ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 3.3.
Assistances and comments will be acknowledged.

3.3 The Queue ADT

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

3.3.1 Description

An abstract queue is an abstract data type that stores objects in an explicitly defined linear order. The insertion and erase operations are significantly restricted:

1. Objects are always *pushed* onto the back of the queue,
2. The *front* of the queue is the object that was least recently pushed onto the queue, and
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.

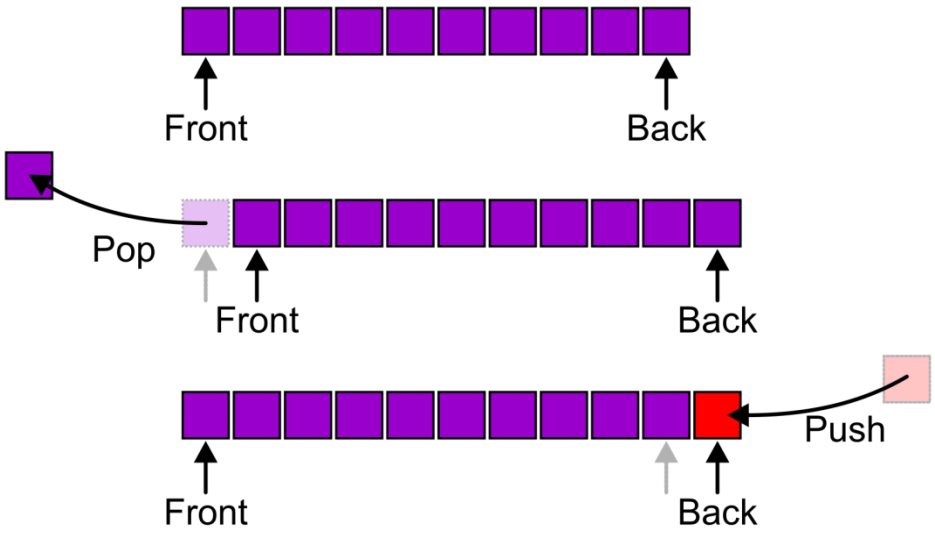

Figure 1. The front, push, and pop operations on a queue.

Common alternative terms for the various operations and parts of a queue include *enqueue* at the *tail* (for *push* at the *back*), *dequeue* from the *head* (for *pop* from the *front*). These alternative terms are shown in Figure 2.

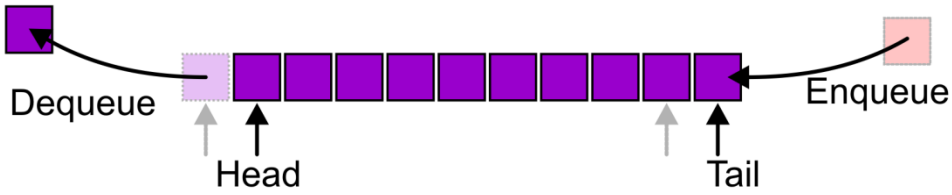

Figure 2. Alternative names for the various operations on a queue.

It is an undefined operation to call front or pop when the queue is empty.

This is also called a *first-in—first-out* (FIFO) behaviour.

3.3.2 Applications

Like a stack, this is an extremely simple ADT, but his many applications in any system that uses the *client-server model*. That is, there is a central server that services requests made from various clients. When clients make their request, the requests are pushed into a queue and the requests are serviced in the order in which they are made.

1. Grocery stores, banks, and airport security,
2. The SSH Secure Shell and SFTP clients,
3. Web servers, databases, mail servers, printers, WOW, *etc*.

For example, using SFTP to copy your code to Unix will show multiple files being queued while a progress bar is shown for the current file being transferred , as is shown in Figure 3.

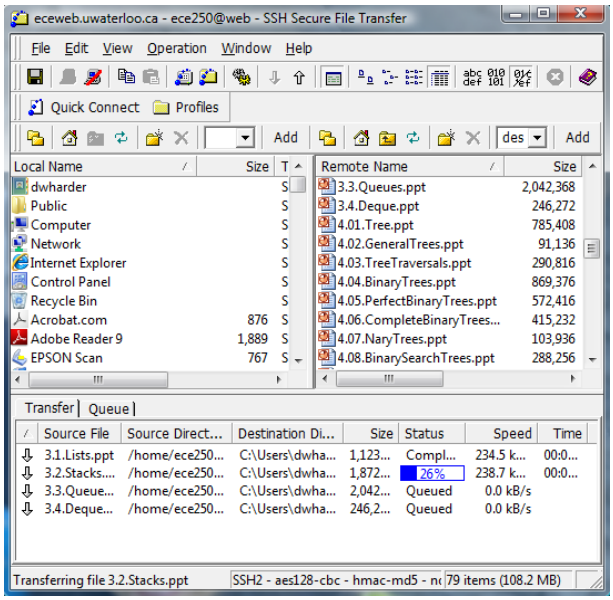

Figure 3. The SFTP client showing the queuing of files waiting for transfer.

3.3.3 Implementation

Like the stack, we will require that all the operations of a queue implementation are Θ(1).

3.3.3.1 Singly Linked List Implementation

A singly linked list both Θ(1) insertions at both the front and the back, but a Θ(1) erase is only possible at the front. In order to satisfy the behaviour of queue, insertions will be at the back while erases are from the front.

```
template <typename Type>
class Queue {
    private:
        Single_list<Type> list;

    public:
        bool empty() const;
        Type front() const;
        void push( const Type & );
        Type pop();
};

template <typename Type>
bool Queue<Type>::empty() const {
    return list.empty();
}


template <typename Type>
void Queue<Type>::push( const Type &obj ) {
    list.push_back( obj );
}

template <typename Type>
Type Queue<Type>::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return list.front();
}

template <typename Type>
Type Queue<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }
    return list.pop_front();
}
```

Again, because the `Queue` class essentially just calls member functions from the `Single_list` class, we call the `Queue` class a *wrapper* class.

3.3.3.2 Two-ended/Circular Array Implementation

A one-ended array only allows Θ(1) insertions and erases at the back and thus we will restrict our operations to that part of the array, and therefore we cannot simulate the behaviour of a queue. Instead, a two-ended queue will allow insertions and erases at both the front and back in Θ(1).

```
#include <algorithm>

template <typename Type>
template <typename Type>
class Queue{
    private:
        int queue_size;
        int queue_front;
        int queue_back;
        int array_capacity;
        Type *array;

    public:
        Queue( int = 10 );
        ~Queue();

        bool empty() const;
        Type front() const;

        void push( const Type &  );
        Type pop();
};

template <typename Type>
Queue<Type>::Queue( int n ):
queue_size( 0 ),
queue_back( -1 ),
queue_front( 0 ),
array_capacity( std::max(1, n) ),
array( new Type[array_capacity] ) {
    // Empty constructor
}

template <typename Type>
Queue<Type>::~Queue() {
    delete [] array;
}
```

```
template <typename Type>
bool Queue<Type>::empty() const {
    return ( queue_size == 0 );
}

template <typename Type>
Type Queue<Type>::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return array[queue_front];
}

template <typename Type>
void Queue<Type>::push( const Type &obj ) {
    if ( queue_size == array_capacity ) {
        throw overflow();
    }

    ++queue_back;
    array[queue_back] = obj;
    ++queue_size;
}

template <typename Type>
Type Queue<Type>::pop() {
    if ( empty() ) {
        throw underflow();
    }

    --queue_size;
    ++queue_front;
    return array[queue_front - 1];
}
```

One major issue that is **not** implemented in this queue is that it will fill up. Suppose we begin with a capacity of 16, perform 16 pushes, and then follow up by performing five pops. We would end up with the situation shown in Figure 4.

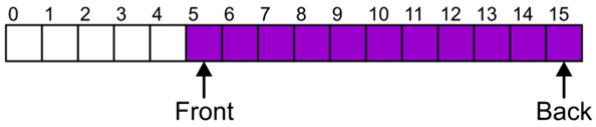

Figure 4. A queue after 16 pushes and five pops.

The capacity is not full, however, we cannot insert any more objects at the back. Instead, consider the entries of the array to be cyclic:

…, 14, 15, 0, 1, 2, …, 14, 15, 0, 1, 2, …, 14, 15, 0, 1, 2, …

Thus, location 0 follows to the right of location 15. This is shown in Figure 5.

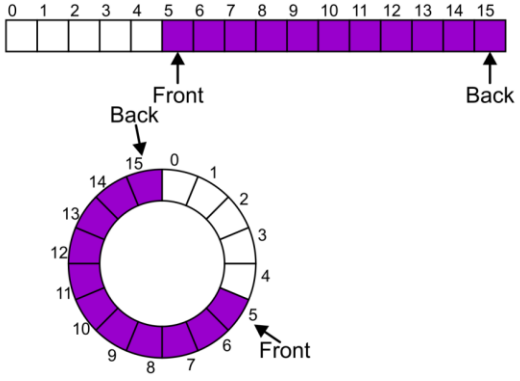

Figure 5. A cyclic array.

Now, the next push may be performed by using the next available location, as is shown in Figure 6.

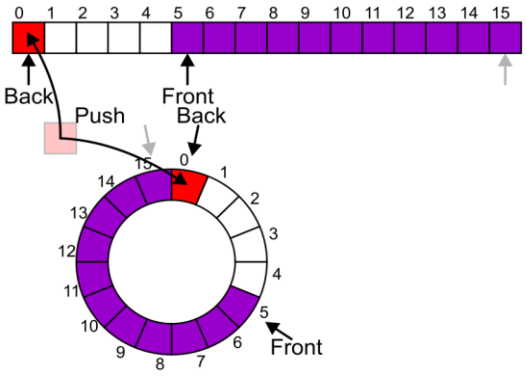

Figure 6. An insertion into a cycle array.

3.3.4 Increasing the Capacity of the Array

Like a Stack ADT, there is no reference in the description of what should happen when the array is full. Often, if the array is of fixed size, new insertions are discarded if the array is full. Alternatively, we can increase the capacity of the array; however, this now has its own issues. Because the queue could end anywhere, it is necessary to be careful about doubling the size of the queue. Just doubling the size will not create a valid queue, as is shown in Figure 7.

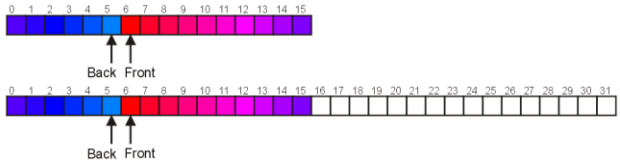

Figure 7. A blind copy of a full array into one with twice the capacity.

Instead, the copies must be made more intelligently, as is shown in Figure 8 or Figure 9.

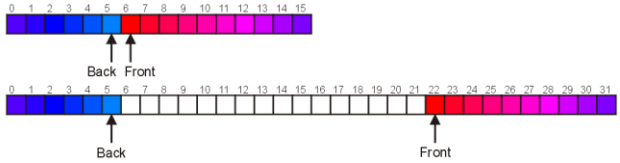

Figure 8. Copying from the front to the end of the array to the end of the new array.

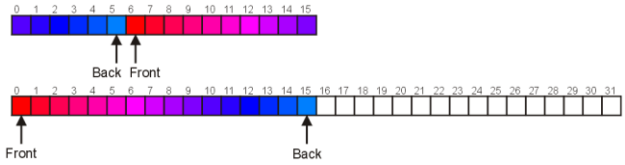

Figure 9. Copying the entries so that the front is at location 0.

3.3.5 Applications

Beyond the obvious client-server model, another use for a queue is in performing a breadth-first traversal of a tree. Suppose you are searching for a file—in all likelihood, in a very complex directory structure, it would be better to search the contents of all sub-directories first before one proceeds to search sub-sub-directories. One such search that satisfies this requirement is a *breadth-first traversal* of the directories. A breadth-first traversal follows a path as is shown in Figure 10.

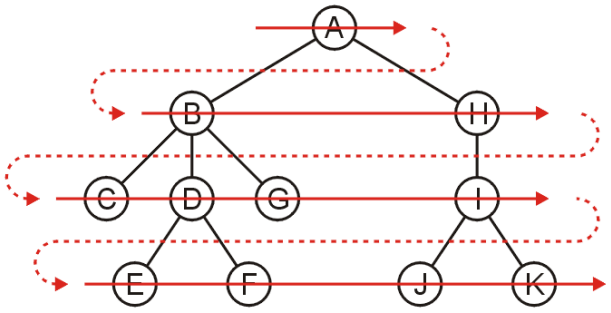

Figure 10. A breadth-first traversal.

To implement such a traversal, consider the following algorithm:

1. Create an empty queue and push the root directory onto the queue, then
2. While the queue is not empty:
    a. Pop to the front directory off of the queue, and
    b. Push all of its sub-directories onto the queue.

Using a circular queue of size seven, we would proceed as follows:

Push A onto the queue.

A						

Pop A from the queue and push its sub-directories, B and H onto the queue:

	B	H				

Pop B off the queue and push its sub-directories, C, D, and G onto the queue:

		H	C	D	G	

Pop H off the queue and push its sub-directory onto the queue:

			C	D	G	I

Pop C off the queue and as it has no sub-directories, we do nothing:

				D	G	I

Pop D off of the queue and push its sub-directories, E and F, onto the queue (recall, it is a circular queue, so we start filling the queue up again at the front):

E	F				G	I

Pop G off the queue and do nothing, it has no sub-directories:

E	F					I

Pop I off the queue and push its two sub-directories, J and K, onto the queue.

E	F	J	K		

At this point, none of the remaining directories in the queue have sub-directories, and therefore we would simply pop them off in the order E, F, J, and K.

Looking back, the order in which the directories were popped off the queue was:

A, B, H, C, D, G, I, E, F, J, K.

This is the order of the breadth-first traversal described above.

3.3.6 Queues in the STL

The queue class in the STL has the following definition:

```
template <typename T>
class queue {
    public:
        queue();                  // not quite true...
        bool empty() const;
        int size() const;
        const T & front() const;
        void push( const T & );
        void pop();
};
```

An example of this queue in use is given here:

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> iqueue;
    iqueue.push( 13 );
    iqueue.push( 42 );

    std::cout << "Top: " << iqueue.front() << std::endl;
    iqueue.pop();                             // no return value

    std::cout << "Top: "  << iqueue.front() << std::endl;
    std::cout << "Size: " << iqueue.size()  << std::endl;

    return 0;
}
```

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.

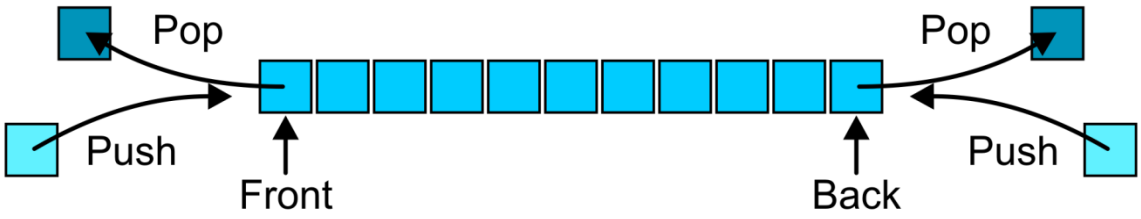

Figure 1. The front, back, push, and pop operations on a deque.

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
`ptr->retrieve();`	`*itr;`
`ptr = ptr->next();`	`++itr;`
`ptr = ptr->previous();`	`--itr;`

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	Returns an iterators that points to…
`deque<T>::iterator begin();`	the entry at the front of the deque.
`deque<T>::iterator end();`	a notional position one past the entry at the back of the deque.
`deque<T>::iterator rbegin();`	the entry at the back of the deque.
`deque<T>::iterator rend();`	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;
}
```