Queues
(Section 4)

Queues - definition
Types of queues
Programming queues in C
Priority queues and their application

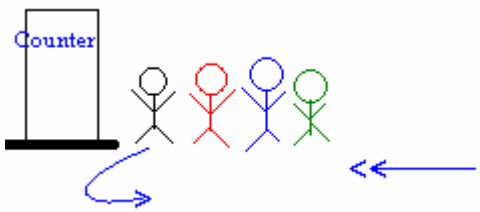
By:

Pramod Parajuli,
Department of Computer Science,
St. Xavier's College,
Nepal.

Queues

Queue is another type of data structure. The primary definition of queue is that; it is an ordered structure (stack like) in which data can be inserted from rear end and removed from front end. The data in the queue must be **continuous**.

One real world scenario is the queue in the ticketing counter where people go into the line from the end and after getting ticket, they leave from the front.



Queue as an ADT

Revision in ADT

Formal definitions

1. In programming, a data set defined by the programmer in terms of the information it can contain and the operations that can be performed with it. The standard example used in illustrating an abstract data type is the stack, a small portion of memory used to store information, generally on a temporary basis. As an abstract data type, the stack is simply a structure onto which values can be pushed (added) and from which they can be popped (removed). **The type of value, such as integer, is irrelevant to the definition.** The way in which the program performs operations on abstract data types is encapsulated, or hidden, from the rest of the program. Encapsulation enables the programmer to change the definition of the data type or its operations without introducing errors to the existing code that uses the abstract data type. **Abstract data types represent an intermediate step between traditional programming and object-oriented programming**.

2. A kind of data abstraction where a type's internal form is hidden behind a set of access functions. Values of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined.

Objects and ADTs are both forms of data abstraction, but objects are not ADTs. Objects use procedural abstraction (methods), not type abstraction. (The identification of objects is based on the load sharing or responsibility sharing.)

A classic example of an ADT is a stack data type for which functions might be provided to create an empty stack, to push values onto a stack and to pop values from a stack.

Data abstraction: Any representation of data in which the implementation details are hidden (abstracted). Abstract data types and objects are the two primary forms of data abstraction.

In general, abstract data type is a type defined for data that contains values and set of operations. The format of values and details of operations are not considered while defining an abstract data type i.e. we do not go for implementation details while defining the abstract data types. Since the definition does not care about the efficiency or time complexity, there is a kind of freedom in defining any kind of ADT.

For example,

If we define a variable ‘counter’ and say it is used to count the number of objects. We do not care the type of the object. In common sense, counters are integers. Mathematically, integer can have

any upper bound and lower bound. The set of operations for counting is either counting up or counting down. Here also, we don't care about how the counting up or counting down, whether the value is increased by 1, or 2, or 5. The 'counter' defined here is an ADT.

What if we considered like, to define integers we use 'int' data type and to count up the value is incremented by 1, and to count down the value is decremented by 1. And if we think that the upper bound for a 16-bit integer is 32767 and lower bound is -32768, then the counter defined here is not an ADT. It is just a data type.

Some of you put up one question that how could we actually write ADT?

You can write in any format to define an ADT. A sentence written like English sentence also works for the definition of an ADT. In some of the books, the writers have defined the ADT using C like syntax. Whatever be the format you used for defining ADT, take care of the following guidelines;

1. Define data holder (liked we defined 'counter' in our example.
2. After defining data holder, define the pre-conditions required for the data holder. For example, for our counting method, we never count down if there are no items.
3. Then, define post-condition, i.e. what will be the situation of the data holder after accomplishing the defined methods.
4. Now, define appropriate methods.

Have a look at;

1. *Fundamentals of Data Structures in C++*, by Horowitz, Sahni, and Mehta. (page - 6).
2. *Data Structures using C and C++*, by Langsam, Augenstein, Tenenbaum (page – 13).
3. *Data Strucutres and Algorithms*, by Aho, Hopcroft, and Ullman (page – 10).

Now, can you think of your own definition of a queue?

From the real representation of a queue, you can see that, we need some space to hold the queue. Every time a new item needs to be **enqueued,** then we just put that item to the rear end and if an item needs to be **dequeued**, then we pull from the front end. So, now, we might want to hold the status of the front and rear end of the queue.

Other two things to remember for the queue operations are;
1. If the queue is empty, then we can not dequeue item from the queue.
2. If the queue is full, then we can not enqueue item in the queue.

```
<ADT definition> Queue

      dataholder[items]       // holds data i.e. lots of items
      counter                 // if counting of the items is necessary

      <process> Process name: <returns none> enqueue
            Pre-condition: the queue must not be full
            Method: push the item at the rear end of dataholder.
            Post-condition: there will be one more item in the queue.
                  If counter is used, then the value will be increased.
      <end process>

      <process> Process name: <returns item> dequeue
            Pre-condition: the queue must not be empty
            Method: pop the item at the front end of the dataholder.
            Post-condition: the queue will have one item less. If
                  counter is used, then the value will be decreased.
      <end process>

<end definition>
```

Types of queues

Simple Queue

In simple queue, data can be inserted from rear end and removed from front end. This method of out putting the first input element first is known as FIFO (first in - first out) - whoever comes first, will get out first. We are going to use an array as a simple queue. Since array is used, it is also known as **sequential** queue.

Since we are doing insertion and removal from different end, we need two pointers to keep records of the rear side and front side.

1	5	6	7
front			rear

Now, if we add new element '2', then the queue will be;

1	5	6	7	2
front				rear

If we remove two item, then the queue will look like;

6	7	2
front		rear

Initialization

The logic behind the pop function is that, whenever we have to pop a data in the queue, we first read the data currently pointed by the pointer, and then decrement the pointer value by '1'. Therefore, the front pointer must be initialized to '0'.

To push the value on the queue, we first increment the pointer value by '1' and then push the value in the array. Therefore, the rear pointer must be initialized to -1.

Queue empty

The queue will be empty if **front > rear**. At point, the queue must be reset by assigning

front = 0
rear = -1

Queue full

Queue will be full whenever the front points to 0, and then rear = MAXSIZE - 1 i.e. maximum numbers of elements in the array.

Elements rolling

If the front pointer points to some other location except 0, and rear = MAXSIZE -1, we can insert new values in the queue but, there are some empty locations in the front. In this situation, we need to roll the elements in the queue.

MAXSIZE = 8

		3	1	10	7	6	9
		front					rear

After roll-back:

3	1	10	7	6	9		
front					rear		

Now we can add more data here.

Algorithm

Push (enqueue) operation

```
1.  First we must check whether the queue is full or not?

      if front == 0 and rear == maxSize -1

            return Push-invalid

2. If the front is greater than rear i.e. the queue is empty then, we
must reset the queue and then push the value

      else if front > rear

            reset queue (i.e. front = 0, rear = -1)
            item[++rear] = x

3. If the rear = maxSize-1 but front is greater than 0, then there are
some empty slots in front.  Therefore, we roll the queue and push.

      else if front != 0 and rear == maxSize -1

            rollQueue
            item[++rear] = x

4. Else, in this case the rear will be less than maxSize-1.

      else if rear < maxSize -1

            item[++rear] = x
```

Pop (dequeue) operation

Here, we just care whether the queue is empty or not. If empty, then pop is invalid, otherwise we can just pop the value from front end.

```
If front > rear  (this is the empty case)

      Return pop-invalid
```

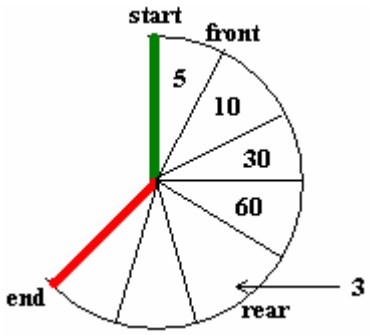
```
Else

      Return item[front++];
```

Circular Queue

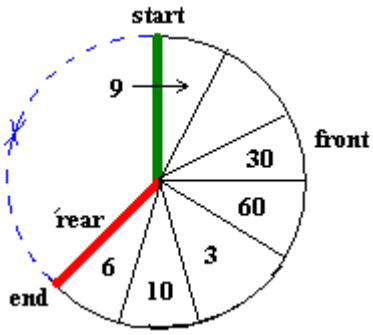
As you can see in the simple queue, every time there are some empty space in the beginning and rear = MAXSIZE - 1, the elements needs to be rolled. The rolling process reduces the performance of the program if queue is accessed thousands of times.

Solution to such problem could be the use of circular queue.

In a circular queue, whenever the elements are pushed i.e. **enqueue**, the rear pointer is increased and if it reaches to MAXSIZE-1, the rear pointer just goes to the front and then starts filling the elements. Therefore, if you just look at the logical representation it would be like;



And if the rear = MAXSIZE - 1 and front > 0, then



The push operation here is a clock-wise operation. And definitely, pop operation as well.

In simple queue, whenever the value of 'rear' goes less than that of 'front', then the queue is supposed to be empty. But in case of circular queue, as you can see, the property of circular queue itself allows less value of 'rear' for non-empty queue. Now, the problem is that, how could we determine whether the queue is empty or not?

Can you think of any logic/method to determine the emptiness of the queue?

Yes, we can use a counter to hold the number of elements. Whenever element is pushed on to the queue, the value of counter is increased and whenever elements are popped from the queue, the value of counter is decreased.

Algorithm

Push (enqueue) operation

```
1. First we must check whether the queue is full or not?

      if counter == maxSize -1

            return Push-invalid

2. If the rear = maxSize-1, and counter < maxSize, then rotate
clockwise and then put the elements

      else if rear == maxSize-1 && counter < maxSize

            rear = 0
            item[rear] = x

3. Else, in this case the number of elements will always be less than
maxSize-1.

      else

            item[++rear] = x
```

Pop (dequeue) operation

Here, we care of two situations.

i. whether the queue is empty or not. If empty, then pop is invalid, otherwise we can just pop the value from front end.

ii. whether we have encountered to the start or not.

```
1. If counter == 0  (this is the empty case)

      Return pop-invalid

2. Else if front > maxSize-1

      front = 0
      return item[front]

3. Else

      Return item[front++];
```

Double-Ended Queue (DE-Queue)

DE-queue - double ended queue is a data structure consisting list of items on which the following operations are possible:

DPush(x)	insert item x on the front end of DE-queue D.
x = DPop()	remove the front item and place it to x.

DInject(x)	insert item x on the rear end.
x = Deject(x)	remove rear item & place to x.

For initiating the DE-queue, set the front and rear pointers to '-1'.

Assumptions

i. Deque D[maxSize] is declared.
ii. rear = front = -1 (initialized).
iii. count = 0 (for number of items count in queue.)

Characteristics

If, f = min i.e. 0, then DPush is invalid.
If, r = max i.e. > maxSize -1, then Dinject is invalid.
If, there is only one element and we are going to pop (either by using DPop or DEject), then it's better that we reset the front and rear pointers to the beginning. Good practice to be a **good citizen**.
If, there is only one element at last, then DInject invalid.

For DPush and DPop operations, we just work with 'front' pointer and for DInject and DEject operations, we work with 'rear' pointer.

Algorithm

DPush(x) routine

```
i. if count = maxSize

      begin
            print "DPush operation is not allowed."
            return
      end

ii. if count = 0

      begin
            front ← 0
            D[front] ← x
            rear ← 0
            count++
            return
      end

iii. if count = 1 and front = 0

      // here, can we just check like count < maxSize and front = 0 ?

      begin
            print "DPush invalid! But you can do DInject operation."
            return
      end

iv. if count != maxSize and front != 0

      begin
            front--
```

```
            D[front] ← x
            count++
            return
      end
```

DPop() routine

```
i. if count = 0

      begin
            print “DPop operation is not allowed”.
            return
      end

ii. if count !=0

      begin
            *x ← D[front]
            front++
      end
```

Can you think of developing algorithm for DInject and DEject?? Do this before moving forward.

DInject(x) routine

```
i. if count = maxSize

      begin
            print “DInject operation is not allowed.”
            Return
      end

ii. if count = 0

      begin
            rear ← 0
            D[rear] ← x
            front ← 0
            count++
            return
      end

iii. if count = 1 and rear = maxSize

      // same as DPush, can we just check count < maxSize and
      // rear == maxSize

      begin
            print “DInject invalid! But you can do DPush operation.”
            return
      end

iv. if count != maxSize and rear != 0

      begin
            rear++
            D[rear] ← x
```

```
            count++
            return
      end
```

DEject(x) routine

```
i. if count = 0

      begin
            print “DEject operation is not allowed”.
            return
      end

ii. if count !=0

      begin
            *x ← D[rear]
            rear--
      end
```

Tracing of algorithms with data.

Programming queues in C

Simple Queue

Look at program 15.

Circular Queue

Look at program 16.

Double-ended Queue

Look at program 17.

Priority queues and their application

The word 'priority' is meant for the determination of higher or lower value of certain objects. In computer, every data is represented by a number. Therefore, to determine the higher or lower values of the items, we just look at the numeric values of the data and then compare i.e. just a number comparison.

A priority queue is also a linear or simple queue. In simple queue, we saw that the elements in the queue are inserted from the rear end and elements are popped from the front end. In case of priority queue, the insertion operation can either be done as in linear queue or at arbitrary position i.e. random. We shall look at random insertion in the '**maintenance of priority queue**.'

Consider that, we have a queue as shown below;

5	10	1	0	3	9		

For linear queue, we just pop data from the front end using front-pointer. But, in case of priority queue, either the largest or the smallest element is popped. If the logic draws **largest element** and then second largest, and so on, then it is known as **descending priority queue** (it's because, the order of the popped elements will be descending). If smallest element is drawn then the queue is known as **ascending priority queue** (here, the order of the popped elements will be ascending).

Insertion (push/enqueue)

For the time being, we just insert according to the enqueue algorithm of linear queue.

Deletion (pop/dequeue)

Descending priority queue

```
i.  Start from the front side, counter = 0, position = 0

ii. Set, max ← item[counter]

iii.If max < item[counter++]
            max = item[counter]
            position = counter

iv. Do step (iii) until counter <= rear pointer.

v. At this point, we know the position of the largest number (position)
      So, let's pop the value
            x = item[position]
                        noOfItems--;
```

Ascending priority queue

```
i.  Start from the front side, counter = 0, position = 0

ii. Set, min ← item[counter]

iii.If min > item[counter++]
            min = item[counter]
            position = counter

iv. Do step (iii) until counter <= rear pointer.

v. At this point, we know the position of the smallest number
(position). So, let's pop the value
            x = item[position]
                        noOfItems--;
```

Empty queue and full queue can be checked by using the algorithms of linear queue.

Maintenance of priority queue

As elements might be deleted from any location, the continuation of the elements will be disturbed each and every time the elements are deleted. Due to the deletion of elements at arbitrary position, the maintenance of priority queue is even more challenging than linear queue. Some proposed maintenance strategies for proper use of all of the memory in the priority queue are given below.

1. **After deleting an element, a value of '-1' is put on the slot**. When the rear pointer reaches to maxSize – 1, then all of the elements are compacted to the front end. Doing so, there will be some empty slots at the rear end. Now new elements can be added.

   If negative numbers or some other data need to be stored then, a new field is created to store the information whether the slot is empty or not. You can store '1' to indicate the slot is reserved and '0' to indicate the slot is free.

   Demerits

   If there are lots if empty slots in zig-zag fashion for a large (really large) queue, then the compaction function itself requires lots of processing.

   During the pop operation, search for the maximum or minimum value will also require lots of time.

2. After deleting, '-1' is kept on the slot but while inserting new element, **the new element is stored in the first empty slot**. This logic requires search for the first empty slot.

3. **Compact the queue after deletion of every elements**!! This solution makes insertion efficient but the deletion process will be time consuming. Modification to this solution is that we shift the elements right to the deleted element or elements left to the deleted elements either to right or to left so that they will form a circular queue. For this modification, we have to take care of both front and rear pointers and measure whether right or left shifting is efficient which is also time consuming.

4. **Create a sorted priority queue**. Here, instead of deletion, elements are inserted in a sorted manner during insertion. One good feature of this solution is that, to search a given element, the search time will greatly reduced as binary search algorithm can be applied.

(Reference: *Data Structures using C and C++*, by Langsam, Augenstein, Tenenbaum)

Applications of priority queue

In most of the resource sharing system, priority queues are used. If two of the devices/processes are requesting same resource, then a priority queue is maintained and then the resource is given to that particular device/process.

For example, a computer is in the network; let's say that there are two programs running in the computer. One of the programs is 'Instant Messenger' and another is 'File Downloader'. If both of the applications send request the operating system to send some data, to server (as operating system is responsible for all of the network protocol management) at the same time, then the OS looks at the priority queue and then chooses 'Instant Messenger' as messengers have got higher priority than 'File Downloader'.

Similar mechanism is also used by CPU for the selection of priority interrupt. An interrupt is a request to the CPU for processing. If two devices like hard-disk and keyboard send interrupt at the same time, then the CPU would respond to the hard-disk interrupt first as in priority queue, the priority of hard-disk interrupt is set higher than that of keyboard already.

Homework 3: Write program to implement priority queue with modification no. 2.

Programming the priority queue.

Look at program 18_1, 18_2, 18_3.