Homework 5 R-8.5

Design an algorithm to count the number of leaves in a binary tree that are the *left* child of their respective parent.

```
Public int countLeftLeaves (Node t){
        If (t == null)
                Return 0;
        If (t.left == null && t. right == null)
                Return 1;
        Else
                Return countLeftLeaves(node.left)
}
```

C-8.28

- The ***path length*** of a tree *T* is the sum of the depths of all positions in *T*. Describe a linear-time method for computing the path length of a tree *T*.

```
Algorithm pathlength(T, node, depth)
        If n == leaf node
                Return depth
        Else
                d = depth
                for each child of node
                        d = d + pathlength(T, child, d + 1)
        return d
```

Recaps of Heaps

- Array-based Heap Implementation
- Bottom-up Heap Construction

Exercise 4

- Construct a heap from the following input sequence: (2, 5, 16, 4, 10, 23, 39, 18, 26, 15).

Homework 6

- Illustrate the execution of the in-place heap-sort algorithm on the following input sequence: (2, 5, 16, 4, 10, 23, 39, 18, 26, 15).

CS 2233-01

Data Structures and Algorithms

Instructor: Dr. Qingguo Wang
College of Computing & Technology
Lipscomb University

Chapter 10 Outline

Maps

Hash Tables

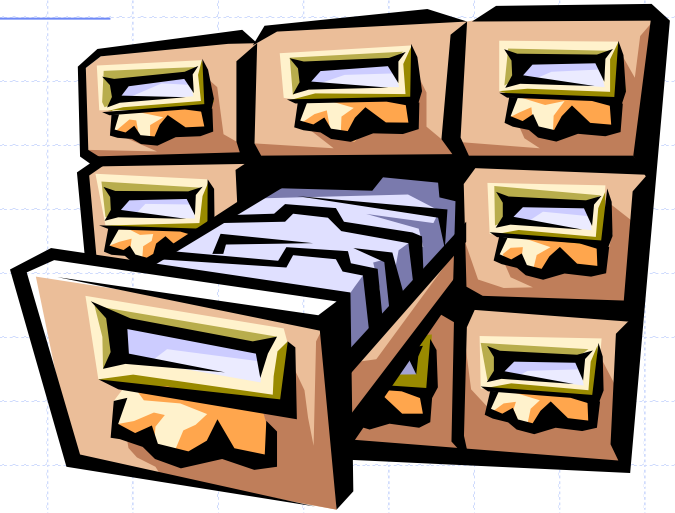
Sorted Maps

Skip Lists

Sets, Multisets, and Multimaps

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Maps



Maps

- A map models a searchable collection of key-value entries
- Stores key-value pairs (k, v), which we call ***entries***. Keys are required to be unique
- Efficiently store and retrieve values based upon a uniquely identifying ***search key*** for each.

Maps

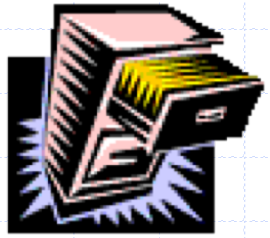
- The main operations of a map are for **searching, inserting**, and **deleting** items
- Multiple entries with the same key are not allowed
- Applications: address book, student-record database

Maps V.S. Arrays

- Maps are also known as ***associative arrays***, because the entry's key serves somewhat like an index into the map, in that it assists the map in efficiently locating the associated entry.

- Unlike a standard array, a key of a map need not be numeric, and does not directly designate a position within the structure.

The Map ADT



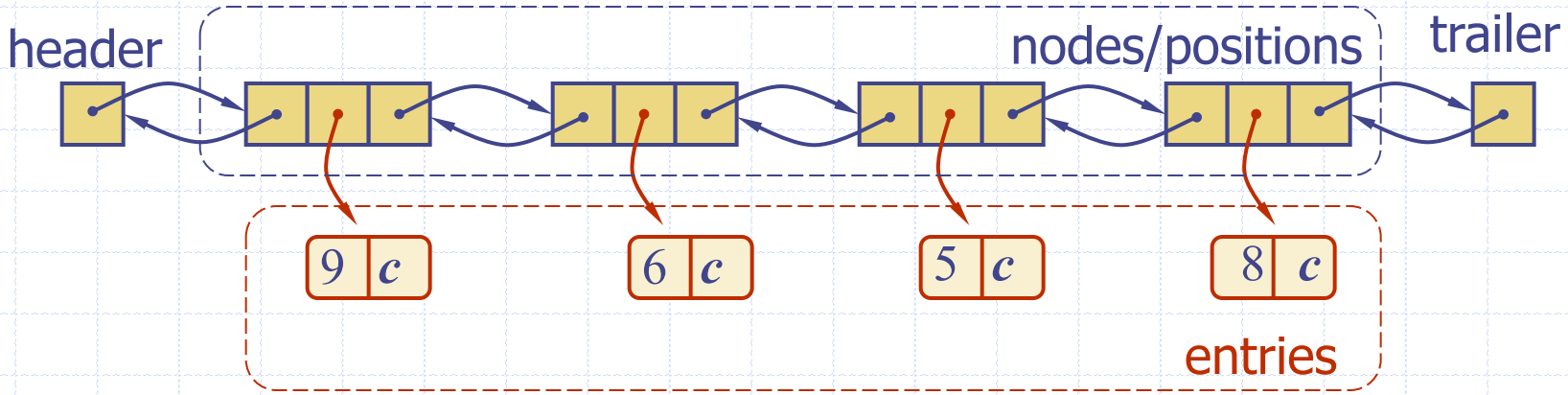
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Example

Operation	*Output*	*Map*
isEmpty()	**true**	Ø
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*)
isEmpty()	**false**	(7,*B*),(8,*D*)

A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doublylinked list), in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()   { the next position in B }
        if p.element().getKey() = k          then
            return p.element().getValue()
    return null {there is no entry with key equal to k}
```

The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
          t = p.element().getValue()
          S.set(p,(k,v))
          return t        {return the old value}
S.addLast((k,v))
n = n + 1       {increment variable storing number of entries}
return null     { there was no entry with key equal to k }
```

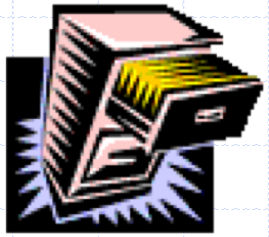
The remove(k) Algorithm

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
        t = p.element().getValue()
        S.remove(p)
        n = n – 1       {decrement number of entries}
        return t        {return the removed value}
return null             {there is no entry with key equal to k}
```

Performance of a List-Based Map

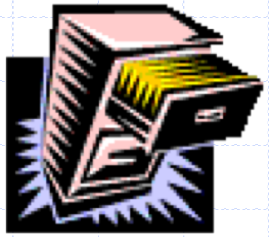
- Performance:
  - put takes $O(n)$ time
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Recall the Map ADT

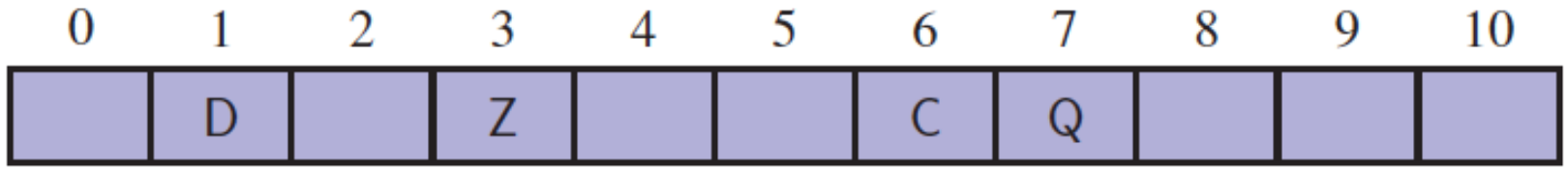


- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

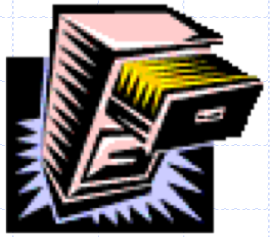
Intuitive Notion of a Map



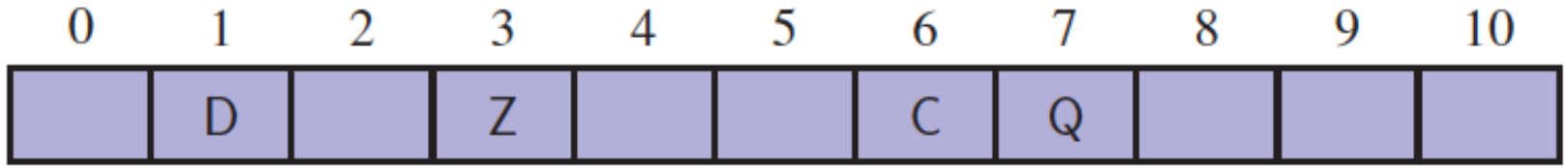
- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- But what should we do if our keys are not integers in the range from 0 to N – 1?



Intuitive Notion of a Map

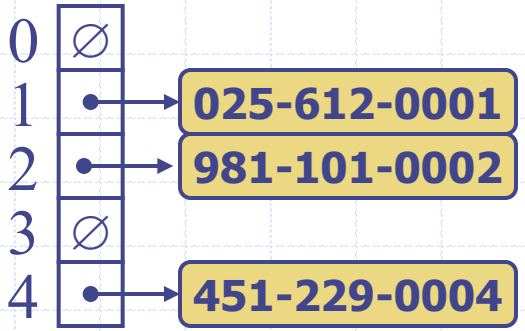


- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- But what should we do if our keys are not integers in the range from 0 to N – 1?
- Use a **hash function** to map general keys to corresponding indices in a table.

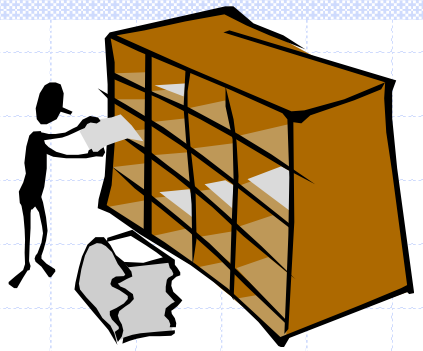


Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Hash Tables



Hash Functions and Hash Tables



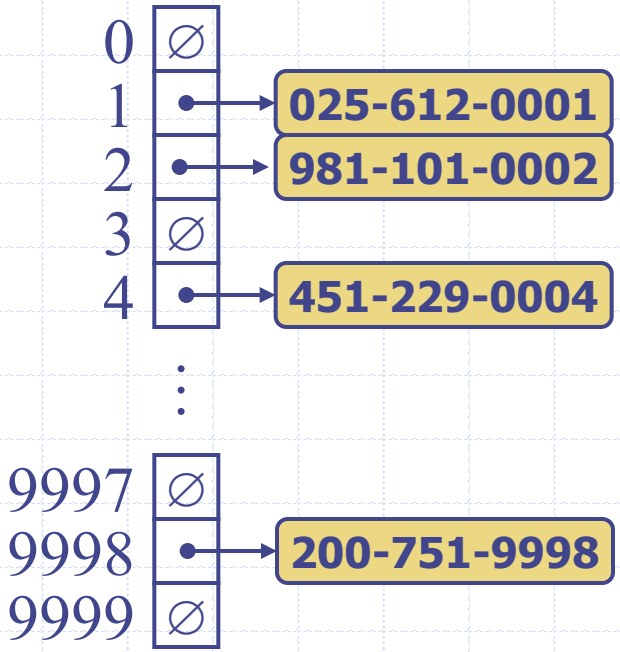
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
  - Example:
    $h(x) = x \bmod N$
    is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N

Implementing a map with a hash table

- Given entry (k, o)
- Calculate index $i = h(k)$
- Store the entry at index i of the hash table (of size N)

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = \text{last four digits of } x$



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

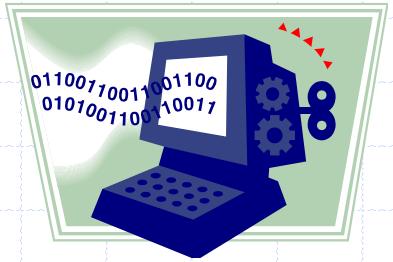
  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

  $$h(x) = h_2(h_1(x))$$

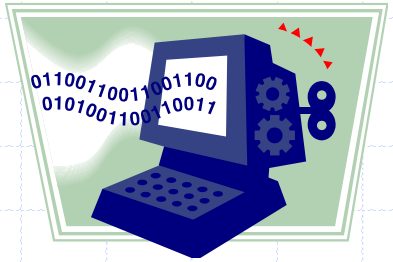
- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
- Integer cast:
  - We reinterpret the bits of the key as an integer
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)

Hash Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0 a_1 \ldots a_{d-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{d-1} z^{d-1}$$
    at a fixed value z, ignoring overflows
  - e.g. $z = 33$
- Polynomial $p(z)$ can be evaluated in $O(d)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{d-1}$$
    $$p_i(z) = a_{d-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, d-1)$$
- We have $p(z) = p_{d-1}(z)$

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0 a_1 \ldots a_{d-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{d-1} z^{d-1}$$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives **at most 6 collisions on a set of 50,000** English words)

- Polynomial $p(z)$ can be evaluated in $O(d)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{d-1}$$
    $$p_i(z) = a_{d-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, d-1)$$
- We have $p(z) = p_{d-1}(z)$

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

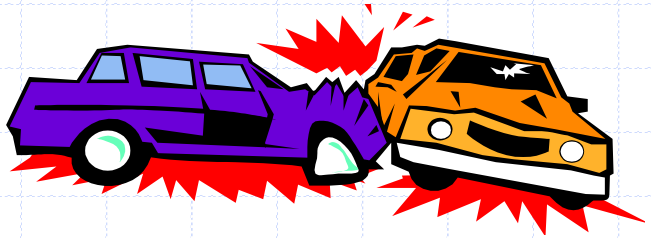
Abstract Hash Map in Java

```
1  public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
2    protected int n = 0;                    // number of entries in the dictionary
3    protected int capacity;                 // length of the table
4    private int prime;                      // prime factor
5    private long scale, shift;              // the shift and scaling factors
6    public AbstractHashMap(int cap, int p) {
7      prime = p;
8      capacity = cap;
9      Random rand = new Random();
10     scale = rand.nextInt(prime−1) + 1;
11     shift = rand.nextInt(prime);
12     createTable();
13   }
14   public AbstractHashMap(int cap) { this(cap, 109345121); }   // default prime
15   public AbstractHashMap() { this(17); }                      // default capacity
16   // public methods
17   public int size() { return n; }
18   public V get(K key) { return bucketGet(hashValue(key), key); }
19   public V remove(K key) { return bucketRemove(hashValue(key), key); }
20   public V put(K key, V value) {
21     V answer = bucketPut(hashValue(key), key, value);
22     if (n > capacity / 2)                 // keep load factor <= 0.5
23       resize(2 * capacity − 1);           // (or find a nearby prime)
24     return answer;
25   }
```

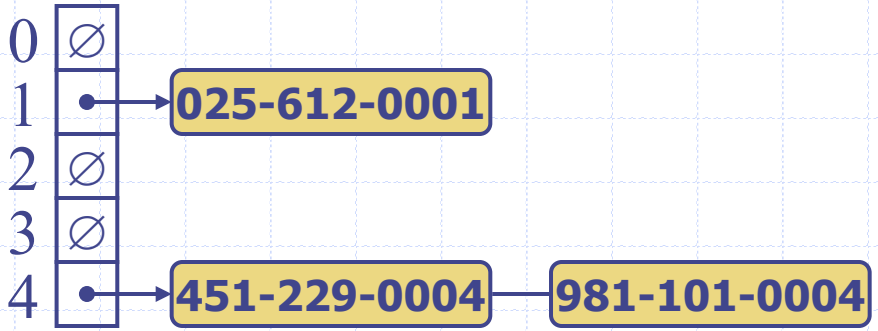
Abstract Hash Map in Java, 2

```
26    // private utilities
27    private int hashValue(K key) {
28      return (int) ((Math.abs(key.hashCode( )*scale + shift) % prime) % capacity);
29    }
30    private void resize(int newCap) {
31      ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32      for (Entry<K,V> e : entrySet( ))
33        buffer.add(e);
34      capacity = newCap;
35      createTable( );                    // based on updated capacity
36      n = 0;                             // will be recomputed while reinserting entries
37      for (Entry<K,V> e : buffer)
38        put(e.getKey( ), e.getValue( ));
39    }
40    // protected abstract methods to be implemented by subclasses
41    protected abstract void createTable( );
42    protected abstract V bucketGet(int h, K k);
43    protected abstract V bucketPut(int h, K k, V v);
44    protected abstract V bucketRemove(int h, K k);
45  }
```

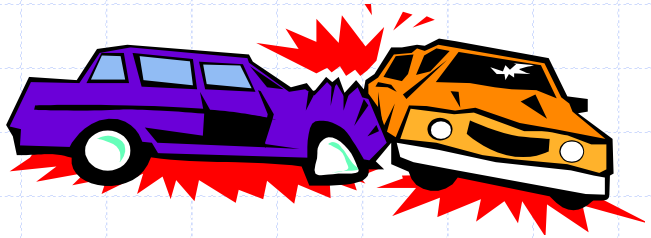
Collision Handling



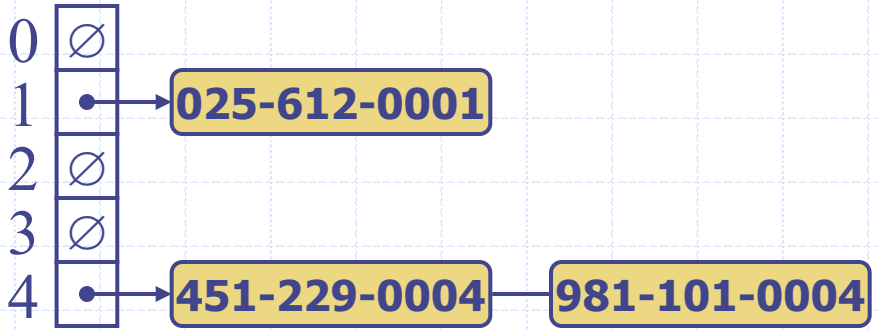
- Collisions occur when different elements are mapped to the same cell



Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



Map with Separate Chaining

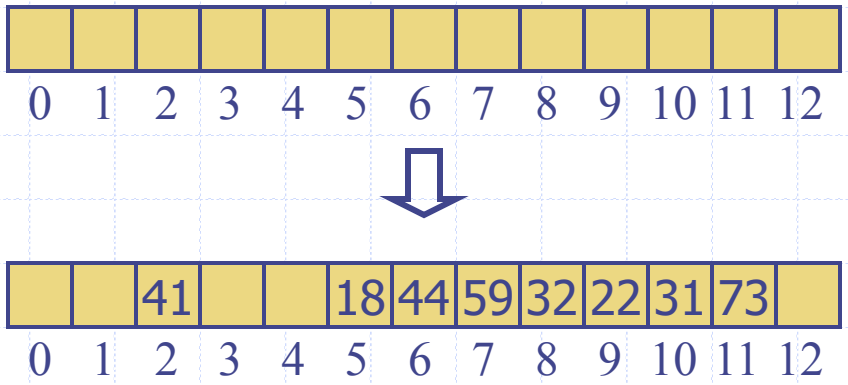
Delegate operations to a list-based map at each cell:

```
Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = null then                    {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ null then                    {k was found}
    n = n - 1
return t
```

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Example

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12

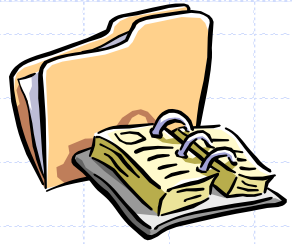
0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	

Example

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.getKey() = k
            return c.getValue()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called $DEFUNCT$, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item $DEFUNCT$ and we return element o
  - Else, we return $null$
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores $DEFUNCT$, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

Probe Hash Map in Java

```
1  public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
2    private MapEntry<K,V>[ ] table;        // a fixed array of entries (all initially null)
3    private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel
4    public ProbeHashMap( ) { super( ); }
5    public ProbeHashMap(int cap) { super(cap); }
6    public ProbeHashMap(int cap, int p) { super(cap, p); }
7    /** Creates an empty table having length equal to current capacity. */
8    protected void createTable( ) {
9      table = (MapEntry<K,V>[ ]) new MapEntry[capacity];   // safe cast
10   }
11   /** Returns true if location is either empty or the "defunct" sentinel. */
12   private boolean isAvailable(int j) {
13     return (table[j] == null || table[j] == DEFUNCT);
14   }
```

Probe Hash Map in Java, 2

```
15    /** Returns index with key k, or −(a+1) such that k could be added at index a. */
16    private int findSlot(int h, K k) {
17      int avail = −1;                          // no slot available (thus far)
18      int j = h;                               // index while scanning table
19      do {
20        if (isAvailable(j)) {                  // may be either empty or defunct
21          if (avail == −1) avail = j;          // this is the first available slot!
22          if (table[j] == null) break;         // if empty, search fails immediately
23        } else if (table[j].getKey( ).equals(k))
24          return j;                            // successful match
25        j = (j+1) % capacity;                  // keep looking (cyclically)
26      } while (j != h);                        // stop if we return to the start
27      return −(avail + 1);                     // search has failed
28    }
29    /** Returns value associated with key k in bucket with hash value h, or else null. */
30    protected V bucketGet(int h, K k) {
31      int j = findSlot(h, k);
32      if (j < 0) return null;                  // no match found
33      return table[j].getValue( );
34    }
```

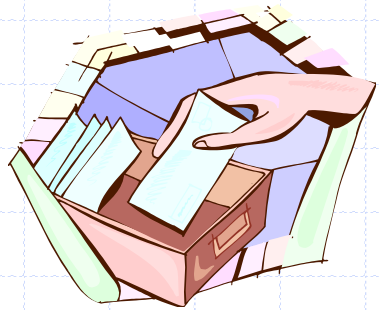
Probe Hash Map in Java, 3

```
35    /** Associates key k with value v in bucket with hash value h; returns old value. */
36    protected V bucketPut(int h, K k, V v) {
37      int j = findSlot(h, k);
38      if (j >= 0)                                  // this key has an existing entry
39        return table[j].setValue(v);
40      table[-(j+1)] = new MapEntry<>(k, v);        // convert to proper index
41      n++;
42      return null;
43    }
44    /** Removes entry having key k from bucket with hash value h (if any). */
45    protected V bucketRemove(int h, K k) {
46      int j = findSlot(h, k);
47      if (j < 0) return null;                      // nothing to remove
48      V answer = table[j].getValue( );
49      table[j] = DEFUNCT;                          // mark this slot as deactivated
50      n--;
51      return answer;
52    }
53    /** Returns an iterable collection of all key-value entries of the map. */
54    public Iterable<Entry<K,V>> entrySet( ) {
55      ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
56      for (int h=0; h < capacity; h++)
57        if (!isAvailable(h)) buffer.add(table[h]);
58      return buffer;
59    }
60  }
```

Exercise

- Draw the 11-entry hash table that results from using the hash function, $h(i) = (3i+5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by **linear probing**.

Double Hashing



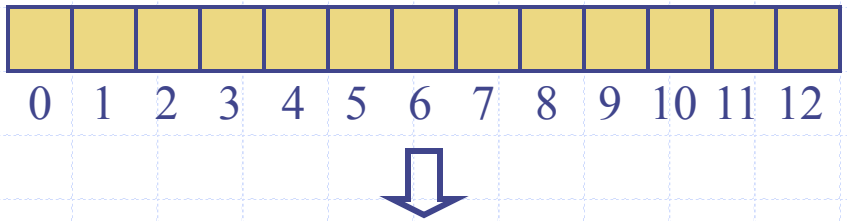
- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jd(k)) \bmod N$$
for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:
$$d_2(k) = q - k \bmod q$$
where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
$$1, 2, \ldots, q$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

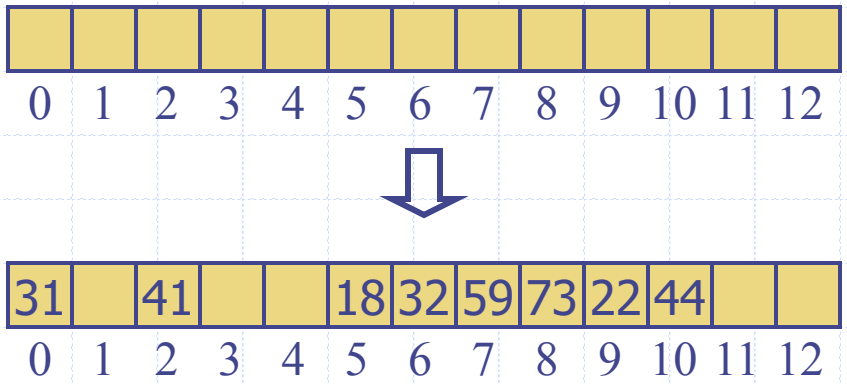
k	$h(k)$	$d(k)$	Probes
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5
59	7	4	7
32	6	3	6
31	5	4	5
73	8	4	8



Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Hashing

- The worst case occurs when all the keys inserted into the map collide.
- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The ratio n/N is called ***load factor***. In practice, hashing is very fast provided the load factor is not close to 100%
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$

Exercises

- How to solve the problem of counting the number of occurrences of words in a document?

- Google interview question: Remove duplicates in an array of numbers.

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Sorted Maps

Inexact search

- The map ADT allows a user to look up the value associated with a given key, the search for that key is a form known as an ***exact search***

- **Inexact search** if we does not find exact match, we can instead return the index at or near where a target might be found.

Maps and inexact search

- Consider an example:
  - Using time stamp as key, record about the event that occurred at that time as the value
  - Map ADT does not provide any way to get a list of all events ordered by the time at which they occur, or to search for which event occurred closest to a particular time.

Sorted Map ADT

Sorted map ADT includes all behaviors of the standard map, plus the following:

firstEntry( ): Returns the entry with smallest key value (or null, if the map is empty).

lastEntry( ): Returns the entry with largest key value (or null, if the map is empty).

ceilingEntry(k): Returns the entry with the least key value greater than or equal to k (or null, if no such entry exists).

floorEntry(k): Returns the entry with the greatest key value less than or equal to k (or null, if no such entry exists).

lowerEntry(k): Returns the entry with the greatest key value strictly less than k (or null, if no such entry exists).

higherEntry(k): Returns the entry with the least key value strictly greater than k (or null if no such entry exists).

subMap(k_1, k_2): Returns an iteration of all entries with key greater than or equal to k_1, but strictly less than k_2.

An Implementation of Sorted Maps

- Store the map's entries in an array list A so that they are in increasing order of their keys.
- We refer to this implementation as a ***sorted search table***.

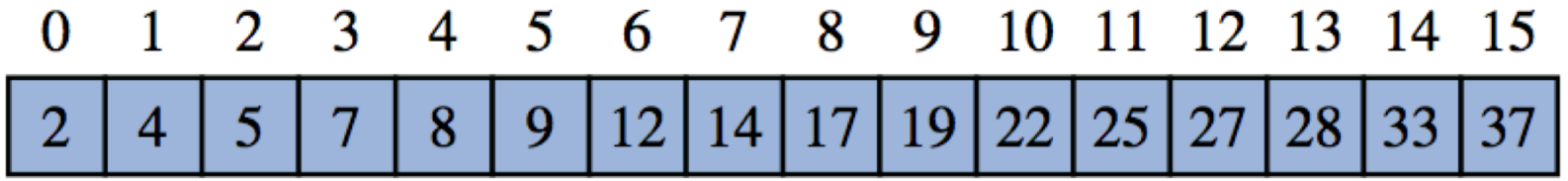


Figure 10.8: Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.

Analysis

Method	Running Time
size	
get	
put	
remove	
firstEntry, lastEntry	
ceilingEntry, floorEntry, lowerEntry, higherEntry	
subMap	
entrySet, keySet, values	

Table 10.3: Performance of a sorted map, as implemented with SortedTableMap. We use n to denote the number of items in the map at the time the operation is performed. The space requirement is $O(n)$.

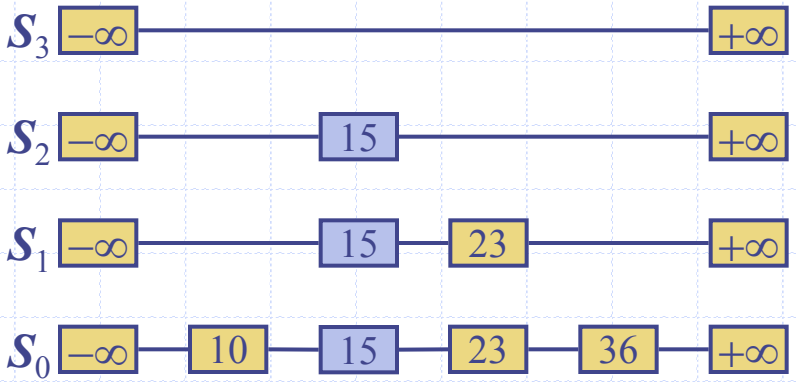
Analysis

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if map has entry with given key
remove	$O(n)$
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s+\log n)$ where s items are reported
entrySet, keySet, values	$O(n)$

Table 10.3: Performance of a sorted map, as implemented with SortedTableMap. We use n to denote the number of items in the map at the time the operation is performed. The space requirement is $O(n)$.

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Skip Lists

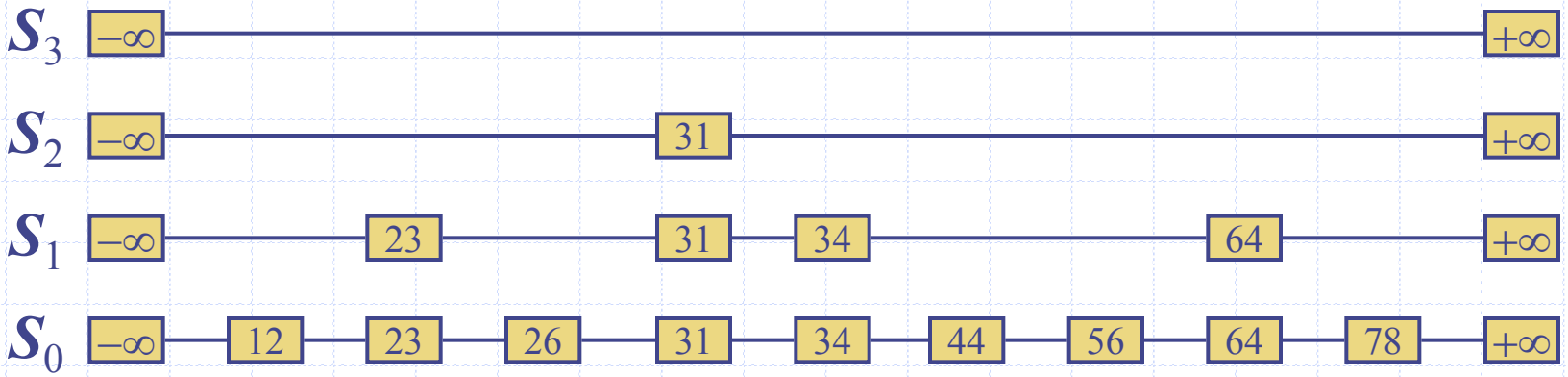


Introduction to Skip Lists

- Skip lists provide a clever compromise to efficiently support search and update operations;
- they are implemented as the java.util.ConcurrentSkipListMap class.

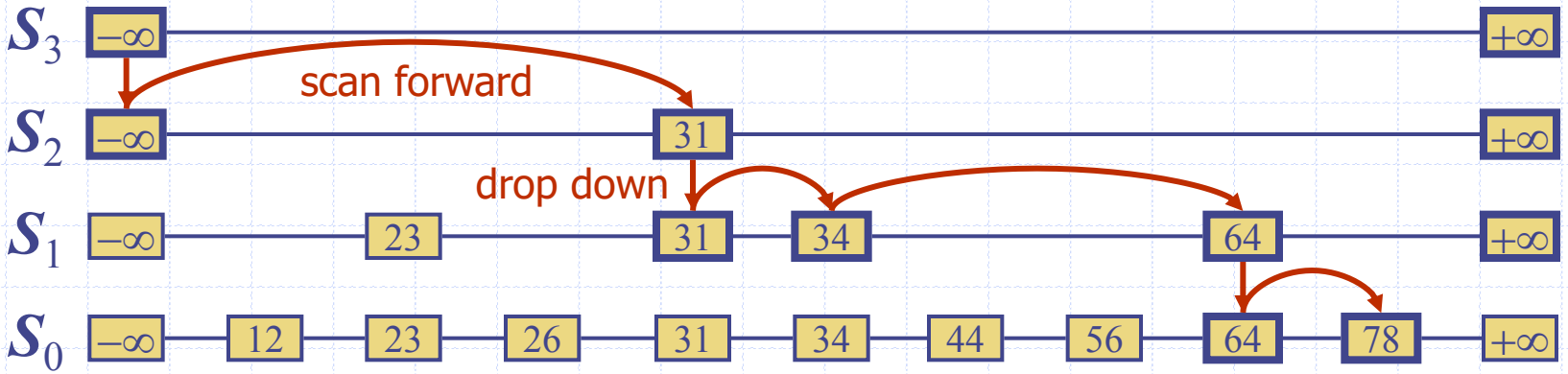
What is a Skip List

- 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.,
    $$S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$$
    where h is called height of skip list.
  - List S_h contains only the two special keys



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 $y \leftarrow key(next(p))$
    - $x = y$: we return $element(next(p))$
    - $x > y$: we "scan forward" until at the rightmost position
    - $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return *null*
- Example: search for 78

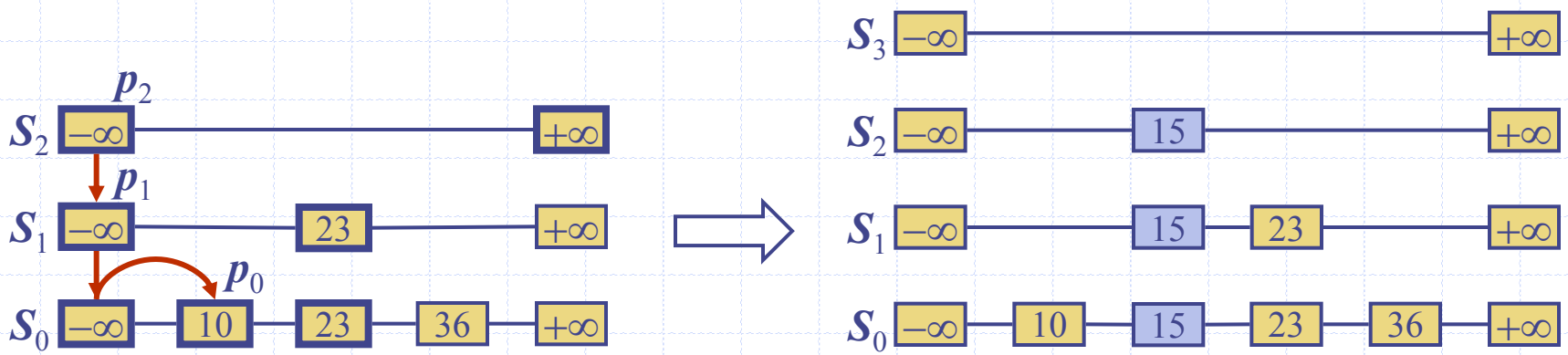


Skip lists and Randomization

- Skip lists are set up so that S_{i+1} contains roughly alternate entries of S_i.
- However, the halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists; instead, randomization is used.
- The entries in S_{i+1} are chosen at random from the entries in S_i by picking each entry from S_i to also be in S_{i+1} with probability 1/2.
- We use a randomized algorithm to insert items into a skip list

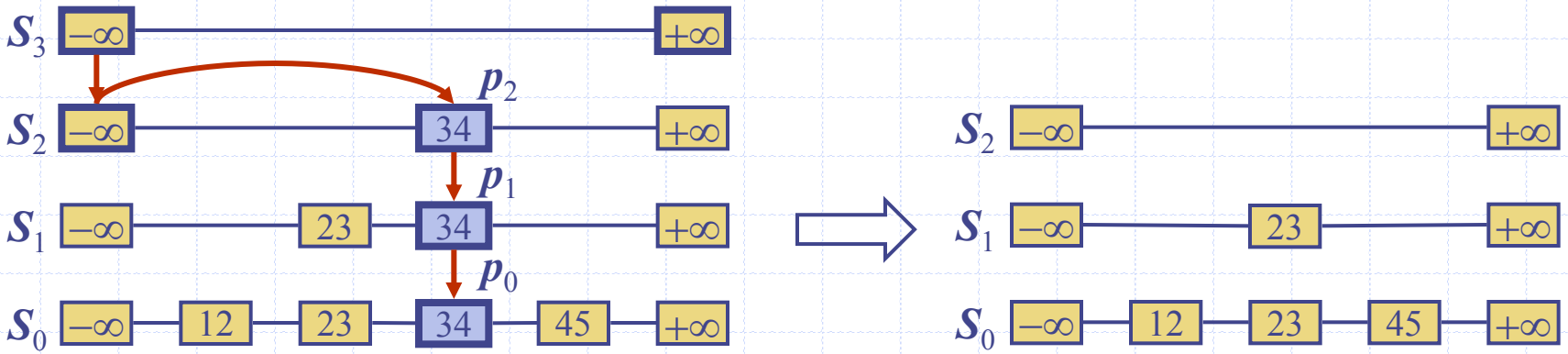
Insertion

- 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 $i = 2$



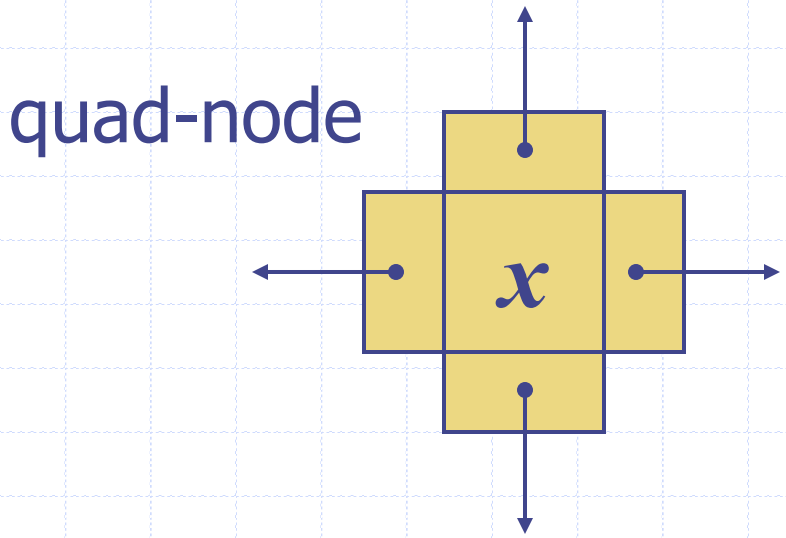
Deletion

- 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

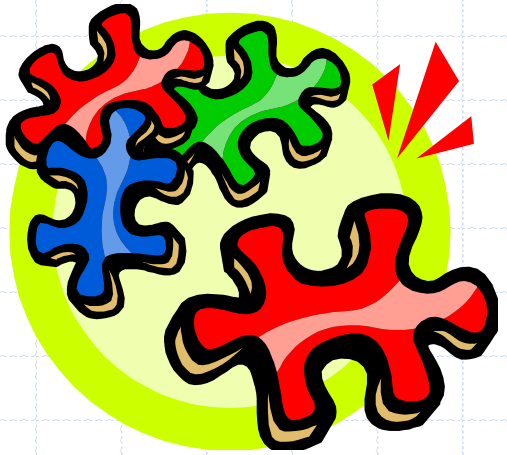
- A skip list is a data structure for maps that uses a randomized insertion algorithm
- In a skip list with n entries
  - The expected space used is $O(n)$
  - The expected search, insertion and deletion time is $O(\log n)$
- Using a more complex probabilistic analysis, one can show that these performance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

Method	Running Time
size, isEmpty	$O(1)$
get	$O(\log n)$ expected
put	$O(\log n)$ expected
remove	$O(\log n)$ expected
firstEntry, lastEntry	$O(1)$
ceilingEntry, floorEntry lowerEntry, higherEntry	$O(\log n)$ expected
subMap	$O(s+\log n)$ expected, with s entries reported
entrySet, keySet, values	$O(n)$

Table 10.4: Performance of a sorted map implemented with a skip list. We use n to denote the number of entries in the dictionary at the time the operation is performed. The expected space requirement is $O(n)$.

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Sets and Multimaps



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

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e:\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Set Implementations

- The Java Collections Framework includes the following set implementations, mirroring similar data structures used for maps:
  - java.util.HashSet provides an implementation of the (unordered) set ADT with a hash table.
  - java.util.concurrent.ConcurrentSkipListSet provides an implementation of the sorted set ADT using a skip list.
  - java.util.TreeSet provides an implementation of the sorted set ADT using a balanced search tree. (Search trees are the focus of Chapter 11.)

After Spring Break

Chapter 11 Search Trees