Project 2 - Recursive List

CS 3358: Data Structures and Algorithms
Due 23/02/2023 11:59 pm CST

Project Instructions

Write code for each of the below questions. Starter code for the project is available in the .zip folder in Canvas. Do not change the function signatures in the starter code. You may edit your code in whatever IDEs or text editors you prefer.

You must submit your final project code via Canvas for grading.

Responses will be graded on three axes: correctness, pseudocode and code logic, and code style. A majority of your grade is determined by how many test cases your code passes. The points awarded for each question break down as follows:

- Test Cases: 60%
    - We Will run your code against predefined test cases to check for correctness.
    - Any solutions that try to game the test cases will receive zero credit. For example, writing a linear search algorithm when we ask for binary search will result in zero credit for this portion of the grade, even if it passes the test cases.
- Pseudocode and Code Logic: 20%
    - For each question, you will be asked to write your pseudocode for your code in the comments. Writing pseudocode benefits you on two fronts: (1) it helps you plan your algorithm before you write your code, and (2) it allows us to give you credit in this category for your logic, even if the final code doesn't pass the "Test Cases". You may also write comments to explain your logic to the grader for any questions on which you're having trouble.
    - Again, any solutions that try to game the test cases or solutions that don't meet the requirements of the question will not receive credit.
- Code Style: 20%
    - We recommend referencing the Google C++ Style Guide, but you do not need to follow it exactly as long as your code style is consistent and readable.

We recommend getting started on the project early so you have time to ask questions in office hours in case you get stuck. You may submit your code as many times as you like, up until the deadline. We will only grade your most recent submission.

Please remember to follow our guidelines for academic integrity. Do not post these project questions publicly, as that is a copyright violation and a breach of the honor code.

Project Overview

This project will give you experience writing recursive functions that operate on recursively-defined data structures and mathematical abstraction. In particular, we will use a recursively-defined list in all of our functions.

Recursive List Definition

A list is a sequence of zero or more numbers. A list is well formed if:

(a) It is the empty list, or
(b) It is an integer followed by a well-formed list

A list is an example of a linear recursive structure: it is recursive because the definition refers to itself. It is linear because there is only one such reference. Here are some examples of well-formed lists:

```
( 1 2 3 4 ) // a list of four elements
( 1 2 4 ) // a list of three elements
( ) // a list of zero elements, a.k.a. the empty list
```

Overview of Starter Code

The starter code for this project contains five files:

- **`recursive_list.h`**
  A simple interface for the recursively defined list data structure described above in the `RecursiveList` Interface section. DO NOT MODIFY THIS FILE.
- **`recursive_list.cpp`**
  Implementation of a simple interface for the recursively defined list data structure described above in the `RecursiveList` Interface section. DO NOT MODIFY THIS FILE.
- **`project2.h`**
  An interface for the functions that you will implement in `project2.cpp`. DO NOT MODIFY THIS FILE.
- **`project2.cpp`**
  This is the file where you will add your function implementations. The function signatures are already provided; DO NOT MODIFY the function signatures. You can think of this file as providing a library of functions that other programs might use, just as `recursive_list.cpp` does. DO NOT INCLUDE a main function in your project2.cpp file.
- **`main.cpp`**
  We have provided a main function in this file which uses your code as a library to test your functions. You are encouraged to modify it to test more cases.

RecursiveList Interface

The file recursive_list.h defines the type RecursiveList. We use a typedef to define RecursiveList, but don't worry if you don't understand what this syntax means. For the purposes of this project, just assume that RecursiveList is the name of a type that you can use just like int or double.

The file recursive_list.h also defines the following operations on recursive lists:

```
// EFFECTS: returns true if list is empty, false otherwise
extern bool ListIsEmpty(RecursiveList list);

// EFFECTS: returns an empty list
RecursiveList MakeList();

// EFFECTS: given the list, make a new list consisting of the new element
//          followed by the elements of the original list
RecursiveList MakeList(int elem, RecursiveList list);

// REQUIRES: list is not empty
// EFFECTS: returns the first element of list
extern int ListFirst(RecursiveList list);

// REQUIRES: list is not empty
// EFFECTS: returns the list containing all but the first element of list
extern RecursiveList ListRest(RecursiveList list);

// MODIFIES: std::cout
// EFFECTS: prints list to std::cout
extern void PrintList(RecursiveList list);
```

Note that the EFFECTS clauses of ListFirst and ListRest are only valid for non-empty lists. To help you in debugging your code, these functions check to see if their lists are empty and will print a warning and exit the program if that is the case. When writing your code, make sure that you first check if a list is empty before calling ListFirst or ListRest.

Note also that ListMake is an overloaded function. If called with no arguments, it produces an empty list. If called with an element and a list, it combines the element and list into a new list.

Your Task: Implementing List Processing Functions

Given the `RecursiveList` interface, you will write the ten list processing functions described in Questions 1-8, each adhering to the following constraints and guidelines:

- Each of your functions MUST be recursive and cannot be iterative.
- You are NOT allowed to use goto, for, while, or do-while. You may not use global variables, static variables, or reference arguments.
- You may use only the C++ standard and `iostream` libraries, and no others.
- You can use any of the functions implemented by the `RecursiveList` Interface. For example, you could call `ListFirst()` and `ListFirst()` in your implementation of the `Sum()` function.
- You may call any of these functions in the implementation of another. For example, you could call `Reverse()` from the `Append()` function, or vice versa.
- You are welcome to define helper functions for any of the recursive functions you need to implement, but you do not have to. See the Appendix for more information about helper functions.
- Only `TailRecursiveSumHelper()` in Question 2 is required to be tail recursive. Your other implementations do not need to be tail recursive. See the Appendix for more information about tail recursive functions.

There are a lot of functions to implement, but many of them are similar to one another, and each can be implemented in under 10-12 lines of code.

Tips and Tricks for Success

1. These project instructions are very detailed, and that is intentional. I recommend reading over the instructions a couple times through before getting started on the project. Ask any clarifying questions you may have about how the existing interfaces work or what is required of you.
2. Remember that these coding projects are expected to take 5-10 hours to complete, which is why you are given 2 weeks to work on the project. Try to do a little bit at a time rather than completing the full project in one go. GIve yourself time to let things marinate in your mind. You might be surprised to see how much processing your brain is doing in the background.
3. It really helps to draw out examples on a whiteboard or on paper. You can start with the example inputs and outputs given in the instructions. Try to follow the 3 b's (base case, break down, build on) that we learned in class. Use concrete examples to guide your thought process.
4. The ten functions you need to implement can be attempted in any order. I have ordered them roughly in order of difficulty according to my experience, but you are more than welcome to skip around.
5. Leave yourself enough time to come to office hours if you get stuck on something.

Question 1 - Sum and Product (20 points)

Implement the `Sum()` and `Product()` functions in `project2.cpp`. Your implementation of `Sum()` should NOT be tail recursive and should NOT use a helper function.

See the Appendix for more information on tail recursion and helper functions.

Function Spec:

```
// EFFECTS: returns the sum of each element in list, or zero if the list is
//          empty
int Sum(RecursiveList list) {
  // Implement this function.
  return 0;
}

// EFFECTS: returns the product of each element in list, or one if the list is
//          empty
int Product(RecursiveList list) {
  // Implement this function.
  return 0;
}
```

Example Input and Output:
Calling `Sum()` on list `( 1 2 3 4 5 )` should return `15`. Calling `Product()` on list `( 1 2 3 4 5 )` should return `120`.

Question 2 - Tail Recursive Sum (10 points)

Implement the `TailRecursiveSumHelper()` function in `project2.cpp`. Your implementation of `TailRecursiveSumHelper()` MUST be tail recursive. `TailRecursiveSumHelper()` is called by the provided function `TailRecursiveSum()`. You should not modify `TailRecursiveSum()`.

See the Appendix for more information on tail recursion and helper functions.

Function Spec:

```
// EFFECTS: adds the next element in the list to the sum so far
int TailRecursiveSumHelper(RecursiveList list, int sum_so_far) {
  // Implement this function.
  return 0;
}

// EFFECTS: returns the sum of each element in list, or zero if the list is
//          empty
// THIS FUNCTION IS PROVIDED AS PART OF THE STARTER CODE.
// DO NOT MODIFY THIS FUNCTION.
int TailRecursiveSum(RecursiveList list) {
  return TailRecursiveSumHelper(list, 0);
}
```

Example Input and Output:
Calling `TailRecursiveSum()` on list `( 1 2 3 4 5 )` should return `15`. Note that `TailRecursiveSum()` has the same expected output as `Sum()`.

Question 3 - Filter Odd and Filter Even (20 points)

Implement the `FilterOdd()` and `FilterEven()` functions in `project2.cpp`.

Function Spec:

```
// EFFECTS: returns a new list containing only the elements of the original list
//          which are odd in value, in the order in which they appeared in list
// For example, FilterOdd(( 4 1 3 0 )) would return the list ( 1 3 )
RecursiveList FilterOdd(RecursiveList list) {
  // Implement this function.
  return list;
}

// EFFECTS: returns a new list containing only the elements of the original list
//          which are even in value, in the order in which they appeared in list
// For example, FilterEven(( 4 1 3 0 )) would return the list ( 4 0 )
RecursiveList FilterEven(RecursiveList list) {
  // Implement this function.
  return list;
}
```

Example Input and Output:
Calling `FilterOdd()` on list `( 1 2 3 4 5 )` should return `( 1 3 5 )`. Calling `FilterEven()` on list `( 1 2 3 4 5 )` should return `( 2 4 )`.

Question 4 - Reverse (10 points)

Implement the `Reverse()` function in `project2.cpp`.

Function Spec:

```
// EFFECTS: returns the reverse of list
// For example, the reverse of ( 3 2 1 ) is ( 1 2 3 )
RecursiveList Reverse(RecursiveList list) {
  // Implement this function.
  return list;
}
```

Example Input and Output:
Calling `Reverse()` on list `( 1 2 3 4 5 )` should return `( 5 4 3 2 1 )`.

Question 5 - Append (10 points)

Implement the `Append()` function in `project2.cpp`.

Function Spec:

```
// EFFECTS: returns the list (first_list second_list)
// For example, if first_list is ( 1 2 3 ) and second_list is ( 4 5 6 ), then
// returns ( 1 2 3 4 5 6 )
RecursiveList Append(RecursiveList first_list, RecursiveList second_list) {
  // Implement this function.
  return first_list;
}
```

Example Input and Output:
Calling `Append()` on lists `( 1 2 3 4 5 )` and `( 11 12 13 14 15 )` should return `( 1 2 3 4 5 11 12 13 14 15 )`.

Question 6 - Chop (10 points)

Implement the `Chop()` function in `project2.cpp`.

Function Spec:

```
// REQUIRES: list has at least n elements
// EFFECTS: returns the list equal to list without its last n elements
RecursiveList Chop(RecursiveList list, unsigned int n) {
  // Implement this function.
  return list;
}
```

Example Input and Output:
Calling `Chop()` on list `( 1 2 3 4 5 )` with `n = 0` should return `( 1 2 3 4 5 )`. Calling `Chop()` on list `( 1 2 3 4 5 )` with `n = 2` should return `( 1 2 3 )`.

Question 7 - Rotate (10 points)

Implement the `Rotate()` function in `project2.cpp`.

Function Spec:

```
// EFFECTS: returns a list equal to the original list with the
//          first element moved to the end of the list n times.
// For example, Rotate(( 1 2 3 4 5 ), 2) yields ( 3 4 5 1 2 )
RecursiveList Rotate(RecursiveList list, unsigned int n) {
  // Implement this function.
  return list;
}
```

Example Input and Output:
Calling `Rotate()` on list `( 1 2 3 4 5 )` with `n = 0` should return `( 1 2 3 4 5 )`.
Calling `Rotate()` on list `( 1 2 3 4 5 )` with `n = 2` should return `( 3 4 5 1 2 )`.

Question 8 - Insert List (10 points)

Implement the `InsertList()` function in `project2.cpp`.

Function Spec:

```
// REQUIRES: n <= the number of elements in first_list
// EFFECTS: returns a list comprising the first n elements of first_list,
//          followed by all elements of second_list, followed by any remaining
//          elements of "first_list"
// For example, InsertList (( 1 2 3 ), ( 4 5 6 ), 2) returns ( 1 2 4 5 6 3 )
RecursiveList InsertList(RecursiveList first_list, RecursiveList second_list,
                         unsigned int n) {
  // Implement this function.
  return first_list;
}
```

Example Input and Output:

Calling `InsertList()` on first list `( 1 2 3 4 5 )`, second list `( 11 12 13 14 15 )`, and `n = 0` should return `( 11 12 13 14 15 1 2 3 4 5 )`. Calling `InsertList()` on first list `( 1 2 3 4 5 )`, second list `( 11 12 13 14 15 )`, and `n = 2` should return `( 1 2 11 12 13 14 15 3 4 5 )`.

Appendix

Tail Recursion

Remember: a tail recursive function is one in which the recursive call happens absent any pending computation in the caller. For example, the following is a tail recursive implementation of factorial:

```
// REQUIRES: n >= 0
// EFFECTS: computes prod * n!
int FactorialHelper(int n, int prod) {
  if (n == 0) {
    return prod;
  } else {
    return FactorialHelper(n-1, prod * n);
  }
}

// REQUIRES: n >= 0
// EFFECTS: computes n!
int Factorial(int n) {
  return FactorialHelper(n, 1);
}
```

Notice that the return value from the recursive call to `FactorialHelper` is only returned again — it is not used in any local computation, nor are there any steps left after the recursive call.

As another example, the following implementation of `Factorial` is NOT tail recursive:

```
// REQUIRES: n >= 0
// EFFECTS: computes n!
int Factorial(int n) {
  if (n == 0) {
    return 1;
  }
  return n * Factorial(n - 1);
}
```

Notice that the return value of the recursive call to `Factorial` is used in a computation in the caller; namely, it is multiplied by `n`.

Helper Functions

Helper functions are useful when you want to add additional parameters to your recursive function. The main recursive function calls the recursive helper function, which then calls itself.

The tail recursive version of `Factorial` requires a helper function (`FactorialHelper`), which is common when writing tail recursive functions. You can choose to utilize helper functions, even if you aren't using tail recursion.