CSCI2100B Data Structures

Assignment 2
Due Date: 24 March 2020

Written Exercises

1. Use **the selection sort algorithm** to sort the following sequence of integers into ascending order:

   8, 6, 8, 7, 5, 9, 1

   Show clearly each step in the process of sorting.

2. What is the computational complexity of the following function?

```
int Mystery1(int n) {

    int sum = 0;
    for (int i=0; i<n; i++) for (int j=2*i; j<4*i; j++) sum += i*j;
    return (sum);
}
```

3. Write the following functions as recursive functions in C. You should use the list operations defined in List ADT shown in the lecture notes. You can simply call exit(EXIT_FAILURE) when error conditions are encountered. Note that your functions should work well with any correct implementation of List ADT.

   a) The function max that accepts a listADT argument and returns an int value that is the maximum value in the list. For examples:
      - max([4,5,6,7]) = 7
      - max([0]) = 0
      - max([]) = ERROR

   b) The function lastButOne that accepts a listADT argument and returns an int value that is the last-but-one value in the list. For examples:
      - lastButOne([3,4,8,**2**,4]) = 2
      - lastButOne([**0**,1]) = 0
      - lastButOne([]) = ERROR
      - lastButOne([4]) = ERROR

   c) The function member that accepts an int argument and a listADT argument, and returns an int value 1 if the int argument is in the listADT argument, or 0 otherwise. For examples:
      - member(8, []) = 0
      - member(2, [3,4,8,2,4]) = 1
      - member(4, [3,5]) = 0

   d) The function firstN that accepts a listADT argument and an int argument N, and returns a listADT value that is the list of the first N elements of the argument. For examples:
      - firstN([3,4,8,2,4], 3) = [3,4,8]
      - firstN([2,4,6], 8) = ERROR
      - firstN([], 0) = []
      - firstN([], 1) = ERROR
      - firstN([8,7,6], 0) = []

e) The function afterFour that accepts an int argument and a listADT argument, and returns a listADT result that is the same as the List argument, except that the int argument appears after the first value 4 in the list. For examples:
   - afterFour(**7**, [3,4,8,2,4]) = [3, 4, **7**, 8, 2, 4]
   - afterFour(**7**, [3,5]) = [3, 5]
   - afterFour(**6**, [2,8,4]) = [2, 8, 4, **6**]
   - afterFour(**6**, [4,2,8,4]) = [4, **6**, 2, 8, 4]

f) The function listEqual that accepts two listADT arguments, and returns an int value 1 if the listADT arguments are the same, or 0 otherwise. For examples:
   - listEqual([], [3,4,8,2,4]) = 0
   - listEqual([], []) = 1
   - listEqual([39,3,8,12,3], [39,3,8,11,3]) = 0
   - listEqual([39,3,8,12,3], [39,3,8,12,3]) = 1

4. In the implementation version 2.0 of List ADT, we use a linked list based implementation of lists. On page 13 of lecture notes LL10-1, we show **list1** and **list2** after a few statements are executed. For each of the code segments shown below, show the linked list based implementation of lists **list1**, **list2** and **list3** after all the statements in the code segment are executed (you should assume that list1, list2 and list3 are all properly declared as listADT variables). You should also indicate what will be outputted from each of the code segments that contain printf statements.

a)
```
list1 = Cons(4, Cons(5, Cons(6, EmptyList())));
list2 = Cons(Head(list1), list1);
```

b)
```
list1 = Cons(4, Cons(5, Cons(6, EmptyList())));
list1 = Cons(Head(list1), Tail(list1));
```

c)
```
list1 = Cons(3, Cons(7, Cons(6, EmptyList())));
list2 = Cons(Head(list1), Tail(list1));
list3 = Cons(0, list2);
```

d)
```
list1 = Cons(8, Cons(4, Cons(5, EmptyList())));
list2 = Tail(list1);
list3 = Cons(Head(list1), list2);
if (listEqual(list1, list3))
        printf("listEqual(list1, list3)\n");
else
        printf("Not listEqual(list1, list3)\n");
if (list1==list3)
        printf("list1==list3");
else
        printf("list1!=list3");
```

e)
```
if (listEqual(EmptyList(), EmptyList()))
        printf("listEqual(EmptyList(), EmptyList())\n");
else
        printf("Not listEqual(EmptyList(), EmptyList())\n");
if (EmptyList() == EmptyList())
        printf("EmptyList() == EmptyList()");
else
        printf("EmptyList() != EmptyList()");
```

f)
```
list1 = Cons(8, Cons(4, Cons(7, EmptyList())));
list3 = Tail(Tail(list1));
list2 = Cons(Head(Tail(list1)), Cons(Head(list1), list3));
```

Programming Exercises

5. **PART I**

In PART I of this assignment we shall write an implementation file for symbol table ADT. In this implementation, the keys are character strings of length of at least 1, and values are list ADT of length of at least 1. We adopt the convention that if an entry with a particular key does not exist in the table, then the Lookup function should return an empty list for that key. In addition, we add a new ShowAll function, the description of which will be provided later.

The header file symtab.h is hence revised as follows.

```
/*
 * File: symtab.h
 */
#include "list.h" // Please refer the lecture notes.

typedef struct symtabCDT *symtabADT;
typedef listADT symtabEntryT;
typedef void (*symtabFnT)(char*, symtabEntryT);

symtabADT EmptySymbolTable(void);
void Enter(symtabADT, char*, symtabEntryT);
symtabEntryT Lookup(symtabADT, char*);
    // It returns an empty list if the entry does not exist.
void forEachEntryDo(symtabFnT, symtabADT);

void ShowAll(symtabADT); // Detailed description will be provided later.
```

We shall implement the symbol table ADT using hash tables **with linear probing**. Part of the implementation file symtab.c is shown below.

```
/*
 * File: symtab.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"
#include "symtab.h"

typedef struct cellT {char *key; symtabEntryT value;} cellT;

struct symtabCDT {
    cellT **bucket; // An array of cellT*, which can be extended.
    int nBuckets; // Current number of buckets (size of the bucket array)
    int nEntries; // Current number of entries.
};
```

```
int Hash(char *s, int nBuckets) { // DO NOT MODIFY.
    unsigned long hashcode = 0;
    for (int i=0; s[i]!='\0'; i++) hashcode = (hashcode<<5) + s[i];
    return (int) (hashcode % nBuckets);
}

symtabADT EmptySymbolTable() { // DO NOT MODIFY.
    symtabADT T1 = (symtabADT) malloc(sizeof(*T1));
    T1->bucket = (cellT*) malloc(5*sizeof(cellT*)); // Initial size is 5.
    T1->nBuckets = 5; T1->nEntries = 0;
    for (int i = 0; i < T1->nBuckets; i++) T1->bucket[i]=(cellT*)NULL;
    return T1;
}

void Enter(symtabADT T1, char* K1, symtabEntryT V1) {
// You need to implement this.
…
}

symtabEntryT Lookup(symtabADT T1, char* K1) {
// You need to implement this.
…
}

void forEachEntryDo(symtabFnT f, symtabADT T1) { // DO NOT MODIFY.
    for (int i = 0; i < T1->nBuckets; i++)
            if (T1->bucket[i] != NULL)
                    f(T1->bucket[i]->key, T1->bucket[i]->value);
}

void printList(listADT L) { // DO NOT MODIFY.
    if (ListIsEmpty(L))
            printf("nil");
    else {
            printf("%d:", Head(L));
            printList(Tail(L));
            }
}

void ShowAll(symtabADT T1){ // DO NOT MODIFY.
    for (int i = 0; i < T1->nBuckets; i++)
            if (T1->bucket[i] != NULL) {
                    printf("Bucket %d\t Key:%s\t Value ", i, T1->bucket[i]->key);
                    printList(T1->bucket[i]->value);
                    printf("\n");
            }
}
```

NOTE

- **typedef listADT symtabEntryT;**

  In this implementation we introduce a new type **symtabEntryT** for symbol table entries, which is the same as **listADT**.

- **typedef struct cellT {char *key; symtabEntryT value;} cellT;**
  ```
  struct symtabCDT {
          cellT **bucket; // An array of cellT*, which can be extended.
          int nBuckets; // Current number of buckets (size of the bucket array)
          int nEntries; // Current number of entries.
    };
  ```

  A hash table is implemented using an array of buckets. Each bucket stores a pointer to a **struct cellT**. Inside each struct cellT store the key (char string) and the value (listADT).

  Note that we use open addressing hashing with linear probing. Therefore, each bucket points to one entry (one **struct cellT**), instead of a linked list of **struct cellT**.

  Note also that the initial size of the bucket array in an empty symbol table is only 5. When the table is full (that is, when **nBuckets == nEntries**), the bucket array will need to be extended (consider using realloc), <u>and rehashing has to be done</u>.

- **symtabADT EmptySymbolTable(void);**
  For an empty symbol table, the initial number of buckets is only 5. All buckets are initialized to contain the NULL pointer. If **bucket[i]** is NULL, then it is vacant. Otherwise, it is in use.

- **void Enter(symtabADT, char*, symtabEntryT);**
  You need to implement this, using linear probing, and rehashing when necessary. Show me that you can do it. <u>**Important**: every time when the hash table is full, you can extend the bucket number by only 7. This is because I want to make sure that rehashing will really be run.</u>

- **symtabEntryT Lookup(symtabADT, char*);**
  You need to implement this, using linear probing, and rehashing when necessary. Show me that you can do it. <u>Note that it returns an empty list if the entry does not exist.</u>

- **void forEachEntryDo(symtabFnT, symtabADT);**
  This is provided. Note that it is different from the version given in the lecture notes, because in the lecture notes we use separate chaining.

- **void ShowAll(symtabADT);**
  This is provided. It prints out, for each entry, the position of the entry, the key, and the value as a list. Note that the list **[3,8,7,4]** will be printed as **3:8:7:4:nil**. The empty list **[]** will be printed as **nil**. (One can imagine that '**:**' is a right associative infix operator for '**Cons**'—that is, **Cons(h,t)** is represented by **h:t**!)

  This function is mainly used for testing. You can call it for debugging.

<u>PART II</u>

In PART II of this assignment we shall write an application program that makes use of the symbol table ADT you implement in PART I.

Your program will first open a data file **data.txt** similar to what is shown below.

```
lek yuen
3
wo che
5
wo che
8
wo che
5
wo che
7
sha kok
0
hin keng
9
lung hang
8
sha kok
2
sha kok
4
lung hang
8
```

In this input data file, lines 1, 3, 5, … are the keys of entries, and lines 2, 4, 6, … are the integers to be added to the listADT values corresponding to the keys in the preceding lines. For example, after the above sample data file is processed, the symbol table should contain the following entries:

key	**value**
"lek yuen"	[3]
"wo che"	[7,5,8,5]
"sha kok"	[4,2,0]
"hin keng"	[9]
"lung heng"	[8,8]

Note that the order of elements in the lists is not important.

Finally, after all data in the input file is processed, the program will print out all the keys in the table, and the corresponding sums of elements in the lists. For the above data file, for example, the output should look like the following:

```
lek yuen     3
wo che       25
sha kok      6
hin keng     9
lung heng    16
```

Note that the order of entries in the output is not important. (Hint: you may consider writing a callback function that prints, for each entry, the key of the entry and sum of elements in the value list of the entry, and pass it to forEachEntryDo.)

—End of Assignment—