CS280 Assignment - Lariat

Luke Cardell, slightly modified by Dmitri Volper

Contents

1 What is Lariat

> It is important to understand that this is not the only way to solve this assignment, merely the way that worked best for me.

Under the hood, Lariat looks like a ”linked list of arrays”.

The main goal is to have a more efficient (compared to array) insert in the middle. Reminder: if array (and consequently `std::vector`) insert an element in the middle, all elements to the right of the insert position have to be shifted 1 position to the right to free a spot for new element. This is $O(N)$ (linear) operation. Another possible problem is if there is not anough space, which will required reallocation if the whole array. Lariat solves the problem by storing small segments of contiguous memory non-contiguously.

During insert one of 2 things can happen (see datail later):

- the contiguous block may have extra space, so only some elements have to be shifted (1)
- block containing insert position is full, then we split it (2)

```
//    0  1            2  3  4        // index
// +------------+  +------------+
// |  1  3     -|->|  4  5  6  -|->  // data
// +------------+  +------------+
// after insert 2 before 3
//    0  1  2         3  4  5         6  7  8
// +------------+  +------------+  +------------+
// |  1  2  3  -|->|  4  5  6  -|->|  9  9  9  -|->
// +------------+  +------------+  +------------+
// only value 3 had to shifted
```

Figure 1: Block containing insert position has extra space

```
// insert 8 before 5
// first split:
//    0  1  2         3  4            5  6            6  7  8
// +------------+  +------------+  +------------+  +------------+
// |  1  2  3  -|->|  4  8     -|->|  5  6     -|->|  9  9  9  -|->
// +------------+  +------------+  +------------+  +------------+
// now insert
//    0  1  2         3  4            5  6            7  8  9
// +------------+  +------------+  +------------+  +------------+
// |  1  2  3  -|->|  4  8     -|->|  5  6     -|->|  9  9  9  -|->
// +------------+  +------------+  +------------+  +------------+
// only values 5 and 6 were copied
```

Figure 2: Block containing the insert position is full

2 Luke's Guide to Templated Templates

"We put some templates in your templates so you could generate code at compile time while you generate code at compile time." -C++

A template is a section of code like any other that gets "written" at compile time, rather than when you actually write templated code. In this sense, writing a template is about using the compiler to your own advantage to avoid doing unnecessary work.

Laziness FTW

A normal function template is declard as such:

```
template<typename T>
void Foo(T bar);
```

The definition requires knowledge of the template parameters in order to work:

```
template<typename T>
void Foo<T>(T bar) {
        //     |
        //      `-[function template ]

        //...
}
```

In the Lariat assignment, you will have to make a templated copy constructor and assignment operator for the Lariat template class. This is similar to writing a static function template, but it's written within another encompassing template, the Lariat class in this case.

The basic declaration of the function is just like a normal template function, but within a class scope.

```
template<typename T>
class Foo {
        //...

        template<typename Bar>
        void Baz(Bar & qux);

        //...
};
```

The definition of the function takes a little bit more work, which may be confusing at first.

```
template<typename T>    // Gives the external class template information
template<typename Bar>  // Gives the internal function template information
            void Foo<T>::Baz<Bar>(Bar & qux) {
  // |          |
  // |           `-[function signature]
  //  `-[class namespace]

  // ...
}
```

The first template declaration belongs to the class, which may be instantiated multiple times with multiple name-mangled signatures based on the compiler's implementation of templates.

The second template declaration belongs to the function, which will likely be instantiated multiple times for every single class instantiation. In this way the structure resembles a tree:

```
          +-source-----------------------------+
          | original code written by programmer |
          +------------------------------------+
            /                              \
      +-class---------+                    +-class---------+
      | instantiation |                    | instantiation |
      | for type A    |                    | for type B    |
      +---------------+                    +---------------+
      /               \                    /               \
+-function------+ +-function------+     +-function------+ +-function------+
| type A        | | type B        |     | type A        | | type B        |
| instantiation | | instantiation |     | instantiation | | instantiation |
+---------------+ +---------------+     +---------------+ +---------------+
```

Figure 3: #dat_ascii

The next little template trick you might run into with this assignment comes when you try to refer to internal member types in the instantiation of code. I personally used this in my helper functions that returned pointers to nodes in the list, but it's useful for any further template work you might be interested in doing.

Dependent types are also instantiated for ech class, with the exact same process as function templates. This means that using just the internal name in external code will lead to type confusion, where the compiler isn't sure which owning class is being referred to.

A class class template with a member type and a member function returning a pointer to the member type:

```
template<typename T>
class Foo {
  //...

  struct Bar {
    //...
  }

  Bar * Baz();

  //...
};
```

The member function here must be defined as such:

```
    template<typename T>
                typename Foo<T>::Bar * Foo<T>::Baz() {
      // |       |    |       |     |
      // |       |    |       |      `-[function signature]
      // |       |    |        `-[class namespace]
      // |       |     `-[member typename]
      // |        `-[class namespace]
      //  `-[declaration of token as user-defined type]

      //...
    }
```

The last real trick I want to share with you pertains directly to the copy constructors this assignment requires. At their instantiation, class templates only have direct access to their own instatiation's member variable. This means that in order to see another instantiation's private variables, there must be a declaration of each instantiation as a friend.

How do you make something to account for any scenario at compile time?

Simple. More templates.

Declare a friend class template in your original class template.

```
template<typename T>>
class Foo {
  //...

  template<typename U>  //[nested template declaration]
  friend class Foo<U>;  //[friend declaration for class template]

  //...
}
```

This has been my brief introduction to templates as they pertain to this assignment. they may seem scary, but templates allow for a greater depth and elegance of code, decreasing the total writing time and reducing the volume of duplicated code.

3 Assignment Details

3.1 Lariat Class Details

3.1.1 Template Parameters

The `Lariat` class has two template parameters which are generally self- explanatory.

`typename T`	The type of data being stored in the data structure
`int Size`	The logical size of the arrays within the node

3.1.2 Member Variables

The `Lariat` class provided by Volper has five member variables.

`head_`	A pointer to the first node in the linked list
`tail_`	A pointer to the last node in the linked list
`size_`	An integer for the total number of elements actively storing data
`nodecount_`	An integer for the number of nodes in the linked list
`asize_`	An integer for the size of the array within each node Interchangeable with the template parameter `<Size>`

3.1.3 Member Type

The `Lariat` class has one member type, the `LNode` struct used in the linked list internal implementation. `LNode` has four member variables of its own.

`next`	A pointer to the next node in the linked list
`prev`	A pointer to the previous node in the linked list
`count`	An integer to track of how many elements the `LNode` is actively storing
`values`	An array of `<Size>` elements of type `<T>`

3.2 Function Details

Remember to check for null at any point you're dealing with pointers. In particular, make sure to check if the head and tail are null before you try to work with them. I find that not dereferencing null is generally the better programming practice.

Also remember to update the `size_` and `nodecount_` member variables where necessary.

- Constructors and Destructor
  - ctor
  - copy ctor (own-type)
  - copy ctor (template)
  - dtor
- Assignment Operators
  - `operator=` (own-type)
  - `operator=` (template)
- Element Addition
  - `insert`
  - `push_back`
  - `push_front`
- Element Removal
  - `erase`
  - `pop_back`
  - `pop_front`
- Element Access
  - `operator[]`
  - `first`
  - `last`
- Data Structure Information
  - `find`
  - `size`
- Data Structure Control
  - `clear`
  - `compact`
- Recommended Helper Functions
  - `split`
  - `findElement`
  - `shiftUp`
  - `shiftDown`

3.3 Constructors and Destructor

3.3.1 Constructor

This constructor is really simple. You don't need to do any logic, just use a member initializer list to initialize

3.3.2 Copy Constructor (own-type)

This is the standard copy constructor. The function should loop through the instance passed in, pushing each element of the other onto the back of the one being constructed.

It should be done using the algorithm for (or directly calling) `push_back` so that all of the nodes are split correctly.

3.3.3 Copy Constructor (template)

This copy constructor is algorithmically identical to the own-type constructor, but is templated to allow for a Lariat type instantiation with different parameters for `<T>` and `<Size>`.

Refer to the above guide to templates for details on how to write the function signature.

3.3.4 Destructor

The destructor is a simple, generic destructor. It's sole purpose is to free all the nodes in the linked list so there are no memory leaks.

3.4 Assignment Operators

3.4.1 `operator=` (own-type)

This assignment operator generally works exactly the same as you might expect. Set the non-pointer members as necessary, clear this instance's data, then walk through the right-hand argument's list adding each element to this instance

3.4.2 `operator=` (template)

This assignment operator follows the exact same algorithm as the own-type version, but is a nested template like I described above.

3.5 Element Addition

3.5.1 `insert`

Insert an element into the data structure at the index, between the element at `[index - 1]` and the element at `[index]` The first thing to this function is to check for an Out of Bounds error. If the index is invalid, throw a `LariatException` with `E_BAD_INDEX` and the description "Subscript is out of range" Make sure to handle the "edge" cases, which allow for insertion at the end of the deque as well as the beginning. I personally suggest calling `push_back` and `push_front` in these cases, as it helps minimize the amount of code written and the algorithm is identical anyways.

The next thing to do is to set up the actual insertion algorithm. First, find the node and local index of the element being inserted. This can be done with the `findElement` helper function detailed in the Recommended Helper Functions section of this guide.

Next, shift all elements past that local index one element to the right. This can be done with the `shiftUp` function detailed in Recommended Helper Functions (4).

```
/*
      [e]
       |
       v
  +------------------+        (0) Starting position, inserting 'e' at
  |[a][b][c][d][_][_]|            index 1 of the node
  +-0--1--2--3--4--5-+

      [e]
       |
       v
  +------------------+        (1) Shift all elements right from index
  |[a][_][b][c][d][_]|
  +-0--1--2--3--4--5-+

      [_]
       |
       v
  +------------------+        (2) Set the newly open element to the value
  |[a][e][b][c][d][_]|            being inserted, 'e' here
  +-0--1--2--3--4--5-+
*/
```

This should work for the most common use-case, but it will cause a problem if the node is full.

If the node is full, you will need to keep track of the last element in the node before you shift all the elements to the right. Again, this is easy to do with the recommended helper function. I call this "popped off" element the overflow.

```
/*
      [g]
       |             [_]
       v             /
  +------------------+        (0) Starting position, inserting 'g' at
  |[a][b][c][d][e][f]|            index 1 of a full node
  +-0--1--2--3--4--5-+

      [g]
       |            [f] <-[overflow]
       v            /
  +------------------+      (1) Shift all elements right from index,
  |[a][_][b][c][d][e]|          keeping track of the overflow, 'f' here
  +-0--1--2--3--4--5-+

      [_]
       |            [f]
       v            /
  +------------------+      (2) Set the newly open element to the value
  |[a][g][b][c][d][e]|          being inserted, 'g' here
  +-0--1--2--3--4--5-+
*/
```

Next, you will need to split the node. I would recommend writing a helper function for this algorithm as it will be used elsewhere. I have detailed the split algorithm in Recommended Helper Functions (4). You will need to transfer the overflow to the last element of the new node created by the split. It is possible to put this part of the algorithm directly in the split function. I will not detail that in this guide, but rather leave it for you to discover on your own. Working out that algorithm will probably make your code much cleaner, so I would definitely recommend figuring it out. As the split algorithm accurately updates the node counts for the split nodes, the only thing left to do is increment the node count.

3.5.2 `push_back`

This is an easy algorithm using the `split` function (4).

- If the tail node is full, split the node and update the `tail_` pointer.
- Set the last element in the tail's array to the value.
- Increment the tail node's count.

3.5.3 `push_front`

This algorithm is more similar to the `insert` function (3.5.1) than the `push_back` algorithm, but is still relatively simple.

- If the head node is empty, increment the node's count.
- If the head node is full, you will need to shift the elements up, in the same way they were shifted in the insert function, making sure to track the overflow.
- Next you will need to split the node. In order to account for splitting the only node in the linked list, you will have to update the `tail_` pointer as necessary.
- If the head node isn't full yet, just shift the head node up an element from element 0 and increase the count.
- Set the 0'th element of the head to the value.

3.6 Element Removal

3.6.1 `erase`

This function uses the `findElement` helper function I have detailed in Recommended Helper Functions (4). Having implemented that, the function itself is relatively simple. You can use the `pop_back` and `pop_front` functions if the index requested is the first or last element.

- First, find the containing node and local index of the requested global index.
- Shift all the elements in the node beyond the local index left one element, covering the element being erased. This is done with the `shiftDown` function detailed in Recommended Helper Functions (4). Make sure to account for the node being only one element large.
- Decrement the node's count.

3.6.2 `pop_back`

Decrement the count of the tail node.

3.6.3 `pop_front`

Shift all elements in the head node down one element. Decrement the head's count.
If the head node is now empty, free the associated memory.

3.7 Element Access

As the const and non-const versions of these functions are algorithmically and syntactically identical, I would recommend (may Shilling forgive me) copypasting the code from one to the other. You can't call one from inside the other, as that would violate the const requirements.

3.7.1 `operator[]`

Find the containing node and local index of the index passed in. Like insert and erase, this is easily done with the `findElement` helper function I detailed in Recommended Helper Functions (4).
Return the element at the local index of the containing node.

3.7.2 `first`

This is one of the easiest functions in this assignment.
Return the first element of the head node.

3.7.3 `last`

This is also an easy function.
Return the last element in the tail node.

3.8 Data Structure Information

3.8.1 `find`

Walk the list in a similar fashion to that detailed in the `findElement` helper function (4), but check equivalence for each element in each node, returning the index when the desired element is found.

If the desired element is not found, return the total number of elements contained in the data structure.

3.8.2 `size`

You should be tracking the `size_` member variable throughout the element addition and removal processes. Return that variable now.

3.9 Data Structure Control

3.9.1 `clear`

This is a relatively simple function with a similar algorithm to the class destructor.

- First, loop through the list, freeing each node in turn.
- Once the list is empty, update the necessary member variables.

3.10 `compact`

Compact takes all the data stored in the linked list and moves it into the smallest number of nodes possible. Then it frees all empty nodes at the end of the list.

The algorithm for this walks the list at two points in parallel. I will refer to the walker having elements shifted into it as the left foot and the walker reading through the elements to shift as the right foot.

- First, loop through the list with both feet until the left node is not already full and the right foot hasn't walked off the list.
- Next, walk through the list while the right foot hasn't lost the list.
- This loop should first store the count of elements in the right node, then set the count of the right node to zero. This allows the left foot to update it as it is given elements to store.
- This loop should have a nested loop that stores each value from the right foot in the left foot's node, and steps the left foot to the next node when it is filled.

```
 --------------------------------------
|.-KEY--------------------------------.|
|| [node]   [values]   [count]        ||
||   |         |           |  [next]  ||
||   v         v           v    |     ||
||  +-+-node-------------+---+  v     ||
||  |#|[_][_][_][_][_][_]|(_)| --->   ||
||  +-+------------------+---+        ||
||     0  1  2  3  4  5  <-[indices]  ||
||____________________________________||
'--------------------------------------'
```

1. The starting position of the inner loop, transfering right's first element into left's fifth space

```
                                [rpos = 0]
                                   |                          [rtot = 3]
+-+-left-------------+---+      +-+-right------------+---+      +-+-...
|0|[a][b][c][d][_][_]|(4)| ---> |1|[e][f][g][_][_][_]|(0)| ---> |2|[...
+-+------------------+---+      +-+------------------+---+      +-+-...
   0  1  2  3  4  5                0  1  2  3  4  5
```

2. Next important point, the left foot is now full. Set the left foot to the next node in the list and continue the algorithm

```
                                      [rpos = 2]
                                         |                    [rtot = 3]
+-+-left-------------+---+      +-+-right------------+---+      +-+-...
|0|[a][b][c][d][e][f]|(6)| ---> |1|[_][_][g][_][_][_]|(0)| ---> |2|[...
+-+------------------+---+      +-+------------------+---+      +-+-...
   0  1  2  3  4  5                0  1  2  3  4  5
```

3. Left and right feet are now standing on the same node, but the algorithm still works because it is based on index in the node rather than the node itself.

```
      [rpos = 2]
          |                                                [rtot = 3]
+-+-left/right-------+---+      +-+-right->next------+---+      +-+-...
|1|[_][_][g][_][_][_]|(0)| ---> |2|[h][i][j][k][_][_]|(4)| ---> |3|[...
+-+------------------+---+      +-+------------------+---+      +-+-...
   0  1  2  3  4  5                0  1  2  3  4  5
```

4. The left foot has had the element at rpos shifted down, and the right foot has stepped to the next node. Note that rtot has been updated to 4, the right node has had its count zeroed out, and rtot has been updated to the right node's original count.

```
                                       [rpos = 1]
                                           |                        [rtot = 4]
+-+-left-------------+---+      +-+-right------------+---+       +-+-...
|1|[g][h][_][_][_][_]|(2)| ---> |2|[_][i][j][k][_][_]|(0)| ---> |3|[...
+-+------------------+---+      +-+------------------+---+       +-+-...
    0  1  2  3  4  5                0  1  2  3  4  5
```

Once the deque has been compacted, remove all the extra nodes from the end of the list.

4 Recommended Helper Functions

4.0.1 `clear`

This is a relatively simple function with a similar algorithm to the class destructor.

- First, loop through the list, freeing each node in turn.
- Once the list is empty, update the necessary member variables.