Homework 9: Linked Parity With Iterator

CSCI 60
Krehbiel

Due 11:59pm Thursday, December 4, 2025

Overview. For your last homework assignment, you will reimplement your `Parity` class from earlier. Your final implementation will be build on dynamically allocated linked lists and it will be complete with an iterator. Your starter code includes a driver program as well as completed interface files and starter implementation files for `Parity`, `node`, and `P_iterator`. You should be particularly careful in reading the invariants in each of the header files to understand how each class is supposed to work. You will only upload your completed `parity.cpp`, `node.cpp`, and `p_iterator.cpp` files.

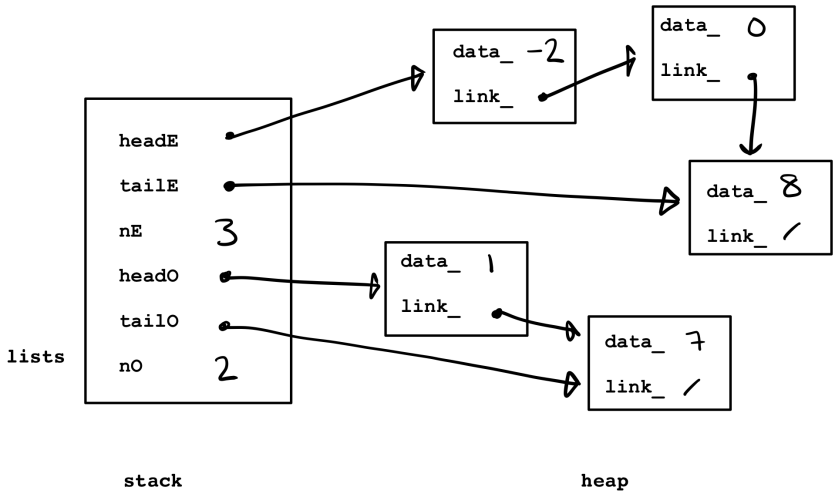
Milestone 0: Write stub implementations and run the basic driver program. The driver program is very minimal and does not yet compile. Write stub functions for each of the functions you are supposed to implement. Specifically, write a stub for `insert`, $\sim$, `<<`, `min`, `max`, `begin`, and `end` in `parity.cpp` as well as for `!=`, $\star$, and `++` (prefix) in `p_iterator.cpp`. You should not expect correct or interesting output when you run it, because you haven't implemented any of the logic yet, but after writing stub functions, you should confirm that your code compiles as follows:

```
g++ main.cpp parity.cpp p_iterator.cpp node.cpp -std=c++11
```

Moving forward, you should *write your own test code as you develop*. We have spent all quarter writing test code as we develop various member functions in class, and you should be comfortable emulating this development style, even when you are not required to turn in evidence of testing your code.

Milestone 1: Implement a function to insert in sorted order. As the only `Parity` mutator function, `insert` is responsible for maintaining the invariants outlined under the private member variable declarations as it changes the state of its calling object. *Read and understand these invariants to make sure you maintain them for all the code you write.* Specifically, `insert` should check whether its parameter `val` is even or odd and add it to the appropriate list, keeping the list *in sorted order*. To do this, `insert` should make an appropriate call to `list_insert_sorted`, a list toolkit function declared in `node.h` that you should implement in `node.cpp`. (This is the only code you'll need to add to `node.cpp`, which contains all other toolkit functionality you'll need.)

The `Parity` insert function should construct nodes dynamically on demand, so the list sizes are always exactly equal to their number of contents, and there is no need for a capacity variable. Unlike our linked-list implementation of `Bag`, the `Parity` class will keep track of the sizes of its respective lists with member variables that the `insert` function should update when updating the lists. If the values 7, 0, -2, 1, 8 are inserted into an object `lists`, the corresponding memory diagram would be as follows:



Milestone 2: Overload << and implement two other accessor functions. Next overload the insertion operator *as a friend function* so you can print out the contents of a parity object. For the object depicted above, `cout << lists;` should output the following to the console:

```
Evens:  -2, 0, 8,
Odds:  1, 7,
```

Even if you don't like the aesthetics of the comma after the last number, you should include it so your code passes the autograder! Then implement the `min` and `max` accessor functions to behave as they did with your array implementation, noting that neither of these functions require iterating through an entire list.

Milestone 3: Implement dynamic memory management. Once you are happy with your regular member functions, implement the copy constructor, destructor, and assignment operator.

- The destructor should release memory associated with the nodes for both lists and restore the default values specified by the default constructor, representing an empty `Parity` object. Note that a function `list_clear` (an iterative implementation for `f4` of last week's homework) is written for you in `node.cpp`, and this will likely be useful for your destructor.
- The copy constructor should initialize values associated with a newly constructed object with a *deep copy* of the lists of the object being copied. For full credit, implement this so the runtime is linear in the size of the parity object, meaning that you don't have to scan through the list in order to determine where to add an element, assuming of course that the object you are copying from is well-formed, meaning its contents are stored sorted.
- The assignment operator should likewise make a deep copy of the `rhs` object in linear time, but any existing memory associated with the calling object should first be released.

Milestone 4: Implement an iterator. The `p_iterator.h` file declares an iterator that can be used (among other ways) to implement range-based for loops for `Parity` objects that iterate through the sorted contents of the iterator, ignoring the parity of each value. Parity iterators maintain current pointers into both the evens and odds lists of an associated parity object, which respectively contain the next even and odd elements; exactly one of them is the next element overall. You need 5 functions to use a range-based for loop:

1. A `begin` function that is a `Parity` member function, and a 2-argument constructor in the `P_iterator` class that `begin` calls to initialize the iterator member variables to point to the head of the two lists associated with the corresponding parity object. Note that outside the class definition the iterator type should be written as `Parity::iterator` or `P_iterator`.
2. A `end` function, also a `Parity` member function that should call a `P_iterator` constructor. Note that the default values declared in the iterator interface files remove the need for a 0-argument constructor that initializes both current pointers to be null.
3. The inequality operator `!=` overloaded for `P_iterator`. Note two iterators are the same only if they are pointing to the exact same locations in the same objects. In other words, you don't need to use the `data` accessor, but rather it suffices to compare memory addresses.
4. The dereference operator `*` overloaded for `P_iterator`.
5. The prefix increment operator `++` overloaded for `P_iterator`.

The last two steps will be the trickiest, because given two pointers, you'll have to decide which one points to the *current* element. Draw pictures of iterators in progress to figure out the necessary logic. To make things easier, you may assume the dereference operator will never be called on an end iterator (so it's okay if your code causes a memory error if it is used in this way). After your implementation is complete, the test code should print out the list `-2 0 1 7 8`.

Extra credit. Some other common iterator functions are declared in the implementation file, which you may implement for extra credit. This include additional comparison operators and the postfix increment operator along with the associated pre- and post-conditions specifying what they should do.