void LinkedDeque::insertFront(const Elem& e) { D.addFront(e);

n++;

} // insert new last element

void LinkedDeque::insertBack(const Elem& e) { D.addBack(e);

n++;

} // remove first element

void LinkedDeque::removeFront() throw(DequeEmpty) { if (empty())

throw DequeEmpty("removeFront of empty deque");

D.removeFront();

n−−;

} // remove last element

void LinkedDeque::removeBack() throw(DequeEmpty) { if (empty())

throw DequeEmpty("removeBack of empty deque");

D.removeBack();

n−−;

}

**Code Fragment 5.22:** The insertion and removal functions for LinkedDeque.

Table 5.2 shows the running times of functions in a realization of a deque by a

doubly linked list. Note that every function of the deque ADT runs in *O*(1) time.

***Operation Time***

size *O*(1)

empty *O*(1)

front, back *O*(1)

insertFront, insertBack *O*(1)

eraseFront, eraseBack *O*(1)

**Table 5.2:** Performance of a deque realized by a doubly linked list. The space usage

is *O*(*n*), where *n* is number of elements in the deque.

5.3.4 Adapters and the Adapter Design Pattern

An inspection of code fragments of Sections 5.1.5, 5.2.5, and 5.3.3, reveals a common

pattern. In each case, we have taken an existing data structure and ***adapted*** it

**5.3. Double-Ended Queues 221**

to be used for a special purpose. For example, in Section 5.3.3, we showed how

the DLinkedList class of Section 3.3.3 could be adapted to implement a deque. Except

for the additional feature of keeping track of the number of elements, we have

simply mapped each deque operation (such as insertFront) to the corresponding

operation of DLinkedList (such as the addFront).

An ***adapter*** (also called a ***wrapper***) is a data structure, for example, a class in

C++, that translates one interface to another. You can think of an adapter as the

software analogue to electric power plug adapters, which are often needed when

you want to plug your electric appliances into electric wall sockets in different

countries.

As an example of adaptation, observe that it is possible to implement the stack

ADT by means of a deque data structure. That is, we can translate each stack

operation to a functionally equivalent deque operation. Such amapping is presented

in Table 5.3.

***Stack Method Deque Implementation***

size() size()

empty() empty()

top() front()

push(*o*) insertFront(*o*)

pop() eraseFront()

**Table 5.3:** Implementing a stack with a deque.

Note that, because of the deque’s symmetry, performing insertions and removals

from the rear of the deque would have been equally efficient.

Likewise, we can develop the correspondences for the queue ADT, as shown in

Table 5.4.

***Queue Method Deque Implementation***

size() size()

empty() empty()

front() front()

enqueue(*e*) insertBack(*e*)

dequeue() eraseFront()

**Table 5.4:** Implementing a queue with a deque.

As a more concrete example of the adapter design pattern, consider the code

fragment shown in Code Fragment 5.23. In this code fragment, we present a class

DequeStack, which implements the stack ADT. It’s implementation is based on

translating each stack operation to the corresponding operation on a LinkedDeque,

which was introduced in Section 5.3.3.

**Chapter 5. Stacks, Queues, and Deques**

typedef string Elem; // element type

class DequeStack { // stack as a deque

public:

DequeStack(); // constructor

int size() const; // number of elements

bool empty() const; // is the stack empty?

const Elem& top() const throw(StackEmpty); // the top element

void push(const Elem& e); // push element onto stack

void pop() throw(StackEmpty); // pop the stack

private:

LinkedDeque D; // deque of elements

};

**Code Fragment 5.23:** Implementation of the Stack interface by means of a deque.

The implementations of the various member functions are presented in Code

Fragment 5.24. In each case, we translate some stack operation into the corresponding

deque operation.

DequeStack::DequeStack() // constructor

: D() { } // number of elements

int DequeStack::size() const

{ return D.size(); } // is the stack empty?

bool DequeStack::empty() const

{ return D.empty(); } // the top element

const Elem& DequeStack::top() const throw(StackEmpty) { if (empty())

throw StackEmpty("top of empty stack");

return D.front();

} // push element onto stack

void DequeStack::push(const Elem& e)

{ D.insertFront(e); } // pop the stack

void DequeStack::pop() throw(StackEmpty)

{ if (empty())

throw StackEmpty("pop of empty stack");

D.removeFront();

}