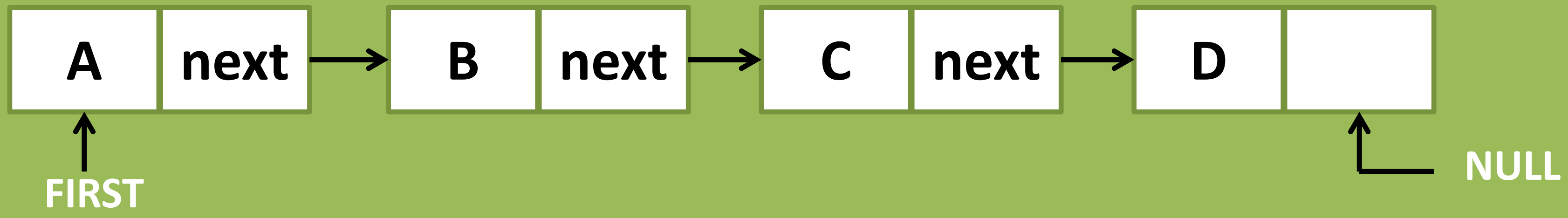


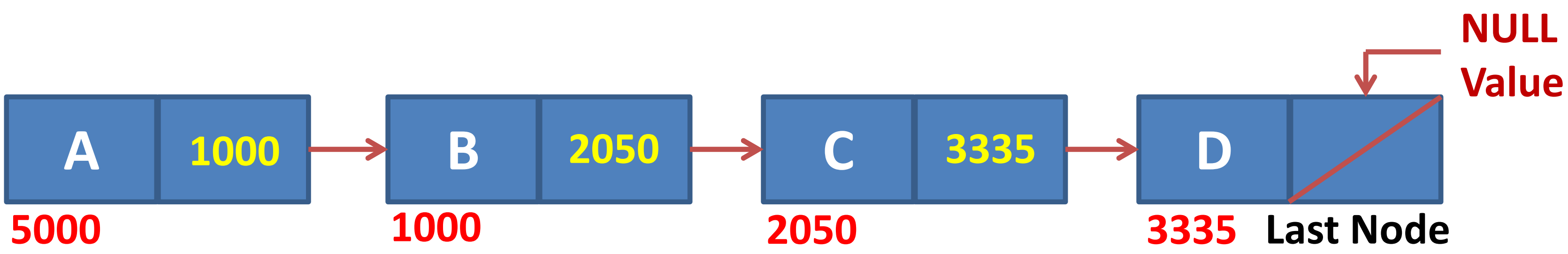
Linked List

Linear Data Structure

Linked Storage Representation

- There are many applications where **sequential allocation** method is **unacceptable** because of following characteristics
  - **Unpredictable storage** requirement
  - **Extensive manipulation** of stored data
- One method of obtaining the address of node is to store address in computer's main memory, we refer this addressing mode as **pointer of link addressing**.
- A simple way to represent a linear list is to expand each node to contain a link or pointer to the next node. This representation is called one-way chain or Singly Linked Linear List.

Linked Storage Representation

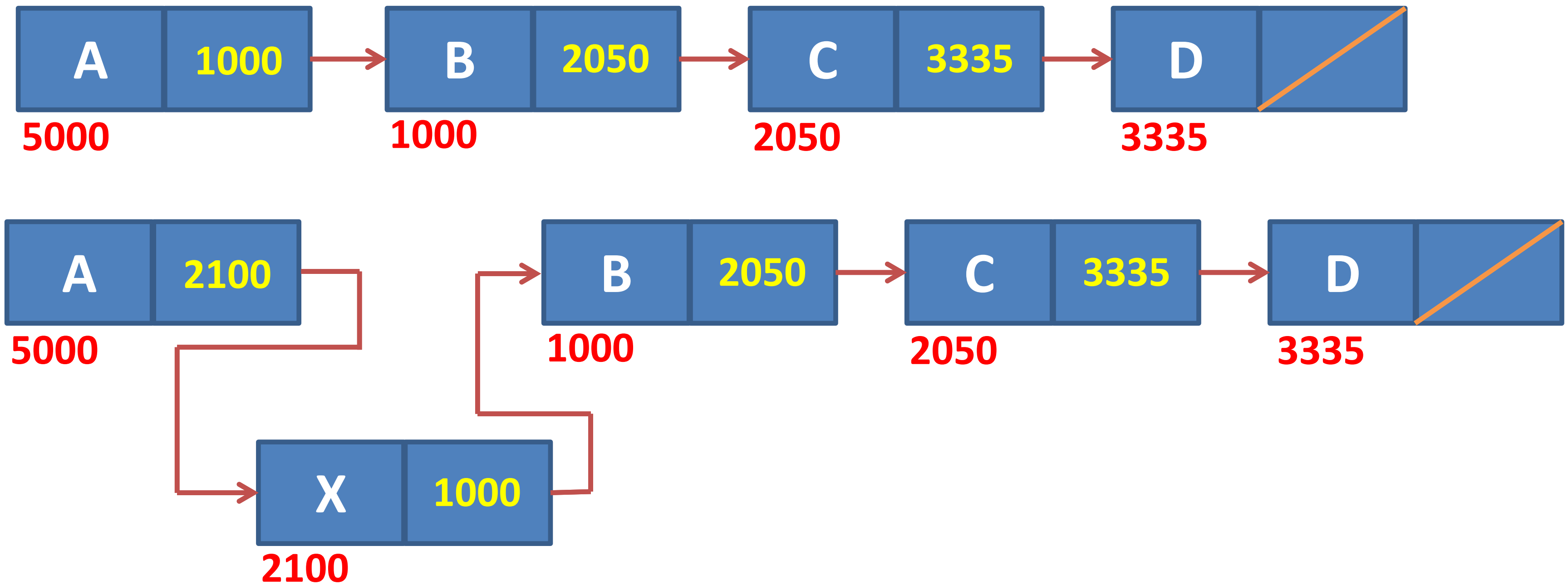


A linked List

- The linked allocation method of storage can result in both efficient use of computer storage and computer time.
  - A linked list is a **non-sequential collection** of data items.
  - Each **node** is **divided** into **two parts**, the **first part** represents the **information** of the element and the **second part** contains the **address of the next node**.
  - The **last node** of the list does not have successor node, so **null value** is stored as the address.
  - It is possible for a list to have no nodes at all, such a list is called empty list.

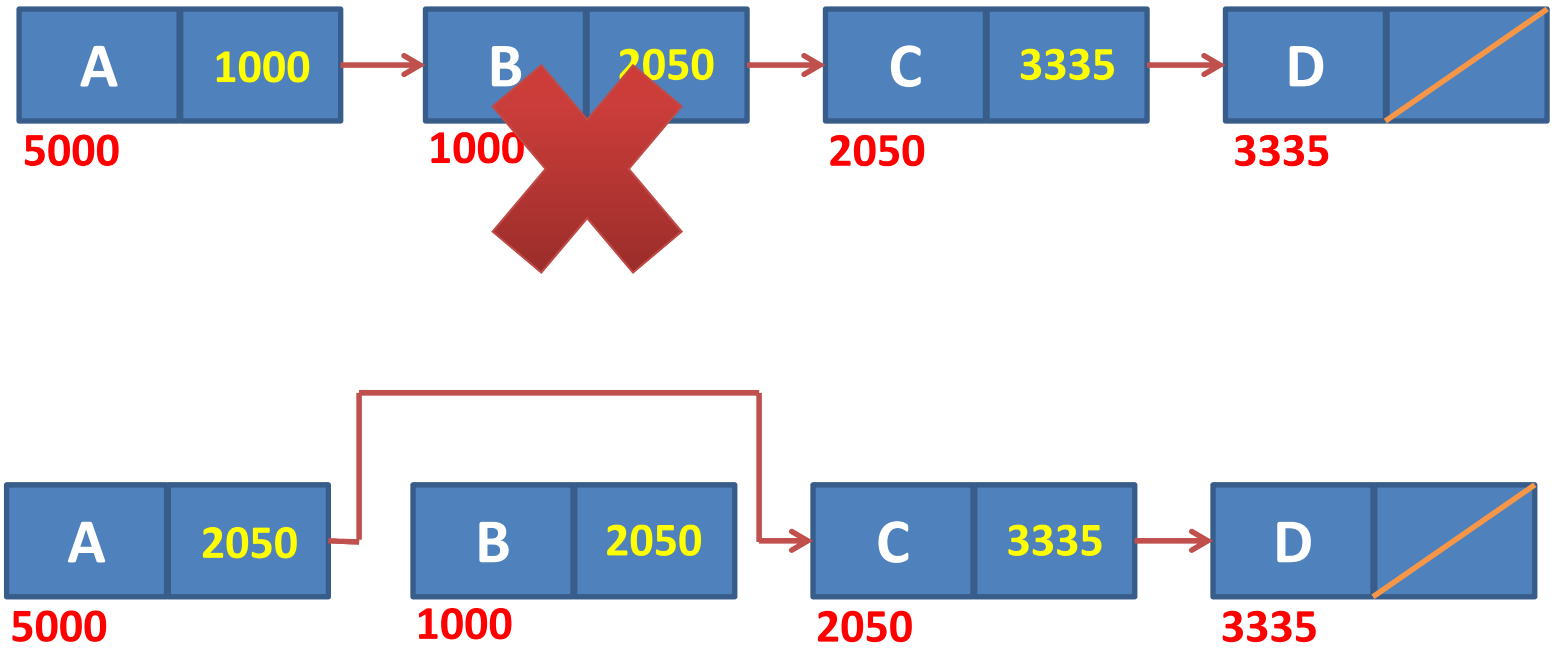
Pros & Cons of Linked Allocation

- **Insertion Operation**
  - we have an n elements in list and it is required to insert a new element between the first and second element, what to do with sequential allocation & linked allocation?
  - Insertion operation is more efficient in Linked allocation.



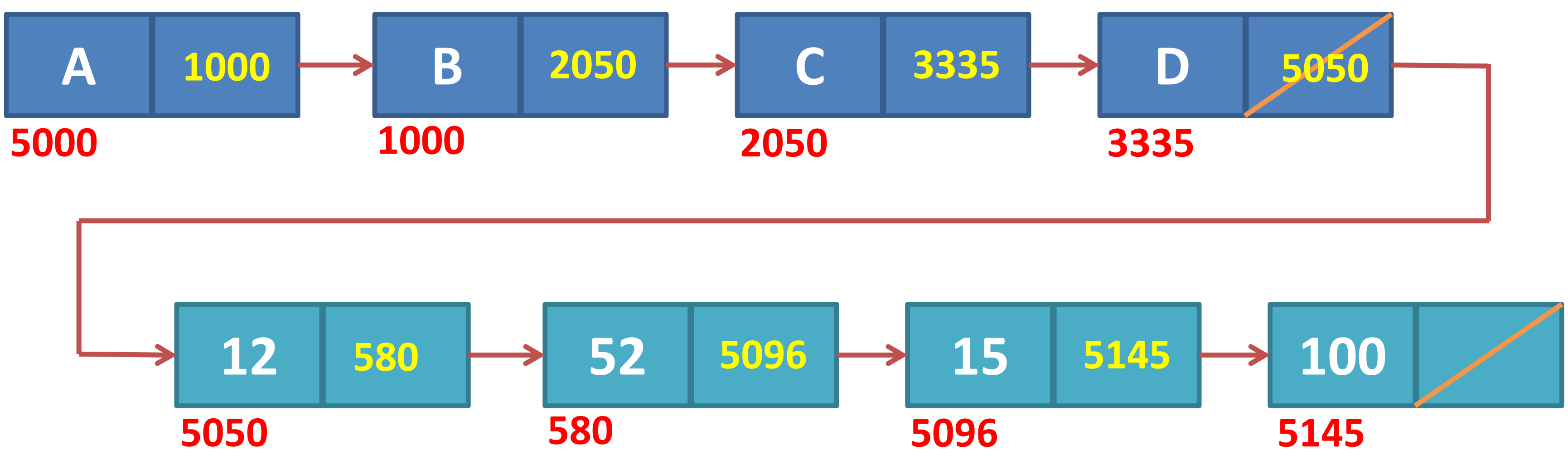
Pros & Cons of Linked Allocation

- **Deletion Operation**
  - Deletion operation is more efficient in Linked Allocation



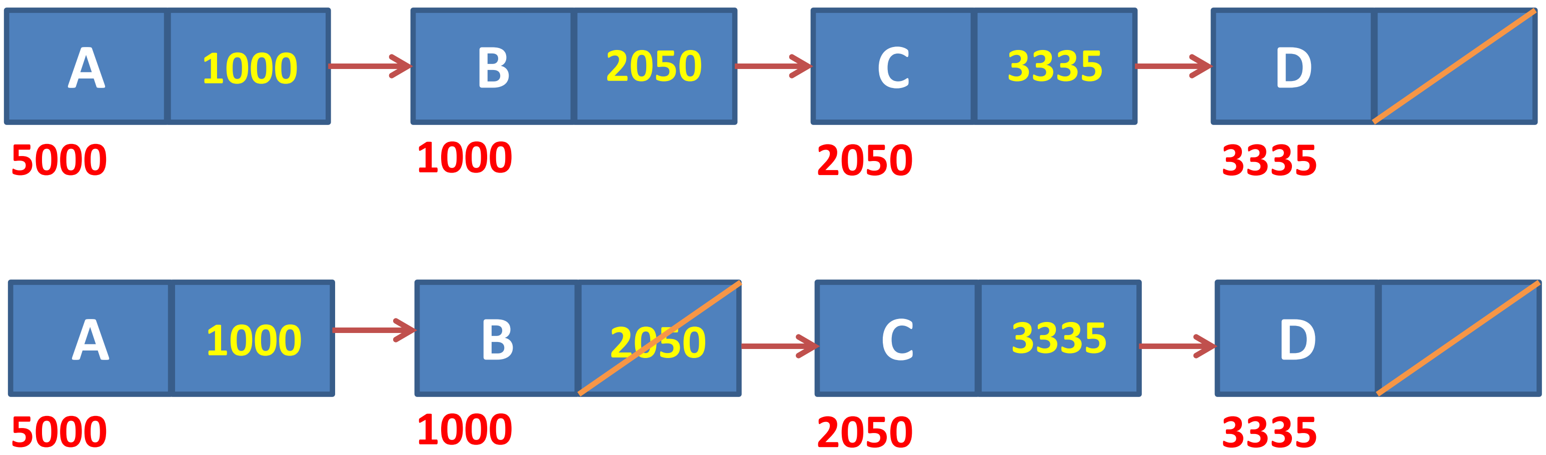
Pros & Cons of Linked Allocation

- **Search Operation**
  - **If particular node in the list is required**, it is necessary to follow links from the first node onwards until the desired node is found, in this situation **it is more time consuming** to go through linked list than a sequential list.
  - Search operation is more time consuming in Linked Allocation.
- **Join Operation**
  - Join operation is more efficient in Linked Allocation.



Pros & Cons of Linked Allocation

- Split Operation
  - Split operation is more efficient in Linked Allocation



Pros & Cons of Linked Allocation

- Linked list require **more memory** compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc…

Operations & Type of Linked List

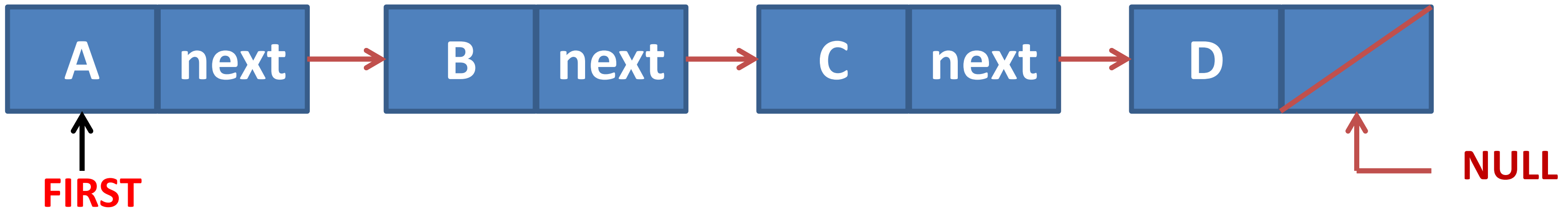
Operations on Linked List

- Insert
  - Insert at first position
  - Insert at last position
  - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

Types of Linked List

- Singly Linked List
- Circular Linked List
- Doubly Linked List

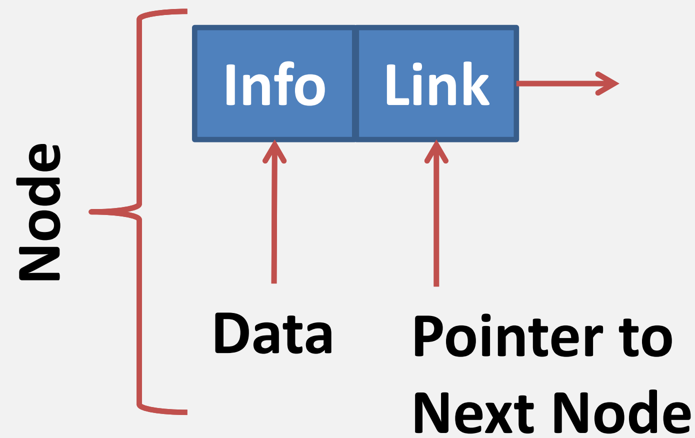
Singly Linked List



- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- First node address is available with pointer variable **FIRST**.
- **Limitation** of singly linked list is **we can traverse only in one direction**, forward direction.

Node Structure of Singly List

Typical Node



Accessing Part of Node

Info (Node)

Link (Node)

C Structure to represent a node

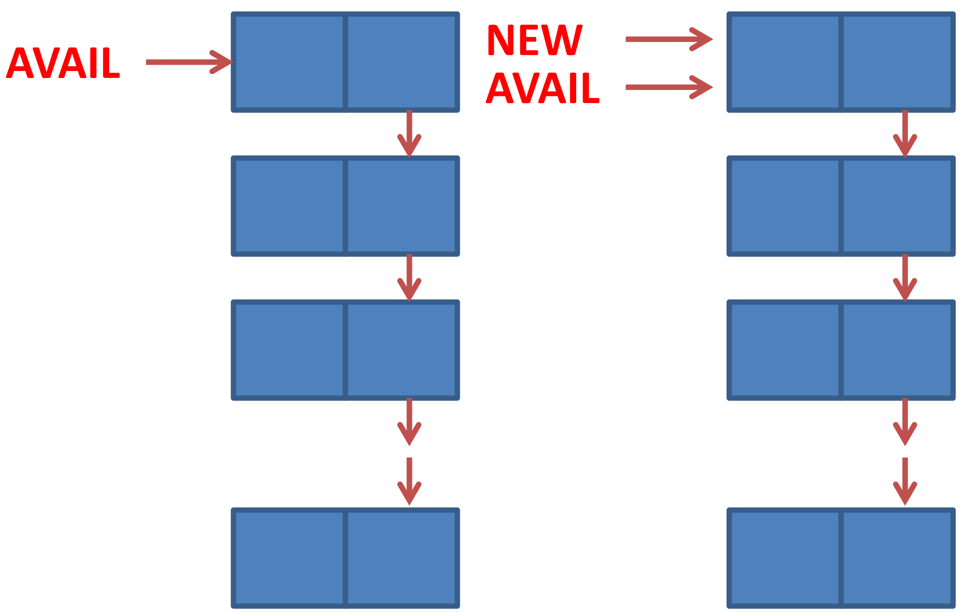
```
struct node
{
        int info;
        struct node *link;
};
```

Algorithms for singly linked list

1. Insert at first position
2. Insert at last position
3. Insert in Ordered Linked list
4. Delete Element
5. Copy Linked List

Availability Stack

- A **pool** or list **of free nodes**, which we refer to as the **availability stack** is maintained in conjunction with linked allocation.
- Whenever a node is to be inserted in a list, a free node is taken from the availability stack and linked to the new list.
- On other end, the deleted node from the list is added to the availability stack.



Check for free node in Availability Stack

IF AVAIL is NULL
THEN Write('Availability Stack Underflow')
Return

Obtain Address of next free node

NEW ← AVAIL

Remove free node from Availability Stack

AVAIL ← LINK(AVAIL)

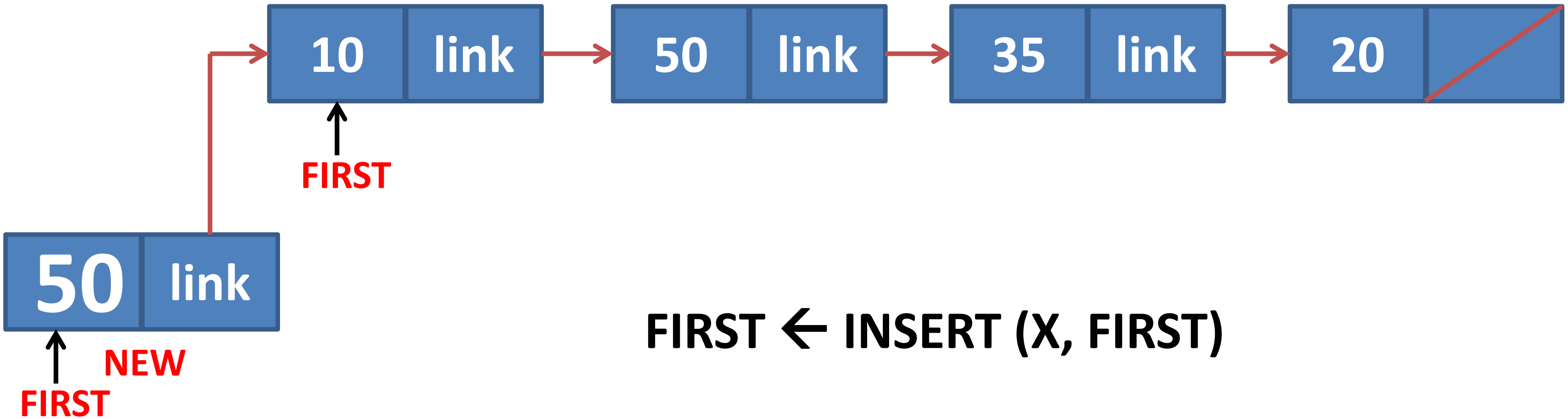
Function: INSERT(X,First)

- This function **inserts a new node at the first position** of Singly linked list.
- This function returns address of **FIRST** node.
- **X** is a new element to be inserted.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW** is a temporary pointer variable.

Function: INSERT(X,FIRST) Cont…

```
1. [Underflow?]
   IF     AVAIL = NULL
   Then   Write ("Availability Stack Underflow")
          Return(FIRST)
2. [Obtain address of next free Node]
    NEW ← AVAIL
3. [Remove free node from availability Stack]
    AVAIL ← LINK(AVAIL)
4. [Initialize fields of new node and
   its link to the list]
    INFO(NEW) ← X
    LINK (NEW) ← FIRST
5. [Return address of new node]
    Return (NEW)
```

Example: INSERT(50,FIRST)



FIRST ← INSERT (X, FIRST)

```
4. [Initialize fields of new node and
   its link to the list]
    INFO(NEW) ← X
    LINK (NEW) ← FIRST
5. [Return address of new node]
    Return (NEW)
```

Function: INSEND(X, FIRST)

- This function **inserts** a new node at the **last position** of linked list.
- This function returns address of **FIRST** node.
- **X** is a new element to be inserted.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW** is a temporary pointer variable.

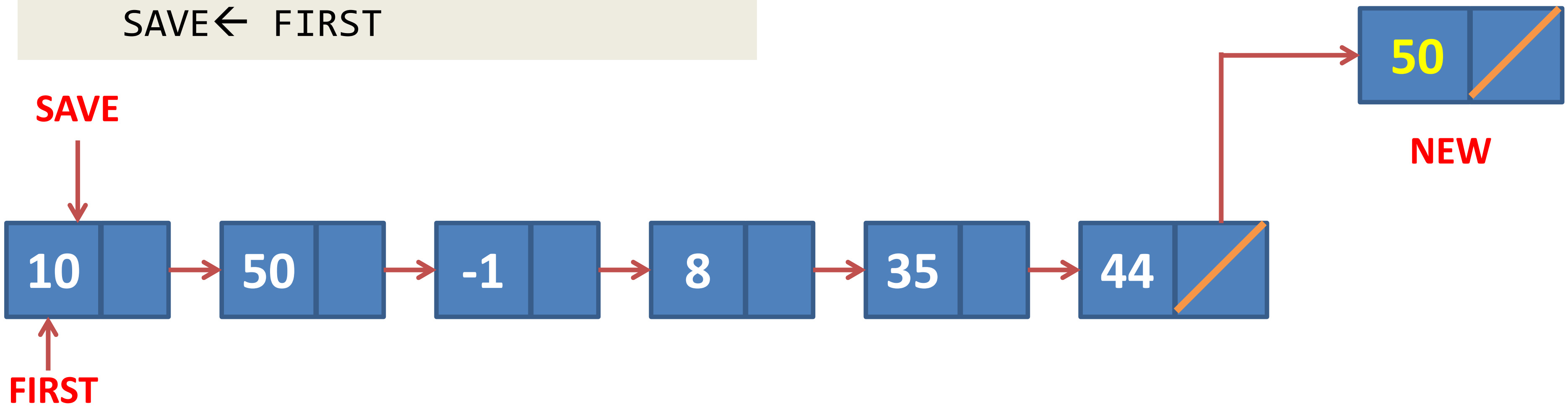
Function: INSEND(X, First) Cont...

```
1. [Underflow?]
   IF     AVAIL = NULL
   Then   Write ("Availability
          Stack Underflow")
          Return(FIRST)

2. [Obtain address of
    next free Node]
    NEW ← AVAIL

3. [Remove free node from
    availability Stack]
    AVAIL ← LINK(AVAIL)

4. [Initialize fields of
   new node]
    INFO(NEW) ← X
    LINK (NEW) ← NULL

5. [Is the list empty?]
    If    FIRST = NULL
    Then  Return (NEW)
6. [Initialize search for
   a last node]
    SAVE← FIRST

7. [Search for end of list]
   Repeat while LINK (SAVE) ≠ NULL
       SAVE ← LINK (SAVE)

8. [Set link field of last node
   to NEW]
    LINK (SAVE) ← NEW
9. [Return first node pointer]
    Return (FIRST)
```

Function: INSEND(50, FIRST)

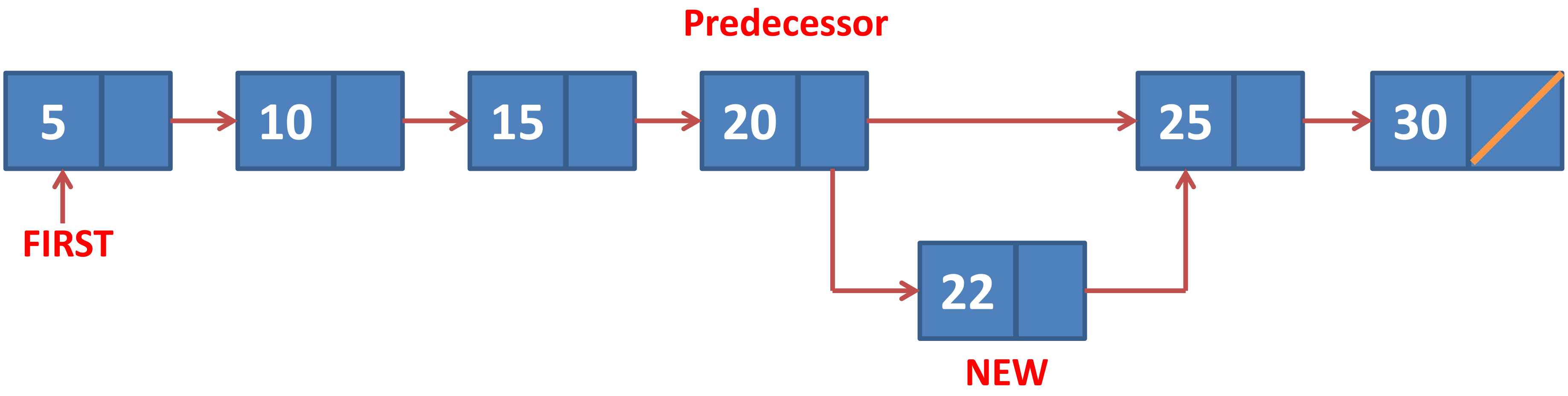
```
4. [Initialize fields of
   new node]
    INFO(NEW) ← X
    LINK (NEW) ← NULL

5. [Is the list empty?]
    If     FIRST = NULL
    Then   Return (NEW)

6. [Initialize search for
   a last node]
    SAVE← FIRST
```

```
7. [Search for end of list]
 Repeat while LINK (SAVE) ≠ NULL
    SAVE ← LINK (SAVE)

8. [Set link field of last node
   to NEW]
    LINK (SAVE) ← NEW

9. [Return first node pointer]
    Return (FIRST)
```



Function: INSORD(X, FIRST)

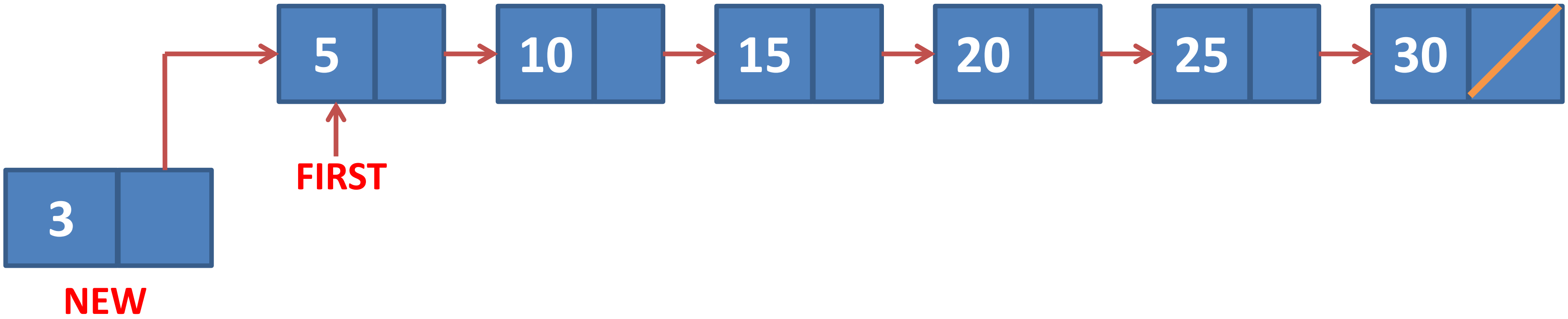
- This function **inserts** a new node such that linked list preserves the ordering of the terms in **increasing order** of their **INFO** field.
- This function returns address of **FIRST** node.
- **X** is a new element to be inserted.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW** is a temporary pointer variable.



Function: INSORD(X, FIRST)

```
1. [Underflow?]
   IF     AVAIL = NULL
   THEN   Write ("Availability
          Stack Underflow")
          Return(FIRST)
2. [Obtain address of
    next free Node]
    NEW ← AVAIL
3. [Remove free node from
    availability Stack]
    AVAIL ← LINK(AVAIL)
4. [Initialize fields of
   new node]
    INFO(NEW) ← X
5. [Is the list empty?]
    IF     FIRST = NULL
    THEN   LINK(NEW) ← NULL
           Return (NEW)
6. [Does the new node precede
   all other node in the list?]
    IF    INFO(NEW) ≤ INFO (FIRST)
    THEN LINK (NEW) ← FIRST
          Return (NEW)
7. [Initialize temporary pointer]
   SAVE ← FIRST
8. [Search for predecessor of
    new node]
   Repeat while LINK (SAVE) ≠ NULL
    & INFO(NEW) ≥ INFO(LINK(SAVE))
        SAVE ← LINK (SAVE)
9. [Set link field of NEW node
   and its predecessor]
    LINK (NEW) ← LINK (SAVE)
    LINK (SAVE) ← NEW
10. [Return first node pointer]
    Return (FIRST)
```

Function: INSORD(3, FIRST)

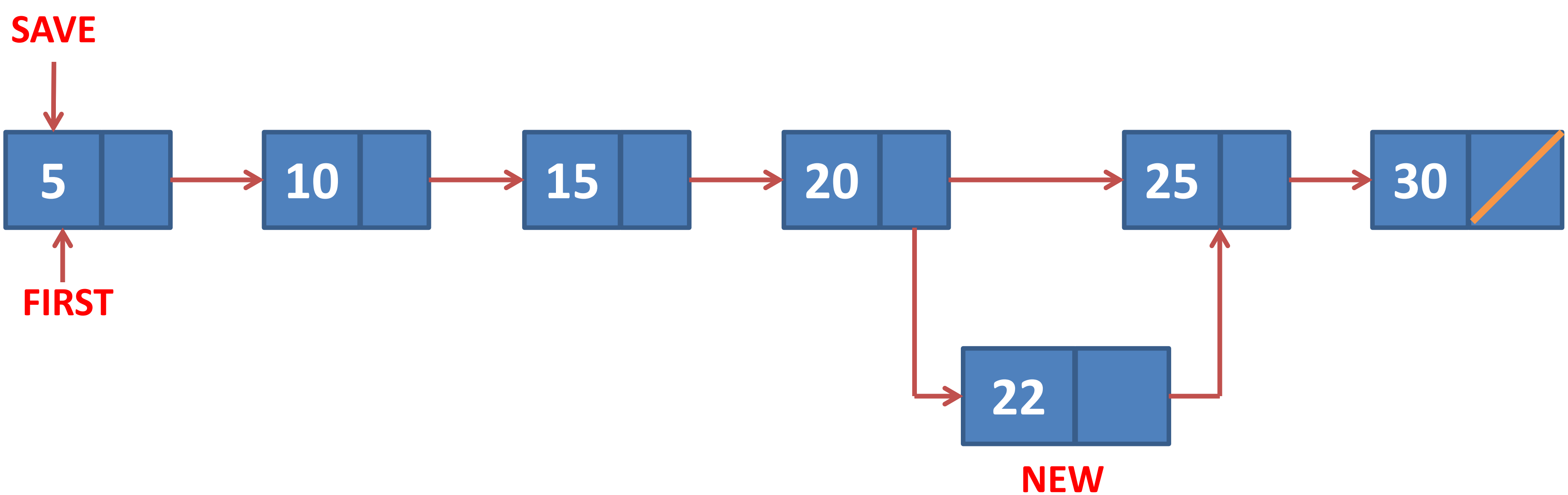



```
6. [Does the new node precede all other node in the
     list?]
     IF   INFO(NEW) ≤ INFO (FIRST)
     THEN LINK (NEW) ← FIRST
           Return (NEW)
```

Function: INSORD(22, FIRST)

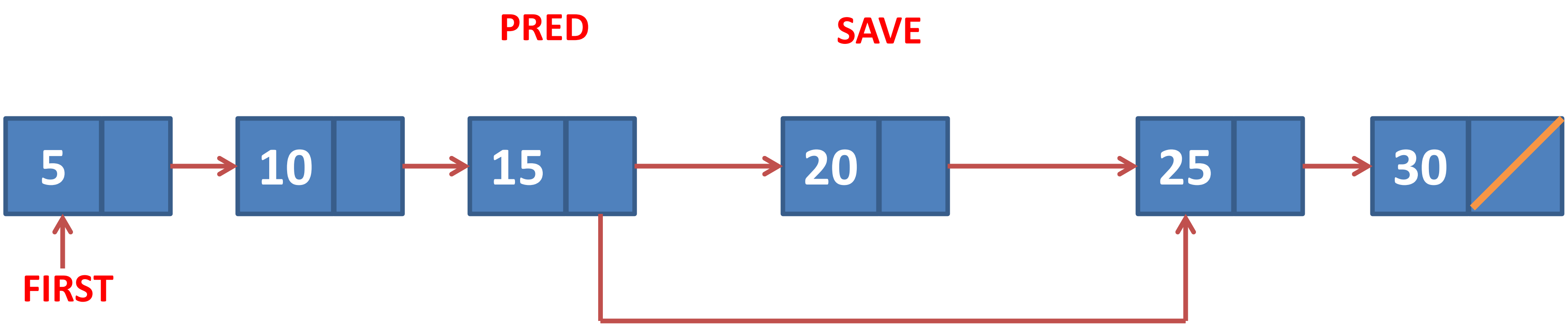
```
7. [Initialize temporary pointer]
     SAVE ← FIRST
8. [Search for predecessor of
     new node]
Repeat while LINK (SAVE) ≠ NULL
 & INFO(NEW) ≥ INFO(LINK(SAVE))
     SAVE ← LINK (SAVE)
```

```
9. [Set link field of NEW node
and its predecessor]
   LINK (NEW) ← LINK (SAVE)
   LINK (SAVE) ← NEW
10. [Return first node
     pointer]
     Return (FIRST)
```



Procedure: DELETE(X, FIRST)

- This algorithm **delete** a node whose address is given by variable **X**.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **SAVE & PRED** are temporary pointer variable.



Procedure: DELETE(X, FIRST)

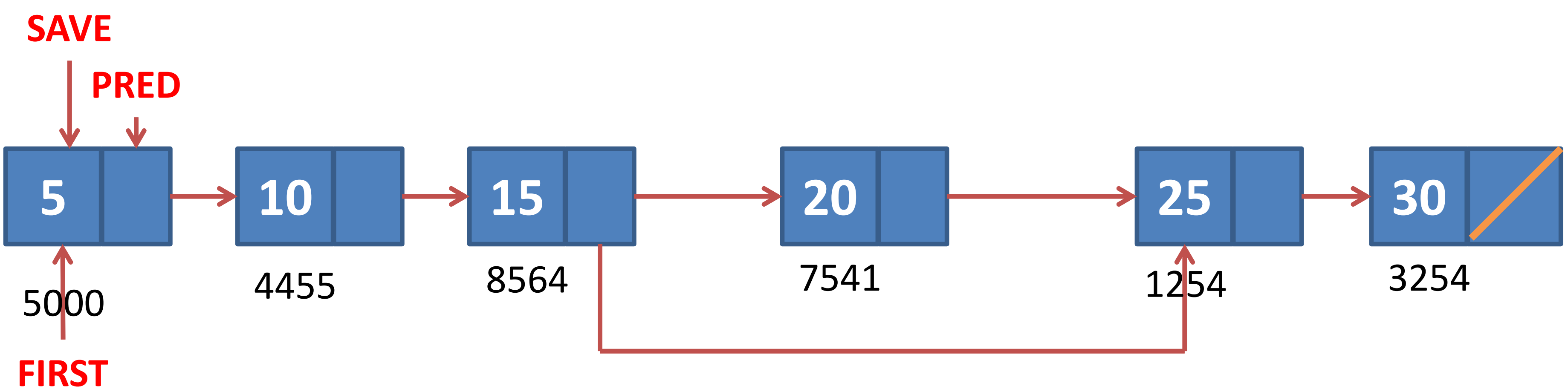
```
1. [Is Empty list?]
IF      FIRST = NULL
THEN    write ('Underflow')
         Return

2. [Initialize search for X]
     SAVE ← FIRST

3. [Find X]
Repeat thru step-5
   while SAVE ≠ X and
   LINK (SAVE) ≠ NULL

[Update predecessor marker]
     PRED ← SAVE
[Move to next node]
     SAVE ← LINK(SAVE)
```

```
4. [End of the list?]
If      SAVE ≠ X
THEN    write ('Node not found')
         Return

5. [Delete X]
If   X = FIRST
THEN    FIRST ← LINK(FIRST)
ELSE    LINK (PRED) ← LINK (X)

8. [Free Deleted Node]
     Free (X)
```

Procedure: DELETE(7541, FIRST)

```
2. [Initialize search for X]
     SAVE ← FIRST
3. [Find X]
Repeat thru step-5
   while SAVE ≠ X and
   LINK (SAVE) ≠ NULL
 [Update predecessor marker]
     PRED ← SAVE
 [Move to next node]
     SAVE ← LINK(SAVE)
```

```
4. [End of the list?]
If      SAVE ≠ X
THEN    write ('Node not found')
         Return
5. [Delete X]
If   X = FIRST
THEN    FIRST ← LINK(FIRST)
ELSE    LINK (PRED) ← LINK (X)
6. [Free Deleted Node]
     Free (X)
```



Function: COUNT_NODES(FIRST)

- This function **counts** number of nodes of the linked list and returns **COUNT**.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **SAVE** is a Temporary pointer variable.

Function: COUNT_NODES(FIRST) Cont...

```
1. [Is list Empty?]
   IF     FIRST = NULL
   Then   COUNT ← 0
          Return(COUNT)

2. [Initialize loop for a last node to update count]
    SAVE← FIRST

3. [Go for end of list]
   Repeat while LINK (SAVE) ≠ NULL
       SAVE ← LINK (SAVE)
       COUNT ← COUNT + 1

4. [Return Count]
    Return (COUNT)
```

Function: COPY (FIRST)

- This function **Copy** a Link List and creates new Linked List
- This function returns address of first node of newly created linked list.
- The **new list** is to contain **nodes** whose **information** and **pointer** fields are denoted by **FIELD** and **PTR**, respectively.
- The address of the **first node** in the newly created list is to be placed in **BEGIN**
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW, SAVE and PRED** are temporary pointer variables.

Function: COPY (FIRST)

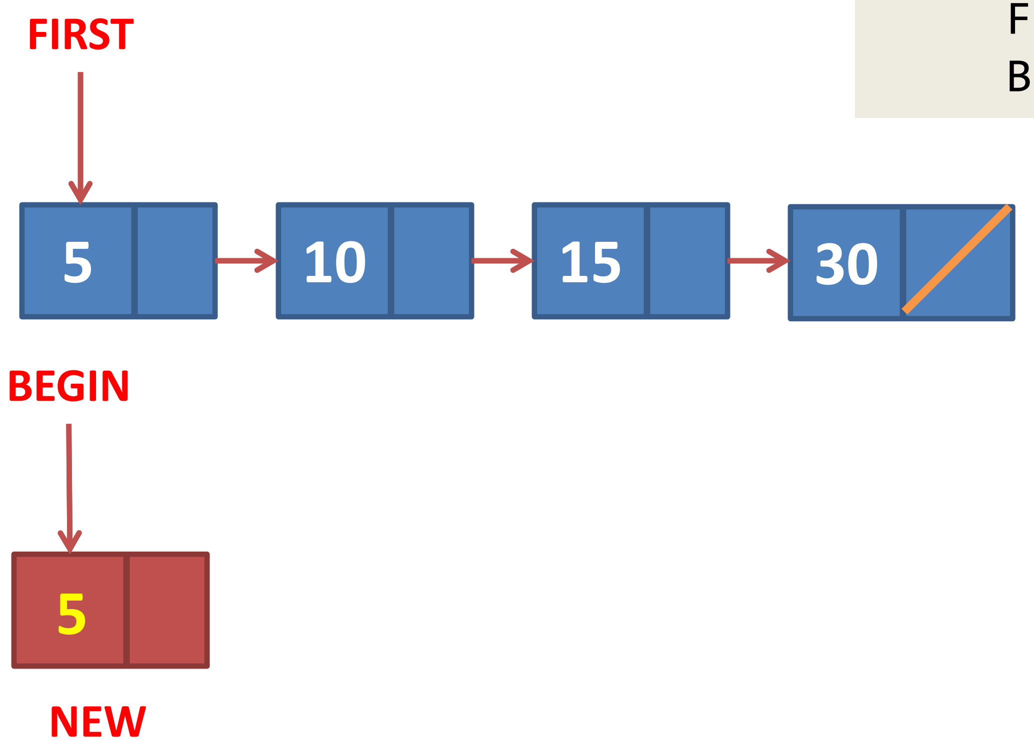
```
1. [Is Empty list?]
   IF    FIRST = NULL
   THEN  Return(NULL)
2. [Copy first node]
IF   AVAIL = NULL
THEN write ('Underflow')
     Return (NULL)
ELSE NEW←AVAIL
     AVAIL←LINK(AVAIL)
     FIELD(NEW)←INFO(FIRST)
     BEGIN ←NEW
3. [Initialize Traversal]
   SAVE ← FIRST
4. [Move the next node if not at
    the end if list]
   Repeat thru step 6
      while LINK(SAVE) ≠ NULL
```

```
5. [Update predecessor and
    save pointer]
   PRED←NEW
   SAVE←LINK(SAVE)
6. [Copy Node]
IF   AVAIL = NULL
THEN write ('Underflow')
     Return (NULL)
ELSE NEW ← AVAIL
     AVAIL ← LINK(AVAIL)
     FIELD(NEW)←INFO(SAVE)
     PTR(PRED)←NEW
7. [Set link of last node and
   return]
   PTR(NEW) ← NULL
   Return(BEGIN)
```

Function: COPY (FIRST)

```
1. [Is Empty list?]
   IF      FIRST = NULL
   THEN    Return(NULL)
```

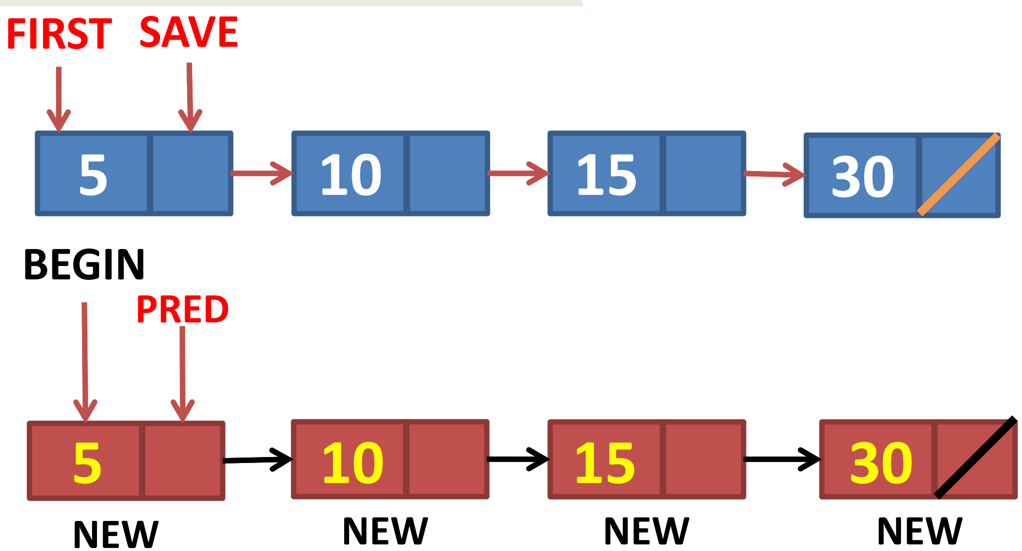
```
2. [Copy first node]
IF    AVAIL = NULL
THEN  write ('Underflow')
      Return (0)
ELSE  NEW←AVAIL
      AVAIL←LINK(AVAIL)
      FIELD(NEW)←INFO(FIRST)
      BEGIN ←NEW
```



Function: COPY (FIRST)

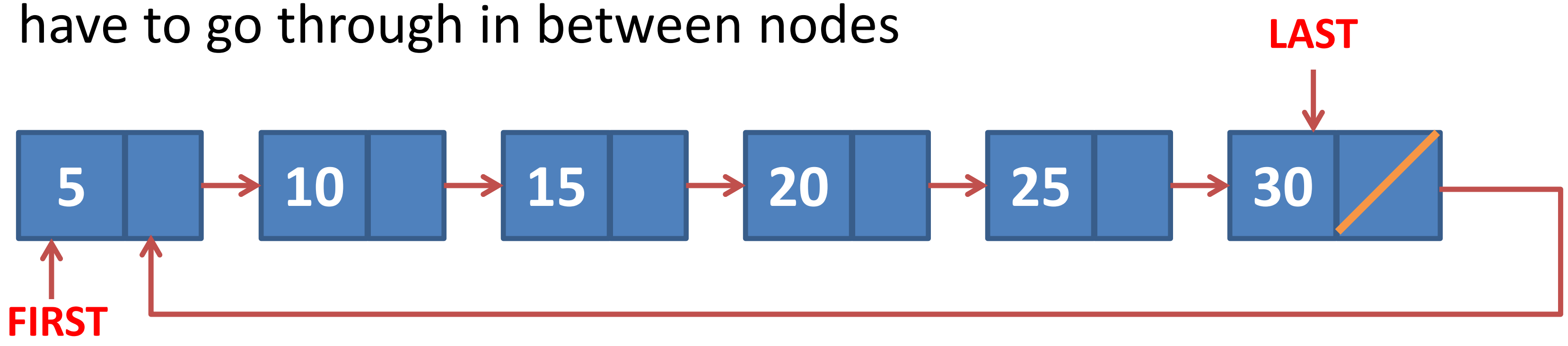
```
3. [Initialize Traversal]
   SAVE ← FIRST
4. [Move the next node if not at
    the end if list]
   Repeat thru step 6
      while LINK(SAVE) ≠ NULL
5. [Update predecessor and
    save pointer]
   PRED←NEW
   SAVE←LINK(SAVE)
```

```
6. [Copy Node]
IF    AVAIL = NULL
THEN write (‘Underflow’)
     Return (0)
ELSE NEW ← AVAIL
     AVAIL ← LINK(AVAIL)
     FIELD(NEW)←INFO(SAVE)
     PTR(PRED)←NEW
7. [Set link of last node & return]
   PTR(NEW) ← NULL
   Return(BEGIN)
```



Circularly Linked Linear List

- If we **replace NULL** pointer of the **last node** of Singly Linked Linear List with the **address of its first node**, that list becomes circularly linked linear list or **Circular List**.
- **FIRST** is the address of first node of Circular List
- **LAST** is the address of the last node of Circular List
- **Advantages of Circular List**
  - In circular list, every node is accessible from given node
  - It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes



Circularly Linked Linear List Cont…

- **Disadvantages of Circular List**
  - It is not easy to reverse the linked list
  - If proper care is not taken, then the problem of infinite loop can occur
  - If we at a node and go back to the previous node, then we can not do it in single step. Instead we have to complete the entire circle by going through the in between nodes and then we will reach the required node

Operations on Circular List

- Insert at First
- Insert at Last
- Insert in Ordered List
- Delete a node

Procedure: CIR_INS_FIRST(X,FIRST,LAST)

- This procedure **inserts a new node at the first position** of Circular linked list.
- **X** is a new element to be inserted.
- **FIRST** and **LAST** are a **pointer to the first & last elements** of a Circular linked linear list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **NEW** is a temporary pointer variable.

Procedure: CIR_INS_FIRST(X,FIRST,LAST)

```
1. [Creates a new empty node]
   NEW ⇐ NODE
2. [Initialize fields of new node and its link]
   INFO (NEW) ← X

   IF   FIRST = NULL
   THEN LINK (NEW) ← NEW
        FIRST ← LAST ← NEW
   ELSE LINK (NEW) ← FIRST
        LINK (LAST) ← NEW
        FIRST ← NEW

   Return
```

Procedure: CIR_INS_FIRST(X,FIRST,LAST)

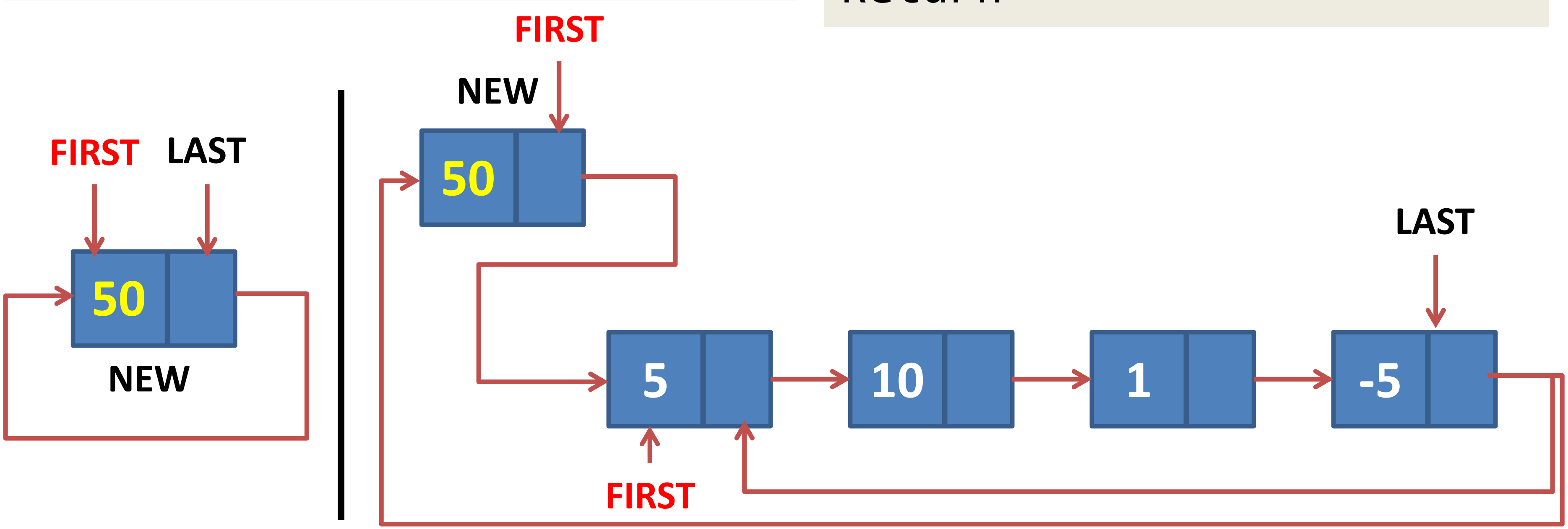
1. [Creates a new empty node]

   NEW ⇐ NODE

2. [Initialize fields of new node and its link]

INFO (NEW) ← X

```
IF   FIRST = NULL
THEN LINK(NEW)←NEW
     FIRST←LAST←NEW
ELSE LINK(NEW)←FIRST
     LINK(LAST)←NEW
     FIRST←NEW
Return
```



Procedure: CIR_INS_LAST(X,FIRST,LAST)

- This procedure **inserts a new node at the last position** of Circular linked list.
- **X** is a new element to be inserted.
- **FIRST** and **LAST** are a **pointer to the first & last elements** of a Circular linked linear list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **NEW** is a temporary pointer variable.

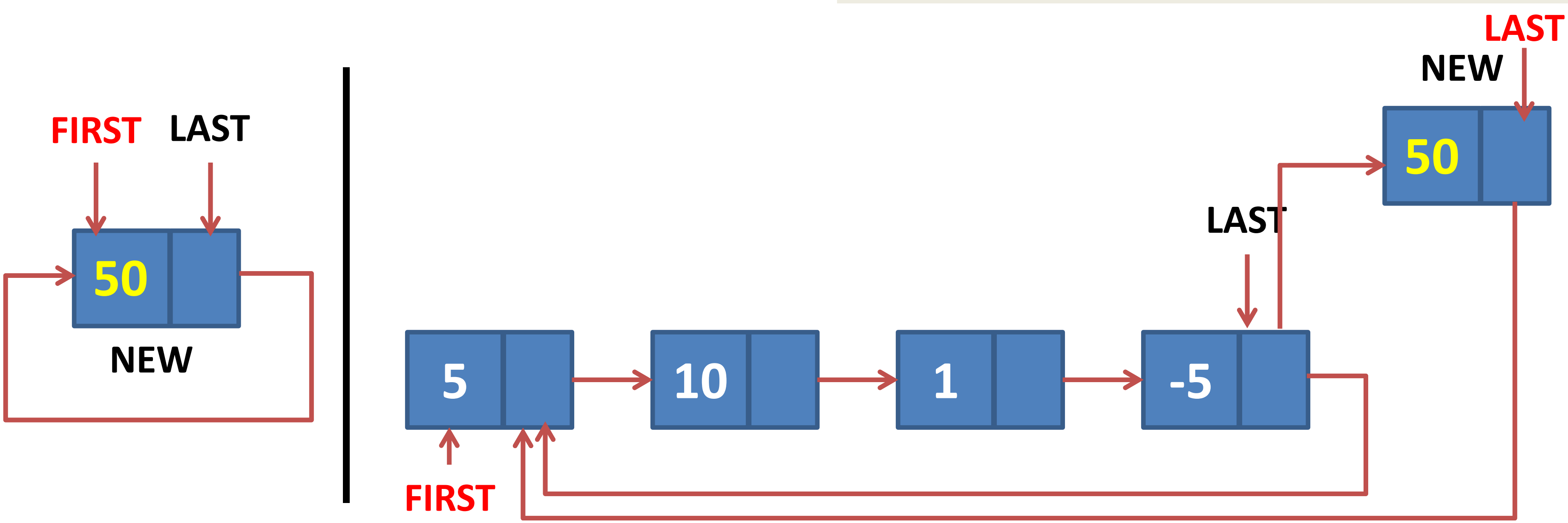
Procedure: CIR_INS_LAST(X,FIRST,LAST)

```
1. [Creates a new empty node]
   NEW ⇐ NODE
2. [Initialize fields of new node and its link]
   INFO (NEW) ← X

   IF   FIRST = NULL
   THEN LINK (NEW) ← NEW
        FIRST ← LAST ← NEW
   ELSE LINK (NEW) ← FIRST
        LINK (LAST) ← NEW
        LAST ← NEW

   Return
```

Procedure: CIR_INS_LAST(X,FIRST,LAST)

```
1. [Creates a new empty
   node]
    NEW ⇐ NODE
2. [Initialize fields of
new node and its link]
INFO (NEW) ← X
```

```
IF    FIRST = NULL
THEN  LINK(NEW)←NEW
      FIRST←LAST←NEW
ELSE  LINK(NEW)←FIRST
      LINK(LAST)←NEW
      LAST←NEW
Return
```



Procedure: CIR_INS_ORD(X,FIRST,LAST)

- This function **inserts** a new node such that linked list preserves the ordering of the terms in **increasing order** of their **INFO** field.
- **X** is a new element to be inserted.
- **FIRST** and **LAST** are a **pointer to the first & last elements** of a Circular linked linear list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **NEW** is a temporary pointer variable.

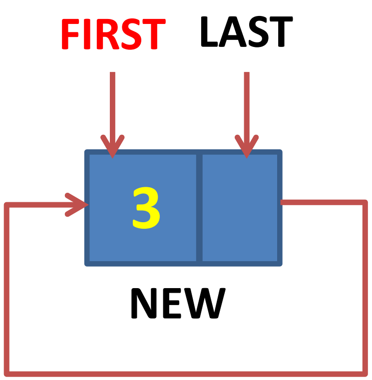
Procedure: CIR_INS_ORD(X,FIRST,LAST)

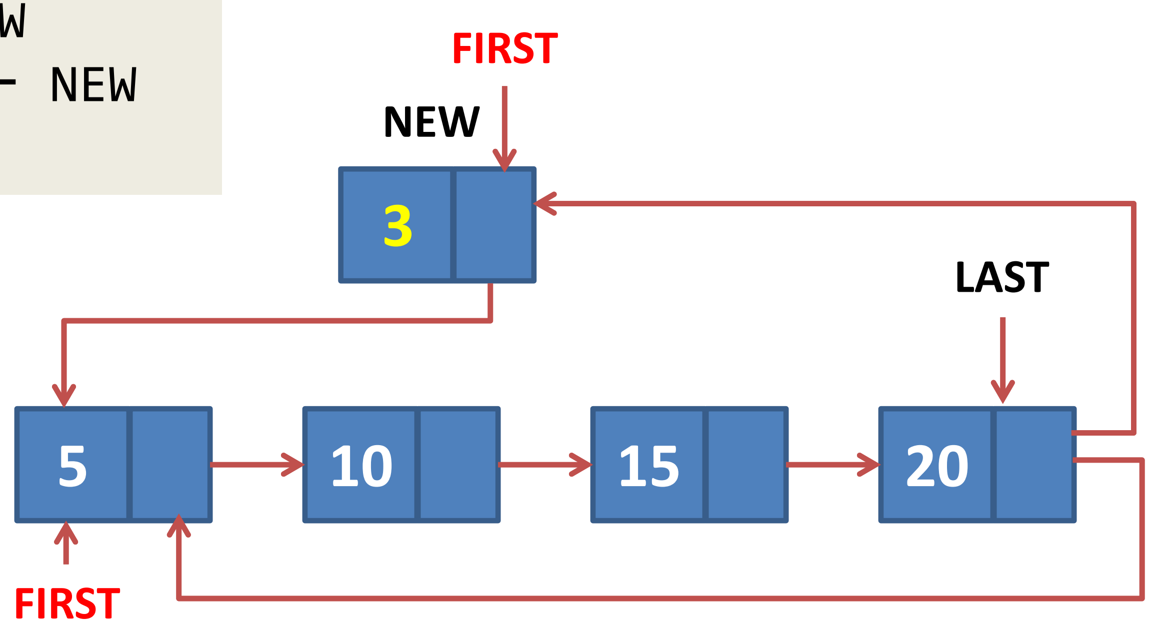
```
1. [Create New Empty Node]
   NEW ⇐ NODE
2. [Copy information content
    into new node]
   INFO(NEW) ← X
3. [Is Linked List Empty?]
   IF   FIRST = NULL
   THEN LINK(NEW) ← NEW
        FIRST ← LAST ← NEW
        Return
4. [Does new node precedes all
    other nodes in List?]
IF    INFO(NEW)≤ INFO(FIRST)
THEN  LINK(NEW) ← FIRST
      LINK(LAST) ← NEW
      FIRST ← NEW
      Return
5. [Initialize Temporary
    Pointer]
   SAVE ← FIRST
6. [Search for Predecessor of
    new node]
Repeat while SAVE ≠ LAST &
  INFO(NEW) ≥ INFO(LINK(SAVE))
      SAVE←LINK(SAVE)
7. [Set link field of NEW
    node and its Predecessor]
   LINK(NEW) ← LINK(SAVE)
   LINK(SAVE) ← NEW

   IF   SAVE = LAST
   THEN LAST ← NEW
8. [Finished]
   Return
```

Procedure: CIR_INS_ORD(3,FIRST,LAST)

```
1. [Create New Empty Node]
   NEW ⇐ NODE
2. [Copy information content
    into new node]
   INFO(NEW) ← X
3. [Is Linked List Empty?]
   IF    FIRST = NULL
   THEN LINK(NEW) ← NEW
        FIRST ← LAST ← NEW
        Return
```

```
4. [Does new node precedes all
    other nodes in List?]
IF     INFO(NEW)≤ INFO(FIRST)
THEN   LINK(NEW) ← FIRST
       LINK(LAST) ← NEW
       FIRST ← NEW
       Return
```



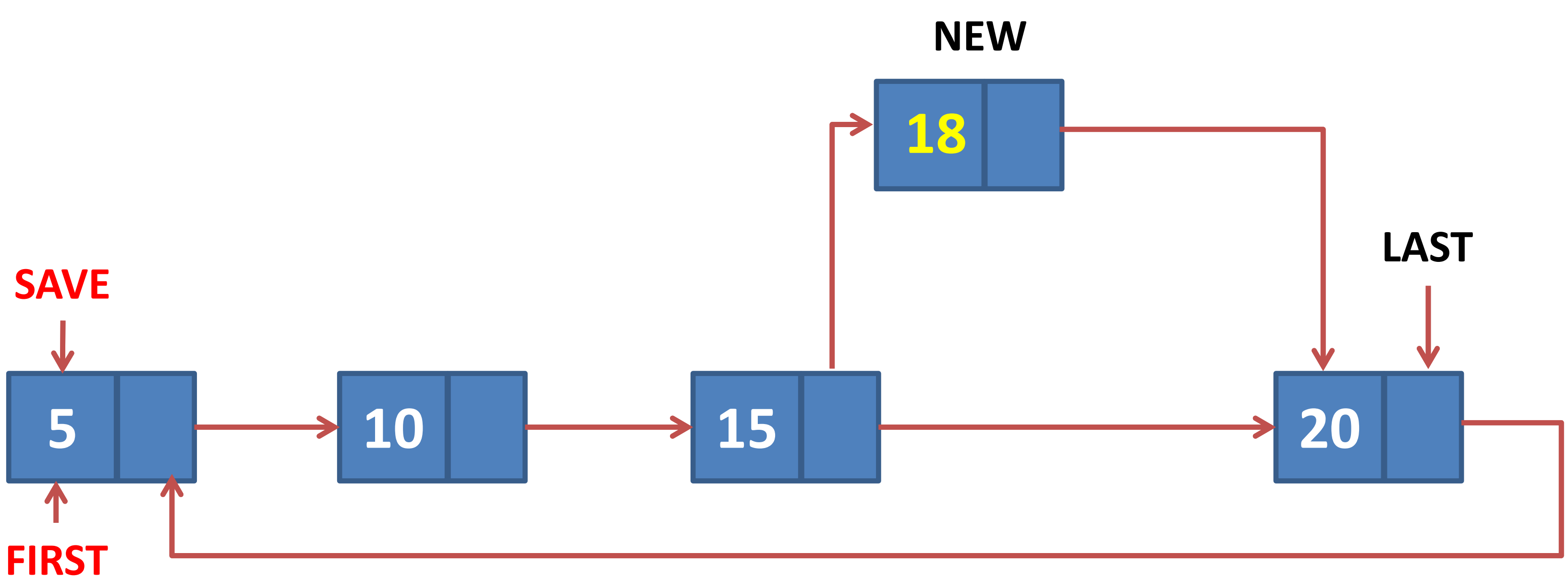


Procedure: CIR_INS_ORD(18,FIRST,LAST)

5. [Initialize Temporary Pointer]
   SAVE ← FIRST
6. [Search for Predecessor of new node]

Repeat while SAVE ≠ LAST & INFO(NEW) ≥ INFO(LINK(SAVE))
SAVE←LINK(SAVE)

7. [Set link field of NEW node and its Predecessor]
   LINK(NEW) ← LINK(SAVE)
   LINK(SAVE) ← NEW
   IF SAVE = LAST
   THEN LAST ← NEW
8. [Finished]
   Return



Procedure: CIR_DELETE(X,FIRST,LAST)

- This algorithm **delete** a node whose address is given by variable **X**.
- **FIRST** & **LAST** are **pointers to the First & Last elements** of a Circular linked list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **SAVE & PRED** are temporary pointer variable.

Procedure: CIR_DELETE(X,FIRST,LAST)

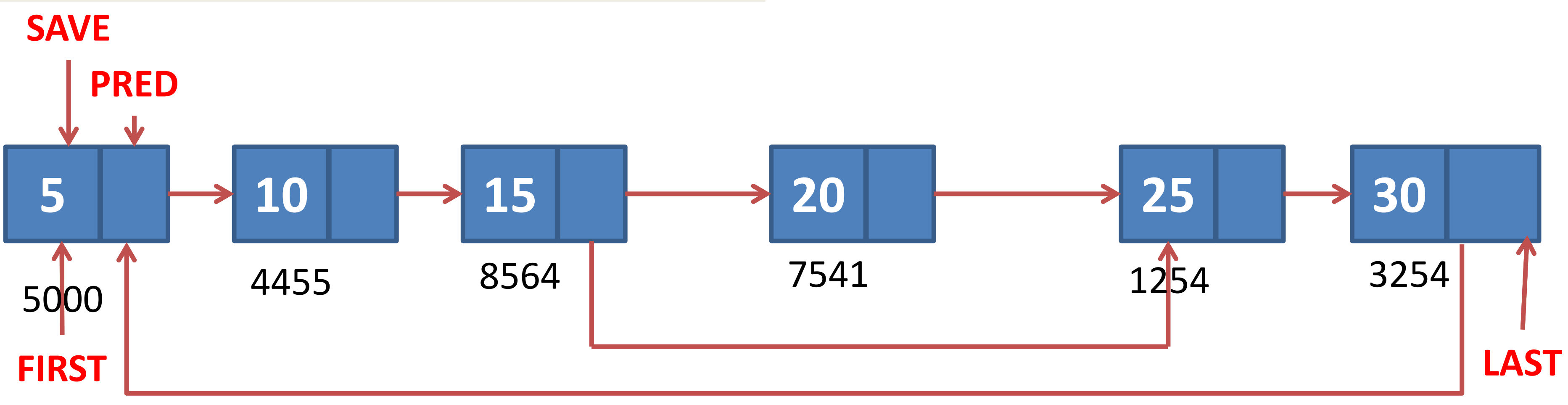
```
1. [Is Empty List?]
IF   FIRST = NULL
THEN write('Linked List is
     Empty')
     Return
2. [Initialize Search for X]
   SAVE ← FIRST
3. [Find X]
Repeat thru step 5
   while SAVE≠X & SAVE≠LAST
4. [Update predecessor marker]
   PRED ← SAVE
5. [Move to next node]
   SAVE ← LINK(SAVE)
```

```
6. [End of Linked List?]
IF      SAVE ≠ X
THEN    write('Node not found')
        return
7. [Delete X]
IF      X = FIRST
THEN    FIRST←LINK(FIRST)
        LINK(LAST)←FIRST
ELSE    LINK(PRED)←LINK(X)
        IF      X = LAST
        THEN    LAST ← PRED
8. [Free Deleted Node]
   Free (X)
```

Procedure: CIR_DELETE(7541,FIRST,LAST)

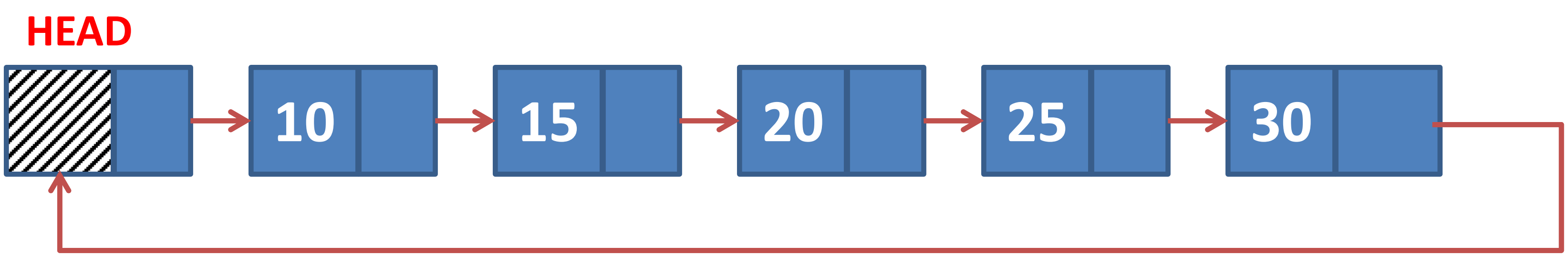
```
1. [Is Empty List?]
IF   FIRST = NULL
THEN write('Linked List is
     Empty')
     Return
2. [Initialize Search for X]
   SAVE ← FIRST
3. [Find X]
Repeat thru step5 while SAVE≠X & SAVE≠LAST
4. [Update predecessor marker]
   PRED ← SAVE
5. [Move to next node]
   SAVE ← LINK(SAVE)
```

```
6. [End of Linked List?]
IF      SAVE ≠ X
THEN    write('Node not found')
        return
7. [Delete X]
IF      X = FIRST
THEN    FIRST←LINK(FIRST)
        LINK(LAST)←FIRST
ELSE    LINK(PRED)←LINK(X)
        IF      X = LAST
        THEN    LAST ← PRED
8. [Free Deleted Node]
   Free (X)
```



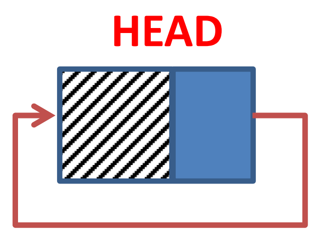
Circularly Linked List with Header Node

- We can have special node, often referred to as **Head node** of Circular Linked List.
- Head node does not have any value.
- Head node is always pointing to the first node if any of the linked list.
- One advantage of this technique is Linked list is never be empty.
- Pointer variable **HEAD** contains the address of head node.



Empty List

LINK(HEAD) ← HEAD

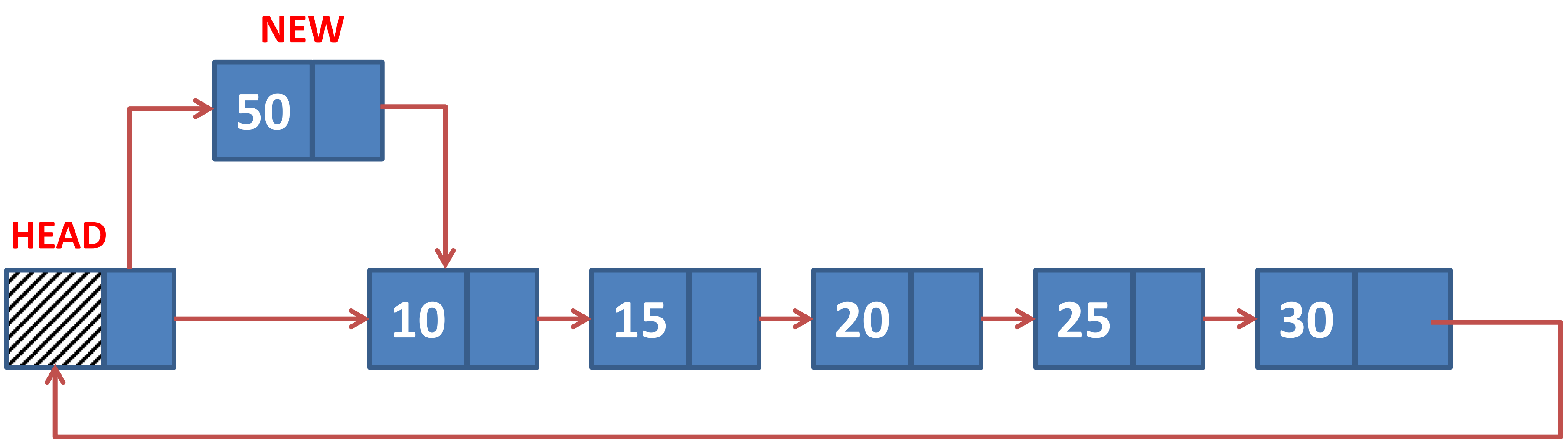


Procedure: CIR_HEAD_INS_FIRST(X,FIRST,LAST)

- This procedure **inserts a new node at the first position** of Circular linked list with Head node.
- **X** is a new element to be inserted.
- **FIRST** and **LAST** are a **pointer to the first & last elements** of a Circular linked linear list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **HEAD** is pointer variable pointing to Head node of Linked List.
- **NEW** is a temporary pointer variable.

Procedure: CIR_HEAD_INS_FIRST(X,FIRST,LAST)

```
1. [Create New Empty Node]
   NEW ⇐ NODE
2. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← LINK(HEAD)
   LINK(HEAD) ← NEW
```

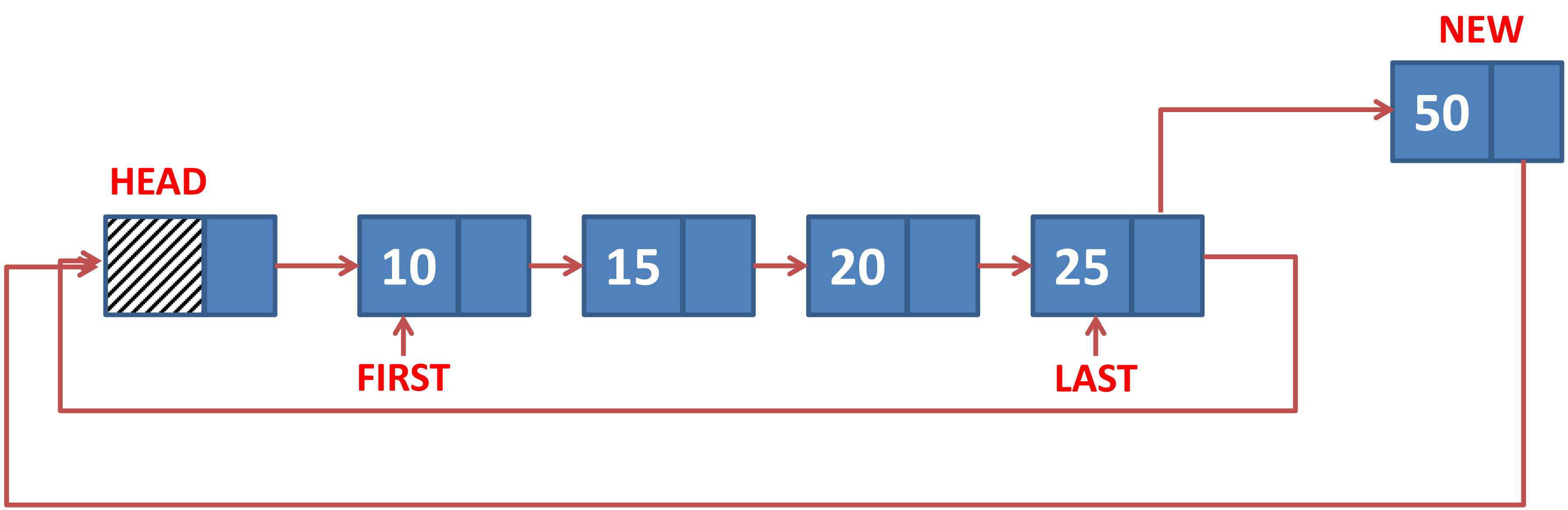


Procedure: CIR_HEAD_INS_LAST(X,FIRST,LAST)

- This procedure **inserts a new node at the last position** of Circular linked list with Head node.
- **X** is a new element to be inserted.
- **FIRST** and **LAST** are a **pointer to the first & last elements** of a Circular linked linear list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **HEAD** is pointer variable pointing to Head node of Linked List.
- **NEW** is a temporary pointer variable.

Procedure: CIR_HEAD_INS_LAST(X,FIRST,LAST)

```
1. [Create New Empty Node]
   NEW ⇐ NODE
2. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← HEAD
   LINK(LAST) ← NEW
   LAST ← NEW
```

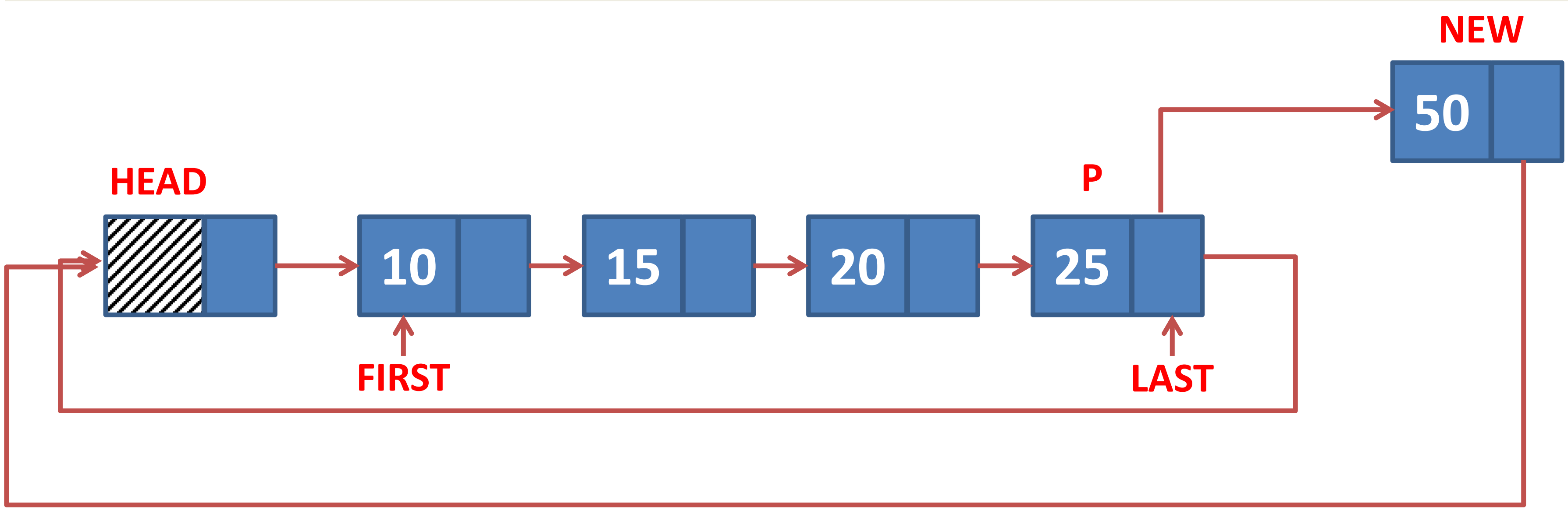


Procedure: CIR_HEAD_INS_AFTER-P (X,FIRST,LAST)

- This procedure **inserts a new node after a node whose address is given by P** of Circular linked list with Head node.
- **X** is a new element to be inserted.
- **FIRST** and **LAST** are a **pointer to the first & last elements** of a Circular linked linear list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **HEAD** is pointer variable pointing to Head node of Linked List.
- **NEW** is a temporary pointer variable.

Procedure: CIR_HEAD_INS_AFTER-P (X,FIRST,LAST)

```
1. [Create New Empty Node]
   NEW ⇐ NODE
2. [Initialize fields of new node and its link to the list]
   INFO(NEW) ← X
   LINK(NEW) ← LINK(P)
   LINK(P) ← NEW
   IF   P = LAST
   THEN LAST ← NEW
```



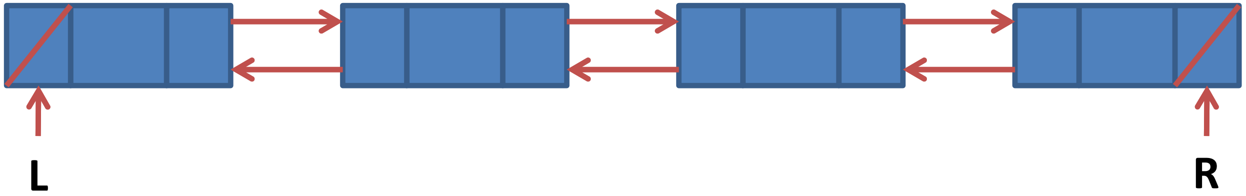
Doubly Linked Linear List

- In certain Applications, it is very desirable that a list be traversed in either forward or reverse direction.
- This property implies that each node must contain two link fields instead of usual one.
- The links are used to denote **Predecessor** and **Successor** of node.
- The link denoting its **predecessor** is called **Left Link.**
- The link denoting its **successor** is called **Right Link.**
- A list containing this type of node is called **doubly linked list** or **two way chain**.

Doubly Linked Linear List

- Typical node of doubly linked linear list contains INFO, LPTR RPTR Fields
- **LPTR** is pointer variable pointing to Predecessor of a node
- **RPTR** is pointer variable pointing to Successor of a node
- Left most node of doubly linked linear list is called **L**, **LPTR** of node L **is always NULL**
- Right most node of doubly linked linear list is called **R**, **RPTR** of node **R is always NULL**

Typical node of Doubly Linked List

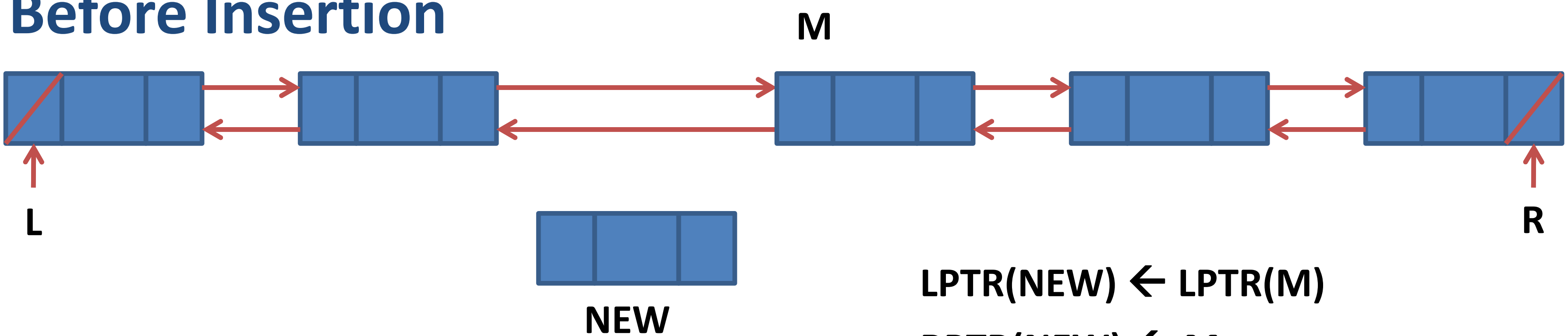


Doubly linked linear list

Insert node in Doubly Linked List

Insertion in the middle of Doubly Linked Linear List

Before Insertion



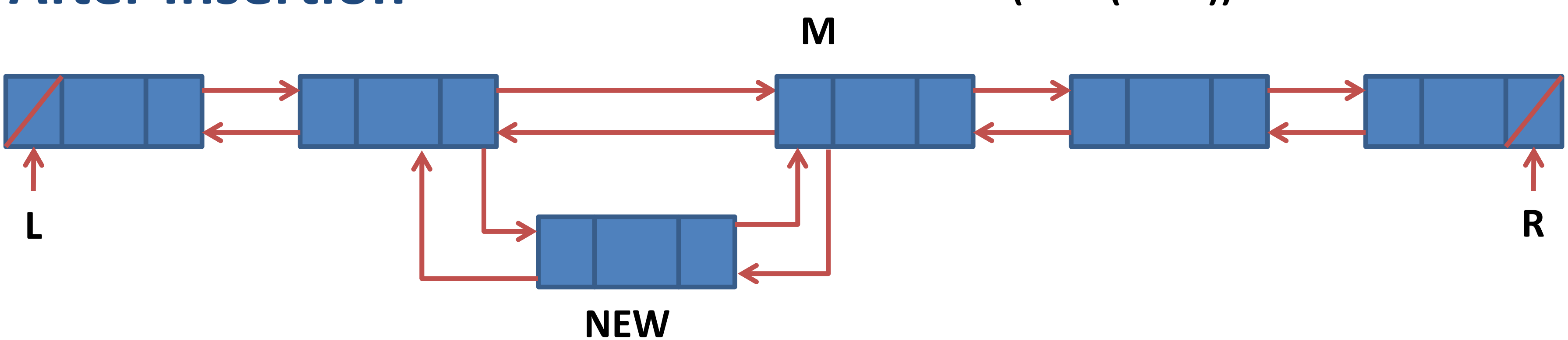
LPTR(NEW) ← LPTR(M)

RPTR(NEW) ← M

LPTR(M) ← NEW

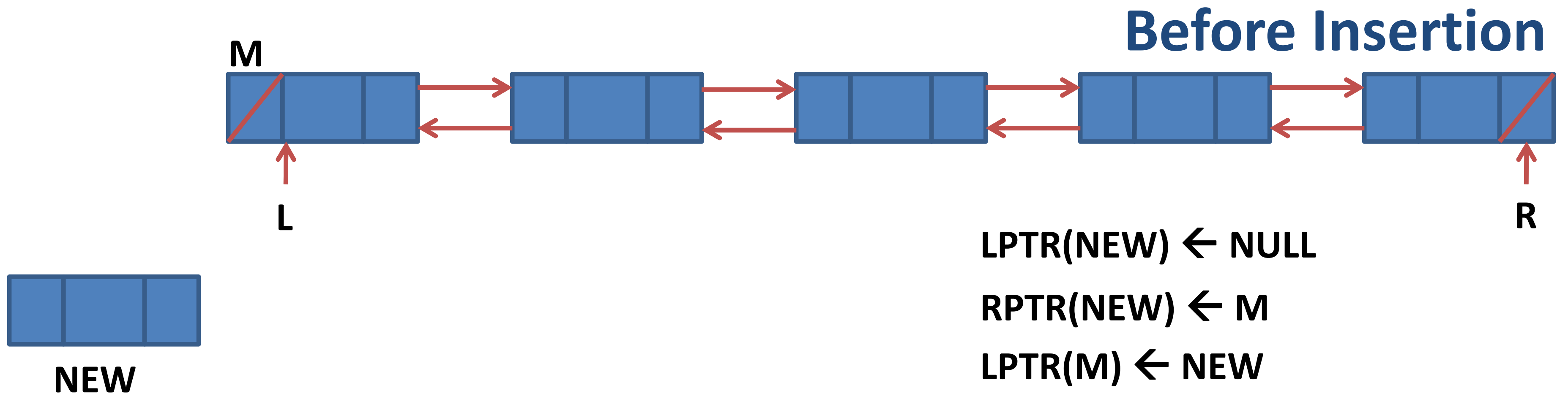
RPTR(LPTR(NEW)) ← NEW

After Insertion



Insert node in Doubly Linked List

Left most insertion in Doubly Linked Linear List

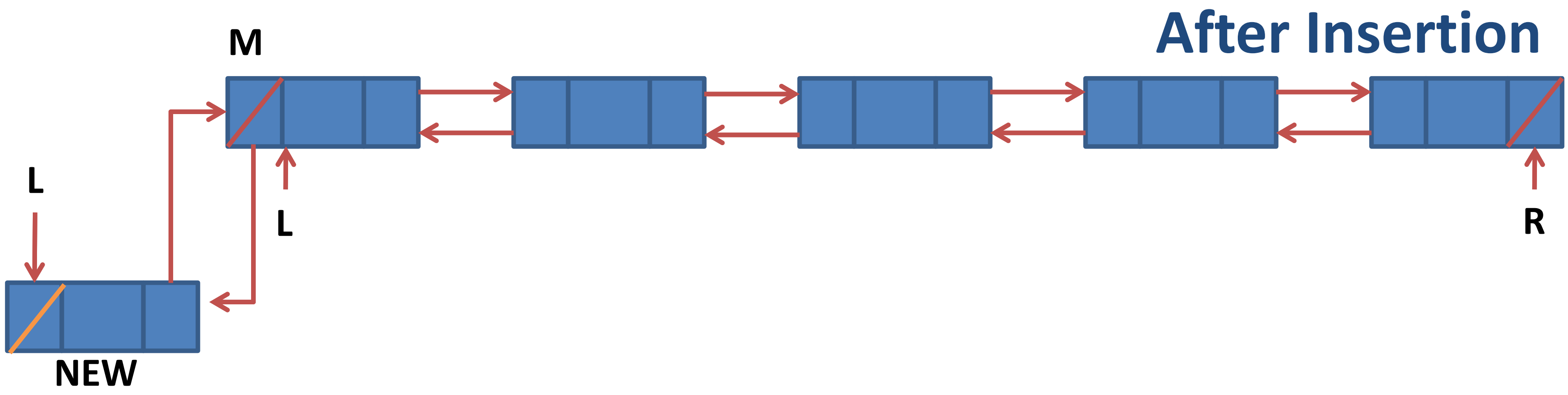


LPTR(NEW) ← NULL

RPTR(NEW) ← M

LPTR(M) ← NEW

L ← NEW

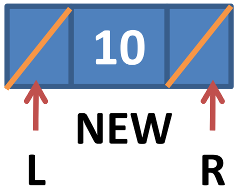


Procedure: DOU_INS (L,R,M,X)

- This algorithm inserts a new node in doubly linked linear list.
- The **insertion** is to be **performed** to the **left of a specific node with** its **address** given by the pointer variable **M**
- Typical node of doubly linked list contains following fields **LPTR**, **RPTR** and **INFO**
- **LPTR** is pointer variable pointing to Predecessor of a node
- **RPTR** is pointer variable pointing to Successor of a node
- **L** & **R** are pointer variables pointing for Leftmost and Rightmost node of Linked List.
- **NEW** is the address of New Node
- **X** is value to be inserted

Procedure: DOU_INS (L,R,M,X)

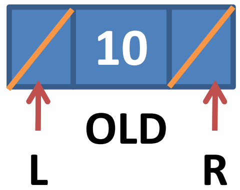
```
1. [Create New Empty Node]
    NEW ⇐ NODE
2. [Copy information field]
    INFO(NEW) ← X
3. [Insert into an empty
    list]
   IF   R = NULL
   THEN LPTR(NEW) ← NULL
        RPTR(NEW) ← NULL
        L ← R ← NEW
        Return
4. [Is left most insertion ?]
   IF   M = L
   THEN LPTR(NEW) ← NULL
        RPTR(NEW) ← M
        LPTR(M)← NEW
        L ← NEW
        Return

5. [Insert in middle]
    LPTR(NEW) ← LPTR(M)
    RPTR(NEW) ← M
    LPTR(M) ← NEW
    RPTR(LPTR(NEW)) ← NEW
    Return
```



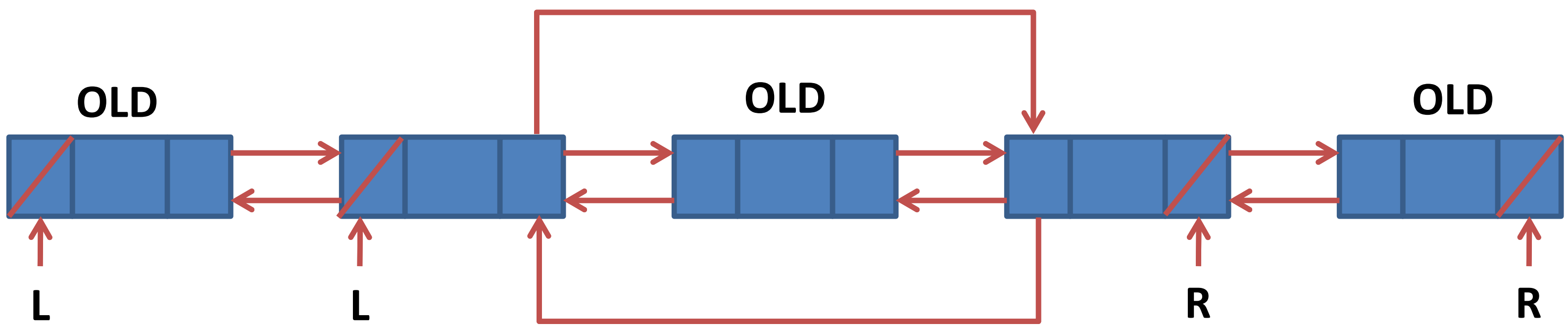
PROCEDURE: DOU _DEL (L, R, OLD)

- This algorithm **deletes the node** whose **address** is contained in the variable **OLD**
- Typical node of doubly linked list contains following fields **LPTR**, **RPTR** and **INFO**
- **LPTR** is pointer variable pointing to Predecessor of a node
- **RPTR** is pointer variable pointing to Successor of a node
- **L** & **R** are pointer variables pointing for Leftmost and Rightmost node of Linked List.

Delete from Doubly Linked List



L ← R ← NULL



L ← RPTR(L)
LPTR (L) ← NULL

RPTR(LTRP(OLD)) ← RPTR(OLD)
LPTR(RTRP(OLD)) ← LPTR(OLD)

R ← LPTR(R)
RPTR (R) ← NULL

PROCEDURE: DOU _DEL (L, R, OLD)

```
1. [Is underflow ?]
   IF   R=NULL
   THEN write ('UNDERFLOW')
        return
2. [Delete node]
IF   L = R (single node in list)
THEN L ← R ← NULL
ELSE IF    OLD = L (left most node)
     THEN  L ← RPTR(L)
           LPTR (L) ← NULL
     ELSE IF    OLD = R (right most)
          THEN  R ← LPTR (R)
                RPTR (R) ← NULL
          ELSE  RPTR(LPTR (OLD)) ← RPTR (OLD)
                LPTR(RPTR (OLD)) ← LPTR (OLD)
3. [FREE deleted node ?]
   FREE(OLD)
```