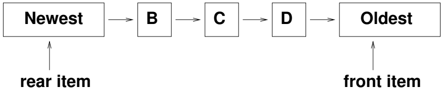
The Queue ADT

A queue is a linear list in which data can only be inserted at one end called the **rear** (or **back**), and deleted from the other end called the **front**. These restrictions ensure that the data are processed through the queue in the order in which they are received. Another name of a queue: FIFO (first-in, first-out). A queue resembles a waiting line.

The basic operations supported by queues are:

Queue ADT (waiting line)



- `enqueue` – insertion of an item at the rear (back) of the queue
- `dequeue` – removal of an item from the front of the queue
- `first` – access of the item at the front of the queue

Informal Queue Interface

The C++ interface corresponding to our stack ADT. It provides to a user useful information about all the public functions of the class `Queue`.

```
class Queue {
public:
   // enqueue object at the end of a queue
   // throwing this exception is not always necessary
   void enqueue(const int elem) throw(QueueFullException);
   // returns first object of  a non-empty queue
   int dequeue() throw(QueueEmptyException);
   // checks whether no elements are stored?
   bool isEmpty() const;
};
```

The operation `dequeu()` cannot be performed if queue is empty. The attempt of removing the first object from an empty queue throws an exception. An exception is also thrown if there is no space to add a new object to a queue.

Circular Array

Assume that `item` is an object to be inserted and a queue is implemented as a *circular array* (using an array of type `int`). That is, the last element of the array is logically followed by the first element of the array (with index 0). This way, a circular array does not have a logical beginning or end. We make use of the operator `%` (modulo operator) to implement it.

Circular Oueue

Note that this implementation: computes correct values for `rear` and `front`: from 0 to `array_length-1` and back to 0, and the array is considered as circular for the array indices `front` and `rear`.

The Circular Queue (cont.)

The insertion and delete operations for queue could be implemented as follows:

```
rear = (rear+1) % array_length; // next position
array[rear] = item;
count++; // increase the number of element
item = array[front]; // get the element
front = (front+1) % array_length; // next position
count--; // decrease the number of elements
```

The Special Conditions

Assume that there can be at most `array_length` entries. The elements can be put in array positions from 0 up to `array_length`−1. We treat the queue as full if we have already `array_length` elements in the array.

Condition	**Special situation**
`back == front`	One-entry queue
`count ==` 0	Empty queue
`count == array_length`	Full queue

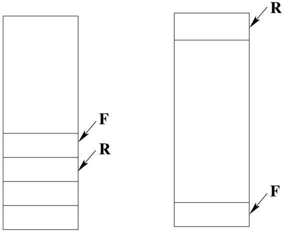
Initially we set `count`, `front` and `rear` as follows:
Initially we set `count`, `front` and `rear` as follows:

```
count = 0;
front = 0;
rear = array_length - 1;
```

Empty and Full Queue

If we *do not* want to use `count`, then the queue is empty when `rear == front-1`. When we include a special case when `rear == array_length-1` and `front == 0` (since the array is circular) the relation for an empty queue in this case can be expressed as

```
(rear-front+1) % array_length == 0
```



Empty Queue

But in this case the queue is full when there are `array_length-1` elements in the queue. If we allow to have `array_length` elements in the queue then the above empty-queue condition is also true for the full array, and we cannot tell whether the queue is empty or full.

Array-Queue Class

```
 class ArrayQueue {
 private: // member data
   int  capacity;      // actual length of stack array
   int* array;         // the stack array (a pointer)
   int front, rear;   //index of the first and last ob-
ject of the queue
   int count;   //index of the first and last object of the queue
public:
   // constructor given max capacity
   ArrayQueue() : capacity(0), array(0), front(0) {}
   ArrayQueue(int cap) : capacity(cap),
       array(new int[cap]), front(0), rear(capacity -1), count(0)
```

The Queue Implementation (cont.)

```
    ArrayQueue(const ArrayQueue& qu); // copy constructor
   // assignment operator
   ArrayQueue& operator=(const ArrayQueue& qu);
   ~ArrayQueue() {  delete [] array; }  // destructor
   bool isEmpty() const { return (count == 0); }// no objects
   // return the first object of the queue
   int first() throw(QueueEmptyException);
   // insert an object onto the queue
   void enqueue(const int elem)  throw(QueueFullException);
   int dequeue() throw(QueueEmptyException);
// remove from the queue
   // number of elements in the queue (auxiliary function)
   int size() const {  return count; }
}; // end of class ArrayQueue
```

The Queue Implementation (cont.)

```
// return the  object of the queue
int ArrayQueue::first() throw(QueueEmptyException)
{  if (isEmpty())
      throw QueueEmptyException("Access to empty queue");
   return array[first];
}
// push object onto the queue
void ArrayQueue::enqueue(const int elem)throw(QueueFullException)
{  if (count == capacity)
      throw QueueFullException("Queue overflow");
   rear = (rear+1)%capacity;
   array[++rear] = elem;
   ++count;
}
```

The Copy Constructor

```
// pop the stack
int ArrayQueue::dequeue() throw(QueueEmptyException)
{
   if (isEmpty())
      throw QueueEmptyException("Access to empty queue");
   --count;
   int item = array[front];
   return array[item];
}
// copy constructor
ArrayQueue::ArrayQueue(const ArrayQueue& qu):
   capacity(qu.capacity),
   array(new int[capacity]),
   front(qu.front);
   rear(qu.rear);
   count(qu.count);
{
   for (int i = 0; i < count; i++) // copy contents
      array[i] = qu.array[i];
}
```

The Implementation of Assignment Operator

```
// assignment operator
ArrayStack& ArrayQueue::
operator=(const ArrayQueue& qu)
{
   if (this != &qu) { // avoid self copy (x = x)
      delete [] array; // delete old contents
      capacity = qu.capacity;
      front = qu.front;
      r rear= qu.rear;
      count = qu.count;
      array = new int[capacity];
      // copy contents
      for (int i = 0; i < count; i++)
         array[i] = qu.array[i];
   }
   return *this;
}
```

A Queue Application

We use the Queue ADT in a simple test. A similar code can be used to process a list of waiting jobs on a computer, where integer numbers are jobs ID numbers.

```
int main() {
   ArrayQueue q; // queue object
   for (int i = 0; i < 10; i++)
      q.enqueue(i+100);
   cout << "Queue contents:" << endl;
   try {
      while(! q.isEmpty())
         cout << q.dequeue()) << " ";
      cout << endl;
   } catch(...) { } // catch any exception
   return 0;
}
```

Exceptions

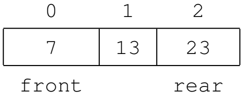
- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "`thrown`" by an operation that cannot be executed
- In the Queue ADT, operations `dequeu` and `first` cannot be performed if the queue is empty
- Attempting `dequeu` or `first` an empty queue throws a `QueueEmpty` exception

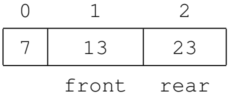
Templated Queue

```
template <typename E> class ArrayQueue
{
   private: E* Q; // array holding the stack
   int cap; // capacity
   int f, r, c; // index of first and last object, number of objec
   public: // constructor given capacity
   ArrayQueue( int c): S(new Q[c]), cap(c), f(0), r(cap -1), c(0)
   void dequeue()
   {
      if (empty()) throw QueueEmpty("Removing from empty stack");
      f++;
   }
   void enqueue(const E& e)
   {
       if (size == cap) throw QueueFull("Adding to full queue");
       S[++r] = e;
       ++c;
   } //... (other methods of Queue interface)
}
```

Example

```
ArrayQueue<int> A;          // A = [ ], count = 0
A.enqueue(7);               // A = [7], count = 1
A.enqueue(13);              // A = [7, 13], count = 2
A.dequeue();                // outputs 7, count = 1
```





Removing an object from a queue it does not mean removing it from a queue-array.

Performance and Limitations

If n is the number of elements in the stack, then

- each stack operation runs in time $O(1)$
- the space used is $O(n)$

However, the size of the stack is fixed at runtime and trying to push a new element into a full stack throws an exception. We can avoid this situation by using an expandable array in the implementation of the stack.

Expandable dynamic arrays were discussed previously. In order to use an expandable dynamic array implementation of the stack we need to add one new function and change `push`.

The Expandable Arrays

- In an insert operation (without an index), we always insert at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
    - Incremental strategy: increase the size by a constant c
    - Doubling strategy: double the size

Doubling Strategy for The Expandable Arrays

```
//a private function double size of an array
void ArrayQueue::doubleSize( )
{  capacity *= 2;
   int newArray[] = new int[capacity];
   for (int i = 0; i <= topOfStack; i++)
      newArray[i] = array[i];
   delete [] array; // remove old contents
   array = newArray; // switch pointers
}
// push an object onto the stack (new implementation)
void ArrayQueue::enquue(const int& elem)
{  if (size() == capacity)
      doubleSize();
   array[++rear] = elem;
   front = 0;
   ++count;
}
```

Radix Sort Algorithm Based on Queue ADT

1. Read in the numbers from a file and create a queue called `Master`.
2. Create 10 other queues called `Subqueue[0]`, `Subqueue[1]`,...,`Subqueue[9]`.
3. Remove a number from the `Master` queue and extract its most right digit (ones).
4. Add the number to the `Subqueue[digit]`
5. Repeat the step 3 and 4 until `Master` queue is empty.
6. Remove the first element from the `Subqueue[0]` and insert it into the `Master` queue.
7. Repeat the previous step until the `Subqueue[0]` is empty.
8. Repeat the last step for consecutive `Subqueue[i]` where `i` takes on values from `1` to `9`.
9. Repeat the procedure starting from (2) `Max` times where the value of `Max` is equal to the number of digits in the largest number of the sequence.

Example (cont.)

Example

An application of the queue data structure to sort numbers using the Radix sort.

$$\text{Master} = \{459, 254, 472, 534, 649, 239, 432, 654, 477\}$$

Subqueue[0] $\rightarrow$

Subqueue[1] $\rightarrow$

Subqueue[2] $\rightarrow$472, 432

Subqueue[3] $\rightarrow$

Subqueue[4] $\rightarrow$ 254, 534, 654

Subqueue[5] $\rightarrow$

Subqueue[6] $\rightarrow$

Subqueue[7] $\rightarrow$ 477

Subqueue[8] $\rightarrow$

Subqueue[9] $\rightarrow$ 459, 649, 239

This first pass of the Radix sort is focusing on the most right digit (ones).We create now the `Master` queue:

Example (cont.)

The second pass of the Radix sort is focusing on the second digit from the right (tens):

Subqueue[0] $\rightarrow$

Subqueue[1] $\rightarrow$

Subqueue[2] $\rightarrow$

Subqueue[3] $\rightarrow$ 432, 534, 239

Subqueue[4] $\rightarrow$ 649

Subqueue[5] $\rightarrow$ 254, 654, 459

Subqueue[6] $\rightarrow$

Subqueue[7] $\rightarrow$ 472, 477

Subqueue[8] $\rightarrow$

Subqueue[9] $\rightarrow$

We create now the `Master` queue:

$$\text{Master} = \{432, 534, 239, 649, 254, 654, 459, 472, 477\}$$

Stack Applications

The third pass of the Radix sort is focusing on the third digit from the right (hundreds)

Subqueue[0] $\rightarrow$

Subqueue[1] $\rightarrow$

Subqueue[2] $\rightarrow$ 239, 254

Subqueue[3] $\rightarrow$

Subqueue[4] $\rightarrow$ 432, 459, 472, 477

Subqueue[5] $\rightarrow$ 534

Subqueue[6] $\rightarrow$ 649, 654

Subqueue[7] $\rightarrow$

Subqueue[8] $\rightarrow$

Subqueue[9] $\rightarrow$

We create now the `Master` queue:

$$\text{Master} = \{239, 254, 432, 459, 472, 477, 534, 649, 654\}$$

The last `Master` queue is sorted.