COMPSCI 221: Programming Assignment 3 #1

Due on Monday, February 26, 2017

Leyk 4:20pm

Robert Quan

Problem 1

Listing 1 shows the Copy Constructor for the TemplateDoublyLinkedList.h

Listing 1: Copy Constructor for Doubly Linked List from TemplateDoublyLinkedList.h

```
// copy constructor
template <typename T>
DoublyLinkedList<T>::DoublyLinkedList(const DoublyLinkedList<T>& dll)
{
  // Initialize the list
  header.next = &trailer;     //Make an empty list
  trailer.prev = &header;

  DListNode<T>* temp = dll.getFirst();
  while(temp != dll.getAfterLast()){
    this->insertLast(temp->getElem());
    temp = temp->getNext();
  }
}
```

1) As show, if we compute the big-O of the function, we find the while loop makes our code run through every element in the Doubly Linked List. This leads us to conclude

$$\boxed{\mathcal{O}(n)} \tag{1}$$

A linear complexity.
Listing 2 shows the Assignment Constructor for the DoublyLinkedList.

Listing 2: Assignemnt Operator for Doubly Linked List from TemplateDoublyLinkedList.h

```
// assignment operator
template <typename T>
DoublyLinkedList<T>& DoublyLinkedList<T>::operator=(const DoublyLinkedList<T>& dll)
{
  if(!isEmpty()){        //Cheack if list is not empty
    while(header.next != &trailer){
        removeFirst();
    }
  }
  DListNode<T>* temp = dll.getFirst();  //Create a temporary node
  while(temp != dll.getAfterLast()){
   this->insertLast(temp->getElem());   //Copy each node over
   temp = temp->getNext();
    }
  return *this;     //Return pointer to new assigned list
}
```

2) We first check whether our doubly linked list is empty. If it is not, we proceed to remove each element of the first list so that we can then assign the elements of the second list to the first. The second while loop will run through each node of the second list and create new nodes for the first list.
Wee see the time complexity to be:

$$\boxed{\mathcal{O}(n)} \tag{2}$$

A linear complexity.

Listing 3: insertFirst() and insertLast() for Doubly Linked List from TemplateDoublyLinkedList.h

```
// insert the object to the first of the linked list
template <typename T>
void DoublyLinkedList<T>::insertFirst(T newobj)
{
  DListNode<T> *newNode = new DListNode<T>(newobj, &header, header.next);
  header.next->prev = newNode;     //Insert a new node into the list
  header.next = newNode;

}

// insert the object to the last of the linked list
template <typename T>
void DoublyLinkedList<T>::insertLast(T newobj)
{
  DListNode<T> *newNode = new DListNode<T>(newobj, trailer.prev,&trailer);
  trailer.prev->next = newNode;    //Insert at the end of the Doubly Linked List.
  trailer.prev = newNode;
}
```

3) This time complexity is a simple

$$\boxed{\mathcal{O}(1)} \tag{3}$$

A constant complexity. This is because we do not need to run through the elements to find the last one. The trailer and header pointers gives us the ability to quickly insert nodes before and/or after these pointer.

Listing 4: removeFirst() and removeLast() for Doubly Linked List from TemplateDoublyLinkedList.h

```
// remove the first object of the list
template <typename T>
T DoublyLinkedList<T>::removeFirst()
{
  if (isEmpty())
    throw EmptyDLinkedListException("Empty Doubly Linked List");
  DListNode<T> *node = header.next;          //Reference the deleted node
  node->next->prev = &header;           //move to next element; repointer
  header.next = node->next;             //Change header to second node
  T obj = node->obj;
  delete node;
  return obj;
}

// remove the last object of the list
template <typename T>
T DoublyLinkedList<T>::removeLast()
{
  if (isEmpty())
    throw EmptyDLinkedListException("Empty Doubly Linked List");
  DListNode<T> *node = trailer.prev;
  node->prev->next = &trailer;
  trailer.prev = node->prev;
  T obj = node->obj;
```

```
  delete node;
  return obj;
}
```

4) The time complexity for these two functions will be a simple

$$\boxed{\mathcal{O}(1)} \tag{4}$$

Because we do not need to iterate through every element of the list to choose the position we want.

Listing 5: Destructor function for Doubly Linked List from TemplateDoublyLinkedList.h

```
// destructor
template <typename T>
DoublyLinkedList<T>::~DoublyLinkedList<T>()
{
  DListNode<T> *prev_node, *node = header.next;   //Create two pointers
  while (node != &trailer) {
    prev_node = node;    //Assigne prev_node to point at node
    node = node->next;
    delete prev_node;    //Go deleting the nodes at pre_node
  }
  header.next = &trailer;
  trailer.prev = &header;
}
```

5) The destructor must run through every element of the doubly linked list to delete the nodes and elements from the list. The time complexity for the while loop in this function will be linear:

$$\boxed{\mathcal{O}(n)} \tag{5}$$

Listing 6: first() and last() for Doubly Linked List from TemplateDoublyLinkedList.h

```
// return the first object
template <typename T>
T DoublyLinkedList<T>::first() const
{
  if (isEmpty())
    throw EmptyDLinkedListException("Empty Doubly Linked List");
  return header.next->obj;
}

// return the last object
template <typename T>
T DoublyLinkedList<T>::last() const
{
  if (isEmpty())
    throw EmptyDLinkedListException("Empty Doubly Linked List");
  return trailer.prev->obj;
}
```

6) The header and trailer pointers give us a quick access to the first and last elements in the Doubly Linked List. We do not need to run though the elements in the list. The time complexity of these two functions will be constant:

$$\boxed{\mathcal{O}(1)} \tag{6}$$

Listing 7: operator¡¡ for Doubly Linked List from TemplateDoublyLinkedList.h

```
// output operator
template <typename T>
ostream& operator<<(ostream& out, const DoublyLinkedList<T>& dll)
{
  DListNode<T>* temp = dll.getFirst();  //Make temp point to the first element
  while(temp != dll.getAfterLast()){    //Iterate through the list
    out << temp->getElem() << " "; //Output the elements in the nodes
    temp = temp->getNext();
  }
  return out;
}
```

7) We see that the while loop depends on the number of nodes in our Doubly Linked List. Therefor the time complexity will be:

$$\boxed{\mathcal{O}(n)} \tag{7}$$

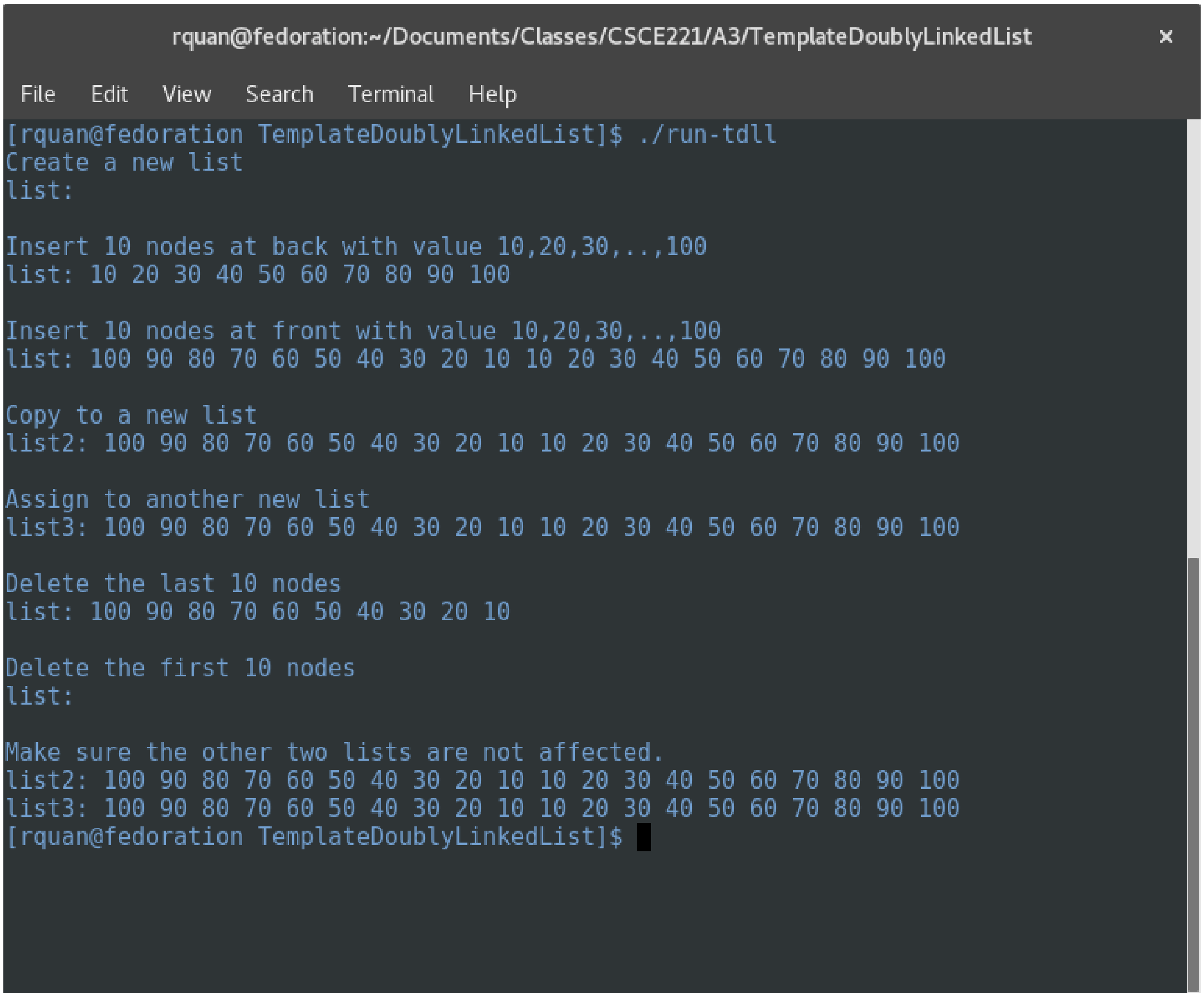

Figure 1: Screenshot of the Templated DoublyLinkedList

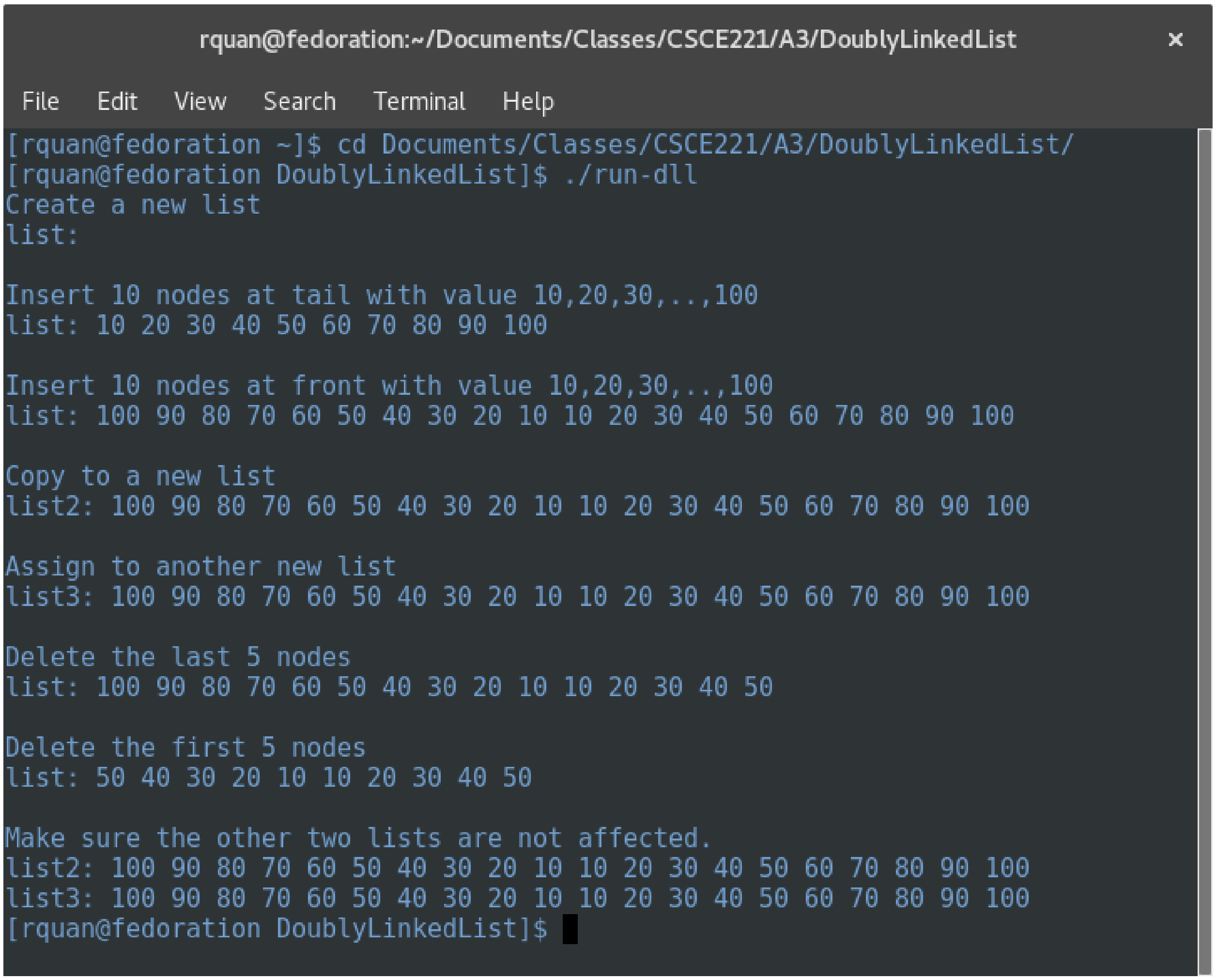

Figure 2: Screenshot of the DoublyLinkedList.cpp