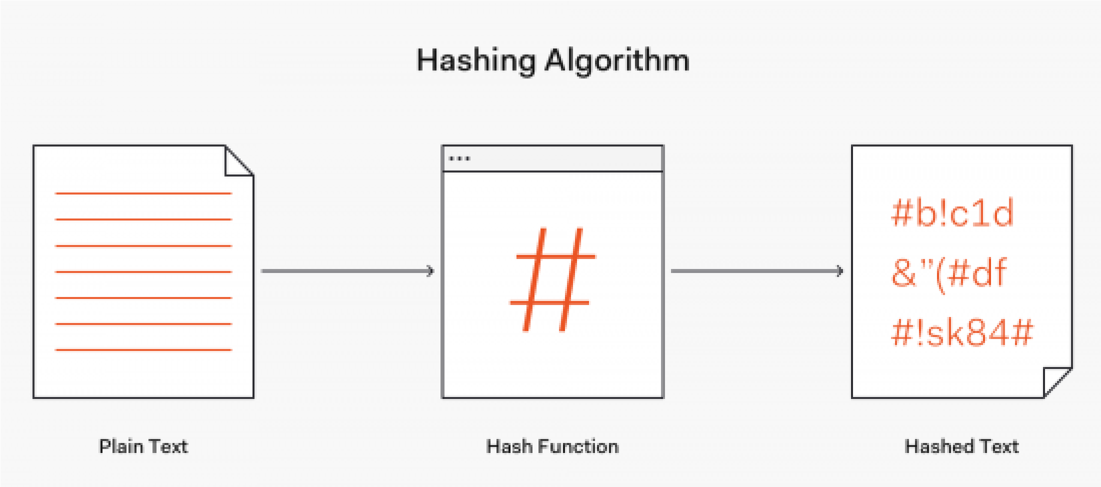
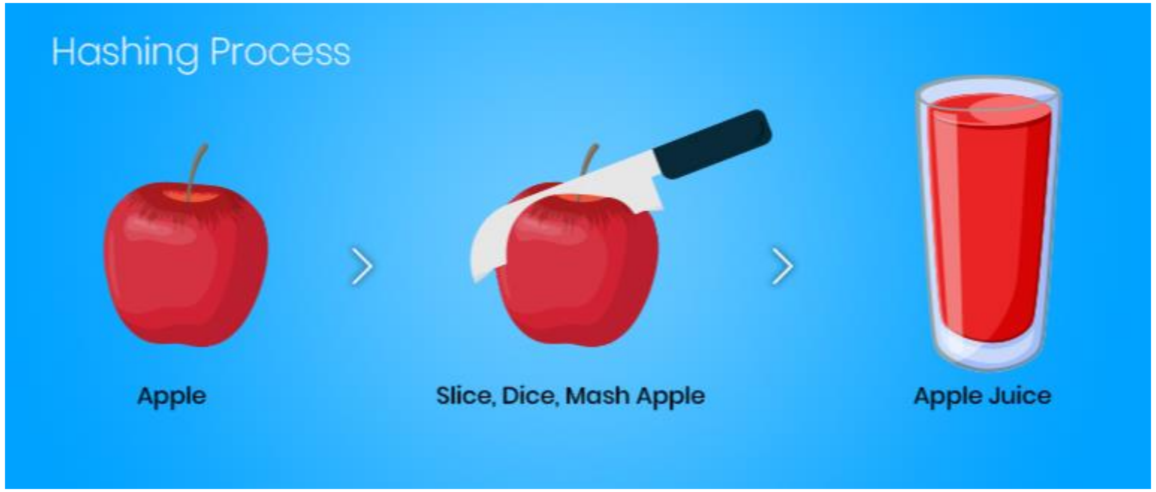
SKIP LISTS AND SETS

School of Artificial Intelligence

PREVIOUSLY ON DS&A

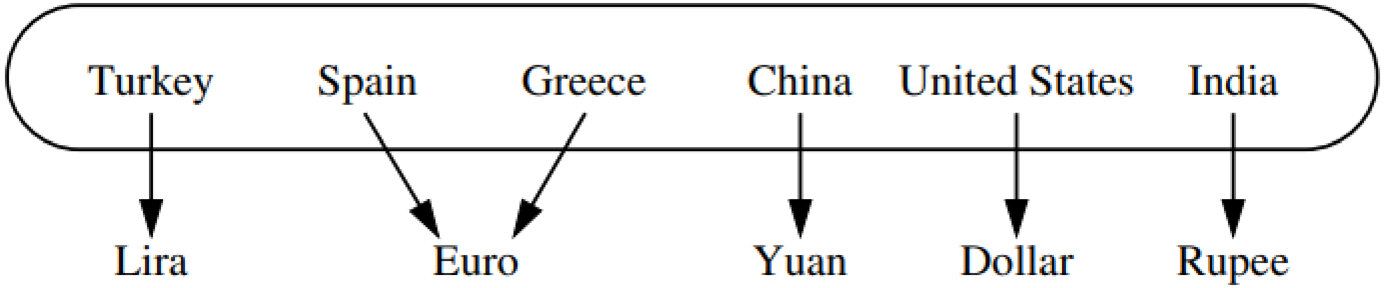
- Maps/dictionaries
- Hash Tables
- Hash Functions
  - Hashing
  - Compressing





MAPS AND DICTIONARIES

- Key->Value pairing
  - Unique association
- Most significant data structure in any programming language
- Often known as **associative arrays** or **maps**
- Keys are (assumed) to be unique, values are not necessarily unique
- Python: dict class
- Use key as ‘index’
- Indices need not be consecutive nor numeric



UNSORTED MAP IMPLEMENTATION

- UnsortedTableMap
- Subclass of MapBase to store key-value pairs in unsorted order in a Python list
- __getitem__: M[k]
- __setitem__: M[k] = v

```
class UnsortedTableMap(MapBase):
  """Map implementation using an unordered list."""

  def __init__(self):
    """Create an empty map."""
    self._table = [ ]

  def __getitem__(self, k):
    """Return value associated with key k (raise KeyE
    for item in self._table:
      if k == item._key:
        return item._value
    raise KeyError('Key Error: ' + repr(k))

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing val
    for item in self._table:
      if k == item._key:
        item._value = v
        return
    # did not find match for key
    self._table.append(self._Item(k,v))
```

HASH TABLES (哈希表)

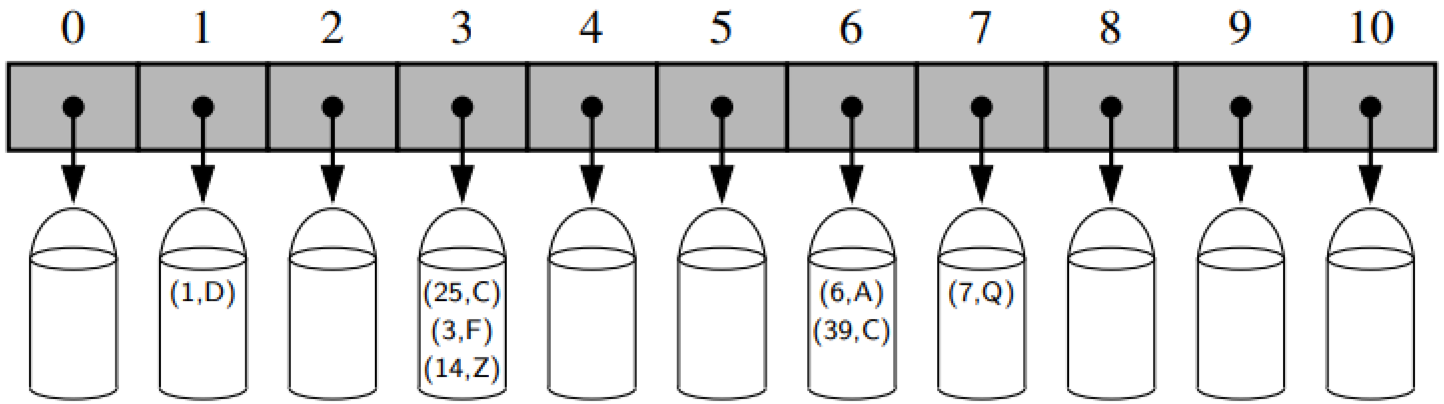
- Most practical data structures for implementing a map
- A map M supports the abstraction of using keys as indices with a syntax such as M[k].
- Assume a map with n items uses integer keys from 0 to N-1 for some N >= n
- We can represent the map like this:

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

- __getitem__, __setitem__ and __delitem__ become O(1)
- Problems?

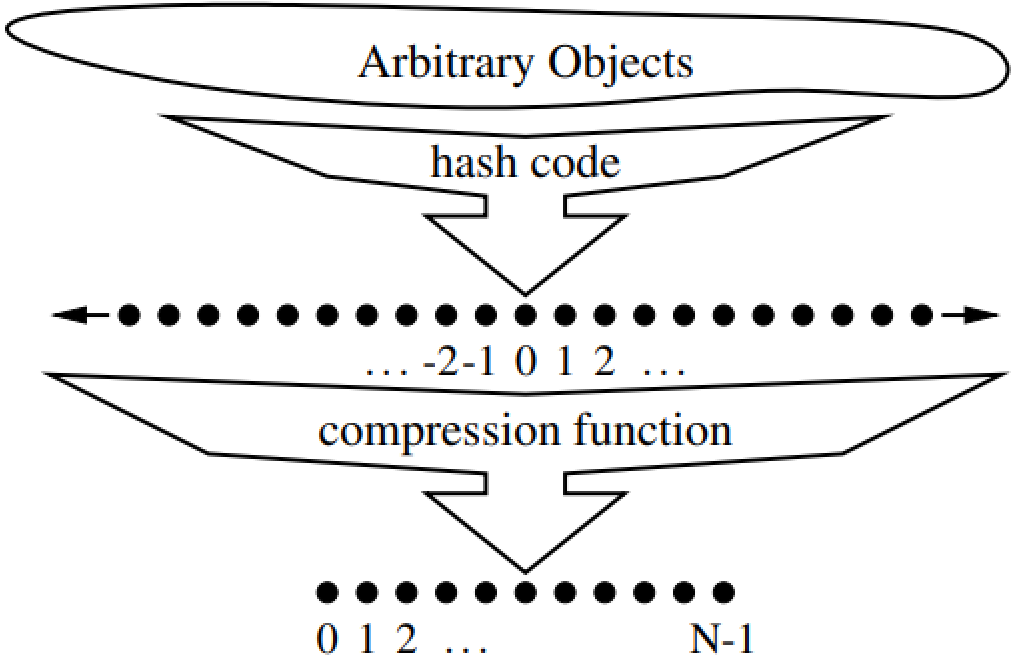
HASH TABLES (哈希表)

- Problems
  - Keys n may not be continuous, therefore an array for the map may have size N >> n
  - A map's key can be other data types, not just integers
- Solution: hash function to map keys to corresponding indices in a table
- Ideally, keys will be distributed in the range from 0 to N-1
- But, there may be two or more distinct keys that get mapped to the same index
- Bucket array



HASH FUNCTION（哈希函数）

- Hash function: h(k)
- **Hashing**: produce a *hash code* that maps a key k to an integer
- **Compressing**: maps the hash code to an integer within a range of indices [0, N-1]
- Why the separation?
  - Independence: hashing is independent of a specific hash table size
  - OO design: hash functions can be overridden



HASH CODES（哈希码）

- Hashing: treating the bit representation as an integer
- For any data type X, its representation in memory can be considered an integer
  - For integer 314, h(314) = 314
  - For floating-point number 3.14, h(3.14) will use its memory representation as an integer
- For type that uses longer than a desired hash code
  - E.g. if we want a 32-bit hash code, if a floating-point number uses a 64-bit representation
  - Approaches: take the first/last 32 bit; add the first/last 32 bit, take exclusive-or of the first/last 32 bits

POLYNOMIAL HASH CODES（多项式哈希码）

- Summation and exclusive-or: NOT good choices for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{n-1})$, where the order of x is significant.
  - E.g. 16-bit hash code for a character string s that sums the Unicode values of the characters in s.
  - “temp01” and “temp10” produces the same hash code.
  - “stop”, “tops”, “pots”, and “spot” produces the same hash code
- A more complicated hashing function is needed, such as

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}.$$

- This hash code is called a polynomial hash code

POLYNOMIAL HASH CODES （多项式哈希码）

- Polynomial – to spread out the influence of each component across the resulting hash code
- For constant a, its polynomial value will periodically overflow the bits used for an integer, but is often ignored
- Therefore a should have nonzero, low-order bits.
- 33, 37, 39, 41 are good choices for a when working with character strings (English)
  - 50,000 English words formed as the union of the word lists for different version of Unix, using a = 33, 37, 39 or 41 produces less than 7 collisions

CYCLIC SHIFT HASH CODES

- Replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits
- 5-bit cyclic shift:
- 00111101100101101010100010101000 to 10110010110101010001010100000111
- Table: comparison of collision behavior for the cyclic-shift hash code to a list of 230,000 English words

```
def hash_code(s):
  mask = (1 << 32) - 1
  h = 0
  for character in s:
    h = (h << 5 & mask) | (h >> 27)
    h += ord(character)
  return h
```

	Collisions	
Shift	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

COMPRESSION FUNCTIONS

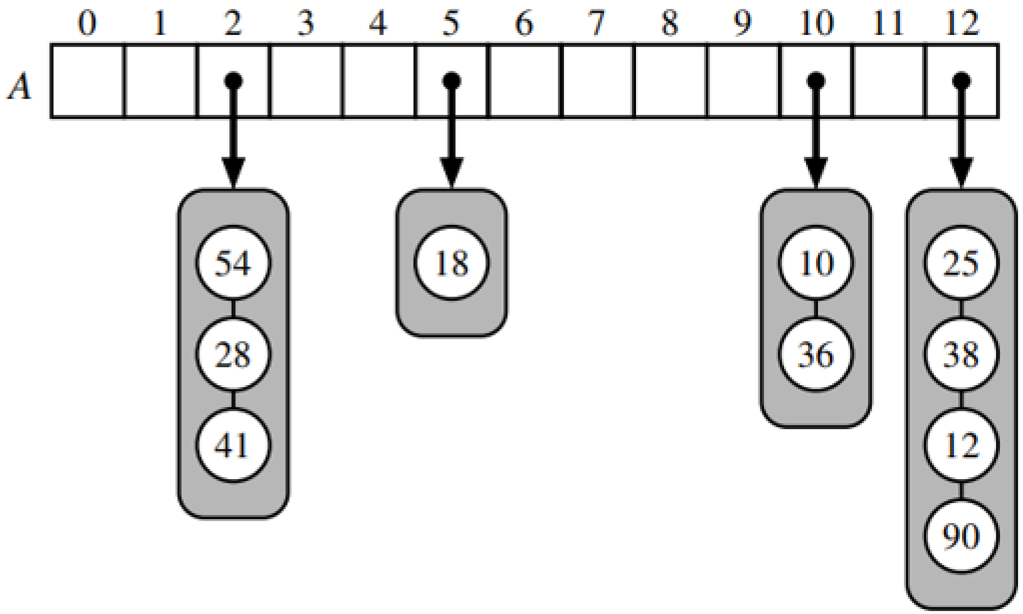
- The hash code for a key *k* may not be suitable for use in a bucket array
- It be negative or may exceed the capacity of the bucket array
- Therefore an additional computation is needed to map the integer into the range [0, N-1] – the **compression function**
- Division method
  - **i mod N**, N = size of the bucket array
  - Choice of N: often prefer prime numbers
  - {200, 205, 210, 215, 220, …, 600} into a bucket array of size 100
  - {200, 205, 210, 215, 220, …, 600} into a bucket array of size 101

COMPRESSION FUNCTIONS

- MAD (Multiply Add and Divide) method
  - **((ai+b) mod p) mod N**, N = size of the bucket array, p is a prime number larger than N, a and b are integers chosen at random from [0, p-1], with a > 0
  - Finding p: in polynomial time
  - Worst case keys k1 != k2, Pr (h(k1) == h(k2)) = 1/N
- Multiplication method
  - h(k) = ((a.k) mod 2^w) >> (w-r), w = w bits computer, bucket array size N = 2^r
  - A better be odd, and should not be close to powers of 2

COLLISION HANDLING

- Separate chaining
- Operations on an individual index (bucket) take time proportional to the size of the bucket
- “Good” hash function:
    - expected size of a bucket: n/N
    - n = number of items in the map
    - N = capacity of the bucket array
    - Core map operations run in O(ceiling(n/N))
- Load factor（负载因子）: λ = n/N
- When λ is O(1), operations on the hash table run in O(1)

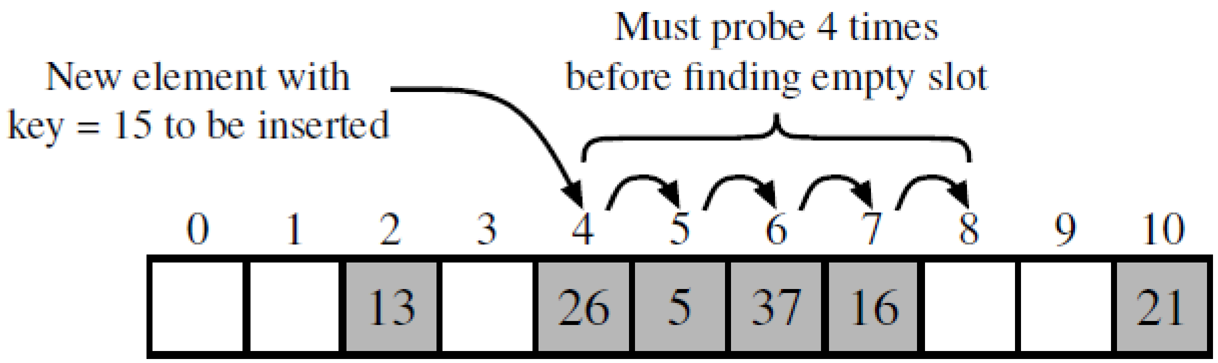


COLLISION HANDLING

- Separate chaining （分离链表）:
  - Advantage: simple implementation
  - Disadvantage: relies on auxiliary data structure - list to hold items with colliding keys
- Alternative approach: open addressing （开放寻址）
  - Always store each item directly in a table slot
  - No auxiliary structures are used
  - Load factor is ways at most 1 and items are stored directly in the cells of the bucket array

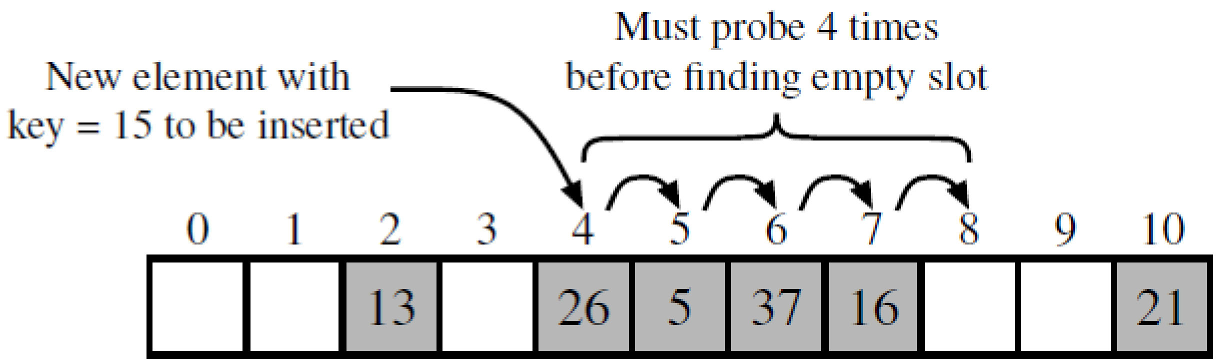
LINEAR PROBING （线性探索）

- Linear probing
- Insert an item (k, v) into a bucket A[j]
- If a[j] is occupied, j = h(k), then try A[(j+1) mod N]
- If A[(j+1) mod N] is occupied, try A[(j+2) mod N], so on
- Need to change the implementation of funcgtions such as __getitem__, __setitem__, __delitem__



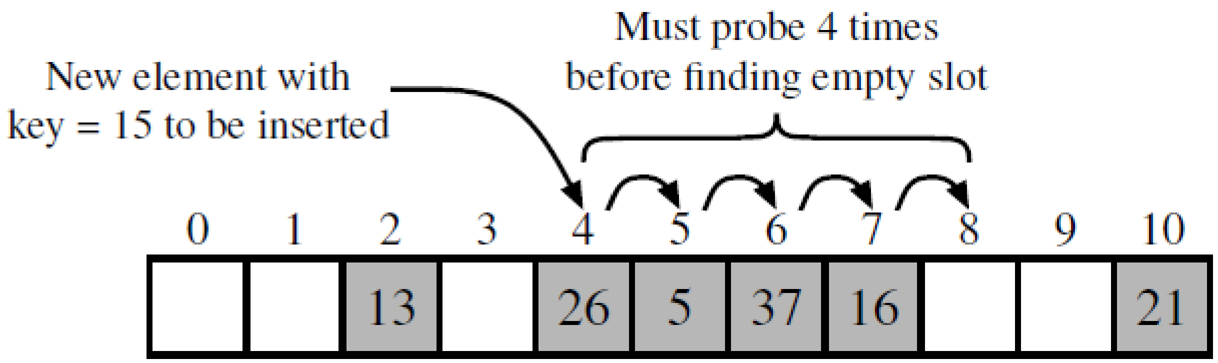
QUADRATIC PROBING （二次探索）

- Iteratively tries A[h(k) + f(i) mod N] for I = 0, 1, 2, …, f(i) = i^2
- Spreads the probing distance over the length N
- Deletion – same strategy as linear probing
- Problems again: secondary clustering （二次聚集）
- When the bucket array is half full, or N is not a prime, quadratic probing does not guarantee to find an empty slot



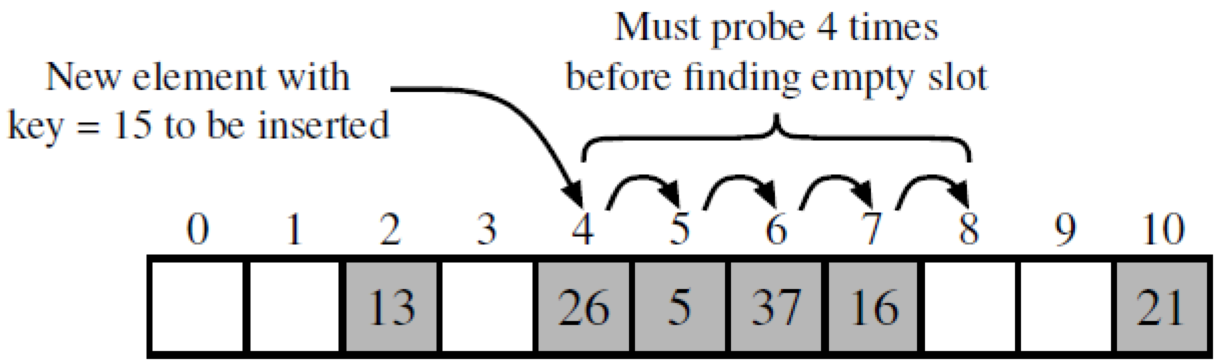
DOUBLE HASHING （二次哈希）

- Secondary hash function h'
- If h maps some key k to a bucket A[h(k)] that is occupied, try A[h(k) + f(i) mod N], for i = 1,2,3, ...
- f(i) = i*h'(k)
- h'(k) cannot be 0
- h'(k) = q – (k mod q), q, N are prime numbers and q < N



ADDITIONAL OPEN ADDRESSING

- A[h(k) + f(i) mod N] where f(i) produces a pseudo-random number
- Repeatable random number

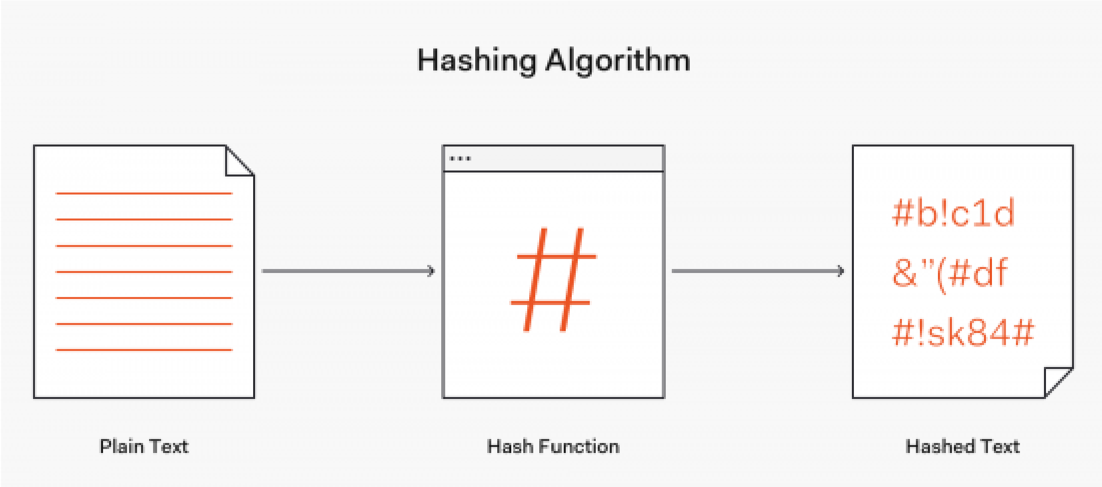


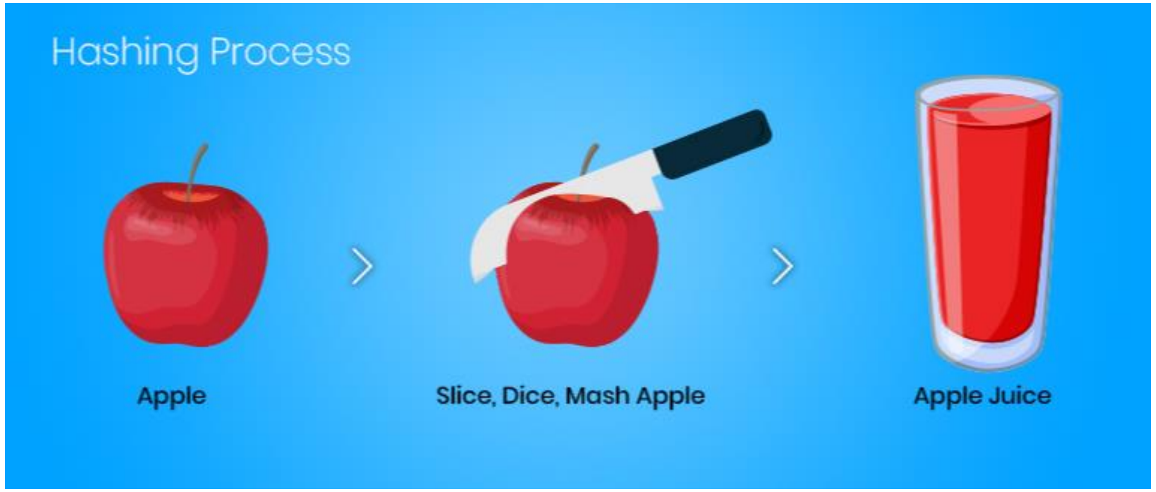
LOAD FACTORS AND REHASING

- Load factor（负载因子）: $\lambda = n/N$ should be kept below 1
- Separate chaining: $\lambda \to 1$ the probability of a collision increases greatly
  - $\lambda < 0.9$ for separate chaining
- Open addressing
  - When λ grows beyond 0.5 and approaches 1, clusters start to show
  - Linear probing: $\lambda < 0.5$
  - Other open addressing: $\lambda < 2/3$
- What happens if an insertion causes the load factor to go beyond 0.5 for linear probing and 2/3 for other open addressing means?
- **Rehashing**: resize the table + reinsert all objects into new table
  - Resize: how?
  - new compression function -> why?

THIS LECTURE

- Skip Lists
- Sets
- Multisets
- Multimaps



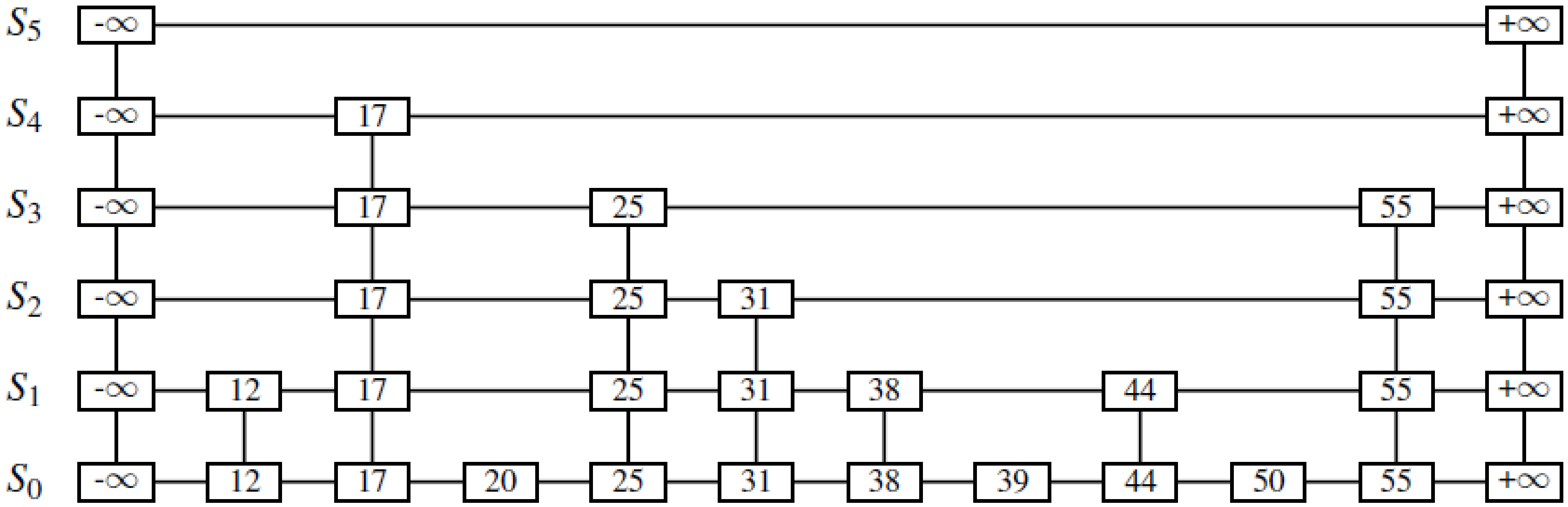


ARRAYS AND LINKEDLISTS

- Arrays vs. Linkedlists
- Search
  - Array(sorted): O(log n)
  - Linkedlist: O(n)
- Update
  - Array(sorted): O(n)
  - Linkedlist: O(1) as long as we know where to update

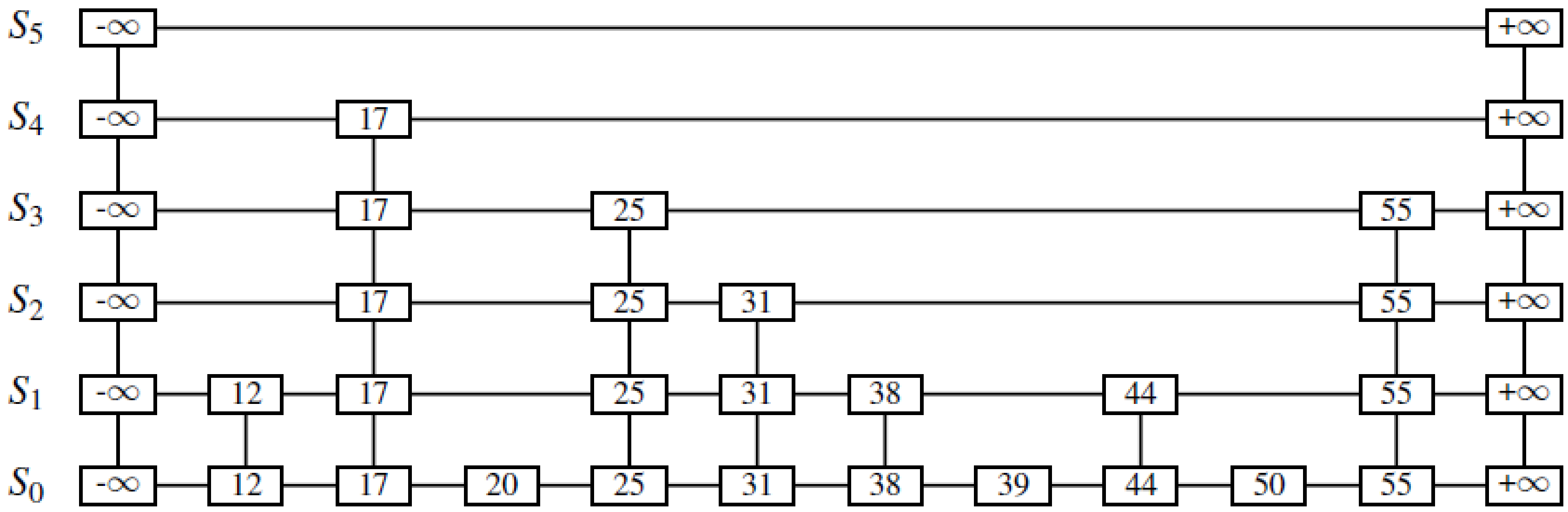
SKIP LISTS

- Skip list S for a map M:
- Series of lists $\{S_0, S_1, \ldots, S_h\}$,
- Each list S_i stores a subset of the items of M sorted by increasing keys
- Sentinel keys -infinity and +infinity



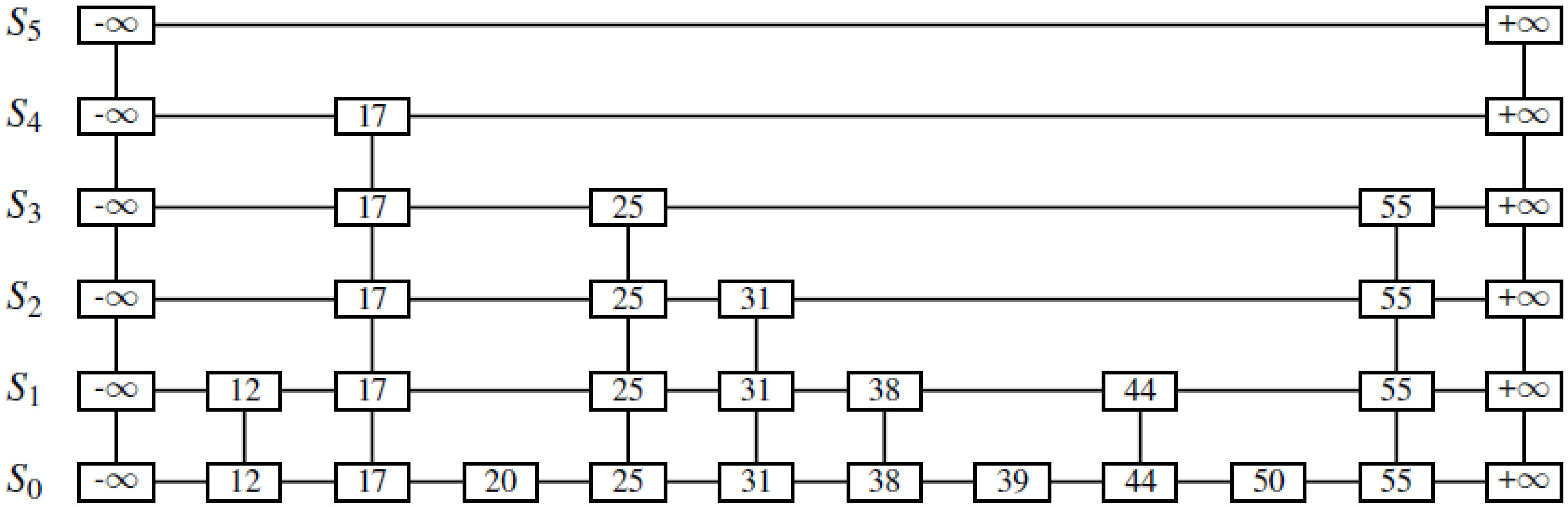
SKIP LISTS

- List S_0 contains every item of the map M (plus the -fininity and +infinity)
- For i = 1, …, h-1, list S_i contains a randomly generated subset of the items in list S_{i-1}
- List S_h contains only –infinity and +infinity



SKIP LISTS

- h: height of skip list S
- S_{i+1} contains more or less alternate items of S_i
- Items in S_{i+1} are chosen at random from the items in S_i by picking each item from S_i to also be in S_{i+1} with probability ½
- S_0: n items
- S_1: n/2 items
- S_i: $n/2^i$ items
- h: about log n



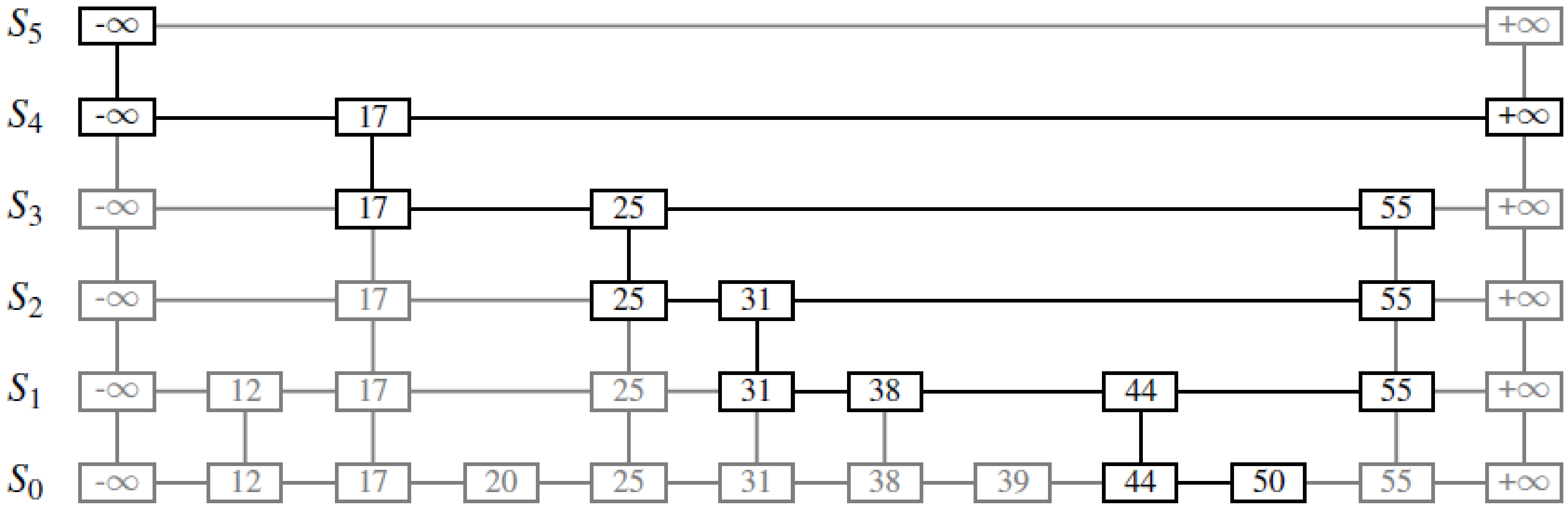
SKIP LISTS

- What can be observed straight away?
- Search and update: O(log n) **on average**
- Running time for skip lists: **expected** other than **worst-case**
- Average time: depends on the use of a random-number generator when elements are inserted into the skip list.
- Skip lists: horizontal **levels** and vertical **towers** （using positions）
- Each level is a list S_i, each tower contains positions storing the same item across lists
- Traversal operations: **next(p)**, **prev(p)**, **below(p) and above(p)**
  - Returns None if the position requested does not exist
  - Time: O(1) for each operation

SKIP LISTS

- Search: SkipSearch(k)
- Key k
- Position p to point to the top left position – **start position**

1. If S.below(p) is None, search is done, obtained key at position p <= k. Otherwise p = S.below(p)
2. Move p forward until it is at the rightmost position, such that k(p) <= k
3. Return to step 1



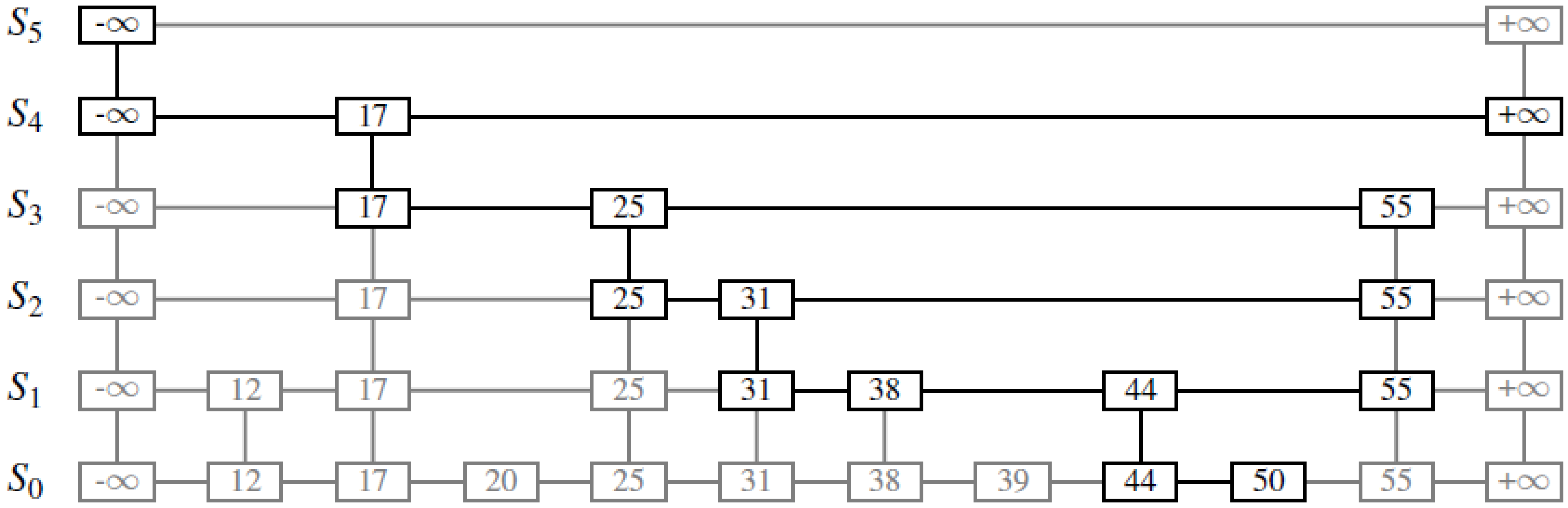
SKIP LISTS

Algorithm SkipSearch(k):

Input: A search key k

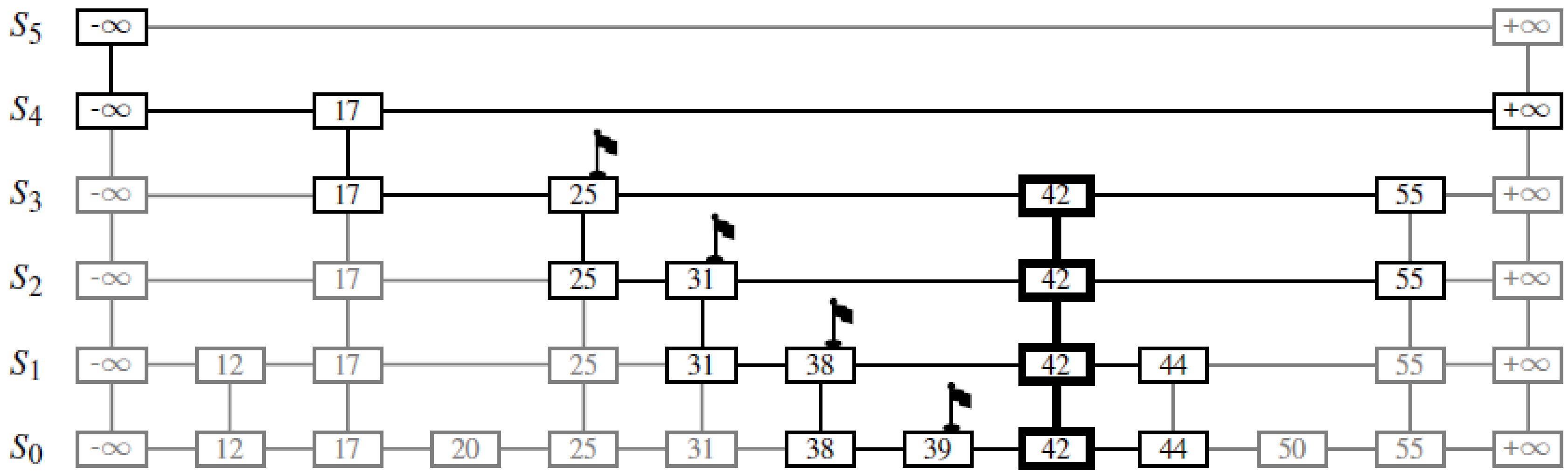
Output: Position p in the bottom list S_0 with the largest key such that $\text{key}(p) \leq k$

```
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.
```



SKIP LISTS

- Insertion
- Begins with SkipSearch(k) to find a position p
- If key(p) == k, update (k, v)
- Otherwise, need to create a new **tower**
- Insert (k, v) immediately after position p within S_0
- Randomise to decide the height of the tower



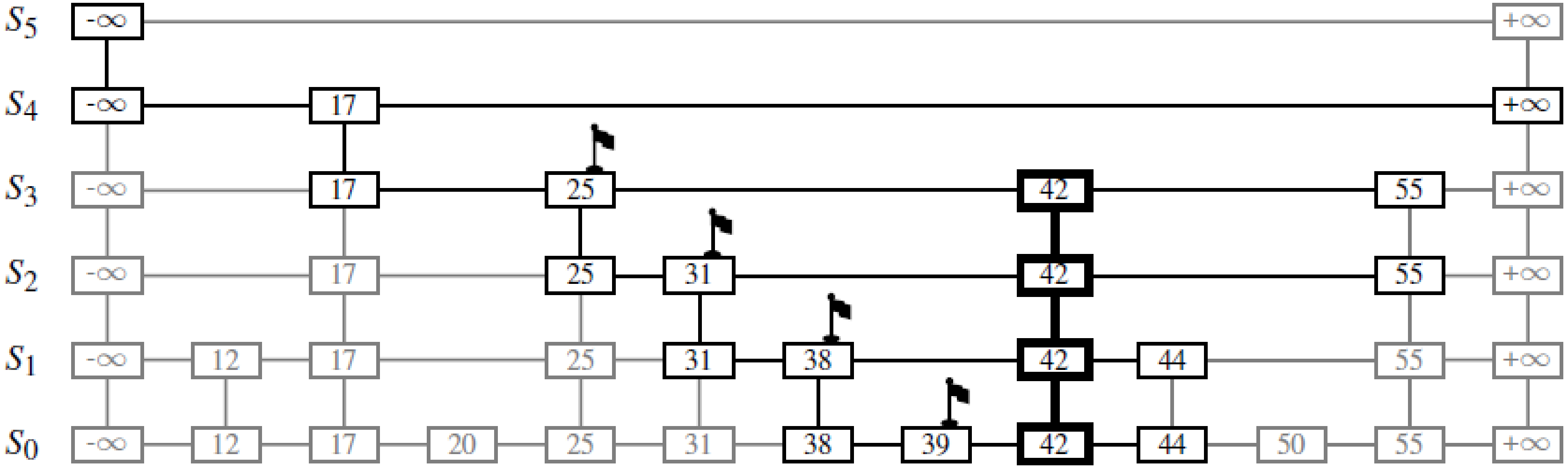
SKIP LISTS

Algorithm SkipInsert(k,v):

Input: Key k and value v

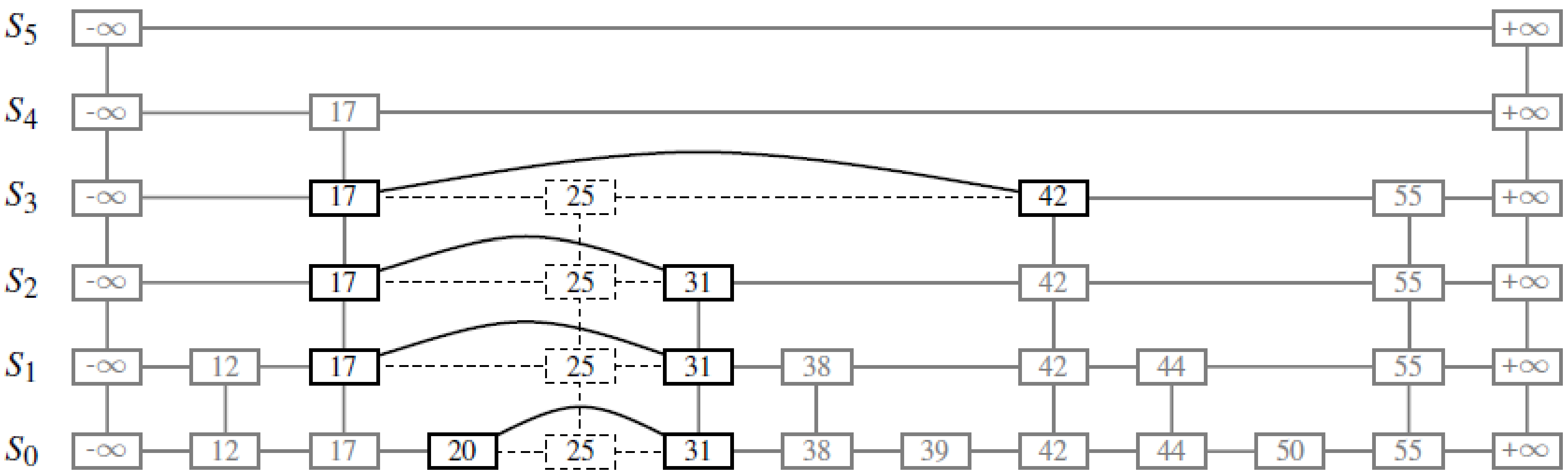
Output: Topmost position of the item inserted in the skip list

```
p = SkipSearch(k)
q = None                       {q will represent top node in n
i = −1
repeat
   i = i + 1
   if i ≥ h  then
      h = h + 1                              {add a new leve
      t = next(s)
      s = insertAfterAbove(None, s, (−∞, None))       {grov
      insertAfterAbove(s, t, (+∞, None))            {grow
   while above(p) is None do
      p = prev(p)
   p = above(p)
   q = insertAfterAbove(p, q, (k, v))
until coinFlip() == tails
n = n + 1
return q
```



SKIP LISTS

- Removal
- Perform SkipSearch(k) to obtain p
- If p is illegal, raise error
- I p is legal, remove the entire tower
- Re-establish links between the horizontal neighbours of each removed position



SKIP LISTS

- Expected value of the height h of S with n entries
- Probability that a given entry has a tower of height i >= 1 equals to the probability of getting i "heads" when flipping a coin: $1/2^i$
- Probability that level i has at least one position is at most: $P_i \leq \frac{n}{2^i},$
- Probability height h of S is larger than i: equal to the probability that level i has at least one position (it is no more than P_i)

$$P_{3\log n} \leq \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}.$$

- Given a constant c > 1, h is larger than c log n with probability at most $1/n^{c-1}$
- The probability that h is smaller than c log n is at least $1-1/n^{c-1}$
- Thus, the height h of S is O(log n)

Algorithm SkipSearch(k):

Input: A search key k

Output: Position p in the bottom list S_0 with the largest key such that 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.
```

- Search time
- 2 nested loops
  - 1st loop: scan forward on a level of S
  - 2nd loop: drops down to the next level
- Height h of S is O(log n): number of drop downs: O(log n)
- Scan-forward steps:
  - n_i: number of keys examined scanning forward at level i: constant
- Search time: O(log n)
- Similar for insertion and removal

SKIP LISTS

- Space usage
- Expected number of positions at level i: $n/2^i$
- Expected total number of positions in S: $$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i}.$$
- Geometric summations:
- Space requirement: O(n)

Operation	Running Time
len(M)	$O(1)$
k **in** M	$O(\log n)$ expected
M[k] = v	$O(\log n)$ expected
del M[k]	$O(\log n)$ expected
M.find_min(), M.find_max()	$O(1)$
M.find_lt(k), M.find_gt(k) M.find_le(k), M.find_ge(k)	$O(\log n)$ expected
M.find_range(start, stop)	$O(s + \log n)$ expected, with s items reported
iter(M), reversed(M)	$O(n)$

SETS, MULTISETS AND MULTIMAPS

- Set: unordered collection of elements, with no duplicates
- Multiset (bag): set-like container that allows duplicates
- Multimap: similar to a traditional map, same key can be mapped to multiple values

SETS, MULTISETS AND MULTIMAPS

- Set ADT:
- S.add(e): add an element e to the set
- S.discard(e): remove element e from the set
- e in S: returns True if set contains element e
- S.remove(e): remove e from S, error if e is not in S
- S.pop(): return an arbitrary element from the set
- S.clear(): remove all elements from the set

SETS, MULTISETS AND MULTIMAPS

- S == T:true if S and T have identical contents
- S != T: true if S and T are not equivalent
- S <= T: true if S is a subset of T
- S < T: true if S is a proper subset of T
- S >= T: true if T is a subset of S
- S > T: true if T is a proper subset of S
- S.isdisjoint(T): true if S and T have no common elements
- S | T: union
- S&T: intersection
- S^T: symmetric difference of S and T

IMPLEMENTATION

- In the practical

QUIZ FOR THIS WEEK

- A business man hires a goldsmith to do a job
- The goldsmith dose the job in a week
- The business man pays him a gold bar for the week
- But the goldsmith wants payment everyday
- How can the business man pay the goldsmith by cutting the gold bar exactly twice?

THANKS

See you in the next session!