```
****************************************************************************
* PROGRAMMED BY : Nick Reardon
* CLASS         : CS1D
* SECTION       : MW - 2:30p
* Assignment #3 : Stacks, Queues, Dequeus
****************************************************************************

                    Assignment #3 - Stacks, Queues, Dequeus

Given the following data:

Input for the string stacks/queues/deques
Mark, Alan, Jennifer, Jordyn, Eric, JoAnn, Bryan

Input for the double stacks/queues/deques
2019.1, 44.44, 888.55, 200.12, 123.123, 8.445,

A. Implement and print (top of stack to bottom) the stacks
   using the STL <stack> with the above data.
B. Delete Jordyn and 200.12 from the above stacks (you will
   need to delete others) using the STL <stack> and print (top of
   stack to bottom) the remaining elements in the stacks.
C. Implement and print (top of stack to bottom) the stacks
   using a singly linked list using the above data.  Do not use the
   STL.
D. Delete Jordyn and 200.12 from the above stacks (you will
   need to delete others) and print (top of stack to bottom) the
   remaining elements in the stacks. Do not use the STL.
E. Implement and print the queues using either a circular array
   or a linked list using the above data.  Do not use the STL.
F. Delete JoAnn and 200.12 from the above queues (you will
   need to delete others) and print the remaining elements in
   the queues. Do not use the STL.
G. Implement and print the deques using a linked list using the
   above data (using push front).    Do not use the STL.
H. Delete JoAnn (pop front) and 200.12 (pop back) from the
   above deques (you will need to delete others) and print the
   remaining elements in the deques. Do not use the STL.

Label your output (part A, part B, part C, etc.)
Do not put deleted elements back on the data structures.

I. Implement the Parentheses Algorithm without using the
   STL).  Test your algorithm with the following mathematical
   statements.
   a. (12x + 6) (2x - 4)
   b. {2x + 5} (6x+4)
   c. {2x + 7) (12x + 6)
   d. {{8x+5) - 5x[9x+3]})
   e. (((4x+8) - x[4x+3])))
   f. [(5x - 5) - 4x[6x + 2]]
   g. {(8x+5) - 6x[9x+3]]
```

```
53
54 J. (extra credit û 3 points) If valid, write software to evaluate
55    the valid expressions above assuming x = -2.
56
57 Your output should CLEARLY demonstrate the above.  Print out
58 the part number before you display the stacks/queues/deques.
59
60 Due on February 3rd
61
62
63 ********************************************************************************
64
65 Reading from file into string stacks/queue/deque
66 Reading from file into double stacks/queue/deque
67
68    --- PART A ---
69
70 Printing STL stack:
71 Bryan
72 JoAnn
73 Eric
74 Jordyn
75 Jennifer
76 Alan
77 Mark
78
79 Printing STL stack:
80 8.445
81 123.123
82 200.12
83 888.55
84 44.44
85 2019.1
86
87
88    --- PART B ---
89
90 Deleting Jordyn from the STL stack
91 deleting Bryan
92 deleting JoAnn
93 deleting Eric
94 deleting Jordyn
95
96 Deleting 200.12 from the STL stack
97 deleting 8.445
98 deleting 123.123
99 deleting 200.12
100
101 Printing STL stack:
102 Jennifer
103 Alan
104 Mark
```

```
105
106 Printing STL stack:
107 888.55
108 44.44
109 2019.1
110
111
112    --- PART C ---
113
114 Printing singly linked list stacks:
115
116 Printing stack:
117 Bryan
118 JoAnn
119 Eric
120 Jordyn
121 Jennifer
122 Alan
123 Mark
124
125 Printing stack:
126 8.445
127 123.123
128 200.12
129 888.55
130 44.44
131 2019.1
132
133
134    --- PART D ---
135
136 Deleting Jordyn from the linked list stack
137 Popped item is Bryan
138 Popped item is JoAnn
139 Popped item is Eric
140 Popped item is Jordyn
141
142 Deleting 200.12 from the linked list stack
143 Popped item is 8.445
144 Popped item is 123.123
145 Popped item is 200.12
146
147 Printing stack:
148 Jennifer
149 Alan
150 Mark
151
152 Printing stack:
153 888.55
154 44.44
155 2019.1
156
```

```
157
158     --- PART E ---
159
160  Printing singly linked list queues:
161
162  Printing queue:
163  2019.1
164  44.44
165  888.55
166  200.12
167  123.123
168  8.445
169
170  Printing queue:
171  Mark
172  Alan
173  Jennifer
174  Jordyn
175  Eric
176  JoAnn
177  Bryan
178
179
180     --- PART F ---
181
182  Deleting Jordyn from the linked list queue
183  deQueued item is Mark
184  deQueued item is Alan
185  deQueued item is Jennifer
186  deQueued item is Jordyn
187
188  Deleting 200.12 from the linked list queue
189  deQueued item is 2019.1
190  deQueued item is 44.44
191  deQueued item is 888.55
192  deQueued item is 200.12
193
194  Printing queue:
195  123.123
196  8.445
197
198  Printing queue:
199  Eric
200  JoAnn
201  Bryan
202
203
204     --- PART G ---
205
206  Printing doubly linked list deques:
207
208  Printing deque front to back:
```

```
Bryan
JoAnn
Eric
Jordyn
Jennifer
Alan
Mark

Printing deque front to back:
8.445
123.123
200.12
888.55
44.44
2019.1


   --- PART H ---

Deleting Jordyn, using pop front, from the doubly linked list deque
removing from front: Bryan
removing from front: JoAnn
removing from front: Eric
removing from front: Jordyn

Deleting 200.12, using pop back, from the doubly linked list deque
removing from back: 2019.1
removing from back: 44.44
removing from back: 888.55
removing from back: 200.12

Printing deque front to back:
Jennifer
Alan
Mark

Printing deque front to back:
8.445
123.123


   --- PART I ---

Testing Parentheses Algorithm - using singly linked list stack


(12x + 6) (2x - 4)
Balanced

{2x + 5} (6x+4)
Balanced

```

```
{2x + 7) (12x + 6)
Not Balanced

{{8x+5) - 5x[9x+3]})
Not Balanced

(((4x+8) - x[4x+3])))
Not Balanced

[(5x - 5) - 4x[6x + 2]]
Balanced

{(8x+5) - 6x[9x+3]]
Not Balanced
Press any key to continue . . .
```

```
/*****************************************************************************
 * AUTHOR           : Nick Reardon
 * Assignment #3    : Stacks, Queues, Dequeus
 * CLASS            : CS1D
 * SECTION          : MW - 2:30p
 * DUE DATE         : 02 / 03 / 20
 *****************************************************************************/
#ifndef _MAIN_H_
#define _MAIN_H_

//Standard includes
#include <iostream>
#include <iomanip>
#include <string>
#include "PrintHeader.h"

//Program Specific
#include<stack>
#include "stackType.h"
#include "linkedQueue.h"
#include "dequeType.h"



//Prints the contents of an STL stack
//Makes a copy and outputs top, then pops in a loop until empty
template <class Type>
void printStackSTL(std::stack<Type> stack)
{
    std::stack<Type> copy = stack;

    cout << "Printing STL stack:" << endl;
    while (copy.size() > 0)
    {
        cout << copy.top() << endl;
        copy.pop();
    }
    cout << endl;
}

//Checks a given strings for parenthesis
bool areParanthesisBalanced(const string& expression);

#endif // _HEADER_H_

```

```cpp
/*****************************************************************************
 * AUTHOR           : Nick Reardon
 * Assignment #3    : Stacks, Queues, Dequeus
 * CLASS            : CS1D
 * SECTION          : MW - 2:30p
 * DUE DATE         : 02 / 03 / 20
 *****************************************************************************/
#include "main.h"

using std::cout; using std::endl;


int main()
{

    /*
     * HEADER OUTPUT
     */
    PrintHeader(cout, "Prompt.txt");

    /*****************************************************************************/

    std::ifstream strFile;
    strFile.open("stringInput.txt");

    std::ifstream numFile;
    numFile.open("doubleInput.txt");

    std::stack<std::string> strStack_STL;
    std::stack<double> numStack_STL;

    linkedStackType<std::string> strStack;
    linkedStackType<double> numStack;

    linkedQueueType<std::string> strQueue;
    linkedQueueType<double> numQueue;

    DLinkedList<std::string> strDeque;
    DLinkedList<double> numDeque;

    cout << "Reading from file into string stacks/queue/deque" << endl;
    while (strFile)
    {
        string temp;
        getline(strFile, temp);
        if (temp != "")
        {
            strStack_STL.push(temp);
            strStack.push(temp);
            strQueue.addQueue(temp);
            strDeque.addFront(temp);
        }
```

```
53      }
54
55      cout << "Reading from file into double stacks/queue/deque" << endl;
56      while (numFile)
57      {
58          double temp = -99999999999999;
59          numFile >> temp;
60          if (temp != -99999999999999)
61          {
62              numStack_STL.push(temp);
63              numStack.push(temp);
64              numQueue.addQueue(temp);
65              numDeque.addFront(temp);
66          }
67      }
68
69      //************************************************
70
71
72      cout << endl << "   --- PART A ---" << endl << endl;
73
74
75      printStackSTL(strStack_STL);
76      printStackSTL(numStack_STL);
77
78
79      cout << endl << "   --- PART B ---" << endl << endl;
80
81
82      cout << "Deleting Jordyn from the STL stack" << endl;
83      std::string tempStr;
84      while (tempStr != "Jordyn")
85      {
86          tempStr = strStack_STL.top();
87          strStack_STL.pop();
88          cout << "deleting " << tempStr << endl;
89      }
90      cout << endl;
91
92      cout << "Deleting 200.12 from the STL stack" << endl;
93      double tempNum = -99999999999999999;
94      while (tempNum != 200.12)
95      {
96          tempNum = numStack_STL.top();
97          numStack_STL.pop();
98          cout << "deleting " << tempNum << endl;
99      }
100     cout << endl;
101
102     printStackSTL(strStack_STL);
103     printStackSTL(numStack_STL);
104
```

```
105
106     cout << endl << "   --- PART C ---" << endl << endl;
107
108
109     cout << "Printing singly linked list stacks: " << endl << endl;
110     strStack.printStack();
111     numStack.printStack();
112
113
114     cout << endl << "   --- PART D ---" << endl << endl;
115
116
117     cout << "Deleting Jordyn from the linked list stack" << endl;
118     tempStr.clear();
119     while (tempStr != "Jordyn")
120     {
121         strStack.pop(tempStr);
122     }
123     cout << endl;
124
125     cout << "Deleting 200.12 from the linked list stack" << endl;
126     tempNum = -99999999999999999;
127     while (tempNum != 200.12)
128     {
129         numStack.pop(tempNum);
130     }
131     cout << endl;
132
133     strStack.printStack();
134     numStack.printStack();
135
136
137     cout << endl << "   --- PART E ---" << endl << endl;
138
139     cout << "Printing singly linked list queues: " << endl << endl;
140
141     numQueue.printQueue();
142     strQueue.printQueue();
143
144
145     cout << endl << "   --- PART F ---" << endl << endl;
146
147
148     cout << "Deleting Jordyn from the linked list queue" << endl;
149     tempStr.clear();
150     while (tempStr != "Jordyn")
151     {
152         strQueue.deQueue(tempStr);
153     }
154     cout << endl;
155
156     cout << "Deleting 200.12 from the linked list queue" << endl;
```

```
157     tempNum = -99999999999999999;
158     while (tempNum != 200.12)
159     {
160         numQueue.deQueue(tempNum);
161     }
162     cout << endl;
163
164     numQueue.printQueue();
165     strQueue.printQueue();
166
167
168     cout << endl << "   --- PART G ---" << endl << endl;
169
170     cout << "Printing doubly linked list deques: " << endl << endl;
171
172     strDeque.printDeque();
173     numDeque.printDeque();
174
175
176     cout << endl << "   --- PART H ---" << endl << endl;
177
178
179     cout << "Deleting Jordyn, using pop front, from the doubly linked list deque"
            << endl;
180     tempStr.clear();
181     while (tempStr != "Jordyn")
182     {
183         tempStr = strDeque.front();
184         strDeque.removeFront();
185     }
186     cout << endl;
187
188     cout << "Deleting 200.12, using pop back, from the doubly linked list deque"
          << endl;
189     tempNum = -99999999999999999;
190     while (tempNum != 200.12)
191     {
192         tempNum = numDeque.back();
193         numDeque.removeBack();
194     }
195     cout << endl;
196
197     strDeque.printDeque();
198     numDeque.printDeque();
199
200
201     cout << endl << "   --- PART I ---" << endl << endl;
202
203     cout << "Testing Parentheses Algorithm - using singly linked list stack " <<
          endl << endl;
204
205     tempStr = "(12x + 6) (2x - 4)";
```

```
206     cout << endl << tempStr << endl;
207     if (areParanthesisBalanced(tempStr))
208         cout << "Balanced" << endl;
209     else
210         cout << "Not Balanced" << endl;
211
212     tempStr = "{2x + 5} (6x+4)";
213     cout << endl << tempStr << endl;
214     if (areParanthesisBalanced(tempStr))
215         cout << "Balanced" << endl;
216     else
217         cout << "Not Balanced" << endl;
218
219     tempStr = "{2x + 7) (12x + 6)";
220     cout << endl << tempStr << endl;
221     if (areParanthesisBalanced(tempStr))
222         cout << "Balanced" << endl;
223     else
224         cout << "Not Balanced" << endl;
225
226
227     tempStr = "{{8x+5) - 5x[9x+3]})";
228     cout << endl << tempStr << endl;
229     if (areParanthesisBalanced(tempStr))
230         cout << "Balanced" << endl;
231     else
232         cout << "Not Balanced" << endl;
233
234
235     tempStr = "(((4x+8) - x[4x+3])))";
236     cout << endl << tempStr << endl;
237     if (areParanthesisBalanced(tempStr))
238         cout << "Balanced" << endl;
239     else
240         cout << "Not Balanced" << endl;
241
242
243     tempStr = "[(5x - 5) - 4x[6x + 2]]";
244     cout << endl << tempStr << endl;
245     if (areParanthesisBalanced(tempStr))
246         cout << "Balanced" << endl;
247     else
248         cout << "Not Balanced" << endl;
249
250
251     tempStr = "{(8x+5) - 6x[9x+3]]";
252     cout << endl << tempStr << endl;
253     if (areParanthesisBalanced(tempStr))
254         cout << "Balanced" << endl;
255     else
256         cout << "Not Balanced" << endl;
257
```

```
258
259     system("pause");
260     return 0;
261
262
263
264
265 }
266
267 bool areParanthesisBalanced(const string& expression)
268 {
269     linkedStackType<char> s;
270     char x;
271
272     for (int i = 0; i < expression.length(); i++)
273     {
274         if (expression[i] == '(' ||
275             expression[i] == '[' ||
276             expression[i] == '{')
277         {
278             s.push(expression[i]);
279         }
280         else
281         {
282             if (!s.isEmptyStack())
283             {
284                 switch (s.top())
285                 {
286                 case '(':
287                     if (expression[i] == ')')
288                     {
289                         s.pop();
290                     }
291                     else if (expression[i] == ']' ||
292                         expression[i] == '}')
293                     {
294                         return false;
295                     }
296                     break;
297
298                 case '[':
299                     if (expression[i] == ']')
300                     {
301                         s.pop();
302                     }
303                     else if (expression[i] == ')' ||
304                         expression[i] == '}')
305                     {
306                         return false;
307                     }
308                     break;
309
```

```
310                 case '{':
311                     if (expression[i] == '}')
312                     {
313                         s.pop();
314                     }
315                     else if (expression[i] == ')' ||
316                         expression[i] == ']')
317                     {
318                         return false;
319                     }
320                     break;
321                 }
322             }
323             else
324             {
325                 if (expression[i] == ')' ||
326                     expression[i] == ']' ||
327                     expression[i] == '}')
328                 {
329                     return false;
330                 }
331             }
332         }
333     }
334     return s.isEmptyStack();
335 }
```

```
#ifndef H_StackType
#define H_StackType

#include <iostream>

using namespace std;

//Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type>* link;
};

template<class Type>
class linkedStackType
{
public:
    const linkedStackType<Type>& operator=
        (const linkedStackType<Type>&);
    //overload the assignment operator
    void initializeStack();
    //Initialize the stack to an empty state.
    //Post condition: Stack elements are removed; head = NULL
    bool isEmptyStack();
    //Function returns true if the stack is empty;
    //otherwise, it returns false
    bool isFullStack();
    //Function returns true if the stack is full;
    //otherwise, it returns false

    Type top();

    void push(const Type& newItem);
    //Add the newItem to the stack.
    //Pre condition: stack exists and is not full
    //Post condition: stack is changed and the newItem
    //     is added to the head of stack. head points to
    //     the updated stack
    void pop(Type& poppedElement);
    void pop();
    //Remove the head element of the stack.
    //Pre condition: Stack exists and is not empty
    //Post condition: stack is changed and the head
    //    element is removed from the stack. The head
    //    element of the stack is saved in poppedElement
    void destroyStack();
    //Remove all elements of the stack, leaving the
    //stack in an empty state.
    //Post condition: head = NULL
    linkedStackType();
```

```
53      //default constructor
54      //Post condition: head = NULL
55      linkedStackType(const linkedStackType<Type>& otherStack);
56      //copy constructor
57      ~linkedStackType();
58      //destructor
59      //All elements of the stack are removed from the stack
60
61      void printStack();
62
63  private:
64      nodeType<Type>* head; // pointer to the stack
65  };
66
67
68  template<class Type> //default constructor
69  linkedStackType<Type>::linkedStackType()
70  {
71      head = NULL;
72  }
73
74  template<class Type>
75  void linkedStackType<Type>::destroyStack()
76  {
77      nodeType<Type>* temp; //pointer to delete the node
78
79      while (head != NULL)  //while there are elements in the stack
80      {
81          temp = head;      //set temp to point to the current node
82          head = head->link; //advance head to the next node
83          delete temp;     //deallocate memory occupied by temp
84      }
85  }// end destroyStack
86
87
88
89  template<class Type>
90  void linkedStackType<Type>::initializeStack()
91  {
92      destroyStack();
93  }
94
95  template<class Type>
96  bool linkedStackType<Type>::isEmptyStack()
97  {
98      return(head == NULL);
99  }
100
101 template<class Type>
102 bool linkedStackType<Type>::isFullStack()
103 {
104     return 0;
```

```
105 }
106
107 template<class Type>
108 Type linkedStackType<Type>::top()
109 {
110     return head->info;
111 }
112
113 template<class Type>
114 void linkedStackType<Type>::push(const Type& newElement)
115 {
116     nodeType<Type>* newNode; //pointer to create the new node
117
118     newNode = new nodeType<Type>; //create the node
119     newNode->info = newElement;   //store newElement in the node
120     newNode->link = head;          //insert newNode before head
121     head = newNode;            //set head to point to the head node
122 } //end push
123
124
125 template<class Type>
126 void linkedStackType<Type>::pop(Type& poppedElement)
127 {
128     nodeType<Type>* temp;       //pointer to deallocate memory
129
130     poppedElement = head->info; //copy the head element into
131                                 //poppedElement
132     cout << "Popped item is " << poppedElement << endl;
133     temp = head;                    //set temp to point to the head node
134     head = head->link;              //advance head to the next node
135     delete temp;                    //delete the head node
136 }//end pop
137
138 template<class Type>
139 void linkedStackType<Type>::pop()
140 {
141     nodeType<Type>* temp;       //pointer to deallocate memory
142     temp = head;                    //set temp to point to the head node
143     head = head->link;              //advance head to the next node
144     delete temp;                    //delete the head node
145 }//end pop
146
147
148 template<class Type>   //copy constructor
149 linkedStackType<Type>::linkedStackType(const linkedStackType<Type>& otherStack)
150 {
151     nodeType<Type>* newNode, * current, * last;
152
153     if (otherStack.head == NULL)
154         head = NULL;
155     else
156     {
```

```
157         current = otherStack.head;  //set current to point to the
158                                     //stack to be copied
159
160             //copy the head element of the stack
161         head = new nodeType<Type>;   //create the node
162         head->info = current->info;  //copy the info
163         head->link = NULL;          //set the link field of the
164                                     //node to null
165         last = head;                 //set last to point to the node
166         current = current->link;    //set current to point to the
167                                     //next node
168
169             //copy the remaining stack
170         while (current != NULL)
171         {
172             newNode = new nodeType<Type>;
173             newNode->info = current->info;
174             newNode->link = NULL;
175             last->link = newNode;
176             last = newNode;
177             current = current->link;
178         }//end while
179     }//end else
180 }//end copy constructor
181
182
183 template<class Type> //destructor
184 linkedStackType<Type>::~linkedStackType()
185 {
186     nodeType<Type>* temp;
187
188     while (head != NULL)     //while there are elements in the stack
189     {
190         temp = head;        //set temp to point to the current node
191         head = head->link;  //advance first to the next node
192         delete temp;       //deallocate the memory occupied by temp
193     }//end while
194 }
195 //end destructor
196
197 template<class Type>
198 inline void linkedStackType<Type>::printStack()
199 {
200     cout << "Printing stack:" << endl;
201     nodeType<Type>* tempPtr = head;
202     for (nodeType<Type>* tempPtr = head; tempPtr != NULL; tempPtr = tempPtr-
        >link)
203     {
204         cout << tempPtr->info << endl;
205     }
206     cout << endl;
207 }
```

```
208
209
210 template<class Type>   //overloading the assignment operator
211 const linkedStackType<Type>& linkedStackType<Type>::operator=
212 (const linkedStackType<Type>& otherStack)
213 {
214     nodeType<Type>* newNode, * current, * last;
215
216     if (this != &otherStack) //avoid self-copy
217     {
218         if (head != NULL)  //if the stack is not empty, destroy it
219             destroyStack();
220
221         if (otherStack.head == NULL)
222             head = NULL;
223         else
224         {
225             current = otherStack.head;  //set current to point to
226                                         //the stack to be copied
227
228                 //copy the head element of otherStack
229             head = new nodeType<Type>;  //create the node
230             head->info = current->info; //copy the info
231             head->link = NULL;          //set the link field of the
232                                         //node to null
233             last = head;                //make last point to the node
234             current = current->link;    //make current point to
235                                         //the next node
236
237                 //copy the remaining elements of the stack
238             while (current != NULL)
239             {
240                 newNode = new nodeType<Type>;
241                 newNode->info = current->info;
242                 newNode->link = NULL;
243                 last->link = newNode;
244                 last = newNode;
245                 current = current->link;
246             }//end while
247         }//end else
248     }//end if
249
250     return *this;
251 }//end operator=
252 #endif
```

```h
#ifndef H_linkedQueue
#define H_linkedQueue

#include <iostream>

using namespace std;


template<class Type>
class linkedQueueType
{
public:
    const linkedQueueType<Type>& operator=
        (const linkedQueueType<Type>&);
    // overload the assignment operator
    bool isEmptyQueue();
    bool isFullQueue();
    void destroyQueue();
    void initializeQueue();
    void addQueue(const Type& newElement);
    void deQueue(Type& deqElement);
    linkedQueueType(); //default constructor
    linkedQueueType(const linkedQueueType<Type>& otherQueue);
        //copy constructor

    void printQueue();

    ~linkedQueueType(); //destructor

private:
    nodeType<Type>* front; //pointer to the front of the queue
    nodeType<Type>* rear;  //pointer to the rear of the queue
};


template<class Type>
linkedQueueType<Type>::linkedQueueType() //default constructor
{
    front = NULL; // set front to null
    rear = NULL;  // set rear to null
}


template<class Type>
bool linkedQueueType<Type>::isEmptyQueue()
{
    return(front == NULL);
}

template<class Type>
bool linkedQueueType<Type>::isFullQueue()
{
```

```
53      return false;
54  }
55
56  template<class Type>
57  void linkedQueueType<Type>::destroyQueue()
58  {
59      nodeType<Type>* temp;
60
61      while (front != NULL)  //while there are elements left in the queue
62      {
63          temp = front;         // set temp to point to the current node
64          front = front->link; // advance front to the next node
65          delete temp;          // deallocate memory occupied by temp
66      }
67
68      rear = NULL;  // set rear to null
69  }
70
71  template<class Type>
72  void linkedQueueType<Type>::initializeQueue()
73  {
74      destroyQueue();
75  }
76
77  template<class Type>
78  void linkedQueueType<Type>::addQueue(const Type& newElement)
79  {
80      nodeType<Type>* newNode;
81
82      newNode = new nodeType<Type>;   //create the node
83      newNode->info = newElement;     //store the info
84      newNode->link = NULL;           //initialize the link field to null
85
86      if (front == NULL)              //if initially queue is empty
87      {
88          front = newNode;
89          rear = newNode;
90      }
91      else                //add newNode at the end
92      {
93          rear->link = newNode;
94          rear = rear->link;
95      }
96  }//end addQueue
97
98  template<class Type>
99  void linkedQueueType<Type>::deQueue(Type& deqElement)
100 {
101     nodeType<Type>* temp;
102
103     deqElement = front->info;  //copy the info of the first element
104
```

```
105     cout << "deQueued item is " << deqElement << endl;
106
107     temp = front;              //make temp point to the first node
108     front = front->link;       //advance front to the next node
109     delete temp;               //delete the first node
110
111     if (front == NULL)          //if after deletion the queue is empty
112         rear = NULL;           //set rear to NULL
113 }//end deQueue
114
115
116
117 template<class Type>
118 linkedQueueType<Type>::~linkedQueueType() //destructor
119 {
120     nodeType<Type>* temp;
121
122     while (front != NULL)     //while there are elements left in the queue
123     {
124         temp = front;         //set temp to point to the current node
125         front = front->link; //advance first to the next node
126         delete temp;          //deallocate memory occupied by temp
127     }
128
129     rear = NULL;  // set rear to null
130 }
131
132 template<class Type>
133 const linkedQueueType<Type>& linkedQueueType<Type>::operator=
134 (const linkedQueueType<Type>& otherQueue)
135 {
136     //Write the definition of to overload the assignment operator
137
138 }
139
140 //copy constructor
141 template<class Type>
142 linkedQueueType<Type>::linkedQueueType(const linkedQueueType<Type>& otherQueue)
143 {
144     //Write the definition of the copy constructor
145 }//end copy constructor
146
147
148 template<class Type>
149 inline void linkedQueueType<Type>::printQueue()
150 {
151     cout << "Printing queue:" << endl;
152     nodeType<Type>* tempPtr = front;
153     for (nodeType<Type>* tempPtr = front; tempPtr != NULL; tempPtr = tempPtr-
        >link)
154     {
155         cout << tempPtr->info << endl;
```

```
156     }
157     cout << endl;
158 }
159
160 #endif
```

```
#ifndef H_dequeElem
#define H_dequeElem

#include <iostream>

using namespace std;
template <class Elem>
struct DNode
{
    Elem elem;
    DNode<Elem>* prev;
    DNode<Elem>* next;
};

template<class Elem>
class DLinkedList // doubly linked list
{
public:
    DLinkedList();              // constructor
    ~DLinkedList();             // destructor
    bool empty() const;             // is list empty?
    const Elem& front() const;          // get front element
    const Elem& back() const;           // get back element
    void addFront(const Elem& e);       // add to front of list
    void addBack(const Elem& e);        // add to back of list
    void removeFront();             // remove from front
    void removeBack();              // remove from back
    void printDeque();
private:                    // local type definitions
    DNode<Elem>* header;                // list sentinels
    DNode<Elem>* trailer;
protected:                  // local utilities
    void add(DNode<Elem>* v, const Elem& e);        // insert new node before v
    void remove(DNode<Elem>* v);            // remove node v
};

template<class Elem>
DLinkedList<Elem>::DLinkedList() {          // constructor
    header = new DNode<Elem>;               // create sentinels
    trailer = new DNode<Elem>;
    header->next = trailer;         // have them point to each other
    trailer->prev = header;
}

template<class Elem>
DLinkedList<Elem>::~DLinkedList() {         // destructor
    while (!empty()) removeFront();     // remove all but sentinels
    delete header;              // remove the sentinels
    delete trailer;
}

template<class Elem>
```

```
53  bool DLinkedList<Elem>::empty() const       // is list empty?
54  {
55      return (header->next == trailer);
56  }
57
58  template<class Elem>
59  const Elem& DLinkedList<Elem>::front() const    // get front element
60  {
61      return header->next->elem;
62  }
63
64  template<class Elem>
65  const Elem& DLinkedList<Elem>::back() const     // get back element
66  {
67      return trailer->prev->elem;
68  }
69
70  template<class Elem>
71  void DLinkedList<Elem>::add(DNode<Elem>* v, const Elem& e) // insert new node
      before v
72  {
73      DNode<Elem>* u = new DNode<Elem>;
74      u->elem = e;        // create a new node for e
75      u->next = v;                // link u in between v
76      u->prev = v->prev;              // ...and v->prev
77      u->prev->next = u;
78      v->prev = u;
79  }
80
81  template<class Elem>
82  void DLinkedList<Elem>::addFront(const Elem& e) // add to front of list
83  {
84      add(header->next, e);
85  }
86
87  template<class Elem>
88  void DLinkedList<Elem>::addBack(const Elem& e)  // add to back of list
89  {
90      add(trailer, e);
91  }
92
93  template<class Elem>
94  void DLinkedList<Elem>::remove(DNode<Elem>* v) {        // remove node v
95      DNode<Elem>* u = v->prev;               // predecessor
96      DNode<Elem>* w = v->next;               // successor
97      u->next = w;                // unlink v from list
98      w->prev = u;
99      delete v;
100 }
101
102 template<class Elem>
103 void DLinkedList<Elem>::removeFront()       // remove from font
```

```
104 {
105     cout << "removing from front: " << header->next->elem  << endl;
106     remove(header->next);
107 }
108 
109 template<class Elem>
110 void DLinkedList<Elem>::removeBack()        // remove from back
111 {
112     cout << "removing from back: " << trailer->prev->elem << endl;
113     remove(trailer->prev);
114 }
115 
116 template<class Elem>
117 void DLinkedList<Elem>::printDeque()        // remove from back
118 {
119     cout << "Printing deque front to back:" << endl;
120     for (DNode<Elem>* tempPtr = header->next; tempPtr != trailer; tempPtr =
        tempPtr->next)
121     {
122         cout << tempPtr->elem << endl;
123     }
124     cout << endl;
125 }
126 #endif
```