Data Structures

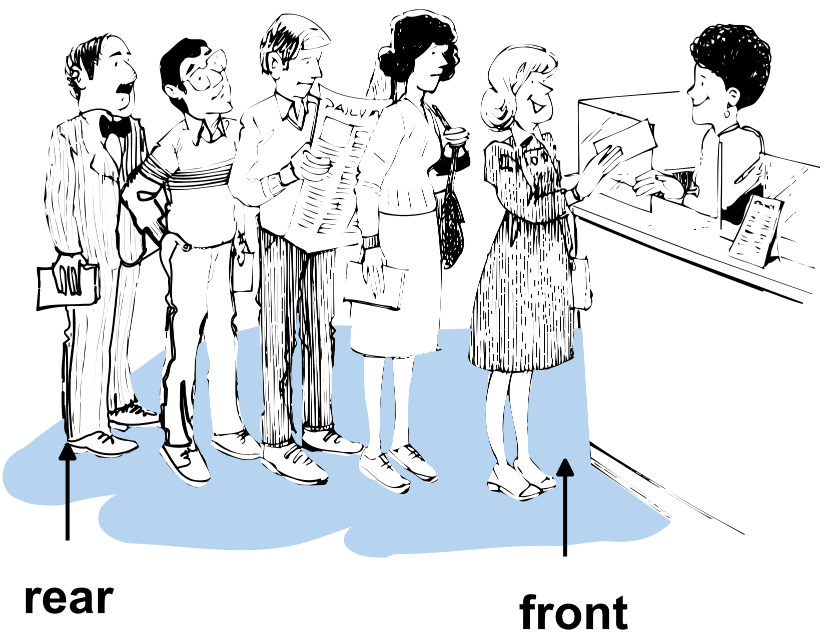
Lecture 5: Queues

Readings: Chapter 4

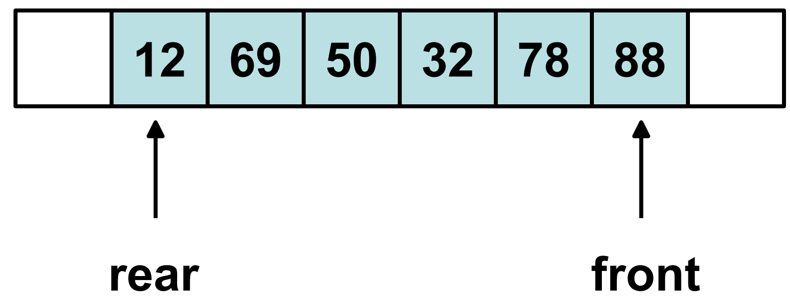
What is a Queue

- *In ordinary English a queue is defined as a waiting line, like a line of people waiting to purchase tickets*
  - *Here, the first person in the line is the first person served*
- A ***queue*** is an ordered collection of items in which all insertions are made at one end (the ***rear***), and all deletions are made at the other end (the ***front***)

Queues



A waiting queue

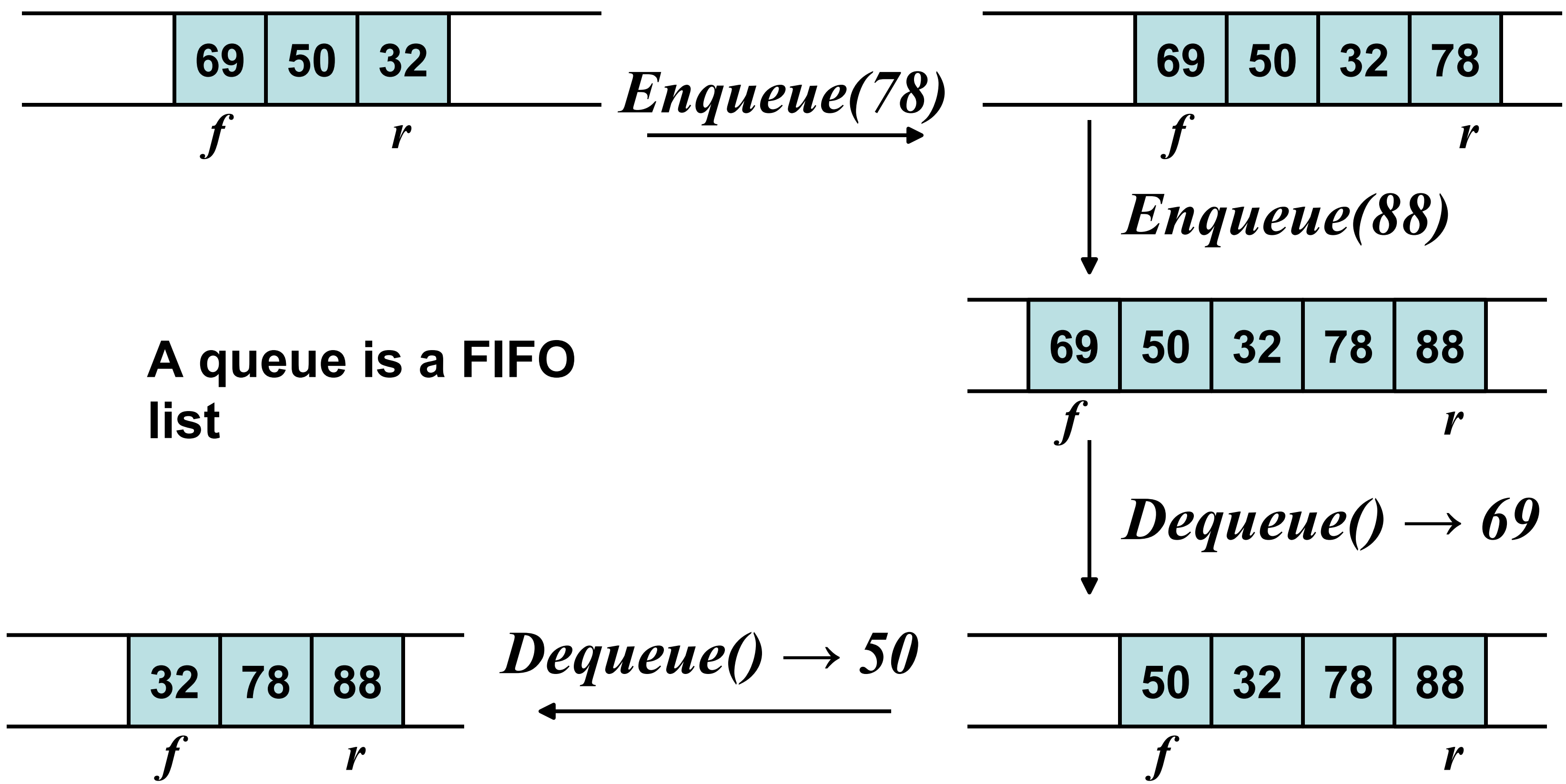


Computer Implementation of queue

Enqueue & Dequeue Operations

- **Enqueue** – Inserts an item at the *rear* of the queue. *Other names – **Add**, **Insert***
- **Dequeue** – Deletes an item from the *front* of the queue. *Other names – **Delete**, **Remove**, **Serve***
- Since it is always the first item to be put into the queue that is the first item to be removed, a queue is a ***first-in***, ***first-out*** or ***FIFO list***
- **Front** and **rear** are also called ***head*** and ***tail*** respectively

Illustration



A queue is a FIFO list

Applications

- **Direct Applications**
  - ***Access to shared resources***
    - *Within a computer system there may be queues of tasks waiting for the printer, for access to disk storage, or even in a time-sharing system, for use of the CPU*
- **Indirect Applications**
  - *Auxiliary data structure for algorithms*
  - *Component of other data structures*

Checking Palindromes

- Read each letter in the phrase. Enqueue the letter into the queue, and push the letter onto the stack.
- After we have read all of the letters in the phrase:
  - Until the stack is empty, Dequeue a letter from the queue and pop a letter from the stack.
  - If the letters are not the same, the phrase is not a palindrome

The Queue ADT

- A ***queue*** of elements of type ***T*** is a finite sequence of elements of ***T*** together with the operations
  - **MakeEmpty(*Q*)** – Create an empty queue ***Q***
  - **Empty(*Q*)** – Determine if the queue ***Q*** is empty or not
  - **Enqueue(*Q*, *x*)** – Insert element ***x*** at the end of the queue ***Q***
  - **Dequeue(*Q*)** – If the queue ***Q*** is not empty, remove the element at the front of the queue
  - **Front(*Q*)** – Retrieve the element at the front of the queue ***Q***, without deleting it

Implementation of Queues

- **The physical model:** *a linear array with the front always in the first position and all entries moved up the array whenever the front is deleted*
- **Array Implementation:**
  - *A linear array with two indices always increasing*
  - *A circular array with front and rear indices and one position left vacant*
- **Linked List**
  - *A linked list with pointers to the front and rear nodes*

Linear Implementation

- For all type of array implementation, ***we set up an array*** to hold the items in the queue
- In linear implementation, ***we use two indices*** to keep track the front and the rear of the queue
- ***To enqueue an item***, we increase the rear by one and put the item in that position
- ***To dequeue an item***, we take it from the position at the front and then increase front by one

The QueueType Declaration

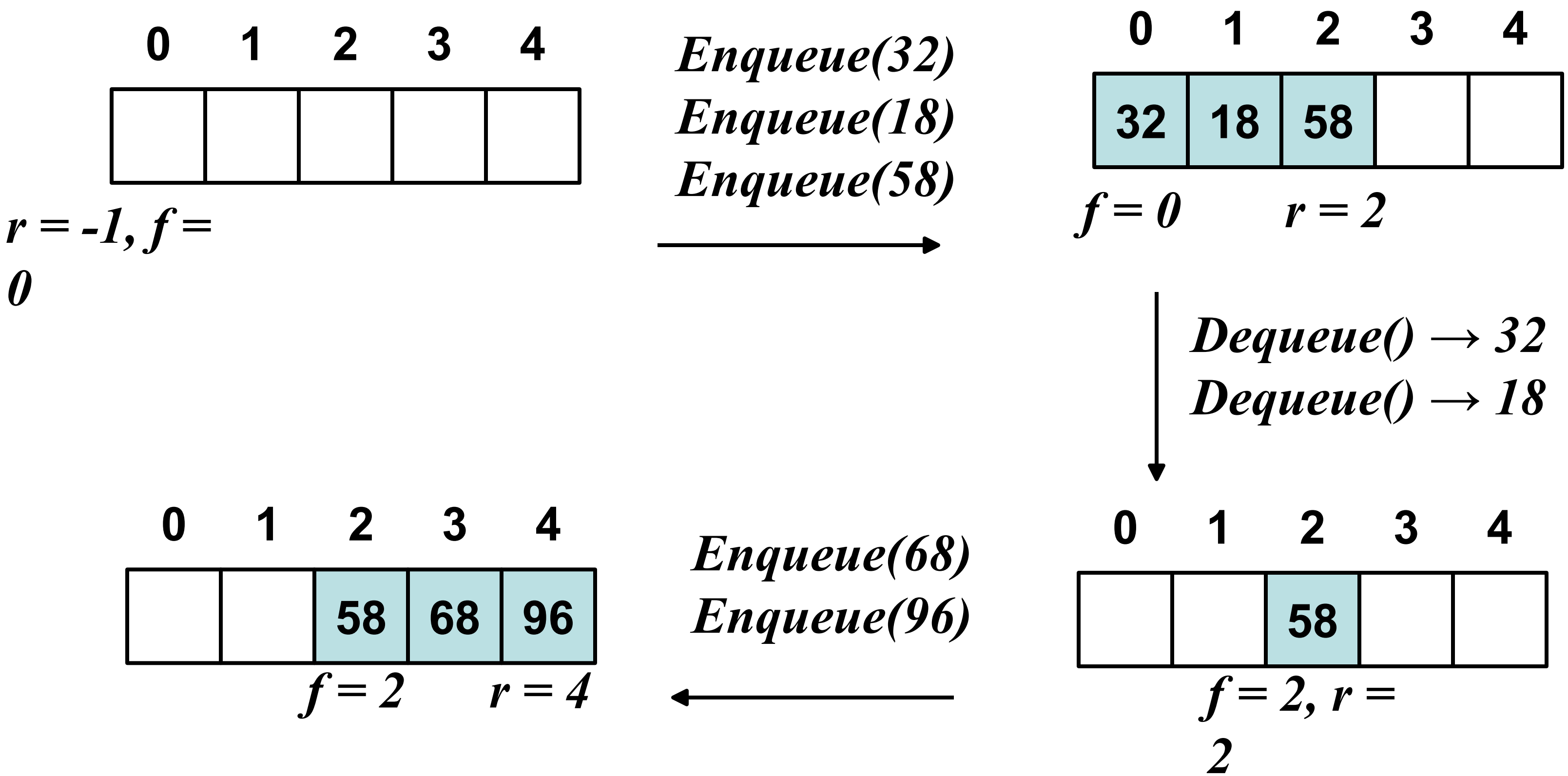
```
#define MAXQUEUESIZE 100

typedef int ItemType;

typedef struct {
     ItemType items[MAXQUEUESIZE];
     int front, rear;
} QueueType;
```

The MakeEmpty Operation

- To initialize an empty queue, we set front to 0 and rear to -1

```
void MakeEmpty(QueueType *pq)
{
     pq->front = 0;
     pq->rear = -1;
}
```

Illustration



A queue with an array of size 5

The Full and Empty function

```
int Empty(QueueType *pq)
{
      return pq->rear < pq->front;
}

int Full(QueueType *pq)
{
      return pq->rear == MAXQUEUESIZE-1;
}
```

The Enqueue function

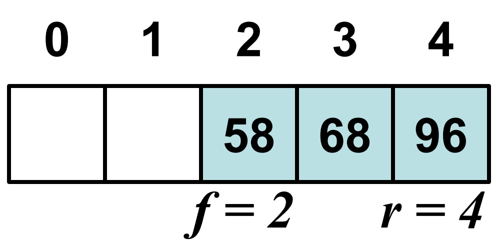
```
void Enqueue(QueueType *pq, ItemType newitem)
{
      if (Full(pq)) {
            printf("Queue Full\n",);
            exit(1);
      }
      pq->items[++pq->rear] = newitem;
}
```

The Dequeue function

```
ItemType Dequeue(QueueType *pq)
{
      if (Empty(pq)) {
            printf("Queue is Empty\n",);
            exit(1);
      }
      return pq->items[pq->front++];
}
```

Problems with Linear Implementation

- Both the rear and front indices are increased but never decreased
- *As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again*

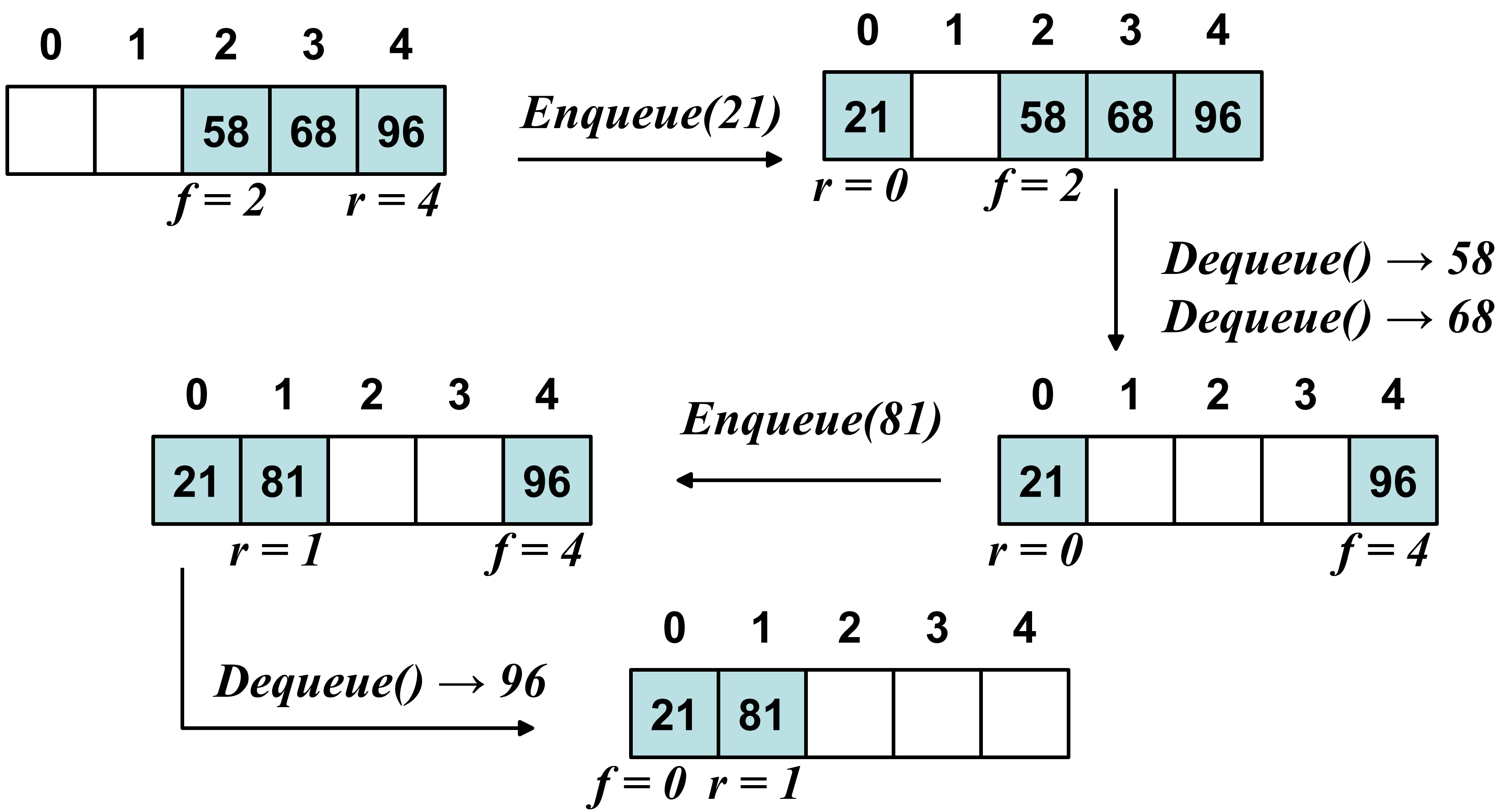


This queue is considered full, even though the space at beginning is vacant

Circular Array Implementation

- View the array, holding the queue, as a circle rather than a straight line
- Imagine the first element of the array (i.e., the element at index 0) as immediately following its last element
- If incrementing either rear or front causes it to go past the array, reset its value to zero
- With this approach, we can always insert an item unless the array is fully occupied

Illustration



Checking Boundary Conditions

- The condition `rear < front` does not hold for empty queue

0	1	2	3	4
21		58	68	96

$r = 0$ $f = 2$

In all cases, rear < front but the queue is not empty

0	1	2	3	4
21	81			96

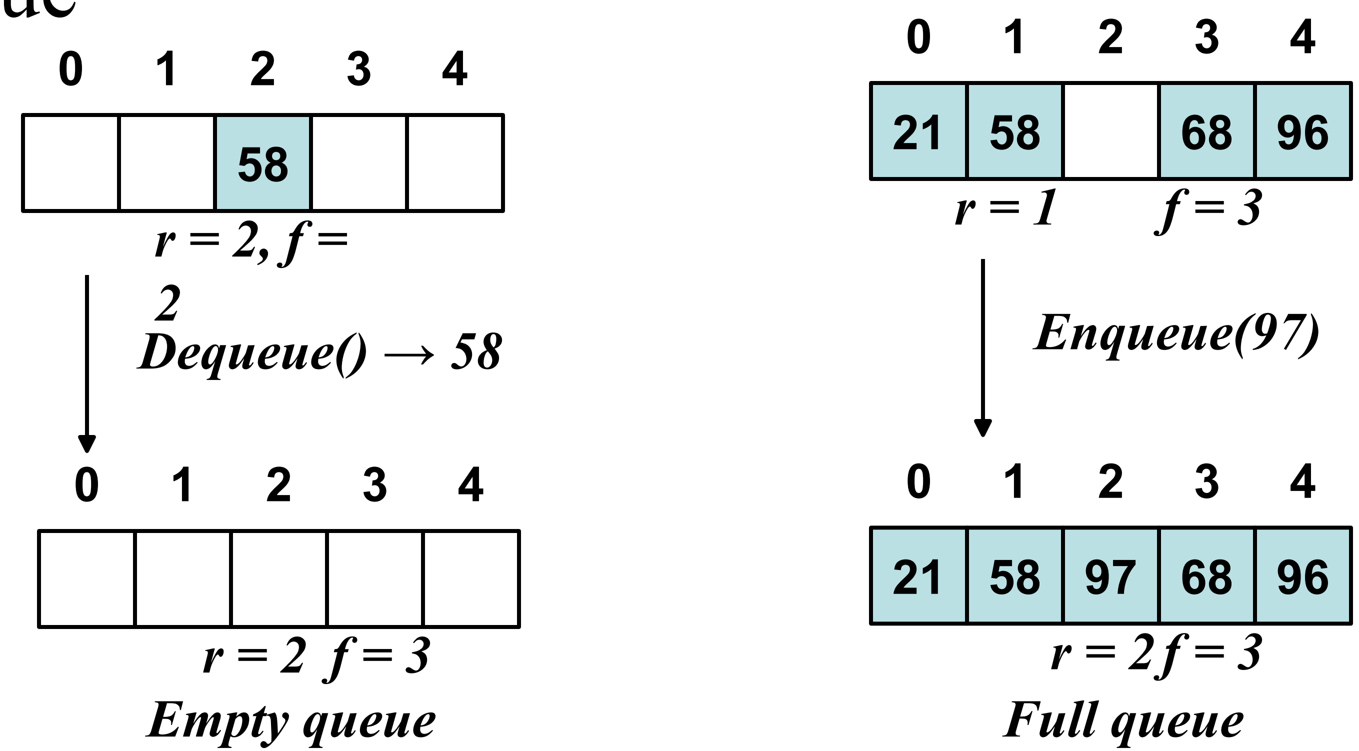
$r = 1$ $f = 4$

0	1	2	3	4
21				96

$r = 0$ $f = 4$

Checking Boundary Conditions

- Same relative positions for an empty and a full queue



Possible Solutions

- At least 3 ways to resolve the problem
  - Leave one empty position in the array so that the queue is considered full when the rear index has moved within two positions ahead of front
    - For convenience, initialize both rear and front to same value for empty queue

We will use this method

Possible Solutions (Cont…)

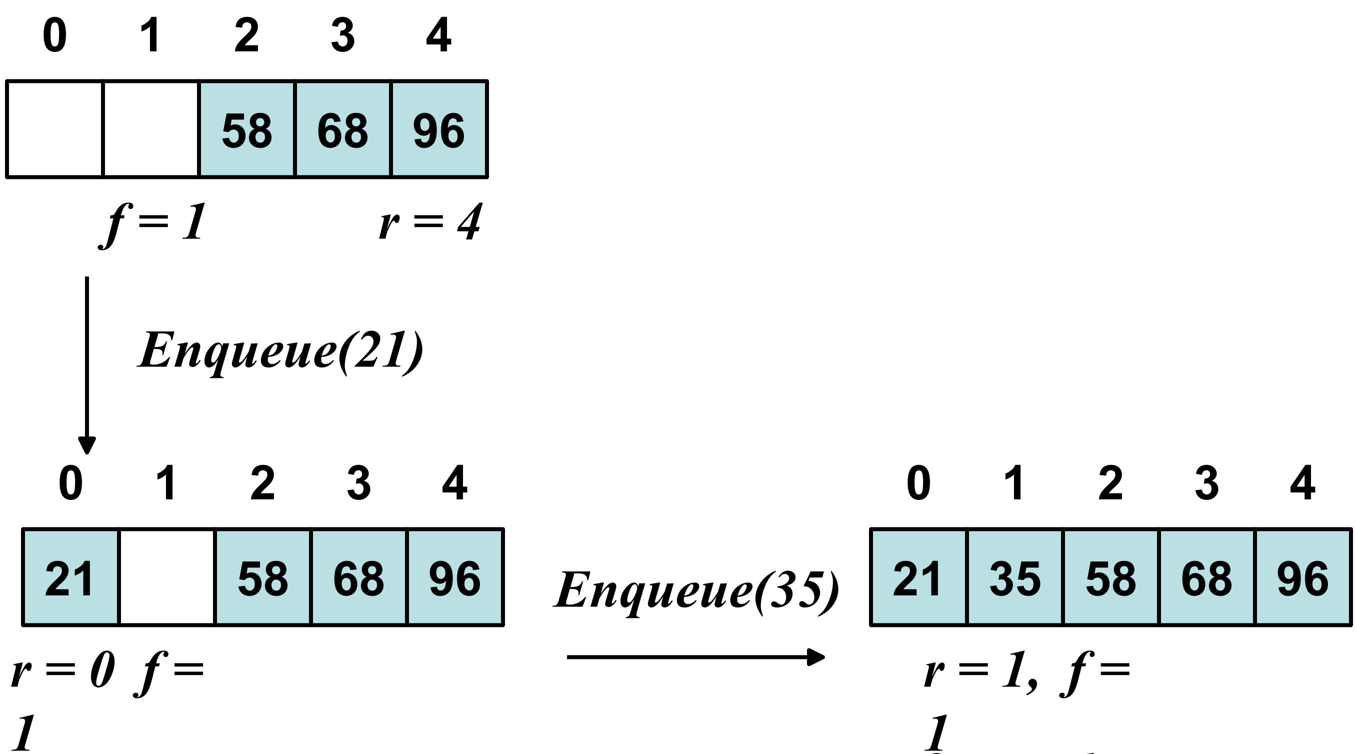
- Introduce a Boolean variable that will indicate whether the queue is full (or empty) or not, or an integer variable that counts the no of items in the queue
- Set one or both of the indices to some values that would otherwise never occur in order to indicate an empty (or full) queue.
  - For e.g., an empty queue could be indicated by setting the rear index to -1

The MakeEmpty and Empty Function

- For circular queue, we set both front and rear to MAXQUEUESIZE-1

```
void MakeEmpty(QueueType *pq)
{
     pq->front = MAXQUEUESIZE-1;
     pq->rear = MAXQUEUESIZE-1;
}
int Empty(QueueType *pq)
{
     return pq->rear == pq->front;
}
```

Why to sacrifice one element?



rear == front, but queue is not empty, so sacrifice one element

The Full Function

- The queue is full, if rear is just one position ahead of front

```
int Full(QueueType *pq)
{
     int newrear;
     newrear = (pq->rear+1)%MAXQUEUESIZE;
     return newrear == pq->front;
}
```

The Enqueue Function

```
void Enqueue(QueueType *pq, ItemType newitem)
{
      if (Full(pq)) {
            printf("Queue Full\n",);
            exit(1);
      }
      pq->rear = (pq->rear+1)%MAXQUEUESIZE;
      pq->items[pq->rear] = newitem;
}
```

The Dequeue Function

```
ItemType Dequeue(QueueType *pq)
{
      if (Empty(pq)) {
            printf("Queue is Empty\n",);
            exit(1);
      }
      pq->front = (pq->front+1)%MAXQUEUESIZE;
      return pq->items[pq->front];
}
```

Summary of Array Implementation of Queues

- A linear array with the front always in the first position and all entries moved up the array whenever the front is deleted
- A linear array with two indices always increasing
- A circular array with front and rear indices and one position left vacant
- A circular array with front and rear indices and either a Boolean variable to indicate fullness (or emptiness) or an integer variable counting entries
- A circular array with front and rear indices taking special values to indicate emptiness

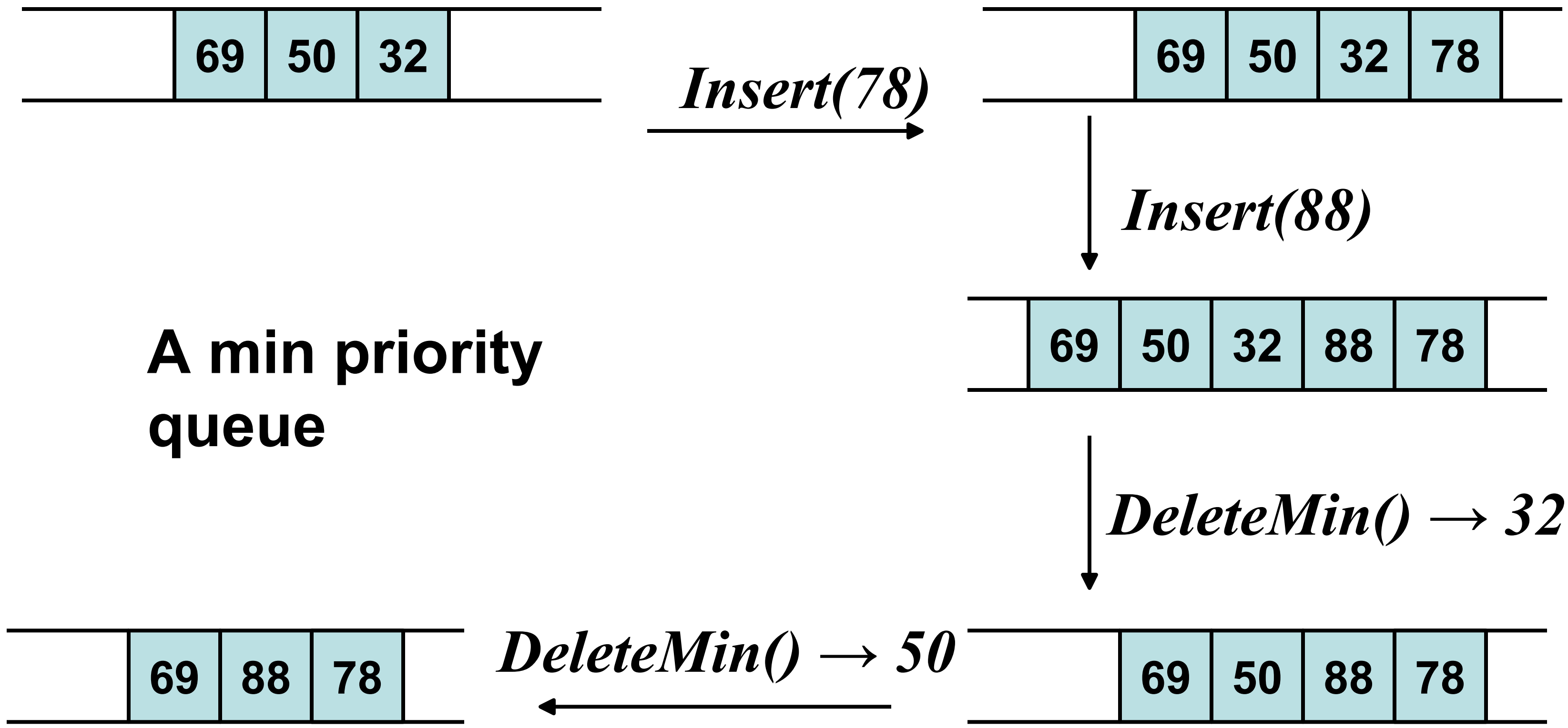
Priority Queues

- In stacks and queues, the elements are ordered are based on the sequence in which they have been inserted
- *A **priority queue** is a data structure in which the intrinsic ordering of the elements determines the results of its basic operations*

Types of Priority Queues

- ***Ascending (Min) Priority Queue***
  - *A collection of items into which items can be inserted arbitrarily but only the **smallest** item can be removed*
- ***Descending (Max) Priority Queue***
  - *A collection of items into which items can be inserted arbitrarily but only the **largest** item can be removed*

Illustration



A min priority queue

Applications of Priority Queue

- In a time-sharing computer system, a large number of tasks may be waiting for the CPU
- Some of these tasks have higher priority than others
- The set of tasks waiting for the CPU forms a priority queue

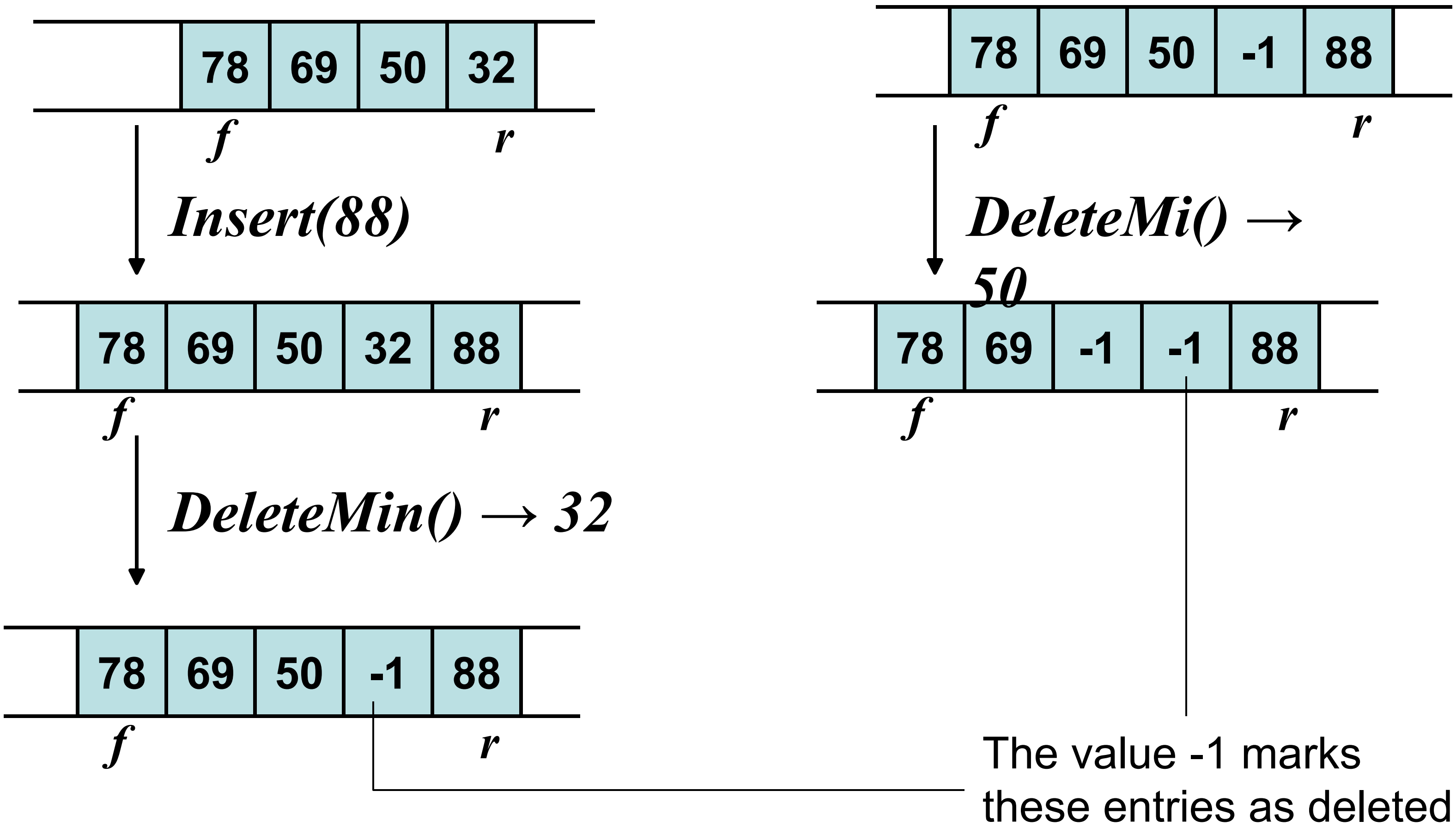
The Priority Queue ADT

- A (***min***) ***priority queue*** of elements of type ***T*** is a finite sequence of elements of ***T*** together with the operations
  - **MakeEmpty(*P*)** – Create an empty priority queue ***P***
  - **Empty(*P*)** – Determine if the priority queue ***P*** is empty or not
  - **Insert(*P*, *x*)** – Add element ***x*** on the priority queue ***P***
  - **DeleteMin(*P*)** – If the priority queue ***P*** is not empty, remove the minimum element of the queue and return it
  - **FindMin(*P*)** – Retrieve the minimum element of the priority queue ***P***

Array Implementation of Priority Queue

- Unordered Array Implementation
  - To insert an item, insert it at the rear end of the queue
  - To delete an item, find the position of the minimum element and
    - either mark it as deleted (lazy deletion) or
    - shift all elements past the deleted element by one position and then decrement rear

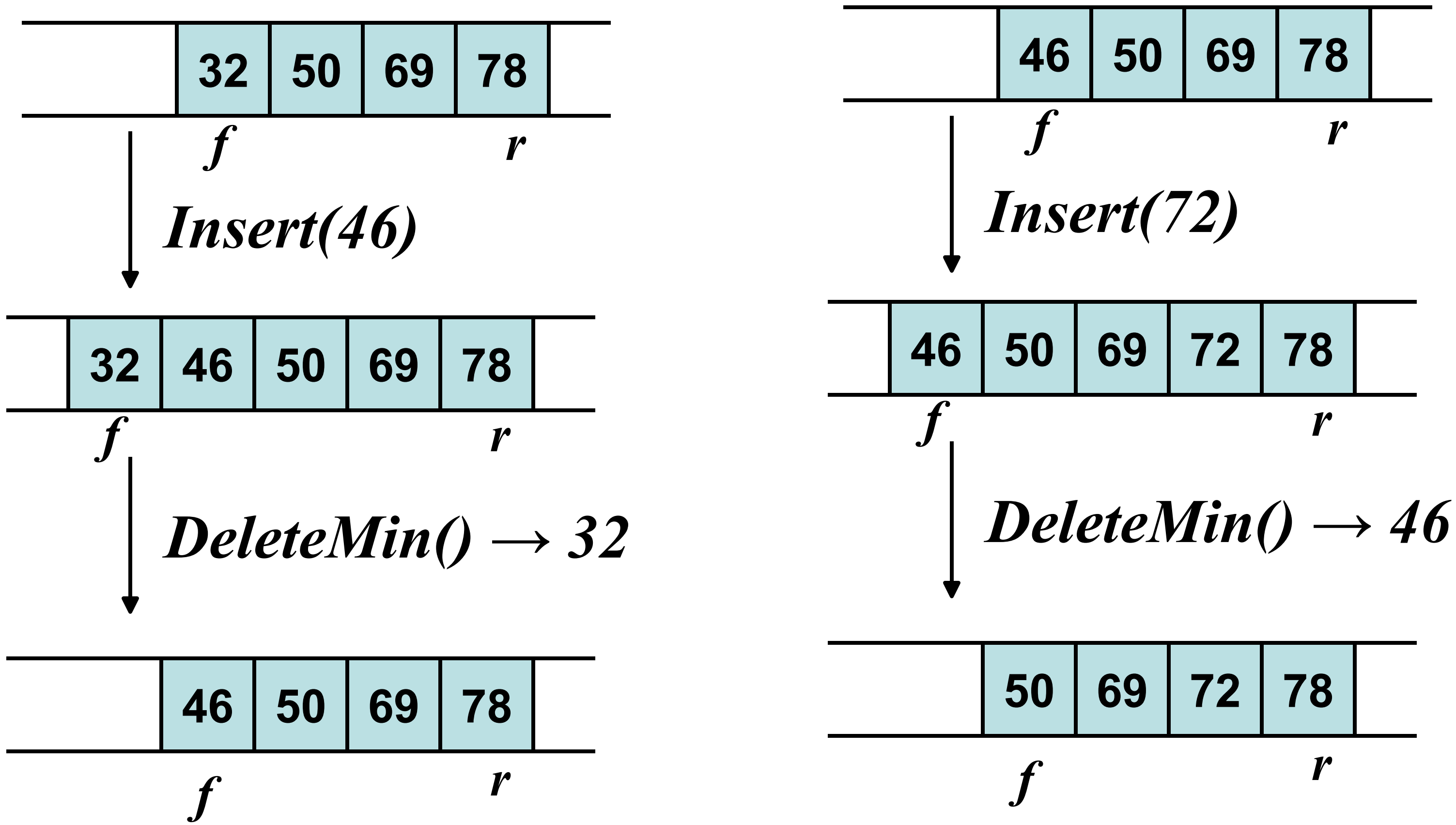
Illustration



Array Implementation of Priority Queue

- Ordered Array Implementation
  - Maintain the queue as a circular ordered array making the front as the position of the smallest element and the rear as the position of the largest element
  - To insert an element, locate the proper position of the new element and shift preceding or succeeding elements by one position
  - To delete the minimum element, increment the front position

Illustration



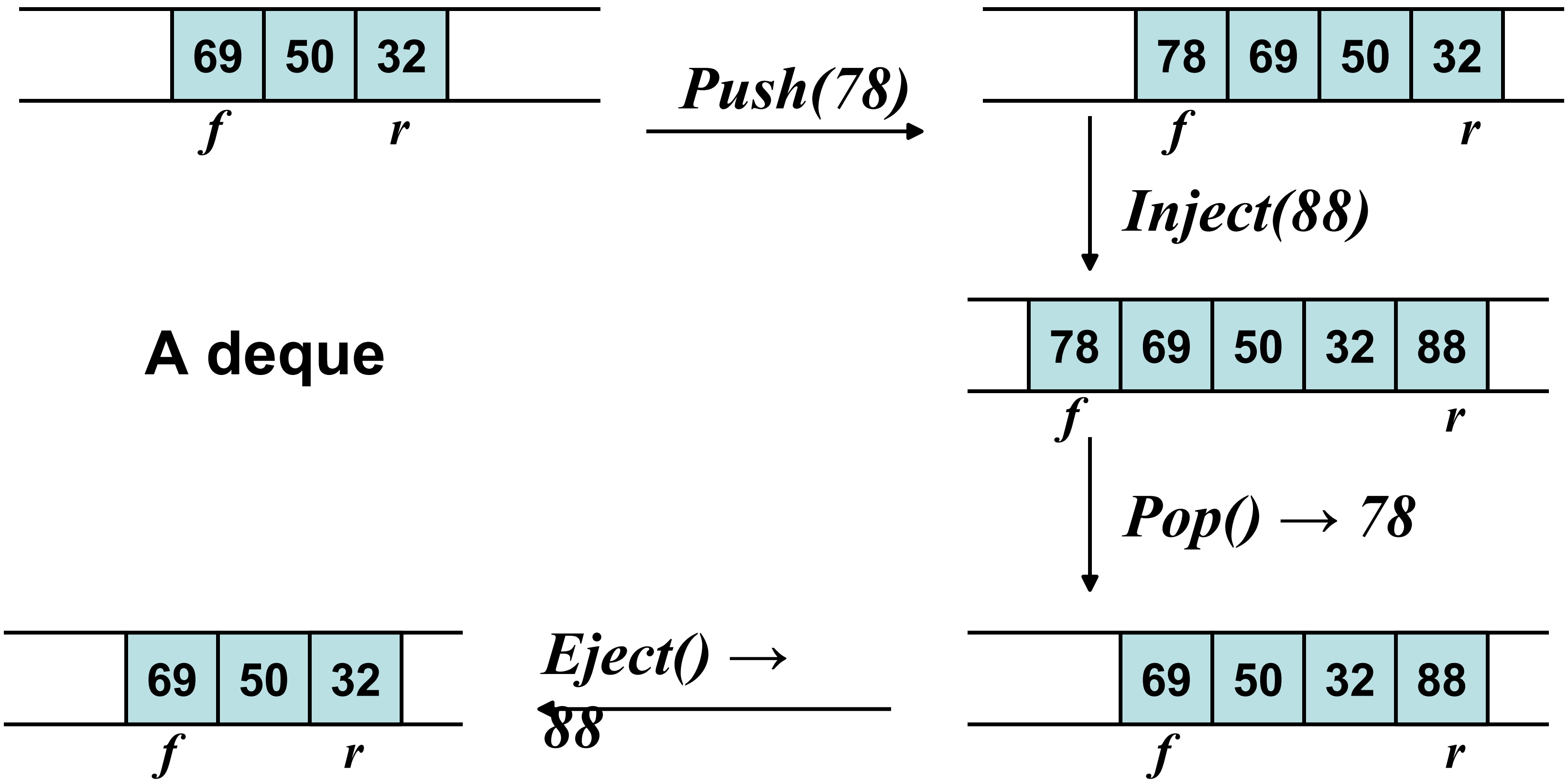
Notes on Array Implementation

- The array implementation of priority queue is not satisfactory
- Either the **Insert** operation or **DeleteMin** operation is proportional to ***n*** (***n*** is the queue size), i.e., $\boldsymbol{O(n)}$
- A heap allows to implement the Insert and DeleteMin in $\boldsymbol{O(\log n)}$ time

Deque

- A **deque** (*double-ended queue*) is a data structure consisting of a list of items, on which the following operations are possible
  - **Push(*D*, *x*)** – Insert item ***x*** on the front of deque ***D***
  - **Pop(*D*)** – Remove the front item from deque ***D*** and return it
  - **Inject(*D*, *x*)** – Insert item ***x*** on the rear end of deque ***D***
  - **Eject(*D*)** – Remove the rear item from deque ***D*** and return it

Illustration



The End