Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

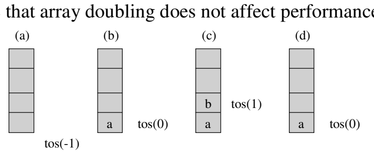
CHAPTER 16
Stacks and Queues: Implementations

Contents

Dynamic Array Implementations

Stacks

- A stack can be implemented with an array and an integer that indicates the index of the top element (*tos* ➔ *topOfStack*).
- Most of the stack routines are applications of previously discussed ideas.
- Note that array doubling does not affect performance in the long run.



How the stack routines works: **(a)** empty stack; **(b)** *push (a)*; **(c)** *push (b)*; and **(d)** *pop*

Stack class -- array implementation: interface

```
// Stack class -- array implementation  ( StackAr.h)

// 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( );

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

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

  private:
    vector<Object> theArray;
    int            topOfStack;
};

#include StackAr.cpp
```

Member function implementations (StackAr.cpp)

```
#include "StackAr.h"

// Construct the stack.
template <class Object>
Stack<Object>::Stack( ) : theArray( 1 )
{  topOfStack = -1; }

// Test if the stack is logically empty. Return true if empty, false, otherwise.
template <class Object>
bool Stack<Object>::isEmpty( ) const
{ return topOfStack == -1; }

// Make the stack logically empty.
template <class Object>
void Stack<Object>::makeEmpty( )
{  topOfStack = -1; }

// Return the most recently inserted item in the stack. Does not alter the stack.
// Throws UnderflowException if stack is already empty.
template <class Object>
const Object & Stack<Object>::top( ) const
{ if( isEmpty( ) ) throw UnderflowException( );
  return theArray[ topOfStack ];
}
```


Member function implementations (cont.)

```
// Remove the most recently inserted item from the stack.
// Throws UnderflowException if stack is already empty.
template <class Object>
void Stack<Object>::pop( )
{
   if( isEmpty( ) ) throw UnderflowException( );
   topOfStack--;
}

// Insert x into the stack.
template <class Object>
void Stack<Object>::push( const Object & x )
{
   if( topOfStack == theArray.size( ) - 1 )
      theArray.resize( theArray.size( ) * 2 + 1 );
   theArray[ ++topOfStack ] = x;
}

// Return and remove most recently inserted item from the stack.
// Throws UnderflowException if stack is already empty.
template <class Object>
Object Stack<Object>::topAndPop( )
{
   if( isEmpty( ) ) throw UnderflowException( );
   return theArray[ topOfStack-- ];
}
```


Test program

```
#include "StackAr.h"
#include <iostream>
using namespace std;

int main( )
{
   Stack<int> s;

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

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

   return 0;
}
```


Dynamic Array Implementations

Queues

- A stack can be implemented with an array and two integers that indicate the index of the first element and the index of the last element (*front, back*).
- Most of the queue routines are applications of previously discussed ideas, but ...
  - Storing the queue items beginning at the start of any array makes dequeueing expensive.
  - A dequeue can be implemented by incrementing the front position.
  - One can use wraparound: wraparound returns front or back to the beginning of the array when either reaches the end. Using wraparound to implement the queue is called a circular array implementation.
    - If the queue is full, we must implement array doubling carefully.
    - When we double the queue array, we cannot simply copy the entire array directly.

Queue class -- array implementation: interface

```
// Queue class -- array implementation  ( QueueAr.h)

// 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( );

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

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

  private:
    vector<Object> theArray;
    int          currentSize;
    int          front;
    int          back;

    void increment( int & x ) const;
    void doubleQueue( );
};
#include "QueueAr.cpp"
```


Member function implementations (QueueAr.cpp)

```
#include "QueueAr.h"

// Construct the queue.
template <class Object>
Queue<Object>::Queue( ) : theArray( 1 )
{  makeEmpty( ); }

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

// Make the queue logically empty.
template <class Object>
void Queue<Object>::makeEmpty( )
{  currentSize = 0;
   front = 0;
   back = theArray.size( ) - 1;
}

// 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 theArray[ front ];
}
```


Member function implementations (cont.)

```
// Return and remove the least recently inserted item from the queue.
// Throws UnderflowException if queue is already empty.
template <class Object>
Object Queue<Object>::dequeue( )
{
   if( isEmpty( ) )
          throw UnderflowException( );

   currentSize--;

   Object frontItem = theArray[ front ];
   increment( front );
   return frontItem;
}

// Insert x into the queue.
template <class Object>
void Queue<Object>::enqueue( const Object & x )
{
   if( currentSize == theArray.size( ) )
         doubleQueue( );
   increment( back );
   theArray[ back ] = x;
   currentSize++;
}
```


Member function implementations (cont.)

```
// Internal method to double capacity.
template <class Object>
void Queue<Object>::doubleQueue( )
{  theArray.resize( theArray.size( ) * 2 + 1 );
   if( front != 0 )
   {
      for( int i = 0; i < front; i++ )
         theArray[ i + currentSize ] = theArray[ i ];
      back += currentSize;
   }
}

// Internal method to increment x with wraparound.
template <class Object>
void Queue<Object>::increment( int & x ) const
{
   if( ++x == theArray.size( ) )
      x = 0;
}
```

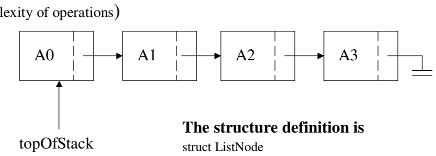
Test program

```
#include "QueueAr.h"
#include <iostream>
using namespace std;

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;
}
```

Linked List Implementations

Stacks

- In implementing the *Stack* class, the top of the stack is represented by the first item in a linked list. (why not the last item?, complexity of operations)




```
ListNode *topOfStack;
```

The structure definition is

```
struct ListNode
{
     Object element;   //Some element
     ListNode *next;
};
```


Stack class – list implementation: interface

```
// Stack class -- linked list implementation ( StackLi.h)
//
// 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;
};
#include "StackLi.cpp"
```


Member function implementations (StackLi.cpp)

```
#include "StackLi.h“

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

// 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( ); }

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


Member function implementations (cont.)

```
// 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.
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;
}
```

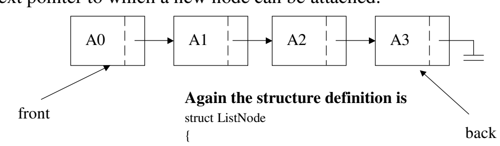
```
#include "StackLi.h"
#include <iostream>
using namespace std;

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;
}
```

Test program

Linked List Implementations

Queues

- A linked list in which we maintain a pointer to the first and last item can be used to implement the queue in constant time per operation.
- Enqueueing the first element is a special case because there is no next pointer to which a new node can be attached.



Again the structure definition is

```
ListNode *front;
ListNode *back;

struct ListNode
{
    Object element;   //Some element
    ListNode *next;
};
```


Queue class – list implementation: interface

```
// Queue class -- linked list implementation ( QueueLi.h)
//
// 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"
```


Member function implementations (QueueLi.cpp)

```
#include "QueueLi.h"

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

// 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( );
}

// 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 );
}
```

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

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

   return frontItem;
}
```

Member function implementations (cont.)


```
#include "QueueLi.h"
#include <iostream>
using namespace std;

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;
}
```

Test program

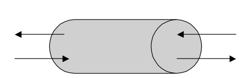
Comparison of the Two Methods

- The array versus linked list implementations represent a classic time-space trade-off.
  - Generally both the array and linked list versions run in constant time per operation.
  - The array versions of these data structures are likely to be faster than their linked list counterparts:
    1. especially, if an accurate estimation of capacity is available (no doubling is performed);
    2. the sequential access provided by an array is typically faster than the potential nonsequential access offered by dynamic memory allocation (linked list).
  - However, the array implementation does have two drawbacks:
    1. the code is more complex (because of wraparound and array doubling);
    2. when doubling we temporary require three times as much as the number of data items suggest.
- Summary
  1. The array implementation is often preferable for small objects
  2. The linked List implementation is preferable for large objects (if *Object* copies are so expensive that the cost of array doubling becomes significant)
  3. However, as stack and queue operations are performed in constant time, which implementation is used probably does not matter either.

Double-Ended Queues

- A double-ended queue (*deque*) allows access at both ends.
- Much of its functionality can be derived from the *Queue* class.
- Implementation of the *deque* is relatively simple with private inheritance.
- We derive it from array-based version of the *Queue* (since deletion from the back of the list is not efficiently supported.

- addFront(x)
- removeFront( )
- getFront( )



- addBack(x)
- removeBack( )
- getBack( )

deque

- All these operations can be implemented to run in constant time.

Deque class -- array implementation: interface

```
// Double ended queue (Deque) class
// -- array implementation  ( DequeAr.h)

// CONSTRUCTION: with no parameters.
// ******************PUBLIC OPERATIONS******
// void addFront( x )    --> Insert x at front
// void addBack( x )    --> Insert x at back
// void removeFront( )      --> Remove front item
// void removeBack( )      --> Remove back item
// Object getFront( )   --> Return front item
// Object getBack( )   --> Return back item
// bool isEmpty( )      --> Return true if empty; else false
// void makeEmpty( )    --> Remove all items
// ******************ERRORS******************
// UnderflowException thrown as needed.

template <class Object>
class Deque : private Queue<Object>
{
  public:
    void addFront( const Object & x );
    void addBack( const Object & x ) { enqueue( x ); }
    void removeFront( ) { dequeue( ); }
    void removeBack( );
    const Object & getFront( ) const { return Queue<Object>::getFront( ); }
    const Object & getBack( ) const;
    bool isEmpty( ) const
      { return Queue<Object>::is Empty( ); }
    void makeEmpty( )
      {Queue<Object>::makeEmpty( ); }
};
```


Homework

1. Problem 16.2 page 562 of the text
2. Problem 16.5 page 562 of the text
   (‘uses’ means ‘adds, removes, and prints the contents)
3. Problem 16.6 page 562 of the text