```cpp
#include "QueueLi.h"

#ifdef USE_DOT_H
    #include <iostream.h>
#else
    #include <iostream>
    using namespace std;
#endif

int main( )
{
    Queue<int> q;

    for( int j = 0; j < 8; j++ )
    {
        for( int i = 0; i < 8; i++ )
            q.enqueue( i );

        while( !q.isEmpty( ) )
            cout << q.dequeue( ) << endl;
    }

    return 0;
}

```

```
#ifndef QUEUELI_H_
#define QUEUELI_H_

#include <stdlib.h>
#include "Except.h"

// Queue class -- linked list implementation.
//
// CONSTRUCTION: with no parameters.
//
// ******************PUBLIC OPERATIONS*********************
// void enqueue( x )      --> Insert x
// void dequeue( )        --> Return and remove least recent item
// Object getFront( )     --> Return least recently inserted item
// bool isEmpty( )        --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// UnderflowException thrown as needed.

template <class Object>
class Queue
{
  public:
    Queue( );
    Queue( const Queue & rhs );
    ~Queue( );
    const Queue & operator= ( const Queue & rhs );

    bool isEmpty( ) const;
    const Object & getFront( ) const;

    void makeEmpty( );
    Object dequeue( );
    void enqueue( const Object & x );

  private:
    struct ListNode
    {
        Object    element;
        ListNode *next;

        ListNode( const Object & theElement, ListNode * n = NULL )
          : element( theElement ), next( n ) { }
    };

    ListNode *front;
    ListNode *back;
};

#include "QueueLi.cpp"
#endif
```

```cpp
#include "QueueLi.h"


// Construct the queue.
template <class Object>
Queue<Object>::Queue( )
{
    front = back = NULL;
}

// Copy constructor.
template <class Object>
Queue<Object>::Queue( const Queue<Object> & rhs )
{
    front = back = NULL;
    *this = rhs;
}

// Destructor.
template <class Object>
Queue<Object>::~Queue( )
{
    makeEmpty( );
}

// Test if the queue is logically empty.
// Return true if empty, false, otherwise.
template <class Object>
bool Queue<Object>::isEmpty( ) const
{
    return front == NULL;
}

// Make the queue logically empty.
template <class Object>
void Queue<Object>::makeEmpty( )
{
    while( !isEmpty( ) )
        dequeue( );
}

// Return the least recently inserted item in the queue
// or throw UnderflowException if empty.
template <class Object>
const Object & Queue<Object>::getFront( ) const
{
    if( isEmpty( ) )
        throw UnderflowException( );
    return front->element;
}

// Return and remove the least recently inserted item from
// the queue. Throw UnderflowException if empty.
```

```cpp
template <class Object>
Object Queue<Object>::dequeue( )
{
    Object frontItem = getFront( );

    ListNode *old = front;
    front = front->next;
    delete old;

    return frontItem;
}

// Insert x into the queue.
template <class Object>
void Queue<Object>::enqueue( const Object & x )
{
    if( isEmpty( ) )
        back = front = new ListNode( x );
    else
        back = back->next = new ListNode( x );
}

// Deep copy.
template <class Object>
const Queue<Object> & Queue<Object>::operator=( const Queue<Object> & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        ListNode *rptr;
        for( rptr = rhs.front; rptr != NULL; rptr = rptr->next )
            enqueue( rptr->element );
    }
    return *this;
}

```

```cpp
#include "StackLi.h"

#ifdef USE_DOT_H
    #include <iostream.h>
#else
    #include <iostream>
    using namespace std;
#endif

int main( )
{
    Stack<int> s, s1;

    for( int i = 0; i < 10; i++ )
        s.push( i );

    s1 = s;

    cout << "s" << endl;
    while( !s.isEmpty( ) )
        cout << s.topAndPop( ) << endl;

    cout << endl << "s1" << endl;
    while( !s1.isEmpty( ) )
        cout << s1.topAndPop( ) << endl;

    return 0;
}
```

```
#ifndef STACKLI_H_
#define STACKLI_H_

#include <stdlib.h>
#include "Except.h"


// Stack class -- linked list implementation.
//
// CONSTRUCTION: with no parameters.
//
// ******************PUBLIC OPERATIONS*********************
// void push( x )         --> Insert x
// void pop( )            --> Remove most recently inserted item
// Object top( )          --> Return most recently inserted item
// Object topAndPop( )    --> Return and remove most recently inserted item
// bool isEmpty( )        --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// UnderflowException thrown as needed.

template <class Object>
class Stack
{
  public:
    Stack( );
    Stack( const Stack & rhs );
    ~Stack( );

    bool isEmpty( ) const;
    const Object & top( ) const;

    void makeEmpty( );
    void pop( );
    void push( const Object & x );
    Object topAndPop( );

    const Stack & operator=( const Stack & rhs );

  private:
    struct ListNode
    {
        Object    element;
        ListNode *next;

        ListNode( const Object & theElement, ListNode * n = NULL )
          : element( theElement ), next( n ) { }
    };

    ListNode *topOfStack;
};

```

```cpp
#include "StackLi.cpp"
#endif


```

```cpp
#include "StackLi.h"


// Construct the stack.
template <class Object>
Stack<Object>::Stack( )
{
    topOfStack = NULL;
}

// Copy constructor.
template <class Object>
Stack<Object>::Stack( const Stack<Object> & rhs )
{
    topOfStack = NULL;
    *this = rhs;
}

// Destructor.
template <class Object>
Stack<Object>::~Stack( )
{
    makeEmpty( );
}

// Test if the stack is logically empty.
// Return true if empty, false, otherwise.
template <class Object>
bool Stack<Object>::isEmpty( ) const
{
    return topOfStack == NULL;
}

// Make the stack logically empty.
template <class Object>
void Stack<Object>::makeEmpty( )
{
    while( !isEmpty( ) )
        pop( );
}

// Return the most recently inserted item in the stack.
// or throw an UnderflowException if empty.
template <class Object>
const Object & Stack<Object>::top( ) const
{
    if( isEmpty( ) )
        throw UnderflowException( );
    return topOfStack->element;
}

// Remove the most recently inserted item from the stack.
// Throw Underflow if the stack is empty.
```

```cpp
template <class Object>
void Stack<Object>::pop( )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    ListNode *oldTop = topOfStack;
    topOfStack = topOfStack->next;
    delete oldTop;
}

// Return and remove the most recently inserted item
// from the stack.
template <class Object>
Object Stack<Object>::topAndPop( )
{
    Object topItem = top( );
    pop( );
    return topItem;
}

// Insert x into the stack.
template <class Object>
void Stack<Object>::push( const Object & x )
{
    topOfStack = new ListNode( x, topOfStack );
}

// Deep copy.
template <class Object>
const Stack<Object> & Stack<Object>::operator=( const Stack<Object> & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
        if( rhs.isEmpty( ) )
            return *this;

        ListNode *rptr = rhs.topOfStack;
        ListNode *ptr  = new ListNode( rptr->element );
        topOfStack = ptr;

        for( rptr = rptr->next; rptr != NULL; rptr = rptr->next )
            ptr = ptr->next = new ListNode( rptr->element );
    }
    return *this;
}
```

```cpp

#include "Except.h"
#include "BinaryHeap.h"

#ifdef USE_DOT_H
    #include <iostream.h>
#else
    #include <iostream>
    using namespace std;
#endif

// Test program
int main( )
{
    int numItems = 10000;
    BinaryHeap<int> h;
    int i = 37;
    int x;

    try
    {
        for( i = 37; i != 0; i = ( i + 37 ) % numItems )
            h.insert( i );
        for( i = 1; i < numItems; i++ )
        {
            h.deleteMin( x );
            if( x != i )
                cout << "Oops! " << i << endl;
        }
        for( i = 37; i != 0; i = ( i + 37 ) % numItems )
            h.insert( i );
        h.insert( 0 );
    }
    catch( const DSException & e )
      { cout << e.toString( )  << endl; }

    return 0;
}

```

```cpp
#ifndef BINARY_HEAP_H_
#define BINARY_HEAP_H_

#include "Except.h"
#include "vector.h"

#include "StartConv.h"

// BinaryHeap class.
//
// CONSTRUCTION: with no parameters or vector containing items.
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void deleteMin( )      --> Remove smallest item
// void deleteMin( min )  --> Remove and send back smallest item
// Comparable findMin( )  --> Return smallest item
// bool isEmpty( )        --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Throws UnderflowException as warranted.

template <class Comparable>
class BinaryHeap
{
  public:
    BinaryHeap( );
    BinaryHeap( const vector<Comparable> & v );

    bool isEmpty( ) const;
    const Comparable & findMin( ) const;

    void insert( const Comparable & x );
    void deleteMin( );
    void deleteMin( Comparable & minItem );
    void makeEmpty( );

  private:
    int                theSize;  // Number of elements in heap
    vector<Comparable> array;    // The heap array

    void buildHeap( );
    void percolateDown( int hole );
};

#include "EndConv.h"
#include "BinaryHeap.cpp"
#endif
```

```cpp
#include "BinaryHeap.h"
#include "StartConv.h"

// Construct the binary heap.
template <class Comparable>
BinaryHeap<Comparable>::BinaryHeap( )
  : array( 11 ), theSize( 0 )
{
}

// Construct the binary heap.
// v is a vector containing the initial items.
template <class Comparable>
BinaryHeap<Comparable>::BinaryHeap( const vector<Comparable> & v )
  : array( v.size( ) + 1 ), theSize( v.size( ) )
{
    for( int i = 0; i < v.size( ); i++ )
        array[ i + 1 ] = v[ i ];
    buildHeap( );
}

// Insert item x into the priority queue, maintaining heap order.
// Duplicates are allowed.
template <class Comparable>
void BinaryHeap<Comparable>::insert( const Comparable & x )
{
    array[ 0 ] = x;   // initialize sentinel
    if( theSize + 1 == array.size( ) )
        array.resize( array.size( ) * 2 + 1 );

      // Percolate up
    int hole = ++theSize;
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}

// Find the smallest item in the priority queue.
// Return the smallest item, or throw UnderflowException if empty.
template <class Comparable>
const Comparable & BinaryHeap<Comparable>::findMin( ) const
{
    if( isEmpty( ) )
        throw UnderflowException( );
    return array[ 1 ];
}

// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( )
{
    if( isEmpty( ) )
```

```cpp
        throw UnderflowException( );

    array[ 1 ] = array[ theSize-- ];
    percolateDown( 1 );
}

// Remove the smallest item from the priority queue
// and place it in minItem. Throw UnderflowException if empty.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( Comparable & minItem )
{
    minItem = findMin( );
    array[ 1 ] = array[ theSize-- ];
    percolateDown( 1 );
}

// Establish heap-order property from an arbitrary
// arrangement of items. Runs in linear time.
template <class Comparable>
void BinaryHeap<Comparable>::buildHeap( )
{
    for( int i = theSize / 2; i > 0; i-- )
        percolateDown( i );
}

// Test if the priority queue is logically empty.
// Return true if empty, false otherwise.
template <class Comparable>
bool BinaryHeap<Comparable>::isEmpty( ) const
{
    return theSize == 0;
}

// Make the priority queue logically empty.
template <class Comparable>
void BinaryHeap<Comparable>::makeEmpty( )
{
    theSize = 0;
}

// Internal method to percolate down in the heap.
// hole is the index at which the percolate begins.
template <class Comparable>
void BinaryHeap<Comparable>::percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];

    for( ; hole * 2 <= theSize; hole = child )
    {
        child = hole * 2;
        if( child != theSize && array[ child + 1 ] < array[ child ] )
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}

#include "EndConv.h"


```