```
1 #include "QueueLi.h"
3 #ifdef USE_DOT_H
      #include <iostream.h>
5 #else
6
      #include <iostream>
7
     using namespace std;
8 #endif
9
10 int main( )
11 {
12
       Queue<int> q;
13
14
       for( int j = 0; j < 8; j++ )</pre>
15
            for( int i = 0; i < 8; i++ )</pre>
16
17
                q.enqueue( i );
18
19
           while( !q.isEmpty( ) )
                cout << q.dequeue( ) << endl;</pre>
20
21
       }
22
23
       return 0;
24 }
25
```

```
1 #ifndef QUEUELI_H_
2 #define QUEUELI_H_
3
4 #include <stdlib.h>
5 #include "Except.h"
7 // Queue class -- linked list implementation.
8 //
9 // CONSTRUCTION: with no parameters.
10 //
12 // void enqueue( x ) --> Insert x
                    --> Return and remove least recent item
13 // void dequeue( )
14 // Object getFront( ) --> Return least recently inserted item
                      --> Return true if empty; else false
15 // bool isEmpty()
16 // void makeEmpty( ) --> Remove all items
18 // UnderflowException thrown as needed.
20 template <class Object>
21 class Queue
22 {
23
    public:
24
      Queue();
25
      Queue( const Queue & rhs );
26
      ~Queue( );
27
      const Queue & operator= ( const Queue & rhs );
28
      bool isEmpty( ) const;
29
      const Object & getFront( ) const;
30
31
32
      void makeEmpty( );
33
      Object dequeue();
34
      void enqueue( const Object & x );
35
36
    private:
37
      struct ListNode
      {
38
          Object element;
39
40
          ListNode *next;
41
42
          ListNode( const Object & theElement, ListNode * n = NULL )
43
            : element( theElement ), next( n ) { }
44
      };
45
      ListNode *front;
46
47
      ListNode *back;
48 };
49
50 #include "QueueLi.cpp"
51 #endif
52
53
```

```
1 #include "QueueLi.h"
 2
 3
 4 // Construct the queue.
 5 template <class Object>
 6 Queue<0bject>::Queue( )
 7 {
 8
       front = back = NULL;
 9 }
10
11 // Copy constructor.
12 template <class Object>
13 Queue<Object>::Queue( const Queue<Object> & rhs )
14 {
       front = back = NULL;
15
       *this = rhs;
16
17 }
18
19 // Destructor.
20 template <class Object>
21 Queue<Object>::~Queue( )
22 {
23
       makeEmpty( );
24 }
25
26 // Test if the queue is logically empty.
27 // Return true if empty, false, otherwise.
28 template <class Object>
29 bool Queue<Object>::isEmpty( ) const
30 {
31
       return front == NULL;
32 }
33
34 // Make the queue logically empty.
35 template <class Object>
36 void Queue<0bject>::makeEmpty( )
37 {
38
       while( !isEmpty( ) )
           dequeue( );
39
40 }
41
42 // Return the least recently inserted item in the queue
43 // or throw UnderflowException if empty.
44 template <class Object>
45 const Object & Queue<Object>::getFront( ) const
46 {
47
       if( isEmpty( ) )
48
           throw UnderflowException( );
       return front->element;
49
50 }
51
52 // Return and remove the least recently inserted item from
53 // the queue. Throw UnderflowException if empty.
```

```
54 template <class Object>
55 Object Queue<Object>::dequeue( )
56 {
57
       Object frontItem = getFront( );
58
59
       ListNode *old = front;
60
       front = front->next;
       delete old;
61
62
63
       return frontItem;
64 }
65
66 // Insert x into the queue.
67 template <class Object>
68 void Queue<Object>::enqueue( const Object & x )
69 {
70
       if( isEmpty( ) )
71
           back = front = new ListNode( x );
72
       else
73
           back = back->next = new ListNode( x );
74 }
75
76 // Deep copy.
77 template <class Object>
78 const Queue<Object> & Queue<Object>::operator=( const Queue<Object> &
     rhs )
79 {
80
       if( this != &rhs )
81
       {
82
           makeEmpty( );
83
           ListNode *rptr;
           for( rptr = rhs.front; rptr != NULL; rptr = rptr->next )
84
85
               enqueue( rptr->element );
86
       }
87
       return *this;
88 }
89
```

```
1 #include "StackLi.h"
 3 #ifdef USE_DOT_H
      #include <iostream.h>
 5 #else
 6
      #include <iostream>
 7
     using namespace std;
 8 #endif
 9
10 int main( )
11 {
12
       Stack<int> s, s1;
13
14
       for( int i = 0; i < 10; i++ )</pre>
15
           s.push( i );
16
17
       s1 = s;
18
19
       cout << "s" << endl;</pre>
       while( !s.isEmpty( ) )
20
21
            cout << s.topAndPop( ) << endl;</pre>
22
      cout << endl << "s1" << endl;</pre>
23
24
       while( !s1.isEmpty( ) )
25
            cout << s1.topAndPop( ) << endl;</pre>
26
27
       return 0;
28 }
```

```
1 #ifndef STACKLI_H_
2 #define STACKLI_H_
3
4 #include <stdlib.h>
5 #include "Except.h"
6
7
8 // Stack class -- linked list implementation.
10 // CONSTRUCTION: with no parameters.
13 // void push( x )
                     --> Insert x
14 // void pop( )
                         --> Remove most recently inserted item
15 // Object top( )
                         --> Return most recently inserted item
16 // Object topAndPop( ) --> Return and remove most recently inserted
    item
17 // bool isEmpty( )
                          --> Return true if empty; else false
                         --> Remove all items
18 // void makeEmpty( )
19 // ***********ERRORS******************
20 // UnderflowException thrown as needed.
21
22 template <class Object>
23 class Stack
24 {
   public:
25
      Stack();
26
27
      Stack( const Stack & rhs );
28
      ~Stack( );
29
30
      bool isEmpty( ) const;
31
      const Object & top( ) const;
32
33
      void makeEmpty( );
      void pop( );
34
35
      void push( const Object & x );
      Object topAndPop( );
37
38
      const Stack & operator=( const Stack & rhs );
39
40
    private:
41
      struct ListNode
42
43
          Object element;
          ListNode *next;
44
45
          ListNode( const Object & theElement, ListNode * n = NULL )
46
47
            : element( theElement ), next( n ) { }
48
       };
49
50
      ListNode *topOfStack;
51 };
52
```

54 #endif

55

56

```
1 #include "StackLi.h"
 3
 4 // Construct the stack.
 5 template <class Object>
 6 Stack<0bject>::Stack( )
7 {
 8
       topOfStack = NULL;
9 }
10
11 // Copy constructor.
12 template <class Object>
13 Stack<Object>::Stack( const Stack<Object> & rhs )
14 {
       topOfStack = NULL;
15
16
       *this = rhs;
17 }
18
19 // Destructor.
20 template <class Object>
21 Stack<Object>::~Stack( )
22 {
23
       makeEmpty( );
24 }
25
26 // Test if the stack is logically empty.
27 // Return true if empty, false, otherwise.
28 template <class Object>
29 bool Stack<Object>::isEmpty( ) const
30 {
       return topOfStack == NULL;
31
32 }
33
34 // Make the stack logically empty.
35 template <class Object>
36 void Stack<Object>::makeEmpty( )
37 {
       while( !isEmpty( ) )
38
39
           pop( );
40 }
41
42 // Return the most recently inserted item in the stack.
43 // or throw an UnderflowException if empty.
44 template <class Object>
45 const Object & Stack<Object>::top( ) const
46 <mark>{</mark>
47
       if( isEmpty( ) )
48
           throw UnderflowException( );
       return topOfStack->element;
49
50 }
51
52 // Remove the most recently inserted item from the stack.
53 // Throw Underflow if the stack is empty.
```

```
template <class Object>
55 void Stack<Object>::pop( )
56 {
57
        if( isEmpty( ) )
            throw UnderflowException( );
58
59
        ListNode *oldTop = topOfStack;
60
        topOfStack = topOfStack->next;
61
62
        delete oldTop;
63 }
64
65 // Return and remove the most recently inserted item
66 // from the stack.
67 template <class Object>
68 Object Stack<Object>::topAndPop( )
69 {
70
        Object topItem = top( );
71
        pop();
        return topItem;
72
73 }
74
75 // Insert x into the stack.
76 template <class Object>
77 void Stack<Object>::push( const Object & x )
78 {
79
        topOfStack = new ListNode( x, topOfStack );
80 }
81
82 // Deep copy.
83 template <class Object>
84 const Stack<Object> & Stack<Object>::operator=( const Stack<Object> &
      rhs )
85 {
86
        if( this != &rhs )
87
88
            makeEmpty( );
89
            if( rhs.isEmpty( ) )
90
                return *this;
91
92
            ListNode *rptr = rhs.topOfStack;
            ListNode *ptr = new ListNode( rptr->element );
93
94
            topOfStack = ptr;
95
96
            for( rptr = rptr->next; rptr != NULL; rptr = rptr->next )
97
                ptr = ptr->next = new ListNode( rptr->element );
98
99
        return *this;
100 }
101
```

```
2 #include "Except.h"
 3 #include "BinaryHeap.h"
 5 #ifdef USE_DOT_H
 6
       #include <iostream.h>
 7 #else
 8
       #include <iostream>
 9
       using namespace std;
10 #endif
11
12 // Test program
13 int main( )
14 {
15
       int numItems = 10000;
       BinaryHeap<int> h;
16
17
       int i = 37;
18
       int x;
19
20
       try
21
       {
22
            for( i = 37; i != 0; i = ( i + 37 ) % numItems )
                h.insert( i );
23
            for( i = 1; i < numItems; i++ )</pre>
24
25
            {
26
                h.deleteMin( x );
                if( x != i )
27
28
                    cout << "Oops! " << i << endl;</pre>
29
30
            for( i = 37; i != 0; i = ( i + 37 ) % numItems )
31
                h.insert( i );
32
            h.insert( 0 );
33
       }
       catch( const DSException & e )
34
35
          { cout << e.toString( ) << endl; }
36
37
       return 0;
38 }
39
```

```
1 #ifndef BINARY_HEAP_H_
2 #define BINARY_HEAP_H_
3
4 #include "Except.h"
5 #include "vector.h"
7 #include "StartConv.h"
8
9 // BinaryHeap class.
10 //
11 // CONSTRUCTION: with no parameters or vector containing items.
12 //
14 // void insert( x )
                       --> Insert x
15 // void deleteMin( )
                          --> Remove smallest item
16 // void deleteMin( min ) --> Remove and send back smallest item
17 // Comparable findMin( ) --> Return smallest item
18 // bool isEmpty( )
                          --> Return true if empty; else false
19 // void makeEmpty( )
                          --> Remove all items
20 // ***********ERRORS******************
21 // Throws UnderflowException as warranted.
22
23 template <class Comparable>
24 class BinaryHeap
25 {
    public:
26
27
      BinaryHeap( );
28
      BinaryHeap( const vector<Comparable> & v );
29
30
      bool isEmpty( ) const;
31
      const Comparable & findMin() const;
32
      void insert( const Comparable & x );
33
34
      void deleteMin( );
      void deleteMin( Comparable & minItem );
35
36
      void makeEmpty( );
37
    private:
38
                        theSize; // Number of elements in heap
39
40
       vector<Comparable> array;  // The heap array
41
42
      void buildHeap( );
      void percolateDown( int hole );
43
44 };
45
46 #include "EndConv.h"
47 #include "BinaryHeap.cpp"
48 #endif
49
```

```
1 #include "BinaryHeap.h"
 2 #include "StartConv.h"
 4 // Construct the binary heap.
 5 template <class Comparable>
 6 BinaryHeap<Comparable>::BinaryHeap( )
 7
     : array( 11 ), theSize( 0 )
 8 {
9 }
10
11 // Construct the binary heap.
12 // v is a vector containing the initial items.
13 template <class Comparable>
14 BinaryHeap<Comparable>::BinaryHeap( const vector<Comparable> & v )
     : array( v.size( ) + 1 ), theSize( v.size( ) )
15
16 {
       for( int i = 0; i < v.size( ); i++ )</pre>
17
18
           array[i+1] = v[i];
19
       buildHeap( );
20 }
21
22 // Insert item x into the priority queue, maintaining heap order.
23 // Duplicates are allowed.
24 template <class Comparable>
25 void BinaryHeap<Comparable>::insert( const Comparable & x )
26 {
       array[ 0 ] = x; // initialize sentinel
27
28
       if( theSize + 1 == array.size( ) )
           array.resize( array.size( ) * 2 + 1 );
29
30
         // Percolate up
31
32
       int hole = ++theSize;
33
       for(; x < array[ hole / 2]; hole /= 2)
34
           array[ hole ] = array[ hole / 2 ];
35
       array[hole] = x;
36 }
37
38 // Find the smallest item in the priority queue.
39 // Return the smallest item, or throw UnderflowException if empty.
40 template <class Comparable>
41 const Comparable & BinaryHeap<Comparable>::findMin() const
42 {
43
       if( isEmpty( ) )
           throw UnderflowException( );
44
45
       return array[ 1 ];
46 }
47
48 // Remove the smallest item from the priority queue.
49 // Throw UnderflowException if empty.
50 template <class Comparable>
51 void BinaryHeap<Comparable>::deleteMin( )
52 {
53
       if( isEmpty( ) )
```

```
54
            throw UnderflowException( );
 55
 56
        array[ 1 ] = array[ theSize-- ];
 57
        percolateDown( 1 );
 58 }
 59
 60 // Remove the smallest item from the priority queue
 61 // and place it in minItem. Throw UnderflowException if empty.
    template <class Comparable>
 63 void BinaryHeap<Comparable>::deleteMin( Comparable & minItem )
 64 {
 65
        minItem = findMin( );
 66
        array[ 1 ] = array[ theSize-- ];
 67
        percolateDown( 1 );
 68 }
 69
 70 // Establish heap-order property from an arbitrary
 71 // arrangement of items. Runs in linear time.
 72 template <class Comparable>
 73 void BinaryHeap<Comparable>::buildHeap( )
 74 {
        for( int i = theSize / 2; i > 0; i-- )
 75
 76
            percolateDown( i );
 77 }
 78
 79 // Test if the priority queue is logically empty.
 80 // Return true if empty, false otherwise.
 81 template <class Comparable>
 82 bool BinaryHeap<Comparable>::isEmpty( ) const
 83 {
 84
        return theSize == 0;
 85 }
 86
 87 // Make the priority queue logically empty.
 88 template <class Comparable>
 89 void BinaryHeap<Comparable>::makeEmpty( )
 90 {
 91
        theSize = 0;
 92 }
 93
 94 // Internal method to percolate down in the heap.
 95 // hole is the index at which the percolate begins.
 96 template <class Comparable>
 97 void BinaryHeap<Comparable>::percolateDown( int hole )
 98 {
        int child;
 99
        Comparable tmp = array[ hole ];
100
101
        for( ; hole * 2 <= theSize; hole = child )</pre>
102
103
104
            child = hole * 2;
            if( child != theSize && array[ child + 1 ] < array[ child ] )</pre>
105
106
                 child++;
```

```
C:\temp\ejercicios\ejemplo\ejemplo\BinaryHeap.cpp
```

3

```
if( array[ child ] < tmp )</pre>
107
                 array[ hole ] = array[ child ];
108
            else
109
110
                 break;
111
        array[ hole ] = tmp;
112
113 }
114
115 #include "EndConv.h"
116
117
```