CSCI 1321 (Principles of Algorithm Design II), Fall 2001

Homework 6[1]

Assigned: November 6, 2001.

Due: November 13, 2001, at 5pm. *Not accepted past noon on November 14.*

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Overview **1**

2 Details **1**

2.1 Introduction 1

2.2 Naive linear implementation 3

2.3 Correctness and testing 4

3 What files do I need? **6**

4 What to turn in **6**

1 Overview

You are to implement a doubly-linked list class using dynamic memory and links. Your class should be similar to the doubly-linked-list class presented in class (`dll`), *except* that it is to be linear rather than circular and is not to use a sentinel link. This implementation is the one programmers usually use if they do not know about the circular implementation with a sentinel. It usually requires keeping track of the location of the first and last links in the list. The first link's pointer to the previous link and the last link's pointer to the next link should "point nowhere". In the following, we will call pointers that point nowhere *null pointers*. If the list is empty (has no elements), the pointers to its first and last links should be null pointers.

2 Details

2.1 Introduction

A *doubly-linked list* is a data structure with objects arranged in linear order and permitting easy access to both previous and next items. For example, the STL `list` class[2] implements a doubly-linked list. (Interestingly, STL creator Alex Stepanov apparently also decided to use a circular

[1]© 2001 Jeffrey D. Oldham (`oldham@cs.stanford.edu`) and Berna L. Massingill (`bmassing@cs.trinity.edu`). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

[2]`http://www.sgi.com/tech/stl/List.html`

implementation with a sentinel.) Your implementation should support the operations listed in the following table.

Function prototype	Example use	Explanation
`dll(void)`	`dll lst;`	creates a list with no items.
`item_type`	`dll::item_type i = 'a';`	type specifying a list entry.
`link`	`dll::link * lnk = lst.erase(lnkPtr);`	type holding a list element (i.e., a link).
`link * insert(link * const pos, const item_type & item)`	`lnkPtr = lst.insert(lnkPtr, 'b');`	adds the specified item *before* the specified position.
`link * erase(link * const linkPtr)`	`dll::link * lnk = lst.erase(lnkPtr);`	removes the specified item from the list.
`link * begin(void) const`	`dll::link * lnkB = lst.begin();`	returns a pointer to the first link in the list.
`link * end(void) const`	`dll::link * lnkE = lst.end();`	returns a pointer past the last link in the list.
`link * pred(link * const lnk) const`	`dll::link * lnk = lst.pred(lnkE);`	returns a pointer to the link just before the given link.
`link * succ(link * const lnk) const`	`dll::link * lnk = lst.succ(lnkB);`	returns a pointer to the link just after the given link.

When inserting an item into a list, the new link (for the inserted item) should be to the left of the user-specified position. If this position is a null pointer, the item should be placed at the right end of the list. Insertion at the left end of the list is accomplished by inserting the desired item before the leftmost link. To erase a link, the user need only specify the link to erase. The `insert()` function returns a pointer to the inserted link; the `erase()` function returns a pointer to the link to the right of the erased link, or a null pointer if the rightmost link was removed. Both functions may assume that their parameter values are valid. (That is, `insert()` can assume that its pointer parameter is either null or points to a valid link, and `erase()` can assume that its parameter points to a valid link.)

The last four functions permit the user to "move" through the items in the list. `begin()` and `end()` return pointers to a list's first link and one past its last link, respectively. (So, `end()` should return a null pointer.) Given a pointer into the list, `pred()` and `succ()` return pointers to its predecessor (the link to its left) and it successor (the link to its right), respectively. The predecessor of the leftmost link is a null pointer as is the successor of the rightmost link. When given a null pointer, `pred()` should assume it points to just past the end of the list and should return a pointer to the rightmost link. When given a null pointer, `succ()` should assume it points to just before the beginning of the list and should return a pointer to the leftmost link.

Please ensure that, if a list is copied, changing the original list's contents does not change the copy and vice versa. (Hint: Write an assignment operator and a copy constructor if necessary.) Be sure not to leak dynamic memory. (Hint: Write a destructor if necessary.)

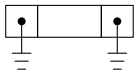
For now, assume the list contains `char`s, but write your code using `item_type` for the type of the list items, so that it would be easy to templatize later.

2.2 Naive linear implementation

In class, we presented algorithms and code (`dll.h`[3]) for a circular implementation with a sentinel link. In this assignment, we will use a linear implementation with no sentinel. For example, here are

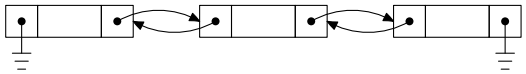
- an empty list,[4]

- a list with one link,



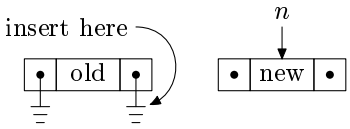
  and

- a list with three links.

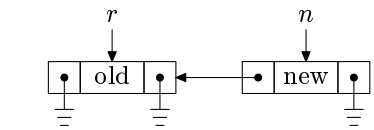


The number of links equals the number of items in the list, and null pointers appear at the left and right ends of the list. (In the above figures, null pointers are represented with a symbol that looks something like the symbol used to represent "ground" in circuit diagrams.) In C++, null pointers are represented as `0`, i.e., zero. (Header file `cstddef` defines a macro `NULL` that you can use instead of 0; I find this somewhat clearer.)

The linear implementation will be very similar to the circular implementation with a sentinel, except that the routines cannot always assume that there is a link to the left and right when inserting and erasing. For example, consider inserting a link at the right end of a list with one link. We are given a null pointer. Assume we have already created a link named n to hold the new item.



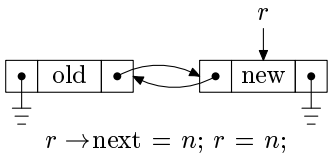
We need to know the link to the left. Thus, let us assume the `dll` object maintains a `link * r` always pointing to the list's rightmost link. The first step is to set n's pointers to the correct locations.



The next step is to change the surrounding links' pointers. Since there is no link to the right of n, we cannot change its link. Finally, we update the object's pointer to the rightmost link.

[3] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/dll.h`

[4] This looks like Darth Vader's ship in episode 4 of Star Wars. Just as Darth was the bane of the rebels' existence, the empty list will be the bane of your existence, at least for a little while.



This implementation differs from the circular implementation with a sentinel in that (i) we needed a pointer to the list's rightmost link and (ii) we did not change the pointer of the link on the right.

2.3 Correctness and testing

Your implementation should support insertion and erasure anywhere in a list (at the beginning, middle, or end) and for a list of any length (empty, one link, or multiple links). Before writing code, I strongly recommend that you draw pictures of all possible cases and annotate with the associated code. Otherwise, the probability of obtaining a correct implementation is minimal.

The recommended strategy for implementing and testing your code is to interleave writing member functions, compiling, and testing. That is, choose an order for implementing member functions that minimizes the amount of code written between compilations and testing. (The `dll` class presented in class can be found in sample program file `dll.h`[5]. You are welcome to use this as a starting point, but I recommend that you begin by removing or commenting out everything except the function prototypes and then proceed as follows.)

1. Write a very simple test program that declares a doubly-linked list class and does nothing else.
2. Write the `dll` skeleton, i.e., `class dll`, braces, a semicolon, and nothing else.
3. Compile (the test program), fixing all errors.
4. Add definitions for `item_type` and `link`.
5. Compile, fixing all errors.
6. Add definitions for `item_type` and `link`, plus a constructor with no arguments. This may require adding private variables and helper functions.
7. Compile, fixing all errors.
8. Add definitions for `begin()`, `end()`, `pred()`, and `succ()`.
9. Compile, fixing all errors.
10. Add a definition for `insert`. This may require adding additional helper functions.
11. Revise the test program to test inserting at the beginning of the list, at the end of the list, and in the middle of the list. (You do not need to spend a lot of time writing a complicated test program; just write a sequence of calls to the functions you have written that test all the possible cases. File `dll-test.cpp`[6] contains a simple starter test program.)
12. Compile, fixing all errors. Check for correctness.
13. Add (if you think it is needed) a destructor function.

[5] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/dll.h`
[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW06/Problems/dll-test.cpp`

14. Compile, fixing all errors. Check for correctness; also check for memory leaks.

15. Continue the process, implementing `erase` last.

It may also be useful to overload the output operator to print a list's contents.

After you have finished the implementation, try using it with the simple-minded (a.k.a. "brain-dead") string editor presented in class. To do this, copy files `editor.cpp`[7] and `buffer.h`[8] into your directory, be sure your definition of the `dll` class is in a file called `dll.h`, and compile `editor.cpp`.

Your implementation should correctly use dynamic memory. For example, dynamic memory should be `delete`d when no longer used, and `delete`d memory should not be used. If you wish to use the `mtrace` command to help find memory-use errors, here are the instructions again:

1. In your test program, add

   ```
   #include <mcheck.h>
   ```

   and, at the beginning of `main()`, add

   ```
   mtrace();
   ```

   Let's assume the executable is named `a.out`.

2. Before running the executable, type

   ```
   declare -x MALLOC_TRACE=foo.txt
   ```

   in a shell. (You can replace `foo.txt` with any filename you choose.)

3. Run the executable. Memory allocation and deallocation information is stored in the file called `foo.txt` (or the filename you chose in the above step).

4. To print memory leak information, type

   ```
   mtrace a.out $MALLOC_TRACE
   ```

   in the same shell.

For more information, see the info pages. You can access the relevant pages by first typing `info` (to start a text-based program to browse info pages) and then typing `m libc`, `m memory allocation`, and `m allocation debugging`.

Some caveats:

1. Ignore any leak information not involving the words "new" or "delete." For example, ignore any leaks involving the word "exit."

2. Ignore the indicated line numbers. Just check your code for `new`s without corresponding `delete`s and vice versa.

3. Using STL `string`s, or other STL classes, may cause memory leak errors; the algorithm used by some STL classes to allocate memory does not completely clean up after itself at the end of the program.

[7] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/editor.cpp`
[8] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/buffer.h`

3 What files do I need?

I suggest starting with the implementation presented in class and converting it to a linear implementation. These two files should be useful:

- `dll.h`[9] contains a slightly revised version that includes all the functions you are required to write, plus an output operator.
- `dll-test.cpp`[10] contains a starter test program. (It assumes that the doubly-linked-list class implementation is contained in file `dll.h`.)

You may also want to get files `editor.cpp`[11] and `buffer.h`[12], to test your code with the string-editor program.

4 What to turn in

Submit only the file containing your revised implementation of the doubly-linked-list class (`dll.h` or `dll.h`), as described in the Guidelines for Programming Assignments[13]. For this assignment use a subject line of "cs1321 hw 6".

[9] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/dll.h`
[10] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/HW06/Problems/dll-test.cpp`
[11] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/editor.cpp`
[12] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/buffer.h`
[13] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/pgmguidelines/index.html`