THE UNIVERSITY
OF THE WITWATERSRAND

Lecturer: **Ritesh Ajoodha**

Laboratory 5
Data Structures and Algorithms
Stacks; Queues; and Design Patterns
26 August, 2015
Due: 2 September, 2015 @ 14:00

In this lab you may use the skeleton code I have provided on Moodle, however sometimes its better to type everything out yourself to that the concepts stick in your head :D

Question 1 Recall the linked-list class interface you implemented in Lab 3. Rewrite the *StringNode* and *StringLinkedList* classes using the **template design pattern**, your new implementation **must** follow the following class interface:

```
#include <string>
#include <iostream>
using namespace std;

template <typename E>
class Node{
   public:
      E elem;
      Node* next;
};

template <typename E>
class LinkedList{
   public:
      LinkedList();
      ~LinkedList();
      bool isEmpty() const;
      const E& front() const;
      const E& back() const;
      void addFront(const E& e);
      void removeFront();
      void addBack(const E& s);
      void removeBack();
      friend ostream& operator << (ostream& out, const LinkedList<E>& obj){
         Node<E>* temp = obj.head;
         if(temp == NULL){out << "[]"; return out;}
         out << "[";
         while(temp != NULL){
            out << temp->elem;
```

```
            if(temp->next != NULL){ out << "] --> [";}
            temp = temp->next;
         }
         out << "]";
         return out;
      }
   private:
      Node<E>* head;
};
```

Input Main:

```
int main(void){
        LinkedList<string>* myList = new LinkedList<string>();
        cout<< *myList << endl;
        //Adding to the front
        cout << myList->isEmpty() << endl;
        myList->addFront("Gandalf");
        cout<< *myList << endl;
        myList->addFront("Aragorn");
        cout<< *myList << endl;
        myList->addFront("Legolas");
        cout<< *myList << endl;
        cout << "Front element: \t"<< myList->front() << endl;
        cout << "Back element: \t"<< myList->back() << endl;
        //Removing from the front
        myList->removeFront();
        cout<< *myList << endl;
        myList->removeFront();
        cout<< *myList << endl;
        myList->removeFront();
        cout<< *myList << endl;
        myList->removeFront();
        //Adding to the back
        myList->addBack("Gollum");
        cout<< *myList << endl;
        myList->addBack("Bilbo Baggins");
        cout<< *myList << endl;
        myList->addBack("Saruman");
        cout<< *myList << endl;
        cout << "Front element: \t"<< myList->front() << endl;
        cout << "Back element: \t"<< myList->back() << endl;
        //Removing from the back
        myList->removeBack();
        cout<< *myList << endl;
        myList->removeBack();
        cout<< *myList << endl;
        myList->removeBack();
        cout<< *myList << endl;
```

```
38          return 0;
39  }
```

Program Output:

```
[]
1
[Gandalf]
[Aragorn] --> [Gandalf]
[Legolas] --> [Aragorn] --> [Gandalf]
Front element:  Legolas
Back element:  Gandalf
[Aragorn] --> [Gandalf]
[Gandalf]
[]
[Gollum]
[Gollum] --> [Bilbo Baggins]
[Gollum] --> [Bilbo Baggins] --> [Saruman]
Front element:  Gollum
Back element:  Saruman
[Gollum] --> [Bilbo Baggins]
[Gollum]
[]
```

Question 2 Implement the *RuntimeException Super-class* that we did in lectures to provide error messages for possible exceptions that could be given by any of the function names marked by "*throw*" in the *linked list* class interface below:

```
1  class RuntimeException{
2     private:
3        string errorMsg;
4     public:
5        RuntimeException(const string& err){errorMsg = err;}
6        string getMessage() const {return errorMsg;}
7  };
8
9  class LinkedListEmpty : public RuntimeException {
10    public:
11       LinkedListEmpty(const string& err) : RuntimeException(err) { }
12 };
13
14 template <typename E>
15 class Node{
16    public:
17       E elem;
18       Node* next;
19 };
20
21 template <typename E>
```

```
class LinkedList{
   public:
      LinkedList();
      ~LinkedList();
      bool isEmpty() const;
      const E& front() const throw(LinkedListEmpty);
      const E& back() const throw(LinkedListEmpty);
      void addFront(const E& e);
      void removeFront() throw(LinkedListEmpty);
      void addBack(const E& s);
      void removeBack() throw(LinkedListEmpty);
      friend ostream& operator << (ostream& out, const LinkedList<E>& obj){
         Node<E>* temp = obj.head;
         if(temp == NULL){out << "[]"; return out;}
         out << "[";
         while(temp != NULL){
            out << temp->elem;
            if(temp->next != NULL){ out << "] --> [";}
            temp = temp->next;
         }
         out << "]";
         return out;
      }
   private:
      Node<E>* head;
};
```

Input Main:

```
int main(void){
        LinkedList<string>* myList = new LinkedList<string>();
        cout<< *myList << endl;
        //Adding to the front
        cout << myList->isEmpty() << endl;
        myList->addFront("Gandalf");
        cout<< *myList << endl;
        myList->addFront("Aragorn");
        cout<< *myList << endl;
        myList->addFront("Legolas");
        cout<< *myList << endl;
        cout << "Front element: \t"<< myList->front() << endl;
        cout << "Back element: \t"<< myList->back() << endl;
        //Removing from the front
        myList->removeFront();
        cout<< *myList << endl;
        myList->removeFront();
        cout<< *myList << endl;
        myList->removeFront();
        cout<< *myList << endl;
```

```
        //Should be able to handle this
        myList->removeFront();
        //Adding to the back
        myList->addBack("Gollum");
        cout<< *myList << endl;
        myList->addBack("Bilbo Baggins");
        cout<< *myList << endl;
        myList->addBack("Saruman");
        cout<< *myList << endl;
        cout << "Front element: \t"<< myList->front() << endl;
        cout << "Back element: \t"<< myList->back() << endl;
        //Removing from the back
        myList->removeBack();
        cout<< *myList << endl;
        myList->removeBack();
        cout<< *myList << endl;
        myList->removeBack();
        cout<< *myList << endl;
        //Should be able to handle this
        myList->removeBack();
        cout<< *myList << endl;
        return 0;
}
```

Program Output:

```
[]
1
[Gandalf]
[Aragorn] --> [Gandalf]
[Legolas] --> [Aragorn] --> [Gandalf]
Front element:  Legolas
Back element:  Gandalf
[Aragorn] --> [Gandalf]
[Gandalf]
[]
Removing the front of an empty linked list
[Gollum]
[Gollum] --> [Bilbo Baggins]
[Gollum] --> [Bilbo Baggins] --> [Saruman]
Front element:  Gollum
Back element:  Saruman
[Gollum] --> [Bilbo Baggins]
[Gollum]
[]
Removing the back of an empty linked list
[]
```

Question 3 Implement the 2 outstanding *housekeeping functions*: the *copy-constructor* and the

assignment copy-constructor. Use the Queues example done in class as a guide.

```
template <typename E>
class LinkedList{
   public:
      LinkedList();
      LinkedList(const LinkedList& obj);
      LinkedList& operator= (const LinkedList& obj);
      ~LinkedList();
      bool isEmpty() const;
      const E& front() const throw(LinkedListEmpty);
      const E& back() const throw(LinkedListEmpty);
      void addFront(const E& e);
      void removeFront() throw(LinkedListEmpty);
      void addBack(const E& s);
      void removeBack() throw(LinkedListEmpty);
      friend ostream& operator << (ostream& out, const LinkedList<E>& obj){
         Node<E>* temp = obj.head;
         if(temp == NULL){out << "[]"; return out;}
         out << "[";
         while(temp != NULL){
            out << temp->elem;
            if(temp->next != NULL){ out << "] --> [";}
            temp = temp->next;
         }
         out << "]";
         return out;
      }
   private:
      Node<E>* head;
};
```

Input Main:

```
int main(void){
        LinkedList<string>* myList = new LinkedList<string>();
        cout<< *myList << endl;
        //Adding to the front
        cout << myList->isEmpty() << endl;
        myList->addFront("Gandalf");
        cout<< *myList << endl;
        myList->addFront("Aragorn");
        cout<< *myList << endl;
        myList->addFront("Legolas");
        cout<< *myList << endl;
        cout << "Front element: \t"<< myList->front() << endl;
        cout << "Back element: \t"<< myList->back() << endl;
        //Removing from the front
        myList->removeFront();
        cout<< *myList << endl;
```

```
17          myList->removeFront();
18          cout<< *myList << endl;
19          myList->removeFront();
20          cout<< *myList << endl;
21          //Should be able to handle this
22          myList->removeFront();
23          //Adding to the back
24          myList->addBack("Gollum");
25          cout<< *myList << endl;
26          myList->addBack("Bilbo␣Baggins");
27          cout<< *myList << endl;
28          myList->addBack("Saruman");
29          cout << "␣1:␣"<<  *myList << endl;
30          LinkedList<string>* myList2 = new LinkedList<string>(*myList);
31          cout << "␣2:␣"<< *myList2 << endl;
32          LinkedList<string>* myList3 = new LinkedList<string>();
33          *myList3 = *myList;
34          cout << "␣3:␣"<< *myList3 << endl;
35          cout << "Front␣element:␣\t"<< myList->front() << endl;
36          cout << "Back␣element:␣\t"<< myList->back() << endl;
37          //Removing from the back
38          myList->removeBack();
39          cout<< *myList << endl;
40          myList->removeBack();
41          cout<< *myList << endl;
42          myList->removeBack();
43          cout<< *myList << endl;
44          //Should be able to handle this
45          myList->removeBack();
46          cout<< *myList << endl;
47          return 0;
48  }
```

Program Output:

```
[]
1
[Gandalf]
[Aragorn] --> [Gandalf]
[Legolas] --> [Aragorn] --> [Gandalf]
Front element:  Legolas
Back element:  Gandalf
[Aragorn] --> [Gandalf]
[Gandalf]
[]
Removing the front of an empty linked list
[Gollum]
[Gollum] --> [Bilbo Baggins]
1: [Gollum] --> [Bilbo Baggins] --> [Saruman]
```

```
2: [Gollum] --> [Bilbo Baggins] --> [Saruman]
3: [Gollum] --> [Bilbo Baggins] --> [Saruman]
Front element:  Gollum
Back element:  Saruman
[Gollum] --> [Bilbo Baggins]
[Gollum]
[]
Removing the back of an empty linked list
[]
```

Question 4 Another very useful design pattern is called the **Adapter design pattern** where already implemented data structures (like linked-lists) are used to create other data structures (like stacks and queues). Implement a queue and stack using the linked list class interface below (you may reuse code from the previous questions). NB: Take note of the getNode() method in the linked list class, I have implemented this for you already.

```cpp
#include <string>
#include <iostream>
using namespace std;

class RuntimeException{
   private:
      string errorMsg;
   public:
      RuntimeException(const string& err){errorMsg = err;}
      string getMessage() const {return errorMsg;}
};

class LinkedListEmpty : public RuntimeException {
   public:
      LinkedListEmpty(const string& err) : RuntimeException(err) { }
};

class QueueEmpty : public RuntimeException {
   public:
      QueueEmpty(const string& err) : RuntimeException(err) { }
};

class StackEmpty : public RuntimeException {
   public:
      StackEmpty(const string& err) : RuntimeException(err) { }
};

template <typename E>
   class Node{
      public:
         E elem;
         Node* next;
};
```

```
34
35 template <typename E>
36 class LinkedList{
37    public:
38       LinkedList();
39       LinkedList(const LinkedList& obj);
40       LinkedList& operator= (const LinkedList& obj);
41       ~LinkedList();
42       Node<E>* getNode(const int n) const;
43       int size() const;
44       bool isEmpty() const;
45       const E& front() const throw(LinkedListEmpty);
46       const E& back() const throw(LinkedListEmpty);
47       void addFront(const E& e);
48       void removeFront() throw(LinkedListEmpty);
49       void addBack(const E& s);
50       void removeBack() throw(LinkedListEmpty);
51       friend ostream& operator << (ostream& out, const LinkedList<E>& obj){
52          Node<E>* temp = obj.head;
53          if(temp == NULL){out << "[]"; return out;}
54          out << "[";
55          while(temp != NULL){
56             out << temp->elem;
57             if(temp->next != NULL){ out << "] --> [";}
58             temp = temp->next;
59          }
60          out << "]";
61          return out;
62       }
63    private:
64       int numberOfElements;
65       Node<E>* head;
66 };
67
68 template <typename E>
69 class Queue{
70    public:
71       Queue();
72       Queue(const Queue& obj);
73       Queue& operator= (const Queue& obj);
74       ~Queue();
75       const int size() const;
76       bool empty() const;
77       const E& front() const throw(QueueEmpty);
78       void enqueue(const E& e);
79       void dequeue() throw(QueueEmpty);
80       friend ostream& operator << (ostream& out, const Queue<E>& obj){
81          if(((obj.myQueue)->size()) > 0){
82             Node<E>* temp = (obj.myQueue)->getNode(0);
```

```
            if(temp == NULL){out << "[]"; return out;}
            out << "[";
            while(temp != NULL){
               out << temp->elem;
               if(temp->next != NULL){ out << "] --> [";}
               temp = temp->next;
            }
            out << "]";
            return out;
         }
         else{
            out << "[]";
            return out;
         }
      }
   private:
      LinkedList<E>* myQueue;
      int sizeOfQueue;
};


template <typename E>
class Stack{
   public:
      Stack();
      Stack(const Stack& obj);
      Stack& operator= (const Stack& obj);
      ~Stack();
      const int size() const;
      bool empty() const;
      const E& top() const throw(StackEmpty);
      void push(const E& e);
      void pop() throw(StackEmpty);
      friend ostream& operator << (ostream& out, const Stack<E>& obj){
         if(((obj.myStack)->size()) > 0){
            Node<E>* temp = (obj.myStack)->getNode(0);
            if(temp == NULL){out << "[]"; return out;}
            out << "[";
            while(temp != NULL){
               out << temp->elem;
               if(temp->next != NULL){ out << "] --> [";}
               temp = temp->next;
            }
            out << "]";
            return out;
         }
         else{
            out << "[]";
            return out;
```

```
132             }
133         }
134     private:
135         LinkedList<E>* myStack;
136         int sizeOfStack;
137 };
138 
139 template <typename E>
140 Node<E>* LinkedList<E>::getNode(const int n) const{
141     Node<E>* temp = head;
142     if(n <= numberOfElements-1){
143         for(int i =0; i < n; i++ ){
144             temp = temp->next;
145         }
146         return temp;
147     }
148 }
```

Input Main:

```
1  int main(void){
2          LinkedList<string>* myList = new LinkedList<string>();
3          cout << myList->size() << endl;
4          cout<< *myList << endl;
5          //Adding to the front
6          cout << myList->isEmpty() << endl;
7          myList->addFront("Gandalf");
8          cout<< *myList << endl;
9          myList->addFront("Aragorn");
10         cout<< *myList << endl;
11         cout << myList->size() << endl;
12         myList->addFront("Legolas");
13         cout<< *myList << endl;
14         Node<string>* myNode = myList->getNode(0);
15         cout << myNode->elem << endl;
16         cout << myList->size() << endl;
17         cout << "Front element: \t"<< myList->front() << endl;
18         cout << "Back element: \t"<< myList->back() << endl;
19         //Removing from the front
20         myList->removeFront();
21         cout<< *myList << endl;
22         cout << myList->size() << endl;
23         myList->removeFront();
24         cout<< *myList << endl;
25         cout << myList->size() << endl;
26         myList->removeFront();
27         cout<< *myList << endl;
28         cout << myList->size() << endl;
29         //Should be able to handle this
```

```
30          myList->removeFront();
31          //Adding to the back
32          myList->addBack("Gollum");
33          cout<< *myList << endl;
34          cout << myList->size() << endl;
35          myList->addBack("Bilbo Baggins");
36          cout<< *myList << endl;
37          cout << myList->size() << endl;
38          myList->addBack("Saruman");
39          cout << " 1: "<<  *myList << endl;
40          LinkedList<string>* myList2 = new LinkedList<string>(*myList);
41          cout << " 2: "<< *myList2 << endl;
42          LinkedList<string>* myList3 = new LinkedList<string>();
43          *myList3 = *myList;
44          cout << " 3: "<< *myList3 << endl;
45          cout << "Front element: \t"<< myList->front() << endl;
46          cout << "Back element: \t"<< myList->back() << endl;
47          //Removing from the back
48          myList->removeBack();
49          cout<< *myList << endl;
50          cout << myList->size() << endl;
51          myList->removeBack();
52          cout<< *myList << endl;
53          cout << myList->size() << endl;
54          myList->removeBack();
55          cout<< *myList << endl;
56          cout << myList->size() << endl;
57          //Should be able to handle this
58          myList->removeBack();
59          cout<< *myList << endl;
60          cout << myList->size() << endl;
61
62          Queue<string>* myQueue = new Queue<string>();
63          myQueue->enqueue("Ritso");
64          myQueue->enqueue("Ritesh");
65          myQueue->enqueue("Ajoodha");
66          cout << *myQueue << endl;
67          myQueue->dequeue();
68          cout << *myQueue << endl;
69          myQueue->dequeue();
70          cout << *myQueue << endl;
71          myQueue->dequeue();
72          cout << *myQueue << endl;
73          myQueue->dequeue();
74          cout << *myQueue << endl;
75          myQueue->dequeue();
76
77          Stack<string>* myStack = new Stack<string>();
78          myStack->push("Ritso");
```

```
79          myStack->push("Ritesh");
80          myStack->push("Ajoodha");
81          cout << *myStack << endl;
82          myStack->pop();
83          cout << *myStack << endl;
84          myStack->pop();
85          cout << *myStack << endl;
86          myStack->pop();
87          cout << *myStack << endl;
88          myStack->pop();
89          cout << *myStack << endl;
90          myStack->pop();
91
92          return 0;
93  }
```

Program Output:

```
0
[]
1
[Gandalf]
[Aragorn] --> [Gandalf]
2
[Legolas] --> [Aragorn] --> [Gandalf]
Legolas
3
Front element:  Legolas
Back element:  Gandalf
[Aragorn] --> [Gandalf]
2
[Gandalf]
1
[]
0
Removing the front of an empty linked list
[Gollum]
1
[Gollum] --> [Bilbo Baggins]
2
1: [Gollum] --> [Bilbo Baggins] --> [Saruman]
2: [Saruman] --> [Bilbo Baggins] --> [Gollum]
3: [Gollum] --> [Bilbo Baggins] --> [Saruman]
Front element:  Gollum
Back element:  Saruman
[Gollum] --> [Bilbo Baggins]
2
[Gollum]
1
```

```
[]
0
Removing the back of an empty linked list
[]
0
[Ajoodha] --> [Ritesh] --> [Ritso]
[Ajoodha] --> [Ritesh]
[Ajoodha]
[]
Removing the back of an empty linked list
[]
Removing the back of an empty linked list
[Ajoodha] --> [Ritesh] --> [Ritso]
[Ritesh] --> [Ritso]
[Ritso]
[]
Removing the front of an empty linked list
[]
Removing the front of an empty linked list
```