Assignment 4

Julien Corman, Tiziano Dalmonte, Werner Nutt

Linked Lists

For this coursework, you are only asked to write code. It consists in the implementation of standard operations on simple linked lists and one of their variants, head tail lists, plus the implementation of some sorting algorithms that you know already for arrays. You have two weeks time for this coursework (not counting the Christmas break).
The modalities for code submissions are:

- Method stubs are available on Codeboard. Your implementation must respect the methods' signatures (but of course, you are free to create and call additional methods if needed).
- The code must be submitted via Codeboard, and a copy of it (zipped .java files) via OLE.
- Use only data structures and methods known from the course.
- Follow the guidelines for good coding style on the OLE pages of the course. This includes adding comments to your code where needed to make it understandable to readers.

1. Operations on Simple Linked Lists

Add methods to the data type list `List` that was introduced in the lectures. You find a Java implementation of such lists and with some of their operations in the sample code of Lab 8. Remember that:

- an object of type `Node` has two fields, an integer `val` and (a pointer to) a Node `next`;
- an object of type `List` has one field, (a pointer to) a Node `head`;

1. `boolean isEmpty();`
2. `int length();`
   returns the number of nodes in the list, which is 0 for the empty list;

3. `int floor(int i);`
   returns the largest value in the list that is less or equal `i` if such a value exists and `Integer.MIN_VALUE` otherwise;

4. `void addAll(List l)`
   appends the list `l` to the last element of the current list, if the current list is nonempty, or lets the `head` of the current list point to the first element of `l` if the current list is empty;

5. `int maxAscent();`
   an ascent in a list is a contiguous sequence of list nodes so that each node in the sequence has a value less or equal the value of the next node in the sequence; the method returns the maximal length of an ascent in the list; it returns 0 for the empty list and a value ≥ 1 for a nonempty list;

6. `boolean sorted()`
   checks whether the current list is sorted, that is, whether the value of each node is less or equal than the value of the next node (the empty list is sorted);

7. `int extractMax();`
   returns the maximal value in the list and removes the node with that value (or one such node if there are more than one), throws a Java RuntimeException if called for the empty list;

8. `List selectionSort();`
   returns a sorted copy of the original list, computed with a list version of the Selection Sort algorithm known for arrays (note that the empty list is a sorted version of the empty list); the method must never return a null pointer as the result;

9. `void addSorted(int i)`
   creates a new node with the integer `i` and adds it in front of the first node with a `val` greater or equal to `val`; therefore, adding a number with this method to a sorted list maintains sortedness;

10. `List insertionSort();`
    returns a sorted copy of the original list, computed with a list version of the Insertion Sort algorithm known for arrays.
    Hint: The idea of Insertion Sort is to repeatedly insert nodes into a sorted list such that the order is preserved. Check out which of the methods defined for your linked list type can be used for implementing `insertionSort`.

(Weight: 50% of this CW)

2. Operations on Head-Tail Lists

Implement a data type `HTList` that realizes head-tail lists consisting of nodes with integer values, as also discussed in the lecture. Nodes and list objects are defined by the following class declarations:

- The nodes of a head-tail list are instances of the class

```
class HTNode{
   int val;
   HTNode next;}
```

- The list objects are of type `HTList` and are similar to objects of type `List`, but have two fields with (i.e., two pointers to) HTNodes `head` and `tail`:

```
class HTList{
   HTNode head, tail;}
```

Modify the following methods for the type `List` from the lecture, the labs and the previous exercise so that they work for head-tail lists:

1. `void addAsHead(int i)`
   creates a new node with the integer and adds it to the beginning of the list;
2. `void addAsTail(int i)`
   creates a new node with the integer and adds it to the end of the list;
3. `void addAll(HTList l)`
   appends the nodes of list `l` to the last element of the current list, if the current list is nonempty, or lets the `head` of the current list point to the first element of `l` if the current list is empty.

(Weight: 10% of this CW)

3. Efficient Sorting with Lists

In this exercise, we want to realize list versions of efficient algorithms that we know already for arrays. Briefly say in comments in your code what each part of your implementation does.

1. Develop a version of Merge Sort for linked lists. Realize it as a method

   ```
   List mergeSort()
   ```

   of the class `List` that returns a sorted copy of the current list.

   Hints: Remember that the steps of Merge Sort are to (i) split a current array segment into two (roughly) equal parts, (ii) sort the parts by recursively applying Merge Sort, and (iii) merging the parts. Translate these steps into analogous operations on lists, while maintaining the asymptotic running time of the array algorithm.

2. Develop a version of Quicksort for head-tail lists. Realize it as a method

   ```
   HTList quickSort()
   ```

   of the class `HTList` that returns a sorted copy of the current list.

   Hint: The idea of Quicksort is to repeatedly choose an element of the current list as pivot and to partition the current list into two lists, one with elements less or equal than the pivot value and another one with elements greater or equal than the pivot value. Then, the two new lists are each sorted recursively and appended.

   Check which of the methods defined for your head-tail list type can be used to simplify the implementation of `quickSort`.

With head-tail lists, one can easily optimize Quicksort so that it has two desirable properties:

- it does not sort elements equal to the pivot and thus has linear running time if called with elements that are all equal;
- it is stable, that is, it does not change the order of elements with equal sorting key.

The second property does not come to bear in the simplified setting where all list elements are integers and there is no further content to list elements. However, you can implement the method in such a way that it would be stable if applied to elements with sorting key and additional fields, like people that are sorted according to their year of birth.

3. Develop such an optimized version of Quicksort for head-tail lists. Realize it as a method

   ```
   HTList quickSortOpt().
   ```

   of the class `HTList`. Explain how you realized the optimization with comments in the code.

(Weight: 40% of this CW)

Deliverables. Submit two copies of your code:

- one via Codeboard (instructions are available here),
- one via the OLE submission page of your lab.

Submission until Monday, 18 December 2022, 23:55, to Codeboard and the OLE submission page.