Data Structures
UNIT – 3: Linked List

Kanak Kalyani

Linked List

- Linked Lists are *dynamic* data structures that grow and shrink one element at a time, normally without some of the inefficiencies of arrays.
- A *linked list* is a series of connected *nodes*

- We create a new node every time we add something to the List and we remove nodes when item removed from list and reclaim memory

Linked List and Arrays

Arrays

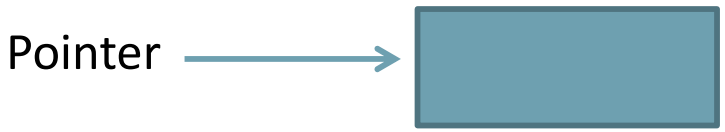
- The array size is fixed once it is created: Changing the size of the array requires creating a new array ad then copying all data from the old array to the new array
- The data items in the array are next to each other in memory: Inserting an item inside the array requires shifting other items

- A linked structure is introduced to overcome limitations of arrays and allow easy insertion and deletion

- A linked structure is introduced to overcome limitations of arrays and allow easy insertion and deletion
  - A collection of nodes storing data items and links to other nodes
  - If each node has a data field and a reference field to another node called next or successor, the sequence of nodes is referred to as a singly linked list
  - Nodes can be located anywhere in the memory, and no wastage of space
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.

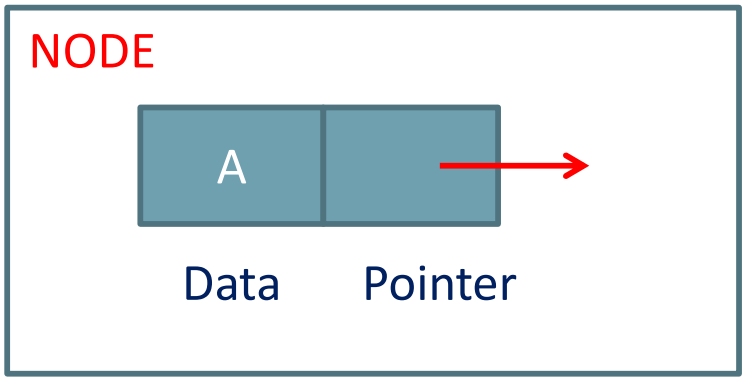
Linked Structures

- An alternative to array-based implementations are *linked structures*
- A linked structure uses pointers to create links between objects



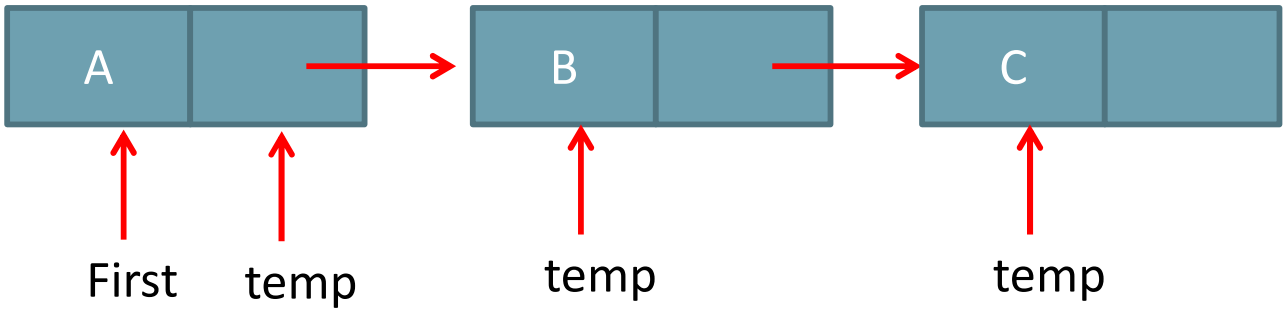
Linked List

- Each node contains at least
  1. A piece of data (any type)
  2. Pointer to the next node in the list
- The last node points to `NULL`




```
struct node
{
        char data;
        struct node *next;
};
```

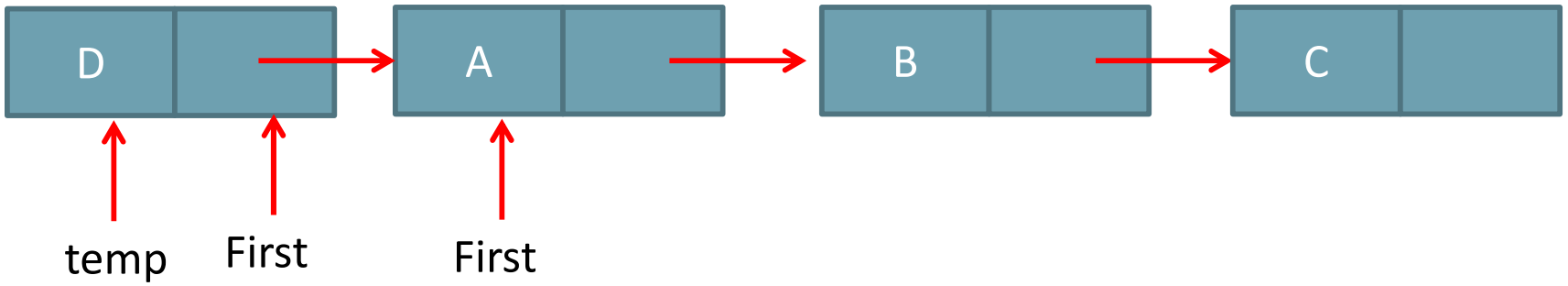
Creating a Linked List



```
void create()
{
        int n,i,x;
        first=NULL;
        printf("how many nodes: ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("Enter data: ");
                scanf(" %d",&x);
                temp=(node*)malloc(sizeof(node));
                temp->data=x;
                temp->next=NULL;
```

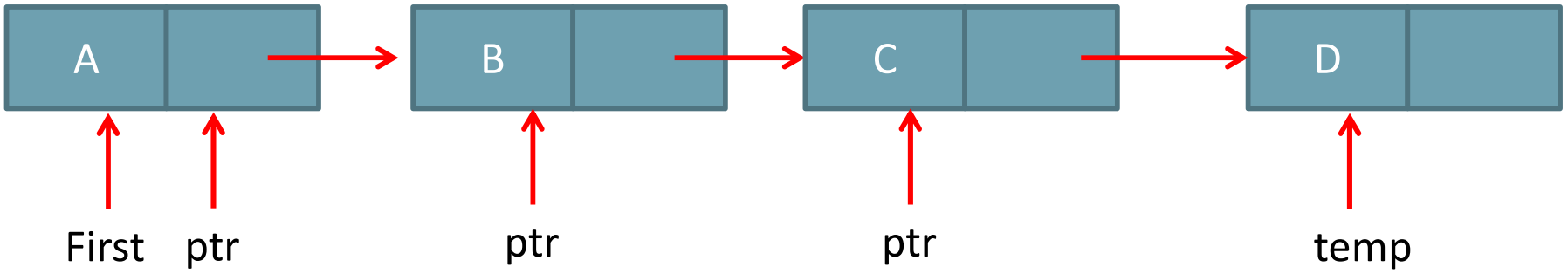
```
if(first==NULL)
                    first=temp;
          else
          {
                    ptr=first;
                    while(ptr->next!=NULL)
                              ptr=ptr->next;
                    ptr->next=temp;
          }
     }
}
```

Inserting a new Node as First Node




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

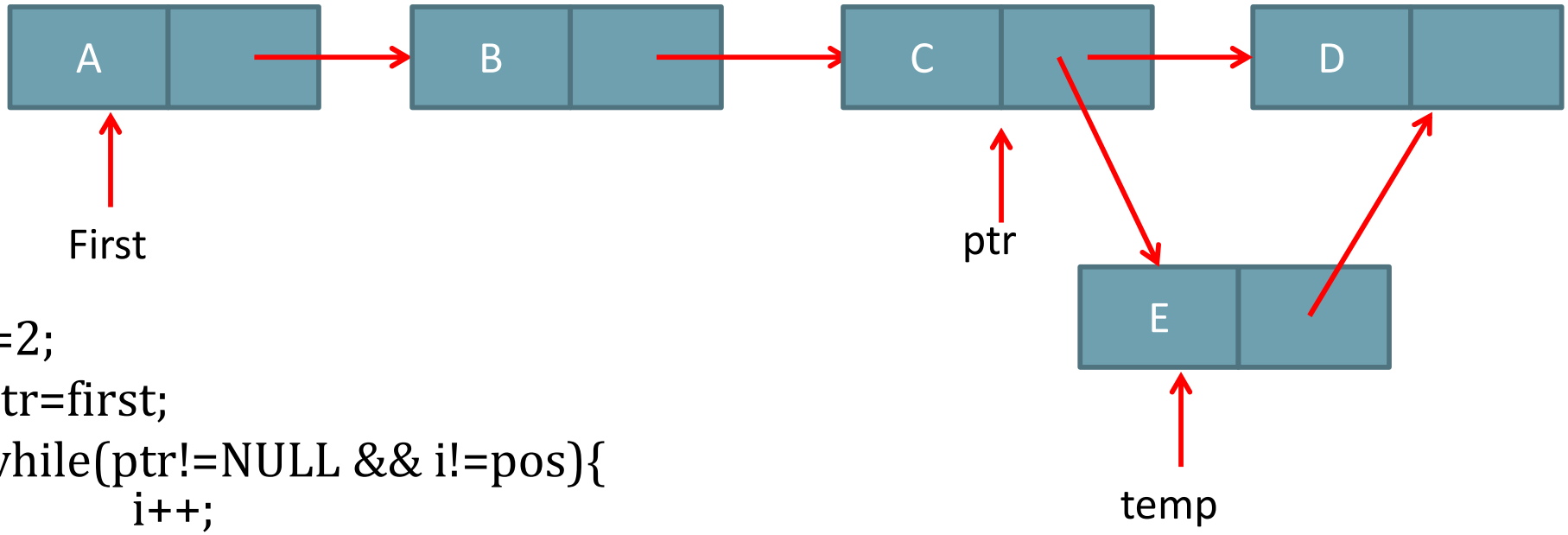
Inserting a new Node as Last Node




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

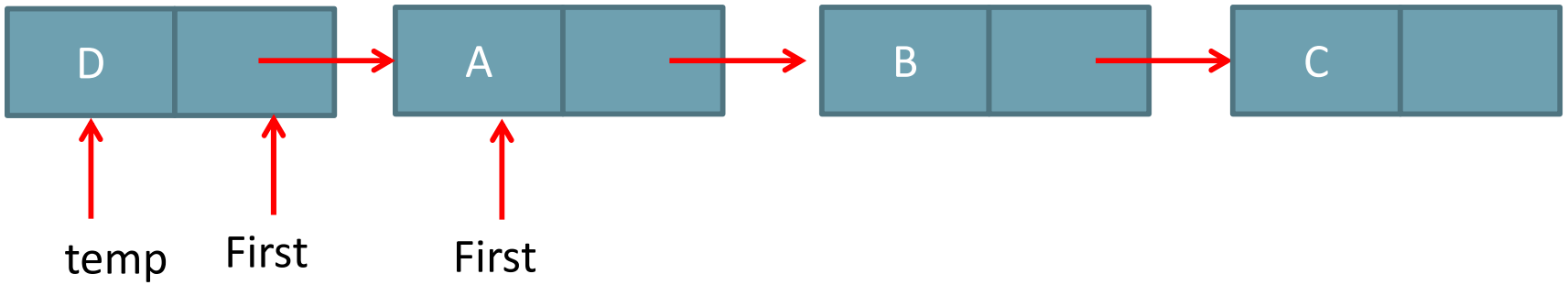
Inserting a new Node: In Between

(Also works for insertion at last)



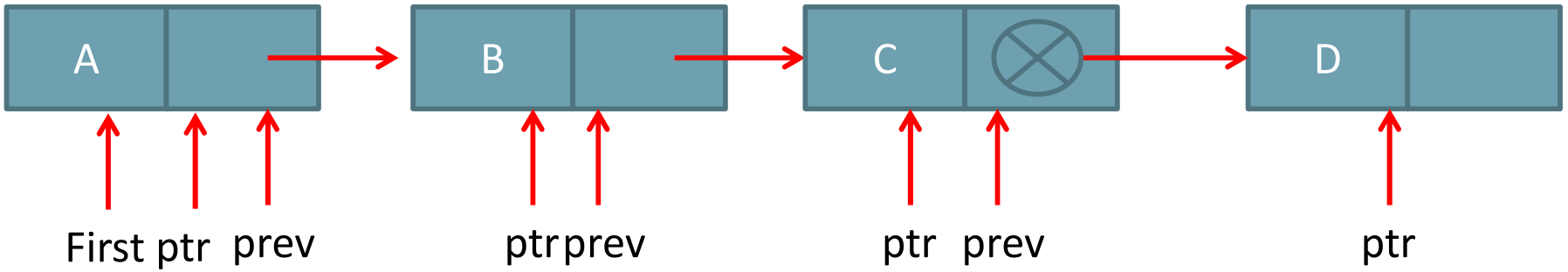

```
i=2;
ptr=first;
while(ptr!=NULL && i!=pos){
        i++;
        ptr=ptr->next;}
if(ptr==NULL)
        printf("insufficient number of nodes");
else{
        temp->next=ptr->next;
        ptr->next=temp;
}
```

Delete a node: First Node



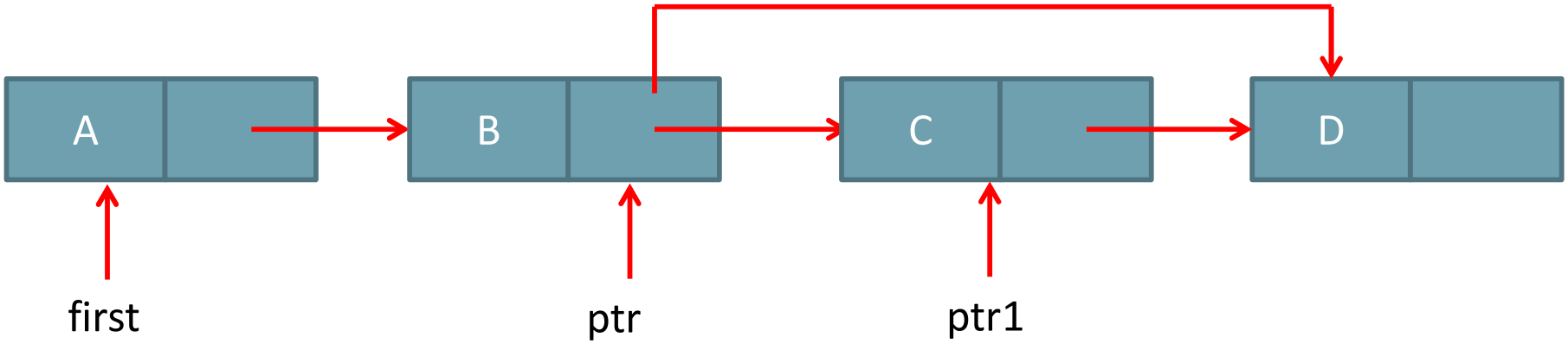

```
temp=first;
first=first->next;
free((void*)temp);
```

Delete a node: Last Node



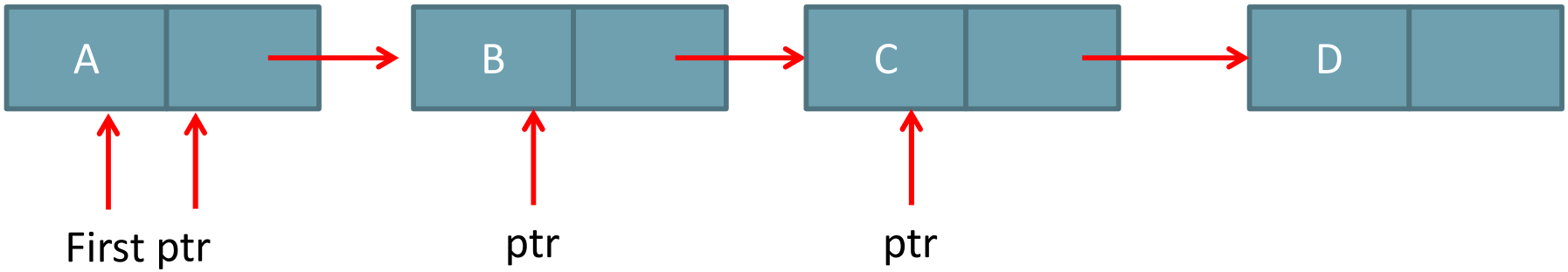

```
ptr=first;
while(ptr->next!=NULL){
        prev=ptr;
        ptr=ptr->next;
}
prev->next=NULL;
free(ptr);
```

Delete a node: Middle Element




```
printf("Enter the position(in between)\n");
scanf("%d",&pos);
ptr=first;
for(i=1;i<pos-1;i++)
        ptr=ptr->next;
if(ptr==NULL)
        printf("there are not sufficient number of nodes");
else{
        ptr1=ptr->next
        ptr->next=ptr1->next;
        free(ptr1);
}
```

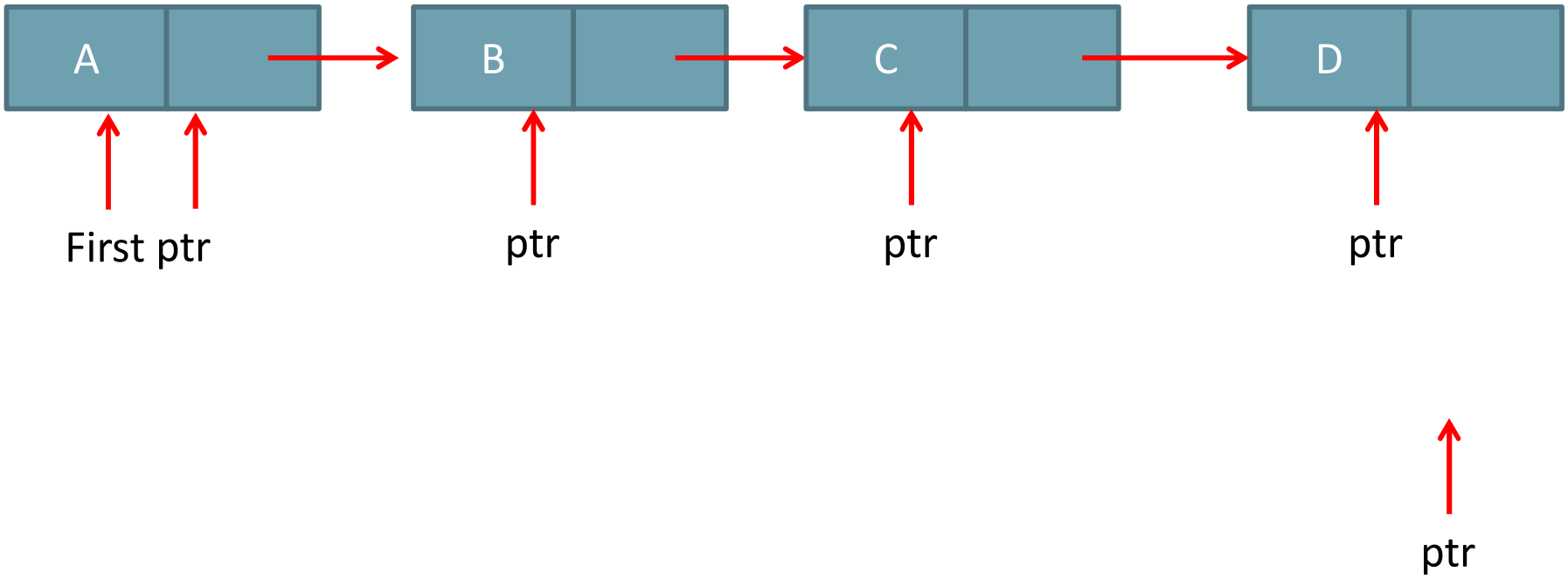
Update the value of a node




```
i=1;
ptr=first;
while(ptr!=NULL && i!=pos){
                i++;
                ptr=ptr->next;}
if(ptr==NULL)
                printf("insufficient number of nodes");
else{
                ptr->data=newdata;
}
```

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

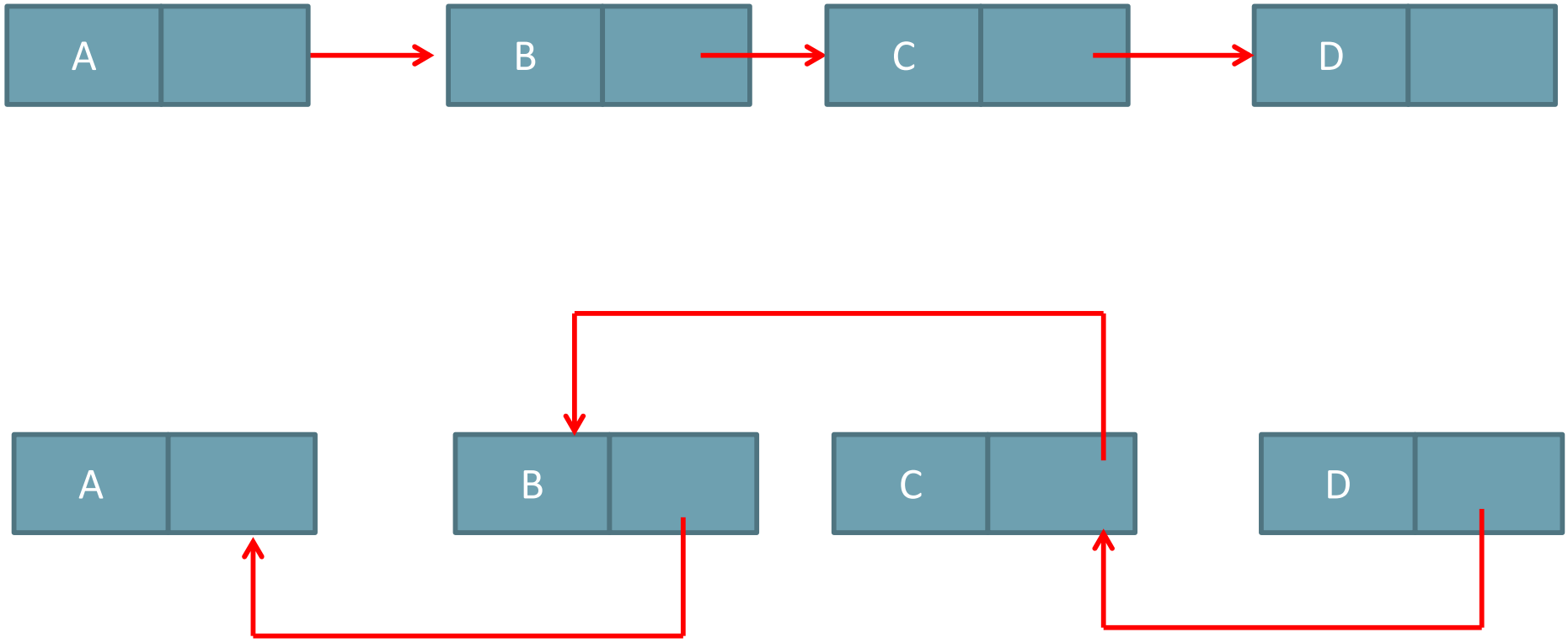
Display



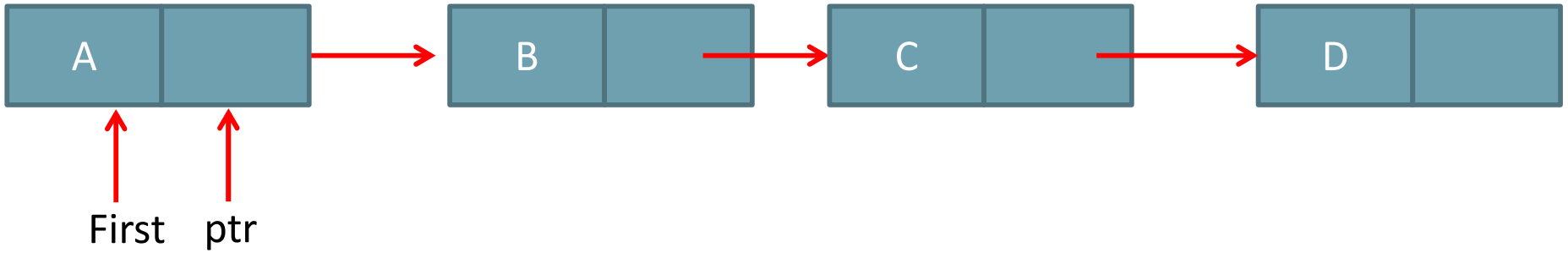
Operations on Linked List

- Reverse the Linked List
- Merge two Linked List
- Sort the Linked List
- Find an element
- Find Max element
- Find Min element

Reverse the Linked List



Reverse the Linked List

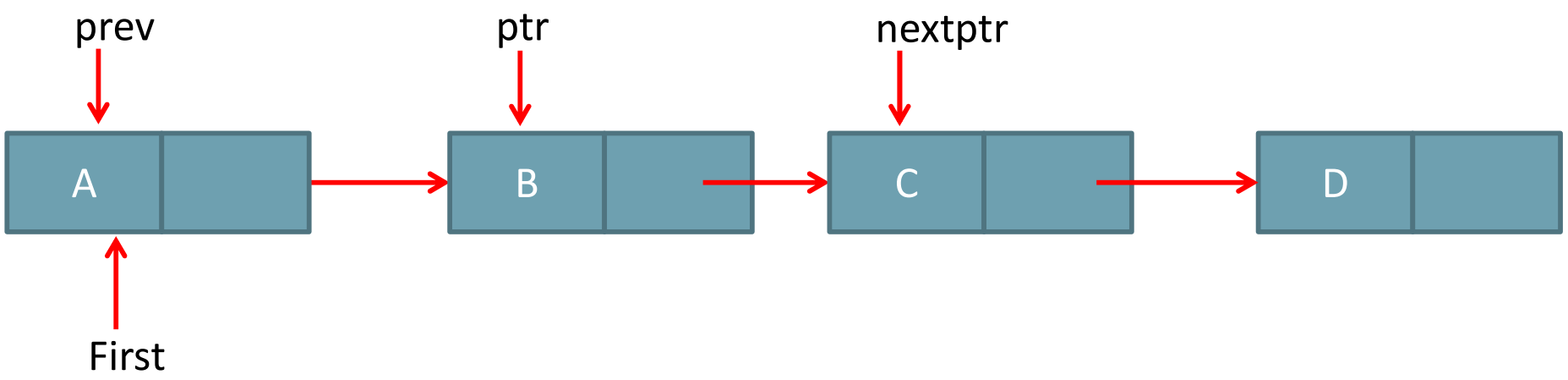


prev= ptr

ptr=ptr->next

nextptr=ptr->next

Reverse the Linked List



Step1:

prev= ptr

ptr=next

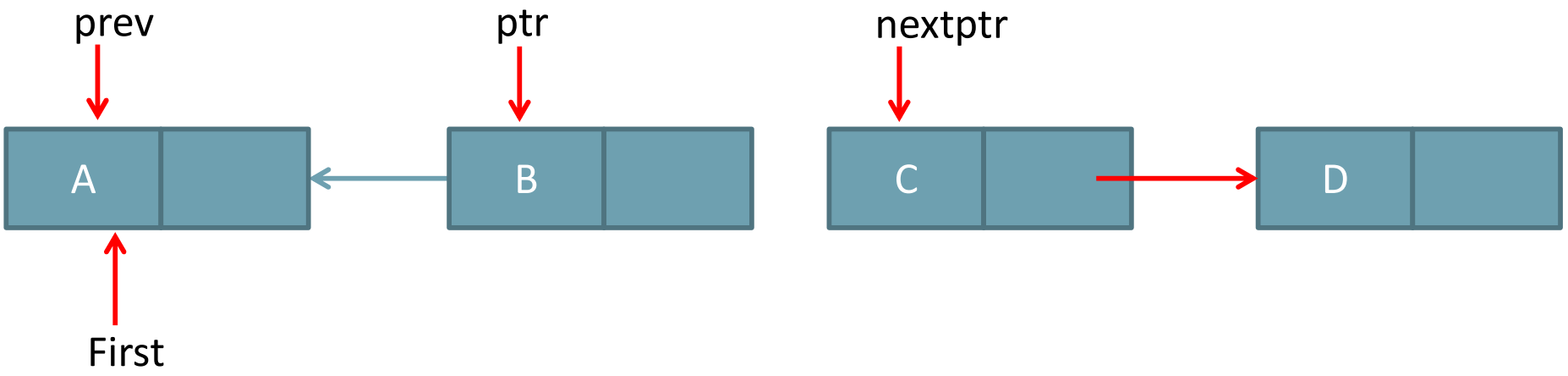
nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

Reverse the Linked List



Step1:

prev= ptr

ptr=ptr->next

nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

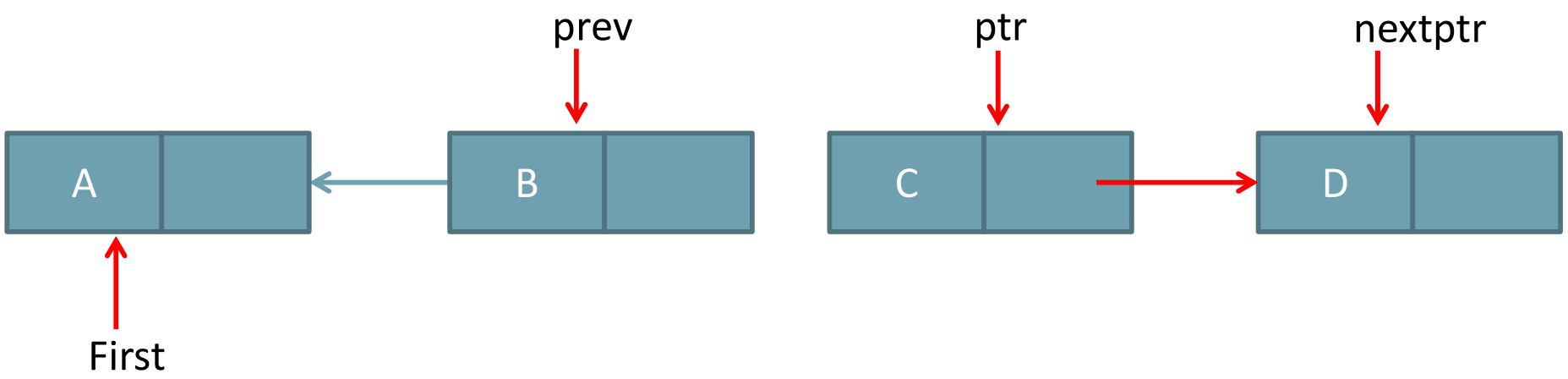
Step3:

prev= ptr

ptr=nextptr

nextptr= nextptr->next

Reverse the Linked List



Step1:

prev= ptr

ptr=ptr->next

nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

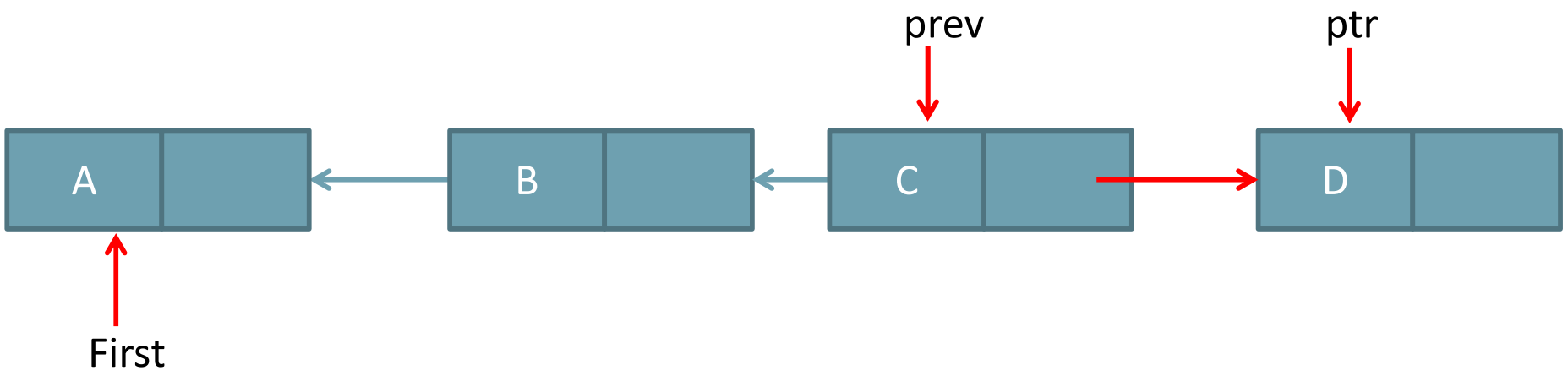
Step3:

prev= ptr

ptr=nextptr

nextptr= nextptr->next

Reverse the Linked List



Step1:

prev= ptr

ptr=ptr->next

nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

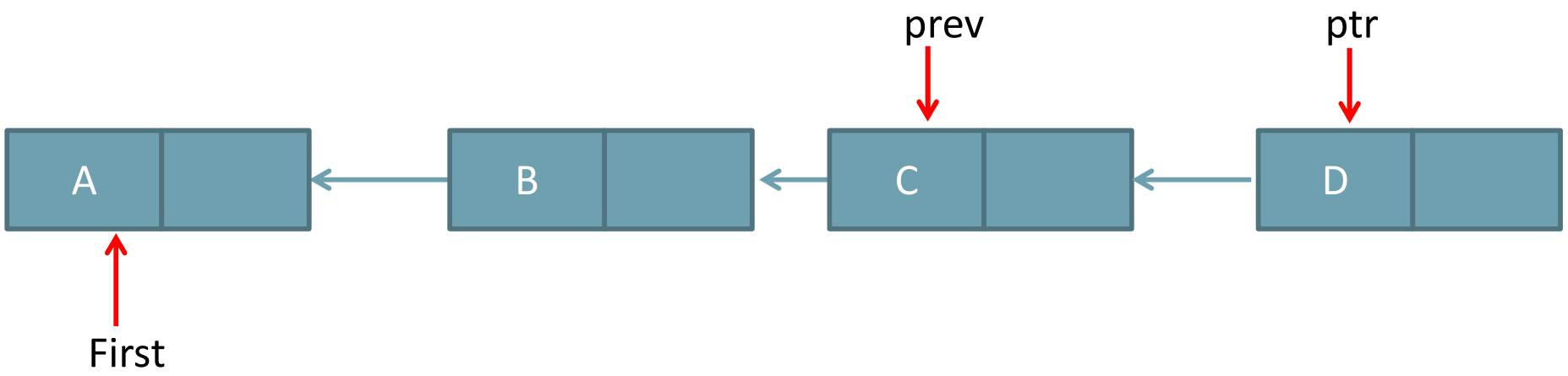
Step3:

prev= ptr

ptr=nextptr

nextptr= nextptr->next

Reverse the Linked List



Step1:

prev= ptr

ptr=ptr->next

nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

Step3:

prev= ptr

ptr=nextptr

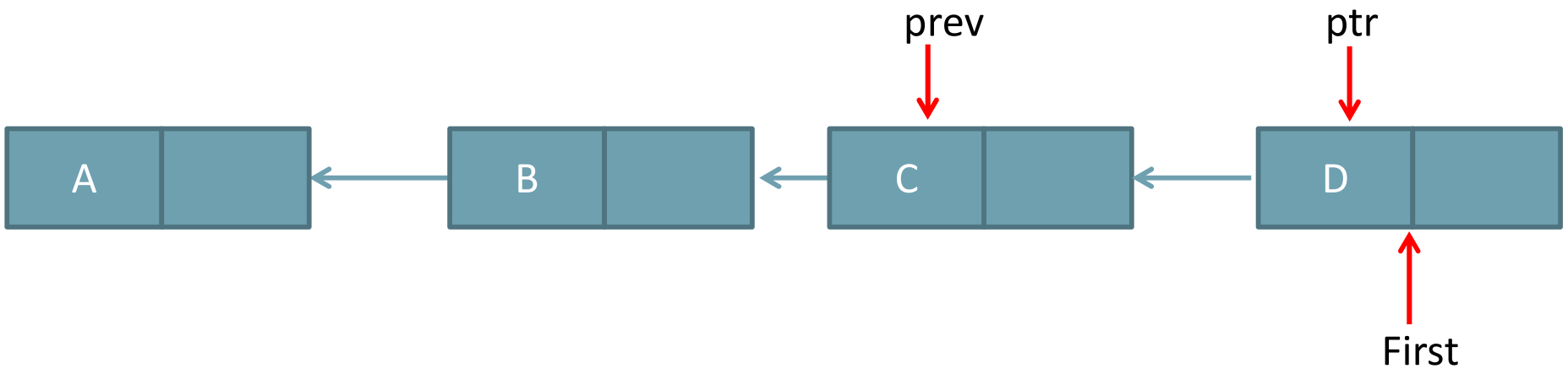
nextptr= nextptr->next

Step 4: change first pointer

first->next=null

first = ptr

Reverse the Linked List



Step1:

prev= ptr

ptr=ptr->next

nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

Step3:

prev= ptr

ptr=nextptr

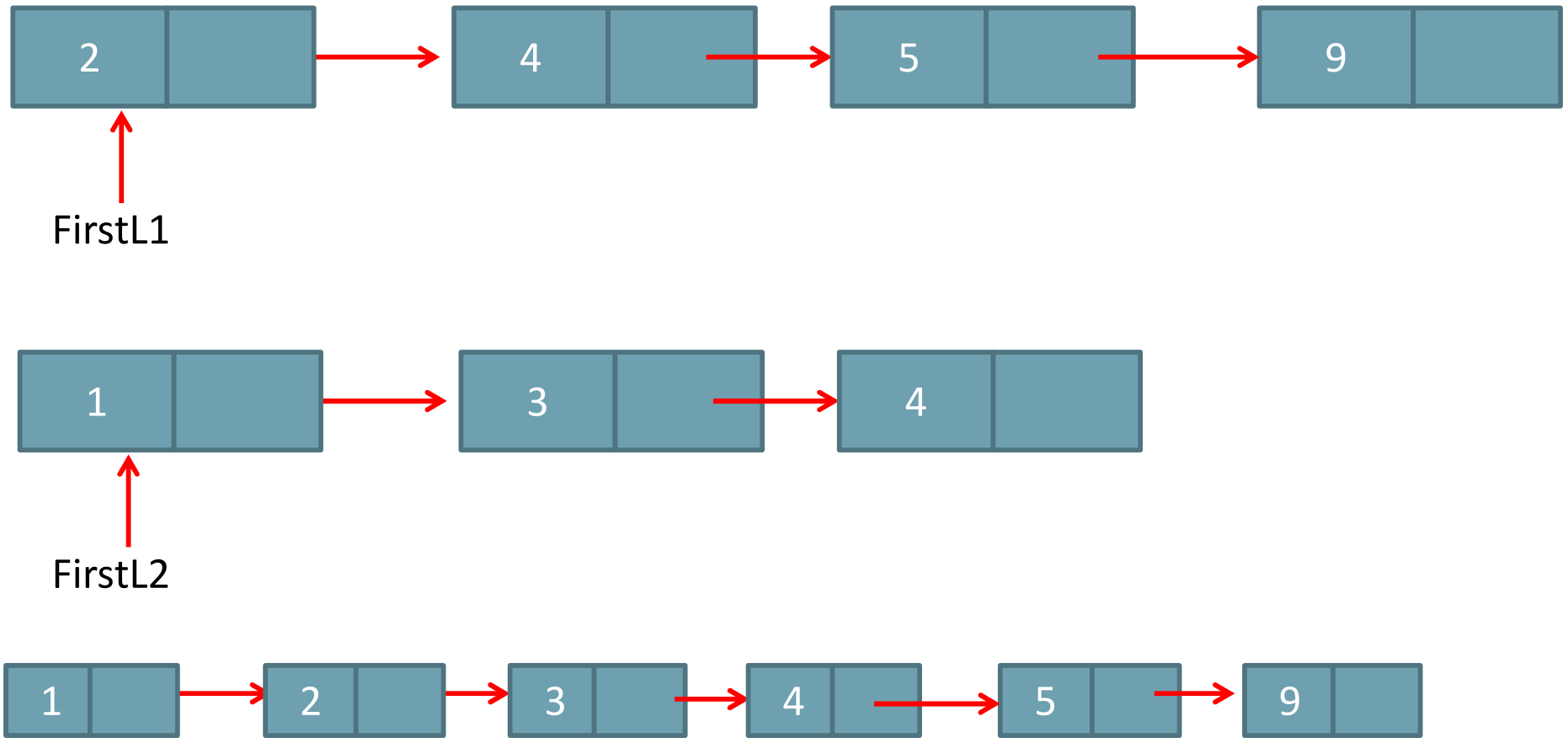
nextptr= nextptr->next

Step 4: change first pointer

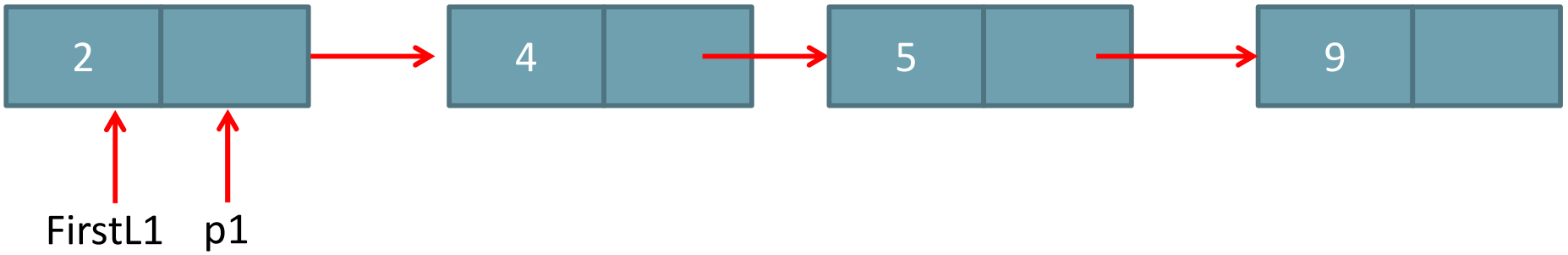
first->next=null

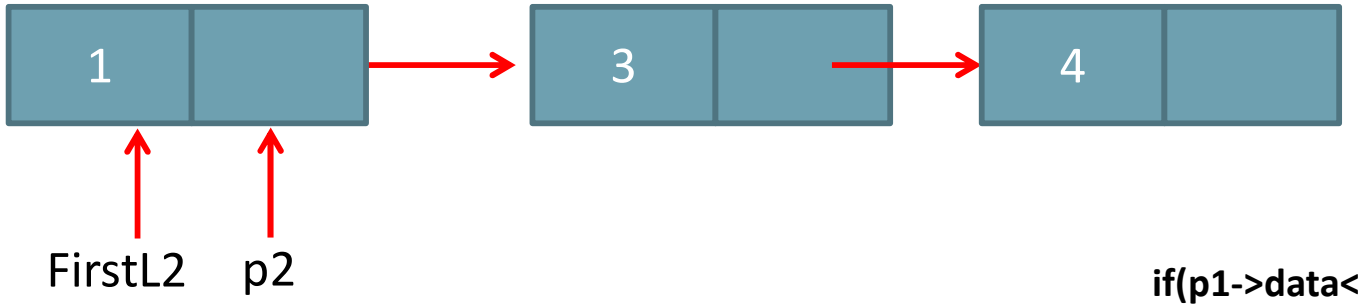
first = ptr

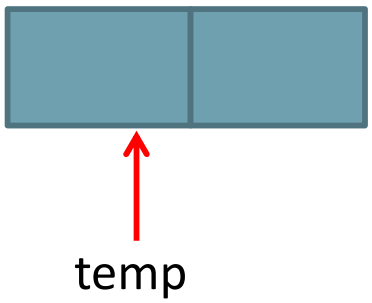
Merge two sorted Linked List



Merge two sorted Linked List

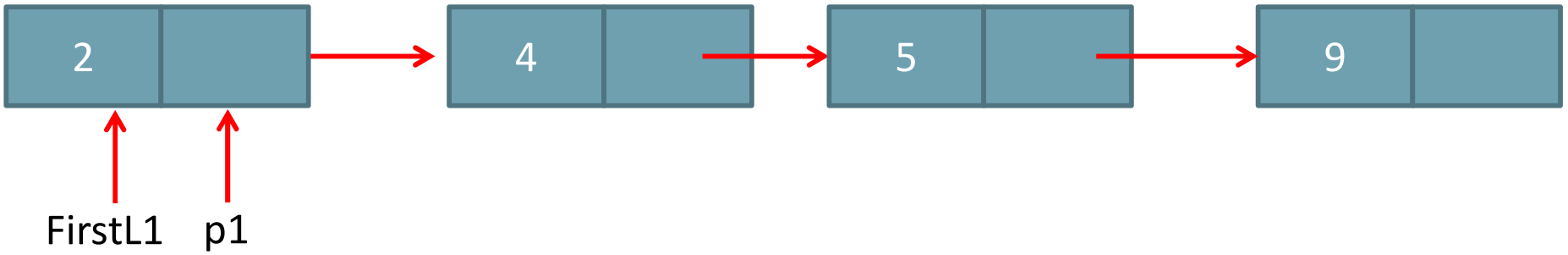


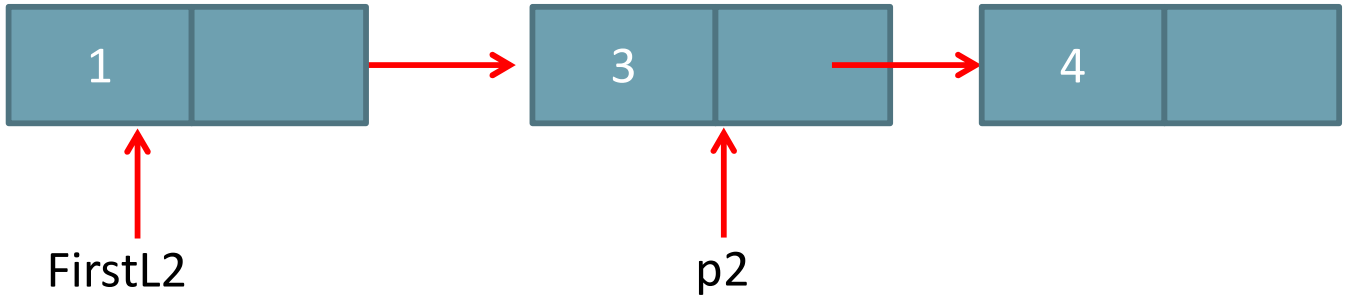


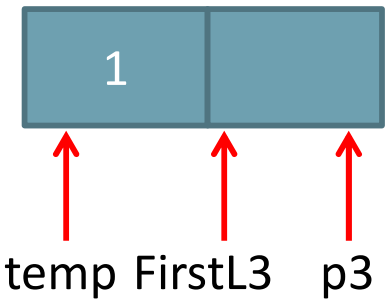



```
if(p1->data<p2->data){
        temp->data=p1->data;
        p1=p1->next; }
else if(p1->data>p2->data){
        temp->data=p2->data;
        p2=p2->next; }
else if(p1->data==p2->data){
        temp->data=p1->data;
        p1=p1->next;
        p2=p2->next;}
```

Merge two sorted Linked List

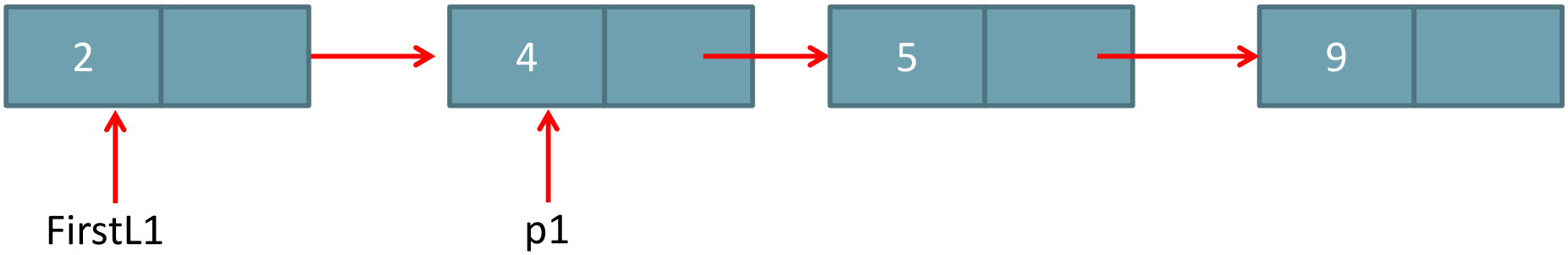


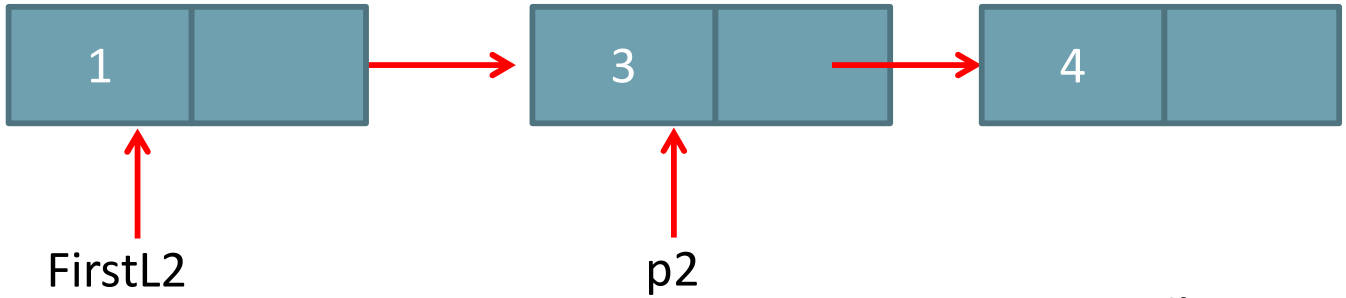


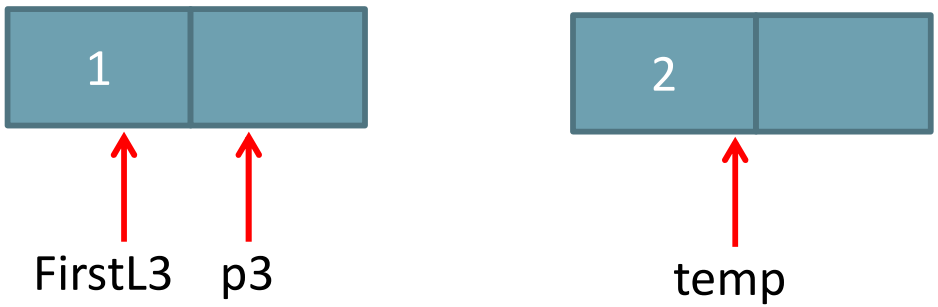



```
if(FirstL3==NULL){
        p3=temp;
        FirstL3=temp;}
else{
        p3-> next =temp;
        p3=temp;}
```

Merge two sorted Linked List

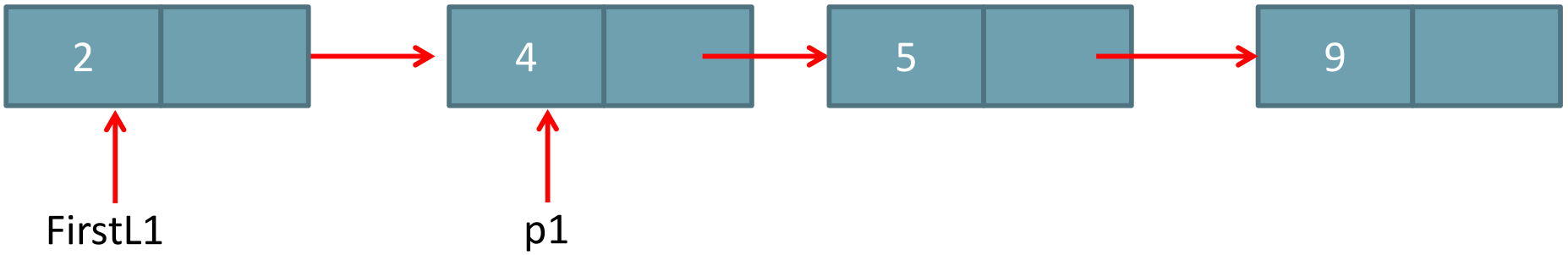


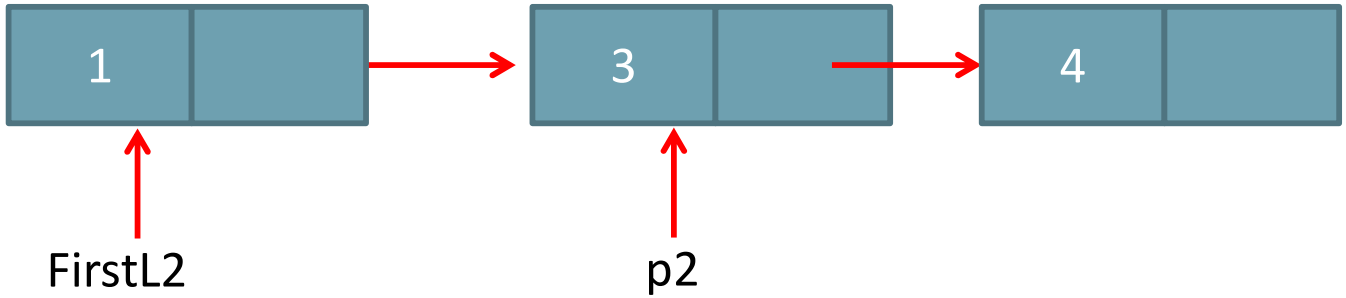


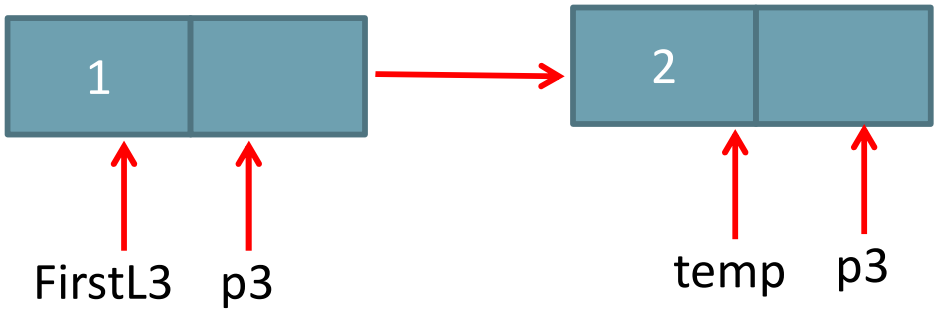



```
if(p1->data<p2->data){
                temp->data=p1->data;
                p1=p1-> next; }
else if(p1->data>p2->data){
                temp->data=p2->data;
                p2=p2-> next; }
else if(p1->data==p2->data){
                temp->data=p1->data;
                p1=p1-> next;
                p2=p2-> next;}
```

Merge two sorted Linked List



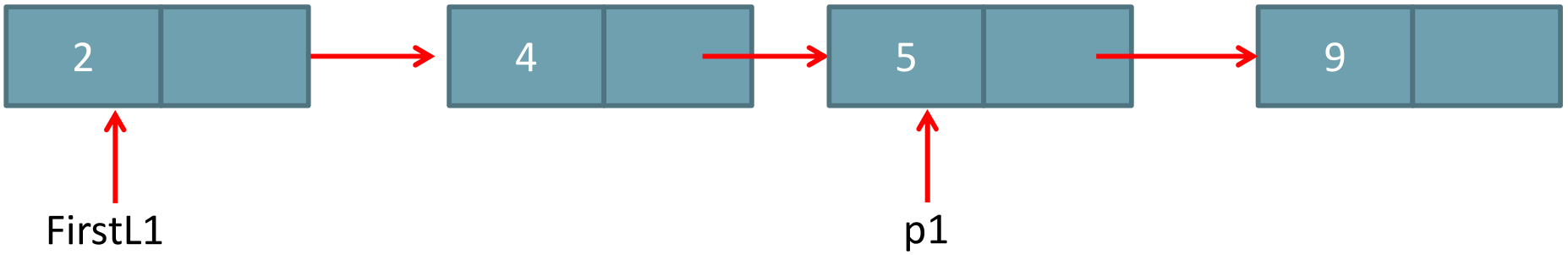


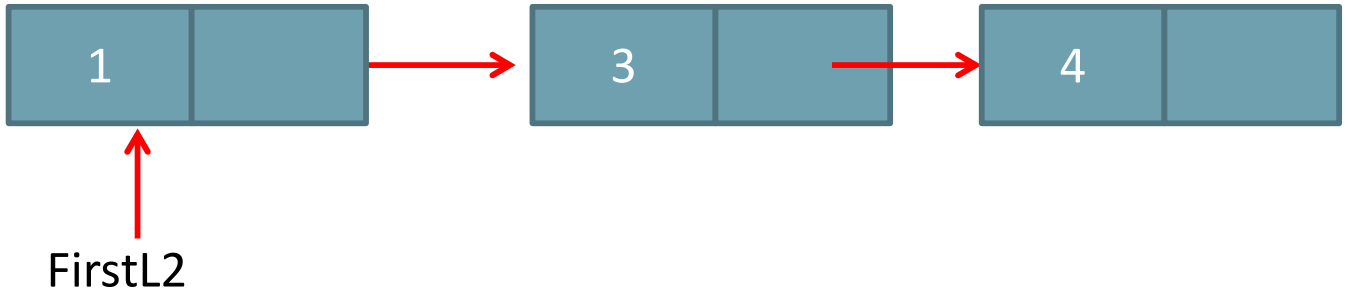


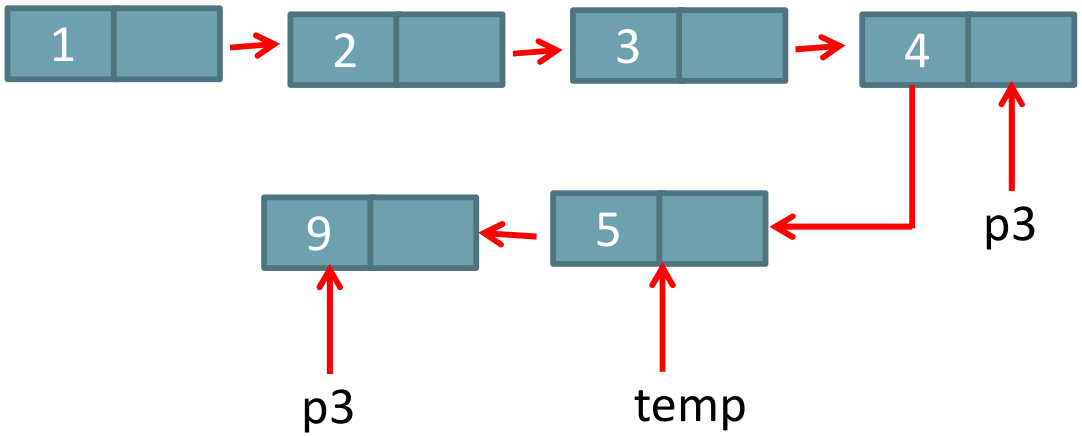

```
if(FirstL3==NULL){
        p3=temp;
        FirstL3=temp;}
else{
        p3-> next =temp;
        p3=temp;}
```


Kanak Kalyani

Merge two sorted Linked List








```
while(p1!=NULL){
  temp=(node *)
     malloc(sizeof(node*));
  temp-> next =NULL;
  temp->data=p1->data;
  p3-> next =temp;
  p3=temp;
  p1=p1-> next;
}
```

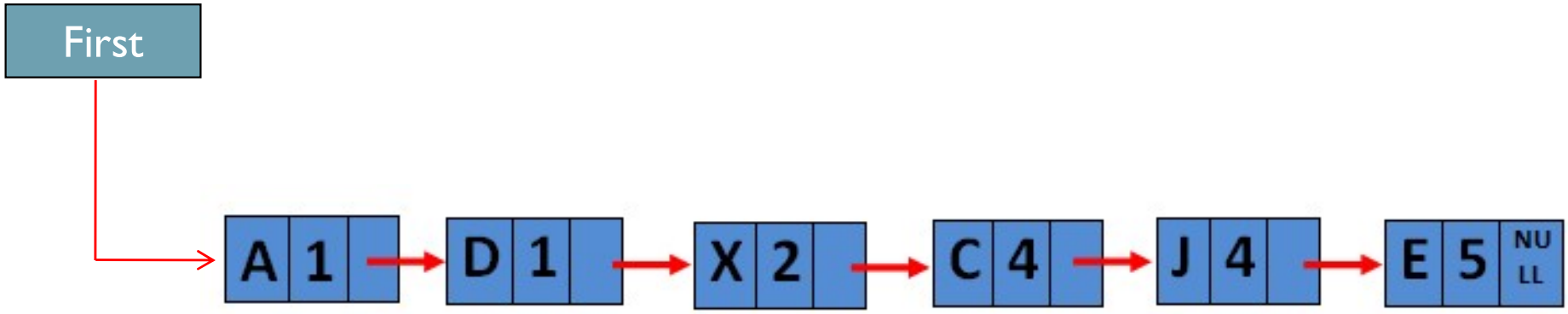
Merge two Linked List

```
node* merge(node* FirstL1,node* FirstL2)
{
   node *p3=NULL, *ptr=NULL, *temp;
   node* p1=FirstL1;
   node* p2-FirstL2;
   while(p1!=NULL && p2!=NULL){
   temp=(node *)malloc(sizeof(node*));
   temp-> next =NULL;
     if(p1->data<p2->data){
              temp->data=p1->data;
              p1=p1-> next;
     else if(p1->data>p2->data){
              temp->data=p2->data;
              p2=p2-> next;}
     else if(p1->data==p2->data){
              temp->data=p1->data;
              p1=p1-> next;
              p2=p2-> next;}
     if(p3==NULL){
              p3=temp;
              FirstL3=temp;}
     else{
              p3-> next =temp;
              p3=temp;
              }}
while(p1!=NULL){
temp=(node *)malloc(sizeof(node*));
temp-> next =NULL;
temp->data=p1->data;
p3-> next =temp;
p3=temp;
p1=p1-> next;
}
while(p2!=NULL)        {
temp=(node *)malloc(sizeof(node*));
temp-> next =NULL;
temp->data=p2->data;
p3-> next =temp;
p3=temp;
p2=p2-> next;
}
   return p3;
}
```

Applications of Linked List

- Stack using Linked List
- Linear Queue using Linked List
- Representing Polynomial using Linked List
- Priority Queue using LL
- Storing records of student in LL

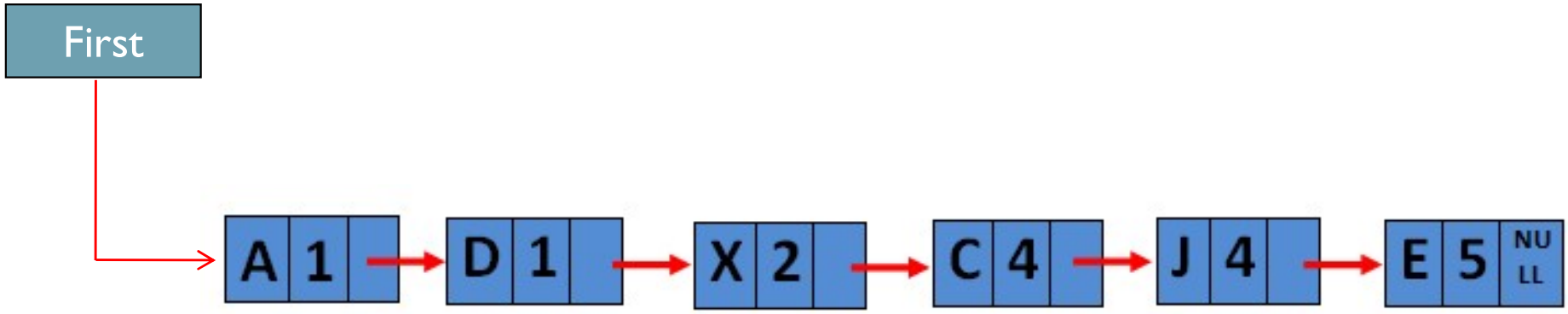
LINKED LIST REPRESENTATION OF A PRIORITY QUEUE




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

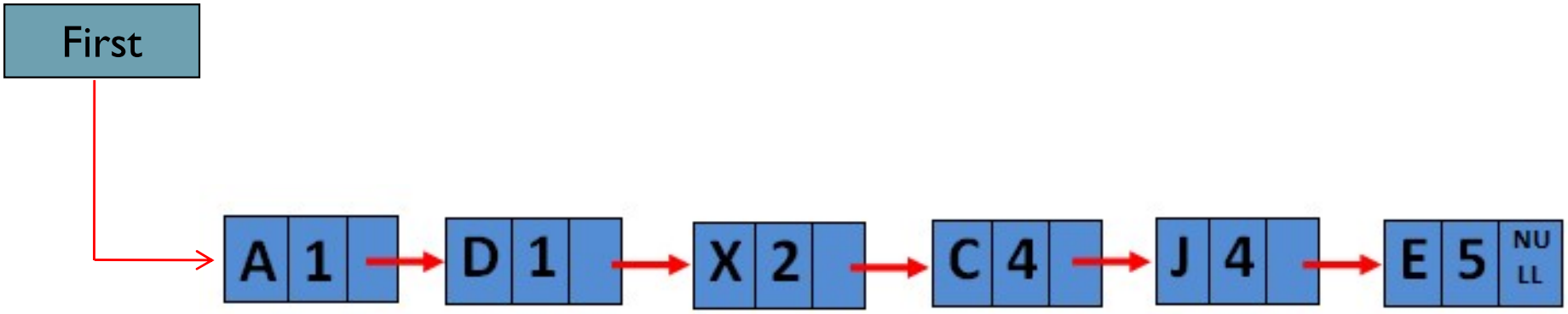
VARIANTS OF PRIORITY QUEUE

- Sorted List



Variant of Priority Queue

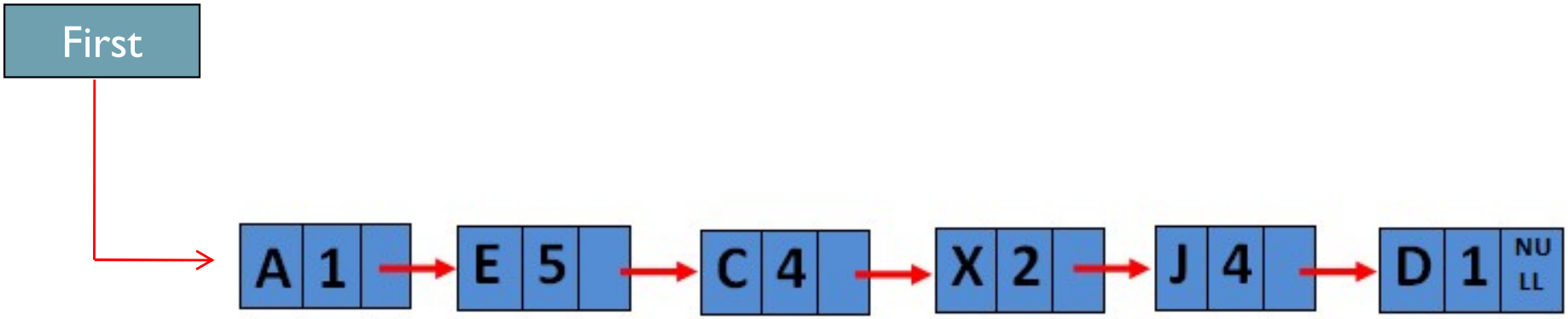
- Sorted List



Complexity of Insertion – O(n)
Complexity of Deletion – O(1)

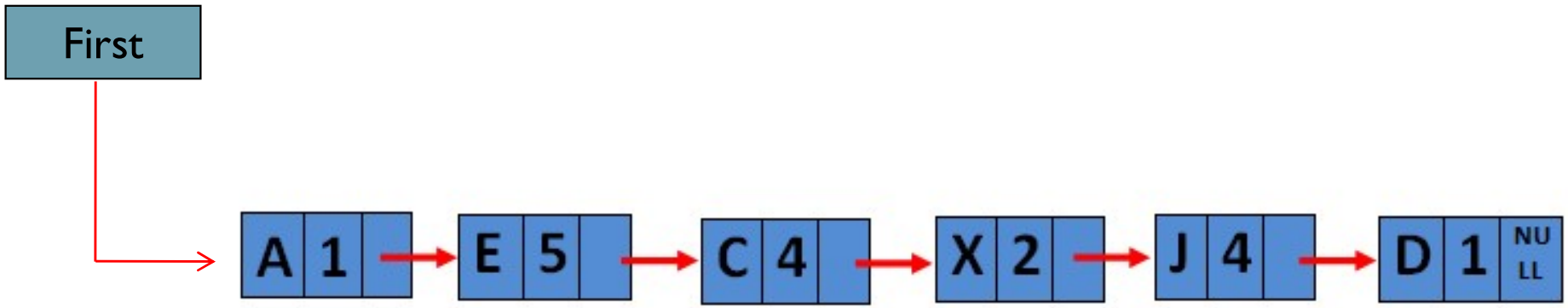
Variants of Priority Queue

- Unsorted List



Variant of Priority Queue

- Unsorted List



Complexity of Insertion – O(1)
Complexity of Deletion – O(n)

PROBLEM DEFINITION

- Suppose a priority queue is implemented as a sorted list. Implement a program to use this queue to schedule processes according to their priority (Priority Scheduling Algorithm).
- In Priority scheduling each process is assigned a priority. Processes with same priority are executed on first come first served basis.
- Consider following set of processes and generate the sequence in which processes get executed.

Process	CPU Burst Time	Priority
P1	9	2
P2	3	5
P3	5	4
P4	2	3
P5	4	4
P6	2	1
P7	8	2

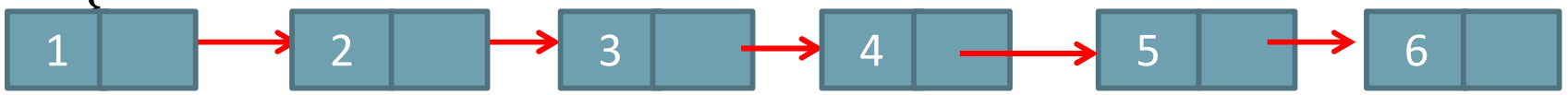
p6->p1->p7->p4->p3->p5->p2

Practice Problems

```
void fun2(struct Node *LL)
{
        if(LL == NULL)
                return;

        if(LL->next != NULL )
                fun2(LL->next);
        printf("%d", LL->data );

}
```

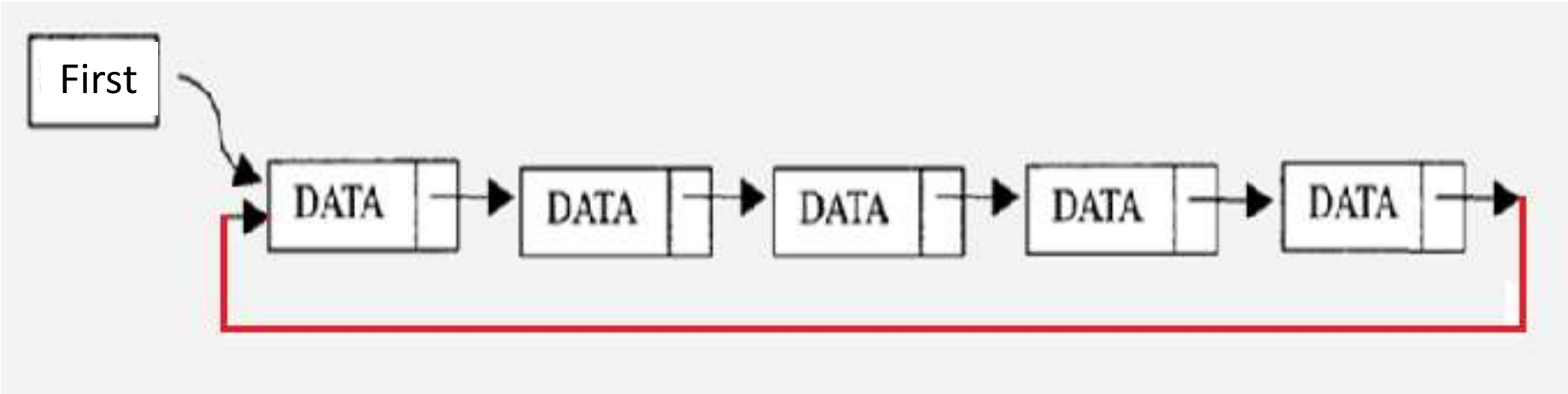


Circular Linked List

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



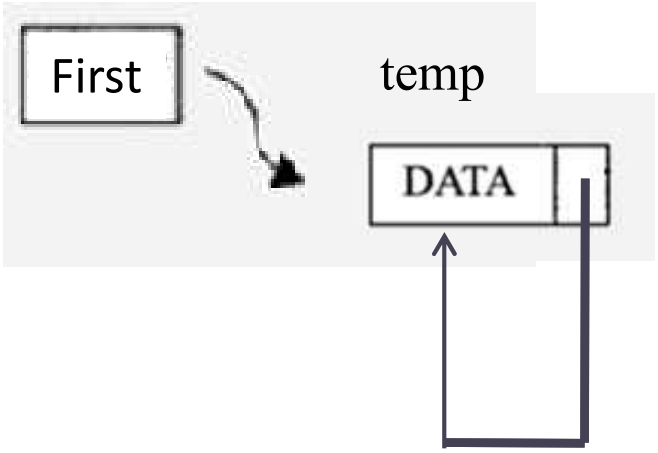
Circular Single Linked list

Node Declaration:

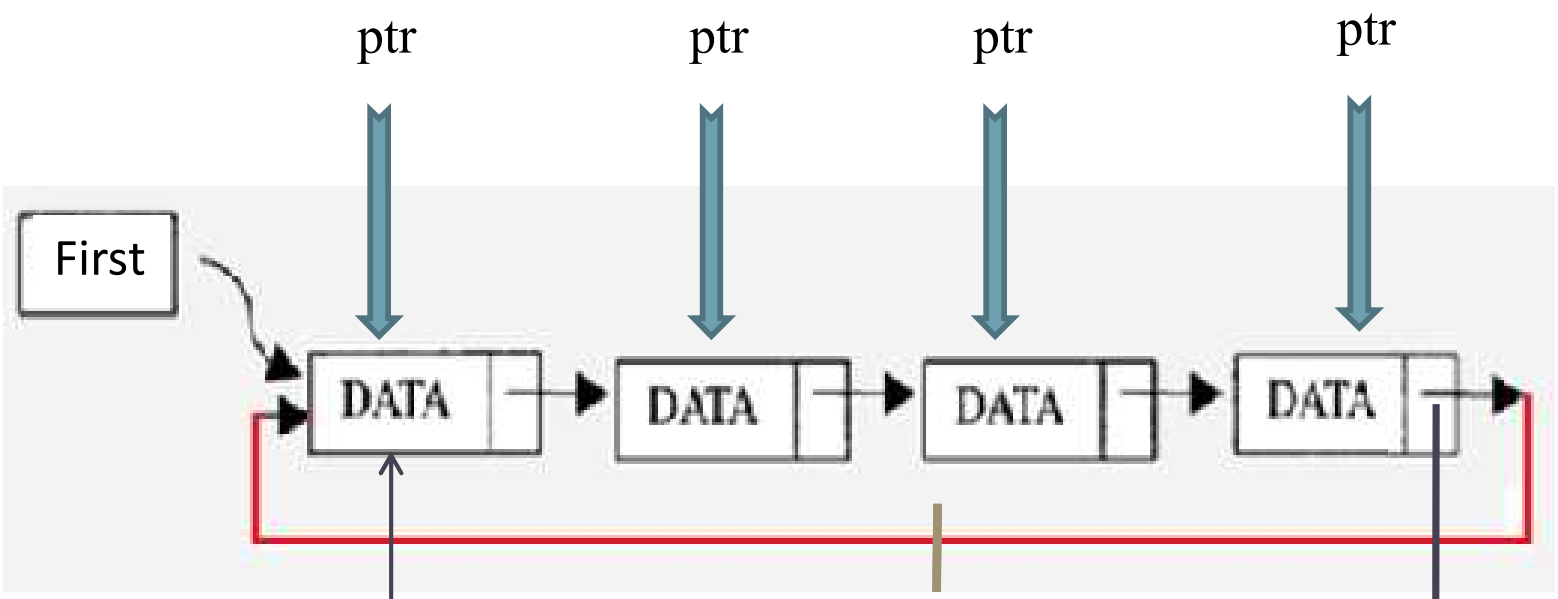
```
typedef struct Node
{
int data;
struct Node *next;
} node;
```

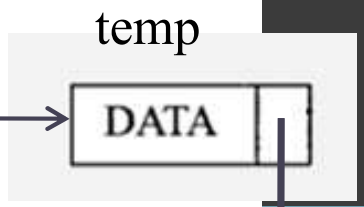
```
node *first=NULL;
```

Circular Linked list creation



Circular Linked list creation






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

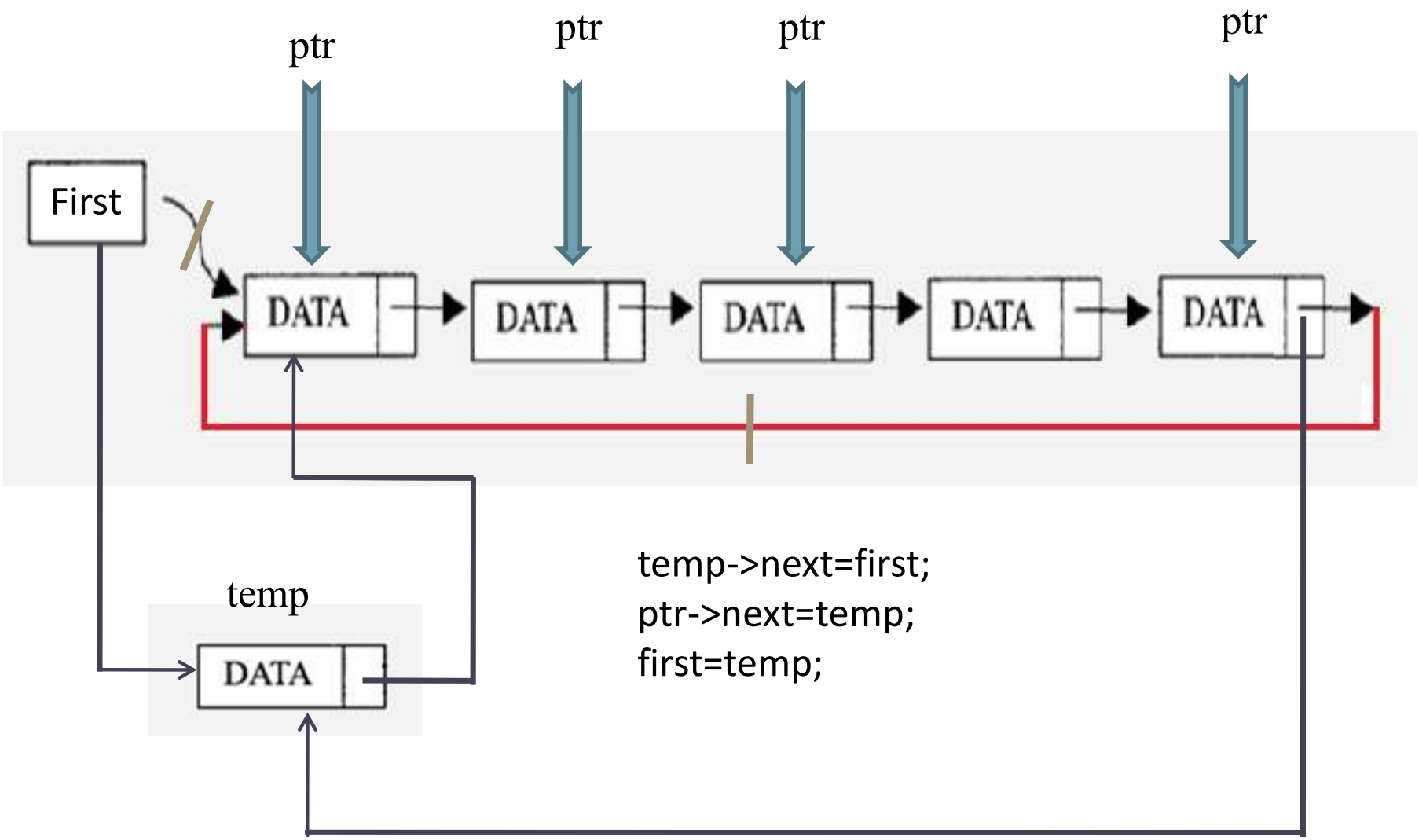
Circular Linked list creation

```
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

Kanak Kalyani

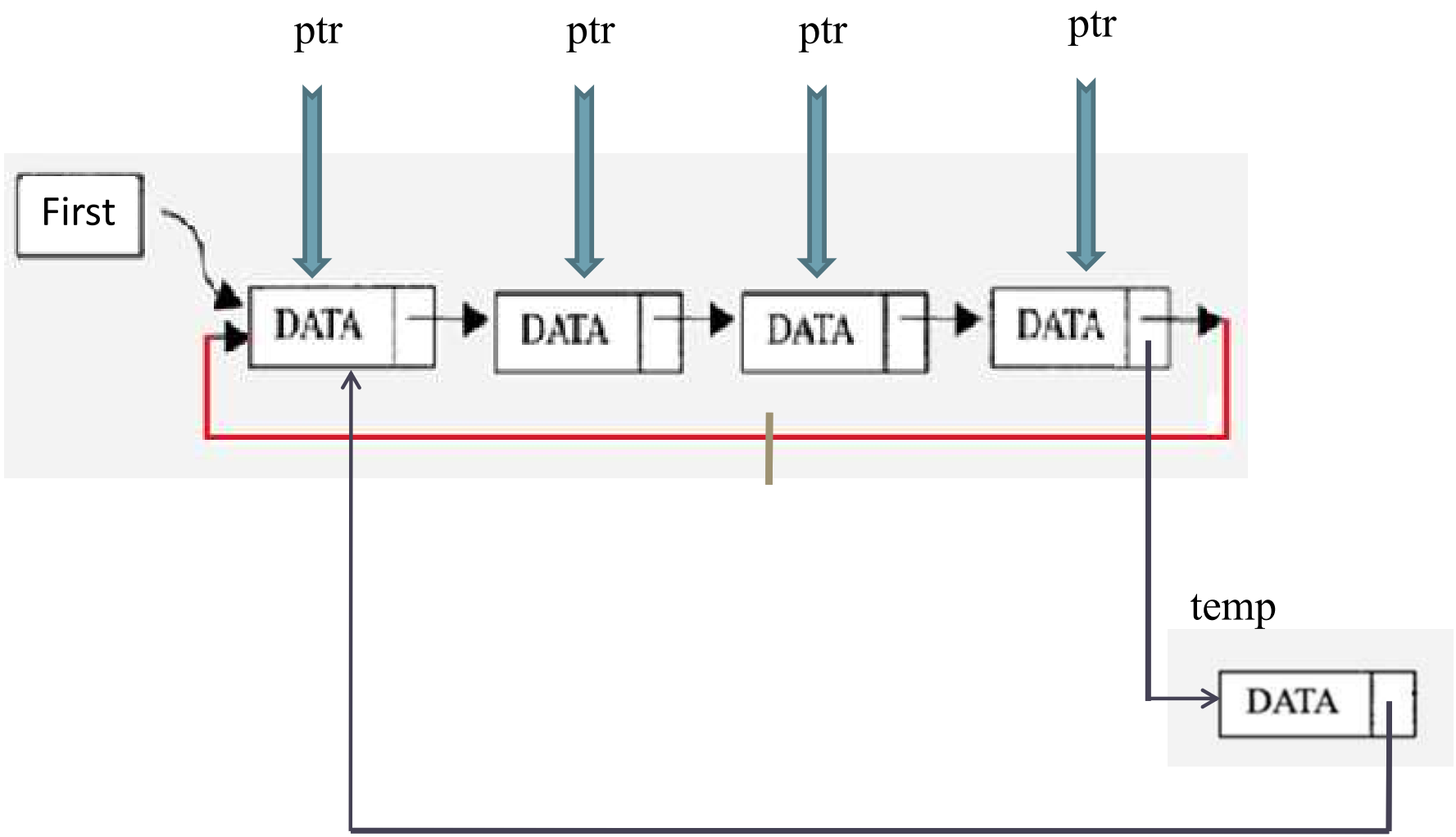
Insertion As First Node



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

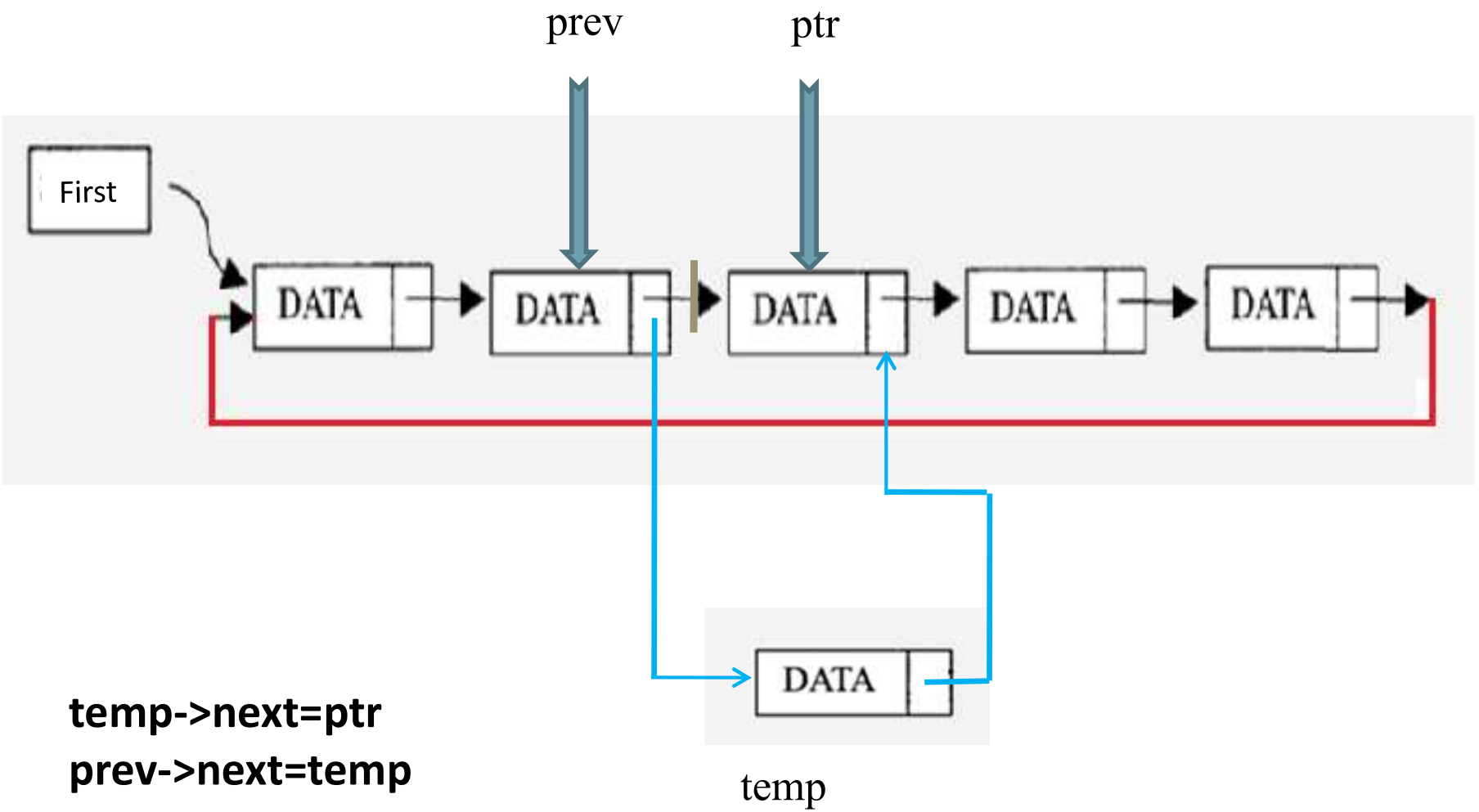


ptr->next=temp;
Temp->next=first

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

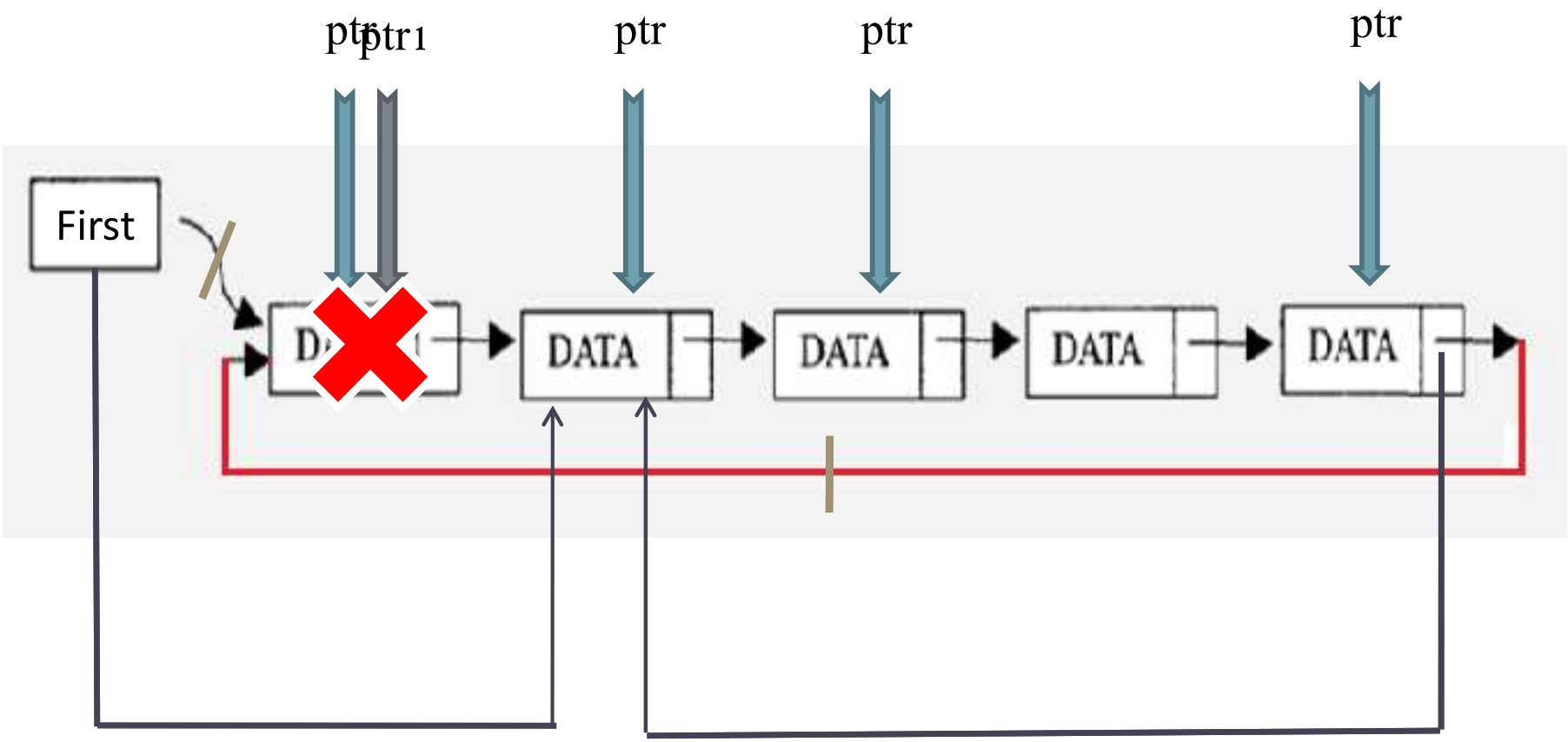
Insertion At The Specified Position

```
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

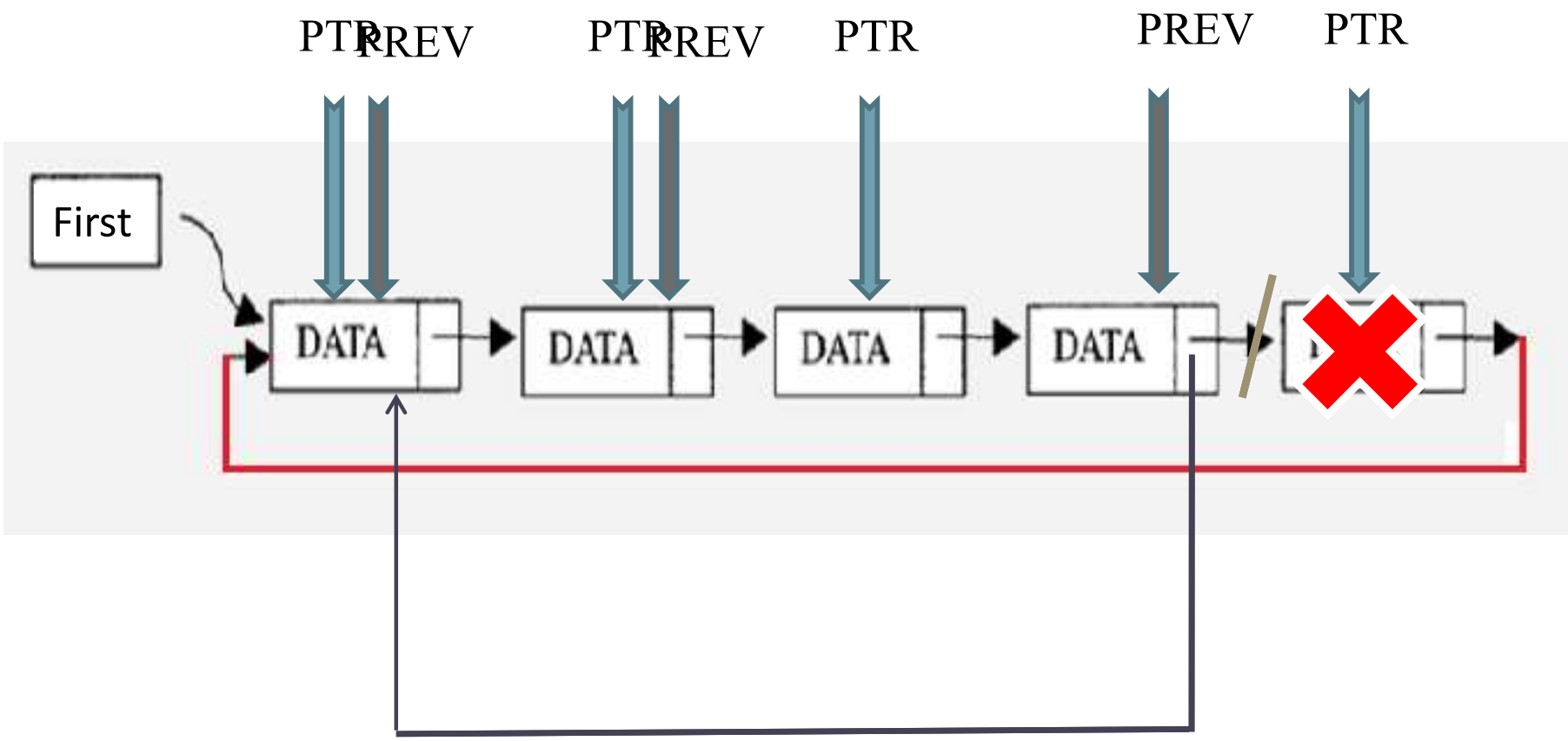



```
ptr1=first;
first=first->next
ptr->next=first
free(ptr1);
```

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

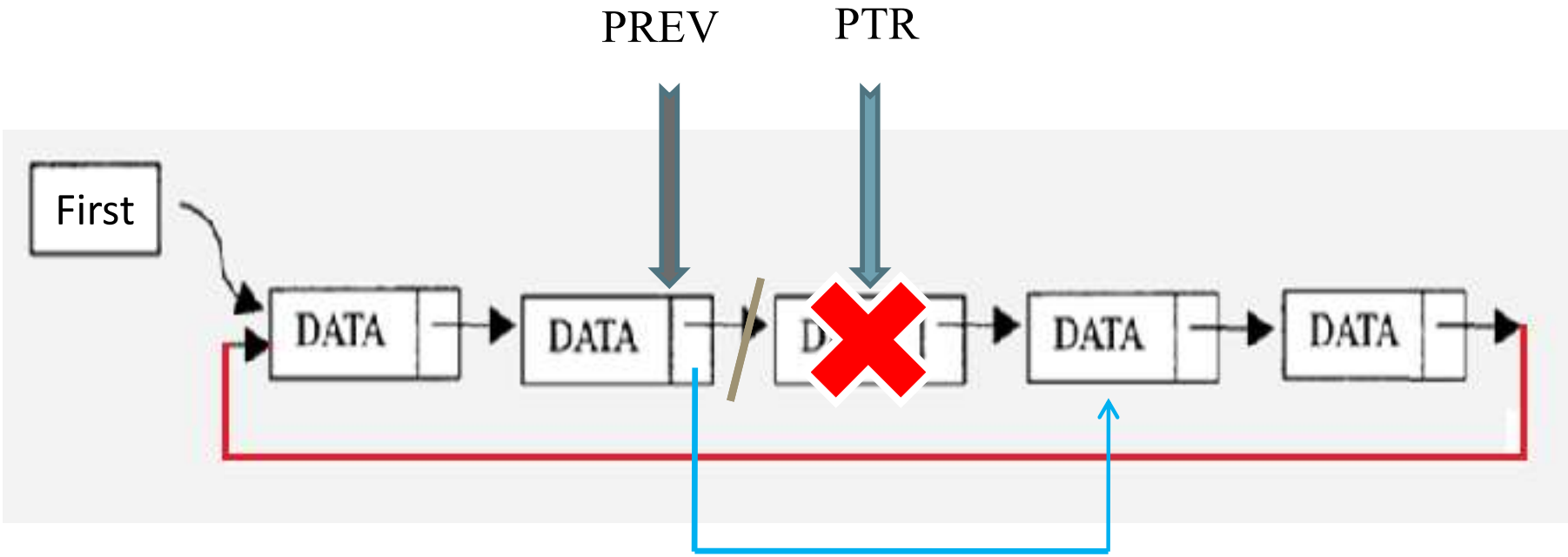


prev->next=first
Free(ptr)

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.

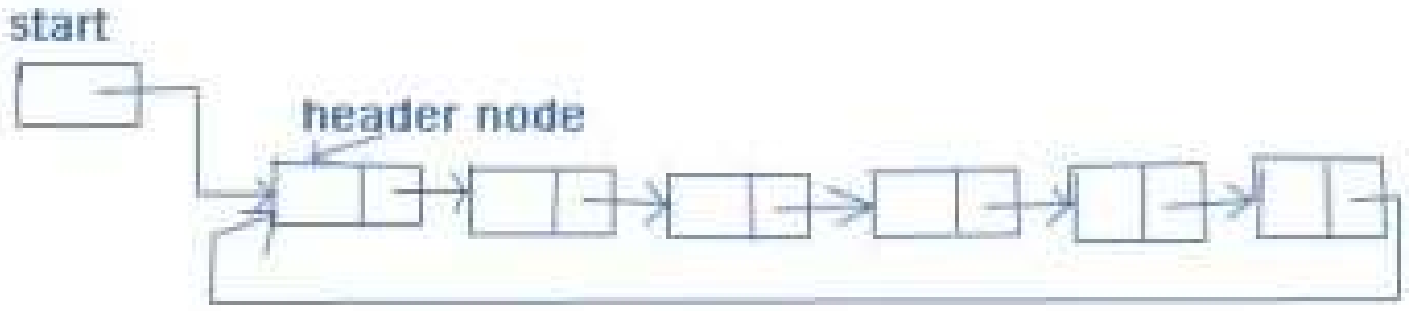
- Circular lists are useful in applications to repeatedly go around the list. For example Round Robin (RR) job scheduling by operating system

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



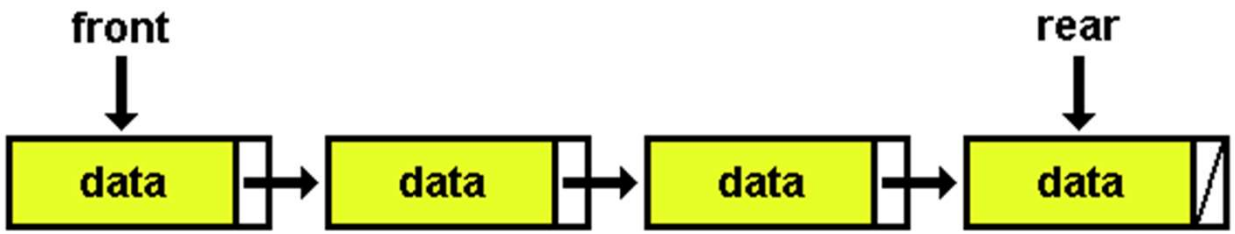
(b) Circular header list.

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

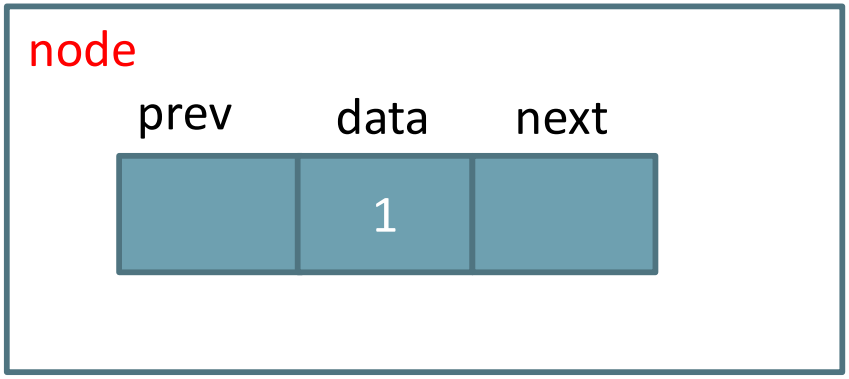
```
void fun1(struct Node* head)
{
if(head == NULL)
        return;

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

Double Linked List

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;
};
```

Advantages Of DLL

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.
- As with a singly linked list, it is the easiest data structure to implement.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only required the pointer which is to be deleted.

Disadvantages Of DLL

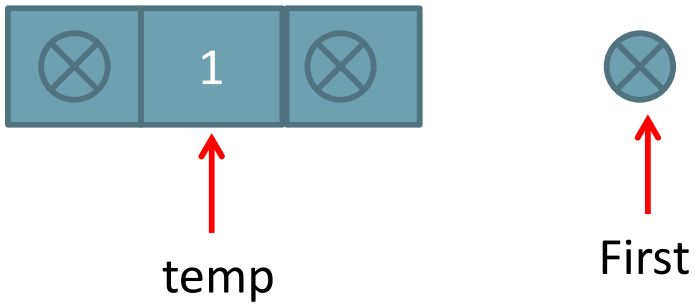
- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.

Applications of Double Linked List

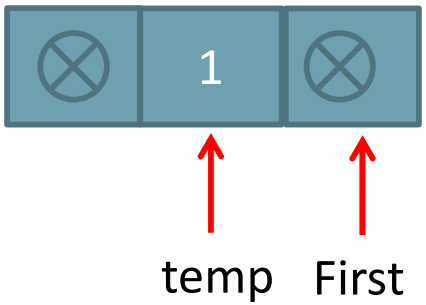
- Used in the navigation systems where front and back navigation is required.
- Used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.
- Used to represent a classic game deck of cards.
- It is also used by various applications to implement undo and redo functionality.
- Used in constructing **MRU**/**LRU** (Most/least recently used) cache.
- Other data structures like stacks, Hash Tables, Binary trees can also be constructed or programmed using a doubly-linked list.
- Also in many operating systems, the **thread scheduler**(the thing that chooses what process needs to run at which time) maintains a doubly-linked list of all processes running at that time.

Double Linked List: Creation

STEP1: Create temp node with data value



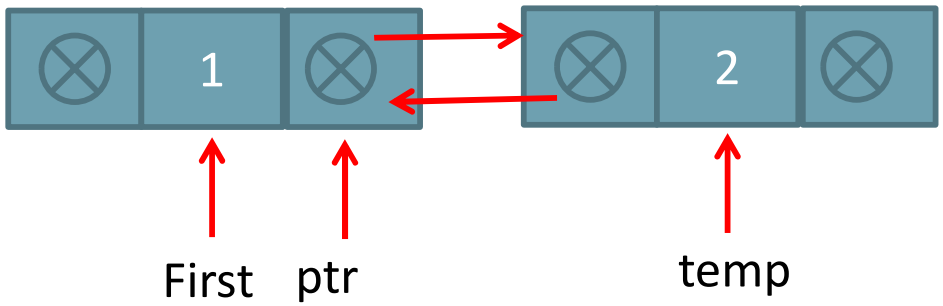
STEP2: if first is NULL then Assign temp to first




```
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

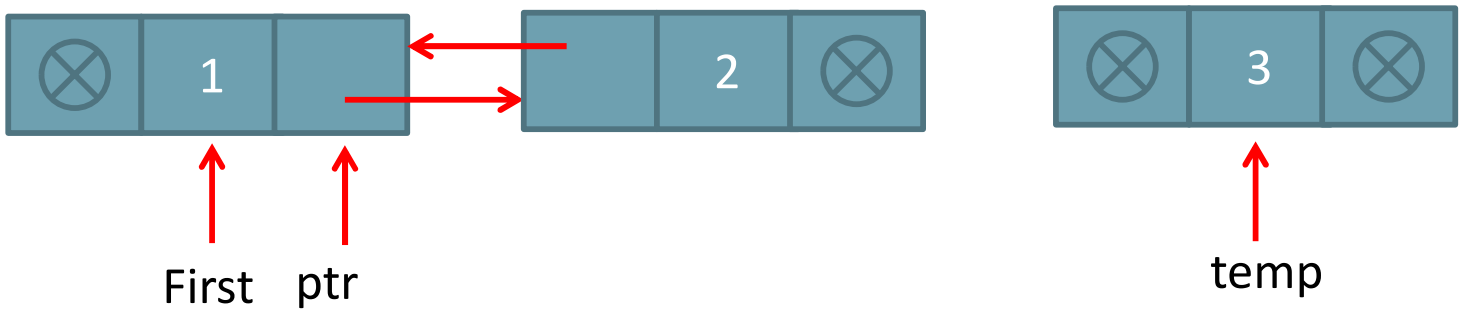
CREATE SECOND NODE




```
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

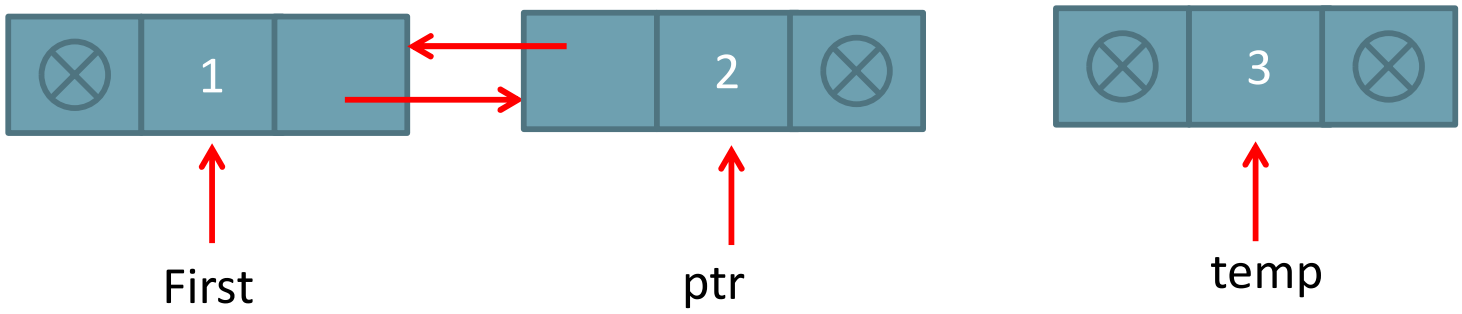
CREATE THIRD NODE




```
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

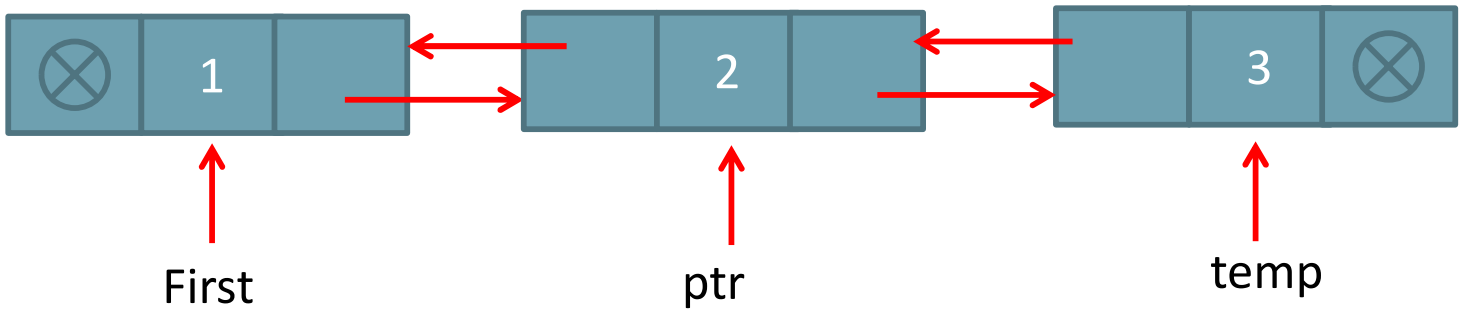
CREATE THIRD NODE




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

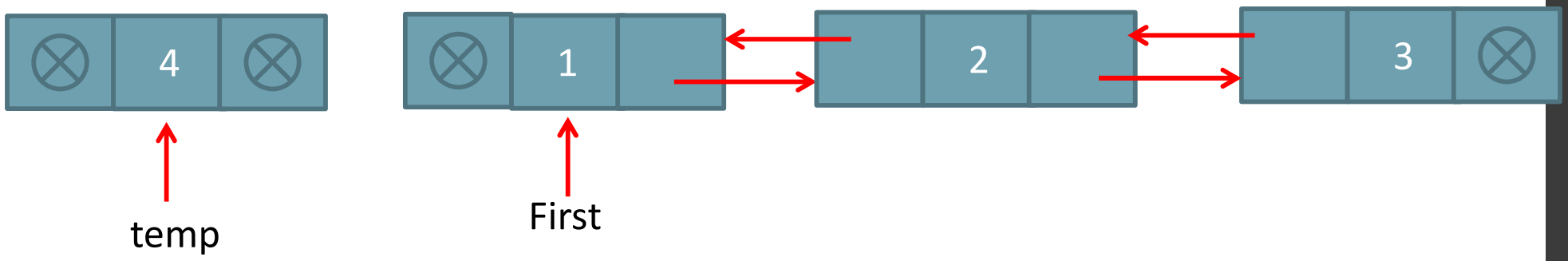
Double Linked List: Creation

CREATE THIRD NODE



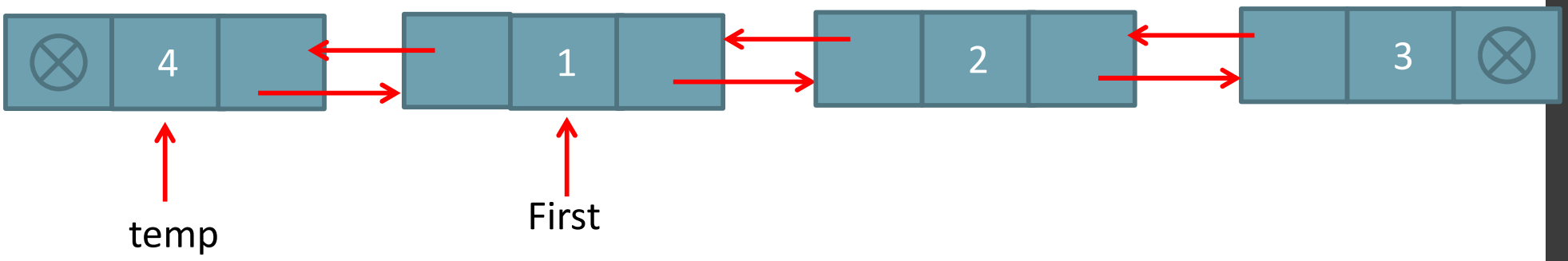

```
if(first==NULL)
        first=temp;
else{
        ptr=first;
        while(ptr->next!=NULL)
                ptr=ptr->next;
        ptr->next=temp;
        temp->prev=ptr;
}
```

Double Linked List
Insertion: At First Position



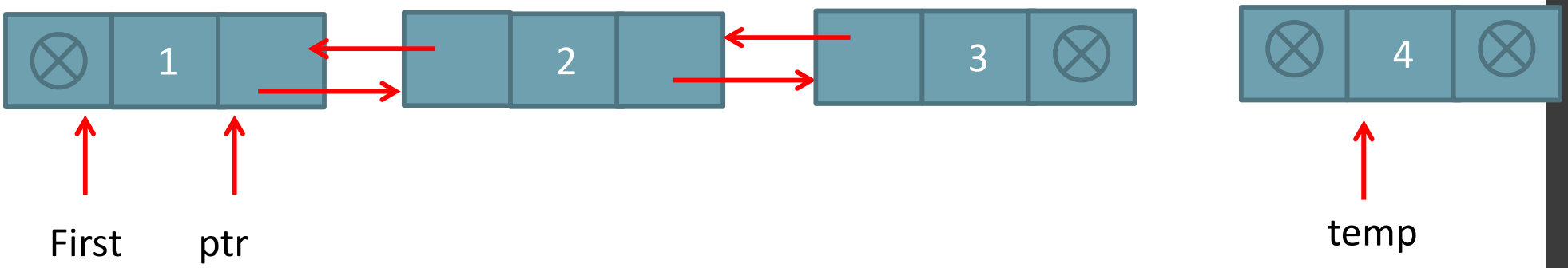

```
temp->next=first;
first->prev=temp;
first=first->prev;
```

Double Linked List
Insertion: At First Position



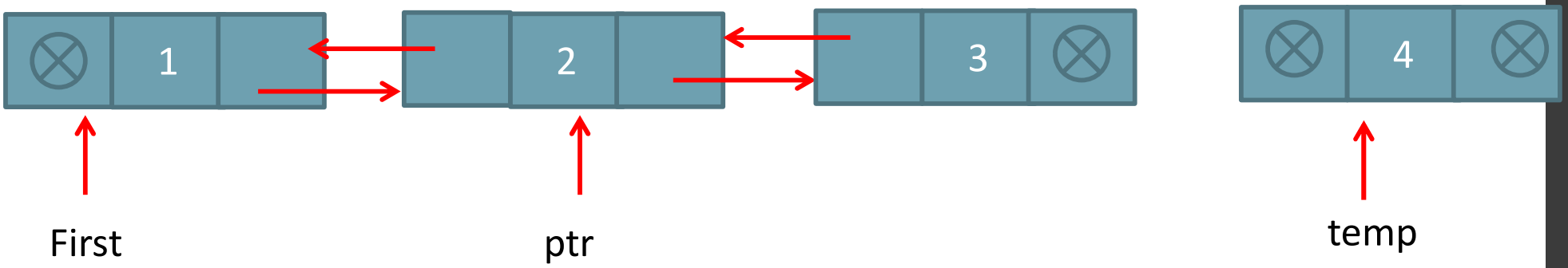

```
temp->next=first;
first->prev=temp;
first=first->prev;
```

Double Linked List
Insertion: At Last Position



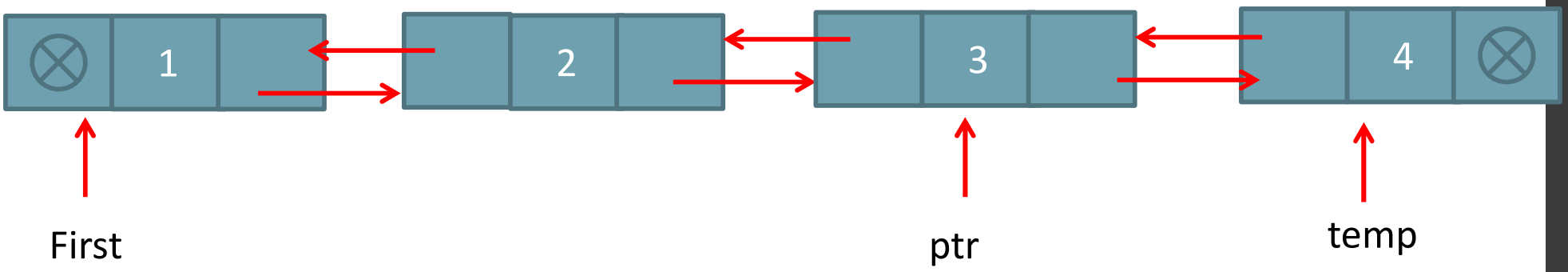

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
temp->prev=ptr;
```

Double Linked List
Insertion: At Last Position



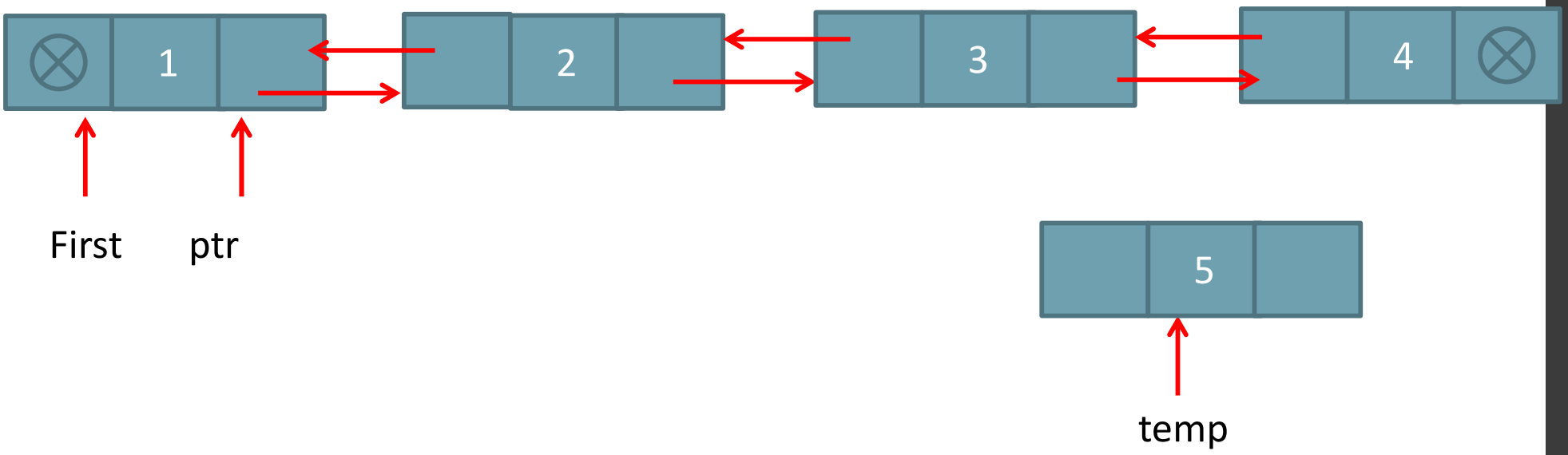

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
temp->prev=ptr;
```

Double Linked List
Insertion: At Last Position



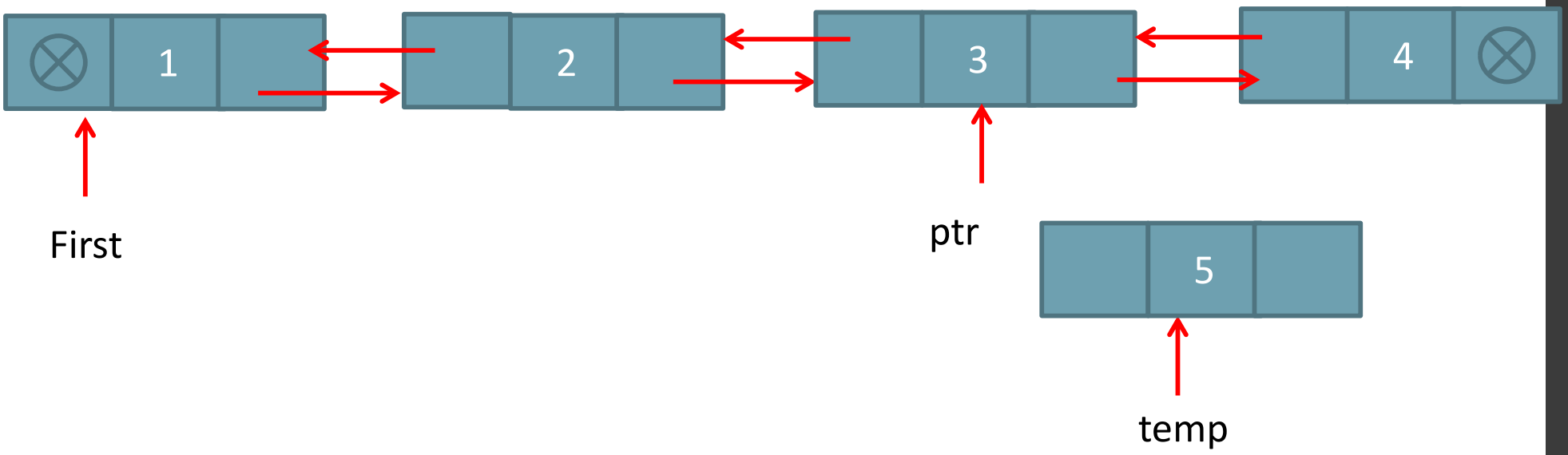

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
temp->prev=ptr;
```

Double Linked List
Insertion: In Between



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

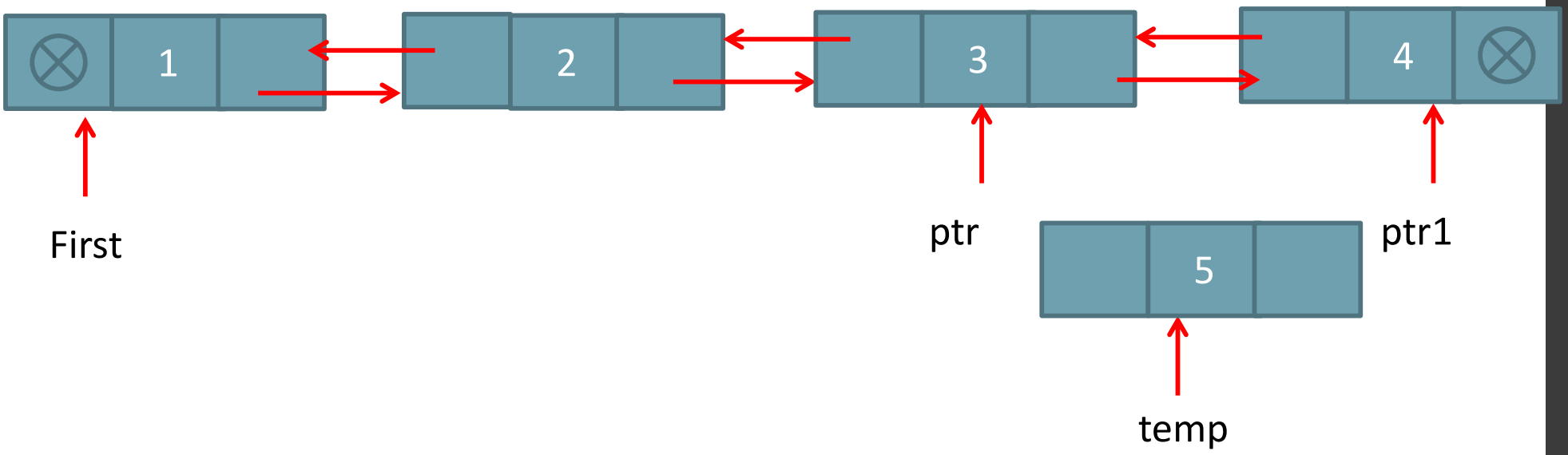
Double Linked List
Insertion: In Between



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

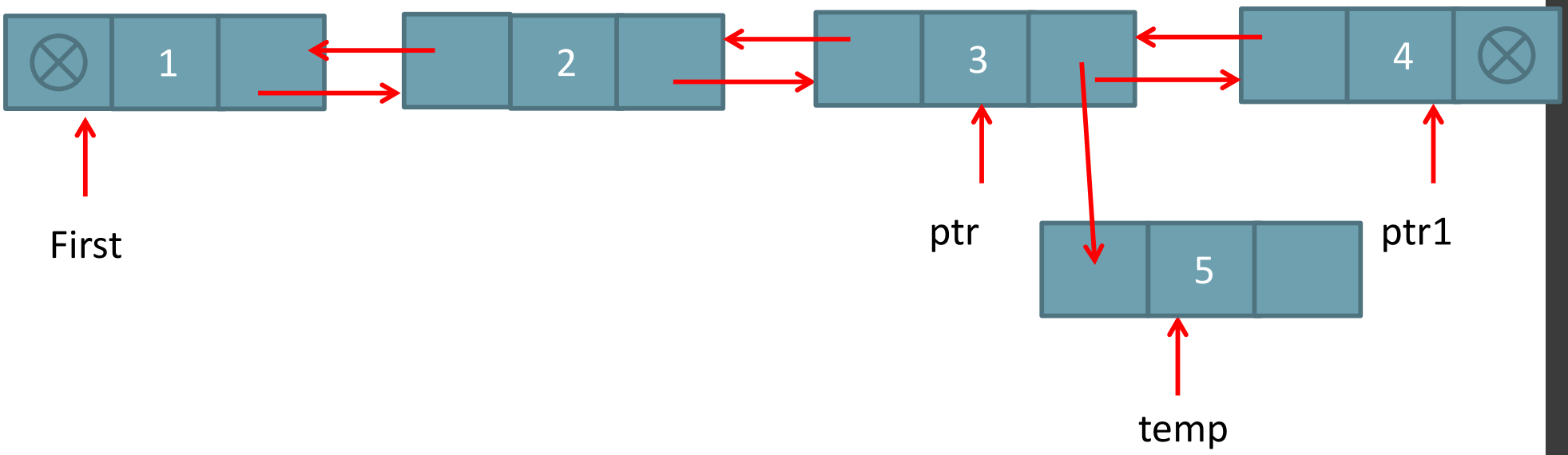


Step1: Enter a position and move ptr pointer reach to position - 1.

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

Step3: ptr1=ptr->next

Double Linked List
Insertion: In Between



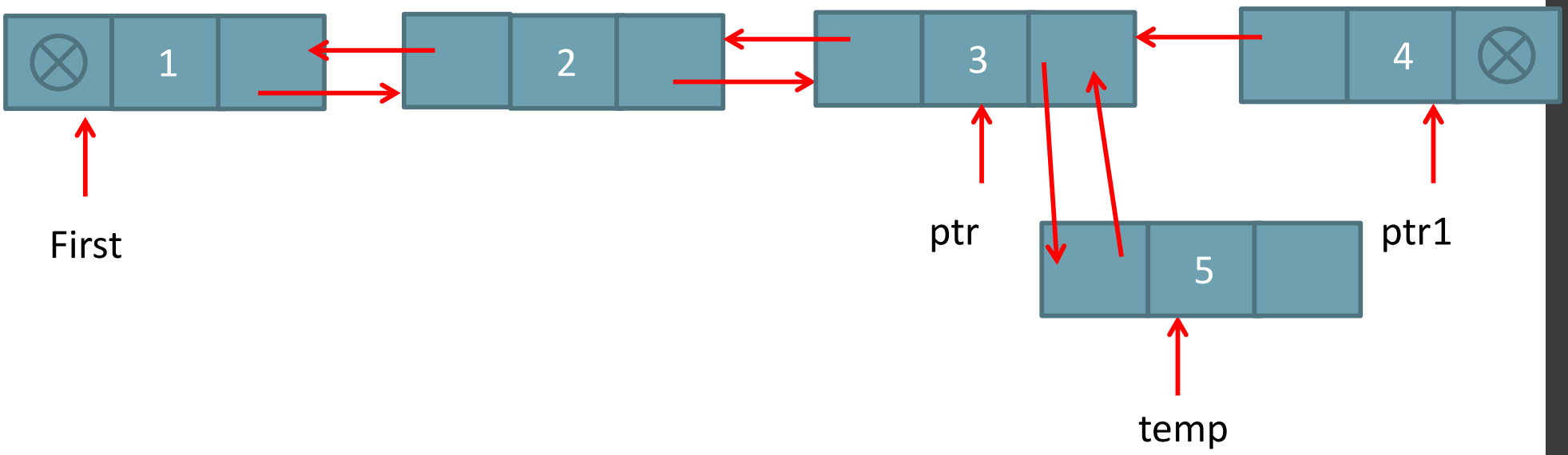
Step1: Enter a position and move ptr pointer reach to position - 1.

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.

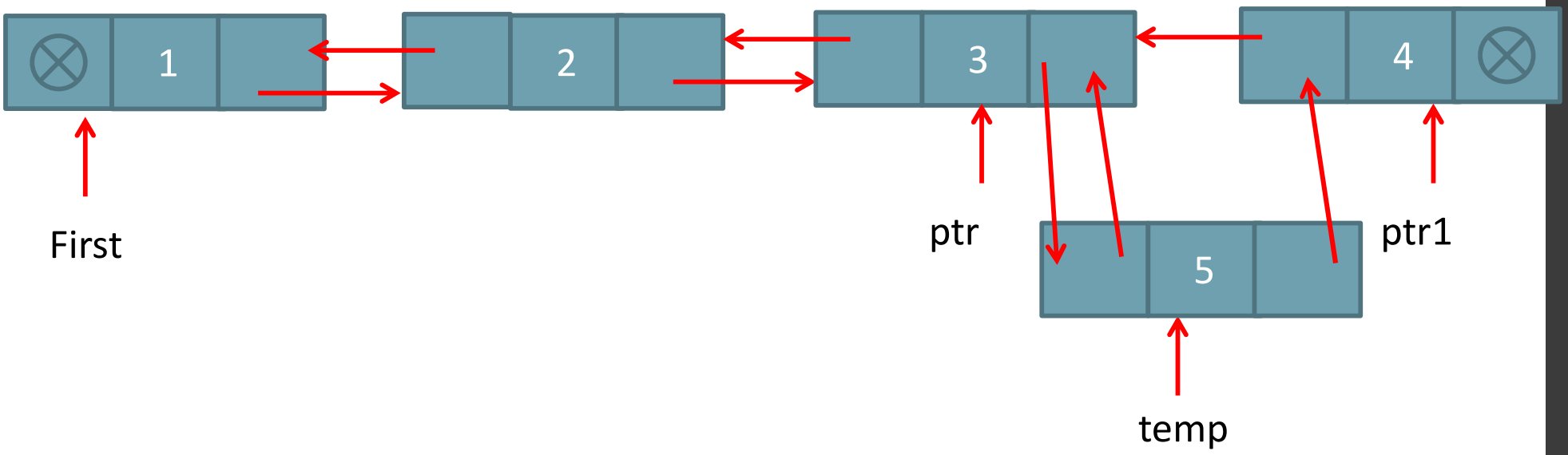
Step2: check for the correctness of ptr, if correct follow the steps below:

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.

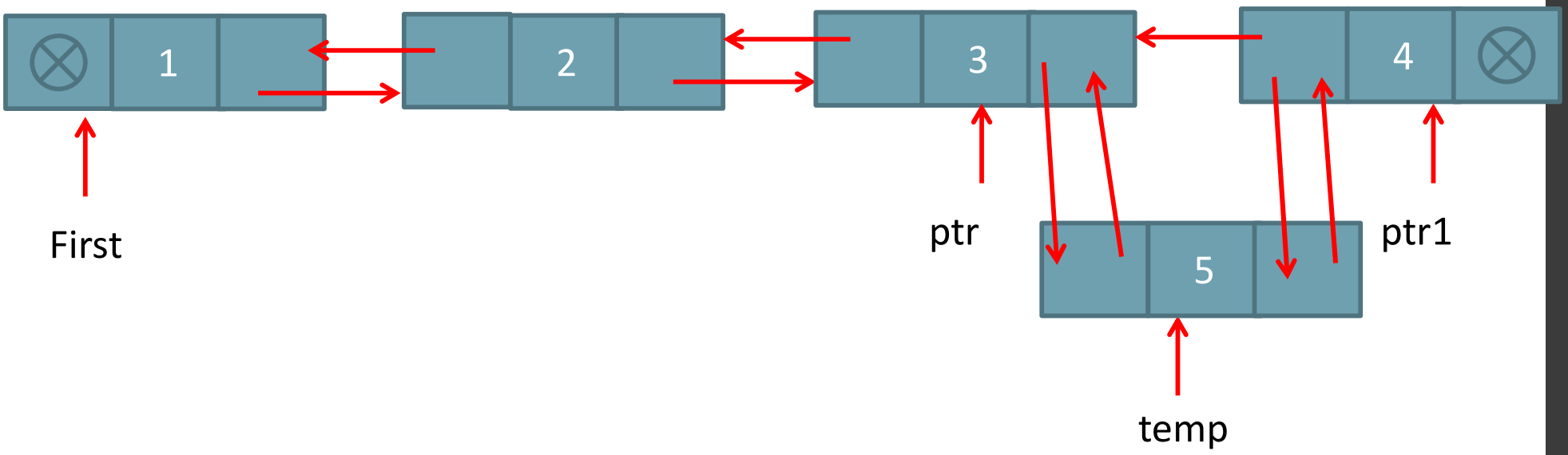
Step2: check for the correctness of ptr, if correct follow the steps below:

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

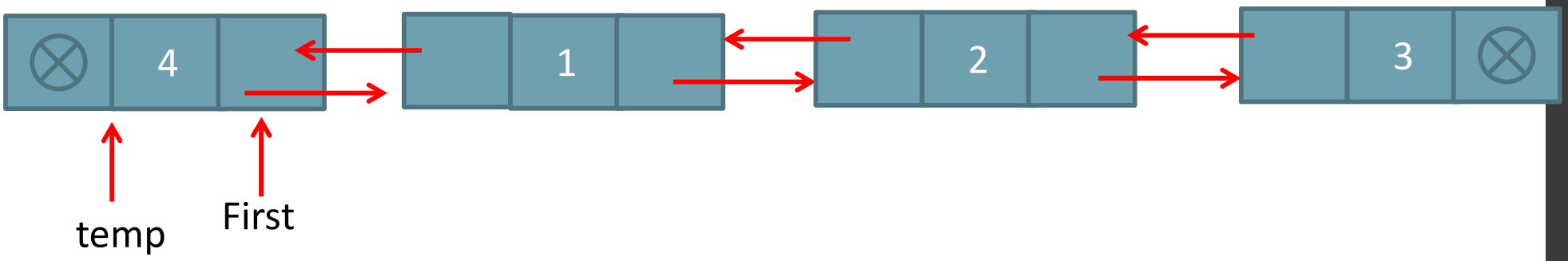
i) ptr->next=temp

ii) temp->prev=ptr

iii) ptr1->prev=temp

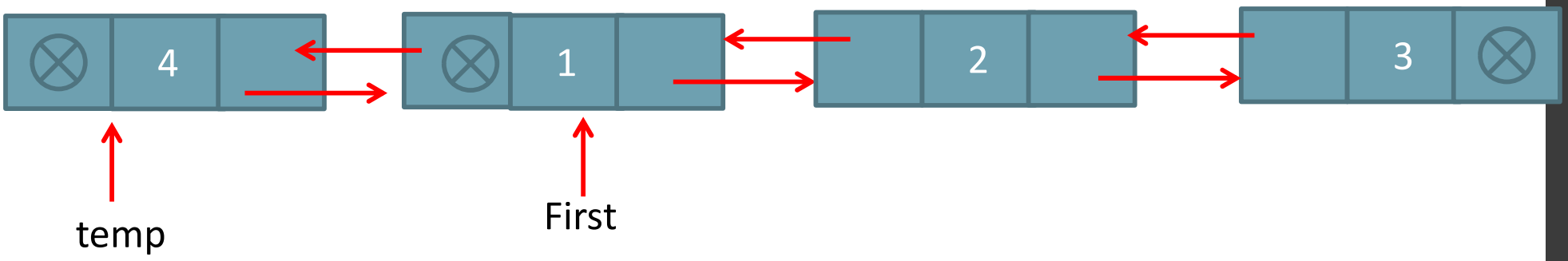
iv) temp->next=ptr1

Double Linked List
Delete node at First Position



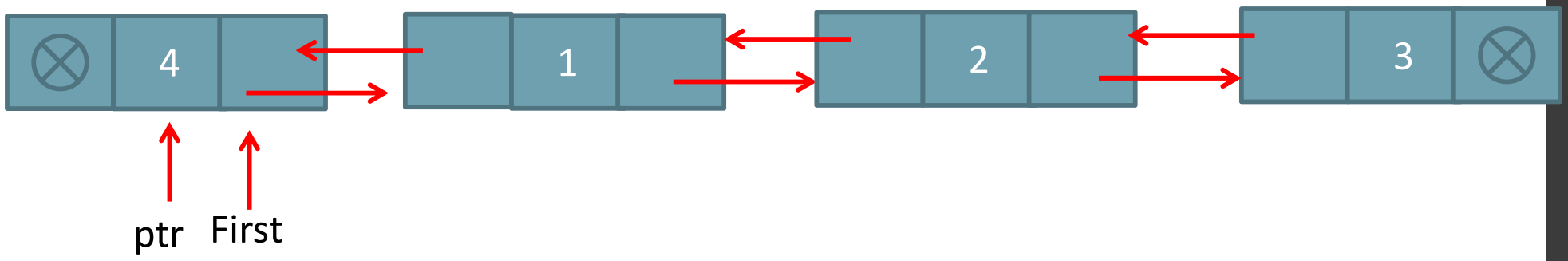

```
temp=first;
first=first->next;
free(temp);
first->prev=NULL;
```

Double Linked List
Delete node at First Position



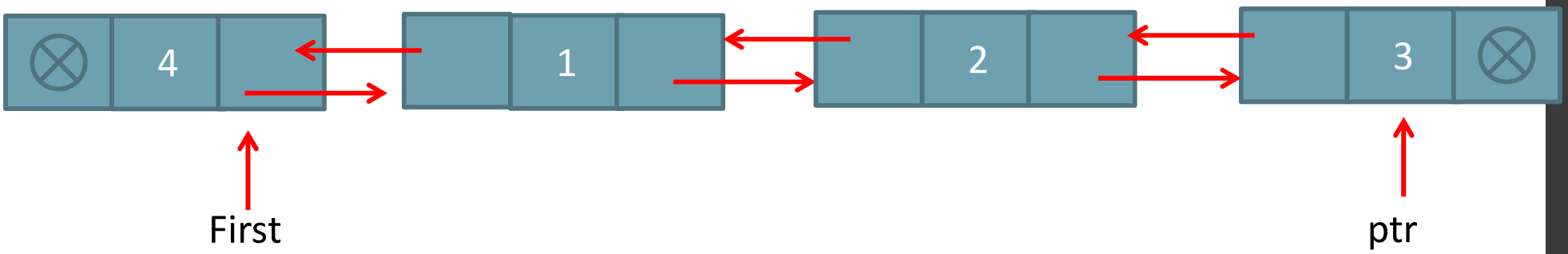

```
temp=first;
first=first->next;
free(temp);
first->prev=NULL;
```

Double Linked List
Delete node at Last Position



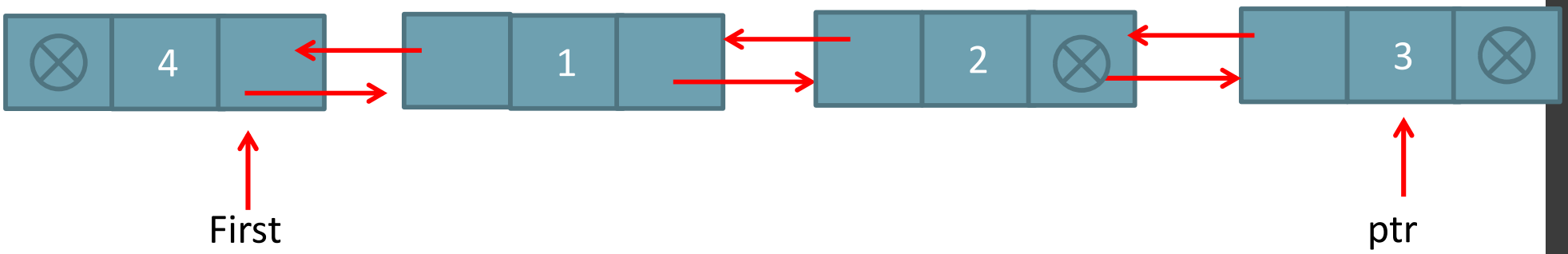

```
ptr=first;
while(ptr->next!=NULL){
        ptr=ptr->next;}
ptr->prev->next=NULL;
free(ptr);
```

Double Linked List
Delete node at Last Position



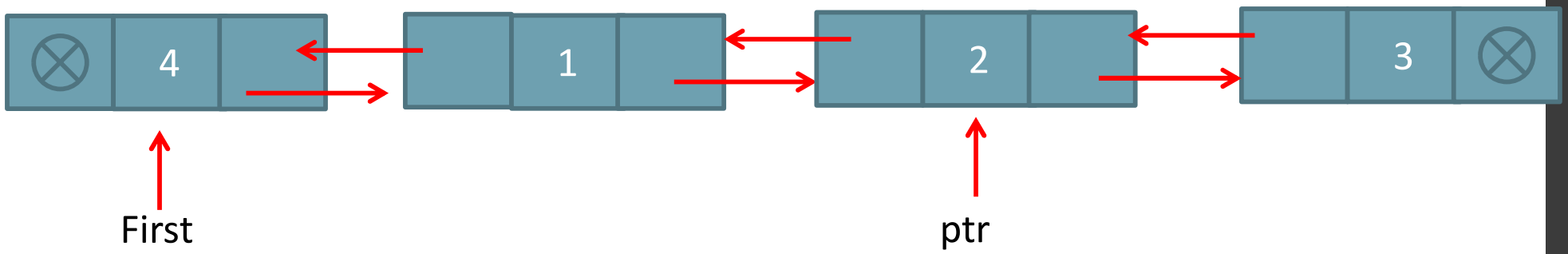

```
ptr=first;
while(ptr->next!=NULL){
        ptr=ptr->next;}
ptr->prev->next=NULL;
free(ptr);
```

Double Linked List
Delete node at Last Position

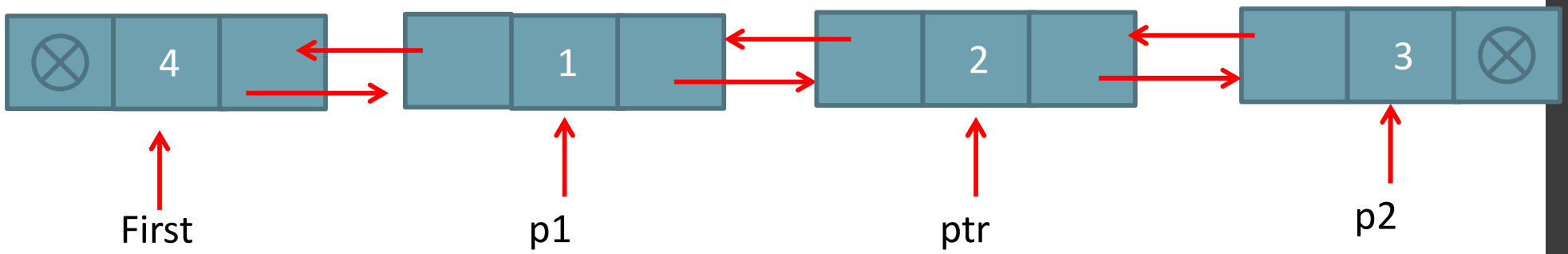



```
ptr=first;
while(ptr->next!=NULL){
        ptr=ptr->next;}
ptr->prev->next=NULL;
free(ptr);
```

Double Linked List
Delete node at some in between position

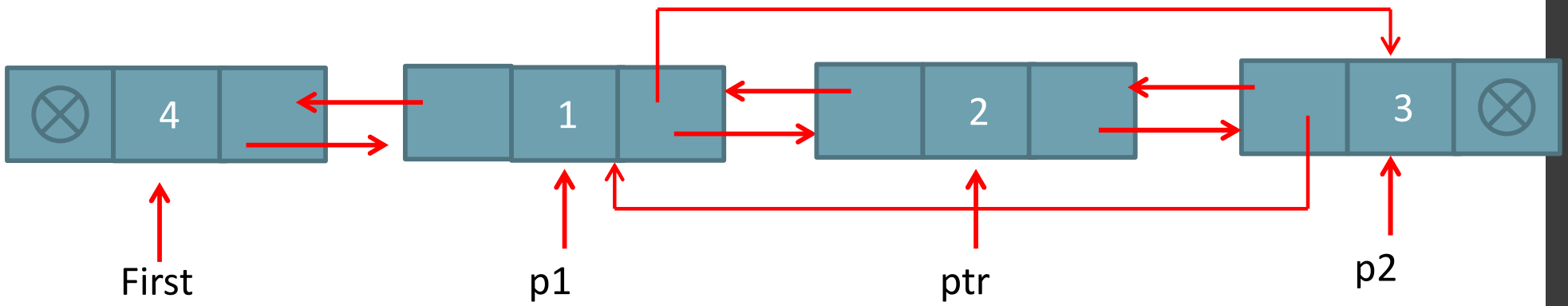


Double Linked List
Delete node at some in between position



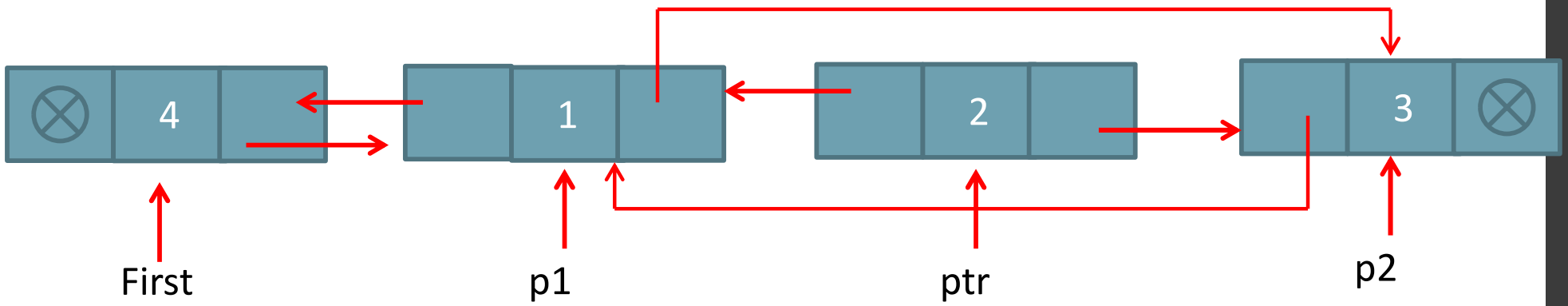

```
p1=ptr->prev;
p2=ptr->next;
p1->next=p2;
p2->prev=p1;
```

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



Method 1

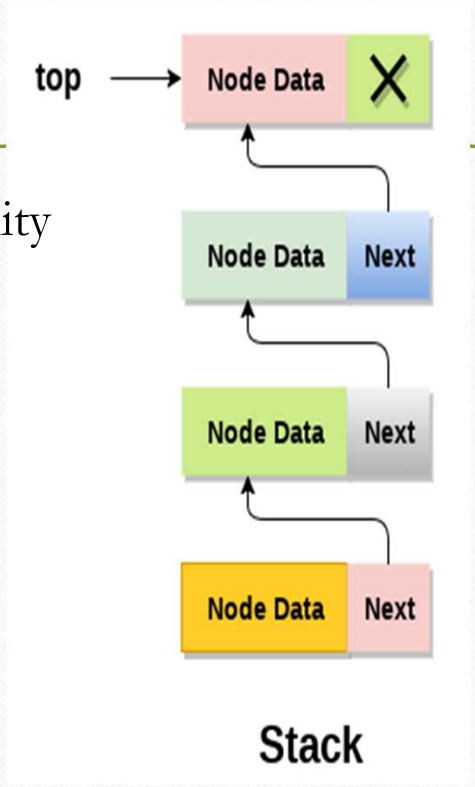
```
p1=ptr->prev;
p2=ptr->next;
p1->next=p2;
p2->prev=p1;
free(ptr);
```

Method 2

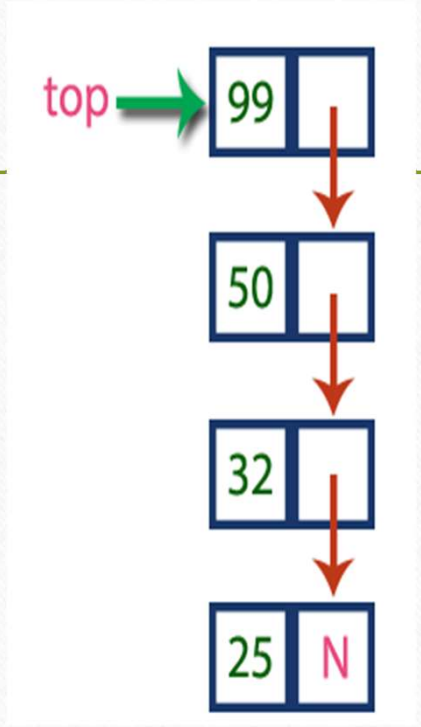
```
ptr->prev->next=ptr->next;
Ptr->next->prev=ptr->prev;
free(ptr);
```

Linked Stacks(Variants)

Complexity
O(n)



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



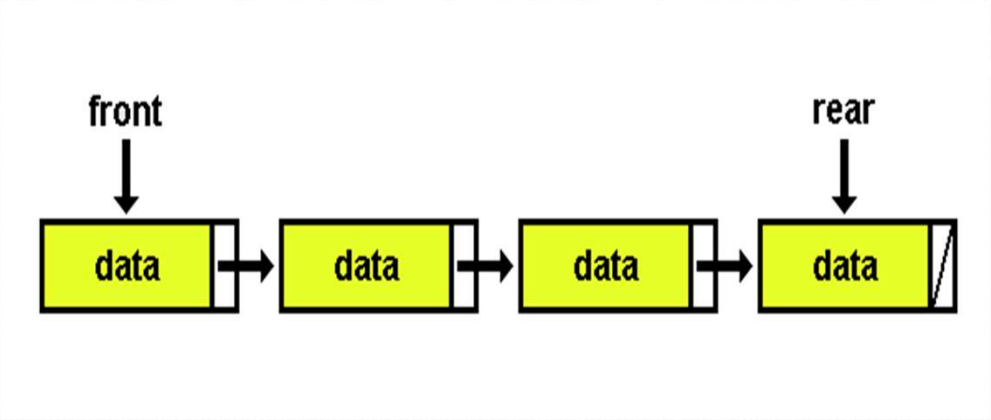
Complexity
O(1)

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