CMSC 341 Spring 2017
Homework #2
Due: Thursday, Feb. 16, 8:59:59pm

Instructions: Save a copy of this Google Doc. Type in your answers. Save your document as PDF. Copy the PDF file to GL. Move the PDF file to the `hw2` directory of your shared submission directories.

Further instructions: When you are asked to provide an "algorithm" in the questions below, you must do two things. First, provide a *brief* English explanation of your *strategy.* This should be a high level explanation that does not involve any variable names. Second, you must provide a C++ code fragment for your algorithm.

For example, if the question asks you to provide an algorithm that counts the number of items in a linked list, your answer should be something like:

> Send a pointer down the linked list until the `NULL` pointer has been reached. Each time a non-`NULL` node is encountered, add one to a counter.
>
> ```
> int counter=0 ;
> Node *ptr=head ;
>
> while (ptr != NULL) {
>    counter++ ;
>    ptr = ptr->next ;
> }
> ```

Question #1:
Log into your Piazza account and post a new thread in the `hw2` folder with your favorite movie quote. The link for the Piazza site for this class is: https://piazza.com/class/iywlcnq7e5q71f
You should have received an email invitation to join Piazza.

Enter the quote here, as well:

Question #2: (R-3.7, page 149 of the textbook)

Give an algorithm for finding the penultimate (second to last) node in a singly-linked list where the last element is indicated by a null next link. You may assume the list has at least two items.

You should use the following definition of the `Node` class:

```
class Node {
public:
   int m_data ;
   Node* next ;
} ;
```

Furthermore, assume that a `Node` pointer `head` points to the first node of the linked list and that the linked list does not use a dummy header.

Don't forget to include your brief English description.

English Description:
Send a pointer down a linked list. Instead of cutting it off at a NULL pointer, the condition for the while loop checks for the next pointers pointer to be equal to NULL. When it is, it stops.

Code Fragment:

```
Node* currentPtr = head;
while (currentPtr->next != NULL) {
        currentPtr = currentPtr->next;
 }
return currentPtr;
```

Question #3: (adapted from R-3.10, page 149 of the textbook)
Describe a nonrecursive function for finding, by link hopping, the middle node of a singly-linked list. You may not use any counters. (*Hint:* use two pointers.)

Use the `Node` class from the previous question. Here's how to handle even versus odd. If the linked list has 9 items, your function should return a pointer to the fifth node. If the linked list has 12 items, your function should return a pointer to the sixth node. If the list is empty, just return `NULL`.

Don't forget to include your brief English description.

English Description:
Iterate through the linked list using 2 different pointers. They are both set to head, but one increments at twice the rate that the other pointer does. The while ends when the fast incrementing pointer hits NULL or is about to hit NULL.

Code Fragment:

```
Node* currentPtr1 = head;
Node* currentPtr2 = head;
while (currentPtr1 != NULL && currentPtr1->next != NULL) {
        currentPtr1 = currentPtr1->next->next;
        currentPtr2 = currentPtr2->next;
        if (currentPtr1 == NULL)
                currentPtr2 = currentPtr2->next;
}
return currentPtr2;
```

Question #4: (adapted from R-3.14, page 150 of the textbook)

Write a short *recursive* C++ function that prints out a random permutation of an array of integers `A[]`. Your function should repeatedly "remove" a random entry from the array, print it out and make a recursive call. (Yes, this alters the array permanently.) Assume that you have access to a function `random(k)`, which returns a random integer between `0` and `k` (inclusive). Use the function prototype:

```
void randPerm(int A[], int size) ;
```

Don't forget to include your brief English description.

English Description:
Uses the random function to find a random entry in the initial array list. It then checks for the size being 1, and if it isn't it prints out the entry from the array it chose. It then swaps the random entry and the last element so it can set the last element to NULL and not use it again. It then uses recursion to repeat the process until the base case is satisfied and it can print out the last remaining element and end the function.

Code Fragment:

```
void randPerm(int A[], int size) {
        int randomEntry = 0;
        randomEntry = random(size-1);
        if (size != 1) {
                cout << A[randomEntry];
                A[randomEntry] = A[size-1];
                A[size-1] = NULL;
                size -= 1;
                randPerm(A[], size);
}
        else {
                cout << A[size-1] << endl;
}
```

}

Question #5: (adapted from C-3.18, page 151 of the textbook)
Describe a recursive algorithm that will rearrange an array of `int` values so that all even values appear before all the odd values. Your algorithm should run in *O(n)* time for arrays with *n* items. To achieve this linear running time, your recursive function should only "handle" each item in the array once and spend only constant time per item.

Use the function prototype:

```
void rearrange(int A[], int start, int end) ;
```

The function should be initially called with `rearrange(A,0,n-1)` where `A[]` has `n` items.

Don't forget to include your brief English description.

English Description:
The function first checks the at the initial start value. It checks it for an even or odd number. If it hits an odd number, it'll go through and see if it can find an even to swap with later down the list. It makes sure to check if it's the last one in the list so that the time is constant for every swap. If it's even, it goes through and heads to the next element. When the final value is reached, the base case is satisfied and the function finishes.

Code Fragment:

```
void rearrange(int A[], int start, int end) {
        int arrayEntry = 0;
        int evenEntry = 0;
        int i = start;
        arrayEntry = A[start];
        if (arrayEntry % 2 == 1) {
                while (evenEntry % 2 != 0 || i != end) {
                        evenEntry = A[i];
```

```
                    i++;
                }
                A[start] = A[i];
                start += 1;
                rearrange(A[], start, end);
        }
        else {
                if (start != end) {
                        start += 1;
                        rearrange(A[], start, end);
        }
        }
}
```