CMPT 115 - Assignment #3

Summer 2016

Assignment Marks: 33

Part 1 Written Questions

Reminder: whenever we ask you to "write an algorithm", we want a complete algorithm, including a header, written in pseudocode. We will deduct 25% if your algorithms are in C++. Remember, pseudocode is a tool for you to focus on larger issues, rather than syntax and minutiae related to coding. This 25% penalty is an attempt to get you to practice designing an algorithm before you implement.

Use a single text file/document to contain all written questions (exercise 1). Microsoft Word documents (doc, docx) should not be handed in, because of incompatibilities between versions. Use text-only format (.txt) or PDF format (.pdf). If you are not sure about this, ask your TA. If the marker cannot open your file (for whatever reason), you will receive a grade of 0 for the written part.

Name your submission file a3-written.txt. Ensure that each answer is clearly marked for the question it relates to; and ensure that your file/document is clearly marked with your name, student number and NSID at the top.

For written questions and algorithms, you can use <- instead of ←.

Exercise 1 Revising the Array based List ADT (10 pts)

When we first examined the array based and node based implementations of the List ADT, there was one difference in the interface: The array based version had a capacity parameter in the CreateList function. This parameter was used to determine the fixed maximum capacity of the List. In this exercise, you will revise the pseudocode for the Array based ADT to remove the fixed capacity limit.

- Only one of the List operations should have a change to its header: Creating an array-based List should now take zero parameters, and should initially create an array based List with a capacity of 1 element. All other operations should have unchanged headers.
- In each of the insert operations, instead of returning false when the array is full, the algorithm should attempt to "grow" the capacity of the list, and then insert. To reduce the workload of this assignment, you only need to provide revised pseudocode for insertTail, not the other insert operations. The other operations would be modified in the same way.
- a new growList operation will be needed to increase the capacity of the list. This operation will be used by insertTail. If we were completely revising the array based List ADT, we would also use this grow operation in insertHead, and insertAfter. The grow operation should:
    1. allocate a new larger array
    2. copy the contents of the old array into the new array
    3. deallocate the old array
    4. update the reference to the array in the List to point to the new array.

The growing operation will be slow (O(n)), so our design should not call the grow operation very often. Thus, every time a list grows, it should double in capacity.

Use the following as your algorithm header for growList:

```
Algorithm growList(rList)
Attempt to double the capacity of rList
Pre: rList :: reference to a list to grow
Post: capacity of rList has been doubled, list contents are unchanged
Return: true if the grow operations succeeds, false otherwise
```

The following pseudocode for createList and insertTail is provided in"ArrayList.txt":

```
Algorithm createList(size)
Create a new list.
Pre: size :: the capacity of the array
Returns: a reference to a newly allocated list that is initialized to be
empty.

      refToList rNewList ← allocate newList
      rList → capacity ← size
      rList → tail ← -1
      rList → numElements ← 0
      rList → elements ← allocate new Element [size]
      return rNewList


Algorithm InsertTail(rList, el)
Pre: rList :: a reference to a list into which to insert
      el :: an Element
Post: el is inserted into the list
Return: true if successful, false otherwise

      if(rList → numElements == rList → capacity)
            return false //Special case when list is full
      else
            //put the new element in the position indexed by numElements
            rList → elements[numElements] ← el
            rList → numElements ← rList → numElements + 1
            rList → tail ← rList → tail + 1
      end if
      return true
```

What to hand in:

In your file a3-written.txt, indicate the question (exercise) number very clearly, so it can be seen easily by a marker. Your revised listing of the array based list should be in this file.

Grading

- 2 marks for properly modifying CreateList
- 2 marks for properly modifying InsertTail
- 6 marks for properly implementing the GrowList operation

Exercise 2 Recursion! (12 pts)

a6q3.cc contains three recursive functions (`collatz`, `removeOdds`, and `findAndReplace`). There is main function with test cases provided for each recursive function. But, each recursive function is missing the implementation.

Complete the implementation of each function, and ensure that all the given test cases pass. You may add additional tests if you like, but it is not required.

The intended behaviour of each function is as follows:

1. The Collatz conjecture states that for all integers >=1, applying the following function repeatedly will eventually result in a value of 1.

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

With the function `collatz`, we want to count how many times this function has to be repeatedly applied to reach the value 1. Thus, the function should return the value in the "Sequence length" column of the table below:

Starting value	Sequence	Sequence length
1	1	0
2	2 1	1
5	5 16 8 4 2 1	5

Write the recursive function to compute the sequence length, starting from an integer n. You may assume that only positive integers will be given as input. Hint: In the lectures, we have discussed recursion in terms of solving subproblems. You may be wondering how to formulate a subproblem for the Collatz function. Typically, we wanted to apply a function recursively to a smaller input such as (n□1). But, the Collatz sequence starting from the number 5 does not build naturally on the sequence starting from 4. Thus, (n□1) is not an appropriate "smaller" parameter for the recursive function call. Instead, because 5 is odd, the Collatz sequence starting from 5 builds on the sequence starting from (3 _ 5 + 1), or 16. It may not seem intuitive but 16 is "closer" to 1 (the end of the sequence) than 5 is. In general, the idea that for all odd n, (3n+1) is closer to 1 in the sequence is a bit of a leap of faith. But, there are exactly zero known counter examples to the Collatz conjecture and the problem has been widely studied.

2. Write a recursive function (`removeOdds`) to remove all nodes containing odd integers from a sequence of nodes. The given sequence should be modified, rather than producing a new sequence containing only the even integers. There are some helper functions given to create, display, and destroy sequences of nodes. You do not need to modify any of these functions. But, you may use them to help create additional tests.

3. Write a recursive function (`findAndReplace`) to find and replace characters in a character array. Recall that each recursive function call should be working on a sub-problem of the overall task. Thus, we need a way of accessing a sub-section of an array. There are multiple ways of doing this. Here's the recommended approach:

- Previously when working with an array, we used the index 0 (array[0]) to access the first element of the array. To work with a sub-array, we will need a "starting position" parameter to use in place of 0 to indicate where the sub-array starts. In the provided function header, this is the startPos parameter.
- As always, we need a way of knowing where the array ends. When we worked with arrays of integers, we used a size/length parameter to denote the number of elements in the array. When working with cstrings (as we are here) we can use the strlen function to determine the number of characters in the cstring.
- Following this approach, your recursive function call should have the following parameters: The character array, the find and replace values, and the starting index of the subproblem.

What to hand in:

The file a3q2.cc, containing your completed implementations. A file a3q2testing.txt containing the output of your testing.

Grading

12 marks: Correctly implementing each function is worth 4 marks.

Exercise 3 Using the List ADT and List Traversal ADT (11 pts)

In this exercise you will gain practice using the List and List Traversal ADTs as well as recursion. Write an application that will perform the following steps:

1. Read the contents of a6q4_input.txt into a List. The file contains one word per line. You should insert each word from the file into the list. Use file I/O (see lab 5), rather than redirecting standard input to read from a file.
2. Print the total number of words read from the file.
3. Use the List Traversal ADT to traverse the list of words and print every word that is a palindrome. A palindrome is a word that is unchanged when the order of the characters is reversed. Some example palindromes are "racecar" and "civic".

   Note: there are alternate solutions to the above tasks that do not use the List ADT, the List Traversal ADT, or recursion. The intent of this exercise is to gain experience with these ADTs and recursion. Your solution must use the List ADT, the List Traversal ADT, and recursion for full marks.

Here are some tips to help you:

- Use the given ADT implementations in NodeList.h, NodeList.cc, ListTraversal.h, and ListTraversal.cc. You do not need to make any changes to these files. If you need to change the datatype being stored in a node, do so in Element.h.
- Write a function bool isPalindrome(char *s, int start, int end) to check if a given word is or is not a palindrome. Write a recursive implementation of this function.

  Note: there are non-recursive ways of writing the function isPalindrome. The intent of this exercise is to gain practice writing recursive functions. your isPalindrome function must be recursive for full marks.

What to hand in:

The file a3q3.cc, containing your application. A file a3q3testing.txt containing the output of your program. You don't have to hand in the List ADT files or the List Traversal ADT files.

Grading

- 2 marks: your program properly uses file I/0 (rather than redirecting standard input) to read the contents of the file.
- 2 marks: the List ADT is used to store the words as they are read in from the file. 0 marks if the List ADT is not used.
- 1 mark: your program properly outputs the total number of words from the file.
- 4 marks: your solution contains a boolean function isPalindrome for checking if a given word is a palindrome.
  - 4 marks for a correct recursive solution. 0 marks for a non-recursive solution.
- 2 marks: your program uses the List Traversal ADT to traverse the list of words, and print the palindromes.
  - 0 marks if the List Traversal ADT is not used.