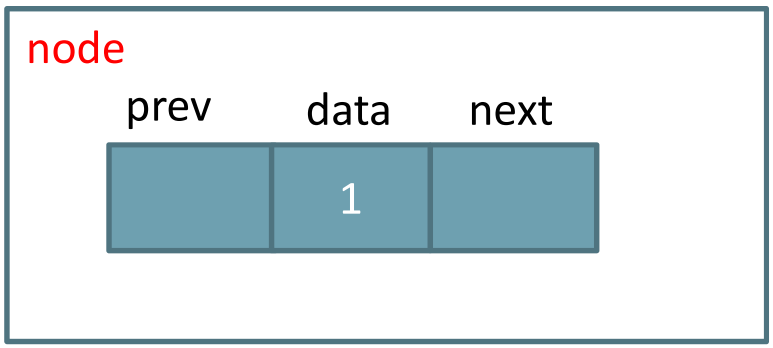
Double Linked List

Kanak Kalyani

Double Linked List

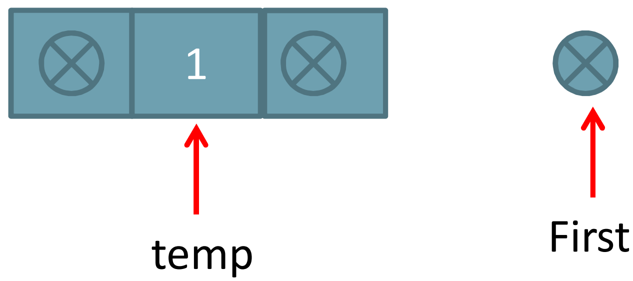
A doubly linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence



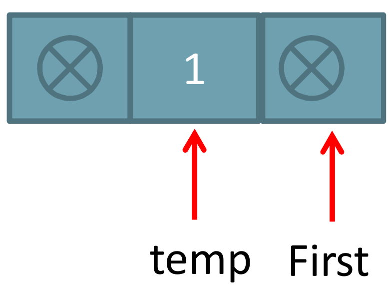

```
struct node
{       struct node *prev;
        int data;
        struct node *next;
};
```

Double Linked List: Creation

STEP1: Create temp node with data value



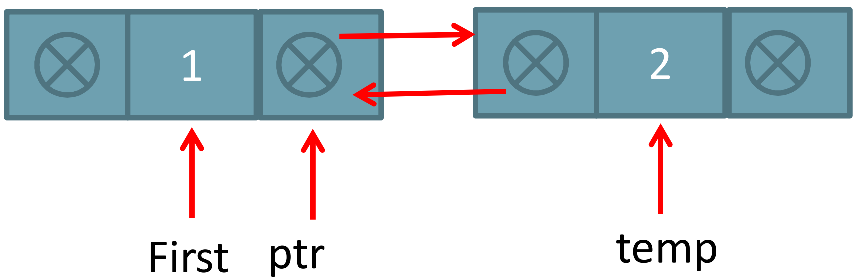
STEP2: if first is NULL then Assign temp to first




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

Double Linked List: Creation

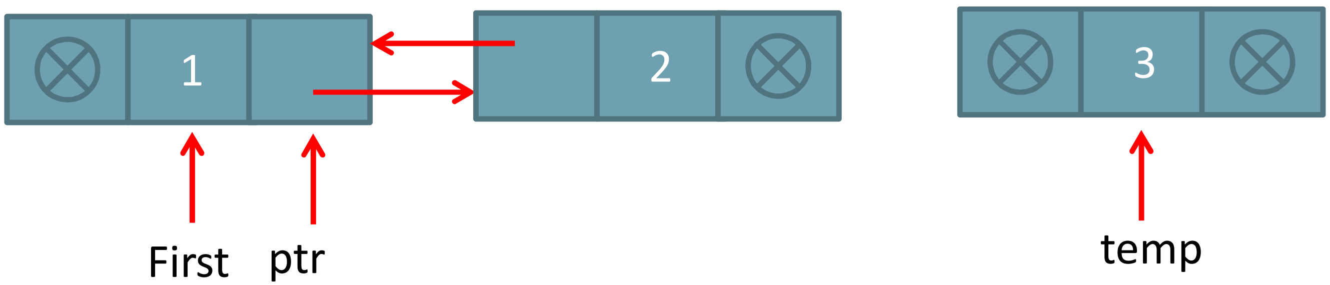
CREATE SECOND NODE




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

Double Linked List: Creation

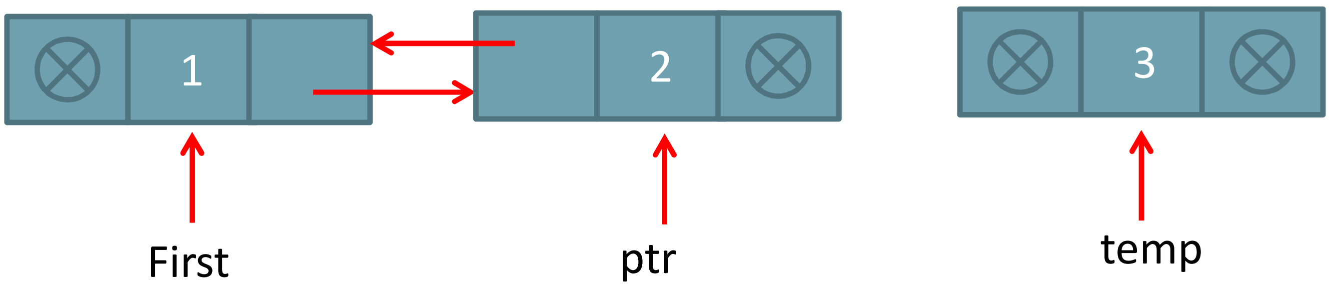
CREATE THIRD NODE




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

Double Linked List: Creation

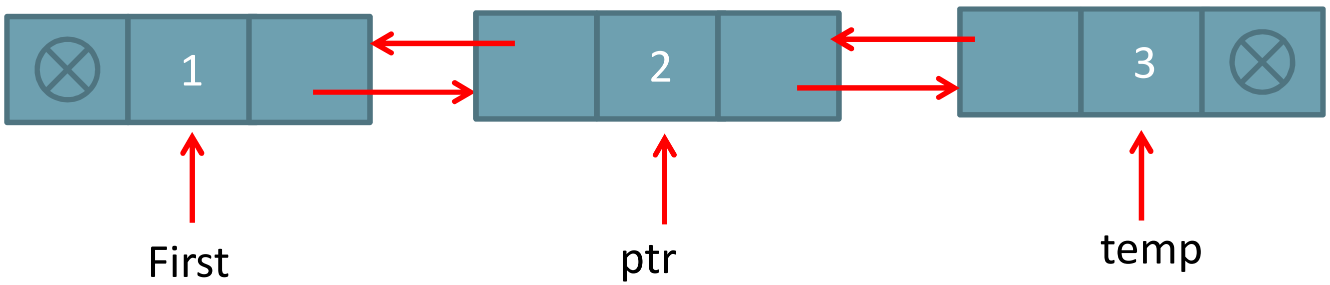
CREATE THIRD NODE




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

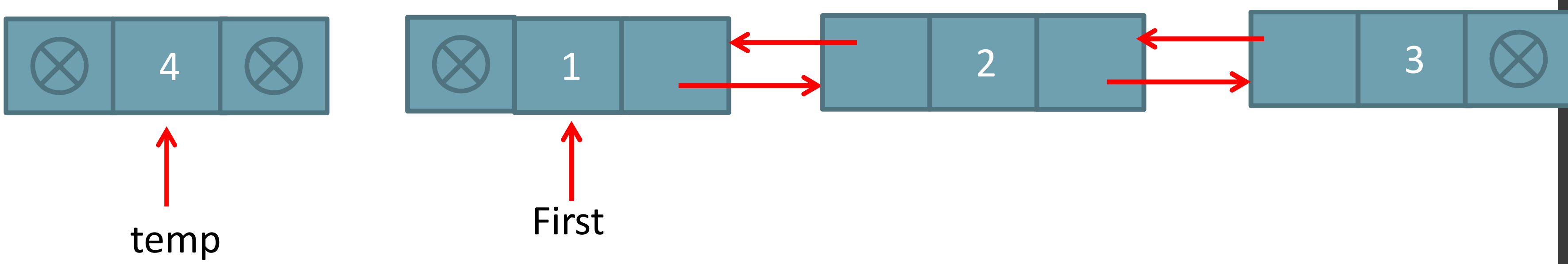
Double Linked List: Creation

CREATE THIRD NODE




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

Double Linked List
Insertion: At First Position



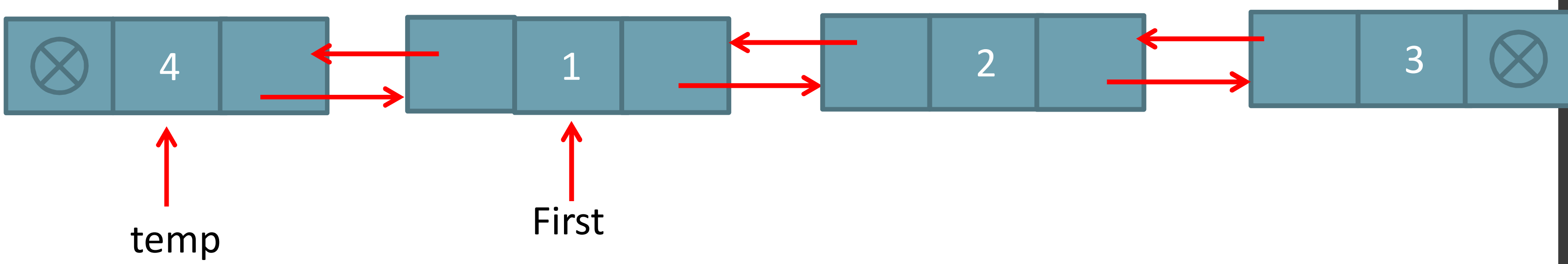
temp->next=first;

first->prev=temp;

first=first->prev;

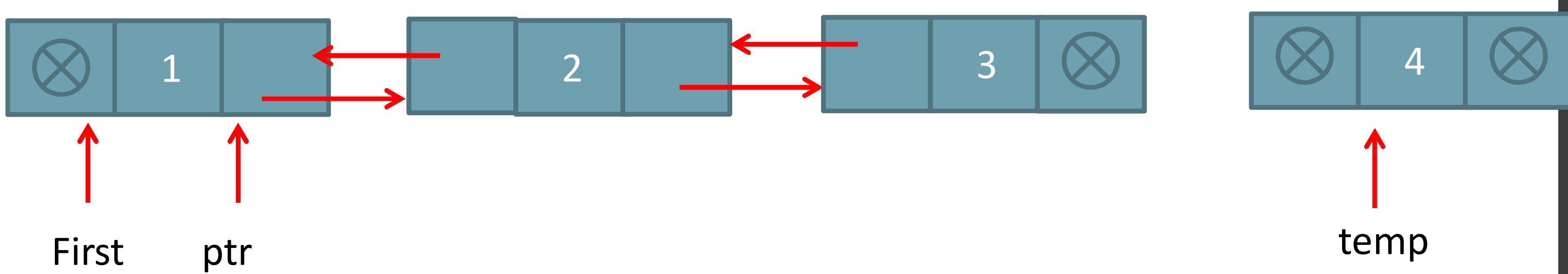
Kanak Kalyani

Double Linked List
Insertion: At First Position



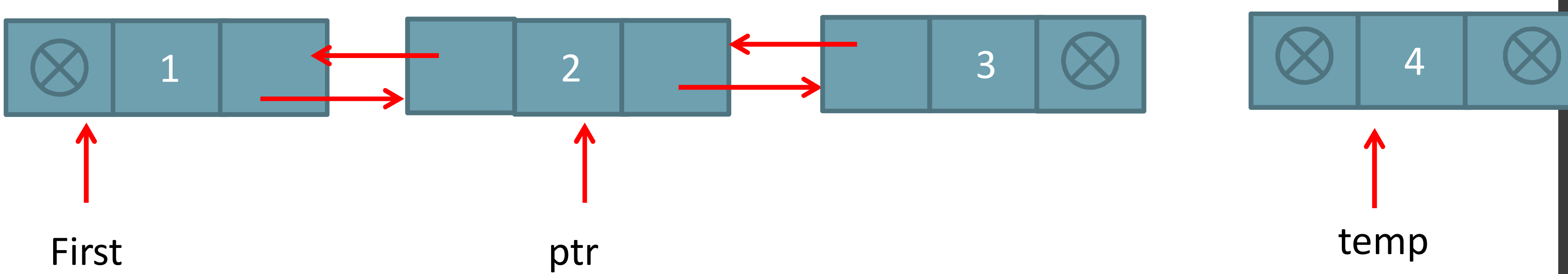

```
temp->next=first;
first->prev=temp;
first=first->prev;
```

Double Linked List
Insertion: At Last Position



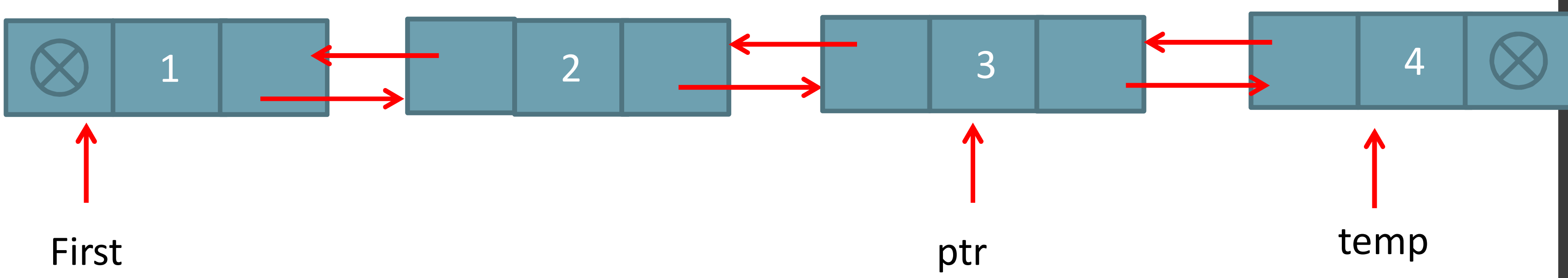

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
temp->prev=ptr;
```

Double Linked List
Insertion: At Last Position



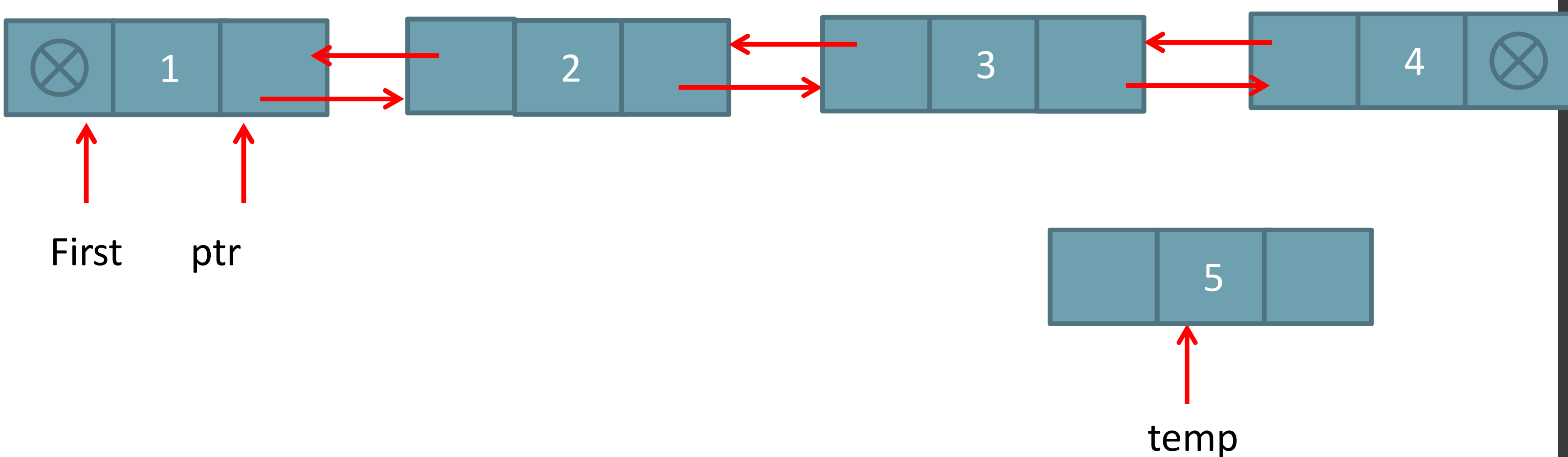

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
temp->prev=ptr;
```

Double Linked List
Insertion: At Last Position



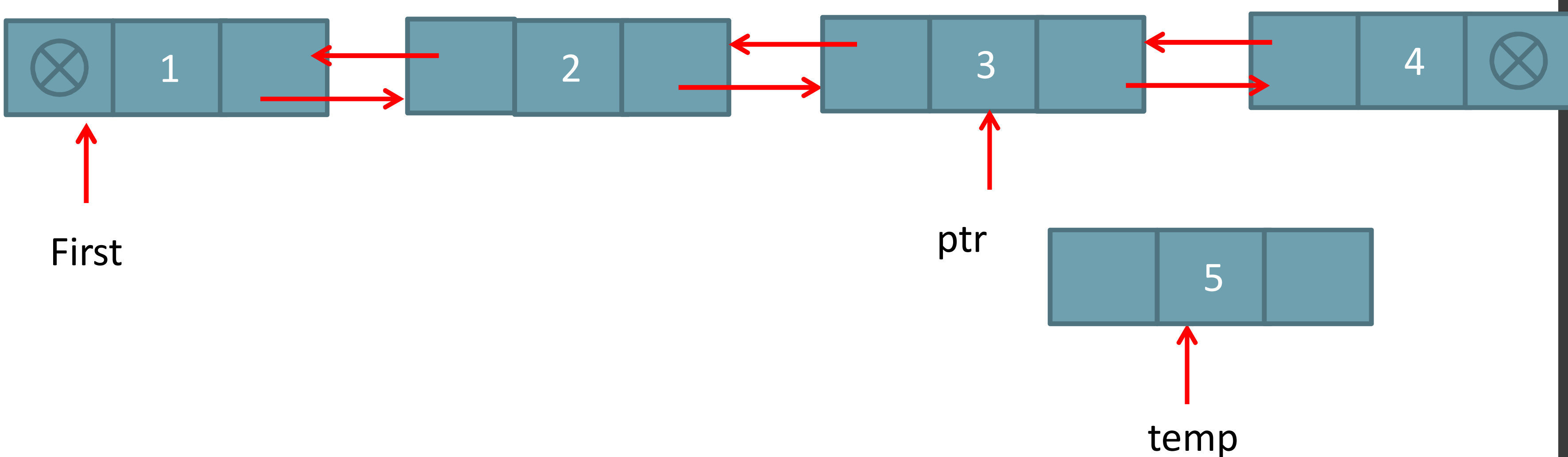

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
temp->prev=ptr;
```

Double Linked List
Insertion: In Between



Step1: Enter a position and move ptr pointer reach to position - 1.

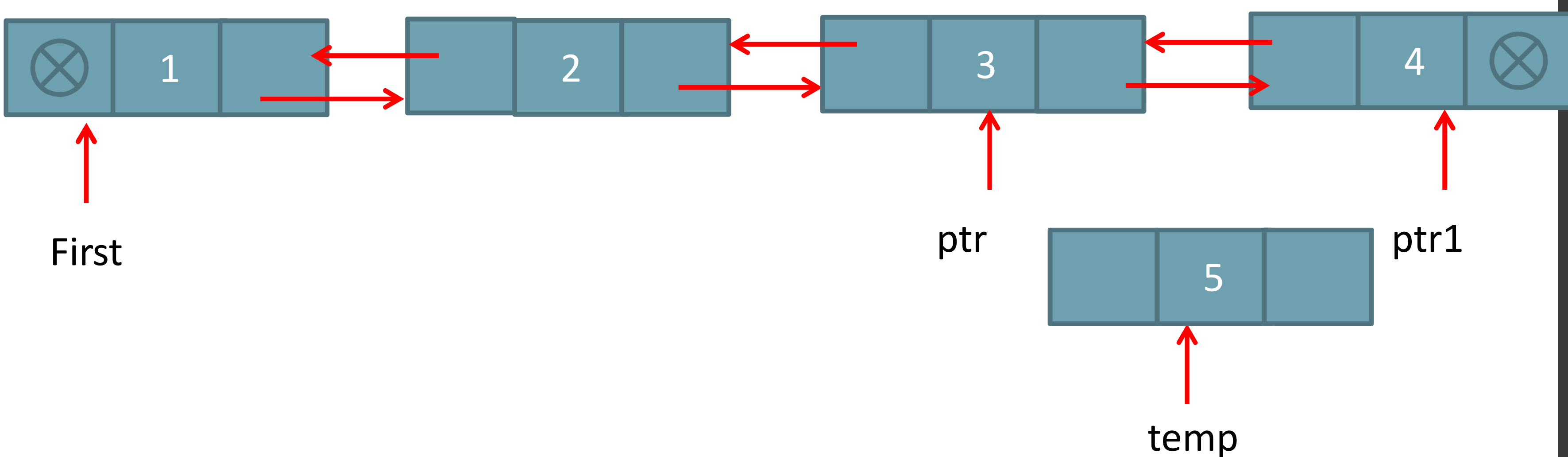
Double Linked List
Insertion: In Between



Step1: Enter a position and move ptr pointer reach to position - 1.

Step2: check for the correctness of ptr, if correct follow the steps below:

Double Linked List
Insertion: In Between

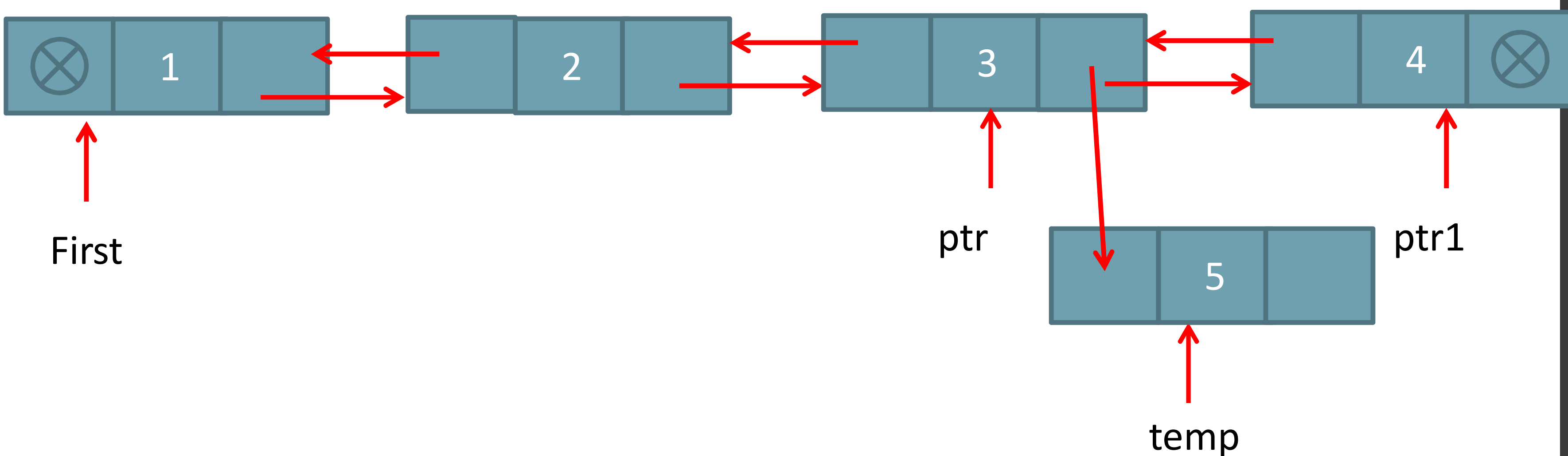


Step1: Enter a position and move ptr pointer reach to position - 1.

Step2: check for the correctness of ptr, if correct follow the steps below:

Step3: ptr1=ptr->next

Double Linked List
Insertion: In Between



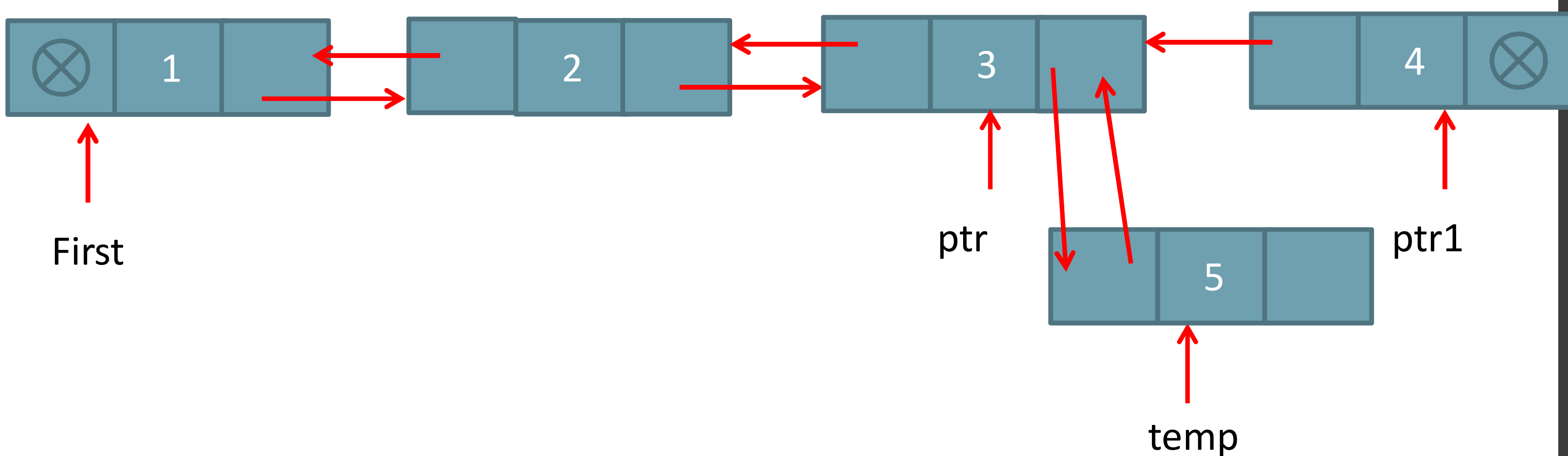
Step1: Enter a position and move ptr pointer reach to position - 1.

Step2: check for the correctness of ptr, if correct follow the steps below:

Step3: ptr1=ptr->next

Step4: i) ptr->next=temp

Double Linked List
Insertion: In Between



Step1: Enter a position and move ptr pointer reach to position - 1.

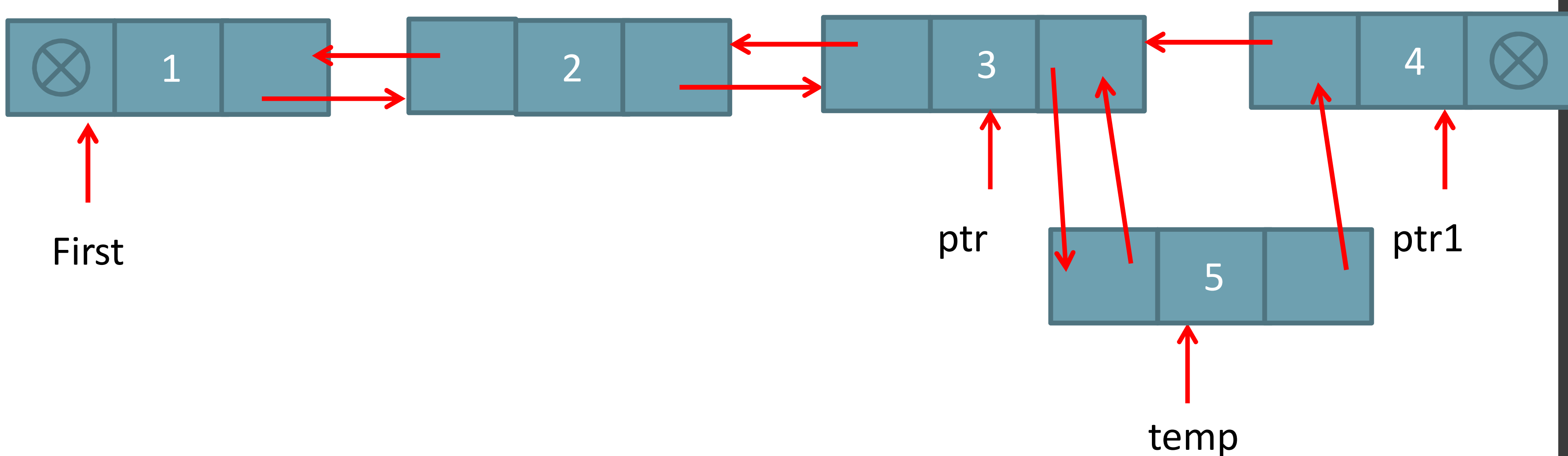
Step2: check for the correctness of ptr, if correct follow the steps below:

Step3: ptr1=ptr->next

Step4: i) ptr->next=temp

ii) temp->prev=ptr

Double Linked List
Insertion: In Between



Step1: Enter a position and move ptr pointer reach to position - 1.

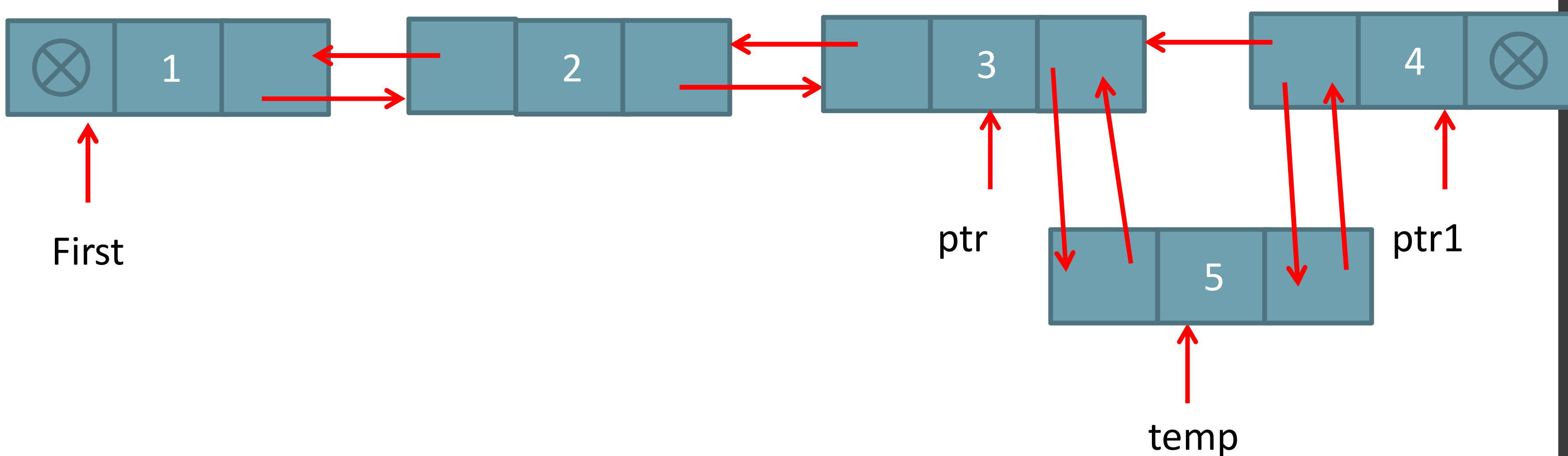
Step2: check for the correctness of ptr, if correct follow the steps below:

Step3: ptr1=ptr->next

Step4: i) ptr1->prev=temp

ii) temp->next=ptr

Double Linked List
Insertion: In Between



Step1: Enter a position and move ptr pointer reach to position - 1.

Step2: check for the correctness of ptr, if correct follow the steps below:

Step3: ptr1=ptr->next

Step4i) ptr1->prev=temp

ii) temp->next=ptr1

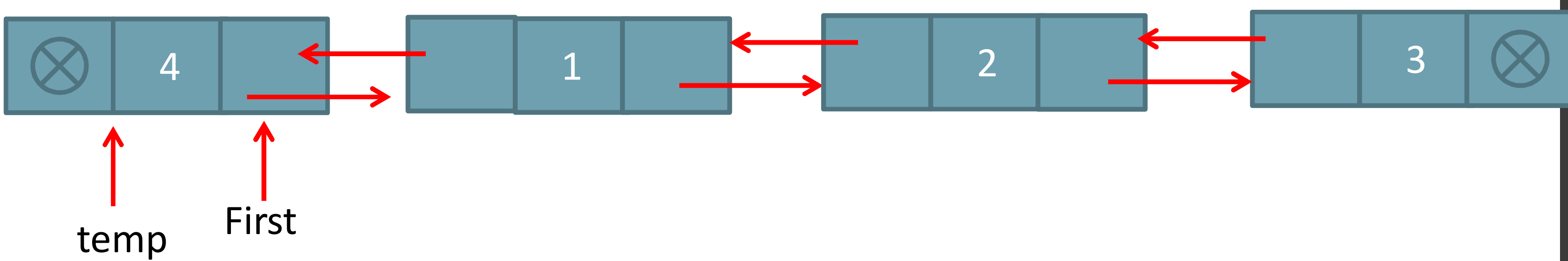
i) ptr->next=temp

ii) temp->prev=ptr

iii) ptr1->prev=temp

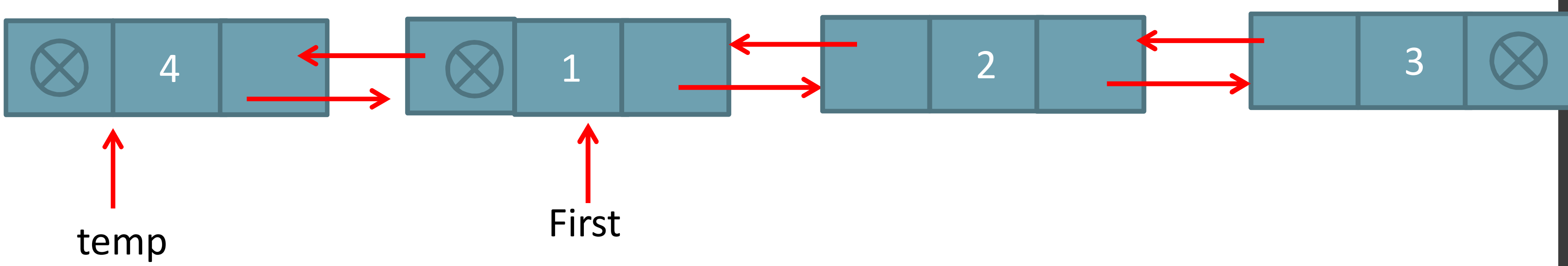
iv) temp->next=ptr1

Double Linked List
Delete node at First Position



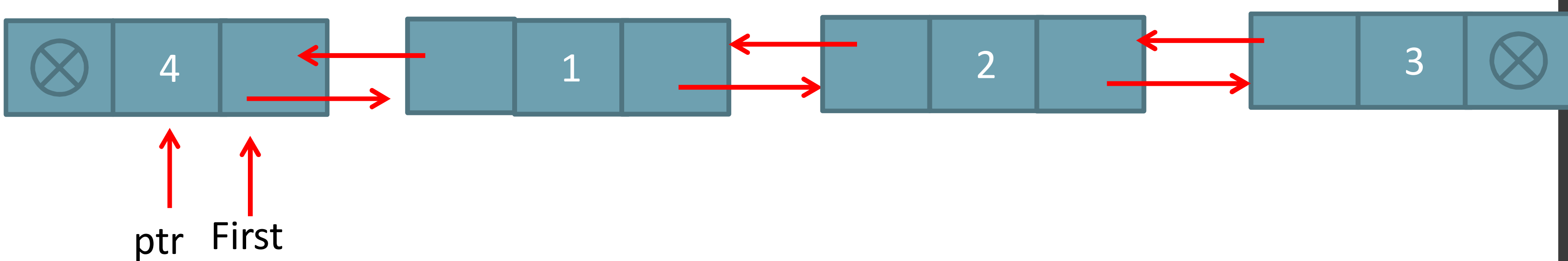

```
temp=first;
first=first->next;
free(temp);
first->prev=NULL;
```

Double Linked List
Delete node at First Position



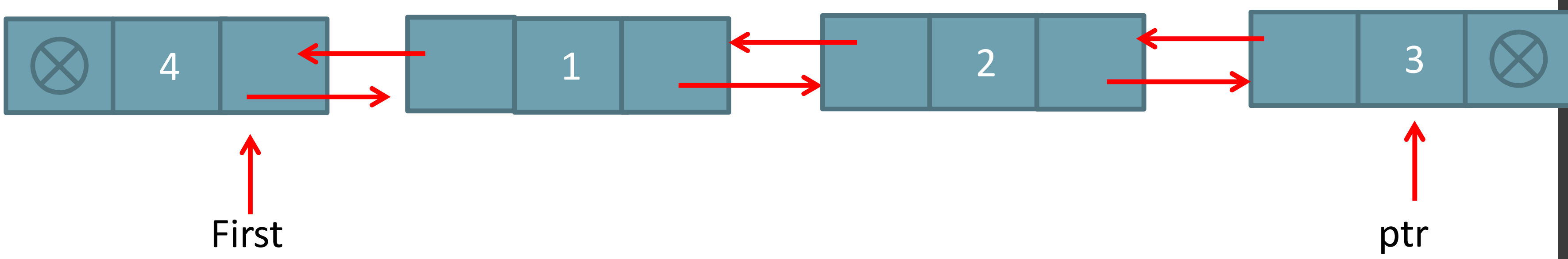

```
temp=first;
first=first->next;
free(temp);
first->prev=NULL;
```

Double Linked List
Delete node at Last Position



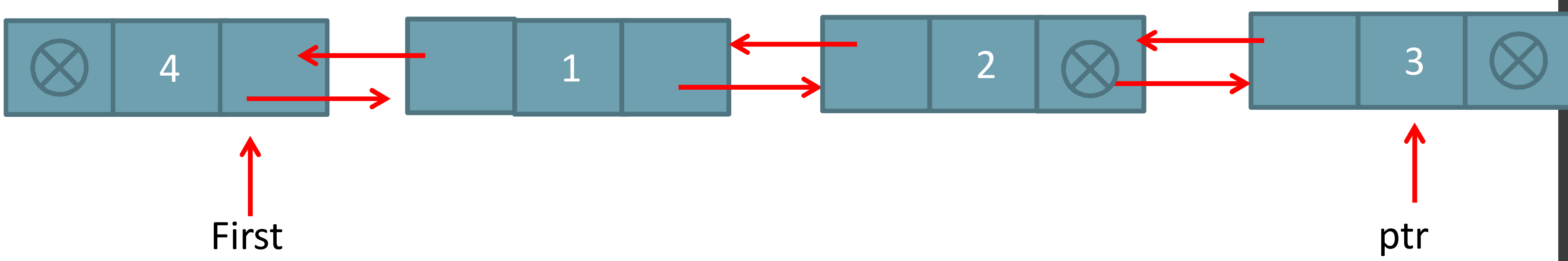

```
ptr=first;
while(ptr->next!=NULL){
            ptr=ptr->next;}
ptr->prev->next=NULL;
free(ptr);
```

Double Linked List
Delete node at Last Position



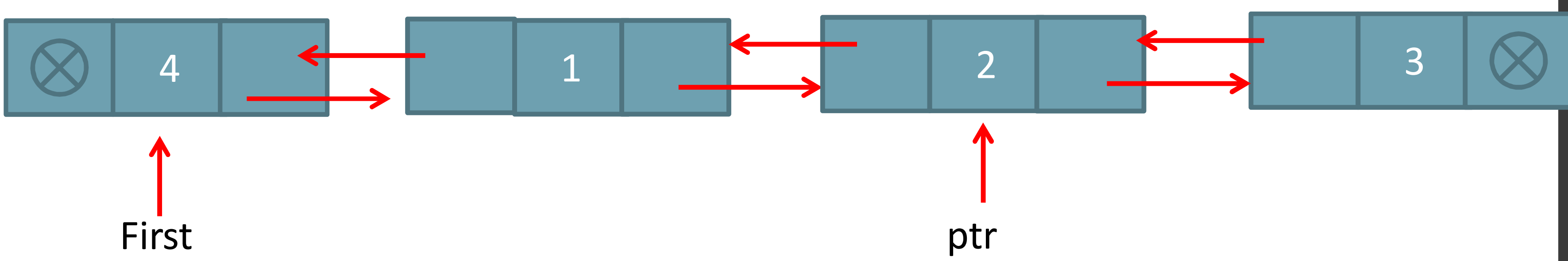

```
ptr=first;
while(ptr->next!=NULL){
        ptr=ptr->next;}
ptr->prev->next=NULL;
free(ptr);
```

Double Linked List
Delete node at Last Position

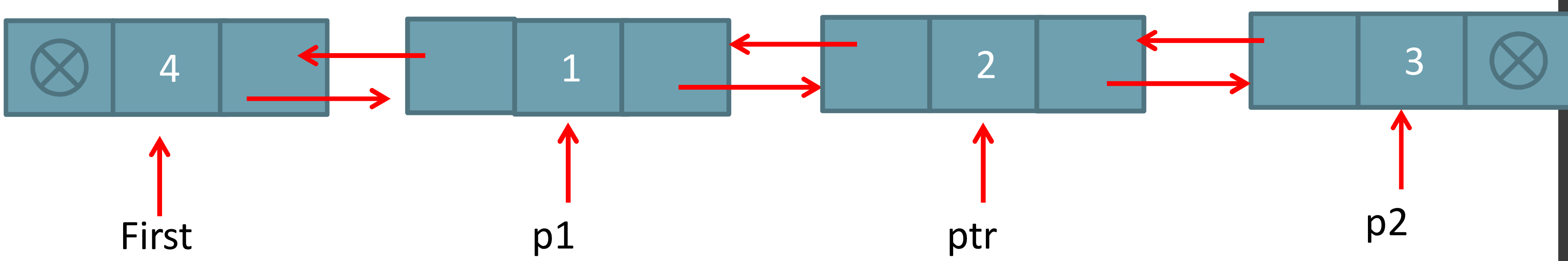



```
ptr=first;
while(ptr->next!=NULL){
        ptr=ptr->next;}
ptr->prev->next=NULL;
free(ptr);
```

Double Linked List
Delete node at some in between position



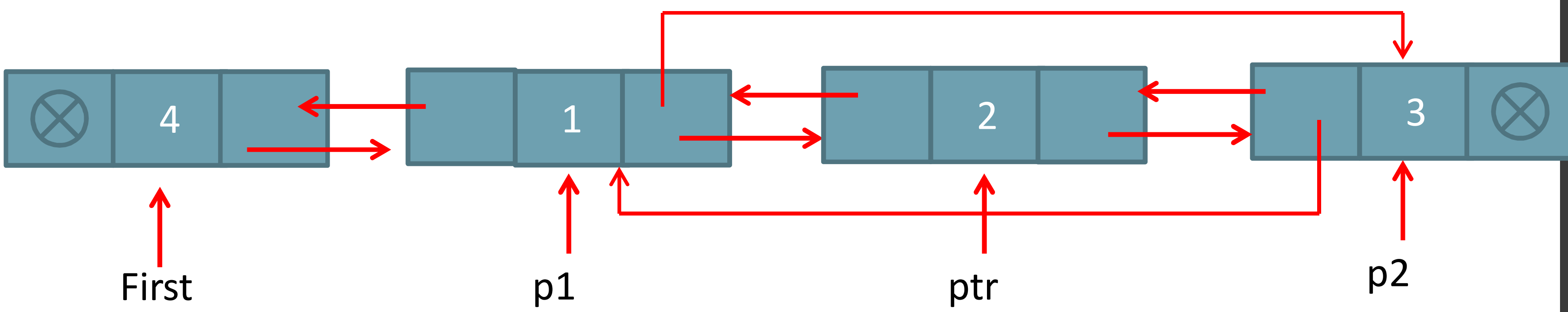
Double Linked List
Delete node at some in between position




```
p1=ptr->prev;
p2=ptr->next;
p1->next=p2;
p2->prev=p1;
```

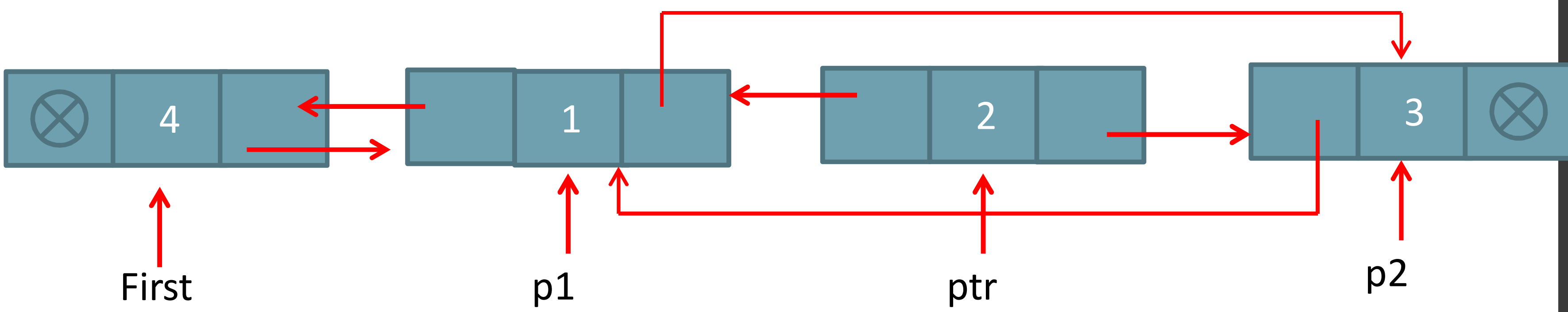

Kanak Kalyani

Double Linked List
Delete node at some in between position



p1=ptr->prev;
p2=ptr->next;
p1->next=p2;
p2->prev=p1;

Double Linked List
Delete node at some in between position




```
p1=ptr->prev;
p2=ptr->next;
p1->next=p2;
p2->prev=p1;
free(ptr);
```

- Sorted Double Linked List
- 1->1 -> 2 -> 2-> 3-> 4-> 5-> 6->6

- Remove duplicate elements from double linked list
- 1-> 2 -> 3-> 4-> 5-> 6

1->1 -> 2 -> 2-> 3-> 4-> 5-> 6->6

```
ptr=first;
while( ptr->next->next!=NULL)
{
        if(ptr->data==ptr->next->data)
        {
                ptr=ptr->next
                p1=ptr->prev;
                p2=ptr->next;
                p1->next=p2;
                p2->prev=p1
                free(ptr);
        }
        else
                ptr=ptr->next
}
Ptr=ptr->next;
P1=ptr->prev;
If(p1->data==ptr->data)
{
        //delete last node
        p1->next=NULL;
        free(ptr);
}
```

1024

First(1000)

1024

First(2000)

1024

First(1000)

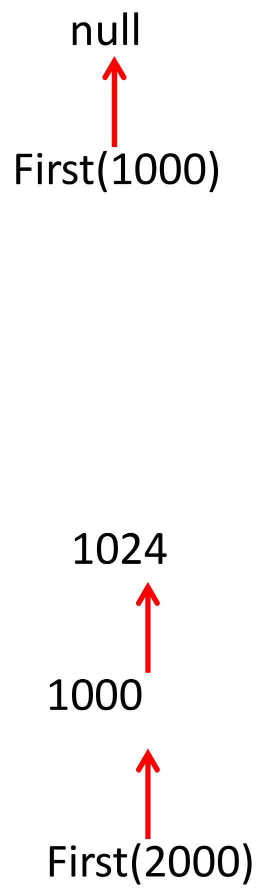
1024

First(1000)

First(2000)

```
Main()
{
Struct node* first;
Create(&first)
Show(first)
}

Void Create(struct node **first)
{
        first =malloc();
```

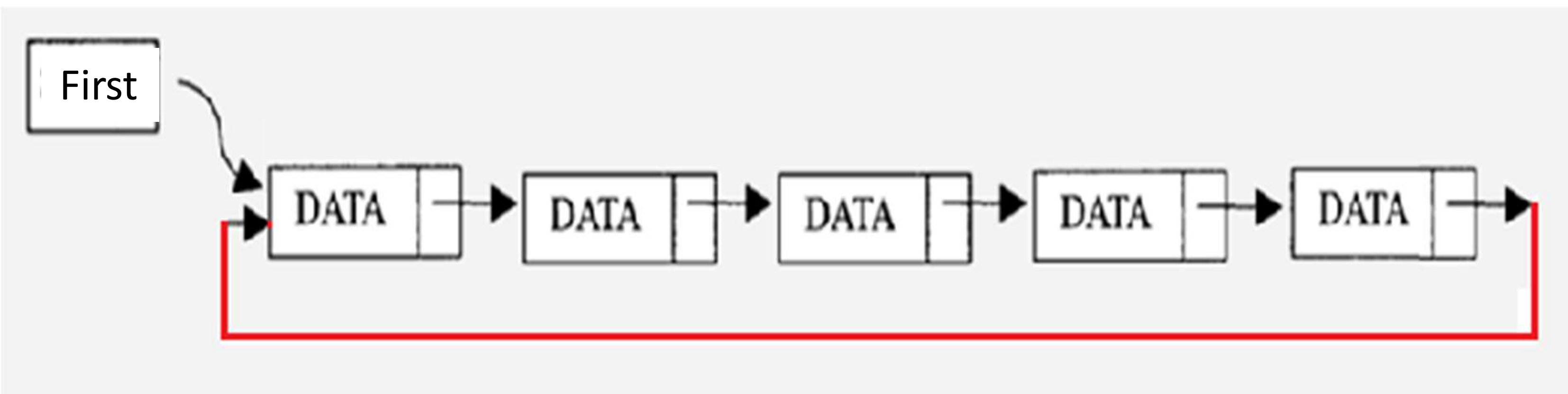


Circular Linked List

Kanak Kalyani

Circular singly Linked list

- Circular singly linked list is a linked list in which last node contains a link to first/start node



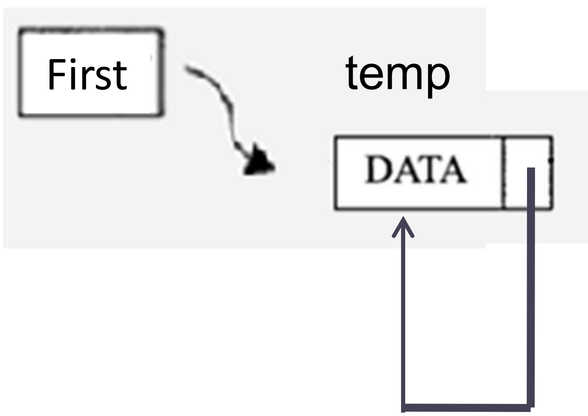
Circular Single Linked list

Node Declaration:

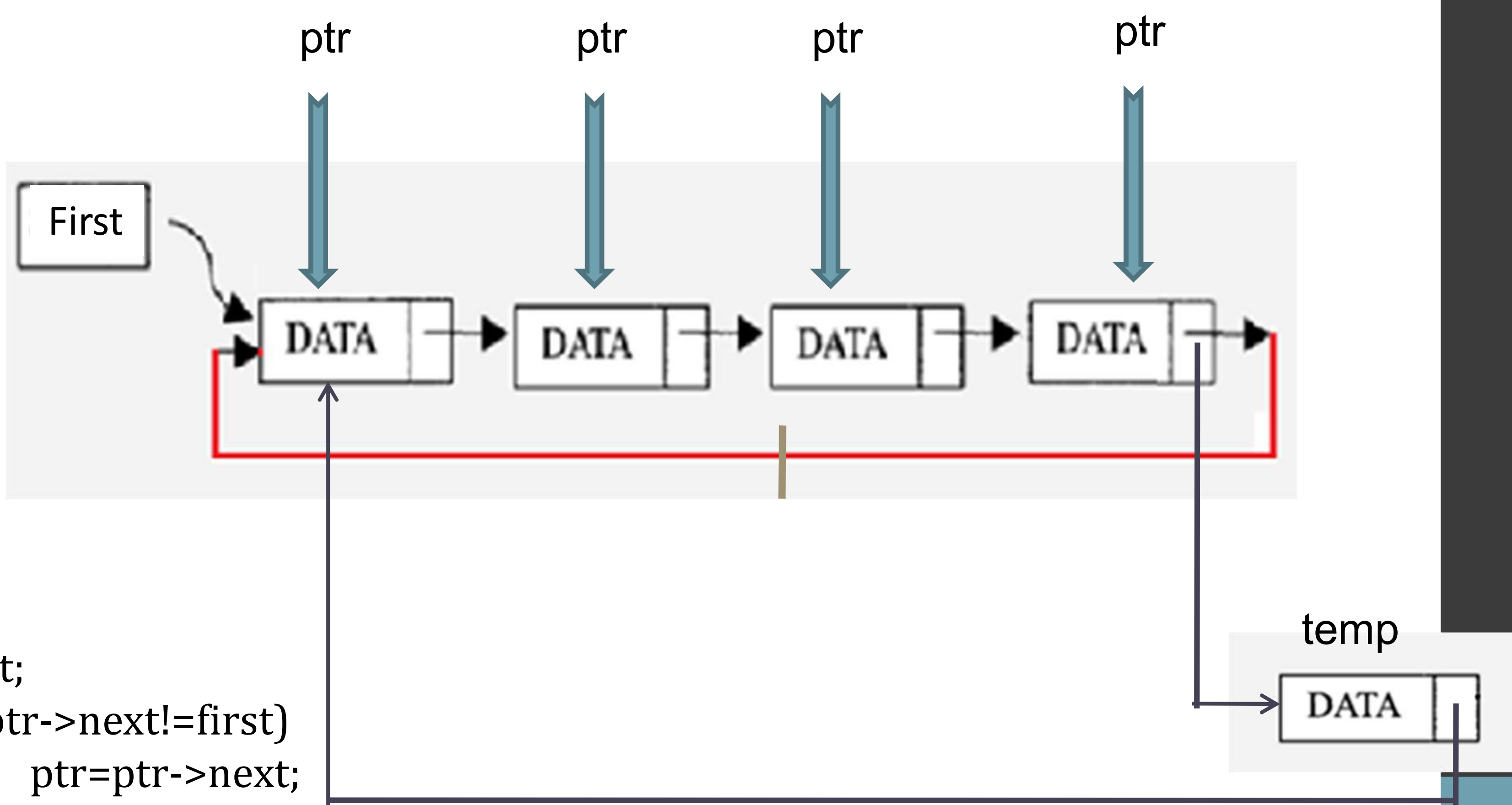
```
typedef struct Node
{
int data;
struct Node *next;
} node;


node *first=NULL;
```

Circular Linked list creation



Circular Linked list creation



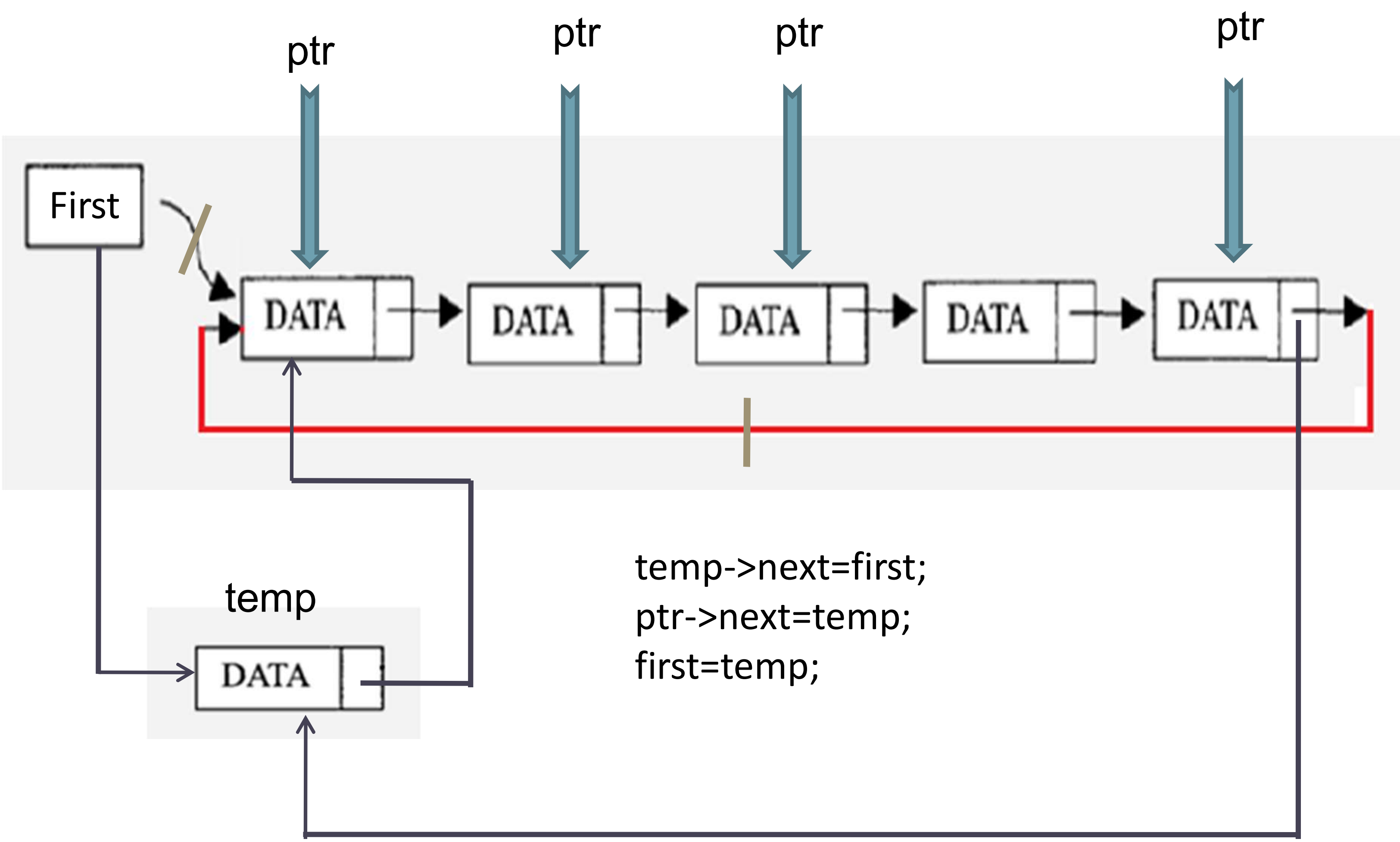
Ptr=first;
While(ptr->next!=first)
ptr=ptr->next;
ptr-> next =temp;
Temp->next=first

Circular Linked list creation

```
temp=(node*)malloc(sizeof(node));
temp -> data=x;
temp -> next=NULL;
if(first == NULL)
{
    first = temp ;
    first -> next=first;
}
else
{
    ptr=first;
    while(ptr -> next != first)
        ptr=ptr -> next;
    ptr -> next= temp;
    temp -> next=first;
}
```

INSERTION

Insertion As First Node

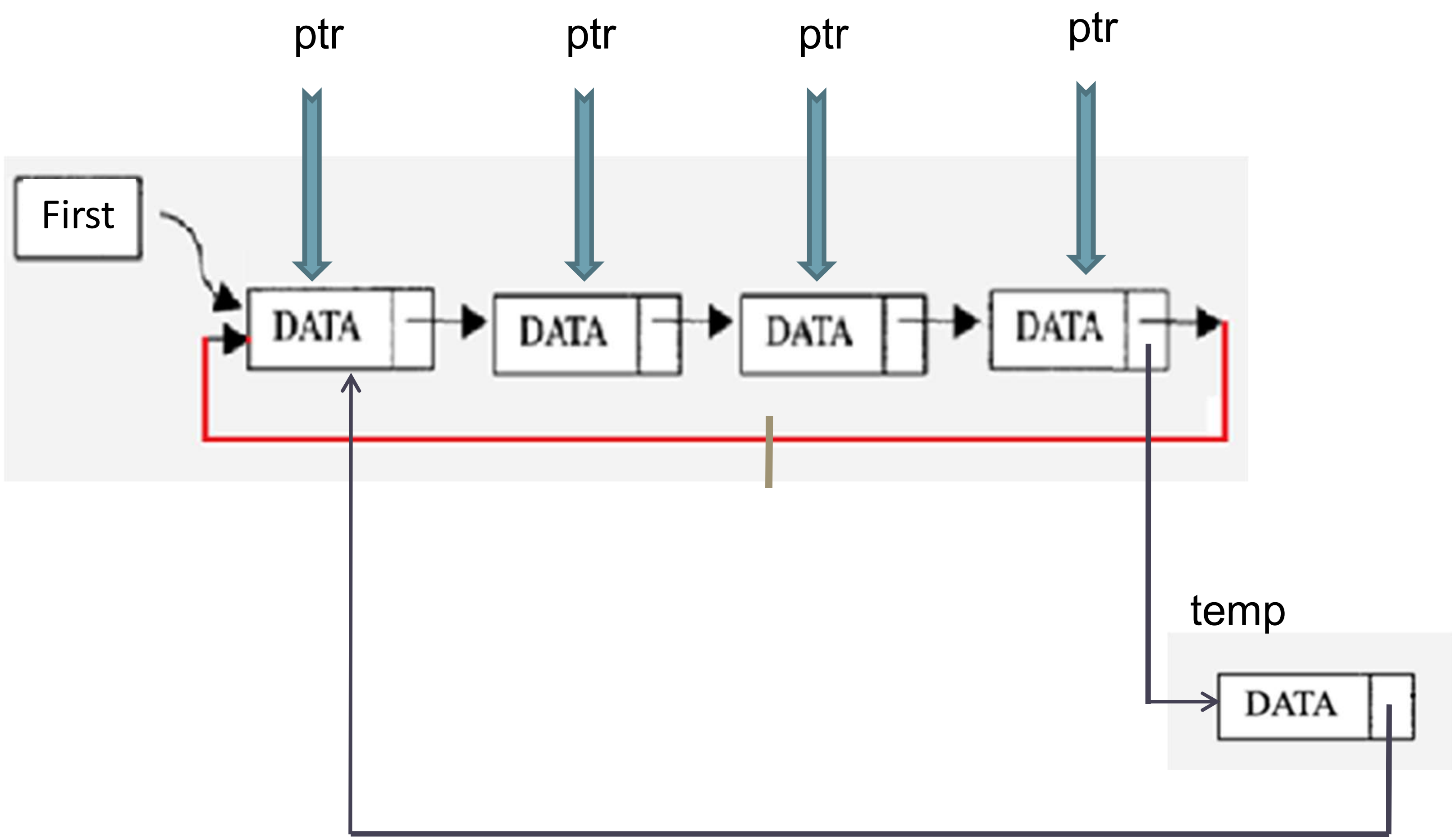


temp->next=first;
ptr->next=temp;
first=temp;

Insertion As First Node

```
ptr=first;
while(ptr -> next != first)
      ptr =ptr -> next;
temp->next=first;
ptr->next=temp;
first=temp;
```

Insertion As Last Node

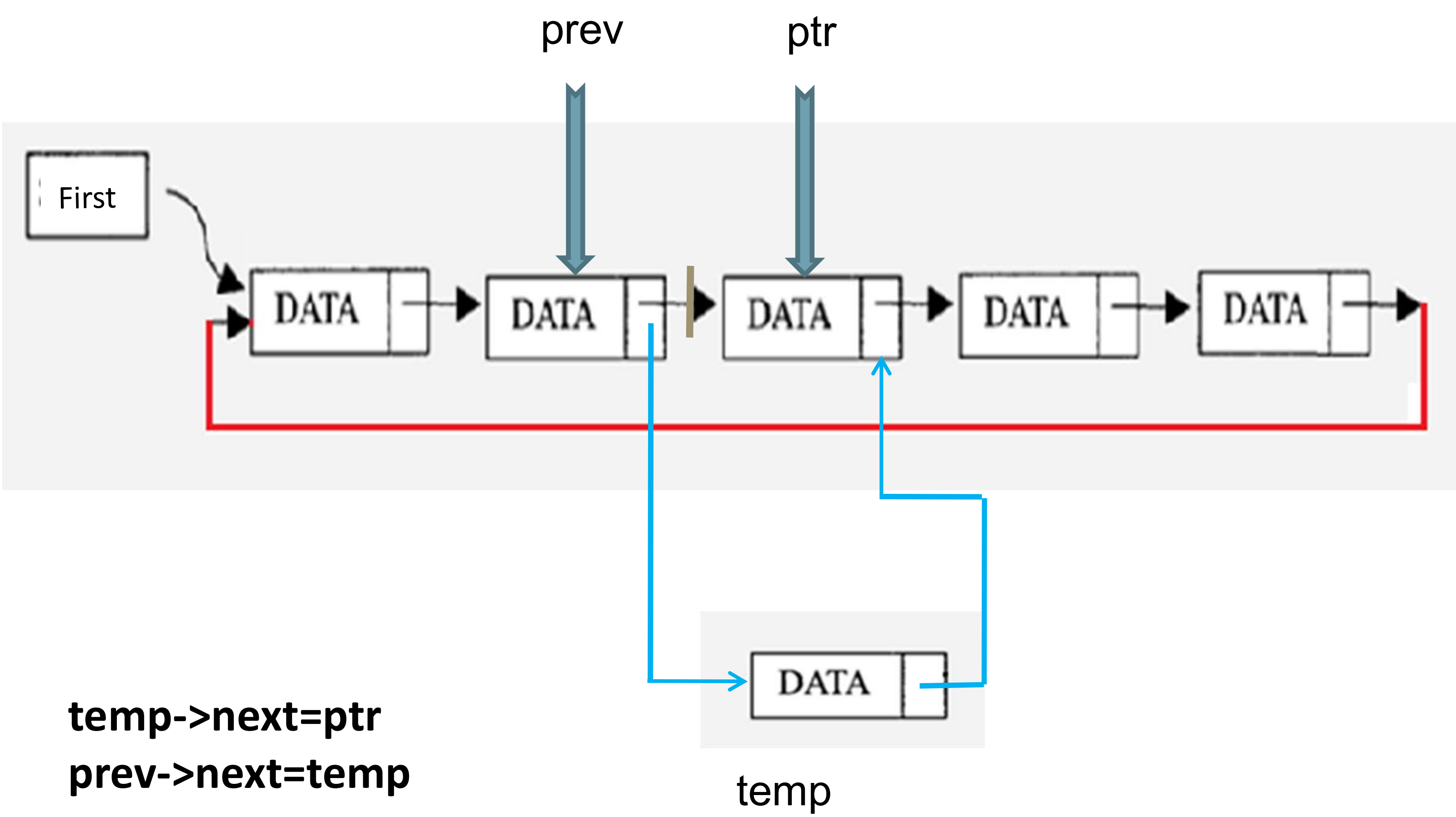


ptr->next=temp;
Temp->next=first

Insertion As Last Node

```
ptr=first;
while(ptr-> next != first)
        ptr= ptr -> next;
ptr -> next = temp;
temp -> next = first;
```

Insertion At The Specified Position



temp->next=ptr
prev->next=temp

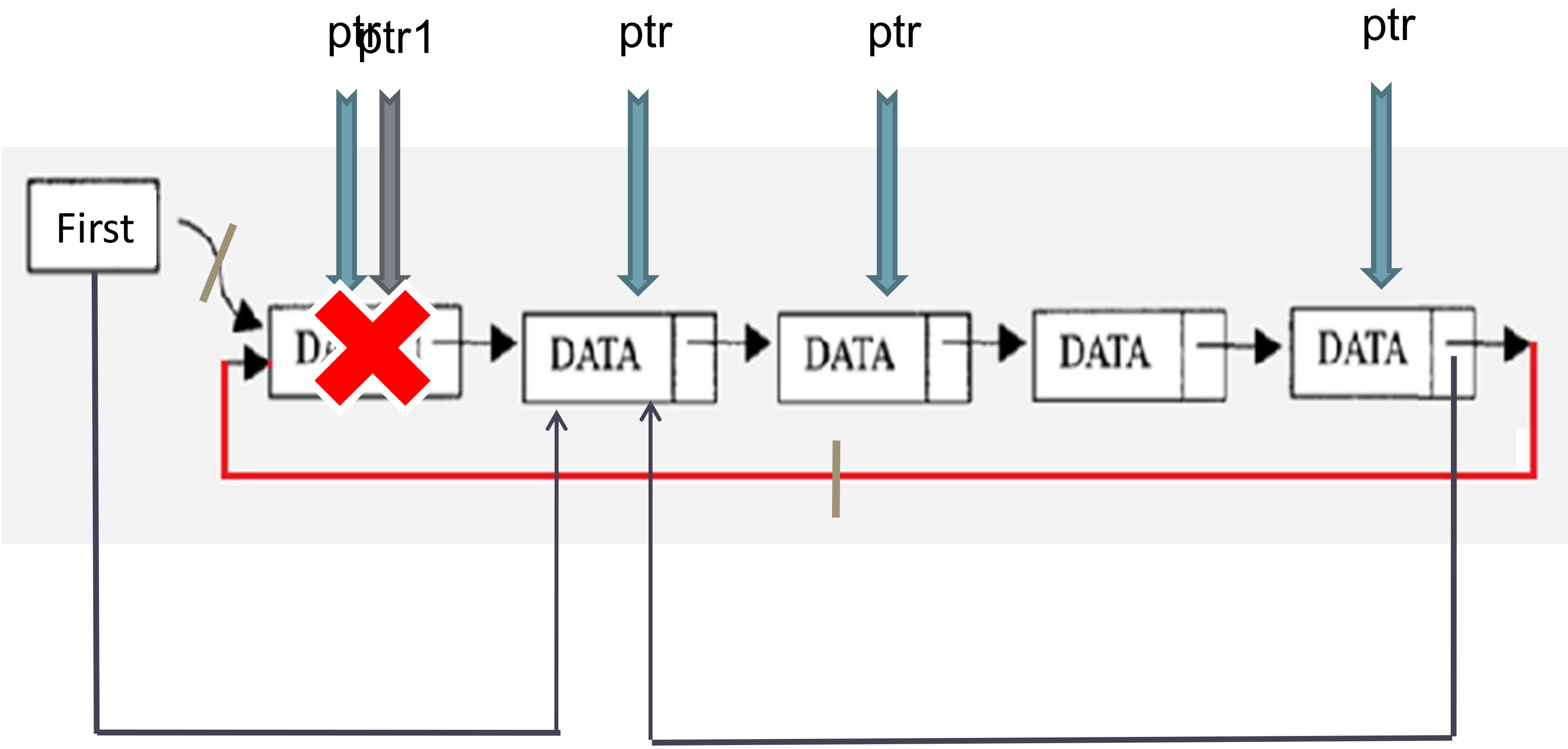
Insertion At The Specified Position

```
ptr=first;
do
{
      prev = ptr;
      ptr=ptr-> next;
      i++;
} while(ptr != first && i != pos);
if(ptr==first)
      printf("No sufficient number of nodes");
else
{
      temp -> next =ptr;
      prev -> next = temp;
}
```

DELETION

Kanak Kalyani

Deletion Of First Node

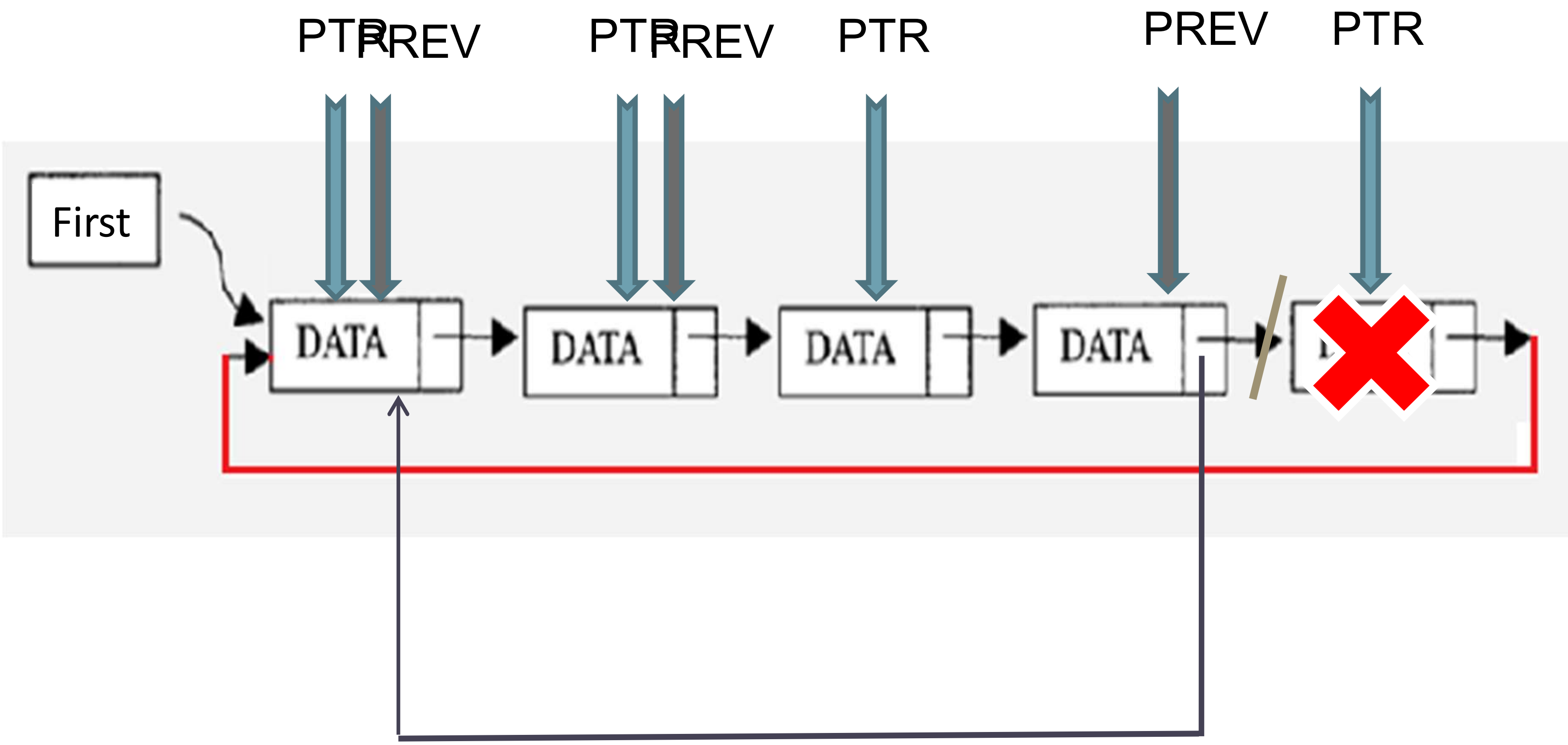


ptr1=first;
first=first->next
ptr->next=first
free(ptr1);

Deletion Of The First Node

```
ptr=first;
while(ptr -> next != first )
        ptr = ptr -> next;
ptr1=first;
first=first->next
ptr->next=first
free(ptr1);
```

Deletion Of The Last Node

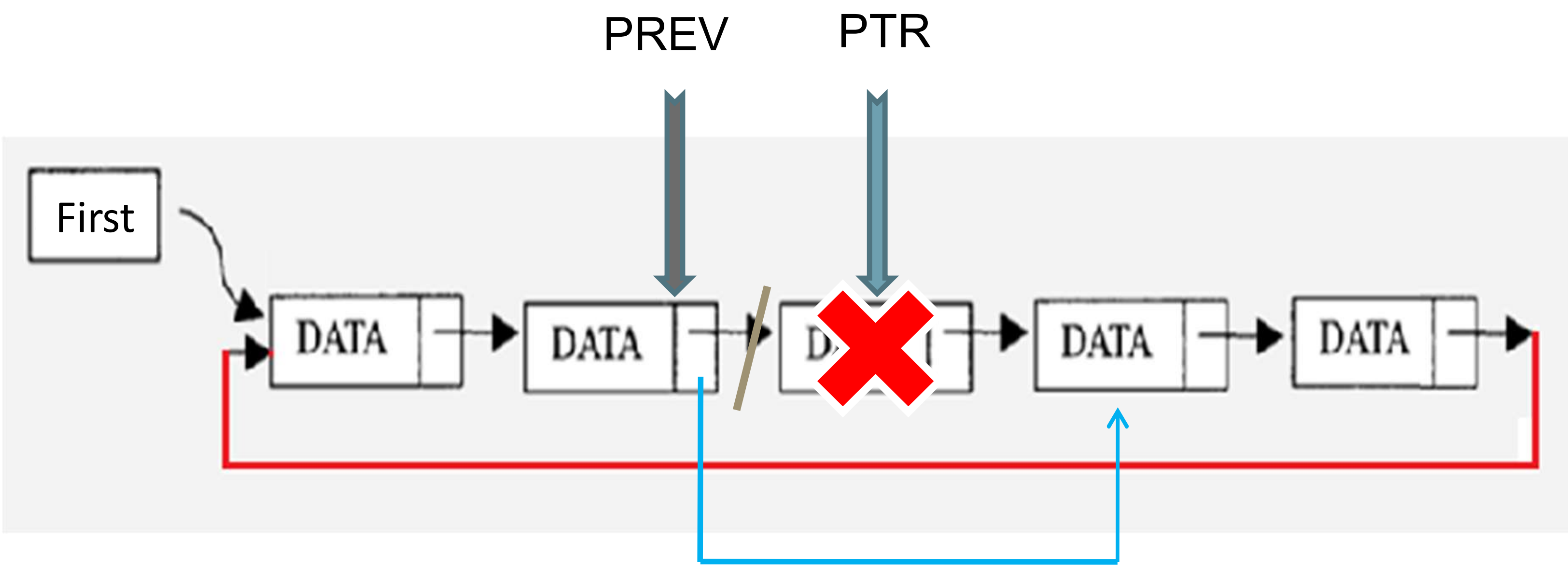


prev->next=first
Free(ptr)

Deletion Of The Last Node

```
ptr=first;
while(ptr -> next != first ){
      prev=ptr;
       ptr = ptr -> next;
}
prev -> next = first;
free( ptr);
```

Deletion Of The Node In Between



prev->next=ptr->next;
free(ptr);

Deletion Of The Node In Between

```
ptr = first;
do
{
        prev = ptr;
        ptr = ptr -> next;
        i++;
} while(ptr != first && i != pos)
if(ptr == first)
        printf("No sufficient number of nodes");
else
{
        prev -> next=ptr-> next;
        free(ptr);
}
```

Update Operation On Circular Linked List

```
ptr=first;
do
{
        if(ptr->data==x)
        {
                ptr->data=xnew;
                break;
        }
        else
                ptr=ptr->next;
} while(ptr!=first);
if ( ptr==first)
        printf("data to be updated is not present");
```

Display Operation On Circular Linked List

```
if(first==NULL)
        printf("*********List is empty********");
else
{
        ptr=first;
        do
        {
                printf("\t%d", ptr->data);
                ptr=ptr->next;
        } while(ptr != first);
}
```

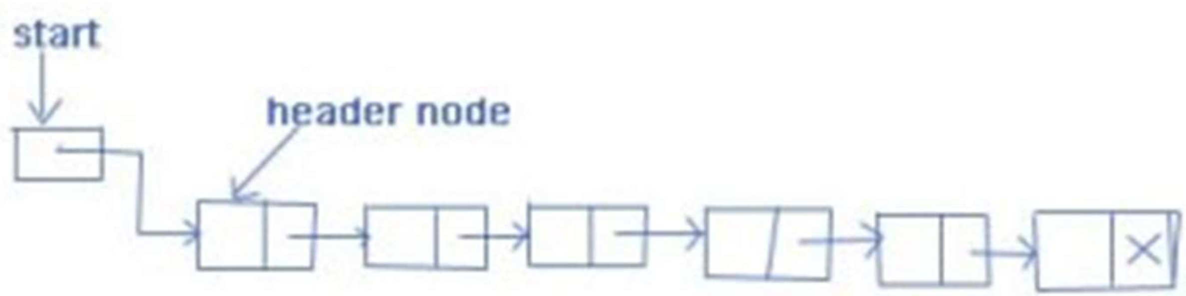
Applications Of Circular Linked List

- An application where any node can be a starting point, we can traverse the whole list by starting from any node and just need to stop when the first visited node is visited again.

- Circular lists are useful in applications to repeatedly go around the list. For example Round Robin (RR) job scheduling by operating system

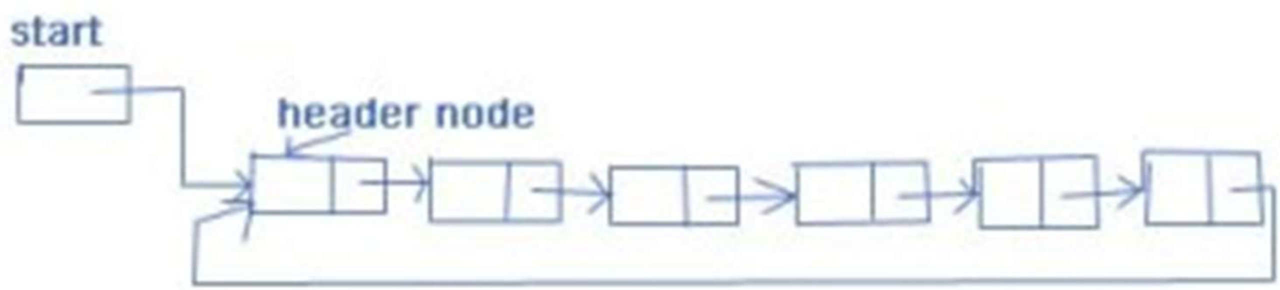
Header Node

- A **header node** is a special **node** that is found at the beginning of the list. A list that contains this type of **node**, is called the **header**-linked list. This type of list is useful when information other than that found in each **node** is needed.



- Example: Header node data may consists of:
  - Number of Nodes in LL
  - Address of Last node
  - Maximum Value in LL

- Circular Header List



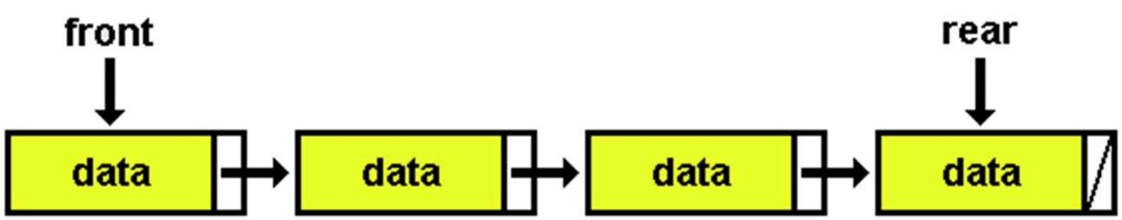
(b) Circular header list.

Practice Problems

- Implement Linked Linear Queue in C.

Linked Queue should have front and rear pointer.

Implement proper enqueue, dequeue operation and display the contents of queue.



Practice Problems

What is the output of the following code

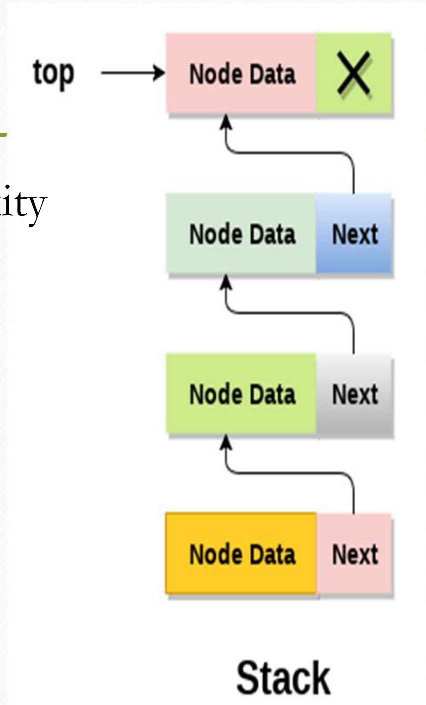
```
void fun1(struct Node* head)
{
if(head == NULL)
        return;

fun1(head->next);
cout << head->data << " ";
}
```

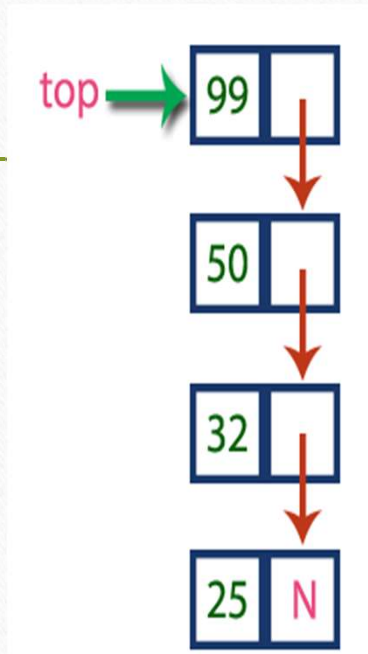
1->2->3->4->5

Linked Stacks(Variants)

Complexity
O(n)



New Item will be added after Top
Top pointed Item will be removed First



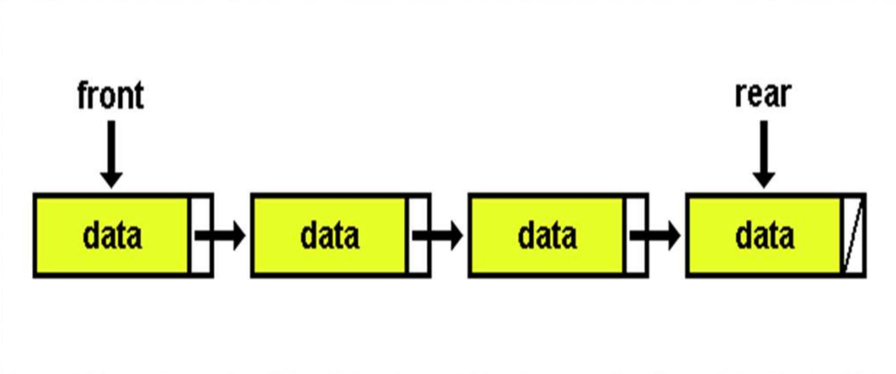
Complexity
O(1)

New Item will be added before Top
Top pointed Item will be removed First

Linked Stacks Implementation

- if(Top==NULL) → Stack is Empty
- Stack will be never full until Newnode allocation is not possible

Linked Queue



Linked Stacks Implementation

- if(Front == NULL) → Queue is Empty
- Queue will be never full until Newnode allocation is not possible
- DeQueue() → Delete operation of First Node
- EnQueue() → Insert operation as Last Node