Prelab 9: Week of April 6th

For this prelab you are to implement six functions for a List ADT:

```
/* This function returns an initialized List variable.
   It must be called before using subsequent functions.
   Parameter is reference to an integer error code. */
List initList(int*)

/* This function inserts the object at the head of
   a given list. Returns an error code of 0 if the
   insertion is successful, otherwise it returns 1. */
int insertHead(void*, List)

/* This function removes and returns the object at the
   head of a list. */
void * removeHead(List)

/* This function inserts the object at the tail of
   a given list. Returns an error code of 0 if the
   insertion is successful, otherwise it returns 1. */
int insertTail(void*, List)

/* This function removes and returns the object at the
   tail of a list. */
void * removeTail(List)

/* This function unitializes a list and frees all
   memory allocated for it by the above functions. */
void freeList(List)
```

You're free to implement these functions however you please, but they all must take only O(1) time. This can easily be achieved using *doubly-linke*d lists. However, that still leaves a lot of implementation options. Would a dummy node simplify things? How about two dummy nodes? What about making the list circular? Or how about circular with a dummy node? Can you implement your functions without having to ever test for special cases (other than malloc failure)?

This is the first time you've been given a specification that includes a performance constraint. From now on your documentation – *both for prelabs and labs* – should include the computational complexity for every function. That's easy for this prelab because all functions are *required* to have O(1) run-time complexity.

- JKU

BONUS TASK:

Sure, some people may choose to spend spring break increasing their risk of skin cancer, liver damage, and STDs, but if you're not one of those then you might have more fun extending your prelab functions with something very different from anything we've discussed thus far: *iterators*. An iterator is a mechanism that allows the user to traverse a list (or other collection ADT). Why is that problematic? Answer: Because we can't allow the user to know or exploit anything about our implementation.

Think about it, the user can't simply set a pointer to a node and then advance it via the *next* pointer of that node because s/he doesn't know anything about our node struct or its members. In fact, the user doesn't know whether we've implemented our list ADT using an array, a linked list, a circular-linked list, a doubly-linked list, or whatever. The only thing the user can do is declare variables of the data types we provide (e.g., List) and call the interface functions we provide.

Here's what we might like for the user to be able to do:

```
List head;
ListIterator p;
Employee *emp;

head = initList();
(user inserts Employee objects into the list)

for (p=listStart(head); notEndOfList(p); p=listNext(p)) {
       emp = getListObject(p);
       (user does something with the returned employee object)
}
```

Notice that there are four new interface functions along with a new data type *ListIterator*. Can you figure out how to make something like this work? I bet you can. Have an enjoyable spring break!!!!

- JKU

Bonus Bonus Task:

Whoa, it's one thing to decide not to spend your spring break licking psychedelic toads while listening to recordings of wombat mating calls, but it's important to recognize the value of setting aside some time for fun and relaxation. What? You say that there's nothing more fun and relaxing than programming? Well then, let me help you out!

If you think about it, we can create a family of list-related functions that allow the user to, e.g., insert a new object before or after the iterator location, or delete the object associated with the iterator, or whatever else the user might want to do when traversing a list. For example, the following could be used to remove a bad employee from a list of employees:

```
for (p=listStart(head); notEndOfList(p); p=listNext(p)) {
      if (getListObj(p) == badEmployee)
         deleteIt(p); // delete bad employee from list
}
```

If you take a shot at designing and implementing some of these functions you'll find that everything is pretty straightforward because our underlying implementation uses doubly-linked lists.

- JKU

Bonus Bonus Bonus Task:

C'mon, you can't spend every waking minute programming. It's too much. Contact your TA and s/he can set you up with a toad and an mp3 of wombat mating calls. But steer clear of those Peruvian magic toads. Trust me.

- JKU