THE UNIVERSITY OF WESTERN ONTARIO

DEPARTMENT OF COMPUTER SCIENCE
LONDON CANADA

Software Tools and Systems Programming
(Computer Science 2211a)

ASSIGNMENT 4
Due date: Tuesday, November 12, 2019, 11:55 PM

Assignment overview
We would like students to experience arrays, strings, structures, pointers and recursion, to understand and use user defined types in C, to use dynamic allocation of memory, and to use multiple source files.

This assignment consists of two parts.

In part one, you are required to write a C programs to implement binary search trees.

In part two, you are to write a C program to calculate the mathematical constant e.

Part one: 80%

1.1 Preliminaries

A *binary tree* is a tree data structure in which each node has at most two children, called the left child and the right child. Binary trees are a very popular data structure in computer science. We shall see in this exercise how we can encode it using C arrays. The formal recursive definition of a binary tree is as follows. A *binary tree* is

- either empty,
- or a node with two children, each of which is a binary tree.

The following terminology is convenient:

- A node with no children is called a leaf and a node with children is called an internal node.
- If a node B is a child of a node A then we say that A is the parent of B.
- In a non-empty binary tree, there is one and only one node with no parent; this node is called the root node of the tree.

A binary tree T up to n nodes can be encoded in an array A with $n+1$ elements. Indeed, one can always label the nodes with the integers $1, 2, \ldots, n$ identifying elements of array A Each element of A is of two parts, one part is the information of the node and the other part are the left child and the right child represented as integers between 0 and n. A zero represents an empty binary tree. Element of A at index zero will not be used to encode a tree node.
A *binary search tree* (BST), is a binary tree with the following properties.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.
- There must be no nodes with duplicate keys.

Generally, the information part of each node is represented as two fields such that one field is key and the other field is data. For the purpose of this exercise, key is a pair consists of a string and an integer and data is an integer. To compare keys, we may use "*int strcmp*(*char*∗, *char*∗);" in *string.h* to compare the strings in keys.
Since we will use an array to encode a binary tree T, we also need to know if an array location is being used as a node of tree T. For this purpose, we will need another array.

1.2 Questions

The purpose of this exercise is to realize a simple C implementation of binary search trees encoded using arrays. We will use a structure with five members to implement binary search trees where one member will be an array of tree nodes, another member will be an array of unsigned integers, used as a stack, to store all locations of unused tree nodes, a third member is used to keep the size of the array, a fourth member is used to indicate the next unused location, and the last member is to store the root of the tree. To guide you toward this goal, we provide a template program hereafter. We ask you to use this template and fill in the missing code.

```
// ====== this is in data.h

typedef struct {char *name; int id;} Key;
typedef int Data;
typedef struct {Key *key; Data data; int left, right;} Node;
Key *key_construct(char *in_name, int in_id);
int key_comp(Key *key1, Key *key2);
void print_key(Key *key);
void print_node(Node node);
```

```
// ====== this is in data.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "data.h"

// Input: 'in_name': a string ends with '\0'
//        'in_id': an integer
// Output: a pointer of type pointer to Key,
//         pointing to an allocated memory containing a Key
// Effect: dynamically allocate memory to hold a Key
//         set Key's id to be in_id
//         dynamically allocate memory for the Key's name
//         so that name will contain what is in 'in_name'.
// Note:   do not use strdup(), use malloc(), strlen(), and strcpy()
Key *key_construct(char *in_name, int in_id) {

}

// Input:  'key1' and 'key2' are two pointers to Key
// Output: if return value < 0, then *key1 < *key2,
//         if return value = 0, then *key1 = *key2,
//         if return value > 0, then *key1 > *key2,
// Note:   use strcmp() to compare key1->name and key2->name
//         if key1->name == key2->name, then compare key1->id with key2->id
int key_comp(Key *key1, Key *key2) {

}

// Input: 'key': a pointer to Key
// Effect: ( key->name key->id ) is printed
void print_key(Key *key) {

}

// Input: 'node': a node
// Effect: node.key is printed and then the node.data is printed
void print_node(Node node) {

}
```

```
// ====== this is in bst.h

#include "data.h"

typedef struct {Node *tree_nodes; unsigned int *free_nodes;
                int size; int top; int root} BStree_struct;
typedef BStree_struct* BStree;
BStree bstree_ini(int size);
void bstree_insert(BStree bst, Key *key, Data data);
void bstree_traversal(BStree bst);
void bstree_free(BStree bst);



// ====== this is in bst.c

#include <stdio.h>
#include <stdlib.h>
#include "bst.h"

// Input: 'size': size of an array
// Output: a pointer of type BStree,
//         i.e. a pointer to an allocated memory of BStree_struct type
// Effect: dynamically allocate memory of type BStree_struct
//         allocate memory for a Node array of size+1 for member tree_nodes
//         allocate memory for an unsigned int array of size+1 for member free_nodes
//         set member size to 'size';
//         set entry free_nodes[i] to i
//         set member top to 1;
//         set member root to 0;
BStree bstree_ini(int size) {

}

// Input: 'bst': a binary search tree
//        'key': a pointer to Key
//        'data': an integer
// Effect: 'data' with 'key' is inserted into 'bst'
//         if 'key' is already in 'bst', do nothing
void bstree_insert(BStree bst, Key *key, Data data) {

}
```

```
// Input: 'bst': a binary search tree
// Effect: print all the nodes in bst using in order traversal
void bstree_traversal(BStree bst) {

}

// Input: 'bst': a binary search tree
// Effect: all dynamic memory used by bst are freed
void bstree_free(BStree bst) {

}
```

A helper function "static int new_node(BStree bst, Key *key, Data data);" may be useful for binary search tree insertion.
Binary search tree insertion, traversal, and free should be implemented using recursion. You will need to use additional "**helper**" functions for the recursion. Binary search tree insertion, or new_node(), should check the array bound.

A testing main program is given below.

```
// ====== this is a sample main program:  bs_tree_main.c
#include <stdio.h>
#include "bst.h"

int main(void) {

BStree bst;

 bst = bstree_ini(256);

 bstree_insert(bst, key_construct("Once", 1),      11);
 bstree_insert(bst, key_construct("Upon", 22),      2);
 bstree_insert(bst, key_construct("a", 3),         33);
 bstree_insert(bst, key_construct("Time", 4),      44);
 bstree_insert(bst, key_construct("is", 5),        55);
 bstree_insert(bst, key_construct("filmed", 6),    66);
 bstree_insert(bst, key_construct("in", 7),        77);
 bstree_insert(bst, key_construct("Vancouver", 8), 88);
 bstree_insert(bst, key_construct("!", 99),         9);
 bstree_insert(bst, key_construct("Once", 5),      50);
 bstree_insert(bst, key_construct("Once", 1),      10);

 bstree_traversal(bst);

 bstree_free(bst);
}
```

The output of the above sample program.

```
( !            99 )      9
( Once          1 )     11
( Once          5 )     50
( Time          4 )     44
( Upon         22 )      2
( Vancouver     8 )     88
( a             3 )     33
( filmed        6 )     66
( in            7 )     77
( is            5 )     55
```

To compile your program, use "gcc -o bs_tree data.c bst.c bs_tree_main.c".
Your program should read from stdin (should also work if redirect from a file) triples consists of string, integer, and integer (a string followed by an int, and followed by another int, i.e. name id data), one triple per line and insert these triples, in the order they are read, into an initially empty binary search tree. To terminate the stdin, type, on a new line, control-d. You should create several test cases, i.e. binary search tree of different sizes, keys (with data) inserted with different orders, and insertion that will cause array out of bound situation.

Part two: 20%

The goal of this exercise is to implement a C program, euler.c, to calculate the mathematical constant e. Please check Chapter 6, programming project 12, pp. 124. You should consider using type **double** and **long long** for the calculation.

(1) The value of the mathematical constant e can be expressed as an infinite series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

(2) Write a program that approximates e by computing the value of

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots \frac{1}{n!}$$

where n is the largest integer such that $\frac{1}{n!} \geq \epsilon$ and ϵ is a small number entered by the user.

(3) Testing your program by inputing ϵ of the following values.
0.0001
0.0000001
0.0000000001
0.0000000000001

Testing your program

You should test your program by running it on compute.gaul. Capture the screen of your testing by using script command. There should be two script files, one for each part.