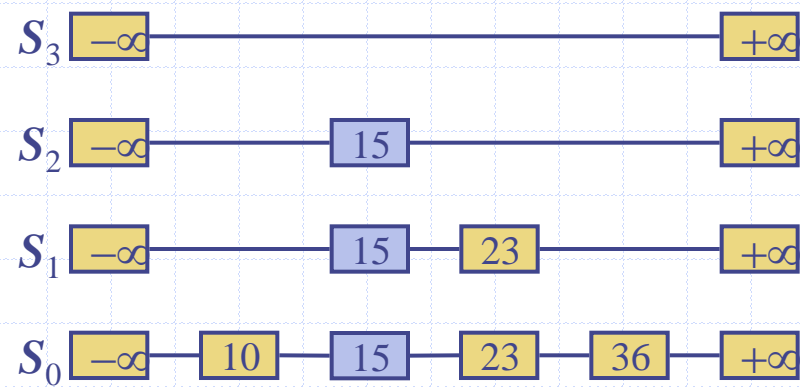


Skip Lists



Flight Databases

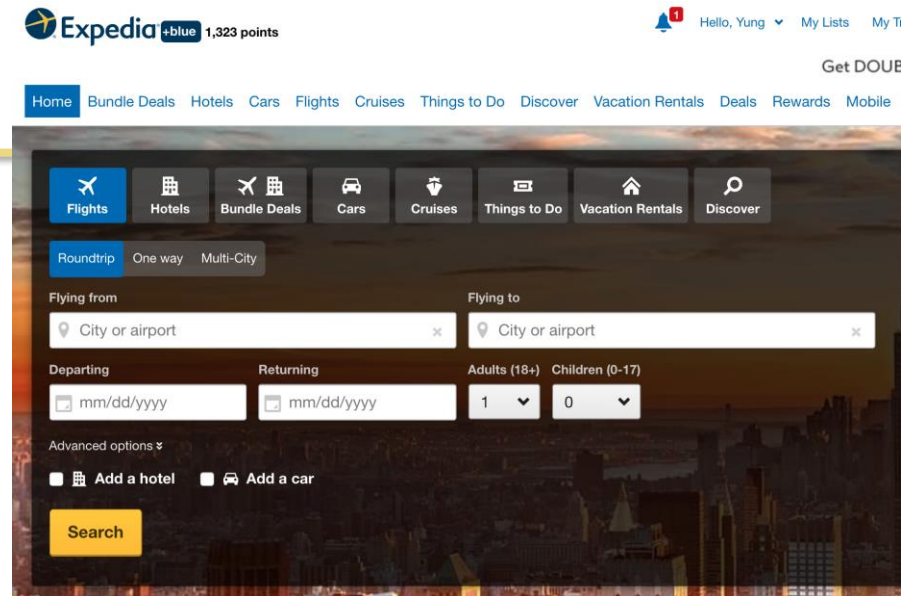
- ◆ Key: (origin, destination, date, time)
- ◆ Value: (flight number, available seats, first or economy, duration, fare, etc)

◆ User

- Happy about “closest” departure time, not simply searching the flights with exact match with the given key

◆ We may need a slightly different data structure from Map ADT

◆ That's an **ordered Map**



The screenshot shows the Expedia website's flight search interface. At the top, the Expedia logo is visible with a blue plus icon and '1,323 points'. To the right, there's a user greeting 'Hello, Yung' with a dropdown arrow, and links for 'My Lists' and 'My Ti'. Below this is a navigation bar with links: Home, Bundle Deals, Hotels, Cars, Flights, Cruises, Things to Do, Discover, Vacation Rentals, Deals, Rewards, and Mobile. The main search area has a dark background with a city skyline. It features a row of icons for Flights, Hotels, Bundle Deals, Cars, Cruises, Things to Do, Vacation Rentals, and Discover. Below these are tabs for Roundtrip, One way, and Multi-City. The search form includes fields for 'Flying from' and 'Flying to', both with placeholder text 'City or airport' and a clear 'x' button. Below these are 'Departing' and 'Returning' date fields with a calendar icon and placeholder 'mm/dd/yyyy'. To the right of these are dropdowns for 'Adults (18+)' (set to 1) and 'Children (0-17)' (set to 0). At the bottom of the form are checkboxes for 'Add a hotel' and 'Add a car', and a prominent yellow 'Search' button.

Ordered Map ADT

◆ Map ADT + the following methods

- `firstEntry(k)`: Return an iterator to the entry with smallest key value; if the map is empty, it returns end.
- `lastEntry(k)`: Return an iterator to the entry with largest key value; if the map is empty, it returns end.
- `ceilingEntry(k)`: Return an iterator to the entry with the least key value greater than or equal to k ; if there is no such entry, it returns end.
- `floorEntry(k)`: Return an iterator to the entry with the greatest key value less than or equal to k ; if there is no such entry, it returns end.
- `lowerEntry(k)`: Return an iterator to the entry with the greatest key value less than k ; if there is no such entry, it returns end.
- `higherEntry(k)`: Return an iterator to the entry with the least key value greater than k ; if there is no such entry, it returns end.

◆ SkipList is one of the efficient way of implementing ordered Maps

Implementing Ordered Map

◆ Natural choice

- Sorted list based implementation
- $O(n)$ searching, insertion, deletion complexity

◆ Lesson from HashTable

- Unordered map can be implemented by $O(1)$ time (on average)

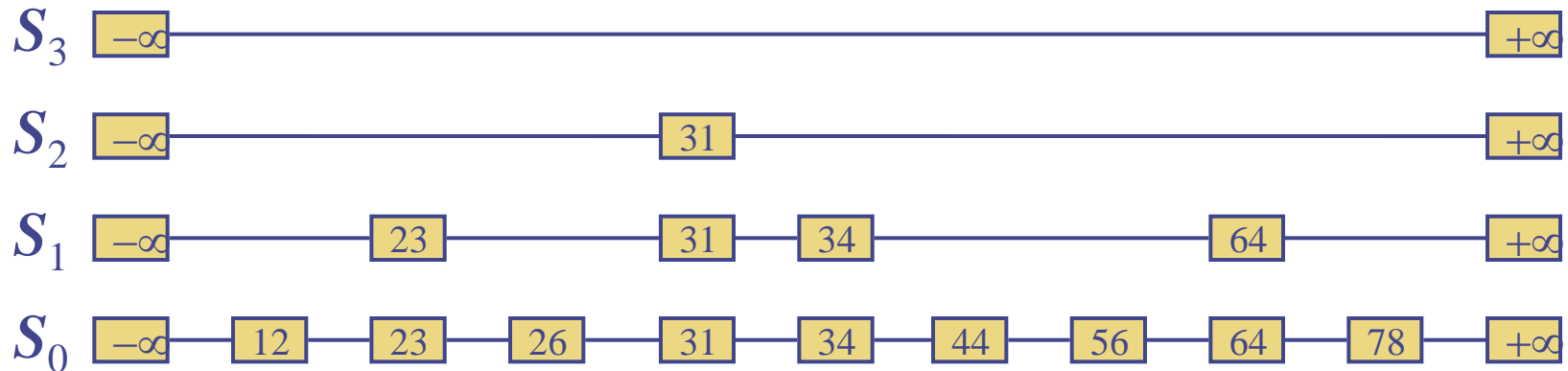
◆ Can we imagine similar things for ordered map?

- SkipList

<i>Operation</i>	<i>Time</i>
size, empty	$O(1)$
firstEntry, lastEntry	$O(1)$
find, insert, erase	$O(\log n)$ (expected)
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$ (expected)

What is a Skip List

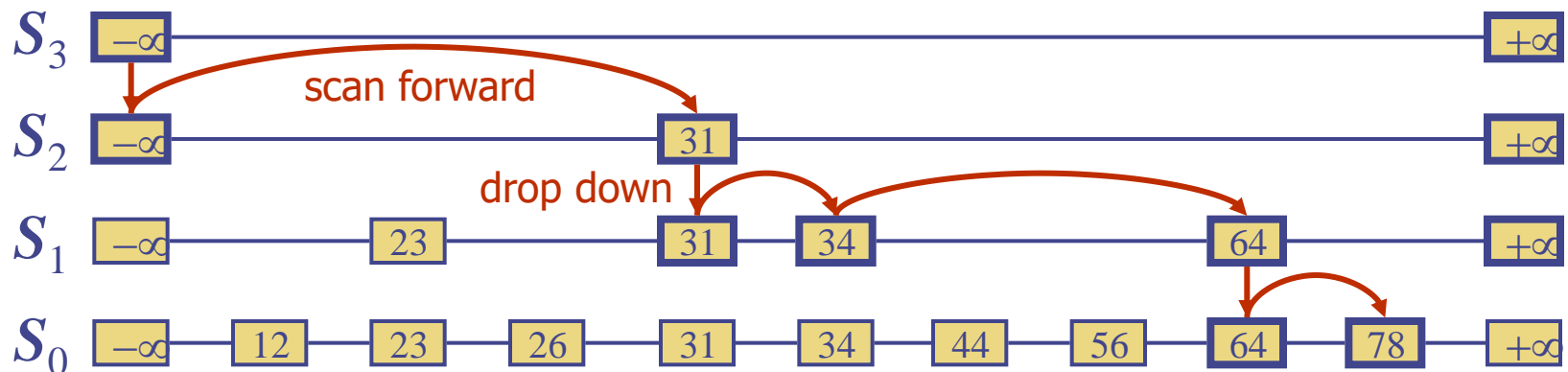
- ◆ A **skip list** for a set S of distinct (key, element) items is a series of lists S_0, S_1, \dots, S_h such that
 - Each list S_i contains the special keys $+\infty$ and $-\infty$
 - List S_0 contains the keys of S in nondecreasing order
 - Each list is a subsequence of the previous one, i.e.,
$$S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$$
 - List S_h contains only the two special keys
- ◆ We show how to use a skip list to implement the ordered MAP ADT



Search

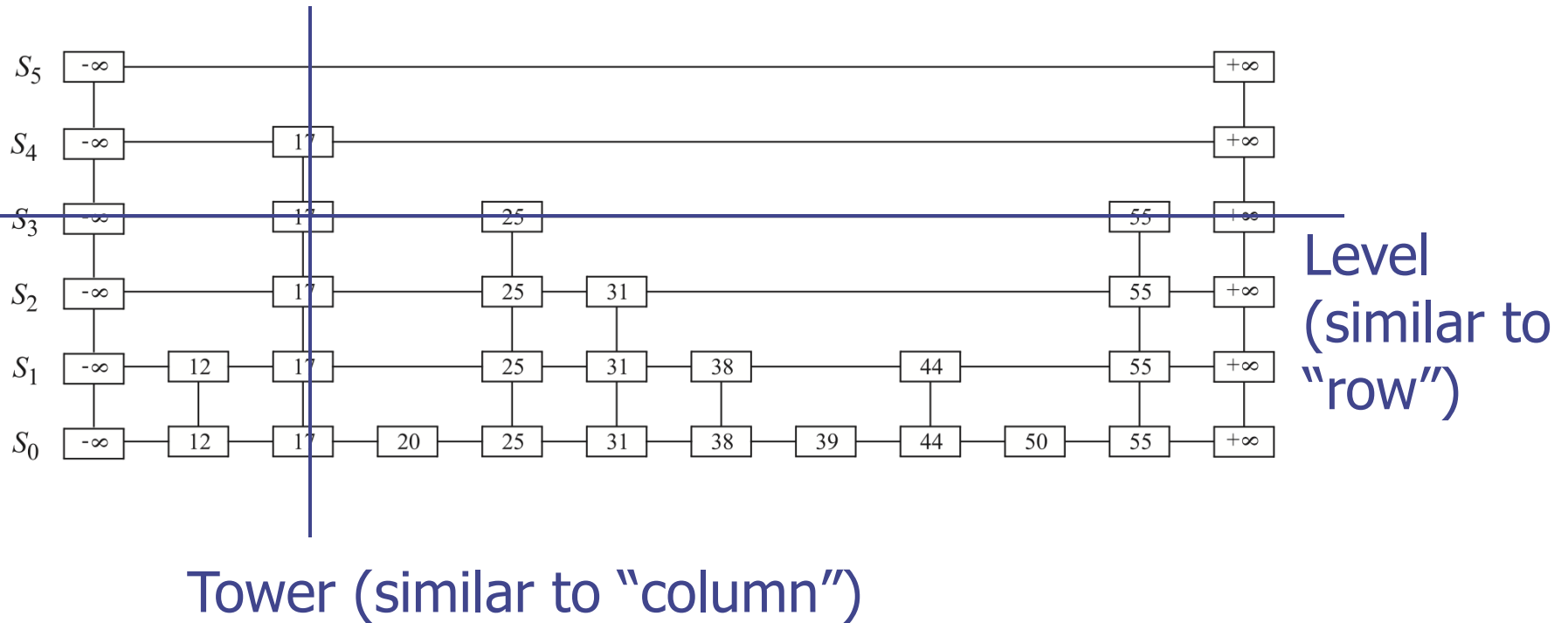
- ◆ We search for a key x in a skip list as follows:
 - We start at the first position of the top list
 - At the current position p , we compare x with $y \leftarrow \text{key}(\text{next}(p))$
 - $x = y$: we return $\text{element}(\text{next}(p))$
 - $x > y$: we “scan forward”
 - $x < y$: we “drop down”
 - If we try to drop down past the bottom list, we return *null*

- ◆ Example: search for 78



Terminology

Height = 5



(Note) Randomized Algorithms

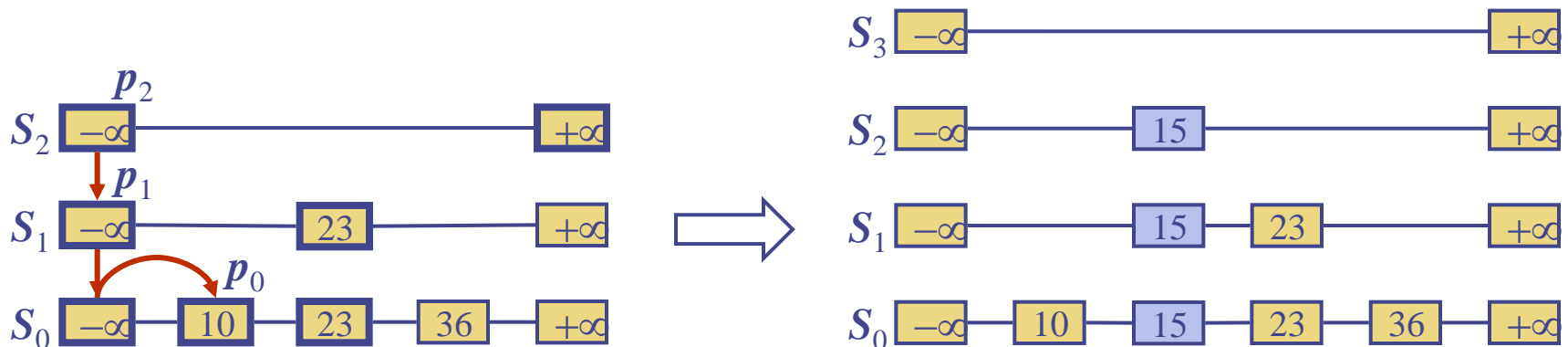
- ◆ A **randomized algorithm** performs coin tosses (i.e., uses random bits) to control its execution
- ◆ It contains statements of the type

```
b ← random()  
if b = 0  
  do A ...  
else { b = 1 }  
  do B ...
```
- ◆ Its running time depends on the outcomes of the coin tosses
- ◆ We analyze the expected running time of a randomized algorithm under the following assumptions
 - the coins are unbiased, and
 - the coin tosses are independent
- ◆ The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give “heads”)
- ◆ We use a randomized algorithm to insert items into a skip list

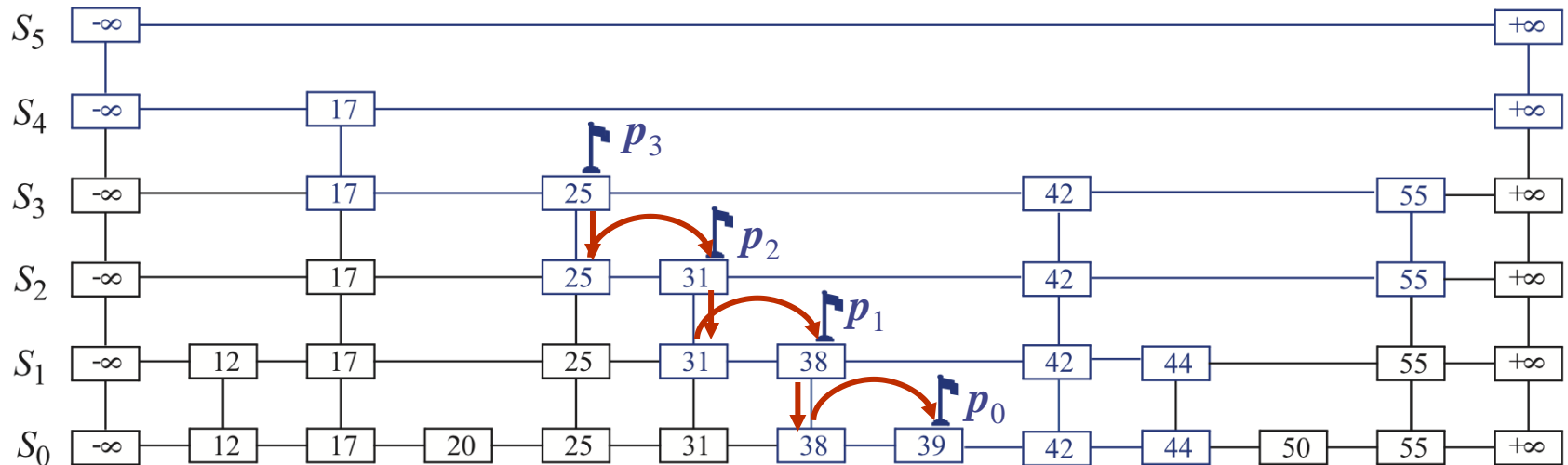
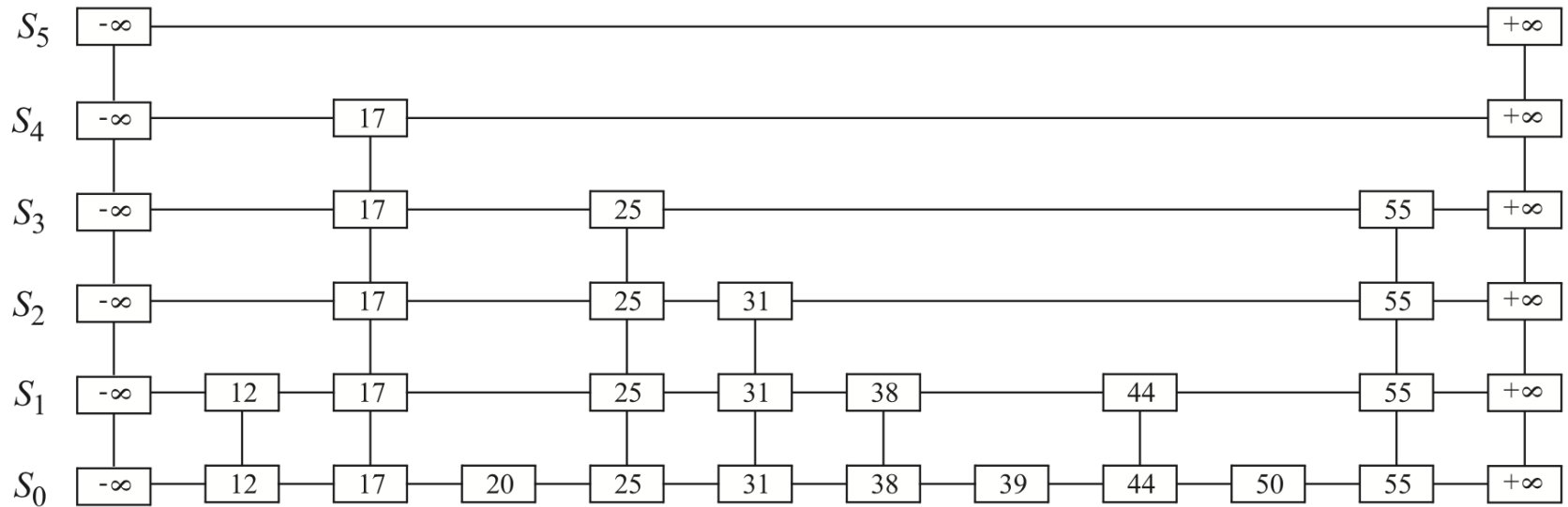
Insertion

- ◆ To insert an entry (x, o) into a skip list, we use a randomized algorithm:
 - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
 - If $i \geq h$, we add (to the skip list) new lists S_{h+1}, \dots, S_{i+1} , each containing only the two special keys, and do nothing, otherwise
 - We search for x in the skip list and find the positions p_0, p_1, \dots, p_i of the items with largest key less than x in each list S_0, S_1, \dots, S_i
 - For $j \leftarrow 0, \dots, i$, we insert item (x, o) into list S_j after position p_j

- ◆ Example: insert key 15, with $i = 2$



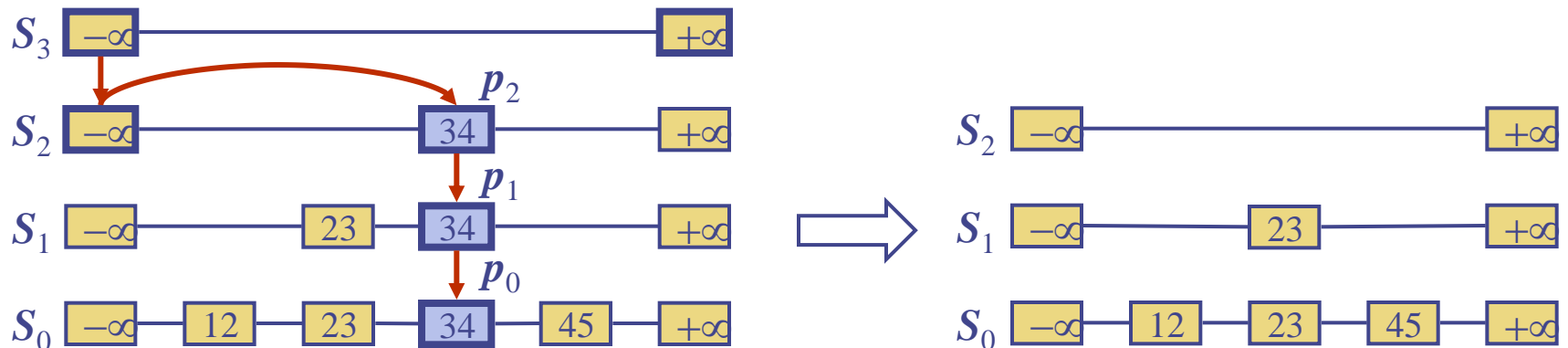
Example of Insertion of Key “42” with $i = 3$



Deletion

- ◆ To remove an entry with key x from a skip list, we proceed:
 - We search for x in the skip list and find the positions p_0, p_1, \dots, p_i of the items with key x , where position p_j is in list S_j
 - We remove positions p_0, p_1, \dots, p_i from the lists S_0, S_1, \dots, S_i
 - We remove all but one list containing only the two special keys

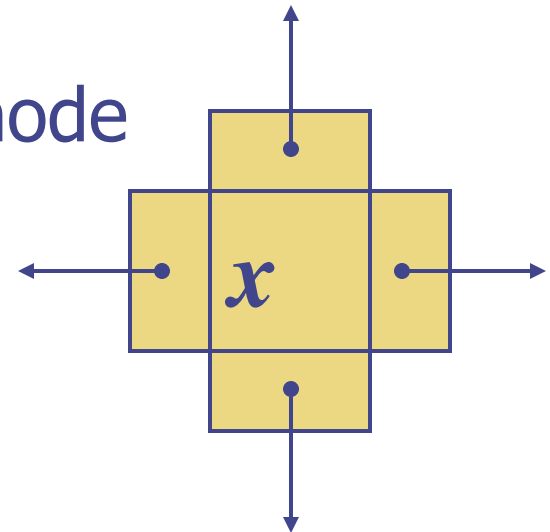
◆ Example: remove key 34



Implementation

- ◆ We can implement a skip list with quad-nodes
- ◆ A quad-node stores:
 - entry
 - link to the node prev
 - link to the node next
 - link to the node below
 - link to the node above
- ◆ Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them

quad-node



Performance of skiplist

◆ Space

- $O(n)$
- Surprising? (Note that an element is stored in multiple places)

◆ Time

<i>Operation</i>	<i>Time</i>
size, empty	$O(1)$
firstEntry, lastEntry	$O(1)$
find, insert, erase	$O(\log n)$ (expected)
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$ (expected)

- ◆ You will see why the probability course helps here. Be ready for math

Space Usage

- ◆ The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm

- ◆ We use the following two basic probabilistic facts:

Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$

Fact 2: If each of n entries is present in a set with probability p , the expected size of the set is np (*expectation of a binomial distribution*)

- ◆ Consider a skip list with n entries
 - By Fact 1, we insert an entry in list S_i with probability $1/2^i$
 - ◆ Why? Because we insert the entry for all levels $\leq i$
 - By Fact 2, the expected size of list S_i is $n/2^i$

- ◆ The expected number of nodes used by the skip list is

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

- ◆ Thus, the expected space usage of a skip list with n items is $O(n)$

Height

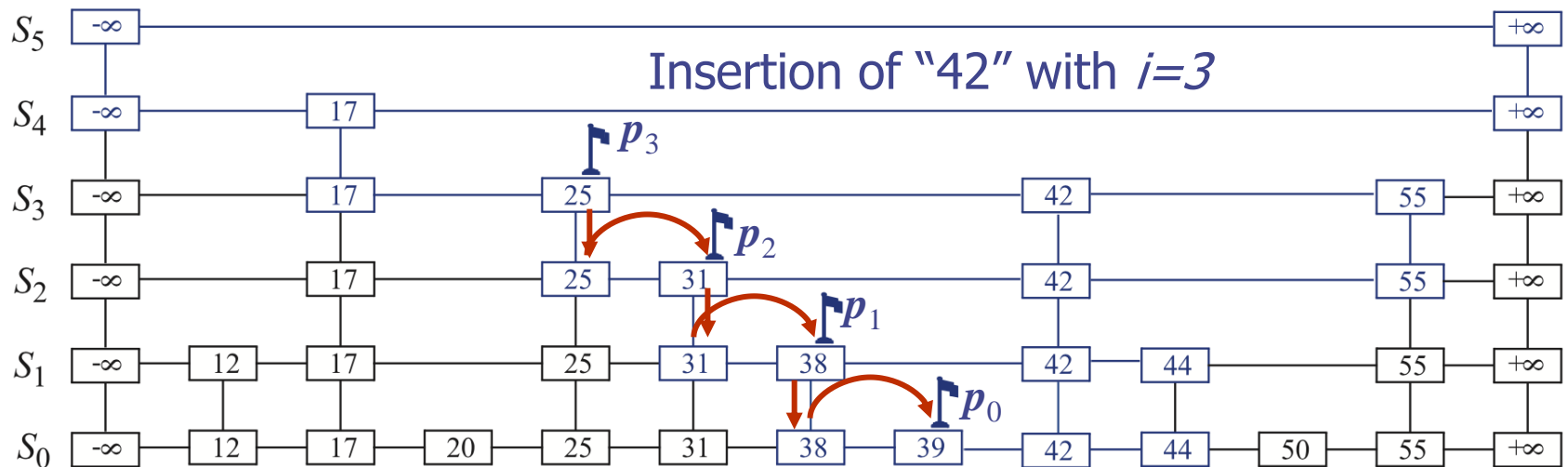
- ◆ The running time of the search and insertion algorithms is affected by the height h of the skip list
- ◆ We show that with **high probability**, a skip list with n items has height $O(\log n)$
- ◆ We use the following additional probabilistic fact:
 - Fact 3:** If each of n events has probability p , the probability that at least one event occurs is at most np (*union bound*)
- ◆ Consider a skip list with n entries
 - By Fact 1, we insert an entry in list S_i with probability $1/2^i$
 - By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
- ◆ By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one entry is at most
$$n/2^{3\log n} = n/n^3 = 1/n^2$$
- ◆ Thus a skip list with n entries has height at most $3\log n$ with probability at least $1 - 1/n^2$
- ◆ The height is $O(\log n)$ with “high probability”

Search and Update Times (1)

- ◆ The search time in a skip list is proportional to
 - the number of drop-down steps, plus
 - the number of scan-forward steps
- ◆ The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability
- ◆ To analyze the scan-forward steps, we use yet another probabilistic fact:
 - Fact 4:** The expected number of coin tosses required in order to get tails is 2

Search and Update Times (2)

- ◆ After the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level $i+1$
 - The probability that any further key is examined is $\frac{1}{2}$
 - How many additional keys should be examined at each level i on average?
- ◆ By Fact 4, in each list the expected number of scan-forward steps is 2, i.e., $O(1)$
- ◆ Thus, the expected number of scan-forward steps is $O(\log n)$
- ◆ We conclude that a search in a skip list takes $O(\log n)$ expected time
- ◆ The analysis of insertion and deletion gives similar results



Summary

- ◆ A skip list is a data structure for dictionaries that uses a randomized insertion algorithm
- ◆ In a skip list with n entries
 - The expected space used is $O(n)$
 - The expected search, insertion and deletion time is $O(\log n)$
- ◆ Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability
- ◆ Skip lists are fast and simple to implement in practice