intro2cs/introcsp/introcst
Ex10: Data Structures

Objectives

This purpose of this exercise is to practice the use and implementation of dynamic data structures, specifically, linked lists. Along the way, we also exercise using nested classes.

Exercise Description

Part 1 - algorithms on linked lists

In this part you will implement a few operations on singly linked lists.

We provide you with a class List (source code) that implements a singly linked list (each node only has a reference to the next node in the list) as defined by its API and the Node API. Node is a nested class of List. Note that the type of the data item stored inside a Node is String.
Your task is to implement the static methods in the ListUtil class according to its API and the description below. You may assume that ListUtil is part of the same package as List, and therefore may access List's protected data members directly.

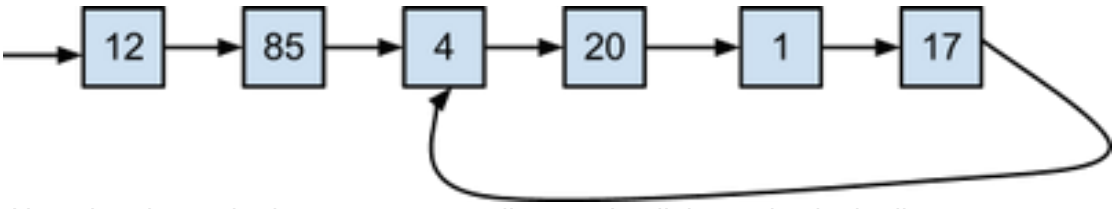
You may assume the list contains no cycles in all methods except the **`containsCycle()`** method.

Here are some more details on methods that you should implement. Note, that in all these methods (unless specified) you should not change the values stored in any of the nodes or replace any nodes in the list (except in the **`mergeLists()`** method, which creates a new list).

1. **`public static List mergeLists(List first, List second)`** – merges two sorted (in ascending order) lists into one new sorted list in an ascending order.

2. **`public static boolean containsCycle(List list)`** – detects if a list contains a cycle. A list contains a cycle if at some point a Node in the list points to a Node that already appeared in the list.
   Hint: This problem is very popular in job interviews. You may find an idea for a solution in the Internet.

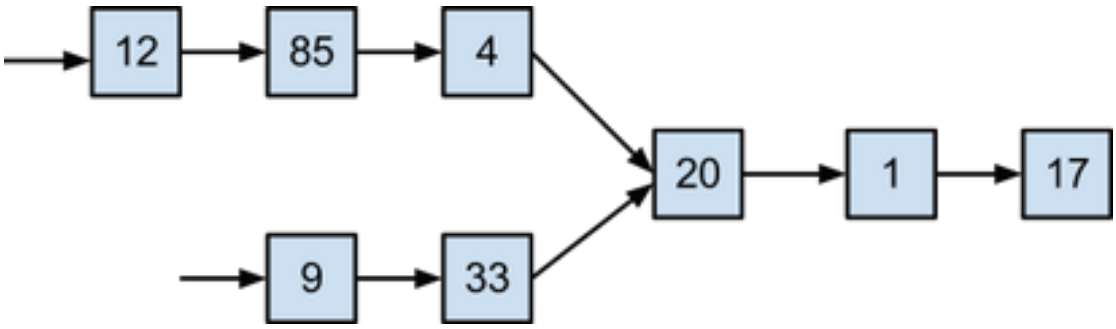
   For example, this method should return true on the following list:

   Note that the cycle does not necessarily contain all the nodes in the list.
3. **`public static void reverse(List list)`** – this method reverses the given list (so the head becomes the last element, and every element points to the element that was previously before it). The complexity of your implementation should be O(n).

4. **`public static boolean isPalindrome(List list)`** – detects if a list is a palindrome. A list is a palindrome if for j=1…n/2 (where n is the number of elements in the list) the element in location j equals to the element in location n-j. Note that you should compare the values stored in the nodes and not the node objects themselves.

5. **`public static boolean haveIntersection(List first, List second)`** – checks if two lists intersect. Two lists intersect if at some point they start to share nodes. Once two lists intersect, they become a single list from that point on and can no longer split apart. For example, the two lists in the next example intersect:



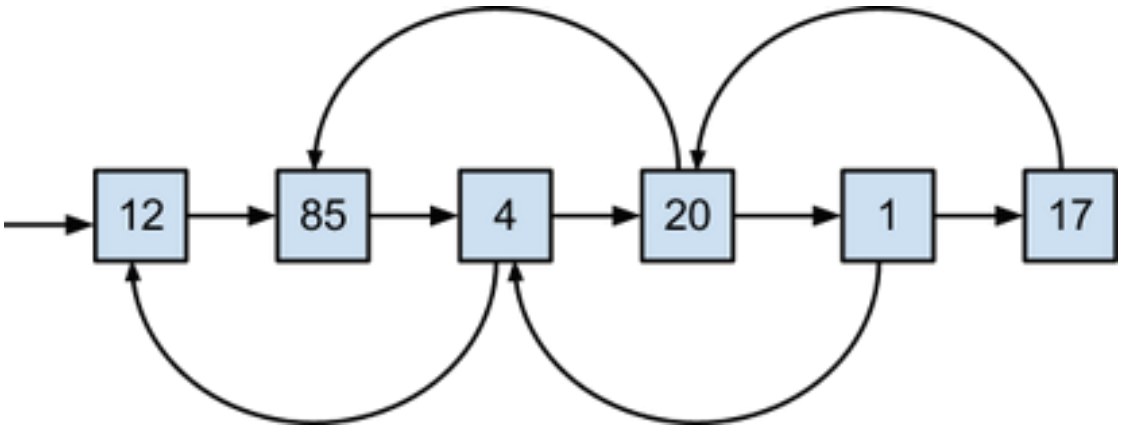
Note that two lists might intersect even if their lengths are not equal, just as in the example above.

6. **`public static void mergeSort(List list)`** – sorts the list into ascending order using the merge-sort algorithm. Make sure that you use the merge-sort algorithm and not any other sorting algorithm. Recall that merge-sort should run at O(n log n).

You should explain the complexity of your implementation for each of the above methods in the README file. A non-efficient implementation may not get a full score.

Part 2 - implementing a special linked list

In this part, you will implement a special linked list, called a **SkipiList**. A SkipiList is composed of **Node**s (a nested class of SkipiList), according to the provided APIs(SkipiList and SkipiList.Node). SkipiList is a special variant of a doubly-linked list, where each Node has one pointer to the next Node in the list, and another pointer to the Node two positions before (**prev-prev)** in the list (hence the prefix “skipi”), which is called the skipBack pointer. In addition, a SkipiList has pointers both to the head and the tail of the list. Initially, the list is created empty with both of these pointers set to **`null`**. The tail node’s next pointer is always **`null`** (in the figure below, note that there is no right arrow from the node whose data is 17). The skipBack pointers of the first two nodes in the list are also always **`null`** (in the figure below, note that there are no left curve arrows from the nodes whose data are 12 and 85).



Testing

It is recommended to check your JAR file by copying it to an empty directory, and running the automatic testers by executing the command: `~introcsp/bin/testers/ex10 ex10.jar` and verifying that all the tests pass successfully.

You may also download the tests JAR from here. Extract the contents of the JAR file using the shell command: `jar xvf ex10testing.jar` and follow the instructions in the file called **`TESTING.`**

Submission

Before you submit, remember to use constants where appropriate and refrain from duplicating code. Make sure to thoroughly test all the classes you implemented.

Don't forget to document your code (including javadoc) and follow all course coding style guidelines.

Submit a JAR file called **`ex10.jar`** containing **only** the files:

- **`ListUtil.java`**
- **`SkipiList.java`**
- **`README`**

Use the "Upload File" link on the course home page, under the ex10 link.

Good luck!