CS 2123 Data Structures

Spring 2016 – Midterm 2 -- March 24, 2016
You have 75 min. Good luck.
You can use the 2-page C reference card posted in the class web page.

Name:……………key………… Score: ……./100

1. (20 pt) Review Questions
   a. (4pt) What will be printed out when the codes in the left and right columns are executed?

`int *p, *q, arr[4]={5,8,3,7};`
`p = q = arr;`
`*p++ = 8;`
`if (*p == *q)`
`   printf("AAA %d", arr[0]);`
`else`
`   printf("BBB %d", arr[0]);`
	`int *p, *q, arr[4]={5,8,3,7};`
	`p = q = arr;`
	`*++p = 5;`
	`if (p == q)`
	`   printf("AAA %d", arr[1]);`
	`else`
	`   printf("BBB %d", arr[1]);`

Output: AAA 8	Output: BBB 5

   b. (4pt) Suppose one of your friends is struggling with the following code segment because it compiles OK, but gives a segmentation fault when executed. What is the problem there and how would you fix it?

The problem is: a is declared as a char pointer but it is currently not pointing any dynamically allocated memory. We have to first allocate enough memory space.

`char *a;`
 	
`strcpy(a, "CS 2123");`
	`/* fixed version */`
	`char *a;`
	
	`a=(char *) malloc(8);`
	`if(a==NULL) exit -1;`
	
	`strcpy(a, "CS 2123");`

c. (4pt) Write a function to check if the given **b**eginning and **e**nding HTML tags do match or not. The format for **b**eginning and **e**nding tags will be given as **"<tag_name other attributes>"** and **"</tag_name>"** respectively. If **tag_name** part matches, it returns 1; otherwise it returns 0.

```
int tag_match(char *b, char *e)
{
        b++;
        e+=2;
        while(*b && *e && *b==*e) { b++; e++; }
        if (*b==' ' && *e=='>') return 1;
        else return 0;
 }
```

d. (4pt) In the following loops, first count the number of operations (i.e., find how many lines will be printed out) when N is 32. Then give the computational complexity using big-O notation for any value of N. (justify your answers)

int i, j, N=32;	Number of lines printed when N=32	big-O notation

`for(i=1; i <= N; i = i + 1)`
`  for(j=1; j <= N; j++)`
`     printf(" line2\n");`
	32*32=1024	$O(N^2)$

`for(i=1; i <= N; i = i * 2)`
`  for(j=1; j <= N; j++)`
`     printf(" line2\n");`
	5*32=160	$O(N \log_2 N)$

e. (4pt) Suppose we want to compute the sum of the integer values in a given array using the following function prototype: `int arr_sum( int arr[], int n );` This function can easily be implemented as an iterative function. **However**, you are asked to think about a recursive strategy and implement `it` as **a recursive function**.

```
int arr_sum( int arr[], int n )
{                        // (No credit will be given for an iterative implementation!)
      if ( n <= 0 )   // base case
          return 0;
       else
          return arr_sum( arr, n - 1 ) +
                 arr[ n - 1 ];
      }
```

2. (20pt) A color image is a 2D array of pixels where each pixel is represented by three integers showing Red, Green, Blue (RGB) components. Suppose we store a color image in a text file using a very simple format as follows: First two integers in the file represent the numbers of rows and columns. Then the file contains that many rows and columns of pixels, where each pixel has three integers to represent RGB components. For example a 3x4 color image would be saved in a file as follows:

3	4										
1	**2**	**3**	1	2	3	**1**	**2**	**3**	1	2	3
3	**6**	**5**	3	34	43	**51**	**43**	**5**	4	8	4
6	**45**	**34**	53	5	4	**8**	**4**	**66**	86	12	43

Complete the following program which takes the file name from the command line and reads the pixel values into a dynamically create 2D array of **`pixelT`** structure defined below. So we can access the colors of each pixel using `img[i][j].r, img[i][j].g, img[i][j].b`

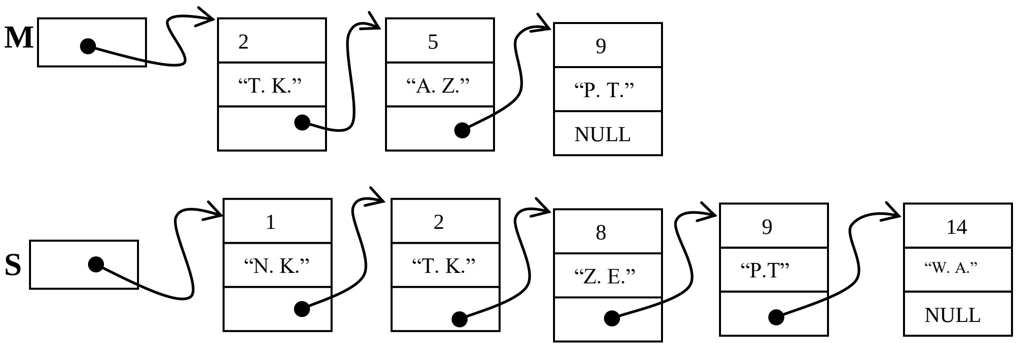
```
/* suppose all standard libraries and our book libs are  included here  */
typedef struct pixel {
    int r, g, b;
} pixelT;

int main(int argc, int *argv[])
{
  FILE *fp;
  int row, col, i, j;
  pixelT **img;


  if (argc<2) {
     printf("Usage: prog filename\n");
      exit(-1);
  }


  if((fp=fopen(argv[1], "r"))==NULL){
     printf("File cannot be opened\n");
    exit(-1);
  }


 fscanf(fp,"%d %d", &row, &col);
```

```
  /* dynamically create 2D array of pixelT */
  img = (pixelT **) malloc(row*sizeof(pixelT *));
  if(img==NULL) ){
    printf("Memory cannot be allocated \n");
   fclose(fp);  exit(-1);
  }
  for(i=0; i < row; i++) {
   img[i] = (pixelT *) malloc(col*sizeof(pixelT));
   if(img[i]==NULL) ){
     for(j=0; j < i; j++) free(img[j]);
     printf("Memory cannot be allocated \n");
     fclose(fp);   exit(-1);
   }
  }




  /* read the RGB values of each pixel into the allocated memory */
  for(i=0; i < row; i++) {
   for(j=0; j < col; j++) {
      fscanf(fp,"%d %d %d",
             &img[i][j].r, &img[i][j].g, &img[i][j].b);
   }
  }



  /* Free the dynamically allocated memory */
  for(i=0; i < row; i++) free(img[i]);
  free(img);
  fclose(fp);
}
```

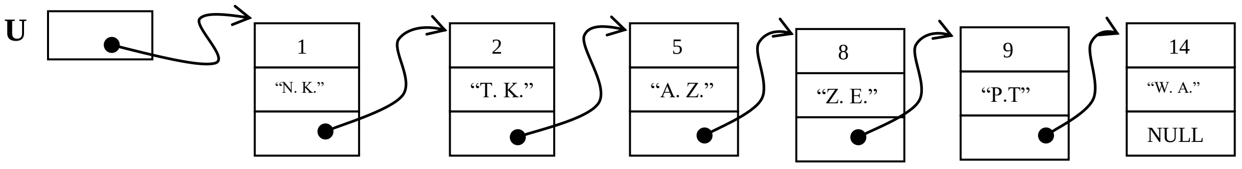
3. (20pt) We use the following cell structure to store the IDs and names of some students in a **single linked list** data structure

```
typedef struct cell  {
   int ID;
   char name[20];
   struct cell  *next;
}  cellT, *cellPtrT;
```

Suppose somehow we have created the following two single linked lists, namely M and S, to store the IDs and names of the students who are taking Math and Science, respectively. Lists are **sorted** w.r.t. IDs.



Now we are interested in creating a new list, say U, to store the IDs and names of the students who are taking Math **or** Science (or both). As you may know this operation is known as **union** of two sets, where each element appears only once in the new list U. So list U will look like as follows (note that elements in list U should be sorted w.r.t. IDs, as in the original lists):



You are asked to implement a function, `cellT *union(cellT *M, cellT *S)`, which can be called as follows to find the union of the given two sets represented by single linked list.

```
cellT *M, *S, *U;

/* somehow the lists pointed by M and S are created */

U = union(M, S);
```

After your function, M and S should be intact. So do not remove the cells from M or S! When needed, create new cells for U and copy the ID and name from the cells in M or S.

use the next page to answer question 3.

Name:………. Answer question 3:

```
cellT *union(cellT *M, cellT *S)
{
     cellT *U=NULL, *target, *tmp, *end;
     while(M || S){
          tmp = (cellT *) malloc(sizeof(cellT));
          if (tmp==NULL) {
              printf("memory cannot be allocated\n");
              exit(-1);
          }
          tmp->next = NULL;
          if (U==NULL)
              U=tmp;
          else
              end->next = tmp;
          end = tmp;

          if (M && S && M->ID == S->ID)) {
              target = M;  // they are the same take one …
              M = M->next;
              S = S->next;
          } else if (S==NULL || (M && M->ID < S->ID)) {
              target=M;
              M = M->next;
          } else {
              target=S;
              S = S->next;
          }
          tmp->ID = target->ID;
          strcpy(tmp->name, target->name);
               //   ! tmp->name = target->name is wrong !

     }
     return U;

}
```

4. (20pt) Recall the list ADT whose interface is extended as below. Mainly we include the following prototype into `list.h`: `listADT` **`CopyReverseList(list`**`ADT a);` which creates **another** copy of a given list but in REVERSE order, and returns the pointer to new list. Old list should be intact. Implement it based on the structures given below.

```
#ifndef _list_h_
#define _list_h_

typedef struct listCDT *listADT;

listADT NewList();
void list_insert_sorted(listADT a, int val);
void list_insert_unsorted(listADT a, int val); // add val to the end
listADT CopyReverseList(listADT a);
#endif
```

```
/* list.c
   using linked list */

#include “list.h”
/* suppose standard and book
libraries are included too */

#include "list.h"

typedef struct point {
   int x;
   struct point *next;
} myDataT;

struct listCDT {
      myDataT *start;
      myDataT *end;
};

listADT NewList()
{
      listADT tmp;
      tmp = New(listADT);
      tmp->start = NULL;
      tmp->end = NULL;
      return(tmp);
}

/* implementations of  other functions */
```
	```
	listADT CopyReverseList(listADT a)
	{
	    listADT r;
	    myDataT *cp, *tmp;

	    r = NewList();

	    cp = a->start;
	    while(cp){
	        tmp = New(myDataT *);
	        tmp->x = cp->x;

	        tmp->next = r->start;
	        r->start == tmp;

	        if (cp == a->start)
	                r->end = tmp;

	        cp = cp->next;
	    }
	  return r;
	}
	```

5. (20pt) Recall the buffer ADT which had the same interface `buffer.h` but three different implementations (i.e., list, array, stack). Suppose we added the following function prototype to `buffer.h`: `void RemoveCharacter(bufferADT buffer, char x);` which removes all the occurrences of a given character x from the buffer after the cursor. It does not change the cursor position. You are asked to implement this function under **list** representation, where we used a dummy cell so cursor were pointing to the cell immediately before the logical insertion point.

   Example: suppose buffer has **`a b c e d | e f g e e h`**. Here the cursor is pointing to the cell containing **`'d'`**

   After we call `RemoveCharacter(buffer, 'e');`

   The buffer should have **`a b c e d | f g h`**

/* listbuf.c */

```
#include "buffer.h"

typedef struct cellT {
    char ch;
    struct cellT *link;
} cellT;

struct bufferCDT {
    cellT *start;
    cellT *cursor;
};

bufferADT NewBuffer(void)
{
 bufferADT buffer;
 buffer = New(bufferADT);
 buffer->start=buffer->cursor
             = New(cellT *);
 buffer->start->link = NULL;
 return (buffer);
}

/* implementations of  other functions */
```

```
void RemoveCharacter(bufferADT buffer, char x)
{  cellT *curr, *prev;

   prev = buffer->cursor;
   curr = buffer->cursor->link;

   while(curr){
     if (curr->ch == x){
        prev->link=curr->link;
        free(curr);
        curr = prev->link;
     } else {
         prev = curr;
         curr = curr->link;
     }
}
```