```
#ifndef CSCI180_LINKED_QUEUE_H
#define CSCI180_LINKED_QUEUE_H

#include <stdexcept>                //   defines std::runtime_error

namespace csci180 {
  /** A queue implementation based upon use of a singly-linked list.
   *  Elements are inserted and removed according to the first-in
   *  first-out principle.
   *
   *  This implementation is based on the discussions from Chapter 4.4
   *  of our text, but it has been adjusted to suit my tastes.
   */
  template <typename Object>
  class LinkedQueue {

  protected:
    struct Node {                                  // a node in the queue
      Object element;                              // element
      Node*  next;                                 // next pointer
      Node(const Object& e = Object(), Node* n = NULL)
        : element(e), next(n) { }                  // constructor
    };

  private:
    Node* head;                                    // pointer to front of the queue
    Node* tail;                                    // pointer to back of the queue
    int sz;                                        // number of items in queue

  public:
    /** Standard constructor creates an empty queue. */
    LinkedQueue() : head(NULL), tail(NULL), sz(0) { }

    /** Returns the number of objects in the queue.
     *  @return number of elements
     */
    int size() const {
      return sz;
    }

    /** Determines if the queue is currently empty.
     *  @return true if empty, false otherwise.
     */
    bool empty() const {
      return sz == 0;
    }

    /** Returns a const reference to the front object in the queue.
     * @return reference to front element
     */
    const Object& front() const {
      if (empty())
        throw std::runtime_error("Access to empty queue");
      return head->element;
    }

    /** Returns a live reference to the front object in the queue.
     * @return reference to front element
     */
    Object& front() {
      if (empty())
        throw std::runtime_error("Access to empty queue");
      return head->element;
    }
```

```
65:     /** Inserts an object at the back of the queue.
66:      *  @param the new element
67:      */
68:     void push(const Object& elem) {
69:       Node* v = new Node(elem, NULL);
70:       if (sz == 0)                // if no other nodes,
71:         head = v;                 //   new node becomes the head (and tail)
72:       else                        // otherwise
73:         tail->next = v;           //   old tail must be informed about new node
74:       tail = v;                   // in either case, new node becomes the tail
75:       sz++;
76:     }
77:
78:     /** Removes the front object from the queue. */
79:     void pop()  {
80:       if (empty())
81:         throw std::runtime_error("Access to empty queue");
82:       Node* old = head;          // node to remove
83:       head = head->next;         // head of list will change (perhaps to NULL)
84:       if (--sz == 0)             // and if we've just removed the last item
85:         tail = NULL;             // the tail is also set to null (for clarity)
86:       delete old;
87:     }
88:
89:   protected:                                        // protected utilities
90:     void removeAll() {                              // remove entire queue contents
91:       while (!empty()) pop();
92:     }
93:
94:     void copyFrom(const LinkedQueue& other) {      // copy from other
95:       head = NULL;
96:       Node* model = other.head;                     // model is current node in other
97:       Node* prev   = NULL;
98:       while (model != NULL) {
99:         Node* v = new Node(model->element, NULL); // make copy of model
100:        if (head == NULL)
101:          head = v;                                // if first node
102:        else
103:          prev->next = v;                          // else link after prev
104:        prev  = v;
105:        model = model->next;
106:      }
107:      tail = prev;                                  // final node (or NULL) is the tail
108:      sz = other.sz;
109:    }
110:
111:  public:
112:    /** Copy constructor */
113:    LinkedQueue(const LinkedQueue& other) {  copyFrom(other); }
114:
115:    /** Destructor */
116:    ~LinkedQueue() {  removeAll(); }
117:
118:    /** Assignment operator */
119:    LinkedQueue& operator=(const LinkedQueue& other) {
120:      if (this != &other) {                         // avoid self copy (x = x)
121:        removeAll();                                // remove old contents
122:        copyFrom(other);                            // copy new contents
123:      }
124:      return *this;
125:    }
126:  };  // end of LinkedQueue class
127: } // end of csci180 namespace
128: #endif
```