|S_i|] = \sum_{i \geq 0} \frac{n}{2^i} = 2n$

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

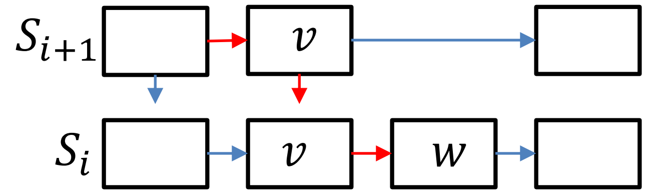
# Skip List Analysis: Expected Running Time




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

# Expected Number of Scan-Forward Operations

- Number 'scan-forward' at level $i$
  - 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 $S_{i+1}$
  - 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 $i$) $= ½$
    - scan forward (i.e. at least one scan) from $v$ to $w$ with probability at most ½
      - 'at most' because we could scan-down down if search $key < w$
    - repeating argument, probability of scan-forward at least $l$ times is at most $(1/2)^l$

$$E[\#\text{ scan-forward at level } i] = \sum_{l \geq 1} l \cdot \mathrm{P}(\text{scans} = l) = \sum_{l \geq 1} \mathrm{P}(\text{scans} \geq l) \leq \sum_{l \geq 1} \frac{1}{2^l} = 1$$

theorem in probability theory

# Expected Number of Scan-Forward Operations

- At level $i < h$: $E$[number of scan-forward] $\leq 1$
- Also, expected number of scan-forward at level $i$ $<$ number of keys at level $S_i$
  - $|S_i|$ is the number of keys in list on level $i$, and $E[|S_i|] = \frac{n}{2^i}$
- For ease of derivation, assume $n$ is a power of 2
- Expected number of scan-forward over all levels

$$\sum_{i\geq 0} E[\text{\# of scan−forward at level } i] =$$

$$= \sum_{i=1}^{\log n} E[\text{\# of scan−for at level } i] + \sum_{i=1+\log n}^{\infty} E[\text{\# of scan−for at level } i]$$

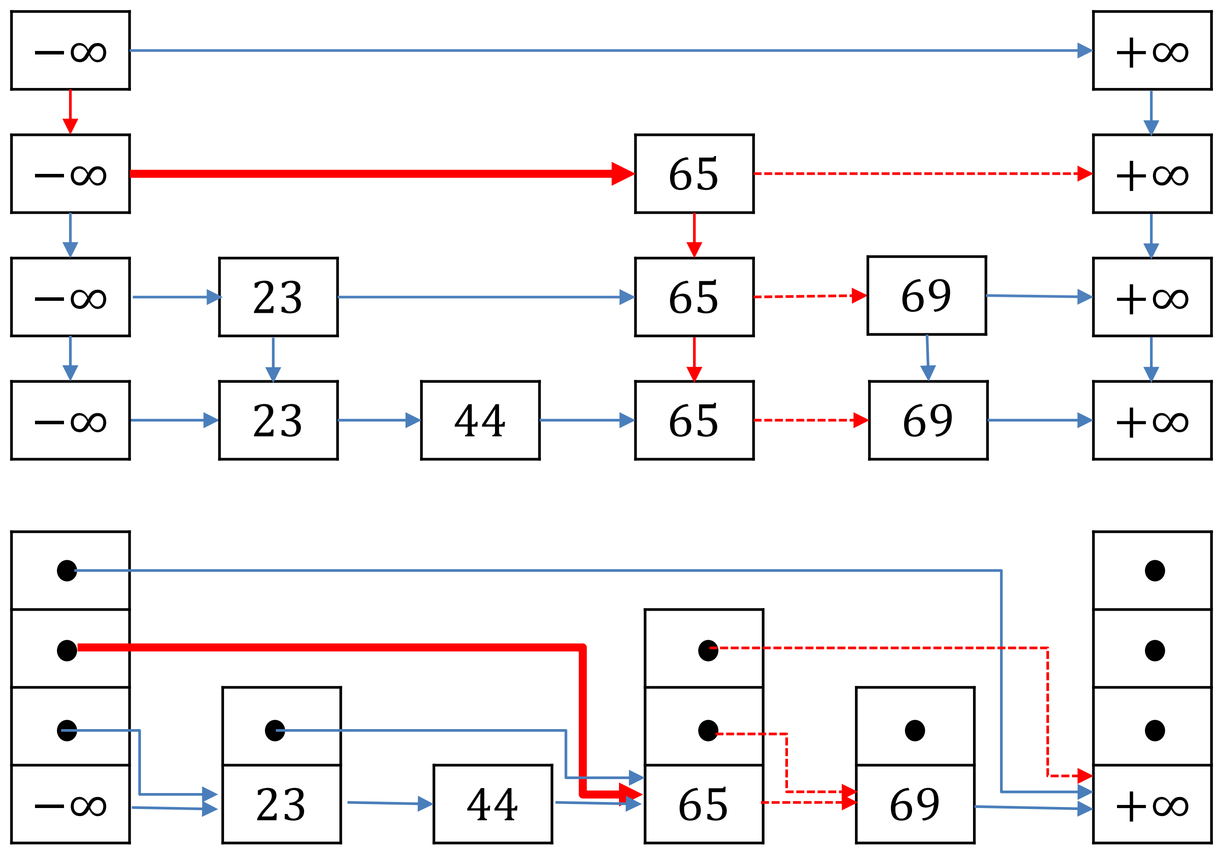
$$\leq \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i}$$

$$\leq \log n + 1$$

- Expected number of scan-forwards is $O(\log n)$

# 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
  - Biased Search Requests

# Improving Unsorted Lists/Arrays

- Unordered lists/arrays are among simplest data structures to implement
- But for Dictionary ADT
    - inefficient *search*: $\Theta(n)$
- Can we make search in unordered lists/arrays more effective in practice?
    - No if items are accessed equally likely
        - can show average-case search is $\Theta(n)$
    - Yes if the search requests are biased
        - some items are accessed much more frequently than others
        - 80/20 rule: 80% of outcomes result from 20% of causes
        - access = insertion or successful search
        - frequently accessed items should be in the front
            - two cases
                - know the access distribution beforehand
                    - optimal static ordering
                - do not know access distribution beforehand
                    - dynamic ordering

# 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 $A$ $B$ $C$ $D$ $E$ has expected cost

$$\frac{2}{26}\cdot 1+\frac{8}{26}\cdot 2+\frac{1}{26}\cdot 3+\frac{10}{26}\cdot 4+\frac{5}{26}\cdot 5 \approx 3.31$$

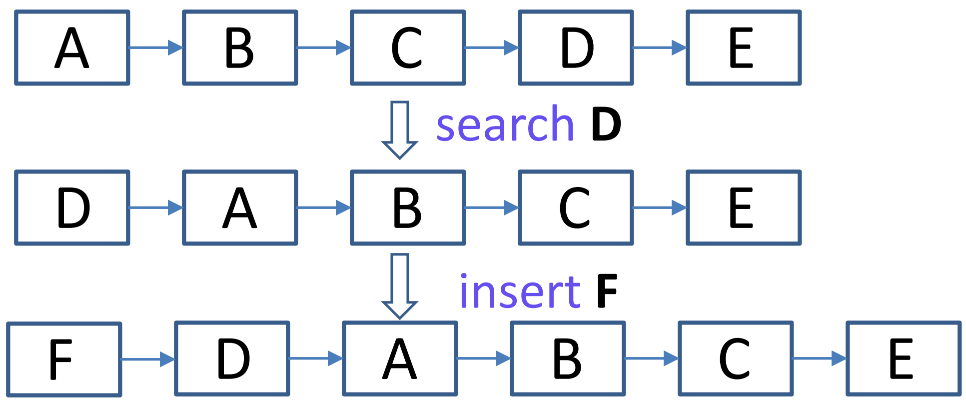
- Order $D$ $B$ $E$ $A$ $C$ has expected cost

$$\frac{10}{26}\cdot 1+\frac{8}{26}\cdot 2+\frac{5}{26}\cdot 3+\frac{2}{26}\cdot 4+\frac{1}{26}\cdot 5 \approx 2.54$$

- Claim: ordering items by non-increasing access-probability minimizes expected access cost, i.e. best *static* ordering
  - *static ordering:* order of items does not change
- Proof Idea: for any other ordering, exchanging two items that are out-of-order according to access probabilities makes total cost increase

# Dynamic Ordering

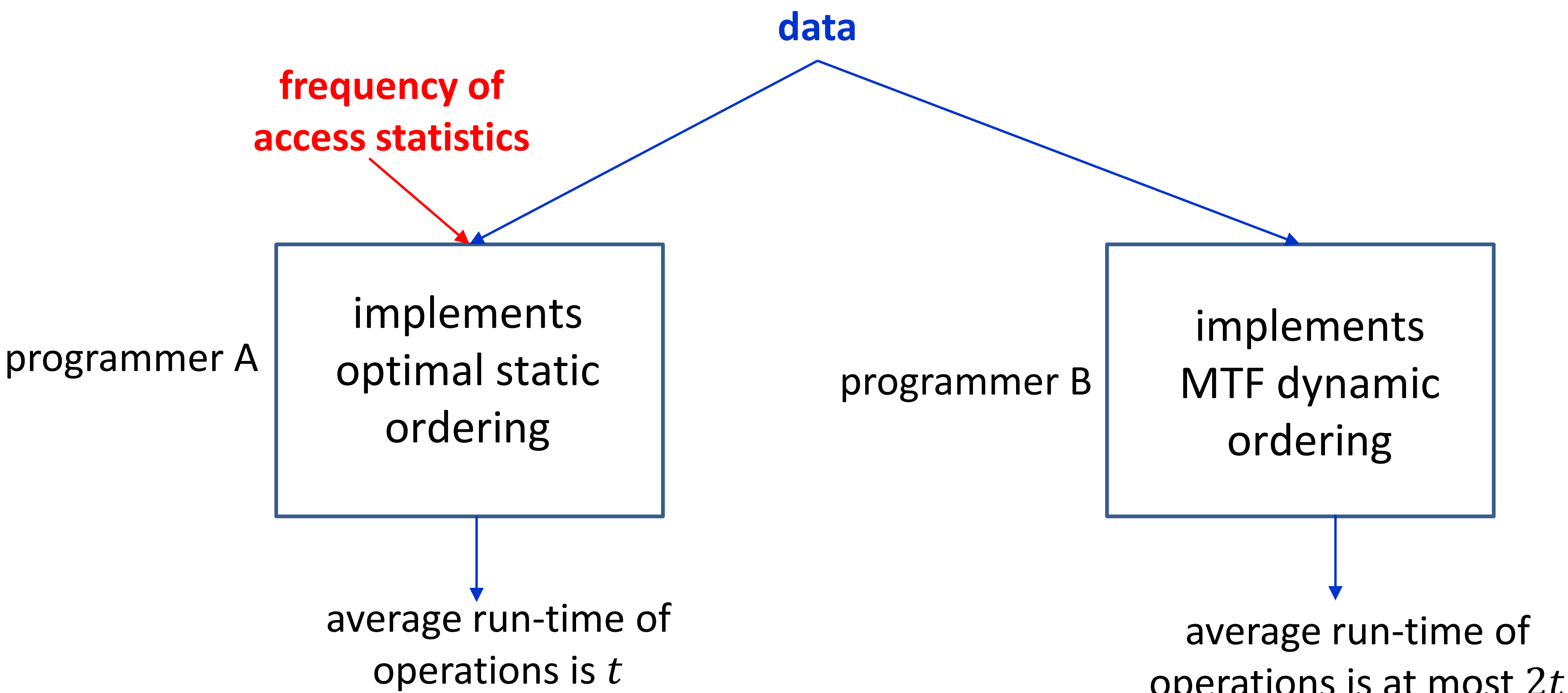
- *Dynamic ordering*: order of items is allowed to change
- What if we do *not know the access probabilities* ahead of time?
- Rule of thumb: recently accessed item is likely to be accessed soon again
- Move-To-Front heuristic (MTF): after search, move the accessed item to the front
  - additionally, in list: always insert at the front




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

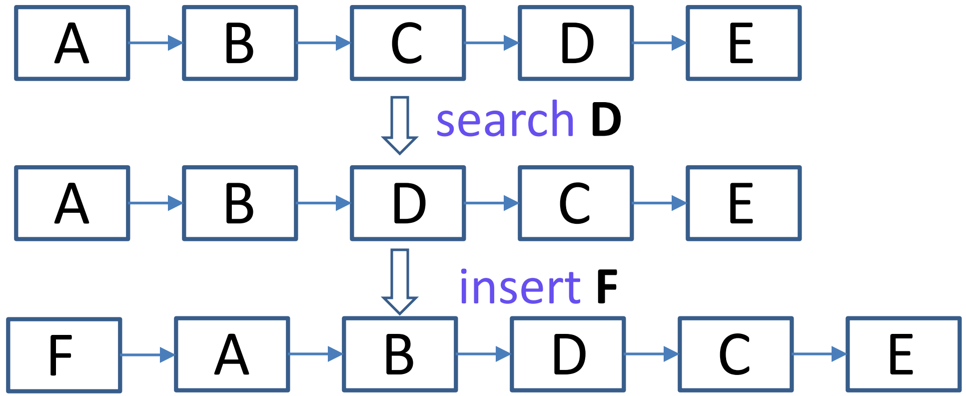
# Dynamic Ordering: MTF

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



# Dynamic Ordering: Other Heuristics

- **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

- **Frequency-count heuristic**: Keep counters how often items were accessed, and sort in non-decreasing order
  - works well in practice, but requires extra space

# Summary of Biased Search Requests

- We are unlikely to know the access-probabilities of items, so optimal static order is mostly of theoretical interest
- For any dynamic reordering heuristic, some sequence will defeat it
  - have $\Theta(n)$ access cost for each item
- MTF and Frequency-Count work well in practice
- For MTF can prove theoretical guarantees
- There is very little overhead for MTF and other strategies, they should be applied whenever unordered arrays or lists are used