CS2001/CS2101 Week 8 Practical

Linked Lists

4th November 2020

Matriculation Number: 200002548

Tutor: Alex Konovalov

Contents

1 Introduction

The aim of this practical was to implement a recursive and an iterative version of a linked list manipulator `IListManipulator`.These were called `RecursiveListManipulator` and `IterativeListManipulator` respectively. The recursive implementation should use recursion (implicitly calling itself) to repeat operations, whereas the iterative implementation should instead use explicit iteration (i.e. conditional and unconditional loops). The ADT `IListManipulator` outlined the methods that were to be featured within both implementations, these are listed below:

i `size(ListNode head)`

ii `contains(ListNode head, Object element)`

iii `count(ListNode head, Object element)`

iv `convertToString(ListNode head)`

v `getFromFront(ListNode head, int n)`

vi `getFromBack(ListNode head, int n)`

vii `equals(ListNode head1, ListNode head2)`

viii `containsDuplicates(ListNode head)`

ix `append(ListNode head1, ListNode head2)`

x `reverse(ListNode head)`

xi `split(ListNode head, int n)`

xii `flatten(ListNode head)`

xiii `isCircular(ListNode head)`

xiv `containsCycles(ListNode head)`

xv `sort(ListNode head, Comparator comparator)`

xvi `map(ListNode head, IMapTransformation transformation)`

xvii `reduce(ListNode head, IReduceOperator operator, Object initial)`

xviii `getFromFront(ListNode head, IFilterCondition condition)`

Basic tests were already provided within the `ListManipulator` class. This allowed for the Test Driven Development (TDD) methodology to be used for implementation. Therefore, unit tests and implementation of the interfaces were carried out sequentially, for all methods in each interface. This should produce robust recursive and iterative implementations of `IListManipulator`.

A `ListNode` class was provided to construct linked lists with. This class featured a `element` and a `next` attribute. These correspond to the data stored by an instance of a `ListNode` and a reference to the next node in the singly linked list respectively. This reference to the next node allows for the linked list to be traversed node by node from the start of the list, as long as the `head` node is maintained. The `IListManipulator` implementations should know when the end of a list has been reached when the `next` attribute equals `null`.

2 Design

As mentioned in the introduction, the two classes `RecursiveListManipulator` and `IterativeListManipulator`, were to be implemented. The latter class was implemented first, since the iterative implementation should make it easier to visualise the flow of execution and is more natural to program. Overall the design of both implementations aims to be as maintainable and efficient as possible. There were some further considerations to be made about the design where the specification became vauge:

- Apart from specific instances whereby the `IListManipulator` specified that a method should not alter the original list, all methods in both the iterative and recursive implementations do not alter the original list(s), whose head(s) is/are passed in.
- The `flatten()` method should only be expected to flatten a list of lists into one list and not any deeper list structures (i.e. list of lists of lists).
- Only the `isCircular()` and `containsCycles()` methods should deal with cyclic lists. All other methods should expect linear linked lists.
- The `IMapTransformation`, `IReduceOperator` and `IFilterCondition` arguments passed into the respective `map()`, `reduce()` and `filter()` methods will be designed to work with the type of objects in the input list.
- A list with zero or one nodes cannot be reversed, split, considered cyclic or considered circular.
- A list with no nodes can already be considered flattened, however a list of one list can still be flattened into a list.

2.1 IterativeListManipulator class

Implementation of the iterative design was straight forward. The lists were enumerated through using a while loop that ran if the current node was not equal to `null`. Then within the loop the list was moved through by setting the current node to the next node in the list using `head = head.next`. If a method needed to iterate through to the n-th node in the list a for loop was used. This loop would move through the list using `head = head.next` until the iterator `i` was equal to `n`. This loop requires caution, as `n` must first be checked to see if it is in the bounds of the list (i.e. greater than or equal to zero and less than the size of the list).

As well as the method specified in the `IListManipulator` interface, other private methods were added to achieve the desired functionality from these public methods. The specifics of the implementation of non-trivial methods is detailed below:

2.1.1 The Size() method

The `size` method was implemented by iterating through the list until the end of the list was reached meanwhile keeping track of the number of iterations and thereby number of nodes in the list. This operation has a complexity of O(n) from iterating through all of the array. Provided this size method was defined within some linked list class that featured add and remove operations the size could have been stored in a variable and maintained. Leading to a time complexity of $O(1)$ for size. However this was not possible within this program as there is no larger linked list class and the specification states that the `size` method should work on any arbitrary list by passing in a `head` node to represent the list.

2.1.2 The getFromFront() method

This method returns the n-th element from the front of the list. It uses a for loop to iterate over the list. If the value of `n` is out of the bounds of the list an `InvalidIndexException` is thrown.

2.1.3 The getFromBack() method

This method returns the n-th element from the back of the list. This method reuses the `getFromFront()` method to reduce complexity, passing in the `size()` of the list, minus `1`, minus the value of `n`.

2.1.4 The equals() method

This method checks if two lists `head1` and `head2` are equal It does this by iterating over both lists at the same time and checking if the nodes contain the same value. If either or both of the next nodes in the lists (`head1` or `head2`) are equal to null then the loop is terminated and the method returns the result of comparing the two last nodes to be iterated over. If lists have identical length then this comparison will return true since both lists were iterated over concurrently so both nodes will be `null`. If `false` this means that one list is not currently equal to `null` and so is longer than the other.

2.1.5 The containsDuplicates() method

Iteratively checks if the list contains any duplicate nodes (nodes that have the same element). This is accomplished by calling the `contains()` method on every node in the list. Therefore for every node in the list, the method checks to see if that node has an element appears in another node in the list.

2.1.6 The append() method

Iteratively joins two lists together, with the second list joined to the back of the first list. It does this by moving to the end of the first list and setting the last node of the first list to reference the head of the second list, thereby creating a longer joined list.

2.1.7 The reverse() method

This method iteratively reverses through a linked list. It does this by iterating through the list and keeping a reference to the current node, next node and previous node in the list. The next node is set to the node after the current node. Then the current node is set to reference the previous node. Then the previous node is set to the current node and the current node set to the next node to move along the list. In the end the previous node should point to the head of the reversed list.

2.1.8 The split() method

Splits the list at the n-th node into two sub-lists and returns a list of those two sub-lists. This method does not change the original list. The method works by calling the iterative method copy that creates a copy of the initial list. Then the method iterates through the copied list to find the node that the list should be split at. A copy of the node after the split node is then created, this is the head of the second list. Finally the split node is set to point to null, creating the separate lists. These lists are then appended together into one larger list.

2.1.9 The copy() method

Creates a new copy of the list iteratively and returns the head `listNode` of this copied list. Is a private iterative method for use within the `split()` method.

2.1.10 The flatten() method

This iterative method converts a list of lists down into a single list, where ordering within the lists is preserved. This method iteratively steps through the list and appends each sub-list to the new list until the list doesn't have any more sub-lists to append. Then the new list is returned. This method does not change the original list passed in as an argument.

2.1.11 The isCircular() method

An iterative method that returns a boolean value, stating if the linked list is perfectly circular (not just cyclic). This method uses an iterative version of

Floyd's cycle-finding algorithm with a $O(n)$ time complexity and $O(1)$ space complexity. This algorithm entails moving through the list iteratively with a node that moves round quickly and one that moves round slowly. If at any point the `slow` node equals the `fast` node the a loop must be present within the list. The method then checks if both the `fast` and `slow` and `head` node are all equal, since if the loop is cyclic and cycles right round back to the head of the list then it is perfectly circular. If at any point the fast node points to a value of `null`, then the the method returns `false`, since if the loop is cyclic, no endpoint should be able to be reached. Note that an empty list or single node list is not considered circular.

2.1.12 The containsCycles() method

An iterative method that returns a boolean value, stating if the linked list is cyclic (has a loop in it). This method is identical to the above `isCircular()` method, however does not include the check if the list is cyclic from the `head` node.

2.1.13 The sort() method

Uses bubble sort for sorting as it is an iterative algorithm. Therefore this `sort()` method has a time complexity of $O(n^2)$. Whilst bubble sort has a worse average time complexity than merge and quick sort, it is easier to implement and runs with the iterative design of this class. Moreover its algorithmic simplicity means it is actually faster for smaller lists. This bubble sort algorithm involves swapping data rather than swapping references between nodes, this is because implementation is simpler. Bubble sort works by bubbling maximum values in the unsorted part of the list up to a sorted part of the list.

2.1.14 The map() method

An iterative method that creates a new list that whereby each element has been transformed a certain way from its initial value in the first list. Each iteration a new node is created whose element has been transformed using some `IMapTransformation`, this is then added to the end of the new list. If the end of the old array has been reached the recursive method returns the head to this new mapped list.

2.1.15 The reduce() method

An iterative reduce method that performs an operation using a `IReduceOperator` called `operator` to combine all elements in the list iteratively and add to all this an initial object to combine with. This combination is performed iteratively until the head variable is `null` and all of the list has been iterated through.

2.1.16 The filter() method

An iterative filter method to remove elements from create a new list of elements that all meet a certain `condition` of type `IFilterCondition` and filter out invalid elements. Each iteration if an element satisfies a condition in the old list it is appended to the new list. If the end of the old list has been reached, we return the list node that is the head of the new filtered list.

2.2 Recursive Implementation

The recursive implementation was more difficult to design as the recursive methods required more thought. However smaller methods like `contains()` and `size()` were trivial to implement, and were more readable than their iterative counterparts. The lists were implicitly iterated through recursion. Whereby each recursive step the next node was used as a parameter using `head = head.next`. A base case for each recursive method is included. This base case checks if the node passed in is equal to `null` and then returns, exiting the recursive method. Some method have a public auxiliary function, with the recursive implementation detailed in a private function. This auxiliary method helps set-up extra implementation and deal with some of the limitations of recursive functions.

Apart from their recursive structure, the recursive methods don't differ wildly from their iterative counterparts (most methods simply just replace a while loop with a recursive method and a if statement containing a base case). Therefore, only the specifics of methods that are completely different from their iterative counterparts are included below:

2.2.1 The reverse() method

This is the first recursive method that differs wildly from its iterative counterpart. In this method the list is traversed recursively, by calling `reverse()` and passing in `head.next` - the next node. When the recursive method gets to the base case, i.e. `head == null` or `head.next == null`, we return the last node, now the new head of the list. Then, now each recursive method call is returned, and the old heads next node points to the previous head node (now end node). Then the previous head node points to `null` (making it the end node). Finally the new head node is returned.

2.2.2 The split() method

This method is similar to its iterative counterpart, apart from it now relies on the `getIndex()` method to get the node of the index that it needs to split at.

2.2.3 The isCircular() method

This method is an auxiliary method for the private recursive `isCircular()` method. This private method has an extra two parameters, a `fast` and a `slow` node. These nodes are needed for Floyd's cycle-finding algorithm and in order to run this recursively, the auxiliary method is needed to set them up from the one `head` node it receives as per the interface design.

2.2.4 The sort() method

This method uses merge sort for sorting as is a recursive divide and conquer algorithm. Therefore this method has a time complexity of $O(nlog(n))$. Merge sort was preferred to quicksort because a split method was is already implemented. Merge sort has the advantage over quick sort in that it doesnt need any extra space when merging the linked lists since the ListNodes can be joined, so no new ListNodes have to be made, giving it a $O(1)$ space complexity (better than the array implementation!). Further more, a reliable quick sort algorithm with a minimal amount of pathological cases requires a random pivot. This random pivot cannot be simply accessed like in a sequential data structure, instead the list would need to be iterated through, actually increasing the complexity of the algorithm.

The merge sort algorithm works as follows. The algorithm first recursively splits the list in two, into smaller and smaller lists (using the already implemented `split()` method) until a base case is reached. This base case is where the length of the lists is one and cannot be split any more. Merge sort then merges all of these separate lists using the `merge()` method. Comparing and combining the lists into sorted larger lists recursively, until a whole sorted list is combined and returned.

2.2.5 The filter() method

This method is another auxiliary method for the private recursive `filter()` method. This private method has an extra parameter, a new head, which is initialised to `null`. The auxiliary method is needed to set up this new head. This leaves the auxiliary parameters unchanged so it keeps with the interface design.

3 Testing

Testing was split into two sections. There were the 22 base tests provided with the practical, and 12 extra tests created to ensure that my implementation met the requirements. The tests are organised under the tests directory within the project folder. This test directory contains a folder for the `base` tests and a folder for the `extra` tests. Each folder contains an abstract test class that features all of the JUnit tests as well as attributes needed for testing and a `@BeforeEach` method that is ran at the start of each test. The other two files in each folder are the concrete implementations of the abstract test class, one for testing the `IRecusiveListManipulator` and the other for testing the `IIterativeListManipulator`. The results from all these tests show that implementation has proven to be successful as all 64 tests in total passed, as shown in figure 1 and 2.

```
│
├─ JUnit Jupiter pass
│  ├─ RecursiveListManipulatorTest pass
│  │  ├─ append() pass
│  │  ├─ equals() pass
│  │  ├─ filter() pass
│  │  ├─ getCountingBackwardsEmptyList() pass
│  │  ├─ getCountingBackwards() pass
│  │  ├─ reduce() pass
│  │  ├─ containsCycles() pass
│  │  ├─ getCountingForwardsEmptyList() pass
│  │  ├─ flatten() pass
│  │  ├─ getCountingForwards() pass
│  │  ├─ contains() pass
│  │  ├─ getCountingBackwardsIndexTooLarge() pass
│  │  ├─ map() pass
│  │  ├─ size() pass
│  │  ├─ sort() pass
│  │  ├─ count() pass
│  │  ├─ split() pass
│  │  ├─ convertToString() pass
│  │  ├─ reverse() pass
│  │  ├─ containsDuplicates() pass
│  │  ├─ getCountingForwardsIndexTooLarge() pass
│  │  └─ isCircular() pass
│  ├─ IterativeListManipulatorTest pass
│  │  ├─ append() pass
│  │  ├─ equals() pass
│  │  ├─ filter() pass
│  │  ├─ getCountingBackwardsEmptyList() pass
│  │  ├─ getCountingBackwards() pass
│  │  ├─ reduce() pass
│  │  ├─ containsCycles() pass
│  │  ├─ getCountingForwardsEmptyList() pass
│  │  ├─ flatten() pass
│  │  ├─ getCountingForwards() pass
│  │  ├─ contains() pass
│  │  ├─ getCountingBackwardsIndexTooLarge() pass
│  │  ├─ map() pass
│  │  ├─ size() pass
│  │  │
```

Figure 1: All base tests passing

```
│    │    ├─ sort() pass
│    │    ├─ count() pass
│    │    ├─ split() pass
│    │    ├─ convertToString() pass
│    │    ├─ reverse() pass
│    │    ├─ containsDuplicates() pass
│    │    ├─ getCountingForwardsIndexTooLarge() pass
│    │    └─ isCircular() pass
│    ├─ ExtraRecursiveListManipulatorTest pass
│    │    ├─ getFromBackInvalidIndex() pass
│    │    ├─ convertToStringWithListOfNulls() pass
│    │    ├─ complexReduce() pass
│    │    ├─ getFromFrontInvalidIndex() pass
│    │    ├─ convertToStringWithDifferentTypes() pass
│    │    ├─ countDealWithDifferentListAndArgumentTypes() pass
│    │    ├─ containsDuplicatesListOfNulls() pass
│    │    ├─ mapNull() pass
│    │    ├─ countDealWithDifferentTypes() pass
│    │    ├─ mapDifferentTypes() pass
│    │    ├─ appendListOfNulls() pass
│    │    └─ countDealWithNullsAndListsOfNulls() pass
│    └─ ExtraIterativeListManipulatorTest pass
│         ├─ getFromBackInvalidIndex() pass
│         ├─ convertToStringWithListOfNulls() pass
│         ├─ complexReduce() pass
│         ├─ getFromFrontInvalidIndex() pass
│         ├─ convertToStringWithDifferentTypes() pass
│         ├─ countDealWithDifferentListAndArgumentTypes() pass
│         ├─ containsDuplicatesListOfNulls() pass
│         ├─ mapNull() pass
│         ├─ countDealWithDifferentTypes() pass
│         ├─ mapDifferentTypes() pass
│         ├─ appendListOfNulls() pass
│         └─ countDealWithNullsAndListsOfNulls() pass
└─ JUnit Vintage pass
Test run finished after 160 ms
[         6 containers found      ]
[         0 containers skipped    ]
[         6 containers started    ]
[         0 containers aborted    ]
[         6 containers successful ]
[         0 containers failed     ]
[        68 tests found           ]
[         0 tests skipped         ]
[        68 tests started         ]
[         0 tests aborted         ]
[        68 tests successful      ]
[         0 tests failed          ]
```

Figure 2: Extra tests passing and testing synopsis

3.1 Extra IListManipulator Tests

The base tests were already very detailed an provided a lot of coverage of test cases. They dealt with `null` being passed into a method as a list node, all exceptions that could be thrown. The extra test cases I wanted to perform where on for different `Object`'s such as `characters` and `floats`, to verify my implementation wasn't specific to one type of `Object`. I also wanted to test how the program would cope with a list containing just one `null` list of multiple `nulls`.

- countDealWithNullsAndListsOfNulls()
  - Tests if count() can deal with null values and lists of null values.
  - Is important to test as a null list should still be counted and return zero. A list of nulls should also be accepted by the method.
- countDealWithDifferentTypes()
  - Tests if count() can deal with different types. The method should be able to count objects of any type. This is to make sure that the internal comparison of objects is correct.
  - This is important to test for as the implementation states that the program should work for any `Object`.
- countDealWithDifferentListAndArgumentTypes()
  - Tests if count() can deal with a list having elements of one type and the item to count for being another type.
  - This is important to test for as the implementation states that the program should work for any `Object`. Also the argument being a different type should not impact the method.
- convertToStringWithListOfNulls()
  - Tests if convertToString() can convert a list of nulls and an empty list into suitable a string.
- convertToStringWithDifferentTypes()
  - Tests if convertToString() can convert a lists of different types into suitable a string.
- convertToStringWithDifferentTypes()
  - Tests if convertToString() can convert a lists of different types into suitable a string.
- getFromFrontInvalidIndex()

- Tests if getFromFront() throws a IndexToLarge error when trying to access index to large.

- getFromBackInvalidIndex()
  - Tests if getFromBack() throws a IndexToLarge error when trying to access index to large.
- appendListOfNulls()
  - Tests is append() can append a list of nulls.
- mapNull()
  - Tests if map() can handle a null value.
- mapDifferentTypes()
  - Tests if map() can work with different types, provided a new IMapTransformation.
- complexReduce()
  - Tests is reduce() can work with more complex IReduceOperator's.
- containsDuplicatesListOfNulls()
  - Tests if containsDuplicates() can deal with null values. Nulls are not considered elements and so contains should be skipped.

4 Conclusion

In this practical I implemented a recursive and iterative version of a linked list manipulator `IListManipulator`. All of the methods outlined in the interface were implemented both recursively and iteratively. The recursive version made use of recursion (implicitly calling itself) to repeat operations and return a desired result. The iterative version used conditional and unconditional loops (explicit iteration). These `RecursiveListManipulator` and `IterativeListManipulator` classes were implemented using TDD, whereby the tests were a collection of provided tests and extra tests for testing more extreme and exceptional test cases.

Implementation in this project was easy at first as the initial iterative and recursive methods outlined in the interface were almost trivial to implement. The implementation soon got harder though and towards the end was definitely testing my understanding of linked lists. The recursive methods were difficult for me to implement. However once you realise how they work, it becomes quite easy to adapt an iterative loop into a recursive method. There is a certain pattern that is similar between the two implementations (for most methods that is). The recursive methods at the end of the `RecursiveListManipulator` were very gruelling to design, especially with some needing auxiliary methods, keeping track of what's being returned at what point from the methods became challenging. That being said, I have definitely learnt a considerable amount about how linked lists work and the nature of designing recursive algorithms.

While it was interesting to implement a class that can perform operations to manipulate linked lists, in the future I would rather implement a whole linked list data structure and provide methods from within the data structure to alter it (i.e. `LinkedList.sort()`). Furthermore, I think it would be interesting to detail exactly the steps involved between transitioning from an iterative to a recursive method. Then create some sort of algorithm that would allow for recursive methods to be automatically generated from basic iterative methods.

References

[1] Wayne Snyder, *Recursion and Linked Lists* , n.a., accessed 04-11-2020, `https://www.cs.bu.edu/fac/snyder/cs112/CourseMaterials/LinkedListNotes.Recursion.html`

[2] JetBrains, *Prepare for testing*, 19-08-2020, accessed 04-11-2020, `https://www.jetbrains.com/help/idea/testing.html`

[3] Educative, *How to sort a linked list using merge sort*, n.d., accessed 04-11-2020, `https://www.educative.io/edpresso/how-to-sort-a-linked-list-using-merge-sort`

[4] Christian Stigen Larsen, *What's the fastest algorithm for sorting a linked list?*, 06-10-2009, accessed 04-11-2020, `https://stackoverflow.com/q/1525117`

[5] Abhijit Rao, *How to detect a loop in a linked list?*, 18-04-2010, accessed 04-11-2020, `https://stackoverflow.com/q/2663115`