SE274 Data Structure

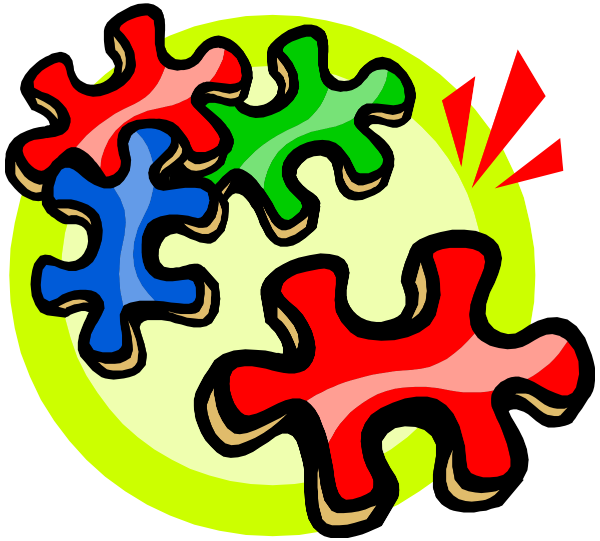
Lecture 6: Maps, Hash Tables, Skip Lists – Part 2

Apr 08, 2020

Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

Sets



Definitions

- A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
    - Elements of a set are like keys of a map, but without any auxiliary values.
- A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.
- A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.
    - For example, the index of a book maps a given term to one or more locations at which the term occurs.

Set ADT

S.add(e): Add element e to the set. This has no effect if the set already contains e.

S.discard(e): Remove element e from the set, if present. This has no effect if the set does not contain e.

e in S: Return True if the set contains element e. In Python, this is implemented with the special __contains__ method.

len(S): Return the number of elements in set S. In Python, this is implemented with the special method __len__.

iter(S): Generate an iteration of all elements of the set. In Python, this is implemented with the special method __iter__.

S.remove(e): Remove element e from the set. If the set does not contain e, raise a KeyError.

S.pop(): Remove and return an arbitrary element from the set. If the set is empty, raise a KeyError.

S.clear(): Remove all elements from the set.

Boolean Set Operations

`S == T`:	Return True if sets S and T have identical contents.
`S != T`:	Return True if sets S and T are not equivalent.
`S <= T`:	Return True if set S is a subset of set T.
`S < T`:	Return True if set S is a *proper* subset of set T.
`S >= T`:	Return True if set S is a superset of set T.
`S > T`:	Return True if set S is a *proper* superset of set T.
`S.isdisjoint(T)`:	Return True if sets S and T have no common elements.

Set Update Operations

S | T: Return a new set representing the union of sets S and T.

S |= T: Update set S to be the union of S and set T.

S & T: Return a new set representing the intersection of sets S and T.

S &= T: Update set S to be the intersection of S and set T.

S ^ T: Return a new set representing the symmetric difference of sets S and T, that is, a set of elements that are in precisely one of S or T.

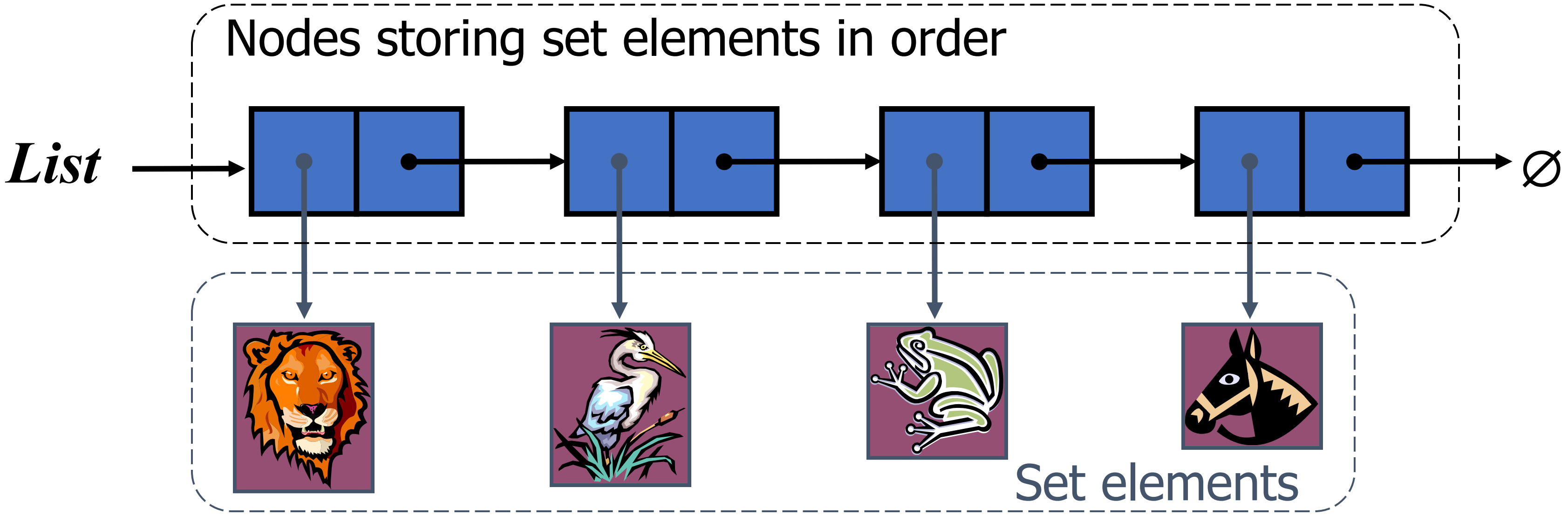
S ^= T: Update set S to become the symmetric difference of itself and set T.

S − T: Return a new set containing elements in S but not T.

S −= T: Update set S to remove all common elements with set T.

Storing a Set in a List

- We can implement a set with a list
- Elements are stored sorted according to some canonical ordering
- The space used is $O(n)$



Generic Merging

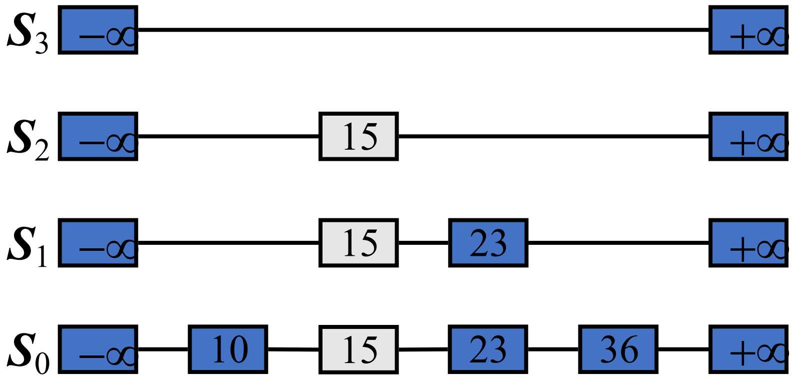
- Generalized merge of two sorted lists A and B
- Template method genericMerge
- Auxiliary methods
  - aIsLess
  - bIsLess
  - bothAreEqual
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        a ← A.first().element();  b ← B.first().element()
        if a < b
            aIsLess(a, S);  A.remove(A.first())
        else if b < a
            bIsLess(b, S);  B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first());  B.remove(B.first())
    while ¬A.isEmpty()
        aIsLess(a, S);  A.remove(A.first())
    while ¬B.isEmpty()
        bIsLess(b, S);  B.remove(B.first())
    return S
```

Using Generic Merge for Set Operations

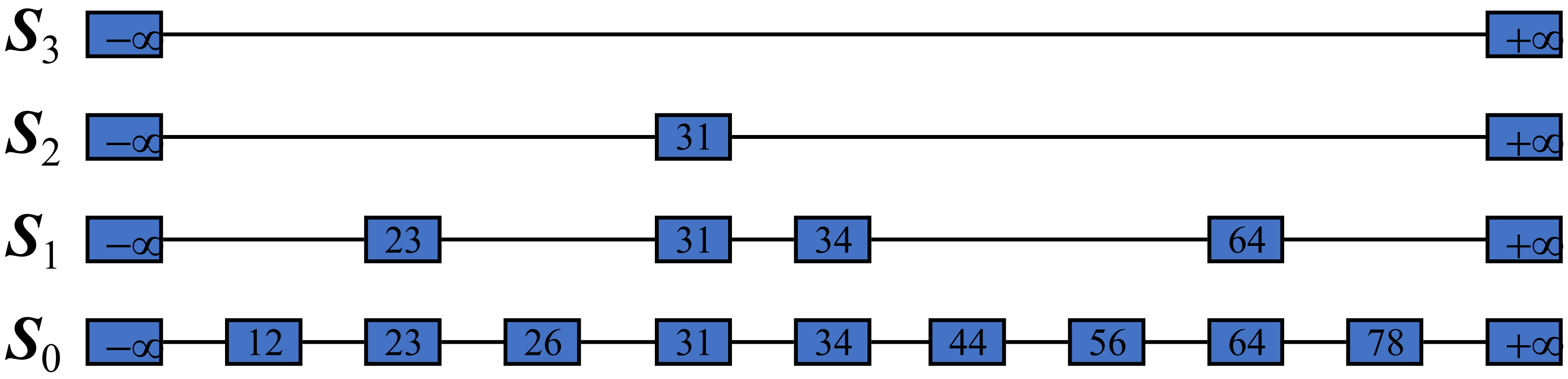
- Any of the set operations can be implemented using a generic merge
- For example:
  - For intersection: only copy elements that are duplicated in both list
  - For union: copy every element from both lists except for the duplicates
- All methods run in linear time

Skip Lists



What is a Skip List

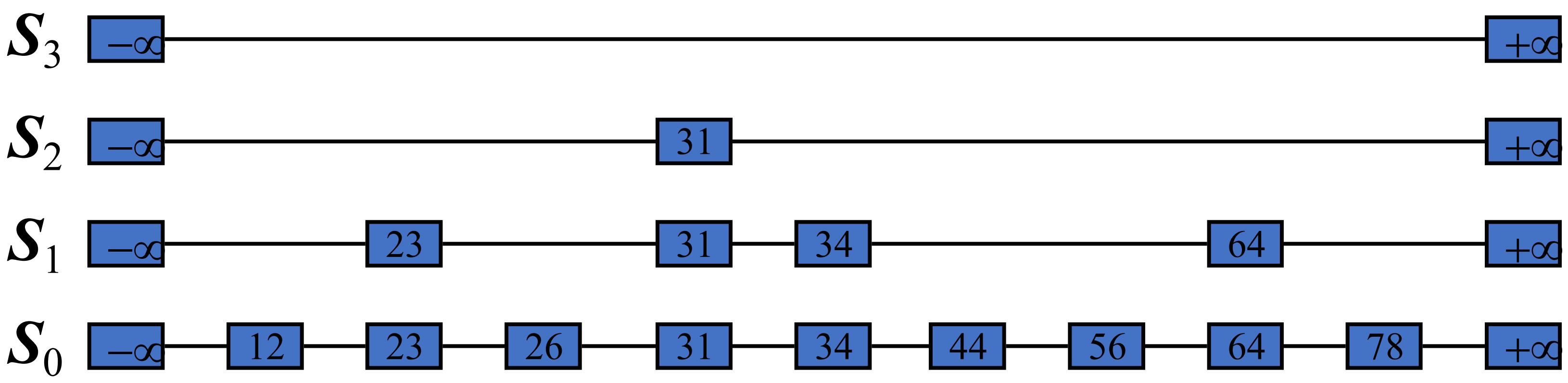
- A skip list for a set S of distinct (key, element) items is a series of lists S_0, $S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in nondecreasing order
  - Each list is a subsequence of the previous one, i.e.,
 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



Actions in a skip list

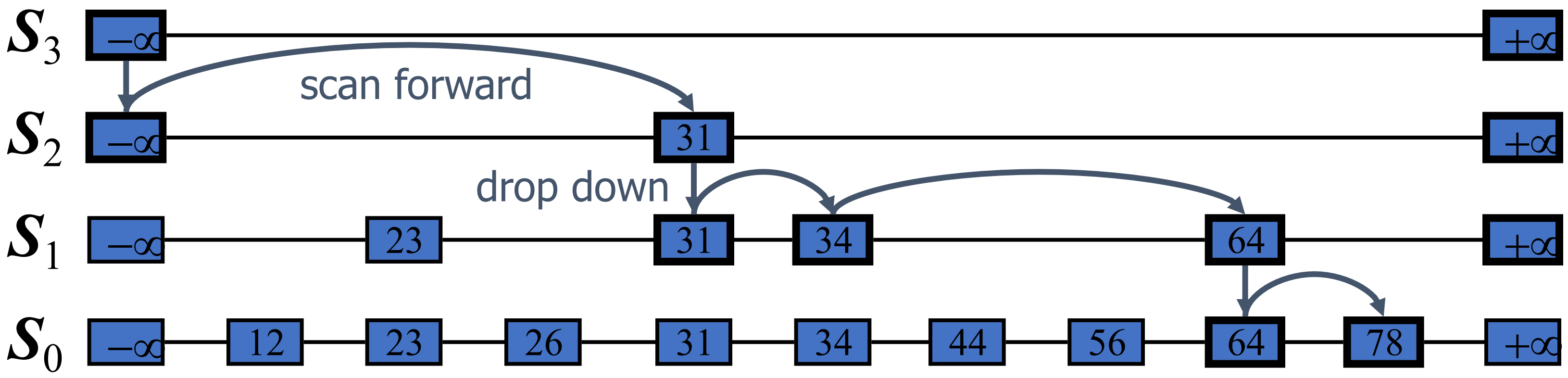
- `next(p)` : Return the position following **p** on the same level.
- `prev(p)` : Return the position preceding **p** on the same level.
- `below(p)` : Return the position below **p** in the same tower.
- `above(p)` : Return the position above **p** in the same tower.

  * The actions return ***None*** if the position requested does not exist



Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(next(p))$
    $x = y$: we return $element(next(p))$
    $x > y$: 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



Search Pseudo-code

Algorithm SkipSearch(k):

Input: A search key k

Output: Position p in the bottom list S_0 with the largest key such that $\text{key}(p) \leq k$

```
p = start                                   {begin at start position}
while below(p) ≠ None do
    p = below(p)                            {drop down}
    while k ≥ key(next(p)) do
        p = next(p)                         {scan forward}
return p.
```

Code Fragment 10.12: Algorthm to search a skip list S for key k.

Randomized Algorithms

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

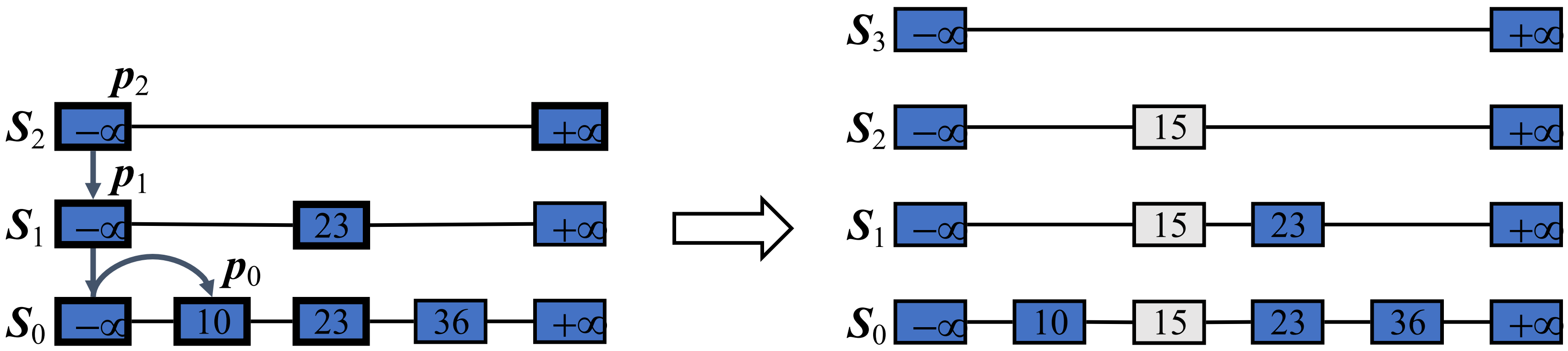
$b \leftarrow \textit{random}()$
if $b = 0$
  do A …
else { $b = 1$ }
  do B …

- 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 $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
  - For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Example: insert key 15, with $i = 2$



Insertion Pseudo-code

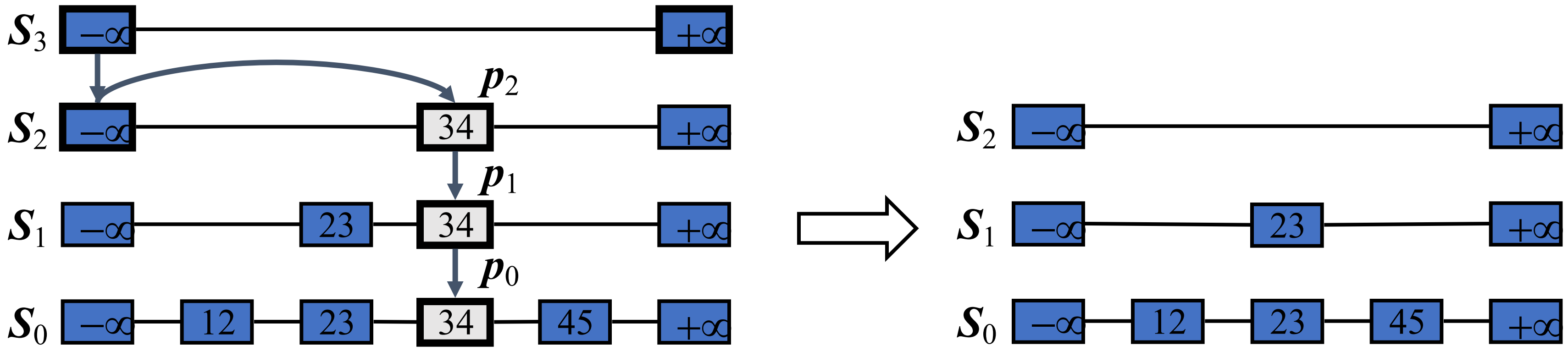
r = insertAfterAbove(p,q,(k,v))
- Inserts a position storing the item (k,v) after position p (on the same level as p) and above position q, returning the new position r.

n: # of entries
h: the height
s: the start node

```
Algorithm SkipInsert(k,v):
  Input: Key k and value v
  Output: Topmost position of the item inserted in the skip list
  p = SkipSearch(k)
  q = None                          {q will represent top node in new item's tower}
  i = −1
  repeat
    i = i+1
    if i ≥ h then
      h = h+1                                  {add a new level to the skip list}
      t = next(s)
      s = insertAfterAbove(None, s, (−∞, None))           {grow leftmost tower}
      insertAfterAbove(s, t, (+∞, None))                  {grow rightmost tower}
    while above(p) is None do
      p = prev(p)                                               {scan backward}
    p = above(p)                                          {jump up to higher level}
    q = insertAfterAbove(p, q, (k, v))        {increase height of new item's tower}
  until coinFlip() == tails
  n = n+1
  return q
```

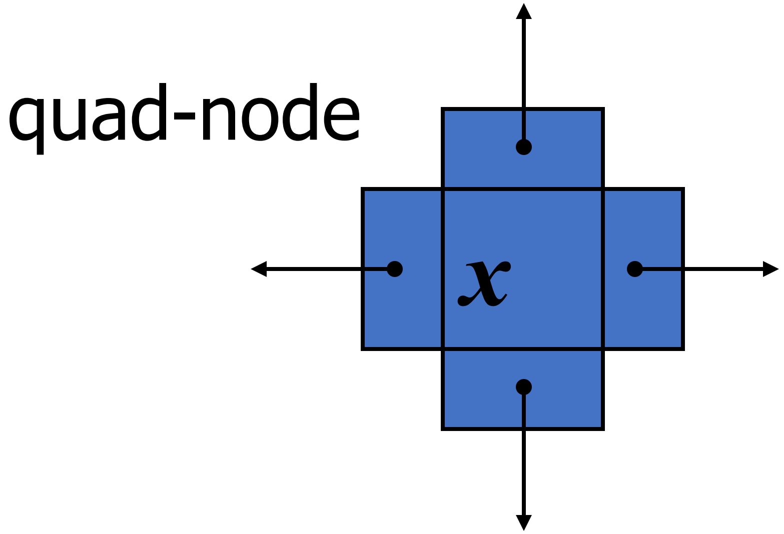
Deletion

- To remove an entry with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - 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



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

- 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 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 an 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

- Consider a skip list with n entires
  - 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 $1 - 1/n^2$

Search and Update Times

- 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

- When we scan forward in a list, the destination key does not belong to a higher list
  - A scan-forward step is associated with a former coin toss that gave tails
- By Fact 4, in each list the expected number of scan-forward steps is 2
- 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