CSCI 2100C Data Structures

Assignment 2
Due Date: 15 March 2022

Written Exercises

1. The following sentence is quoted from *Data Structures and Algorithms* by Aho, Ullman and Hopcroft:

   'We say that $T(n)$ is $O(f(n))$ if there are constants c and n_0 such that $T(n) \leq cf(n)$ whenever $n \geq n_0$.'

   Based on this definition,

   a) Show that the function $T(n) = n$ is $O(n)$. (3pts)
   Let $c = 1$ and $f(n) = n$ then $T(n) = n \leq cf(n)$ for $n_0 = 1$

   b) Show that the function $T(n) = 2n + 1$ is $O(n)$. (3pts)
   Let $c = 3$ and $f(n) = n$ and $n_0 = 1$, then when $n > n_0$, we have $2n + 1 < 3n$, i.e. $T(n) = 2n + 1 < 3f(n)$

   c) Show that the function $T(n) = 5n^4 + 2n^2$ is $O(n^4)$. (3pts)
   Let $c = 7$ and $f(n) = n^4$ and $n_0 = 1$, then when $n > n_0$, we have $5n^4 + 2n^2 < 7n^4$, i.e. $T(n) = 5n^4 + 2n^2 < 7f(n)$

2. Write the function mergeLists that merges two argument lists into one:

   listADT mergeLists(listADT, listADT);

   so that if the two argument lists are already sorted into ascending order, then the returned list is naturally sorted in ascending order.

   a) Write this function as a recursive function. *Hint: Call mergeLists with one of the argument lists and the tail of the other argument list.* (5pts)

```
listADT mergeLists(listADT l1, listADT l2) {
    if (ListIsEmpty(l1)) {
        return l2;
    }
    else if (ListIsEmpty(l2)) {
        return l1;
    }
    else if (Head(l1) <= Head(l2)) {
        listADT tail = mergeLists(Tail(l1), l2);
        return Cons(Head(l1), tail);
    }
    else {
        listADT tail = mergeLists(l1, Tail(l2));
        return Cons(Head(l2), tail);
    }
}
```

b) Write this function as a nonrecursive function. (5pts)

```
listADT mergeLists(listADT l1, listADT l2) {
    stackADT stack = EmptyStack();
    while(!ListIsEmpty(l1) && !ListIsEmpty(l2)) {
        if (Head(l1) <= Head(l2)) {
            Push(stack, Head(l1));
            l1 = Tail(l1);
        }
        else{
            Push(stack, Head(l2));
            l2 = Tail(l2);
        }
    }
    while(!ListIsEmpty(l1)){
        Push(stack, Head(l1));
        l1 = Tail(l1);
    }
    while(!ListIsEmpty(l2)){
        Push(stack, Head(l2));
        l2 = Tail(l2);
    }
    listADT result = EmptyList();
    while(!StackIsEmpty(stack)){
        result = Cons(Pop(stack), result);
    }
    return result;
}
```

3. Write the function splitList that takes the first argument list and splits it into two: a list of all the elements at the odd position (the second argument), and a list of all the elements at the even position (the third argument):

    void splitList(listADT list, listADT* oddList, listADT* evenList);

    Note that the first argument is the input and the other two arguments are output.

    a) Write this function as a recursive function. (5pts)

```
void splitList(listADT list, listADT* L1, listADT* L2) {
    if (ListIsEmpty(list))
        *L1 = *L2 = EmptyList();
    else {
        /* order of L1 & L2 is reversed */
        splitList(Tail(list), L2, L1);
        *L2 = Cons(Head(list), *L2);
    }
}
```

b) Write this function as a nonrecursive function. (5pts)

```
void splitList(listADT list, listADT* oddList, listADT* evenList) {
    *oddList = EmptyList();
    *evenList = EmptyList();
    int even = 1;
    while(!ListIsEmpty(list)){
        if (even == 1)  {
            *evenList = Cons(Head(list), *evenList);
            even = 0;
        }
```

```
        else {
            *oddList = Cons(Head(list), *oddList);
            even = 1;
        }
        list = Tail(list);
    }
}
```

4. Using mergeLists and splitList, Write the mergeSort function that sorts the argument list into ascending order: (6pts)

listADT mergeSort(listADT);

```
listADT mergeSort(listADT list) {
    if (ListIsEmpty(list)) return list;
    listADT oddList;
    listADT evenList;
    splitList(list, &oddList, &evenList);
    oddList = mergeSort(oddList);
    evenList = mergeSort(evenList);
    return mergeLists(oddList, evenList);
}
```

5. Use **the quicksort algorithm** to sort the following sequence of integers into ascending order:

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

   Draw a series of diagrams to show clearly each step in the process of sorting. (8pts)

6. 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 listIsSorted that accepts a listADT argument and returns an int value 1 if the argument list is sorted into descending order, or 0 otherwise. (3pts) For examples:
      - listIsSorted([7,8,1,5,3]) = 0
      - listIsSorted([7,6,5,4]) = 1
      - listIsSorted([0]) = 1
      - listIsSorted([]) = 1

```
int listIsSorted(listADT list) {
    if (ListIsEmpty(list)) return 1;
    if (ListIsEmpty(Tail(list))) return 1;
    if (Head(list) >= Head(Tail(list)))
        return listIsSorted(Tail(list)));
}
```

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

```
int member(int value, listADT list) {
    if (ListIsEmpty(list)) {
        return 0;
    }
    if (Head(list) == value) {
        return 1;
    }
    return member(value, Tail(list));
}
```

c) 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, until the end of the list. For examples:
   - firstN([3,4,8,2,4], 3) = [3,4,8]
   - firstN([2,4,6], 8) = [2,4,6]
   - firstN([], 0) = []
   - firstN([], 1) = []
   - firstN([8,7,6], 3) = [8,7,6]

```
listADT firstN(listADT list, int n) {
    if (ListIsEmpty(list) || n == 0) {
        return EmptyList();
    }
    return (Cons(Head(list), firstN(Tail(list), n - 1)));
}
```

d) The function oddOnly that accepts an int argument and a listADT argument, and returns a listADT result that is the same as the List argument, except that all the even numbers are removed. (3pts) For examples:
   - oddOnly([3,4,8,2,4]) = [3]
   - oddOnly([3,5]) = [3,5]
   - oddOnly([2,8,4]) = []
   - oddOnly([]) = []

```
listADT oddOnly(listADT list) {
    if (ListIsEmpty(list)) {
        return EmptyList();
    }
    if (Head(list) % 2 == 1)
        return (Cons(Head(list), oddOnly(Tail(list))));
    else
        return oddOnly(Tail(list));
}
```

e) The function allDifferent that accepts a listADT argument, and returns an int value 1 if all elements of the listADT arguments are different, or 0 otherwise. (3pts) For examples:
   - allDifferent([3,4,8,2,4]) = 0
   - allDifferent([2]) = 1
   - allDifferent([]) = 1
   - allDifferent([39,3,8,12,4]) = 1
   - allDifferent([39,4,8,12,4]) = 0

```
int allDifferent(listADT list) {
    if (ListIsEmpty(list)) return 1;
    return allDifferent(Tail(list)) ? !member(Head(list),
Tail(list)) : 0;
}
```

7. In the implementation version 2.0 of list ADT, we use a linked list-based implementation of lists. 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. (3pts x 4)

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

   b) list1 = Cons(3, Cons(7, Cons(6, EmptyList())));
      list2 = Cons(Head(list1), TailTail((list1)));
      list3 = Cons(0, list2);

   c) listElementT firstElement, secondElement;
      list1 = Cons(8, Cons(4, Cons(7, EmptyList())));
      firstElement = Head(list1);
      secondElement = Head(Tail(list1));
      list3 = Cons(secondElement, Tail(Tail(list1)));
      list2 = Cons(secondElement, Cons(firstElement, list3));

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

Programming Exercises

8. There are two parts in this programming exercise.
   Programming basisc – (6Pts)

   a) **(Part 1** - 12pts**)** In the lecture notes we have seen how to implement symbol tables using Hash tables. In the version we present in the lecture notes, we use the technique of *separate chaining* to resolve collisions. We shall call this version 1.0.

      Implement version 2.0 of symtabADT, in which you use *linear probing* (instead of separate chaining) for collision resolution. You should use F(*i*)=*i* in your implementation. Also you should use the #define directive to define an int constant INITIALTABLESIZE in symtab.c, which will be used in the program as the size of the Hash table. You should use the following hash function.

```
int Hash(char *s, int n) {
  unsigned long hashcode = 0UL;
  for (int i=0; s[i]!='\0'; i++) hashcode = (hashcode<<6) + (unsigned long) s[i];
  return (int) (hashcode % n);
}
```

In your implementation, you should use the following definition for struct symtabCDT:

```
typedef struct bucketT {
  int inUse; // This is 1 if the bucket is in use, or 0 otherwise
  char *key;
  void *value; } bucketT;

struct symtabCDT { bucketT *bucket; };
```

You shall need to implement at least four functions: EmptySymbolTable, Enter, Lookup, and ForEachEntryDo.

Important Notes:

- Use 31 as the value of INITIALTABLESIZE for testing.
- In the EmptySymbolTable function, you should initialize the variable bucket in symtabCDT to be an array of size INITIALTABLESIZE. Remember to set the variable inUse of each bucket to 0.
- In the function Enter, if the table is full and no vacant buckets are available, then the array bucket in symtabCDT will need to be extended by 10. After that, rehashing will be performed.

You only need to submit **symtab.c** that implements version 2.0 of symtabADT.

b) **(Part 2** - 12pts**)** Prof. Chan is teaching a course that is open to both postgraduate and undergraduate students. Prof. Chan decides to use different scales for postgraduate and undergraduate students to convert from numeric marks to grades.

The class list with marks of individual students is stored in a student data file **studentData.txt**. The format of the student data file is shown as follows:

```
218432
CHAN TAI MAN
78
200987
CHEUNG MAN MING
90
217748
MAN TAI SUN
75
216639
YEUNG WING KEUNG
98
```

A student's information is stored in four separate lines in the student data file. The first line is a 6-digit student number, the second line is the student's name, and the third line the student's mark (which is an integer). Whether a student is a postgraduate student or an undergraduate student is indicated by his or her student number: if the second digit is 0, then the student is a postgraduate student; otherwise, the student is an undergraduate student. The total number of students is not known in advance, so you need to read the file until the end of file is reached.

There is another file called the data correction file **dataCorrection.txt**. If there is any mistake in the **studentData.txt** file, then the correction information will appear in the data correction file for correction purpose. A student's information is stored in two separate lines in the data correction file. The first line is a 6-digit student number, and the second line the student's correct mark (which is an integer). We assume that there is only one data correction file for all teachers, *that is*, it might contain information about a student whose student number does not appear in the original student data file. This is because it is actually a correction for the student data file of another teacher. If this happens, the you can simply ignore it. The following is an example of the data correction file:

```
218432
80
200100
82
216639
98
```

In this case the student whose student number is 200100 does not appear in the data file, so it can be ignored.

Write a program that first reads the student data file and then the data correction file, and outputs the grades of each of the students in a file. The following is a sample output for the above input file (the order is not important).

```
218432 CHAN TAI MAN
B
200987 CHEUNG MAN MING
A
217748 MAN TAI SUN
B
216639 YEUNG WING KEUNG
A
```

Note 1

You must use the following implementation:

1. You must first use typedef to define a structure type **StudentT** for a structure with 3 fields.
   - The first field **sName** is a string that stores the name of the student.
   - The second field **level** is a value of type **studentLevelT** (**studentLevelT level**), where **studentLevelT** is an enum type defined as
     typedef enum {UG, PG} studentLevelT;
   - The last field **mark** is an int value (**int mark**) that stores the student's examination mark.

2. There is a function **char getGrade(StudentT)** to determine the grade for each of the students.

   The conversion schemes for undergraduates and postgraduates are as follows:

Undergraduate	
86 – 100	A
71 – 85	B
61 – 70	C
50 – 60	D
< 50	F

Postgraduate	
85 – 100	A
71 – 84	B
60 – 70	C
< 60	F

Note 2

You should use a symbol table to store the students' information in your program. The key is the student IDs. The data stored in the table should be pointers of type StudentT*, which point to structures of type StudentT. For simplicity, **you can assume that the table will never be full.**

Your program should work well using the symtab.h and symtab.c in the lecture notes and in Part 1 of the question.

Note 3

To output the grades of all students, you must use a call back function
void displayStudentNameAndGrade(char*, StudentT*)
to display all entries in the table (the order is not important). You may use the mapping function **forEachEntryDo** defined in the lecture notes.

(IMPORTANT) Grading Explanation for Question 8:

Full Score = (6+12+12=30Pts)

The first 6 points is for programming basics. If the code can be compiled, and contains all necessary components, then you can get full (6) marks.

If you only implement one sub-question, e.g. main.c or symtab.c, the highest mark you can obtain is 3 marks (for programming basics).

If your program contains all necessary components but also has some compilation errors, then the mark will be partially deducted.

If you falsely place some function prototypes, function implementations or type definitions in the wrong position (e.g. header file(s)), then the mark will be partially deducted.

In Q8a and Q8b, file symtab.c and main.c will be tested separately.

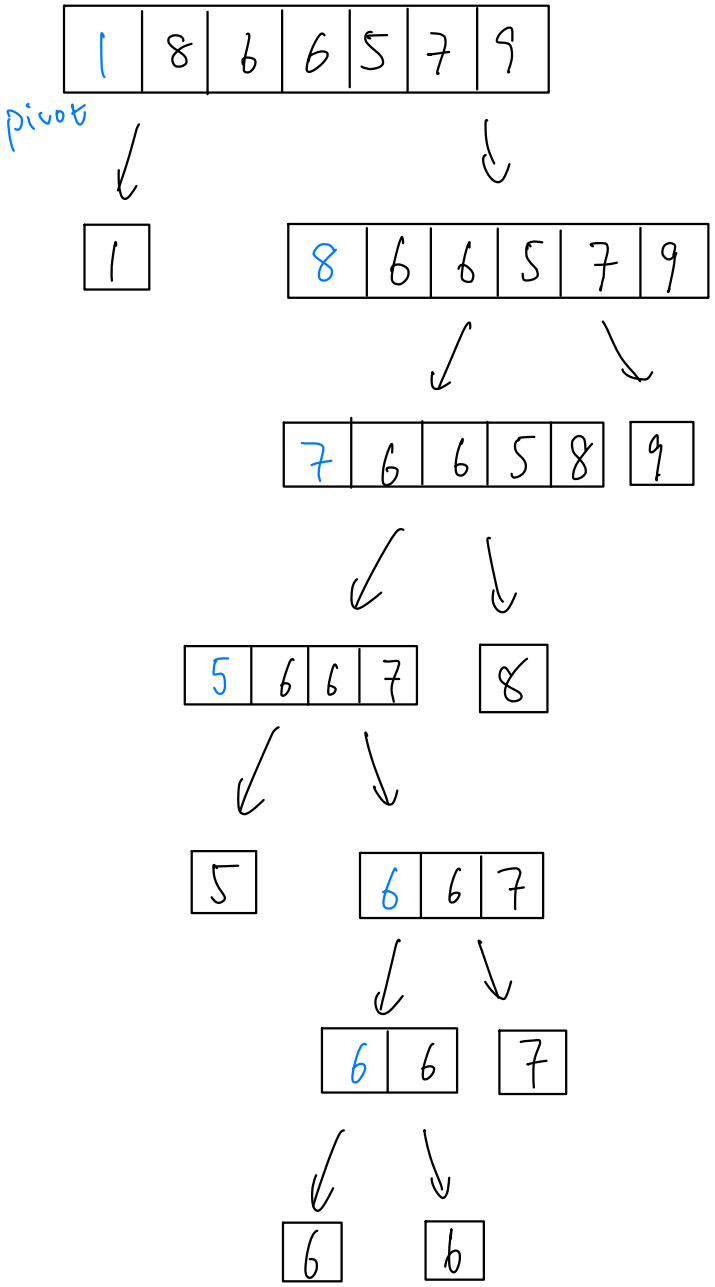
Specifically, for Q8a, your symtab.c implementation will be tested with a correct main.c implementation and predefined symtab.h header file. The setting is similar for Q8b.

For each sub-question, we use gcc compiler to compile your code and test your code with 6 test cases. Each test case values 2 pts (in each sub-question). You can get full (2) marks for a

test case only if your program can generate an output which is identical with the standard answer in run mode. Please be noted that:
1) The order of grade output is not important.
2) If the program can generate desired solution in debug mode only, marks will not be given.
3) The successful execution of your code should not depend on specific operating systems and/or IDEs.
4) Partial marks will be given if the code generates very similar output compared with the standard answer. (e.g. only one calculation of grade is wrong).

CSCI 2100C Assignment 2 Figure drawing problems solution.

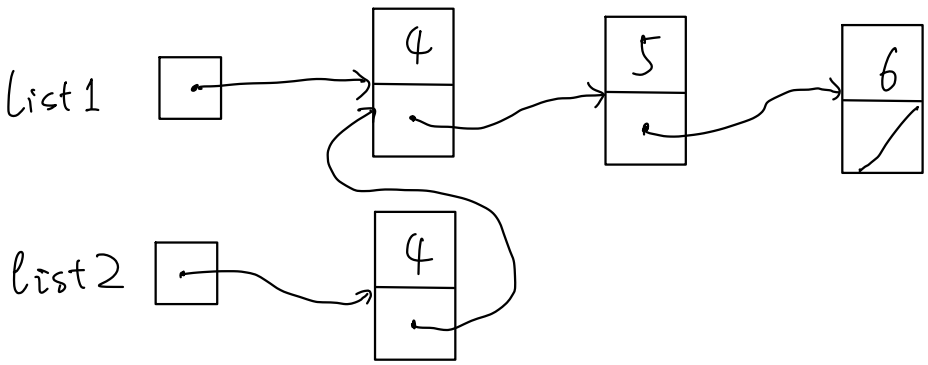
Q5.



⇒ The sorted array: 1 5 6 6 7 8 9

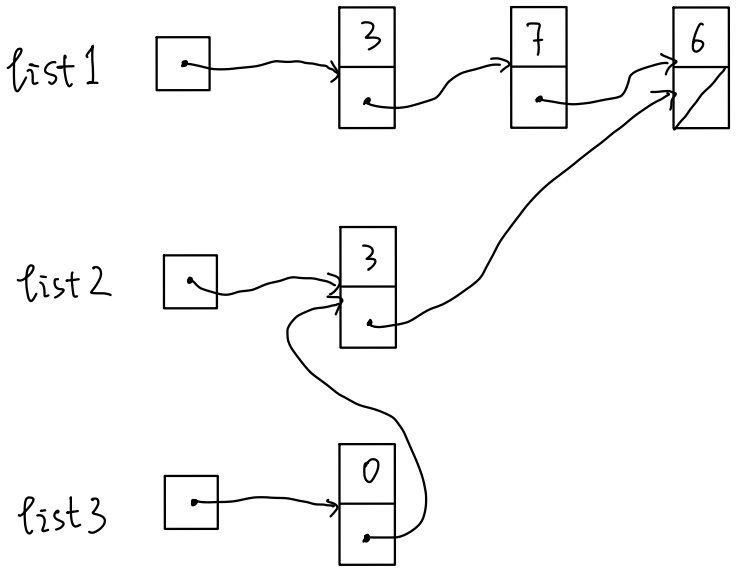
Q7

a)

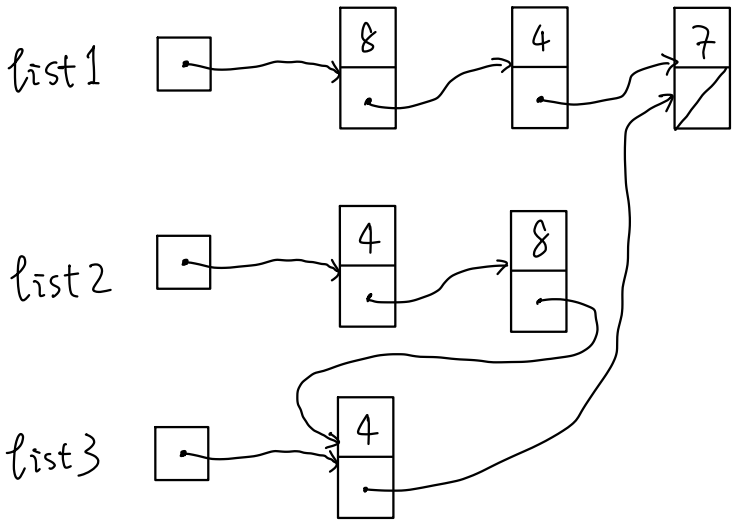


Q7

b)



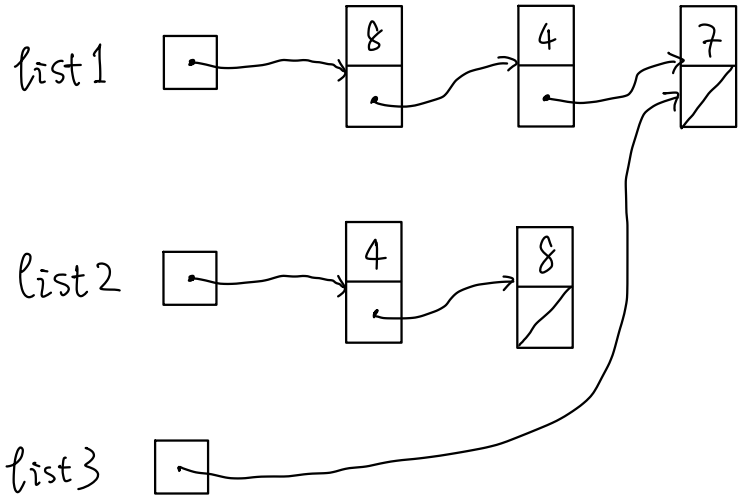
c)



firstElement = 8

secondElement = 4

d)

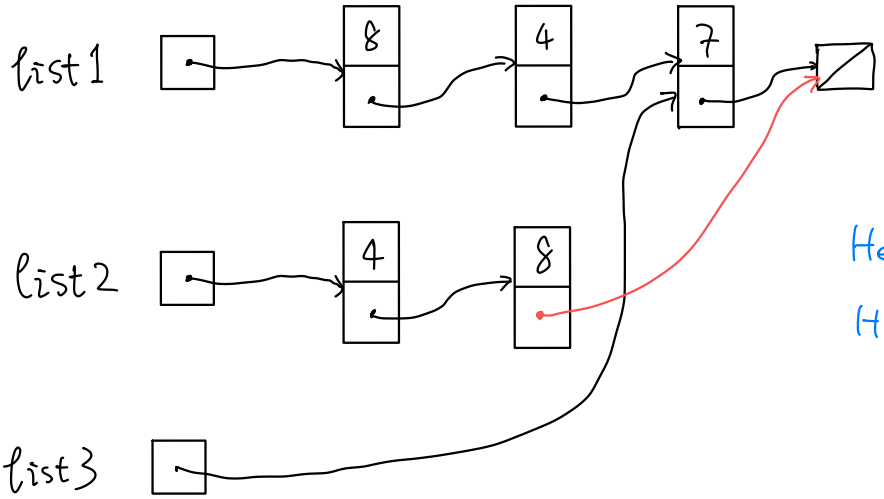


Head (Tail (list 1)) = 4

Head (list 1) = 8

The following solution is also appropriate.

d)



Head (Tail (list 1)) = 4

Head (list 1) = 8