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 13, 2018, 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 wrote a C program to calculate the mathematical constant π.

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 can be encoded in an array A with n+1 elements. Indeed, one can always label the nodes with the integers 1, 2, . . . , n such that:

- the root has label 1,

- if an internal node has label i then its left child (if any) has label 2i and its right child (if any) has label 2i + 1.

Using this observation, we can store the nodes of T in A as follows:

- the root of the tree is in A[1], and its left and right children are stored in A[2] and A[3] respectively,
- given the index i of an internal node, different from the root, the indices of its parent Parent(i), left child Left(i), and right child Right(i) can be computed simply by:
  - Parent(i) = $\lfloor i/2 \rfloor$
  - Left(i) = $2i$
  - Right(i) = $2i + 1$

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 represented by each node is of two fields such that one field is key and the other field is data. For the purpose of this exercise, key is a string and an integer pair, and data is an integer. To compare keys, we can 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 three members to implement binary search trees where one member will be an array of tree nodes, another member will be an array of unsigned char to indicate if a location is being used or not, and a third member is used to keep the size of the array. 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 struct {Key *key; int data;} 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 <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:   may use strdup()
Key *key_construct(char *in_name, int in_id) {

}

// Input:  'key1' and 'key2' are two Keys
// 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 char *is_free; int size;} BStree_struct;
typedef BStree_struct* BStree;
BStree bstree_ini(int size);
void bstree_insert(BStree bst, Key *key, int 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 char array of size+1 for member is_free
//         set all entries of is_free to 1
//         set member size to 'size';
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, int 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 memory used by bst are freed
void bstree_free(BStree bst) {

}
```

Binary search tree insertion and traversal should be implemented using recursion. You may need to use "**helper**" functions. Binary search tree insertion should check the array bound.

A sample main program is given below.

```
// ====== this is a sample main program

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

int main(void) {

BStree bst;

 bst = bstree_ini(1000);

 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
```

Your program should read from stdin (or redirect from a file) triples 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. 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 to calculate the mathematical constant π. You should consider using type **double** and **long long** for the calculation. You should consider using **compute.gaul.csd.uwo.ca** since this exercise is time consuming.

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

$$\pi = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{4}{2i-1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} \ldots$$

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

$$\pi \approx \sum_{i=1}^{n} (-1)^{(i+1)} \frac{4}{2i-1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} \ldots (-1)^{n+1} \frac{4}{2n-1}$$

where n is the smallest integer such that $\frac{4}{2(n+1)-1} < \epsilon$ for an user entered small (double precision) number ϵ.

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

Testing your program

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