CMSC 341

Deques, Stacks and Queues

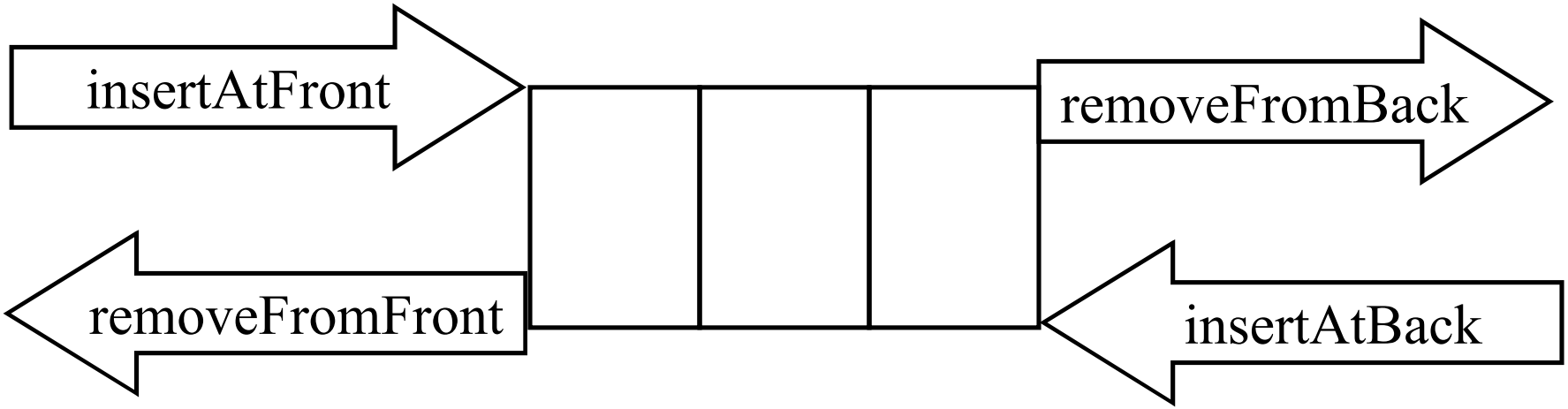
The Double-Ended Queue ADT

The double ended queue is referred to as a Deque (rhymes with “check”)

Restricted List

- add to the end
- remove from the end
- add to the front
- remove from the front

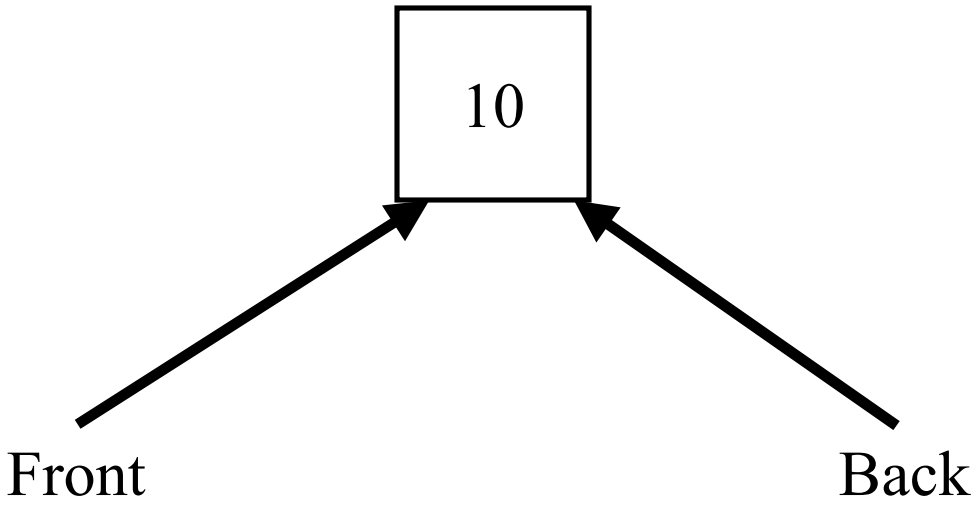
Stacks and Queues are often implemented using a Deque


insertAtFront
removeFromBack
removeFromFront
insertAtBack

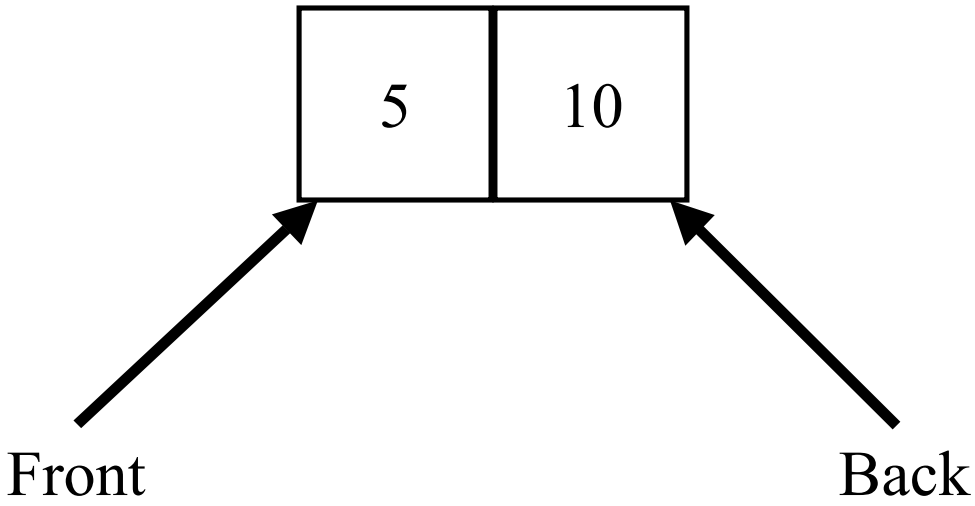
Front

Back

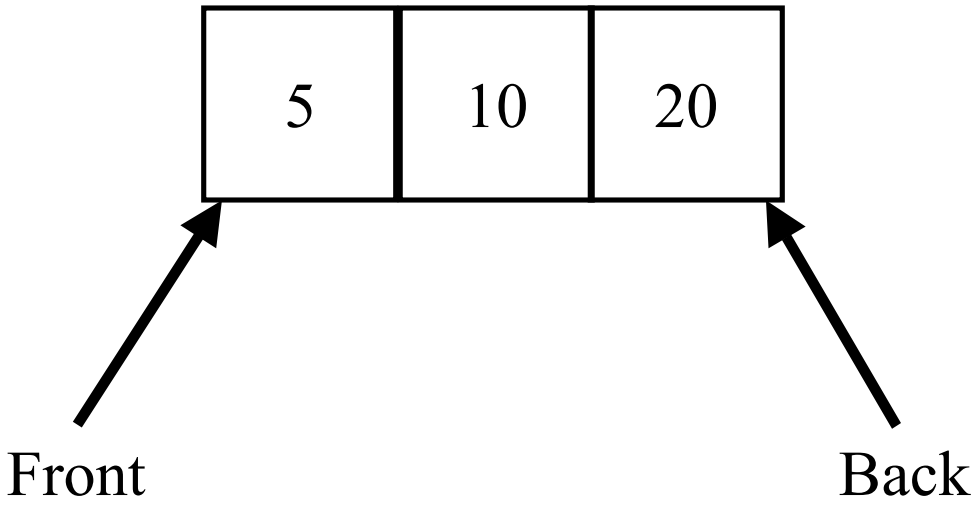
insertAtFront(10)



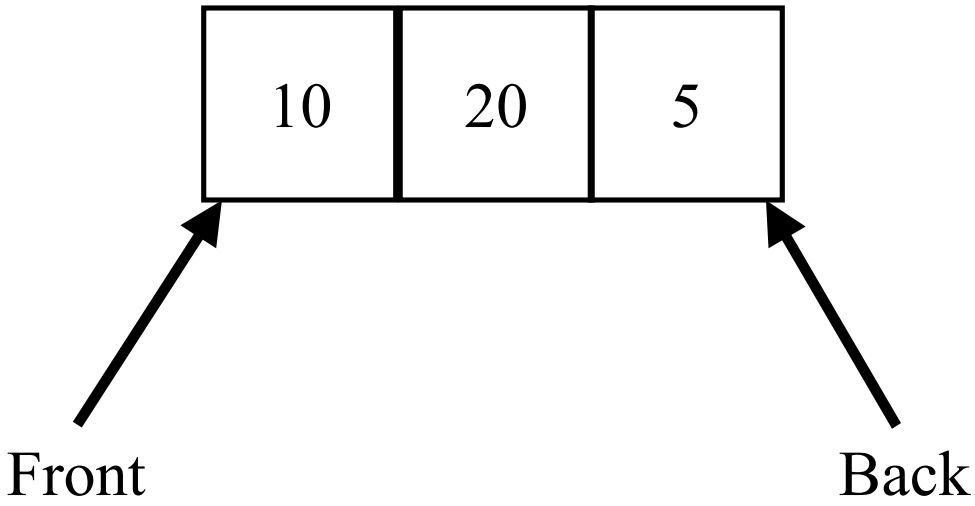
insertAtFront(5)



insertAtBack(20)



insertAtBack(removeFromFront())



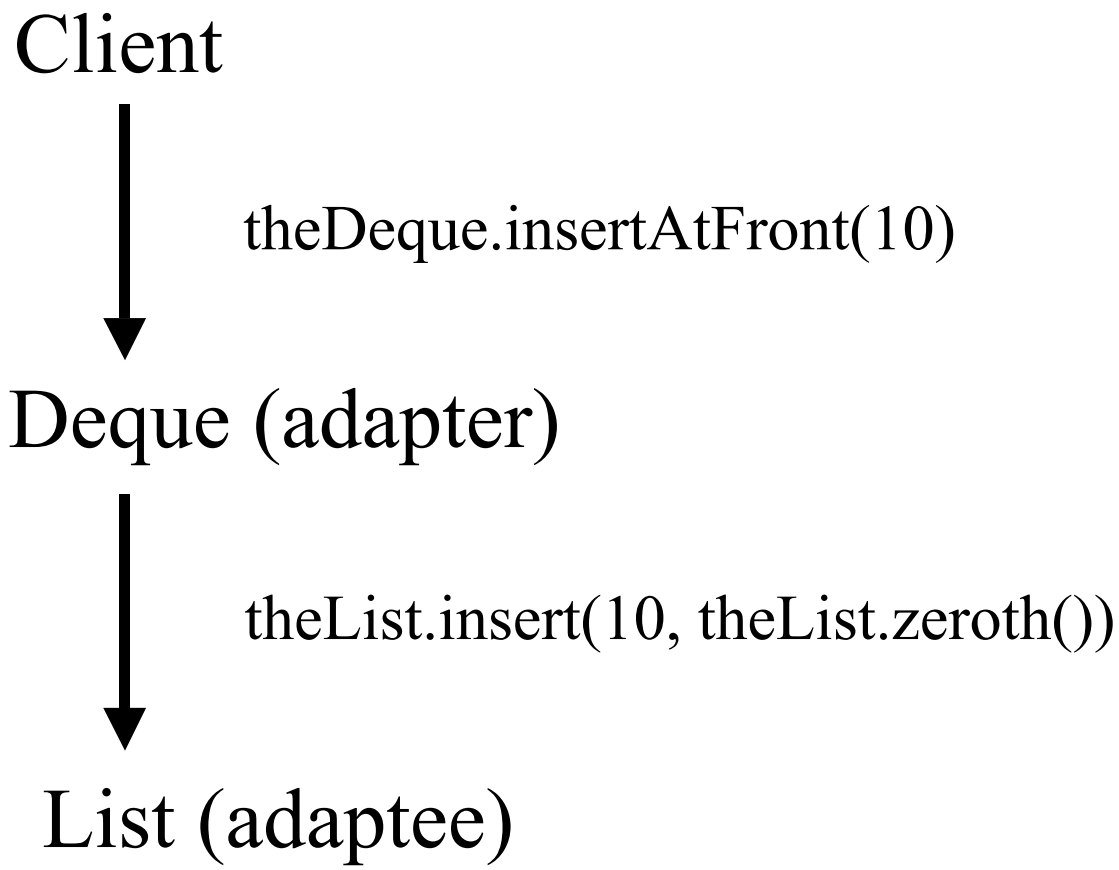
Adapting Lists to Implement Deques

List interface (Weiss)

- Does not directly support Deque operations
- Could support them with a little extra code
- Any good list interface will support Deque operations

Adapter Design Pattern

- Allow a client to use a class whose interface is different from the one expected by the client
- Do not modify client or class, write adapter class that sits between them


Client
theDeque.insertAtFront(10)
Deque (adapter)
theList.insert(10, theList.zeroth())
List (adaptee)

Deque.H

```
#include "LinkedList.H"
template <class Object>
class Deque {
public:
   Deque();
   Deque(const Deque &deq);
   ~Deque();
   bool isEmpty() const;
   void makeEmpty();
   void insertAtFront(const Object &x);
   void insertAtBack(const Object &x);
   Object removeFromFront();
   Object rremoveFromBack();
   const Deque &operator=(const Deque &deq);
```

Deque.H (cont)

```
private:
   List<Object> m_theList;
};
```

New List Methods Assumed to Exist

ListItr<Object> last( )

- Returns iterator that points to last object in list

void remove(ListItr<Object> itr)

- Removes the object pointed to by itr
- What is the state of itr after remove( )?

What are the complexity of last( ) and remove( )?

- Singly linked
- Doubly linked

Deque.C

```
template <class Object>
Deque<Object>::Deque()
{
   // no code
}

template <class Object>
Deque<Object>::Deque(const Deque &deq)
{
   m_theList = deq.m_theList;
}

template <class Object>
Deque<Object>::~Deque()
{
   // no code
}
```

Deque.C (cont)

```
template <class Object>
bool Deque<Object>::isEmpty( ) const
{
   return m_theList.isEmpty( );
}


template <class Object>
void Deque<Object>::makeEmpty ( )
{
   m_theList.makeEmpty();
}
```

Deque.C (cont)

```
template <class Object>
Object Deque<Object>::removeFromFront( )
{
   if (isEmpty())
       throw DequeException("remove on empty deque");
   Object tmp = m_theList.first().retrieve();
   m_theList.remove(m_theList.first( ));
   return tmp;
}


template <class Object>
void Deque<Object>::insertAtFront(const Object &x)
{
   m_theList.insert(x, m_theList.zeroth( ));
}
```

Deque.C (cont)

```
template <class Object>
Object Deque<Object>::removeFromBack( )
{
   if (isEmpty())
       throw DequeException("remove on empty deque");
   Object tmp = m_theList.last().retrieve();
   m_theList.remove(m_theList.last( ));
   return tmp;
}


template <class Object>
void Deque<Object>::insertAtBack(const Object &x)
{
   m_theList.insert(x, m_theList.last( ));
}
```

Deque.C (cont)

```
template <class Object>
const Deque<Object> &Deque<Object>::
operator=(const Deque &deq)
{
   if (this != &deq)
       m_theList = deq.m_theList;
   return *this;
}
```

DequeException.H

```
class DequeException
{
 public:
  DequeException(); // Message is the empty string
  DequeException(const string & errorMsg);
  DequeException(const DequeException & ce);
  ~DequeException();
  const DequeException &
       operator=(const DequeException & ce);
   const string & errorMsg() const; // Accessor for msg

 private:
  string m_msg;
};
```

DequeException.C

```
DequeException::DequeException( ){ /* no code */ }

DequeException::DequeException(const string & errorMsg)
{
  m_msg = errorMsg;
}

DequeException::DequeException(const DequeException &ce)
{
  m_msg = ce.errorMsg();
}

DequeException::~DequeException( ){ /* no code }
```

DequeException.C (cont)

```
const DequeException &
DequeException::operator=(const DequeException & ce)
{
  if (this == &ce)
    return *this;   // don't assign to itself
  m_msg = ce.errorMsg();
  return *this;
}

const string & DequeException::errorMsg() const
{
  return m_msg;
}
```

TestDeque.C

```
int main (){
  Deque<int> deq;

  deq.insertAtBack(1);
  deq.insertAtBack(2);
  printDeque(deq);

  Deque<int> otherdeq;
  otherdeq = deq;
  printDeque(otherdeq);

  cout << deq.removeFromFront() << endl;
  cout << deq.removeFromFront() << endl;
```

TestDeque.C (cont)

```
  printDeque(deq);
  printDeque(otherdeq);

  try {
      deq.removeFromFront();
  }
  catch (DequeException & e){
      cout << e.errorMsg() << endl;
  }
}
```

Queue ADT

Restricted List

- only add to end
- only remove from front

Examples

- line waiting for service
- jobs waiting to print

Implement using a Deque

Queue.H

```
template <class Object>
class Queue {
public:
   Queue();
   ~Queue();
   bool isEmpty() const;
   void makeEmpty();
   Object dequeue();
   void enqueue (const Object & x);

private:
   Deque<Object> m_theDeque;
}
```

Queue.C

```
template <class Object>
Queue<Object>::Queue()
{   /* no code */   }

template <class Object>
Queue<Object>::~Queue()
{  /* no code * /  }

template <class Object>
void Queue<Object>::makeEmpty( )
{
   m_theDeque.makeEmpty();
}
```

Queue.C (cont'd)

```
template <class Object>
void Queue<Object>::enqueue(const Object &x)
{
   m_theDeque.insertAtBack(x);
}


template <class Object>
Object Queue<Object>::dequeue()
{
   return m_theDeque.removeFromFront( );
}
```

An Alternative Queue.H

```
template <class Object>
class Queue {
public:
   Queue(int capacity = 10);
   ~Queue();
   bool isEmpty() const;
   void makeEmpty();
   Object dequeue();
   void enqueue (const Object & x);

private:
   vector<Object> m_theArray;
   int m_currentSize;
   int m_front;
   int m_back;
   void increment (int &x);
}
```

Queue.C

```
template <class Object>
Queue<Object>::Queue( int capacity )
   : m_theArray( capacity )
{
   makeEmpty( );
}

// make queue logically empty
template <class Object>
void Queue<Object>::makeEmpty( )
{
    m_currentSize = 0;
    m_front = 0;
    m_back = -1;
}
```

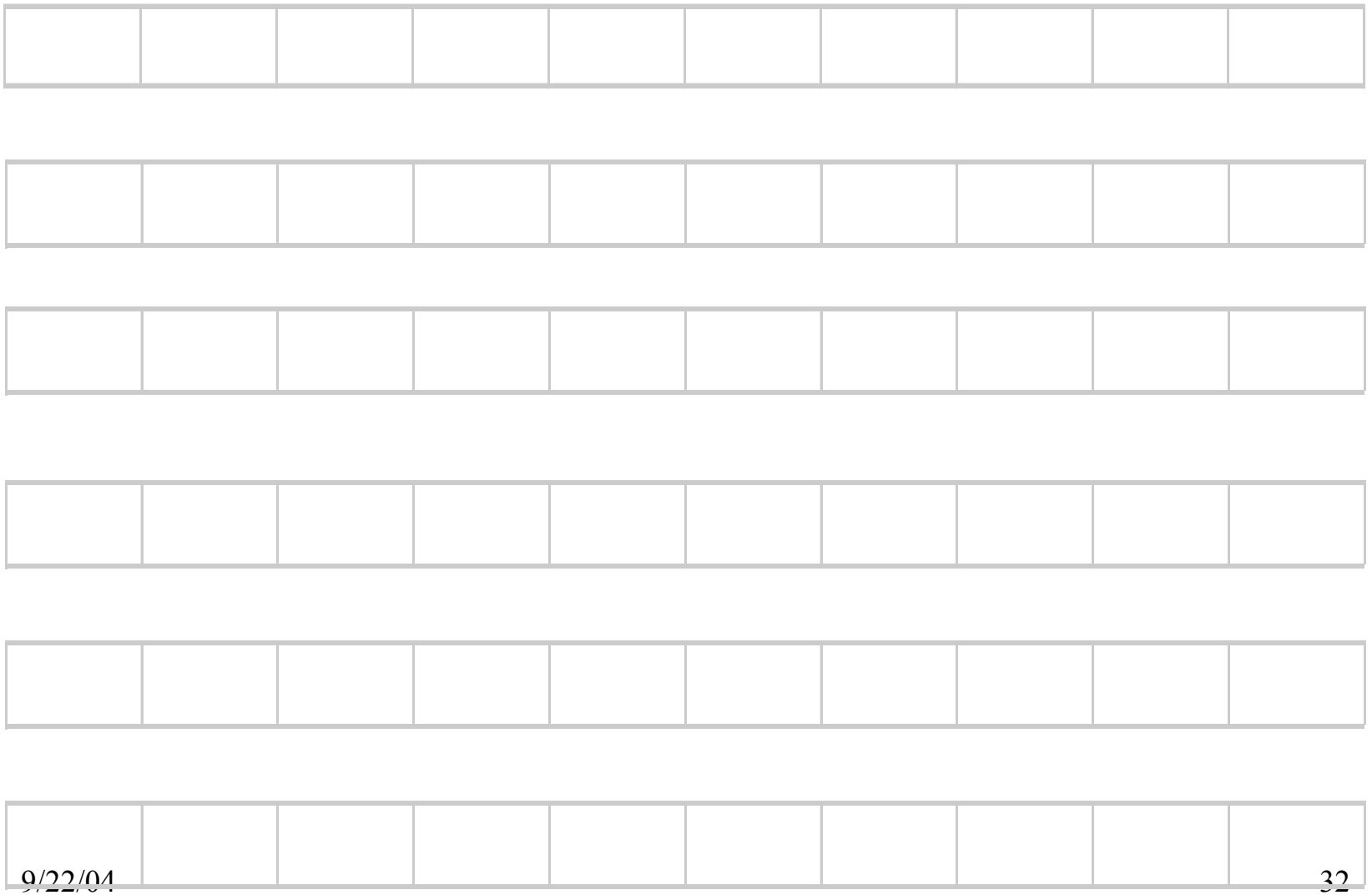
Queue.C (cont'd)

```
template <class Object>
void Queue<Object>::enqueue(const Object &x)
{
   if (isFull())
       throw Overflow();
   increment (m_back);
   m_theArray[m_back] = x;
   m_currentSize++;
}
template <class Object>
void Queue<Object>::increment(int &x)
 {
   if (++x == m_theArray.size())
       x = 0;
}
```

Queue.C (cont)

```
template <class Object>
Object Queue<Object>::dequeue()
{
   if (isEmpty())
       throw Underflow();
   m_currentSize--;
   Object frontItem = m_theArray[m_front];
   increment(m_front);
   return frontItem;
}
```

theArray



Stack ADT

Restricted List

- only add to top
- only remove from top

Examples

- pile of trays
- partial results
- local state

Implement using a Deque

Stack.H

```
template <class Object>
class Stack {
public:
   Stack( );
   ~Stack( );
   bool isEmpty( ) const;
   void makeEmpty( );

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

private:
   Deque<Object> m_theDeque;
};
```

Stack.C

```
template <class Object>
Stack<Object>::Stack()
{  /* no code */  }

template <class Object>
Stack<Object>::~Stack()
{  /* no code */ }

template <class Object>
void Stack<Object>::makeEmpty( )
{
   m_theDeque.makeEmpty();
}
```

Stack.C (cont'd)

```
template <class Object>
void Stack<Object>::push(const Object &x)
{
   m_theDeque.insertAtFront(x);
}


template <class Object>
Object Stack<Object>::pop()
{
   return m_theDeque.removeFromFront( );
}
```