CSE221 Data Structures
Lecture 17: Ordered Maps and Skip Lists

Antoine Vigneron
antoine@unist.ac.kr

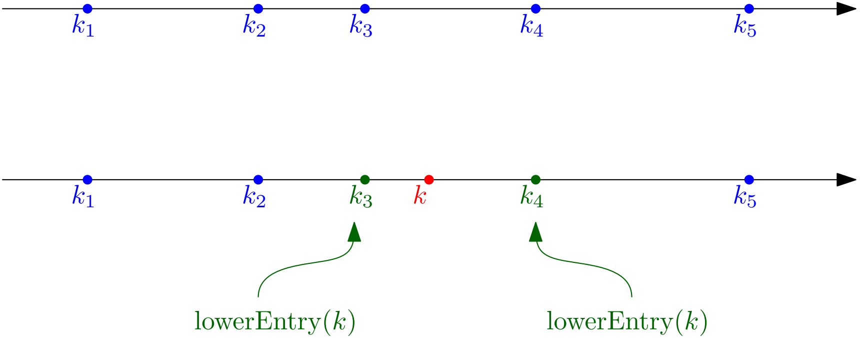
Ulsan National Institute of Science and Technology

November 15, 2021

Introduction

- I updated attendance records. They can be found in the portal under E-attendance.
- I will post Assignment 3 tonight.
- Reference for this lecture: Textbook Chapter 9.3–9.4.

Ordered Maps



- An *ordered map* ADT is similar with a map ADT, but uses an ordering of the keys.
- In particular, it may use key *comparators*. (See Lecture 15.)

Ordered Maps

- In addition to the map ADT, the *ordered map* ADT supports the following functions:

- **firstEntry**(k): Return an iterator to the entry with smallest key value.
- **lastEntry**(k): Return an iterator to the entry with largest key value.
- **ceilingEntry**(k): Return an iterator to the entry with the least key value greater than or equal to k.
- **floorEntry**(k): Return an iterator to the entry with the greatest key value less than or equal to k.
- **lowerEntry**(k): Return an iterator to the entry with the greatest key value less than k.
- **higherEntry**(k): Return an iterator to the entry with the least key value greater than k.

(When there is no such entry, we return an iterator `end`.)

Ordered Maps

- Intuitively, a map implemented with a hash table would not be a good implementation of an ordered map, because the entries are not placed in any particular order in the table. In fact, the hash function is chosen in such a way that the positions look random.
- So we will need different implementation.

Example

- A database records flights in an ordered map, with keys

$$k = (\text{origin}, \text{destination}, \text{date}, \text{time}).$$

  in lexicographical order.
- Suppose you want to take a flight from ICN to CDG after Nov 17, 9:30 am.
- You can find the 4 earliest such flights by calling `ceilingEntry`(ICN, CDG, 17Nov, 09:30), followed by 3 calls to `higherEntry` on the successive results:

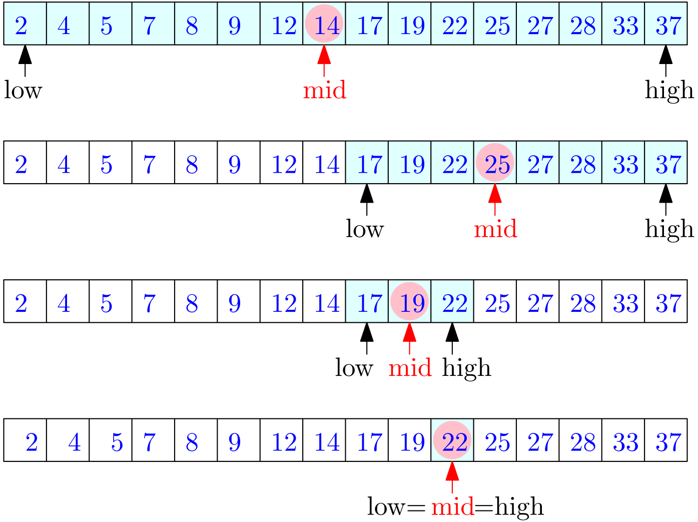
$$\begin{array}{c}
((\text{ICN, CDG, 17Nov, 09:55}), (\text{AF237} \ldots)) \\
((\text{ICN, CDG, 17Nov, 15:15}), (\text{KE516} \ldots)) \\
((\text{ICN, CDG, 17Nov, 19:50}), (\text{OZ218} \ldots)) \\
((\text{ICN, CDG, 18Nov, 00:45}), (\text{AF354} \ldots))
\end{array}$$

Ordered Search Tables

0	1	2	3	4	5	6	7	8	9	10
4	6	9	12	15	16	18	28	34		

- An *ordered search table* is a vector L where the entries are stored by increasing key values.
- Assuming we grow and shrink L in an appropriate manner, the space usage is $O(n)$.
- Inserting an entry (k, v) takes $O(n)$ time as we may need to shift a linear number of entries.
- Deleting an element also takes $O(n)$ time.
- On the other hand, we will see that it allows us to perform other operations to run in $O(\log n)$ time.

Binary Search: Example with Find(22)



Binary Search

- We are given a vector L of n entries, with index 0 to $n-1$.
- The `find` operation of the map ADT can be implemented by *binary search*.
- We recurse on an interval of indices [low, high] where $0 \leqslant \text{low} \leqslant \text{high} \leqslant n-1$. Initially, low$=0$ and high$=n-1$.
- We make sure that, if the key is in L, its index must be in the interval [low, high].
- Let e be the entry with index

$$\text{mid} = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor.$$

Binary Search

- If $k = e$.key, then we are done: We found k.
- If $k < e$.key, then we recurse on the range [low, mid−1].
- If $k > e$.key, then we recurse on the range [mid+1, high].

Pseudocode

```
procedure BinarySearch(L, k, low, high)
    if low > high then
        return end
    mid ← ⌊(low + high)/2⌋
    e ← L[mid]
    if k = e.key then
        return e
    if k < e.key then
        return BinarySearch(L, k, low, mid−1)
    return BinarySearch(L, k, mid+1, high)
```

Analysis

- Each recursive call takes $O(1)$ time.
- Let $n_i =$high$-$low$+1$ denote the size of the range under consideration at step i
- Initially, $n_0 = n$. At each step i, we have $n_{i+1} \leqslant n_i/2$.
- So at step i, we have $1 \leqslant n_i \leqslant n/2^i$.
- It follows that $2^i \leqslant n$ and thus $i \leqslant \log n$.
- In summary, there are at most $\log n$ steps, each step takes $O(1)$ time, so the running time of binary search is

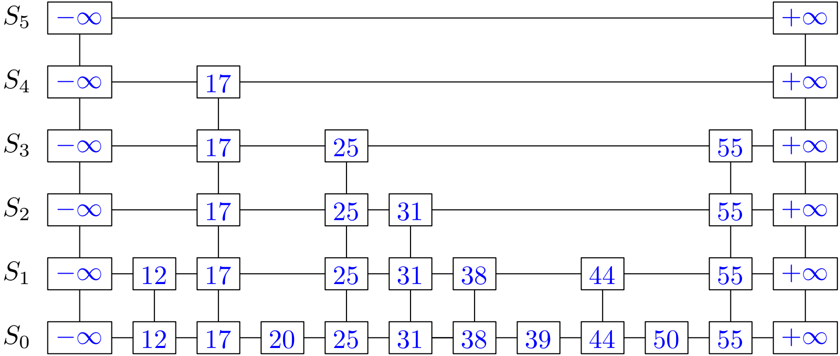
$$\mathrm{O}(\log \mathrm{n}).$$

Comparing Map Implementations

method	List	Hash Table	Search Table
size, empty	$O(1)$	$O(1)$	$O(1)$
find	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(\log n)$
insert	$O(1)$	$O(1)$	$O(n)$
erase	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(n)$

- We will now show that *skip lists* allow us to perform all the operations of the *ordered* map ADT efficiently.

Skip Lists

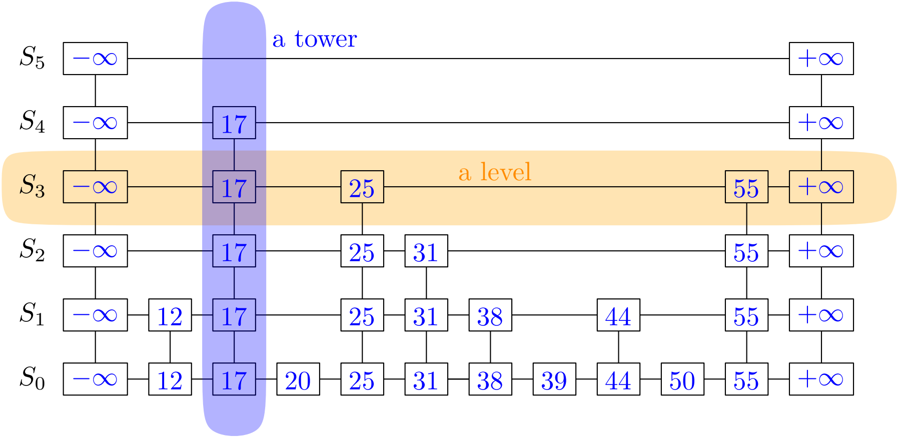


S_5 $-\infty$ $+\infty$
S_4 $-\infty$ 17 $+\infty$
S_3 $-\infty$ 17 25 55 $+\infty$
S_2 $-\infty$ 17 25 31 55 $+\infty$
S_1 $-\infty$ 12 17 25 31 38 44 55 $+\infty$
S_0 $-\infty$ 12 17 20 25 31 38 39 44 50 55 $+\infty$

Skip Lists

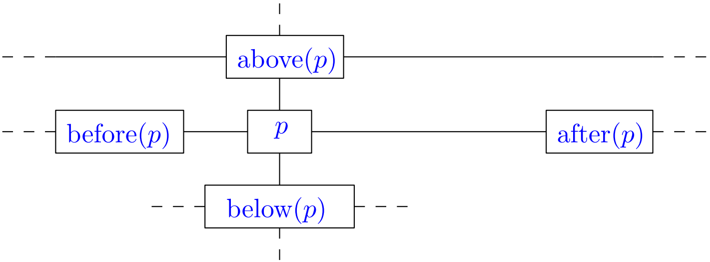
- A *skip list* for a map M consists of a sequence of lists $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$.
- h is the *height* of the skip list.
- Each element of S_i appears in S_{i+1} with probability $1/2$.
- So the size of S_{i+1} is roughly half of the size of S_i.
- This is called *randomization*: We make random choices to construct the data structure.
- Then we will show that, *on average*, a search or update operation takes $O(\log n)$ time. This is an *expected* time bound, as opposed to worst-case time bounds for ordered search tables.

Skip Lists



a tower
a level
S_5 $-\infty$ $+\infty$
S_4 $-\infty$ 17 $+\infty$
S_3 $-\infty$ 17 25 55 $+\infty$
S_2 $-\infty$ 17 25 31 55 $+\infty$
S_1 $-\infty$ 12 17 25 31 38 44 55 $+\infty$
S_0 $-\infty$ 12 17 20 25 31 38 39 44 50 55 $+\infty$

Skip Lists

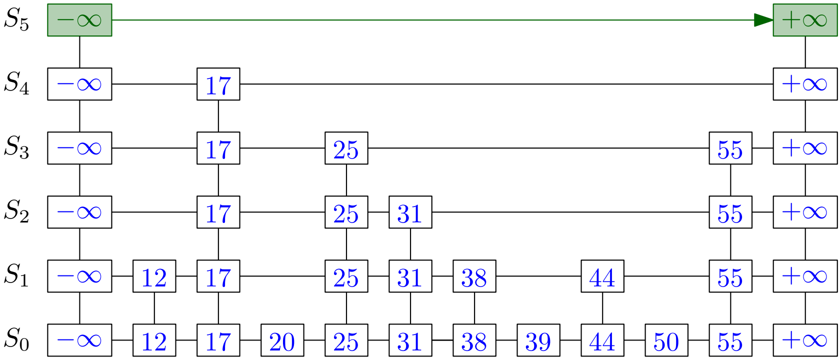


Operations for traversing a skip list

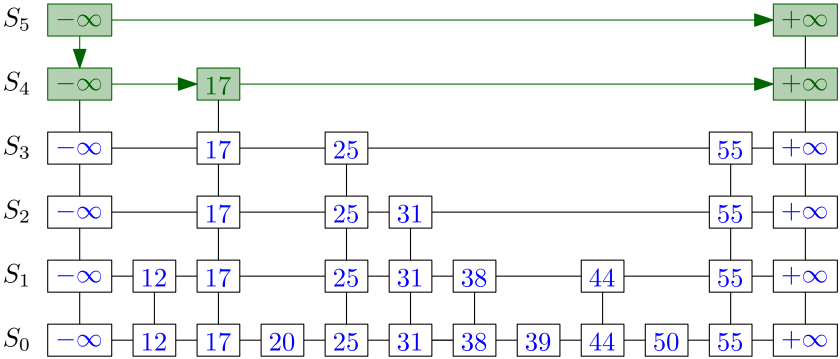
- **after**(p): Return the position following p on the same level.
- **before**(p): Return the position preceding p on the same level.
- **below**(p): Return the position below p in the same tower.
- **above**(p): Return the position above p in the same tower.

- Using a linked structure,each of these operations takes $O(1)$ time

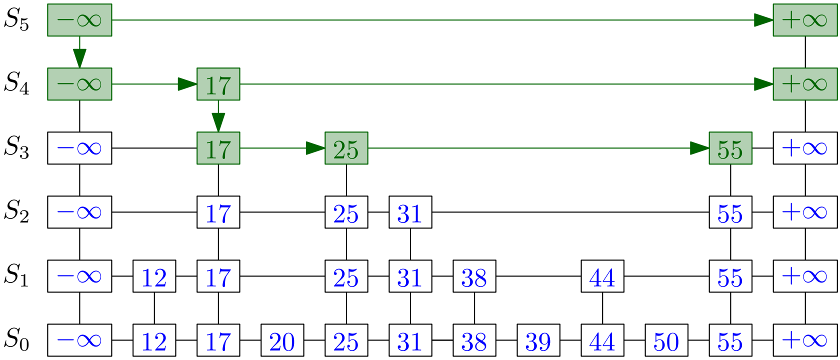
Searching for Key 50



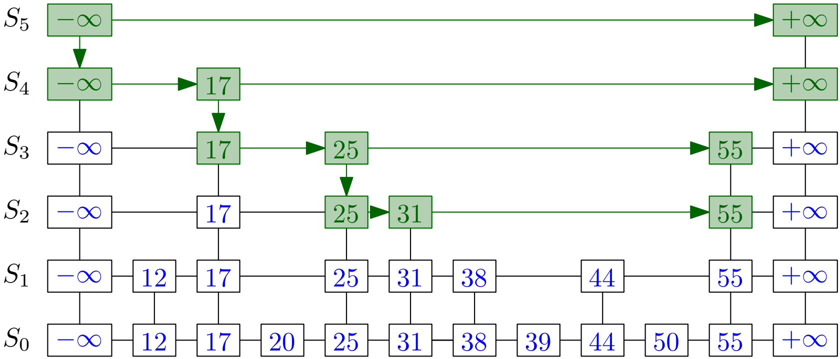
Searching for Key 50



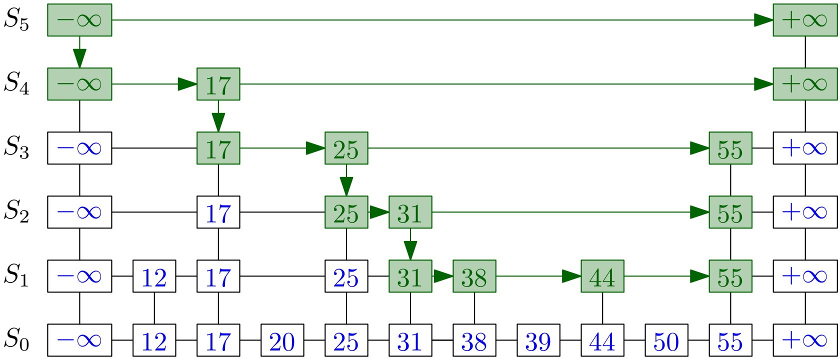
Searching for Key 50



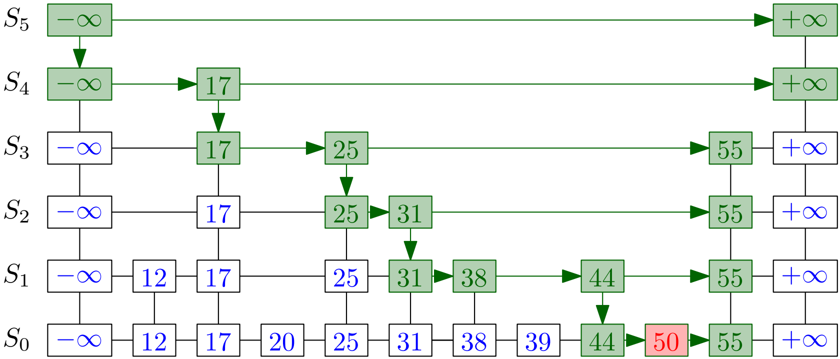
Searching for Key 50



Searching for Key 50



Searching for Key 50



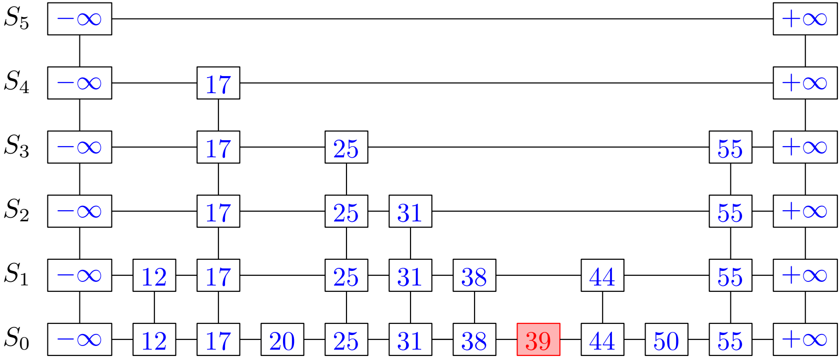
Searching in a Skip List

Pseudocode

1: **procedure** SkipSearch(k)
2:     $p \leftarrow s$
3:     **while** below(p) $\neq$ NULL **do**
4:         $p \leftarrow$ below(p) $\triangleright$ drop down
5:         **while** $k \geqslant$ key(after(p)) **do**
6:             $p \leftarrow$ after(p) $\triangleright$ scan forward

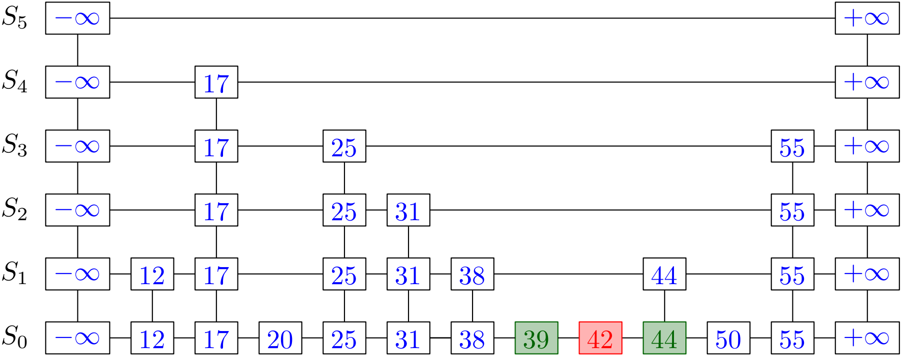
- Searching in a skip list S. Variable s holds the start position of S.
- This function returns the position p in the bottom list S_0 such that the entry at p has the largest key less than or equal to k.
- The operation $p \leftarrow$ below(p) is called *drop down*. We drop down until we reach the *bottom*, in which case we are done.
- At Line 5–6, we *scan forward*: We look for the right-most position p such that key(p) $\leqslant k$.

Inserting 42



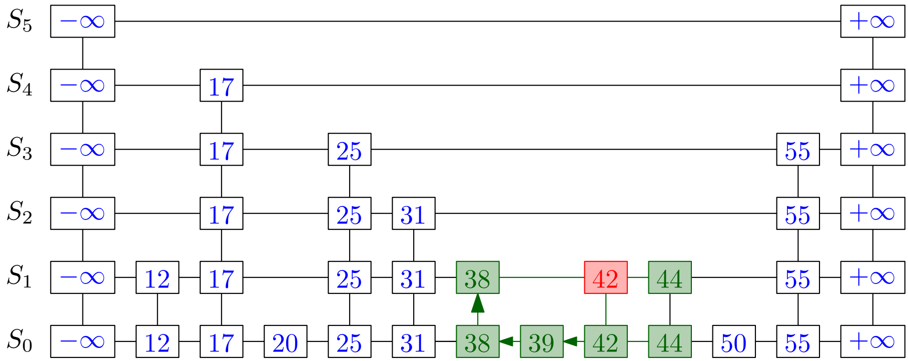
- First perform SkipSearch(42).

Inserting 42



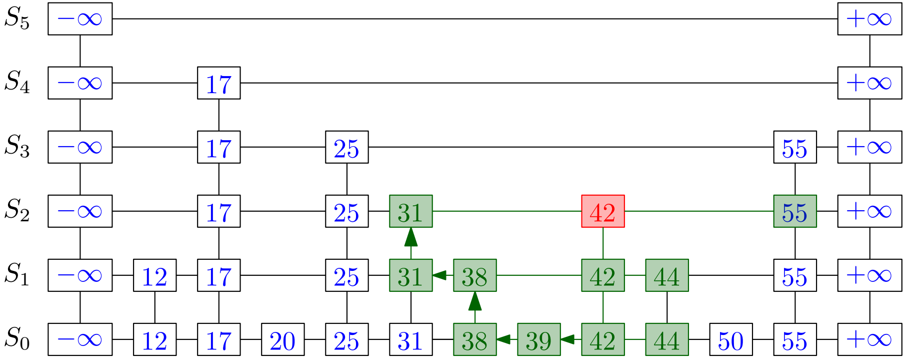
- Insert new node at the bottom.

Inserting 42



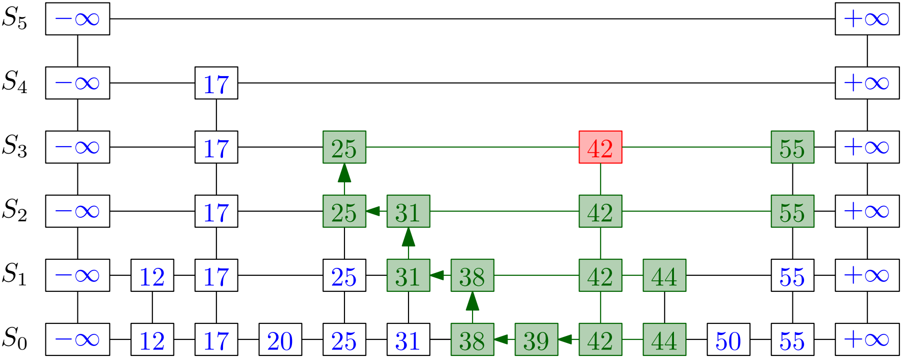
- With probability $1/2$: coinFlip()=tails. Grow the tower.

Inserting 42



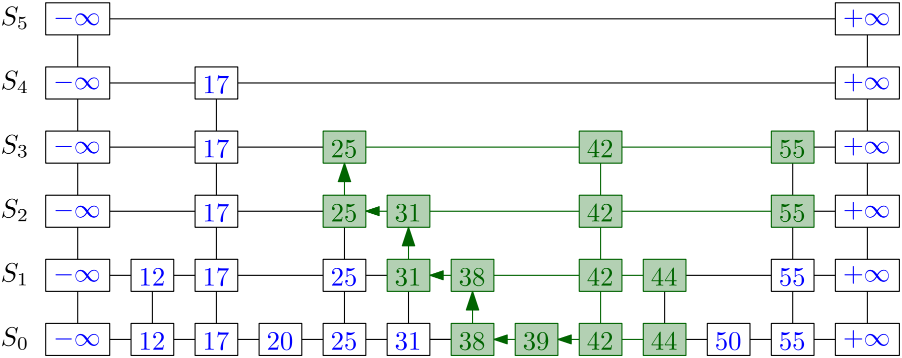
- coinFlip()=tails. Grow the tower.

Inserting 42



- coinFlip()=tails. Grow the tower.

Inserting 42

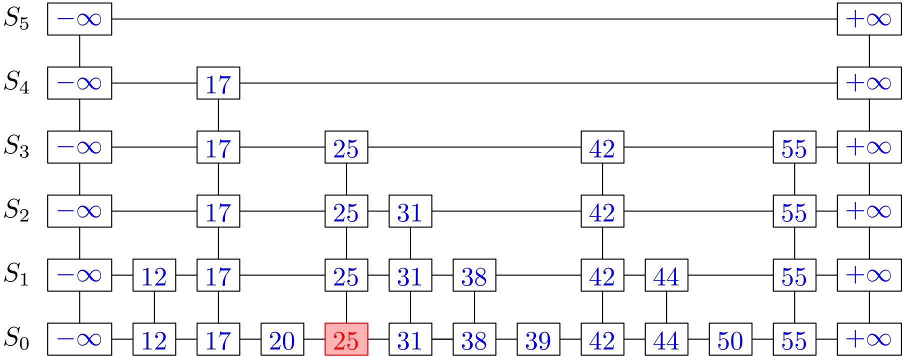


- coinFlip()=heads: We are done

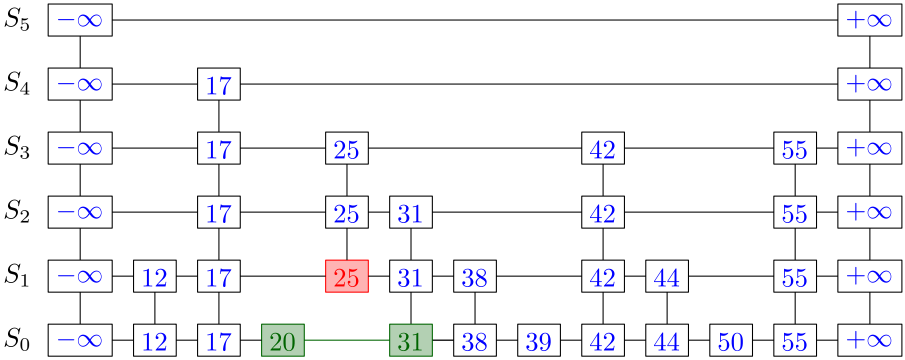
Skip Lists

- Detailed pseudocode is given in the textbook.
- Remark: If the tower grows beyond S_h, we need to add new levels to the skip list.

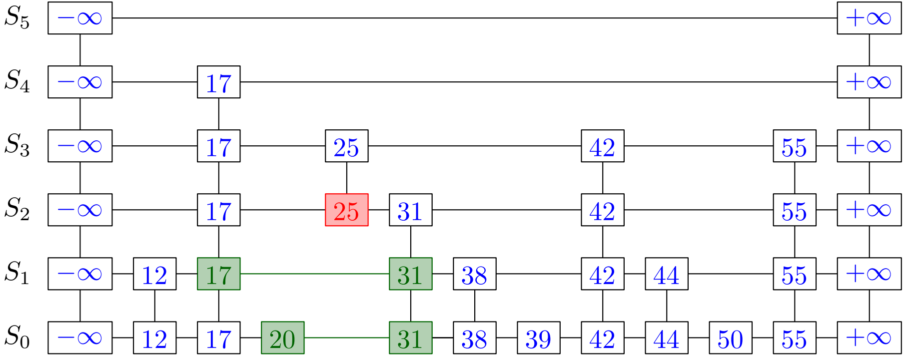
Removal of 25: First perform SKIPSEARCH(25).



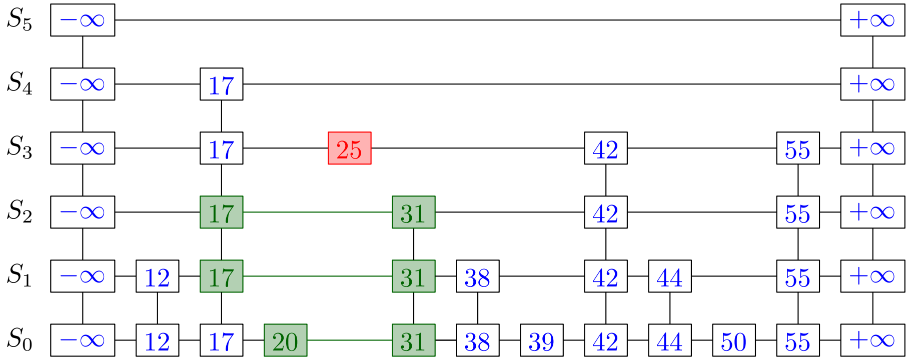
Removal of 25



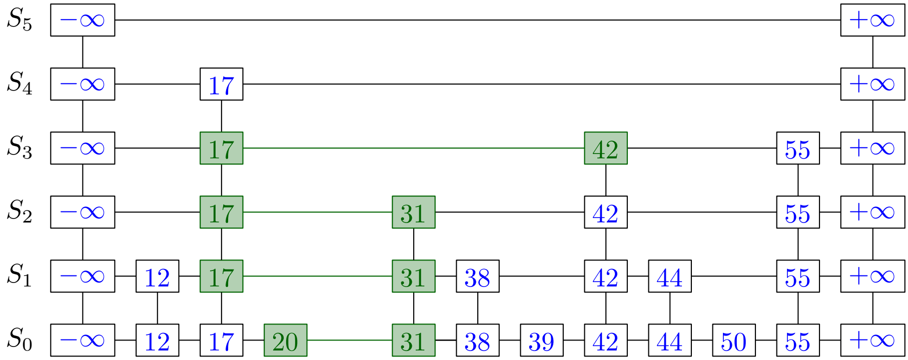
Removal of 25



Removal of 25



Removal of 25



Probabilistic Analysis of Skip Lists

- We first bound the height of a skip list.
- Let e be an entry of a skip list, and $h(e)$ the height of its tower.
- Then

$$\Pr[h(e) \geqslant i] = \Pr[\text{ first } i \text{ coin flips are "tails"}] = 1/2^i$$

- The probability that level i is non-empty is

$$\begin{aligned}\Pr[S_i \neq \emptyset] &= \Pr[\text{ there is an entry } e \text{ such that } h(e) \geqslant i] \\ &\leqslant \sum_{e \in S} \Pr[h(e) \geqslant i] \\ &= n \cdot (1/2^i).\end{aligned}$$

Probabilistic Analysis of Skip Lists

- So the probability that the height h of S is at most $3 \log n$ satisfies:

$$\begin{aligned} \Pr[h \geqslant 3 \log n] &\leqslant n \cdot \frac{1}{2^{3 \log n}} \\ &= n \cdot \frac{1}{n^3} = \frac{1}{n^2}. \end{aligned}$$

- We say that $h = O(\log n)$ with *high probability* because $h < c \log n$ with probability at least $1 - 1/n^{c-1}$, for some constant $c > 1$. (Here $c = 3$.)

Example

If S records 1,000 entries, then the probability that its height is more than $3 \log 1000 \approx 30$ is less than 10^{-6}.

Probabilistic Analysis of Skip Lists

- With high probability, $h = O(\log n)$.
- So in a search operation, the number of *drop-down* moves is $O(\log n)$.
- What about the number of *scan-forward*?
- Let n_i be the number of keys examined while scanning forward at level i.
- None of these n_i keys was in S_{i+1}, otherwise they would have been scanned at the previous step.
- Each key of S_i is in S_{i+1} with probability $1/2$.
- Therefore, the expected value of n_i is exactly equal to the expected number of times we must flip a fair coin before it comes up heads. This expected value is 2.

Probabilistic Analysis of Skip Lists

- So for each level, we perform 1 drop-down and an expected number of 2 scan-forward.
- Hence the expected running time is $O(h) = O(\log n)$ with high probability.
- It follows that the expected search time is $O(\log n)$.
- Similarly, we can prove that the expected insertion and removal times are $O(\log n)$.