CMPT 115 Assignment 4

Summer 2016

Written Question

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. The answer to exercise one part two subpart c is one zero five zero and the top number on the stack is ten. 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 a4-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 Trees and stacks and queues and rocks and trees . . . (25 pts)

In lectures, we've seen stacks and queues and lists and trees. These are important data organizational tools used in a wide variety of ways in advanced computer science. We've only seen hints of the importance of these tools in the lecture. We did, however, talk about arithmetic expressions. A computer program can read arithmetic expressions, and understand their meaning, and evaluate them (for example, the C++ compiler converts arithmetic expressions to machine code when you compile your own programs!).

In this question, we will look at two representative algorithms. These are still exercises for learning about data organization, but they are relevant and reflective of real processes that go on in a compiler like C++. In other words, this exercise is not exactly what C++ does, but it's strongly related.

In your mathematics classes, you were taught how to write expressions using precedence rules (order of operations). For example, you know that 3 + 4 x 5 = 23, because multiplication is done before addition, unless an expression has brackets, such as (3 + 4) x 5 = 35. The rules that are used to evaluate arithmetic expressions are only conventions (i.e., mathematicians had to agree on them, but they do not come from nature). But these conventions impose mental burden of memorization on everyone (which you realized when you first were told to learn them).

There are two ways to represent arithmetic expressions that do not require memorization of order of operations. We have seen both in lecture. The first is post-fix notation, and the second is expressions trees. In this question we will convert from one to the other.

As a reminder: a post-fix expression puts the operator after the operands. For example, when you'd write "3 + 4" in normal arithmetic, the post-fix expression is "3 4 +"; similarly, the expression "3 + 4

x 5" would be written in post-fix: "3 4 5 x +". The key to post-fix notation is the idea of a stack (LIFO). To evaluate a post-fix expression, we use the following algorithm:

```
Algorithm EvaluatePostFix(expression, result)
Evaluates a postfix expression
Pre: expression :: C-string
     result :: refToNumber a valid reference to a number
Post: the value of result is changed if the expression can be evaluated
Return: true if the expression is valid; false otherwise

  S = createStack()
  Q = createQueue()

  break the expression into tokens (numbers, operator symbols)
  enqueue all the tokens into Q in the order they appear in expression

  if empty(Q) then
      return false
  end

  while size(Q) > 0 do
    dequeue(Q,&T)
    if T is a number then
       push(S,T)
    else if T is an operator then
       Let Op be the new name for T
       Let k be the number of operands needed by Op
       // e.g., k=2 for addition and multiplication
       if size(S) >= k then
           pop k numbers from S
           let V be the result of applying Op to the k numbers just popped
           // e.g., if Op is +, then add the two numbers
           push(S, V)
       else
           destroyQueue(Q)
           destroyStack(S)
           return false
       end if
    end if
 done
 pop(S,&result)
 return true
```

If our expression is "30 40 50 x +", then the tokens are "30", "40", "50", "+" and "x". For simplicity, we'll only worry about operators that take k=2 operands, like multiplication and addition.

1. (10 marks) Use the above algorithm to evaluate the following post-fix expressions. To avoid any ambiguities, we'll work only with multiplication and addition, but you should be confident that

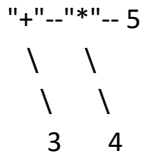
the algorithm works for any set of operators, including things like division, sin(), cos(), sqrt(), subtraction, etc.

(a) "3 4 5 x +"
(b) "3 4 + 5 x"
(c) "3 12 + 8 2 + x 7 x"
(d) "2 2 2 2 2 2 + x x + +"
(e) "2 2 2 + 2 x 2 + 2 x +"

The results of the expressions are not important. Understanding the process is important. So, to show that you understand the process, you are to provide the answer returned by the algorithm, and the contents of the stack S just before the coloured operator in each expression (write your stack so that the bottom of the stack is on the left, the top is on the right). For example, we'll give you the first answer in the format we want:

1. Stack: 3 20; Final result: 23

2. (10 marks) Here's an interesting fact: A post-fix expression is a post-order traversal of an expression tree. Using this fact, and the knowledge of trees and expression trees from lecture, draw the 5 expression trees for the post-fix expressions above. To answer this question, you have to ask yourself "What does a tree look like for a given post-order traversal?" A post-order traversal of a tree is unique, i.e., there is only one post-order traversal for any given tree! You can write trees in text, or you can upload a PDF document with your trees (if you use a separate document, please name your document in a helpful way. If the marker cannot find your work, you will get zero.) Here's a tree diagram that you might use in text. The tree is sideways, with the root on the left, and the branches going to the right.



3. (5 marks) Using your knowledge of trees, stacks, post-fix expressions and expression trees, write an algorithm that takes an post-order expression, and creates an expression tree from it. Use the Tree ADT (e.g., insertLeft(), insertRight(), etc), to build the tree. Remember that empty trees are NULL. Also, assume that your tree nodes can store numbers or operators (say, as C-strings). Also, assume that you can look at a token and decide if it is a number or a token (as the above algorithm did). Also, you can break the expression into tokens as we did in the algorithm above.

What to hand in:

In your file a4-written.txt, indicate the question (exercise) number very clearly, so it can be seen easily by a marker. You may use a second document for your trees. Please name the document clearly, and maybe even mention in a4-written.txt where to find your trees.

Grading

1. 10 marks. For each example: 1 mark for the contents of the stack just prior to the indicated operation, and one for the final result.
2. 10 marks. For the five examples: 2 marks for each tree. A post-order traversal of the tree gives the post-fix expression.
3. 5 marks. Your algorithm shows how to build an expression tree from a postfix expression, using stacks queues and trees, at a level of abstraction similar to the example above.

Programming Question

All C++ source code files must contain your name, student number, and NSID; and the course, assignment number, and question number as comments at the top of the file. All C++ programs must compile under g++ without errors or warnings, using the standard flags described in tutorial, namely -Wall -pedantic. Use of String class or other object-oriented code (except for cin or cout and related file-stream objects, which are allowed) will result in a flat deduction of 25%. (That means: don't use advanced techniques to avoid learning the concepts we are attempting to teach you!)

Exercise 3 Implement the revised Array based List ADT (19 pts)

In Assignment 3 Exercise 1, you were asked to modify the design of the Array based List. In this exercise you will implement and test the revised Array based List.

1. Start from the provided files ArrayList.h, ArrayList.cc, Element.h, and a7q3.cc. These files contain a partial implementation of the Array based List ADT and some tests. In particular, there are only 6 operations provided: createList, destroyList, emptyList, lengthList, insertTail, and deleteTail. This was specifically done to reduce the amount of work required for this question. You do not need to implement the other List operations we have studied in the lectures.
2. Following your pseudocode design from Assignment 3 Exercise 1, modify the Array based List to remove the fixed capacity limit by modifying createList, and insertTail and adding a growList operation.
3. The file a7q3.cc contains some tests for the given List implementation. Add additional tests to ensure that the new functionality you have added works correctly.
    - When modifying existing code it is good practice to ensure that the code you are working from actually works. You can run a7q3.cc to verify that the existing list operations behave as expected. Read over the test cases given in a7q3.cc so that you know what to expect from the output. Then, check the output before modifying the List code and verify that the List behaves as expected.
    - Write additional tests for your modified implementation.
    - Changes to one part of an implementation can have unexpected changes elsewhere. Thus, after you modify the ArrayList code, you should verify that the old tests still pass.

- Note that some of the old tests will need to be modified to work with the new List implementation. In particular, the old tests specify the size of the list to create, but the parameter has been removed from createList. That's OK. When the existing tests don't compile (or do compile, but don't run as expected) always ask yourself if the test needs to be changed to suit the new implementation, or if you have introduced a bug into the implementation. The tests will only need to be modified if you have introduced a change to the interface, or if your tests violate the encapsulation of the ADT.

What to hand in:

The files ArrayList.h and ArrayList.cc, containing your revised ADT. A file a4q3.cc containing your code for testing the new ADT. A file a4q3testing.txt containing the output from your testing.

Grading

- 1 mark for properly modifying the function prototypes in ArrayList.h
- 2 marks for modifying createList in ArrayList.cc
- 4 marks for modifying insertTail in ArrayList.cc
- 6 marks for implementing growList in ArrayList.cc
- 6 marks for adding new tests for the growable list in a4q3.cc. These tests should either directly or indirectly call the grow operation, and verify that it works as expected.