Adapters

Section 5.3.4

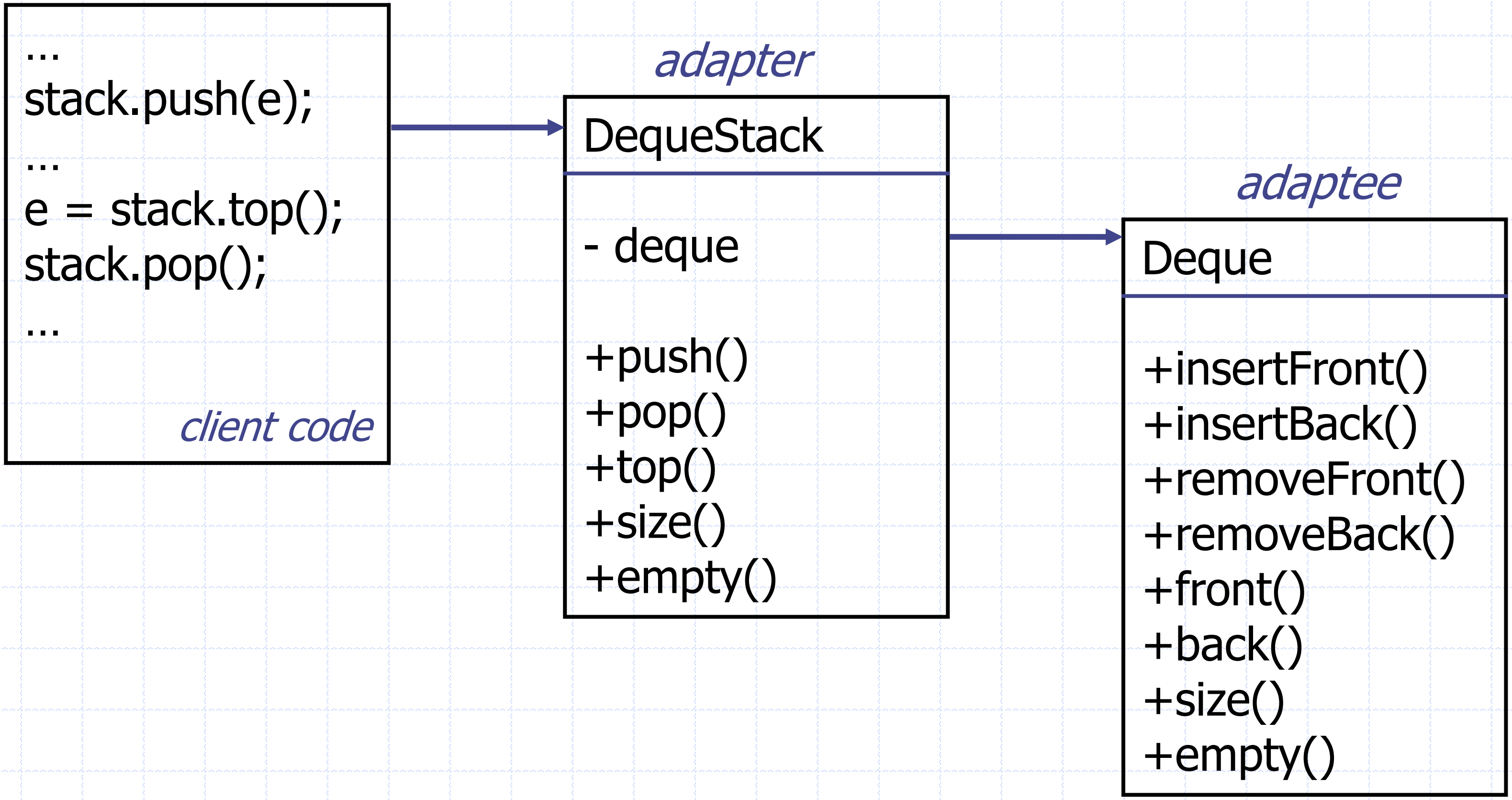
Adapter Design Pattern

- A design pattern is a way of creating objects that is a solution to a recurring design problem.
- The Adapter design pattern (sometimes called the Wrapper design pattern) is a solution to the problem that occurs when:
  - You have an object class that does what you want, or almost what you want,
  - This class has the wrong interface for the task,
  - And you can not or should not modify this class.

Example

- Suppose you have a class Deque that implements a deque with the standard Deque ADT interface.
- Suppose further that you need a stack with its standard interface.
- Here, the deque can handle the job of the stack, but it doesn't have the correct interface.
- There are three solutions that spring to mind:
  - Rewrite your Deque class so that it inserts and erases only from the front, and rename its functions to match that of the stack interface.
  - Rewrite the code that uses the stack to have it use a deque instead, calling insertFront instead of push, removeFront instead of pop, etc.
  - Build a class that implements a stack by using a deque. This is the adapter.

Example



```
…
stack.push(e);
…
e = stack.top();
stack.pop();
…
```

client code

adapter

DequeStack

- deque

+push()
+pop()
+top()
+size()
+empty()

adaptee

Deque

+insertFront()
+insertBack()
+removeFront()
+removeBack()
+front()
+back()
+size()
+empty()

UML Class Diagrams

- The preceding slide was an (informal) example of a class diagram.
- Class names are given at the top of a box representing the class, with a line underneath.
- Horizontal arrows from sides of boxes represent HAS_A relations.
- Vertical arrows from the tops of boxes (not shown) represent IS_A relations.
- Inside class boxes, + indicates public members and – indicates private members.

Coding an Adapter

- In an adapter, most functions use the adaptee in their implementation.
- When the implementation of a function is simply a call to another function, or the return of another function, it is called delegation.

```
typedef string Elem;
class DequeStack {
public:
   DequeStack();
    int size() const;
    bool empty() const;
    const Elem& top() const
        throw(StackEmpty);
    void push (const Elem& e);
    void pop()
        throw(StackEmpty);
private:
    LinkedDeque deque;
};
```

Coding an Adapter

```
DequeStack::DequeStack()
    : deque() { }

int DequeStack::size() const
    { return deque.size(); }

bool DequeStack::empty() const
    { return deque.empty(); }

const Elem& DequeStack::top() const
        throw(StackEmpty) {
    if(empty())
        throw StackEmpty(“top … “);
    return deque.front();
}
```

```
void DequeStack::push(const Elem& e)
    { deque.insertFront(e); }

void DequeStack::pop()
        throw(StackEmpty) {
    if(empty())
        throw StackEmpty(“pop …”);
    deque.removeFront();
}
```

An alternative: Exception translation

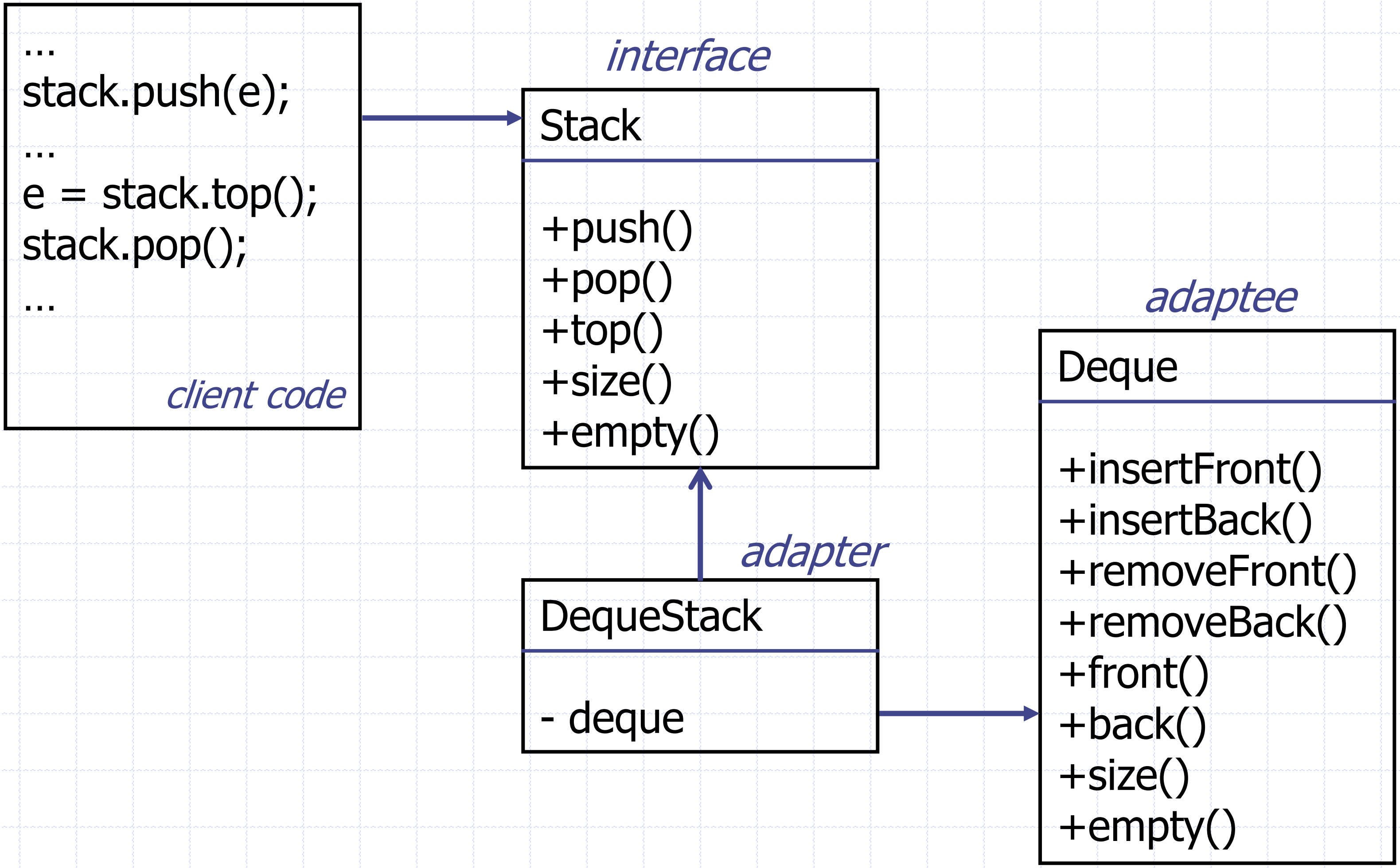
```
const Elem& DequeStack::top() const
      throw(StackEmpty) {

  try {
      return deque.front();
   }
   catch(DequeEmpty& exception) {
      throw StackEmpty(“top … “);
   }
}
```

```
void DequeStack::pop()
      throw(StackEmpty) {

   try {
      deque.removeFront();
   }
   catch(DequeEmpty& exception) {
      throw StackEmpty(“pop …”);
   }
}
```

Adapting to a Known Interface

- Often the adapter needs to have the interface of a known class or abstract class.
- In this case, the adapter should subclass the known class, and override its member functions.
- For example, suppose we already had an abstract class Stack for the stack interface.

Adapting to a Known Interface

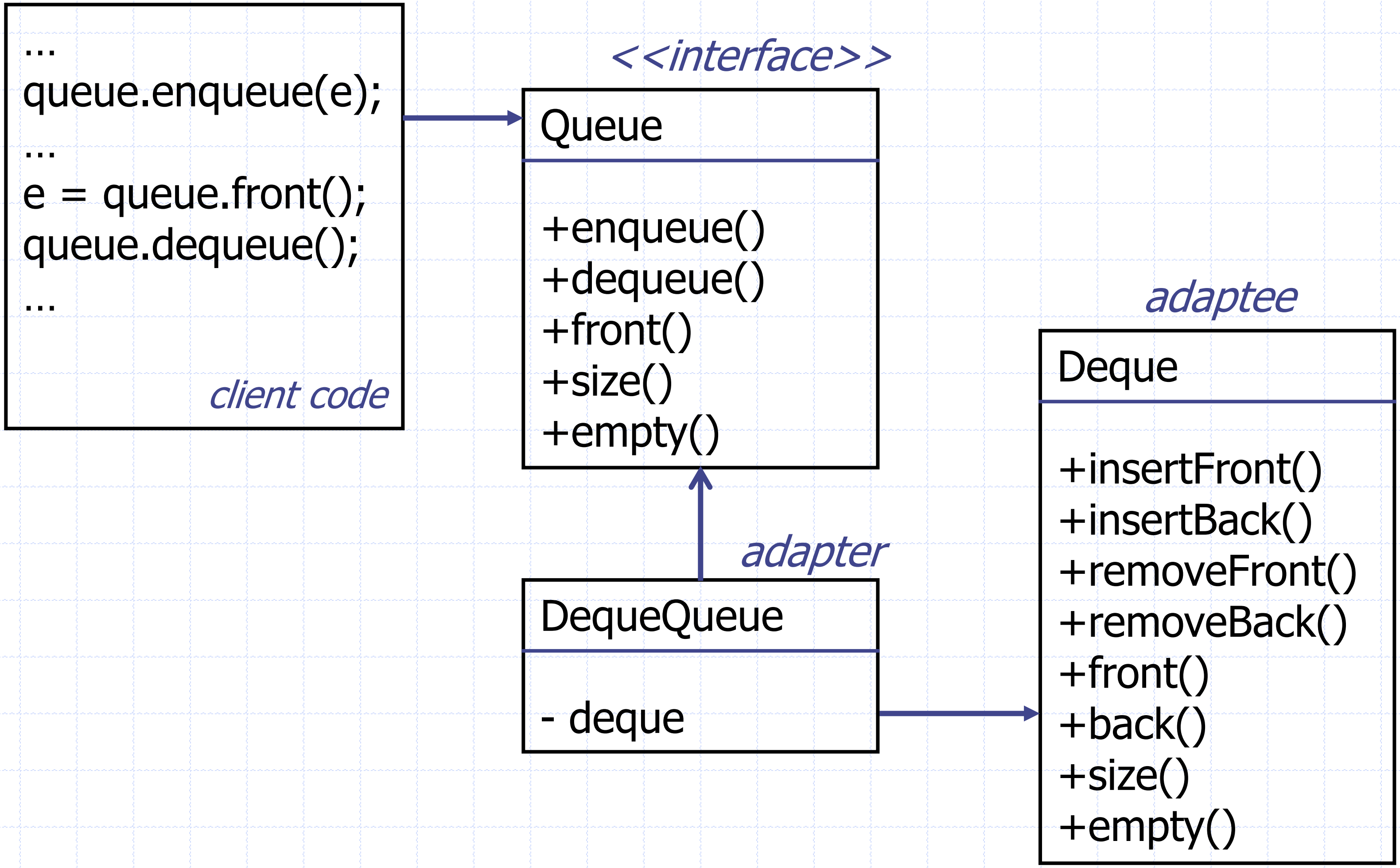


Code for Adapting to a Known Interface

```
typedef string Elem;
class Stack {
public:
    virtual int size() const = 0;
    virtual bool empty() const = 0;
    virtual const Elem& top() const
        throw(StackEmpty) = 0;
    virtual void push (const Elem& e) = 0;
    virtual void pop()
        throw(StackEmpty) = 0;
};
```

```
typedef string Elem;
class DequeStack : public Stack {
public:
   DequeStack();
    virtual int size() const;
    virtual bool empty() const;
    virtual const Elem& top() const
        throw(StackEmpty);
    virtual void push (const Elem& e);
    virtual void pop()
        throw(StackEmpty);
private:
    LinkedDeque deque;
};
```

Adapting a Deque to be a Queue



…
queue.enqueue(e);
…
e = queue.front();
queue.dequeue();
…

client code

<<interface>>

Queue

+enqueue()
+dequeue()
+front()
+size()
+empty()

adapter

DequeQueue

- deque

adaptee

Deque

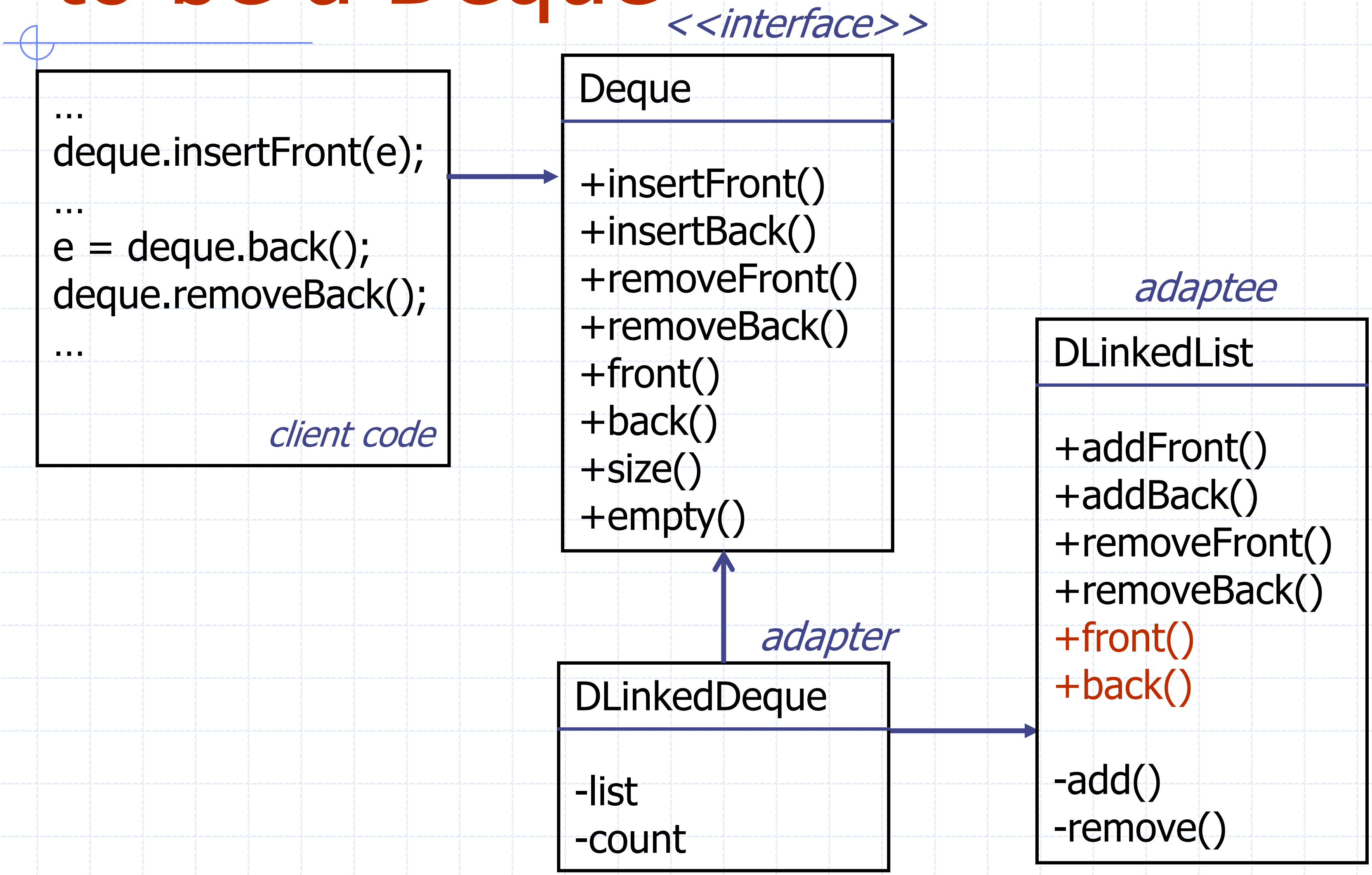
+insertFront()
+insertBack()
+removeFront()
+removeBack()
+front()
+back()
+size()
+empty()

Deque-based Queue

Queue Method	*Deque Implementation*
size()	size()
empty()	empty()
front()	front()
enqueue(e)	insertBack(e)
dequeue()	eraseFront()

```
int DequeQueue::size() const
    { return deque.size(); }

const Elem& DequeQueue::dequeue() const throw(QueueEmpty) {
    if(empty()) throw QueueEmpty(“dequeue called on empty queue“);
    deque.eraseFront();
}
```

Adapting a Doubly-Linked List to be a Deque



```
…
deque.insertFront(e);
…
e = deque.back();
deque.removeBack();
…
```

client code

<<interface>>

Deque

+insertFront()
+insertBack()
+removeFront()
+removeBack()
+front()
+back()
+size()
+empty()

adapter

DLinkedDeque

-list
-count

adaptee

DLinkedList

+addFront()
+addBack()
+removeFront()
+removeBack()
+front()
+back()

-add()
-remove()

Interface for Deque and Header for DLinkedDeque

Deque.h

```
typedef string Elem;
class Deque {
public:
    virtual int size() const = 0;
    virtual bool empty() const = 0;
    virtual const Elem& front() const
        throw(DequeEmpty) = 0;
    virtual const Elem& back() const
        throw(DequeEmpty) = 0;
    virtual void insertFront(const Elem& e) = 0;
    virtual void insertBack(const Elem& e) = 0;
    virtual void removeFront()
        throw(DequeEmpty) = 0;
    virtual void removeBack()
        throw(DequeEmpty) = 0;
};
```

DLinkedDeque.h

```
class DLinkedDeque : public Deque {
public:
    virtual int size() const;
    virtual bool empty() const;
    virtual const Elem& front() const
        throw(DequeEmpty);
    virtual const Elem& back() const
        throw(DequeEmpty);
    virtual void insertFront(const Elem& e);
    virtual void insertBack(const Elem& e);
    virtual void removeFront()
        throw(DequeEmpty);
    virtual void removeBack()
        throw(DequeEmpty);
private:
    DLinkedList list;
    int count;
};
```

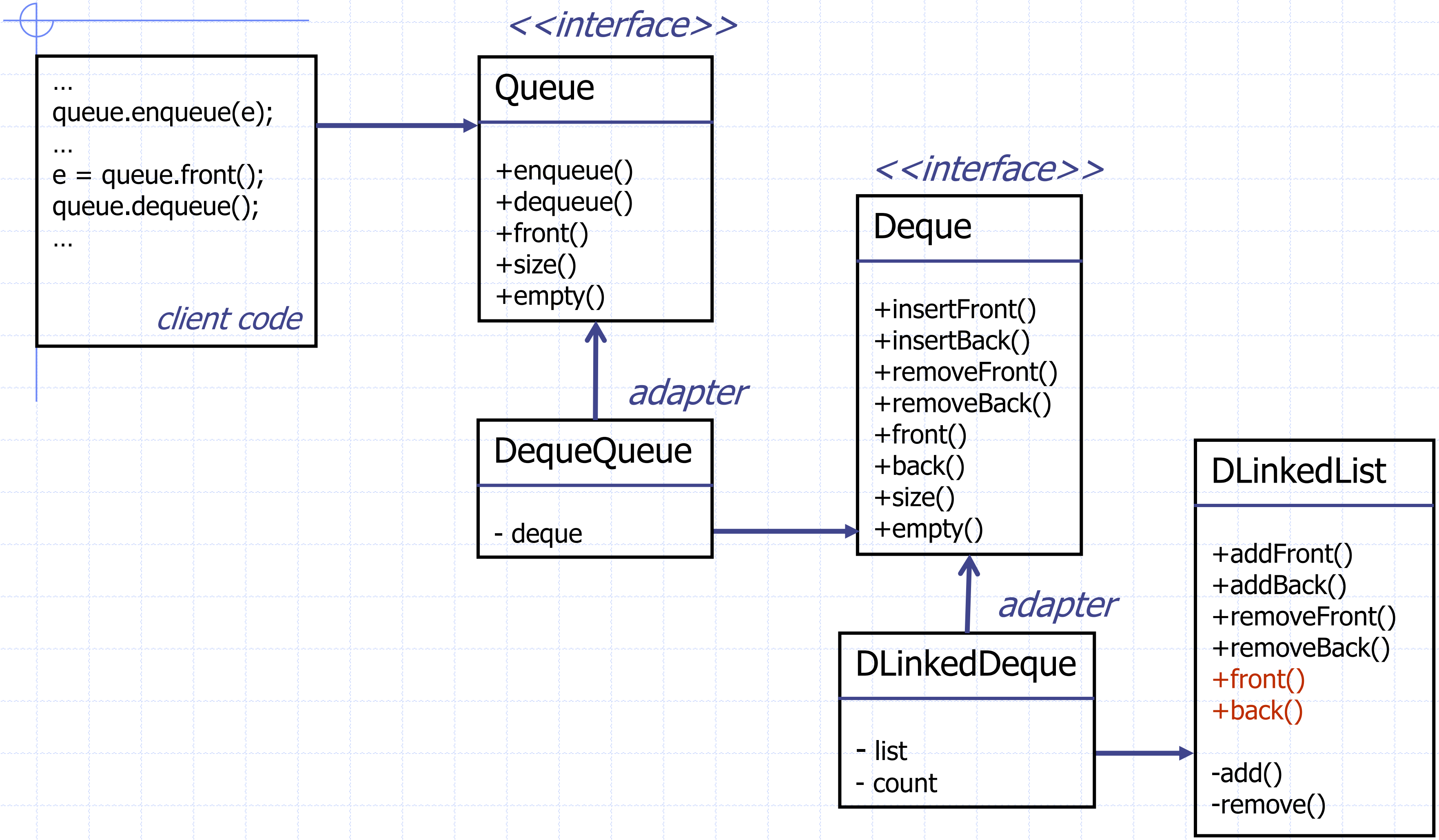
Implementation of DLinkedDeque

DLinkedDeque.cpp

```
DLinkedDeque::DLinkedDeque() :
    list(),
    count(0) { }

int DLinkedDeque::size() const {
    return count;
}

bool DLinkedDeque::empty() const {
    return count == 0;
}

const Elem& DLinkedDeque::front() const {
    if(empty()) throw DequeEmpty("front…");
    return list.front();
}       // back() is similar
```

```
void DLinkedDeque::insertFront(
            const Elem& e) {
    list.addFront(e);
    count++;
}    // insertBack is similar

void DLinkedDeque::removeFront()
            throw(DequeEmpty) {
    if(empty()) throw DequeEmpty("remove…");
    list.removeFront();
    count--;
}    // removeBack is similar
```

A Chain of Two Adapters



```
…
queue.enqueue(e);
…
e = queue.front();
queue.dequeue();
…
```

client code

<<interface>>

Queue

+enqueue()
+dequeue()
+front()
+size()
+empty()

adapter

DequeQueue

- deque

<<interface>>

Deque

+insertFront()
+insertBack()
+removeFront()
+removeBack()
+front()
+back()
+size()
+empty()

adapter

DLinkedDeque

- list
- count

DLinkedList

+addFront()
+addBack()
+removeFront()
+removeBack()
+front()
+back()

-add()
-remove()