

3.4 The Deque ADT

We will look at the deque as our third abstract data type and we will see two different implementations: one using a doubly linked list, the other a circular array.

3.4.1 Description

An abstract doubly ended queue (or *deque*) is an abstract data type that stores objects in an explicitly defined linear order. The insertion and erase operations are significantly restricted:

1. Objects may be *pushed* onto either the front or the back of the deque, and
2. Objects may be *popped* from either the front or the back of the deque.
3. When an object is *popped* from the queue, it erases the current front of the queue.

Visually, we have these operations as shown in Figure 1.

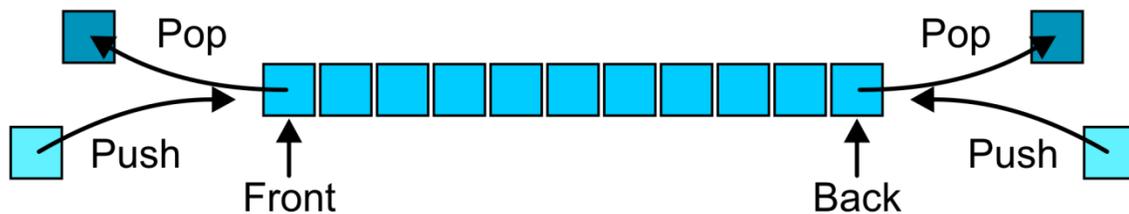


Figure 1. The front, back, push, and pop operations on a deque.

These operations will be named

front	push_front	pop_front
back	push_back	pop_back

3.4.2 Applications

A deque can be used as a queue or a stack. It can also be used in problem solving: Consider finding a path through a maze. As the path is added to, it can be pushed on the front, while if backtracking is necessary, the path can be popped from the front. Finally, when a path is found, the solution can be read from the back of the deque.

3.4.3 Implementation

Like the stack and queue, we will require that all the operations of a deque implementation are $\Theta(1)$.

3.4.3.1 Doubly Linked List Implementation

Only a doubly linked list allows both insertions and erases at both the front and back in $\Theta(1)$ time. We will therefore use a doubly linked list instead of a singly linked list.

```
template <typename Type>
class Deque {
private:
    Double_list<Type> list;

public:
    bool empty() const;
    Type front() const;
    Type back() const;
    void push_front( const Type & );
    void push_back( const Type & );
    Type pop_front();
    Type pop_back();
};
```

The implementation is reasonably straight-forward: like our implementations of stacks and queues using the `Single_list` class, each of the member functions will call the corresponding member functions in the `Double_list` class—as before, this will be a *wrapper* class.

3.4.3.2 Circular Array Implementation

The implementation of a deque using a circular queue is reasonably straight-forward. It is left as an exercise to the you to determine the details.

3.4.4 Deques in the STL

The deque class in the STL has the following definition:

```
template <typename T>
class deque {
public:
    deque();                // not quite true...
    bool empty() const;
    int size() const;
    const T & front() const;
    const T & back() const;
    void push_front( const T & );
    void push_back( const T & );
    void pop_front();
    void pop_back();
};
```

An example of this deque in use is given here:

```
#include <iostream>
#include <deque>

int main() {
    deque<int> ideque;

    ideque.push_front( 5 );
    ideque.push_back( 4 );
    ideque.push_front( 3 );
    ideque.push_back( 6 );           // 3 5 4 6

    std::cout << "Is the deque empty? " << ideque.empty() << std::endl;
    std::cout << "Size of deque: " << ideque.size() << std::endl;

    for ( int i = 0; i < 4; ++i ) {
        std::cout << "Back of the deque: " << ideque.back() << std::endl;
        ideque.pop_back();
    }

    std::cout << "Is the deque empty? " << ideque.empty() << std::endl;

    return 0;
}
```

Given a deque with n entries, there are two methods to access an arbitrary entry from $0 \leq k < n$:

`ideque[k]` and `ideque.at(k)`

The only difference between the two is that the second does range checking. If $k < 0$ or $k \geq n$, the member function `at` will throw an `out_of_range` exception. Consequently, using indexing is faster, but less safe.

The third mechanism for walking through the entries of the deque is to use an *iterator*. To motivate this, consider the design of Project 1:

```
Single_list<int> list;

for ( int i = 0; i < 10; ++i ) { list.push_front( i ); }

for ( Single_node<int> *ptr = list.head(); ptr != 0; ptr = ptr->next() ) {
    cout << ptr->retrieve();
}
```

What happens if the user were to, for example, execute:

```
delete list.head();
```

Now, the internal structure of the linked list is no longer consistent: the memory for the first node is no longer allocated. Thus, the internal state of the linked list is no longer valid. One of the goals of object-oriented programming is to avoid such situations. Therefore, we need a different mechanism for stepping through a linked list.

3.4.5 Iterators

Within the deque class is a nested class with the declaration:

```
template< typename T >
class deque<T>::iterator;
```

Given the instance `ideque` from above, we may now declare

```
deque<int>::iterator itr = ideque.begin();
```

This is equivalent to the Project 1 call:

```
Single_node<int> *ptr = list.head();
```

The operations to move forward and backward, however, are much more elegant:

Project 1	Iterators
<code>ptr->retrieve();</code>	<code>*itr;</code>
<code>ptr = ptr->next();</code>	<code>++itr;</code>
<code>ptr = ptr->previous();</code>	<code>--itr;</code>

while operators appear to be the same as for pointers, they are actually overloaded operators.

Numerous member functions of the deque class will return an iterator:

Member Function

```
deque<T>::iterator begin();  
deque<T>::iterator end();  
deque<T>::iterator rbegin();  
deque<T>::iterator rend();
```

Returns an iterators that points to...

the entry at the front of the deque.
a notional position one past the entry at the back of the deque.
the entry at the back of the deque.
a notional position one prior to the entry at the front of the deque.

Now, one might wonder why `end` and `rend` point to notional positions either past the back or prior to the front of the deque, respectively. This is so that the standard for loops still work as expected:

```
for ( deque<int>::iterator itr = ideque.begin(); itr != ideque.end(); ++itr ) { // ...  
for ( deque<int>::iterator itr = ideque.rbegin(); itr != ideque.rend(); --itr ) { // ...
```

The first will iterate from the first entry to the last, while the second will iterate from the last entry back to the first. As well as accessing, it is also possible (as with pointers) to assign to entries already in the deque:

```
deque<int> ideque;  
ideque.push_front(3);  
deque<int>::iterator itr = ideque.begin();  
*itr = 8;  
std::cout << *itr << std::endl;    // This would print out 8
```

While this is valid, it is invalid to add to a deque by assigning to the end iterator:

```
// Invalid method  
// *ideque.end() = 4;  
// Correct method  
ideque.push_back( 4 );
```

Why use iterators? For numerous reasons:

1. They hide the implementation—the underlying data structure could be an array, a linked list, or something more complex; however, this can be changed and so long as the code for the iterator is adjusted, the behaviour of the iterator will not change,
2. All operations on an iterator are performed in $\Theta(1)$ time, and
3. It provides a common interface across all containers within the STL.

3.4.6 Iterators for the `Single_list` Class (Optional)

The following provides for a forward traversing iterator for the `Single_list` class.

3.4.6.1 Changes to the `Single_list` Class Definition

The member functions `head` and `tail` would be made private:

```
private:
    Single_node<Type> *head() const;
    Single_node<Type> *tail() const;
```

The iterator class would be public and we would introduce three new member functions: `begin`, `end` and `find`.

```
public:
    class iterator {
    private:
        iterator( Single_node<Type> * );
        Single_node<Type> *current;
    public:
        Type &operator*();
        iterator operator++();
        iterator operator++( int );
        bool operator==( iterator const & ) const;
        bool operator!=( iterator const & ) const;

        friend class Single_list;
    };

    iterator begin() const;
    iterator end() const;
    iterator find( Type const & ) const;
```

The implementation of the three member functions is straight-forward:

```
template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::begin() const {
    return iterator( head() );
}

template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::end() const {
    return iterator( 0 );
}

template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::find( const Type &obj ) const {
    for ( Single_node<Type> *ptr = list_head; ptr != 0; ptr = ptr->next() ) {
        if ( ptr->retrieve() == obj ) {
            return iterator( ptr );
        }
    }

    return iterator( 0 );
}
```

The implementation member functions of the iterator class simply update or access the `Single_node` member variable.

```
// Create a new iterator pointing to the Single_node passed as an argument
template <typename Type>
Single_list<Type>::iterator::iterator( Single_node<Type> *ptr ) :
current( ptr ) {
    // empty constructor
}

// Return the element stored in the current node
// It is returned by reference and can therefore be changed
template <typename Type>
Type &Single_list<Type>::iterator::operator*() {
    return current->element;
}

// If we are not already at the end, push the pointer forward one in the linked list
// Return an iterator also pointing to the current object
template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::iterator::operator++() {
    if ( current != 0 ) {
        current = current->next();
    }

    return iterator( current );
}

// If we are not already at the end, push the pointer forward one in the linked list
// Return an iterator pointing to the original node in the linked list
template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::iterator::operator++( int ) {
    Single_node<Type> *previous = current;

    if ( current != 0 ) {
        current = current->next();
    }

    return iterator( previous );
}

// Check if two iterators are equal by comparing the values of the pointers
template <typename Type>
bool Single_list<Type>::iterator::operator==( Single_list<Type>::iterator const &itr ) const {
    return current == itr.current;
}

// Check if two iterators are different by comparing the values of the pointers
template <typename Type>
bool Single_list<Type>::iterator::operator!=( Single_list<Type>::iterator const &itr ) const {
    return current != itr.current;
}
```