CMSC 341

Deques, Stacks and Queues

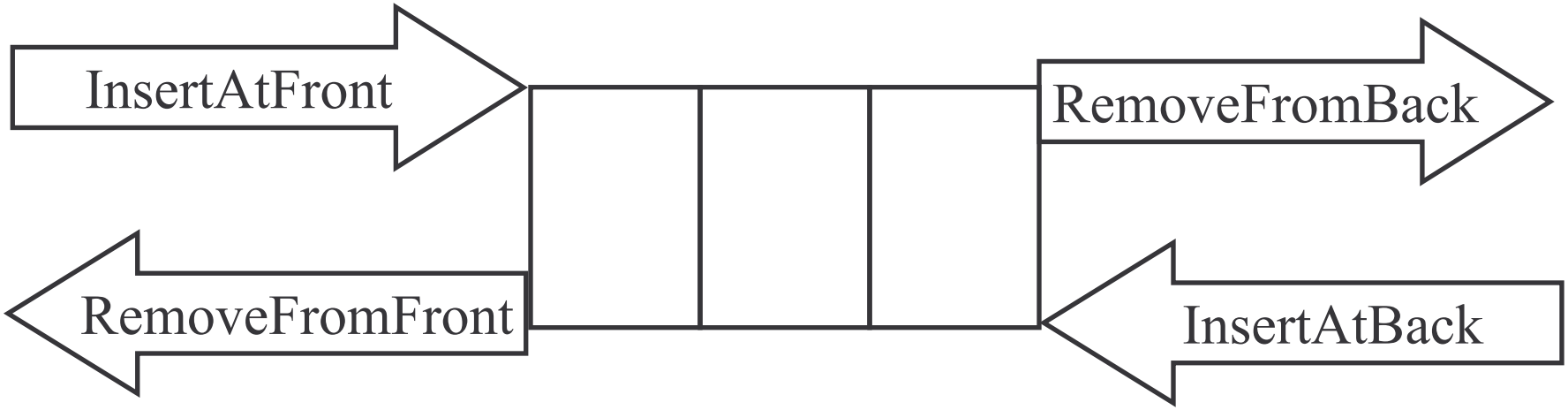
The Double-Ended Queue ADT

A Deque (rhymes with “check”) is a “Double Ended QUEue”.

A Deque is a restricted List which supports only

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

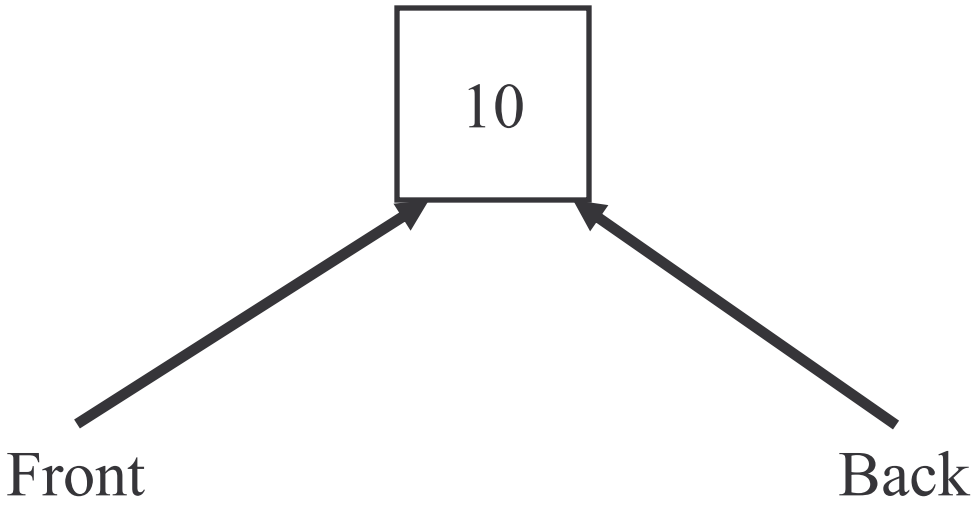
Stacks and Queues are often implemented using a Deque


InsertAtFront
RemoveFromBack
RemoveFromFront
InsertAtBack

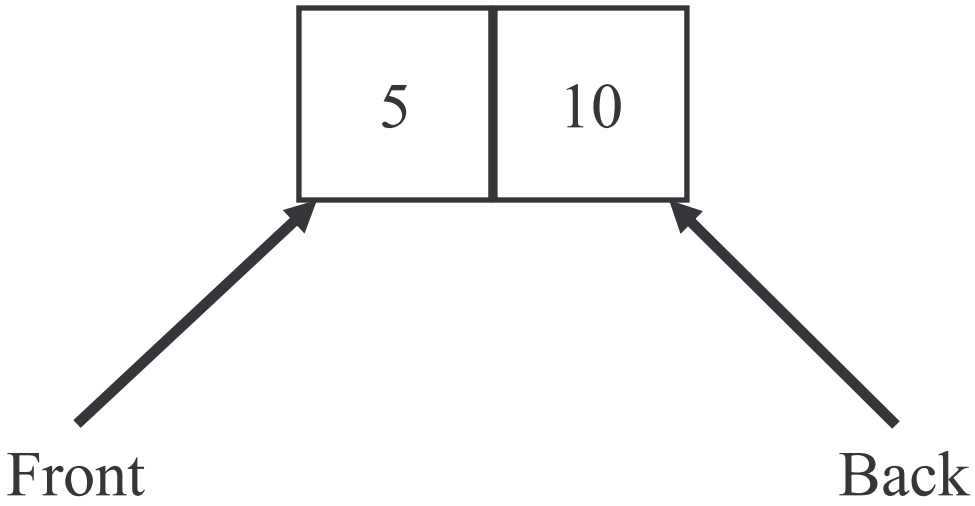
Front

Back

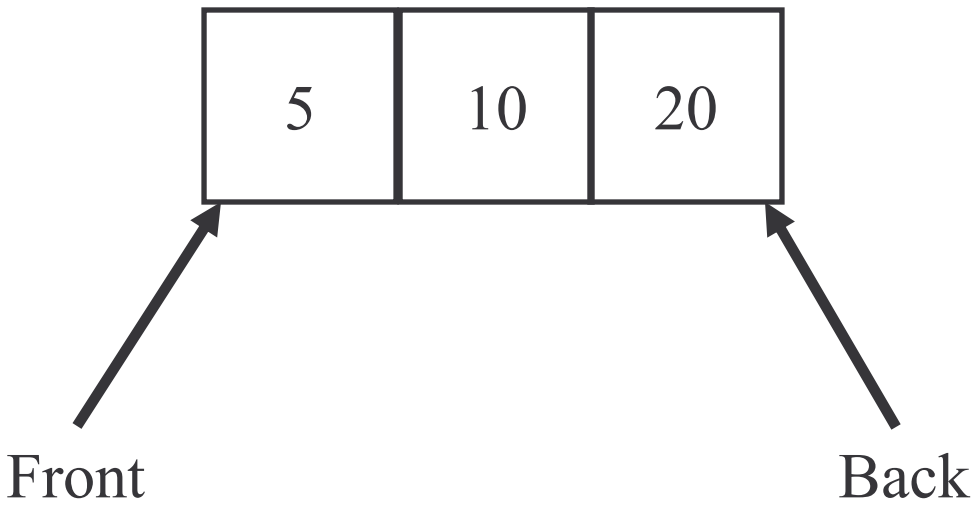
InsertAtFront(10)



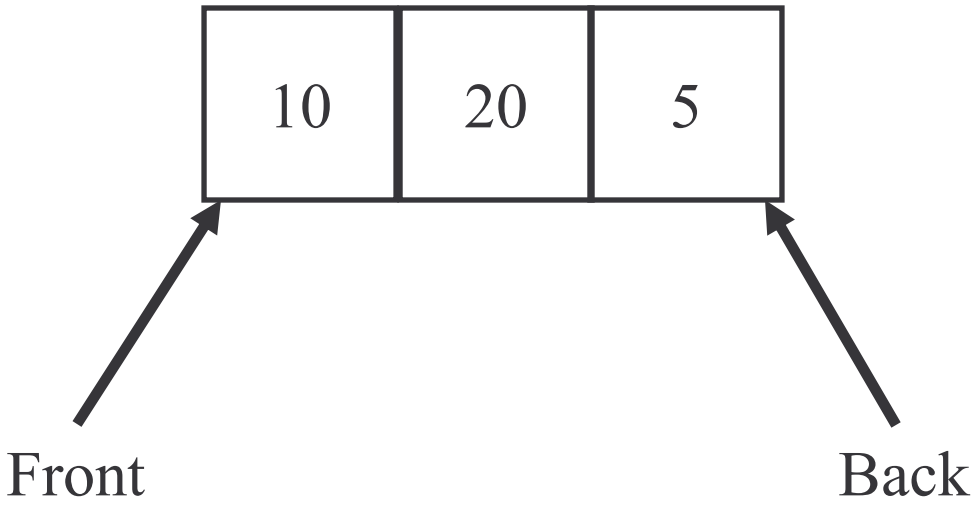
InsertAtFront(5)



InsertAtBack(20)



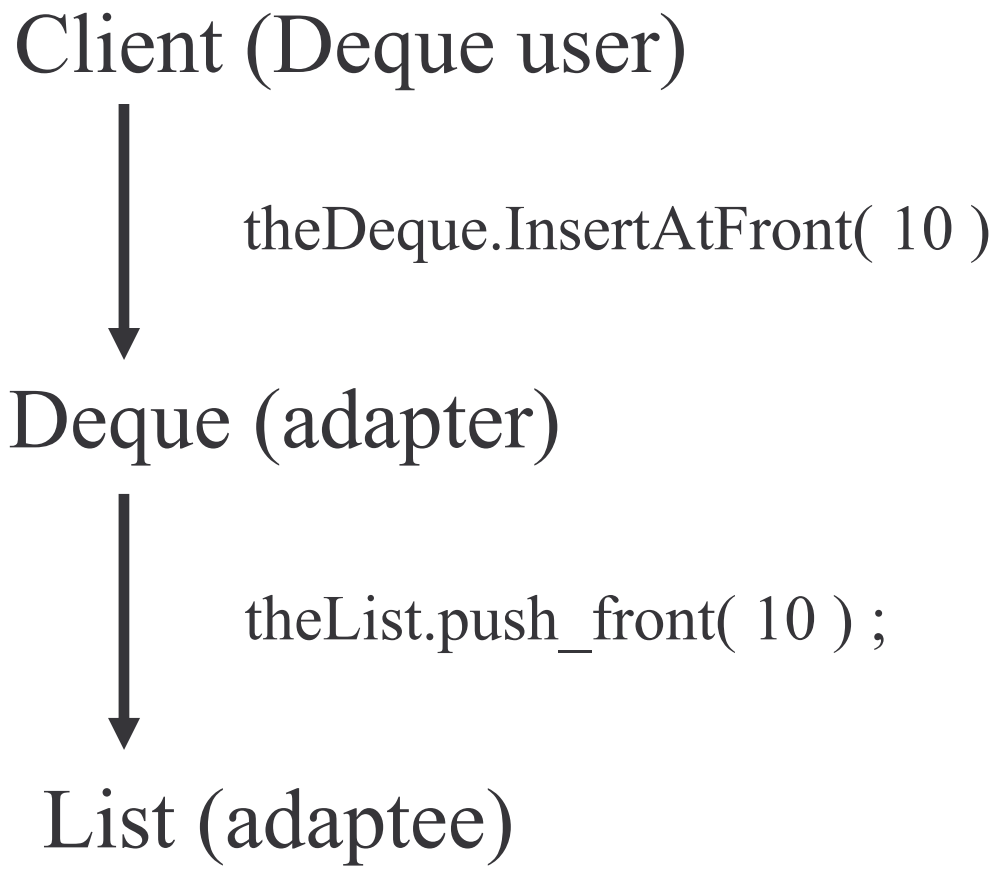
InsertAtBack(removeFromFront())



Adapting Lists to Implement Deques

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
- In this case, the Deque is an adapter for the List. The client (user) calls methods of the Deque which in turn calls appropriate List method(s).


Client (Deque user)
theDeque.InsertAtFront(10)
Deque (adapter)
theList.push_front(10) ;
List (adaptee)

Deque.h

```
#include “List.h”
template <typename 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 RemoveFromBack();
   Object Front( ) const;
   Object Back( ) const;
   const Deque &operator=(const Deque &deq);
private:
   List<Object> m_theList;
};
```

Deque methods

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

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

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

Deque methods (2)

```
template <typename Object>
bool Deque<Object>::IsEmpty( ) const
{
   return m_theList.empty( );
}


template <typename Object>
void Deque<Object>::MakeEmpty ( )
{
   m_theList.clear( );
}
```

Deque methods (3)

```
template <typename Object>
Object Deque<Object>::RemoveFromFront( )
{
   if (isEmpty())
       throw DequeException("remove on empty deque");
   Object tmp = m_theList.front( );
   m_theList.pop_front( );
   return tmp;
}
```

```
template <typename Object>
void Deque<Object>::InsertAtFront(const Object &x)
{
   m_theList.push_front( x );
}
```

Deque methods (4)

```
template <typename Object>
Object Deque<Object>::RemoveFromBack( )
{
   if (isEmpty())
       throw DequeException("remove on empty deque");
   Object tmp = m_theList.back( );
   m_theList.pop_back( );
   return tmp;
}
```

```
template <typename Object>
void Deque<Object>::InsertAtBack(const Object &x)
{
   m_theList.push_back( x );
}
```

Deque methods (5)

```
// examine the element at the front of the Deque
template <typename Object>
Object Deque<Object>::Front( ) const
{
   if (isEmpty())
       throw DequeException("front on empty deque");
   return m_theList.front( );
}

// examine the element at the back of the Deque
template <typename Object>
Object Deque<Object>::Back( ) const
{
   if (isEmpty())
       throw DequeException("back on empty deque");
   return m_theList.back( );
}
```

Deque methods (6)

```
template <typename 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.cpp

```
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.cpp (cont'd)

```
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.cpp

```
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;
    }

    return 0;
}
```

Queue ADT

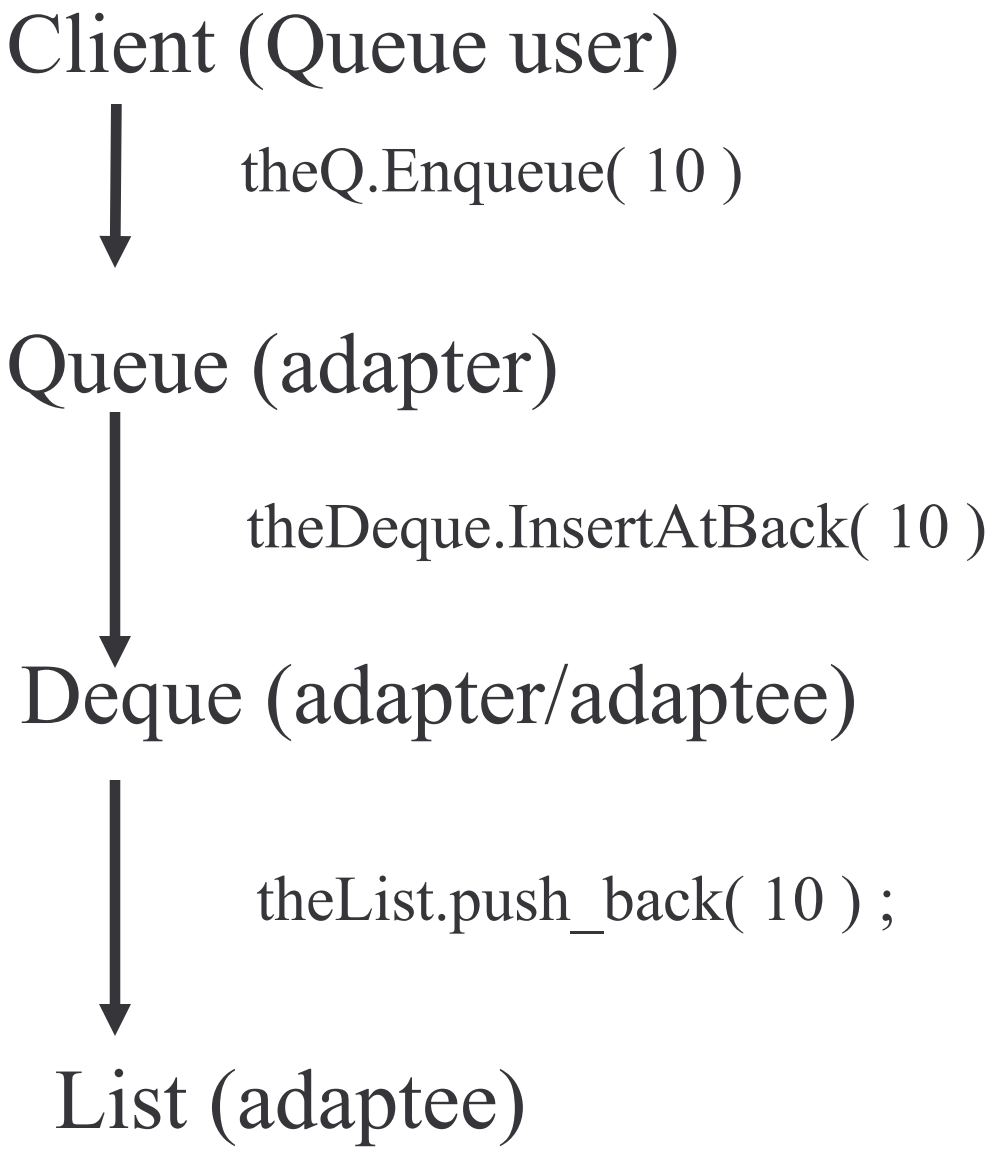
Restricted Deque

- only add to end
- only remove from front

Examples

- line waiting for service
- jobs waiting to print

Implement as an adapter of Deque


Client (Queue user)
theQ.Enqueue(10)
Queue (adapter)
theDeque.InsertAtBack(10)
Deque (adapter/adaptee)
theList.push_back(10) ;
List (adaptee)

Queue.h

```
template <typename Object>
class Queue
{
  public:
       Queue( );
       ~Queue( );
       bool IsEmpty( ) const;
       void MakeEmpty( );
       Object Dequeue( );
       void Enqueue (const Object & x);

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

Queue methods

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

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

template <typename Object>
void Queue<Object>::MakeEmpty( )
{
   m_theDeque.MakeEmpty( );
}
```

Queue methods (2)

```
template <typename Object>
void Queue<Object>::Enqueue(const Object &x)
{
   m_theDeque.InsertAtBack( x );
}


template <typename Object>
Object Queue<Object>::Dequeue( )
{
   return m_theDeque.RemoveFromFront( );
}
```

An Alternative Queue Implementation

```
template <typename Object>
class Queue {
   public:
        Queue(int capacity = 10);
        ~Queue();
        bool IsEmpty( ) const;
        bool IsFull ( ) 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 );
};
```

Alternate Queue methods

```
template <typename Object>
Queue<Object>::Queue( int capacity )
   : m_theArray( capacity )
{
   MakeEmpty( );
}

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

Alternate Queue methods (2)

```
template <typename Object>
void Queue<Object>::Enqueue(const Object &x)
{
   if ( IsFull( ) )
       throw Overflow( );
   Increment (m_back);
   m_theArray[m_back] = x;
   m_currentSize++;
}
template <typename Object>
void Queue<Object>::Increment( int &x )
 {
   if ( ++x == m_theArray.size( ) )
       x = 0;
}
```

Alternate Queue methods (3)

```
template <typename Object>
Object Queue<Object>::Dequeue( )
{
   if ( IsEmpty( ) )
       throw Underflow( );
   m_currentSize--;
   Object frontItem = m_theArray[m_front];
   Increment(m_front);
   return frontItem;
}
```

Stack ADT

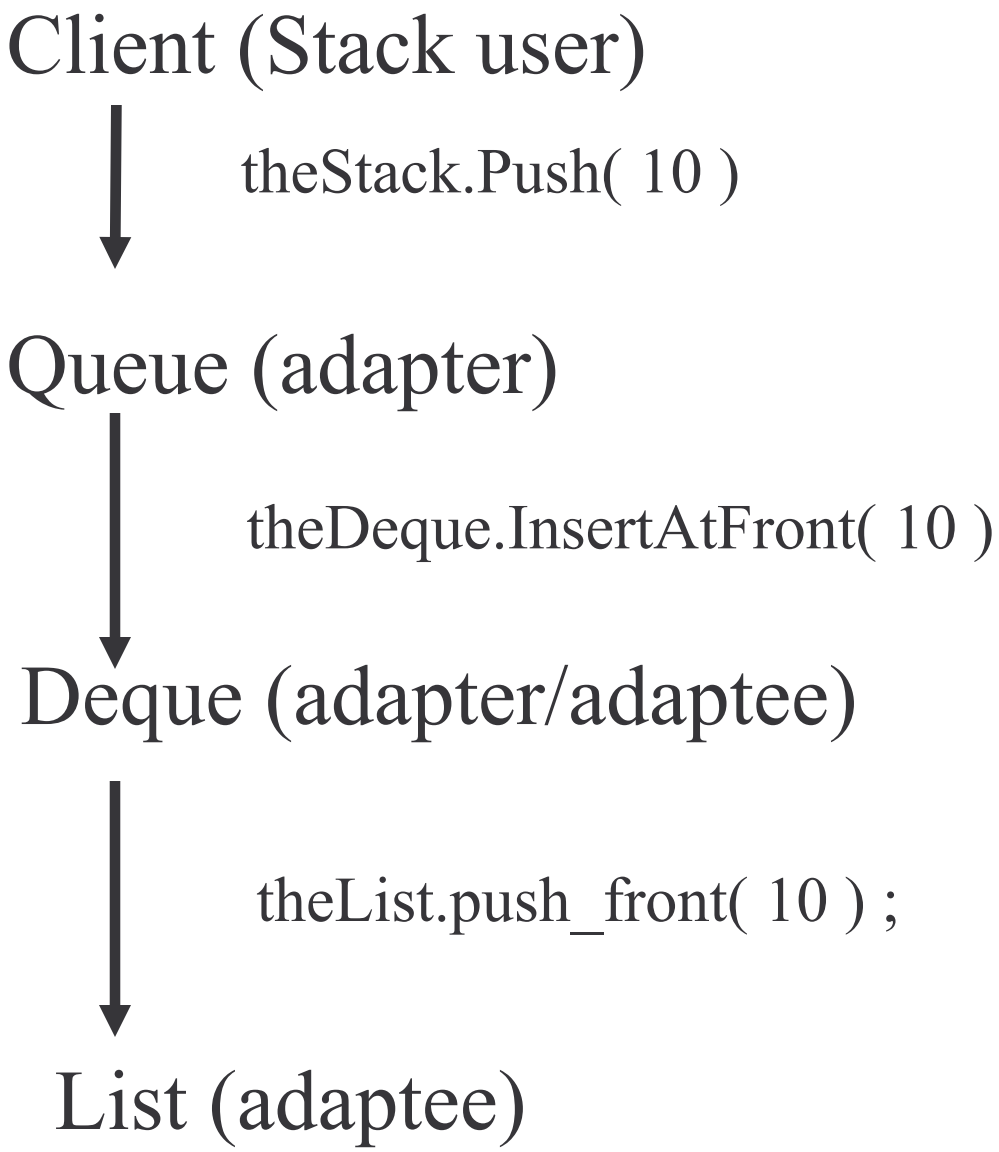
Restricted Deque

- only add to top (front)
- only remove from top (front)

Examples

- pile of trays
- partial results
- local state
- balancing parentheses

Implement as an adapter of Deque


Client (Stack user)
theStack.Push(10)
Queue (adapter)
theDeque.InsertAtFront(10)
Deque (adapter/adaptee)
theList.push_front(10) ;
List (adaptee)

Stack.h

```
template <typename Object>
class Stack {
public:
   Stack( );
   ~Stack( );
   bool IsEmpty( ) const;
   void MakeEmpty( );

   void Pop( );
   void Push( const Object &x );
   Object Top( ) const;

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

Stack methods

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

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

template <typename Object>
void Stack<Object>::MakeEmpty( )
{
   m_theDeque.MakeEmpty( );
}
```

Stack methods (2)

```
// “insert” a new element at the top of the stack
template <typename Object>
void Stack<Object>::Push( const Object &x )
{
   m_theDeque.InsertAtFront( x );
}

// remove the element at the top of the stack
template <typename Object>
Object Stack<Object>::Pop( )
{
   return m_theDeque.RemoveFromFront( );
}
// return the element at the top of the stack
template <typename Object>
Object Stack<Object>::Top( ) const
{
   return m_theDeque.Front( );
}
```