MP 3 Images and Lists

Extra credit: Friday, September 25 at 11:59 PM
Due: Friday, October 2 at 11:59 PM
Doxygen for MP 3

Direct links to MP 3.1 and MP 3.2

Solo MP

This MP will be a solo MP. You are required to complete the MP without working with any other students. This is just like the previous MPs, except that your "group" consists of only you. Your `partners.txt` file must include your and only your NetID.

Don't worry! Not all the MPs will be like this. But we on course staff feel having a few solo MPs will help ensure everyone learns the material themselves. Good luck!

You are welcome to get help on the MP from course staff, via open lab hours, or Piazza!!

Goals

In this MP (machine problem) you will:

- learn to manipulate linked memory by writing functions to modify linked lists
- practice using templates

Also note that you **MUST WORK ALONE** for completing this MP, i.e., no partner submissions are allowed.

Do This First!

iList is an interactive visualization for linked lists. We **strongly** recommend completing all of the activities at iList before beginning the MP. We promise it will help make the rest of the MP go faster :). If you feel you'd benefit by working with someone else for this part of the MP, feel free to work with a partner on iList (but not the rest of the MP).

- To get started with iList, you'll need to create an account: http://www.digitaltutor.net/signup.php
- If you are on an EWS machine, launch iList from the command line with:

```
javaws http://www.digitaltutor.net/ilist-1.0/ilist-1.0.jnlp &
```
TERMINAL

- If you are on your own computer, you can download the launch file and double click it to run with Java Web Start: http://www.digitaltutor.net/download.php
- If you're stuck trying to figure out how to use the program, here's a tutorial

Checking Out the Code

To check out the provided code simply run

```
svn up
```
TERMINAL

from your `cs225` directory.

This should update your directory to contain a new directory called `mp3`. These files are used for both parts of the MP: MP 3.1 and MP 3.2.

Background Information: Template Classes

Template classes provide the ability to create generic container classes. In this MP, you will be writing a `List` container class. Note that the syntax for things is slightly different here:

```
template <class T>
class List {
    // implementation
};
```
C++

This simply says that our class `List` has a parametrized type that we will call `T`. Similarly, the constructor will look like this:

```
template <class T>
List<T>::List() {
    // implementation
}
```
C++

We need the `template <class T>` above all of our functions—it becomes part of the function signature.

Template classes need access to the implementation for compilation. Every time a different class is used as the template, the code must be compiled to support containing it. For example, if you want to make a `List<int>`, the compiler must take the generic `List<T>` implementation code and replace all the `T`s with `int`s inside it, and compile the result (this process is called **template instantiation**). Our solution to this is to `#include "list.cpp"` at the bottom of our `list.h` file. This ensures that whenever a client includes our header file, he/she also gets the implementation as well for compilation purposes (there are other solutions, but this is how we will solve it in this course).

Background Information: Linked Lists

The interface of this `List` class is slightly different from what you have seen in lecture. This `List` has no sentinel nodes; the first node's `prev` pointer, and the last node's `next` pointer, are both `NULL`. In lieu of these sentinels, we keep a pointer `head` to the first node, and a pointer `tail` to the last node in the `List`. (In an empty list, both `head` and `tail` are `NULL`.) The `List` class also has an integer member variable, `length`, which represents the number of nodes in the `List`; you will need to maintain this variable.

General MP Requirements

- You are required to comment the MP as per the commenting standard described by the Coding Style Policy.
- You must name all files, public functions, public member variables (if any exist), and executables **exactly** as we specify in this document.
- Your code must produce the **exact** output that we specify: nothing more, nothing less. Output includes standard and error output and files such as `PNG`s.
- Your code must compile on the EWS machines using `clang++`. Being able to compile on a different machine is not sufficient.
- Your code must be submitted correctly by the due date and time. Late work is not accepted.
- Your code must not have any memory errors or leaks for full credit. ASAN tests will be performed separately from the functionality tests.
- Your public function signatures must match ours **exactly** for full credit. If using different signatures prevents compilation, you will receive a zero. Tests for `const`-correctness may be performed separately from the other tests (if applicable).

MP 3.1: A Linked List Implementation

In your `mp3` folder, you will find the following files:

- `list.h`
- `list.cpp`

We have provided you with a skeleton for the functions needed for this part of the MP, but you will need to write the implementations. They are designed to force you to write pointer manipulation code. You will write code for these functions, which are declared in `list.h` but not defined in `list.cpp`. You must add your implementation in `list.cpp`.

See the Doxygen for MP 3 for details of the `List` class.

The `List` class is being implemented as a **doubly linked list**. Thus, you're responsible for verifying that **ALL** `next` pointers and **ALL** `prev` pointers are correctly set. Presently, if you just do

```C++
cout << list1 << endl;
```

then you're not checking the correctness of the `prev` pointers because of how we

overloaded the `operator<<` function. If you want, you can modify it in `list_given.cpp` to add this functionality, or you can check their correctness in some other way.

Notes on testing

The Makefile provided for this MP will create two useful executables when you run `make`, namely `mp3test` and `mp3test-asan`. So when you want to test a specific part of your MP you can use either of those. For example, running

```
./mp3test reverse
```
TERMINAL

will run the tests for `reverse()`. You can also run the ASAN version in the same way:

```
./mp3test-asan reverse
```
TERMINAL

Additionally, you're free to run Valgrind on the normal executable:

```
valgrind ./mp3test reverse
```
TERMINAL

MP 3.1: `~List()` and `clear()`

Since the `List` class has dynamic memory associated with it, we need to define all of the Big Three. We have provided you with the Copy Constructor and overloaded `operator=`.

- You will need to implement the `List` destructor (`~List()`) and the `clear()` helper function called by `operator=` (the assignment operator)
- Both the `List` destructor and `clear()` function should free all memory allocated for `ListNode` objects.

MP 3.1: Insertion

MP 3.1: The `insertFront` Function

(See the Doxygen for `insertFront`.)

- This function takes a data element and prepends it to the beginning of the list.
- If the list is empty before `insertFront` is called, the list should have one element with the same value as the parameter.
- You may allocate new `ListNodes`.

Example

For example, if `insertFront` is called on the list of integers

```
< 5 4 7 >
```

with the parameter `6`, then the resultant list should be

```
< 6 5 4 7 >
```

MP 3.1: The `insertBack` Function

(See the Doxygen for `insertBack`.)

- This function takes a data element and appends it to the end of the list.
- If the list is empty before `insertBack` is called, the list should have one element with the same value as the parameter.
- You may allocate new `ListNodes`.

Example

For example, if `insertBack` is called on the list of integers

```
< 5 4 7 >
```

with the parameter `6`, then the resultant list should be

```
< 5 4 7 6 >
```

MP 3.1: Testing Your `insert` Functions

Once you have completed `insertFront` and `insertBack`, you should compile and test them:

TERMINAL

```
make
./mp3test inserts
./mp3test-asan inserts
```

These tests are deliberately insufficient. You should write more tests to make sure your code works as expected.

MP 3.1: Pointer Manipulation

MP 3.1: The `reverse` Helper Function

(See the Doxygen for `reverse`.)

In `list.cpp` you will see that a public `reverse` method is already defined and given to you. You are to write the helper function that the method calls.

- This function will reverse a chain of linked memory beginning at `startPoint` and ending at `endPoint`.
- The `startPoint` and `endPoint` pointers should point at the new start and end of the chain of linked memory.
- The `next` member of the `ListNode` before the sequence should point at the new start, and the `prev` member of the `ListNode` after the sequence should point to the new end.
- You may **NOT** allocate new `ListNodes`.

Example

For example, if we have a list of integers

```
< 1 2 3 4 5 6 7 >
```

(with `head` pointing at `1` and `tail` pointing at `7`) and call the public function `reverse()`

The resulting list should be

```
< 7 6 5 4 3 2 1 >
```

(with `head` pointing at `7` and `tail` pointing at `1`)

Your helper function should be as general as possible! In other words, **do not** assume your `reverse()` helper function is called only to reverse the entire list—**it may be called to reverse only parts of a given list**.

Additionally, the pointers `startPoint` and `endPoint` that are parameters to this function should at its completion point to the beginning and end of the new, reversed sublist.

MP 3.1: The `reverseNth` Function

(See the Doxygen for `reverseNth`.)

- This function accepts as a parameter an integer, n, and reverses blocks of n elements in the list.
- The order of the blocks should not be changed.
- If the final block (that is, the one containing the `tail`) is not long enough to have n elements, then just reverse what remains in the list. In particular, if n is larger than the length of the list, this will do the same thing as reverse.
- You may **NOT** allocate new `ListNodes`.

Example

For example, if `reverseNth` is called on the list of integers

```
< 1 2 3 4 5 6 7 8 9 >
```

then the call to `reverseNth(3)` should result in

```
< 3 2 1 6 5 4 9 8 7 >
```

For the list of integers

```
< 1 2 3 4 5 6 >
```

the call to `reverseNth(4)` should result in

```
< 4 3 2 1 6 5 >
```

Hint

You should try to use your `reverse()` helper function here.

MP 3.1: Testing Your `reverse` Functions

Once you have completed `reverse` and `reverseNth`, you should compile and test them.

TERMINAL

```
make
./mp3test reverse
./mp3test-asan reverse
```

These tests are deliberately insufficient.

MP 3.1: The `waterfall` Function

(See the Doxygen for `waterfall`.)

- This function modifies the list in a cascading manner as follows.
- Every other node (starting from the second one) is removed from the list, but appended at the back, becoming the new `tail`.
- This continues until the next thing to be removed is either the `tail` (not necessarily the original `tail`!) or `NULL`.
- You may **NOT** allocate new `ListNode`s.
- Note that since the `tail` should be continuously updated, some nodes will be moved more than once.

Example

For example, if `waterfall` is called on the list of integers

```
< 1 2 3 4 5 6 7 8 >
```

then the call to `waterfall()` should result in

```
< 1 3 5 7 2 6 4 8 >
```

(Do you see the pattern here?)

Step-by-Step Example

We will look again at the list

```
< 1 2 3 4 5 6 7 8 >
```

When we call `waterfall`, this is how it should look step-by-step:

```
< 1 2 3 4 5 6 7 8 > - Skip the 1
  ^               ^
curr             tail

< 1 3 4 5 6 7 8 2 > - Remove the 2 and move it at the end
    ^             ^
  curr           tail

< 1 3 5 6 7 8 2 4 > - Skip the 3, and move the 4 to the end
      ^           ^
    curr         tail

< 1 3 5 7 8 2 4 6 > - Skip the 5 and move the 6 to the end
        ^         ^
      curr       tail

< 1 3 5 7 2 4 6 8 > - Skip the 7 and move the 8 to the end
          ^       ^
        curr     tail

< 1 3 5 7 2 6 8 4 > - We have moved past the original tail of the
            ^   ^       This is okay! Skip the 2 and move the 4 to
          curr tail     now for the second time!
```

```
< 1 3 5 7 2 6 4 8 > Skip the 6 and move the 8 to the end, now for
                ^ ^
            curr tail
```

We are done now because we skip over the `4` and get to the `tail` of the list. The `8` stays in place, and we have finished. If you were keeping track of moves, you would notice that a number (they happen to be in order here for convenience) gets moved the same amount of times as it is divisible by 2! Technically this might not be true for the `8`, but we could have moved it that last time, it just would have stayed where it was (remove it from the `tail` and put it back to the `tail`). Kinda neat, huh?

MP 3.1: Testing Your `waterfall` Function

Once you have completed `waterfall`, you should compile and test it.

TERMINAL
```
make
./mp3test waterfall
./mp3test-asan waterfall
```

These tests are deliberately insufficient.

MP 3.1: Testing

Compile your code using the following command:

TERMINAL
```
make
```

After compiling, you can run **all** of the MP 3.1 tests at once with the following command:

TERMINAL
```
./mp3test mp3.1
./mp3test-asan mp3.1
```

You should be able to `diff` the respective `.png` files with their solutions as in previous MPs.

These tests are deliberately insufficient.

Notes

- These tests are deliberately insufficient. We strongly recommend augmenting these tests with your own.
- Be sure to think carefully about reasonable behavior of each of the functions when called on an empty list, or when given an empty list as a parameter.
- It is **highly advised** to test with lists of **integers** before testing with lists of `RGBAPixels`.

- Printing out a list both forward and backwards is one way to check whether you have the double-linking correct, not just forward linking. Printing the size may also help debug other logical errors.

⚠ **DOUBLE CHECK** that you can confidently answer "no" to the following questions:

- Did I allocate new memory in functions that disallow it?
- Did I modify the data entry of any `ListNode`?
- Do I leak memory?

MP 3.1: Extra Credit Submission

For extra credit, you can submit the code you have implemented and tested for part one of MP 3. You must submit your work before the extra credit deadline (noted above). Follow the instructions in the MP 3 Submission section for handing in your code.

MP 3.2: Sorting

You will be implementing the helper functions for one more member function of the `List` template class: `sort`. This is designed to help you practice pointer manipulation and solve an interesting algorithm problem. In the process of solving this problem, you will implement several helper functions along the way—we have provided public interfaces for these helper functions to help you test your code.

MP 3.2: The `split` Helper Function

(See the Doxygen for `split`.)

- This function takes in a pointer `start` and an integer `splitPoint` and splits the chain of `ListNodes` into two completely distinct chains of `ListNodes` after `splitPoint` many nodes.
- The split happens after `splitPoint` number of nodes, making that the `head` of the new sublist, which should be returned. In effect, there will be `splitPoint` number of nodes remaining in the current list.
- You may **NOT** allocate new `ListNodes`

ⓘ **Example**

For example, if `split` is called on the list of integers

```
list1 = < 1 2 3 4 5 >
```

then after calling `list2 = list1.split(2)` the lists will look like

```
list1 == < 1 2 >
list2 == < 3 4 5 >
```

MP 3.2: Testing Your `split` Function

Once you have completed `split`, you should compile and test it.

TERMINAL

```
make
./mp3test split
./mp3test-asan split
```

You should see images `split_*.png` created in the working directory (these are generated by repeatedly splitting `in_07.png`). Compare them against `soln_split_*.png`.

MP 3.2: The `merge` Helper Function

(See the Doxygen for `merge`.)

- This function takes in two pointers to heads of sublists and merges the two lists into one in sorted order (increasing).
- You can assume both lists are sorted, and the final list should remain sorted.
- You should use `operator<` on the data fields of `ListNode` objects. This allows you to perform the comparisons necessary for maintaining the sorted order.
- You may **NOT** allocate new `ListNodes`!

Example

For example, if we have the following lists

```
list1 = < 1 3 4 6 >
list2 = < 2 5 7 >
```

then after calling `list1.mergeWith(list2)` the lists will look like

```
list1 == < 1 2 3 4 5 6 7 >
list2 == < >
```

MP 3.2: Testing Your `merge` Function

Once you have completed `merge`, you should compile and test it.

TERMINAL

```
make
```

```
./mp3test merge
./mp3test-asan merge
```

You should see the image `merge.png` created in the working directory if your program terminates properly. (This is generated by merging the images `in_08.png` and `in_09.png`.) Compare this against `soln_merge.png`.

MP 3.2: The `mergesort` Helper Function

(See the Doxygen for `mergesort`.)

- This function sorts the list using the merge sort algorithm, explained below.
- You should use `operator<` on the data fields of `ListNode` objects. This allows you to perform the comparisons necessary for sorting.
- You should use the private helper functions you wrote above to help you solve this problem.
- You may **NOT** allocate new `ListNode`s
- This function's runtime will be graded for efficiency (correct Big-Oh runtime)

Example

For example, if `sort` is called on the list of integers

```
< 6 1 5 8 4 3 7 2 9 >
```

the resulting list should be

```
< 1 2 3 4 5 6 7 8 9 >
```

Merge Sort — Algorithm Details

Merge Sort is a recursive sorting algorithm that behaves as follows:

- **Base Case**: A list of size 1 is sorted. Return.
- **Recursive Case**:
  - Split the current list into two smaller, more manageable parts
  - Sort the two halves (this should be a recursive call)
  - Merge the two sorted halves back together into a single list

In other words, Merge Sort operates on the principle of breaking the problem into smaller and smaller pieces, and merging the sorted, smaller lists together to finally end up at a completely sorted list.

MP 3.2: Testing

Compile your code using the following command:

```
make
```
TERMINAL

After compiling, you can run the MP 3.2 tests at once with the following command:

```
./mp3test mp3.2
./mp3test-asan mp3.2
```
TERMINAL

If execution goes smoothly, images named `in_01_shuffled_1.png` and `in_01_shuffled_60.png` will be created in your working directory. These files are shuffled versions of `in_01.png`. Additionally, images named `unshuffled_1.png` and `unshuffled_60.png` will be created in your working directory. These are generated by calling your `sort` function. The solution image for these files is `in_01.png`. Do not assume that if two images look similar that they match perfectly. **Use a utility such as `diff` to check for correctness.**

Notes

- These tests are deliberately insufficient. We strongly recommend augmenting these tests with your own.
- Be sure to think carefully about reasonable behavior of each of the functions when called on an empty list, or when given an empty list as a parameter.
- It is **highly advised** to test with lists of **integers** before testing with lists of `RGBAPixels`.
- Printing out a list both forward and backwards is one way to check whether you have the double-linking correct, not just forward linking. Printing the size may also help debug other logical errors.

DOUBLE CHECK that you can confidently answer "no" to the following questions:

- Did I allocate new memory in functions that disallow it?
- Did I modify the data entry of any `ListNode`?
- Do I leak memory?

MP 3: Submission

To facilitate anonymous grading, **do not** include any personally-identifiable information (like your name, your UIN, or your NetID) in any of your source files. Instead, before you hand in this assignment, create a file called `partners.txt` that contains your own NetID in this file. You are not allowed to work with others, so it should contain exactly your NetID.

We will be automatically processing this information, so do not include anything else in the file.

Our grading system will checkout your most recent (**pre-deadline**) commit for grading. Therefore, to hand in your code, all you have to do is commit it to your Subversion repository.

Be sure your working directory is the `mp3` folder that was created when you checked out the code. To hand in your code, you first need to add the new files you created to the working copy of your repository by typing:

```
svn add partners.txt
```
TERMINAL

(Since this is a solo MP, `partners.txt` should only have one line in it, with your NetID on that line.)

Adding the files to your working copy only needs to be done once.

To commit your changes to the repository type:

```
svn commit -m "mp3 submission"
```
TERMINAL

Grading Information

The following files are used to grade MP 3:

- `list.h`
- `list.cpp`
- `partners.txt`

All other files including any testing files you have added will not be used for grading.

Good Luck!