CST238 Fall 2019

Lab 10 - Basic Linked List

Part 1. (20 points)

1. Fork the Simple Linked List starting code
2. Implement the following functions, in order:
   `LinkedList::empty()`
   `Node::hasNext()`
   `LinkedList::append(int data)`
   `LinkedList::display()`
   `LinkedList::search()`
3. When done, submit ONLY the LinkedList.cpp file

When completed correctly, you should be able to run the code and see the following output:

```
Empty!
1 2 3
Is 2 in the linked list? true
Is 3 in the linked list? true
Is 5 in the linked list? false
```

There isn't much code to write, so this must be an easy lab, right?

Think again.

The toughest part is that we're dealing with pointers, not objects. You will have to remind yourself repeatedly how to access member functions and variables. If I have the following class

```
class Foo {
public:
    Foo(double bar) { myBar = bar; };
    void setBar(double bar) { myBar = bar; };
    double getBar() const { return myBar; };
private:
    double myBar;
};
```

And I want to or need to use pointers, I need to remember a few things:

1. I can't create new objects the old way:
   `Foo * f;`
   This creates the pointer, but no memory is assigned! I need to do both:
   `Foo * f = new Foo(2.5);`
   Because I'm using parentheses, not brackets, I'm creating one object, not an array
2. Because I have a pointer, I have to dereference the pointer before calling a member function or variable.
   Start with an pointer:
   `Foo * f = new Foo(2.5);`

Now, to call getBar, I can either dereference the pointer, then call getBar with the dot operator (using parentheses to get the order right):

```
cout << (*f).getBar << endl;
```

or I can do both operations together, with the arrow (dash, greater-than sign) operator

```
cout << f->getBar << endl;
```

3. If I have a reference and want a pointer, I can use the reference operator (the ampersand):

```
Foo f(2.5);
Foo * fPtr = &f;
```

4. If I have a pointer and want a reference, I can use the dereference operator (asterisk, aka star):

```
Foo * f = new Foo(2.5);
Foo p = *f;
cout << p.getBar() << endl;
```

5. To compare two objects to make sure they are the same object (not just the same values), I need pointers:

```
Foo * g = new Foo(3.4);
Foo * h = g;
Foo * k = new Foo(3.4);
if (g == h) {
    cout << "g and h are the same!" << endl;
} else {
    cout << "g and h are the not same!" << endl;
}

if (g == k) {
    cout << "g and k are the same!" << endl;
} else {
    cout << "g and k are the not same!" << endl;
}
```

6. References are not comparable at all; this won't compile

```
Foo m(2.5);
Foo o(m);
if (o == m) {
    cout << "o and m are the same!" << endl;
} else {
    cout << "o and m are the not same!" << endl;
}
```

As a final hint, remember that there are three kinds of loops: for loops, while loops, and do-while loops.

NOTE: You do NOT need to use recursion here.

Part 2. (10 point)

1. Implement the power function as discussed in class *using recursion*. Not using recursion will result in a zero. The power function takes in two inputs, x and n, and returns xn, so that it works as below. In this case, the function should NOT print, just calculate.

   ```
   Input x: 3
   Input n: 3
   3 to the power of 3 is 81
   ```

   or

   ```
   Input x: 4
   Input n: 2
   4 to the power of 2 is 16
   ```

2. Implement a slightly modified version of the sumArray function discussed in class *using recursion*. Not using recursion will result in a zero. The sumArray function should print out each element as it adds (either direction is fine). In this case, the function SHOULD print the values, and return the sum to be printed by main.

   If the input array is 2, 5, 7, 12:

   ```
   12 + 7 + 5 + 2 = 26
   ```

   If the input array has no elements:

   ```
   0
   ```

   If the input array has one element, 5:

   ```
   5 = 5
   ```