THE UNIVERSITY OF WESTERN ONTARIO

DEPARTMENT OF COMPUTER SCIENCE
LONDON   CANADA

Software Tools and Systems Programming
(Computer Science 2211a)

ASSIGNMENT 3
Due date: Tuesday, November 14, 2023, 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 num;} Key;
typedef struct {Key key; int data;} Node;
Key key_construct(char *in_name, int in_num);
int key_comp(Key key1, Key key2);
void print_key(Key key);
void print_node(Node node);
```

```c
// ====== 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_num': an integer
// Output: a Key,
// Effect: copy in_name to Key's name
//         set Key's num to be in_num
Key key_construct(char *in_name, int in_num) {

}


// 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.num with key2.num
int key_comp(Key key1, Key key2) {

}

// Input: 'key': a Key
// Effect: key.num key.name are 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) {

}
```

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

#include "data.h"

typedef struct {Node *tree_nodes; unsigned char *is_free; int size;} 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 BStree,
// Effect: 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 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) {

}
```

4

```
// 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) {

}
```

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(600);

 bstree_insert(bst, key_construct("Oxford Street", 70),        2500);
 bstree_insert(bst, key_construct("3 Valleys Place", 8),       2000);
 bstree_insert(bst, key_construct("Western Road", 101),        2500);
 bstree_insert(bst, key_construct("3rd Street", 74),           3000);
 bstree_insert(bst, key_construct("4 Oaks Crescent", 4),       2100);
 bstree_insert(bst, key_construct("Abbott Street", 10),        2300);
 bstree_insert(bst, key_construct("1st Street", 100),          1000);
 bstree_insert(bst, key_construct("Beech Drive", 204),         1800);
 bstree_insert(bst, key_construct("Beechbank Crescent", 36),   2400);
 bstree_insert(bst, key_construct("Beechmount Crescent", 36),  2500);
 bstree_insert(bst, key_construct("Oxford Street", 46),        2600);

 bstree_traversal(bst);

 bstree_free(bst);
}
```

The output of the above sample program.

```
100  1st Street                              1000
  8  3 Valleys Place                         2000
 74  3rd Street                              3000
  4  4 Oaks Crescent                         2100
 10  Abbott Street                           2300
204  Beech Drive                             1800
 36  Beechbank Crescent                      2400
 36  Beechmount Crescent                     2500
 46  Oxford Street                           2600
 70  Oxford Street                           2500
101  Western Road                           2500
```

Your program should work for reading from stdin or redirect from a file. First, read the size of the tree_node array of the binary search tree. After initialize the binary search tree, your program should read a sequence of key and data for tree nodes, each with two lines, one line for the key which contains an int followed by a string with spaces, and the next line with one int for the data. Insert these key and data pairs, 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.

Hint: to read the line with key, use:

```
scanf("%d ", &number);
fgets(str, sizeof(str), stdin);
```

This will read an integer to **number**, skip the spaces after the integer and then read a string to **str** which contains all the characters up to and include the newline. You may replace '\n' with '\0' to remove the newline from **str**.

Hint: to compile multiple .c files, use:

```
gcc -o bs_tree data.c bst.c main.c
```

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:

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

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

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

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.