COP 3530 Project 1: Part I

Data Structures & Algorithms
Fall, 2014

Part I: Lists

Due Tuesday, 9/23

Design, implement, and *thoroughly* test all four varieties of list classes as described below.

The List varieties

Simple Singly-Linked List (SSLL)

A basic implementation: whenever an item is added to the list, a new node is allocated to hold it; whenever an item is removed, the node that held it is deallocated.

Pool-using Singly-Linked List (PSLL)

Similar to the SSLL, except the node allocation/deallocation strategy is different. The list maintains a *pool* of unused nodes. Whenever an item is added to the list, use a node in the pool of free nodes; if there are no free nodes, then a new node is allocated. Whenever an item is removed, the node is added to the pool.
Because we don't want the pool to waste too much memory, whenever the list contains ≥ 100 items AND the pool contains more nodes than half the list size, reduce the number of pool nodes to half the list size (by deallocating the excess).

Simple Dynamic Array-based List (SDAL)

A basic implementation: the initial array size is passed as a parameter to the constructor; if no value is passed then default to a backing array with 50 slots. Whenever an item is added and the backing array is full, allocate a new array 150% the size of the original, copy the items over to the new array, and deallocate the original one.
Because we don't want the list to waste too much memory, whenever the array's size is ≥ 100 slots and fewer than half the slots are used, allocate a new array 50% the size of the original, copy the items over to the new array, and deallocate the original one.

Chained Dynamic Array-based List (CDAL)

This is the one described in the discussion session. The idea again is that a linked-list of arrays is used as the backing store. Each array has 50 slots. The chain starts off containing just a single array. When the last array in the chain is filled, and a new item is inserted, a new array is added to the chain.
Because we don't want the list to waste too much memory, whenever the more than half of the arrays are unused (they would all be at the end of the chain), deallocate half the unused arrays.

List operations

These are the methods your list classes must support. Hopefully, the function names are self explanatory, but just in case here are brief descriptions. The term ”shifting right“ means that the positions indices are increased, while ”shifting left“ means that the positions indices are decreased

- **`replace( element, position )`** — replaces the existing element at the specified position with the specified element and returns the original element.
- **`insert( element, position )`** — adds the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the ”right.“

- **`push_back( element )`** — appends the specified element to the list.
- **`push_front( element )`** — prepends the specified element to the list.
- **`remove( position )`** — removes and returns the the element at the specified position, shifting the subsequent elements one position to the ”left.“
- **`pop_back()`** — removes and returns the element at the list's tail.
- **`pop_front()`** — removes and returns the element at the list's head.
- **`item_at( position )`** — returns (without removing from the list) the element at the specified position.
- **`is_empty()`** — returns `true` IFF the list contains no elements.
- **`size()`** — returns the number of elements in the list.
- **`clear()`** — removes all elements from the list.
- **`contains( element, equals_function )`** — returns `true` IFF one of the elements of the list matches the specified element.
- **`print( ostream )`** — If the list is empty, inserts " <empty list>" into the ostream; otherwise, inserts, enclosed in square brackets, the list's elements, separated by commas, in sequential order.

Implememntation

Code template

I've provided a skeleton SSLL.H to get you started with implementing the SSLL. You may add additional private functions, but may not add any additional public functions. I've also provided a simple_test.cpp, that should, if studied, provide some insights as to how a couple functions are used and work (as well as providing examples of the `print()` function's output) —it is not a substitute for a through test program. For the other list classes, you are on your own.

Parameter value checking

For this part of the project, you may assume that all calls made to your SLL implementation will be made with valid arguments (e.g., for the function `item_at( int position)`, *position* will always be non-negative and less than the list's size).

Things to remember

This, and *all* programming assignments are to be tested using the **g++** (GNU C++) compiler v4.8.2 running on OpenBSD 5.5 (i386). Also, remember the deliverable requirements as outlined on the course policies page. *A detailed description of your testing process is expected, along with the test code, and its output.*