# QUEUES

# INTRODUCTION

- Linear list of elements.
- Concept- FIFO (first in first out).


Whenever element is added → REAR=REAR+1
Whenever element is deleted→ FRONT=FRONT+1

# ARRAY REPRESENTATION

Array is empty

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |

Insert A,B,C

| A | B | C | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

Front=1
Rear=3

Delete A

| | B | C | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

Front=2
Rear=3

Insert D,E,F

| | B | C | D | E | F |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

Front=2
Rear=6

Insert G

| G | B | C | D | E | F |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

Front=2
Rear=1

# ALGORITHM TO INSERT AN ELEMENT-ENQUEUE

```
QINSERT (queue,N,front,rear,item)

Tf front=1 and rear=N
            then queue is full and overflow
If front=null, then
            set front=1
                 rear=1
Else if
            rear=N, then
            set rear=1
Else
            set rear=rear+1

Set queue[rear]= item

return
```

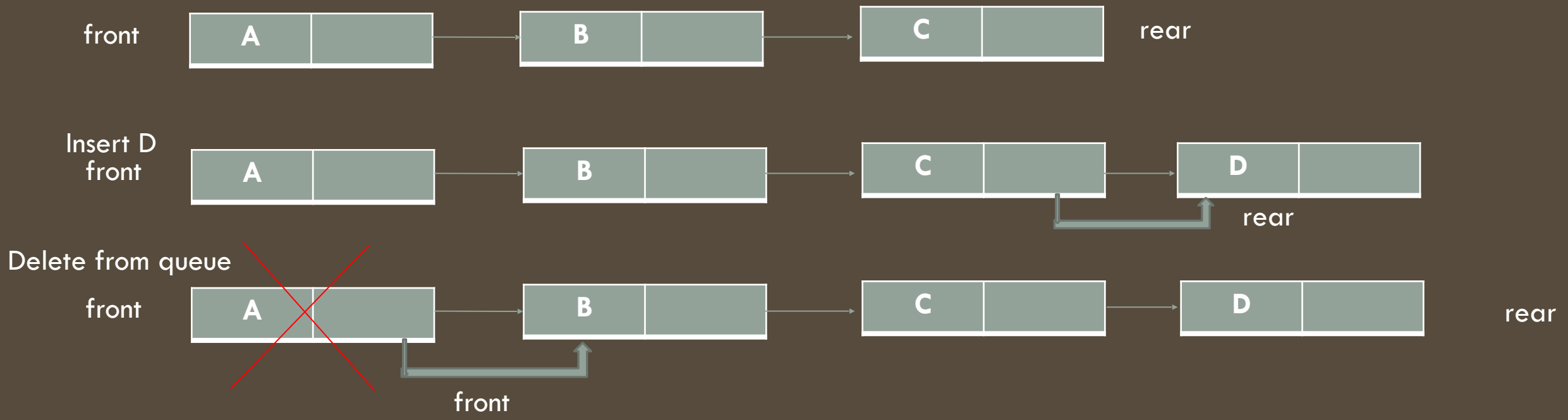
# ALGORITHM TO DELETE AN ELEMENT-DEQUEUE

```
QDELETE (queue,N,front,rear,item)

If front=null
            then queue is empty; underflow
Set item= queue[front]

If front=rear
            set front=null
                rear=null
Else if     front=N
            Set front=1
Else
            Set front=front+1
return
```

# LINKED REPRESENTATION



# ALGORITHM TO INSERT AN ELEMENT

LINKQ_INSERT(INFO,LINK,FRONT,REAR,AVAIL,ITEM)

```
If        AVAIL=NULL
          overflow;

Set       NEW=AVAIL
          AVAIL=LINK[AVAIL]

Set       INFO[NEW]=ITEM
          LINK[NEW]=NULL

If        FRONT=NULL
Then      FRONT=REAR=NEW

Else set  LINK[REAR]=NEW
          REAR=NEW
```

# ALGORITHM TO DELETE AN ELEMENT

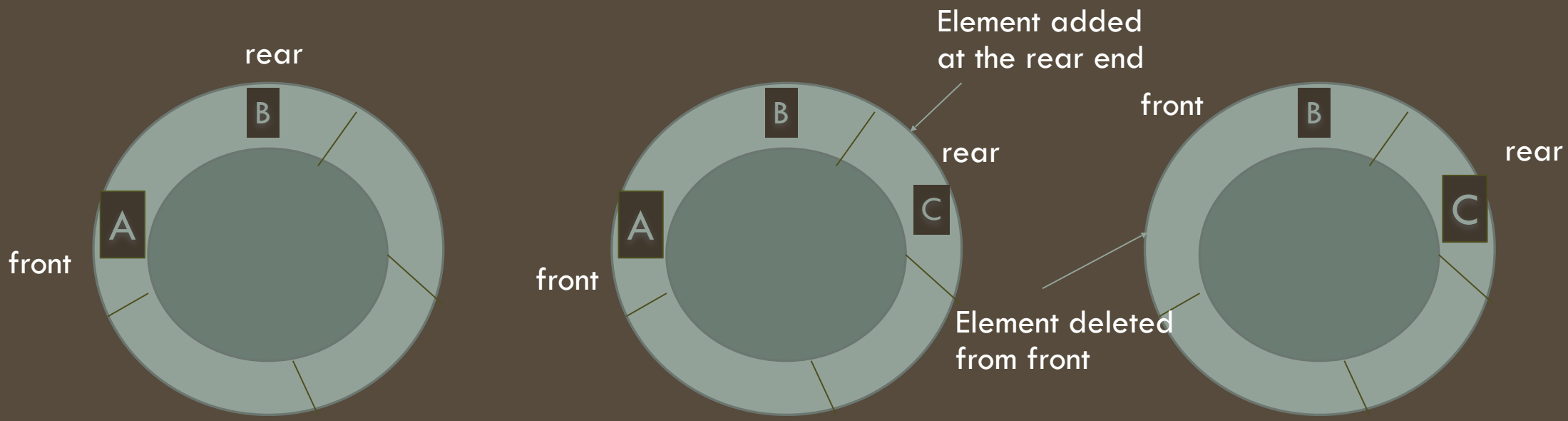
```
LINKQ_DELETE(INFO,LINK,FRONT,REAR,AVAIL,ITEM)

If FRONT=NULL
            underflow;
Set         TEMP=FRONT
            ITEM=INFO[TEMP]
Now
            FRONT=LINK[TEMP]
            LINK[TEMP]=AVAIL
            AVAIL=TEMP
Return deleted node TEMP to AVAIL
exit
```

# CIRCULAR QUEUE

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue.

Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.



# ADDITION IN CIRCULAR QUEUE

```
CIR_QUEUE_INSERT (FRONT,REAR,N,ITEM )

If (FRONT == 1 and REAR == N)
                        Overflow
Else
            If (REAR == 0) Then [Check if QUEUE is empty]
Set FRONT = 1
Set REAR = 1
            Else If (REAR == N) Then [If REAR reaches end of QUEUE]
Set REAR = 1
            Else
Set REAR = REAR + 1 [Increment REAR by 1]
End
Set QUEUE[REAR] = ITEM
ITEM inserted
End
Exit
```

# DELETION IN CIRCULAR QUEUE

```
CIR_QUEUE_DELETE (FRONT,REAR,N,ITEM )

If (FRONT == 0)
            Underflow
Else
            ITEM = QUEUE[FRONT]
If (FRONT == REAR) Then [If only element is left]
            Set FRONT = 0
            Set REAR = 0
Else If (FRONT == N) Then [If FRONT reaches end if QUEUE]
            Set FRONT = 1
Else
            Set FRONT = FRONT + 1 [Increment FRONT by 1]
End
ITEM deleted
Exit
```

# PRIORITY QUEUES

**Priority queues**: Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

A priority queue is a queue where:

- The element with highest priority is processed first (deleted from the queue)
- Two elements with the same priority are processed according to the order in which they were added to the queue.

**Usage of queues**:
In resource management: several users waiting for one and the same resource.
for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates.

# REPRESENTATION OF PRIORITY QUEUE

Representation of Priority queue: (One-Way List)

Each node in the list contains 3 items of information:

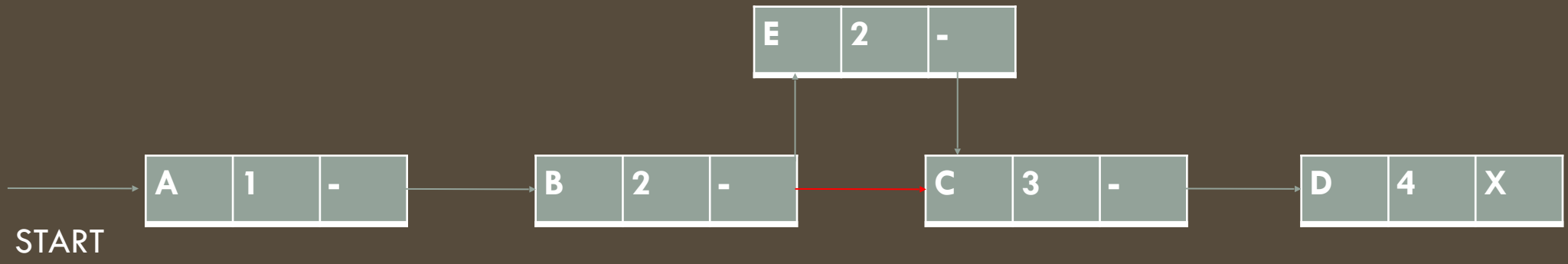
- INFO
- PRN
- LINK

# ADDING ELEMENTS IN PRIORITY QUEUE

Adding an element is more complicated than deleting an element in priority queue.

ADD_PQ (ITEM, N)

Traverse the queue to find the element N whose priority number is greater than ITEM
Insert ITEM in front of N
Else
Insert ITEM as the last element of the list



# DELETING ELEMENTS IN PRIORITY QUEUE

DELETE_PQ

The element with highest priority
i.e PRN= Highest(PRN)
Is deleted from the queue
This element is the first element in the priority queue.

# DEQUE

An deque is a linear list in which elements can be added or removed at either end but not in the middle.

Two variations od deque are:

- Input-restricted deque: it is a deque that allows insertion at only one end but allow deletion at both ends of the list.
- Output-restricted deque: it is a deque that allows deletion at only one end but allow insertion at both ends of the list.

# SINGLY LINKED LIST

A linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is, by storing the address of the next element in the link field of the previous element.

Link list used for the dynamic memory allocation.

Array and link list both are the linear data structure.

Representation of link list:

Link list consists a series of structure. Each structure consists of a data field and address field. Data field consists data part and the address field contains the address of the successors.

# SINGLY LINKED LIST CONT..

Advantage of Link list

1. Link list is an example of dynamic data structure. They can grow and shrink during the execution of program.
2. Efficient memory utilization. Memory is not pre allocated like static data structure. The allocation of memory depends upon the user.
3. Insertion and deletion easily performed.

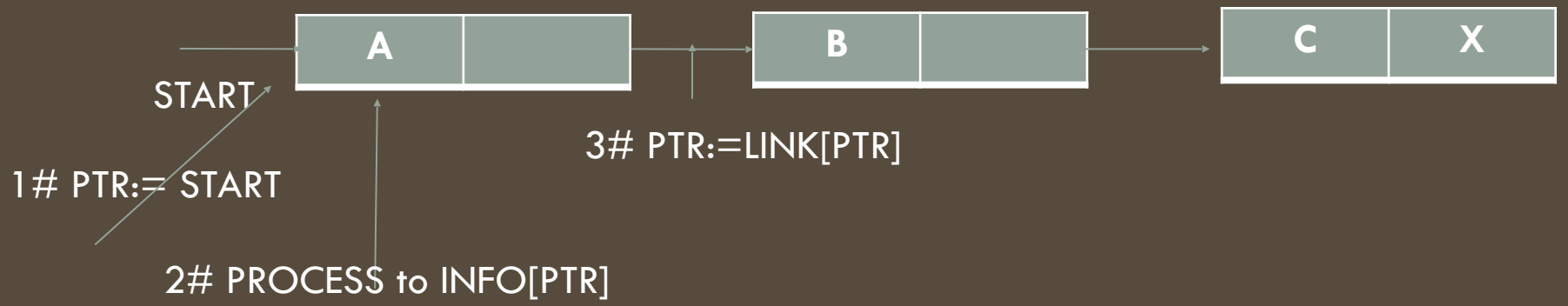
**Basic operations of a singly-linked list are:**

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
Print – Prints the list.

# TRAVERSING A LINKED LIST

The algorithm traverses a linked list, applying an operation PROCESS to each element in the list.
The variable PTR points to the node currently being processed.

Set PTR:= START //initialise the pointer
Repeat step 3 and 4 while PTR not equal to NULL
Apply PROCESS to INFO[PTR]
Set PTR:=LINK[PTR] //PTR points to next node
Exit



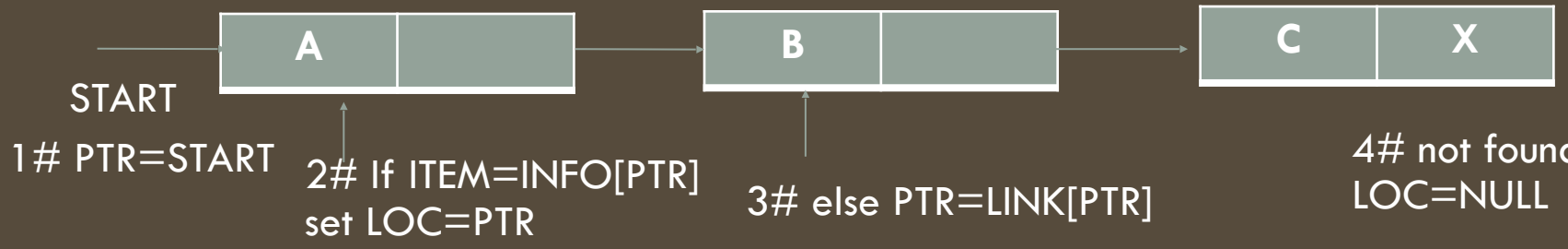
# SEARCHING A LINKED LIST- UNSORTED LIST

We need to search an ITEM in the LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST.

SEARCH(INFO,LINK,START,ITEM,LOC)

This algorithm finds the location LOC of the node where ITEM first appears or sets LOC=NULL

```
Set PTR=START
        repeat step 3 while PTR!=NULL
If ITEM=INFO[PTR] then
        set LOC=PTR and exit
Else
        set PTR=LINK[PTR] //PTR points to next node
End
End
Set LOC=NULL //unsuccessful
exit
```



# SEARCHING A LINKED LIST-SORTED LIST

SEARCH_SL(INFO,LINK,START,ITEM,LOC)

The list is sorted in memory. This algorithm finds the location LOC of the node where ITEM first appears or sets LOC=NULL

Set PTR=START

repeat step 3 while PTR!=NULL

If ITEM<INFO[PTR]

set PTR=LINK[PTR] //PTR points to next node

Else If ITEM=INFO[PTR] then

set LOC=PTR and exit

Else

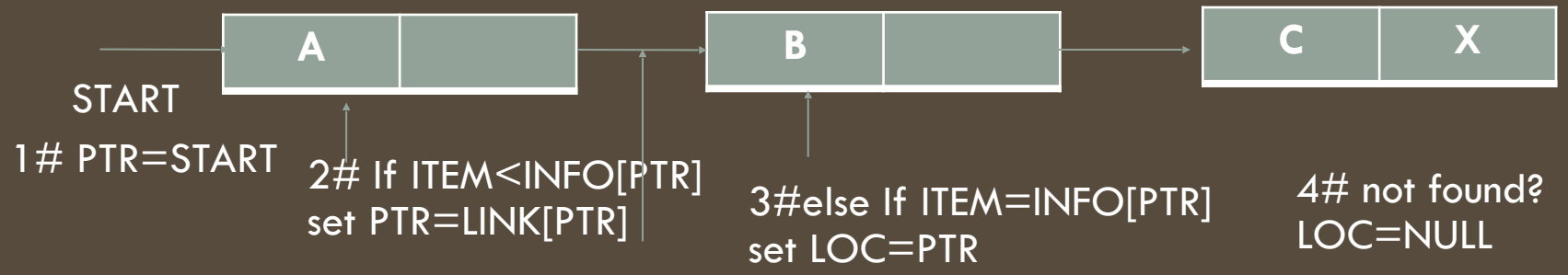
set LOC=NULL and exit

End

End

Set LOC=NULL //unsuccessful

exit



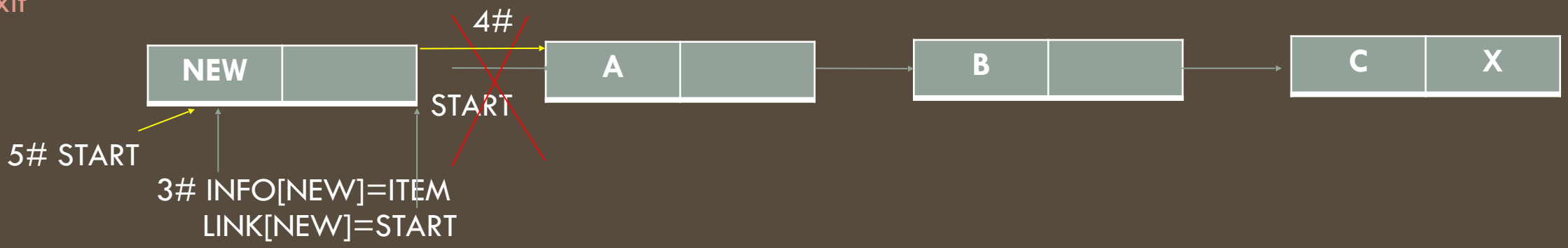
# INSERTION-AT BEGINNING

This algorithm inserts ITEM as the first node in the list.

INSFIRST(INFO,LINK,START,AVAIL,ITEM)

If AVAIL=NULL then error ;
Overflow;
Set NEW=AVAIL //remove first node from avail
AVAIL=LINK[AVAIL]
Set INFO[NEW]=ITEM
LINK[NEW]=START //new node points to original start node
Set START=NEW

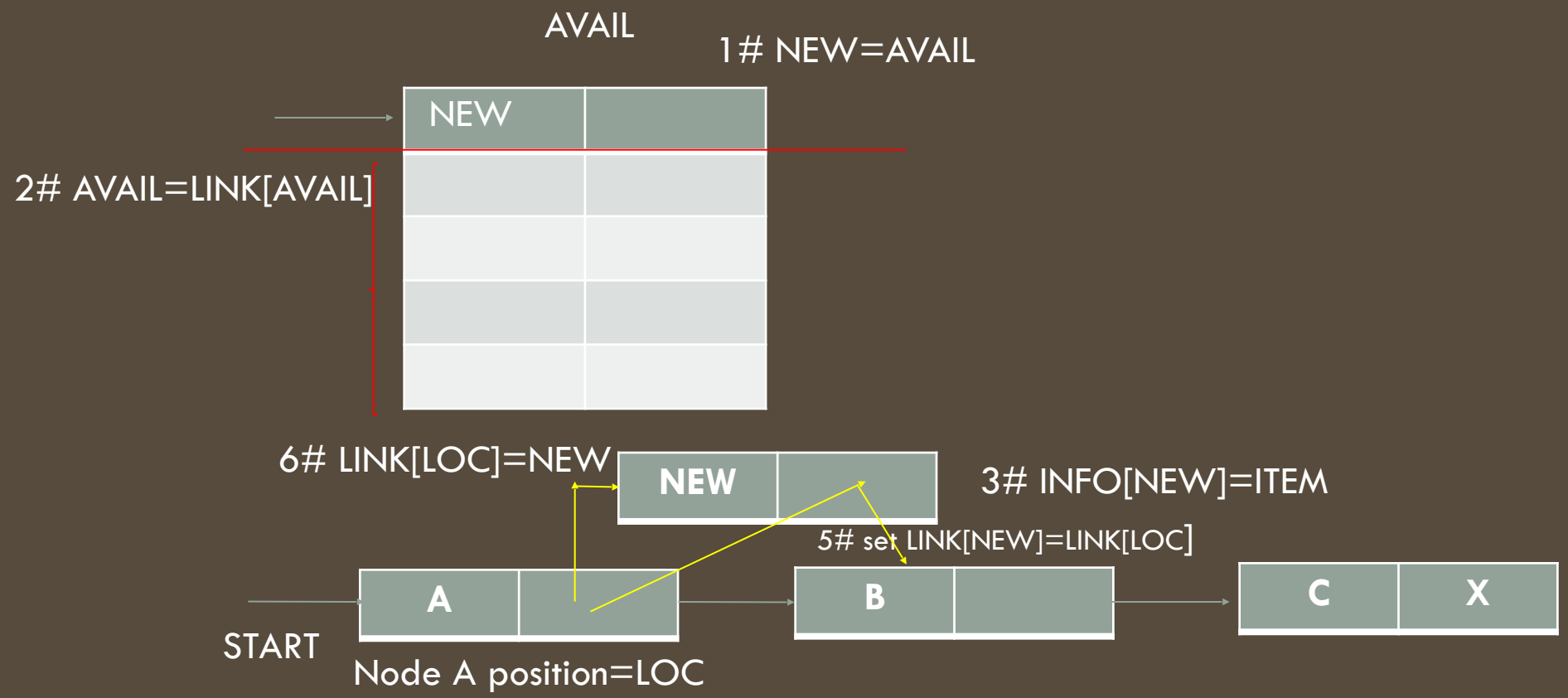
exit

# INSERTION-AFTER A GIVEN NODE

This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as first node when LOC=NULL.
INSLOC(INFO,LINK,START,AVAIL,LOC,ITEM)

```
If AVAIL=NULL then error ;
            Overflow;
Set NEW=AVAIL //remove first node from avail
    AVAIL=LINK[AVAIL]
Set INFO[NEW]=ITEM
If LOC=NULL then
            set LINK[NEW]=START //insert at first position
            and START=NEW
Else
            set LINK[NEW]=LINK[LOC]
            and LINK[LOC]=NEW
exit
```




4# LOC=NULL then
set LINK[NEW]=START //insert at first position
and START=NEW

# INSERTING INTO A SORTED LIST

Item is to be inserted into a sorted list. The item must be inserted between node A and B, such that:

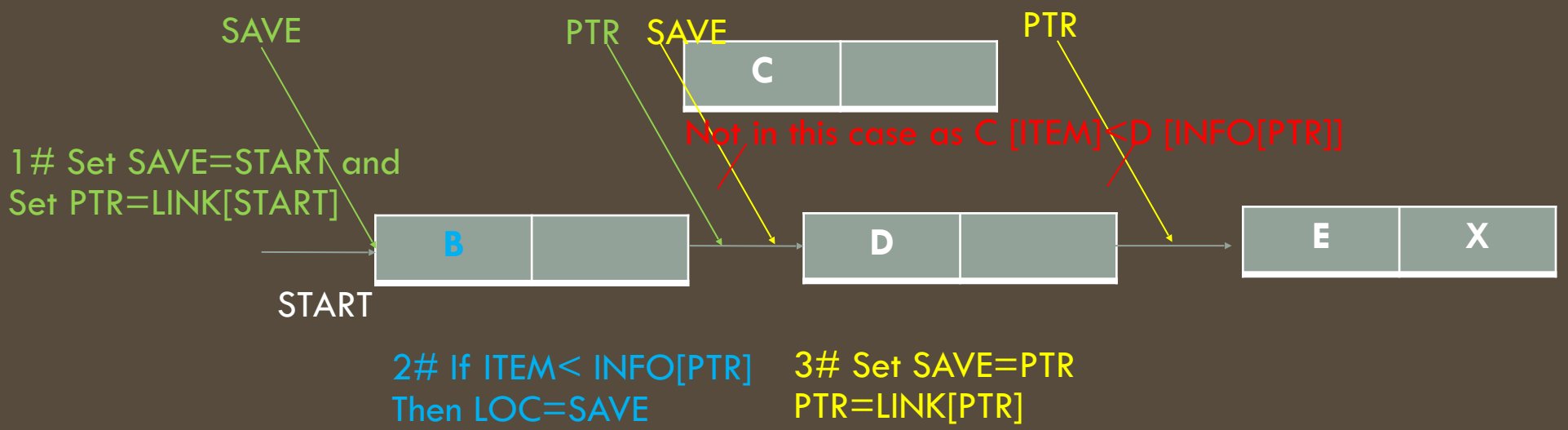
INFO[A]<ITEM<=INFO[B]

First we find the location where the element is to be inserted.

```
FIND(INFO,LINK,START,ITEM,LOC)
If START=NULL then
        set LOC=NULL //list is empty
If ITEM<INFO[START]
        set LOC=NULL and return
set SAVE=START and
Set PTR=LINK[START]
Repeat step 5 and 6 while PTR!=NULL
If ITEM< INFO[PTR]
Then set    LOC=SAVE and return
end
Set SAVE=PTR
        PTR=LINK[PTR]
End
Set LOC=SAVE
return
```

# INSERTING INTO A SORTED LIST.. CONT..

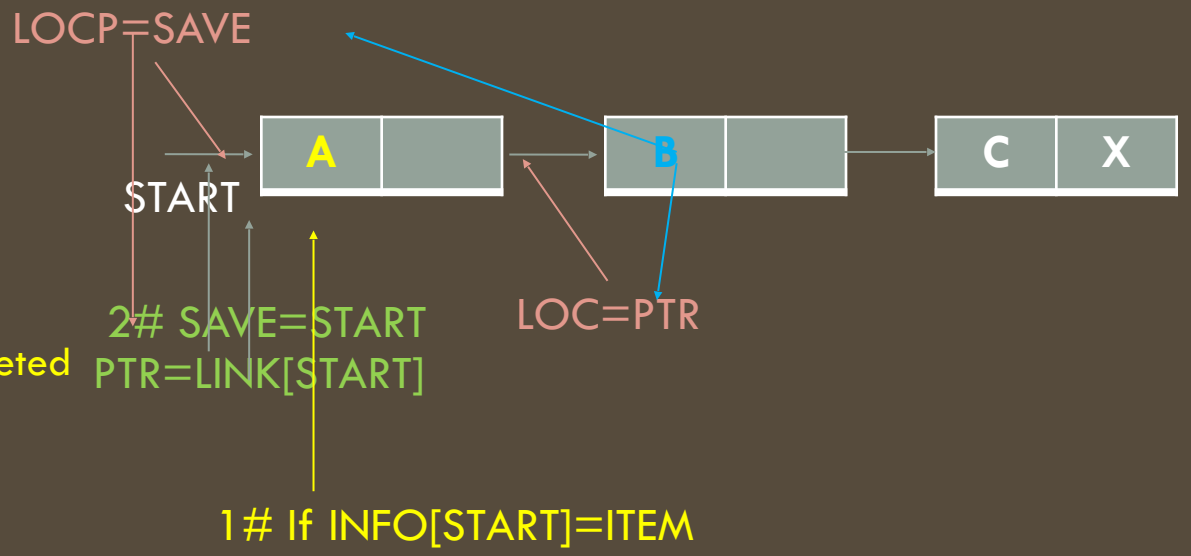
INSERT_S(INFO,LINK.START,AVAIL,ITEM)
Find the location of the node preceding the ITEM
call FIND(INFO,LINK,START,ITEM,LOC)
Insert the ITEM after the node with location LOC
call INSLOC(INFO,LINK,START,AVAIL,LOC,ITEM)



# DELETION FROM A LINKED LIST

This algorithm deletes the first element from the list containing ITEM or an element at a location LOC containing ITEM and a preceding node at location LOCP

```
FIND(INFO,LINK,START,ITEM,LOC,LOCP)
If START=NULL then
            set LOC=NULL
            and LOCP=NULL //list is empty
If INFO[START]=ITEM, then
            set LOC=START
            and LOCP=NULL // first node has to be deleted
Set SAVE=START
PTR=LINK[START]
Repeat step 5 and 6 while PTR!=NULL
If INFO[PTR]=ITEM
Then        LOC=PTR
            LOCP=SAVE return
Set SAVE=PTR
            PTR=LINK[PTR]
Set LOC=NULL
return
```



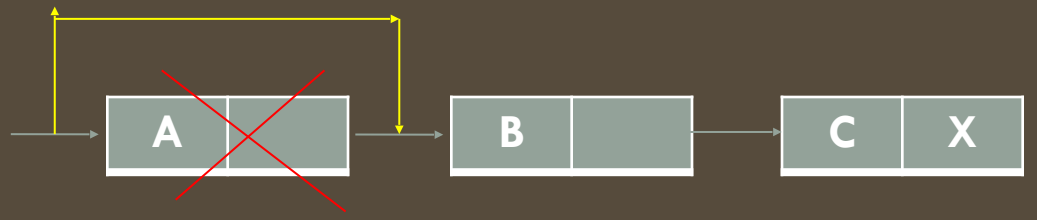
# DELETION FROM A LINKED LIST CONT..

```
DELETE(INFO,LINK,START,AVAIL,ITEM)
Call FIND(INFO,LINK,START,ITEM,LOC,LOCP)

If LOC=NULL
        item not in the list
If LOCP=NULL
        then set START=LINK[START] //delete first node
Else
Set LINK[LOCP]=LINK[LOC]
End

LINK[LOC]=AVAIL // return node to avail
AVAIL=LOC

exit
```



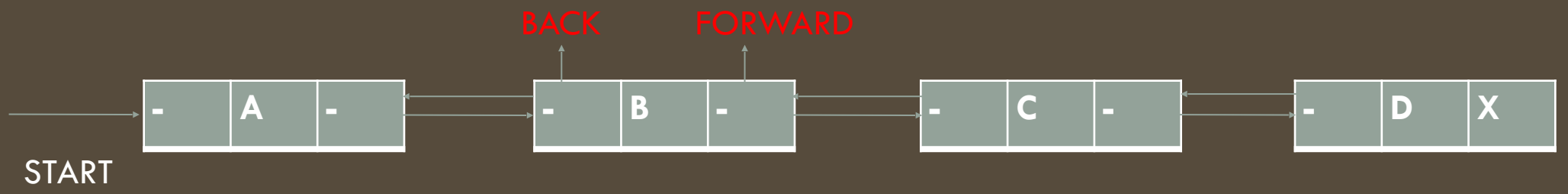

LINK[LOCP]=LINK[LOC]

# DOUBLY LINKED LIST (TWO WAY LISTS)

In this type of liked list each node holds two-pointer field. Pointers exist between adjacent nodes in both directions. The list can be traversed either forward or backward.

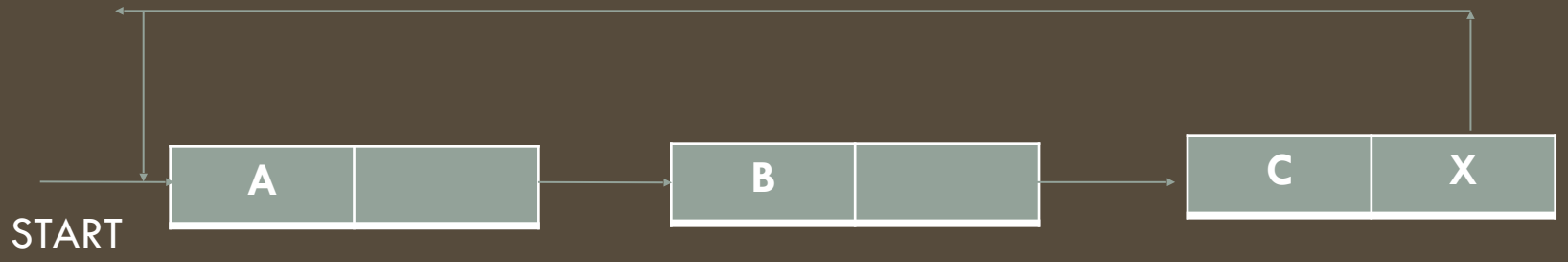
Doubly Linked List are more convenient than Singly Linked List since we maintain links for bi-directional traversing

Each Node contains two fields, called Links, which are references to the previous and to the Next Node in the sequence of Nodes.



# CIRCULAR LINKED LIST

Circular list is a list in which the link field of the last node is made to point to the start/first node of the list.



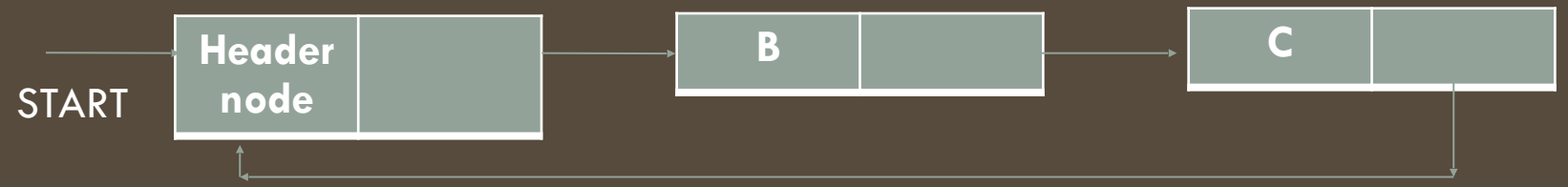
# HEADER LINKED LIST

A header linked list is a linked list which always contain a special node called the HEADER NODE, at the beginning of The list.

Two kinds of widely used header linked lists are:

-A grounded header list: it is a header list where last node contains the null pointer.


-A circular header list: it is a header list where last node points back to the header node.



# APPLICATIONS

Queues are mostly used in operating systems.

- Waiting for a particular event to occur.
- Scheduling of processes