```
#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
```