29 Lists using LinkedList

1 LinkedList

Apart from arrays and array lists, Java provides another class for handling lists, namely `LinkedList<T>`. An instance of `LinkedList<T>` is called a *linked list*. The constructors for `LinkedList<T>` are

LinkedList<T>() LinkedList<T>(LinkedList<T>)

They behave just like the corresponding constructors for `ArrayList`. All the methods of `ArrayList` are also provided by `LinkedList`. Indeed, if in a program you replace all occurrences of `ArrayList` with `LinkedList` the program will still work as before. The difference lies in the underlying implementations. Linked lists are not implemented using arrays, with the consequence that indexing into linked lists is expensive. So operations such as `t.get(i)` and `t.set(i,o)` are best avoided when `t` is a linked list. On the other hand, insertions and deletions to/from linked lists are more efficient than for array lists. Many applications require us to maintain lists which grow and shrink by adding or deleting items at either end of the list, and which do not require indexing into the list. For these applications, linked lists are to be preferred over array lists.

`LinkedList<T>` (but not `ArrayList<T>`) includes some methods for operating on either end of a list. These bring some added convenience but no added power.

void addFirst(*T*)
T getFirst()
T getLast()
T removeFirst()
T removeLast()

(As usual, `T` stands for the type of the elements in the list.) **`t.addFirst(o)`** inserts `o` at the beginning of list `t`. (Recall that `t.add(o)` appends `o` to the end of `t`.) **`t.getFirst()`**

returns a reference to the first element in list `t`. `t` must not be empty. **`t.getLast()`** returns a reference to the last element in list `t`. `t` must not be empty. **`t.removeFirst()`** removes the first element from list `t` and returns a reference to it. `t` must not be empty. **`t.removeLast()`** removes the last element from list `t` and returns a reference to it. `t` must not be empty.

The following code illustrates the behaviour of the above methods.

```
LinkedList<String> s = new LinkedList<String>();  // s is empty
s.add("dog"); s.add("cat");                       // s: dog, cat
s.addFirst("pig");                                // s: pig, dog, cat
s.addFirst("cow");                                // s: cow, pig, dog, cat
System.out.print(s.getLast());                    // cat appears
s.removeFirst();                                  // s: pig, dog, cat
String word1 = s.removeLast();                    // s: pig, dog
System.out.print(word1);                          // cat appears
```

We can access all the elements in an array list in turn by indexing over the array, from 0 up to the list size less 1. However, indexing into linked lists is expensive, and it is preferable to use a for-each loop.

Example 1: reverse a list of integers

The following program reads a sequence of integers from the keyboard, and displays them in reverse order.

```
import java.util.*;
class ReverseInts {
   public static void main(String[] args) {
      LinkedList<Integer> nums = new LinkedList<Integer>(); // for numbers in input
      while (!Console.endOfFile()) {
         int k = Console.readInt();
         nums.addFirst(k); //using autoboxing
      }
      for (Integer k: nums)  {
         System.out.print(k + "  ");
      }
   }
}
```
■

A *stack* is a list of items to which elements are added and from which elements are removed at one end only. A stack is also called a *last-in-first-out* or *LIFO* list. Some languages provide specially for them. In the case of Java, `LinkedList` provide all the functionality needed. A *queue* is a list to which items are added at one end and from which items are removed at the other end. Items are never inserted are removed at any other position. A queue is also called a *first-in-first-out* or *FIFO* list. Queues occur commonly in some application areas. For example, a program controlling a printer shared by many users must keep a list of jobs waiting to be printed. It is only fair that when the printer becomes available, the job that has been waiting the longest should be the first to print. The list of waiting jobs in the controlling program will therefore be a queue. Some languages provide specially for queues, but in Java the functionality required is provided by `LinkedList`.

There's a big banana skin lurking whenever we use collections such as array lists or linked lists. Inserting an object in a collection is effected by aliasing. In other words, a *reference* to the original object is inserted in the collection, rather than copying the object and inserting a reference to the copy. This means that if you insert an item in a list (or any other kind of collection), and then subsequently in another part of the program alter its state, the change is also effected in the object as stored in the list. This may well be what you want. However, if you intend to change an object after inserting it in a list, and you want the original object to remain unchanged in the list, you should make a copy of the object and insert the copy. Note that constructors with collection arguments always introduce aliasing. For example, `ArrayList<Integer> w = new ArrayList<Integer>(t)` results in the creation of a new list `w`, but the elements in `w` and `t` are shared via aliasing. However, `w` and `t` are themselves distinct; for example, appending a new element to `t` has no effect on `w`.

We mention some additional methods provided by both `ArrayList<T>` and `LinkedList`<T>. They are not used so much (and they are expensive) so it suffices to pass lightly over them. We will not be using them.

boolean addAll(*Collection<T>*)
boolean retainAll(*Collection<?>*)
boolean removeAll(*Collection<?>*)
boolean containsAll(*Collection<?>*)

Again we write `Collection<T>` to stand for `ArrayList<T>` or `LinkedList<T>` or any one of several other kinds of collection in the Java library. The question mark in `Collection<?>` stands for an arbitrary type, although in practice it will nearly always be the same as `T`. **`t.addAll(c)`** appends all the elements in collection `c` to the end of list `t`. When `c` is of type `LinkedList` or `ArrayList`, the elements are appended in the same order as they occur in `c`. When `c` is of type `HashSet`, the order in which the elements are appended is not specified. When `c` is of type `TreeSet`, the elements are appended in ascending order. A boolean is returned indicating whether `t` changed as a result of the call. **`t.retainAll(c)`** removes from list `t` every element not in collection `c`. A boolean is returned indicating whether `t` changed as a result of the call. The method is more expensive for `t` of type `ArrayList`. **`t.removeAll(c)`** removes from list `t` all the elements in collection `c`. A boolean is returned indicating whether `t` changed as a result of the call. The method is more expensive for `t` of type `ArrayList`. **`t.containsAll(c)`** returns a boolean indicating whether list `t` contains all the elements in collection `c`.

2 List utilities

The collections classes includes a class **`Collections`** which provides some useful static methods on lists, including the following:

```
static void sort(List<T>)
static void shuffle(List<T>)
```

We write `List<T>` above to indicate that the actual parameter may be an instance of `LinkedList<T>` or `ArrayList<T>`. `T` is a type parameter. **`Collections.sort(t)`** sorts list `t` into ascending order; elements are compared using `compareTo() which must be provided by class E`. **`Collections.shuffle(t)`** rearranges the elements in list `t` in some random way.

The following code illustrates these methods.

```
ArrayList<String> s = new ArrayList<String> ();  // s is empty
s.add("dog"); s.add("hen");                      // s: dog, hen
s.add("cat"); s.add("hen");                      // s: dog, hen, cat, hen
System.out.print(s);                             // [dog, hen, cat, hen] appears
Collections.sort(s);                             // s: cat, dog, hen, hen
Collections.shuffle(s);                          // s (e.g.): hen, dog, hen, cat
```

Example 1: sorted word list

We write a program `WordList` to print the words in a text file in ascending order. No word appears more than once in the output, and case is not significant in comparing words. For example, the words in *The Tempes*t can be listed by invoking

```
java WordList TheTempest.txt
```

where `TheTempest.txt` contains the text of *The Tempest* (you can find Shakespearean plays and many other literary works on the web). :

```
import java.util.*;
import java.io.*;
class WordList {
    public static void main(String[] args) {
        // Phase 1: generate a sorted list of words in the input file (lower case)
        ArrayList<String> words = new ArrayList<String>(); // for words in input file
        Scanner text = null;
        try {
            text = new Scanner(new File(args[0]));
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
```

```
            while (text.hasNext()) {
                String w = text.next();  // next word
                // convert w to lower case, & strip any trailing punctuation mark
                w = w.toLowerCase();
                if (!Character.isLetterOrDigit(w.charAt(w.length()-1)))
                    w = w.substring(0, w.length()-1);
                words.add(w);
            }
            Collections.sort(words);
            // Phase 2: print words omitting duplicates
            String lastWord = ""; // last word processed
            for (String w: words) {
                if (!w.equals(lastWord)) {
                    System.out.println(w);
                    lastWord= w;
                }
            }
        }
    }
```

Recall that `readToken()` reads *tokens* which are almost but not quite the same as words. Tokens may end in a punctuation mark such as a comma or period, and these have to be removed. The words are stored in lower case.

3 equals and compareTo

Many of the methods in the list classes compare elements using `equals()`. Although `equals()` comes for free with all classes, it compares objects based on references rather than contents. If an `equals()` based on object contents is not provided, then some methods in the collection classes will not behave as expected.

`Collections.sort(t)` assume that the constituent elements come equipped with a comparison method `compareTo()` that behaves like `compareTo()` for strings. An invocation of `p.compareTo(q)` returns (i) a negative integer if `p` precedes `q` in the ordering we have in mind; (ii) a positive integer if `q` precedes `p`; and (iii) 0 if `p` and `q` are equal. Additionally, the class definition must announce in its header that objects of the class are comparable, as in the following example of a class `Person`.

```
    class Person implements Comparable<Person> {
        private String name;
        private int age;

        Person (String s, int years) {
            name = s; age = years;
        }
        public int compareTo(Person other) {
            if (name.equals(other.name)) {
                if (age==other.age) return 0;
                else if (age<other.age) return -1;
                else return 1;
            }
            else return name.compareTo(other.name);
        }

        // ... more ...
    }
```

Method compareTo() above compares Person objects alphabetically by name, with any tie being resolved by comparing ages. A class Person with compareTo() must include the phrase implements Comparable<Person> after the class name; if you omit this, the system will fail to find compareTo() (technically speaking, Comparable is an interface in the Java library). The header of compareTo() must be precisely public int compareTo(Person other) (the name of the parameter can be any identifier, however). It is common practice to return –1 for *less than* and 1 for *greater than*, but this is not required. For example, the following alternative version of compareTo() for Person acceptable:

```
    public int compareTo(Person other) {
        if (name.equals(other.name))
            return age-other.age;
        else return name.compareTo(other.name);
    }
```

You must define compareTo() so that it really is a comparison operation – just returning 257, say, for every invocation will not give you meaningful results. If you proceed sensibly you will not meet any difficulties. If you are in doubt, check that your definition satisfies the following requirements (the *sign* of a number is –1, 0, or 1 according to whether it is negative, zero, or positive, respectively):

(i) The sign of p.compareTo(q) equals -1 * the sign of q.compareTo(p).
(ii) If p.compareTo(q)>0 and q.compareTo(r)>0 then p.compareTo(r)>0.
(iii) If p.compareTo(q) equals 0 then p.compareTo(r) and q.compareTo(r) have the same sign.

If objects of the class are also subject to equality testing, equals() and compareTo() must be consistent in the sense:

(iv) p.equals(q) yields true if and only if p.compareTo(q) yields 0.

The primitive wrapper classes all provide a meaningful compareTo().