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

COMP2054-ADE

Map ADT and hashtables

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

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.
  - Not guaranteed, and design of the table needs to be done carefully if want the access to be "reliably O(1)"

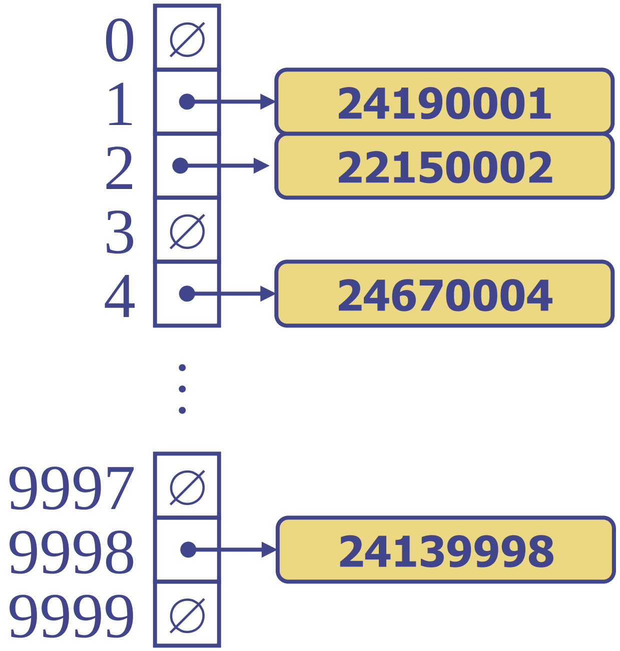
Hash Functions and Hash Tables

- 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 $i = h(k)$

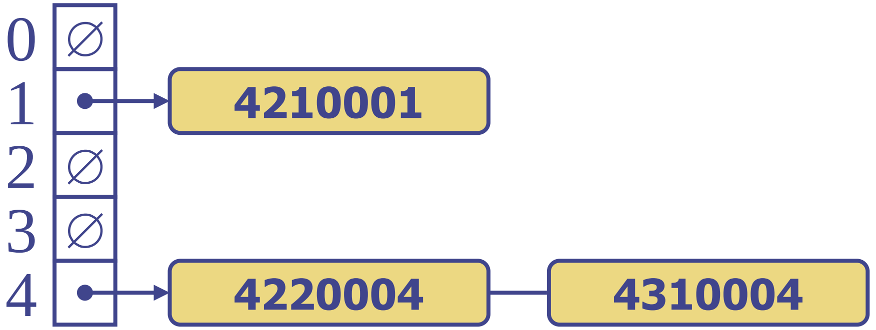
Example

- We design a hash table for a map storing entries as (SID, Name), where SID (student identity number) is an eight-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $\boldsymbol{h}(\boldsymbol{x})$ = last four digits of $\boldsymbol{x}$
  = *"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:

  $\boldsymbol{h}_1$: keys $\rightarrow$ integers

  Compression function:

  $\boldsymbol{h}_2$: integers $\rightarrow [0, N-1]$

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

  $$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$$

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

Hash functions: “dispersal”?

- Re: The goal of the hash function is to “disperse” the keys in an “apparently random” way
- Questions:
    - 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
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
    $$\ldots + a_{n-1} z^{n-1}$$
    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

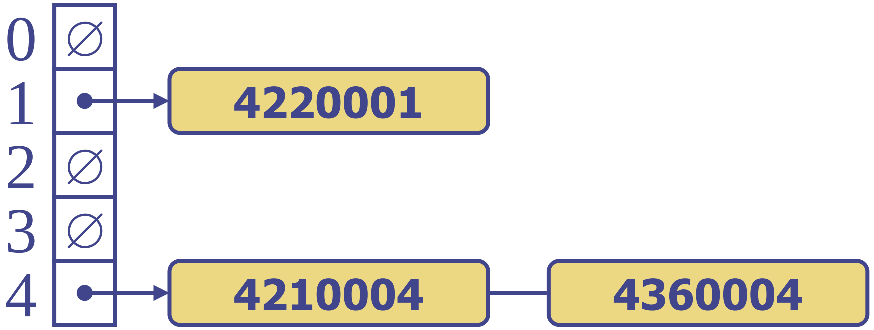
  (hash codes will tend to spread better)

- 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), (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)

// Simply delegates 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)

// Simply delegates 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)

// Simply delegates 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

Access is still O(n), but usually the relevant n is much smaller, because it usually builds many small lists, e.g. length n/N on average.
So this method might be okay, sometimes.

Separate Chaining

- Separate chaining is simple and fast, but requires additional memory outside the table.
- When memory is critical then we try harder to remain within 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)
  - “Circular array” – once get to the right-hand end, then just start again at the beginning of the array
- 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

Open addressing : Example

- Example: $h(x) = x \bmod 13$, c=1
  - Insert keys 18, 41, 22, 44, 59, 32, 5 in this order
  - 18,41,22 have no collisions, giving

		41			**18**				**22**			

- But 44mod13=5 collides with 18mod13=5, so use index 5+1

		41			**18**	**44**			**22**			

- Now 59mod13=7 has no collisions

		41			**18**	**44**	**59**		**22**			

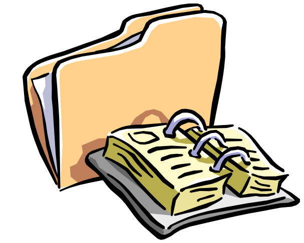
- Now 32mod13=6 collides with the 44, and so walk past the 44 and 59 to find an empty space

		41			**18**	**44**	**59**	**32**	**22**			

- Now 5mod13=5 has a long walk to find an empty space:

		41			**18**	**44**	**59**	**32**	**22**	**5**		

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)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *null*
        else if $c.key\,() = k$
            return $c.element()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return *null*

Open addressing : Example

- Example: $h(x) = x \bmod 13$, c=1
  - Insert keys 18, 41, 22, 44, 59, 32, 5 in this order
  - Gave

		41			**18**	**44**	**59**	**32**	**22**	**5**		

- Suppose we now remove(22).
  It seems we should just simply obtain:

		41			**18**	**44**	**59**	**32**		**5**		

  - Do you see a problem?
  - Suppose we try to do get(5)
  - get(5) would fail to find the 5
  - as the scan stops at the empty cell where 22 used to be

Exercise

- How do you safely 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/tutorials if needed!)

Exercise

How do you safely remove an element x?
Another solution:

- "**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, skip over a 'deleted' entry rather than stopping

(See Tutorials)

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 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

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

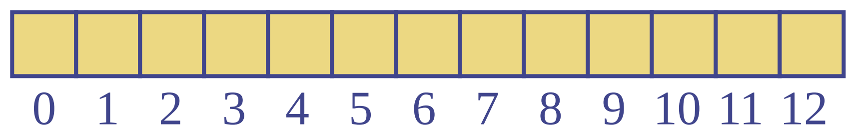
Remarks

- "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 only positions scanned are 4,8,0,4, …
    - So we miss many cells that might have space
  - With N=11 then the positions are 4,8,1,5,9,2, …
- With a prime N, then eventually all table positions will be probed

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		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

- Exercise: do the details yourself, and see tutorials

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 may be good to "roughly double" the table size each rehash
    - pick a new (prime) close to twice the current size
- 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