

COMP20012

2: Hash Tables

Maps

The Map ADT is used when information needs to be stored and accessed via a *key*

The key is often a string, but could be any Java Object.

For example:

- Dictionaries
- Symbol Tables
- Associative Arrays (eg in perl, awk) In perl we can use

```
$room_no{"Gough"} = "Kilburn 2.115";
```

- Associated with each key is some *data*
- The Map is essentially a *set* of (key,data) pairs, with the property that, for any k, d, d' , if (k, d) and (k, d') are in the set, then $d = d'$.
- In other words a Map provides a function $Key \longrightarrow Data$.
- For example the key might be a word and the data its dictionary definition, or the key an employee National Insurance number and the data the employment record associated with that employee.

The Map Interface

Object	<code>put(K key, V value)</code>	Associates the specified value with the specified key in this map (optional operation).
Object	<code>get(Object key)</code>	Returns the value to which this map maps the specified key.
boolean	<code>containsKey(Object key)</code>	Returns true if this map contains a mapping for the specified key.
boolean	<code>containsValue(Object value)</code>	Returns true if this map maps one or more keys to the specified value.
boolean	<code>equals(Object o)</code>	Compares the specified object with this map for equality.

<code>boolean</code>	<code>isEmpty()</code>	Returns true if this map contains no key-value mappings.
<code>void</code>	<code>putAll(Map<? extends K> t)</code>	Copies all of the mappings from the specified map to this map (optional operation).
<code>Object</code>	<code>remove(Object key)</code>	Removes the mapping for this key from this map if present (optional operation).
<code>void</code>	<code>clear()</code>	Removes all mappings from this map (optional operation).
<code>int</code>	<code>hashCode()</code>	Returns the hash code value for this map.
<code>int</code>	<code>size()</code>	Returns the number of key-value mappings in this map.

Set	<code>entrySet ()</code>	Returns a set view of the mappings contained in this map.
Set	<code>keySet ()</code>	Returns a set view of the keys contained in this map.
Collection	<code>values ()</code>	Returns a collection view of the values contained in this map

Implementing Maps

Maps can be implemented in many ways

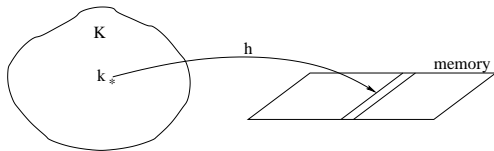
- Linked lists of (key,data) pairs
- Tree based representations
 - Ordered binary trees of (key,data) pairs
 - Balanced trees, eg 2-3 trees, Red-Black trees, B-trees, Splay trees
- Hash tables

Hash tables

- When storing and accessing indexed information, most efficient access mechanism is *direct addressing*.
- Simplest example is an array a
- Can access, or update, any array element $a[i]$ in constant time, if we know the index i .

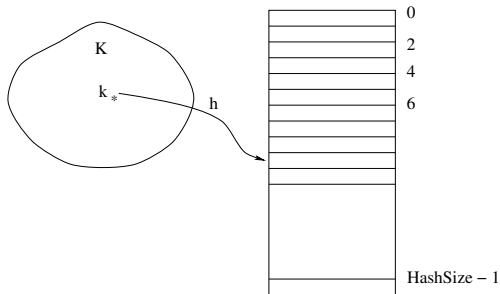
- If data is stored in other forms of data structure, such as lists, or binary search trees, the access and update cost depends on position of the data in the data structure.
- Arrays ok for data indexed by integers
- Need similar mechanisms for data with other types of index, e.g. strings of characters

- Suppose we are given a set K of keys.
- For *direct addressing* need a function h , such that, for any $k \in K$
 $h(k)$ is the address of the location of data associated with k



- h would ideally have the following properties:-
 - It is easy (and fast) to calculate $h(k)$, given k
 - It is 1-1 (i.e. no two keys give the same value of h)
 - Uses reasonable amount of memory (i.e. the range of h is not too large)

- All these properties are satisfied by array indexing, but in general are too much to ask for.
- *Hash tables* give a way round the problem by relaxing the 1-1 condition.
- A hash table is basically an array



- A *hash function* is a function

$$h : K \longrightarrow (0 \dots HashSize - 1)$$

- h is used whenever access to the hash table is required – e.g. search, insertion, deletion

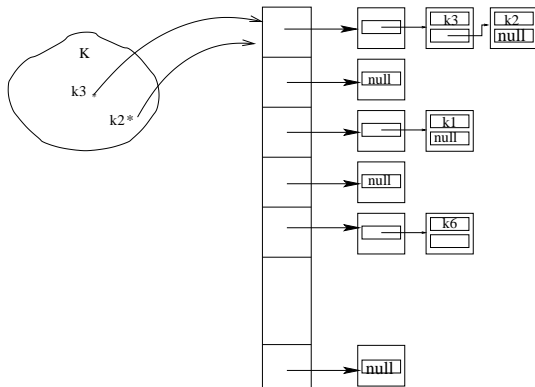
- A problem arises when two keys k_1, k_2 have

$$h(k_1) = h(k_2)$$

- Where does data associated with these keys go?
- Two mechanisms for resolving this type of conflict
 - *Open hashing* (otherwise known as *separate chaining*)
 - *Closed hashing* (otherwise known as *open addressing*)

Open hashing

- No data is stored in the hash table itself
- Hash table just contains pointers to linked lists of data cells



- Have drawn cells containing only a key and a reference to the next list node.
- The cells would normally contain references to the values associated with the keys.

The principal operations on hash tables are to initialise, find using a key and insert new data, again using a key.

- Initialise a hash table, given its size
 - Create array (of appropriate size) of Lists
 - Initialise each List object.

- Find a key k
 - Calculate $h(k)$
 - look along the list with header at $h(k)$ to see if k is there
- Insert a key k
 - If already there, do nothing
 - Otherwise, add a new list element containing k to the list starting at $h(k)$

- Delete a key k
 - If not there, do nothing
 - Otherwise, delete cell containing k from the list starting at $h(k)$.

Cost of hashing

Search time for a key k has two components

- Calculating $h(k)$
- Searching the linked list for k

Ignoring, for time being, calculation of $h(k)$, search time depends on

Maximum length of linked lists

= Maximum number of collisions for any k

Aim to choose size of hash table and hash function so that this is as small as possible

Hash functions

If hash table size is H , then need a function

$$h : K \longrightarrow 0 \dots (H - 1)$$

Should have properties

- It is easy to compute
- For given values in K , values of $h(k)$ are uniformly distributed over the range $0 \dots (H - 1)$

For example, no use having $H = 1000$ and all values of $h(k)$ in the region 200 to 300.

These properties are difficult to achieve, since the keys themselves are often not randomly distributed

Example hash functions for strings

Simple additive function. Adds character values (as ints) and reduces mod the table size.

```
static int hash1( String key)
{
    int hashVal=0;

    for (int i = 0 ; i < key.length() ; i++)
        hashVal += key.charAt(i);

    return( hashVal );
}
```

This function takes no account of the position of characters within the key

e.g. 'cat' and 'act' hash to the same value, $99 + 97 + 116 = 312$

This function weights the second and third characters with powers of the size of the alphabet (in this case 27). It ignores all characters after the third.

```
static int hash2( String key )
{
    int hashVal;
    if ( key.length() == 0 )
        hashVal = 0;
    else if ( key.length() == 1 )
        hashVal = key.charAt(0);
    else if ( key.length() == 2 )
        hashVal = (key.charAt(0) + 27*key.charAt(1));
    else
        hashVal = (key.charAt(0) + 27*key.charAt(1) +
                    729*key.charAt(2));
    return(hashVal);
}
```

With this one we have

cat: 87282

act: 87334

catalogue: 87282

We can generalise this approach to use the entire string.
For a key of length n , calculate the polynomial

$$k_0 * C^{n-1} + k_1 * C^{n-2} + \dots k_{n-1}$$

where the value of C is 32, and $k_0, \dots k_{n-1}$ are the characters of the key.

We can do this in an efficient way by rewriting the polynomial in the following way:-

$$(\dots ((k_0 * C) + k_1) * C) + k_2 + \dots k_{n-2}) + k_{n-1}$$

An example of this evaluation strategy is

$$3C^3 + 5C^2 + 2C + 7 = ((3C + 5)C + 2)C + 7$$

Only 3 multiplications are used instead of $3 + 2 + 1 = 6$

```
static int hash3( String key )
{
    int hashVal=0;
    int c=32;
    for (int i = 0 ; i < key.length() ; i++)
    {
        hashVal = ( hashVal * c ) + key.charAt(i);
    }
    return(hashVal);
}
```

or equivalently

```
static int hash3alt( String key, int hashSize )
{
    int hashVal=0;

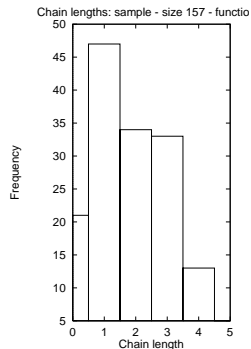
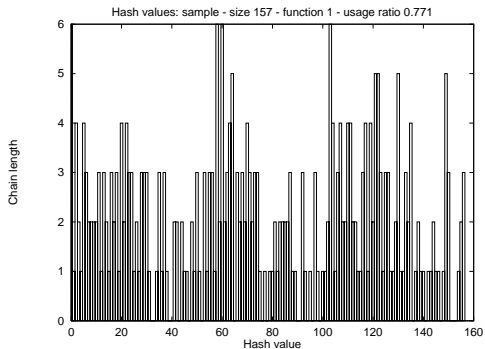
    for (int i = 0 ; i < key.length() ; i++)
    {
        hashVal = ( hashVal << 5 ) + key.charAt(i);
    }
    return(hashVal);
}
```

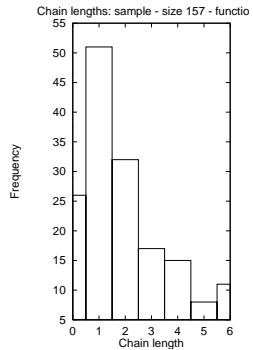
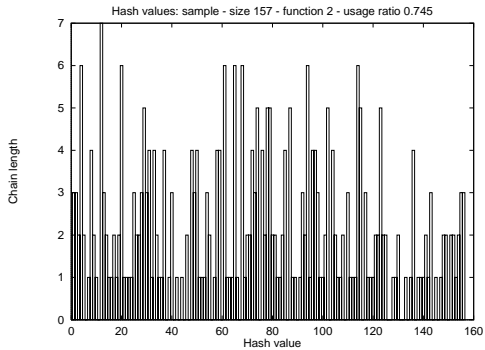
The Java method `hashCode` in the class `String` is defined in a similar way to `hash3`, except that the constant used is 31, rather than 32.

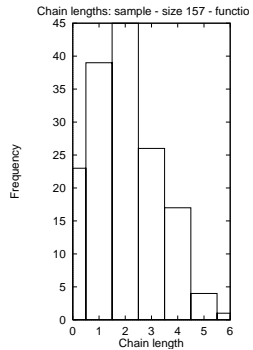
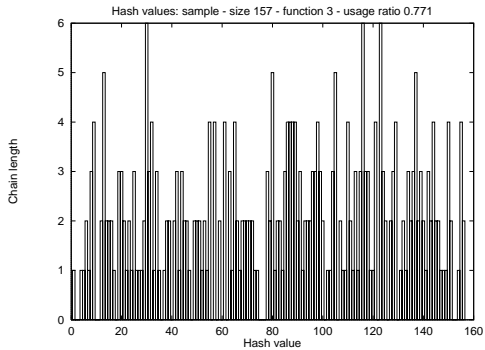
Hash function performance

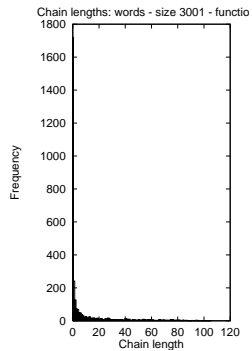
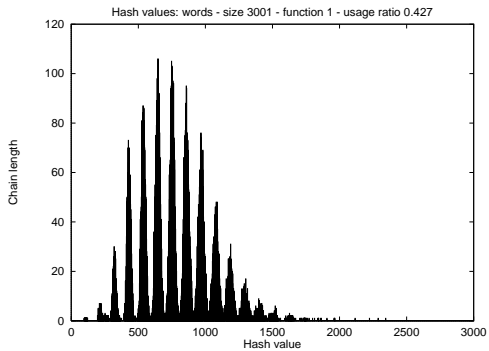
The following slides show the performance of the above hash functions under two sets of conditions

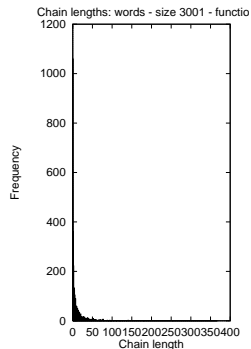
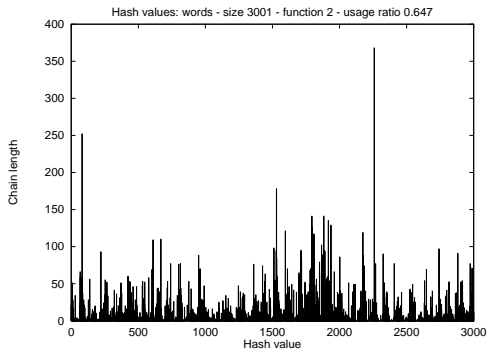
- Data set of 315 words, hash table size 157
- Data set of 25144 words (`/usr/share/dict/words`), hash table size 3001

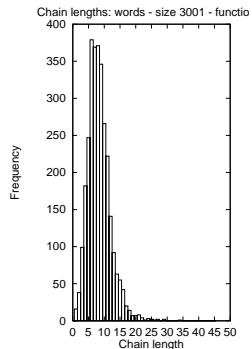
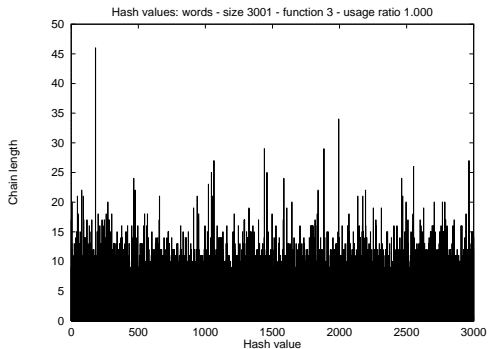












Closed Hashing (Open addressing)

Open hashing has several disadvantages, including

- needs separate data structure for chains, and code to manage it

Closed hashing has neither of these problems

- all data is stored in the hash table array itself
- when collision occurs, look elsewhere in the array for space

Given key k , hash function $hash$, adopt following procedure to search for k .

- Look at the location s_0 given by $hash(k)$
- If k found at location s_0 , return s_0
- Otherwise, if location s_0 is empty, halt search, since k is not in the table
- Otherwise, use second function p to find s_1 , given by

$$s_1 = hash(k) + p(1) \pmod{HashSize}$$

- Repeat above with

$$s_i = hash(k) + p(i) \pmod{HashSize}$$

until either find k or an empty cell.

- The function p called *collision resolution function* or *probe function*
- Simplest function is

$$p(i) = i$$

So $s_i = \text{hash}(k) + i$

- Start at s_0 and look at consecutive cells until find space.
- This is *linear probing*

Hash table insertion – linear probing

This example uses integer keys and the hash function

$$h(k) = k \bmod \text{hash_size}$$

Data to be inserted is

23 3 2 22 45 68 24 47 91

Using *linear probing* gives

insertions ---->

	23	3	2	22	45	68	24	47	91
0	23	23	23	23	23	23	23	23	23
1					45	45	45	45	45
2			2	2	2	2	2	2	2
3		3	3	3	3	3	3	3	3
4						68	68	68	68
5							24	24	24
6								47	47
7									91
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22				22	22	22	22	22	22

Everything OK until try
to insert 45.

$$45 \bmod 23 = 22 \quad \text{full}$$

$$22 + 1 \bmod 23 = 0 \quad \text{full}$$

$$22 + 2 \bmod 23 = 1 \quad \text{insert}$$

- Note that this produces a block of occupied cells – this effect is known as *clustering*.
- Using linear probing means that if a new key is hashed *into the cluster*, we need several probes to find space for it, and it then adds to the cluster.
- One way to avoid this is to use *quadratic probing*.

$$p(i) = i^2$$

Hash table insertion – quadratic probing

Same data and hash function as before, but using *quadratic probing*

0		23		23		23		23		23		23		23	
1										24		24		24	
2						2		2		2		2		2	
3				3		3		3		3		3		3	
4															
5												47		47	
6															
7															
8								45		45		45		45	
9															
10															
11															
12														91	
13															
14															
15										68		68		68	
16															
17															
18															
19															
20															
21															
22								22		22		22		22	

Again everything OK until
try to insert 45.

$45 \bmod 23 = 22$ full
 $(22 + 1^2) \bmod 23 = 0$ full
 $(22 + 2^2) \bmod 23 = 3$ full
 $(22 + 3^2) \bmod 23 = 8$ insert

Quadratic probing

Linear probing is *guaranteed* to find an empty space if one exists.

Not the case for quadratic probing

Consider the following hash table. It isn't full, but an attempt to insert 30, using hash function as before and quadratic probing, will fail.

=====		$30 \bmod 10 = 0$	full
0 10		$(0 + 1^2) \bmod 10 = 1$	full
1 21		$(0 + 2^2) \bmod 10 = 4$	full
2		$(0 + 3^2) \bmod 10 = 9$	full
3		$(0 + 4^2) \bmod 10 = 6$	full
4 34		$(0 + 5^2) \bmod 10 = 5$	full
5 45		$(0 + 6^2) \bmod 10 = 6$	full
6 56		$(0 + 7^2) \bmod 10 = 9$	full
7			
8			
9 69			
=====			

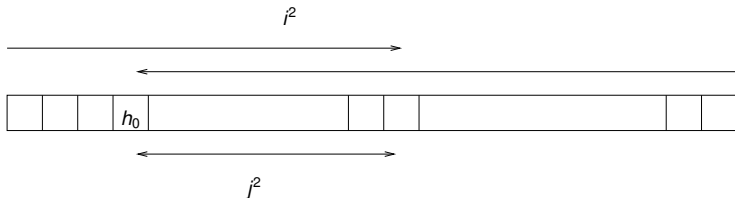
Never get any values other than 0, 4, 5, 6, 9, so
never hit free space

Can be shown that can guarantee to find a place for a new key k using quadratic probing if following conditions hold

- The table size is prime
- The table is less than half full

This means that, if quadratic probing used, need table size to be at least twice as big as maximum number of data items

- Suppose table has size H , H an odd prime greater than 3.
- Claim that the first $\lfloor H/2 \rfloor$ alternative locations probed by quadratic probing are different
- Suppose we hit the same location for probes i and j , where $i \geq j$, and h_0 is the initial starting point given by the hash function



- So we have

$$\begin{aligned}h_0 + i^2 &= h_0 + j^2 \pmod{H} \\ i^2 &= j^2 \pmod{H} \\ (i-j)(i+j) &= 0 \pmod{H}\end{aligned}$$

- This means that $(i-j)(i+j)$ is a multiple of H .
- Since H is prime, one of $i-j$, $i+j$ must be divisible by H .
- Since both i and j are $\leq \lfloor H/2 \rfloor$, we must have $i+j < H$
So $i-j = 0$.

- This justifies the claim that the first $\lfloor H/2 \rfloor$ alternative locations probed are different
- If the table is less than half full, at least one of these $\lfloor H/2 \rfloor$ locations must be free.

Double hashing

- Although quadratic probing eliminates primary clustering there is still a *secondary clustering* effect, in that if two keys have the same hash values, they result in probing the same sequence of slots.
- One way of eliminating this is by using *double hashing*.
- Choose a second hash function, h_2 , and use the collision resolution function

$$p(i) = h_2(k) * i$$

- Need to make sure that $h_2(k)$ never takes the value 0 (Why?)

If the table size is prime and p is some prime smaller than the hash table size, the function

$$h_2(k) = p - (k \bmod p)$$

usually works well. (Note that we are assuming integer keys here, but can easily be generalised to string keys)

Using double hashing with the above example, with secondary hash function

$$p - (k \bmod p)$$

with $p = 7$

Can insert 10, 21, 34, 45, 56, 69 with no problems – now try to insert 70 and 19

=====									
0		10		10		10			
1		21		21		21			
2									
3						19			
4		34		34		34			
5		45		45		45			
6		56		56		56			
7				70		70			
8									
9		69		69		69			
=====									

						$h_2(70)$	=	$7 - (70 \bmod 7) = 7$	
						$70 \bmod 10$	=	0	full
						$(0 + 7) \bmod 10$	=	7	insert

						$h_2(19)$	=	$7 - (19 \bmod 7) = 2$	
						$19 \bmod 10$	=	9	full
						$(9 + 2) \bmod 10$	=	1	full
						$(9 + 2 * 2) \bmod 10$	=	3	insert

Deletion in closed hashing

- Deleting an entry from a hash table based on closed hashing is not as straight forward as in open hashing
- Why is this?

- Suppose we have used linear probing and the obvious hash function to create the following table:-

=====				=====			
0		10		0		10	
1		21		1		21	
2		31		2			
3		41		3		41	
4		44		4		44	
5		71		5		71	
6				6			
7				7			
8				8			
9				9			
10				10			
=====				=====			

- Now delete 31 and then search for 71.
- If we just delete, we come across an empty space which suggests our search is unsuccessful, even though 71 is there.

- To get round this problem we need to flag each entry in the table to say whether it is deleted or not.
- Only if we find an empty slot which is not flagged as deleted do we end a search

Rehashing

- Two major disadvantages to closed hashing
 - Need to know upper bound on amount of data before building the hash table
 - As the hash table fills up, more clashes occur and insertion becomes more expensive. At some stage insertion will fail (at best when the table is full).
- Rather than make the hash table extremely large in first place, can get round both problems by *rehashing*.
- This involves building a new hash table at least double the size of original, inserting all data from the original hash table into the new table, and freeing memory resources taken by old table.

- This means that we are not limited by size of data expected in advance,
- Can rehash if run out of space.
- Rehash can take place either
 - When becomes full – probably a bit late
 - When table loading reaches some predetermined factor (70% often used)
- Rehashing should be triggered automatically by an insert, without user intervention.
- Easy to implement

The lab exercise: A simple spell checker

- First lab session week of April 8.
- The exercise is to implement a very simple spelling checker which uses a set in which to store all the words in a dictionary.
- A text is then read and words which are in the text which can not be found in the dictionary are reported to the user, together with the line numbers on which these words occur in the text.

- The dictionary is to be contained in a collection which implements the following very simplified collection interface

```
public interface Coll
{
    public boolean add(Object o);
    public boolean contains(Object o);
    public void printColl();
}
```

- The behaviour of the `add` and `contains` methods should be the same as the corresponding methods in `Set`. `printColl` should print the contents of collection to standard output in a suitable manner.

Tasks

- Your tasks for this lab are to provide two *different* implementations of this interface, both using *Hash Tables*, one using *Open Hashing* and the other using *Closed Hashing*.
- You should then implement a class `SpellCheck` which uses either of these implementations to produce a simple spell checker, as described above.

Task 1: Open Hashing based implementation

- The first task is to create a class `HashSetOpen` which implements the `Coll` interface using hash tables based on *open hashing*, so that collisions are dealt with by using a list of entries.
- You can probably reuse a modified form of your `LinkedList` code from lab 1 for list handling purposes.
- In addition to the methods required for the interface you will need suitable constructors and a `main` method which includes code to test your methods.

Task 2: Closed Hashing based implementation

- The second task is to write a call `HashSetClosed`, which implements the `Coll` interface type using hash tables based on *closed hashing*, so that collisions are dealt with by using a collision resolution mechanism, such as *linear probing*.
- You could alternatively use quadratic probing or double hashing.
- For this style of hash table you will also need to implement a `rehash` method, which is invoked by `add` when the the table reaches a predetermined load factor, otherwise the table risks overfilling.
- Your implementation should, of course, include suitable testing.

- For both types of hash table it is preferable to use hash tables whose size is a prime number.
- A class `Primes` is provided in `/opt/info/courses/lab3/Primes.java`, to help you identify suitable prime numbers. In particular the method `nextPrime` returns the first prime number greater than or equal to its argument.

Task 3: The Spell Checker

- The final task is to write a class `SpellCheck` which uses either of these implementations to produce a simple spell checker, as described above. The text file to be checked should be specified via a command line argument, as should the `Coll` implementation to be used and the dictionary.

- One possible example of how this might be used is:-

```
java SpellCheck -o -d /usr/share/dict/words test-text
```

This should use the dictionary `/usr/share/dict/words` and the open hashing based hash table implementation to spell check the file `test-text`.

- The simpler command

```
java SpellCheck test-text
```

might use a default dictionary and text file.

- One way to switch between the two implementations is to have a `SpellCheck` constructor which takes an argument of type `Coll` which is then used to store the dictionary.
- This can then be used in conjunction with a method to do the spell check, which can be invoked by that object.

- To populate the dictionary and also to read the text to be checked, you need to read a file and identify the words in it.
- For this purpose we regard a word as a sequence of consecutive alphabetic characters. Any non-alphabetic character can be regarded as a separator.
- You may find the `StringTokenizer` class useful here.

Take a look in the directory

`/opt/info/courses/COMP20012/lab3/.`, where there is a sample dictionary and text file to be spell checked, together with a file containing some examples of command line handling.