CSCI2100B Data Structures

Assignment 3
Due Date: 21 April 2020

Written Exercises

1. Suppose t1 is initially an empty AVL tree. Nodes with the following keys are then inserted, in the following sequence, into t1:

   5, 6, 13, 7, 2, 1, 3, 9, 12, 4, 10, 11, 8

   Show clearly what t1 should look like after each node is inserted. Remember to perform rotations whenever necessary.

2. Repeat the above question, assuming that the tree is only an ordinary binary search tree (not an AVL tree). You do not need to show what the tree should look like after each node is inserted. Only the final tree needs to be shown.

Programming Exercises

3. Write a function quicksortList that receives a listADT argument, sorts the list into ascending order, and returns the sorted list as a listADT value.

   ```
   listADT quicksortList(listADT);
   ```

   For example, if you run the following main program

   ```
   main() {
       listADT list1, list2;
       list1 = Cons(9, Cons(3, Cons(8, Cons(7, EmptyList()))));
       list2 = quicksortList(list1);
   }
   ```

   then list1 should be the list [9, 3, 8, 7] and list2 should become the list [3, 7, 8, 9].

   Hint: If the list to be sorted is not empty, you can choose the head of the list as the pivot. You write a function SmallerList that returns all the elements that are less than the pivot, and a function LargerOrEqList that returns all the elements that are greater than or equal to the pivot:

   ```
   listADT SmallerList(listADT);
   listADT LargerOrEqList(listADT);
   ```

   For example, if listA=[5, 2, 3, 8, 9, 1, 7, 0], then the function calls

   ```
   SmallerList(listA) = [2, 3, 1, 0] /* not necessarily in this order */
   LargerOrEqList(listA) = [8, 9, 7] /* not necessarily in this order */
   ```

   Note that listA should not be changed after these function calls. Also the head (5) is in neither of these two lists.

4. In C, you write functions that have variable number of arguments. For example, one can write a function test1 that prints an arbitrary number of integers, as follows:

```
#include <stdio.h>
#include <stdarg.h>

void test1(int number, ...) {
    va_list args;
    int x;

    va_start(args, number);
    for (int k=0; k<number; k++) {
            x = va_arg(args, int);
            printf("%d ", x);
    }

    va_end(args); // va_end is used, instead of return.
}
```

then you can write the following function calls:

```
    test1(1, 3); /* this prints 3 */
    test1(3, 7, 8, 9); /* this prints 7 8 9 */
    test1(2, 4, 5); /* this prints 4 5 */
```

and the integer actual arguments will all be printed.

To use this feature, you must include <stdarg.h>, and declare a variable of type va_list in the function. Note that you can have a number of named arguments in the function definition, followed by a number of unnamed arguments (collectively represented by three dots ('...')). Before you access the arguments, you must first call va_start(a, x), where a is a variable of type va_list, and x is the identifier of the last named argument (the one just before the ', ...'). After that, you can access the arguments one after another by calling va_arg(a, t), where a is the va_list variable and t is the type of the arguments. Finally, you must call va_end(a) to implement the clean-up process and the return operation, where a is the va_list variable. If you call return before va_end, no one knows what will happen.

Now write a new function ListOf for the List ADT, so that you can create a list of many elements. For example, you can create the list [9, 5, 8, 7] and assign it to list L1 using

```
listADT L1 = ListOf(4, 9, 5, 8, 7); /* the first argument is the number of elements */
```

You can also create the list [7, 1, 3] and assign it to list L2 using

```
listADT L2 = ListOf(3, 7, 1, 3);
```

The list [8] can be created and assigned to list L3 using

```
listADT L3 = ListOf(1, 8);
```

The empty list [] can be created and assigned to list L4 using

```
listADT L4 = ListOf(0);
```

Finally, the first argument cannot be a negative number!

```
listADT L5 = ListOf(-1); // ERROR! //
```

5. There is another way to create a listADT with a number of given elements. For example, one can write a new function ListOf2 for the List ADT, that you can create a list of many elements. You can first put all the elements into an array of intergers, and pass the array to ListOf2. For example, you can create the list [9, 5, 8, 7] and assign it to list L1 using

   ```
   listADT L1 = ListOf2(4, E);
   ```

   where E is an integer array that contains the four integers 9, 5, 8, and 7. You can also create the list [7, 1, 3] and assign it to list L2 using

   ```
   listADT L2 = ListOf2(3, E);
   ```

   if E is an integer array that contains the three integers 7, 1, and 3.

   The empty list [] can be created and assigned to list L4 using

   ```
   listADT L4 = ListOf2(0, E);
   ```

   where no one cares what E contains. Finally, the first argument cannot be a negative number!

   ```
   listADT L5 = ListOf2(-1, E); // ERROR! //
   ```

6. Write a function PrintList that has a listADT argument and prints the list.

   ```
   void PrintList(listADT);
   ```

   For example, if you run the following program

   ```
   main() {
       listADT list1, list2;
       printf("Testing case 1:\n");
       list1 = ListOf(4, 9, 3, 8, 7);
       printf("Before sorting: "); PrintList(list1); printf("\n");
       list2 = quicksortList(list1);
       printf("After sorting: "); PrintList(list2); printf("\n");
       printf("Testing case 2:\n");
       list1 = EmptyList();
       printf("Before sorting: "); PrintList(list1); printf("\n");
       list2 = quicksortList(list1);
       printf("After sorting: "); PrintList(list2); printf("\n");
   }
   ```

   then the following lines should be printed:

   ```
   Testing case 1:
   Before sorting: [9, 3, 8, 7]
   After sorting: [3, 7, 8, 9]
   Testing case 2:
   Before sorting: []
   After sorting: []
   ```

(**Hint**: you can write either a recursive or a non-recursive version of PrintList. You are encouraged to try both.)

7. Finally, write a simple main program to test all these functions. The main program should open a text file data.txt, and for each line in the file, perform the following task:

   (1) Print the string "Line " and the line number.
   (2) Construct a list, using all the integers shown on the line as elements. (Should you use ListOf? Should you use ListOf2? Or should you use Cons only?)
   (3) Print out the list using PrintList.
   (4) Sort the list using quicksortList.
   (5) Print out the sorted list using PrintList.
   (6) Print the string "Done with Line " and the line number.
   (7) Print a blank line

   For example, if data.txt contains the following 4 lines (note that the third line is blank)

```
3 7 2 1 9
8

234 0 2 -1
```

   Then the output should look like this

```
Line 1
[3, 7, 2, 1, 9]
[1, 2, 3, 7, 9]
Done with Line 1

Line 2
[8]
[8]
Done with Line 2

Line 3
[]
[]
Done with Line 3

Line 4
[234, 0, 2, -1]
[-1, 0, 2, 234]
Done with Line 4
```

8. Write a function parent for the BinaryTreeADT

```
TreeNodeADT parent(BinaryTreeADT, TreeNodeADT);
```

   that returns the parent of a node (the second argument) in a binary tree (the first argument). If the node does not have a parent, then you can simply return the special error node.

Write a testing main program that open a text file data.txt, and for each line in the file, perform the following task:

(1) Print the string "Tree " and the line number.
(2) Construct a <u>binary **search tree**</u> by inserting nodes into the tree, using all the integers shown on the line as keys of the nodes, in the order of the integers. You do not need to care about the data in the tree nodes (who cares?). You may freely use any functions in the lecture notes.
(3) Preform an <u>in-order</u> traversal of the tree. For each of the tree nodes visited during the traversal, print out the key of the node and the key of its parent node.
(4) Print the string "Done with Tree " and the line number.
(5) Print a blank line

For example, if data.txt contains the following 1 line:

```
3 7 2 1 9
```

The output should look like this

```
Tree 1
Parent of 1 is 2
Parent of 2 is 3
Parent of 3 does not exist
Parent of 7 is 3
Parent of 9 is 7
Done with Tree 1
```