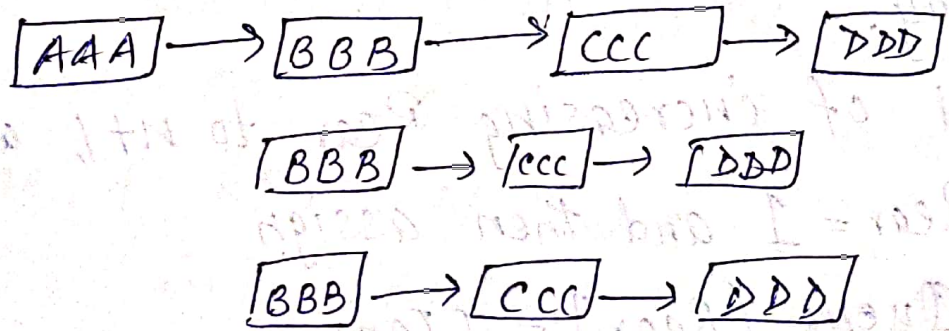
Queues :→ A queue is a linear list of elements in which deletions can take place only at one end called the front and insertions can take place only at the other end called the rear. The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

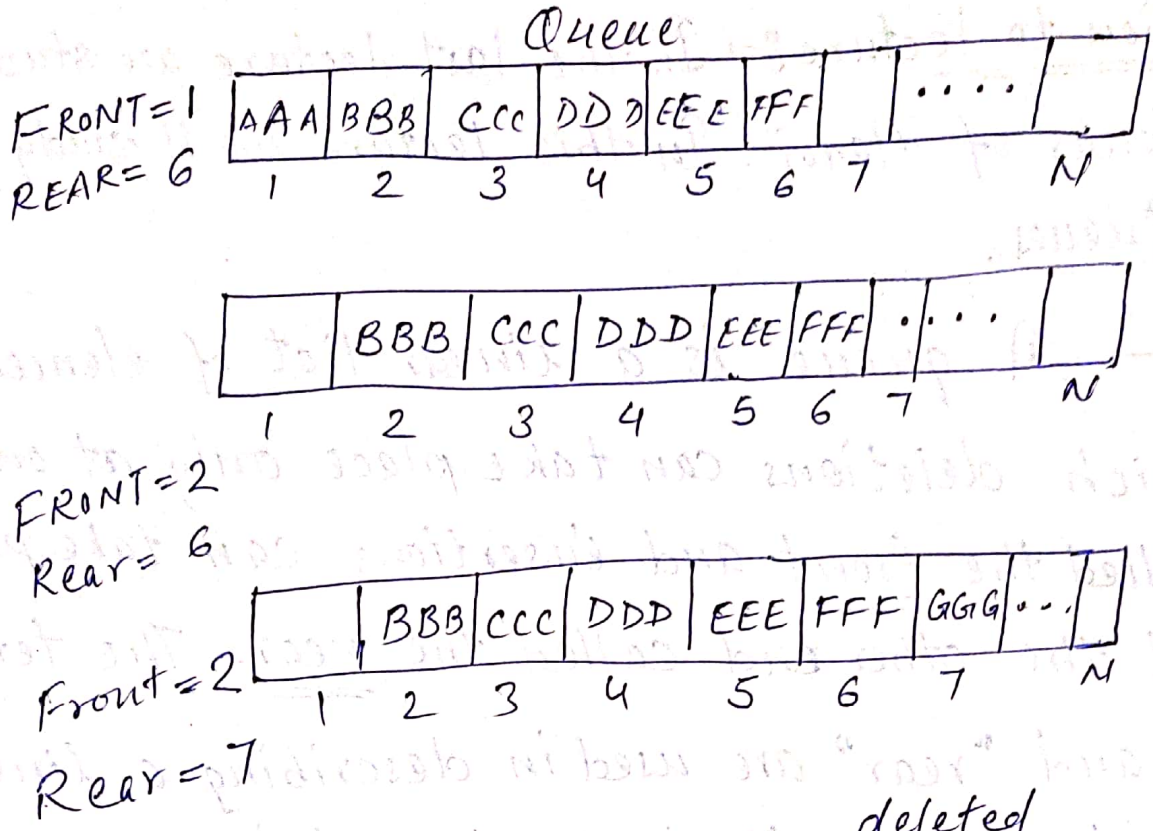
↳ Queues are also called first-in-first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue.

Representation of Queues :→

↳ Queues will be maintained by a linear array Queue and two pointer variables : FRONT & REAR.

↳ The cond^{n} FRONT = NULL will indicate that the queue is empty.





↳ Whenever an element is ~~deleted~~ deleted from the queue, the value of FRONT is increased by 1 i.e.

FRONT := FRONT + 1.

↳ Whenever an element is added to the queue, the value of REAR is increased by 1;

this is :- REAR := REAR + 1.

↳ We assume that the array ~~Queue~~ Queue is circular that is Queue[1] comes after Queue[N] in the array.

∴ instead of increasing Rear to n+1, we reset Rear = 1 and then assign

Queue [Rear] := Item.

Similarly if Front=N and an element of Queue is deleted, we reset Front=1 instead of increasing Front to n+1.

↳ When queue contains only one element i.e.

Front = Rear ≠ Null

& suppose that the element is deleted, Then we assign

Front := Null and Rear := Null

to indicate that the queue is empty.

Procedure :→ 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.

Procedure :- QDELETE (QUEUE, N, FRONT, REAR, ITEM)

This procedure deletes an element from a queue and assigns it to the variable ITEM.

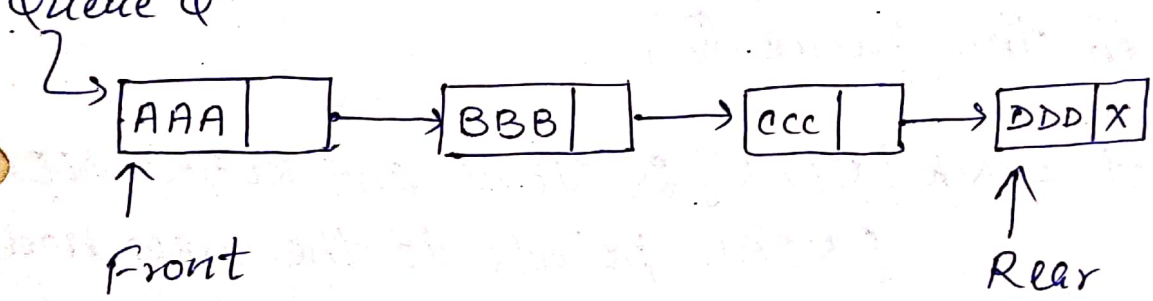
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW and Return.
2. Set ITEM := QUEUE[FRONT]
3. [Find new value of FRONT.]
   If FRONT = REAR, then: [Queue has only one element to start]
   Set FRONT := NULL and REAR := NULL
   Else if FRONT = N, then:
   Set FRONT := 1
   Else:
   Set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

Introduction to Lecture:→ In the last lecture we studied about implementation of queues. In this lecture we'll study about: Linked Representation of Queues.

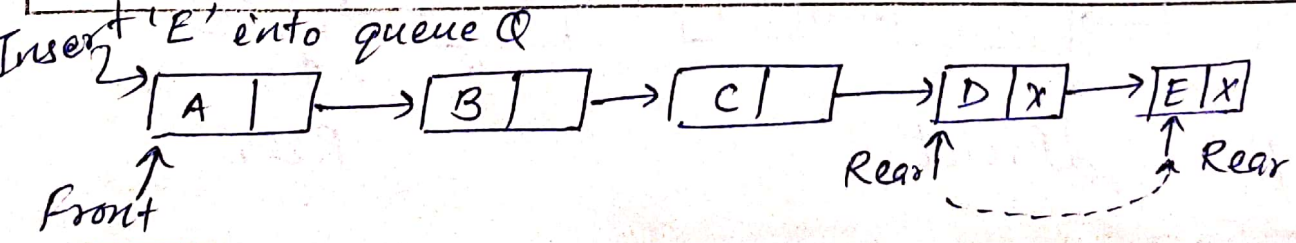
Linked Representation of Queues:→

↳ A linked queue is a queue implemented as a linked list with two pointer variables Front & Rear.

↳ 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.



↳ In the case of insertion into a linked queue, a node borrowed from the AVAIL list and carrying the item to be inserted is added as the last node of the linked list representing the queue.



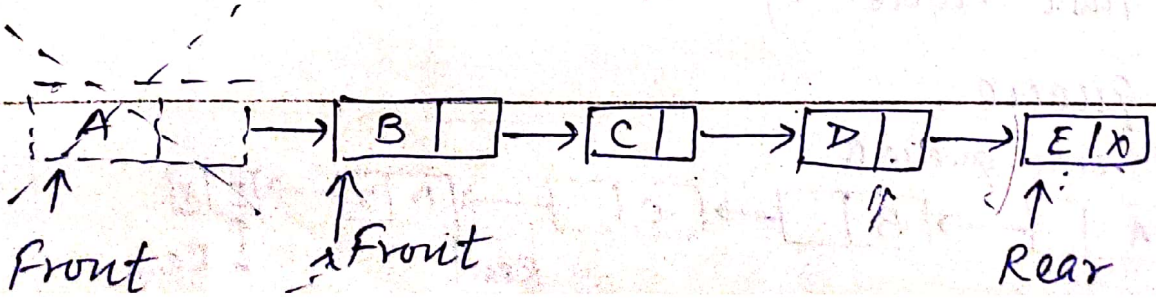
Insertion:→

Procedure:→ 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]
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 list]
5. Exit.

Deletion in Linked Queue:→



Procedure:- 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]
   (TEMP ↓ node.)
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.

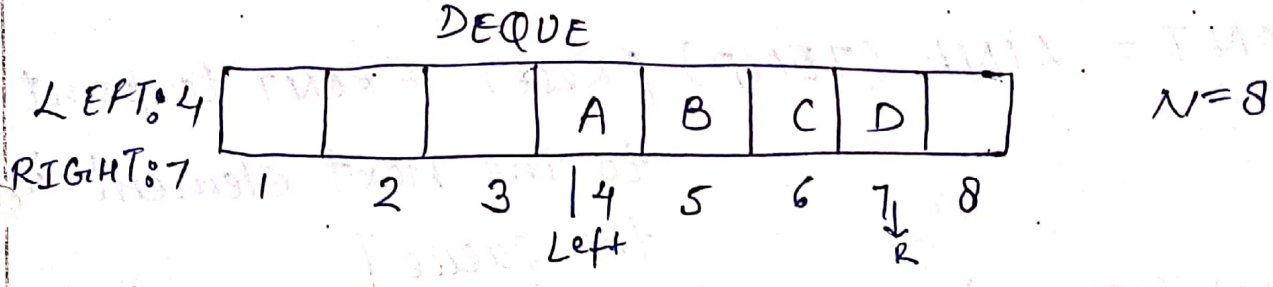
Introduction to Deques :→ [Double Ended Queue].

A deque (either deck or dequeue) is a linear list in which elements can be added or removed at either end but not in the middle.

Representation of a deque in a computer :→

We'll assume our deque is maintained in memory by a circular array DEQUE with pointers LEFT & RIGHT, which point to the two ends of the queue.

↳ The term "Circular" indicates that DEQUE[1] comes after DEQUE[N] in the array.

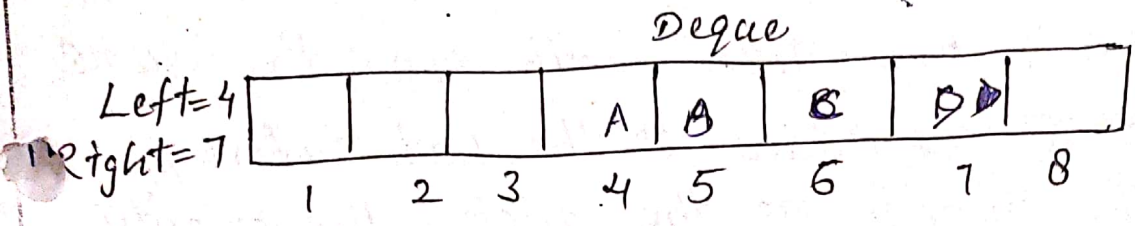


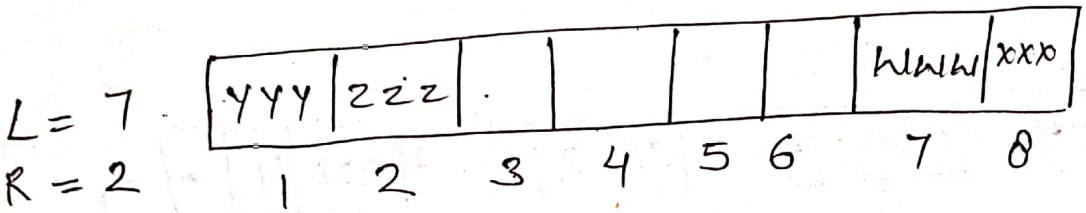
When LEFT = NULL, queue is empty.

* There are two types of deques →

① Input-restricted deque :→ which allows insertions at only one end of the list but allows deletions at both ends of the list.

(2) Output-restricted deque :→ which allows deletions at only one end of the list but allows insertions at both ends of the list.





* Priority Queues :→ A priority Queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules :

(1) An element of higher priority is processed before any element of lower priority.

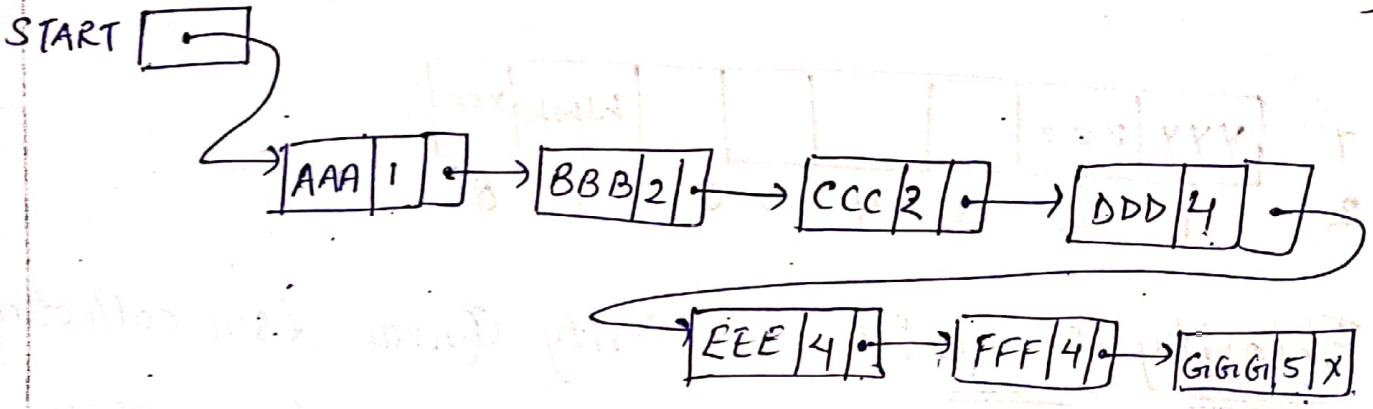
(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

One-Way list Representation of a Priority Queue:→

↳ Each node in the list will contain three items of information: an info^m field INFO, a priority no. PRN and a link no.

↳ X precedes a node Y, when X has higher priority than Y or when both has the same priority but X was added to the list before Y.

↳ the lower the priority no, the higher the priority



↳ Deletion in priority Queue:→

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].
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Addition :→ 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.

Algo^m :- This algorithm adds an ITEM with priority no. N to a priority queue which is maintained in memory as a one-way list:-

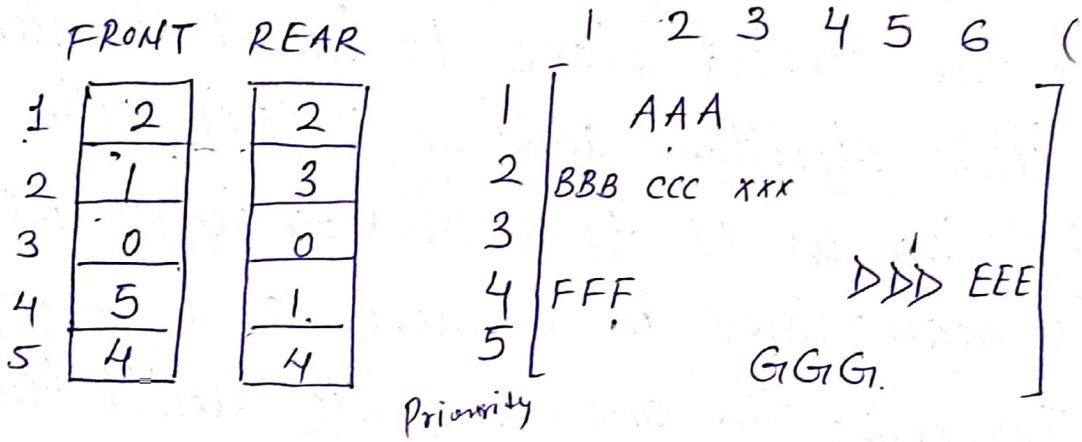
(a) Traversing the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

(b) If no such node is found, insert ITEM as the last element of the list.

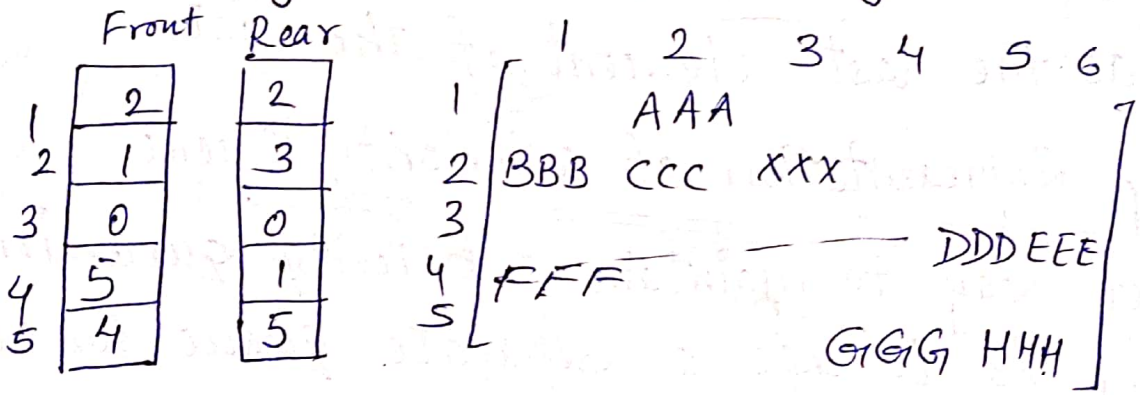
Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority no.). Each such queue will appear in its own circular array and must have its own pair of pointers.

FRONT and REAR. A two-dimensional array QUE... can be used instead of the linear arrays. Here 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.



Now suppose one more element HHH is inserted in the priority Queue having priority 5. then changes in the matrix ~~given~~ areas:-



Algo^m of deletion:- This algo^m deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

Step 1:- [find the first non-empty queue]
Find the smallest K such that FRONT[K] ≠ NUL

Step 2:- Delete and process the front elemen in row K of QUEUE.

Step 3:- Exit.

Algo^m of Addition:→ This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEU

Step 1:- Insert ITEM as the rear element in row M of QUEUE.

Step 2:- Exit.

Representation of Linked Lists in memory :→

Let LIST be a linked list. Then LIST will be maintained in memory.

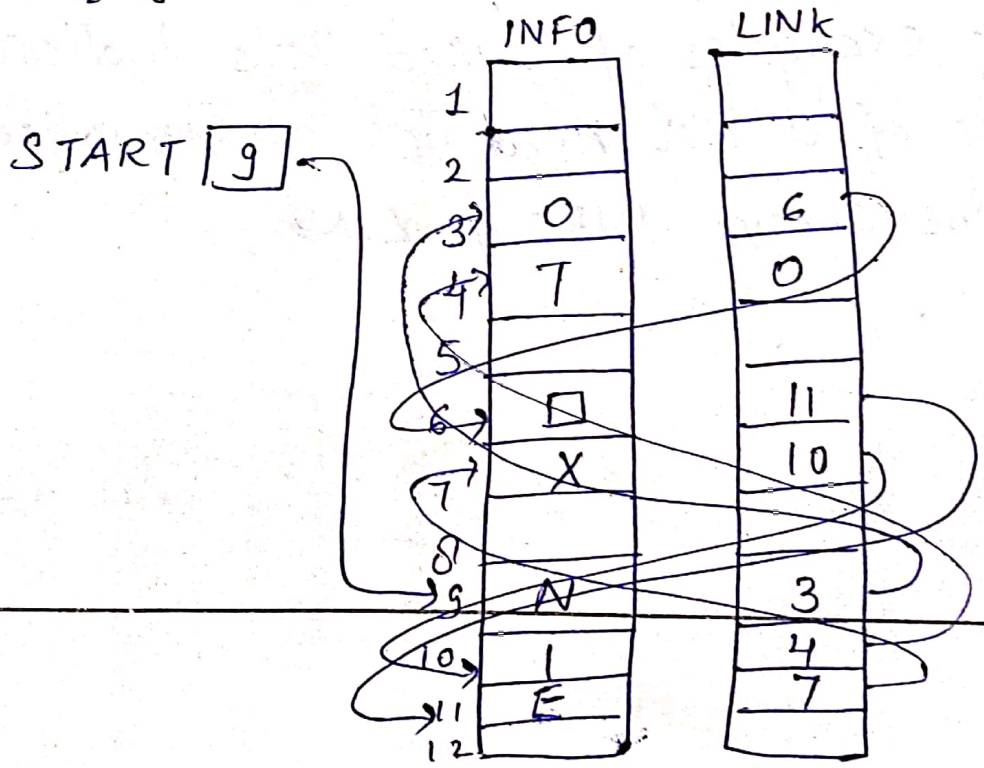
↳ LIST requires two linear arrays., they are :- INFO[K] and LINK[K] i.e. the information part & the nextpointer field of a node of LIST.

↳ LIST also requires a variable name - suchas START - which contains the location of the beginning of the list and a next pointer sentinel - denoted by NULL - which indicates the end of the list.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays INFO & LINK.

The following figure pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters or in other words, the string as follows:-

START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = O is the second char^c.
LINK[3] = 6, so INFO[6] = □ (blank) is the 3rd char^c.
LINK[6] = 11, so INFO[11] = E is the fourth char^c.
LINK[11] = 7, so INFO[7] = X is the fifth char^c.
LINK[7] = 10, so INFO[10] = I is the sixth char^c.
LINK[10] = 4, so INFO[4] = T is the seventh char^c
LINK[4] = 0, the NULL value, so the list has ended.



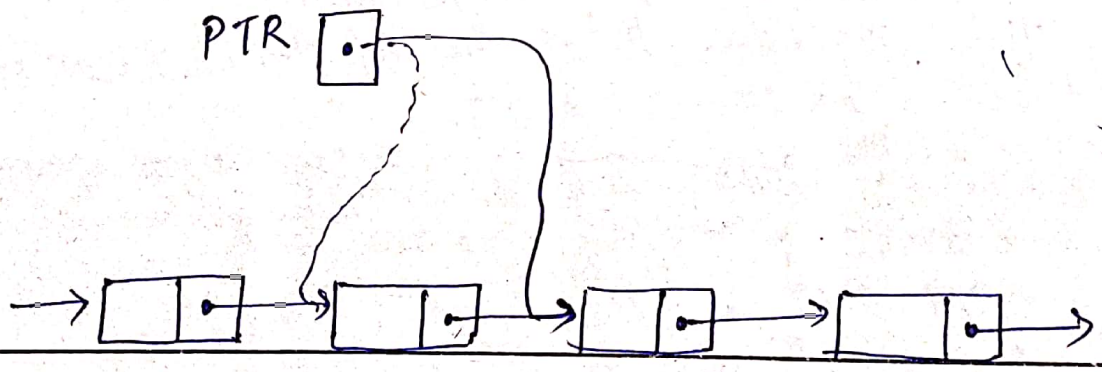
"NO EXIT"
Character string

Traversing a Linked List :→ Let LIST be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST. Suppose we want to traverse LIST in order to process each node exactly once.

↳ Traversing algo^m uses a pointer variable PTR which points to the node that is currently being processed. LINK[PTR] points to the next node to be processed. Thus the assignment

PTR := LINK[PTR]

moves the pointer to the next node in the list.



PTR := LINK[PTR]

Algorithm :→ (Traversing a Linked List) Let LIST be a linked list in memory. This algo^m traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR := START [Initializes pointer PTR]
2. Repeat Steps 3 & 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR]
4. Set PTR := LINK[PTR] [PTR now points to the next node]

   [End of Step 2 loop]
5. Exit

Algo^m for printing info^m at each node.

Procedure :- PRINT (INFO, LINK, START)

This procedure prints the info^m at each node of the list.

1. Set PTR := START
2. Repeat Steps 3 & 4 while PTR ≠ NULL
3. Write : INFO[PTR]
4. Set PTR := LINK[PTR] [updates ptr]

   [End of Step 2 loop]
5. Return.

Introduction :→ In the last lectures we studied about sorting & creating of linked lists. In this lecture we will study about Insertion & deletion into a linked list.

Overflow and Underflow :→

Sometimes new data are to be inserted into a data structure but there is no available space i.e. the free storage list is empty. This situation is usually called overflow. The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays.

↳ Overflow will occur with our linked lists when AVAIL = NULL & there is an insertion.

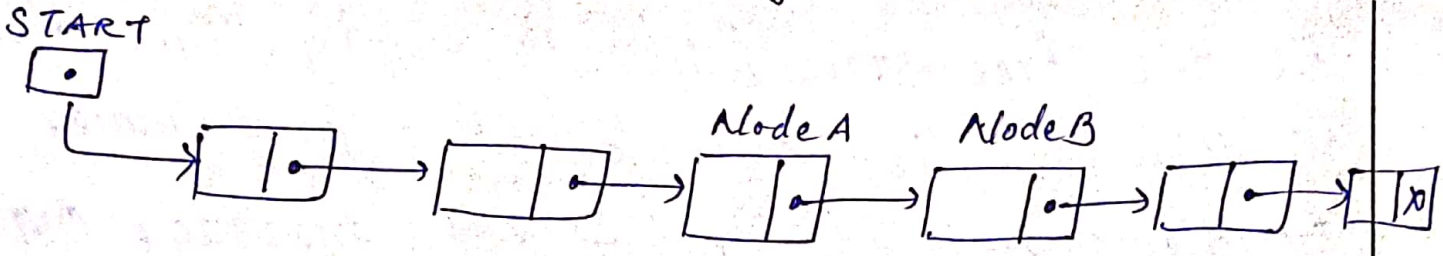
↳ The term underflow refers to the situation where one wants to delete data from a data structure that is empty.

↳ The programmer may handle ~~over~~ underflow by printing the message UNDERFLOW.

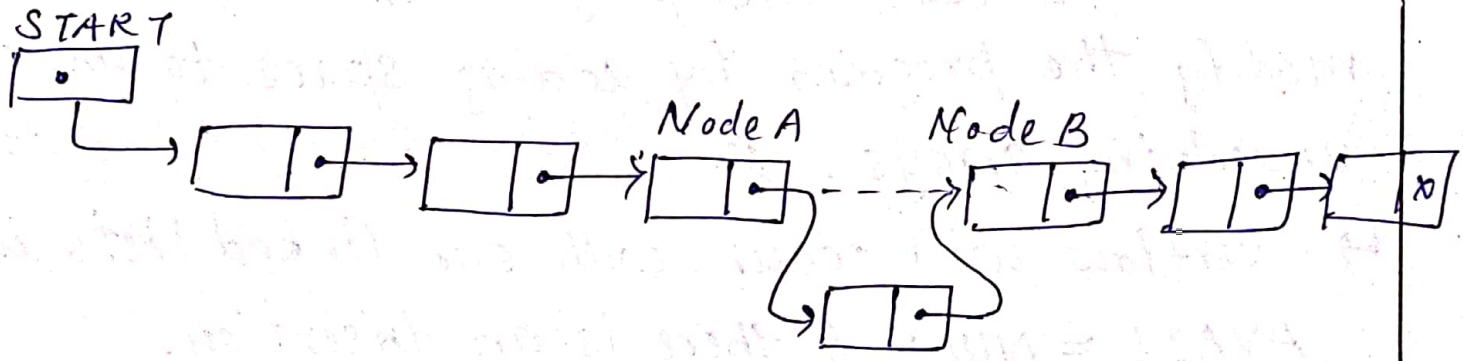
↳ UNDERFLOW will occur with our linked lists when START = NULL & there is a deletion.

Insertion into a Linked List :→

Let LIST be a linked list with successive nodes A and B, as pictured in Fig.



(a) Before insertion.



(b) After insertion.

↳ Suppose a node N is to be inserted into the list between nodes A and B.

→ That is, node A now points to the new Node N, and node N points to node B, to which A previously pointed.

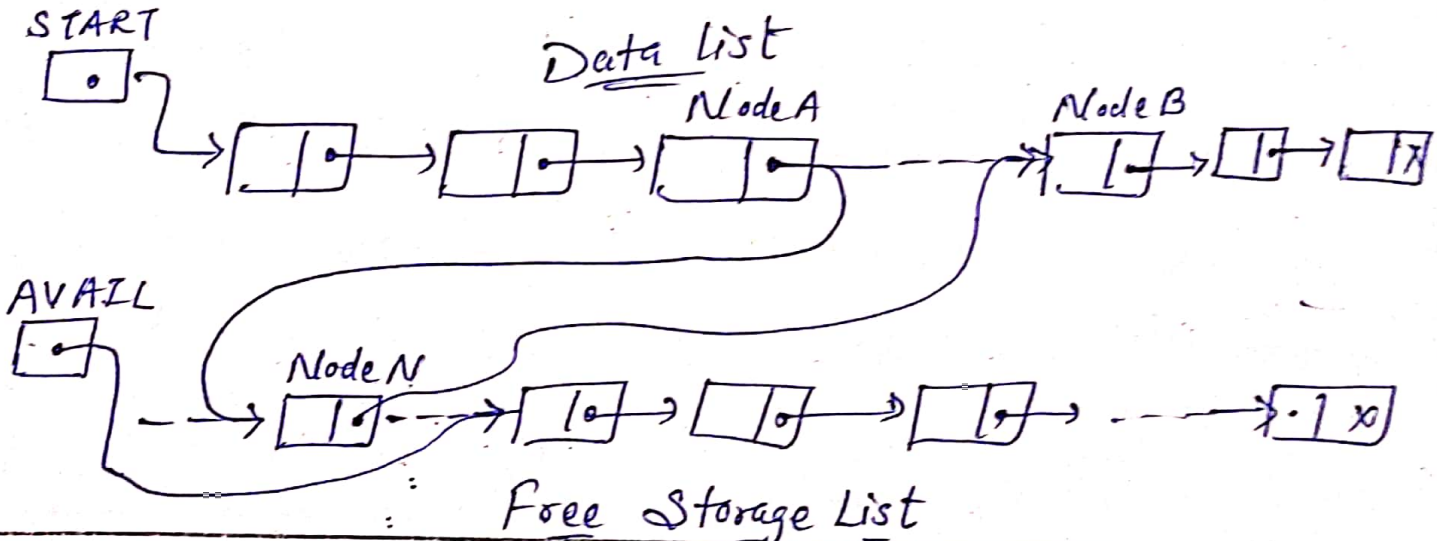
Suppose our linked list is maintained in memory in the form.

LIST (INFO, LINK, START, AVAIL)

↳ The memory space for the new node N will come from the AVAIL List.

↳ For easier processing, the first node in the AVAIL list will be used for the new node N. Thus a more exact schematic diagram is as :-

(1) The next pointer field of node A now points to the new node N, to which AVAIL previously pointed.



(2) AVAIL now points to the second node in the free pool, to which node N previously pointed.

(3) The nextpointer field of node N now points to node B, to which node A previously pointed.

Insertion Algorithms :→ contain various situations.

3 of them are :-

(1) inserts node at the beginning of the list.

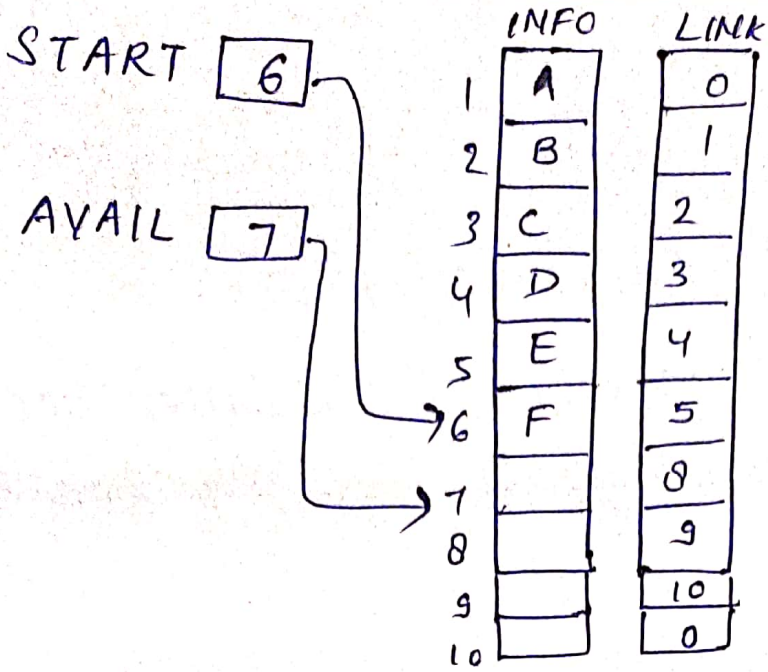
(2) one inserts a node after the node with a given location

(3) inserts a node into a sorted list.

Linked list in memory in the form

LIST(INFO, LINK, START, AVAIL)

and that the variable ITEM contains the new info to be added to the list.



↳ Since our insertion algorithms will use a node in the AVAIL List, all algo^m will use following steps :-

(a) Checking to see if space is available in the AVAIL list. If not i.e. if AVAIL = NULL then the algo^m will print the msg OVERFLOW.

(b) Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this can be implemented as:

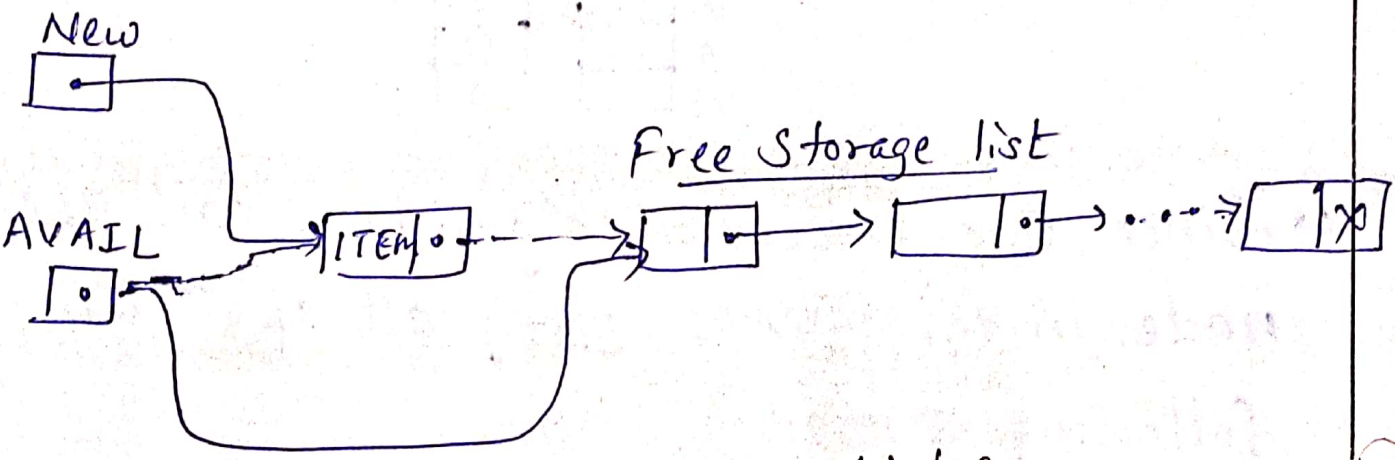
NEW := AVAIL, AVAIL := LINK[AVAIL]

(c) Copying new info^m into the new node.

In other words,

INFO [NEW] := ITEM

The schematic Diagram of the latter two steps is as :-



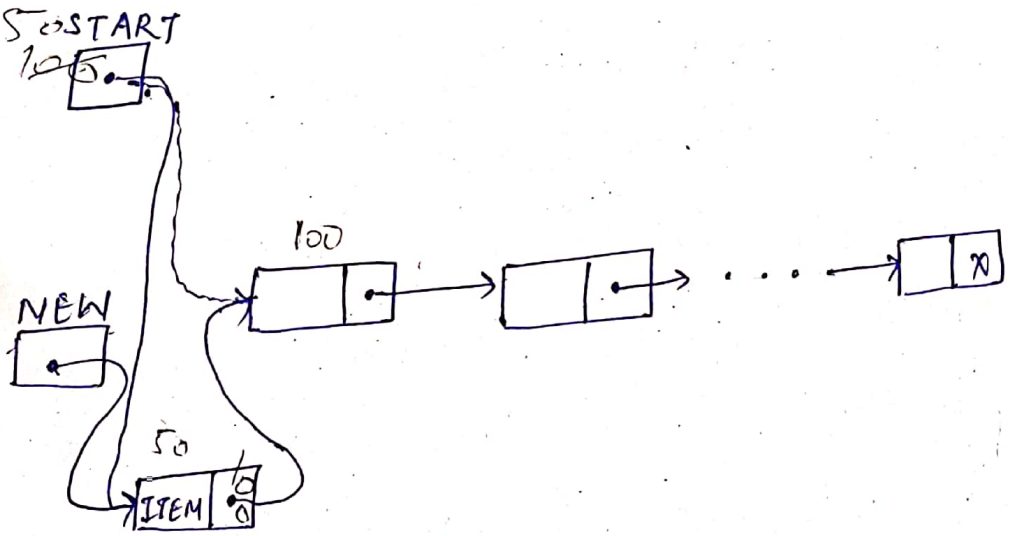
* Inserting at the beginning of a List :-

↳ If our list is not sorted, then the easiest place to insert the node is at the beginning of the list.

Algo^m :- INSFIRST (INFO, LINK, START, AVAIL, ITEM)
This algo^m inserts ITEM as the first node in the list.

1. [OVERFLOW?] 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 [Copies new data into new node]
4. Set LINK[NEW] := START. [New Node now points to original first node]
5. Set START := NEW [Changes START so it points to the new node]
   addr
6. Exit.



Insertion at the Beginning of a List