Unit 12

Dynamic arrays and linked lists

Summary

- Limitations of arrays
- Dynamic memory management
- Definitions of lists
- Typical operations on lists: iterative implementation
- Typical operations on lists: recursive implementation

12.1 Limitations of arrays

We have seen that we can use **arrays** whenever we have to store and manipulate collections of elements. However, the use of arrays for this purpose presents several limitations related to the way arrays are handled in Java:

- the dimension of an array is determined the moment the array is created, and cannot be changed later on;
- the array occupies an amount of memory that is proportional to its size, independently of the number of elements that are actually of interest;
- if we want to keep the elements of the collection ordered, and insert a new value in its correct position, or remove it, then, for each such operation we may need to move many elements (on the average, half of the elements of the array); this is very inefficient.

12.2 Dealing with the dimension of an array

Some of the previous limitations can be overcome by suitably manipulating arrays, in such a way that the memory necessary to store the data of the application is allocated and deallocated dynamically during the program execution, depending on the needs.

The following example shows a class for dealing with a list of persons (objects of the class `Person`) that manages the size of the array dynamically.

```
public class ListOfPersonsArray {

  private Person[] a;
  private int n;

  public ListOfPersonsArray() {
    a = new Person[10];
    n = 0;
  }

  // Add a person to the List of Persons in the last position
  public void add (Person p) {
    if (n == a.length) {
      // The array is full!!!
      // We have to create a new, bigger array and copy all elements
      Person[] b = new Person[a.length*2];
      for (int i=0; i<a.length; i++)
        b[i] = a[i];
      a = b;
    }
    // Now we are sure that n < a.length
    a[n] = p;
```

```
      n++;
    }

    // Remove the person in position k
    public void remove (int k) {
      if ((k >= 0) && (k < n)) {
        // We have to move all elements that follow k
        for (int i = k; i < n; i++)
          a[i] = a[i+1];
        n--;
      }
      // We reduce the dimension of the array if it is sufficiently empty
      if ((a.length > 10) && (n < a.length/4)) {
        Person[] b = new Person[a.length/2];
        for (int i = 0; i < n; i++)
          b[i] = a[i];
        a = b;
      }
    }
  }
```

Note that the dynamic management of the array is not done by dynamically changing its dimension (which is not possible), but by creating a new bigger or smaller array according to the application needs, and copying all values to the new array each time this becomes necessary.

The dimension of a new dynamically created array is doubled when new elements are necessary, and halved when the array is sufficiently empty. The choice of considering the array sufficiently empty when `n < a.length/4` allows us to minimize the number of times it is necessary to create a new array, since at most half of the elements of the newly created array are used the moment the array is created.

12.3 Dynamic memory management

We have just given an example of *dynamic memory management*, in which memory resources are allocated whenever they are needed, and freed whenever they are not necessary anymore. This is a very important aspect, since it defines the modalities by which programs interact with machine memory in order to store the data they require.

Actually, we have already seen at the beginning of the course that in Java we have the ability to allocate memory by creating new objects. This is done by means of the `new` operator that invokes a constructor of a class. In this case, we need to use a distinct variable for each element that we want to store and manipulate in the program.

When we want to define and manipulate collections of objects, we need an additional mechanism to allocate memory dynamically, that should take into account that, in general, the number of elements in the collection we have to deal with is not known a priori. **Linked lists**, which we are introducing now, are defined precisely in such a way as to allow for dynamically allocating and deallocating memory, depending on the requirements of the application.

As opposed to the dynamic memory management provided by lists, we could say that with arrays we have a *static memory management*, since their size cannot be changed, once they are created.

12.4 Linked lists

A very flexible mechanism for dynamic memory management is provided by *linked structures*, which are realized in such a way as to allow an easy insertion and deletion of elements in specific positions, and, more generally, an easy modification of their structure.

In this course, we deal only with *linear linked structures*, called also linked lists, or simply **lists**. Such structures allow us to store collections of elements organized in the form of a linear sequence of nodes. The fact that a sequence is *linear* means that each element has at most one successor element (possibly none, if the element is the last one in the sequence).

Note that, in general, the number of nodes of a list is not known in advance, and we do not have variables containing references to all nodes. Instead we maintain only a reference to the first node of the list, and access the whole list through its first node, by following the links.

Hence:

- a nonempty list is represented by a *reference to its first node*;
- the *empty list* is represented by `null`.

12.5 The class `ListNode`

The basic class for a linked lists is a class whose objects represent the information associated to a single element (or *node*) of the structure.

```
public class ListNode {
  public ElementType info;
  public ListNode next;
}
```

Such a class defines two public instance variables:

- `info`, containing the information of interest, which could be of any type;
- `next`, containing a reference to the next node of the list.

In the case where we need a list only as an auxiliary data structure for a certain class `C`, we can also define the class `ListNode` in the same file `C.java` in which we define the class `C`, as follows:

```
class ListNode {
  ElementType info;
  ListNode next;
}

public class C {
  ...
}
```

Since the class `ListNode` and its instance variables `info` and `next` are not declared `public` (or `private`), the visibility of `ListNode` and of its instance variables is at the level of *package* (i.e., it is visible only to classes defined in the same directory in which the file `C.java` is placed). Note that, if instead we define `ListNode` as a public class, we must place it in a separate file `ListNode.java`.

Note that, in both cases (when the class list is `public`, and when it is visible at the level of package), the instance variables `info` and `next` are *not* declared as `private`, because we want to directly access them outside the class `ListNode` in order to manipulate lists (the class `ListNode` itself has no methods of its own, besides the default constructor without parameters).

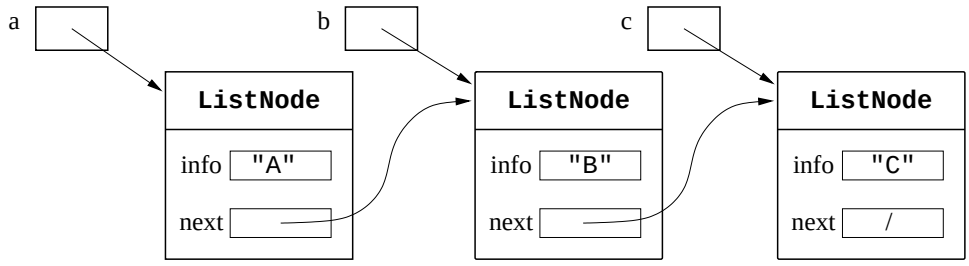
12.6 Creation and linking of nodes

The following code fragment creates a linear structure with 3 nodes containing strings.

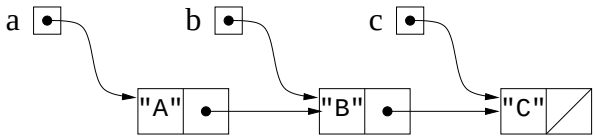
```
class ListNode {
  String info;
  ListNode next;
}

public class TestList {
  public static ListNode create3NodesABC() {
    ListNode a = new ListNode();
    ListNode b = new ListNode();
    ListNode c = new ListNode();
    a.info = "A";
    a.next = b;
    b.info = "B";
    b.next = c;
    c.info = "C";
    c.next = null;
    return a;
  }
}
```

The following figure shows a representation in memory of the list created by the method above.

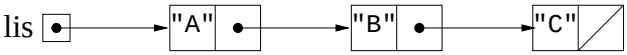


Note: In the following, we will use a more compact graphical representation of `ListNode` objects and lists. Specifically, the above list will be represented as follows.



12.7 Operations on linked lists

Suppose that we have already created a linked list in memory, and that a variable `lis` of type `ListNode` contains a reference to the first element of the list.



We can perform various operations on such a list. The most common operations are:

- *checking* whether the list is empty;
- *accessing* a node to modify it or to obtain the information in it;
- *traversing* the list to access all elements (e.g., to print them, or to find some specific element);
- determining the *size* (i.e., the number of elements) of the list;
- *inserting* or *removing* a specific element (e.g., the first one, the last one, or one with a certain value);
- creating a list by *reading* the elements from an input stream;
- *converting* a list to and from an array, string, etc.

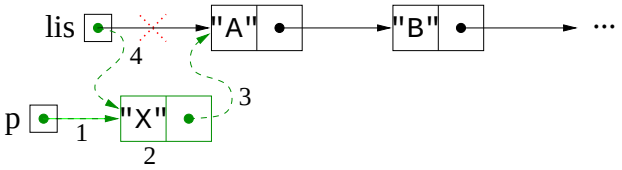
Note that some of the operations above do not modify the list at all, some modify only the information field of a node, and some modify the structure of the list, by changing how the nodes are connected to each other. We will realize each operation through a static method:

- The method takes as one of its parameters a reference to the first node of the list.
- If the method modifies the list, it also returns a reference to the first node of the modified list as its return value.

12.8 Inserting a new element as the first one of a list

To insert a new element as the first one of a list:

1. we allocate a new node for the element (note that we are given the element (e.g., a string), but we still must create the node for it),
2. we assign the element to the `info` instance field,
3. we concatenate the new node with the original list,
4. we make the newly created node the first one of the list.



Note that we do not need to actually access the elements of the list to perform this operation.

Implementation:

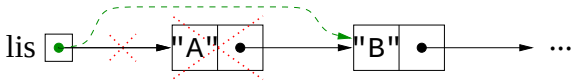
```
public static ListNode insertFirst(ListNode lis, String s) {
  ListNode p = new ListNode();     // 1
  p.info = s;                      // 2
  p.next = lis;                    // 3
  lis = p;                         // 4
  return lis;
}
```

12.9 Deleting the first element of a list

In general, *deleting* an element of a list means to modify the list in such a way that the element is no longer connected to its predecessor and successor, while bridging the deleted element to maintain the connection of the other elements.

To delete the first element of a list we proceed as follows:

- If the list is empty, we don't do anything.
- Otherwise, we return the list starting at the element that follows the first one.



Implementation:

```
public static ListNode deleteFirst(ListNode lis) {
  if (lis != null)
    lis = lis.next;
  return lis;
}
```

Note that the method works correctly also in the case where the element to delete is the only one of the list (i.e., when the value of `lis.next` is `null`). In this case, the method returns the empty list (i.e., `null`).

Note: The possibility of deleting a node by simply modifying the reference to it in such a way that this reference points to the following node, without needing to take care of releasing the occupied memory, is a peculiarity of Java (and in general, of those languages that make use of *garbage collection* for the dynamic management of objects). Indeed, the garbage collector will take care of checking whether the "deleted" node is not referenced anymore, and hence the memory occupied by it can be made available on the heap for other objects. Other programming languages, such as Pascal, C, or C++, require that the programmer explicitly frees the memory occupied by those objects that are not needed anymore.

12.10 Accessing successive nodes of a list

To perform an operation on all elements of a list, we have to reach each element starting from the first one, by following the `next` references. One way of doing this is through iteration. The loop scheme to access all elements of a list whose first element is referenced by a variable `lis` is the following.

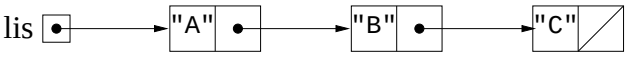
```
ListNode lis = ...
ListNode p = lis;
while (p != null) {
  process the node referenced by p
  p = p.next;
}
```

Note that we start the "scan" of the list by using `p` as a reference to the *current node*. We initialize `p` to the value of `lis`, and then, at each iteration of the loop, we advance `p` to the next node through the statement `p = p.next`. Note that this statement does not modify in any way the list itself.

12.11 Printing the elements of a list on an output stream

We want to print on a `PrintStream` the elements of a list in the order in which they appear in the list (separating them by a `' '`).

Example: For the list



the output should be:

```
A B C
```

We realize this operation through a static method that takes as parameters the reference to the first element of the list and a `PrintStream`.

```
public static void print(ListNode lis, PrintStream ps) {
  ListNode p = lis;
  while (p != null) {
    ps.print(p.info + " ");
    p = p.next;
  }
  ps.println();
}
```

Note that, in this case, we could have directly used `lis` to scan the list, since inside the method we do not need to access anymore the nodes we have already processed (i.e., printed out). We obtain the following implementation.

```
public static void print(ListNode lis, PrintStream ps) {
  while (lis != null) {
    ps.print(lis.info + " ");
    lis = lis.next;
  }
  ps.println();
}
```

12.12 Lists as inductive structures

A different way to perform an operation on all elements of a list is by exploiting the fact that lists are inductive structures:

- the empty list (i.e., the list with no nodes) is a list;
- if *lis* is a list and *elem* is an element, then we obtain a list by taking *elem* as first node and *lis* as rest;
- nothing else is a list.

Exploiting the inductive characterization of lists, we can realize all operations on list also through recursion. The typical structure of a recursive method that operates on a list is as follows:

```
if (the list is empty) {
  perform the operation for the empty list
} else {
  perform the operation on the first element of the list
  call the method recursively on the rest of the list
}
```

12.13 Printing the elements of a list – recursive version

We can characterize recursively the operation of printing the elements of a list as follows:

1. if the list is empty, don't do anything (*base case*);
2. otherwise, print the first element, and then print recursively the rest of the list (*recursive case*).

Recursive implementation:

```
public static void print(ListNode lis, PrintStream ps) {
  if (lis == null)
    ps.println();                // base case
  else {
    ps.print(lis.info + " ");    // process the first element
    print(lis.next, ps);         // recursive call
  }
}
```

12.14 Searching an element in a list

To check whether an element in present in a list, we can "scan" the list until:

- we find a node with the element we are looking for, or
- we reach the end of the list.

Iterative implementation:

```
public static boolean search(ListNode lis, String s) {
  while (lis != null) {
    if (lis.info.equals(s)) return true;
    lis = lis.next;
  }
  return false;
}
```

We can also characterize recursively the operation of checking whether an element *elem* is present in a list *lis* as follows:

1. if *lis* is empty, then return `false` (*base case*);
2. otherwise, if *elem* is the first element of *lis*, then return `true` (*base case*);
3. otherwise, return the result of searching *elem* in the rest of *lis* (*recursive case*).

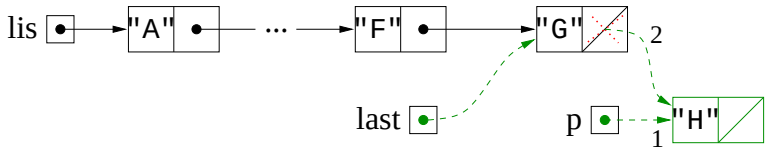
Given such a characterization, the recursive implementation is immediate. We leave it as an exercise.

12.15 Inserting a new element as the last one of a list

To insert `elem` as the last element of `lis`:

```
create a new node whose info field is elem;
if (lis is empty)
  return the list constituted only by the new node;
else {
  scan the list stopping when the current reference points to the last node
  concatenate the new node with the last one
}
```

The situation in memory after the scan of the list is finished and `last` references the last element of the list is shown below.



Iterative implementation:

```
public static ListNode insertLast(ListNode lis, String s) {
  ListNode p = new ListNode();      // note: p.next == null
  p.info = s;

  if (lis == null)
    return p;                       // the list contains only the new node
  else {
    ListNode last = lis;
    while (last.next != null)       // find last element
      last = last.next;
    last.next = p;
    return lis;
  }
}
```

12.16 Inserting a new element as the last one of a list – recursive version

Recursive characterization of the insertion of *elem* as the last element of *lis*:

1. if *lis* is the empty list, then return the list constituted only by one node containing *elem* (*base case*);
2. otherwise, return the list whose first element coincides with the one of *lis*, and whose rest is obtained by inserting *elem* in the rest of *lis* (*recursive case*).

Recursive implementation:

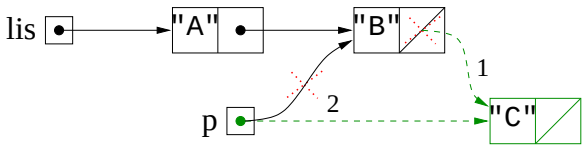
```
public static ListNode insertLast(ListNode lis, String s) {
  if (lis == null) {
    ListNode res = new ListNode();    // note: res.next == null
    res.info = s;
    return res;
  } else {
    lis.next = insertLast(lis.next, s);
    return lis;
  }
}
```

12.17 Creation of a list whose elements are read from a file

We want to create a list by reading its elements from a file, and we want the order of the elements to be as in the file. This means that each newly read element must be inserted at the end of the list.

To avoid that for each new element read from the file we have to scan the entire list already constructed, we maintain a reference to the last element read.

Example: If the first three lines of the file contain the strings `"A"`, `"B"`, and `"C"` in that order, the third element of the list is concatenated to the preceding ones as shown here:



For the iterative implementation, to avoid having to deal separately with the first element of the list, we use the **generator node** technique:

- the generator node is a node whose `info` field does not contain any significant information;
- it is created and placed at the beginning of the list before starting the loop; in such a way, all elements of the list, including the first one, have a predecessor node and can be dealt with in the same way;
- before terminating the method, we have to remove the generator node.

Iterative implementation:

```
public static ListNode read(BufferedReader br) throws IOException {
  // note: the value EOF (^D) in input terminates the insertion
  ListNode lis = new ListNode();    // create generator node
  ListNode p = lis;

  String s = br.readLine();
  while (s != null) {
    p.next = new ListNode();        // note: p.next.next == null
    p = p.next;
    p.info = s;
    s = br.readLine();
  }

  lis = lis.next;                   // delete generator node
  return lis;
}
```

The recursive characterization and implementation are left as an exercise.

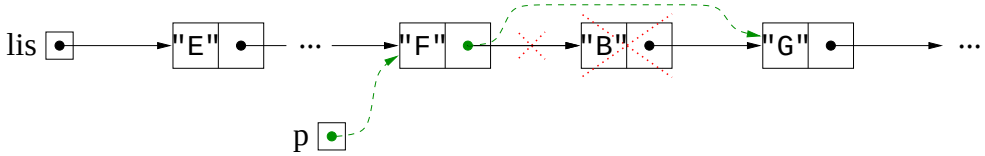
12.18 Deleting the first occurrence of an element from a list

To delete the first occurrence of an element *elem* from a list through iteration:

- we scan the list searching for *elem*;
- if *elem* does not appear in the list, then we do nothing;
- otherwise, depending on where *elem* appears in the list, we distinguish three cases:
  1. *elem* is the first element of the list: we return the reference to the second element of the list;
  2. *elem* is neither the first element nor the last element of the list: we update the `next` field of the element preceding *elem* in such a way that it points to the element following *elem*;
  3. *elem* is the last element of the list: as case 2, with the only difference that the `next` field of the preceding element is set to `null`.

Observations:

- To be able to update the `next` field of the element preceding *elem*, we have to *stop scanning the list when we have a reference to the element preceding elem* (and not when we have reached *elem* itself).
  Example: Suppose we have to delete the element `"B"` from the list shown in the figure:



- To stop scanning the list after we have found and deleted the element, we use a boolean sentinel.
- To avoid having to deal separately with the first element, we can again use the generator node technique.

Iterative implementation:

```
public static ListNode delete(ListNode lis, String s) {
  ListNode p = new ListNode();     // create the generator node
  p.next = lis;
  lis = p;

  boolean found = false;
  while ((p.next != null) && !found) {
    if (p.next.info.equals(s)) {
      p.next = p.next.next;        // delete the element
      found = true;                // forces exit of the loop
    } else
      p = p.next;
  }

  return lis.next;                 // delete generator node
}
```

The recursive characterization and implementation are left as an exercise.

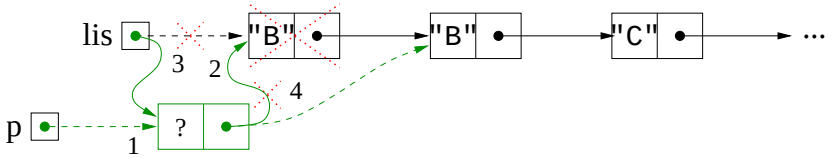
12.19 Deleting all occurrences of an element from a list

To delete all occurrences of an elements from a list, we can proceed as for the deletion of the first occurrence. The differences are that:

- after having found and deleted the element, we have to continue scanning the list;
- we stop only when we have reached the end of the list.

Observations:

- In this case, it is definitely better to use the generator node technique, since after having deleted the first element of the list, we may again be in the situation where we have to delete the first element.
  Example: Suppose we have to delete the element `"B"` from the list shown in the figure:



- We do not need a sentinel to stop the loop.

Iterative implementation:

```
public static ListNode deleteAll(ListNode lis, String s) {
  ListNode p = new ListNode();    // create the generator node
  p.next = lis;
  lis = p;

  while (p.next != null) {
    if (p.next.info.equals(s))
      p.next = p.next.next;       // delete the element
    else
      p = p.next;
  }

  return lis.next;                // delete generator node
}
```

12.20 Deleting all occurrences of an element from a list – recursive version

We can characterize recursively the operation of deleting all occurrences of an element *elem* from a list *lis* as follows:

1. if *lis* is the empty list, then return the empty list (*base case*);

2. otherwise, if the first element of *lis* is equal to *elem*, then return the list obtained by deleting all occurrences of *elem* from the rest of *lis* (*recursive case*);

3. otherwise, return the list whose first element coincides with the first element of *lis*, and whose rest is obtained by deleting all occurrences of *elem* from the rest of *lis* (*recursive case*).

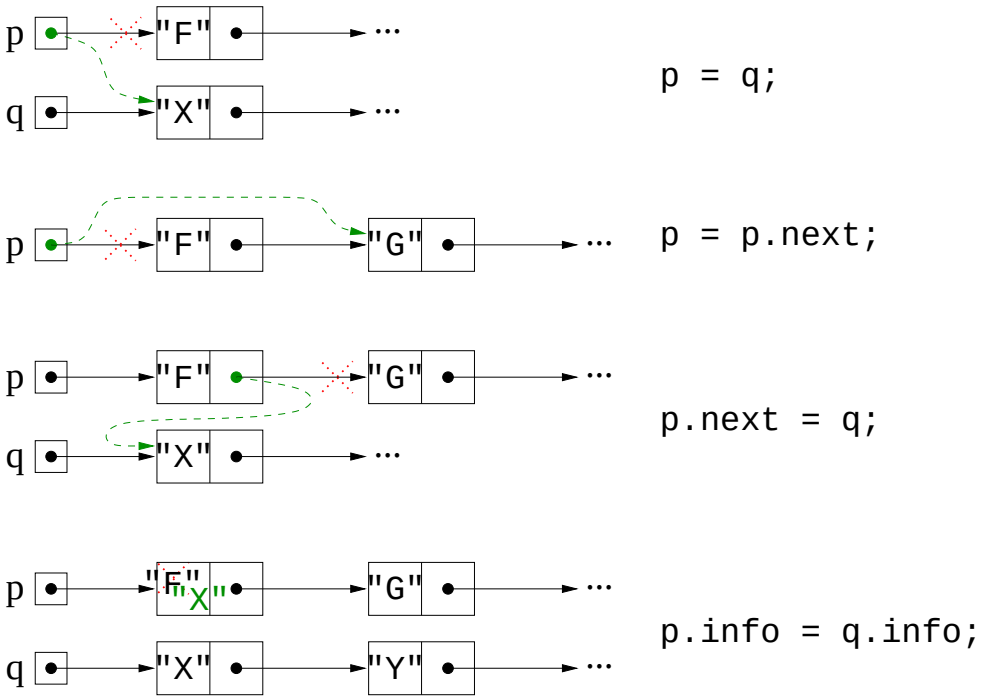
Recursive implementation:

```
public static ListNode deleteAll(ListNode lis, String s) {
  if (lis == null)
    return null;
  else if (lis.info.equals(s))
    return deleteAll(lis.next, s);
  else {
    lis.next = deleteAll(lis.next, s);
    return lis;
  }
}
```

Note: It does not make sense to apply the generator node technique when we implement an operation on lists using recursion. Such a technique makes only sense for an iterative implementation, and typically only when the list is modified by the operation.

12.21 Summary of the graphical representation of list manipulation statements

The following figure summarizes graphically the effect of the various statements manipulating references to list elements:



Exercises

Exercise 12.1. Provide an iterative implementation of the following operations on lists of strings:

- `public static int length(ListNode lis)`
  that returns the length of `lis`;
- `public static ListNode insertAfter(ListNode lis, String s, String given)`
  that returns the list obtained by modifying `lis` by inserting a new element `s` after a `given` element, if such an element is present, and returns `lis` unmodified otherwise;
- `public static ListNode modify(ListNode lis, String old, String ne)`
  that returns the list obtained by modifying `lis` by changing the first occurrence of `old` to `ne`;
- `public static ListNode modifyAll(ListNode lis, String old, String ne)`
  that returns the list obtained by modifying `lis` by changing all occurrences of `old` to `ne`;
- `public static ListNode insertBefore(ListNode lis, String s, String given)`
  that returns the list obtained by modifying `lis` by inserting a new element `s` before a `given` element, if such an element is present, and returns `lis` unmodified otherwise;
- `public static ListNode copy(ListNode lis)`
  that returns a copy of the list `lis`, i.e., a list containing the same sequence of elements as `lis`, but using new nodes;
- `public static ListNode invert(ListNode lis)`
  that modifies `lis` by inverting the links among its nodes, and returns a reference to the inverted list;
- `public static ListNode deleteDoubles(ListNode lis)`
  that returns the list obtained by modifying `lis` by deleting all occurrences of an element apart from the first one.
- `public static boolean searchSequence(ListNode lis1, ListNode lis2)`
  that checks whether the lis `lis1` contains a subsequence of consecutive elements that coincides with `lis2`. For example, if the sequence of elements of `lis1` is `(A B B C D E)`, and the sequence of elements of `lis2` is `(B B C)`, the result should be `true`. Instead, if `lis1` is again `(A B B C D E)` and `lis2` is `(A B C)`, the result should be `false`.

Exercise 12.2. Provide a recursive implementation of the operations of Exercise 12.1.

Exercise 12.3. Whenever on the domain of the elements of a list an order is defined, we can consider ordered lists, in which the sequence of elements in a list is ordered. Provide a recursive and an iterative implementation of the following operations on ordered lists of integers:

- check whether a list is actually ordered;
- search of a given element, exploiting the fact that the list is ordered to interrupt the search when the element has not been found and we know that it cannot appear in the rest of the list;
- insertion of a new element, maintaining the list ordered;
- deletion of a given element;
- merge of two ordered lists into a new ordered list containing the elements of both lists (but whose nodes are newly created);
- merge of two ordered lists into an ordered list that reuses the nodes of the two lists (hence, the two lists are destroyed by the method call);
- intersection of two ordered lists, to produce a new ordered list containing exactly those elements that are present in both lists.

Exercise 12.4. Define a class `ListOfPersons`, each of whose objects maintains the information about a sequence of objects of a class `Person` (for each person, the information of interest are the name, surname, age, and city of residence). The class `ListOfPersons` should provide, besides the methods for insertion, deletion and modification of a person, a method to print all persons in the list, a method to count the number of persons in the list that live in a given city passed as parameter, and a method that computes the average age of all persons in the list.

Exercise 12.5. Solve Esercise 7.12 by removing the assumption that at most 10 persons live in an apartment, and handling such an information by means of a linked list, possibly using the class defined in the preceding exercise. Add to the class `Apartment` a method `countPersons()` that returns the number of persons living in the apartment, and `checkFamily()` that checks whether all persons living in the apartment have the same surname.

Exercise 12.6. Define a class `ListCDs` that maintains information about a sequence of objects of a class `CD` (for each CD, the information of interest are author, title, publication year, and price). The class `ListCDs` should provide, besides the methods for inserting, deleting and printing the elements of a list, a method that calculates the total price of the CDs contained in the list, and a method that returns the list of CDs of a given author passed as parameter.

Exercise 12.7. Define a class `MyString`, that handles sequences of characters (by means of a list of `char`) and implements some methods similar to those of the class `String`. Specifically, implement a constructor that creates an object of the class starting from a string, and the methods `concat()`, `equals()`, `substring()`, `length()`, `charAt()`, `indexOf()`, and `toUpperCase()`. Realize also a method, external to the class `MyString`, that behaves in the same way as the method `Integer.parseInt()`. More precisely, given an object of the class `MyString`, the method should return the integer value corresponding to the numeric digits represented in the `MyString` object, or should throw an exception if the object contains characters that are different from the digits `'0'`, ..., `'9'`. Implement the latter method both using iteration and using recursion.

Exercise 12.8. Define a class `BigNumber`, whose objects represent positive integers with an unbounded number of digits, realized by means of a list of integers. The class should implement a constructor that creates a `BigNumber` object starting from a string of numeric digits, the methods to calculate the sum and difference of two `BigNumber`s, the predicates to compare whether two `BigNumber`s are equal and whether one is smaller than another, and the method `toString` that constructs the string corresponding to the number. For the difference, return the object representing the number 0 in the case where the result of the subtraction would be negative. We suggest to represent a `BigNumber` using a list of integers in which the least significant digit is the first element of the list, and the most significant digit the last element of the list.