

### 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.

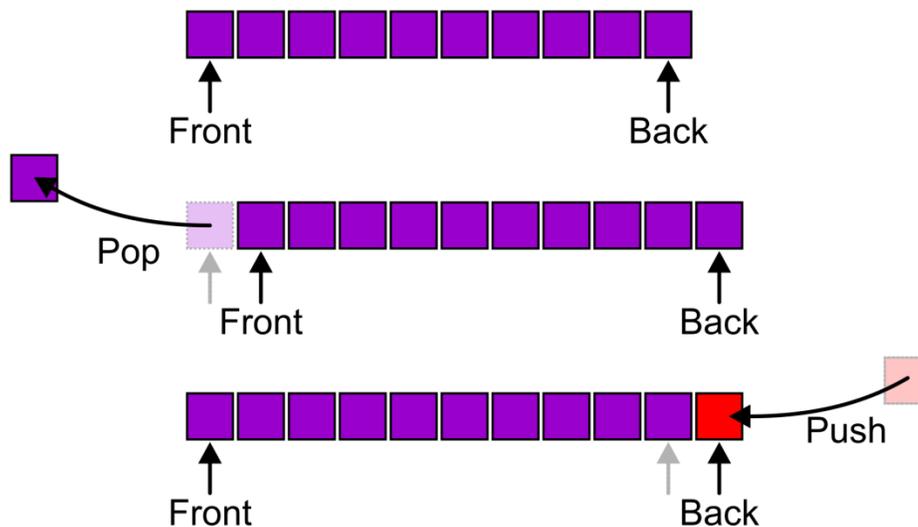


Figure 1. The front, push, and pop operations on a queue.

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.

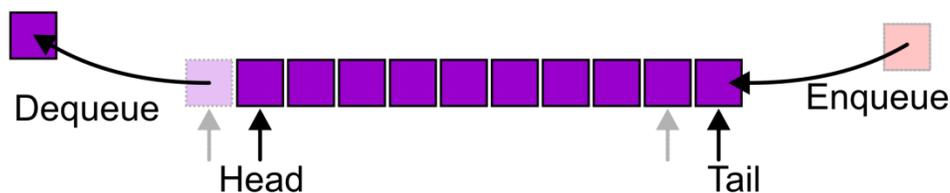


Figure 2. Alternative names for the various operations on a queue.

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 has 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.

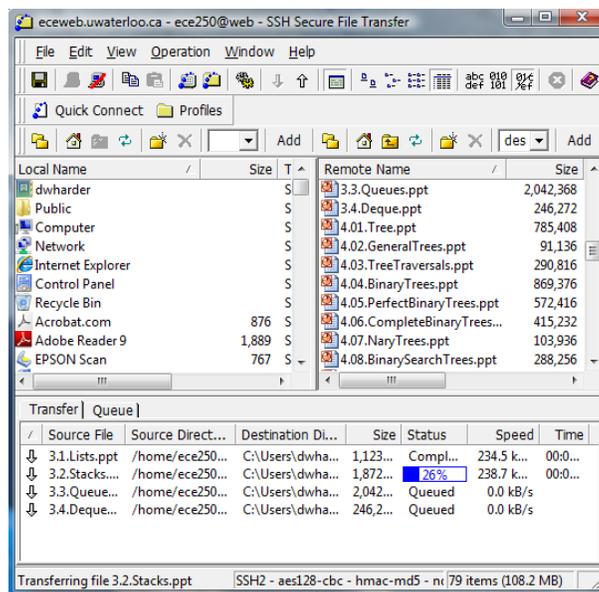


Figure 3. The SFTP client showing the queuing of files waiting for transfer.

### 3.3.3 Implementation

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

#### 3.3.3.1 Singly Linked List Implementation

A singly linked list both  $\Theta(1)$  insertions at both the front and the back, but a  $\Theta(1)$  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  $\Theta(1)$  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  $\Theta(1)$ .

```
#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.

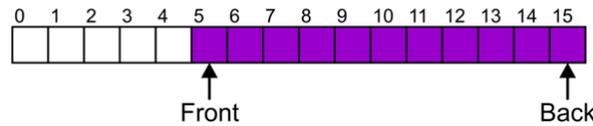


Figure 4. A queue after 16 pushes and five pops.

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.

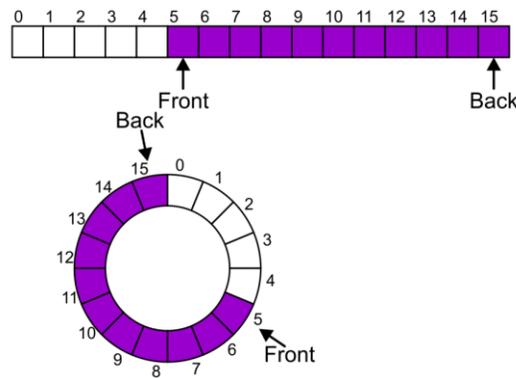


Figure 5. A cyclic array.

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

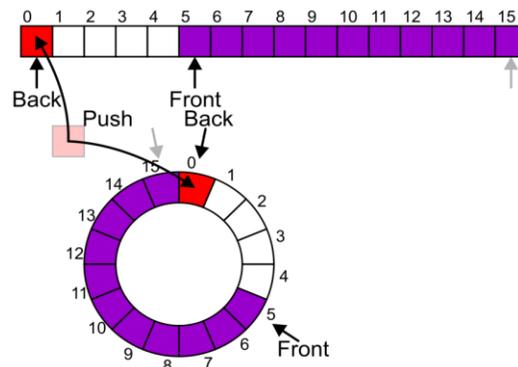


Figure 6. An insertion into a cycle array.

### 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.

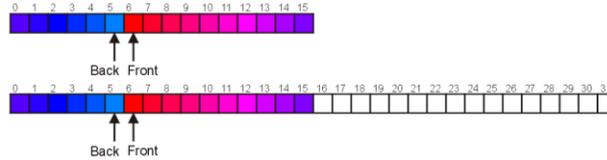


Figure 7. A blind copy of a full array into one with twice the capacity.

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

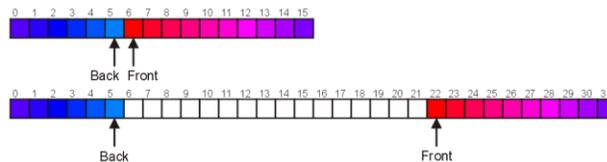


Figure 8. Copying from the front to the end of the array to the end of the new array.

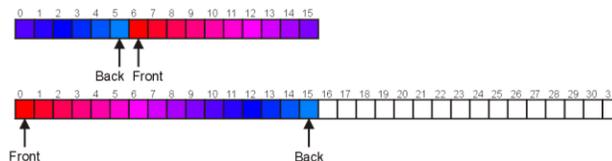


Figure 9. Copying the entries so that the front is at location 0.

### 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.

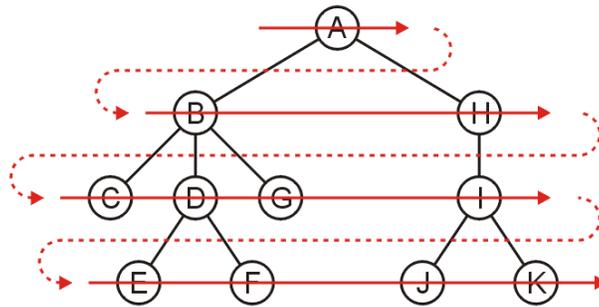


Figure 10. A breadth-first traversal.

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.



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



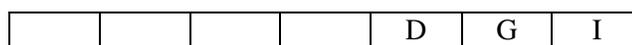
Pop B off the queue and push its sub-directories, C, D, and G onto the queue:



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



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



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;
}
```