Project Two: Recursion

Due: Mar. 19, 2024

Motivation

This project will give you experience in writing recursive functions that operate on recursively-defined data structures and mathematical abstractions.

Lists

A "list" is a sequence of zero or more numbers in no particular order. 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 element--the empty list
```

The file `recursive.h` in the `Project-2-Related-Files.zip` defines the type `list_t` and the following operations on lists:

```
bool list_isEmpty(list_t list);
// EFFECTS: returns true if "list" is empty, false otherwise

list_t list_make();
// EFFECTS: returns an empty list

list_t list_make(int elt, list_t list);
// EFFECTS: given "list", make a new list consisting of
// the new element followed by the elements of the
// original list

int list_first(list_t list);
// REQUIRES: "list" is not empty
```

```
// EFFECTS: returns the first element of list

list_t list_rest(list_t list);
// REQUIRES: "list" is not empty
// EFFECTS: returns the list containing all but the first
// element of list

void list_print(list_t list);
// MODIFIES: cout
// EFFECTS: prints "list" to cout
```

Note: `list_first` and `list_rest` are both partial functions; their EFFECTS clauses are only valid for nonempty lists. To help you write your code, these functions actually check whether the list is empty or not--if they are passed an empty list, they fail gracefully by warning you and exiting; if you are running your program under the debugger, it will stop at the exit point. Note that such checking is not required! It would be perfectly acceptable to write these in such a way that they fail quite ungracefully if passed empty lists. Note also that `list_make` is an overloaded function. If called with no arguments, it produces an empty list. If called with an element and a list, it combines them.

Given this `list_t` interface, you will write the following list processing procedures. Each of these procedures **must be recursive**. **For full credit, your routines must provide the correct result and provide an implementation that is recursive.** In writing these functions, you may use only recursion and selection. You are **NOT** allowed to use goto, for, while, do-while, global variables, pointers (except function pointers), and references (including constant references).

Hint: in implementing some functions recursively, you may need to define some recursive helper functions. If you define **any** functions yourself (such as the recursive helper functions), be sure to declare them "**static**", so that they are **not visible** outside this file. This will prevent any name conflicts in case you give a function the same name as one in the test cases. (For further information on "static" functions, please read some online tutorials/references. In the past, some students got a zero score simply because they forgot to declare their support functions as **static** functions. Be aware of this!)

Below is an example where we implement the factorial function with a recursive helper function. Note that the function `factorial_helper` is defined as a static function.

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

int factorial(int n)
// REQUIRES: n >= 0
// EFFECTS: computes n!
{
     factorial_helper(n, 1);
}
```

Below are the functions you are to implement. There are a number of them, but many of them are similar to one another, and the longest is at most tens of lines of code, including support functions.

```
int size(list_t list);
// EFFECTS: Returns the number of elements in "list".
//          Returns zero if "list" is empty.

int sum(list_t list);
// EFFECTS: Returns the sum of all elements in "list".
//          Returns zero if "list" is empty.

int product(list_t list);
// EFFECTS: Returns the product of all elements in "list".
//          Returns one if "list" is empty.

list_t reverse(list_t list);
// EFFECTS: Returns the reverse of "list".
//
// For example: the reverse of ( 3 2 1 ) is ( 1 2 3 )
```

```
list_t append(list_t first, list_t second);
// EFFECTS: Returns the list (first second).
//
// For example: append(( 2 4 6 ), ( 1 3 )) gives
// the list ( 2 4 6 1 3 ).

list_t filter(list_t list, bool (*fn)(int));
// EFFECTS: Returns a list containing precisely the elements of "list"
//          for which the predicate fn() evaluates to true, in the
//          order in which they appeared in list.
//
// For example, if predicate bool odd(int a) returns true if a is odd,
// then the function filter(list, odd) has the same behavior as the
// function filter_odd(list).

list_t insert_list(list_t first, list_t second, unsigned int n);
// REQUIRES: n <= the number of elements in "first".
//
// EFFECTS: Returns a list comprising the first n elements of
//          "first", followed by all elements of "second",
//           followed by any remaining elements of "first".
//
//     For example: insert (( 1 2 3 ), ( 4 5 6 ), 2)
//            gives ( 1 2 4 5 6 3 ).

list_t chop(list_t list, unsigned int n);
// REQUIRES: "list" has at least n elements.
//
// EFFECTS: Returns the list equal to "list" without its last n
//          elements.

bool issorted_list(list_t list);
// EFFECTS: Returns true if the "list" is ascending sorted or empty,
//          false otherwise.
```

Binary Trees

A binary tree is another fundamental data structure we will use in this project. A binary tree is well-formed if:

a) It is the empty tree, or
b) It consists of an integer element (called the root element), plus two children, called the left subtree and the right subtree, each of which is a well-formed binary tree.

Additionally, we say a binary tree is a "leaf" if and only if both of its children are the EMPTY TREE.

The file `recursive.h` in `Project-2-Related-Files.zip` defines the type `tree_t` and the following operations on trees:

```
bool tree_isEmpty(tree_t tree);
// EFFECTS: returns true if "tree" is empty, false otherwise

tree_t tree_make();
// EFFECTS: creates an empty tree

tree_t tree_make(int elt, tree_t left, tree_t right);
// EFFECTS: creates a new tree, with "elt" as its root element,
// "left" as its left subtree, and "right" as its right subtree

int tree_elt(tree_t tree);
// REQUIRES: "tree" is not empty
// EFFECTS: returns the element at the top of "tree"

tree_t tree_left(tree_t tree);
// REQUIRES: "tree" is not empty
// EFFECTS: returns the left subtree of "tree"

tree_t tree_right(tree_t tree);
// REQUIRES: "tree" is not empty
// EFFECTS: returns the right subtree of "tree"

void tree_print(tree_t tree);
// MODIFIES: cout
// EFFECTS: prints "tree" to cout.
// Note: this uses a non-intuitive, but easy-to-print format
```

There are several functions you are to write for binary trees. These must be recursive, and cannot use any looping structures. Once again, if you need to define any support functions, be sure to define them as static functions.

```
int tree_sum(tree_t tree);
// EFFECTS: Returns the sum of all elements in "tree".
//          Returns zero if "tree" is empty.

bool tree_search(tree_t tree, int key);
// EFFECTS: Returns true if there exists any element in "tree"
//          whose value is "key". Otherwise, returns “false”.

int depth(tree_t tree);
// EFFECTS: Returns the depth of "tree", which equals the number of
//          layers of nodes in the tree.
//          Returns zero if "tree" is empty.
//
// For example, the tree
//
//                             4
//                           /   \
//                          /     \
//                         2       5
//                        / \     / \
//                          3       8
//                         / \     / \
//                        6   7
//                       / \ / \
//
// has depth 4.
// The element 4 is on the first layer.
// The elements 2 and 5 are on the second layer.
// The elements 3 and 8 are on the third layer.
// The elements 6 and 7 are on the fourth layer.

int tree_min(tree_t tree);
// REQUIRES: "tree" is non-empty.
// EFFECTS: Returns the smallest element in "tree".

list_t traversal(tree_t tree);
// EFFECTS: Returns the elements of "tree" in a list using an
```

```
//          in-order traversal. An in-order traversal prints
//          the "left most" element first, then the second-left-most,
//          and so on, until the right-most element is printed.
//
//          For any node, all the elements of its left subtree
//          are considered as on the left of that node and
//          all the elements of its right subtree are considered as
//          on the right of that node.
//
// For example, the tree:
//
//                            4
//                          /   \
//                         /     \
//                        2       5
//                       / \     / \
//                          3
//                         / \
//
// would return the list
//
//       ( 2 3 4 5 )
//
// An empty tree would print as:
//
//       ( )


bool tree_hasPathSum(tree_t tree, int sum);
// EFFECTS: Returns true if and only if "tree" has at least one
//          root-to-leaf path such that adding all elements along
//          the path equals "sum".
//
// A root-to-leaf path is a sequence of elements in a tree starting
// with the root element and proceeding downward to a leaf (an element
// with no children).
//
// An empty tree has no root-to-leaf path.
//
// For example, the tree:
//
```

```
//                           4
//                         /   \
//                        /     \
//                       1       5
//                      / \     / \
//                     3   6
//                    / \ / \
//
// has three root-to-leaf paths: 4->1->3, 4->1->6 and 4->5.
// Given sum = 9, the path 4->5 has the sum 9, so the function
// should return true. If sum = 10, since no path has the sum 10,
// the function should return false.
```

We can define a special relation between trees "is covered by" as follows:

- An empty tree is covered by all trees.
- The empty tree covers only other empty trees.
- For any two non-empty trees, A and B, A is covered by B if and only if the root elements of A and B are equal, the left subtree of A is covered by the left subtree of B, and the right subtree of A is covered by the right subtree of B.

For example, the tree:

```
        4
       / \
      /   \
     2     5
    / \   / \
       3
      / \
```

covers the tree:

```
         4
        / \
       /
      2
     / \
```

but not the trees:

```
          4                    5
         /                    / \
        /          or
       3
      / \
```

In light of this definition, write the following function:

```
bool covered_by(tree_t A, tree_t B);
// EFFECTS: Returns true if tree A is covered by tree B.
```

With the definition of “covered by”, we can define a relation “contained by”. A tree A is contained by a tree B if

- A is covered by B, or,
- A is covered by a subtree of B.

Note that in the above definition, a **subtree** of a tree T is an empty tree or a non-empty tree composed of a node S in T together with **all** downstream nodes of the node S in T (called the descendants of S in T).

For example, for the tree T

```
          4
        /   \
       /     \
      2       5
     / \     / \
        3       8
       / \     / \
      6   7
     / \ / \
```

the tree

```
        3
       / \
      6   7
     / \ / \
```

is a subtree of T. However, the tree

```
    3
   / \
      7
     / \
```

is not.

Based on the definition of "contained by", the trees

```
      4                     5
     / \                   / \
    /          and
   2
  / \
```

are contained by the tree

```
      4
     / \
    /   \
   2     5
  / \   / \
    3
   / \
```

but this tree is not:

```
      4
     /
    /
   3
  / \
```

Please write a function implementing the relation "contained by":

```
bool contained_by(tree_t A, tree_t B);
// EFFECTS: Returns true if tree A is covered by tree B
//          or a subtree of B.
```

There exists a special kind of binary tree, called the sorted binary tree. A sorted binary tree is well-formed if:

1. It is a well-formed binary tree **and**
2. One of the following is true:
   a) The tree is empty.
   b) The left subtree is a sorted binary tree, and any element in the left subtree is strictly less than the root element of the tree. The right subtree is a sorted binary tree, and any element in the right subtree is greater than or equal to the root element of the tree.

For example, the following trees are all well-formed sorted binary trees:

```
       4                 1
     /   \              / \
    /     \                1
   2       6              / \
  / \     / \                2
 1   3   5   7              / \
/ \ / \ / \ / \
```

while the following trees are not:

```
   1           1            4
  / \         / \          / \
 1           2   3        3   6
/ \                      / \   \
                        2   1   7
```

You are to write the following function for creating sorted binary trees:

```
tree_t insert_tree(int elt, tree_t tree);
// REQUIRES: "tree" is a sorted binary tree.
//
// EFFECTS: Returns a new tree with "elt" inserted as a leaf such that
//          the resulting tree is also a sorted binary tree.
//
// For example, inserting 1 into the tree:
```

```
//
//                           4
//                         /   \
//                        /     \
//                       2       5
//                      / \     / \
//                         3
//                        / \
//
// would yield
//                           4
//                         /   \
//                        /     \
//                       2       5
//                      / \     / \
//                     1   3
//                    / \ / \
//
// Hint: There is only one unique position for any element to be
// inserted.
```

Also, you can write the following function to check whether the tree is sorted binary tree.

```
bool issorted_tree(tree_t tree);
// EFFECTS: returns true if the tree is sorted or empty,
//          false otherwise.
```

<u>Files</u>

There are several files located in the `Project-2-Related-Files.zip` of our Canvas Resources:

`p2.h`	The header file for the functions you must write
`recursive.h`	The `list_t` and `tree_t` interfaces
`recursive.cpp`	The implementations of `list_t` and `tree_t`.

You should copy the above files into your working directory. **DO NOT modify these files!** You should put **all** of the functions you write in a single file, called **`p2.cpp` (exactly like this!)**. You may only use `<iostream>` and `<cstdlib>` libraries, and no others. You may **not** use global variables. You can think of `p2.cpp` as providing a library of functions that other programs might use, just as `recursive.cpp` does.

Testing

You can use the following two functions to check the equivalence of two lists and the equivalence of two trees, respectively.

```
bool list_equal(list_t l1, list_t l2)
    // EFFECTS: returns true iff l1 == l2.
{
    if(list_isEmpty(l1) && list_isEmpty(l2))
    {
        return true;
    }
    else if(list_isEmpty(l1) || list_isEmpty(l2))
    {
        return false;
    }
    else if(list_first(l1) != list_first(l2))
    {
        return false;
    }
    else
    {
        return list_equal(list_rest(l1), list_rest(l2));
    }
}

bool tree_equal(tree_t t1, tree_t t2)
    // EFFECTS: returns true iff t1 == t2
{
    if(tree_isEmpty(t1) && tree_isEmpty(t2))
    {
        return true;
    }
    else if(tree_isEmpty(t1) || tree_isEmpty(t2))
    {
```

```
        return false;
    }
    else
    {
        return ((tree_elt(t1) == tree_elt(t2))
            && tree_equal(tree_left(t1), tree_left(t2))
            && tree_equal(tree_right(t1), tree_right(t2)));
    }
}
```

To test your code, you should create a family of test case programs that exercise the functions we ask you to write. Here is a simple illustration to get you started:

```
#include <iostream>
#include "recursive.h"
#include "p2.h"
using namespace std;

static bool list_equal(list_t l1, list_t l2)
    // EFFECTS: returns true iff l1 == l2.
{
    if(list_isEmpty(l1) && list_isEmpty(l2))
    {
        return true;
    }
    else if(list_isEmpty(l1) || list_isEmpty(l2))
    {
        return false;
    }
    else if(list_first(l1) != list_first(l2))
    {
        return false;
    }
    else
    {
        return list_equal(list_rest(l1), list_rest(l2));
    }
}

int main()
```

```
{
    int i;
    list_t listA, listA_answer;
    list_t listB, listB_answer;

    listA = list_make();
    listB = list_make();
    listA_answer = list_make();
    listB_answer = list_make();

    for(i = 5; i>0; i--)
    {
        listA = list_make(i, listA);
        listA_answer = list_make(6-i, listA_answer);
        listB = list_make(i+10, listB);
        listB_answer = list_make(i+10, listB_answer);
    }

    for(i = 5; i>0; i--)
    {
        listB_answer = list_make(i, listB_answer);
    }

    listB = append(listA, listB);
    listA = reverse(listA);

    if(!list_equal(listA, listA_answer))
        return -1;

    if(!list_equal(listB, listB_answer))
        return -1;

    return 0;
}
```

Note that in the above test program, the return value will be -1 if there is any error in the function `reverse` and `append`. If the return value is 0, then you pass the above test case.

Suppose the above test code is written in the file `test.cpp`. Compile `test.cpp` with `recursive.cpp` and your `p2.cpp` using the following Linux command:

```
g++ -Wall -o test test.cpp recursive.cpp p2.cpp
```

To check the return value, you should first run the program in Linux by typing

```
./test
```

Then you can check the return value by typing

```
echo $?
```

If the return value is -1, it indicates an error.

You may also find it helpful to add error messages to your output or print out the list or tree using the functions `list_print` and `tree_print`. You can find two more test examples `simple_test.cpp` and `treeins_test.cpp` in the `Project-2-Related-Files.zip` on Canvas.

Submitting and Due Date

You only need to submit your source code file `p2.cpp` (**name it exactly like this!**). The source code file should be submitted via the online judgment system. The due date is 11:59 pm on Mar. 19th, 2024.

Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

An example of Functional Correctness is whether or not your reverse function reverses a list properly in all cases. An example of an Implementation Constraint is whether reverse() is recursive. General Style speaks to the cleanliness and readability of your code.