项目二：递归

截止日期：2024 年 10 月 30 日

动机

递归是应用过程抽象思想的一种很好的方式，因为递归函数会调用自身，这是一种抽象。本项目将为您提供编写递归函数的经验，这些函数作用于递归定义的数据结构和数学抽象。

列表

“列表”是零个或多个数字的序列，没有特定的顺序。列表格式正确的情况包括：a) 它是空列表，或者

b) 它是一个整数，后面跟着一个格式良好的列表。

列表是线性递归结构的一个示例：它“递归”是因为其定义涉及自身。它“线性”是因为存在这样一种引用且仅此一种。

以下是一些格式良好的列表示例:

```
( 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
```

在“Project-2-Related-Files.zip”中的“recursive.h”文件定义了列表类型“list_t”以及针对列表的以下操作:

```
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", makes 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
```

注意：list_first 和 list_rest 都是部分函数；它们的 EFFECTS 子句仅对非空列表有效。为了帮助您编写代码，这些函数实际上会检查列表是否为空——如果向它们传递一个空列表，它们会通过向您发出警告并退出而优雅地失败；如果您在调试器下运行程序，它会在退出点停止。请注意，这种检查并非必需！如果以这样一种方式编写这些函数，即如果传递空列表，它们会相当不优雅地失败，那也是完全可以接受的。还要注意，list_make 是一个重载函数。如果调用时未提供任何参数，它将生成一个空列表。如果调用时传递了一个元素和一个列表，它将把它们合并。

给定这个 list_t 接口，您将编写以下列表处理程序。这些程序中的每一个都**必须是递归的**。要获得**满分**，**您**的**例程必须提供正确**的**结果**，**并**提供**递归的实现**。在编写这些函数时，您只能使用递归和选择。**不**允许使用跳转、for、while、do-while、全局变量、指针（除了函数指针）和引用（包括常量引用）。

提示：在递归实现某些函数时，您可能需要定义一些辅助函数（例如递归辅助函数）。如果您自己定义**任何**函数，请确保将其声明为“静态”函数，这样它们就**不会**在这个文件之外**可见**。这将防止在您给一个函数与测试用例中的函数同名的情况下发生任何名称冲突。（关于“静态”函数的更多信息，请阅读一些在线教程/参考资料。过去，一些学生因为忘记将辅助函数声明为**静态**函数而得到零分。请注意这一点！）

以下是一个示例，其中我们使用递归辅助函数来实现阶乘函数。请注意，函数 factorial_helper 被定义为静态函数。

```
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);
}
```

以下是您要实现的功能。它们数量众多，但其中许多彼此相似，而且最长的功能（包括支持函数）最多也就几十行代码。

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

bool memberOf(list_t list, int val);
// EFFECTS: Returns true if the value "val" appears in "list".
//          Returns false otherwise.

int dot(list_t v1, list_t v2);
// REQUIRES: Both "v1" and "v2" are non-empty
//
// EFFECTS: Treats both lists as vectors. Returns the dot
//          product of the two vectors. If one list is longer
//          than the other, ignore the longer part of the vector.

bool isIncreasing(list_t v);
// EFFECTS: Checks if the list of integers is increasing.
//          A list is increasing if and only if no element
//          is smaller than its previous element. By default,
//          an empty list and a list of a single element are
//          increasing.
//
```

```
//          For example: (), (2), (1, 1), and (1, 2, 3, 3, 5) are
//          all increasing. (2, 1) and (1, 2, 3, 2, 5) are not.

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 ).

bool isArithmeticSequence(list_t v);
// EFFECTS: Checks if the list of integers forms an
//          arithmetic sequence. By default, an empty list and
//          a list of a single element are arithmetic sequences.
//
//          For example: (), (1), (1, 3, 5, 7), and (2, 8, 14, 20)
//          are arithmetic sequences. (1, 2, 4), (1, 3, 3),
//          and (2, 4, 8, 10) are not.

list_t filter_odd(list_t list);
// 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, if you apply filter_odd to the list
//          ( 3 4 1 5 6 ), you would get the list ( 3 1 5 ).

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 unique(list t list);
```

// 效果：返回一个新列表，其中包含“列表”中的每个唯一元素 // 。顺序由每个唯一元素在“列表”中的首次出现决定。

//

//　　例如，如果您将“unique”应用于列表
//　　（1 1 2 1 3 5 5 3 4 5 4），你会得到（1 2 3 5 4）。
//　　如果您将“unique”应用于列表（0 1 2 3），您将会
//　　获取（0 1 2 3）

list_t insert_list(list_t first, list_t second, unsigned int n)；
// 要求：n ≤ “first”中元素的数量 。
//
// 效果：返回一个列表，其中包含前 n 个元素
//　　“首先”，接着是“其次”的所有要素
//　　随后是“first”的任何剩余元素。
//
// 例如：insert((1 2 3), (4 5 6), 2) // 给出 （1 2 4 5 6 3） 。

list_t 截断（list_t 列表，无符号整数 n）；
// 要求：“列表”至少有 n 个元素。

//
// 效果：返回与“列表”相等但不包含其最后 n 个元素的列表。

二叉树——

二叉树是我们在这个项目中使用的另一种基本数据结构。二叉树如果满足以下条件则格式良好：

a) 这是棵空树，或者
b) 它由一个整数元素（称为根元素）以及两个子树（称为左子树和右子树）组成，每个子树都是一个格式良好的二叉树。

此外，我们说一棵二叉树是“叶子”当且仅当它的两个子树均为空树。

在“Project-2-Related-Files.zip”中的“recursive.h”文件定义了“tree_t”类型以及关于树的以下操作：

```
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
```

您需要为二叉树编写几个函数。这些函数必须是递归的，并且不能使用任何循环结构。再次强调，如果您需要定义任何辅助函数，请务必将它们定义为静态函数。

```
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 val);
// EFFECTS: Returns true if the value "val" appears in "tree".
//          Returns false otherwise.

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_max(tree_t tree);
// REQUIRES: "tree" is non-empty.
//
// EFFECTS: Returns the largest 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_hasMonotonicPath(tree_t tree);
// EFFECTS: Returns true if and only if "tree" has at least one
//          root-to-leaf path such that all the elements along the
//          path form a monotonically increasing or decreasing
//          sequence.
//
//          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.
//
//          A monotonically increasing (decreasing) sequence is a
//          sequence of numbers where no number is smaller (larger)
//          than its previous number.
//
```

```
// For example, the tree:
//
//                              4
//                            /   \
//                           /
//                         8
//                        / \
//                       3   16
//                      / \  / \
//
// has two root-to-leaf paths: 4->8->3 and 4->8->16.
// Since the numbers on the path 4->8->16 form a monotonically
// increasing sequence, the function should return true.
// If we change 8 into 20, there is no such a path.
// Thus, the function should return false.
```

bool tree_allPathSumGreater(tree_t tree, int sum)；

// 要求：树不为空

//

// 效果：当且仅当对于每个从根节点到叶节点的

//　　“tree”（树）的路径，即沿该路径的所有元素的总和

//　　大于“总和”

//

//　　从根到叶的路径是树中一系列的元素。

//　　从根元素开始并向下推进

//　　对于一个叶子节点（没有子节点的元素）

//

// 例如，这棵树：

//

//　　4

//　　/　　\

//

//　　1　　5

/　　\　　/\

//　　3　　6

//　　/\/\

//

// 有三条从根节点到叶节点的路径：4 -> 1 -> 3、4 -> 1 -> 6 以及 4 -> 5 。

// 给定总和为 9，路径 4 -> 5 的总和为 9，所以该函数

// 应该返回 false。如果总和为 7，因为所有路径的总和都是

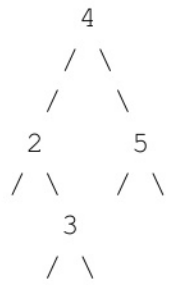
// greater than 7, the function should return true.

我们可以将树之间的特殊关系“被覆盖”定义如下:

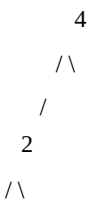
一棵空树被所有的树覆盖着。

- 空树只覆盖着其他空树。
- 对于任意两个非空的树 A 和 B，A 被 B 覆盖当且仅当 A 和 B 的根元素相等，A 的左子树被 B 的左子树覆盖，并且 A 的右子树被 B 的右子树覆盖。

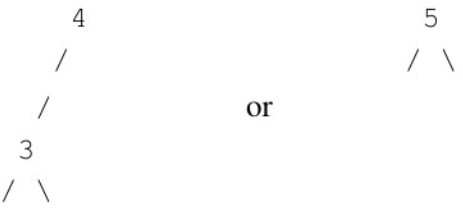
例如，这棵树:



覆盖着这棵树:



但不是那些树:



鉴于这个定义，编写以下函数:

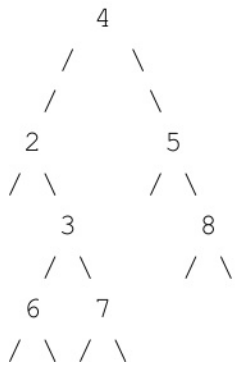
```
bool covered_by(tree_t A, tree_t B);
// EFFECTS: Returns true if tree A is covered by tree B.
```

根据“被覆盖”的定义，我们可以定义一种关系“被包含”。如果一棵树 A 被一棵树 B 包含，那么

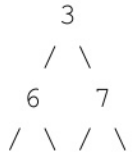
- A 被 B 覆盖，或者，
- A 被 B 的一个子树所覆盖。

请注意，在上述定义中，**一棵树 T 的子**树是一棵空树或者一棵非空树，由树 T 中的一个节点 S 以及树 T 中节点 S 的**所有**下游节点（称为树 T 中节点 S 的后代）组成。

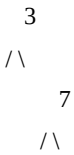
例如，对于这棵树 T



这棵树



是 T 的一个子树。然而，这棵树

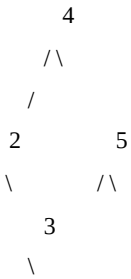


不是。

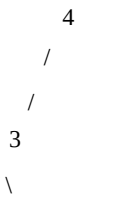
基于“包含”这一定义，这些树

并且

包含在树内



但这棵树不是:



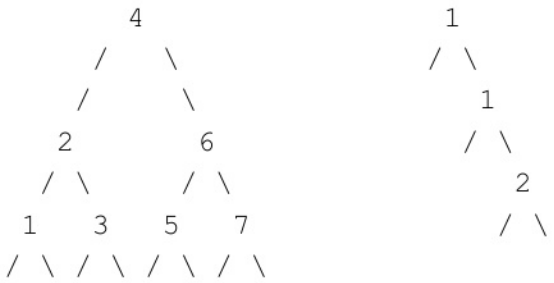
请编写一个函数来实现“包含”关系:

```
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.
```

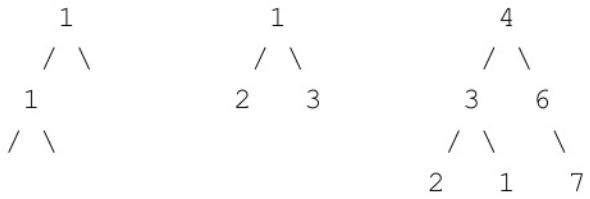
存在一种特殊的二叉树，称为有序二叉树。一个有序二叉树如果满足以下条件则格式良好:

1. 这是一个结构良好的二叉树**并且**
2. 以下内容中有一项是正确的:
   a) 这棵树是空的。
   b) 左子树是一个有序二叉树，左子树中的任何元素都严格小于树的根节点。右子树是一个有序二叉树，右子树中的任何元素都大于或等于树的根节点。

例如，以下这些树均为格式良好的排序二叉树：



而以下的树木则不是：



您需要编写以下用于创建排序二叉树的函数：

```
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.
```

文件

在 Canvas 上的“项目 2 相关文件.zip”中有几个文件：

p2.h	您必须编写的函数的头文件
recursive.h 文件	列表_t 和树_t 接口
“recursive.cpp”	list_t 和 tree_t 的实现。

您应该将上述文件复制到您的工作目录中。**请勿修改这些文件！**您应该将**所有编写的函数都**放在一个名为 p2**.cpp** 的单个文件中（**必须完全如此**！）。您只能使用 <iostream> 和 <cstdlib> 库，不能使用其他库。您不能使用全局变量。您可以**将 p**2.cpp 视为提供了一个函数库，其他程序可能会使用它，就像 recursive.cpp 一样。

测试—

您可以使用以下两个函数分别检查两个列表的等同性以及两个树的等同性。

```
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)));
    }
}
```

为了测试您的代码，您应该创建一组测试用例程序，以测试我们要求您编写的函数。以下是一个简单的示例，供您开始使用：

```
#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;
}
```

请注意，在上述测试程序中，如果函数 reverse 和 append 存在任何错误，返回值将为 -1。如果返回值为 0，那么您通过了上述测试用例。

假设上述测试代码写在“test.cpp”文件中。使用以下 Linux 命令将“test.cpp”与“recursive.cpp”以及您的“p2.cpp”一起编译：

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

要检查返回值，您应该首先在 Linux 中通过键入来运行程序

```
./test
```

然后您可以通过输入来检查返回值。

```
echo $?
```

如果返回值为 -1，则表示出现错误。

您可能还会发现向输出中添加错误消息，或者使用 list_print 和 tree_print 函数打印列表或树形结构会有所帮助。您可以在 Project-2-Related-Files.zip 中再找到两个测试示例 simple_test.cpp 和 treeins_test.cpp 。

提交与截止日期

您只需要提交您的源代码文件 p2.cpp（名称必须**完全一致！**）。源代码文件应通过在线评分系统提交。更多细节请查看助教的公告。截止日期为2024年10月30日晚上11点59分。

评分

你的项目将根据三项标准进行评分：

1. 功能正确性
2. 实施限制
3. 一般风格

功能正确性的一个例子是您的反转函数在所有情况下是否能正确反转一个列表。实现约束的一个例子是 reverse（） 是否是递归的。一般风格指的是您代码的整洁性和可读性。