```cpp
/*
Elliot Shaw
DoublyLinkedList_1
280
*/
#include <iostream>
#include <string>
using namespace std;

struct Node {
    int data;
    Node* next, * prev;
};

class DoublyLinkedList {
private:
    Node* top, * bottom;
    int size;
public:
    //constructors
    DoublyLinkedList();
    //setters
    void addNodeTop(int);
    void addNodeBottom(int);
    void deleteTop();
    void deleteBottom();
    void deleteAt(Node* p);
    void insertBefore(Node* p, int v);
    void insertAfter(Node* p, int v);
    void set(Node* p, int v);
    //getters
    int get(Node* p);
    int getSize();
    int getTop();
    Node* getTopP();
    int getBottom();
    int howMany(int v);
    Node* findFirst(int v);
    Node* findLast(int v);
    //utility
    void displayFromTop();
    void displayFromBottom();
};  //DoublyLinkedList

DoublyLinkedList::DoublyLinkedList() {
    top = bottom = nullptr;
    size = 0;
} //DoublyLinkedList

```

```
50  void DoublyLinkedList::addNodeTop(int v) {
51      Node* p = new Node;
52      p->data = v;
53      p->next = top;
54      p->prev = nullptr;
55      if (size == 0)
56          bottom = p;
57      else
58          top->prev = p;
59      top = p;
60      size++;
61  } //addNodeTop      O(1)
62
63  void DoublyLinkedList::addNodeBottom(int v) {
64      Node* p = new Node;
65      p->data = v;
66      p->prev = bottom;
67      p->next = nullptr;
68      if (size == 0)
69          top = p;
70      else
71          bottom->next = p;
72      bottom = p;
73      size++;
74  } //addNodeBottom       O(1)
75
76  void DoublyLinkedList::deleteTop() {
77      if (top != nullptr) {
78          Node* temp = top;
79          if (top == bottom)
80              top = bottom = nullptr;
81          else {
82              top = top->next;
83              top->prev = nullptr;
84          }
85          delete temp;
86          size--;
87      }
88  } //deleteTop       O(1)
89
90  void DoublyLinkedList::deleteBottom() {
91      if (bottom != nullptr) {
92          Node* temp = bottom;
93          if (bottom == top)
94              bottom = top = nullptr;
95          else {
96              bottom = bottom->prev;
97              bottom->next = nullptr;
98          }
```

```
 99          delete temp;
100          size--;
101      }
102  } //deleteBottom        O(1)
103
104  void DoublyLinkedList::deleteAt(Node* p) {
105      if (p == top) {
106          deleteTop();
107          size--;
108      }
109      else if (p == bottom) {
110          deleteBottom();
111          size--;
112      }
113      else {
114          p->next->prev = p->prev;
115          p->prev->next = p->next;
116          delete p;
117          size--;
118      }
119  } //deleteAt        O(1)
120
121  void DoublyLinkedList::insertBefore(Node* p, int v) {
122      Node* temp = new Node;
123      temp->data = v;
124      if (p == top) {
125          temp->prev = nullptr;
126          temp->next = p;
127          top->prev = temp;
128          top = temp;
129      }
130      else {
131          temp->prev = p->prev;
132          temp->next = p;
133          p->prev = temp;
134          temp->prev->next = temp;
135      }
136      size++;
137  } //insertBefore        O(1)
138
139  void DoublyLinkedList::insertAfter(Node* p, int v) {
140      if (p == bottom) {
141          Node* temp = new Node;
142          temp->data = v;
143
144          temp->next = nullptr;
145          temp->prev = p;
146          bottom->next = temp;
147          bottom = temp;
```

```
148         size++;
149     }
150     else {
151         insertBefore(p->next, v);
152     }
153
154 } //insertAfter     O(1)
155
156 Node* DoublyLinkedList::findFirst(int v) {
157     Node* runner = top;
158
159     while (runner != nullptr) {
160         if (runner->data == v)
161             return runner;
162         runner = runner->next;
163     }
164     return nullptr;
165 } //findFirst       O(n)
166
167 Node* DoublyLinkedList::findLast(int v) {
168     Node* runner = bottom;
169
170     while (runner != nullptr) {
171         if (runner->data == v)
172             return runner;
173         runner = runner->prev;
174     }
175     return nullptr;
176 } //findLast        O(n)
177
178 int DoublyLinkedList::howMany(int v) {
179     Node* runner = top;
180     int count = 0;
181     while (runner != nullptr) {
182         if (runner->data == v)
183             count++;
184         runner = runner->next;
185     }
186     return count;
187 } //howMany     O(n)
188
189 void DoublyLinkedList::set(Node* p, int v) {
190     p->data = v;
191 } //set     O(1)
192
193 int DoublyLinkedList::get(Node* p) {
194     return p->data;
195 } //get     O(1)
196
```

```
197 int DoublyLinkedList::getTop() {
198     return top->data;
199 } //getTop      O(1)
200
201 int DoublyLinkedList::getBottom() {
202     return bottom->data;
203 } //getBottom       O(1)
204
205 void DoublyLinkedList::displayFromTop() {
206     Node* runner = top;
207     while (runner != nullptr) {
208         cout << runner->data << endl;
209         runner = runner->next;
210     }
211 } //displayFromTop      O(n)
212
213 void DoublyLinkedList::displayFromBottom() {
214     Node* runner = bottom;
215     while (runner != nullptr) {
216         cout << runner->data << endl;
217         runner = runner->prev;
218     }
219 } //displayFromBottom       O(n)
220
221 int DoublyLinkedList::getSize() {
222     return size;
223 }; //getSize        O(1)
224
225 Node* DoublyLinkedList::getTopP() {
226     return top;
227 } //getTopP     O(1)
228
229
230 int main() {
231
232     DoublyLinkedList dl1 = DoublyLinkedList();
233
234
235     dl1.addNodeTop(12);
236
237     dl1.addNodeTop(50);
238
239     dl1.addNodeTop(-88);
240
241     dl1.addNodeBottom(-1);
242
243     dl1.addNodeBottom(-2);
244
245     dl1.addNodeBottom(-3);
```

```
246
247     cout << "Display from top 1..." << endl;
248
249     dl1.displayFromTop();
250
251     cout << "Display from bottom 1..." << endl;
252
253     dl1.displayFromBottom();
254
255     cout << endl;
256
257
258
259     dl1.deleteBottom();
260
261     cout << "Display from top 2..." << endl;
262
263     dl1.displayFromTop();
264
265     cout << "Display from bottom 2..." << endl;
266
267     dl1.displayFromBottom();
268
269     cout << "Size: " << dl1.getSize() << endl << endl;
270
271
272
273     Node* temp = dl1.getTopP(); //write this temporary function for testing,
          all it does is return the top pointer
274
275                                 //it's used so that we can have a pointer to
                  deleteAt
276
277     Node* r = temp->next; //change this line
          ------------------------------------------------
278
279     dl1.deleteAt(r);
280
281     cout << "Display from top 3..." << endl;
282
283     dl1.displayFromTop();
284
285     cout << "Display from bottom 3..." << endl;
286
287     dl1.displayFromBottom();
288
289     cout << "Size: " << dl1.getSize() << endl << endl;
290
291
```

```
292
293     r = dl1.findFirst(12);
294
295     cout << "Value found: " << r->data << endl << endl;
296
297
298
299     dl1.insertAfter(r, 999);
300
301     cout << "Display from top 4..." << endl;
302
303     dl1.displayFromTop();
304
305     cout << "Display from bottom 4..." << endl;
306
307     dl1.displayFromBottom();
308
309     cout << "Size: " << dl1.getSize() << endl << endl;
310
311 } //main
312
```