# p Ch 10 Maps, Hash Tables, Multimaps

## 10.1 Map

***Map:*** an abstract data type designed to efficiently store and retrieve values based upon a uniquely identifying search key for each.

It ***stores*** key value pairs (k,v), which we call ***entries***, where k is the key and v is its corresponding value. ***Keys are unique***.

Maps are 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.

***Applications***: address book; student-record database.

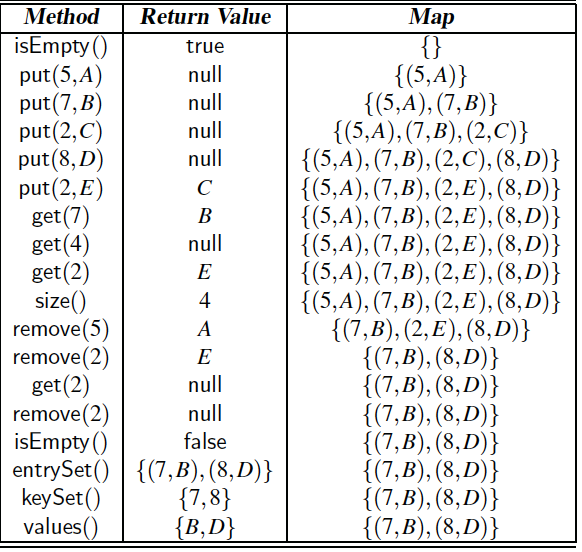
### Map ADT methods

|  |  |
| --- | --- |
| ***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. |

### Comparison to java.util.Map

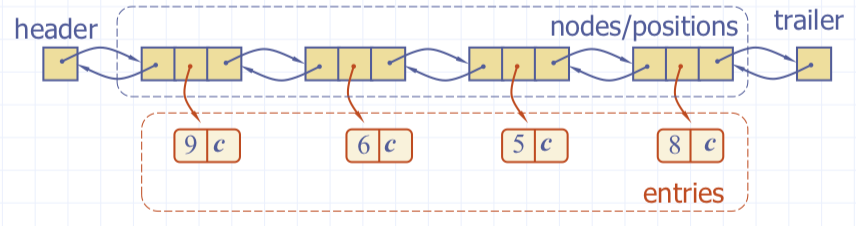
|  |  |
| --- | --- |
| ***Map ADT Methods*** | ***java.util.Map Methods*** |
| size()  isEmpty()  get(k)  put(k,v)  remove(k)  keys()  values() | size()  isEmpty()  get(k)  put(k,v)  remove(k)  keySet().iterator()  values().iterator() |

Example of methods:



### 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 doubly-linked list), in arbitrary order.



|  |
| --- |
| ***The get(k) Algorithm*** |
| **Algorithm** get(k)  B = S.positions() {B is an iterator of positions in S}  **while** B.hasNext() **do**  p = B.next() {the next position in B}  **if** p.element().key() = k **then**  **return** p.element().value()  **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().key() = k **then**  t = p.element().value()  B.replace(p, (k,v))  **return** t {return the old value}  S.insertLast((k,v))  n = n+1 {increment variable storing number of entries}  **return null** {there was no previous 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().key() = k **then**  t = p.element().value()  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} |

On a map with n entries, each of the fundamental methods takes ***O(n) time*** in the ***worst case*** because of the need to scan through the entire list when searching for an existing entry.

Each of the fundamental methods get(k), put(k, v), and remove(k) ***requires an initial scan of the array to determine whether an entry with key equal to k exists***.

**Conclusion**: unsorted list implementation is only effective for small size maps.

## 10.2 Hash Tables

Hash tables is one of the most efficient data structures for implementing a map, and the one that is used most in practice.

***Hash* function** *h* used to map general keys of a given type to corresponding indices (integers in a fixed interval [0, N − 1]) in a table.

h(x) = x mod N

integer h(x) is called the hash value of key x

***Hash table*** for a given key type consists of

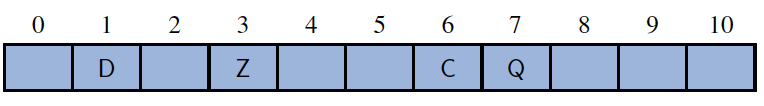
• Hash function *h* • Array (called table) of size *N*

When implementing a dictionary with a hash table, the goal is to store item (k, o) at index i = h(k)

Example

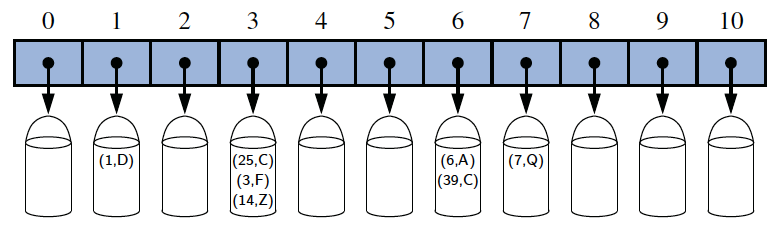
|  |  |
| --- | --- |
| * We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a ten-digit positive integer * Our hash table uses an array of size N = 10,000 and the hash function   h(x) = last four digits of x | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 10.59.57 AM.png |

***Lookup Array***



We store the value associated with key k at index k of the table (presuming that we have a distinct way to represent an empty slot). Basic map operations get, put, and remove can be implemented in O(1) worst-case time.

***Bucket Array***



### Hash functions

***collision***: If there are two or more keys with the same hash value, then two different

entries will be mapped to the same bucket in A.

A hash function, h(k), consists of two portions:

* a ***hash code map***

that maps a key k to an integer

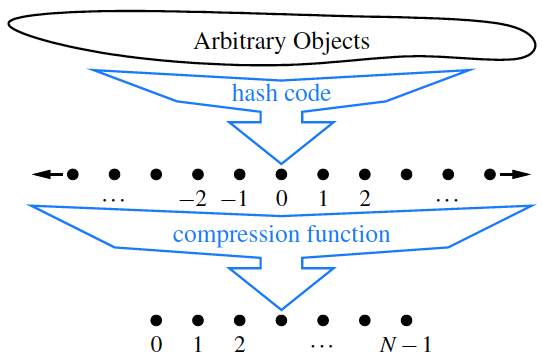
h1: keys → integers

* a ***compression function***

that maps the hash code to an integer within a range of indices, [0,N −1], for a bucket array.

h2: integers → [0, N − 1]

The hash code map is applied first, and the compression map is applied next on the result, i.e., h(x) = h2(h1(x))



The goal of the hash function is to “disperse” the keys in an apparently random way.

### Hash Code Maps

The first action is to take arbitrary key k in our map and compute an integer that is called the hash code for k; this integer need not be in the range [0,N −1], and may even be negative.

***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)
* What about strings?
* "stop", "tops", "pots", "spot" ?
* Character codes: s=115, t=116, o=111, p=112
* h("stop") = 454
* h("tops") = ?

***Polynomial accumulation***

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

a0a1 ... an−1

* We evaluate the polynomial p(z) = a0 + a1 z + a2 z2 + ... + an−1 zn−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)
* Character codes: s=115, t=116, o=111, p=112, z=33

h("stop") = 4149766

h("tops") = 4258502

h("pots") = 4262854

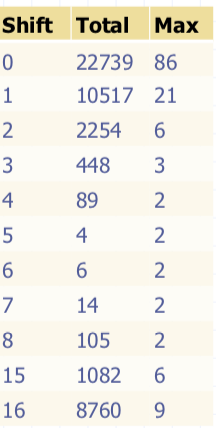
h("spot") = 4293382

* What about long strings?
* 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

p0(z) = an−1

pi (z) = an−i−1 + zpi−1(z)

(i = 1, 2, ..., n −1)

* We have p(z) = pn−1(z)

***Cyclic Shift Hash Codes***

* Instead of multiplying with some integer we can cyclically shift bits
* Bit shifts are very efficient and implement a multiplication by 2n
* Comparison of collision behaviour for cyclic shift variant of polynomial hash code applied to a list of just over 25,000 English words (table to the right)

### Compression Map

***Compression Map***: second action performed as part of an overall hash function.

A good compression function is one that minimizes the number of collisions for a given set of distinct hash codes.

Two types:

|  |  |
| --- | --- |
| ***Division Method*** | ***Multiply-Add-and-Divide Method*** |
| * Size N of hash table is usually chosen to be a prime * h2(y) = y mod N | * h2(y) = [(ay + b) mod p] mod N * a and b are nonnegative integers randomly chosen from the interval [0, p - 1], a > 0 * • p is a prime number with p > N |

### Collision Handling

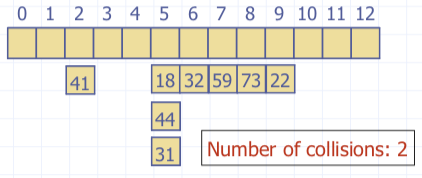
When we have two distinct keys, k1 and k2, such that h(k1) = h(k2). The existence of such collisions prevents us from simply inserting a new entry (k,v) directly into the bucket A[h(k)]. It also complicates our procedure for performing insertion, search, and deletion operations.

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

***Types to handle collision***: separate chaining, linear probing, quadratic probing, double hashing

***Separate chaining***

* let each cell in the table point to a linked list of elements that map there
* requires additional memory outside the table
* handles collisions by placing the colliding item into the map (doubly linked list) associated with the table cell
* Map size - 1 is the number of collisions for each cell



Ex:

h(x) = x mod 13

Insert keys 18, 41, 22,44,59,32,31, 73, in this order

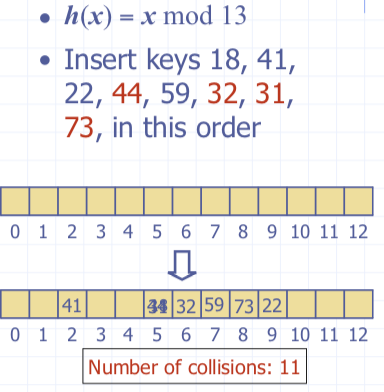
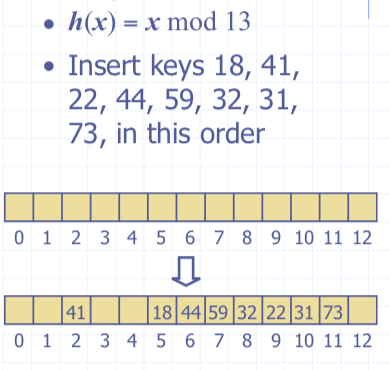
**TU T’EN FICHES DES COLLISIONS TU FAIS HASH FCT ET TU METS LES TRUCS OU ILS VONT**

***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

**TU FAIS LA HASH FCT ET SI Y A UNE COLLISION TU TRAVERSE L’ARRAY JUSQU’À QU’IL Y AIT UN ESPACE VIDE**

Example



***Search with 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 * empty cell is found or * N cells have been unsuccessfully probed | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 11.54.55 AM.png |

***Updates with Linear Probing***

To handle insertions and deletions, we introduce a special obj, called AVAILABLE, which replaces deleted elements

|  |  |
| --- | --- |
| **remove**(k)   * We search for an item with key k * If such an item (k, o) is found, we replace it with the special item AVAILABLE and we return element o * Else, we return null | **put**(k,o)   * We throw exception if table 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 AVAILABLE, or * N cells have been unsuccessfully probed * We store item (k, o) in celli |

***Quadratic Probing***

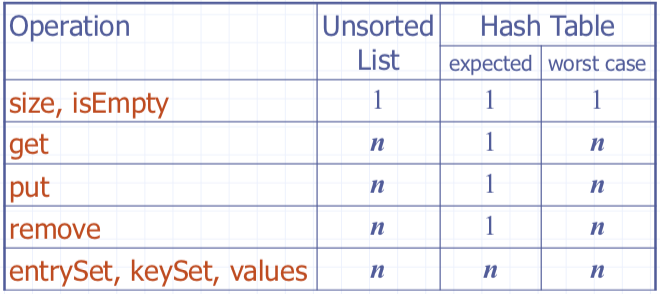
|  |  |
| --- | --- |
| Quadratic probing handles collisions by placing the colliding item in the next (circularly) available table cell of the series  (*i + j2*) mod ***N***  for *j* = 1, ... , ***N*** − 1; *i=h*(*k*)  Avoids some problems of linear probing but makes removal more complicated The table size ***N*** must be a prime and the table less than half full to allow probing of all the cells | Example:  • h(x)=xmod13  • Insert keys 18,41,22,**44**,59, **32**, **31**, 73 in this orderMacintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 12.04.46 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 12.04.51 PM.png |

***Double Hashing***

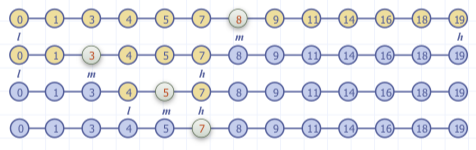
|  |  |
| --- | --- |
| * k means key * uses a secondary hash fct d(k) and handles collisions by placing an item in the first available cell of the series (i +j \* d(k)) mod N * secondary hash fct d(k) cannot have zero values * table size N must be a prime * secondary hash function:   d(k) = q – (k mod q) where q < N and q is a prime   * possible values d(k) are 1, 2, … q | ***Example***  N = 13  h(k) = k mod 13  d(k) = 7 – (k mod7)  collision for cell i = (i + j\* d(k)) modN  ***i is from h(k) and j will increment if more than 2 collisions***  Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 12.13.03 PM.png  Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 12.13.07 PM.png |

### Performance of Hashing

|  |  |
| --- | --- |
| * ***Worst case***: searches, insertions and removals on a hash table take O(n) time. It occurs when all keys inserted into dictionary collide. * ***load factor*** α = n/N affects the performance of a hash table * Assuming that hash values are like random numbers, ***expected number of probes*** for an insertion with open addressing is: 1/(1- α) | * Expected runtime of all dictionary ADT operations in hash table is O(1) * In practise, hashing is very fast provided with load factor is not close to 100% * Applications of hash tables: small databases; compilers; browser caches |



## 10.3 Multimaps



Multimap ***stores*** entries that are key-value pairs (k,v), where k is the key and v is the value. It ***allows*** multiple entries to have the ***same key***.

The multi map (ordered maps) ADT ***models a searchable collection*** of key-element entries.

The ***main operations*** of a multimap are searching, inserting, and deleting items Multiple items with the same key are allowed.

***Applications***:

word-definition pairs

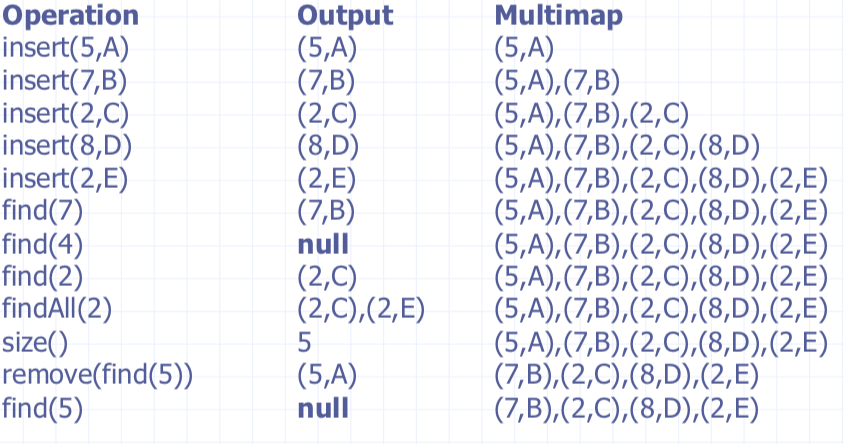
credit card authorizations

DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

### Multimaps ADT Methods

|  |  |
| --- | --- |
| **get**(k) | if the multimap has an entry with key k, returns it, else, returns null |
| **getAll**(k) | returns an iterable collection of all entries with key k |
| **put**(k,o) | inserts and returns the entry (k, o) |
| **remove**(e) | remove the entry e from the multimap |
| **entrySet**() | returns an iterable collection of the entries in the multimap |
| **keySet**() | Returns a nonduplicative collection of keys in the multimap. |
| **size**() | Returns the number of entries of the multiset (including multiple associations). |
| **isEmpty**() |  |

Example



### List-Based Multimap

***log file/audit trail***: is a multimap implemented by means of an unsorted sequence.

* We store the items of multimap in a sequence (based on a doubly-linked list or array), in arbitrary order

***Performance***:

* *puts* takes O(1) time since we can insert the new item at the beginning or at the end of the sequence
* *get* and *remove* take O(n) time since in worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

The log file is *effective only for dictionaries of small size* or *for dictionaries on which insertions are the most common operations*, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

|  |
| --- |
| ***The getAll(k) Algorithm*** |
| **Algorithm**  getAll(k)  **Input:**  a key k  **Output:** iterator of entries with key equal to k  Create an initially-empty list L  **for** e: D **do**  **if** e.getKey() = k  **then**  L.addLast(e)  **return** L |

|  |
| --- |
| ***The put(k,v) Algorithm*** |
| ***Algorithm*** put(k,v):  ***Input***: A key  ***Output***: The entry (k,v) ad  Create a new entry e = (k,v)  S.addLast(e) {S is unordered}  ***return*** e |

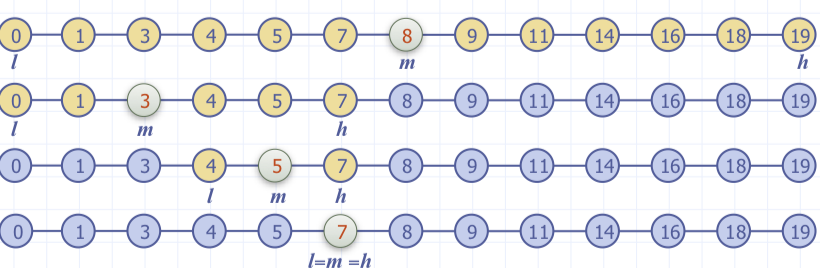
|  |
| --- |
| ***The remove(e) Algorithm*** |
| **Algorithm** remove(e)  **Input**: An entry e  **Output**: The removed entry e or null if e was not in D  {We don’t assume here that e stores its location in S}  B=S.positions()  **while** B.hasNext() do  p = B.next()  **if** p.element() = e **then**  S.remove(p)  **return** e  **return** null |

### Binary Search (ch 5.1.3 )

Binary Search performs operation **get**(k) on a multimap implemented by means of an array-based sequence, sorted by key

* similar to the high-low game
* at each step, the number of candidate items is halved
* terminates after a logarithmic number of steps

Example: **get**(7)



### Search Table

***Search table***: is a multimap implemented by means of a sorted array

* We store the items of the multimap in an array-based sequence, sorted by key
* We use an external comparator for the keys

***Performance***:

* **get** takes **O(log n)** time, using binary search
* **put** takes **O(n)** time since in the worst case we have to shift n/2 items to make room for the new item
* **remove** takes **O(n)** time since in the worst case we have to shift n/2 items to compact the items after the removal

A ***search table*** is ***effective*** only for ***dictionaries of small size*** or for ***dictionaries on which searches are the most common operations***, while insertions and removals are rarely performed (e.g., credit card authorizations or a history of logins to workstations, servers, etc.)

### Comparing Map implementations

