built on 2022/06/13 at 08:48:00 ***due:*** *Tuesday, June 28th @ 11:59pm*

Be sure to read this problem set thoroughly, especially the sections related to collaboration and the hand-in procedure.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Hand-in Instructions

To submit this assignment, please follow the steps below:

1. Make sure your programs compile and run on Syskill.
2. Check for any memory leak using `valgrind --tool=memcheck --leak-check=full ./box < test.in`
3. Zip up all `.c` files and name the zip file `a3.zip` i.e.

```
> zip a3.zip box.c queue.c bubblesort.c det.c
```

4. Find out the MD5 hash of your zip file. You will need to submit this code on Canvas. We use it for keeping track of your submission time. You may resubmit your work but the MD5 hash has to match.

```
> md5sum a3.zip
```

5. Copy the zip file to the directory `/handin/a3/u6312345` where `u6312345` is your student ID.

```
> cp a3.zip /handin/a3/u6312345
```

6. Log on to Canvas, go to assignment 3 submission page and enter the MD5 hash.

Task 1: Magic Box (10 points)

In this problem, you will work on a data structure called (Magic) Box. This Box is magical because it can hold as many items as you want! (well, not really but as long as memory permits)

Your Task: Complete the implement of the following functions in `box.c`,

- `void createBox(Box **b, int init_cap)` – Creates a new box with capacity of `init_cap`. If the box `b` has already been allocated, this function does nothing.
- `void insert(Box *b, int elem)` – Inserts an element into a given box. If the box is full, you must fist increase its capacity by **doubling the capacity** with `realloc` and then add the element.

- `void removeAll(Box *b, int elem)` – Removes all occurances of an element from a box. After removal, the elements in the box must be condensed i.e. no gap between any two elements. Also, this function must not change the capacity of the box.
- `void printBox(Box *b)` – Prints out elements of a box, one element per line.
- `double getMean(Box *b)` – Returns the mean of data in a box.
- `void dealloc(Box **b)` – Deallocates the box along with its data.

Expected Behavior: After you have finish implementing `box.c`, you can check your own work by:

```
> ./box < test.in
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) and the `main` function will be ignored in the grading environment. So you should not need to and must not modify the header files and the `main` function! Your program should not leak any memory.

Task 2: Queue of Words (10 points)

Queue (or a FIFO) is another basic data structure that is very common in low-level system implementations i.e. system libraries and OS kernels. You will work on a simple implementation of a dynamic queue where each element is a string. The underlying implmentation of your queue will be similar to a listed list.

Your Task: Complete the implement of the following functions in `queue.c`,

- `void push(Queue **q, char *word)` – Pushes a string `word` to the back of a queue `q`. Instead of keeping the pointer to the array, you must instead keep a **COPY** of the word inside the queue. Also, if `q` hasn't been allocated i.e. `q` is NULL, you must allocate space for it here as well. Also, please note that the input is a double pointer as the address to your queue can change in the case `q` has not been allocated.
- `char *pop(Queue *q)` – Returns the pointer to the string at the front of the queue and remove it from the queue. The caller of `pop` is responsible for freeing the returned pointer. When `q` is empty, this function returns NULL.
- `int isEmpty(Queue *q)` – Returen `true` if the queue is empty and `false` otherwise.
- `void print(Queue *q)` – Prints out the elements of a give queue `q`, front to back. If `q` is empty, prints out `No items`.
- `void delete(Queue *q)` – Deallocates the queue as well as all items in it.

Expected Behavior: After you have finish implementing `queue.c`, you can check your own work by running:

```
> ./queue
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) and the `main` function will be ignored in the grading environment. So you should not need to and must not modify the header files and the `main` function! Your program should not leak any memory.

Task 3: Bubblesort (10 points)

In this task, you will be implementing a simple sorting algoithm *Bubble Sort* in C. To keep this things simple, we have provided you with the pseudo code; you only need to translate it to C.

Let `Entry` be a C-struct with the following fields: data (`int`), name (`char *`). We have already defined them for you in `bubblesort.h`. Entries are sorted by using `data` as the key. Your task is to finish the implementation in sort() that also takes in a function pointer representing our comparator function: `sort` and also the main function `main`.

- `sort` – Sort the data based on the input with size n

```
// input is the input data you want to sort,
// n is the size and
// comparator is the comparator function used by your implementation of bubblesort
void sort(Entry *input, int n, void* comparator) {
    # Traverse through all array elements
    for i from 0 to n:
        # Last i elements are already in place
        for j from 0 to (n-i-1):
            # Swap if the element found is greater
            # than the next element
            if comparator(input[j], input[j+1]) > 0:
                swap input[j] and input[j+1]
}
```

- `main` – The "main" function

```
main() {
         Read the number of items from the STDIN.
        Allocate an array for the input.
        Read data from STDIN and store them the array.
        Create a comparator function that would allow you to sort our data from smallest to largest.
        Call sort() on the array.
        Print out the entries.
        Free all the memory that you have allocated.
}
```

Input & Output Your program should read input from the **standard input**. The format of the input is as follows:

```
<number of items>
<data_1> <name_1>
<data_2> <name_2>
...
<data_n> <name_n>
```

where `data_i` is an `int` and `name_i` is a string of length strictly smaller than `MAX_NAME_LENGTH` which is defined in `mergesort.h`. Here is an example input:

```
4
4 aa
7 bb
2 cc
1 dd
```

After you are done, you can check your own work by running:

```
> ./bubblesort < test.in
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) will be ignored in the grading environment. So you should not need to and must not modify the header files! For this problem, the `main` function will not be replaced. Your program should not leak any memory.

Task 4: Determinant (10 points)

There are several ways to compute the determinant of an $n \times n$ matrix. In this task, we ask you to implement a simple method that considers elements from the top row and their respective minors. The determinant of a matrix A, $\det(A)$ is given by an alternate sum of the products between each element of the top row with its minor.

$$det(A) = \begin{vmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{vmatrix} = a \begin{vmatrix} f & g & h \\ j & k & l \\ n & o & p \end{vmatrix} - b \begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix} + c \begin{vmatrix} e & f & h \\ i & j & l \\ m & n & p \end{vmatrix} - d \begin{vmatrix} e & f & g \\ i & j & k \\ m & n & o \end{vmatrix}$$

As you notice, this is a recursive definition that can be easily implemented with recursion. You can use the following fact to define the base case.

For any 1×1 matrix $A = [a]$,

$$det(A) = \begin{vmatrix} a \end{vmatrix} = a$$

Input & Output Your program should read input from the **standard input**. The first line of the input is n and each of the next n lines contain n integers a_{ij} ($-1000 \leq a_{ij} \leq 1000$).

The output of your program should be the value of the determinant accurate to 5 decimal places.

Input:

```
4
1 2 3 4
5 2 3 1
2 5 0 1
3 2 4 2
```

Output:

```
-105.00000
```

Important: Your program should not leak any memory.