CS101 Algorithms and Data Structures

Queue

Textbook Ch 10.1

Outline

- Queue ADT
- Implementation
- Deque

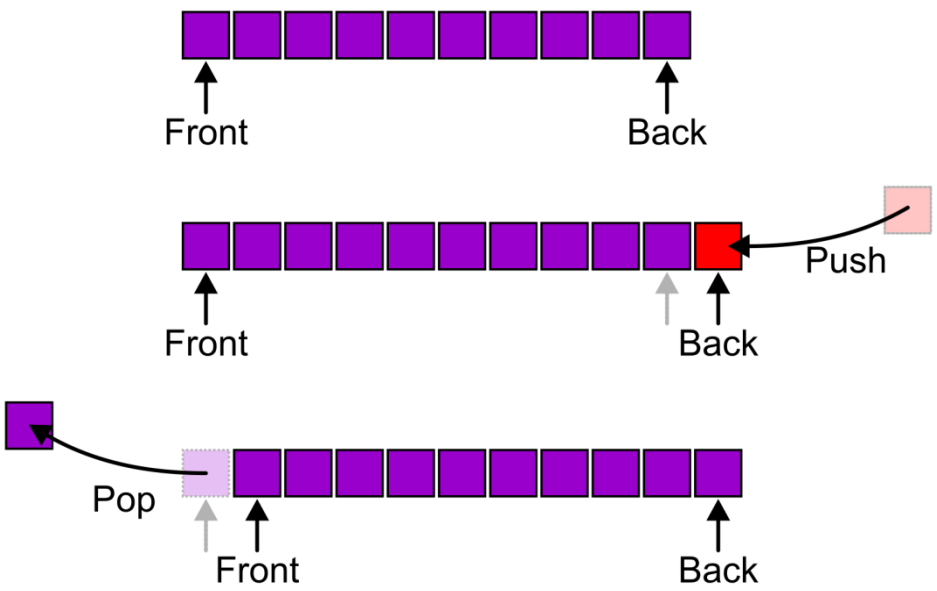
Queue ADT

- Uses an explicit linear ordering
- Two principal operations
  - *Push*: insert an object at the back of the queue
  - *Pop*: remove the object from the front of the queue

Queue ADT

Also called a *first-in–first-out* (FIFO) data structure

– Graphically, we may view these operations as follows:



Applications

Grocery stores, banks, airport security…

Applications

Tree traversals, graph traversals

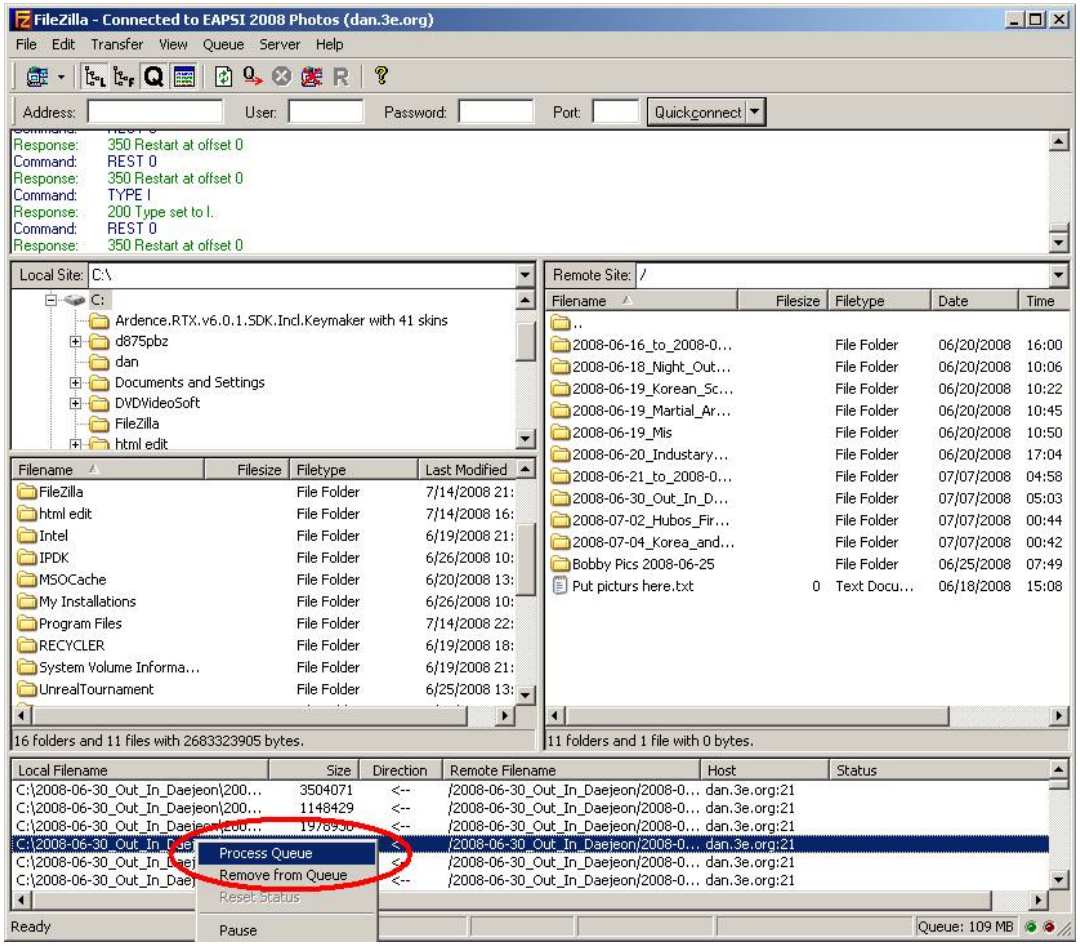
- Will see in coming lectures

The most common application is in client-server models (web, file, ftp, database, mail, printers, etc.)

- Multiple clients may be requesting services from one or more servers
- Some clients may have to wait while the servers are busy
- Those clients are placed in a queue and serviced in the order of arrival

Applications

Example:
When downloading files from a web server, the requests not currently being downloaded are marked as “Queued”



Outline

- Queue ADT
- Implementation
- Deque

Implementations

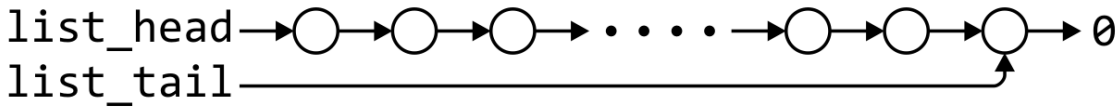
We will look at two implementations of queues:

- Singly linked lists
- Circular arrays

All queue operations run in $\Theta(1)$ time

Linked-List Implementation

List head/tail → Queue front/back?



	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

Removal is only possible at the front with $\Theta(1)$ run time

The desired behavior of an Abstract Queue may be produced by performing insertions at the back and removal at the front

`Single_list` Definition

The definition of single list class:

```
template <typename Type>
class Single_list {
        public:
                int size() const;
                bool empty() const;
                Type front() const;
                Type back() const;
                Single_node<Type> *head() const;
                Single_node<Type> *tail() const;
                int count( Type const & ) const;

                void push_front( Type const & );
                void push_back( Type const & );
                Type pop_front();
                int erase( Type const & );
};
```

Queue-as-List Class

The queue class using a singly linked list has a single private member variable: a singly linked list

```
template <typename Type>
class Queue{
    private:
        Single_list<Type> list;
    public:
        bool empty() const;
        Type front() const;
        void push( Type const & );
        Type pop();
};
```

Queue-as-List Class

The implementation is similar to that of a Stack-as-List

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

```
template <typename Type>
void Queue<Type>::push( Type const &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();
}
```

Array Implementation

A one-ended array does not allow all operations to occur in $\Theta(1)$ time

	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Remove	$\Theta(n)$	$\Theta(1)$

Array Implementation

Using a two-ended array, $\Theta(1)$ are possible by pushing at the back and popping from the front

	Front/1^{st}	Back/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Remove	$\Theta(1)$	$\Theta(1)$

Array Implementation

We need to store an array:

```
Type *array;
```

We need additional information, including:

- The number of objects currently in the queue and the front and back indices

```
int queue_size;
int ifront;       // index of the front entry
int iback;        // index of the back entry
```

- The capacity of the array

```
int array_capacity;
```

Queue-as-Array Class

The class definition is similar to that of the Stack:

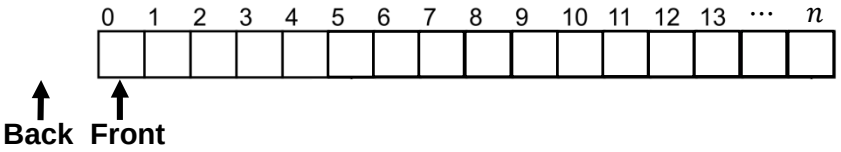
```
template <typename Type>
class Queue{
    private:
        int queue_size;
        int ifront;
        int iback;
        int array_capacity;
        Type *array;
    public:
        Queue( int = 10 );
        ~Queue();
        bool empty() const;
        Type front() const;
        void push( Type const & );
        Type pop();
};
```

Constructor

We must initialize the values

- Allocate memory for the array
- Initialize the member variables
- iback is initialized to -1

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



Member Functions

```
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[ifront];
}
```

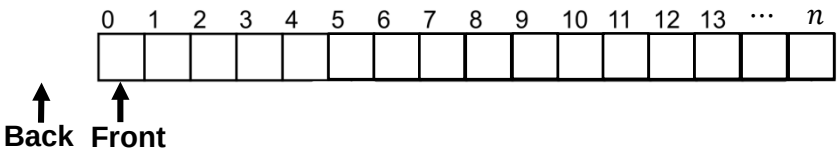
Member Functions

A naïve implementation of push and pop:

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

    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

queue_size=0

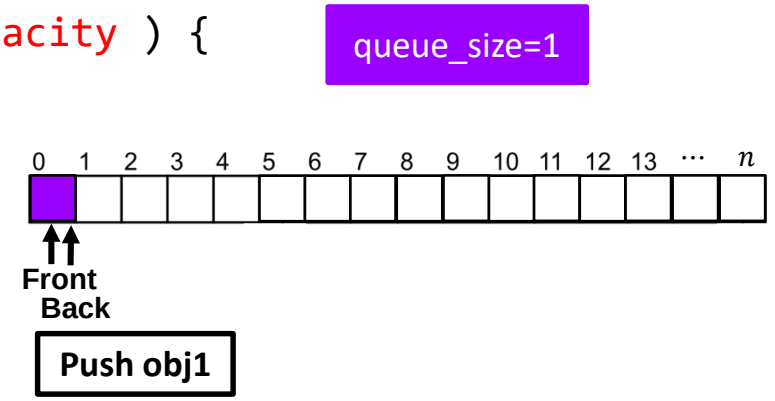


Member Functions

A naïve implementation of push and pop:

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

    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

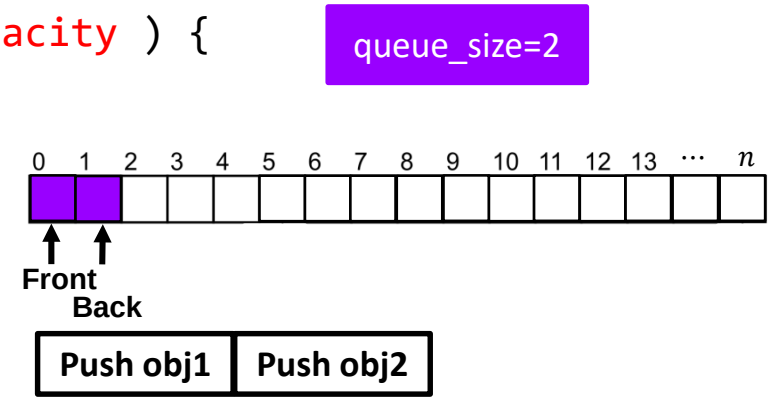


Member Functions

A naïve implementation of push and pop:

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

    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

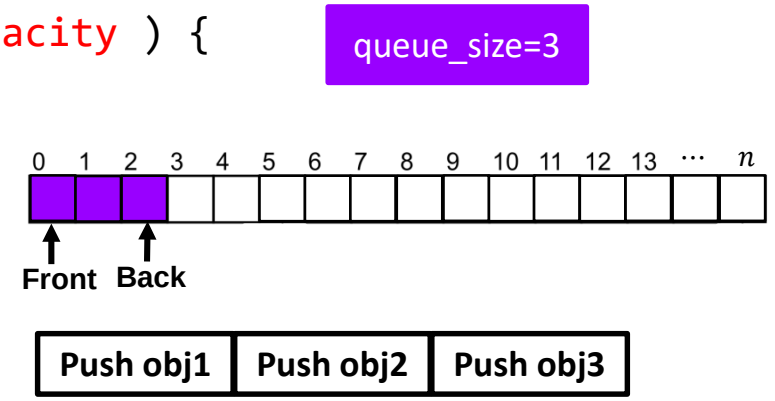


Member Functions

A naïve implementation of push and pop:

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

    ++iback;
    array[iback] = obj;
    ++queue_size;
}
```

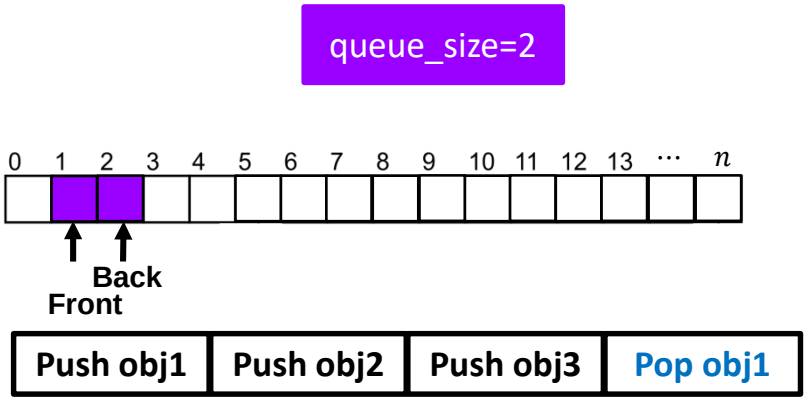


Member Functions

A naïve implementation of push and pop:

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

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

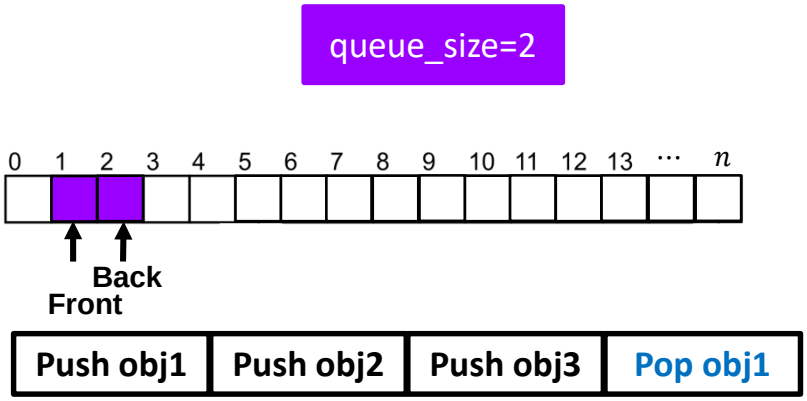


Member Functions

A naïve implementation of push and pop:

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

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



Problem?

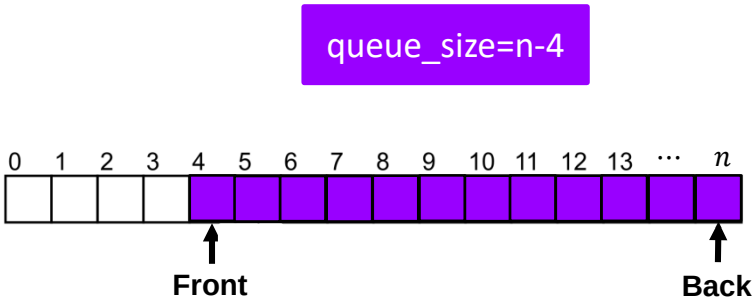
Member Functions

A naïve implementation of push and pop:

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

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

Problem?

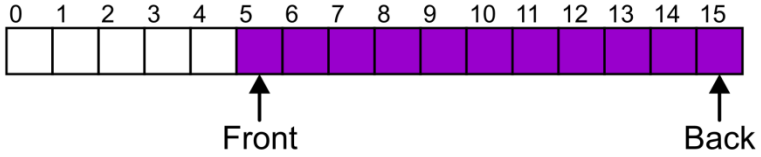


Push obj1	Push obj2	Push obj3	Pop obj1
Push obj4	Push obj5	Push obj6	Push obj7
Pop obj2	Pop obj3	Pop obj4	Push obj8
Push obj9	⋯	Push obj (n-1)	Push obj n

Member Functions

Suppose that:

- The array capacity is 16
- We have performed 16 pushes
- We have performed 5 pops
  - The queue size is now 11



- We perform one further push

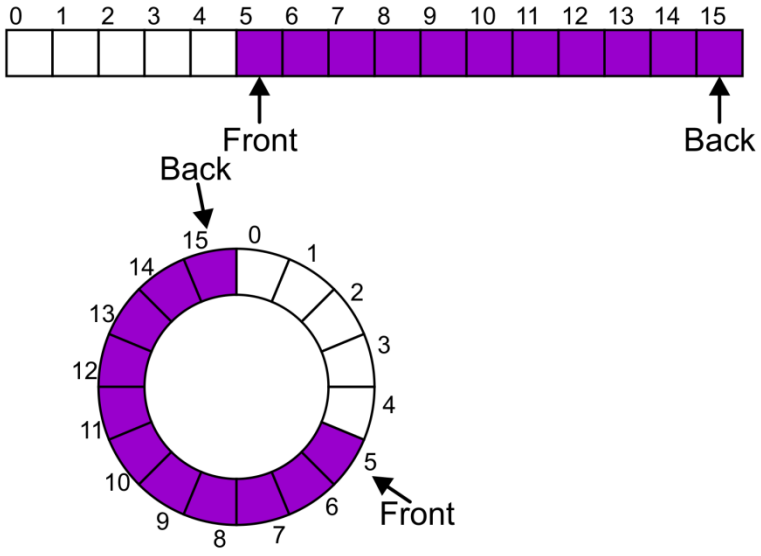
In this case, the array is not full and yet we cannot place any more objects in to the array

Member Functions

Instead of viewing the array on the range $0, \ldots, 15$, consider the indices being cyclic:

$$\ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots$$

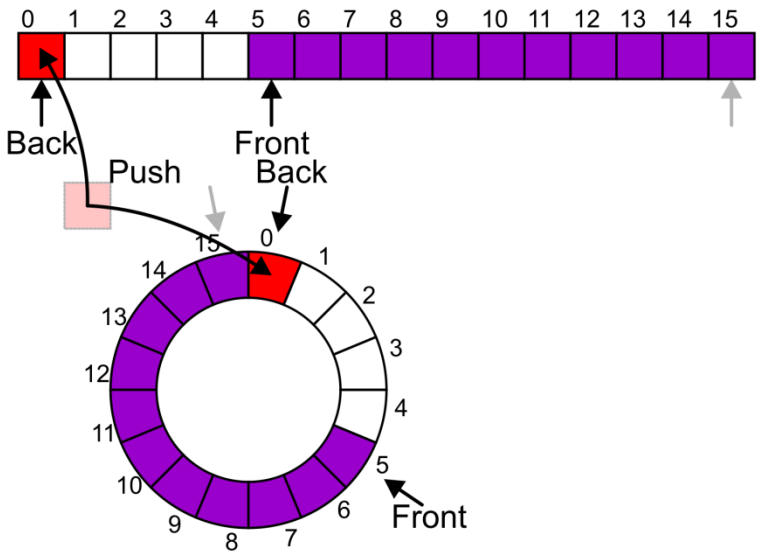
This is referred to as a *circular array*



Member Functions

Now, the next push may be performed in the next available location of the circular array:

```
++iback;
if ( iback == capacity() ) {
    iback = 0;
}
```



Exceptions

As with a stack, there are a number of options which can be used if the array is filled

If the array is filled, we have five options:

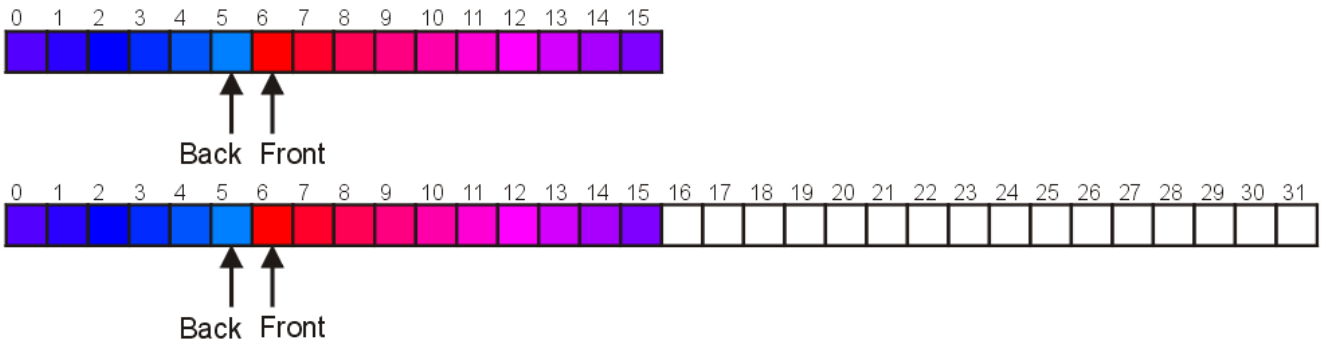
- Increase the size of the array
- Throw an exception
- Ignore the element being pushed
- Put the pushing process to “sleep” until something else pops the front of the queue

Include a member function **`bool full()`**

Increasing Capacity

When the array is full, increasing the capacity is slightly more complex than in the case of stack:

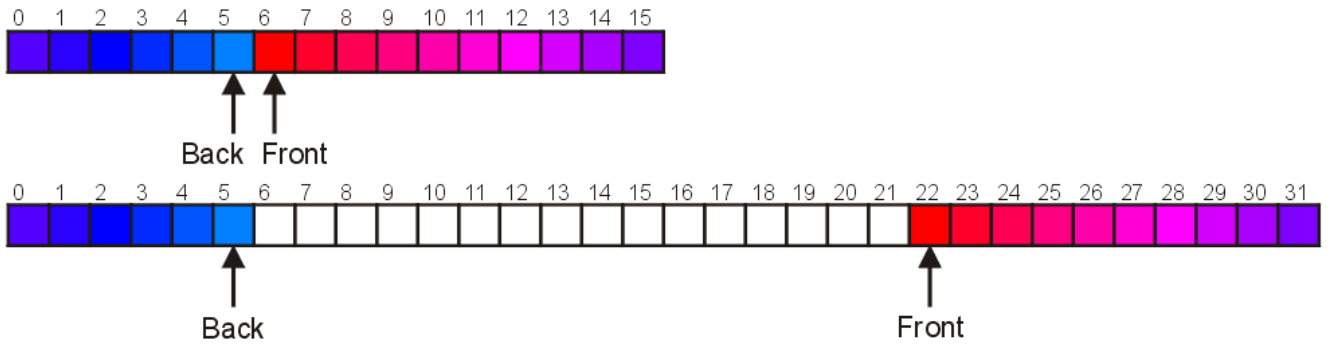
– A direct copy does not work:



Increasing Capacity

One solution:

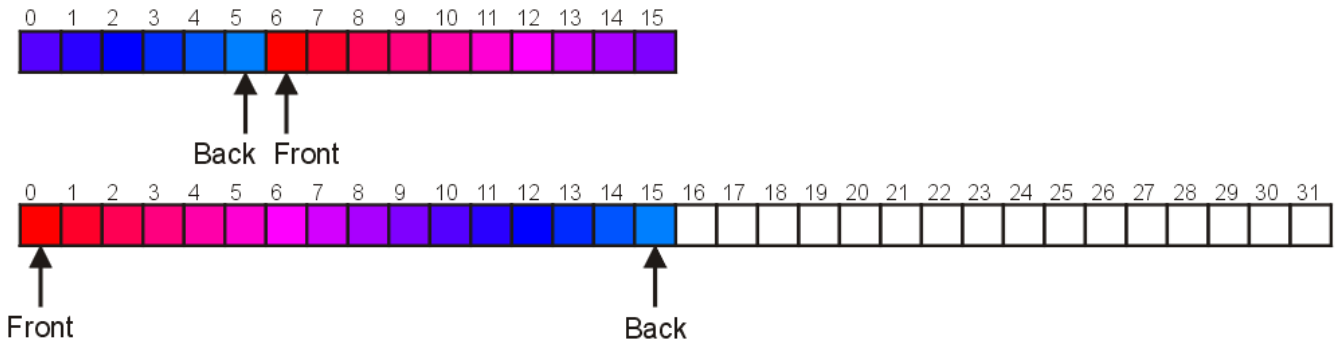
- Move those beyond the front to the end of the array
- The next push would then occur in position 6



Increasing Capacity

An alternate solution is normalization:

- Map the front at position 0
- The next push would then occur in position 16



Destructor

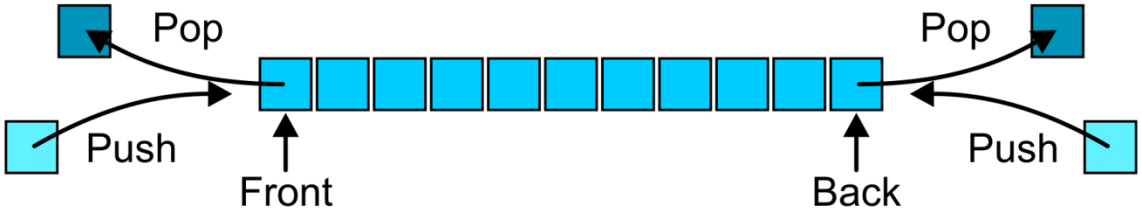
```
template <typename Type>
Queue<Type>::~Queue() {
    delete [] array;
}
```

Outline

- Queue ADT
- Implementation
- Deque

Deque ADT

- Deque = Double-ended queue
  - pronounced like "deck"
- Uses an explicit linear ordering
- Allows insertion/removal at both the front and the back of the deque



Applications

Useful as a general-purpose tool:

- Can be used as either a queue or a stack

Can be used in certain job scheduling algorithms for parallel programming

Implementations

Can we use linked list?

- Pop_back requires $\Theta(n)$

Two efficient implementations:

- Doubly linked list
- Circular array

Summary

- Queue ADT
  - Push, pop, FIFO
- Implementation
  - Singly linked lists
  - Circular arrays
- Deque

Standard Template Library

An example of a queue in the STL is:

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue <int> iqueue;

    iqueue.push( 13 );
    iqueue.push( 42 );
    cout << "Head: " << iqueue.front() << endl;
    iqueue.pop();                          // no return value
    cout << "Head: " << iqueue.front() << endl;
    cout << "Size: " << iqueue.size() << endl;

    return 0;
}
```