

**CSC 402 - Data Structures II, Fall, 2010**  
**Priority Queue code and Assignment 3, Dr. Dale E. Parson**  
**Assignment 3 is due by 11:59 PM on November 4**  
**See comments in file schedulerTest.cxx below for assignment details.**

**/export/home/faculty/parson/AdvDataStructures/priorityq/PriorityQueue.h**

```
1  /*      PriorityQueue.h -- Concrete template container class definition
2          that implements a priority queue.
3
4          CSC402, Fall, 2009, Dr. Dale Parson.
5  */
6
7  #ifndef PRIORITY_QUEUE_H
8  #define PRIORITY_QUEUE_H
9  #include <iostream>
10 #include <map>
11 using namespace std;
12
13 /*      Class:          PriorityQueue
14
15          PriorityQueue is a template class that implements a
16          priority heap using an array of template-type values.
17          These values may be primitive types, pointers, or class
18          objects; class objects must supply the assignment operator,
19          0-parameter constructor and a copy constructor.
20          Insertion and removal are log(N) time complexity, except
21          when insertion entails growth of the underlying array,
22          which is then O(n). Construction of an array having the requisite
23          capacity circumvents the O(n) problem. Lookup via peek is O(1).
24  */
25 template <typename ElemType>
26 class PriorityQueue {
27 public:
28     /**
29     *   A comparator is a function for comparing two ElemType objects
30     *   for priority with respect to the priority heap. It returns
31     *   a negative value, 0, or a positive value when the left parameter
32     *   is <, ==, or > the right parameter in priority. The highest
33     *   priority element rises to the top of the heap. When there is
34     *   a tie for the highest, the relative order is arbitrary.
35     */
36     typedef int (*comparator)(const ElemType *const left,
37                               const ElemType *const right);
38     /**
```

```

39     * Construct a PriorityQueue with a comparison function and
40     * an initial non-0 capacity.
41     *
42     * Parameters:
43     *
44     *     comparefunction is a non-NULL comparator function for
45     *     measuring relative priority of two ElemType objects.
46     *     See typedef for comparator above.
47     *
48     *     capacity when > 0 is the initial size of the heap.
49     *     Making this big enough to hold all insertions guarantees
50     *     O(log(n)) mutation time. A default value of 0 or a value < 0
51     *     triggers allocation of some implementation-dependent
52     *     default size for the underlying array.
53     **/
54     PriorityQueue(comparator comparefunction, int capacity=0);
55     /**
56     *     Destructor clean up.
57     **/
58     ~PriorityQueue();
59     /**
60     *     A qhandle is an opaque type returned by enqueue to allow client
61     *     code to change and/or move an entry in the priority queue soem time
62     *     after it has been enqueued. See enqueue and move below.
63     **/
64     typedef unsigned long qhandle ;
65     /**
66     *     Insert value into the heap according to its priority.
67     *     Return an opaque handle to the object's location in the heap
68     *     that can be used later in a call to operation move().
69     *     Insertion is O(log(n)) on the size of the heap unless the underlying
70     *     array must grow, in which case it is O(n).
71     **/
72     qhandle enqueue(const ElemType & value);
73     /**
74     *     Adjust an element's position in the priority heap based on
75     *     a change in its priority. The element may move in the heap.
76     *
77     *     Parameter handle is the value returned by enqueue for this object.
78     *
79     *     Parameter newvalue is the value with a modified priority and
80     *     possibly other fields that takes the place of the previous value
81     *     in the queue associated with handle. If ElemType is a pointer that
82     *     has not changed, but the object referenced by the pointer has
83     *     changed with respect to priority, client code can supply the same
84     *     pointer as newvalue that was supplied as enqueue's value; in that case

```

```

85     * the call to move simply repositions that pointer in the priority queue.
86     *
87     * Return value is true for a valid handle, false if the handle is
88     * invalid or if the object associated with the handle has been removed.
89     **/
90     bool move(qhandle handle, const ElemType & newvalue);
91     /**
92     * Retrieve a copy of the element with highest priority.
93     *
94     * Reference parameter isValid returns true if the heap has contents,
95     * else false.
96     *
97     * Return value is a copy of the highest priority element if
98     * size() > 0 for the heap.
99     **/
100    ElemType peek(bool &isValid) const ;
101    /**
102    * Retrieve a copy of the element with highest priority, and remove that
103    * element from the top of the priority heap.
104    *
105    * Reference parameter isValid returns true if the heap has contents
106    * before removal, else false.
107    *
108    * Return value is a copy of the highest priority element if an only if
109    * size() > 0 for the heap before removal. Removal decrements size().
110    **/
111    ElemType dequeue(bool &isValid);
112    /**
113    * Return the number of elements in this priority queue.
114    **/
115    int size() const { return count ; }
116 private:
117     ElemType * heaparray ;           // array of sorted elements
118     int count ;                     // how many elements currently stored
119     int mycapacity ;                // total number of elements in heaparray
120     comparator comparisonFunction ;
121     // We must maintain two mappings in order to implement move().
122     // handle2index maps a handle returned by enqueue to an index in heaparray.
123     // index2handle maintains the inverse mapping.
124     // 1. Enqueue sets up these mappings. It uses and post-increments seqno
125     //     to assign a unique qhandle to each element in this PriorityQueue.
126     //     (Parson's implementation avoids reusing in-use handles after
127     //     wrapping seqno at 3^32; students need not bother about
128     //     seqno overflow.)
129     // 2. Whenever enqueue, dequeue, or move moves an element, it stores
130     //     the element's index in handle2index, and also sets index2handle

```

```

131     //      for that index to the element's handle.
132
133     map<qhandle, int> handle2index ;
134     map<int, qhandle> index2handle ;
135     PriorityQueue() ;      // This constructor is not implemented
136     qhandle seqno ;      // Sequence number is bumped on each insertion.
137     // moveup moves the element at heaparray[qindex] up in the heap
138     // towards the root as long as its priority is > than it parent's,
139     // adjusting handle2index and index2handle as needed.
140     void moveup(int qindex, qhandle hndl); // movement helper functions
141     // movedown moves the element at heaparray[qindex] down in the heap
142     // towards the leaves as long as its priority is < than the greater
143     // of its children, always bringing the greater child up at each step,
144     // adjusting handle2index and index2handle as needed.
145     void movedown(int qindex, qhandle hndl);
146     // higherPriorityChild takes the index of a left child in heaparray
147     // and returns:
148     // A. -1 if leftchild is > count. (Elements are stored at 1 .. count).
149     // B. leftchild if leftchild == count (there is not right child).
150     // C. Either leftchild or (leftchild+1), the index of the greater
151     //      priority child. This function only inspects one or
152     //      two children; it does not inspect their parent's priority.
153     int higherPriorityChild(int leftchild) ; // higher priority child's ix or -1
154 };
155
156 #endif

```

**/export/home/faculty/parson/AdvDataStructures/priorityq/PriorityQueue.cxx**

```

1  /*      PriorityQueue.cxx -- Method implementations for
2          PriorityQueue.h.
3
4          CSC237, Fall, 2008, Dr. Dale Parson.
5  */
6
7  #ifndef PRIORITY_QUEUE_CXX
8  #define PRIORITY_QUEUE_CXX
9  #include "PriorityQueue.h"
10
11  static const int MINCAPACITY = 1025 ;
12  static const int GROWSIZE = 1024 ;
13
14  template <typename ElemType>
15  PriorityQueue<ElemType>::PriorityQueue(comparator comparefunction, int capacity) {
16      if (capacity < MINCAPACITY) {
17          mycapacity = MINCAPACITY ;

```

```

18     } else {
19         mycapacity = capacity + 1 ;
20     }
21     heaparray = new ElemType [ mycapacity ];
22     count = 0 ;
23     comparisonFunction = comparefunction ;
24     seqno = 1 ;
25 }
26
27 template <typename ElemType>
28 PriorityQueue<ElemType>::~~PriorityQueue() {
29     delete [] heaparray ;
30     // Being paranoid about dangling references here:
31     heaparray = 0 ;
32     count = mycapacity = 0 ;
33 }
34
35 template <typename ElemType>
36 typename PriorityQueue<ElemType>::qhandle
37 PriorityQueue<ElemType>::enqueue(const ElemType & value) {
38     // Wrapping the sequence number won't be a problem until at least after
39     // 2^32 insertions, but let's be paranoid!
40     map<qhandle, int>::iterator hiter = handle2index.find(seqno);
41     while (hiter != handle2index.end()) {
42         seqno++ ;
43         hiter = handle2index.find(seqno);    // Search for the next free handle.
44     }
45     qhandle myhandle = seqno ;
46     seqno++ ;    // Advance for next time.
47     int myindex = count + 1 ;    // start heap at index 1.
48     if (myindex == mycapacity) {
49         // It's O(n) growth time on the old array.
50         int newcapacity = mycapacity + GROWSIZE ;
51         ElemType * newarray = new ElemType [ newcapacity ] ;
52         // element 0 is never used
53         for (int i = 1 ; i < myindex ; i++) {
54             newarray[i] = heaparray[i];
55         }
56         mycapacity = newcapacity ;
57         delete [] heaparray ;
58         heaparray = newarray ;
59     }
60     heaparray[myindex] = value ;
61     handle2index[myhandle] = myindex ;
62     index2handle[myindex] = myhandle ;
63     count++ ;

```

```

64     moveup(myindex, myhandle);
65     return myhandle ;
66 }
67
68 template <typename ElemType>
69 bool
70 PriorityQueue<ElemType>::move(qhandle handle, const ElemType & newvalue) {
71     map<qhandle, int>::iterator miter = handle2index.find(handle);
72     if (miter == handle2index.end()) {
73         return false ;
74     }
75     int index = handle2index[handle];
76     heaparray[index] = newvalue ;
77     int parent = index / 2 ;
78     if (parent > 0 && (*comparisonFunction)(&newvalue,
79         &(heaparray[parent])) > 0) { // my priority > parent
80         moveup(index, handle);
81         return true ;
82     }
83     int child = higherPriorityChild(index * 2) ;
84     if (child != -1 && (*comparisonFunction)(&newvalue,
85         &(heaparray[child])) < 0) { // my priority < child
86         movedown(index, handle);
87         return true ;
88     }
89     return true ;
90 }
91
92 template <typename ElemType>
93 void
94 PriorityQueue<ElemType>::moveup(int qindex, qhandle hndl) {
95     bool ismovement = false ;
96     int parent = qindex / 2 ;
97     ElemType myvalue = heaparray[qindex] ; // get a local copy
98     while (parent > 0 && (*comparisonFunction)(&myvalue,
99         &(heaparray[parent])) > 0) { // my priority > parent
100         heaparray[qindex] = heaparray[parent];
101         qhandle parenthndl = index2handle[parent]; // retrieve premove handle
102         index2handle[qindex] = parenthndl ;
103         handle2index[parenthndl] = qindex ;
104         qindex = parent ;
105         parent = qindex / 2 ;
106         ismovement = true ;
107     }
108     if (ismovement) { // Do not waste time otherwise.
109         // I go into the qindex after all movement is done.

```

```

110         heaparray[qindex] = myvalue ;
111         handle2index[hndl] = qindex ;
112         index2handle[qindex] = hndl ;
113     }
114 }
115
116 template <typename ElemType>
117 void
118 PriorityQueue<ElemType>::movedown(int qindex, qhandle hndl) {
119     bool ismovement = false ;
120     int child = higherPriorityChild(qindex * 2) ;
121     ElemType myvalue = heaparray[qindex] ; // get a local copy
122     while (child != -1 && (*comparisonFunction)(&myvalue,
123         &(heaparray[child])) < 0) { // my priority < child
124         heaparray[qindex] = heaparray[child];
125         qhandle childhndl = index2handle[child]; // retrieve premove handle
126         index2handle[qindex] = childhndl ;
127         handle2index[childhndl] = qindex ;
128         qindex = child ;
129         child = higherPriorityChild(qindex * 2) ;
130         ismovement = true ;
131     }
132     if (ismovement) { // Do not waste time otherwise.
133         // I go into the qindex after all movement is done.
134         heaparray[qindex] = myvalue ;
135         handle2index[hndl] = qindex ;
136         index2handle[qindex] = hndl ;
137     }
138 }
139
140 template <typename ElemType>
141 int
142 PriorityQueue<ElemType>::higherPriorityChild(int leftchild) {
143     if (leftchild > count) { // elements are at 1..count
144         return -1 ;
145     }
146     if (leftchild == count) {
147         return leftchild ; // there is no rightchild
148     }
149     if ((*comparisonFunction)(&(heaparray[leftchild]),
150         &(heaparray[leftchild+1])) >= 0) { // left is >= right
151         return leftchild ;
152     } else {
153         return(leftchild+1);
154     }
155 }

```

```

156
157 template <typename ElemType>
158 ElemType
159 PriorityQueue<ElemType>::peek(bool &isValid) const {
160     if (count > 0) {
161         isValid = true ;
162         return heaparray[1];      // stored in 1..count
163     } else {
164         ElemType garbage ;
165         isValid = false ;
166         return garbage ;
167     }
168 }
169
170 template <typename ElemType>
171 ElemType
172 PriorityQueue<ElemType>::dequeue(bool &isValid) {
173     ElemType result = peek(isValid);
174     if (isValid) {
175         qhandle deadhandle = index2handle[1];
176         handle2index.erase(deadhandle);
177         if (count > 1) {
178             qhandle lasthandle = index2handle[count];
179             index2handle.erase(count);
180             heaparray[1] = heaparray[count];
181             handle2index[lasthandle] = 1 ;
182             index2handle[1] = lasthandle ;
183             count-- ;
184             movedown(1, lasthandle);
185         } else {
186             index2handle.erase(1);
187             count-- ;
188         }
189     }
190     return result ;
191 }
192
193 #endif

```

**/export/home/faculty/parson/AdvDataStructures/priorityq/PriorityQueueTest.cxx**

```

1  /*      PriorityQueueTest.cxx -- test driver for priority queue algorithms.
2
3          CSC402 Fall 2009.
4          Updated Fall 2010 to include a non-primitive custom data type
5          in the priority queue. See extern function schedulerTest() in

```

```

6         file schedulerTest.cxx. STUDENT WORK GOES INTO schedulerTest.cxx.
7     */
8
9     // FOLLOWING EXAMPLE EXIT CODES DEMONSTRATE POSSIBLE RETURN
VALUES FROM MAIN.
10    static const int    SUCCESS    =    0 ; /* successful program execution */
11    static const int    USAGERR    =    1 ; /* invalid command line usage */
12    static const int    DATAERR   =    2 ; /* invalid input test data */
13
14    #include <iostream>
15    #include <fstream>
16    #include <string>
17    using namespace std;
18
19    #include "StringToInteger.h"
20    #include "PriorityQueue.cxx"
21    extern void schedulerTest(); // See schedulerTest.cxx.
22
23    /*
24        Function:          main
25
26        main is a test driver that test some graph classes.
27    */
28
29    // Here is an example comparator:
30    int compareints(const int *const left, const int *const right) {
31        if (*left < *right) {          // Lesser value gets higher priority.
32            return 1 ;                  // send the least value out first
33        } else if (*left > *right) {   // If priority of left is > right.
34            return -1 ;
35        } else {                       // Priorities are the same.
36            return 0 ;
37        }
38    }
39    int comparestrings(const string *const left, const string *const right) {
40        if (*left < *right) {          // Lesser value gets higher priority.
41            return 1 ;                  // send the least value out first
42        } else if (*left > *right) {   // If priority of left is > right.
43            return -1 ;
44        } else {                       // Priorities are the same.
45            return 0 ;
46        }
47    }
48    static const int QSIZE = 1024 ;    // Initial PriorityQueue capacity.
49
50    int

```

```

51 main(const int argc, char *argv[]) {
52     PriorityQueue<int> intqueue(compareints, QSIZE);
53     PriorityQueue<string> stringqueue(comparestrings, QSIZE);
54     // Test all of the PriorityQueue operations.
55     bool sawstring = false, sawint = false ;
56     string sbuf, minstr, maxstr ;
57     int ibuf, minint, maxint ;
58     PriorityQueue<int>::qhandle mininthndl, maxinthndl ; ;
59     PriorityQueue<string>::qhandle minstrhndl, maxstrhndl ;
60     bool isvalid ;
61     cin >> sbuf ; // istream requires a read before setting eof flag
62     while (! cin.eof()) {
63         bool isminstr = false, ismaxstr = false, isminint = false,
64             ismaxint = false ;
65         PriorityQueue<string>::qhandle shndl = stringqueue.enqueue(sbuf);
66         cout << "MAPPED STRING " << sbuf << " TO HANDLE " << shndl << endl ;
67         if ((! sawstring) || sbuf < minstr) {
68             minstr = sbuf ; // save for a later move() call
69             minstrhndl = shndl ;
70         }
71         if ((! sawstring) || sbuf > maxstr) {
72             maxstr = sbuf ; // save for a later move() call
73             maxstrhndl = shndl ;
74         }
75         sawstring = true ;
76         ibuf = StringToInteger(sbuf.c_str(), isvalid);
77         if (isvalid) {
78             PriorityQueue<int>::qhandle ihndl = intqueue.enqueue(ibuf);
79             cout << "MAPPED INT " << ibuf << " TO HANDLE " << ihndl << endl ;
80             if ((! sawint) || ibuf < minint) {
81                 minint = ibuf ; // save for a later move() call
82                 mininthndl = shndl ;
83             }
84             if ((! sawint) || ibuf > maxint) {
85                 maxint = ibuf ; // save for a later move() call
86                 maxinthndl = shndl ;
87             }
88             sawint = true ;
89         }
90         cin >> sbuf ; // next read
91     }
92     // Replace min values with big max values and vice versa.
93     cout << endl ;
94     if (sawstring) {
95         isvalid = stringqueue.move(minstrhndl, string("~") + minstr);
96         if (! isvalid) {

```

```

97         cout << "INVALID false returned from move minstrhdl" << endl ;
98     } else {
99         cout << "MOVED HANDLE " << minstrhdl << " TO STRING "
100         << (string("~") + minstr) << endl ;
101     }
102     isvalid = stringqueue.move(maxstrhdl, string("+") + maxstr);
103     if (! isvalid) {
104         cout << "INVALID false returned from move maxstrhdl" << endl ;
105     } else {
106         cout << "MOVED HANDLE " << maxstrhdl << " TO STRING "
107         << (string("+") + maxstr) << endl ;
108     }
109     if (sawint) {
110         isvalid = intqueue.move(mininthhdl, 0x3fffffff);
111         if (! isvalid) {
112             cout << "INVALID false returned from move mininthhdl" << endl ;
113         } else {
114             cout << "MOVED HANDLE " << mininthhdl << " TO int "
115             << 0x3fffffff << endl ;
116         }
117         isvalid = intqueue.move(maxinthhdl, (-1 * 0x3fffffff));
118         if (! isvalid) {
119             cout << "INVALID false returned from move maxinthhdl" << endl ;
120         } else {
121             cout << "MOVED HANDLE " << maxinthhdl << " TO int "
122             << (-1 * 0x3fffffff) << endl ;
123         }
124     }
125 }
126 cout << "\nINTS" << endl << endl ;
127 while (intqueue.size() > 0) {
128     int oldsize = intqueue.size() ;
129     ibuf = intqueue.peek(isvalid);
130     if (! isvalid) {
131         cout << "INVALID FALSE isvalid flag on integer peek" << endl ;
132     } else {
133         cout << "I PEEK\t" << ibuf << endl ;
134     }
135     ibuf = intqueue.dequeue(isvalid);
136     if (! isvalid) {
137         cout << "INVALID FALSE isvalid flag on integer dequeue" << endl ;
138     } else {
139         cout << "I DEQUEUE\t" << ibuf << endl ;
140     }
141     if (intqueue.size() != (oldsize-1)) {
142         cout << "INVALID integer dequeue failed to drop size!" << endl ;

```

```

143         return(DATAERR);           // It could loop forever.
144     }
145 }
146 ibuf = intqueue.peek(isvalid);
147 if (isvalid) {
148     cout << "INVALID TRUE isvalid flag on integer peek" << endl ;
149 }
150 ibuf = intqueue.dequeue(isvalid);
151 if (isvalid) {
152     cout << "INVALID TRUE isvalid flag on integer dequeue" << endl ;
153 }
154 // Now do the same with stringqueue.
155 cout << "\nSTRS" << endl << endl ;
156 while (stringqueue.size() > 0) {
157     int oldsize = stringqueue.size() ;
158     sbuf = stringqueue.peek(isvalid);
159     if (! isvalid) {
160         cout << "INVALID FALSE isvalid flag on string peek" << endl ;
161     } else {
162         cout << "I PEEK\t" << sbuf << endl ;
163     }
164     sbuf = stringqueue.dequeue(isvalid);
165     if (! isvalid) {
166         cout << "INVALID FALSE isvalid flag on string dequeue" << endl ;
167     } else {
168         cout << "I DEQUEUE\t" << sbuf << endl ;
169     }
170     if (stringqueue.size() != (oldsize-1)) {
171         cout << "INVALID string dequeue failed to drop size!" << endl ;
172         return(DATAERR);           // It could loop forever.
173     }
174 }
175 sbuf = stringqueue.peek(isvalid);
176 if (isvalid) {
177     cout << "INVALID TRUE isvalid flag on integer peek" << endl ;
178 }
179 sbuf = stringqueue.dequeue(isvalid);
180 if (isvalid) {
181     cout << "INVALID TRUE isvalid flag on integer dequeue" << endl ;
182 }
183 schedulerTest();
184 return(SUCCESS);
185 }

```

**/export/home/faculty/parson/AdvDataStructures/priorityq/schedulerTest.cxx**

```
1  /*      schedulerTest.cxx -- test driver for priority queue.
2
3      CSC402 Fall 2010 tests to use a non-primitive custom data type
4      in the priority queue. See function schedulerTest() below.
5      project is due on November 4.
6      STUDENT NAME
7
8      Write and document the test driver for a priority queue
9      Return to the caller upon completion.
10 */
11
12
```

/\*\*\*\*\*\*

**13 STUDENT PROJECT REQUIREMENT:**

```
14
15      cp ~parson/AdvDataStructures/priorityq.assn3.zip ~/AdvDataStructures
16      cd ~/AdvDataStructures
17      unzip priorityq.assn3.zip
18      cd ./priorityq
19
```

20 Your test driver must manipulate two pairs of files,  
21 schedulerTest1.txt / schedulerTest1.out (I am supplying schedulerTest1.txt),  
22 and schedulerTest2.txt / schedulerTest2.out (you will supply both).  
23 Therefore, write function schedulerTest() below as a two-pass  
24 loop that manipulates file names starting with “schedulerTest1” on the  
25 first pass and “schedulerTest2” on the second pass. Make sure to close()  
26 any files that you open.

27  
28 1. Attempt to open text file “schedulerTest1.txt” as an ifstream  
29 (input file stream) for reading test data. See documentation  
30 in <http://cplusplus.com/reference/iostream/> and also in  
31 ~parson/fall09AdvDataStructures/externalsort, file\_queueOfStrings.cxx  
32 and mergesort.cxx for opening an ifstream.

33  
34 NOTE: All stream class types in iostream have three boolean predicate  
35 functions for testing completion of the most recent use of that stream:  
36 eof() (end-of-file), fail() and bad(). These status bits  
37 are set only \*AFTER\* attempting an operation such as an open or  
38 a read using operations such as “>>” or getline(). Therefore, you can  
39 check for fail() after attempting an open(), or check for eof() after  
40 attempting to read from the file; you may wish to break out of a  
41 loop after attempting a read and then finding eof() to be true.

42  
43 2. Open text file “schedulerTest1.out” as an output stream for writing,

44 truncating any previous contents *\*if\** the file already exists.

45

- 46 3. If the file in step 1 opens successfully, it will consist of a  
47 series of lines like this (with possible bad lines):

48

49 noteon pitch time

50 noteoff pitch time

51

52 Each line has three fields separated by white space:

53

54 The word “noteon” or “noteoff” as a literal.

55 Pitch is an integer  $\geq 0$  &&  $\leq 127$ .

56 Time is an integer  $\geq 0$ .

57

58 You must read each line, store it in a structure with this definition:

59

```
60 struct noteinfo {
```

```
61     boolean isnoteon ; // true for “noteon”, false for “noteoff”
```

```
62     unsigned int pitch ;
```

```
63     unsigned int time ;
```

```
64 } ;
```

65

66 Use a local variable for storing the information from one line

67 of the file into a noteinfo struct. You can use function

68 StringToInteger() similar to my code examples to convert a

69 numeric string into an integer and check for conversion errors.

70

71 If your input loop reads any line that does not conform to the  
72 above format (string other than “noteon” or “noteoff”, invalid int  
73 or int outside valid range, or an improperly formatted line),  
74 print a warning *\*including the line number\** to cerr and go on to  
75 process the next input line. The line number must be correct in  
76 your error message to cerr.

77

- 78 4. For each valid line stored into a noteinfo object, insert the  
79 noteinfo object into a PriorityQueue object (see PriorityQueue.h)  
80 that you have constructed after opening the input file and before  
81 processing its lines. ElemType of this PriorityQueue is the  
82 noteinfo class. In order to construct the PriorityQueue object  
83 you must supply it with the constructor’s comparator comparefunction  
84 parameter. See PriorityQueueTest.cxx for some example comparators.  
85 Your comparator function must give the highest priority to the  
86 noteinfo object with the lowest time value (i.e., the next upcoming  
87 time). For noteinfo objects with the same value for time, the  
88 comparator function must give higher priority to noteinfo objects  
89 with the isnoteon field to set to false. In other words, if there

```

90         are multiple noteinfo objects with the same lowest time (making this
91         a minqueue), the noteoff objects should dequeue ahead of the noteon
92         objects at that time.
93
94     5. After reaching end-of-file on the input stream, close() it, and then
95        iterate through the priority queue. In each pass of the loop dequeue
96        the highest priority noteinfo object at the front of the queue,
97        and print out a line to the file opened in step 2, using the same
98        format as the input file, with one noteinfo entry per line.
99
100    6. After emptying the priority queue, close the output file.
101
102    7. After processing files “schedulerTest1.txt” and “schedulerTest1.out”,
103        loop back and repeat the above steps for “schedulerTest2.txt” and
104        “schedulerTest2.out”. After this second iteration, return to the
105        calling function. All previously opened files are closed at this
106        point, and if any objects were constructed using “new” they have been
107        deleted using “delete” before returning.
108
109    8. Add tests into the makefile to
110        diff schedulerTest1.out schedulerTest1.ref > schedulerTest1.dif
111        diff schedulerTest2.out schedulerTest2.ref > schedulerTest2.dif
112        at the appropriate places. Make sure that these two .ref files
113        are correct.
114
115    9. Invoke “gmake turnitin” before the end of November 4 with your updated
116        schedulerTest.cxx, makefile, schedulerTest1.ref, schedulerTest2.txt
117        and schedulerTest2.re files.
118    **/
119
120    // STUDENT CODE GOES BELOW
121
122    void schedulerTest() {
123    }

```