CSE214 Data Structures

Maps

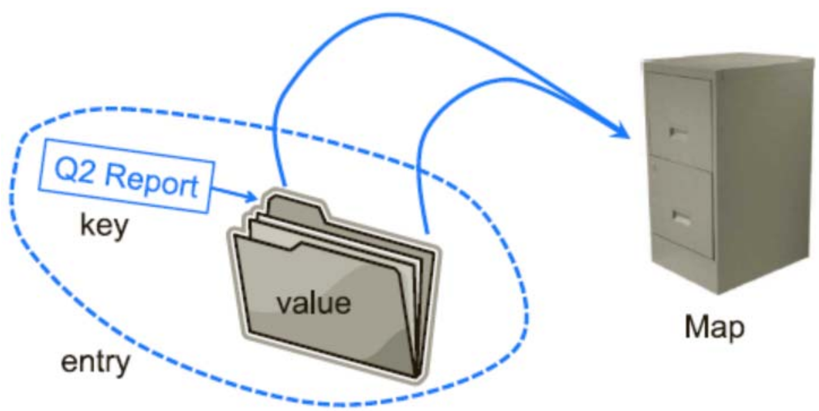
YoungMin Kwon

Map Abstract Data Type

- Map
  - Map is an abstract data type for efficiently storing and retrieving values based on unique search keys
  - Maps store key-value pairs (k, v) called entries
  - Maps are also known as associative arrays
    - Keys serve somewhat like indexes into the map

Map Abstract Data Type

- Map examples



  - Keys are labels
  - Values are folders
  - Map is the file cabinet

Map Abstract Data Type

- Map examples
  - URL (http://datastructures.net) and the page contents
  - A student ID and the student's record
  - DNS maps host name (www.suny.ac.kr) to IP address 221.143.20.101
  - In computer graphics, color name (Crimson) to RGB (0xdc, 0x14, 0x3c)

Map Abstract Data Type

size( ): Returns the number of entries in M.

isEmpty( ): Returns a boolean indicating whether M is empty.

get(k): Returns the value v associated with key k, if such an entry exists; otherwise returns null.

put(k, v): If M does not have an entry with key equal to k, then adds entry (k, v) to M and returns null; else, replaces with v the existing value of the entry with key equal to k and returns the old value.

remove(k): Removes from M the entry with key equal to k, and returns its value; if M has no such entry, then returns null.

keySet( ): Returns an iterable collection containing all the keys stored in M.

values( ): Returns an iterable collection containing all the *values* of entries stored in M (with repetition if multiple keys map to the same value).

entrySet( ): Returns an iterable collection containing all the key-value entries in M.

Map Abstract Data Type (Operations)

Method	*Return Value*	*Map*
isEmpty()	true	$\{\}$
put$(5,A)$	null	$\{(5,A)\}$
put$(7,B)$	null	$\{(5,A),(7,B)\}$
put$(2,C)$	null	$\{(5,A),(7,B),(2,C)\}$
put$(8,D)$	null	$\{(5,A),(7,B),(2,C),(8,D)\}$
put$(2,E)$	C	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(7)	B	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(4)	null	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(2)	E	$\{(5,A),(7,B),(2,E),(8,D)\}$
size()	4	$\{(5,A),(7,B),(2,E),(8,D)\}$
remove(5)	A	$\{(7,B),(2,E),(8,D)\}$
remove(2)	E	$\{(7,B),(8,D)\}$
get(2)	null	$\{(7,B),(8,D)\}$
remove(2)	null	$\{(7,B),(8,D)\}$
isEmpty()	false	$\{(7,B),(8,D)\}$
entrySet()	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
keySet()	$\{7,8\}$	$\{(7,B),(8,D)\}$
values()	$\{B,D\}$	$\{(7,B),(8,D)\}$

Java Interface for Map ADT

```java
public interface Map<K, V> {
    int size();
    boolean isEmpty();
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    Iterable<K> keys();
    Iterable<V> values();
    Iterable<Entry<K,V>> entries();
}

public interface Entry<K, V> {
    public K key();
    public V value();
    public void setKey(K key);
    public void setValue(V value);
}
```

Application: Word Counter

```
/** A program that counts words in a document, printing the most frequent. */
public class WordCount {
  public static void main(String[ ] args) {
    Map<String,Integer> freq = new ChainHashMap<>( ); // or any concrete map
    // scan input for words, using all nonletters as delimiters
    Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
    while (doc.hasNext( )) {
      String word = doc.next( ).toLowerCase( ); // convert next word to lowercase
      Integer count = freq.get(word); // get the previous count for this word
      if (count == null)
        count = 0;                    // if not in map, previous count is zero
      freq.put(word, 1 + count);      // (re)assign new count for this word
    }
    int maxCount = 0;
    String maxWord = "no word";
    for (Entry<String,Integer> ent : freq.entrySet( )) // find max-count word
      if (ent.getValue( ) > maxCount) {
        maxWord = ent.getKey( );
        maxCount = ent.getValue( );
      }
    System.out.print("The most frequent word is '" + maxWord);
    System.out.println("' with " + maxCount + " occurrences.");
  }
}
```

Hashing

- Hash table
  - One of the most efficient data structures for implementing a map (also used most in practice)

  - Intuitive example:
    - Keys are integers
    - Lookup table is an array of length N

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

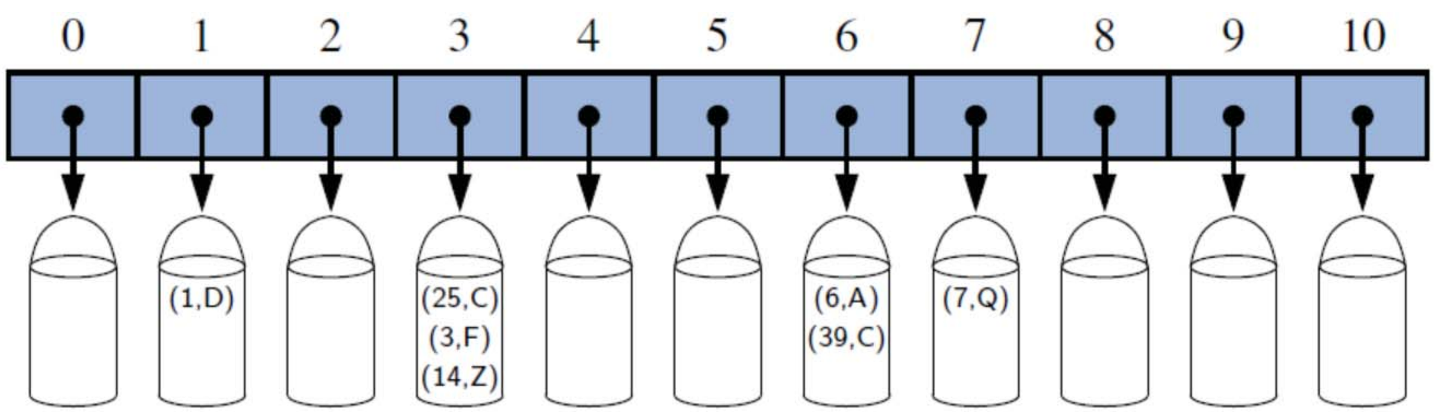
    - A table of length 11 containing (1,D), (3,Z), (6,C), and (7,Q).

Hashing

- Two challenges
  - We may NOT wish to devote an array of length N when $N \gg n$
  - Map's key may not be an integer in general

Hashing

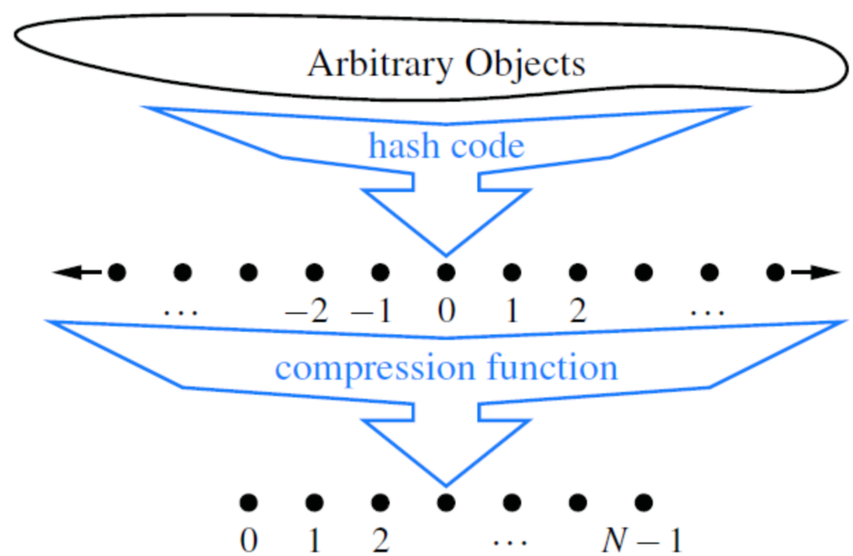
- Hash function
  - maps a general key to 0 ~ N-1
    - N is the capacity of bucket array
  - Two or more distinct keys can be mapped to the same index ⇒ bucket array



Hashing

- Hash function
  - Use the hash function value h(k) as an index
  - Store an entry (k, v) in a bucket A[h(k)]
  - Collision: two or more entries are mapped to the same bucket in A

- Two parts of a hash function
  - Hash code: maps a key k to an integer
  - Compression function: maps the hash code to [0, N-1]
  - Hash code is independent to the bucket array size



Hash Codes

- Treating bit representations as an integer
  - byte, short, int, char, float ⇒ int

- How about long, double: larger than 32bit
  - Ignore the half: can collide easily
  - Combine them: *add* or *xor* the two halves
  - General objects of any size (x_0, x_1, …, x_{n-1})
    - $x_0 + x_1 + \ldots + x_{n-1}$
    - $x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$

Hash Codes

- Polynomial hash codes
  - Using *add, xor* for $(x_0, x_1, \ldots, x_{n-1})$: easy to collide when the order of x_i's is significant
    - E.g. "temp01", and "temp10"
    - "stop", "tops", "pots", and "spot"

  - Polynomial hash code (for $a \neq 1$)
    - $x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-2} a + x_{n-1}$
    - $x_{n-1} + a(x_{n-2} + a(x_{n-3} + \ldots + a(x_2 + a(x_1 + a x_0)) \ldots))$
    - Ignore the overflows
    - 3, 37, 39, 41 for $a \Rightarrow$ fewer than 7 collision for 50,000 English words

Hash Codes

- Cyclic-shift hash codes
  - A variant of the polynomial hash code
  - Multiplication by a ⇒ cyclic shift of a partial sum
    - 0011 1101 1001 0110 1010 1000 1010 1000 ⇒ 1011 0010 1101 0101 0001 0101 0000 0111

```
static int hashCode(String s) {
  int h=0;
  for (int i=0; i<s.length(); i++) {
    h = (h << 5) | (h >>> 27); // 5-bit cyclic shift of the running sum
    h += (int) s.charAt(i);    // add in next character
  }
  return h;
}
```

Shift right, fill zero

Hash Code in Java

- Object class includes hashCode() method
  - A 32 bit integer for the object's memory address

- Issue with equals
  - Equivalent keys should have the same hash code
    - Otherwise, map may not function correctly
  - If x.equals(y), then x.hashCode() == y.hashCode()

Compression Functions

- Compression function
  - Maps the hash code into the range [0, N-1]
- The division method
  - *i mod N*
  - Making N a *prime number* helps spread out the distribution of the hashed values
    - Hash codes of {200, 205, 210, 215, ..., 595} will have 3 collisions when N is 100
    - No collisions when N is 101

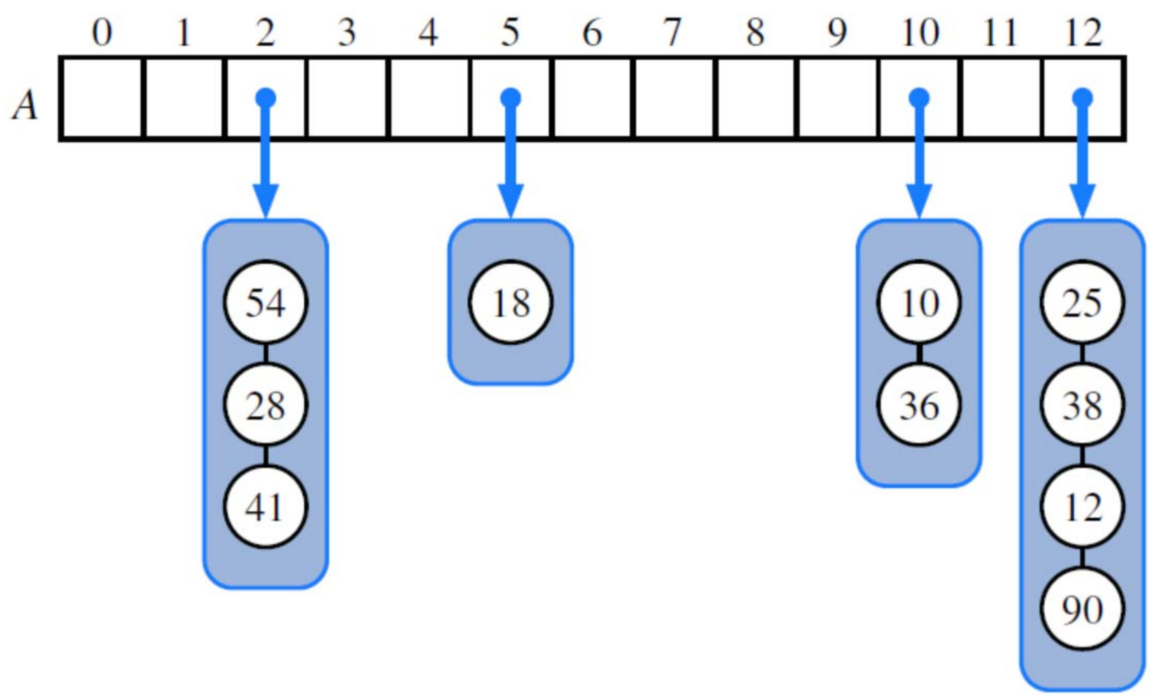
Compression Functions

- The MAD method
  - Multiply-Add-and-Divide (MAD)
  - $[\,(a \cdot i + b) \bmod p\,] \bmod N$
    - N is the size of the bucket array
    - p is a prime number larger than N
    - a and b are integers chosen from [0, p-1] with $a > 0$

Collision Handling Schemes

Separate Chaining

- Separate chaining
  - Have each bucket A[j] store its own container for all entries (k, v) with h(k) = j



Collision Handling Schemes
Separate Chaining

- Load factor
  - Assuming the use of a good hash function
  - Expected size of a bucket is n / N
    - n: number of entries in the map
    - N: the number of buckets
  - The search in a chain will take $O(n / N)$
  - The ratio $\lambda = n / N$ is called the load factor

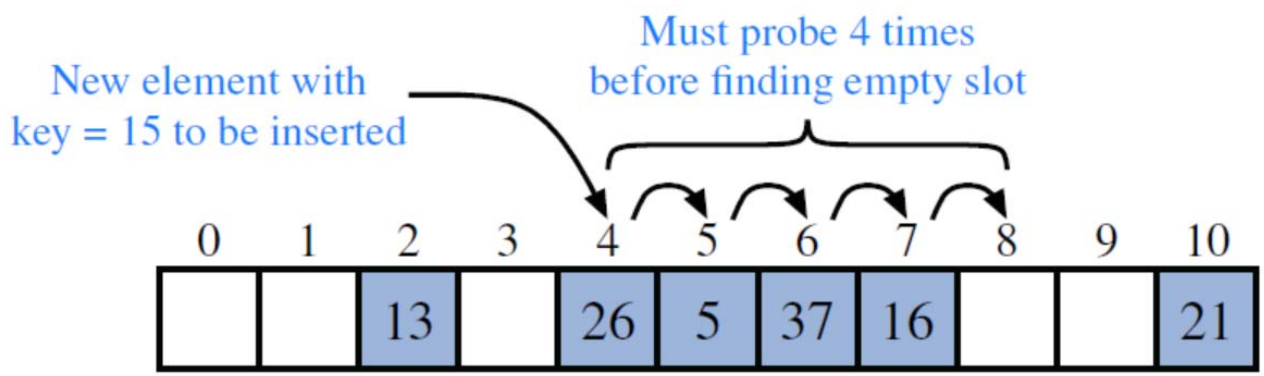
Collision Handling Schemes
Open Addressing

- Separate chaining
  - Requires an auxiliary data structure (bucket)

- Open addressing
  - Stores entries directly in a table slot without using an auxiliary data structure
  - Several variants of this approach is collectively referred to as open addressing

Collision Handling Schemes
Open Addressing

- Linear probing
  - Let *h*(*k*) = *j* for an entry (k, v).
  - If A[*j*] is occupied, try A[(*j+1*) *mod N*], A[(*j+2*) *mod N*], and so on



Collision Handling Schemes

Open Addressing

- Linear proving
  - To delete: mark the entry with a defunct object
  - To search: from A[h(k)], look for an entry with k until an empty slot is encountered while skipping the defunct objects
  - Disadvantage: it tends to cluster entries into a contiguous runs

Collision Handling Schemes
Open Addressing

- Double hashing
  - Use a secondary hash function h'(k)
  - If A[h(k)] is occupied try A[(h(k) + f(i)) mod N], for i = 1, 2, 3, …
    where $f(i) = i \cdot h'(k)$

- Another approach
  - On collision, try A[(h(k) + f(i)) mod N], where f(i) is based on a pseudo random number generator

Rehashing

- Efficiency of hash table
  - It depends on keeping the load factor $\lambda = n/N$ low
  - Separate chaining: large λ increases the entries in a bucket
  - Open addressing: large λ grows the cluster of entries
- Rehashing
  - If λ go above a specified threshold, resize the table reapply the compression function to each entry
  - New table size: a prime number about the double of the current the size
    - Amortization: *put is an $O(1)$ operation*

Efficiency of Hash Tables

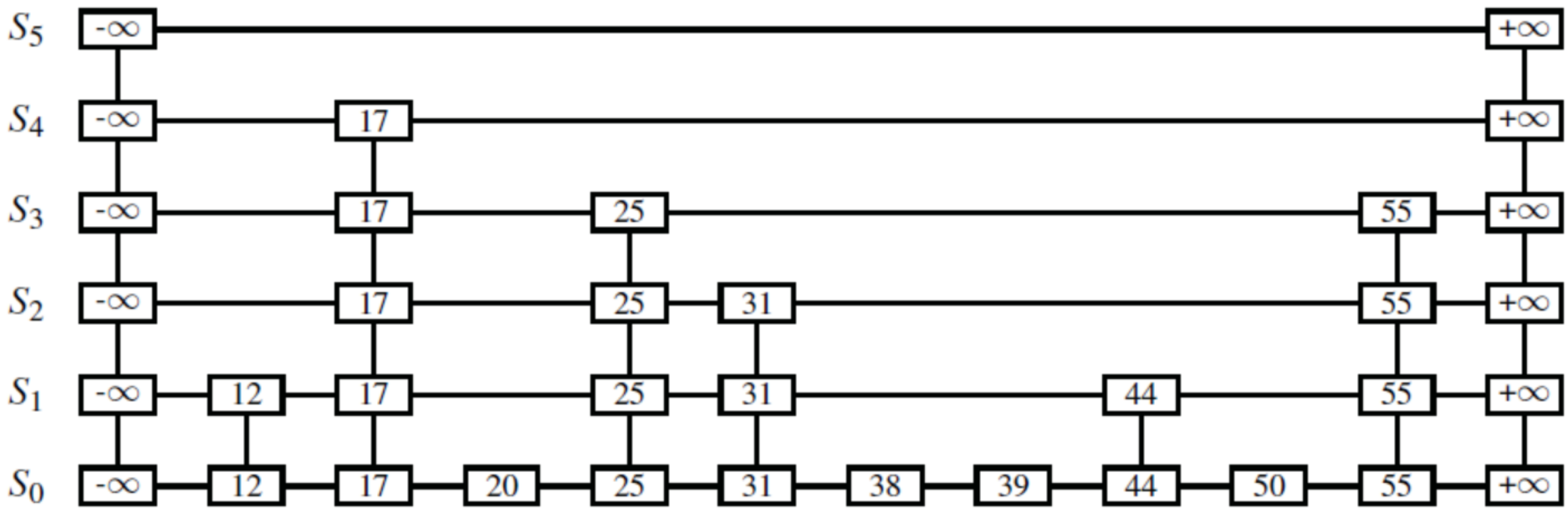
Method	**Hash Table**	
	expected	worst case
get	$O(1)$	$O(n)$
put	$O(1)$	$O(n)$
remove	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(n)$

Skip Lists

- A *skip list* S for a map M
  - Consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$
  - Each list S_i stores a subset of entries of M sorted by keys and two sentinel keys denoted by $-\infty$ and $+\infty$

- Lists in S satisfy
  - S_0 contains every entry of M plus $-\infty$ and $+\infty$
  - S_i contains a randomly generated subset of S_{i-1} plus $-\infty$ and $+\infty$ for i = 1, …, h-1,
  - S_h contains only $-\infty$ and $+\infty$

Skip Lists



- Intuitively, S_{i+1} contains roughly alternate entries of S_i
- Randomization: for each entry in S_i, we flip a coin and add it to S_{i+1} if head comes up

Skip Lists

- Representation
  - Two dimensional collection of positions (levels and towers)
    - Each *level* is a list S_i
    - Each *tower* contains positions storing the same entry

- Positions in a skip list

next(p): Returns the position following p on the same level.

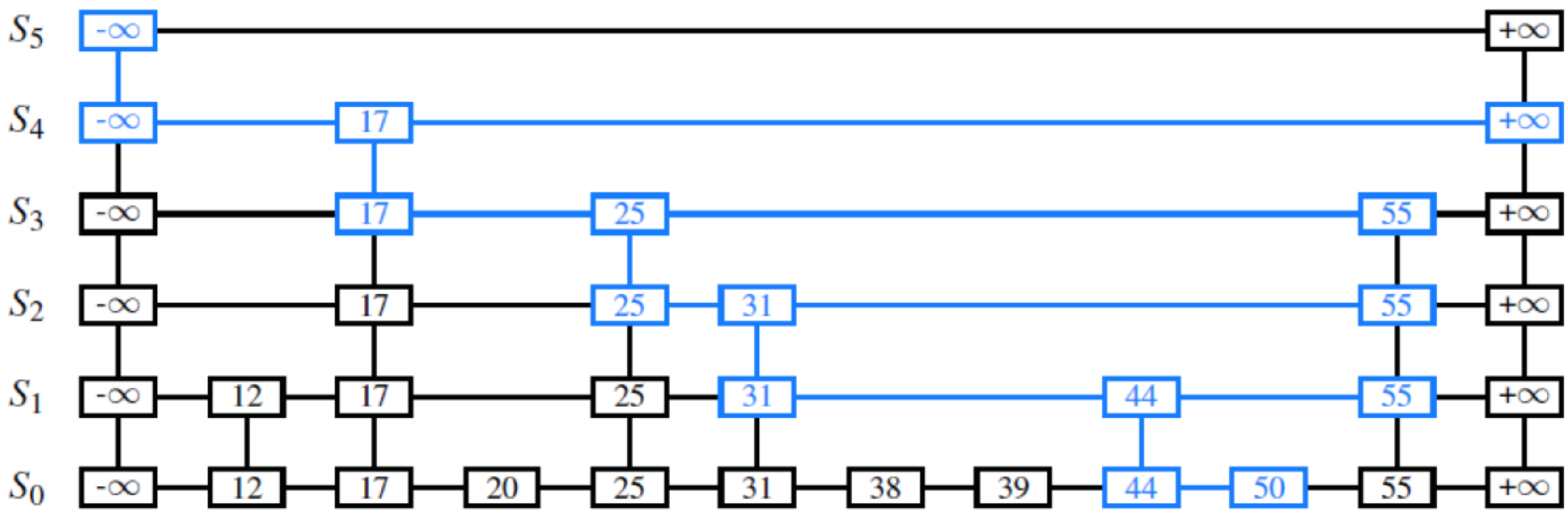
prev(p): Returns the position preceding p on the same level.

above(p): Returns the position above p in the same tower.

below(p): Returns the position below p in the same tower.

Skip Lists

- Search
  - E.g. searching for 50



Skip Lists

- Search

Algorithm SkipSearch(k):

Input: A search key k

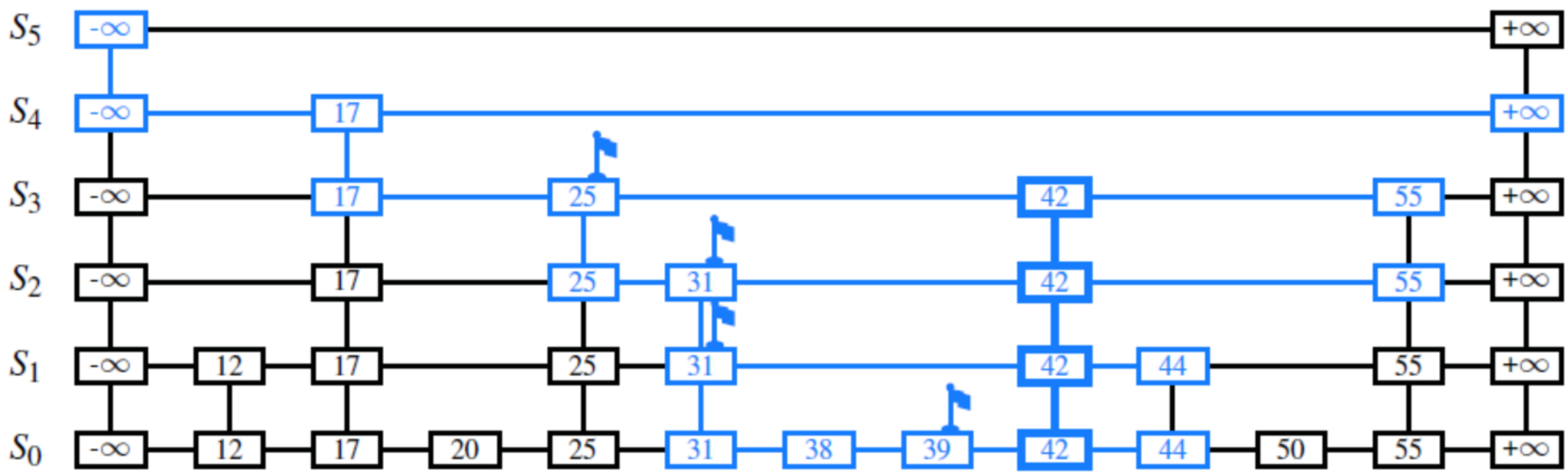
Output: Position p in the bottom list S_0 with the largest key having $\text{key}(p) \leq k$

```
p = s                                   {begin at start position}
while below(p) ≠ null do
    p = below(p)                        {drop down}
    while k ≥ key(next(p)) do
        p = next(p)                     {scan forward}
return p
```

- Expected running time: O(log n)

Skip Lists

- Insertion
  - E.g. Inserting 42



  - New entries are in thick lines; their preceding entries are flagged
  - Expected running time: O(log n)

Algorithm SkipInsert(k, v):

Input: Key k and value v

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

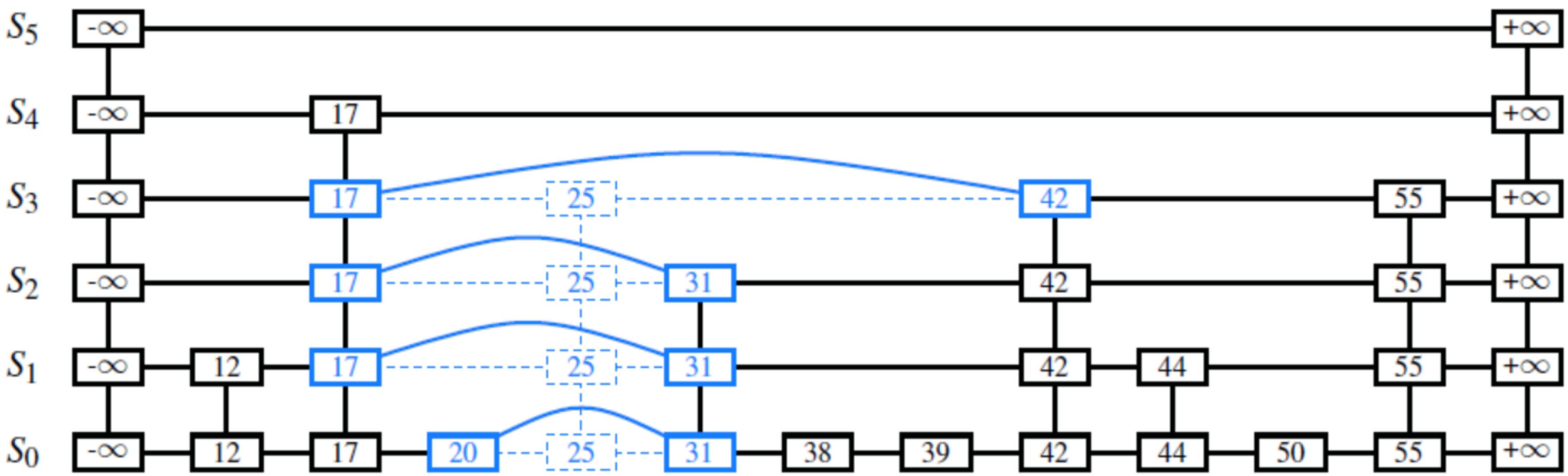
```
p = SkipSearch(k)        {position in bottom list with largest key less than k}
q = null                 {current node of new entry's tower}
i = −1                   {current height of new entry's tower}
repeat
   i = i + 1             {increase height of new entry's tower}
   if i ≥ h then
      h = h + 1          {add a new level to the skip list}
      t = next(s)
      s = insertAfterAbove(null, s, (−∞, null))   {grow leftmost tower}
      insertAfterAbove(s, t, (+∞, null))          {grow rightmost tower}
   q = insertAfterAbove(p, q, (k, v))             {add node to new entry's tower}
   while above(p) == null do
      p = prev(p)        {scan backward}
   p = above(p)          {jump up to higher level}
until coinFlip() == tails
n = n + 1
return q                 {top node of new entry's tower}
```

s: top of the first tower
t: top of the last tower

Add (k,v) after p
and above q

Skip Lists

- Removal
  - E.g. Removing 25



  - Expected running time: O(log n)