International Islamic University Chittagong
Department of Computer Science & Engineering
Spring - 2021

Course Code: CSE-2321
Course Title: Data Structures

Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

Lecture – 9

Queues

Queues

A **Queue** is a linear data structure in which the insertion and deletion operations are performed at two different ends.

- In a queue data structure, the insertion operation is performed at a position which is known as **'rear'** and the deletion operation is performed at a position which is known as **'front'**.
- Queue **follows FIFO (First In First Out)** principle.

- The elements in a queue are processed like customers standing in a grocery check-out line: the first customer in line is the first one served.

Applications of Queues

There are numerous applications of queue in computer science.

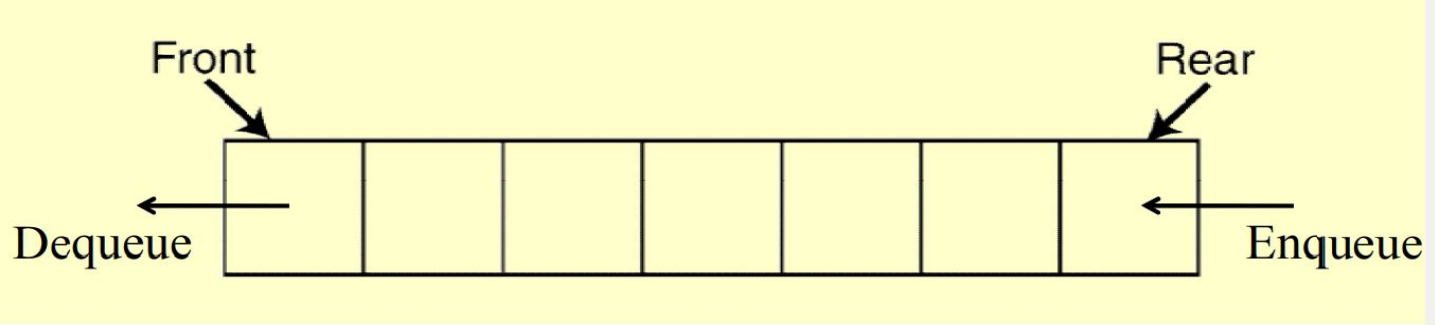
- Various real-life applications, like railway ticket reservation, banking system are implemented using queue.
- One of the most useful applications of queue is in simulation.
- Another application of queue is in operating system to implement various functions like CPU scheduling in multiprogramming environment, device management (printer or disk), etc.
- Communications software also uses queues to hold information received over networks and dialup connections. Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.
- Besides these, there are several algorithms like level-order traversal of binary tree, breadh-first search in graphs, etc., that use queues to solve the problems efficiently.

Representation of Queues

Queues may be represented in the computer in various ways, usually by means at one-way lists or linear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing .the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty.

Queue Operations

The two main operations of queue are insertion and deletion of items which are referred as **enqueue** and **dequeue** respectively. In *enqueue* operation, an item is **added** to the **rear** end of the queue. In *dequeue* operation, the item is **deleted** from the **front** end of the queue.



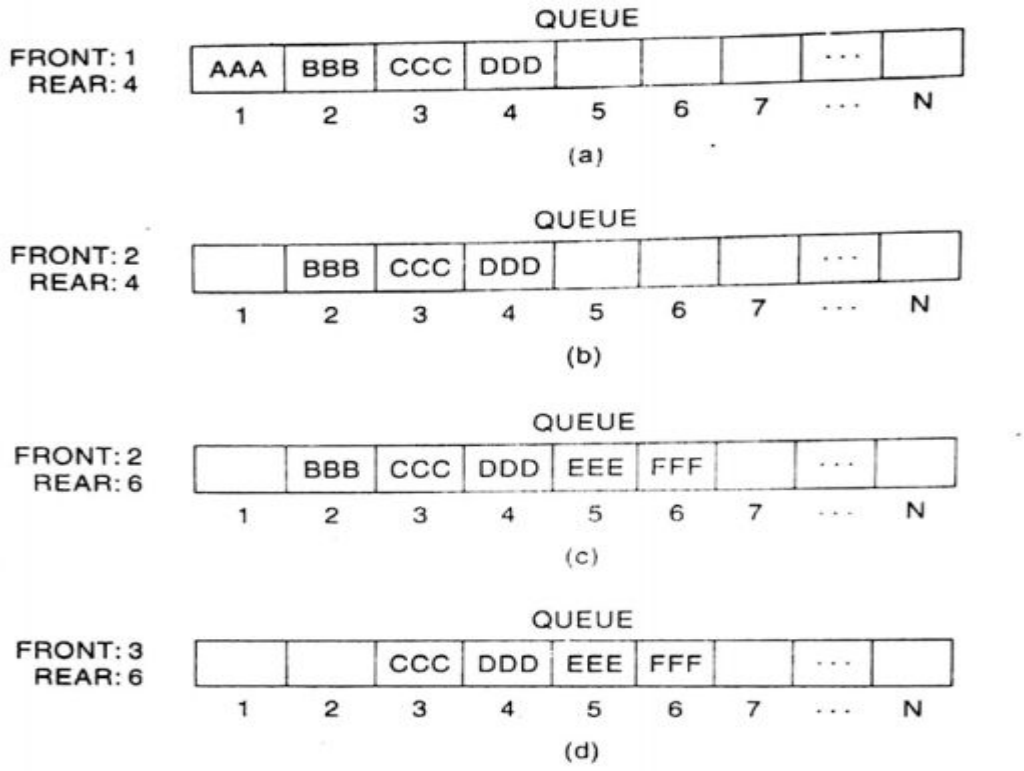
Insert at Rear End

To insert an item into the queue, first it should be verified whether the queue is full or not. If the queue is full, a new item cannot be inserted into the queue. The condition ***FRONT=NULL indicates that the queue is empty***. If the queue is not full, items are inserted at the rear end. When an item is added to the queue, the value of rear is incremented by 1 i.e. **REAR := REAR +1**

Queue Operations

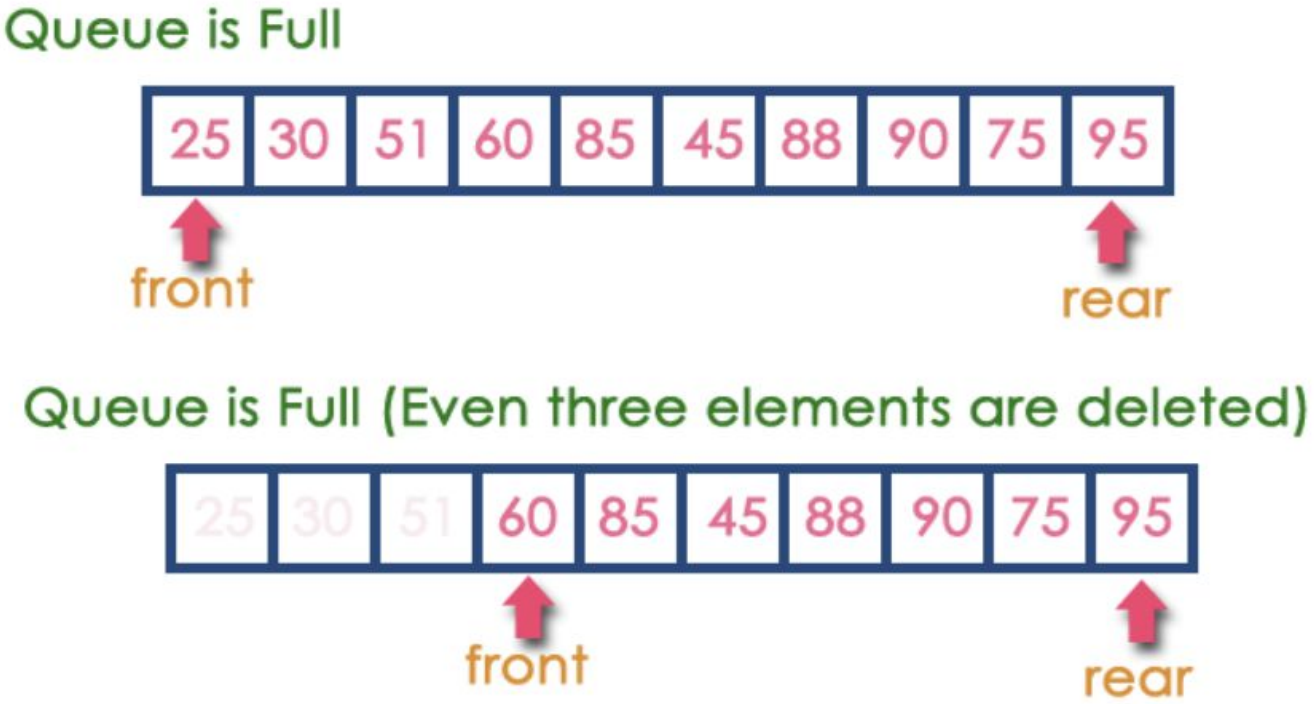
Delete from the Front End

To delete an item from the stack, first it should be verified that the queue is not empty. If the queue is not empty, the items are deleted at the front end of the queue. When an item is deleted from the queue, the value of the front is incremented by 1 i.e **FRONT := FRONT + 1**



Problem with Normal Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we can not insert the next element until all the elements are deleted from the queue

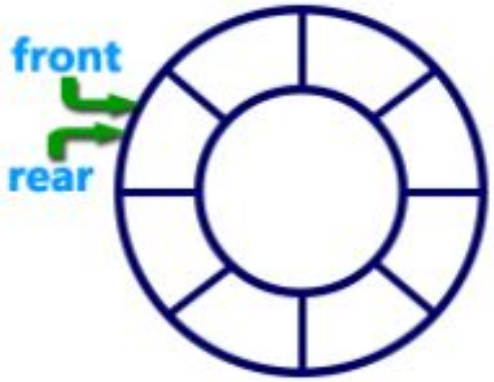


'rear' is still at last position. In above situation, *even though we have empty positions in the queue we can not make use of them to insert new element*.

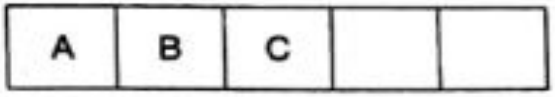
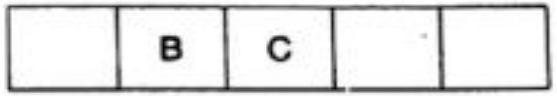
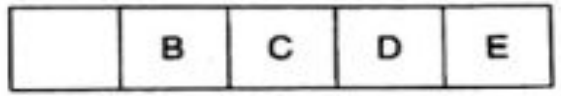
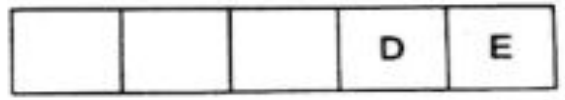
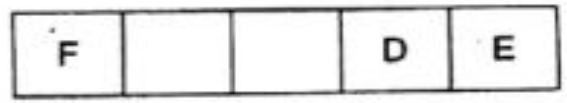
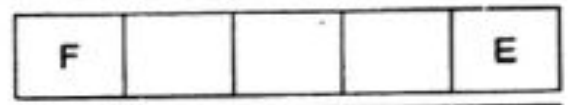
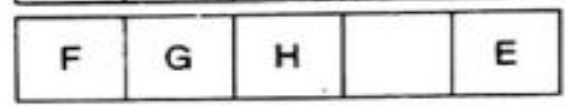
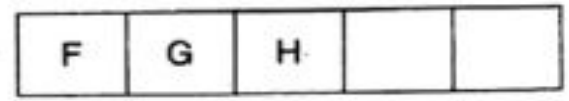
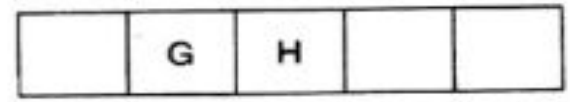
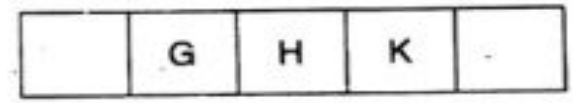
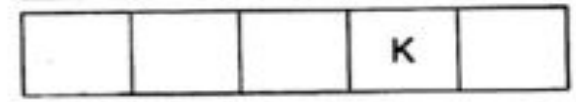
Circular Queue

A **Circular Queue** is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle, that is, **QUEUE[1] comes after QUEUE[N].**

Graphical representation of a circular queue is as follows:



- If REAR = N and an ITEM is inserted into the QUEUE when there is free space in the QUEUE, then we reset REAR = 1 instead of increasing REAR to N+1.
- Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N+1.

		QUEUE
(a) Initially empty:	FRONT: 0 REAR: 0	
		1 2 3 4 5
(b) A, B and then C inserted:	FRONT: 1 REAR: 3	
(c) A deleted:	FRONT: 2 REAR: 3	
(d) D and then E inserted:	FRONT: 2 REAR: 5	
(e) B and C deleted:	FRONT: 4 REAR: 5	
(f) F Inserted:	FRONT: 4 REAR: 1	
(g) D deleted:	FRONT: 5 REAR: 1	
(h) G and then H inserted:	FRONT: 5 REAR: 3	
(i) E deleted:	FRONT: 1 REAR: 3	
(j) F deleted:	FRONT: 2 REAR: 3	
(k) K inserted:	FRONT: 2 REAR: 4	
(l) G and H deleted:	FRONT: 4 REAR: 4	
(m) K deleted, QUEUE empty:	FRONT: 0 REAR: 0	

Queue Operations

QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
   Write: OVERFLOW, and Return.

2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
   Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
   Set REAR := 1.
   Else:
   Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.

: QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
   If FRONT = REAR, then: [Queue has only one element to start.]
   Set FRONT := NULL and REAR := NULL.
   Else if FRONT = N, then:
   Set FRONT := 1.
   Else:
   Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.

Linked Representation of Queues

In this section we discuss the linked representation of a queue. A linked queue is a queue implemented as a linked list with two pointer variables FRONT and REAR pointing to the nodes which is in the FRONT and REAR of the queue. The INFO fields of the list hold the elements of the queue and the LINK fields hold pointers to the neighboring elements in the queue. Fig. 6.22 illustrates the linked representation of the queue shown in Fig. 6.16(a).

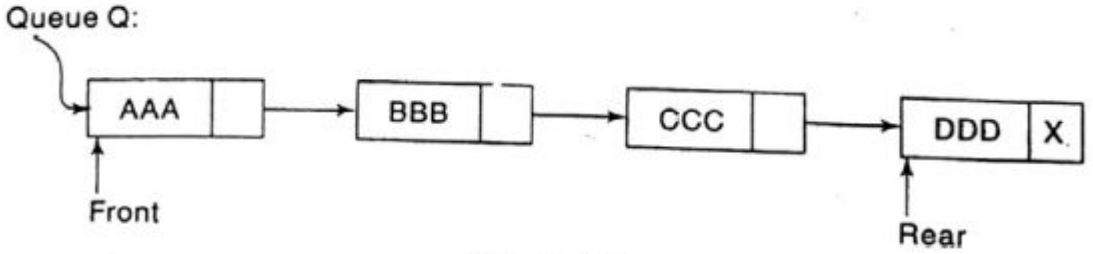


Fig. 6.22

Procedure 6.15: LINKQ_INSERT(INFO,LINK, FRONT, REAR,AVAIL,ITEM)
This procedure inserts an ITEM into a linked queue

1. [Available space?] If AVAIL = NULL, then Write OVERFLOW and Exit
2. [Remove first node from AVAIL list]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL]
3. Set INFO[NEW] := ITEM and LINK[NEW]=NULL
   [Copies ITEM into new node]

Linked Representation of Queues

4. If (FRONT = NULL) then FRONT = REAR = NEW
   [If Q is empty then ITEM is the first element in the queue Q]
   else set LINK[REAR] := NEW and REAR = NEW
   [REAR points to the new node appended to the end of the list]
5. Exit.

Procedure 6.16: LINKQ_DELETE (INFO, LINK, FRONT, REAR, AVAIL, ITEM)
This procedure deletes the front element of the linked queue and stores it in ITEM

1. [Linked queue empty?] if (FRONT = NULL) then Write: UNDERFLOW and Exit
2. Set TEMP = FRONT [If linked queue is nonempty, remember FRONT in a temporary variable TEMP]
3. ITEM = INFO (TEMP)
4. FRONT = LINK (TEMP) [Reset FRONT to point to the next element in the queue]
5. LINK(TEMP) =AVAIL and AVAIL=TEMP [return the deleted node TEMP to the AVAIL list]
6. Exit.

Example- 6.12 [Page- 6.38-6.39]

Deques

Deque (short form of Double-Ended Queue) is a linear list in which elements can be inserted or deleted at either end but not in the middle.

- Pronounced as “DECK” or dequeue

There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term “circular” comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array. Figure 6.26 pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

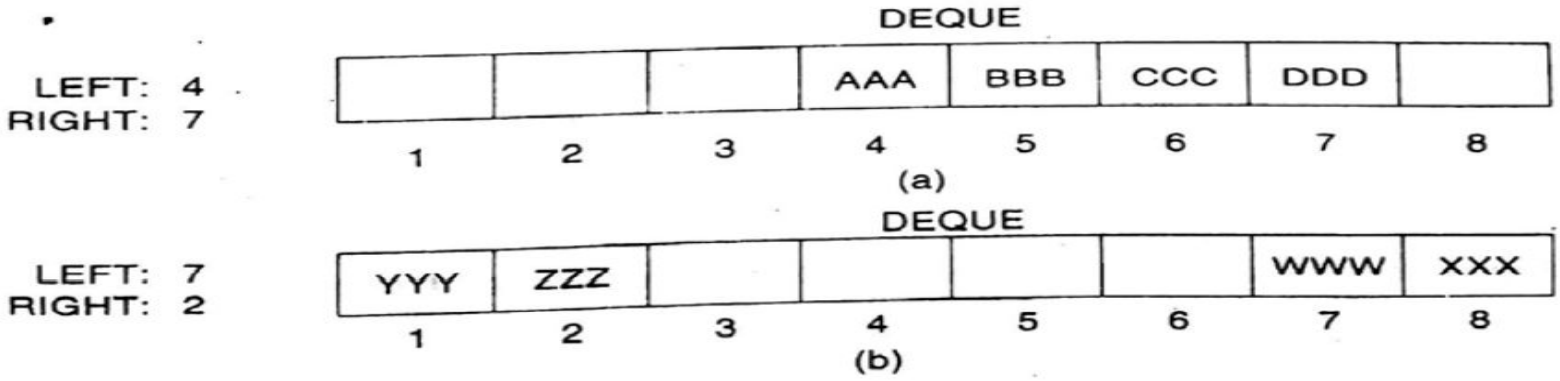


Fig. 6.26

Deques

There are two variations of a deque that are as follows:

1. **Input restricted deque**: It allows insertion of elements at one end only but deletion can be done at both ends.

2. **Output restricted deque**: It allows deletion of elements at one end only but insertion can be done at both ends.

Priority Queues

A **priority queue** is a type of queue in which each element is assigned a **priority** and such that the order in which elements are deleted and processed comes from the following rules :

1. The element with higher priority is processed before any element of lower priority.
2. The elements with the same priority are processed according to the order in which they were added to the queue.

- The elements are inserted at the rear.
- The elements are deleted according to the priority.
- The lower priority number indicates the higher priority.

Applications of Priority Queue:

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
3. All queue applications where priority is involved.

Array Representation of Priority Queues

A priority queue can be represented in many ways. Here we will discuss two dimensional Array representation of priority queue.

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays. Figure 6.30 indicates this representation for the priority queue in Fig. 6.29. Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Array Representation of Priority Queues

The followings are outlines of algorithms for deleting and inserting elements in a priority queue that is maintained in memory by a two-dimensional array QUEUE..

Algorithm 6.19: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm 6.20: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

One-way List Representation of Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows:

(a) Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

(b) A node X precedes a node Y in the list (1) when X has higher priority than Y or (2) when both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Priority numbers will operate in the usual way: the lower the priority number, the higher the priority.

Example 6.13

Figure 6.27 shows a schematic diagram of a priority queue with 7 elements. The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list. Figure 6.28 shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK. (See Sec. 5.2.)

One-way List Representation of Priority Queue

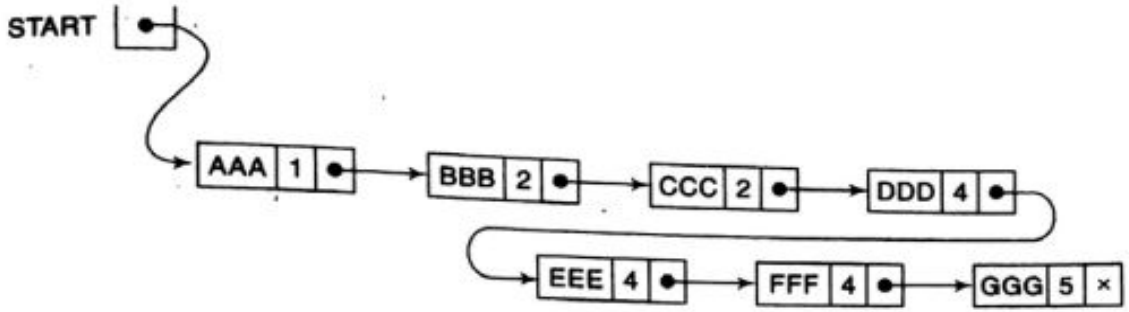


Fig. 6.27

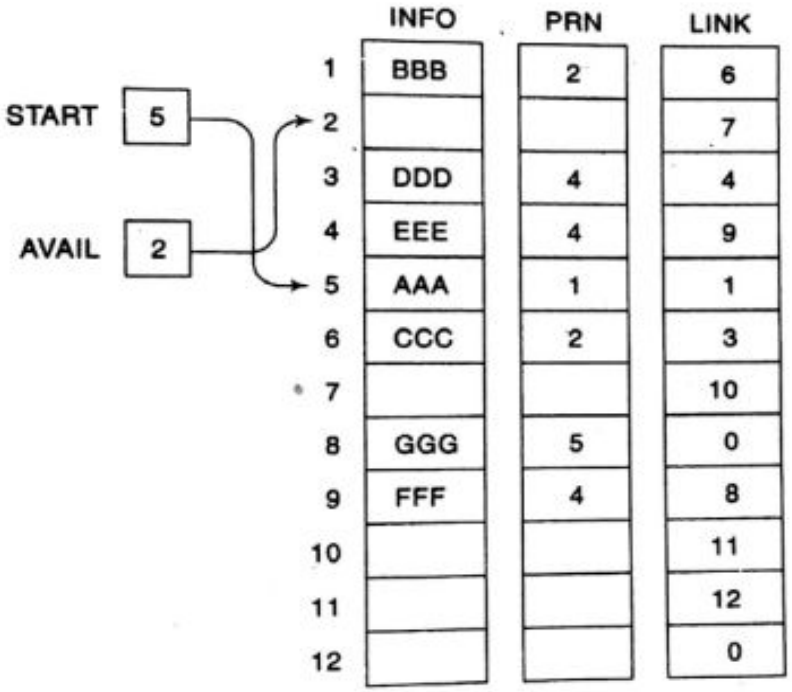


Fig. 6.28

One-way List Representation of Priority Queue

Algorithm 6.17: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM := INFO[START]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Adding an element to our priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element. An outline of the algorithm follows.

Algorithm 6.18: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

(a) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.
(b) If no such node is found, insert ITEM as the last element of the list.

One-way List Representation of Priority Queue

processes the first element in a priority queue

Example 6.14

Consider the priority queue in Fig. 6.27. Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers. Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig. 6.29. Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the list. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

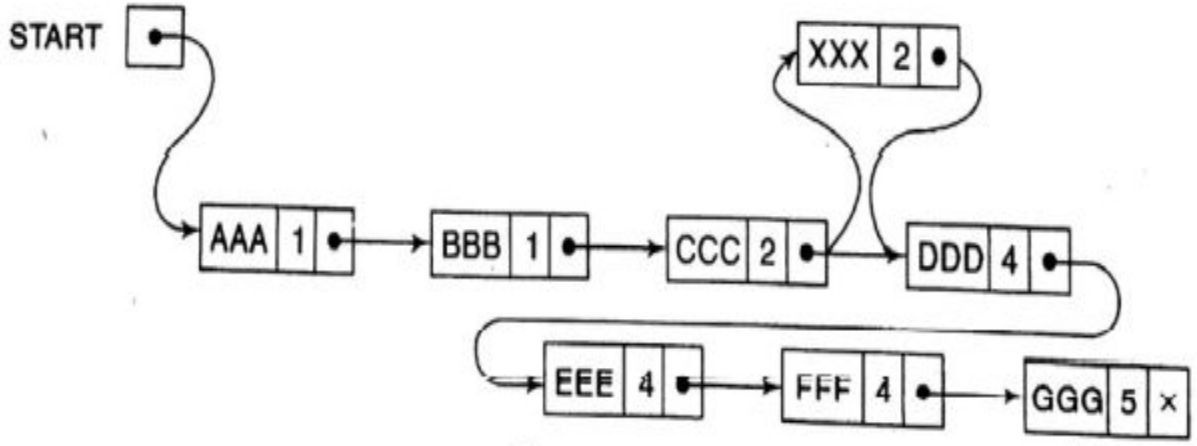


Fig. 6.29

Comparison between Array & One-way List Representation of Priority Queue

Once again we see the time-space tradeoff when choosing between different data structures for a given problem. The array representation of a priority queue is more time-efficient than the one-way list. This is because when adding an element to a one-way list, one must perform a linear search on the list. On the other hand, the one-way list representation of the priority queue may be more space-efficient than the array representation. This is because in using the array representation, overflow occurs when the number of elements in any single priority level exceeds the capacity for that level, but in using the one-way list, overflow occurs only when the total number of elements exceeds the *total* capacity. Another alternative is to use a linked list for each priority level.