Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

G52ACE 2017-18

Map ADT and hashtables

Abstract Data Type: Maps

- A map models a collection of key-value entries that is searchable `by the key'
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
  - (may be allowed in a 'multi-map')

Common Applications of Maps

- Address book:
  - key=name, value=address
- Student-record database
  - key=student-ID, value=name,marks,etc
- Dictionaries:
  - key=word, value='meaning'
- Symbol table:
  - key=symbol, value='something internally useful'
  - standard component of compilers, and any program that has to read data from a file

Also known as "associative arrays" (perl, php) because they act like an array in which the index need not be an integer. Maps are very useful in many coding tasks

The Map ADT over <K,V>

Map ADT methods:

- V get(K k): if the map M has an entry with key k, return its associated value; else, return null
- V put(K k, V 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
- V remove(K k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- int size(), boolean isEmpty()
- {K} keys(): return an iterable collection of the keys in M
- {V} values(): return an iterable collection of the values in M
- {<K,V>} entries(): return an iterable collection of the entries in M

Comparison to java.util.Map

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

(No, I don't know why the textbook authors changed the names ☺)

VERY VERY IMPORTANT

Be careful not to confuse different usages of binary trees:

- **Priority Queue &**
  "Binary Tree implementations of heaps" (in other lectures)

with

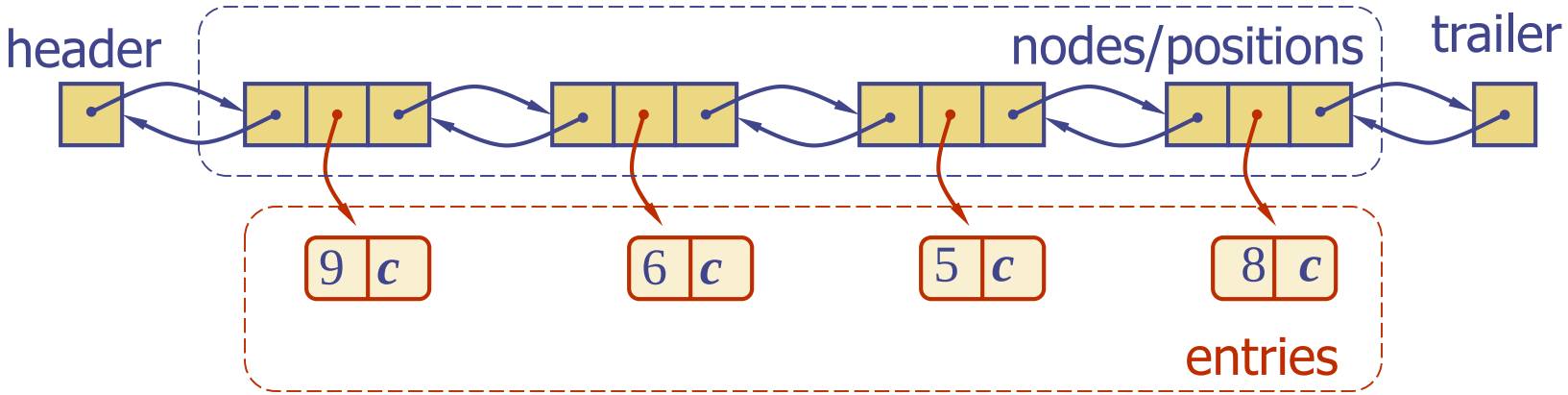
- **MAP &**
  "Binary Search Trees" (in next lecture)

Both insert entries based on a key.
The crucial difference is the way entries are accessed:

- MAP: access **any** key
- PQ: access only the **minimum** key

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 doubly-linked list), in arbitrary order
  - (and, as usual, a size counter, n)



The get(k) Algorithm

```
Algorithm get(k):
    p ← S.positions() // B is an iterator of the positions in S
    while p.hasNext() do
        p ← p.next()
        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):
p ← S.positions()
while p.hasNext() do
    p ← p.next()
    if p.element().key() = k  then
        t ← p.element().value()
        p.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):
p ← S.positions()
while p.hasNext() do
    p ← p.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
```

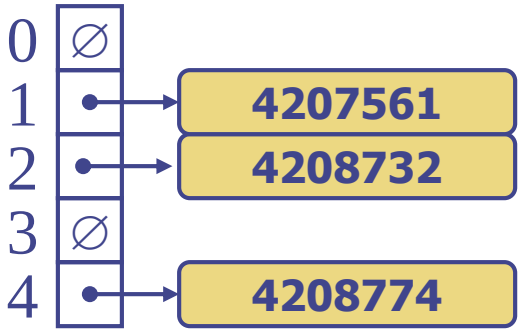
Performance of a List-Based Map

- Performance:
  - put would have taken $O(1)$ time if we could just insert the new item at the beginning (or end) of the sequence
    - but we have to check if the key occurs in the map, so it is $O(n)$.
  - 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 suitable only for maps of small size

Remark: Prototyping ADTs

- List version of a Map is
  "simple but inefficient"
- Quite common that ADTs can be implemented very easily using lists & arrays.
  - Easier to be correct
  - Even though slow, they can be useful
    - allow the rest of the project to proceed without waiting for the efficient version
    - can be used for a regression test of a "faster but trickier" later version
    - e.g. use in a "shadow mode" where the "slow-obviously-correct" version is used together with the "fast-possibly-buggy" version and the results checked against each other. (Might be relevant to G52GRP, etc).

Hash Tables (Hash maps)



Hash Tables

- Hash tables are a concrete data structure which is suitable for implementing maps.
- **Basic idea: convert each key into an index into a (big) array.**
- Look-up of keys and insertion and deletion in a hash table usually runs in O(1) time.

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:
    $h(k) = k \bmod N$
    is a hash function for integer keys
- The integer $h(k)$ is called the hash value of key k

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, v) at index $i = h(k)$

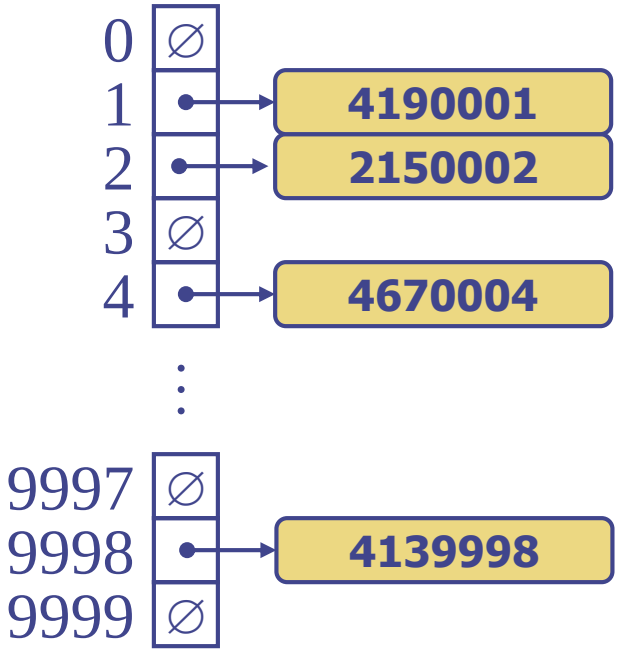
Example

- We design a hash table for a map storing entries as (SID, Name), where SID (student identity number) is a seven-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

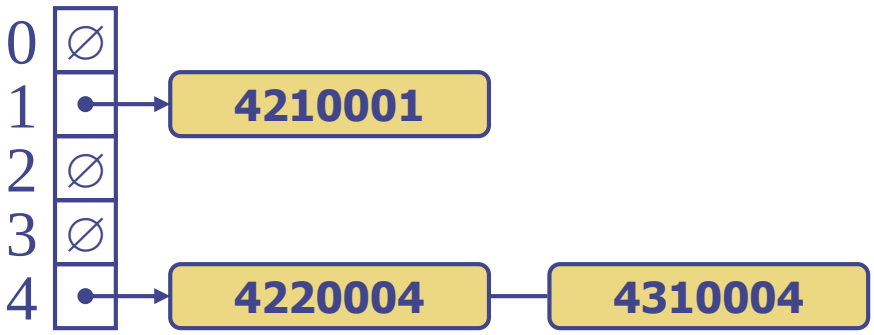
  = *"x mod 10000"*

  (details depends if SID is stored as an int or a string)



Collision Handling

- Collisions occur when different elements are mapped to the same cell
- A lot of the theory and practice of hashing consists of devising better ways to avoid or handle collisions



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 functions: “dispersal”?

- The goal of the hash function is to “disperse” the keys in an apparently random way
- Exercise (online):
  - Why disperse?
  - Why random?

Hash functions: “dispersal”?

- Why disperse?
  - to reduce numbers of collisions
- Why random?
  - random means ‘no pattern’
  - if there is an obvious pattern then the incoming data might have a matching pattern that leads to many collisions
  - “sometimes ‘no pattern’ is the only safe pattern” (e.g. rock-paper-scissors game)

Hash Codes [Not assessed]

- 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.) [Not assessed]

- 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)

- Or, representing a string as a number on base z
- Compare with base 10:

$365 = 3*10^2 + 6*10^1 + 5*10^0$

- Base 27 (26 characters + blank):

$cab = 3*27^2 + 1*27^1 + 2*27^0$

where

a = 1, b = 2, c = 3 and z = 26

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime

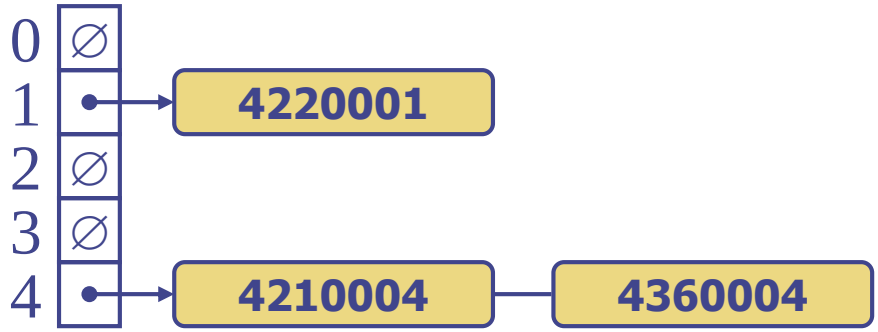
  (hash codes will be spread better)

- 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

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to (e.g.) a linked list of entries that map there
  - Note: In practice, should use a more efficient Map; e.g. a Binary Search Tree (BST), such as a "red-black tree" (see later lectures)



Map Methods with Separate Chaining used for Collisions

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

Algorithm get(k):

Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map

return $A[h(k)]$.get(k)

{delegate the get to the list-based map at $A[h(k)]$}

Map Methods with Separate Chaining used for Collisions

Algorithm put(k,v):

Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**

$t \leftarrow A[h(k)]$.put(k,v)
{delegate the put to the list-based map at $A[h(k)]$}

if t = **null then** {k is a new key}
    $n \leftarrow n + 1$
return t

Map Methods with Separate Chaining used for Collisions

Algorithm remove(k):

Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map

$t \leftarrow A[h(k)]$.remove(k)

{delegate the remove to the list-based map at $A[h(k)]$}

if $t \neq$ **null then** {k was found}

$n \leftarrow n$ - 1

return t

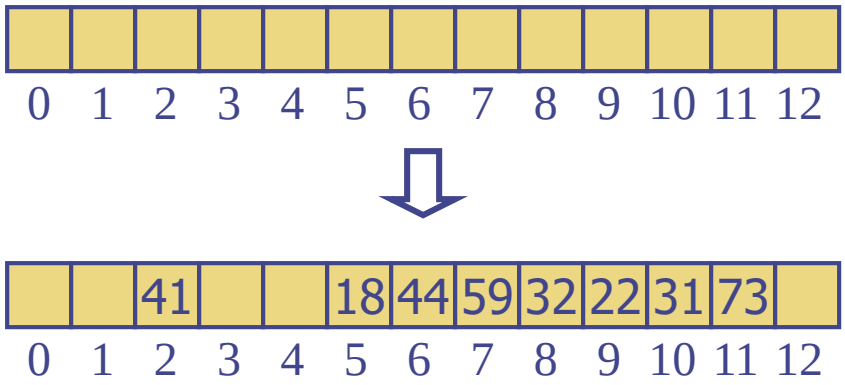
Separate Chaining

- Separate chaining is simple and fast, but requires additional memory outside the table.
- When memory is critical then try harder to stick with using the existing memory:

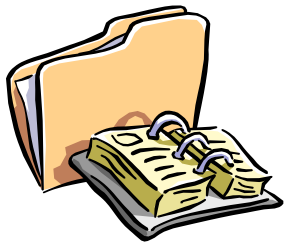
Open addressing

- 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 (variant: cell + c where c is a constant)
- Each table cell inspected is referred to as a "probe"
- Disadvantage: 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



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.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Exercise

- How do you remove an element?

Exercise

- How do you remove an element x?
- One answer:
  - (not the best answer, but simplest):
  - Find x using `get' and set the entry back to blank, i.e. null or empty (which sometimes write as `#')
  - Fix the sequence on its right-hand-side
    - WHY!?: If any entry on the right used linear probing then it might no longer be discoverable by `get' because it will stop at the blank!!!
    - Fix: Move such entries, e.g. by removing them and then re-inserting them all.
  - **EXERCISE (offline)** Figure out the details and write pseudo-code for this and do examples. (Ask in labs if needed!)

Exercise (online)

How do you remove an element x?

- "**Lazy deletion**": don't mark the entry as a blank, but as a 'deleted' and fix the entries later. E.g. see
- http://opendatastructures.org/versions/edition-0.1e/ods-java/5_2_LinearHashTable_Linear_.html
- E.g. in the find then skip over a 'deleted' entry rather than stopping

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
- Linear probing is just $d(k)=1$
- 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:

  $$d(k) = q - (k \bmod q)$$

  where
  - $q < N$
  - q is a prime
- The possible values for $d(k)$ are

  $$1, 2, \ldots, q$$

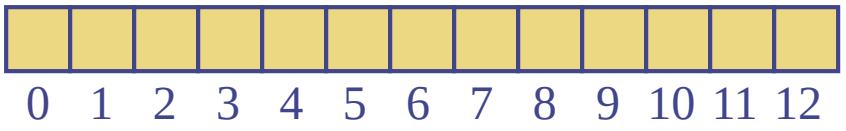
Remarks

- "The table size ***N*** must be a prime to allow probing of all the cells"
- E.g. consider d(k) = 4
  - With N=12 then the positions are 4,8,0,4, …
  - With N=11 then the positions are 4,8,1,5,9,2, …
- With a prime N, then eventually all table positions will be reached

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

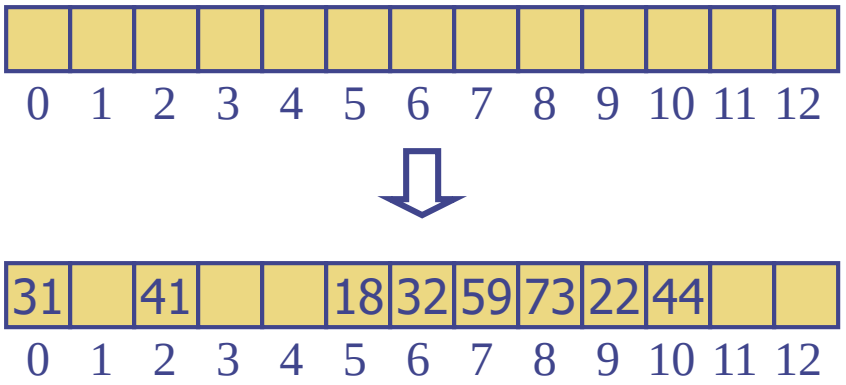
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



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

- 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
- In Java, maximal load factor is 0.75 (75%) – after that, rehashed
  - as for Vector, it is good to double the table size each rehash
- 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%

Re-Hashing

When the table gets too full then "re-hash":

Create a new larger table and new hash function.

- Need to (eventually) transfer all the entries from the old table to the new one
- If do so immediately, then
  - one can amortise the cost over many entries (as for Vector) and so get an average cost of O(1) again
  - but the worst case might be O(n) when the table is rehashed and this might be bad for a real time system
  - Option:
    do not transfer all entries "in one go" but do "a few at a time"
  - Keep both tables until the transfer is complete; but only do insertions into the new table.

Exercise (offline): consider this in more detail, and read the relevant part of the text book (Section 9.2.7 Load factors and Reshashing) or the wiki page http://en.wikipedia.org/wiki/Hash_table#Dynamic_resizing

Applications of Hashing

- Direct applications of hash tables:
  - small databases
  - compilers
  - browser caches – the weird and wonderful filenames in the browser cache folder are hashcodes of something?
- Hash tables as an auxiliary data structure in a program:
  - Look-up table: if you want to check whether some object has been seen before, for example in a graph or list traversal, keep a hashtable of (object,"seen before") pairs, where the key is the reference to the object, and the value is some arbitrary marker.

Comparison of HashMap and PQ

- HashMap does not use the ordering of keys
  - E.g. does not implement min()
  - In a hash table it would need a scan of all the keys in the table, so O(n) (or worse)
- PQ does not allow direct access to a key
  - E.g. there is no easy way to do get(k)
  - In a (standard) heap we would have to walk through all the entries

Minimum Expectations

- Map ADT, and its usage
- basic concepts of hash tables
- hash codes and compression functions
- Options to handle collisions
  - Separate chaining, linear probing, double hashing
  - How to insert, find/get, remove for all these systems

GoTa Textbook:

- study Sections 9.1 and 9.2
- do some relevant "Reinforcement" exercises of section 9.6