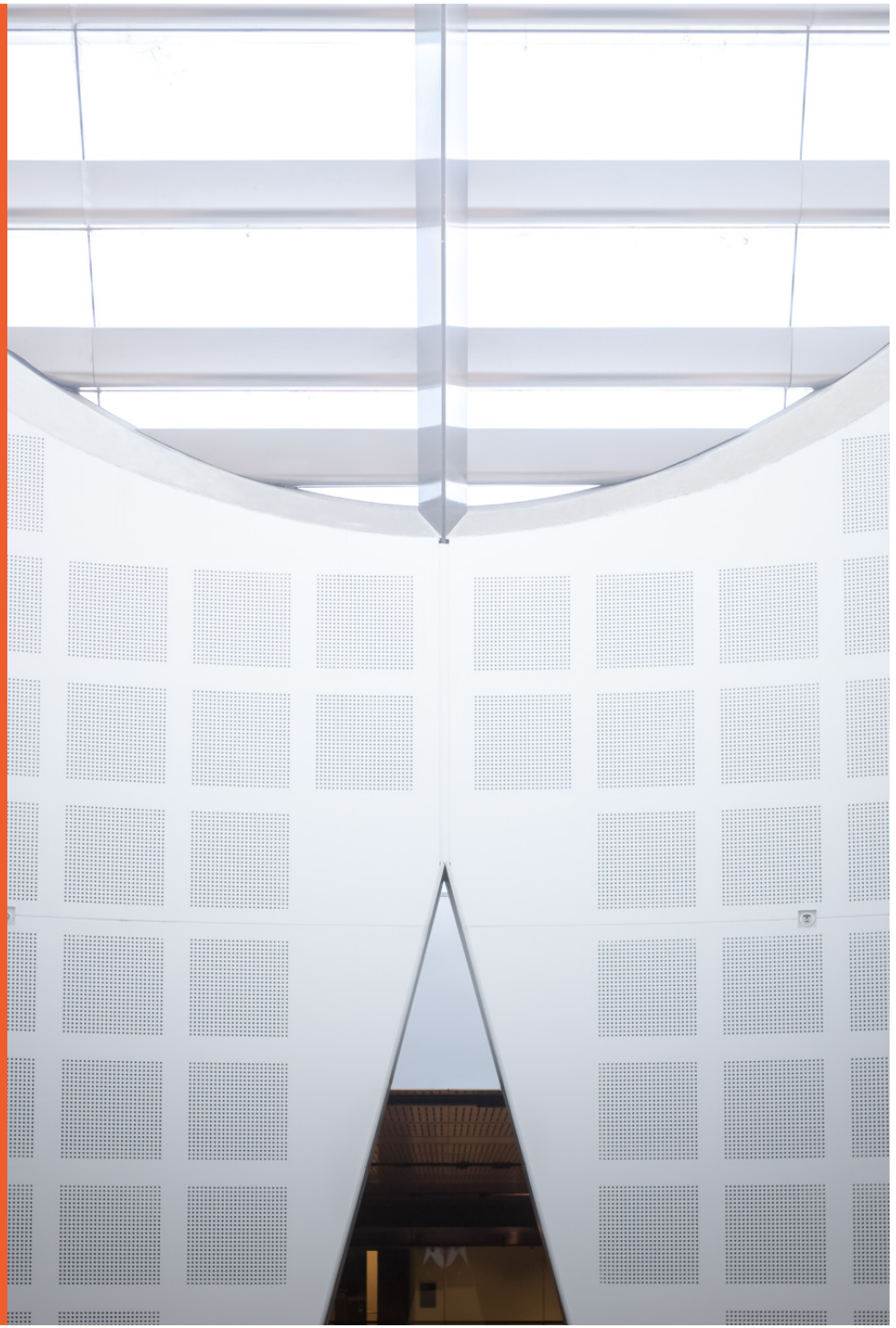
INFO1105/1905
Data Structures
Week 9: Hashtables

see textbook sections 10.2, 10.5

Professor Alan Fekete
Dr John Stavrakakis
School of Information Technologies

using material from the textbook
and A/Prof Kalina Yacef



THE UNIVERSITY OF SYDNEY

Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5^{th} & 6^{th} edition)
- With modifications and additions from the University of Sydney

- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

Reminder: mid-semester break

- No classes next week (Sept 26-30) or on Monday Oct 3
- But week 9 weekly challenge task is due on Monday Sept 26
- “Week 10” will begin on Tuesday October 4

Reminder! Quiz 4

- Quiz 4 will take place during lab in week 10, starting Oct 4(*)
- Done online, over a 20 minutes duration,
  - during the last 30 minutes of the lab period, or as indicated by your tutor
- A few multiple choice questions,
  - covering material from lectures of weeks 7 and 8 (labs 8 and 9)
    - graph definitions and properties
    - graph representations
    - BFS
    - DFS
    - directed and weighted graph algorithms

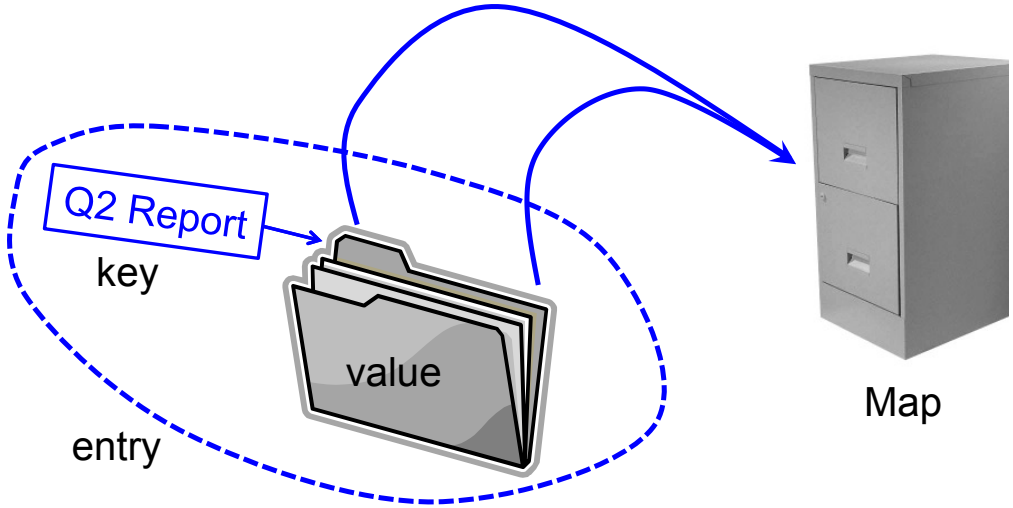
(*)Students with Monday lab: special arrangements, check email from your tutor

Outline

- Revision: Maps
  - ADT
- Hash tables
  - Hash functions, hash codes, compression functions
  - Collision handling: separate chaining
  - Performance
  - Choice of hash function
- Open addressing
- Related ADTs: SortedMap, Set, MultiSet

Recall: Maps

- A map models a searchable collection of key-value entries
  - Elements can be located quickly using keys
- Key = unique identifier
  - Multiple entries with the same key are not allowed
- The main operations of a map are for
  - searching,
  - inserting, and
  - deleting items
- Applications:
  - address book
  - student-record database
  - Web



Recall: 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 iterable collection of the values in M

Recall: Map Example

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

Simple Implementation with restricted keys

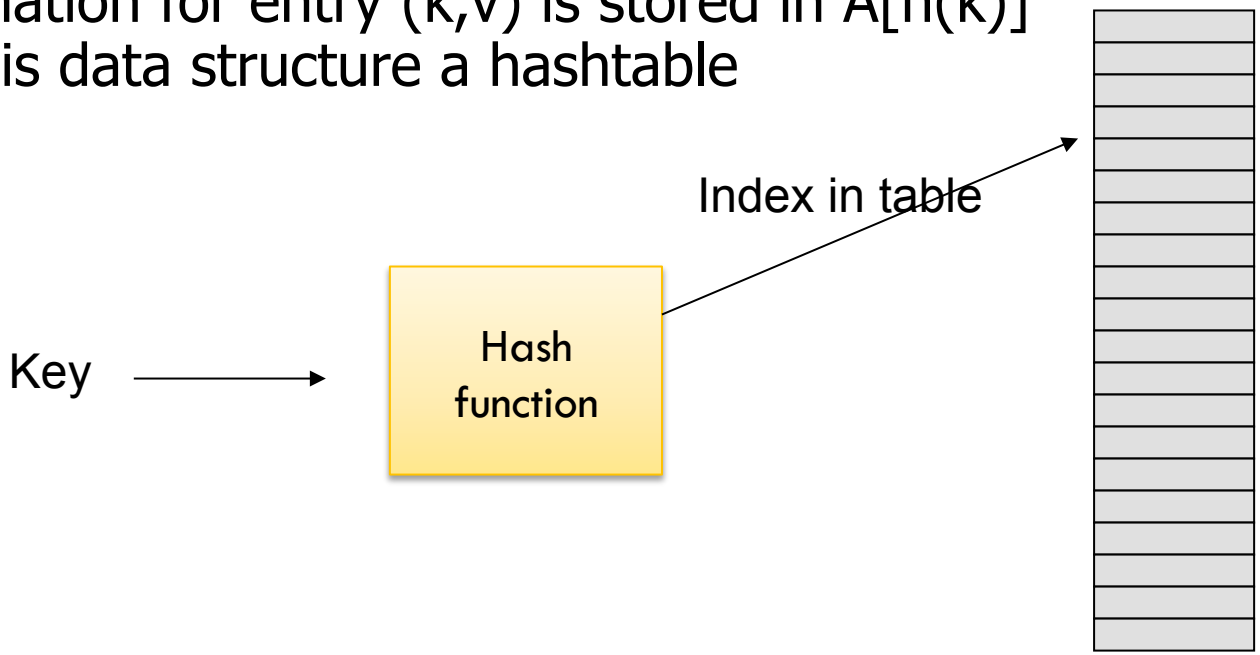
- Intuitively, a map M supports the abstraction of using keys as addresses to help locate an entry.
- Consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N − 1, for some N ≥ n.
  - Data structure as an array of size N
  - Key can be index so entries can be located directly
  - O(1) operations (get, put, remove)
- Eg StudentID is 9 digits, so a Map with StudentID key can be stored in array of 10,000,000,000 entries.

Evaluation of this structure

- Really good worst-case runtime
- Often, bad space utilization
  - when actual keys are sparse in the space of possible keys
  - as in StudentID example
- What about keys that are not from a bounded range of integers
  - eg web cache has URL as key (ie key is a String)

More General Kinds of Keys

- What should we do if our keys are not integers in the range from 0 to N – 1, or not integers at all?
  - Use a **hash function h** to map general keys to corresponding indices in an array A
  - h is a mathematical function (always gives same answer for any particular k
  - h is fairly efficient to compute
- the information for entry (k,v) is stored in A[h(k)]
- We call this data structure a hashtable



Hash functions

- Choosing a good hash function is not straightforward
- We discuss this later
- For our examples, we usually use very simple (and not good) choices that can be calculated by hand
  - eg for unbounded integer key in array of size 11, we might use remainder mod 11 as hash function
    - so h(25) = 3
  - eg for String key, in array of size 10 we might do an example where h(s) = last digit of position in alphabet of first character of s
    - so h("Mary") = 3 since M is 13-th character in alphabet

Important mathematical notion for hash tables: Arithmetic modulo N

x mod N is mathematical notation for remainder

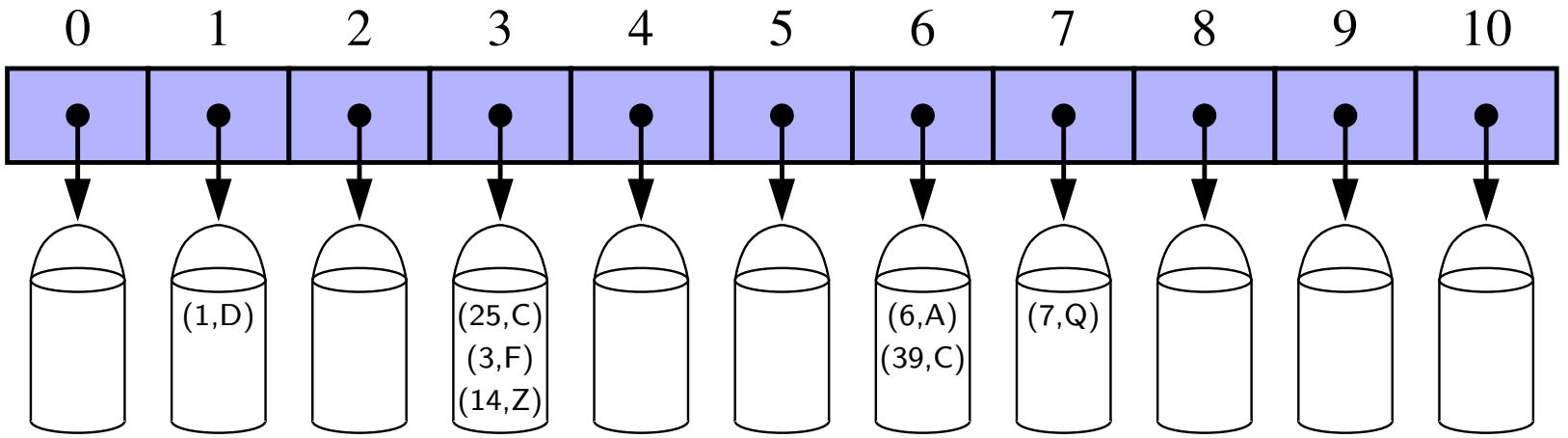
- If x = c * N + r with 0 <= r < N then r = x mod N
- Also r = x – N*(floor(x/N))
- So numbers wrap-around when working mod N
  - 35 mod 10 = 5
- Java operator (x % N)

Collisions

- Usually, the hash function h will not be injective
- That is, there can be different keys k1 and k2, such that h(k1) = h(k2)
- This is called a collision
- A good hash function makes collisions rare
- Dealing with collisions makes lots of complexity in design of hashtable data structures

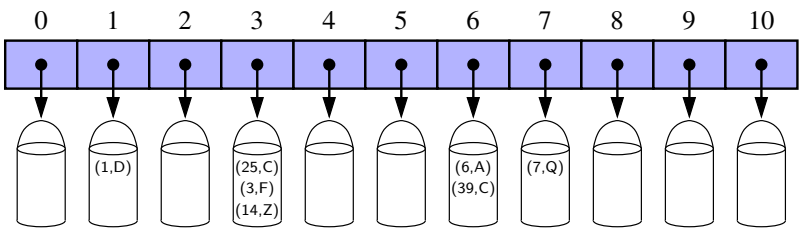
Handling collisions: separate chaining

- Instead of our simple array representation of a map
  - each place in array stores 0 or 1 entries
- We provide a bucket array, in which each bucket manages a collection of entries corresponding to the index calculated by the hash function
- Example below uses h(k) = k mod 11
  - so h(25) = 3



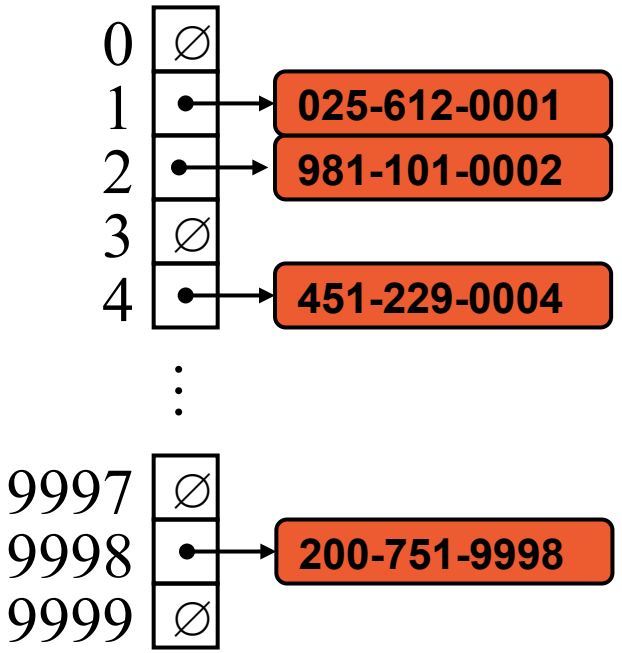
Summary: Hash Functions and Hash Tables

- A hash function ***h*** maps keys of a given type to integers in a fixed interval [0, ***N*** - 1]
  - Example:
    h(***x***) = ***x*** mod ***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
  - Array A (called table) of size ***N***
  - Hash function ***h*** which maps keys to array indices.
    - Eg h(1)=1, h(25)=3, h(14)=3 etc.
  - we store entry (k,v) in A[h(k)]



Example

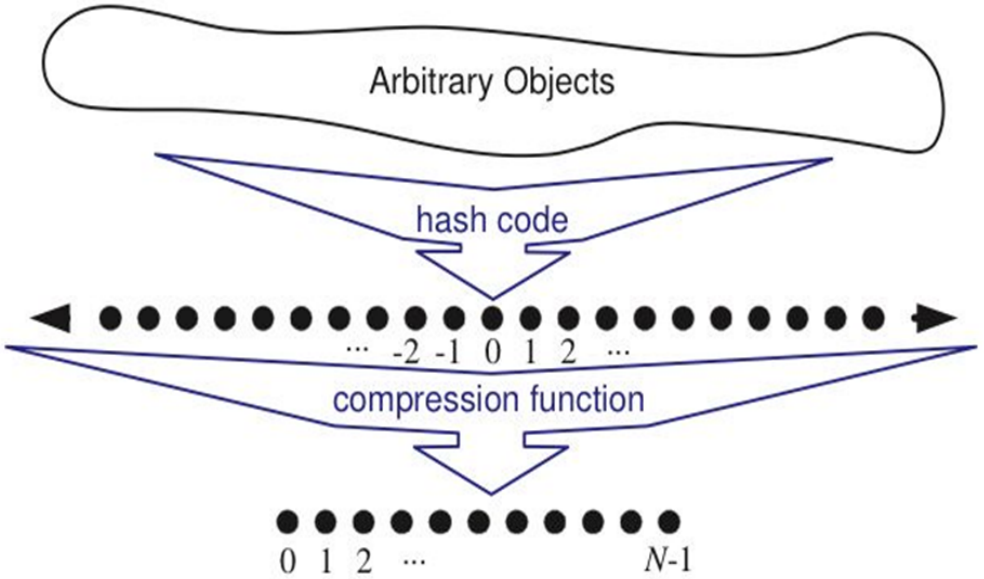
- We design a hash table for a map storing entries as (ID, Name), where ID is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x)$ = last four digits of x
- that is $h(x) = x \bmod 10000$



Hashing

- Many types of keys to start from : integers, floating point numbers, strings, or arbitrary objects (for example a whole binary search tree)
- Usually there are two steps to produce a hash value from an object:
  - Map the object to a corresponding integer
  - From this integer produce a resulting hash value in the desired interval

Hash Functions



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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

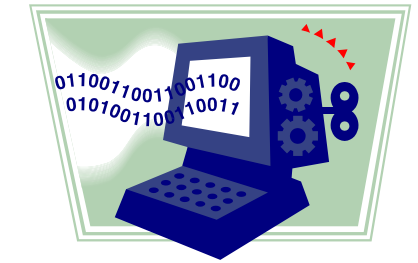
  h_2: integers $\rightarrow [0, N-1]$

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

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

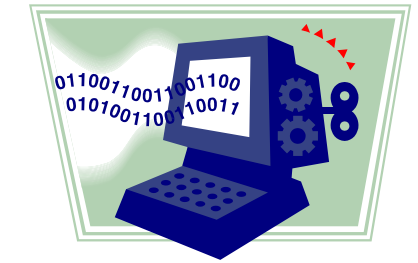
- The goal of the hash function is to "disperse" the keys in an apparently random way and avoid collisions

Common Hash Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hashCode() of all Java objects)
  - Often works ok, but problems with numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Directly 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)

Dealing with long keys



- Simplest approach: high order bits are ignored
  - Just use the lowest part of the key
  - This leads to lots of collisions
    - eg URLs often end in ".html"
- 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)
  - Component sum hash code not suited for character strings or n-tuples

```
static int hashCode(long i) {return (int)((i >> 32) + (int) i);}
```

Other approaches for long keys

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $a_0 a_1 \ldots a_{n-1}$
  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$

    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)
- Variant: cyclic shift hash code
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $p_0(z) = a_{n-1}$

    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$

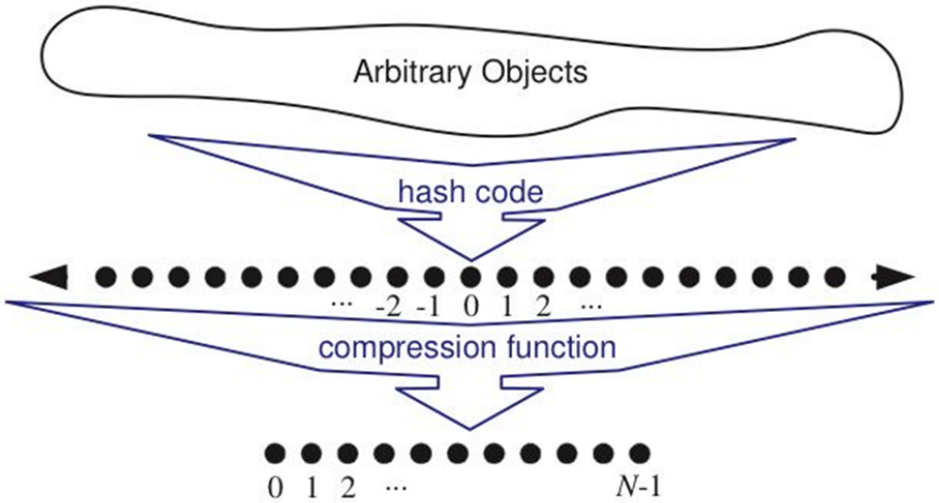
    $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

Hashcode function properties

- It is vital that hashcode be a mathematical function for key objects
- So, if two key objects are equal, they must have the same hashcode
- If a Java type redefines equals(), make sure to redefine hashcode appropriately,
  - so hashcode() uses no information except what is considered in equals() method
- Eg Date objects, String objects, other classes that use value-based equality

Compression functions

- Hash code for a key k typically not suitable for immediate use without compression
  - Wide range of k
  - Negative values
  - Need to map it to [0,N-1]
- Good compression function minimises collisions



Common Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime to minimise collisions
    - The reason has to do with number theory and is beyond the scope of this course
  - Eg (200, 205, 210, 215, …600)
    - If N = 100?
    - If N = 101?
  - Collisions if repeated patterns pN+q
- Multiply, Add and Divide (MAD):
  - Eliminate repeated patterns
  - $h_2(y) = (ay + b) \bmod N$
  - 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

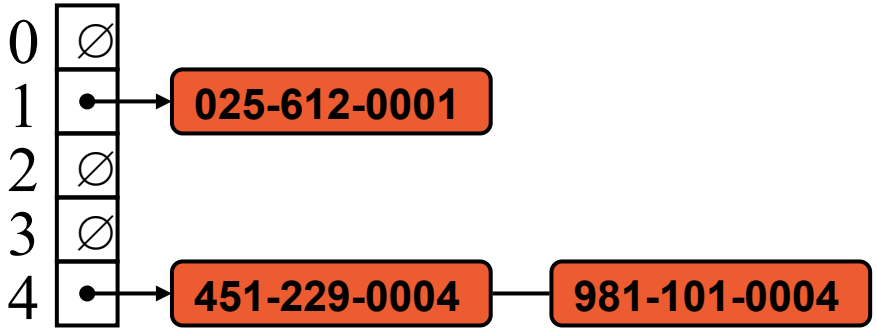
```
26    // private utilities
27    private int hashValue(K key) {
28      return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
29    }
30    private void resize(int newCap) {
31      ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32      for (Entry<K,V> e : entrySet())
33        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

- Recall: Collisions occur when different elements are mapped to the same cell
- We first describe a simple technique (separate chaining)

Separate chaining

- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

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
```

Performance

- Worst case is terrible: all keys collided
  - So one bucket has a list of length n
- Search and delete must traverse this list; insert needs to traverse also (checking in case key already exists)
- So worst case runtime is O(n) for all the operations!
- But in practice, the operations run *much* faster; in practice, we think of HashMap as having O(1) operations.

Typical Performance

- In practice, a good hash function distributes the keys evenly (that is, it is like a random allocation of buckets)
- Each bucket is *expected* to have length n/N
- Usually n is at most some constant times N
  - resize the table if n grows to big, re-calculating hash function for enlarged table size
  - So operations are expected time O(1)

Hash Table with Chaining

```
1  public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
2    // a fixed capacity array of UnsortedTableMap that serve as buckets
3    private UnsortedTableMap<K,V>[ ] table;   // initialized within createTable
4    public ChainHashMap( ) { super( ); }
5    public ChainHashMap(int cap) { super(cap); }
6    public ChainHashMap(int cap, int p) { super(cap, p); }
7    /** Creates an empty table having length equal to current capacity. */
8    protected void createTable( ) {
9      table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
10   }
11   /** Returns value associated with key k in bucket with hash value h, or else null. */
12   protected V bucketGet(int h, K k) {
13     UnsortedTableMap<K,V> bucket = table[h];
14     if (bucket == null) return null;
15     return bucket.get(k);
16   }
17   /** Associates key k with value v in bucket with hash value h; returns old value. */
18   protected V bucketPut(int h, K k, V v) {
19     UnsortedTableMap<K,V> bucket = table[h];
20     if (bucket == null)
21       bucket = table[h] = new UnsortedTableMap<>( );
22     int oldSize = bucket.size( );
23     V answer = bucket.put(k,v);
24     n += (bucket.size( ) − oldSize);      // size may have increased
25     return answer;
26   }
```

Hash Table with Chaining, 2

```
27    /** Removes entry having key k from bucket with hash value h (if any). */
28    protected V bucketRemove(int h, K k) {
29      UnsortedTableMap<K,V> bucket = table[h];
30      if (bucket == null) return null;
31      int oldSize = bucket.size( );
32      V answer = bucket.remove(k);
33      n -= (oldSize - bucket.size( ));      // size may have decreased
34      return answer;
35    }
36    /** Returns an iterable collection of all key-value entries of the map. */
37    public Iterable<Entry<K,V>> entrySet( ) {
38      ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
39      for (int h=0; h < capacity; h++)
40        if (table[h] != null)
41          for (Entry<K,V> entry : table[h].entrySet( ))
42            buffer.add(entry);
43      return buffer;
44    }
45  }
```

Open addressing

- An alternative approach to building Map based on hashing
- It does not use an auxiliary data structure to hold entries with colliding keys

- If the table entry is already occupied, divert to another cell in the table. Different methods to find "another" cell:
  - Linear Probing
  - Quadratic Probing
  - Double hashing

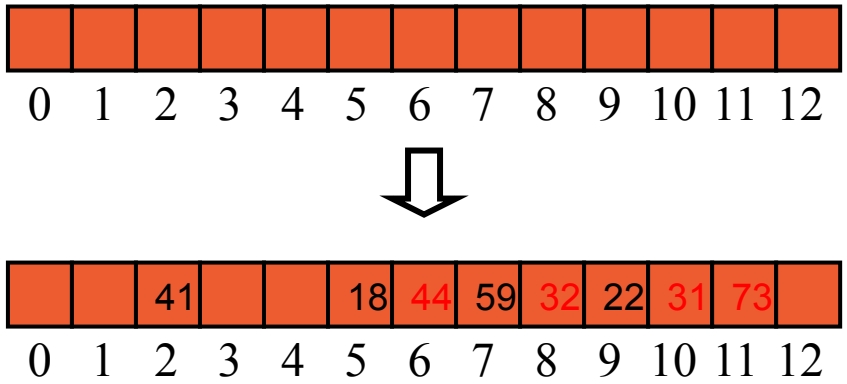
Linear Probing

- 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"
- Problem: Colliding items lump together, causing future collisions to cause a longer sequence of probes

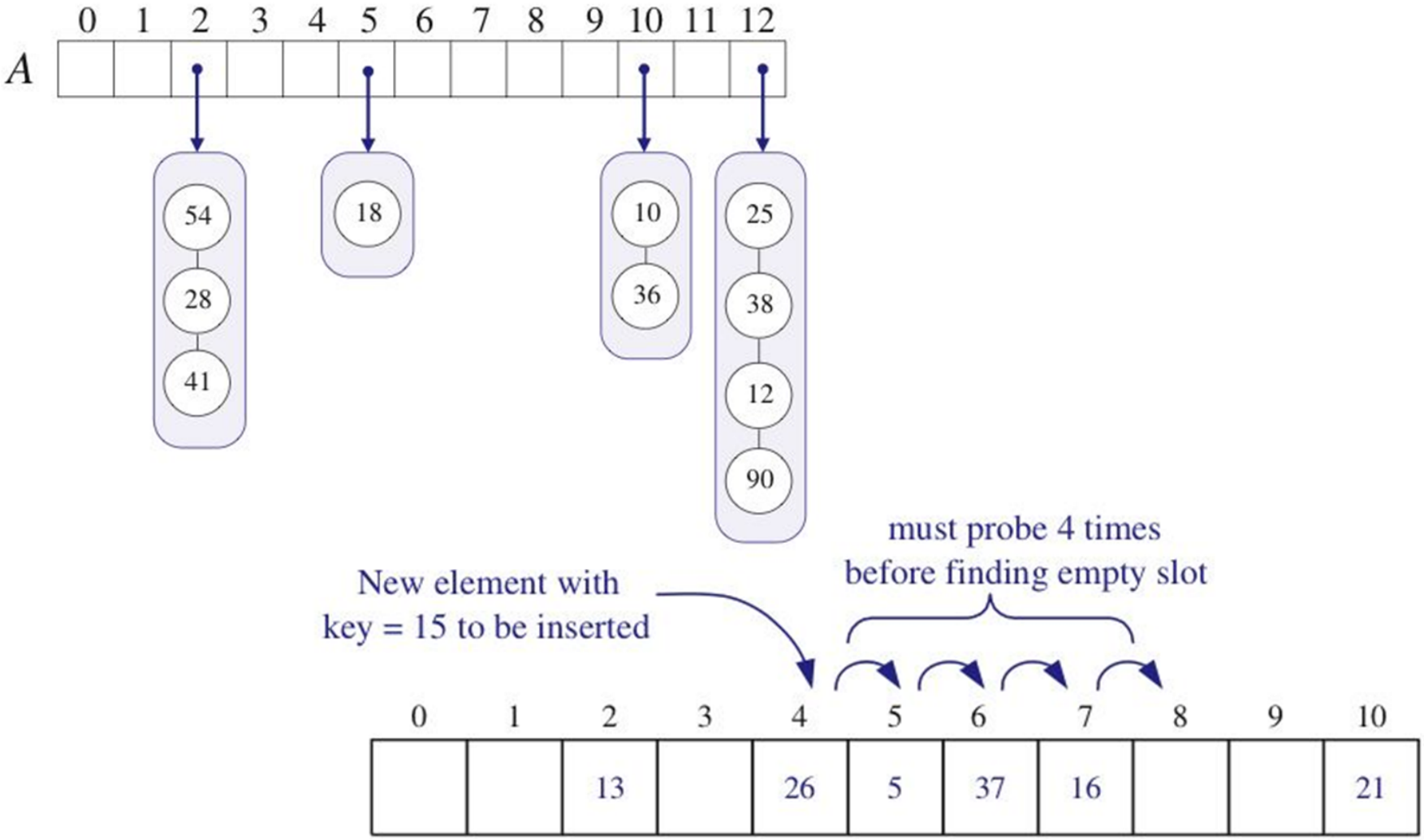
Modulo values

18	5
41	2
22	9
44	5
59	7
32	6
31	5
73	8

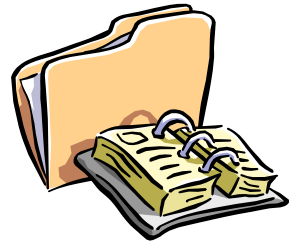
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Chaining versus probing



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
  - Equivalently, we examine cells at locations

    $(h(k)+j) \bmod N$

    for $j = 0, 1, \ldots, N-1$

```
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
- get(k) must pass over cells with DEFUNCT and keep probing
- 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  }
```

Quadratic probing

- handles collisions by placing an item in the first available cell of the series
  $(h(k)+j^2) \bmod N$
  for $j = 0, 1, \ldots, N-1$

- The table size N must be a prime to allow probing of all the cells
- Table must be less than half full

- Problems:
  - Can also produce a clustering pattern (although different to the linear probing)
  - If table fills up, not guaranteed to find an empty spot

Double Hashing

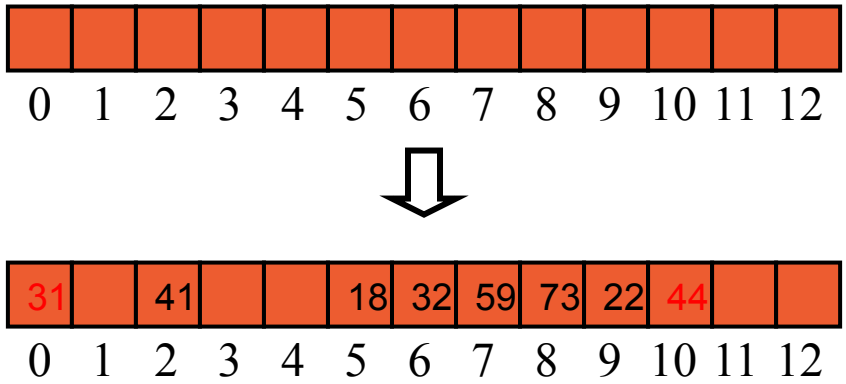
- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $(h(k) + j.d(k)) \bmod N$
  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow 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
  $1, 2, \ldots, q$

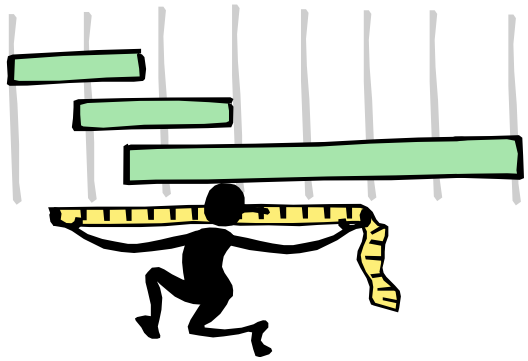
Example of Double Hashing

- 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	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of open addressing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1 / (1 - \alpha)$
- The expected running time of all the map ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Load factors

- Load factor of a hash table $\lambda = n/N = keys\ in\ the\ table / array\ size$
- Experimental analysis suggests we should aim for
  - $\lambda < 0.5$ for open addressing
  - $\lambda < 0.9$ for separate chaining
- Some Java implementations of hash tables will use the threshold of 0.75 for triggering a re-hashing to a new hash table
- Rehashing: pick larger table size, and new parameters for the hash function
- When rehashing to a new table, we typically at least double the size

Recall: sorted map ADT

- Similar to the map ADT, but it also keeps entries "sorted" according to some total order
  - Order can be defined by a given comparator for the entries
- If the data structure is aware of this order among the entries, then we can perform additional operations
  - map ADT plus order-related operations

Sorted map ADT (extra methods)

firstEntry() returns the entry with smallest key; if map is empty, returns null

lastEntry() returns the entry with largest key; if map is empty, returns null

ceilingEntry(k) returns the entry with least key that is greater than or equal to k (or null, if no such entry exists)

floorEntry(k) returns the entry with greatest key that is less than or equal to k (or null, if no such entry exists)

lowerEntry(k) returns the entry with greatest key that is strictly less than k (or null, if no such entry exists)

higherEntry(k) returns the entry with least key that is strictly greater than k (or null, if no such entry exists)

subMap(k1,k2) returns an iteration of all the entries with key greater than or equal to k1 and strictly less than k2

Sorted map implementation

- Hash tables *are not appropriate* for the implementation of ordered maps
  - They are not aware of any ordering between the keys/entries
- Ordered maps can be implemented reasonably efficiently using Binary Search Trees (from week 6)
  - Operations cost O(log n) provided the tree can be kept balanced

Definition

- 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.

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 implemented via Map

- Use a Map to store the keys, and ignore the value
- Allows contains(k) to be answered by get(k)
- Similarly for add and remove
- Using HashMap for Map, gives main Set operations that are usually O(1)

MultiSet

- Like a Set, but allows duplicates
  - also called a Bag
  - operation count(e) says how many occurrences of e in collection
  - remove(e) removes ONE occurrence (provided e is in the collection already)
- Implement by Map where the element is the key, and the associated value is the number of occurrences

Summary

- Read sections 10.2, 10.5 of textbook
- Hash tables for Map ADT
  - Wise choice of hash code and compression function to minimise collisions
  - Separate chaining
  - Open addressing, with different probing rules
  - Performance (worst case bad, usual case is really good as long a sload factor is reasonable and hash function is well-chosen)
- Related ADTs: SortedMap, Set, MultiSet