Com S 228

Project 3: A Stout List (100 pts)

> Due at 11:59 p.m. on Wednesday, Apr 17 (Saturday, Apr 20 with automatic extension)

If you need help, see one of the instructors or TAs. Please make sure you understand the "Academic dishonesty" section of the syllabus. The course Piazza page is a good place to post general questions. Please do not post or attach any source code for the assignment. Watch Canvas announcements and Piazza for corrections, hints, and answers to common questions. Since no test code is being submitted for this assignment, you are welcome to post unit tests.

1. Introduction

The purposes of this assignment are

- to practice working with a linked data structure
- to become intimately familiar with the `List` and `ListIterator` interfaces

This assignment will also provide many opportunities to practice your debugging skills.

1.1. Summary of Tasks

Implement a class `StoutList` that extends `AbstractSequentialList` along with inner classes for the iterators. A skeleton is provided. See the implementation and suggestions for getting started sections for details.

1.2. Overview

In this assignment you will implement a somewhat peculiar-looking linked list. The list will be a doubly-linked list with dummy nodes for the head and tail. An empty list has the following form:

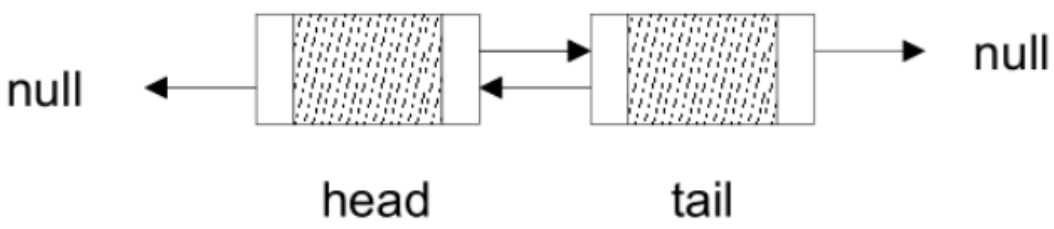

Figure 1- an empty list

So far so good. Now, let *M* be a fixed, positive even number. The twist is that each node can store up to *M* data elements, so the number of linked nodes may not correspond to the number of elements. For

example, after some sequence of add and remove operations, one of these lists might have the following form:

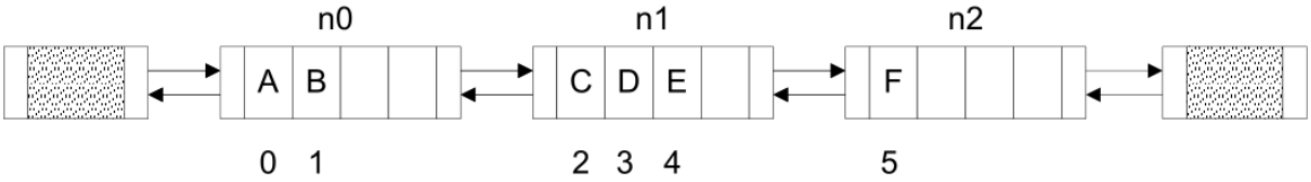

Figure 2 - a possible list of size 6, where M=4

Note that there are 6 elements, and their logical indices are shown below the nodes. The number of actual linked nodes may vary depending on the exact sequence of operations. Each node contains an array of the element type having fixed length *M*. Therefore, each logical index within the list is represented by two values: the node n containing the element, and the offset within that node's array. For example, the element E shown above at logical index 4 is in node n_1 at offset 2. There are special rules for adding and removing elements to ensure that all nodes, except possibly the last one, have at least $M/2$ elements. These rules are described in detail in a later section. Note that you can get started and make significant progress on the assignment without needing all the add and remove rules. See the section suggestions for getting started.

2. Implementation of StoutList

For the implementation, you must implement the class `StoutList` extending `AbstractSequentialList`. `AbstractSequentialList` is a partial implementation of the `List` interface in which the `size()` and `listIterator()` methods are abstract. All other operations have default implementations using the list iterator. The `StoutList` should NOT allow `null` elements. Your add methods (those within the iterator as well as those implemented without the iterator) should explicitly throw a `NullPointerException` if a client attempts to add a null element. A skeleton of the code can be found in project3_template.zip. The skeleton code includes a constructor, an inner class Node, and two methods `toStringInternal()`. There is also some code in place to support *logging*.

2.1. Methods to override

In addition to implementing the iterators described below, you must override the following methods of AbstractList *without* using the iterator:

```
int size()
boolean add(E item)
void add(int pos, E item)
E remove(int pos)
Iterator<E> iterator()
ListIterator<E> listIterator()
ListIterator<E> listIterator(int pos)
```

The methods above must conform to the `List` interface, with the added restriction that the `add()` methods must throw a `NullPointerException` if the item argument is `null`. The purpose of asking you to override `add()` and `remove()` is primarily to help ensure that you can get partial credit for implementing the split and merge strategies even if your iterator isn't completely correct.

2.2. The Node inner class

A minimal `Node` class is provided for you. It has methods for adding an element at a given offset and removing an element at a given offset. Note that empty cells within the array (i.e., those with index >= count) should always be `null`.

You can add additional features to this class if you find it helpful to do so.

2.3. The toStringInternal methods

These methods show the internal structure of the nodes and are useful for debugging. Normally, such a method would not be public (since it reveals implementation details), but we are making it public to simplify unit testing. For example, if the list of Figure 3 contained String objects “A”, “B”, “C”, “D” and “E”, an invocation of `toStringInternal()` would return the string:

[(A, B, -, -), (C, D, E, -)]

where the elements are displayed by invoking their own `toString()` methods and empty cells in the array inside each node (which should always be `null`) are displayed as “-”. A second version takes a `ListIterator` argument and will show the cursor position as a character “|” according to the `nextIndex()` method of the iterator. For example, if iter is a `ListIterator` for the list above and has `nextIndex()` = 3, the invocation `toStringInternal(iter)` would return the string:

[(A, B, -, -), (C, | D, E, -)]

Do not modify these methods.

2.4. Finding indices

Your index-based methods `remove(int pos)`, `add(int pos, E item)` and `listIterator(int pos)` method must not traverse every element in order to find the node and offset for a given index; you should be able to skip over nodes just by looking at the number of elements in the node. For example, for the list of Figure 9, to find the node and offset for logical index 7, you can see 3 elements in the first node, plus 2 in the second node, which makes 5, plus 4 in the third node, which is 9. Since 7 is greater than 5 and less than or equal to 9, index 7 must be in the third node. You may find it helpful to represent a node and offset using a simple inner class similar to the following:

```
private class NodeInfo
{
public Node node;
public int offset;
    public NodeInfo(Node node, int offset)
    {
        this.node = node;
        this.offset = offset;
    }
}
```

Then you can create a helper method something like the following:

```
// returns the node and offset for the given logical index
NodeInfo find(int pos){...}
```

3. Implementation of StoutIterator and StoutListIterator

You must provide a complete implementation of an inner class `StoutIterator` implementing `Iterator<E>` and a class `StoutListIterator` implementing `ListIterator<E>`. The `StoutIterator` does NOT need to implement `remove()` (can throw `UnsupportedOperationException`). Optionally, if you are confident that your `StoutListIterator` is correct, you can return an instance of `StoutListIterator` from your `iterator()` method and forget about `StoutIterator`. However, you are encouraged to keep them separate when you first start development, since the basic one-directional Iterator, without a remove operation, is relatively simple, while the add and remove operations for the full `ListIterator` are tricky.

4. Suggestions for Getting Started

1. Implement `add(E item)`. Adding an element at the end of the list is relatively straightforward and does not require any split operations. (e.g. see Figures 3 – 5). You can use `toStringInternal()` to check.

2. Implement the `hasNext()` and `next()` methods of StoutIterator and implement the `iterator()` method. At this point the List methods `contains(Object)` and `toString()` should work.

3. Start StoutListIterator. Implement ListIterator methods `nextIndex()`, `previousIndex()`, `hasPrevious()`, and `previous()`. These methods are straightforward. Implement the `listIterator()` method. You should then be able to iterate forward and backward, and you can check your iterator position using `toStringInternal(iter)`. The `indexOf(Object obj)` method of List should work now.

4. Implement the `set()` method of `StoutListIterator`. You will need to keep track of whether to act on the element before or after the cursor (and possibly throw an `IllegalStateException`), but the method is not complicated since it doesn't have to change the structure of the list or reposition the cursor.

5. Implement a helper method such as the "find" method described above under finding indices. Then you can easily implement the `listIterator(int pos)` method. After that, the `get(int pos)` of List should work.

6. Implement the `add(int pos, E item)` method. Now you will need to carefully review the rules for adding elements. Here is a suggestion for keeping things organized. Write a helper method whose arguments include the node and offset containing the index pos at which you want to add, e.g., `private NodeInfo add(Node n, int offset, E item){...}`
   (If pos = size, the arguments should be the tail node and offset 0.) The return value should be the actual node and offset at which the new element was placed. (In some cases this will be the given node and offset, in some cases it will be the previous node, and in case of a split there might be a completely new node.) You don't need the return value for the add method, but you will need it in the iterator.

7. Implement the `remove(int pos)` method. Again, you will need to carefully review the rules for removing elements and merging.

8. Implement the `add()` method of listIterator. This is not too bad if you have the helper method from (6). The catch is that after adding an element, you have to update the logical cursor position to the element after the one that was added.

9. Implement the `remove()` method of listIterator. The tricky part is that you have to update the cursor position differently depending on whether you are removing ahead of the cursor or behind the cursor, and depending on whether there was a merge operation.

5. The add and remove Rules

5.1. Adding an element (see Figures 3 – 8)

The rules for adding an element *X* at index are as follows. Remember that adding an element without specifying an index is the same as adding at index *i* = size. For the sake of discussion, assume that the logical index = size corresponds to node = tail and offset = 0.

Suppose that index occurs in node *n* and offset *off*. (Assume that index = size means *n* = tail and *off = 0*)

- if the list is empty, create a new node and put *X* at offset 0
- otherwise if *off = 0* and one of the following two cases occurs,
    - if *n* has a predecessor which has fewer than *M* elements (and is not the head), put *X* in *n*'s predecessor
    - if *n* is the tail node and *n*'s predecessor has *M* elements, create a new node and put *X* at offset 0
- otherwise if there is space in node *n*, put *X* in node *n* at offset *off*, shifting array elements as necessary
- otherwise, perform a *split* operation: move the last *M/2* elements of node *n* into a new successor node *n'*, and then
    - if $\text{off} \leq M/2$, put *X* in node *n* at offset *off*
    - if $\text{off} > M/2$, put *X* in node *n'* at offset $(\text{off} - M/2)$

5.2. Removing an element (see Figures 9 – 14)

The rules for removing an element are:

- if the node *n* containing *X* is the last node and has only one element, delete it;
- otherwise, if *n* is the last node (thus with two or more elements) , or if *n* has more than $M/2$ elements, remove *X* from *n*, shifting elements as necessary;
- otherwise (the node *n* must have at most $M/2$ elements), look at its successor *n'* (note that we don't look at the predecessor of *n*) and perform a *merge* operation as follows:
    - if the successor node *n'* has more than $M/2$ elements, move the first element from *n'* to *n*. (mini-merge)
    - if the successor node *n*' has $M/2$ or fewer elements, then move all elements from *n*' to *n* and delete *n*' (full merge)

5.3 Examples: adding some elements to a list

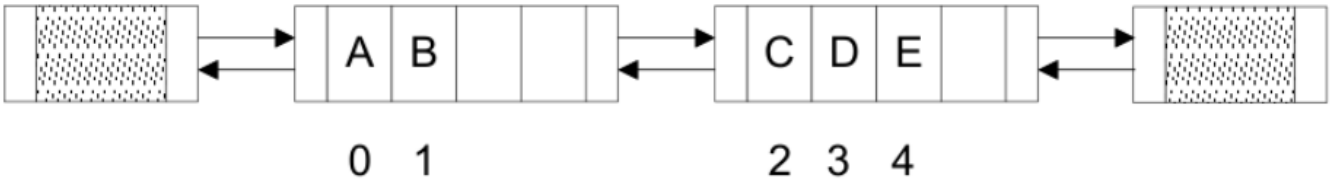

Figure 3- an example list

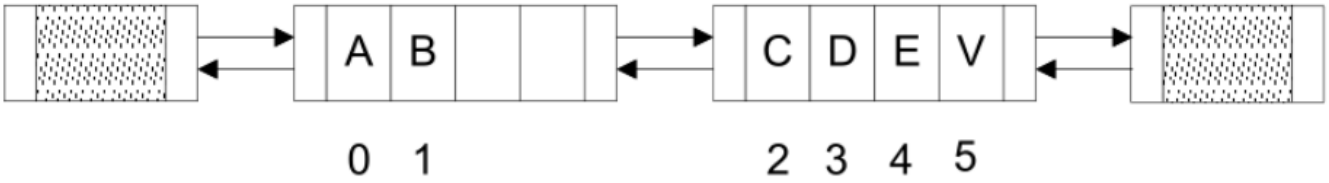

Figure 4 - after add(V)

Figure 5 - after add(W)

Figure 6 - after add(2, X)

Figure 7 - after add(2, Y)

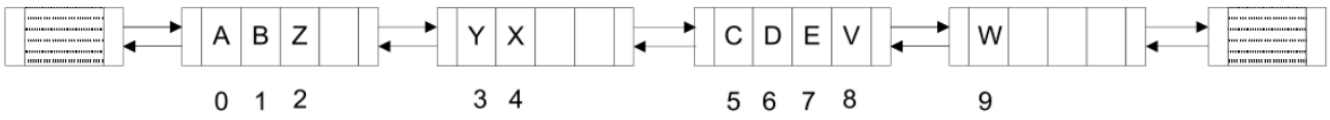

Figure 8 - after add(2, Z)

5.4. Examples: removing some elements from a list

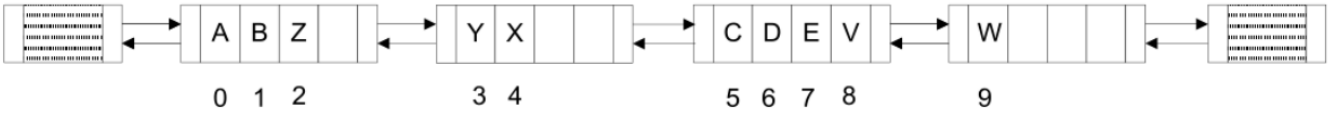

Figure 9 - example list

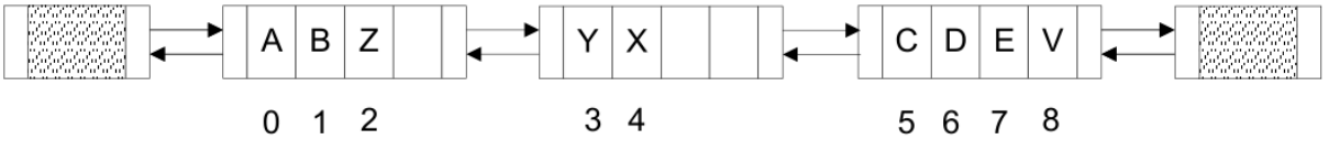

Figure 10 - after removing W

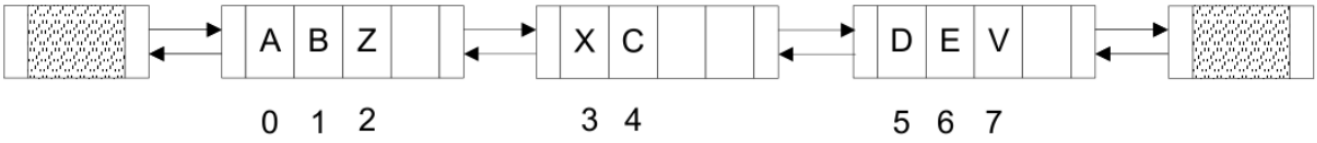

Figure 11 - after removing Y (mini-merge)

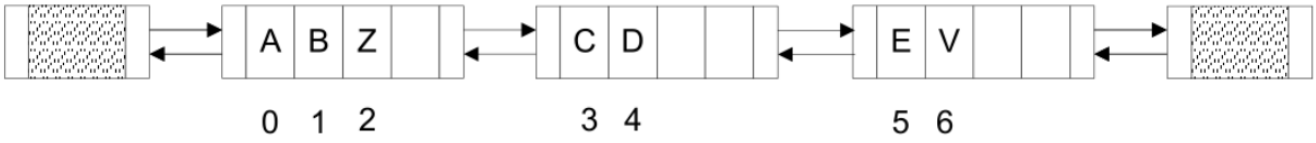

Figure 12 - after removing X (mini-merge)

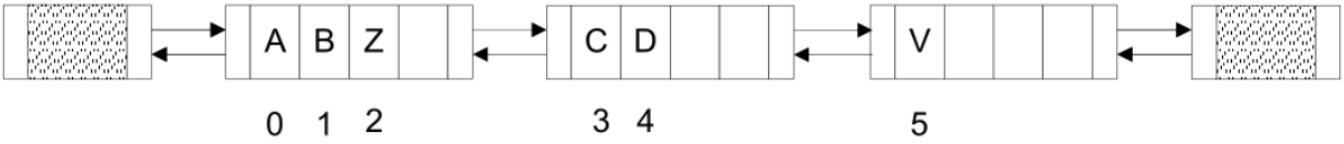

Figure 13 - after removing E (no merge with predecessor node)

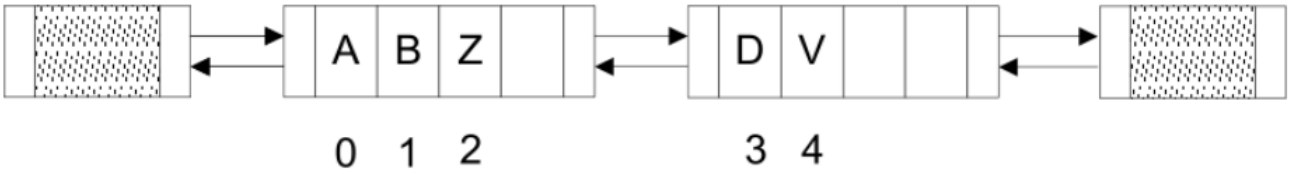

Figure 14 - after removing C (full merge with successor node)

6. Sorting

Implement the following two sorting methods within the `StoutList` class:

```
    public void sort();
    public void sortReverse();
```

The `sort()` method implements insertion sort by calling a private generic method:

```
private void insertionSort(E[] arr, Comparator<? super E> comp);
```

On return from calling `sort()`, the elements in the stout list are in the *non-decreasing* order, and every node (except the last one) stores the maximum number of elements. One implementation consists of the following steps:

- Traverse the list and copy its elements into an array.
- Destroy all storage nodes in the list during the traversal.
- Sort the array.
- Create new nodes in the list and add elements back from the sorted array.

The `reverseSort()` method sorts elements in the non-increasing order using *Bubble Sort*, which is described in Section 6.1. It calls another private generic method:

```
private void bubbleSort(E[] arr);
```

In `bubbleSort()` you are required to use the `compareTo()` method from an expected implementation of the `Comparable` interface by `E` or `? super E`. The list must, after the sorting, have every node (except the last one) storing elements to its full capacity, just as is required for the `sort()` method.

6.1. Bubble Sort

Bubble sort is performed in multiple passes. Suppose we want to sort an array of elements in the non-decreasing order (which is opposite to the order in `bubbleSort()`). In every pass, it steps through the entire list, comparing each pair of adjacent numbers and swapping them if the preceding number in the pair is greater than the succeeding one. If no swaps have been performed during a pass, the list is sorted and the algorithm terminates. Intuitively, the algorithm works in a way that smaller elements "bubble" to the front of the list. Wikipedia offers a step-by-step example to illustrate the working of the algorithm. It also has some cool simulations. Below is a bigger example from *The Art of Computer Programming*, vol. 3, 2nd edition, p. 106, authored by Donald E. Knuth.

```
503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703 (input)

087 503 061 512 170 897 275 653 426 154 509 612 677 765 703 908 (pass 1)

087 061 503 170 512 275 653 426 154 509 612 677 765 703 897 908 (pass 2)

...

061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908 (pass 9)
```

As illustrated in the above, each pass of bubble sort guarantees to put the next biggest element in place. The algorithm has worst-case complexity $O(n^2)$ for an input of n integers.

7. Unit Tests

You are strongly encouraged to write unit tests as you develop your solution. However, you do not have to turn in any test code. You may therefore post your tests on Piazza, should you wish to share them with your

colleagues.

8. Documentation and Style

Up to 10% of the points will be for documentation and style. See the style guidelines posted on Canvas.

9. Grading

You are strongly encouraged to develop your code incrementally as outlined in the Suggestions for getting started. You can get partial credit for working features at the top of the list, since they can be tested even if features toward the bottom of the list are not implemented. We will be unit testing your code to check conformance with the `List` and `ListIterator` API methods, and many of these methods will work correctly even if your iterator does not support add and remove. Note that we will also check the internal structure of your list (conformance with the add and remove rules) via the `toStringInternal` methods.

The iterator's add and remove methods, which can be somewhat challenging to get right, will be worth at most 10% of the total points.

10. What to Turn In

All of your code should be in the one class `edu.iastate.cs228.hw3.StoutList`. Submit a zip file, containing your source code only, in the correct package structure:

`edu/iastate/cs228/hw3/StoutList.java`

> **Remember:** CHECK your submission after it is uploaded to Canvas by downloading it and looking at it. Do not submit .class files. Submit a **zip** file, **not** a rar or tar or gzip or 7z or anything else.