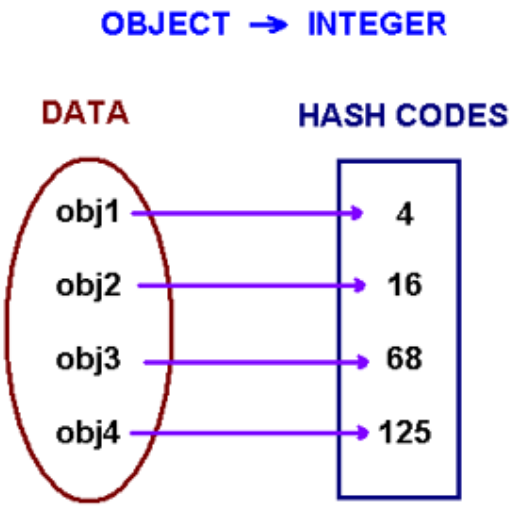
Concept of Hashing

Introduction

The problem at hands is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function**. A hash function is a function which when given a key, generates an address in the table.



The example of a hash function is a *book call number*. Each book in the library has a *unique* call number. A call number is like an address: it tells us where the book is located in the library. Many academic libraries in the United States, uses Library of Congress Classification for call numbers. This system uses a combination of letters and numbers to arrange materials by subjects.

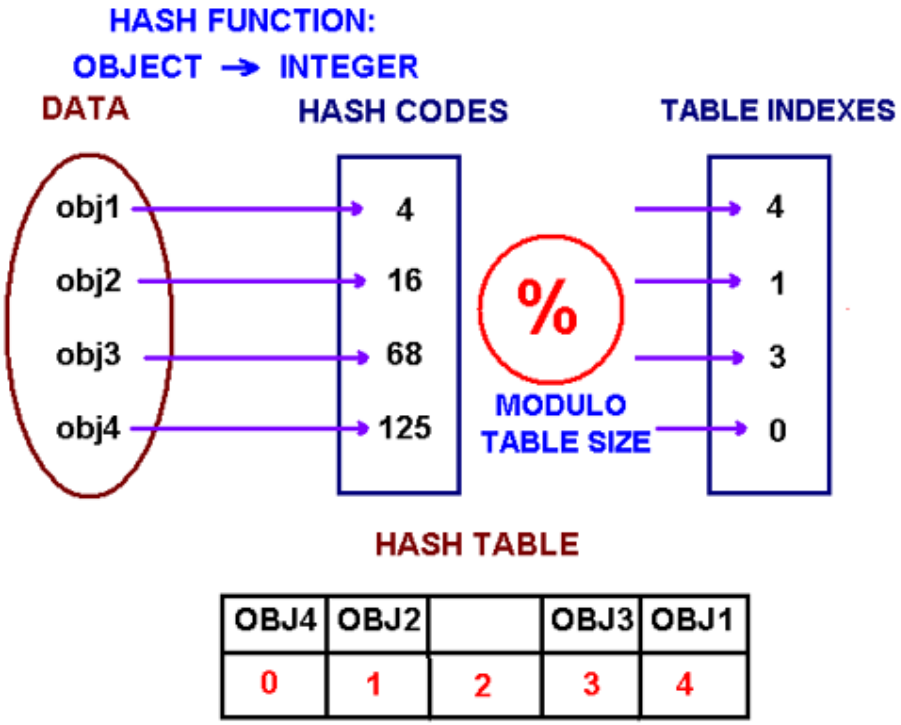
A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The later is always possible only if you know (or approximate) the number of objects to be proccessed.

Thus, we say that our hash function has the following properties

- it always returns a number for an object.
- two equal objects will always have the same number
- two unequal objects not always have different numbers

The precedure of storing objets using a hash function is the following.

Create an array of size *M*. Choose a hash function *h*, that is a mapping from objects into integers *0, 1, ..., M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. Such array is called a **hash table**.



How to choose a hash function? One approach of creating a hash function is to use Java's *hashCode()* method. The hashCode() method is implemented in the Object class and therefore each class in Java inherits it. The hash code provides a numeric representation of an object (this is somewhat similar to the toString method that gives a text representation of an object). Conside the following code example

```
Integer obj1 = new Integer(2009);
String obj2 = new String("2009");
System.out.println("hashCode for an integer is " + obj1.hashCode());
System.out.println("hashCode for a string is " + obj2.hashCode());
```

It will print

```
hashCode for an integer is 2009
hashCode for a string is 1537223
```

The method hasCode has different implementation in different classes. In the String class, hashCode is computed by the following formula

$$s.charAt(0) * 31^{n-1} + s.charAt(1) * 31^{n-2} + ... + s.charAt(n-1)$$

where *s* is a string and *n* is its length. An example

$$"ABC" = 'A' * 31^2 + 'B' * 31 + 'C' = 65 * 31^2 + 66 * 31 + 67 = 64578$$

Note that Java's *hashCode* method might return a negative integer. If a string is long enough, its hashcode will be bigger than the largest integer we can store on 32 bits CPU. In this case, due to integer overflow, the value returned by *hashCode*can be negative.

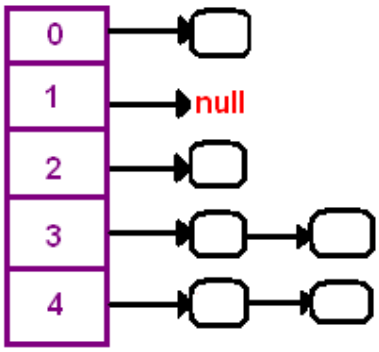
Review the code in HashCodeDemo.java.

Collisions

When we put objects into a hashtable, it is possible that different objects (by the *equals()* method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key: .

```
"Aa" = 'A' * 31 + 'a' = 2112
"BB" = 'B' * 31 + 'B' = 2112
```

How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution.



The big attraction of using a hash table is a constant-time performance for the basic operations `add, remove, contains, size`. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list, that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that our lists won't become too long. This is usually implemnted by maintaining a *load factor* that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and *rehash* all elements from the old table into the new one.

Another technique of collision resolution is a *linear probing*. If we cannoit insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on. This is quite simple approach but it requires new thinking about hash tables. Do you always find an empty slot? What do you do when you reach the end of the table?

HashSet

In this course we mostly concern with using hashtables in applications. Java provides the following classes HashMap, HashSet and some others (more specialized ones).

HashSet is a regular set - all objects in a set are distinct. Consider this code segment

```
String[] words = new String("Nothing is as easy as it looks").split(" ");

HashSet<String> hs = new HashSet<String>();

for (String x : words) hs.add(x);

System.out.println(hs.size() + " distinct words detected.");
System.out.println(hs);
```

It prints "6 distinct words detected.". The word "as" is stored only once.

HashSet stores and retrieves elements by their content, which is internally converted into an integer by applying a hash function. Elements from a HashSet are retrieved using an Iterator. The order in which elements are returned depends on their hash codes.

Review the code in HashSetDemo.java.

The following are some of the HashSet methods:

- set.add(key) -- adds the key to the set.
- set.contains(key) -- returns true if the set has that key.
- set.iterator() -- returns an iterator over the elements

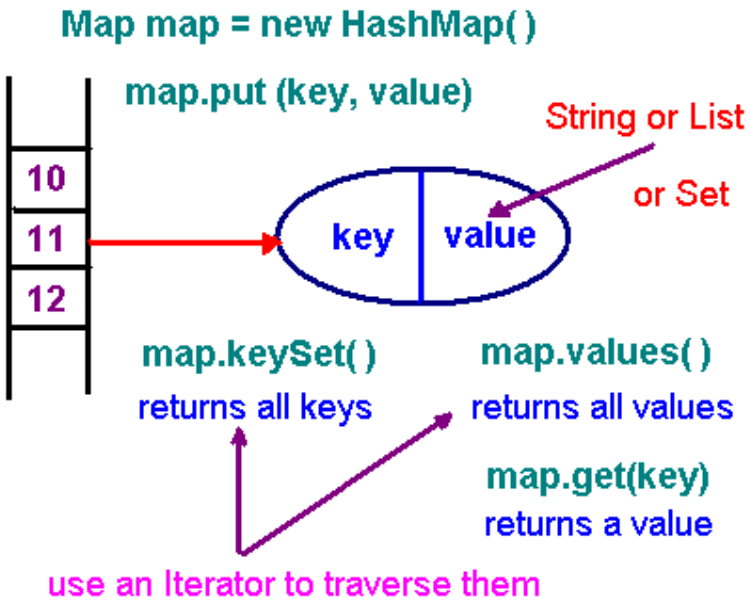
Spell-checker

You are implement a simple spell checker using a hash table. Your spell-checker will be reading from two input files. The first file is a dictionary located at the URL *http://www.andrew.cmu.edu/course/15-121/dictionary.txt* . The program should read the dictionary and insert the words into a hash table. After reading the dictionary, it will read a list of words from a second file. The goal of the spell-checker is to determine the misspelled words in the second file by looking each word up in the dictionary. The program should output each misspelled word.

See the solution here Spellchecker.java.

HashMap

HashMap is a collection class that is designed to store elements as key-value pairs. Maps provide a way of looking up one thing based on the value of another.




```
Iterator itr = map.keySet().iterator();
while( itr.has.Next())
{
   Object key = itr.next();
   System.out.println(map.get(key));
}
```

We modify the above code by use of the HashMap class to store words along with their frequencies.

```
String[] data = new String("Nothing is as easy as it looks").split(" ");

HashMap< String, Integer> hm = new HashMap< String, Integer>();

for (String key : data)
{
  Integer freq = hm.get(key);
  if(freq == null) freq = 1; else freq ++;
  hm.put(key, freq);
}
System.out.println(hm);
```

This prints `{as=2, Nothing=1, it=1, easy=1, is=1, looks=1}`.

HashSet and HashMap will be printed in no particular order. If the order of insertion is important in your application, you should use *LinkeHashSet* and/or *LinkedHashMap* classes. If you want to print dtata in sorted order, you should use *TreeSet* and or *TreeMap* classes

Review the code in SetMapDemo.java.

The following are some of the HashMap methods:

- map.get(key) -- returns the value associated with that key. If the map does not associate any value with that key then it returns null. Referring to "map.get(key)" is similar to referring to "A[key]" for an array A.
- map.put(key,value) -- adds the key-value pair to the map. This is similar to "A[key] = value" for an array A.
- map.containsKey(key) -- returns true if the map has that key.
- map.containsValue(value) -- returns true if the map has that value.
- map.keySet() -- returns a set of all keys
- map.values() -- returns a collection of all value

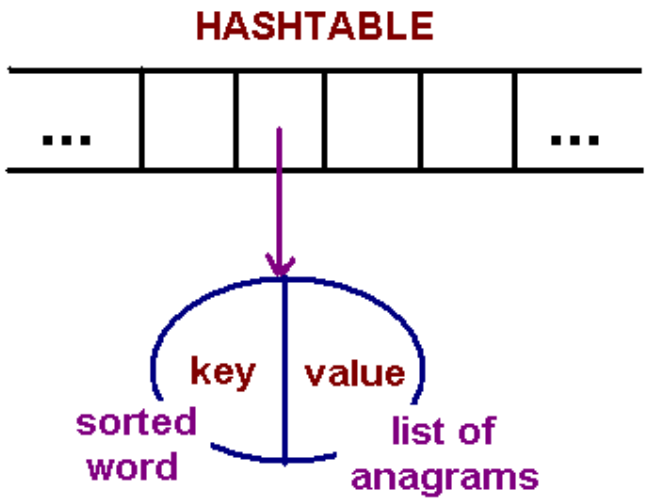
Anagram solver

An anagram is a word or phrase formed by reordering the letters of another word or phrase. Here is a list of words such that the words on each line are anagrams of each other:

```
barde, ardeb, bread, debar, beard, bared

bears, saber, bares, baser, braes, sabre
```

In this program you read a dictionary from the web site at *http://www.andrew.cmu.edu/course/15-121/dictionary.txt* and build a **Map( )** whose key is a sorted word (meaning that its characters are sorted in alphabetical order) and whose values are the word's anagrams.



See the solution here Anagrams.java.

Priority Queue

We are often faced with a situation in which certain events/elements in life have higher or lower priorities than others. For example, university course prerequisites, emergency vehicles have priority over regular vehicles. A Priority Queue is like a queue, except that each element is inserted according a given priority. The simplest example is provided by real numbers and $\leq$ or $\geq$ relations over them. We can say that the smallest (or the largest) numerical value has the highest priority. In practice, priority queues are more complex than that. A priority queue is a data structure containing records with numerical keys (priorities) that supports some of the following operations:

- Construct a priority queue
- Insert a new item.
- Remove an item.with the highest priority
- Change the priority
- Merge two priority queues

Observe that a priority queue is a proper generalization of the stack (remove the newest) and the queue (remove the oldest).

Elementary Implementations

There are numerous options for implementing priority queues. We start with simple implementations based on use of unordered or ordered sequences, such as linked lists and arrays. The worst-case costs of the various operations on a priority queue are summarized in this table

	insert	*deleteMin*	*remove*	*findMin*	*merge*
`ordered array`	*N*	1	*N*	1	*N*
`ordered list`	*N*	1	1	1	*N*

unordered array	1	*N*	1	*N*	*N*
unordered list	1	*N*	1	*N*	1

Later on in the course we will see another implementation of a priority queueu based on a binary heap.

Comparable and Comparator interfaces

The Comparable interface contains only one method with the following signature:

```
public int compareTo(Object obj);
```

The returned value is negative, zero or positive depending on whether this object is less, equals or greater than parameter object. Note a difference between the equals() and compareTo() methods. In the following code example we design a class of playing cards that can be compared based on their values:

```
class Card implements Comparable<Card>
{
   private String suit;
   private int value;

   public Card(String suit, int value)
   {
      this.suit = suit;
      this.value = value;
   }
   public int getValue()
   {
      return value;
   }
   public String getSuit()
   {
      return suit;
   }
   public int compareTo(Card x)
   {
      return getValue() - x.getValue();
   }
}
```

It is important to recognize that if a class implements the Comparable interface than compareTo() and equals() methods must be correlated in a sense that if `x.compareTo(y)==0`, then `x.equals(y)==true`. The default equals() method compares two objects based on their reference numbers and therefore in the above code example two cards with the same value won't be equal. And a final comment, if the equals() method is overriden than the hashCode() method must also be overriden, in order to maintain the following properety: if `x.equals(y)==true`, then `x.hashCode()==y.hashCode()`.

Suppose we would like to be more flexible and have a different way to compare cards, for example, by suit. The above implementation doesn't allow us to do this, since there is only one compareTo method in Card. Java provides another interface which we can be uses to solve this problem:

```
public interface Comparator<AnyType>
{
   compare(AnyType first, AnyType second);
}
```

Notice that the compare() method takes two arguments, rather than one. Next we demonstrate the way to compare two cards by their suits, This method is defined in its own class that implements Comparator:

```
class SuitSort implements Comparator<Card>
{
   public int compare(Card x, Card y)
   {
      return x.getSuit().compareTo(  y.getSuit() );
   }
}
```

Objects that implement the Comparable interface can be sorted using the sort() method of the Arrays and Collections classes. In the following code example, we randomly generate a hand of five cards and sort them by value and then by suit:

```
String[] suits = {"Diamonds", "Hearts", "Spades", "Clubs"};
Card[] hand = new Card[5];
Random rand = new Random();

for (int i = 0; i < 5; i++)
   hand[i] = new Card(suits[rand.nextInt(4)], rand.nextInt(12));

System.out.println("sort by value");
Arrays.sort(hand);
System.out.println(Arrays.toString(hand));

System.out.println("sort by suit");
Arrays.sort(hand, new SuitSort() );
System.out.println(Arrays.toString(hand));
```

Objects can have several different ways of being compared. Here is another way of comparing cards: first by value and if values are the same then by suit:

```
class ValueSuitSort implements Comparator<Card>
{
   public int compare(Card x, Card y)
   {
      int v = x.getValue() - y.getValue();

      return ( v == 0) ? x.getSuit().compareTo(y.getSuit()) : v;
   }
}
```
