Data Structures – Week #4

Queues

Outline

- Queues
- Operations on Queues
- Array Implementation of Queues
- Linked List Implementation of Queues
- Queue Applications

Queues (Kuyruklar)

- A *queue* is a list of data with the restriction that
  1. *data can be inserted from the "rear" or "tail," and*
  2. *data can be retrieved from the "front" or "head"*

  of the list.
- By "*rear*" we mean a *pointer pointing to the element that is last added to the list whereas "front" points to the first element.*
- A *queue* is a *first-in-first-out (FIFO)* structure.

Operations on Queues

- *Two basic operations* related to queues:
  - ***Enqueue*** (*Put data to the rear of the queue*)
  - ***Dequeue*** (*Retrieve data from the front of the queue*)

Data Structures for Queues

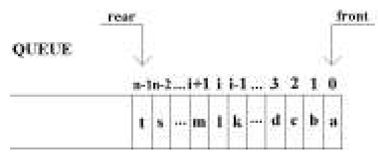
- Queues can be implemented using
  - *arrays*, or
  - *linked lists*.

Array Implementation of Queues

- Queues can be *implemented using arrays*.
- During the execution, queue can grow or shrink within this array. The array has *two "open" ends*.
- One end of the doubly-open-ended array is the *rear where the insertions are made*. The other is the *front where elements are removed*.

Array Implementation of Queues

- Initialization of an *n*-node queue:
  - ***front=0; rear=-1;***
- Condition for an empty queue:
  - In general: ***rear+1 = front***
  - In particular: ***rear = -1***;
- Condition for a full queue
  - In general: ***rear-(n-1) = front***;
  - In particular: ***rear ≥ n-1***;



Sample C Implementation

```
#define queueSize …;
struct dataType {
   …
}
typedef struct dataType dataType;
struct queueType {
   int front;
   int rear;
   dataType content[queueSize];
}
typedef struct queueType queueType;
queueType queue;
```

Sample C Implementation… isEmpty() and isFull()

```
//Initialize Queue (i.e., set value of front and rear to 0)
queue.rear=-1; queue.front=0;
int isEmpty(queueType q)
{
   return (q.rear < q.front);
}
int isFull(queueType q, int n)
{
   return (q.rear >= q.front + (n-1));
}
```


Enqueue() Operation

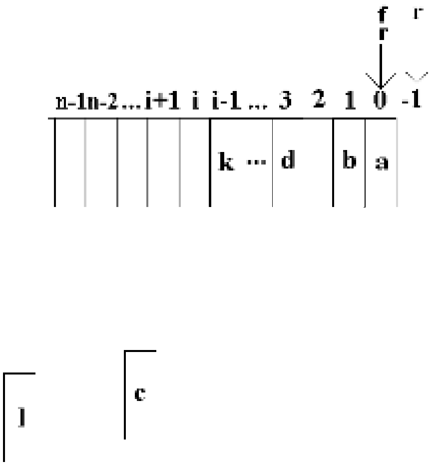
```
int enqueue(queueType *qp,int n,dataType item)
{
   if isFull(*qp,n) return 0;  //unsuccessful insertion
   (*qp).content[++(*qp).rear]=item;
   return 1;  //successful insertion
}
```

Running time of enqueue ***O(?)***

An ***<u>O(1)</u>*** operation

Enqueue Operation Animated

Empty Queue
a enqueued
b enqueued
c enqueued
d enqueued
…
k enqueued
l enqueued



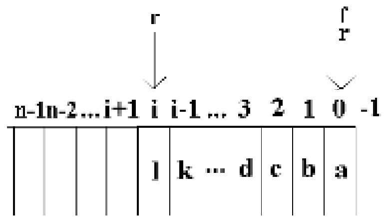
Dequeue Operation

```
int dequeue(queueType *qp,dataType *item)
{
   if isEmpty(*qp) return 0; //unsuccessful removal
   *item = (*qp).content[0]; // always: front = 0
   for (i=1; i <= (*qp).rear; i++)
         (*qp).content[i-1]= (*qp).content[i];
   (*qp).rear--;
   return 1;  //successful removal
} O(?)
```

An ***O(n)*** operation

O(n) Dequeue Operation Animated

a dequeued
b dequeued
c dequeued
d dequeued
…
k dequeued
l dequeued
Empty Queue



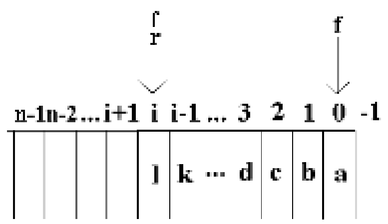
Improved Dequeue Operation

```
int dequeue(queueType *qp,dataType *item)
{
   if isEmpty(*qp) return 0; //unsuccessful removal
    *item = (*qp).content[(*qp).front++];
   return 1;  //successful removal
}
```

An ***O(1)*** operation

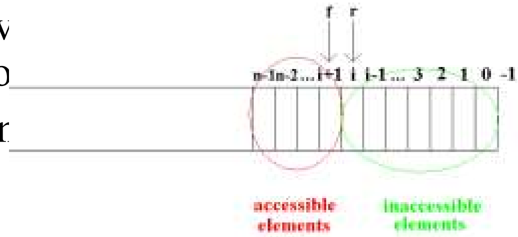
O(1) Dequeue Operation Animated

a dequeued
b dequeued
c dequeued
d dequeued
…
k dequeued
l dequeued
Empty Queue



Problem of *O(1)* Dequeue

- As *front* proceeds towards the larger indexed elements in the queue, we get *supposedly available but inaccessible array cells* in the queue (i.e., all elements with indices less than that pointed to by front).
- Whenev … nt, a shift operatio
- Solutior … ue to the start!!! … ***queues***.



Circular Queues

- Since with the existing conditions an empty and full circular queue is indistinguishable, we redefine the conditions for empty and full queue following a new convention:
- Convention: *front* points to the preceding cell of the cell with the data to be removed next.
- *Empty* circular queue condition: ***front=rear***
- *Full* queue condition: ***front=(rear+1) mod n***

Circular Queues (CQs)

```
//Initialize Queue (i.e., set value of front and rear to n-1)
queue.rear=n-1; queue.front=n-1; // i.e., -1 mod n
int isEmptyCQ(queueType cq)
{
   return (cq.rear == cq.front);
}
int isFullCQ(queueType cq, int n)
{
   return (cq.rear == (cq.front-1 % n));
}
```

Enqueue Operation in CQs

```
int enqueueCQ(queueType *cqp,dataType item)
{
  if isFullCQ(*cqp,n) return 0;//unsuccessful
   insertion
  (*cqp).content[++(*cqp).rear % n]=item;
   return 1;  //successful insertion
}
```

An ***O(1)*** operation

Dequeue Operation in CQs

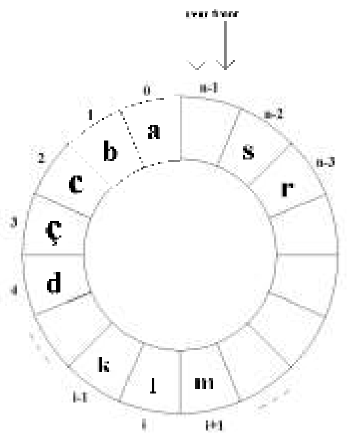
```
int dequeueCQ(queueType *cqp,dataType *item)
{
  if isEmptyCQ(*cqp) return 0;//unsuccessful removal
    *item = (*cqp).content[++(*cqp).front % n];
   return 1;  //successful removal
}
```

An ***O(1)*** operation

Circular Queues

```
int enqueueCQ(queueType *cqp,dataType item)
{
  if isFullCQ(*cqp) return 0;//unsuccessful insertion
  (*cqp).content[++(*cqp).rear%n]=item;
              return 1;  //successful insertion
}

int dequeueCQ(queueType *cqp,dataType *item)
{
 if isEmptyCQ(*cqp) return 0;//unsuccessful removal
 *item = (*cqp).content[++(*cqp).front%n];
 return 1;  //successful removal
}
```



Linked List Implementation of Queues

```
//Declaration of a queue node

Struct QueueNode {
   int data;
   struct QueueNode *next;
}
typedef struct QueueNode QueueNode;
typedef QueueNode * QueueNodePtr;
…
```

Linked List Implementation of Queues

```
QueueNodePtr NodePtr, rear, front;
…
…
NodePtr = malloc(sizeof(QueueNode));
rear = NodePtr;
front = rear;
NodePtr->data=2;                // or rear->data=2
NodePtr->next=NULL;             // or rear->next=NULL;
Enqueue(&rear,&NodePtr);
…
Dequeue( );
…
```


Enqueue and Dequeue Functions

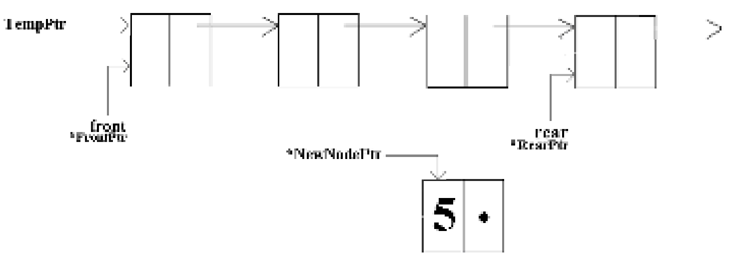
```
Void Enqueue (QueueNodePtr *RearPtr, QueueNodePtr *NewNodePtr) {
        *NewNodePtr = malloc(sizeof(QueueNode));
        (*NewNodePtr)->data=5;
        (*NewNodePtr)->next =NULL;
        (*RearPtr)->next=*NewNodePtr;
        *RearPtr = (*RearPtr)->next;
}

Void Dequeue(QueueNodePtr *FrontPtr) {
        QueueNodePtr TempPtr;
        TempPtr= *FrontPtr;
        *FrontPtr = (*FrontPtr)->next;
        free(TempPtr); // or you may return TempPtr!!!
}
```

Linked List Implementation of Queues

```
Void Dequeue(QueueNodePtr *FrontPtr) {
        QueueNodePtr TempPtr;
        TempPtr= *FrontPtr;
        *FrontPtr = (*FrontPtr)->next;
        free(TempPtr); // or return TempPtr!!!
}
```

```
Void Enqueue (QueueNodePtr *RearPtr,
        QueueNodePtr *NewNodePtr) {
        *NewNodePtr = malloc(sizeof(QueueNode));
        (*NewNodePtr)->data=5;
        (*NewNodePtr)->next =NULL;
        (*RearPtr)->next=*NewNodePtr;
       *RearPtr = (*RearPtr)->next;
}
```



Queue Applications

- All systems where a queue (a FIFO structure) is applicable can make use of queues.
- Possible examples from daily life are:
  - Bank desks
  - Market cashiers
  - Pumps in gas stations
- Examples from computer science are:
  - Printer queues
  - Queue of computer processes that wait for using the microprocessor

Priority Queues

- While a regular queue functions based on the arrival time as the only criterion as a FIFO structure, this sometimes degrades the overall performance of the system.
- Consider a printer queue in a multi-processing system where one user has submitted, say, a 200-page-long print job seconds before many users have submitted print jobs of only several pages long.
- A regular queue would start with the long print job and all others would have to wait. This would cause the average waiting time (AWT) of the queue to increase. AWT is an important measure used to evaluate the performance of the computer system, and the shorter the AWT, the better the performance of the system.

Priority Queues

- What may be done to improve the performance of the printer queue?
- Solution: Assign priority values to arriving jobs
- Then, jobs of the same priority will be ordered by their arrival time.

Priority Queues

- Assume a printer queue of jobs with three priorities, *a*, *b*, and *c*, where jobs with *a* (*c*) have the highest (lowest) priority, respectively.
- That is, jobs with priority *a* are to be processed first by their arrival times, and jobs of priority *c* last.

Priority Queues