Queues

Queue - an ordered collection of data items from which the oldest at the front may be deleted. New items are inserted at the rear.

No limit on number of items.

No limit on nature of items.

Implementations usually will place limits

Distinction between front and rear is arbitrary but traditional.

1

Queues

FIFO - First In, First Out

The ordering is essentially chronological

Analogy to waiting line for Space Mountain ride at Disney world.

2

Uses of Queues

Operating systems often use FIFO queues to process jobs (eg. requests for resources).

Use queues when application has a FIFO characteristic (First Come - First Served)

3

ADT Queue

Data An **empty** list of values with a reference to the first (front) and last (rear) items

Methods

QEmpty

Input:	None
Precondition:	None
Process:	Check if the queues contains any data items.
Postcondition:	None
Output:	Return 1 or true if queue is empty and 0 or false otherwise.

QDelete

Input:	None
Precondition:	Queue contains meaningful data values
Process:	Remove an item from the front of the queue
Postcondition:	The queue contains one less data item
Output:	Return the deleted value.

QInsert

Input:	A data item to be stored in the queue
Precondition:	None
Process:	Store an item at the rear of the queue
Postcondition:	The queue contains one additional data item
Output:	None

QPeek

Input:	None
Precondition:	Queue contains meaningful data values
Process:	Retrieve the value of the data item at the front of the queue
Postcondition:	The queue is unchanged.
Output:	Return the value of the data item at the front of the queue.

end ADT Queue

4

Queue ADT

Standard queue operations

- Queue insert
- Queue delete
- Constructor
- data: data values; front & rear pointers.

Optional methods include

Queue copy, etc.

5

Queue Implementations

- Array Implementation
- Linked Implementation

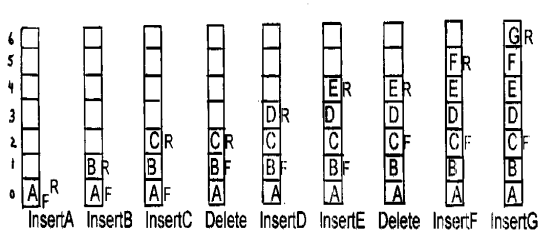
6

Queue: Array Implementation

☹Requires queue to be homogeneous
☹Places size limits - static allocation
☺Exploits Random Access
😐Need front and rear pointers

7

Queue: Array Implementation



If we use array like we did for stacks

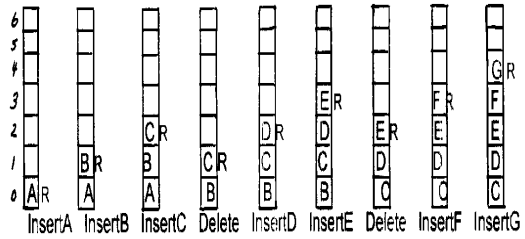
8

Queue: Array Implementation

☺ Inserts & Deletes are **O**(1)
😐 Locations 0 & 1 still contain A & B
😐 Need pointers to separate data & garbage.
😐 Queue travels through the array.
☹ No place to insert another value.
☹ Array not fully utilized - **false overflow**

9

Queue: Array Implementation: Alternative



10

Queue: Array Implementation: Alternative

Now we can easily insert additional items without a false overflow.

☺Array fully utilized

☺ Inserts still O(1)

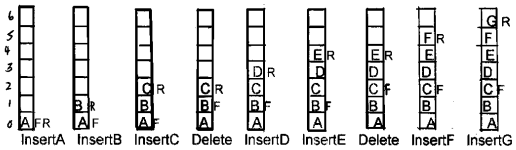
☹Entire queue shifts on every delete O(n)

☹Computationally expensive if n large

😐 Can we do better ?

11

Queue: Array Implementation: Alternative 2



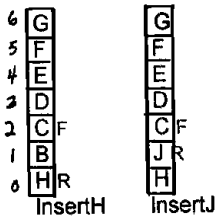
Exploit that available space is contiguous.

Wrap around to beginning of array.

12

Queue: Array Implementation: Alternative 2

Now, we can insert more items.



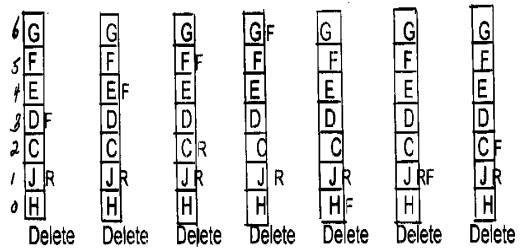
13

Queue: Array Implementation: Alternative 2

☹Need pointers to identify data/garbage
☹Queue still travels through array.
☺Additional insertion done easily
☺Array fully utilized
☹No shifting
☹F and R can occur in either order

14

Queue: Array Implementation: Alternative 2



Queue: Array Implementation: Alternative 2

Delete 7 times

F and R now have the same relationship as in the previous example when the queue full.

Usually solved by keeping a length parameter in the private section.

Standard array implementation of Queue

16

Queue: Linked Implementation

- ☹ Requires queue to be homogeneous
- ☺ No size limits - Dynamic allocation
- ☹ Forces sequential access
- 😐 Need front and rear pointers

17

Queue Code

A simple Linked implementation

```
class QNode {
   Datatype Data;          // any appropriate type
   QNode   Next;
}
```

18

```
public class LQueue {
   private QNode Front, Rear;
   private int size;
                          //optional size parameter

public LQueue() {                    //constructor
   Front = null;
   Rear = null;
   size = 0;                                //if used;
}
```

19

20

```
 public void QInsert ( Datatype Item) {
    QNode Temp = new QNode; //allocate space
    Temp.Data = Item;                  //stuff in info
    Temp.Next = null;              //new item is last
    if (Front == null) Front = Temp;
    else Rear.next = Temp;      //Attach to queue
    Rear = Temp;                //reset Rear pointer
 }

 } //end LQueue
```

21

22

```
public Datatype QDelete () {
        Datatype Item;
    QNode Temp = Front;
    if (Front == null) "error handling"
    else {
        Item = Front.Data;              //Extract information
        Front = Front.Next;                     //Reset front
       if (Front == null) Rear = null           //delete last
           Temp.Next = null;             //disconnect node
           Temp.Data = "a string of blanks";
          return Item;                //Always return deleted
    value
    }
```

23