Section #5

Ashley Taylor and Shreya Shankar (Based on handouts by previous CS106B instructors and TAs)

Pointers and Linked Lists

1. Pointer Trace

Given the following structs:

```
typedef struct Quidditch {
    int quaffle;
    int *snitch;
    int bludger[2];
} Quidditch;
```

```
typedef struct Hogwarts {
    int wizard;
    Quidditch harry;
    Quidditch *potter;
} Hogwarts;
```

Draw the memory diagram for the following pointer trace.

```
Quidditch * hufflepuff(Hogwarts * cedric) {
    Quidditch *seeker = &(cedric->harry);
    seeker->snitch = new int;
    *(seeker->snitch) = 2;
    cedric = new Hogwarts;
    cedric->harry.quaffle = 6;
    cedric->potter = seeker;
    cedric->potter->quaffle = 8;
    cedric->potter->snitch =
        &(cedric->potter->bludger[1]);
    seeker->bludger[0] = 4;
    return seeker;
}
```

```
void gryffindor() {
    Hogwarts *triwizard = new Hogwarts[3];
    triwizard[1].wizard = 3;
    triwizard[1].potter = NULL;
    triwizard[0] = triwizard[1];
    tri
    wizard[2].potter =
    hufflepuff(triwizard);
    triwizard[2].potter->quaffle = 4;
}
```

2. Mirror (Arrays)

Write a member function named `mirror` that could be added to the `ArrayIntList` class. Your function should double the size of the list of integers by appending the mirror image of the original sequence to the end of the list. The mirror image is the same sequence of values in reverse order. For example, suppose a variable named list stores the following values: $\{1, 3, 2, 7\}$. If we make the call of `list.mirror();` then it should store the following values after the call: $\{1, 3, 2, 7, 7, 2, 3, 1\}$. The list has been doubled in size by having the original sequence appearing in reverse order at the end of the list. If the list is empty, it should also be empty after the call. Assume that you are adding this method to the `arrayIntList` class as defined below:

```
class ArrayIntList {
private:
    int* elements;
    int mysize;
    int capacity;
```

```
public:
    ...
};
```

3. Sorted (Linked Lists)

Write a member function named `isSorted` that accepts a pointer to a `ListNode` representing the front of a linked list. Your function should return true if the list is in sorted (non-decreasing) order and false otherwise. An empty list is considered to be sorted. Assume that you are using the `ListNode` structure as defined below:

```
struct ListNode {
    int data; // value stored in each node
    ListNode* next; // pointer to next node in list (nullptr if none)
}
```

4. Stutter (Linked Lists)

Write a function named stutter that accepts as a parameter a reference to a pointer to a `ListNode` representing the front of a linked list. Your function should double the size of a list by replacing every integer with two consecutive occurrences of that integer. For example, if a variable named front points to the front of a list containing $\{1, 8, 19, 4, 17\}$, after a call of `stutter(front);`, the list should store $\{1, 1, 8, 8, 19, 19, 4, 4, 17, 17\}$. Assume that you are using the `ListNode` structure as defined below:

```
struct ListNode {
    int data; // value stored in each node
    ListNode* next; // pointer to next node in list (nullptr if none)
}
```

5. Braid (Linked Lists)

Write a function named `braid` that accepts as a parameter a reference to a pointer to a `ListNode` representing the front of a linked list. Your function should interleave the reverse of the list into the original, with an element from the reversed list appearing after each element of the original list. For example, if a variable named front points to the front of a chain containing $\{10, 20, 30, 40\}$, then after a call of `braid(front);`, it should store $\{10, 40, 20, 30, 30, 20, 40, 10\}$. Assume that you are using the `ListNode` structure as defined below:

```
struct ListNode {
    int data; // value stored in each node
    ListNode* next; // pointer to next node in list (nullptr if none)
}
```