Queues

Slides

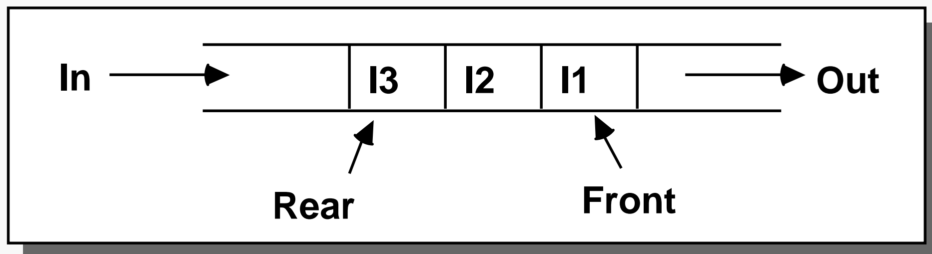
1. Table of Contents
2. Queue Definition
3. Queue Application: Op Sys
4. Multi-Level Feedback Queuing Sys
5. Op Sys Processes
6. Dynamic Priorities
7. Sequential Queue Implementations
8. Circular Queue
9. Circular Queue: Interface
10. Circular Queue: Implementation
11. Linked-Queue: Implementation
12. Linked-Queue: Implementation (cont)
13. Drop-Out Stack
14. DOS Implementations
15. Deque

 ©1995-2001 Barnette ND, McQuain WD

Queue Definition

Description

- Restricted (two-tailed) list structure
- Dynamic FIFO Storage Structure
  - † Size and Contents can change during execution of program
  - † First in First Out
  - † Elements are inserted (enqueue) into the rear and retrieved (dequeue) from front.
- Single Type Element (not generic)
- Real life: Any Ticket Line.



Insertion Order:
I1, I2, I3

Operations

- **Queue** ( ) ;
  - † set queue to be empty
- bool **Empty** ( ) ;
  - † check if queue is empty
- bool **Full** ( ) ;
  - † check if queue is full
- **Enqueue** (const Item& item) ;
  - † *Insert* item into the queue
- Item **Dequeue** ( ) ;
  - † *Remove* & return the item at the front of the queue

```
Queue Que ;
Que.Enqueue(  I1 ) ;
Que.Enqueue(  I2 ) ;
Que.Enqueue(  I3 ) ;
```

Some implementations define:
Item Front( ) ;
Returns first item in the queue, but does not remove it.
Dequeue() ;
In this case removes the first item in the queue, but does not return it.

 ©1995-2001 Barnette ND, McQuain WD

Operating System (Fictional)

- A (fictional) batch operating system queues jobs waiting to execute according to the following scheme:

- Users of the system have relative priorities according to their ID number :

users 100-199	highest
users 200-299	next highest
users 300-399	
• •	•
users 800-899	next to lowest
users 900-999	lowest (jobs run only when no other jobs are present in the system)

Batch Operating Sys executes each job to completion before executing other jobs.

- Within each priority group, the jobs execute in the same order that they arrive in the system. (FIFO)

- If there is a highest-priority job queued, it will execute before any other job; if not, if there is a next-to-highest-priority job queued, it will run before any lower-priority jobs, and so on. That is, a lower-priority job will only run when there are no higher-priority jobs waiting.

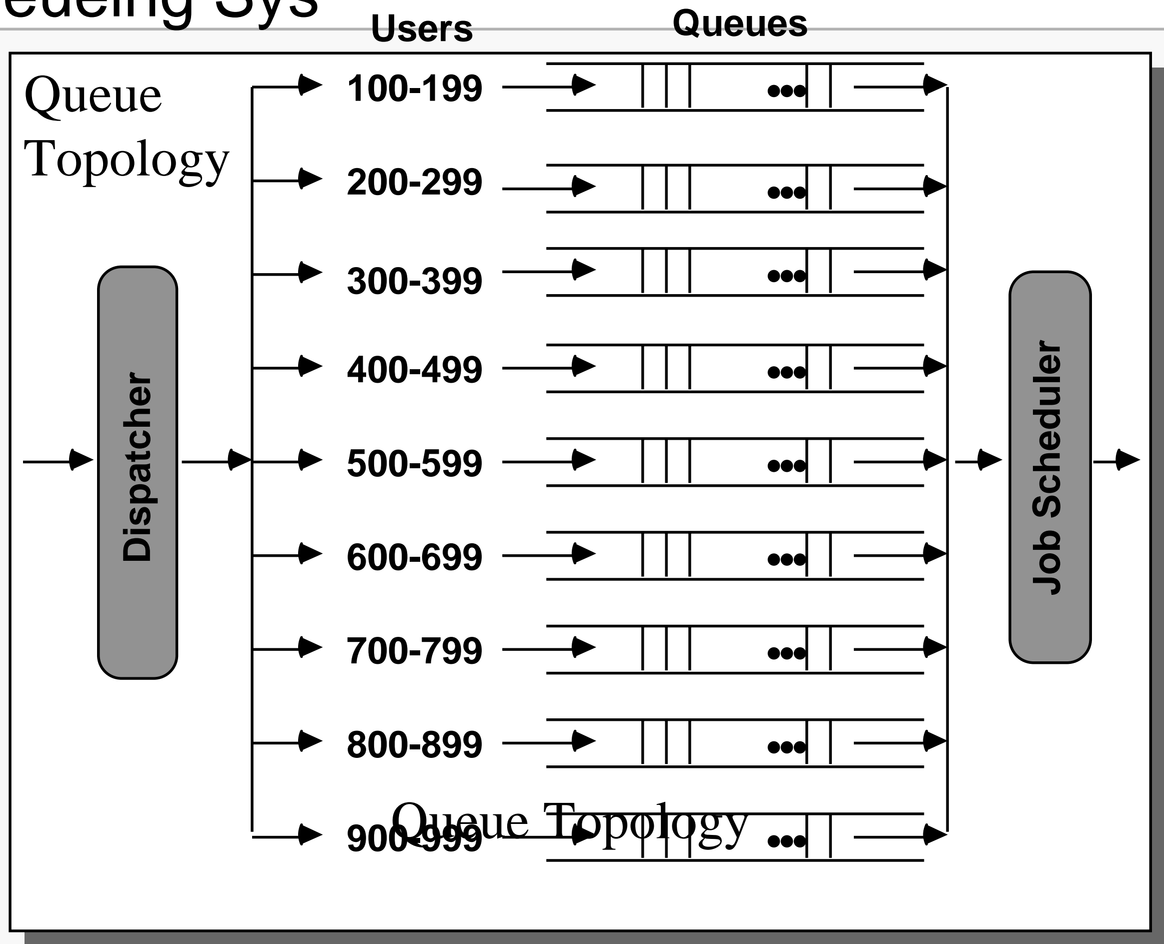
- The system has an array of queues

```
typedef Queue mlfQueSys [MAXLEVELS];
mlfQueSys jobs;
```

- to hold the queues for the various priority levels.

Problem taken from: PASCAL Plus Data Structures, 4th ed., N. Dale and S.C. Lilly, D.C. Heath Pub., ©1995

Multi-Level Feedback Queueing Sys



Queue Topology

Queue Item type :

```
class Item { // PCB (Process Control Block)
private:
   int userid, processid, priority;
   state processState;     // running ready blocked
   tablePtr procMem;          // pointers process's
  memory
                              // segment/page table
   rscrPtr rscrs;             // pointers to allocated
                              //         resources
   regPtr regs;             // register save area
public:
   // public function prototypes . . .
};
```

Op Sys Functions

- Any of the standard queue procedures (Enque, Deque, etc.) may be called to accomplish the following:
- Write a procedure ADDJOB (**Dispatcher**) that receives a user ID and a token (representing the job to be executed) and adds the token to an appropriate queue for the user's priority level.
- Write a procedure GETNEXTJOB (**Job Scheduler**) that returns the token for the next highest-priority job queued for execution.
- The system is going down for maintenance. All jobs waiting for execution will be purged from the job queues. However, this is a very user friendly system that notifies users when their jobs are being killed, so they will know to resubmit the jobs later.

  PROCEDURE Notify (Token, Messageid)
  // takes care of notificaton.
- Write a procedure CLEANUPJOBS that sends message 7 to all the users with queued jobs. (Call Procedure NOTIFY). Of course, send messages to the highest-priority users first.

Time-Sharing

- Jobs in each queue are given time slices of the CPU
- Assume PCB contains pcb.time, i.e. num of time slices
- Problem : Prevent large jobs from "**hogging**" the CPU
- Solution : code a function **adjustPriority** (called by the op sys when necessary) to move "hogging" jobs down to next lower priority queue.
- Hogging Job - program that has received more than some **HOG** (system constant) time slices
- Only jobs from current highest queue need be moved to free up the system.

 ©1995-2001 Barnette ND, McQuain WD

```
void adjustpriority ( mlfQueSys& jobs , bool& err ) {

   Item      proc;
   int       q = 0;
   bool      found = false;
   Queue     temp ;

   // find current highest que with jobs
   while ( ( !found )  && (q < MAXLEVELS-1) )
      if ( !jobs[q].Empty() )
         found = true;
      else ++q;

   err = !found; //sys empty or all jobs in lowest Que
   if ( !err) { // adjust
      while ( ! jobs[q].Empty() ) {
         proc = jobs[q].Dequeue();
         if ( proc.GetTime() > HOG )
          jobs[q+1].Enqueue( proc );
         else temp.Enqueue( proc );
            }//while
      while ( ! temp.Empty()) {
         // restore small jobs
         proc = temp.Dequeue();
         jobs[q].Enqueue( proc );
           }//while
   }//if
}// adjustpriority
```

'*Hogging*' jobs whose priority has been changed dynamically by insertion in next lower level queue are still considered 'large' jobs (i.e. their number of time slices received has NOT been reset).

©1995-2001 Barnette ND, McQuain WD

Sequential Queue Implementations

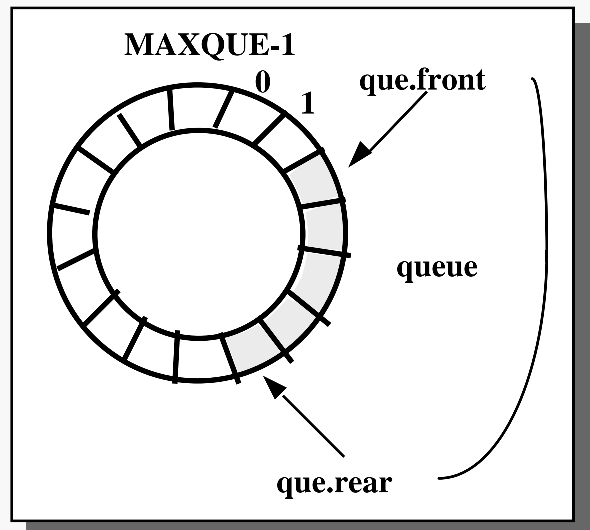
Linear Array

- Front or Rear must be fixed at one end of the array
- Enqueing or Dequeing requires inefficient array shifting.

Circular Array

- Code operations to force array indicies to 'wrap-around'
  - † front = (front + 1) % MAXQUE ;
  - † rear = (rear + 1) % MAXQUE ;

- front and rear indicies delimit the bounds of the queue contents

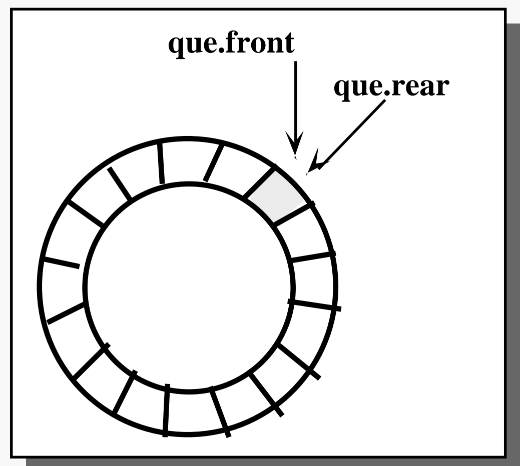


- Enqueue
  - † Move the que.rear pointer 1 position clockwise & write the element in that position.

- Dequeue
  - † Return element at que.front and move que.front one position clockwise

- Count (queue size) is stored and maintained or boolean full status flag maintained.

One-element Queue



©1995-2001 Barnette ND, McQuain WD

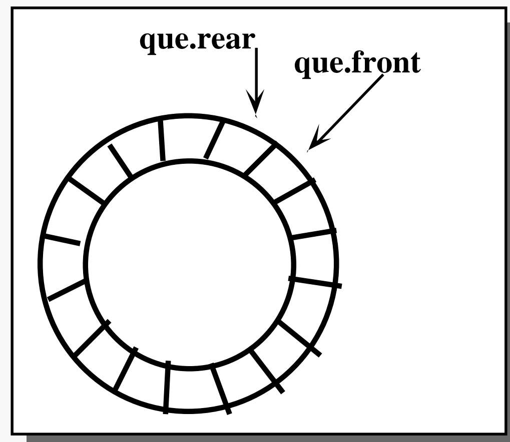
Efficient Implementation

– Empty or Full Queue?

Assume queue has 1 element.

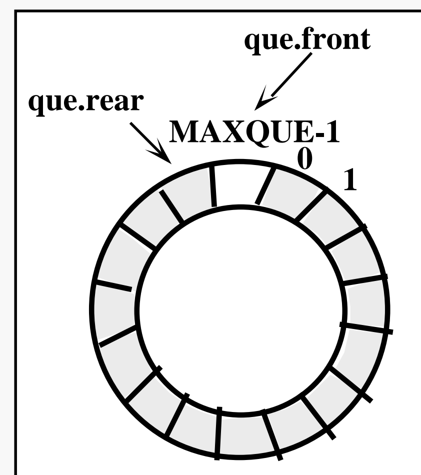
Dequeue the element.
Where are the indicies?

Fill up the queue.
Where are the indicies?



– Solution

- † design implementation to ensure that different states of the queue are represented distinctly
- † Eliminates need to maintain a queue size count.
- † *Front* refers to the position preceding actual front element
- † full queue:
  - ‡ contains (max - 1) elements.



– **Tradeoff**:

- † one memory location saves processing (maintaining queue size count)

– Distinct States

- † Full Queue:
  `(que.rear + 1) % MAXQUE == que.front`
- † Empty Queue:
  `(que.rear == que.front )`
- † One-element Queue:
  `(que.front + 1) % MAXQUE == que.rear`

 ©1995-2001 Barnette ND, McQuain WD

Circular Queue: Interface

Array Representation

– **Queue.h**

```
const int MAXQUE = 100;
//typedef        arbitrary Itemtype;
#include "Item.h"

class Queue {
 private:
  int         Front;
  int         Rear;
  Item        Items[MAXQUE];
 public:
  Queue();
  bool Empty();
  bool Full();
  void Enqueue (const Item& item);
  Item Dequeue ();
};
```

No count variable required.

Queue Interface does NOT change, unaffected by underlying representation.

Considerations

– Requires establishment of conventions for the unique representation of queue states.

– Consistency of conventions must be maintained between all operation functions

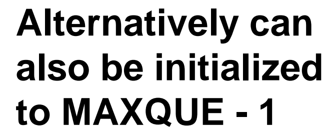
– Deque'ed items will remain in the queue (array) until they are overwritten

©1995-2001 Barnette ND, McQuain WD

Circular Queue: Implementation

Queue.cpp

```
#include "Queue.h"

Queue::Queue() {
   Front = 0;
   Rear  = 0;
}

bool Queue::Empty ( ) {
   return ( Front == Rear );
}

bool Queue::Full ( ) {
   return ( ((Rear+1) % MAXQUE) == Front );
}

void Queue::Enqueue(const Item& item ) {
   Rear = (Rear + 1) % MAXQUE;
   Items[Rear] = item;
}

Item Queue::Dequeue( ) {
      Front = (Front + 1) % MAXQUE;
      return( Items[Front] );
}
```



©1995-2001 Barnette ND, McQuain WD

Linked-List Representation

- Queue is a structure containing two pointers:
  - front: points to the head of the list
  - rear: points to the end of the list (last node)
- Enque operates upon the rear pointer, inserting after the last node.
- Deque operates upon the front pointer, always removing the head of the list.
- Empty queue is represented by NULL front & rear pointers

List Class Implementation

- Queue.h

```
#include "LinkList.h"
//typedef        arbitrary Item
#include "Item.h"

class Queue {
private:
  LinkList que;
public:
  //Queue(); //LinkList constructor
  bool Empty();
  bool Full();
  void Enqueue (const Item& Item);
  Item Dequeue ();
};
```

Class aggregation

(implement using Class List operations)

No change should be made to Queue interface to achieve abstraction and information hiding.

©1995-2001 Barnette ND, McQuain WD

Linked Que Implementation

Queue.cpp

```
#include "Queue.h"

bool Queue::Empty ( ) {
  return (que.isEmpty() );
}

bool Queue::Full ( ) {
  Item*  newNode= new(nothrow) Item;
  if (newNode == NULL )
     return true;
  delete newNode;
  return false;
}

void Queue::Enqueue(const Item& item ) {
   que.gotoTail();
   que.Insert(item);
}

Item Queue::Dequeue( ) {
  Item temp;

  que.gotoHead();
  temp = que.getCurrentData();
  que.DeleteCurrentNode();
  return( temp );
}
```


©1995-2001 Barnette ND, McQuain WD

Drop-Out Stack (dos)

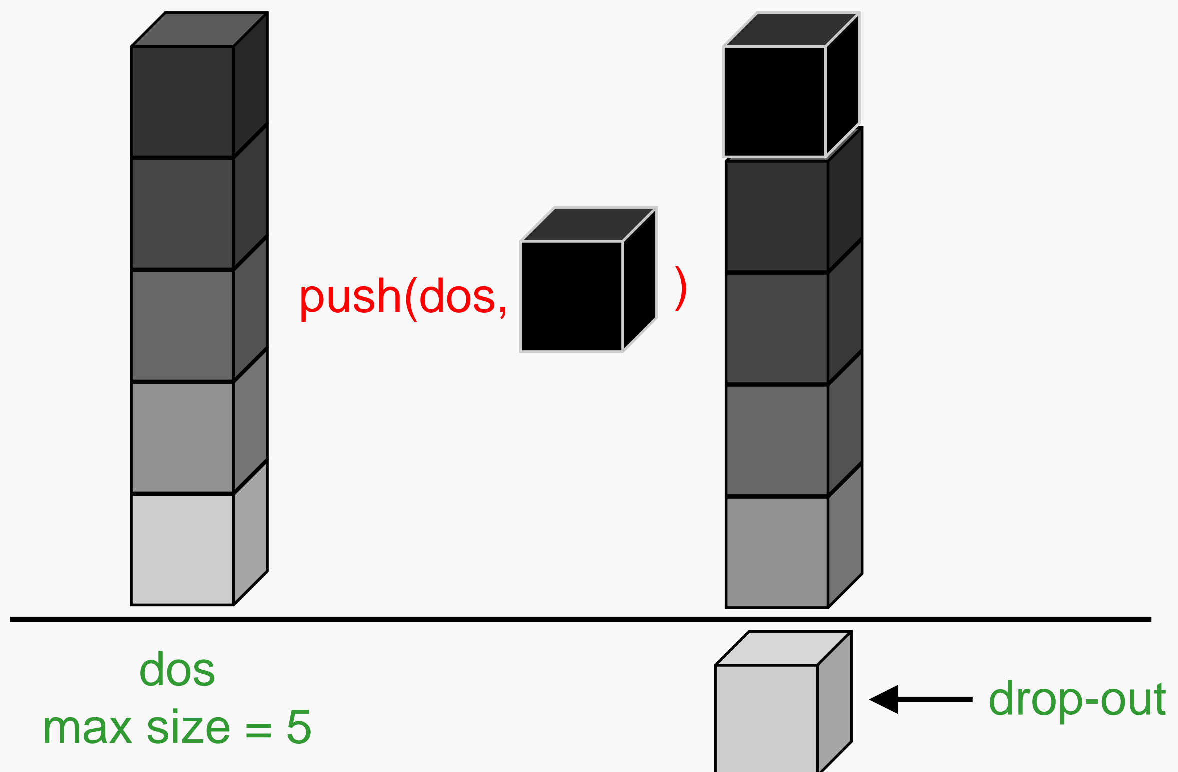
"Bottomless" Stack

- Variation of a regular stack.
  - † No fullstack operation (i.e. a **dos** can never become full).
- "Drop-Out" Stack of size N has following behavior:

  Let the integers 1 , 2 ... be the first elements PUSHed onto the stack respectively.

  After the Nth integer element is PUSH'ed, integer 1 is at the "bottom" of the stack, with 2 immediately above it.

  After the N+1 integer is PUSHed, 1 **Drops-Out** of the bottom and integer 2 is now at the bottom of the stack.
- Note: any element that Drops-Out of the stack never **reenters** the stack automatically from the bottom due to POPs being performed.

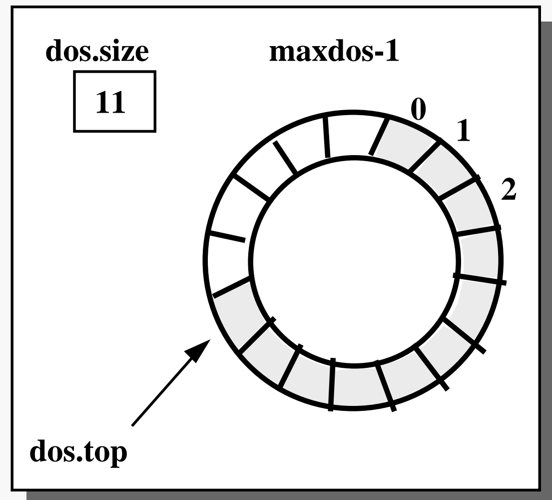


©1995-2001 Barnette ND, McQuain WD

DOS: Implementations

Representations

- Linear Array
  - † Bottom is fixed at first index.
    - ‡ Problem : inefficient down shifting of elements required when Drop-Outs must occur.
- Circular Array
  - † Elements PUSHed when Dropping-Out occurs simply store over elements at bottom.
- Problem: when are all the elements POP'ed off?



  - † Solution #1:

    Size counter stores number of stack elements.

    Requires extra processing & checking.

  - † Solution #2:

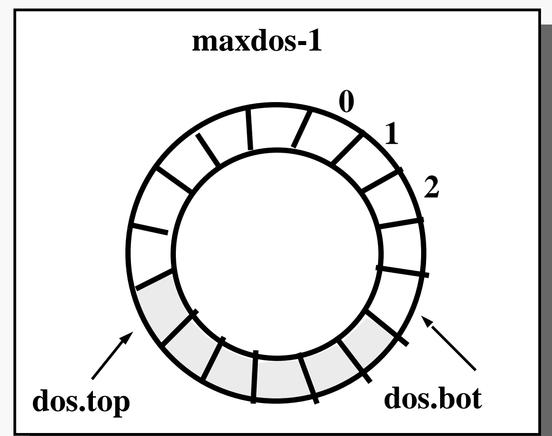
    Bottom index.

    Empty Stack ?

    when top = bottom.

    Drop-Outs ?

    When PUSHing the bottom index is moved forward.



Analogous to circular queue implementation.

 ©1995-2001 Barnette ND, McQuain WD

Deque

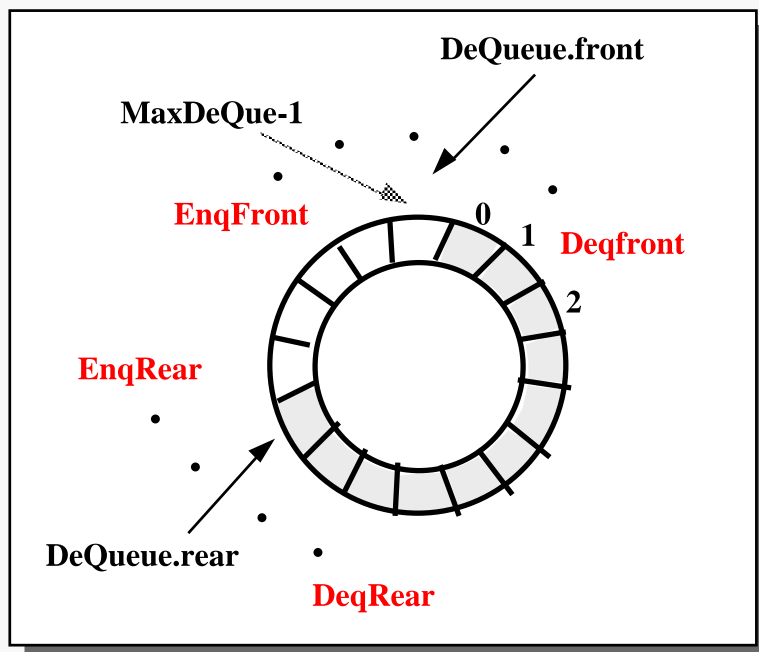
“Double-Ended” Queue

- variation of a regular queue.
- elements can be added and removed at either the rear or front of the queue, but nowhere else in the queue.
- operations:

  Deque(), Empty(), Full(), EnqRear(), EnqFront(), DeqFront(), DeqRear()
- generalization of both a stack and a queue.

Design

- Linear Array
  - † Front or Rear is fixed at first or last index.
  - † Inefficient down shifting of elements required when Enqueuing or Dequeuing to the fixed end.
- Circular Array
  - † Front & Rear move both forward & backward around the array.



©1995-2001 Barnette ND, McQuain WD