Hash Tables

- Hash tables
- Hash functions
- Collisions, separate chaining, open addressing (three strategies)
- Complexity of insertion, deletion and search in a hash table
- Java hash tables.

Hash Table ADT

- Logical domain:
  collection of pairs (key, value). Key is used to access the value, just as an array index is used to access the corresponding element in the array. Array indices have to be of type int. Hash table keys can be anything.

Hash Table ADT

Methods (assume store ItemTypes, in Java would be Objects):

- **`void put(key, value)`**
  pre: key is a valid key
  post: the pair is inserted in the hash table. If there was another value associated with this key before, it is replaced by the new value.
- **`ItemType get(key)`**
  pre: pair (key, value) is in the table for some value
  post: the value is returned

Hash Table ADT

- **`boolean containsKey(key)`**
  pre: key is valid
  post: returns true or false depending on whether there is an item with this key in the table
- **`boolean contains(value)`**
  post: returns true or false depending on whether there is such a value in the table
- **`void remove(key)`**
  pre: key is valid
  post: removes item with the specified key

Hashing

- General idea:
  - get a large array
  - design a *hash function h* which converts keys into array indices
  - insert (key, value) in the array at *h(key)* position
  - can also be used with data stored in external memory (addresses instead of array indices).

Example: English dictionary

- Suppose there are 50 000 words in the dictionary.
- How do we convert words into array indices?
- ASCII code: letter - integer
  a 97
  b 98
  c 99
  Words: ?

Naive encoding

- Represent words as numbers on base 256 (number of symbols).
- Recall decimal representation:
  - digits: 0,1,2,...9
  - number 602 = $6 * 10^2 + 0 * 10^1 + 2 * 10^0$
- So, "cab" can be represented as

$99 * 256^2 + 97 * 256^1 + 98 * 256^0$.

Problems with naive encoding

- Even with a more sensible base (0 to 26) will get VERY large numbers for indices!
- We provide a unique encoding for all possible combinations of letters, not just meaningful words. And there are infinitely many such combinations.
- We don't know what the largest index will be (the longest English word?)
- Each word has a unique index, but there is no array large enough to have all the indices.

How to fit an array of size k

- How to map arbitrary large numbers to numbers in the range *0.....k-1*?
- A possible solution: division hashing.
  index = largeNumber % k
  (remainder of division by *k*).
  However, even this is not very practical in our case (largeNumber is too large).
- Another problem is that the same index may now be associated with different words.

Requirements for Hash Functions

The following are general requirements for hash functions:

- if the hash table has size *k*, hash function should return numbers in the range *0....k-1*.
- hash function must be quick to calculate
- hash function should minimise *collision*s (when several keys are hashed to the same number).
- The latter requirement is satisfied if each key is equally likely to map to any of the *k* indices (uniform distribution). Usually we cannot completely avoid collisions (as in the example with words).

Perfect Hashing

In exceptional circumstances, can avoid collisions completely:

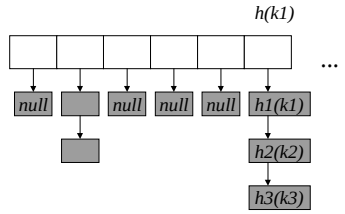
- for example when the entire set of values is available in advance and is not going to change (as with a particular edition of the dictionary). Then can design a ***perfect*** hash function for that particular collection.

Handling collisions

- open addressing: probe array for the "next" slot which is still empty. Different methods for finding the "next" slot:
  - linear probing
  - quadratic probing
  - double hashing
- separate chaining: each slot in the array contains a reference to a bucket of items with the keys hashed to that slot. Usually a linked list.

Separate Chaining

If *h(k1)=h(k2)=h(k3):*



Complexity of Separate Chaining

- The worst case: all keys hashed to the same slot.
- Items kept in a linked list of size N (size of the collection), search O(N).
- Best case: hash function distributes the keys uniformly, the size of the bucket is N/k where N is the number of items and k the capacity (number of buckets). Usually N<k.

Open addressing

- compute the index given the key
- go to the index
- if it's free, insert the item
- if it is not free, go to the next possible slot.

To find the next possible slot, generates a "probing sequence".

Probing Sequence

- A sequence of indices in the array where should try to insert the item (first try, second try, etc.)
- Different ways to generate this sequence:
  - linear probing
  - quadratic probing
  - double hashing

Linear Probing

- Search sequentially for the next free slot.
- The simplest probing sequence is

  $h(k), h(k) + 1, h(k) + 2, \ldots$

but in general can do

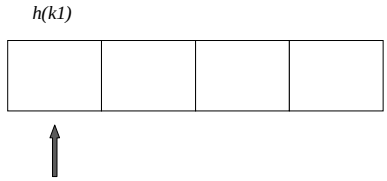
$h(k), h(k) + c, h(k) + 2c, h(k) + 3c, \ldots$

where c is some constant.

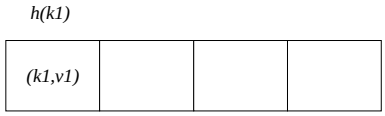
Problem: clustering

- If several keys are mapped to the same index, we get large clusters of occupied cells with empty spaces in between. The larger the cluster, the longer the search.
- Suppose *k1*, *k2* and *k3* are all mapped to the same index. Then searching for/inserting *(k1,v1)* is OK, inserting *(k2,v2)* takes an extra step, inserting *(k3,v3)* takes extra two steps etc.

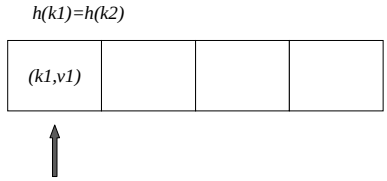
Example: insert *(k1,v1)*



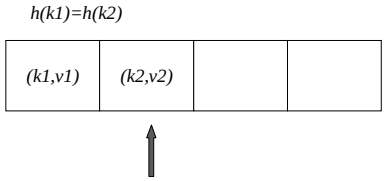
Example: insert *(k1,v1)*



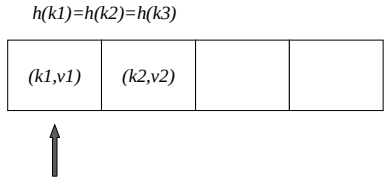
Example: insert *(k2,v2)*



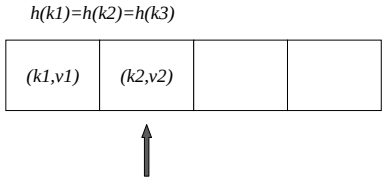
Example: insert(k2,v2)



Example: insert *(k3,v3)*

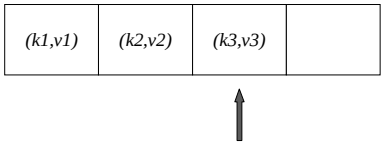


Example: insert *(k3,v3)*



Example: insert *(k3,v3)*

h(k1)=h(k2)=h(k3)



Quadratic probing

- The offset is squared at every step.
- Jumps around much more, but the items with the same keys make exactly the same jumps.
- Secondary clustering: each new item with the same hash value requires a longer probing sequence

Double Hashing

- Offset is calculated by a second hashing function *h'*, which is different from the first one:

h(k), h(k) + h'(k), h(k) + 2h'(k), h(k) + 3h'(k), ...

- Solves the problem of clustering.

Drawbacks of hash tables

- If too full, performance may deteriorate to linear.
- Not ***well*** suited for enumerating all elements kept in the table. If too empty, iteration inefficient (lots of gaps in the array).

Drawbacks of hash tables

- Expensive to expand once the table is created. Need to:
  - modify the hash function (if old h gave values between 0 and k-1, now need h' with the range of values between for example 0 and 2k - 1).
  - create a larger array (say 2k -1)
  - rehash all items from the old table into the larger one (for each item, compute h' and insert into the new table).

Useful rules for the table size

Unless it is possible to map each key to a unique index in the array,

- if modulo division or linear probing is used, the table size should be a prime number
- if open addressing is used, should be about twice larger than the number of values stored. Optimal ***load factor*** = ***size/capacity*** for open addressing is 1/2 (the table twice larger than the number of items).

Useful rules for the table size

- For separate chaining, default load factor is usually 3/4, or 75% .
- Java's HashMap is resized and rehashed if the load factor exceeds 3/4. The capacity (number of buckets) is roughly doubled.

Java hash tables

- Java hash table classes (HashMap, HashTable or HashSet) use separate chaining. Buckets are searched sequentially.
- Each Object has a default hashCode() method; the hash value is determined by the object's address. User-defined classes may overwrite it (usually not worth bothering).
- If you are using HashMap, HashTable or HashSet for your coursework, you may assume that insertion, deletion and search are constant time, and iteration is linear in the number of items.

Summary

- Hash tables are very useful when the size of collection is known in advance and search, insertion and deletion should be fast.
- If the table is large enough and the hash function is chosen well, these operations are almost constant.
- Difficult to resize and to iterate through all items (array contains gaps).