CMSC 202 — Fall 2015 — Prof. Marron

Project 5: Templated Linked List

Due: Monday, December 7, 9:00PM

Objective
The objectives of this project are 1) to practice designing and implementing templated C++ classes, and 2) to experiment with running times of data structures.

Background
One problem with the design of container classes (e.g., `list` and `vector` in the Standard Template Library) is to provide a method for the client to traverse the data structure efficiently. The C++ Standard Template Library uses iterators. In this project, we will try a different approach.

For a singly-linked list, we can overload the `[]` operator, to retrieve the i-th item of a linked list. The intention is that we can use a for loop, as in the following program fragment which increments each item of the linked list.

```
LinkedList<int> L ;

...

for (unsigned int i = 0 ; i < L.size() ; i++) {
   L[i] = L[i] + 1 ;
}
```

The problem with this approach is that each time `L[i]` is used, the code for the `operator[]` function must go to the beginning of the linked list and follow the links to the i-th node of the list. In fact, `L[i]` is used twice in the body of this loop, so we would have to traverse the linked list twice. This is horribly inefficient. The loop above should run in *O(n)* time, but a naive implementation takes $O(n^2)$ time.

There is a simple solution to this problem: *caching*. We can modify the linked list data structure so that it "remembers" or caches a pointer to the i-th node of the linked list. This pointer must be saved in a data member of the linked list along with the value of i. When the `operator[]` function is called, it checks to see if the index requested is the same as the cached index. If so, it can reuse the cached pointer instead of looking for the i-th node from the beginning of the linked list. The cache can also be used when an index greater than the cached index is requested. In fact, using the cached pointer, advancing one step in the linked list takes constant time because computing `L[i+1]` involves moving just one node from the cached position. Each time the `operator[]` function is invoked, it should cache a new pointer and index. (The newly cached values replace the previous cache.) Thus, the entire loop will run in linear time.

One problem with caching is that the cached values can become invalid. Whenever we add a node to or remove a node from the linked list, the position of the nodes in the linked list change. Thus, we have to declare that the cached values, if any, are no longer valid. Thus, the linked list must also have some mechanism to remember whether the cached pointer and index is valid.

This basic strategy can be expanded. For example, we could cache multiple pointers and indices instead of just one pointer and one index. When we add a node to the front of the linked list, we can add 1 to the indexed cache (if one exists) instead of invalidating the cache. For doubly linked lists, we can use the cache in both directions. However, for this project, we will keep thing simple and just have singly-linked list with a single cached pointer and index.

Assignment

Your assignment is to design and implement a *templated* singly-linked list that uses the caching strategy described above. You should have a templated class called `Node` and a templated class called `LinkedList`. Your clients will not use the `Node` class directly, so

```
#include "list.h"
```

must be sufficient to use all of the member functions of the `LinkedList` class.

Your templated `LinkedList` class must have the following member functions:

- A default constructor.
- A copy constructor with signature

  ```
  LinkedList(const LinkedList<T>& otherList) ;
  ```

  Here `T` is the type parameter of your templated class.
- A destructor.
  Note: your destructor should loop through the linked list explicitly and deallocate each node in the linked list. You should not use the `Node` destructor to destroy subsequent nodes in the linked list. This is because for very long linked lists (with millions of nodes), the sequence of destructor calls will cause memory error.
- A member function `addFront` that adds a new node to the beginning of the linked list with the given data. It should have the following signature:

  ```
  void addFront(const T& moreData) ;
  ```

  Again, `T` is the type parameter of your templated class. This function should run in constant time.
- A member function `deleteFront` that removes the first node of the linked list and returns the value stored in that node. It should have the following signature:

  ```
  T deleteFront() ;
  ```

  If the list is empty, then `deleteFront()` should throw a `range_error()` exception:

  ```
  throw range_error("deleteFront() on empty list") ;
  ```

  You must

  ```
  #include <stdexcept>
  ```

  to make `range_error` available to your program. This function should run in constant time.
- A member function `remove` that looks for a node with a given value and removes it from the linked list, if found. If there is more than one node with the given value only the first occurrence is removed. The function must have the signature:

  ```
  bool remove(const T& searchData) ;
  ```

  If there is no node with value equal to `searchData`, then calling `remove` should have no effect. The return value indicates whether the value is found. (True means found.)
- A member function `size` that returns the number of nodes in the linked list (not counting the dummy header). It should have the signature:

```
unsigned int size() const ;
```

This function must run in constant time. Thus, you should store the size of the linked list in a data member and update it as necessary. Do not compute the number of nodes in the linked list by traversing the linked list. That will make our for loop above run in quadratic time.

- An overloaded `[]` operator with the signature:

```
T& operator[](unsigned int i) ;
```

It should return a reference to the data in the i-th node of the linked list. This function must implement the caching strategy described above. The function returns a reference so we can assign values to the i-th node of the linked list. (See example loop above.)

- An overloaded assignment operator.

```
const LinkedList<T>& operator=(const LinkedList<T>& otherList) ;
```

This should follow the usual guidelines for implementing assignment and also make a deep copy of the linked list. Note that the new list will not have a valid cache even if `otherList` has a valid cache.

- A member function `print` that prints out the data in each node in the linked list, one item per line. You may assume that objects of the templated type work with the insertion operator `<<`. This function should have the signature:

```
void print() ;
```

You are allowed to have other member functions. No `list.h` or `node.h` file is provided — you must make your own. However, your implementation must be compatible with the following main programs. (These files are also available on GL in the `/afs/umbc.edu/users/c/h/chang/pub/cs202stuff/proj5` directory.)

- `p5main1.cpp`
  A very simple test of [] operator.
  Sample output: p5main1-output.txt

- `p5main2.cpp`
  Checks copy constructor and assignment operator.
  Sample output: p5main2-output.txt

- `p5main3.cpp`
  Simple test of remove() and deleteFront(). Also, checks that cache invalidation works.
  Sample output: p5main3-output.txt

- `p5main4.cpp`
  Testing with list of vectors as 2D array. A `LinkedList` of `vectors` can be used with [][].
  Sample output: p5main4-output.txt

- `p5main5.cpp`
  Simple timing run to show that addFront() is O(1) time. You should adjust the `base` variable in the `main()` so that the program runs in reasonable time on your machine.
  Sample output: p5main5-output.txt

- `p5main6.cpp`
  Simple timing run to show that looping thru a LinkedList using a for loop in increasing order is linear time. Again, adjust `base` as needed.
  Sample output: p5main6-output.txt

- `p5main7.cpp`
  Simple timing run to show that looping thru a LinkedList in reverse order takes quadratic time because we cannot take advantage of caching the index.
  Sample output: p5main7-output.txt

- p5main8.cpp
  Checks that caching still works when the indices go up by a factor of 2 instead of increasing by +1. Also uses very large linked lists.
  Sample output: p5main8-output.txt

You should run all of these programs with `valgrind` and no memory leaks should be indicated.

Implementation Issues

1. You do not have to start from scratch. You are allowed to modify your code from Project 3, or the linked list code from Lab 6, or the linked list code from your text book.
2. Remember that the templated header files have to include the `.cpp` files at the end. Also, **both** the `.h` files **and** the `.cpp` files must be guarded.
3. If you want your templated `LinkedList` class to be a friend of your `Node` class, you have to make the forward declaration:

   ```
   template <class T> class LinkedList ;
   ```

   in `node.h` and in the class declaration of `Node` say:

   ```
   friend class LinkedList<T> ;
   ```

4. Remember to update or invalidate the cache! Think of which member functions have to update, which have to invalidate and then check that your implementation actually does this.
5. If the `operator[ ]` is given a bad index, it should throw a `range_error`.
6. You can call the `operator[]` function for the host object with either:

   ```
   operator[](i)
   ```

   or

   ```
   (*this)[i]
   ```

   This might be useful in `print()`.

Submitting your program

You should submit these files to the `proj5` subdirectory: `node.h`, `node.cpp`, `list.h`, `list.cpp`,

You should also submit a main program that works for you. This allows you to demonstrate what you have accomplished if you are not able to implement all of the required features. Call this file `proj5.cpp`.
