

Reading

Wednesday 27 April: Sethi, Sections 6.2, 6.7
Friday 29 April: Sethi, Sections 7.1–7.4
Monday 2 May: Sethi, Sections 7.5, 7.6

Exercises

Due date: Tuesday 3 May. Please leave your paper in the Taylor Hall homework box (it's located in the breezeway between 2.132 and 2.136) **by 4pm.**

1. [20] Use the `IntArray` class defined in Slides 293–302 (and available at <http://www.cs.utexas.edu/users/ham/UTCS/CS345/Homework/IntArray.cp>) in a program which

- declares two 10-element `IntArray`s A and B
- fills A with integers 100–109, and B with integers 200–209
- declares `IntArray` C, initializing it as a copy of A
- prints the contents of A, B, and C
- assigns B to A
- prints the contents of A, B, and C
- assigns A+C to B
- prints the contents of A, B, and C

Add some “instrumentation” to `IntArray`'s constructors, assignment operator, and destructor, in the form of output statements that identify the function being executed and print the heap addresses involved. Make the instrumentation's activity dependent on the value of a global variable, and run your program twice, once with the instrumentation off, and again with it on. Add comments to the output to explain what's happening.

2. [20] Change the definition of `IntArray` to obtain reference semantics:
- remove the destructor
 - replace the copy constructor and assignment operator with ones that mimic the default copy constructor and the default assignment operator, which perform shallow copies (add instrumentation as in Exercise 1).
 - add a member function `dispose` which deallocates an `IntArray`'s heap storage
 - add a member function `clone` which returns a new `IntArray` object equivalent—but not identical—to the one being cloned.
 - include instrumentation in `clone` and `dispose` similar to that which you added to `IntArray`'s constructors, assignment operator, and destructor in Exercise 1.
- a. Run the program you wrote for Exercise 1 with no changes except the `IntArray` changes listed above; run it twice, once with instrumentation and once without. Does it get the same results as the Exercise 1 version? Are there any storage leaks?
- b. Revise the program of part (a), using `clone` and `dispose` as needed to get the same results as in Exercise 1, including the instrumentation output.

3. [30] Define in C++ a class template `List<T>`. Lists defined by this template contain zero or more items, and include the notion of *current* item; these lists can be traversed in *either direction*. The interface includes the following member functions:

- `ins`: inserts a copy of a given item before the list's current item (if none, inserts at end); the new item is then current.
- `del`: deletes the current item (error if none)
- `next`: if the current item is the list's last item, *current* becomes undefined; otherwise, the current item's successor becomes current
- `atEnd`: true if there is no current item (the list is empty or next has advanced past the last item)
- `prev`: if `atEnd` and the list is not empty, the last item becomes current; otherwise, the current item's predecessor becomes current
- `atFront`: the list's first item is current
- `cur`: returns a reference to the current item (error if none)
- `reverse`: reverses the order of elements in the list (this can be done in constant time).

The design should include a companion class template `Cell<T>`, which contains two data members:

- the value of a list item
- the address of the next or previous list cell

The `next` and `prev` member functions should operate by reversing `Cell` pointers so that the items preceding the current item are linked in reverse order.

Define the `List<T>` class template so that it has *value semantics*; that is, provide an appropriate copy constructor, assignment operator, and destructor.

Demonstrate the class template using some lists of integers.

4. [10] Define an ADT `List a` in Haskell similar to the C++ ADT specified in Exercise 3, using Haskell's built-in lists and defining corresponding interface functions.

Suggestion: Implement the `List` ADT with a pair of Haskell lists. One list would contain the current item and all of the following items, and the other would contain the preceding items, in reverse order. Note that each of the interface functions that "updates" a list in C++ will produce a new list in Haskell.

Does Haskell support value semantics, reference semantics, both, or neither? Explain.

5. [20] Sethi, Exercise 7.11. This is a very nice problem involving class derivation, but Sethi's main that tests it (Figure 7.22) is a little weak. Improve it by declaring a few variables of type `tree*` (rather than `tree`), and make it construct and print a tree corresponding to the following expression:

$$((1 + 2) * (3 - 4)) + (5 * 6)$$

6. [40] Define in C++ a noncircular version of class `List` in Sethi, Section 6.6. Call the new class `LinkList`, and use pointers to the front and rear cells in the list. Define `LinkList::pop` so that if the list is empty it calls `exit`.

- a. For this exercise, no copy constructor or assignment operator is needed, but the default versions would be dangerous. To prevent trouble, define a *private* copy constructor and a *private* assignment operator which do nothing. Then demonstrate what happens if you try to compile code that calls these functions.

Then use `LinkList` to define `Queue` and `Stack` classes in two different ways:

- b. Define `Queue` and `Stack` by *private derivation*, using `LinkList` as a base class.
c. Define `Queue` and `Stack` using *member variables* of type `LinkList`.

7. [20] Repeat Exercise 6c, adding to `LinkList` a provision for throwing exceptions when an attempt is made to refer to an element of an empty list. The `Stack` and `Queue` ADTs should catch the `LinkList` exceptions and re-throw them as `Stack` and `Queue` exceptions.

CS 345 Richards

Assignment **10**

Authors:

1. _____

2. _____

3. _____

4. _____

Since the grading of this set of homework exercises will not be completed before the last class day, graded papers will be available for pickup only at my office.