

## Lookup Tables and Hashing

24.1	Prologue	2
24.2	Lookup Tables	2
24.3	<i>Lab</i> : Cryptogram Solver	5
24.4	Hash Tables	8
24.5	<code>java.util</code> 's <code>HashSet</code> and <code>HashMap</code>	11
24.6	<i>Lab</i> : Search Engine	13
24.7	Summary	15
	Exercises	16

## 24.1 Prologue

In this chapter we continue our discussion of different ways to store and retrieve data in programs. But first let us briefly review what we already know about data structures.

In a list, data elements are arranged in a sequence; each element has an index assigned to it, its position in the sequence. If we want to find a given value in a list, we can do a sequential search, traversing a list from the beginning until we find the value. If we represent a list as an array, it is easy to get access to an element with a given index, but inserting and removing values at the beginning or in the middle of the list takes some time. In a linked list we have to follow the links to get access to an element with a given index, but it is easy to insert and remove elements.

Stacks and queues are used for temporary storage. In these structures, access to values is defined by the order in which they were stored. A stack provides immediate access to the value stored last, whereas in a queue values are retrieved in the order of their arrival.

BSTs (Binary Search Trees) support quick searching, similar to Binary Search in a sorted array, and relatively easy insertion and removal of values, as in a linked list.

Is there a way to store and retrieve data even faster than in a BST? The answer is yes, if we can set up a *lookup table* or a *hash table*. In the remainder of this chapter we will discuss lookup tables and hashing and get more practice with `java.util`'s `HashSet` and `HashMap` classes, which use hashing.

## 24.2 Lookup Tables

The idea of a lookup table is to avoid searching altogether: each key in a map itself tells us exactly where we can find it. The key is converted either directly or through some simple formula into an integer, which is used as an index into the lookup table array. The associated value is stored in the element of the array with that index. One special reserved value (for example, `null`) may be used to indicate that a particular slot in the table is empty — the value is not in the set. The mapping from all valid keys to the computed indices must be unambiguous, so that we can go directly to the corresponding lookup table entry and store or fetch the data.

Suppose, for example, that an application such as entering shipping orders requires a database of U.S. zip codes that would quickly find the town or locality with a given code. Suppose we are dealing with 5-digit zip codes, so there are no more than 100,000 possible values, from 00000 to 99999. Actually, only a fraction of the 5-digit numbers represent real zip codes used by the U.S. Postal Service. But in this application it may be important to make the zip code lookup as quick as possible. This can be accomplished using a table with 100,000 entries. The 5-digit zip will be used directly as an index into the table. Those entries in the table that correspond to a valid zip code will point to the corresponding record containing the locality name; all the other entries will remain unused (Figure 24-1).

0	<null>
1	<null>
...	
600	<null>
601	Adjuntas, PR
...	
1004	Amherst, MA
1005	Barre, MA
1006	<null>
1007	Belchertown, MA
1008	Blanford, MA
1009	Bondsville, MA
...	
99950	Ketchikan, AK
99951	<null>
...	
99998	<null>
99999	<null>

**Figure 24-1. A lookup table for United States zip codes**

This type of data access takes  $O(1)$  time, but some space may be wasted if many lookup table entries are empty.

Lookup tables are useful for many other tasks, such as data compression or translating one symbolic notation into another. In graphics applications and in hardware, for example, a “logical” color code (usually some number, say, from 0 to 255) can be converted into an actual screen color by fetching its red, green, and blue components from three lookup tables.

Another common use of lookup tables is for tabulating functions when we need to speed up time-critical computations. The function argument is translated into an integer index that is used to fetch the function value from its lookup table. In some

cases, when the function argument may have only a small number of integer values, the lookup table may actually take less space than the code that would be needed to implement the function! If, for example, we need to compute  $3^n$  repeatedly for  $n = 0, \dots, 9$ , the most efficient way, in terms of both time and space, is to use a lookup table of 10 values:

```
private static int[] n_thPowerOf3 =
    {1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683};
...
public int powOf3(int n)
    // precondition: 0 <= n < 10
    {
        return n_thPowerOf3[n];
    }
```

In another example, an imaging application may need to count quickly the number of “black” pixels (picture elements) in a scan line (for instance, in order to locate lines of text). In a large black and white image, pixels may be packed eight per byte to save space. The task then needs a method that finds the number of bits in a byte that are set to 1. This method can easily do the job by testing individual bits:

```
// Count the number of bits in byte b that are set to 1:
int count = 0;
if ((b & 0x01) != 0) count++; // bit 0
if ((b & 0x02) != 0) count++; // bit 1
if ((b & 0x04) != 0) count++; // bit 2
...
if ((b & 0x80) != 0) count++; // bit 7
```

But, if performance is important, a lookup table with 256 elements that holds the bit counts for all possible values of a byte (0-255) may be a more efficient solution:

```
private static int bitCounts[] =
{
    0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4, // 00000000 - 00001111
    1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5, // 00010000 - 00011111
    ...
    4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8 // 11110000 - 11111111
};

...
int i = (int)b; // Unfortunately, in Java, byte type
if (i < 0) i += 256; // values are signed: -128 <= b <= 127
int count = bitCounts[i];
```

## 24.3 Lab: Cryptogram Solver

Almost everyone is sooner or later tempted to either send a message in code or to decode an encrypted message. In a simple cryptogram puzzle, a small fragment of text is encrypted with a substitution cipher in which each letter is substituted with another letter. Something like: “Puffm, Cmafx!” To solve a cryptogram we usually look for familiar patterns of letters, especially in short words. We also evaluate frequencies of letters, guessing that in English the most frequent letters stand for ‘e’ or ‘t’, while the least frequent letters stand for ‘j’ or ‘q’. The purpose of this lab is to solve a cryptogram and to master the use of lookup tables and distributions in the process.

Our *Cryptogram Solver* program is interactive. After opening an encrypted text file, the user sees the encrypted text on the left side of the screen and the decoded text on the right. Initially, nothing is decoded — the decoded text has only dashes for all the letters. The user then enters substitutions for one or several letters, clicks the “Refresh” button, and immediately sees an updated version of the decoded text. In addition, the program can offer decoding hints based on the frequencies of letters in the encrypted text.

*Cryptogram Solver* can also create cryptograms. If you enter random substitutions for all letters (or click the “Encode” menu item) and then enter your text fragment on the left side (by typing it in, loading it from a text file, or cutting and pasting it from another program), then the text shown on the right side will be an encrypted version of your text.

Your task is to write a class `Enigma`, named after the famous German encryption machine. <sup>★enigma</sup> Enigma was invented in 1918 and was first used in the banking business, but it very quickly found its way into the military. Enigma’s codes were considered “unbreakable,” but they were eventually broken, first by Polish codebreakers in 1932 and later, during WWII, by the codebreakers of the Bletchley Park project in Great Britain, led by Alan Turing, <sup>★turing</sup> one of the founders of modern computer science. The battle between Enigma codemakers and codebreakers lasted through WWII, and the dramatic successes of Allied cryptanalysts provided invaluable intelligence information.



Your `Enigma` class should maintain a lookup table for substitutions for the letters ‘A’ through ‘Z’. The easiest way to store the table is simply in a string of text. Initially the lookup table contains only dashes. For example:

```
private static String lookupTable = "-----";
```

As decryption proceeds, the table is gradually filled and updated. You can change your guess for a letter as often as you want.

The `getNumericValue(char ch)` static method of the `Character` class returns consecutive integers for letters ‘A’ through ‘Z’ (10 for ‘A’, 11 for ‘B’, etc.; 0 through 9 are returned for digits between ‘0’ and ‘9’). Therefore, if `ch` is an upper-case letter, then

```
int i = Character.getNumericValue(ch) -
        Character.getNumericValue('A');
```

sets `i` to an integer in the range from 0 to 25, which can be used as an index into our lookup table (array or string) of 26 letters.

Your `Enigma` class should define three static methods:

```
void setSubstitutions(String subs);
String decode(String text);
String getHints(String text, String lettersByFrequency);
```

The first two methods support decoding (or encoding) of text; the last one supports computer-generated hints based on letter counts.

The `setSubstitutions(String subs)` method assumes that `subs` contains 26 characters and saves it as the lookup table.

The `decode(String text)` method decodes all the letters in `text` according to the current lookup table. `decode` leaves all characters that are not letters unchanged and preserves the upper or lower case of letters. It returns the decoded string, which has the same length as `text`.

The `getHints(String text, String lettersByFrequency)` method returns computer-generated hints for each letter in the encrypted text. It works as follows. First it counts the number of occurrences for each of the letters ‘a’ through ‘z’ in `text` (case blind) and saves these 26 counts in an array. Write a separate private method for that:

```
private static int[] countLetters(String text)
```

You should count all letters in one sweep over `text`. Start with zeroes in all counts, then increment the appropriate count for each letter.

After getting the counts for all letters, `getHints` creates and returns a `String` of “hints.” The returned string `hints` should hold a permutation of letters ‘A’ through ‘Z’; `hint.charAt(k)` will be displayed as a computer-generated hint for decoding the  $k$ -th letter of the alphabet. The hints should be based on comparing the order of letters by frequency in letter counts in encrypted text with the order of letters by frequency in plain (unencrypted) text. The `lettersByFrequency` parameter contains the letters ‘A’ through ‘Z’ arranged in increasing order of their frequencies in a sample text file. Suppose `lettersByFrequency`, passed to `getHints` is “JQXZKBVWFUYPGCLSDHROANITE”. Then ‘J’ should be the hint for the least frequent letter in `text`, ‘Q’ for the second least frequent letter, and so on. The method’s code is quite short once you figure out the algorithm. Try to figure it out yourself or read the (encrypted) hint below.

```
Mlk ze zvcyqutbf lufuvzq ty “lyqtuec ps hymetuec,” nklhqupkn ue Hbzwtkq 18. Jyq
kzhb kvkfket hymetl [d] ue tbk vkttkq hymetl zqqzs, juen tbk emfpkq yj kvkfketl
hymet [r] lmhb tbzt hymet [r] < hymet [d] yq hymet [r] == hymet [d] zen r
< d. Tbul emfpkq (hzvv ut qzed) ul tbk qzed yj tbk d-tb vkttkq yj tbk zvwbpkt ue
tkqfl yj utl jqkimkehs ue tbk kehqswtkn tkot. Ly buetl [d] lbymvn pk lkt ty
vkttkqlPsJqkimkehs.hbzqZt(qzed).
```

(The above paragraph is also available in `JM\Ch24\Cryptogram\hint.txt` file. You can use *Cryptogram Solver* on it even before you get the hints part working correctly. Just use a “stub” method for `getHints` that returns an arbitrary string of 26 characters.)

We generated our `lettersByFrequency` string by counting occurrences for the 26 letters in the file `sample.txt`. Therefore, if you load `sample.txt` (a plain text file) into the program, the hint displayed for each letter should be that same letter. This is an easy way to test your `countLetters` and `getHints` methods.

Combine your `Enigma` class with the `Cryptogram` and `CryprogramMenu` classes located in `JM\Ch24\Cryptogram`. Test your program with `sample.txt`, then try to decode `secret.txt`. Both these files are in `JM\Ch24\Cryptogram`, too.

Unfortunately, as you can see, our computer-generated hints turn out to be entirely unhelpful, except for the most frequent letter ‘e’. Apparently we need a more sophisticated tool for solving cryptograms automatically — perhaps counting 2-D distributions for all pairs of adjacent letters, or even 3-D distributions for all triplets of letters, or a way to look for other patterns in the text.

## 24.4 Hash Tables

The technique of *hashing* builds on the lookup table concept. In a lookup table, a key is either used directly or converted through a very simple formula into an integer index. Different keys translate into different indices in the lookup table. This method is not practical, however, when the range of possible key values is large. It is also wasteful when the mapping from keys to integer indices is very sparse — many lookup table entries remain unused.

We can avoid these problems by using a tighter system of mapping from keys to integer indices in the table. The purpose of the mapping is to map all possible key values into a narrower range of indices and to cover that range more uniformly. Such a transformation is called a *hash function*; a table used with it is a *hash table*.

The price of hashing is that we lose the one-to-one correspondence between keys and table entries: two different keys may be mapped into the same location in the hash table. Thus when we try inserting a new value into the table, its slot may already be occupied. These situations are called *collisions*. We have to devise some method of dealing with them. When we retrieve a value, we have to verify that its key indeed matches the target; therefore, the key must be explicitly stored in the table with the rest of the record.

The design of a hash table thus hinges upon successful handling of two problems: how to choose a good hash function and how to handle collisions. There is room for ingenious solutions for both.

A good hash function must have the following properties:

1. It must be easy to calculate.
2. It must map all possible values of keys onto a range that is not too large.
3. It must cover that range uniformly and minimize collisions.

To devise such a function, we can try some “random” things akin to the transformations used for generating random numbers in a specified range. If the key is a string of characters, we can use some numeric codes (for instance, Unicode) for them. We can then chop the key into pieces and combine these together using bitwise or arithmetic operations — hence the term “hashing.” The result must be an integer in the range from 0 to `tableSize-1`.

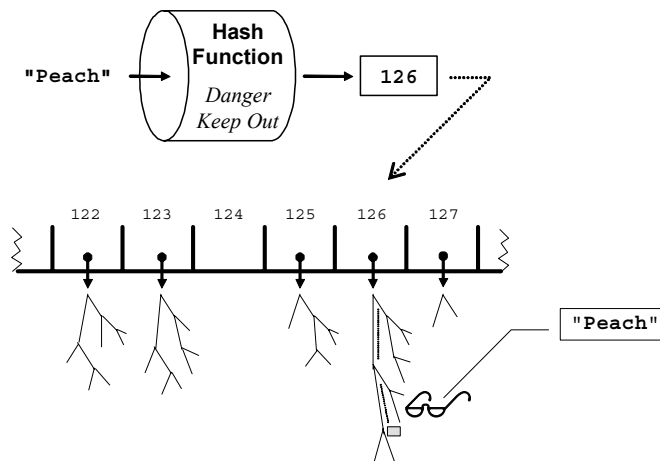
Overly simplistic hash functions, such as simply truncating the key or converting it modulo the table size —

```
public int hashCode(long key) { return (int)(key % tableSize); }
```

— may create unexpected clusters of collisions resulting from some peculiar clustering in the data. Fortunately, we can evaluate our hash function on some simulated and real data before using it in an application.



There are two principal approaches to resolving collisions. In the first approach, each entry in the hash table is itself implemented as a structure that can hold more than one value. This approach is called *chaining* and the table entry is referred to as a *bucket*. A bucket may be implemented as a linked list, a sorted array, or even a binary search tree (Figure 24-2). This approach works well for densely populated hash tables.



**Figure 24-2. Resolving collisions in a hash table by chaining**

The second approach to resolving collisions is by storing the colliding value in a different slot of the same hash table. This approach is known as *probing*. We calculate the index into the table using the hash function as usual. If the slot is already occupied, we use some *probing function* to convert that index into a new index; we repeat this step until we find a vacant slot:

```

...
int index = hashCode(target.getKey());
while (hashTable[index] != null)
    index = probe(index);
hashTable[index].setValue(target.getValue());
...

```

The same probing function, of course, must be used for retrieving a value:

```

...
int index = hashCode(key);
while (hashTable[index] != null &&
    !key.equals(hashTable[index].getKey()))
    index = probe(index);
target = hashTable[index];
...

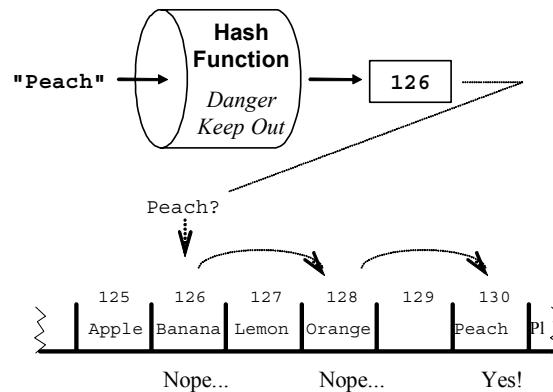
```

The simplest form of the probing function is to increment the index by one or by some fixed number:

```
int probe(index) { return (index + INCR) % tableSize; }
```

This is called *linear probing* (Figure 24-3). After the table has been in use for a while, linear probing may degrade the uniform distribution of the hash table population — a condition called *clustering*. In so-called *quadratic probing*, the sequence of examined slots is

```
index, index+1, index+4, index+9, ...
```



**Figure 24-3. Resolving collisions in a hash table by linear probing**

In more elaborate probing schemes, the next location may depend not only on the consecutive number of the attempt, but also on the value of the key. In addition, some rehash function may be used instead of `% tableSize`:

```
...
int index = hashCode(target.getKey());
int attempt = 1;
while (hashTable[index] != null)
    index = rehash(index, hashTable[index].getKey(), attempt++);
...
```

Probing should be used only with relatively sparsely populated hash tables so that probing sequences are kept short. The sequence of probing attempts required to insert a value is repeated each time we search for that value.

As we can see, the performance of a search in a hash table varies greatly with the details of implementation. In the best case, data access is  $O(1)$ . But with many collisions, the performance may deteriorate. The ratio of the number of items stored in a hash table to the number of buckets is called the hash table's *load factor*. If a load factor is too small, a lot of space is wasted. When the load factor becomes very high, all the advantages of hashing are lost. Then it may be time to rehash all the items into a new table with a larger number of buckets. A reasonable load factor may range from 0.5 to 2.0.

**One disadvantage of hash tables over a binary search tree or a sorted list is the difficulty of quickly traversing the table in ascending order of keys. This may be a serious consideration in some applications.**

## 24.5 java.util's HashSet and HashMap

**java.util's HashSet and HashMap classes resolve collisions through chaining.**

HashSet's no-args constructor creates an empty hash table with a default initial capacity (the number of buckets) of 16 and a load factor limit of 0.75. If you know in advance the maximum number of entries to be stored in the table, it is better to use another constructor —

```
HashSet(int initialCapacity)
```

— to avoid unnecessary reallocation and rehashing of the table when it runs out of space. `initialCapacity` is the number of buckets to be used in the table; it should

be roughly twice the number of entries expected to be stored. The third constructor takes two arguments, the initial capacity and the load factor limit.

The `HashMap` class provides similar constructors.

The main difference between Java's BST classes and hash table classes is that objects stored in a `HashSet` and keys mapped in a `HashMap` do not have to be `Comparable` and do not use comparators. An iterator for a hash set traverses the set's values in no particular order. Instead of the ordering among the set elements, you have to pay attention to providing a reasonable hash function for them.

In Java, when an object is stored in a `HashSet` or serves as a key in a `HashMap`, that object's `hashCode` method is called. `hashCode` is a method of the `Object` class, so every object has it. Unfortunately, this method is hardly usable because it is based on the object's current address in memory and does not reflect the object's properties. Programmers usually override `Object`'s `hashCode` in their classes.

**It is important to override `Object`'s `hashCode` method with a `hashCode` method appropriate for your class if you plan to store the objects of your class in a `HashSet` or use them as keys in a `HashMap`.**

Java library classes such as `String`, `Double`, and `Integer` have suitable `hashCode` methods defined for their objects. For example, `String`'s `hashCode` method computes the hash code as

$$s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \dots + s_{n-1}$$

where  $s_i$  is Unicode for the  $i$ -th character in the string. (This computation is performed using integer arithmetic, ignoring overflows.)

**Normally, you do not have to invent your own hash code formula for your class from scratch but can rely on the library `hashCode` methods available for one or several fields of your class.**

For example:

```
public class MsgUser
{
    ...
    private String screenName;
    ...
    public int hashCode()
    {
        return screenName.hashCode();
    }
}
```

But, no matter how simple it is, it is important not to forget to supply a `hashCode` if you plan to use your objects in a `HashSet` or a `HashMap`. In fact, it is a good idea to always supply a reasonable `hashCode` method for your class if there is any chance that you or someone else will reuse this class in a future project where its objects (or objects of a derived class) will be placed into a hash table.

`HashSet` and `HashMap` also use the `equals` method when they look for a target within a bucket.

**Your `hashCode` method must agree with `equals`, in the sense that if `x.equals(y)` is true, `x.hashCode() == y.hashCode()` must also be true.**

You don't have to worry about the range of values returned by a `hashCode` method.

**Java library `hashCode` methods return an integer from the whole integer range. `HashSet` and `HashMap` methods further map the hash code onto the range of valid table indices for a particular table (by taking it modulo table size).**

## 24.6 *Lab*: Search Engine

We have already built a couple of games, a browser, and a messaging program, but one can hardly imagine life without a search engine. In this lab we'll design and program our own miniature search engine, which we'll call "Giggle." Rather than searching the Internet for keywords or phrases, *Giggle* searches a single file for all lines of text that contain a given word. *Giggle*'s code uses lists, BSTs, and hash tables. More precisely, it takes advantage of `java.util`'s `List`, `Set`, and `Map` interfaces and `LinkedList`, `TreeSet`, and `HashMap` classes.

Before you start searching, you need to create an index of all the words in the file you are going to search. This is analogous to the indexing process in which real search engines constantly update their indices and “crawl” the web looking for new web pages. In *Giggle*, the index is a map that associates each word in a text file with a list of all lines in the file that contain that word. With a little more work, we can upgrade *Giggle* to build an index for multiple files.



As usual, I’ll supply a GUI class, *Giggle*, that loads files, accepts user input, and displays the search results. You’ll write the *SearchEngine* class that builds the index for a file and generates the search results. Your class must use a `Map` object to hold the word index. In this map, a key is a word (in lowercase letters) and the associated value is a `List<String>`. The list holds all the lines in the file that contain the corresponding keyword.

Here are the specs for your class:

Fields:

```
private String myURL;
    // holds the name for the "url" (file name)

private Map<String, List<String>> myIndex;
    // holds the word index
```

Constructor:

```
public SearchEngine(String url)
```

Saves `url` in `myUrl`; initializes `myIndex` to an empty `HashMap` with an initial capacity of 20,000. Note: this constructor does not load the file; the *Giggle* class reads the file and passes one line at a time to *SearchEngine*.

Public methods:

```
public String getURL()
```

Returns `myUrl`. I call this method from *Giggle* to display the name of the file in which hits were found. In the present version I already know the file name, but eventually an expanded version of *Giggle* may need to index several files.

```
public void add(String line)
```

Extracts all words from `line`, and, for each word, adds `line` to its list of lines in `myIndex`. This method obtains a set of all words in `line` by calling a private method `parseWords(line)` (see below). Use a `LinkedList<String>` object to represent a list of lines associated with a word.

```
public List<String> getHits(String word)
```

Returns the list of lines associated with `word` in `myIndex`.

#### Private methods:

```
private Set<String> parseWords(String line)
```

Returns a set of all words in `line`. Use the same technique for extracting all the words from `line` as you did in the *Index Maker* lab in Section <...>: call `line.split("\\W+")`. Add all the elements from the resulting array to a `TreeSet`. Skip empty words and convert each word to lower case before adding it to the set. `parseWords` uses a set, as opposed to a list, because we don't want to index the same line multiple times when the same word occurs several times in it. When we add words to the set, duplicates are automatically eliminated.

Combine your class with the `Giggle` class, located in `JM\\Ch24\\Giggle`. Test `Giggle` thoroughly on a small text file. Be sure to try searching for words that are not in the file as well as those that appear in several lines and multiple times in the same line.

## 24.7 Summary

In a *lookup table*, each key is converted through some simple formula into a non-negative integer, which is used as an index into the lookup table array. The associated value is stored in the element of the array with that index. Lookup tables can be used when the keys can be easily mapped onto integers in a relatively narrow range. All allowed keys correspond to valid indices in the table, and different keys correspond to different indices. Lookup tables provide access to data in  $O(1)$  time, but a sparsely populated lookup table may waste a lot of space.

In the *hashing* approach, a hash function converts the key into an integer that is used as an index into a hash table. Different keys may be hashed into the same index, causing *collisions*. The *chaining* technique resolves collisions by turning each slot in the hash table into a “bucket” that can hold several values. The *probing* technique stores the colliding objects in alternative slots chosen according to a predefined probing function. The performance and space requirements for hash tables may vary widely depending on the implementation. In the best scenario, data access time in a hash table is  $O(1)$ , but the performance may deteriorate with a lot of collisions. One disadvantage of hash tables over a binary search tree or a sorted list is the difficulty of quickly traversing the table in ascending order of keys.

`java.util`'s `HashSet` and `HashMap` classes implement the `Set` and `Map` interfaces, respectively, using hash tables. The objects kept in `HashSet` and the keys in a `HashMap` do not have to be `Comparable` and do not use comparators. Instead, you have to make sure that these objects have a reasonable `hashCode` method defined for them, which agrees with the `equals` method. Java library classes, such as `String`, `Double`, and `Integer`, have suitable `hashCode` methods defined. You can often write a reasonable `hashCode` method for your own class by calling library `hashCode` methods for one or several fields of your class.

A `hashCode` method may return an integer from the whole integer range. `HashSet` and `HashMap` methods further map the hash code onto the range of valid table indices for a particular table.

## Exercises

1. Define:

<i>hashing</i>	<i>chaining</i>
<i>hash function</i>	<i>bucket</i>
<i>collisions</i>	<i>probing</i>

2. Mark true or false and explain:

- (a) Access time in a lookup table of size  $n$  is proportional to  $n$  \_\_\_\_\_ ✓
- (b) Probing is feasible only when the population of a hash table is relatively sparse. \_\_\_\_\_
- (c) It is easy to traverse a hash table in ascending order of keys. \_\_\_\_\_
- (d) One of the advantages of hash tables over binary search trees is the faster access time when adding and removing data values. \_\_\_\_\_ ✓

3. A class `LookupState` implements a lookup table that helps to find full names of states from their two-letter postal abbreviations (in which both letters are uppercase). The lookup table is implemented as an array of 676 entries (676 is 26 times 26), of which only 50 are used.

```
public class LookupState
{
    private static String stateNames[] = new String[676];

    public static void add(String abbr, String name)
    {
        ...
    }

    public static String find(String abbr)
    {
        ...
    }

    private static int lookupIndex(String abbr)
    {
        ...
    }
}
```

Devise a method for mapping a two-letter state abbreviation into an integer index from 0 to 675 returned by the `lookupIndex` method, write the `add` and `find` methods, and test your class.  $\Leftarrow$  Hint: Recall from the Cryptogram lab that `Character.getNumericValue(ch)` returns consecutive integers for letters 'A' - 'Z'.  $\Rightarrow$

4. Write a method

```
public List sortByFirstLetter(List words)
```

that takes a list of words, all of which start with capital letters, and returns a new list (a `LinkedList`) in which all the words that start with an "A" are followed by all the words that start with a "B," and so on. All the words that start with a particular letter must be in the same order as in the original list. Use the following algorithm:

1. Create an array of 26 queues.
2. Scan the list once from the beginning and put each word into the appropriate queue based on its first letter.
3. Collect all the words from the queues into a new `LinkedList`.

5. The class `PhoneCall` represents a record of a telephone call:

```
public class PhoneCall
{
    ...
    public int getStartHour() { ... } // European time: 0 -- 23
    public int getStartMin() { ... }
    public int getDuration() { ... } // in seconds
    ...
}
```

Write a method

```
public int busiestHour(List<PhoneCall> dayCalls)
```

that returns the hour (a value from 0 to 23) in which the largest number of calls originated. Count only those calls that lasted at least 30 seconds. Your method must scan the list only once, using a “for each” loop or an iterator. ✓

6. A class `RecordsHashTable` implements a set of `Record` objects as a hash table —

```
private ListNode[] buckets = new ListNode[1000];
```

It resolves collisions by chaining, with buckets implemented as linked lists (references to `ListNode`).

- (a) Assuming that a class `Record` has a method `hashCode` that returns an integer from 0 to 999 and a method `equals(Object other)`, write a `RecordsHashTable`’s method

```
public boolean contains(Record record)
```

that returns `true` if `record` is found in the set, `false` otherwise.

- (b) ■ Suppose a reference takes 4 bytes and `Record` information takes 20 bytes. The average number of collisions is 5. Suppose we can convert our hash table into a lookup table by using 12 times more slots and a different `hashCode` method. Will we use more or less space? By approximately what percentage? How many times faster, on average, will the retrieval operation run, assuming that computing the old and the new `hashCode` method and comparing two records takes the same time? ✓

7. ■ A hash table has sixty entries. Devise and test a hash function for English words such that all the different words from this paragraph are hashed into the table with no more than four collisions. Do not call any `hashCode` methods.

Compare the performance of your method with the more standard  
`Math.abs(word.hashCode()) % 60`.

8. ♦ Implement *Radix Sort* for a list of words.

Radix Sort is a sorting method that is not based on comparing keys but rather on applying the lookup or hashing idea to them. Suppose we have a large list of integers with values from 0 to 9. We can create 10 buckets, corresponding to the 10 possible values. In one pass through the list we add each value to the appropriate bucket. Then we scan through the ten buckets in ascending order and collect all the values together. The result will be the list sorted in ascending order.

Now suppose we have some data records with keys that are integers in the range from 0 to 99999. Suppose memory limitations do not allow us to use 100000 buckets. The Radix Sort technique lets us sort the keys one digit at a time: we can complete the task with only 10 buckets, but we will need five passes through the list. We have to make sure that the buckets preserve the order of inserted values; for instance, each bucket can be a list with the values inserted at the end (or a queue). We start with the least significant digit in the key (the units digit) and distribute the data values into buckets based on that digit. When we are done, we scan all the buckets in ascending order and collect the data back into one list. We then take the second digit (the tens digit) and repeat the process. We have to make as many passes through the data as there are digits in the longest key. After the last pass, the list is sorted.

The Radix Sort method works for data with any keys that permit positional representation. For integers, using hexadecimal digits or whole bytes is actually more appropriate than decimal digits. To sort words in lexicographic order we can use radix sort with a bucket for each letter or symbol. We have to pad (logically) all the words with “spaces” to the maximum length and start sorting from the rightmost character position.

*Continued* ↗

Implement Radix Sort for a list of words (`String` objects):

```
public LinkedList<String> sort(LinkedList<String> words)
```

The method takes a list of words and returns a linked list of these words sorted alphabetically in ascending order. Assume that all the words are made up of uppercase letters 'A' through 'Z'. Recall that `Character.getNumericValue(ch)` returns consecutive integers for letters 'A' through 'Z'. Use an array of `LinkedList<String>` to hold the temporary buckets (each bucket is a `LinkedList` of words). Don't forget to "pad" (logically) shorter words with spaces: the first bucket should be reserved for words that are shorter than the letter position at the current pass, so you'll need to use 27 buckets. Use a "for each" loop for each bucket to append the words from that bucket to the new list (or use `LinkedList`'s `addAll` method).

9. ■ Write a `hashCode` method for the `TicTacToeBoard` class. An object of this class represents a tic-tac-toe position. It has a method

```
public char charAt(int row, int col)
```

that returns the character at the `(row, col)` position ( $0 \leq \text{row} < 3$ ;  $0 \leq \text{col} < 3$ ): space, 'o', or 'x'. There are  $3^9$  possible configurations on a 3 by 3 board (although only a fraction of them can occur in a real tic-tac-toe game). Define your `hashCode` method in such a way that it returns different values for all  $3^9$  board configurations.