```
 1: #ifndef ALT_LINKED_DEQUE_H
 2: #define ALT_LINKED_DEQUE_H
 3:
 4: /**
 5:  * An implementation of a deque using a doubly-linked list WITHOUT sentinels.
 6:  */
 7: template <typename Object>
 8: class AltLinkedDeque {
 9:
10: public:
11:     AltLinkedDeque();                   // construct empty deque
12:     int size() const;                   // number of items in deque
13:     bool empty() const;                 // is the deque empty?
14:     const Object& front() const;        // access the front element
15:     const Object& back() const;         // access the back element
16:     void push_front(const Object& e);   // add element to the front of the deque
17:     void push_back(const Object& e);    // add element to the back of the deque
18:     void pop_front();                   // remove the front item from the deque
19:     void pop_back();                    // remove the back item from the deque
20:
21:     // housekeeping functions
22:     AltLinkedDeque(const AltLinkedDeque& other);
23:     ~AltLinkedDeque();
24:     AltLinkedDeque& operator=(const AltLinkedDeque& other);
25:
26: protected:
27:
28:     struct Node {
29:         Object elem;                    // element
30:         Node*  prev;                    // prev pointer
31:         Node*  next;                    // next pointer
32:         Node(const Object& e = Object(), Node* prv = NULL, Node* nxt = NULL)
33:             : elem(e), prev(prv), next(nxt) { }        // constructor
34:     };
35:
36: private:
37:     // instance variables
38:     Node* head;                                 // node with first deque element
39:     Node* tail;                                 // node with last deque element
40:     int n;                                      // number of items in deque
41:
42:     // utilities for housekeeping
43:     void removeAll();                           // utility for housekeeping
44:     void copyFrom(const AltLinkedDeque& other); // utility for housekeeping
45:
46: };  // end of AltLinkedDeque class
47:
48: #include "AltLinkedDeque.tcc"
49: #endif
```

```
 1: #include <stdexcept>               //   defines std::runtime_error
 2:
 3: /* Standard constructor creates an empty deque. */
 4: template <typename Object>
 5: AltLinkedDeque<Object>::AltLinkedDeque() : head(NULL), tail(NULL), n(0) { }
 6:
 7: /* Return the number of objects in the deque. */
 8: template <typename Object>
 9: int AltLinkedDeque<Object>::size() const {
10:     return n;
11: }
12:
13: /* Determine if the deque is currently empty. */
14: template <typename Object>
15: bool AltLinkedDeque<Object>::empty() const {
16:     return n == 0;
17: }
18:
19: /* Return a const reference to the front object in the deque. */
20: template <typename Object>
21: const Object& AltLinkedDeque<Object>::front() const {
22:     if (empty()) throw std::runtime_error("Access to empty deque");
23:     return head->elem;
24: }
25:
26: /* Return a const reference to the back object in the deque. */
27: template <typename Object>
28: const Object& AltLinkedDeque<Object>::back() const {
29:     if (empty()) throw std::runtime_error("Access to empty deque");
30:     return tail->elem;
31: }
32:
33: /* Insert an object at the front of the deque. */
34: template <typename Object>
35: void AltLinkedDeque<Object>::push_front(const Object& e) {
36:
37:     // YOUR CODE HERE
38:
39: }
40:
41: /* Remove the back object from the deque. */
42: template <typename Object>
43: void AltLinkedDeque<Object>::pop_back()  {
44:
45:     // YOUR CODE HERE
46:
47: }
48:
49: /* Insert an object at the back of the deque. */
50: template <typename Object>
51: void AltLinkedDeque<Object>::push_back(const Object& e) {
52:
53:     // would be symmetric to push_front
54: }
55:
56: /* Remove the front object from the deque. */
57: template <typename Object>
58: void AltLinkedDeque<Object>::pop_front()  {
59:
60:     // would be symmetric to pop_back
61:
62: }
```

```
 63:
 64: // housekeeping functions remain
 65:
 66: template <typename Object>
 67: void AltLinkedDeque<Object>::removeAll() {   // remove entire deque contents
 68:     while (!empty()) pop_front();         // for simplicity, reuse existing method
 69: }
 70:
 71: template <typename Object>
 72: void AltLinkedDeque<Object>::copyFrom(const AltLinkedDeque& other) {
 73:     Node* walk(other.head);                      // the front node
 74:     while (walk != NULL) {
 75:         push_back(walk->elem);
 76:         walk = walk->next;
 77:     }
 78: }
 79:
 80: /* Copy constructor */
 81: template <typename Object>
 82: AltLinkedDeque<Object>::AltLinkedDeque(const AltLinkedDeque& other)
 83:     : head(NULL), tail(NULL), n(0)
 84: {
 85:     copyFrom(other);
 86: }
 87:
 88: /* Destructor */
 89: template <typename Object>
 90: AltLinkedDeque<Object>::~AltLinkedDeque() {
 91:     removeAll();
 92: }
 93:
 94: /* Assignment operator */
 95: template <typename Object>
 96: AltLinkedDeque<Object>&
 97: AltLinkedDeque<Object>::operator=(const AltLinkedDeque& other) {
 98:     if (this != &other) {                            // avoid self copy (x = x)
 99:         removeAll();                                 // remove old contents
100:         copyFrom(other);                             // copy new contents
101:     }
102:     return *this;
103: }
```