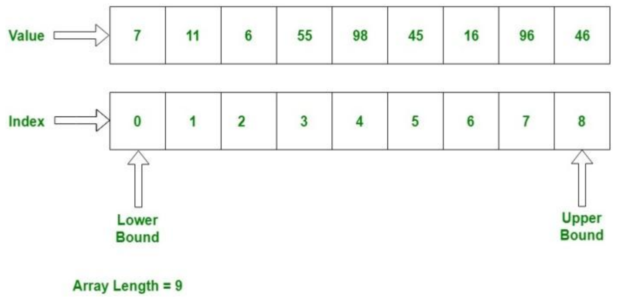
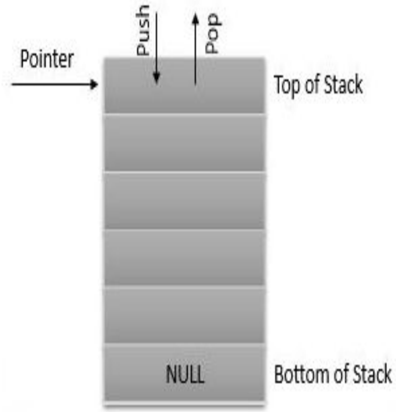
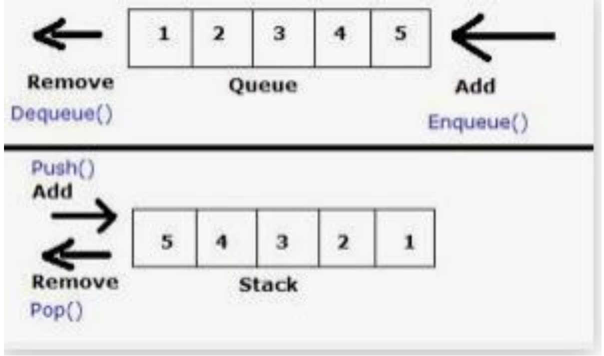


Value
7
11
6
55
98
45
16
96
46
Index
0
1
2
3
4
5
6
7
8
Lower Bound
Upper Bound
Array Length = 9

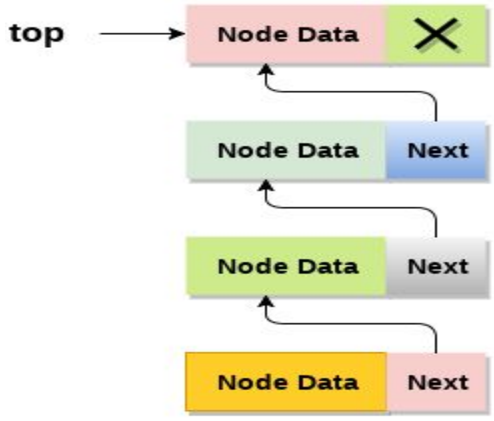


Push
Pop
Pointer
Top of Stack
NULL
Bottom of Stack

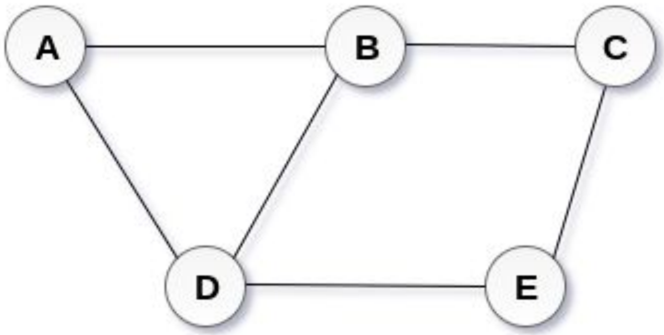


Remove
Dequeue()
1
2
3
4
5
Queue
Add
Enqueue()
Push()
Add
5
4
3
2
1
Remove
Pop()
Stack

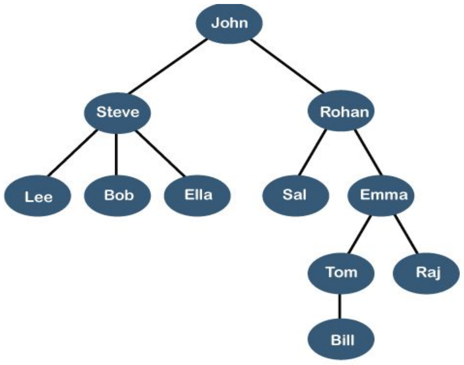
Linked List



top
Node Data
Node Data
Next
Node Data
Next
Node Data
Next



A
B
C
D
E



John
Steve
Rohan
Lee
Bob
Ella
Sal
Emma
Tom
Raj
Bill

Outline

- Introduction to Singly Linked List
- Representation of Singly Linked List using Array and Pointer
- Implementing Operations on Singly Linked List
  - Insertion as a First Node
  - Insertion as a Last Node
  - Insertion of a Node at Specific Location
  - Deletion of First Node
  - Deletion of Last Node
  - Deletion of a Desired Node
  - Searching for the Particular Element in List
  - Sorting the Linked List
  - Reversing the Linked List
  - Traversing a Linked List.

Outline

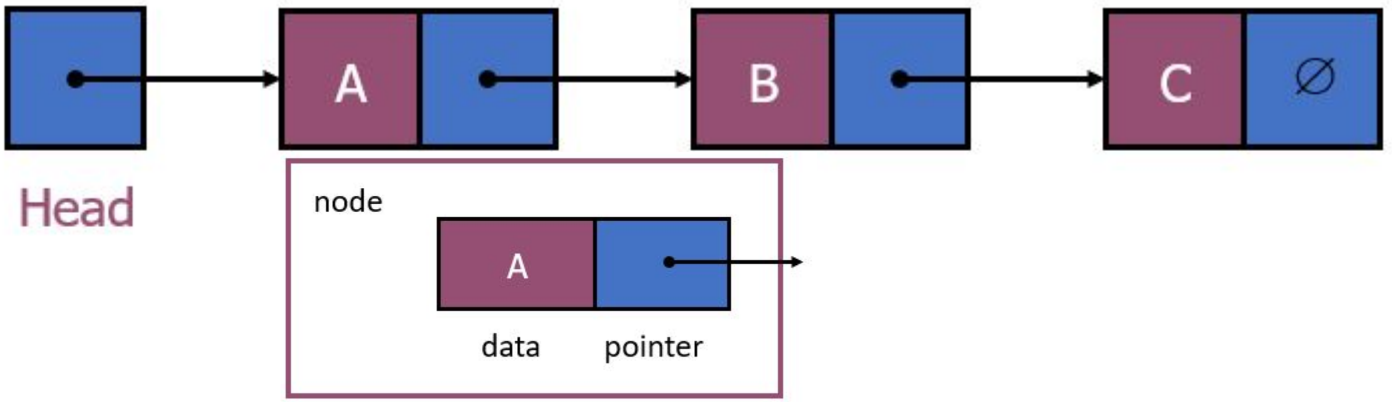
- Introduction to Circular Linked List
- Representation of Circular Linked List
- Implementing Operation of Circular Linked List
  - Inserting and Deleting a Node in Circular Linked List
  - Traversing a Circular Linked List
- Implementing Stack and Queue Operations using Singly Linked List.

Outline

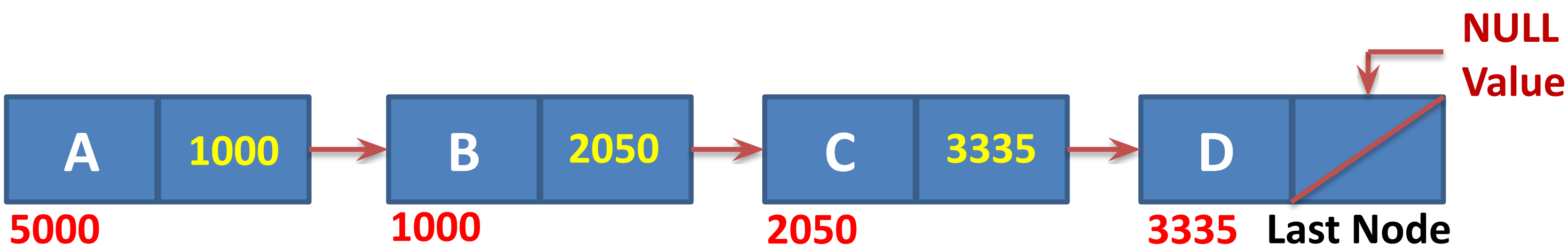
- Introduction to Doubly Linked List
- Representation of Doubly Linked List
- Implementing Operations of Doubly Linked List
  - Insertion as a First Node
  - Insertion as a Last Node
  - Insertion of a Node at Specific Location
  - Deletion of First Node
  - Deletion of Last Node
  - Deletion of a Desired Node
  - Searching for the Particular Element in Doubly Linked List
  - Sorting the Doubly Linked List
  - Traversing a Linked List.

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 mode**.
  - 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**.

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- List size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Array vs Linked List

- Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations.
- Another point of difference between an array and a linked list is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner.
- But like an array, insertions and deletions can be done at any point in the list in a constant time.
- Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array.
- For example, if we declare an array as int marks[20], then the array can store a maximum of 20 data elements only. There is no such restriction in case of a linked list.

START

1

	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	-1

START 1

1 (Biology)

2

START 2

(Computer Science)

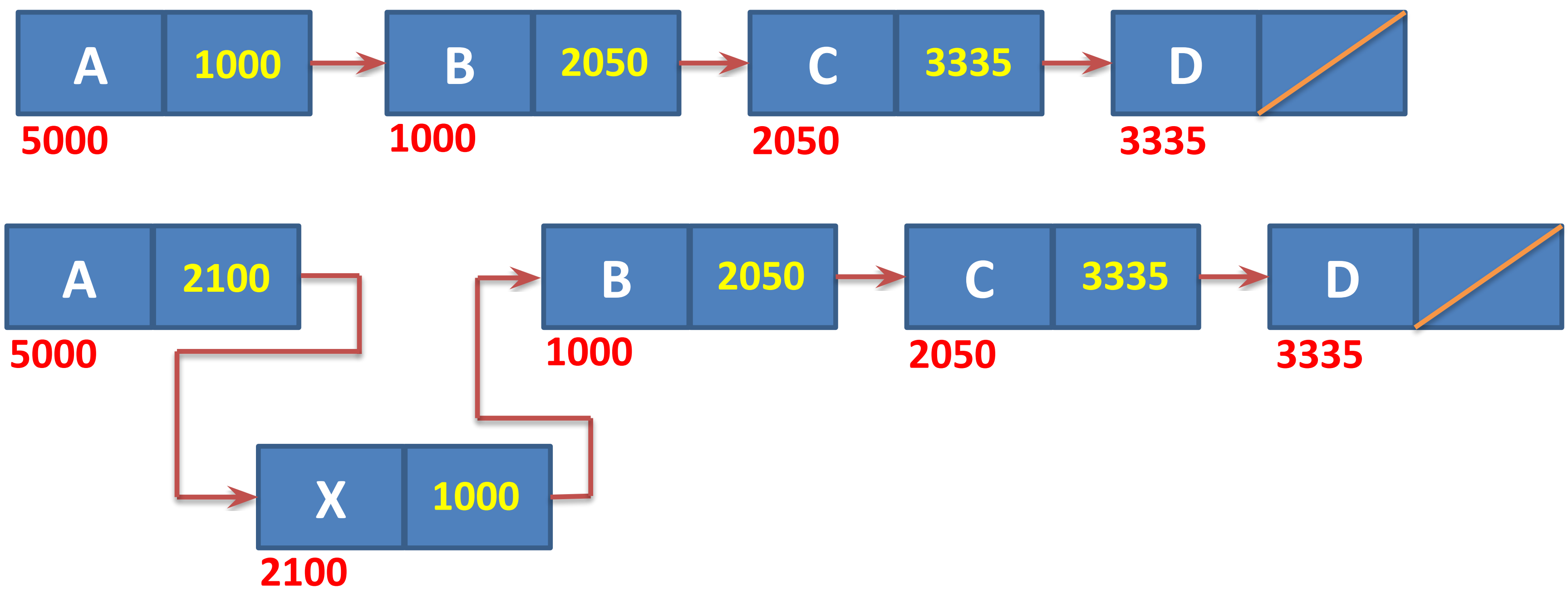
	Roll No	Next
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	-1
13	S10	15
14		
15	S11	-1

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?

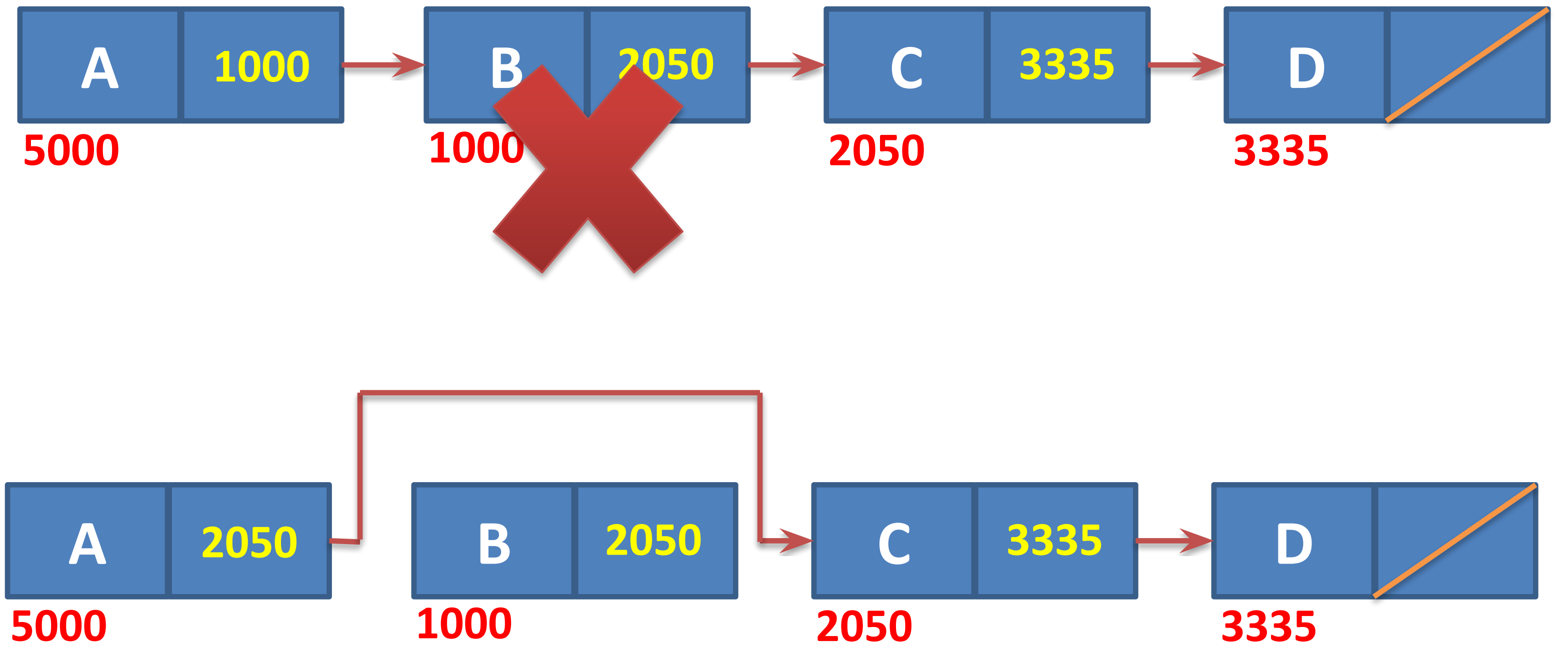
Insertion operation is more efficient in Linked allocation.



Linked Allocation

Deletion Operation

Deletion operation is more efficient in Linked Allocation



Linked Allocation

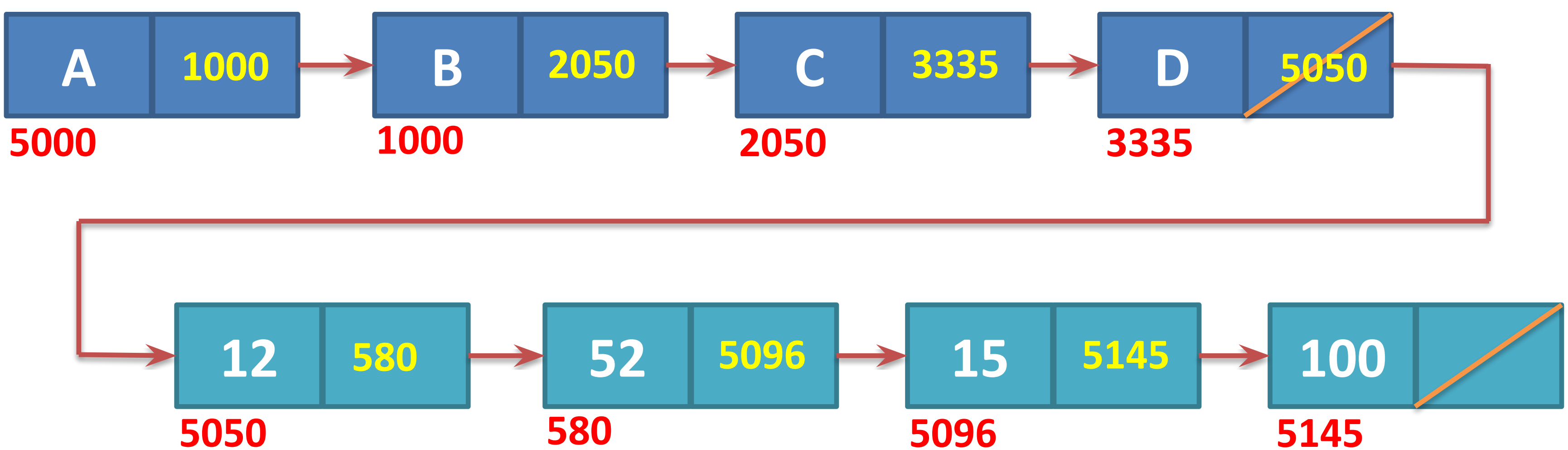
Search Operation

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

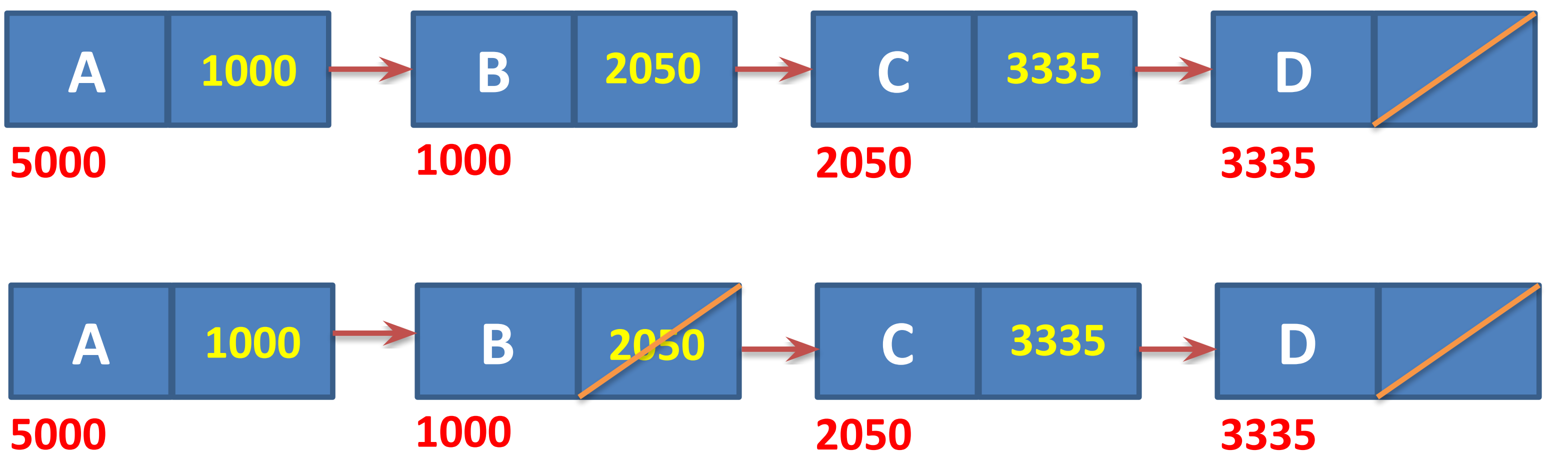
Join operation is more efficient in Linked Allocation.



Linked Allocation

Split Operation

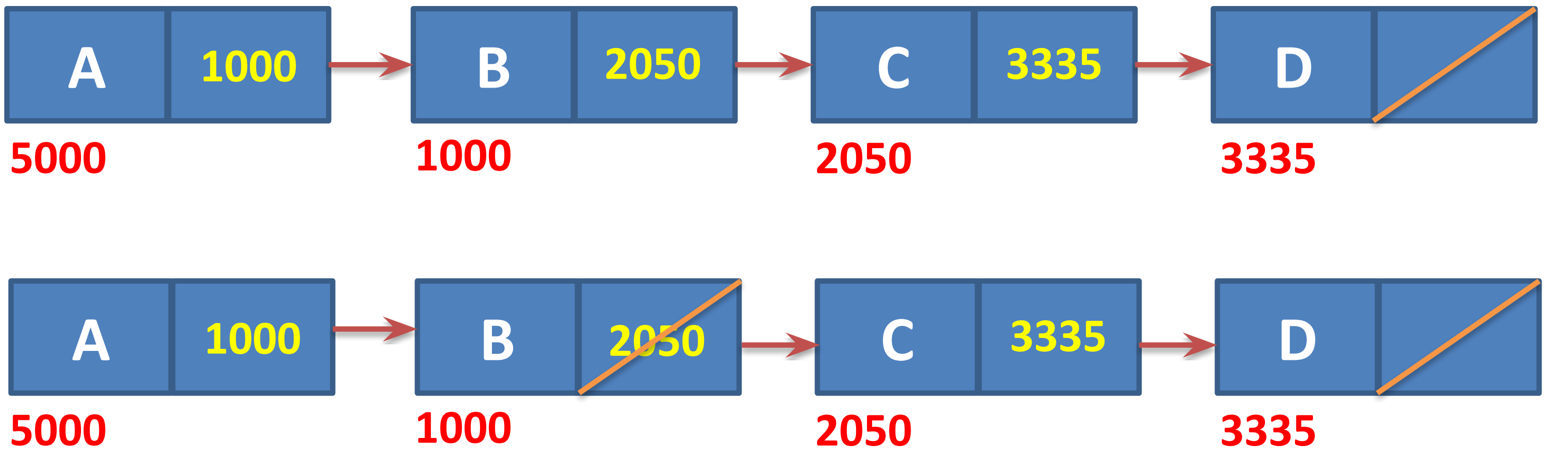
Split operation is more efficient in Linked Allocation



Linked Allocation

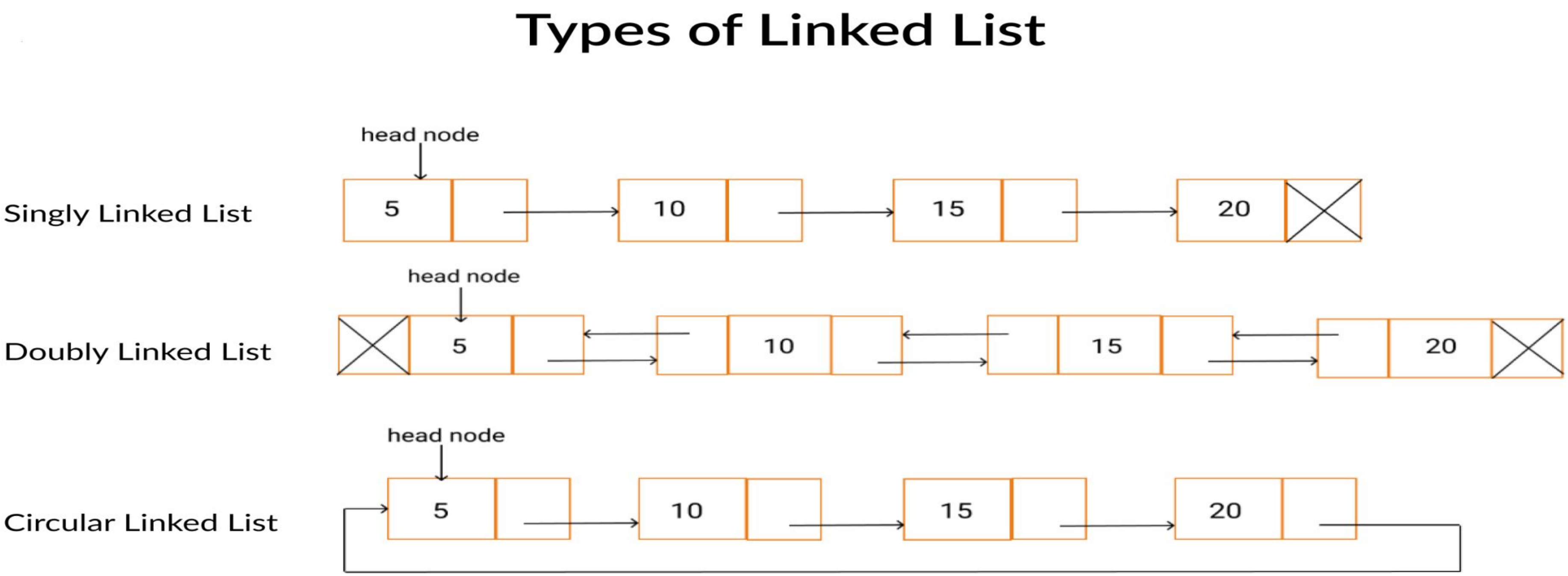
Split Operation

Split operation is more efficient in Linked Allocation



Types of Linked List

- **Simple Linked List/Singly Linked List −** Item navigation is forward only.
- **Doubly Linked List −** Items can be navigated forward and backward.
- **Circular Linked List −** Last item contains link of the first element as next and the first element has a link to the last element as previous.



Operations on Linked List

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Operations on Linked List

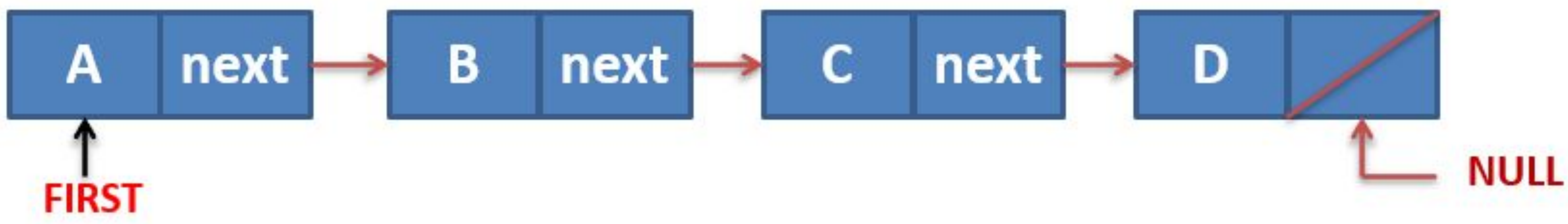
SN	Operation	Description
3	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
4	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
5	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

Operations on Linked List

SN	Operation	Description
6	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
7	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

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/Head**.
- Limitation of singly linked list is **we can traverse only in one direction**, forward direction.



Algorithms for Singly Linked List

- Insert at first position
- Insert at last position
- Insert in Ordered Linked list
- Delete Element
- Copy Linked List

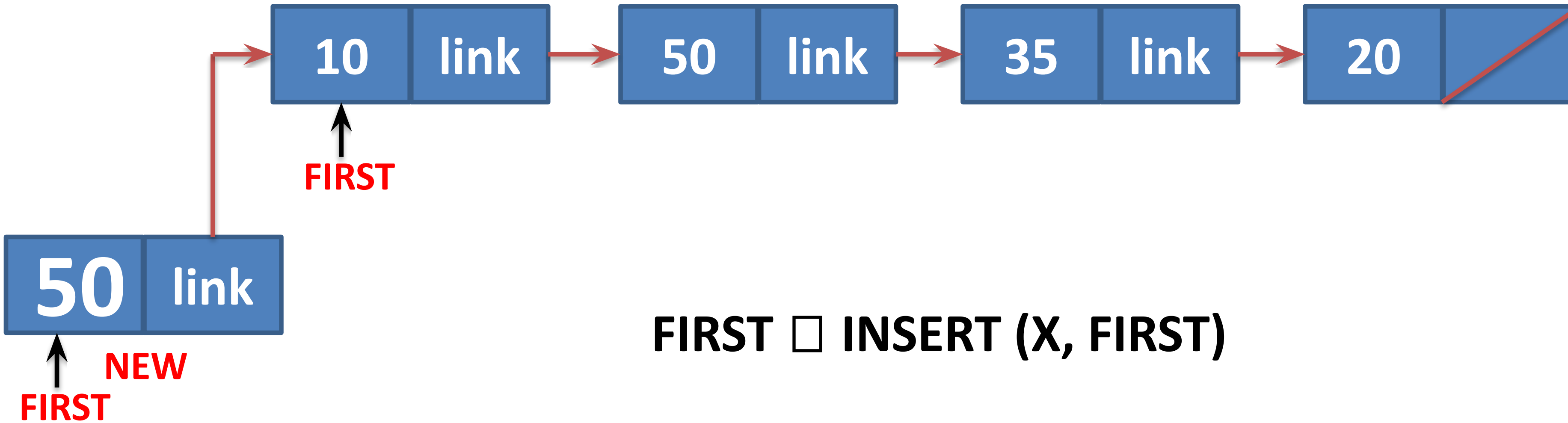
Function: INSERT(X,First)

- 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)

```
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)
```

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL NEXT
Step 4: SET DATA = VAL
Step 5: SET NEW_NODE NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

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)

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]

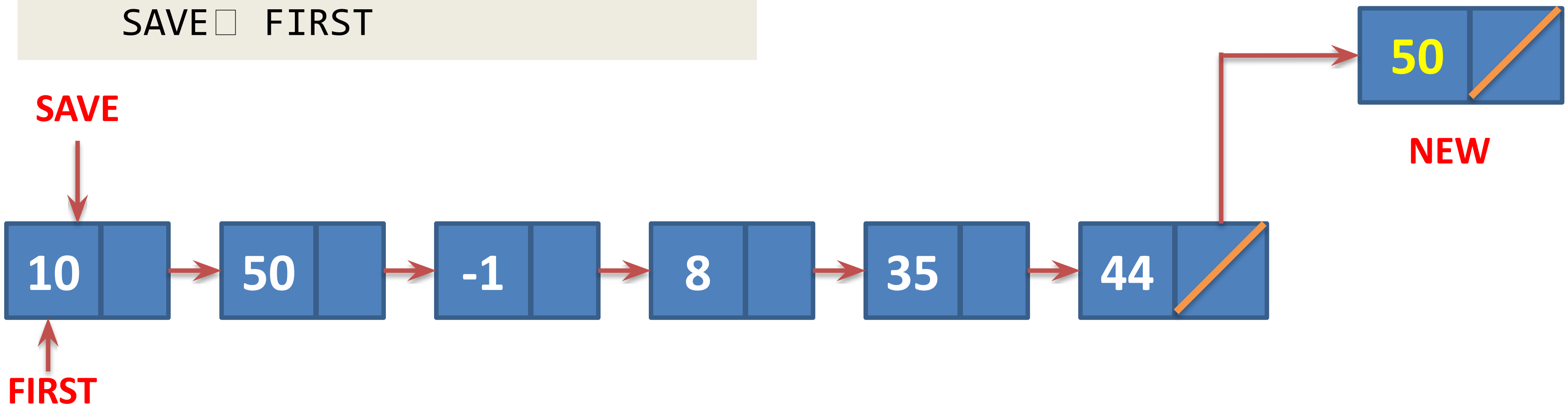
```
LINK (SAVE) □ NEW
```

9. [Return first node pointer]

```
Return (FIRST)
```

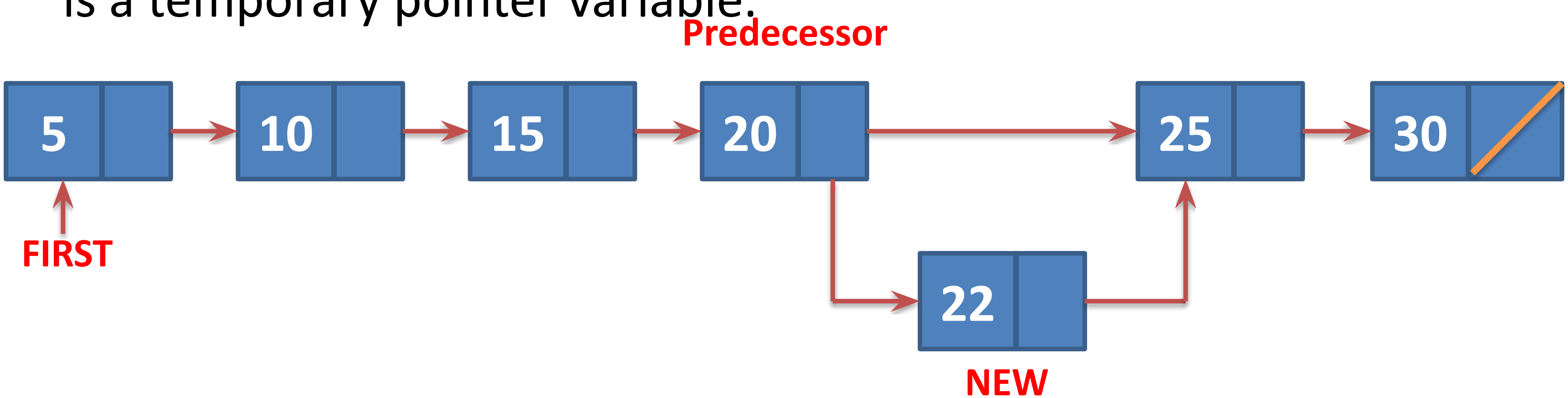
Function: INSEND(50,First)

```
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)

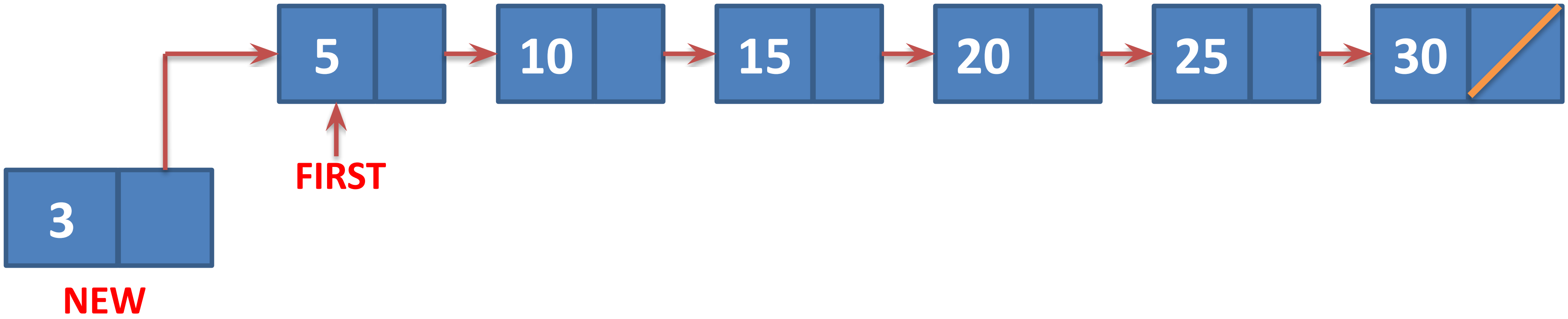
- This function **inserts** a new node such that linked list preserves the ordering of the terms in **increasing order** of their **INFO** field.
- This function returns address of **FIRST** node.
- **X** is a new element to be inserted.
- **FIRST** is a pointer to the first element of a Singly linked 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)

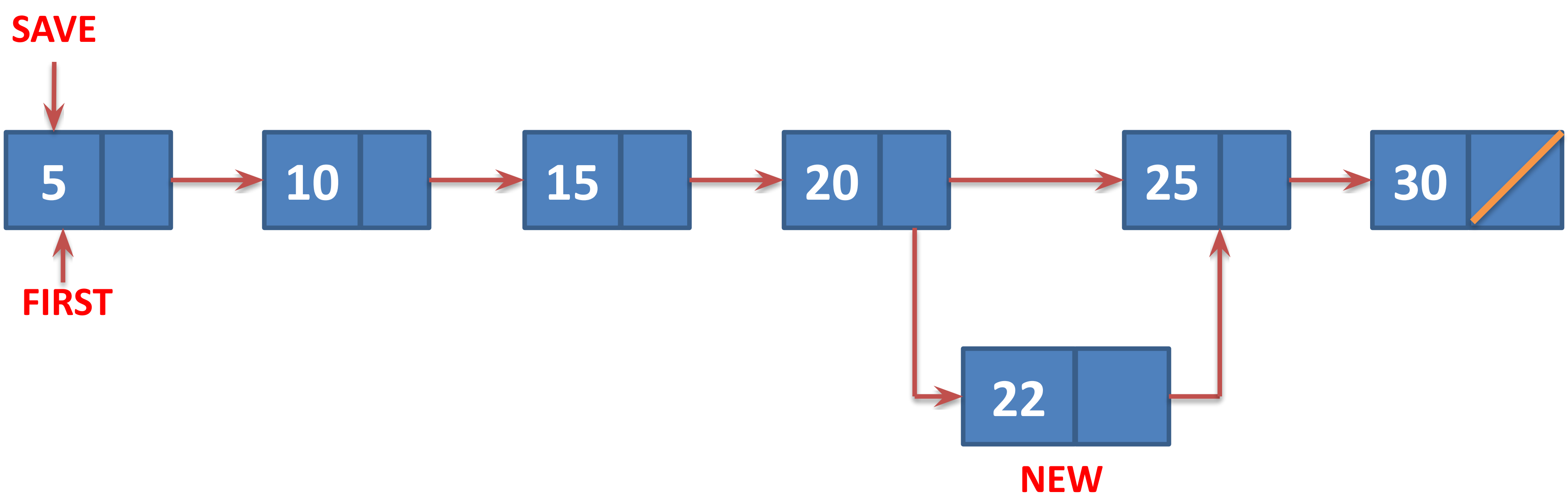
9. [Set link field of NEW node and its predecessor]

    LINK (NEW) ← LINK (SAVE)

    LINK (SAVE) ← NEW

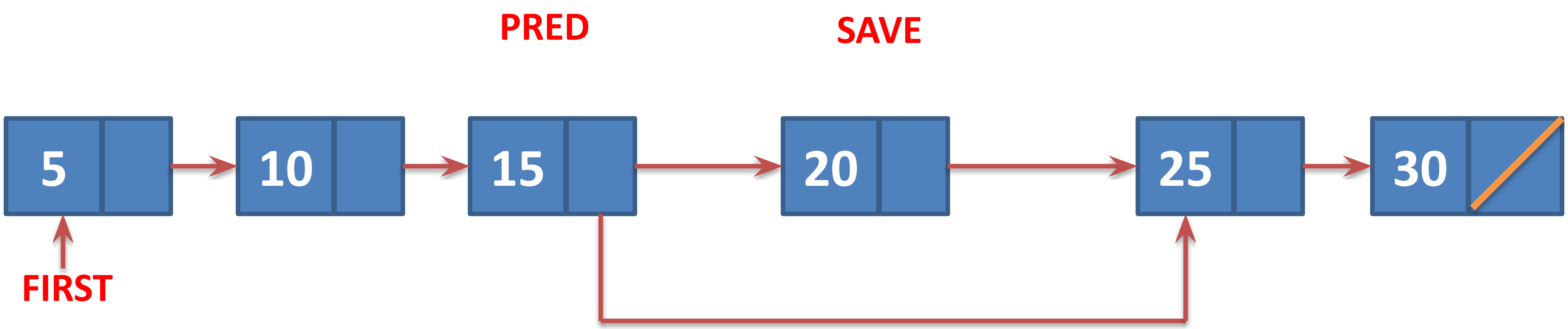
10. [Return first node pointer]

    Return (FIRST)



Function: 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.



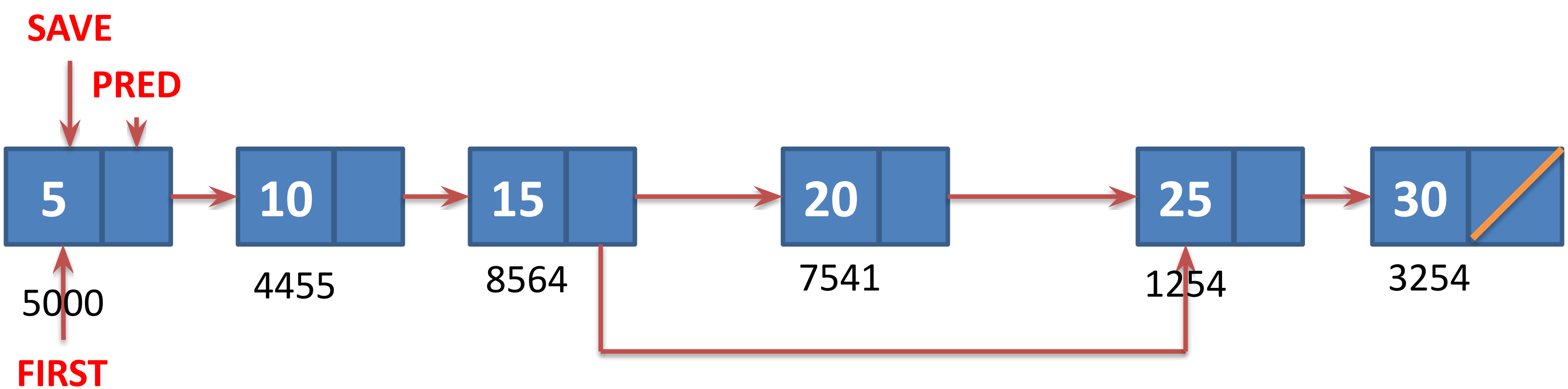
Function: DELETE(X,First)

```
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

4. [Update predecessor marker]
    PRED □ SAVE
5. [Move to next node]
    SAVE □ LINK(SAVE)

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

7. [Delete X]
If   X = FIRST
THEN   FIRST □ LINK(FIRST)
ELSE   LINK (PRED) □ LINK (X)

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

Function: DELETE(7541,First)

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

```
6. [End of the list?]
If  SAVE ≠ X
THEN   write (‘Node not found’)
          Return
7. [Delete X]
If   X = FIRST
THEN   FIRST  LINK(FIRST)
ELSE   LINK (PRED)  LINK (X)
8. [Free Deleted Node]
     Free (X)
```



Function: COUNT_NODE(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_NODE(First)

```
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)

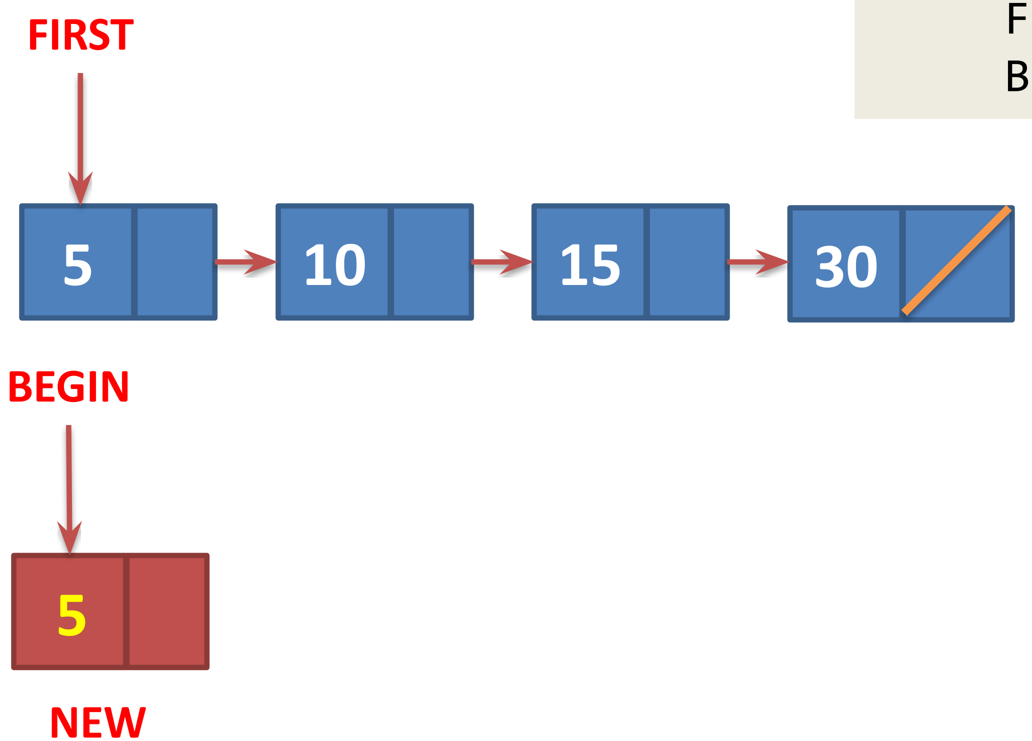
```
1. [Is Empty list?]
   IF    FIRST = NULL
   THEN  Return(NULL)
2. [Copy first node]
IF   AVAIL = NULL
THEN write (‘Underflow’)
     Return (NULL)
ELSE NEW□AVAIL
     AVAIL□LINK(AVAIL)
     FIELD(NEW)□INFO(FIRST)
     BEGIN □NEW
3. [Initialize Traversal]
   SAVE □ FIRST
4. [Move the next node if 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)
```

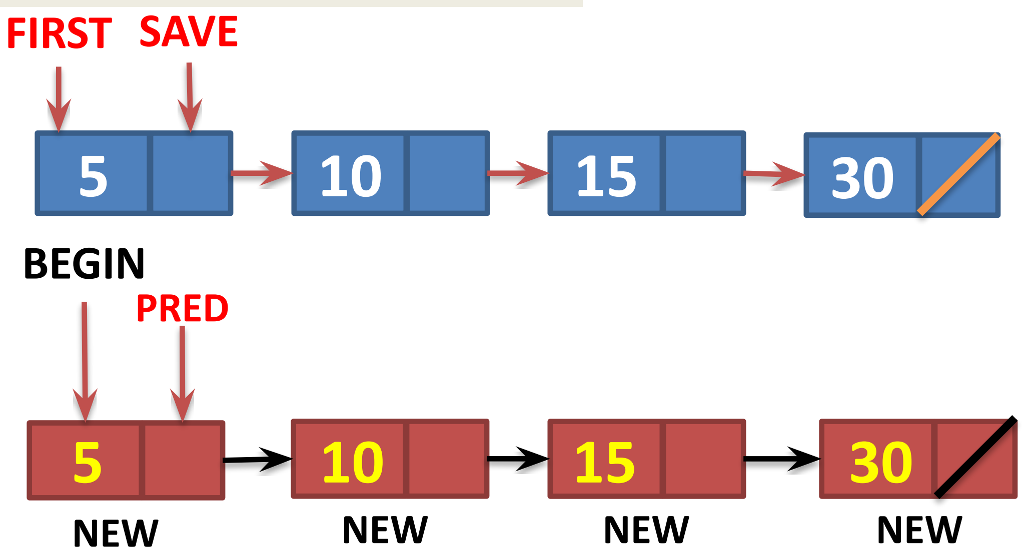
```
2. [Copy first node]
IF   AVAIL = NULL
THEN write (‘Underflow’)
     Return (0)
ELSE NEW□AVAIL
     AVAIL□LINK(AVAIL)
     FIELD(NEW)□INFO(FIRST)
     BEGIN □NEW
```



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)
```



Linked List in C: Program

```
#include <stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *p;

 // function to add node at the beginning
void addbeg(struct node *q,int num)
{
   p=(struct node*)malloc(sizeof(struct node));
   p->link=q;
   p->data=num;

}
```

Linked List in C: Program

```
// function to add node after a given node
void addafter(struct node *q,int c,int num)
{
  struct node *tmp;
  int i;
  for(i=0;i<c;i++)
  {
     q=q->link;
     if(q->link==NULL)
     {
        printf("\nthere are less than %d elements",c);
        return;
     }
  }
  tmp=(struct node*)malloc(sizeof(struct node));
  tmp->link=q->link;
  tmp->data=num;
  q->link=tmp;

}
```

Linked List in C: Program

```
// function to add node at last
void append(struct node *q,int num)
{
   if(q==NULL)
   {
      p=(struct node*)malloc(sizeof(struct node));
      p->data=num;
      p->link=NULL;
   }
   else
   {
      //printf("hi:%p",q->link);
      while(q->link != NULL)
         q=q->link;
      q->link=(struct node*)malloc(sizeof(struct node));
         //printf("%p",q->link);
      q->link->data=num;
      q->link->link=NULL;

   }
}
```

Linked List in C: Program

```
// function to display the content of the list
void display(struct node *q)
{
   printf("\n");
   while(q !=NULL)
   {
      if(q->link !=NULL)
      {
         printf("%d -> %p\t",q->data,q->link);
         q=q->link;
      }
      else
      {
         printf("%d -> NULL\t",q->data);
         q=q->link;
      }
   }
}
```

Linked List in C: Program

```
// function to count how many nodes are there in the list
int count(struct node *q)
{
    int c=0;
    while(q !=NULL)
    {
        q=q->link;
        c++;
    }
    return(c);
}
```

Linked List in C: Program

```
// function to delete a specified nodes in the list
void delete(struct node *q,int num){
    if(q->data==num)    {
        p=q->link;
        free(q);
        return;    }
    while(q->link->link !=NULL)    {
        if(q->link->data == num)        {
            q->link=q->link->link;
            free(q->link);
            return;        }
        q=q->link;    }
    if(q->link->data==num)
    {
        free(q->link);
        q->link=NULL;
        return;
    }
    printf("\nElment %d not found",num);
}
```

Linked List in C: Program

```
// Main function
void main()
{
   p=NULL;
   printf("\nNo. of elements in linked list %d\n",count(p));
   append(p,11);
   append(p,22);
   append(p,33);
   append(p,44);
   append(p,55);
   append(p,66);
   display(p);
   printf("\nNo. of elements in linked list %d\n",count(p));
   addbeg(p,100);
   addbeg(p,200);
   addbeg(p,300);
   display(p);
   printf("\nNo. of elements in linked list %d\n",count(p));
   addafter(p,3,333);
   addafter(p,6,444);
```

Linked List in C: Program

```
// Main function
    display(p);
    printf("\nNo. of elements in linked list %d\n",count(p));
    delete(p,300);
    delete(p,66);
    delete(p,0);
    display(p);
    printf("\nNo. of elements in linked list %d\n",count(p));
}
```

C Program to Reverse a Singly Linked List

```
#include <stdio.h>
#include <stdlib.h>

/* Structure of a node */
struct node {
   int data; //Data part
   struct node *next; //Address part
}*head;

/* Functions used in the program */
void createList(int n);
void reverseList();
void displayList();
```

C Program to Reverse a Singly Linked List

```
int main()
{
   int n, choice;
    /*  Create a singly linked list of n nodes       */
   printf("Enter the total number of nodes: ");
   scanf("%d", &n);
   createList(n);
    printf("\nData in the list \n");
   displayList();
    /* Reverse the list      */
   printf("\nPress 1 to reverse the order of singly linked list\n");
   scanf("%d", &choice);
   if(choice == 1)    {
      reverseList();    }
    printf("\nData in the list\n");
   displayList();

   return 0;
}
```

C Program to Reverse a Singly Linked List

```
 // Create a list of n nodes
void createList(int n)
{
    struct node *newNode, *temp;
    int data, i;

    if(n <= 0)
    {
        printf("List size must be greater than zero.\n");
        return;
    }
   head = (struct node *)malloc(sizeof(struct node));

   /* If unable to allocate memory for head node     */
    if(head == NULL)
    {
        printf("Unable to allocate memory.");
    }
```

C Program to Reverse a Singly Linked List

```
else
{
   /* Read data of node from the user        */
   printf("Enter the data of node 1: ");
   scanf("%d", &data);
   head->data = data; // Link the data field with data
   head->next = NULL; // Link the address field to NULL
  temp = head;

   /*  Create n nodes and adds to linked list        */
   for(i=2; i<=n; i++)
   {
      newNode = (struct node *)malloc(sizeof(struct node));

      /* If memory is not allocated for newNode */
      if(newNode == NULL)
```

C Program to Reverse a Singly Linked List

```
      {
     printf("Unable to allocate memory.");
     break;
   }
   else    {
       printf("Enter the data of node %d: ", i);
       scanf("%d", &data);
      newNode->data = data; // Link the data field
                      of newNode with data
      newNode->next = NULL; // Link the address
                     field of newNode with NULL
      temp->next = newNode; // Link previous node
                 i.e.temp to the newNode
      temp = temp->next;     }
  }

     printf("SINGLY LINKED LIST CREATED SUCCESSFULLY\n");
  }
}
```

C Program to Reverse a Singly Linked List

```
 /*Reverse the order of nodes of a singly linked list */
void reverseList()
{
   struct node *prevNode, *curNode;

   if(head != NULL)
   {
      prevNode = head;
      curNode = head->next;
      head = head->next;

      prevNode->next = NULL; // Make first node as last node
```

C Program to Reverse a Singly Linked List

```
 while(head != NULL)
    {
        head = head->next;
        curNode->next = prevNode;

        prevNode = curNode;
        curNode = head;
    }

    head = prevNode; // Make last node as head

    printf("SUCCESSFULLY REVERSED LIST\n");
  }
}
```

C Program to Reverse a Singly Linked List

```
/* Display entire list */
void displayList()
{
   struct node *temp;

   /*If the list is empty i.e. head = NULL     */
   if(head == NULL)
   {
      printf("List is empty.");
   }
   else
   {
      temp = head;
```

C Program to Reverse a Singly Linked List

```
        while(temp != NULL)
        {
            printf("Data = %d\n", temp->data);
             // Print the data of current node
            temp = temp->next;                // Move to next node
        }
    }
}
```

C Program to Sort Linked List

```
#include <stdio.h>
 #include<stdlib.h>
   //Represent a node of the singly linked list
   struct node{
      int data;
      struct node *next;
   };

 //Represent the head and tail of the singly linked list
   struct node *head, *tail = NULL;
```

C Program to Sort Linked List

```
//addNode() will add a new node to the list
 void addNode(int data) {
    //Create a new node
    struct node *newNode = (struct node*)malloc(sizeof(struct
                                          node));
    newNode->data = data;
    newNode->next = NULL;

    //Checks if the list is empty
    if(head == NULL) {
       //If list is empty, both head and tail will point to new node
       head = newNode;
       tail = newNode;
    }
```

C Program to Sort Linked List

```
else {
        //newNode will be added after tail such that tail's next will
          point to newNode
        tail->next = newNode;
        //newNode will become new tail of the list
        tail = newNode;
    }
}
```

C Program to Sort Linked List

```
//sortList() will sort nodes of the list in ascending order
 void sortList() {
        //Node current will point to head
        struct node *current = head, *index = NULL;
        int temp;
        if(head == NULL) {
              return;
        }
        else {
```

C Program to Sort Linked List

```
while(current != NULL) {
        //Node index will point to node next to current
        index = current->next;

        while(index != NULL) {
          //If current node's data is greater than index's node data, swap the
          data between them
          if(current->data > index->data) {
            temp = current->data;
            current->data = index->data;
            index->data = temp;
          }
          index = index->next;
        }
        current = current->next;
      }
    }
  }
```

C Program to Sort Linked List

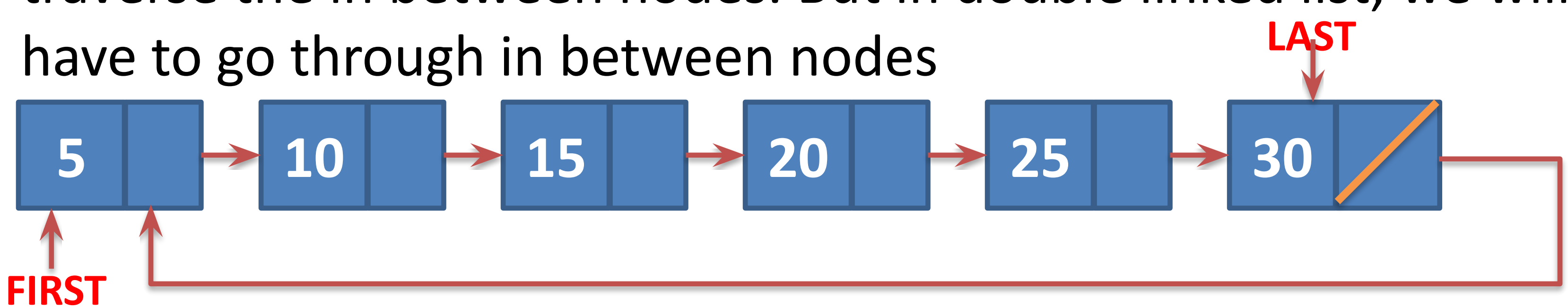
```
//display() will display all the nodes present in the list
  void display() {
    //Node current will point to head
    struct node *current = head;
    if(head == NULL) {
      printf("List is empty \n");
      return;
    }
    while(current != NULL) {
      //Prints each node by incrementing pointer
      printf("%d ", current->data);
      current = current->next;
    }
    printf("\n");
  }
```

C Program to Sort Linked List

```
//Main begins
int main()     {
    //Adds data to the list
    addNode(9);
    addNode(7);
    addNode(2);
    addNode(5);
    addNode(4);
    //Displaying original list
    printf("Original list: \n");
    display();
    //Sorting list
    sortList();
    //Displaying sorted list
    printf("Sorted list: \n");
    display();
    return 0;
  }
```

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

- **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)

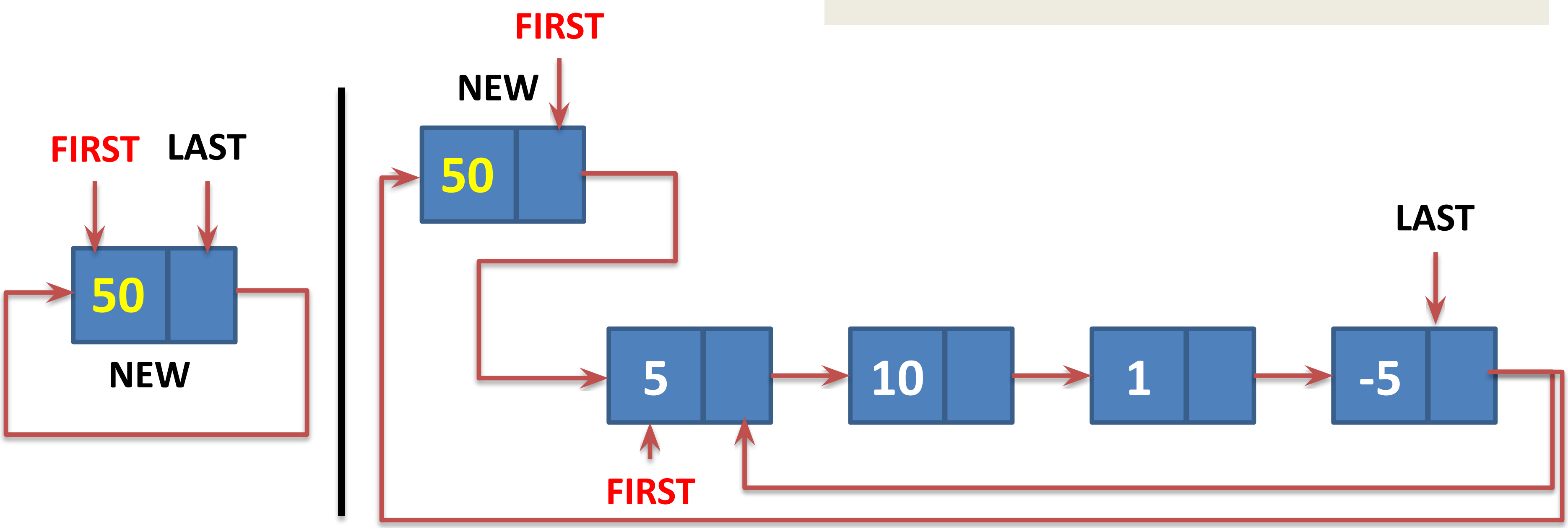
1. [Creates a new empty node]

   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.

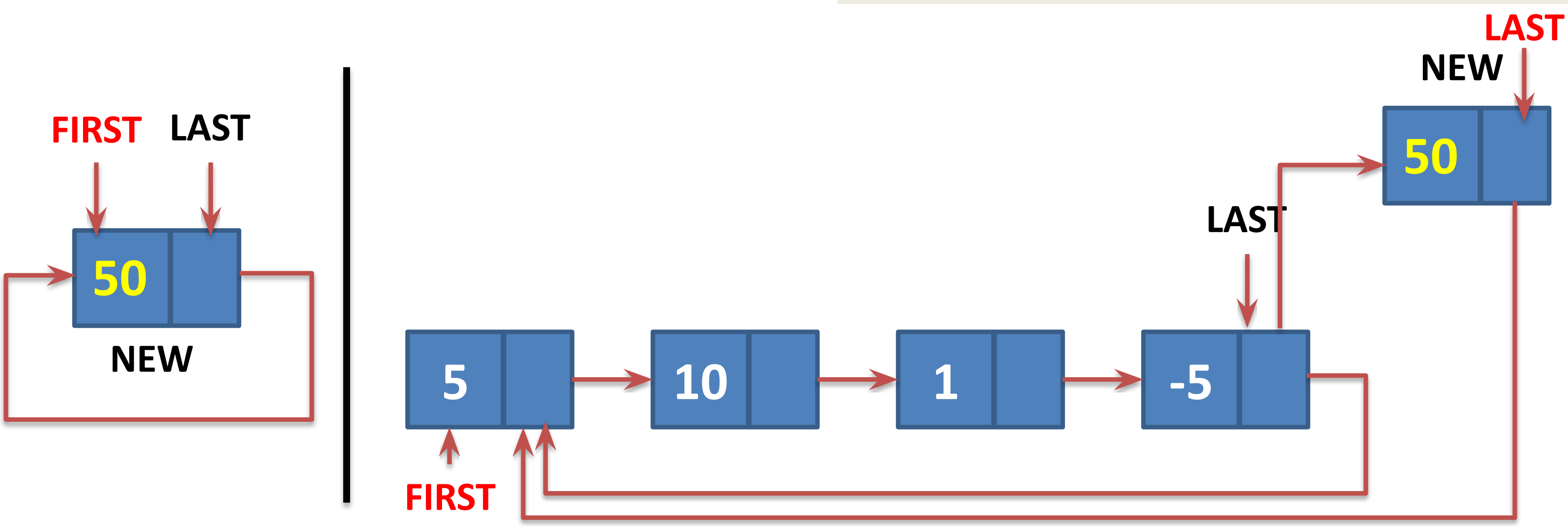
Procedure: CIR_INS_LAST(X,FIRST,LAST)

```
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 procedure **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.

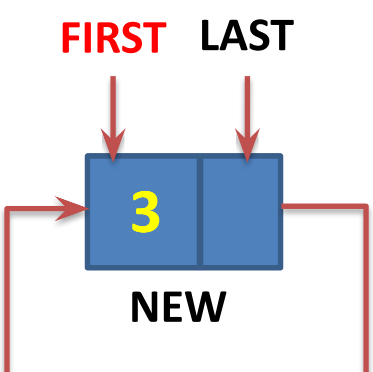
Procedure: CIR_INS_ORD(X,FIRST,LAST)

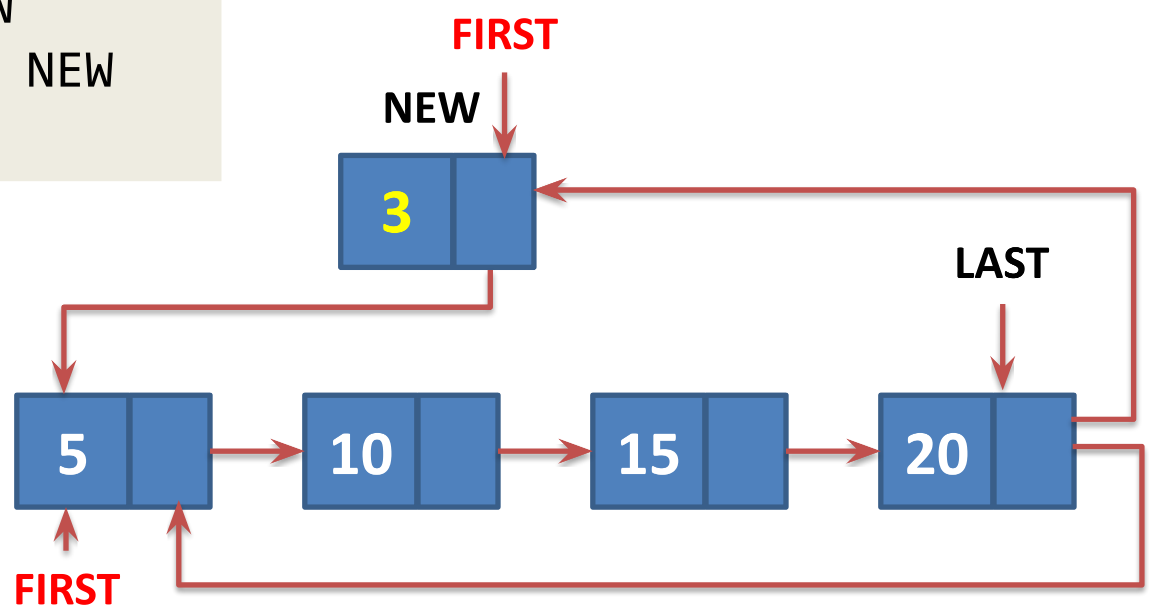
```
1. [Create New Empty Node]
   NEW     NODE
2. [Copy information content
    into new node]
   INFO(NEW) □ X
3. [Is Linked List Empty?]
   IF   FIRST = NULL
   THEN LINK(NEW) □ NEW
        FIRST □ LAST □ NEW
        Return
4. [Does new node precedes all
    other nodes in List?]
IF     INFO(NEW)≤ 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(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
```

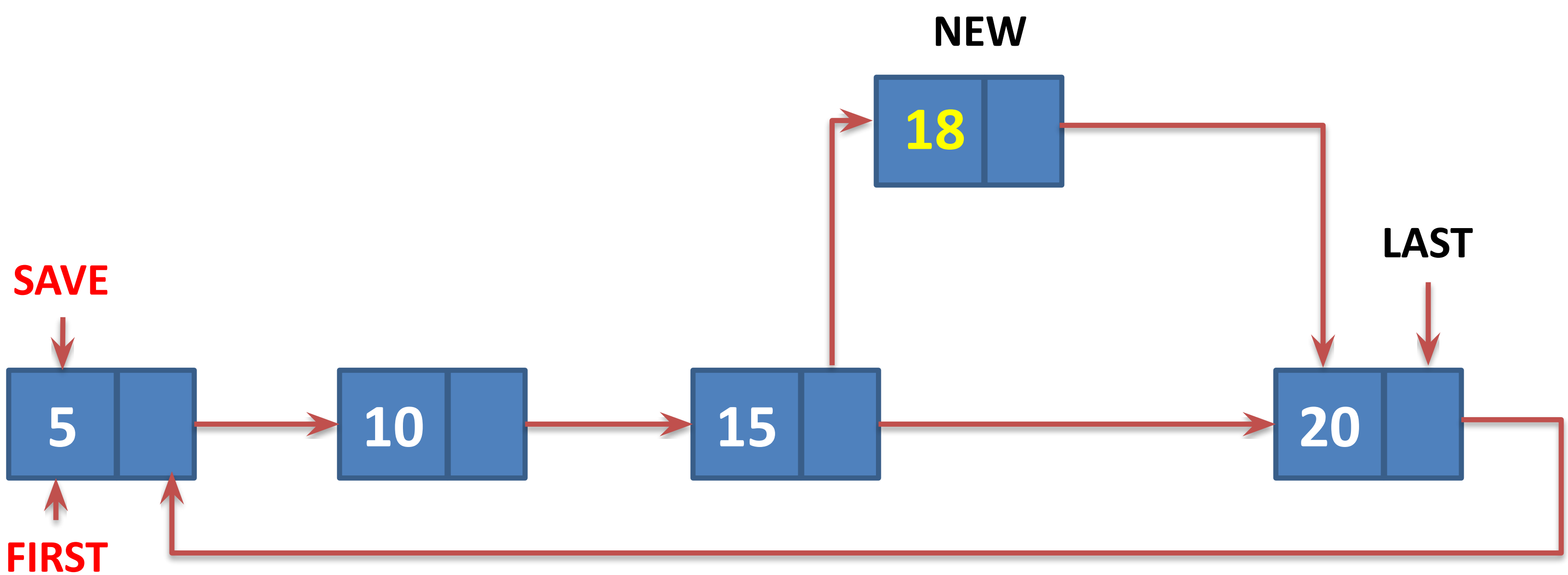




Procedure: CIR_INS_ORD(X,FIRST,LAST)

```
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)

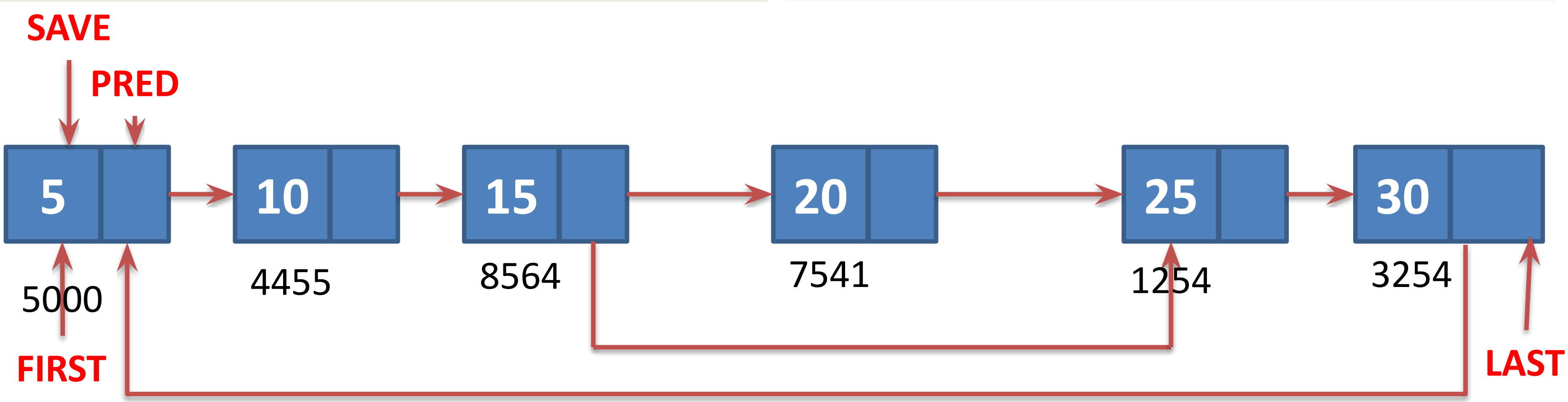
- This algorithm **delete** a node whose address is given by variable **X**.
- **FIRST** & **LAST** are pointers to the First & Last elements of a Circular linked list, respectively.
- Typical node contains **INFO** and **LINK** fields.
- **SAVE** & **PRED** are temporary pointer variable.

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)
```

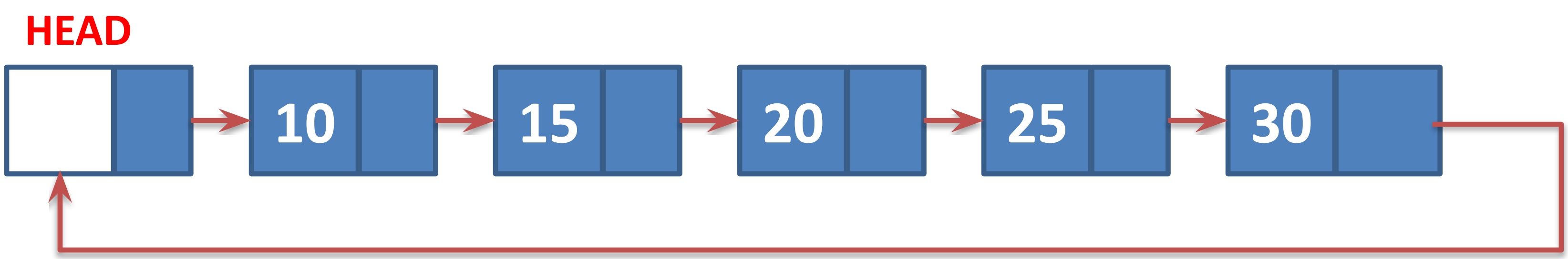
Procedure: CIR_DELETE(X,FIRST,LAST)

```
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 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
     THENLAST □ PRED
8. [Free Deleted Node]
   Free (X)
```



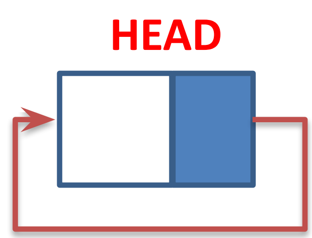
Circularly Linked List with Header Node

- 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

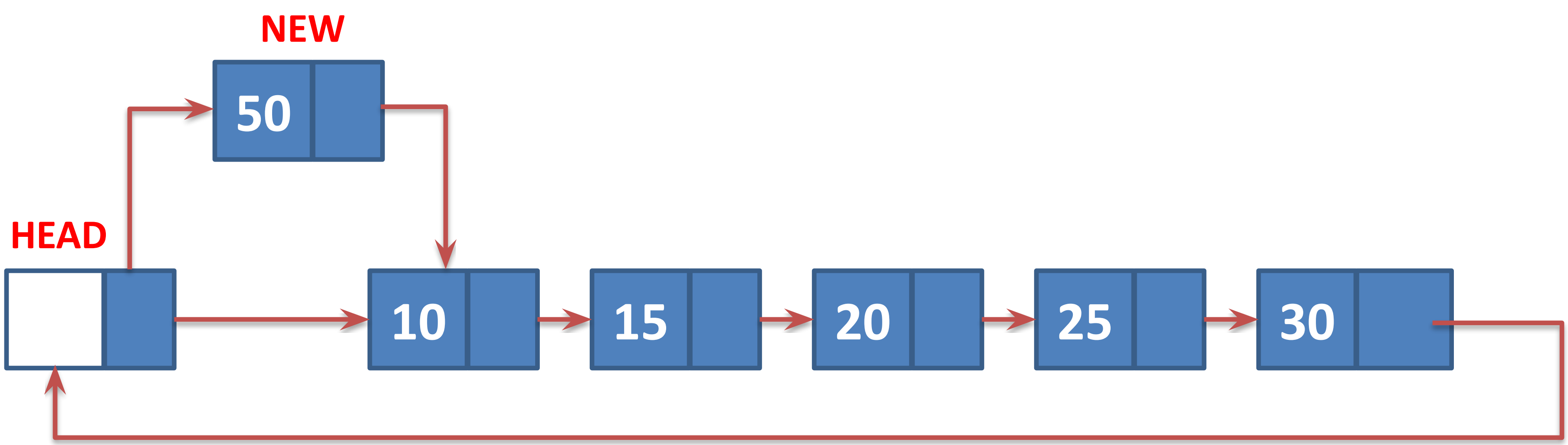


Procedure: CIR_HEAD_INS_FIRST(X,FIRST,LAST)

- 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
```

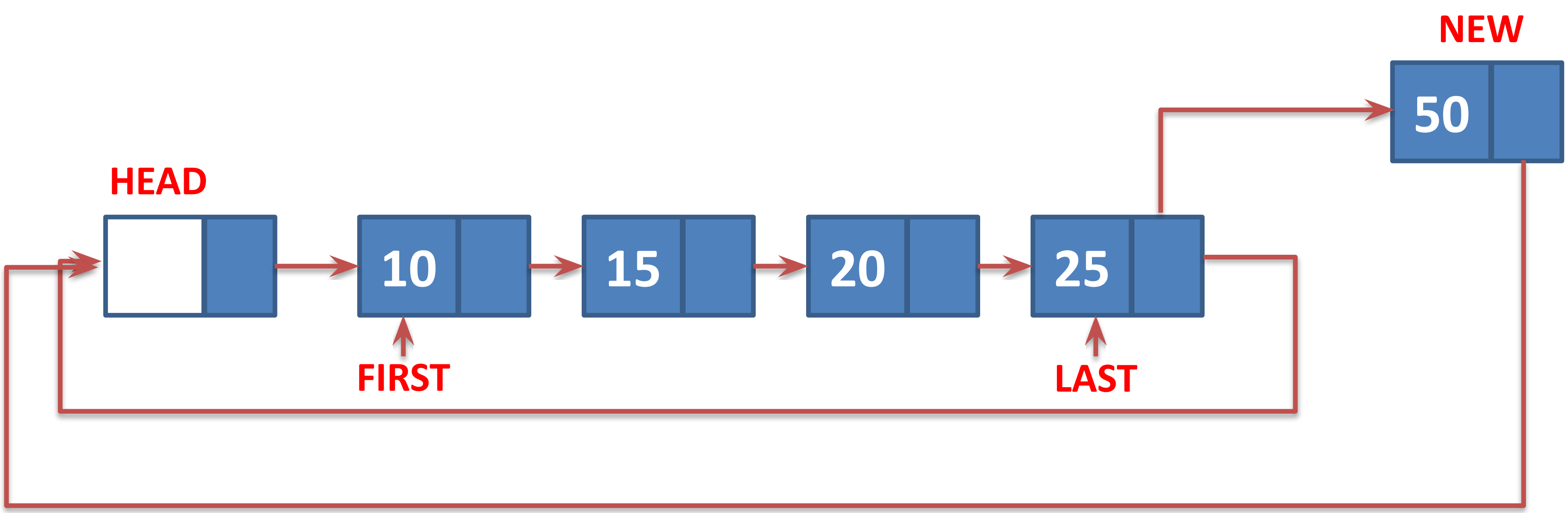


Procedure: CIR_HEAD_INS_LAST(X,FIRST,LAST)

- 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
```

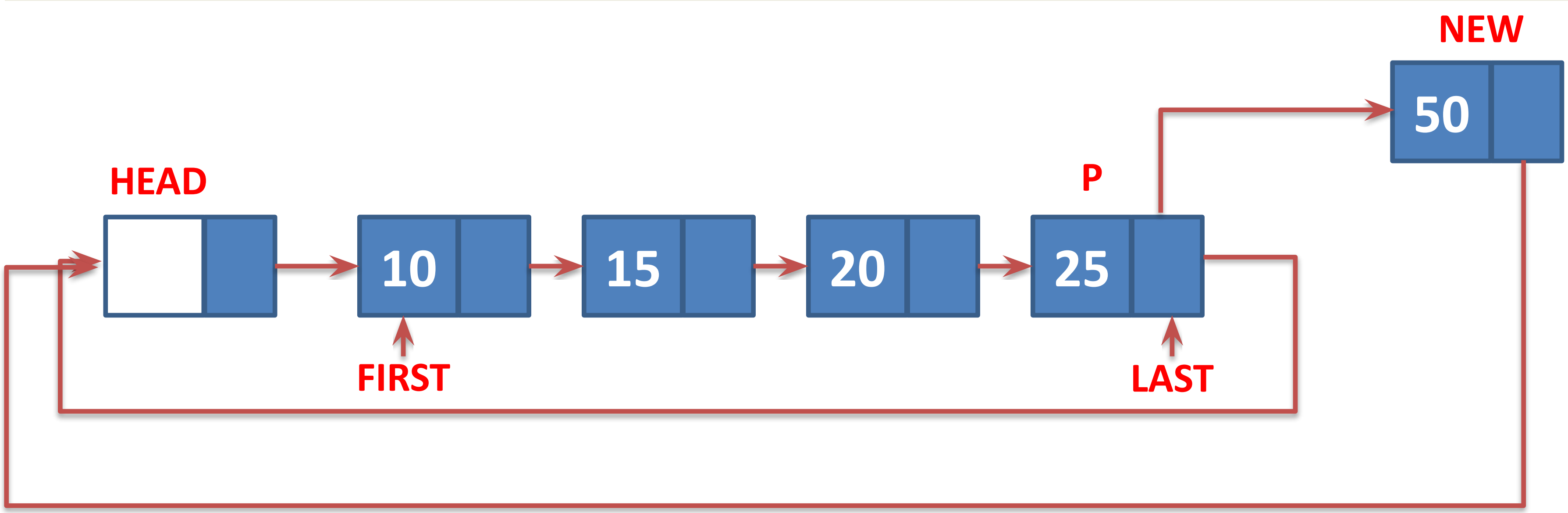


Procedure: CIR_HEAD_INS_ AFTER-P(X,FIRST,LAST)

- 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
```



Program in C implementing all operations on circular singly linked list

```
#include<stdio.h>
#include<stdlib.h>
    struct node
    {
        int data;
        struct node *next;
    };
    struct node *head;

    void beginsert ();
    void lastinsert ();
     void begin_delete();
    void last_delete();
    void display();
    void search();
```

Program in C implementing all operations on circular singly linked list

```
void main ()
  {
     int choice =0;
     while(choice != 7)
     {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete
        from last\n5.Search for an element\n6.Show\n7.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
           case 1:
           beginsert();
           break;
           case 2:
           lastinsert();
           break;
```

Program in C implementing all operations on circular singly linked list

```
        case 3:
        begin_delete();
        break;
        case 4:
        last_delete();
        break;
        case 5:
        search();
        break;
        case 6:
        display();
        break;
        case 7:
        exit(0);
        break;
        default:
        printf("Please enter valid choice..");
      }
   }
}
```

Program in C implementing all operations on circular singly linked list

```
void beginsert()
 {
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
       printf("\nOVERFLOW");
    }
    else
    {
       printf("\nEnter the node data?");
       scanf("%d",&item);
       ptr -> data = item;
```

Program in C implementing all operations on circular singly linked list

```
        if(head == NULL)
         {
            head = ptr;
            ptr -> next = head;
         }
         else
         {
            temp = head;
            while(temp->next != head)
               temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
         }
          printf("\nnode inserted\n");
     }

}
```

Program in C implementing all operations on circular singly linked list

```
void lastinsert()
 {
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
       printf("\nOVERFLOW\n");
    }
    else
    {
       printf("\nEnter Data?");
       scanf("%d",&item);
       ptr->data = item;
```

Program in C implementing all operations on circular singly linked list

```
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }

        printf("\nnode inserted\n");
    }

}
```

Program in C implementing all operations on circular singly linked list

```
void begin_delete()
  {
     struct node *ptr;
     if(head == NULL)
     {
        printf("\nUNDERFLOW");
     }
     else if(head->next == head)
     {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
     }
```

Program in C implementing all operations on circular singly linked list

```
    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

    }
}
```

Program in C implementing all operations on circular singly linked list

```
void last_delete()
  {
      struct node *ptr, *preptr;
      if(head==NULL)
      {
         printf("\nUNDERFLOW");
      }
      else if (head ->next == head)
      {
         head = NULL;
         free(head);
         printf("\nnode deleted\n");

      }
```

Program in C implementing all operations on circular singly linked list

```
 else
  {
     ptr = head;
     while(ptr ->next != head)
     {
        preptr=ptr;
        ptr = ptr->next;
     }
     preptr->next = ptr -> next;
     free(ptr);
     printf("\nnode deleted\n");

  }
}
```

Program in C implementing all operations on circular singly linked list

```
void search()
 {
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
       printf("\nEmpty List\n");
    }
    else
    {
       printf("\nEnter item which you want to search?\n");
       scanf("%d",&item);
```

Program in C implementing all operations on circular singly linked list

```
if(head ->data == item)
 {
 printf("item found at location %d",i+1);
 flag=0;
 }
 else
 {
 while (ptr->next != head)
 {
    if(ptr->data == item)
    {
       printf("item found at location %d ",i+1);
       flag=0;
       break;
    }
```

```
            else
            {
               flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        }
        if(flag != 0)
        {
            printf("Item not found\n");
        }
    }

}
```

Program in C implementing all operations on circular singly linked list

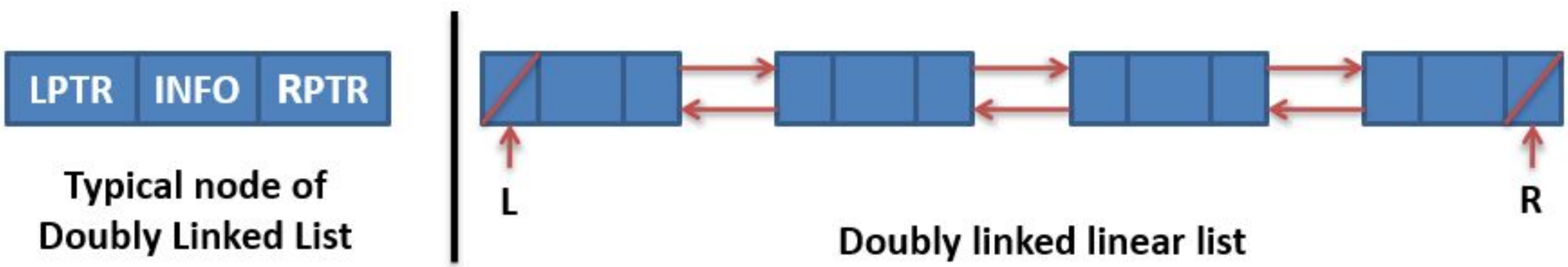
```
void display()     {
      struct node *ptr;
      ptr=head;
      if(head == NULL)        {
         printf("\nnothing to print");         }
      else        {
         printf("\n printing values ... \n");
         while(ptr -> next != head)            {
            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
         }
         printf("%d\n", ptr -> data);
      }
}
```

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

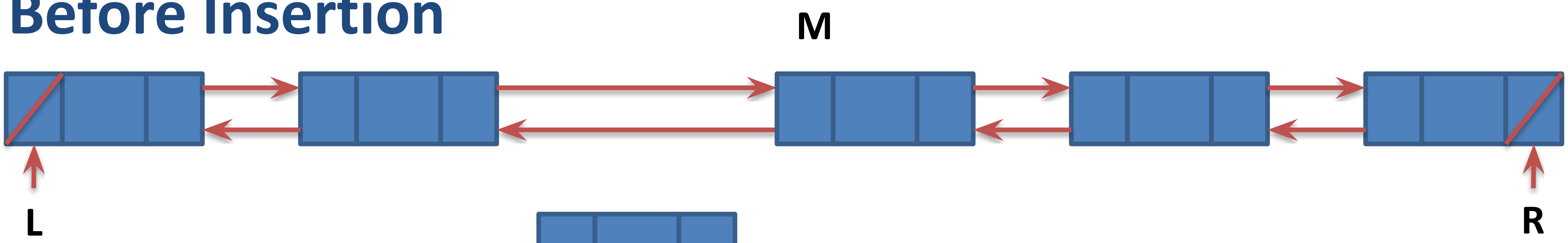
- Typical node of doubly linked linear list contains INFO, LPTR RPTR Fields
- **LPTR** is pointer variable pointing to Predecessor of a node
- **RPTR** is pointer variable pointing to Successor of a node
- Left most node of doubly linked linear list is called **L, LPTR** of node **L is always NULL**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

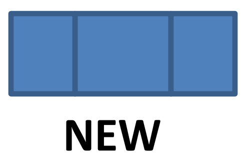


Insert node in Doubly Linked List

Insertion in the middle of Doubly Linked Linear List

Before Insertion





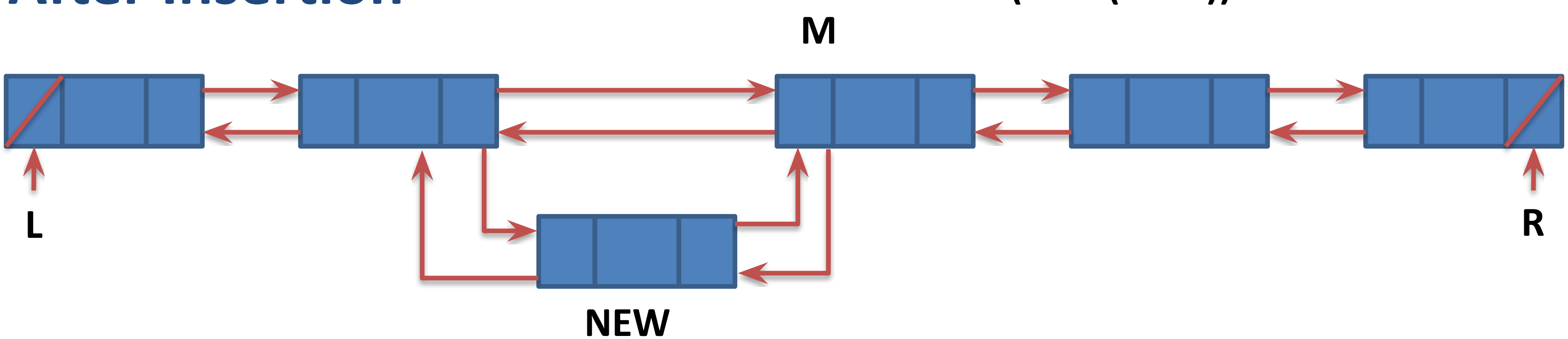
LPTR(NEW) □ LPTR(M)

RPTR(NEW) □ M

LPTR(M) □ NEW

RPTR(LPTR(NEW)) □ NEW

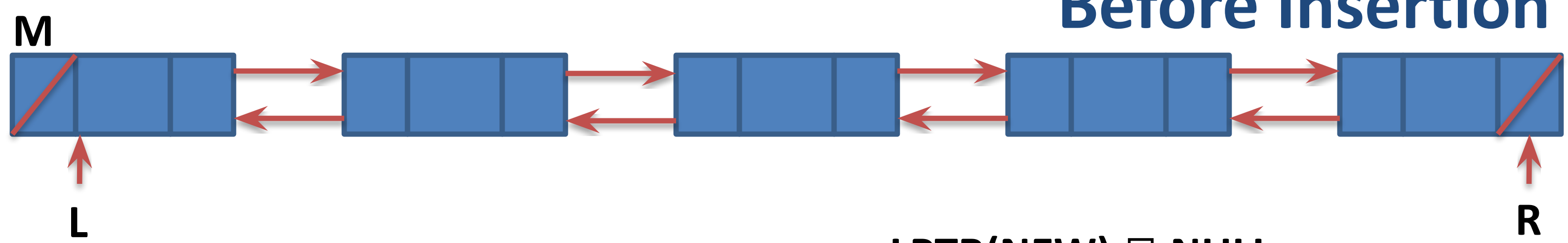
After Insertion

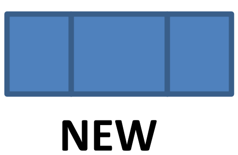


Insert node in Doubly Linked List

Left most insertion in Doubly Linked Linear List

Before Insertion





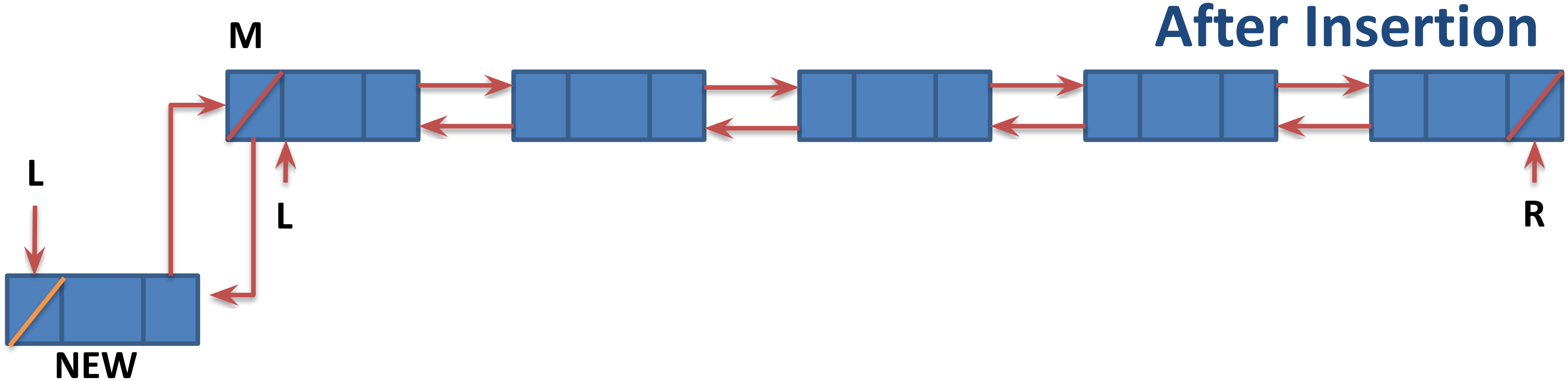
LPTR(NEW) □ NULL

RPTR(NEW) □ M

LPTR(M) □ NEW

L □ NEW

After Insertion



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
```

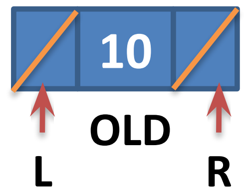
5. [Insert in middle]

```
LPTR(NEW) □ LPTR(M)
RPTR(NEW) □ M
LPTR(M) □ NEW
RPTR(LPTR(NEW)) □ NEW
Return
```

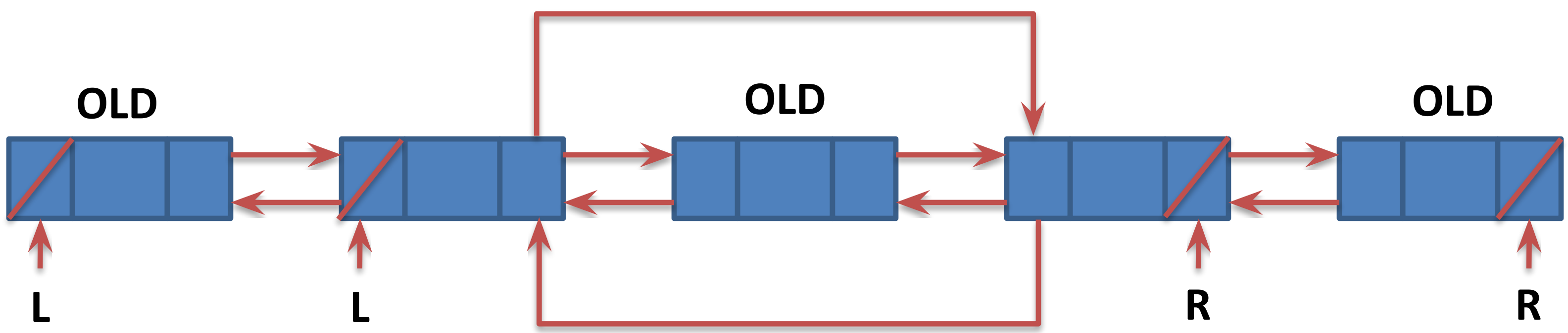
Procedure: DOU _DEL (L, R, OLD)

- 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)
```

Program for Doubly Linked List

```
#include <stdio.h>
#include<stdlib.h>
#define NULL 0
struct node
{
  struct node *prev;
  int data;
  struct node *next;
};
struct node *p;
 //add node at the begining
void addbeg (struct node *q, int num) {
  p = (struct node *) malloc (sizeof (struct node));
  p->prev = NULL;
  p->next = q;
  p->data = num;
  q->prev = p;
}
```

Program for Doubly Linked List

```
//adding node after a given node
void addafter (struct node *q, int c, int num){
 struct node *tmp;
 int i;
 for (i = 0; i < c; i++)    {
    q = q->next;
    if (q == NULL)  {
       printf ("\nthere are less than %d elements", c);
       return; }
  }
 q = q->prev;
 tmp = (struct node *) malloc (sizeof (struct node));
 tmp->prev = q;
 tmp->next = q->next;
 tmp->data = num;
 tmp->next->prev = tmp;
 q->next = tmp;
}
```

Program for Doubly Linked List

```
//adding node at the end
void append (struct node *q, int num){
 struct node *r;
 if (q == NULL)    {
    p = (struct node *) malloc (sizeof (struct node));
    p->data = num;
    p->prev = NULL;
    p->next = NULL;    }
 else    {
    while (q->next != NULL)
      q = q->next;
    r = (struct node *) malloc (sizeof (struct node));
    r->data = num;
    r->next = NULL;
    r->prev = q;
    q->next = r;

   }
}
```

Program for Doubly Linked List

```
//Displaying the list
void display (struct node *q){
 int k = 1;
 printf ("\n");
 while (q != NULL)    {
    if (k == 1)     {
      printf ("NULL -> %d -> %p\t\t", q->data, q->next);
      q = q->next;
      k++;     }
    if (q->next != NULL)      {
       printf ("%p  ->%d -> %p\t\t", q->prev, q->data, q->next);
       q = q->next;       }
    else   {
       printf ("%p -> %d -> NULL\t\t", q->prev, q->data);
       q = q->next;


      }
   }
}
```

Program for Doubly Linked List

```
//count number of node in the list
int count (struct node *q)
{

  int c = 0;
  while (q != NULL)
   {
     q = q->next;
     c++;
   }
  return (c);
}
```

Program for Doubly Linked List

```
//Delete a specified node in the list
void delete (struct node *q, int num){
 while (q != NULL)    {
   if (q->data == num)       {
     if (q->next != NULL)
      q->next->prev = q->prev;
     if (q->prev != NULL)
      q->prev->next = q->next;
     if (p == q)
      p = q->next;
     free (q);
     return;
    }
   q = q->next;
  }
 printf ("\nElment %d not found", num);
 return;
}
```

Program for Doubly Linked List

```
//main function
void main (){
 p = NULL;
 printf ("\nNo. of elements in linked list %d\n", count (p));
 append (p, 11);
 append (p, 22);
 append (p, 33);
 append (p, 44);
 append (p, 55);
 append (p, 66);
 append (p, 77);
 display (p);
 printf ("\nNo. of elements in linked list %d\n", count (p));
 addbeg (p, 100);
 addbeg (p, 200);
 addbeg (p, 300);
 addbeg (p, 400);
 display (p);
```

Program for Doubly Linked List

```
//main function
printf ("\nNo. of elements in linked list %d\n", count (p));
 addafter (p, 3, 333);
 addafter (p, 6, 444);
 addafter (p, 8, 555);
 addafter (p, 9, 655);
 addafter (p, 9, 666);
 display (p);
 printf ("\nNo. of elements in linked list %d\n", count (p));
 delete (p, 300);
 delete (p, 66);
 delete (p, 0);
 delete (p, 444);
 display (p);
 printf ("\nNo. of elements in linked list %d\n", count (p));
}
```

Program for sorting Doubly Linked List

```
#include <stdio.h>
#include<stdlib.h>
  //Represent a node of the doubly linked list

struct node{
      int data;
      struct node *previous;
      struct node *next;
};

  //Represent the head and tail of the doubly linked list
struct node *head, *tail = NULL;
```

Program for sorting Doubly Linked List

```
//addNode() will add a node to the list
void addNode(int data) {
    //Create a new node
    struct node *newNode = (struct node*)malloc(sizeof(struct
    node));
    newNode->data = data;
    //If list is empty
    if(head == NULL) {
        //Both head and tail will point to newNode
        head = tail = newNode;
        //head's previous will point to NULL
        head->previous = NULL;
        //tail's next will point to NULL, as it is the last node of the list
        tail->next = NULL;
    }
```

Program for sorting Doubly Linked List

```
    else {
      //newNode will be added after tail such that tail's next will
     //point to newNode
      tail->next = newNode;
      //newNode's previous will point to tail
      newNode->previous = tail;
      //newNode will become new tail
      tail = newNode;
      //As it is last node, tail's next will point to NULL
      tail->next = NULL;
    }
}
```

Program for sorting Doubly Linked List

```
//sortList() will sort the given list in ascending order
 void sortList() {
     struct node *current = NULL, *index = NULL;
     int temp;
     //Check whether list is empty
     if(head == NULL) {
        return;       }
     else {
        //Current will point to head
        for(current = head; current->next != NULL; current = current->next) {
           //Index will point to node next to current
           for(index = current->next; index != NULL; index = index->next) {
              //If current's data is greater than index's data, swap the data of current and index
              if(current->data > index->data) {
                 temp = current->data;
                 current->data = index->data;
                 index->data = temp;                 }
           }
        }
     }
   }
```

Program for sorting Doubly Linked List

```
//sortList1() will sort the given list in descending order
 void sortList()1 {
    struct node *current = NULL, *index = NULL;
    int temp;
    //Check whether list is empty
    if(head == NULL) {
       return;       }
    else {
       //Current will point to head
       for(current = head; current->next != NULL; current = current->next) {
          //Index will point to node next to current
          for(index = current->next; index != NULL; index = index->next) {
             //If current's data is greater than index's data, swap the data of current and index
             if(current->data < index->data) {
                temp = current->data;
                current->data = index->data;
                index->data = temp;                  }
          }
       }
    }
 }
```

Program for sorting Doubly Linked List

```
//display() will print out the nodes of the list
void display() {
    //Node current will point to head
    struct node *current = head;
    if(head == NULL) {
        printf("List is empty\n");
        return;
    }
    while(current != NULL) {
        //Prints each node by incrementing pointer.
        printf("%d ",current->data);
        current = current->next;
    }
    printf("\n");
}
```

Program for sorting Doubly Linked List

//Main () begins

```
int main()     {
    //Add nodes to the list
    addNode(7);
    addNode(1);
    addNode(4);
    addNode(5);
    addNode(2);
    //Displaying original list
    printf("Original list: \n");
    display();
    //Sorting list
    sortList();
    //Displaying sorted list
    printf("Sorted list: Ascending Order \n");
    display();
    sortList1();
    printf("Sorted list: Descending Order \n");
    display();
    return 0;     }
```

Program for implementing stack using linked list

```
#include <stdio.h>

#include <stdlib.h>

void push ();

void pop ();

void display ();

struct node

{

  int val;

  struct node *next;

};

struct node *head;
```

Program for implementing stack using linked list

```
void main ()
{
  int choice = 0;
  printf ("\n*********Stack operations using linked list*********\n");
  printf ("\n----------------------------------------------\n");
  while (choice != 4)    {
     printf ("\n\nChose one from the below options...\n");
     printf ("\n1.Push\n2.Pop\n3.Show\n4.Exit");
     printf ("\n Enter your choice \n");
     scanf ("%d", &choice);
     switch (choice)   {
       case 1:      {
          push ();
          break;    }
       case 2:      {
          pop ();
          break;    }
       case 3:      {
          display ();
          break;    }
```

Program for implementing stack using linked list

```
      case 4:
       {
         printf ("Exiting....");
         break;
       }
      default:
       {
         printf ("Please Enter valid choice ");
       }
      };
   }
}
```

Program for implementing stack using linked list

```
void push (){
 int val;
 struct node *ptr = (struct node *) malloc (sizeof (struct node));
 if (ptr == NULL)    {
    printf ("not able to push the element");    }
 else    {
    printf ("Enter the value");
    scanf ("%d", &val);
    if (head == NULL)       {
        ptr->val = val;
        ptr->next = NULL;
        head = ptr;     }
    else    {
        ptr->val = val;
        ptr->next = head;
        head = ptr;
      }
    printf ("Item pushed");


  }
}
```

Program for implementing stack using linked list

```
void pop ()
{
  int item;
  struct node *ptr;
  if (head == NULL)
    {
      printf ("Underflow");
    }
  else
    {
      item = head->val;
      ptr = head;
      head = head->next;
      free (ptr);
      printf ("Item popped");

    }
}
```

Program for implementing stack using linked list

```
void display ()
{
 int i;
 struct node *ptr;
 ptr = head;
 if (ptr == NULL)
  {
   printf ("Stack is empty\n");
  }
 else
  {
   printf ("Printing Stack elements \n");
   while (ptr != NULL)
    {
     printf ("%d\n", ptr->val);
     ptr = ptr->next;
    }
  }
}
```

Program for implementing queue using linked list

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
   int data;
   struct node *next;
};

struct node *front;
struct node *rear;
void insert();
void delete();
void display();
```

Program for implementing queue using linked list

```
void main ()
{
   int choice;
   while(choice != 4)
   {
      printf("\n**********************Main Menu*****************************\n");
      printf("\n=============================================================\n");
     printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
      printf("\nEnter your choice ?");
      scanf("%d",& choice);
      switch(choice)       {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
```

Program for implementing queue using linked list

```
        case 4:
        exit(0);
        break;
        default:
        printf("\nEnter valid choice??\n");
      }
   }
}
```

Program for implementing queue using linked list

```
void insert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)     {
        printf("\nOVERFLOW\n");
        return;     }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
```

Program for implementing queue using linked list

```
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
```

Program for implementing queue using linked list

```
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

Program for implementing queue using linked list

```
 void display()
{
   struct node *ptr;
   ptr = front;
   if(front == NULL)
   {
      printf("\nEmpty queue\n");
   }
   else
   {
     printf("\nprinting values .....\n");
      while(ptr != NULL)
      {
         printf("\n%d\n",ptr -> data);
         ptr = ptr -> next;
      }
   }
}
```