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 2025

Outline

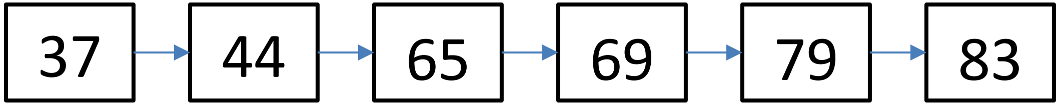
- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Biased Search Requests
    - optimal static ordering
    - dynamic ordering: MTF

Outline

- Dictionaries with Lists Revisited
    - Dictionary ADT
        - implementations so far
    - Skip Lists
    - Biased Search Requests
        - optimal static ordering
        - dynamic ordering: MTF

Dictionary ADT: Implementations thus far

- A *dictionary* is a collection of *key-value pairs* (KVPs)
    - *search*, *insert*, and *delete*
- Realizations we have seen so far
    - **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?

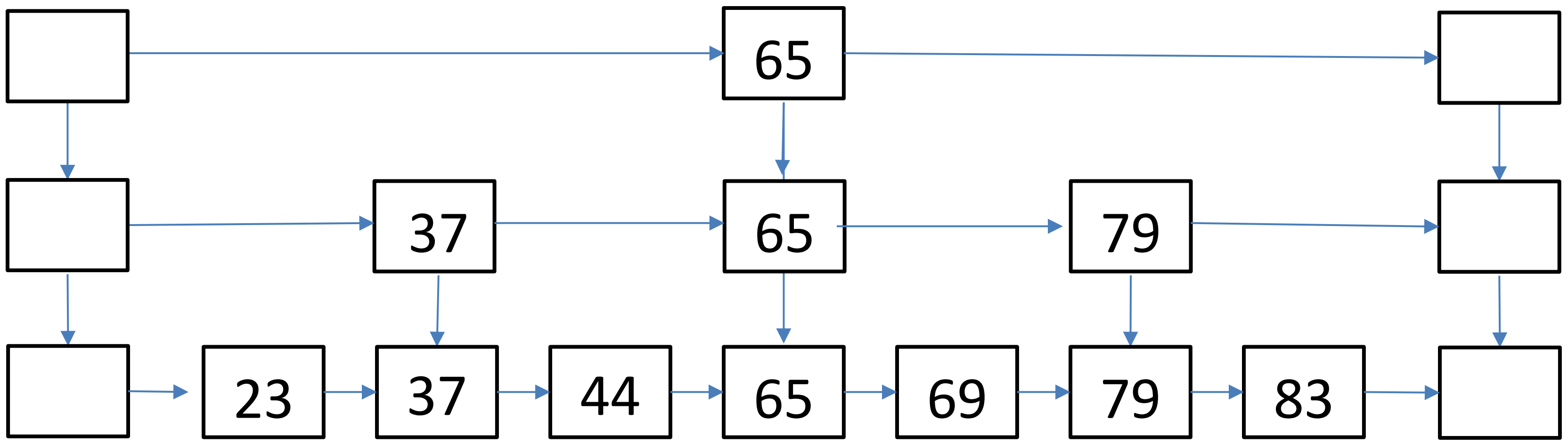


Outline

- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Re-ordering items
    - optimal static ordering
    - dynamic ordering: MTF

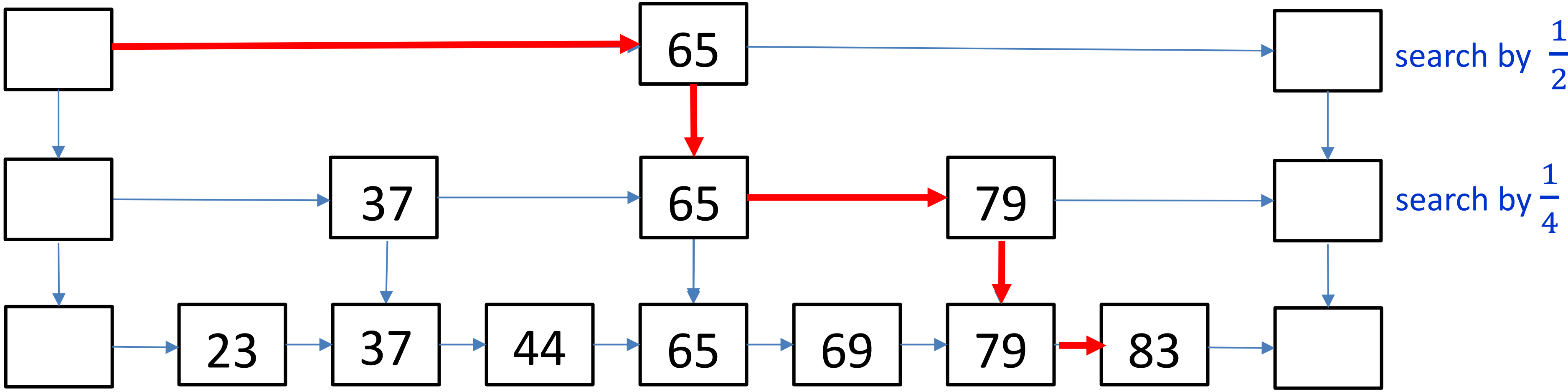
Skip Lists: Motivation

- Build a *hierarchy* of linked lists to imitate binary search in ordered linked list
    - 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



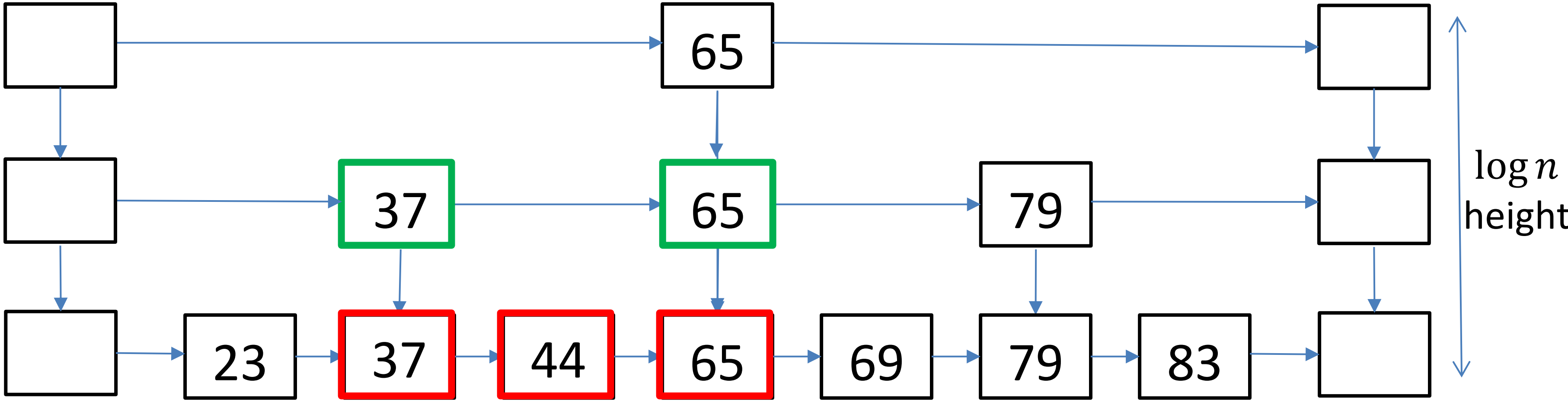
Skip Lists: Motivation

- 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

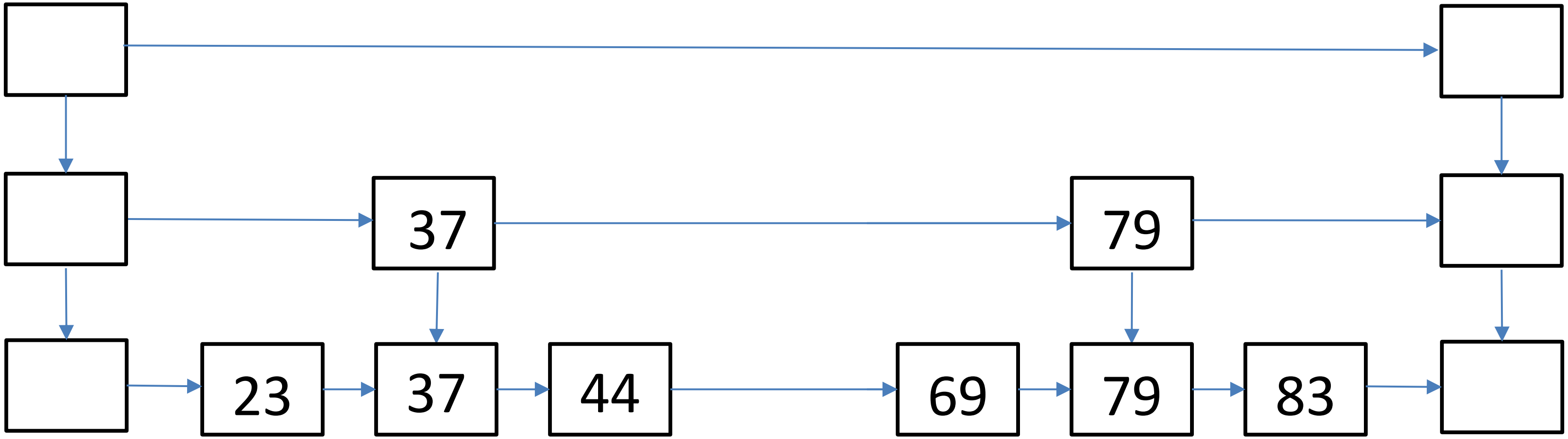
- Hierarchy of linked lists
  - each list has $1/2$ of items from the list below
  - total number of linked lists (height) is $\log n$
  - total number of nodes $\leq 2n$



- 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

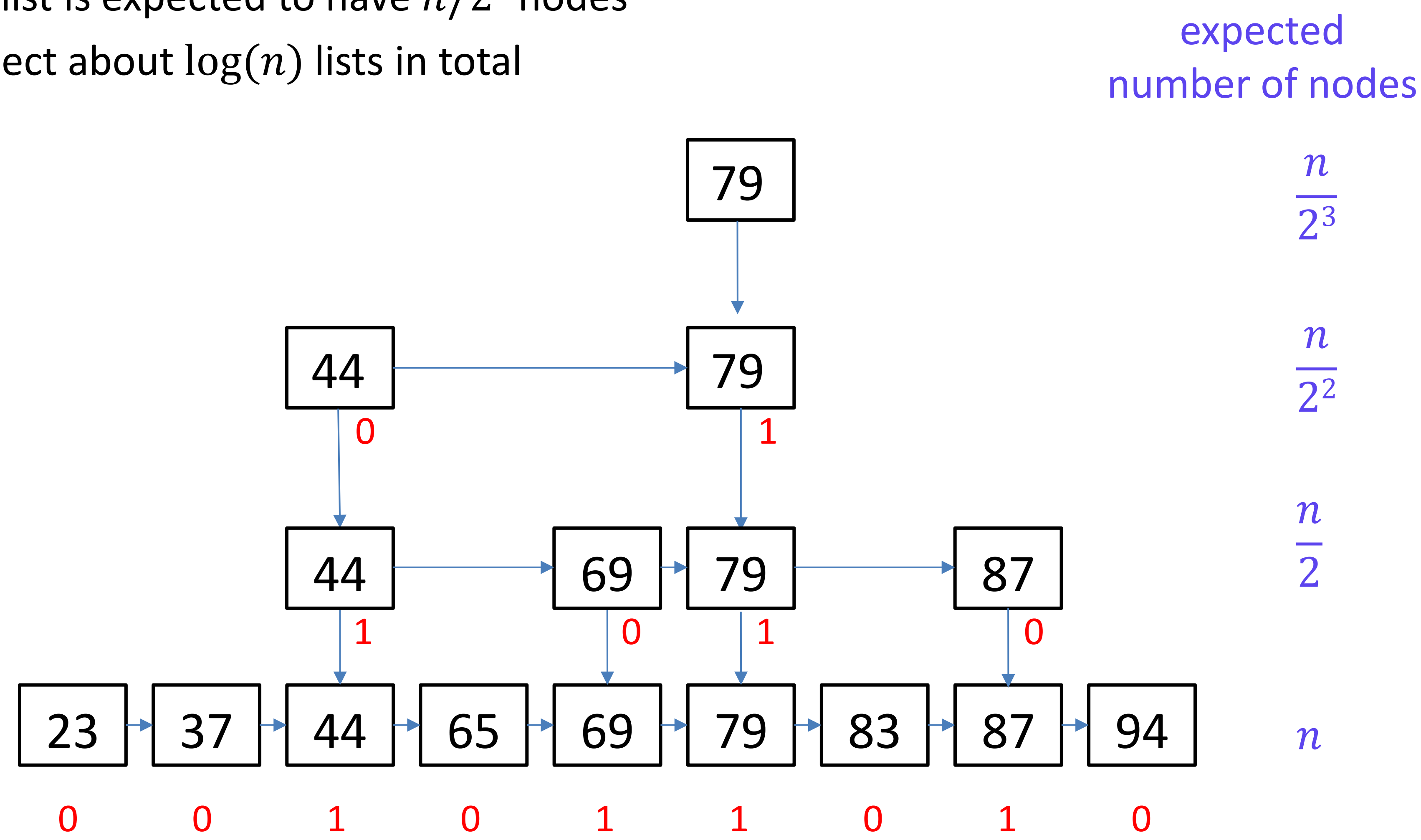
- Deleted 65, no longer every second item is in the list above



- **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

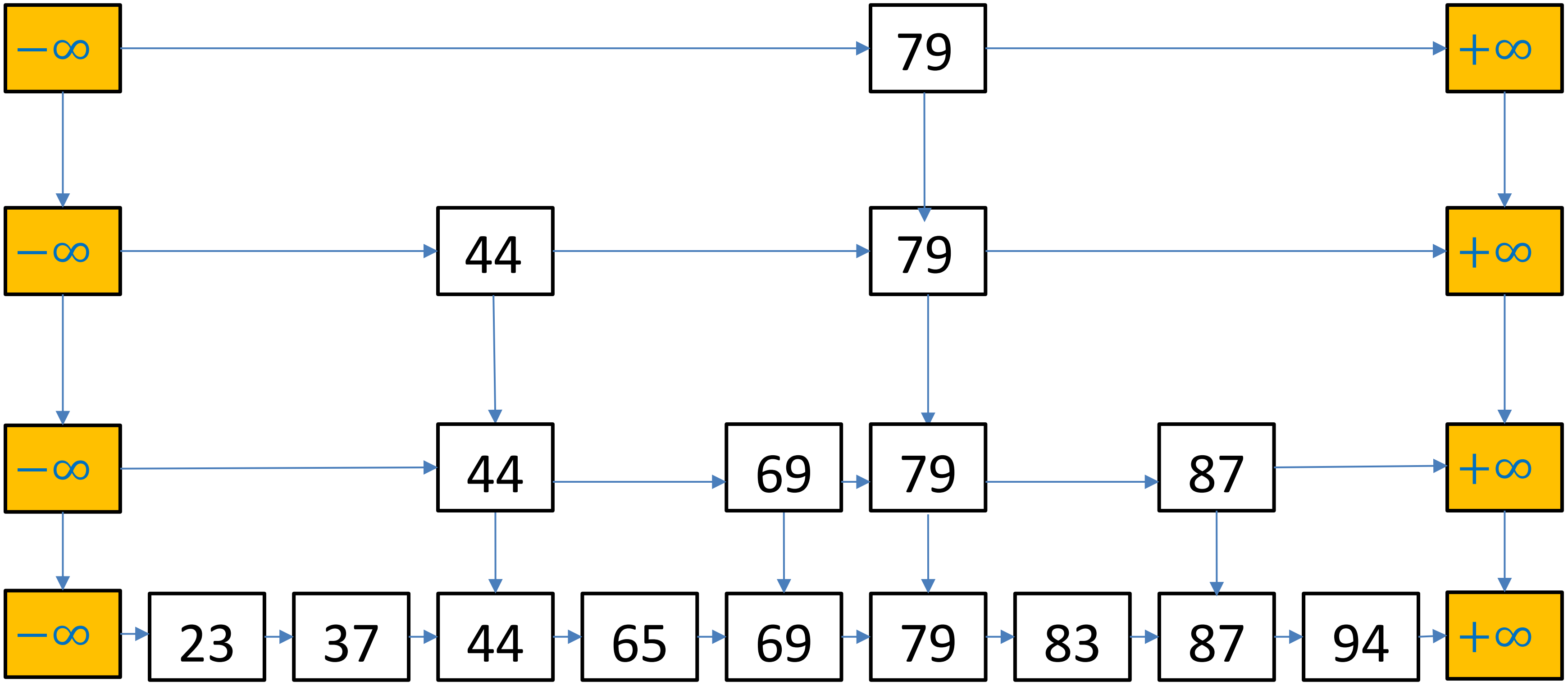
Skip Lists: Motivation

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



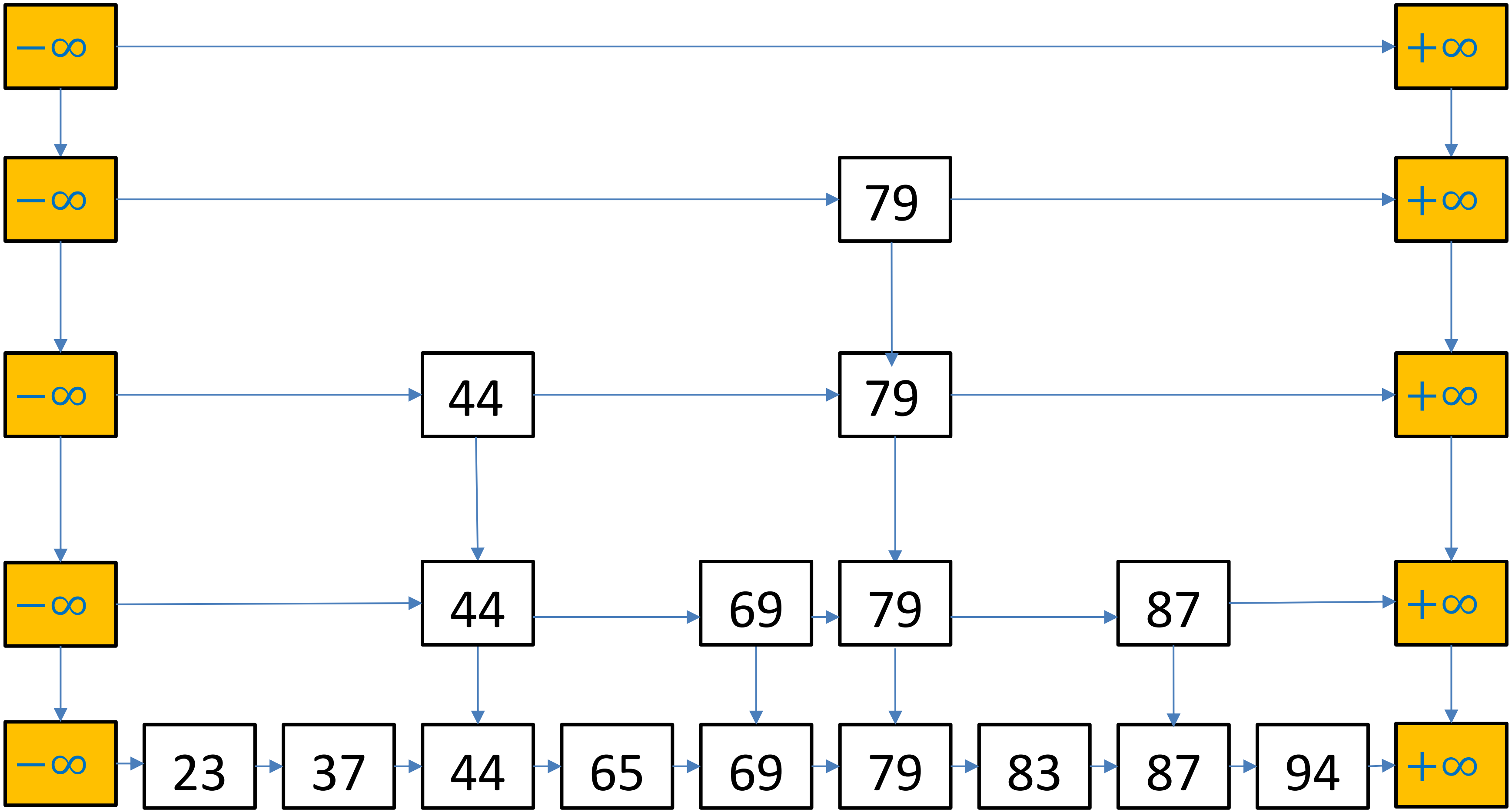
Skip Lists: Motivation

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



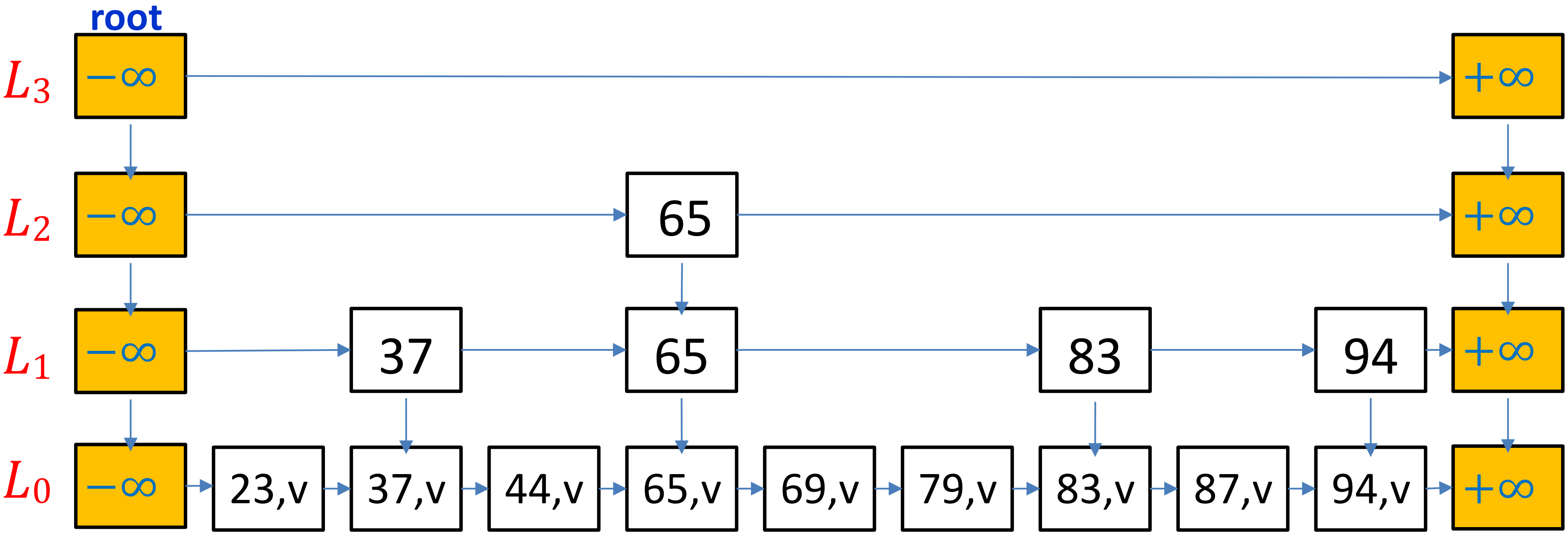
Skip Lists: Motivation

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



Skip Lists [Pugh'1989]

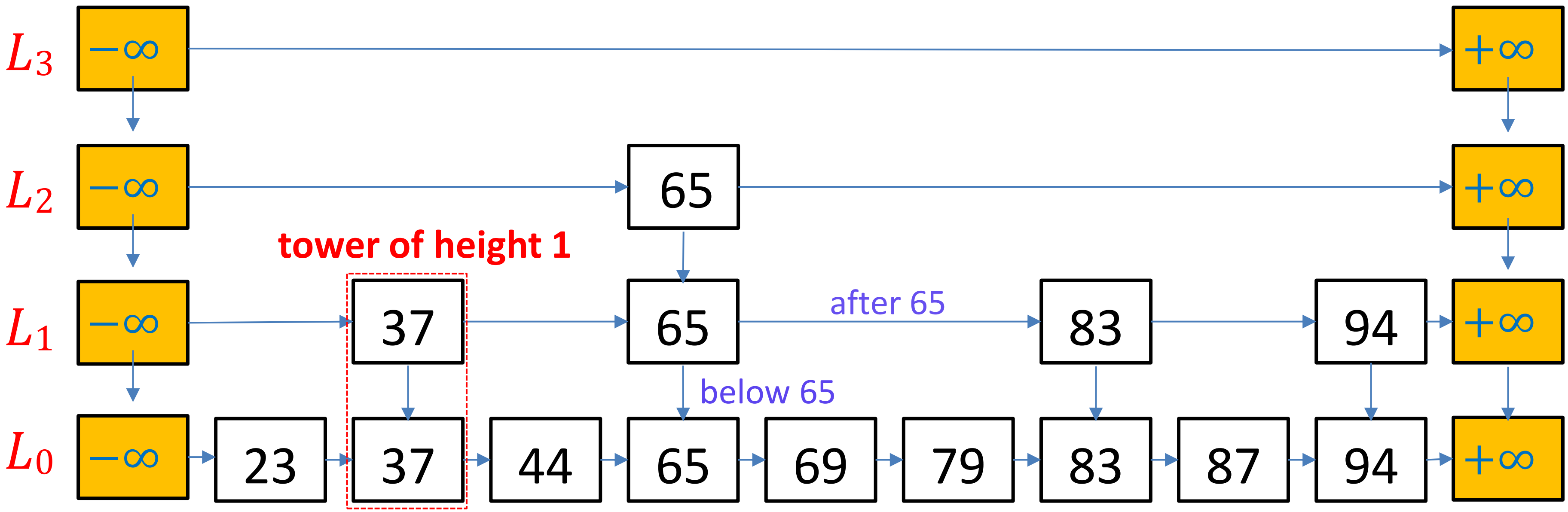
- A hierarchy *L* of ordered linked lists (*levels*) $L_0, L_1, \dots, L_h$
  - L_0 contains the KVPs of some S in non-decreasing order
  - other lists store only keys
  - each L_i contains special keys (sentinels) $-\infty$ and $+\infty$
  - each list is a subsequence of previous one, i.e. $L_0 \supseteq L_1 \supseteq \dots \supseteq L_h$
  - L_h contains only sentinels, the left sentinel is the root



- node is entry in one list vs. KVP is one non-sentinel entry in L_0
- n is number of KVP, here, $n = 9$ (number of nodes is 22)

Skip Lists

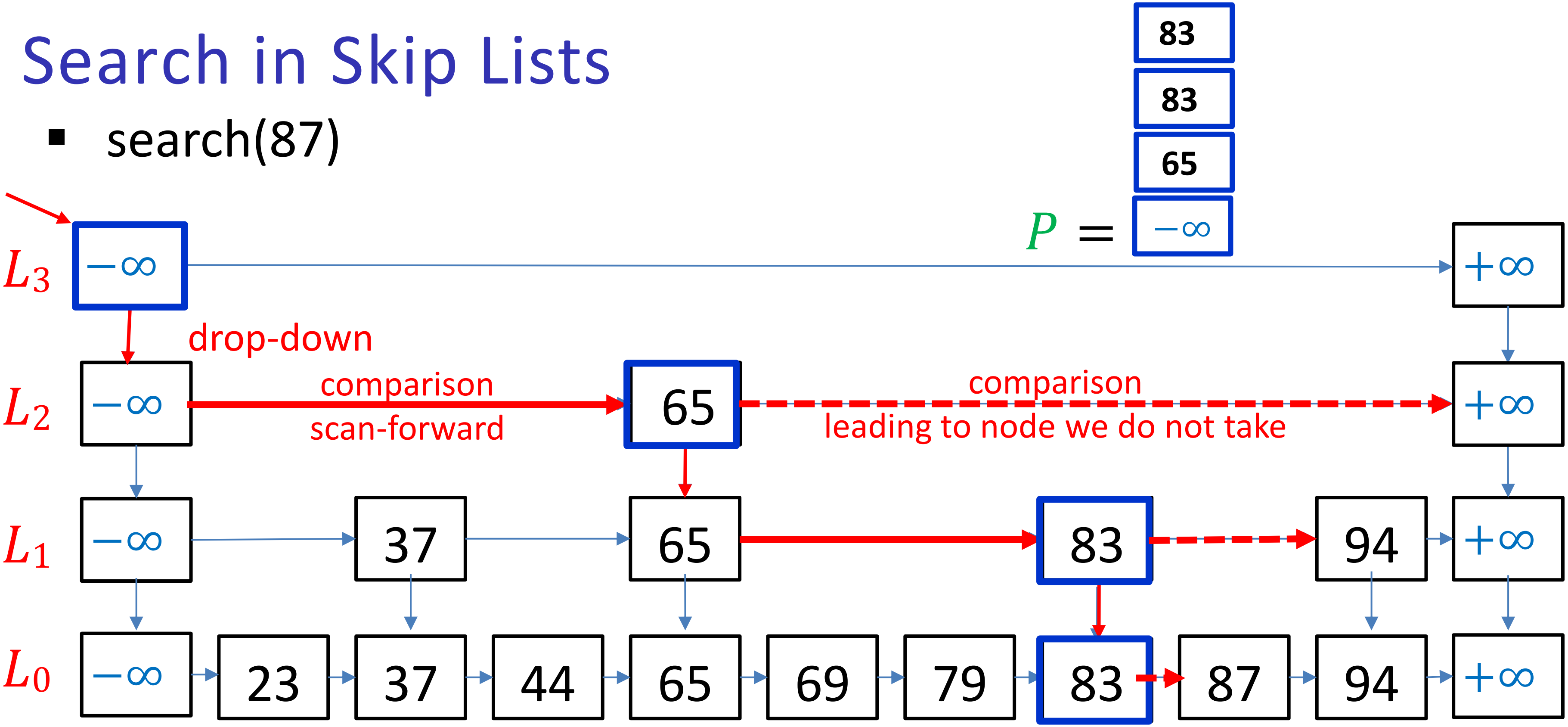
- Show only keys from now on



- Each key k belongs to a *tower* of nodes
  - height of tower for k : largest i s.t. $k \in L_i$
- Height of the skip list is the maximum height of any tower
  - which is the same as largest h for which L_h exists
  - height is 3 in this example
- Each node p has references to after(p) and below(p)

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 L_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 $p.below \neq NULL$ do // keep dropping down until reach $L_0$
            $p \leftarrow p.$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 $L_0$
        if $q.after.key = k$ return $q.after$
        else return 'not found, but would be after $q$'
```

Insert in Skip Lists

- No choice as to where put the tower of k
- The only choice is how talls hould we make the tower of k

L_3 ← if in L_2, then insert new item with probability ½

L_2 ← if in L_1, then insert new item with probability ½

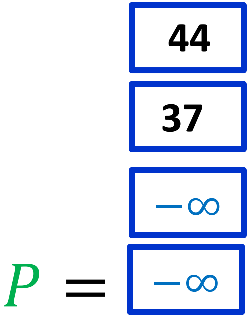
L_1 ← insert new item with probability ½

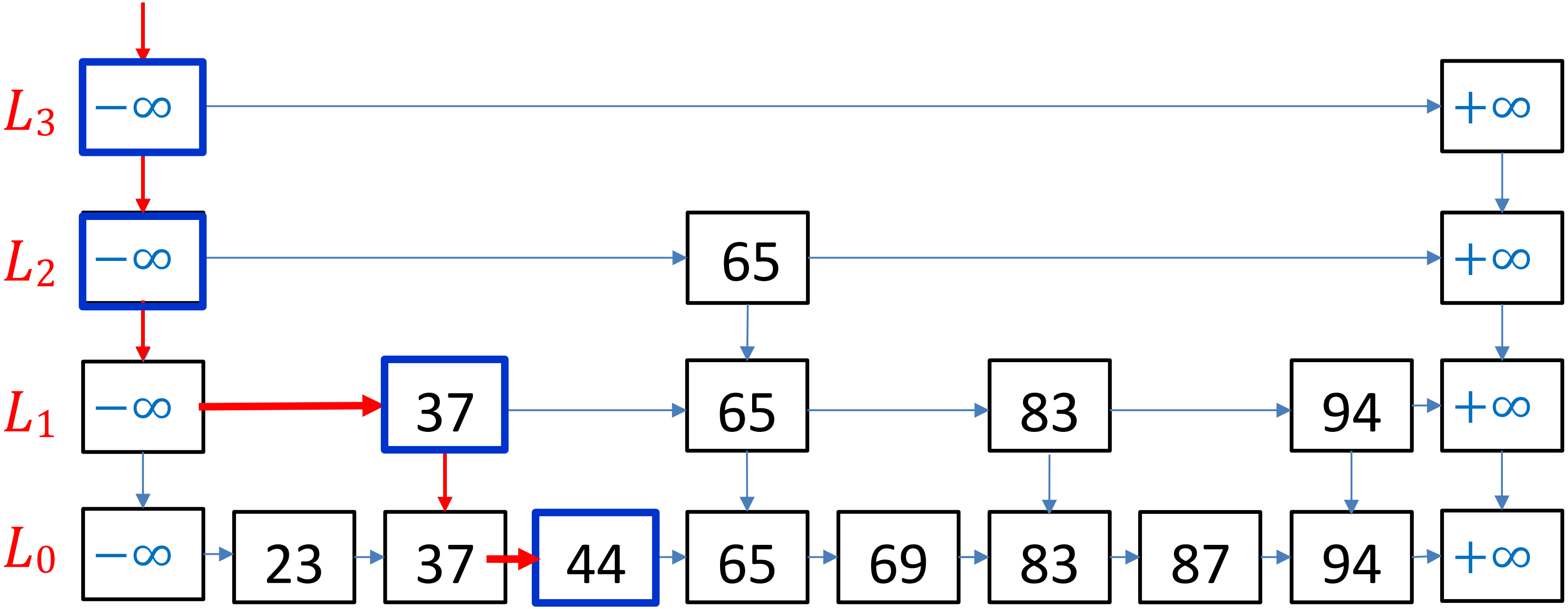
L_0 ← insert new item

- Keep “tossing a coin” until T appears
- Insert into L_0 and as many other L_i as there are heads
- Examples
    - H, H, T (insert into L_0, L_1, L_2) ⇒ will say $i = 2$
    - H, T (insert into L_0, L_1) ⇒ will say $i = 1$
    - T (insert into L_0) ⇒ will say $i = 0$

Insert in Skip Lists: Example 1

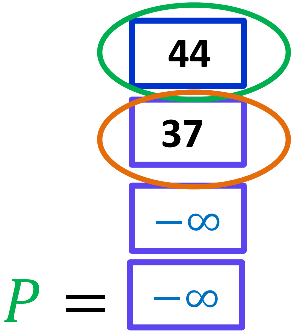
- *skipList::insert*$(52, v)$
- coin tosses: $H, T \Rightarrow i = 1$
- *getPredecessors*(52)

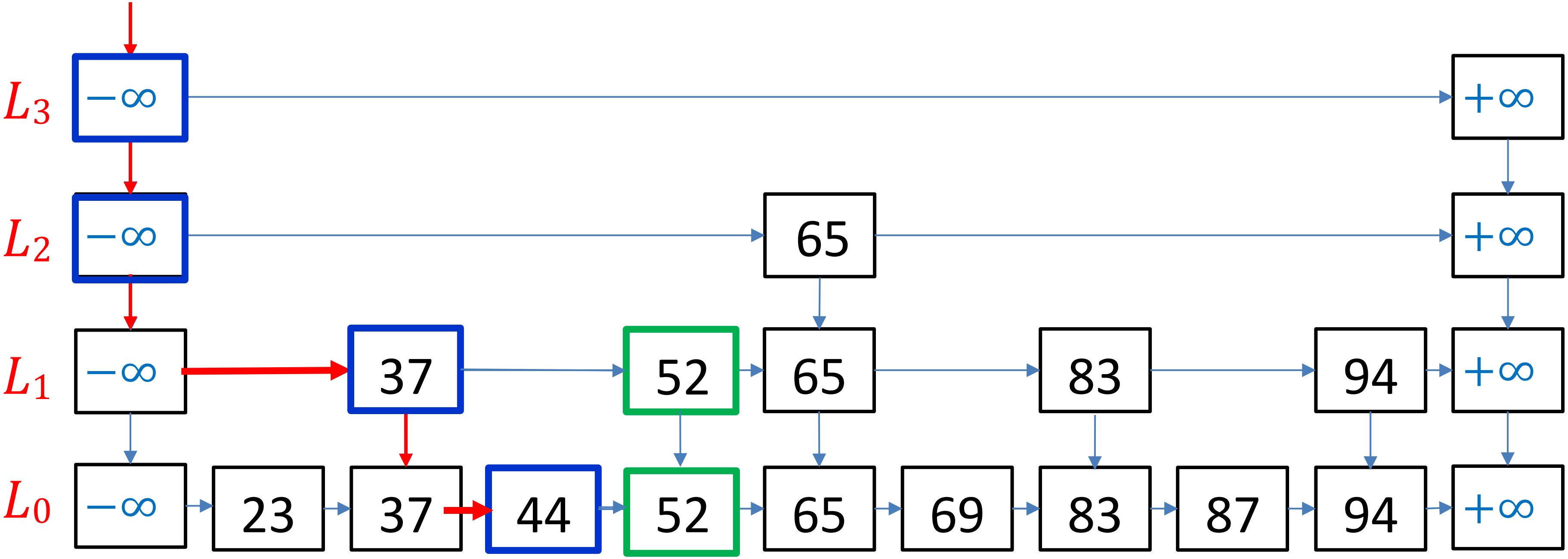




Insert in Skip Lists: Example 1

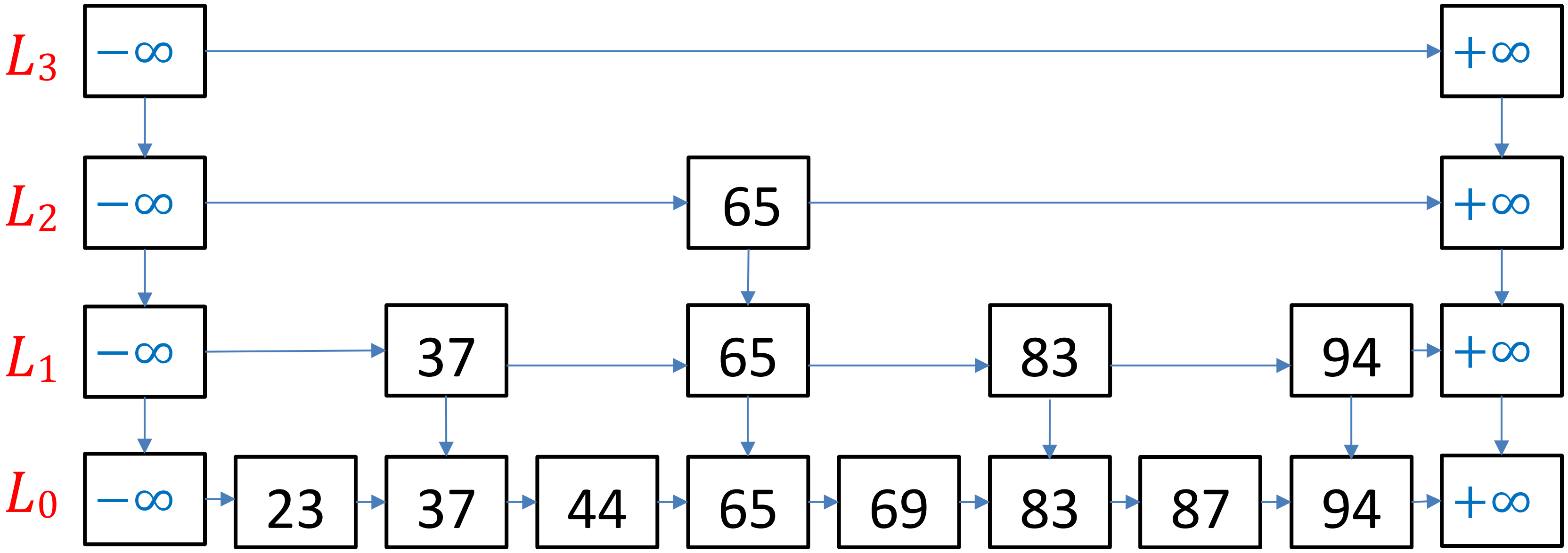
- *skipList::insert*$(52, v)$
- coin tosses: $H, T \Rightarrow i = 1$
- *getPredecessors*(52)
- now insert into L_0 and L_1





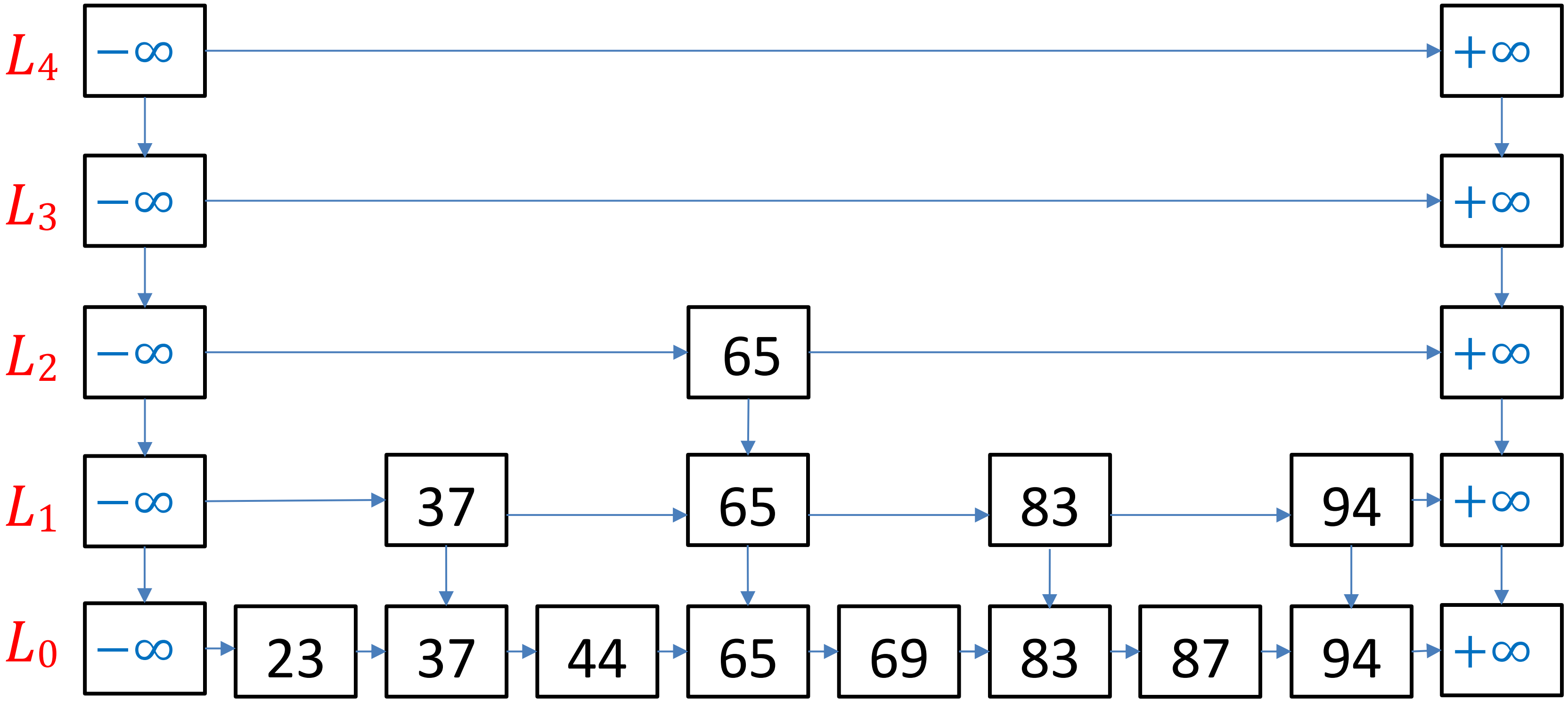
Insert in Skip Lists: Example 2

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



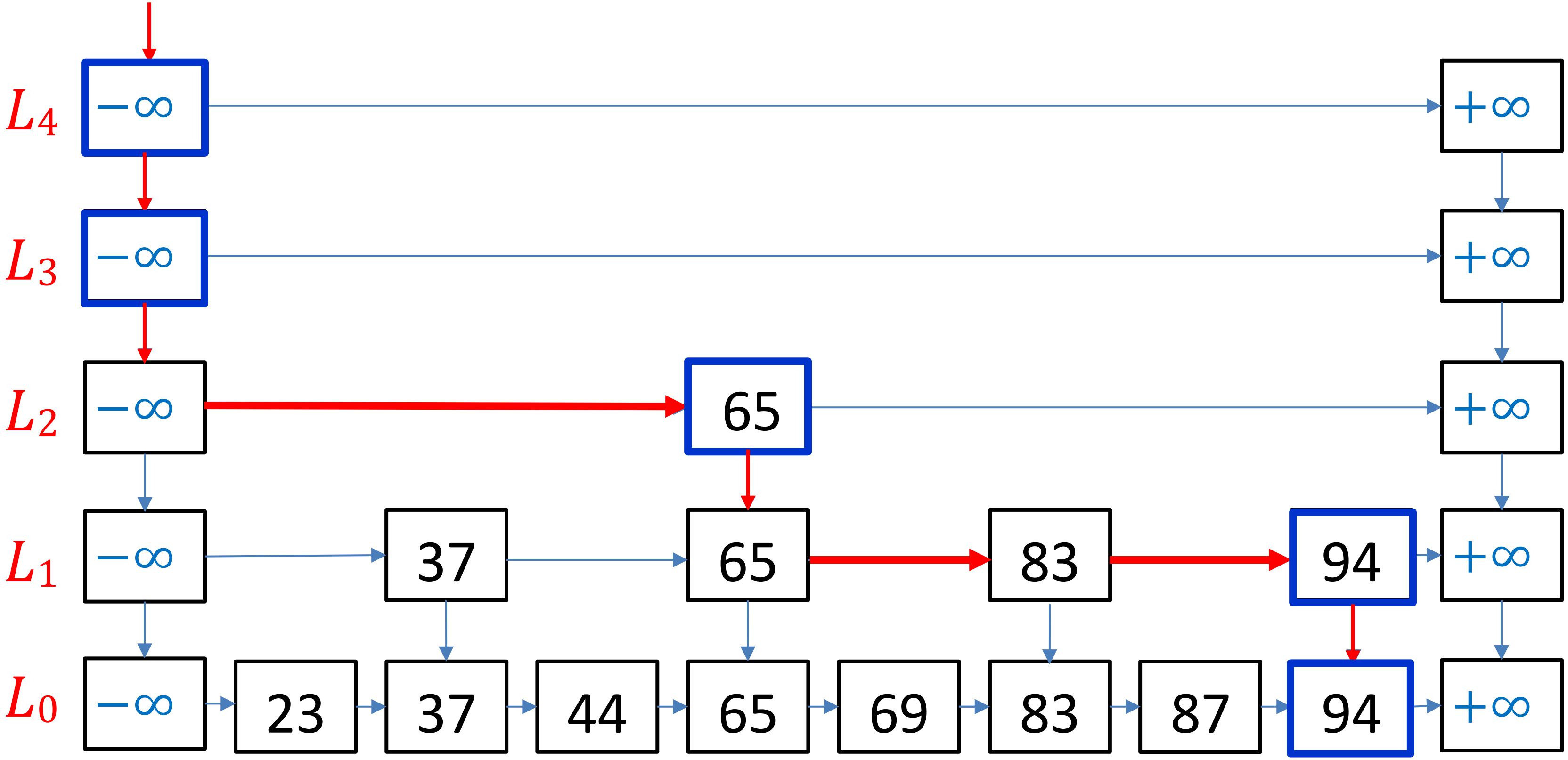
Insert in Skip Lists: Example 2

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



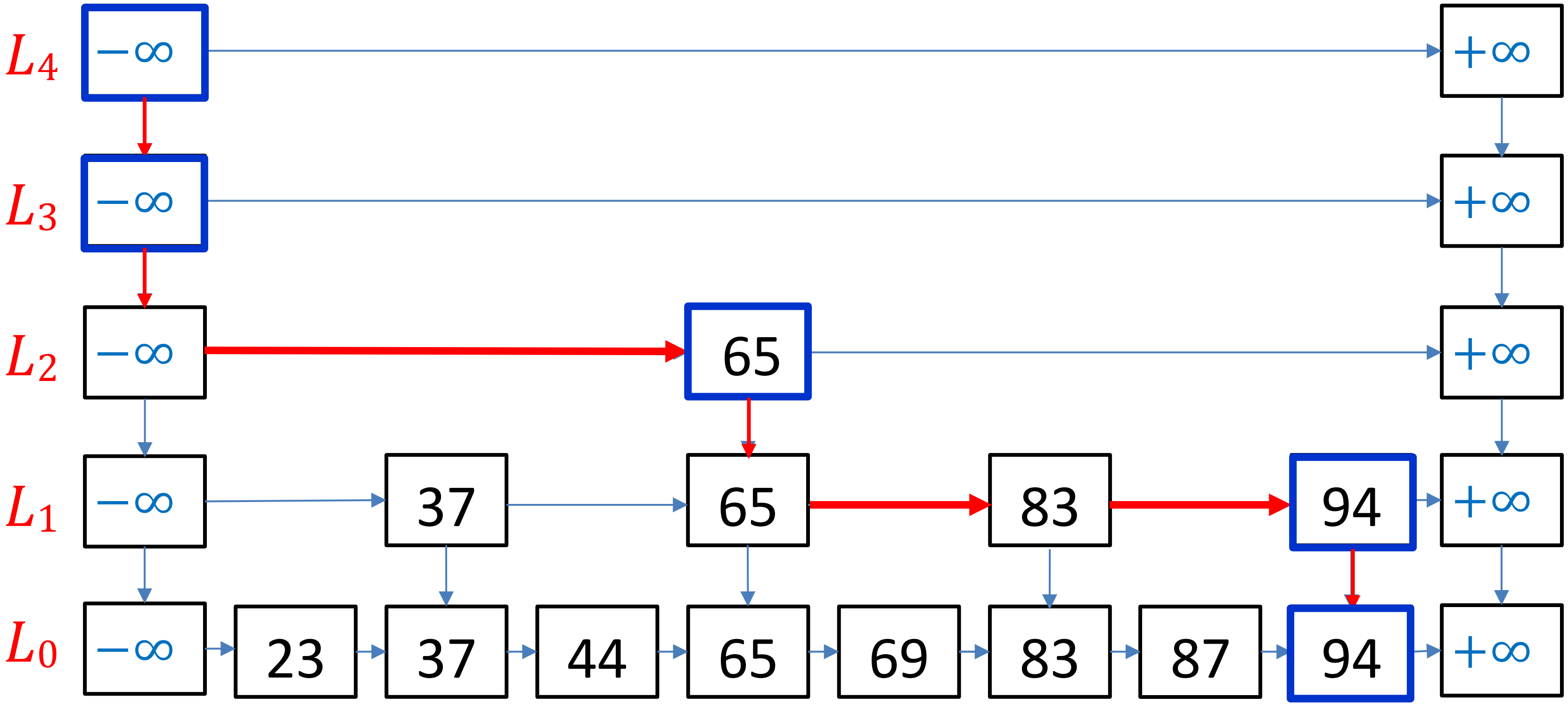
Insert in Skip Lists: Example 2

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



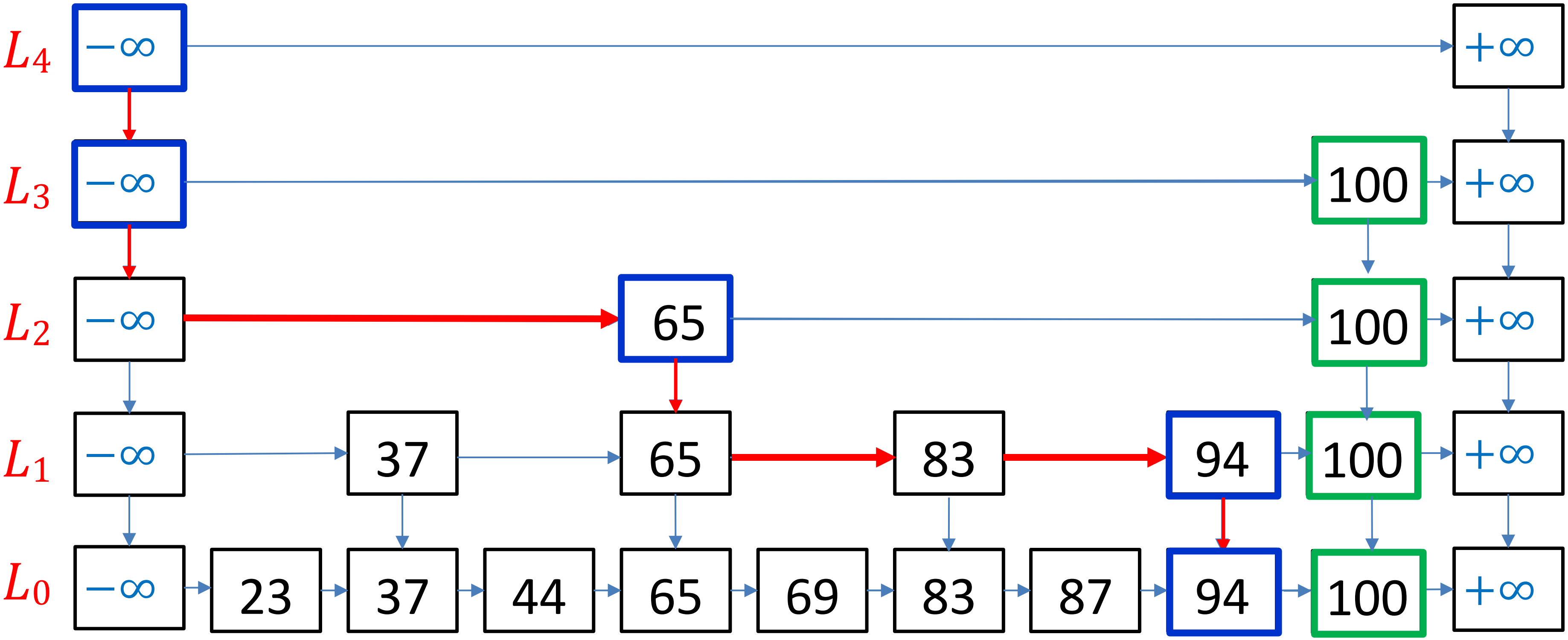
Insert in Skip Lists: Example 2

- *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

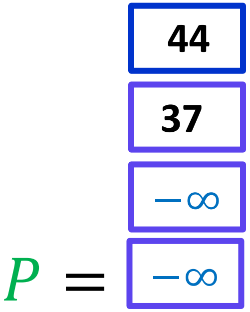


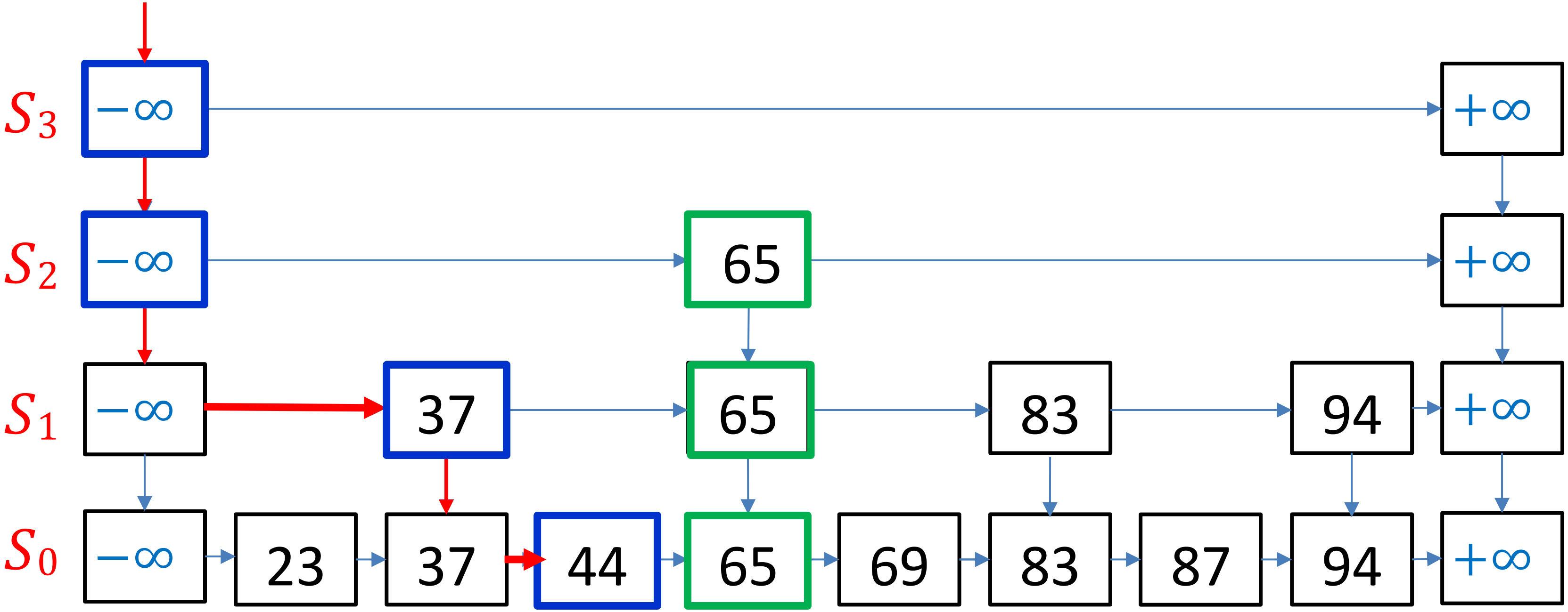
Insert in Skip Lists

```
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
        create new sentinel-only list; link it in below topmost level
        $h{+}{+}$
    $P \leftarrow$ getPredecessors$(k)$
    $p \leftarrow P.pop()$
    $zBellow \leftarrow$ new node with $(k, v)$ inserted after $p$        // insert $(k, v)$ in $L_0$
    while $i > 0$        // insert $k$ in $L_1$ $L_2$,..., $L_i$
        $p \leftarrow P.pop()$
        $z \leftarrow$ new node with $k$ added after $p$
        $z.below \leftarrow zBellow$
        $zBellow \leftarrow z$
        $i \leftarrow i - 1$
```

Example: Delete in Skip Lists

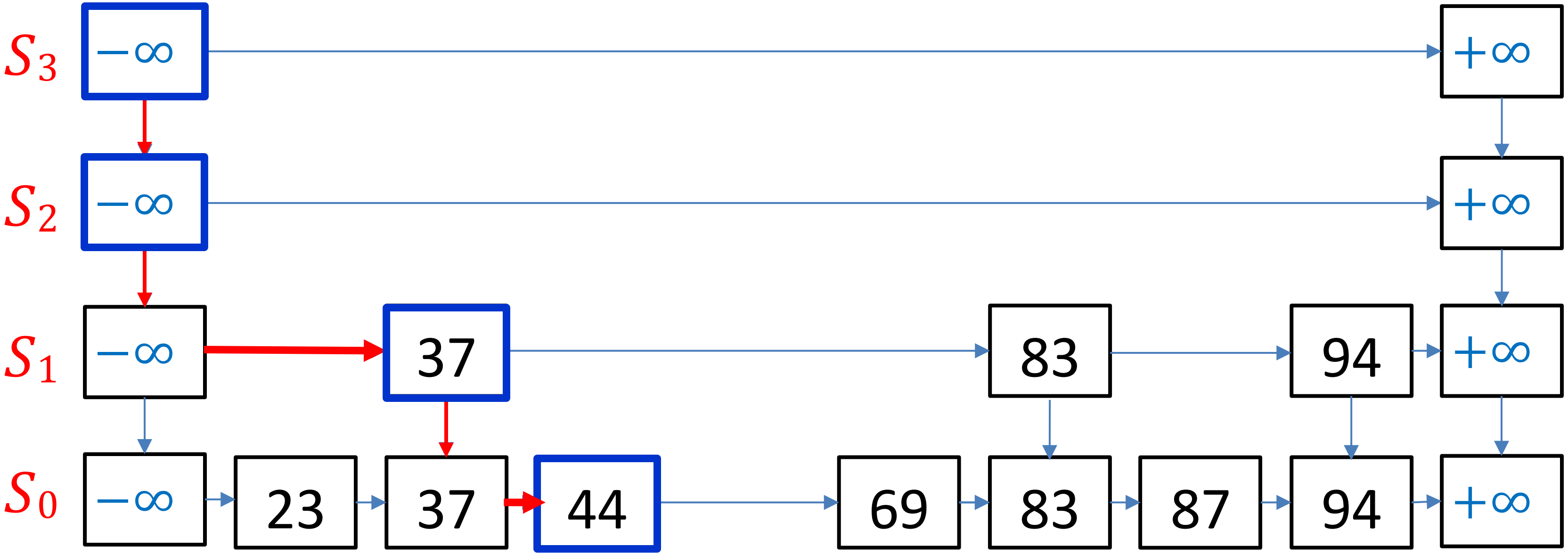
- *skipList::delete*(65)
  - first *getPredecessors*$(S, 65)$
  - then delete key 65 from all S_i
    - P has predecessor of each node to be deleted





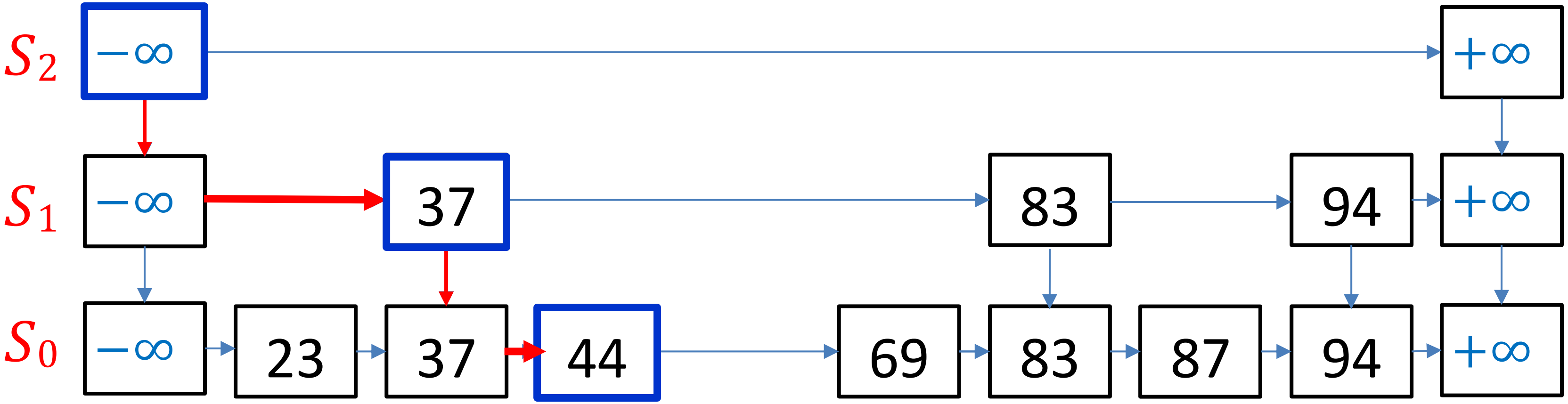
Example: Delete in Skip Lists

- *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

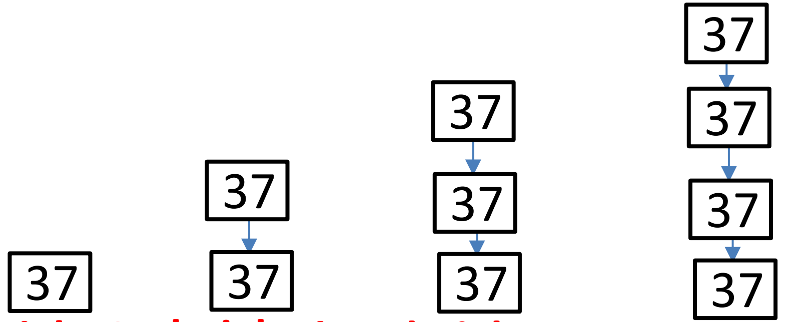
- *skipList::delete*(65)
  - first *getPredecessors*$(S, 65)$
  - then delete key 65 from all S_i
    - P has predecessor of each node to be deleted
  - height decrease: delete all unnecessary S_i, if any



Delete in Skip Lists

```
skipList::delete($k$)
        $P \leftarrow$ 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 $\infty$ 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



height 0 toss T

height 1 toss HT

height 2 toss HHT

height 3 toss $HHHT$

- Let X_k be the height of tower for key k

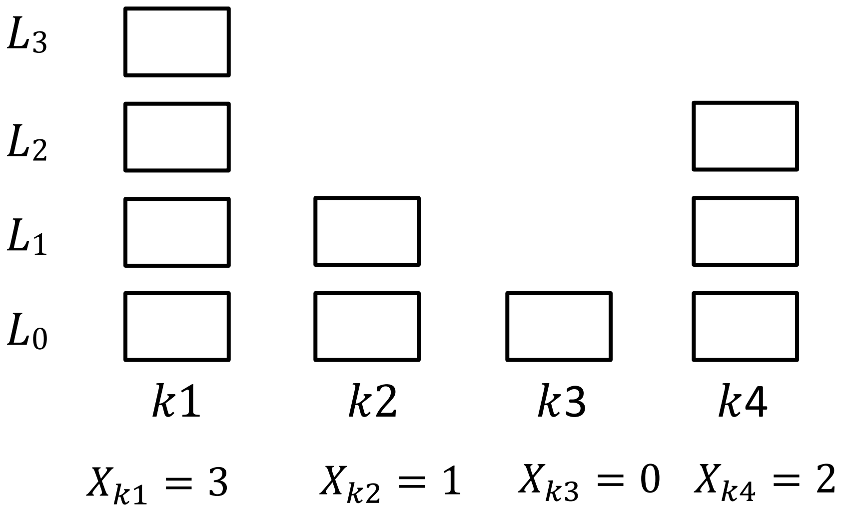
$P(X_k \geq 1) = \frac{1}{2}$ toss $H....$

$P(X_k \geq 2) = \frac{1}{2} \cdot \frac{1}{2}$ toss $HH....$

$P(X_k \geq 3) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2}$ toss $HHH....$

- 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
- 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



$|L_3| = 1$

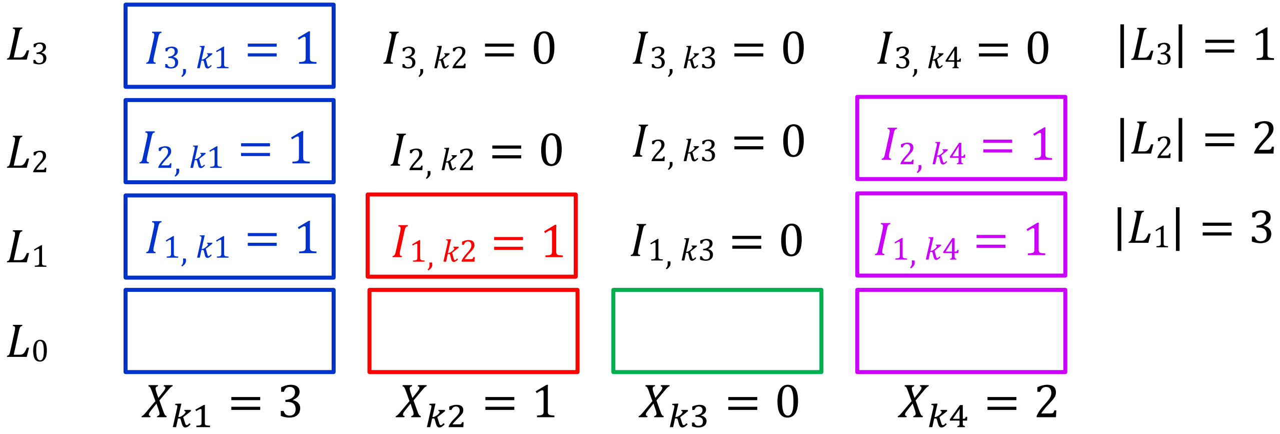
$|L_2| = 2$

$|L_1| = 3$

$|L_0| = 4$

- 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 L_i includes key k
- Let $|L_i|$ be the number of keys in list L_i
    - sentinels do not count towards the length
    - L_0 always contains all n keys

Skip List Analysis

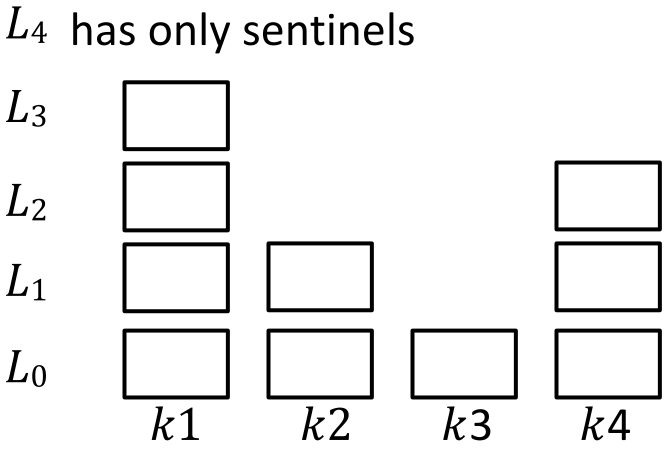


L_3	$I_{3,\,k1} = 1$	$I_{3,\,k2} = 0$	$I_{3,\,k3} = 0$	$I_{3,\,k4} = 0$	$\lvert L_3\rvert = 1$
L_2	$I_{2,\,k1} = 1$	$I_{2,\,k2} = 0$	$I_{2,\,k3} = 0$	$I_{2,\,k4} = 1$	$\lvert L_2\rvert = 2$
L_1	$I_{1,\,k1} = 1$	$I_{1,\,k2} = 1$	$I_{1,\,k3} = 0$	$I_{1,\,k4} = 1$	$\lvert L_1\rvert = 3$
L_0					
	$X_{k1} = 3$	$X_{k2} = 1$	$X_{k3} = 0$	$X_{k4} = 2$	

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

Skip List Analysis

- $|L_i|$ is number of keys in list L_i
  - $E[|L_i|] = \frac{n}{2^i}$
- Let $I_i = \begin{cases} 0 & \text{if } |L_i| = 0 \\ 1 & \text{if } |L_i| \geq 1 \end{cases}$



$I_4 = 0$

$I_3 = 1$

$I_2 = 1$

$I_1 = 1$

- $h = 1 + \sum_{i \geq 1} I_i$ (here +1 is for the sentinel-only level)
- Since $I_i \leq 1$ we have that $E[I_i] \leq 1$
- Since $I_i \leq |L_i|$ we have that $E[I_i] \leq E[|L_i|] = \frac{n}{2^i}$
- 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

L_4 has only sentinels $\quad I_4 = 0$

$L_3 \quad I_3 = 1$

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

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

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

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

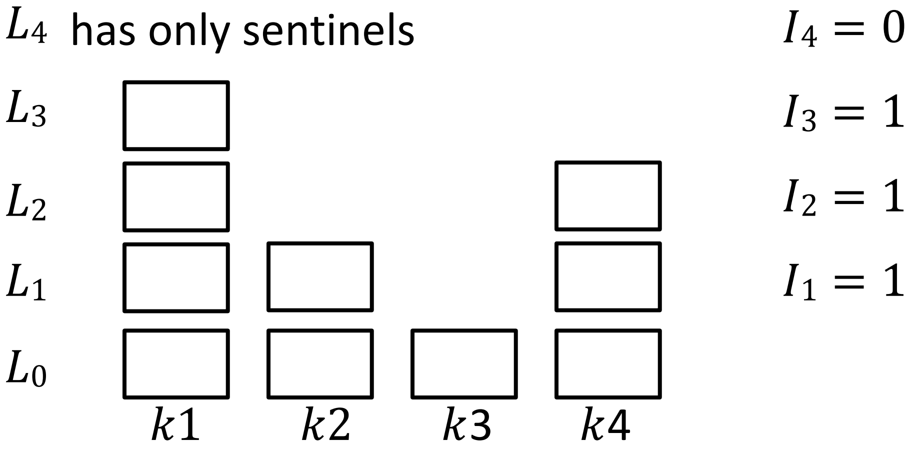
$$S = \sum_{i=0}^{\infty} \frac{1}{2^i} = \cancel{1} + \cancel{\frac{1}{2}} + \cancel{\frac{1}{2^2}} + \cancel{\frac{1}{2^3}} + \cdots$$

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

$$2S - S = 2$$

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

Skip List Analysis


L_4 has only sentinels

$I_4 = 0$

$I_3 = 1$

$I_2 = 1$

$I_1 = 1$

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

Skip List Analysis: Expected Space

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

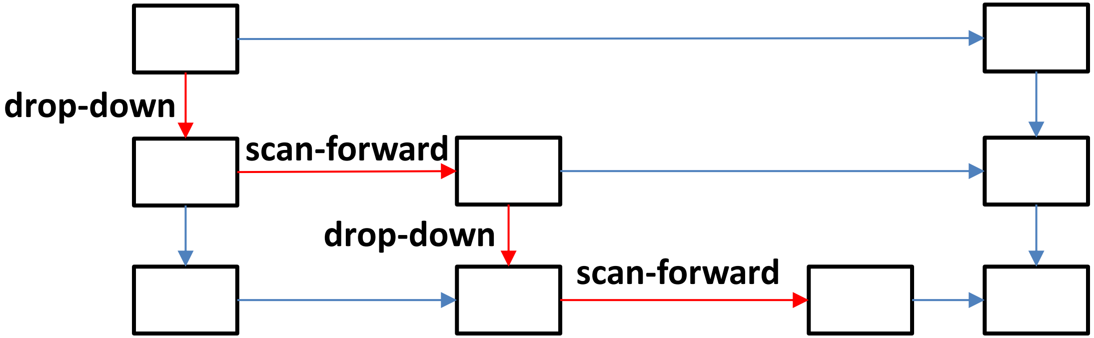
1. Space for nodes storing sentinels
    - there are $2h + 2$ sentinels, where h be the skip list height
    - $E[h] \leq 2 + \log n$
    - expected space for sentinels is at most

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

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

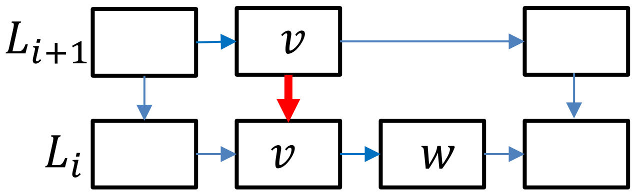
- Total expected space is $\Theta(n)$

Skip List Analysis: Expected Running Time

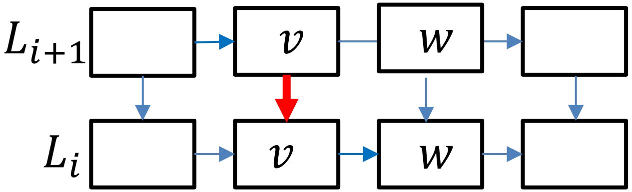


- *search*, *insert*, and *delete* are dominated by the runtime of *getPredecessors*
- So we analyze the expected time of *getPredecessors*
    - runtime is proportional to number of ‘drop-down’ and ‘scan-forward’
- We ‘drop-down’ h times, where h is skip list height
    - expected height h is $O(\log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

Expected Number of Scan-Forwards at Level i



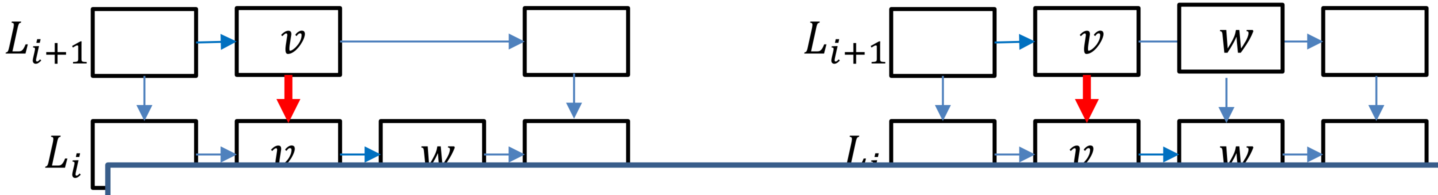
could scan forward from v to w in L_i



would scan from v to w in L_{i+1}, not in L_i

- Assume $i < h$
    - if $i = h$, then we are at the top sentinel-only list and do not scan forward at all
- Let v be leftmost key in L_i we visit during search
    - we v reached by dropping down from L_{i+1}
- Let w be the key right after v
    - height of tower of w is *at least* i, but could be more than i
- What is the probability of scanning from v to w (i.e. at least one scan) in L_i?
    - if scan-forward from v to w, then w is not in L_{i+1}
    - thus tower of w has height exactly i
        - and we already know that tower of w has height at least i

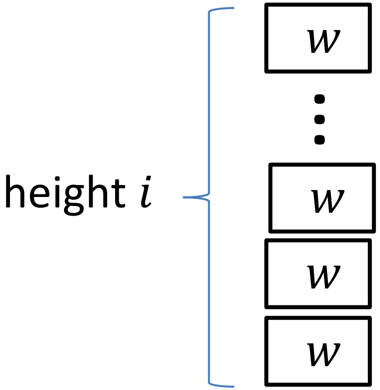
Expected Number of Scan-Forwards at Level i



coul

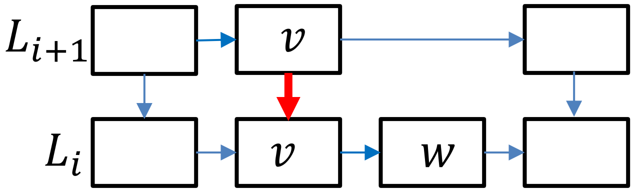
- Assum
- Let v b
- Let w
- What

$P(\text{tower of } w \text{ has height } i \mid \text{tower of } w \text{ has height at least } i) = ½$

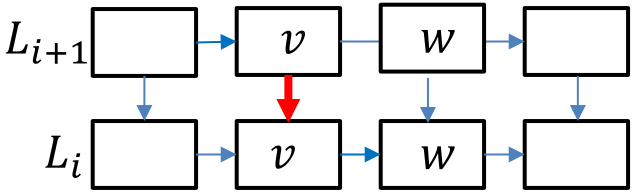


next toss is $T \Rightarrow$ tower stops growing

Expected Number of Scan-Forwards at Level i



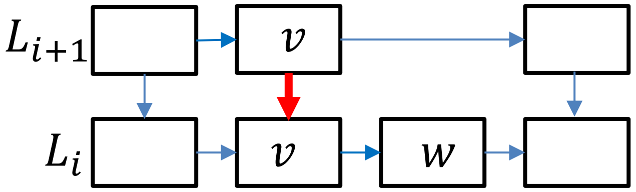
could scan forward from v to w in L_i



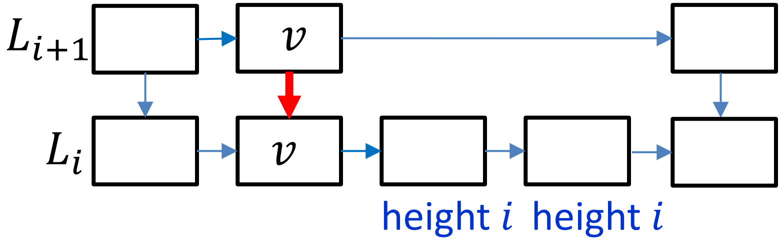
would scan from v to w in L_{i+1}, not in L_i

- Assume $i < h$
    - if $i = h$, then we are at the top sentinel-only list and do not scan forward at all
- Let v be leftmost key in L_i we visit during search
    - we v reached by dropping down from L_{i+1}
- Let w be the key right after v
    - height of tower of w is *at least* i, but could be more than i
- What is the probability of scanning from v to w (i.e. at least one scan) in L_i?
    - if scan-forward from v to w, then w is not in L_{i+1}
    - thus tower of w has height exactly i
        - and we already know that tower of w has height at least i
    - $P($tower of w has height $i\,|$ tower of w has height at least $i) = ½$
    - thus scan forward from v to w with probability at most ½
        - ‘at most’ because we could scan-down down if search $key < w$

Expected Number of Scan-Forwards at Level i



could scan forward from v to w in L_i



could scan forward twice from v in L_i

- What is the probability of scanning twice (i.e. at least 2 scans) from v in L_i?
    - scan forward at least twice from v with probability at most $(1/2)^2$
        - 'at most' because we could scan-down down
- In general, probability of scan-forward at least l times is at most $(1/2)^l$
    - i.e. $\mathrm{P}(\text{scans} \geq l) \leq (1/2)^l$

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

theorem in probability theory

Expected Number of Scan-Forward Operations

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

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

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

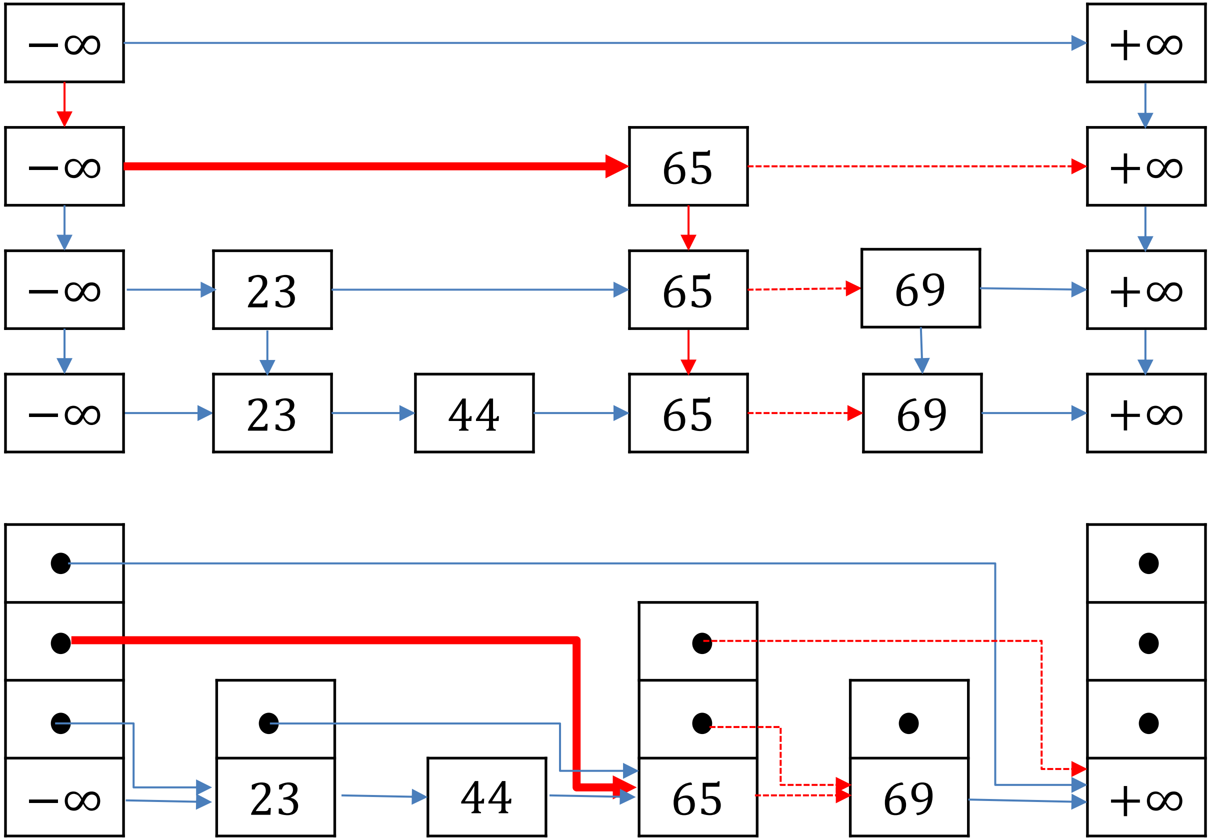
$$\leq \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i}$$

$$\leq \log n + 1$$

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

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 $O(\log n)$
- Lists make it easy to implement
    - easy to add more operations: *successor*, *merge*, …
- 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
        - expected space $<\ 2n$, less than for BST

Outline

- Dictionaries with Lists Revisited
  - Dictionary ADT
    - implementations so far
  - Skip Lists
  - Biased Search Requests
    - optimal static ordering
    - dynamic ordering: MTF

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
        - Intuition: 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

Outline

- Dictionaries with Lists Revisited
    - Dictionary ADT
        - implementations so far
    - Skip Lists
    - Biased Search Requests
        - optimal static ordering
        - dynamic ordering: MTF

Optimal Static Ordering

- Scenario: We know access distribution, and want to find the best list order

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

- Let the cost of search for key located at position i be i

$$T^{exp}(n) = \sum_{I \in I_n} T(I) \cdot \Pr(\text{randomly chosen instance } I)$$

$$= \sum_{i} i \cdot \Pr(\text{search for key at position } i)$$

$$= \sum_{i} i \cdot (\text{access probability for key at position } i)$$

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 $A \longrightarrow B \longrightarrow C \longrightarrow D \longrightarrow E$ 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 $D \longrightarrow B \longrightarrow E \longrightarrow A \longrightarrow C$ has expected cost

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

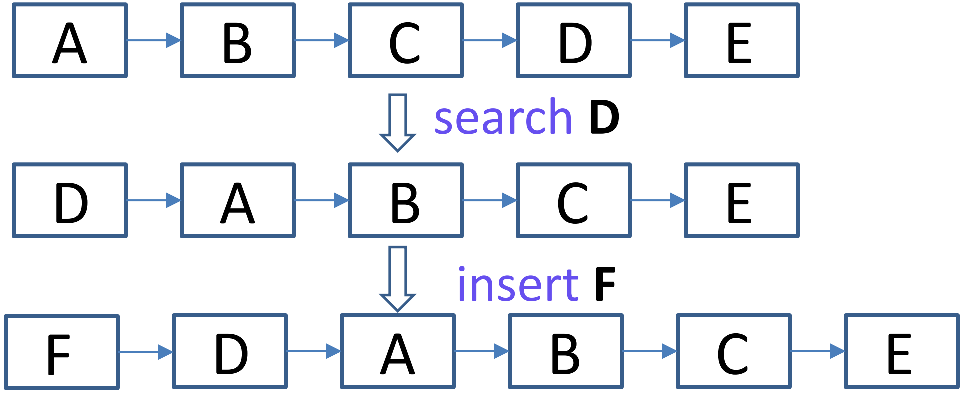
- Claim: ordering items by non-increasing access-probability minimizes expected access cost, i.e. best *static* ordering
    - *static ordering:* order of items does not change
- Proof Idea: for any other ordering, exchanging two items that are out-of-order according to access probabilities makes total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

Outline

- Dictionaries with Lists Revisited
    - Dictionary ADT
        - implementations so far
    - Skip Lists
    - Biased Search Requests
        - optimal static ordering
        - dynamic ordering: MTF

Dynamic Ordering

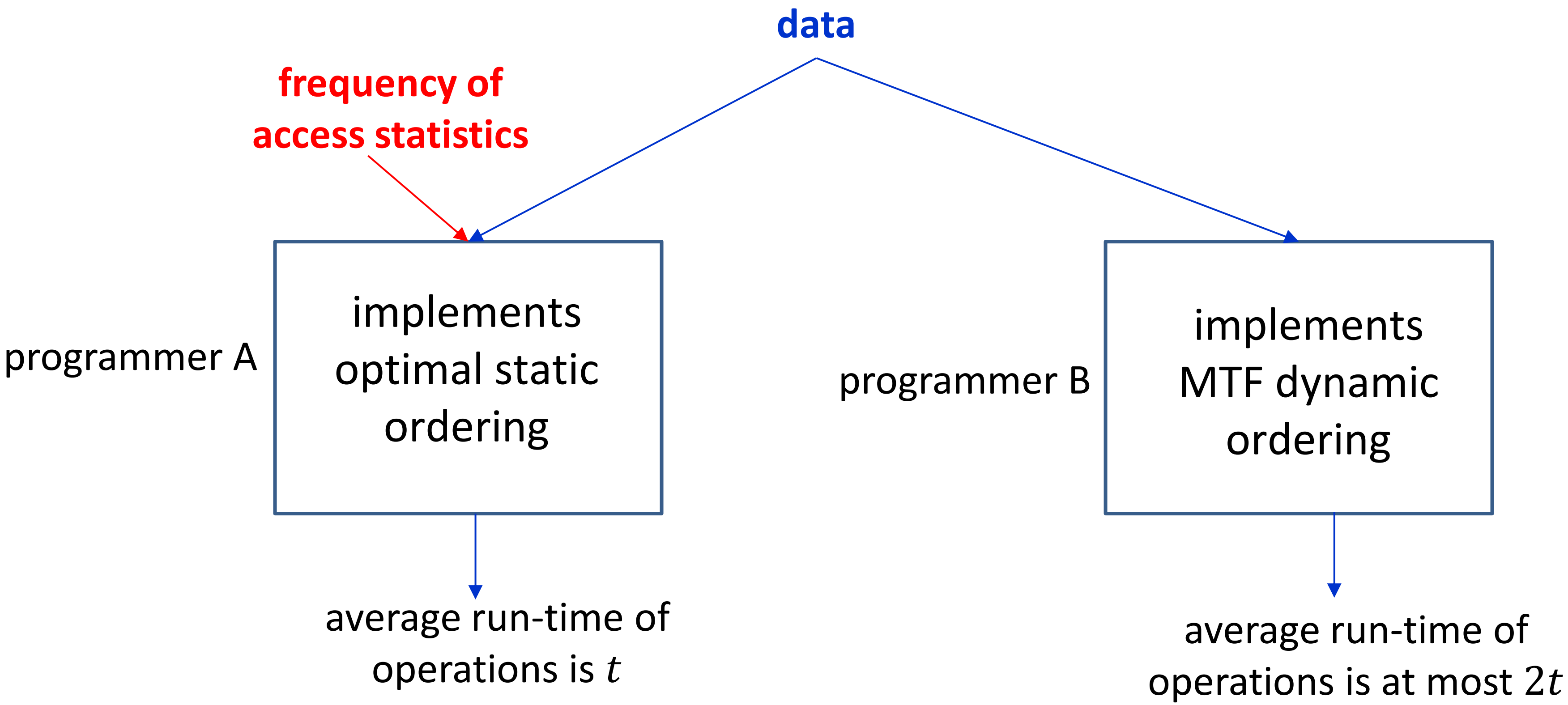
- Scenario: we do not know the access probabilities ahead of time
- *Idea:* modify the order dynamically, i.e. while we are accessing
- Rule of thumb: recently accessed item is likely to be accessed soon again
- Move-To-Front heuristic (MTF): after search, move the accessed item to the front
    - additionally, 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



data

frequency of access statistics

programmer A

implements optimal static ordering

average run-time of operations is t

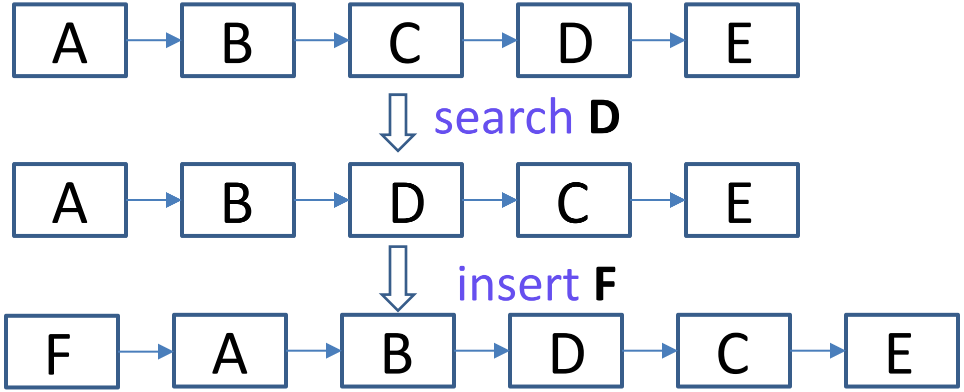
programmer B

implements MTF dynamic ordering

average run-time of operations is at most $2t$

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 auxiliary 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
    - hashing, text compression