Data Structure and Algorithm
Linked List

Kanak Kalyani

Linked List and Arrays

Arrays

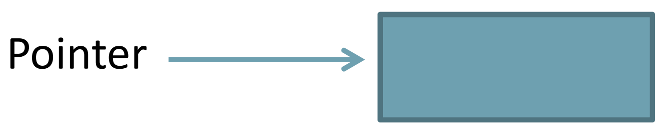
- The array size is fixed once it is created: Changing the size of the array requires creating a new array ad then copying all data from the old array to the new array
- The data items in the array are next to each other in memory: Inserting an item inside the array requires shifting other items

- 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



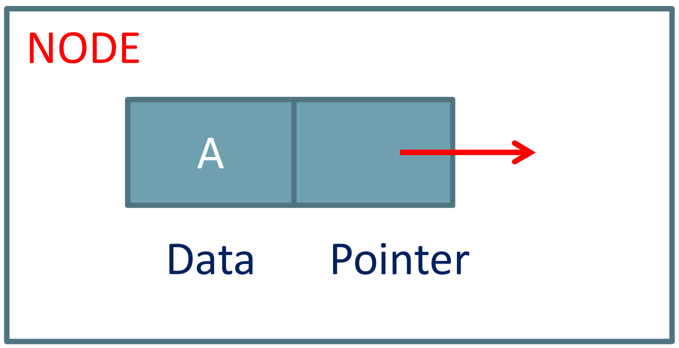
Linked List

- 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

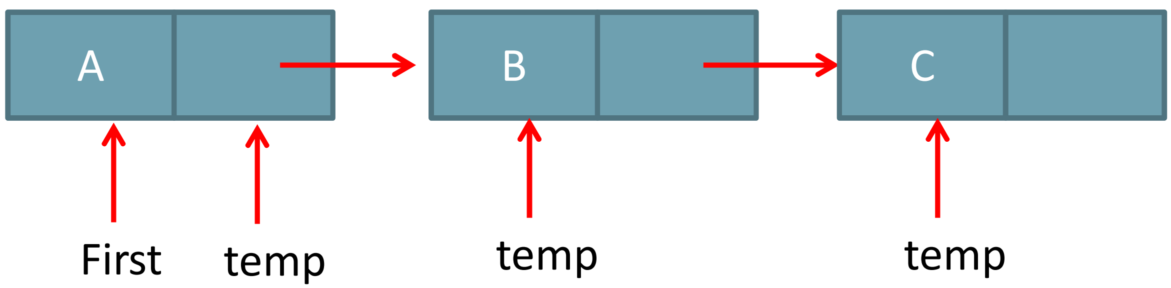
Linked List

- 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`



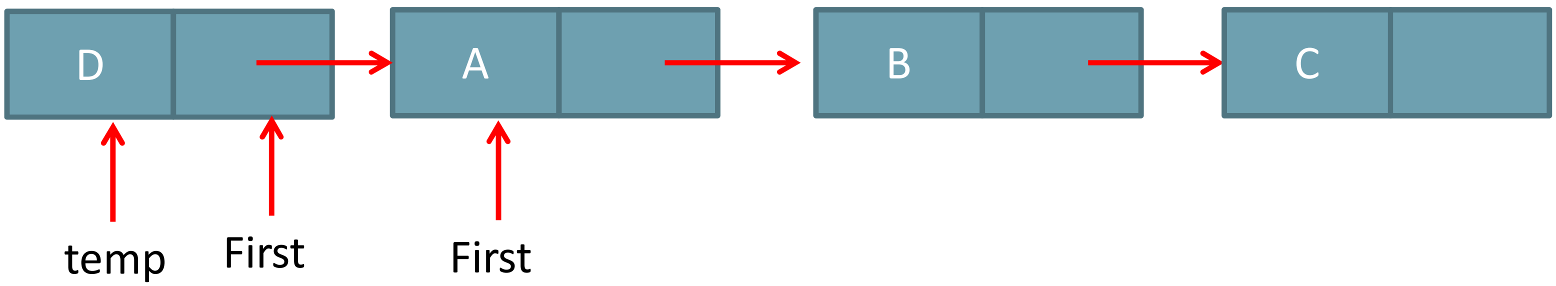

```
struct node
{
        char data;
        struct node *next;
};
```

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;
                }
        }
}
```

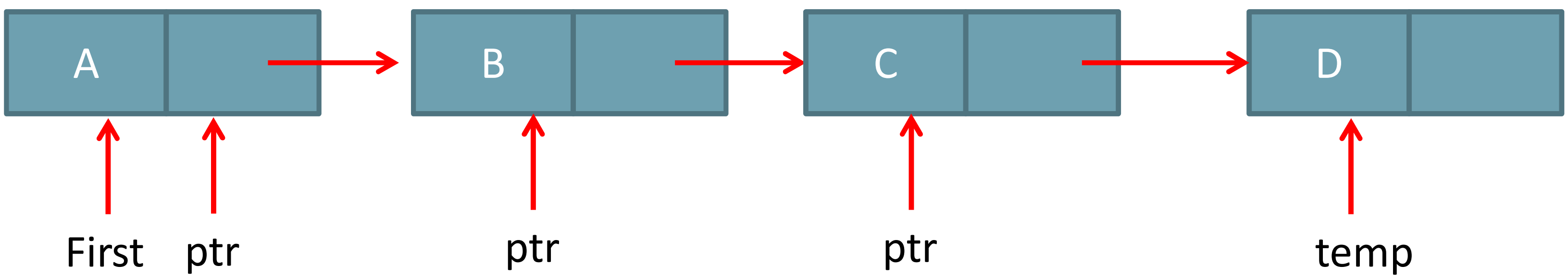
Inserting a new Node as First Node



temp->next=first;

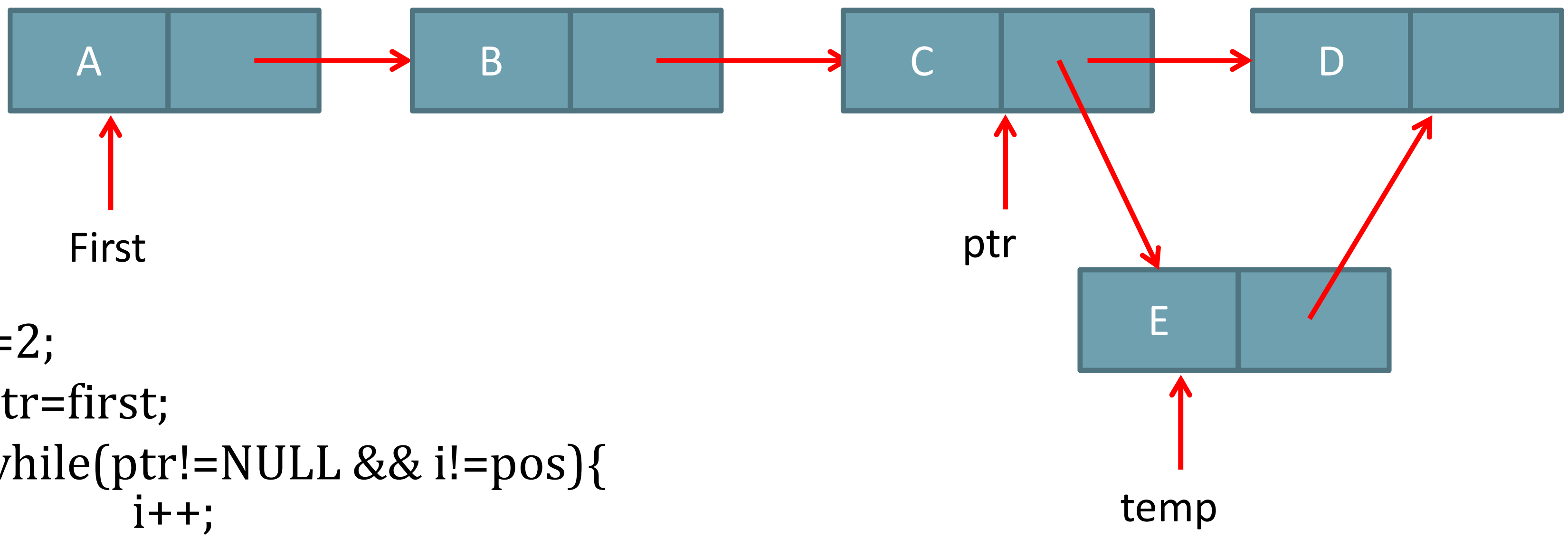
first=temp;

Inserting a new Node as Last Node



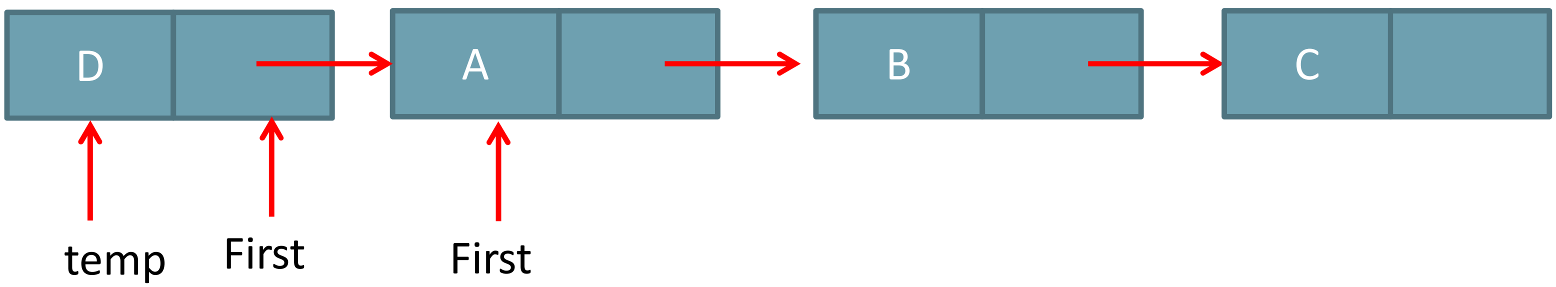

```
ptr=first;
while(ptr->next!=NULL)
        ptr=ptr->next;
ptr->next=temp;
```

Inserting a new Node: In Between



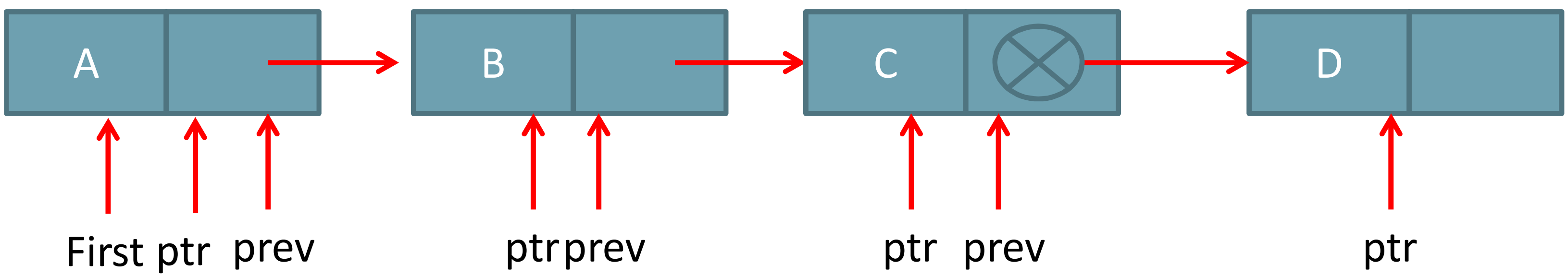

```
i=2;
ptr=first;
while(ptr!=NULL && i!=pos){
        i++;
        ptr=ptr->next;}
if(ptr==NULL)
        printf("insufficient number of nodes");
else{
        temp->next=ptr->next;
        ptr->next=temp;
}
```

Delete a node: First Node



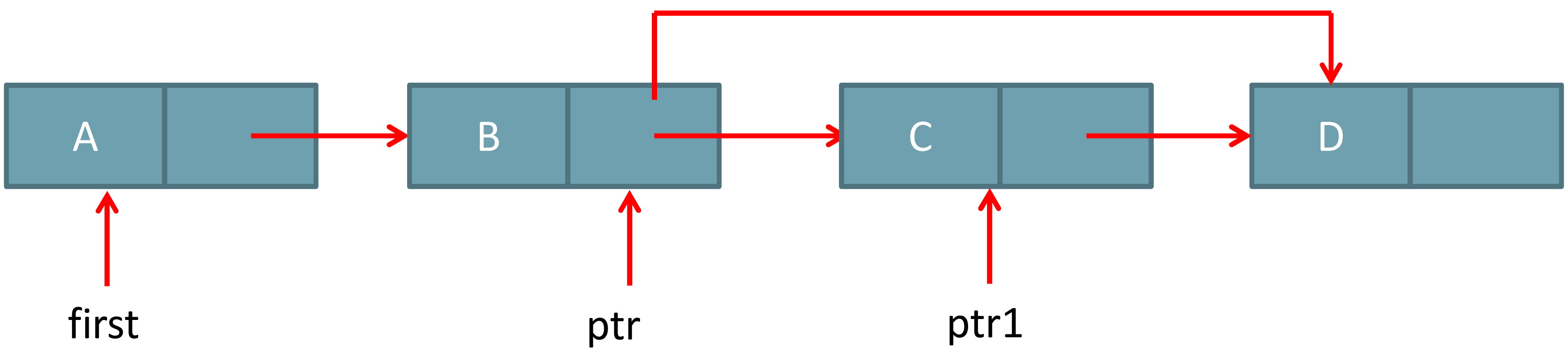

```
temp=first;
first=first->next;
free((void*)temp);
```

Delete a node: Last Node



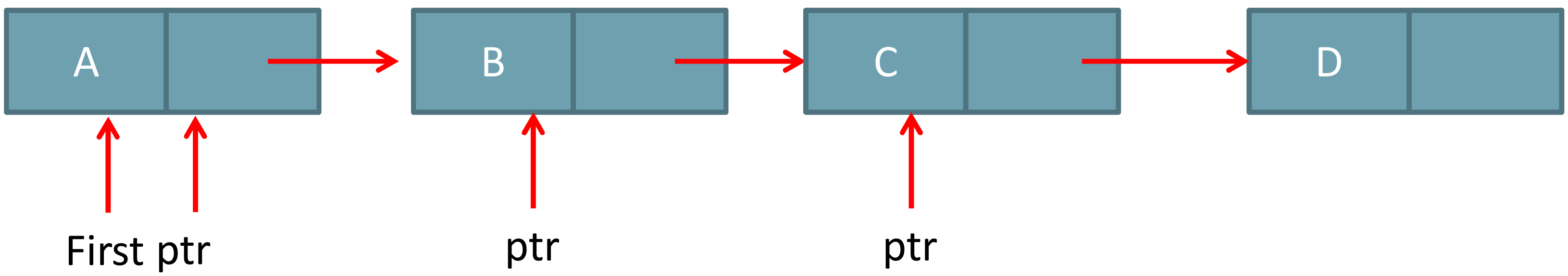

```
ptr=first;
while(ptr->next!=NULL){
        prev=ptr;
        ptr=ptr->next;
}
prev->next=NULL;
free(ptr);
```

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);
}
```

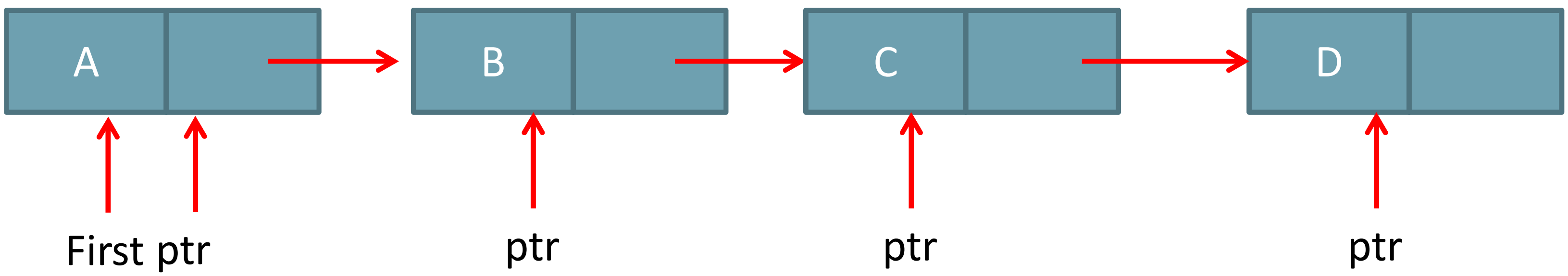
Update the value of a node




```
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

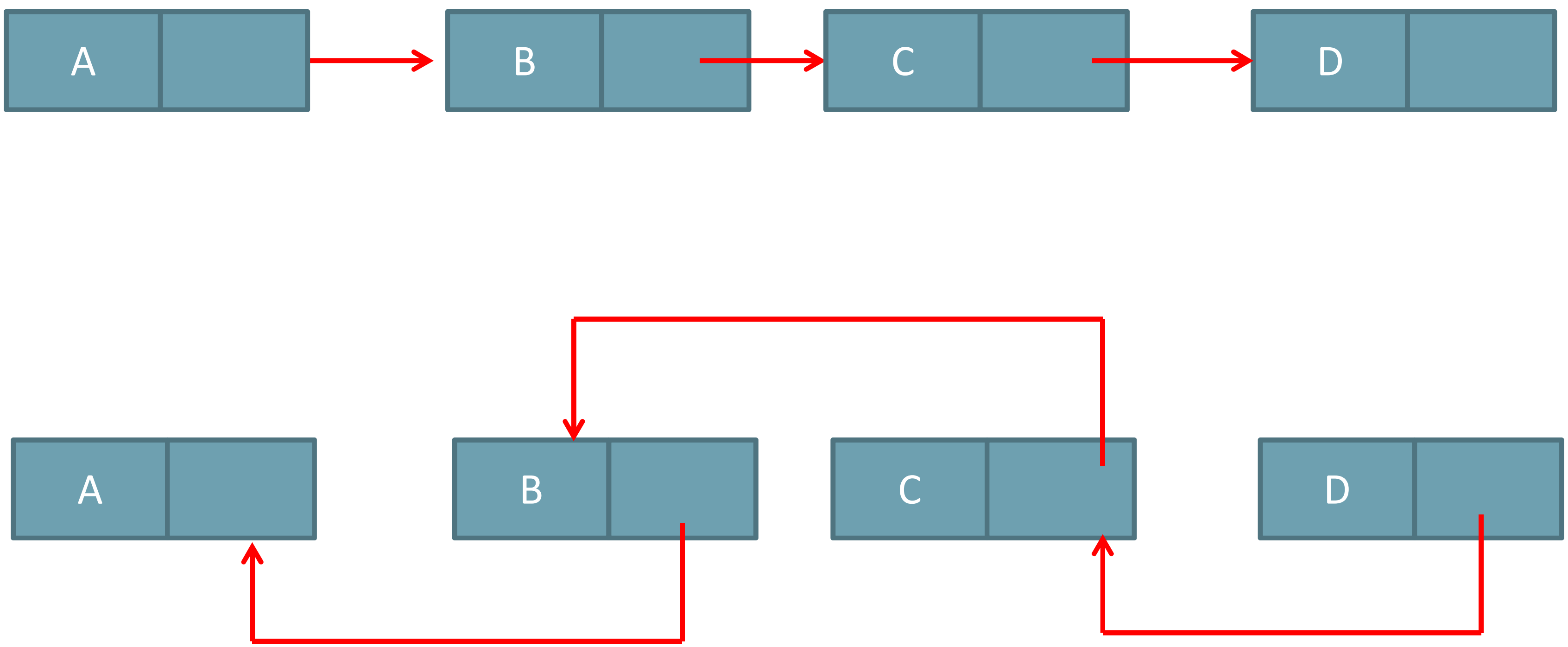



```
if(first==NULL)
        printf("List is empty");
else
{
        ptr=first;
        while(ptr!=NULL){
        printf("\t%d",ptr->data);
        ptr=ptr->link;
        }
}
```

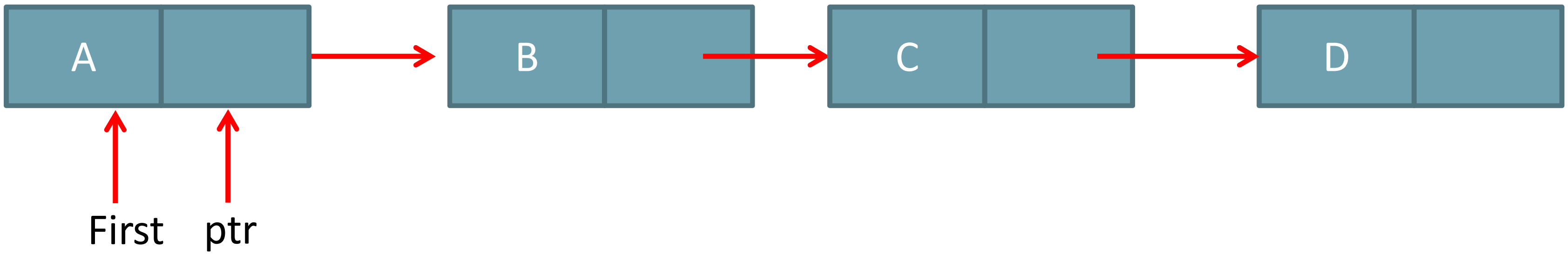
Operations on Linked List

- 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

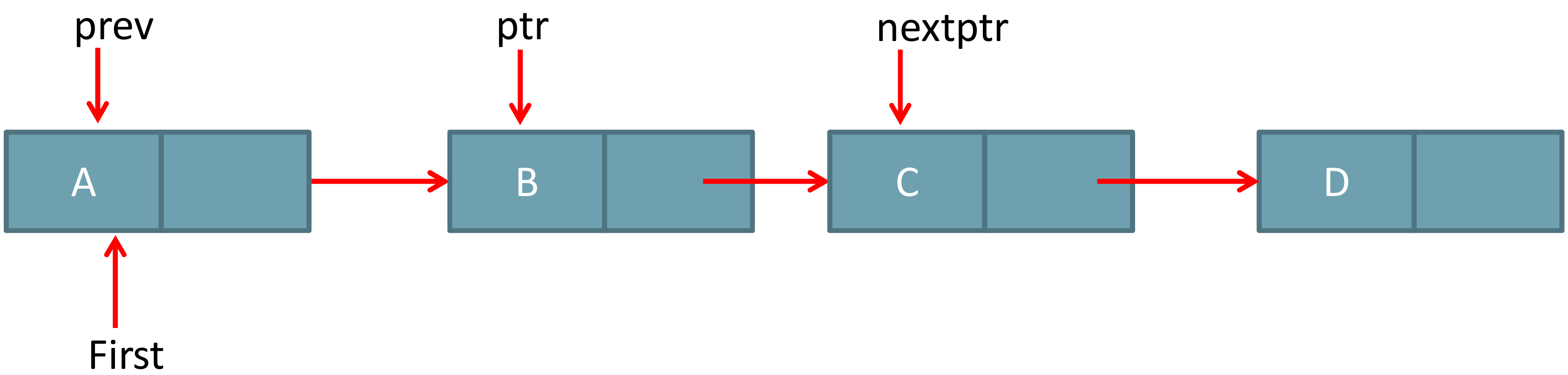


prev= ptr

ptr=ptr->next

nextptr=ptr->next

Reverse the Linked List



Step1:

prev= ptr

ptr=next

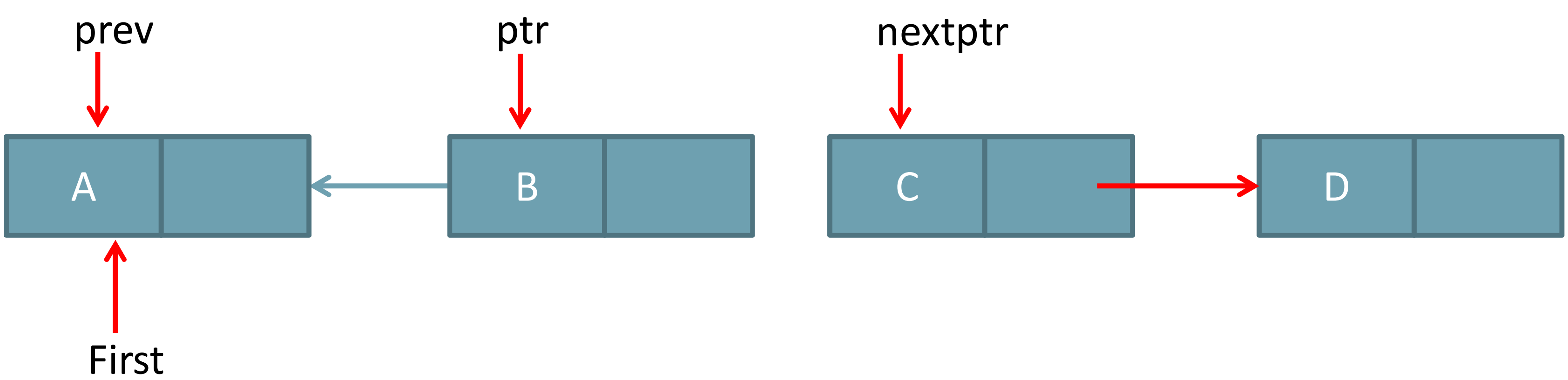
nextptr=ptr->next

Step2:

Reverse the link

ptr->next=prev

Reverse the Linked List

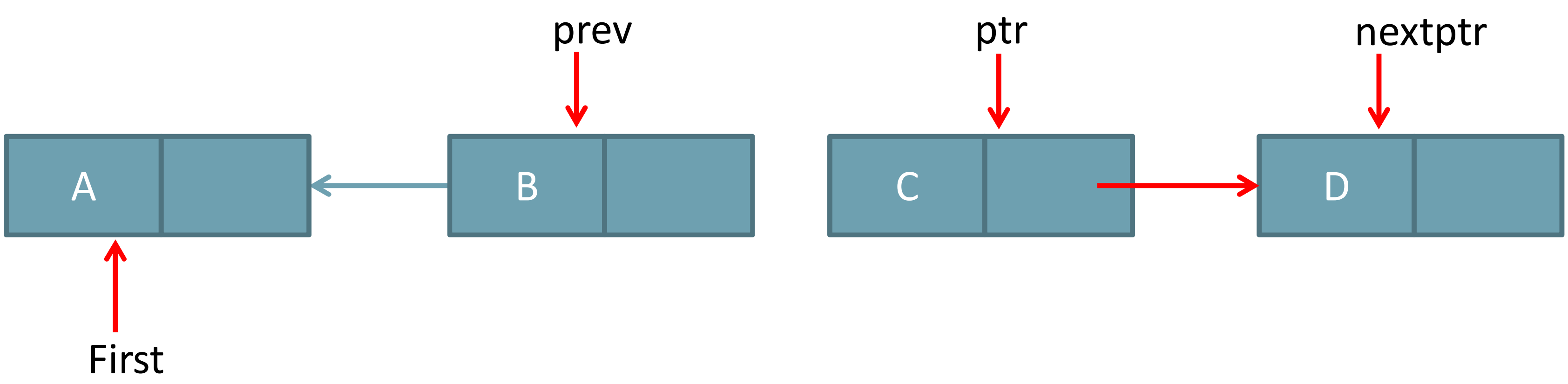


Step1:
prev= ptr
ptr=ptr->next
nextptr=ptr->next

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

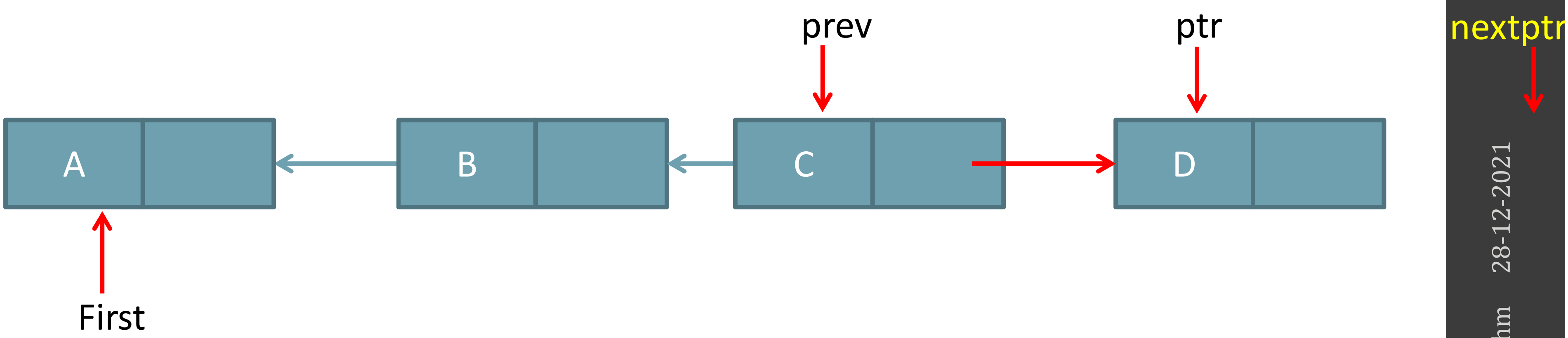
Step3:

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

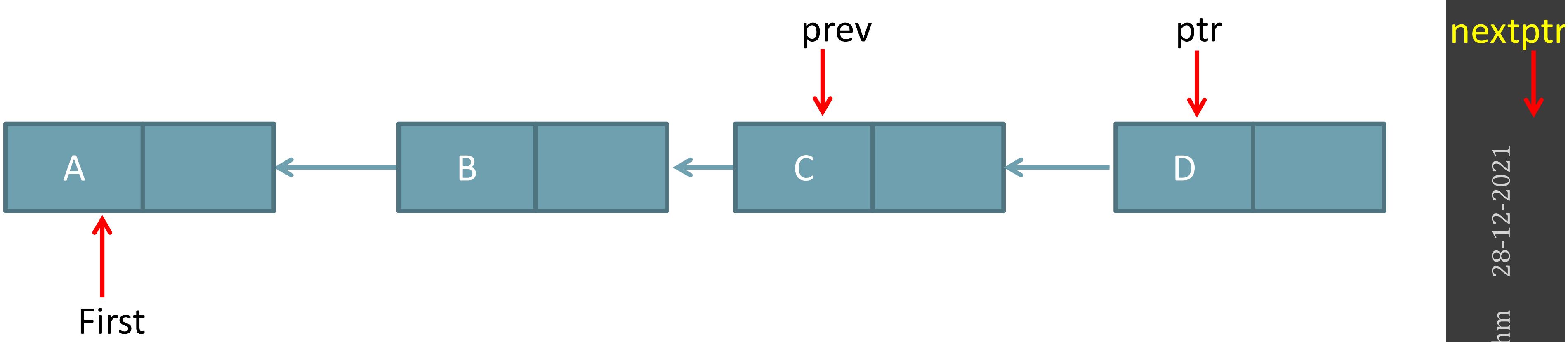
Step3:

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

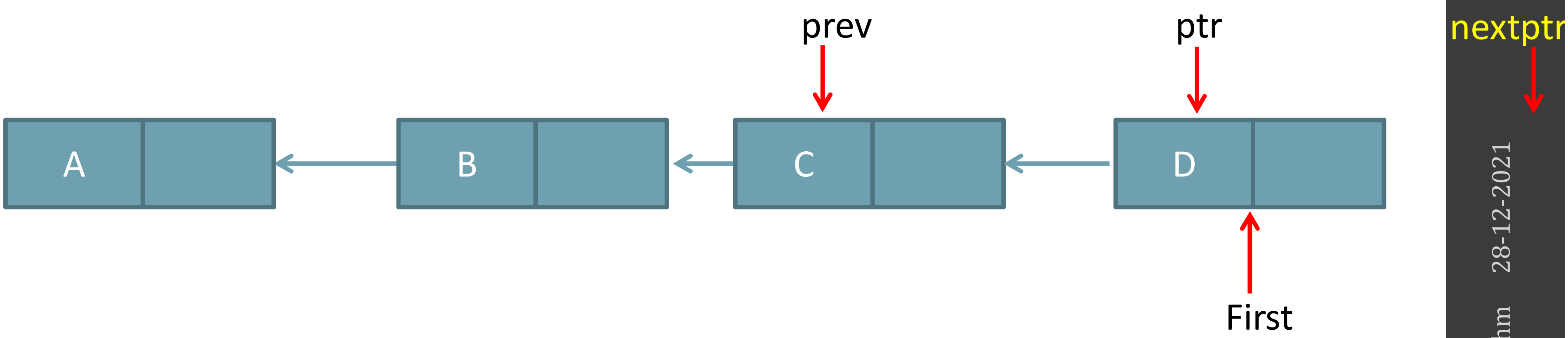
nextptr= nextptr->next

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

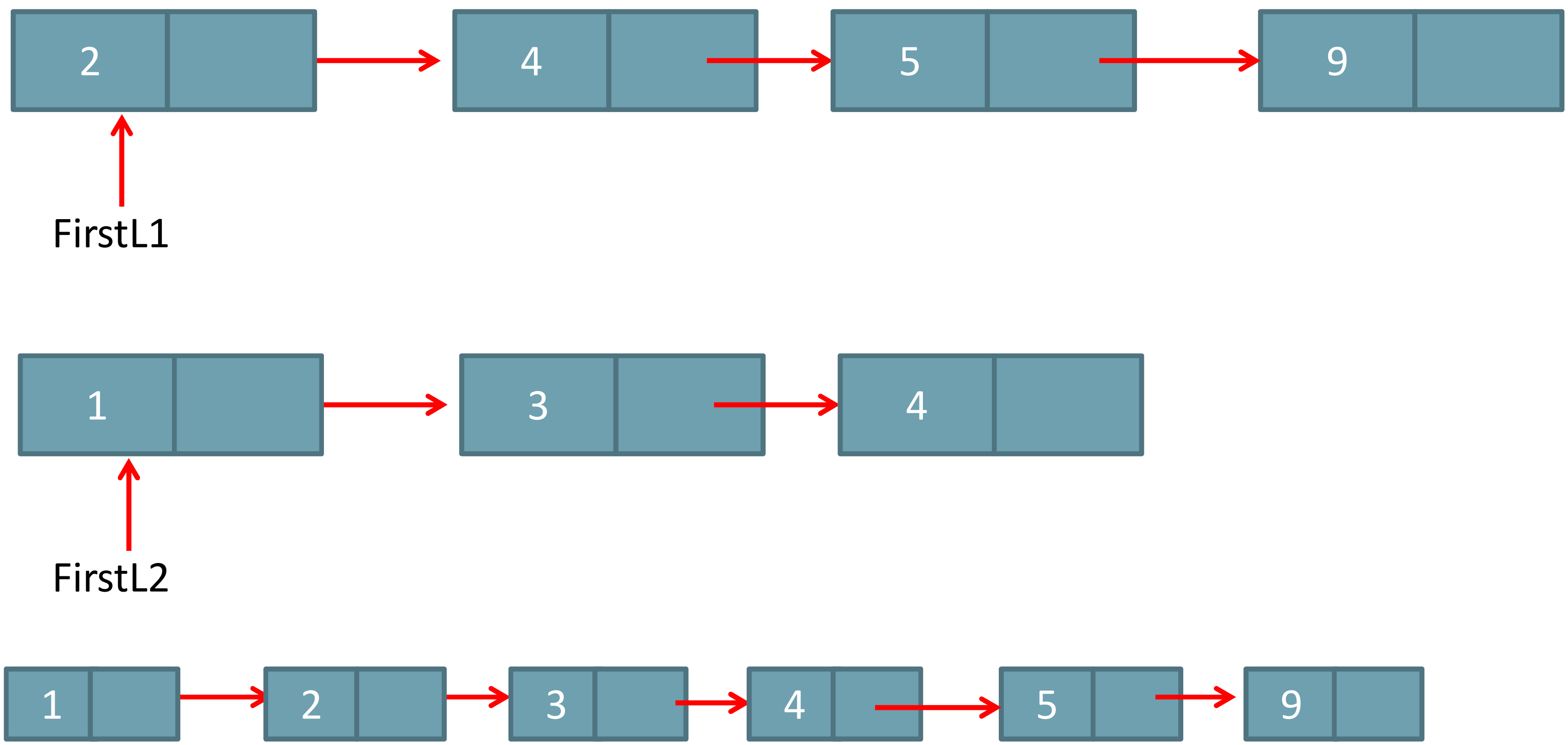
nextptr= nextptr->next

Step 4: change first pointer

first->next=null

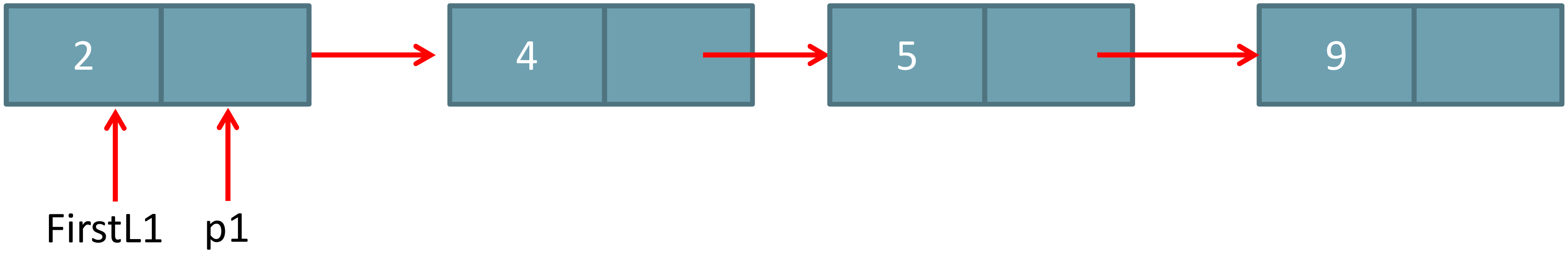
first = ptr

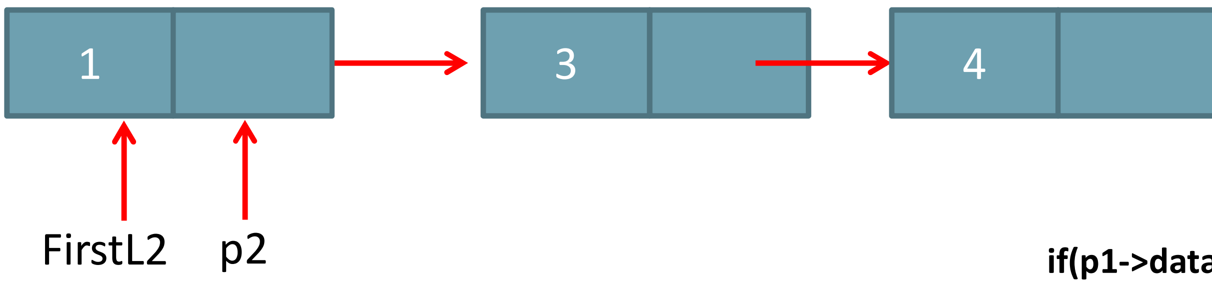
Merge two sorted Linked List

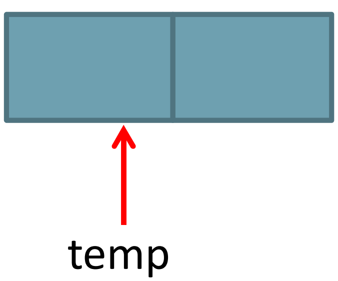


Kanak Kalyani

Merge two sorted Linked List

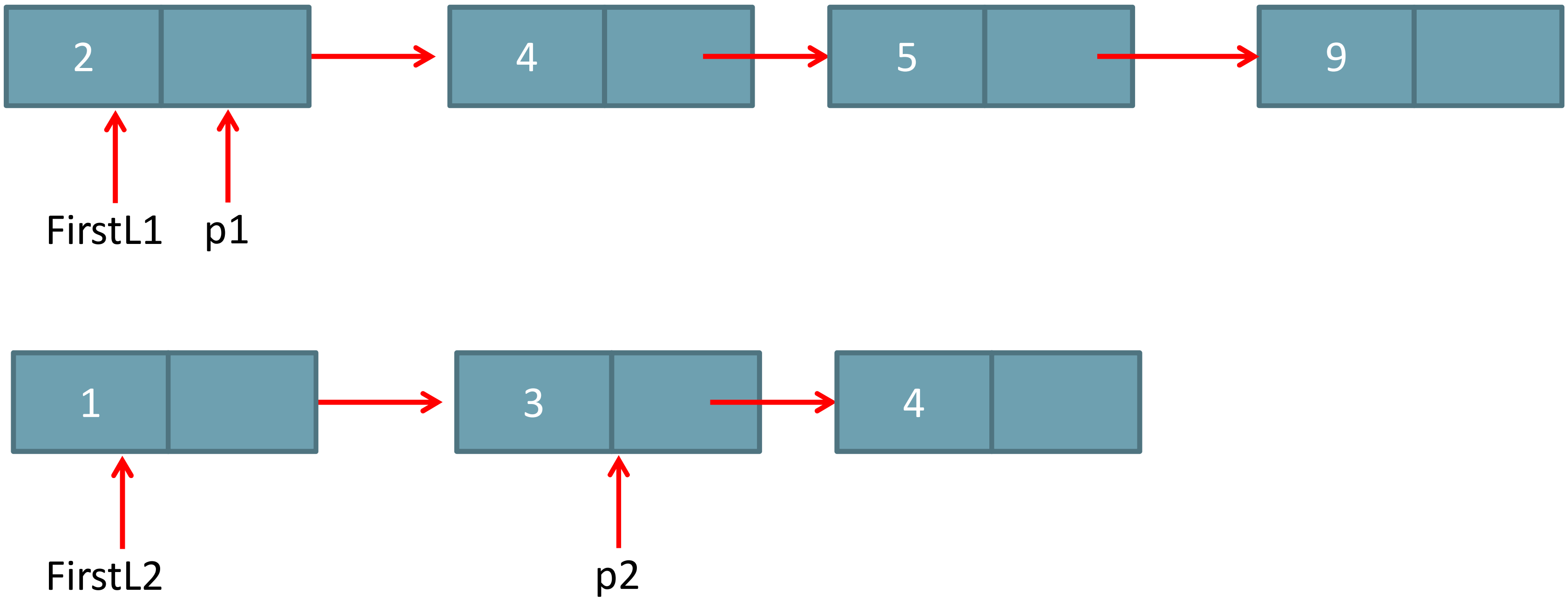


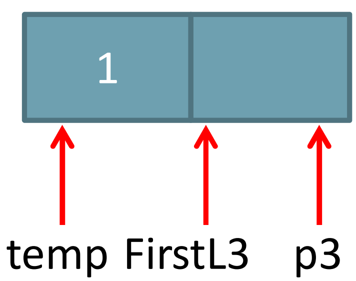





```
if(p1->data<p2->data){
        temp->data=p1->data;
        p1=p1->next; }
else if(p1->data>p2->data){
        temp->data=p2->data;
        p2=p2->next; }
else if(p1->data==p2->data){
        temp->data=p1->data;
        p1=p1->next;
        p2=p2->next;}
```

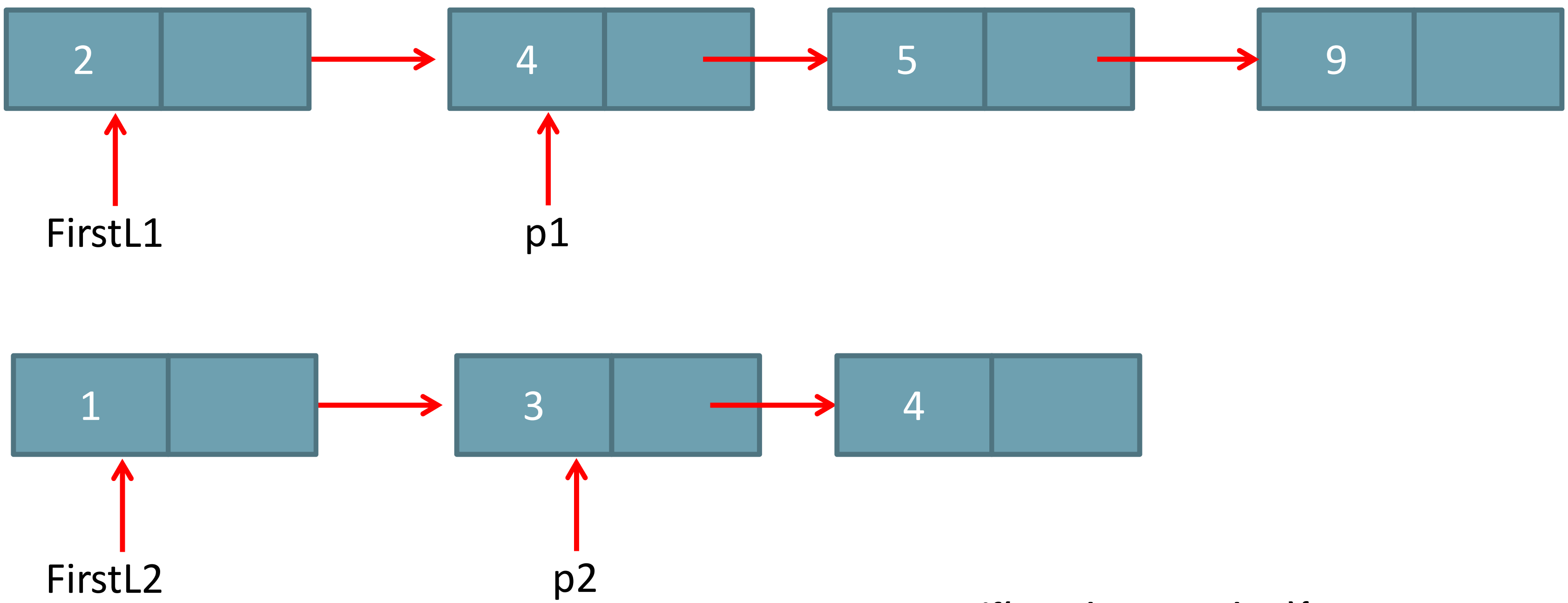
Merge two sorted Linked List

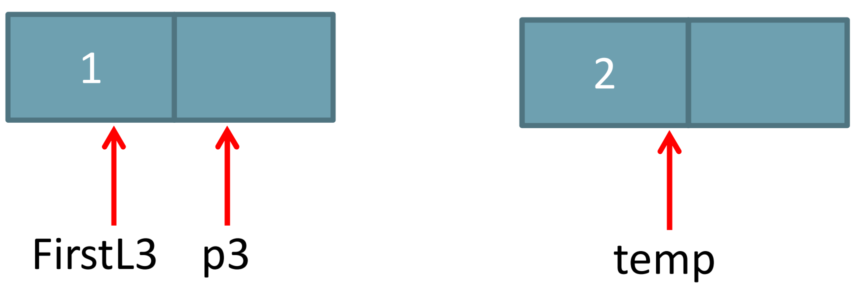





```
if(FirstL3==NULL){
        p3=temp;
        FirstL3=temp;}
else{
        p3-> next =temp;
        p3=temp;}
```

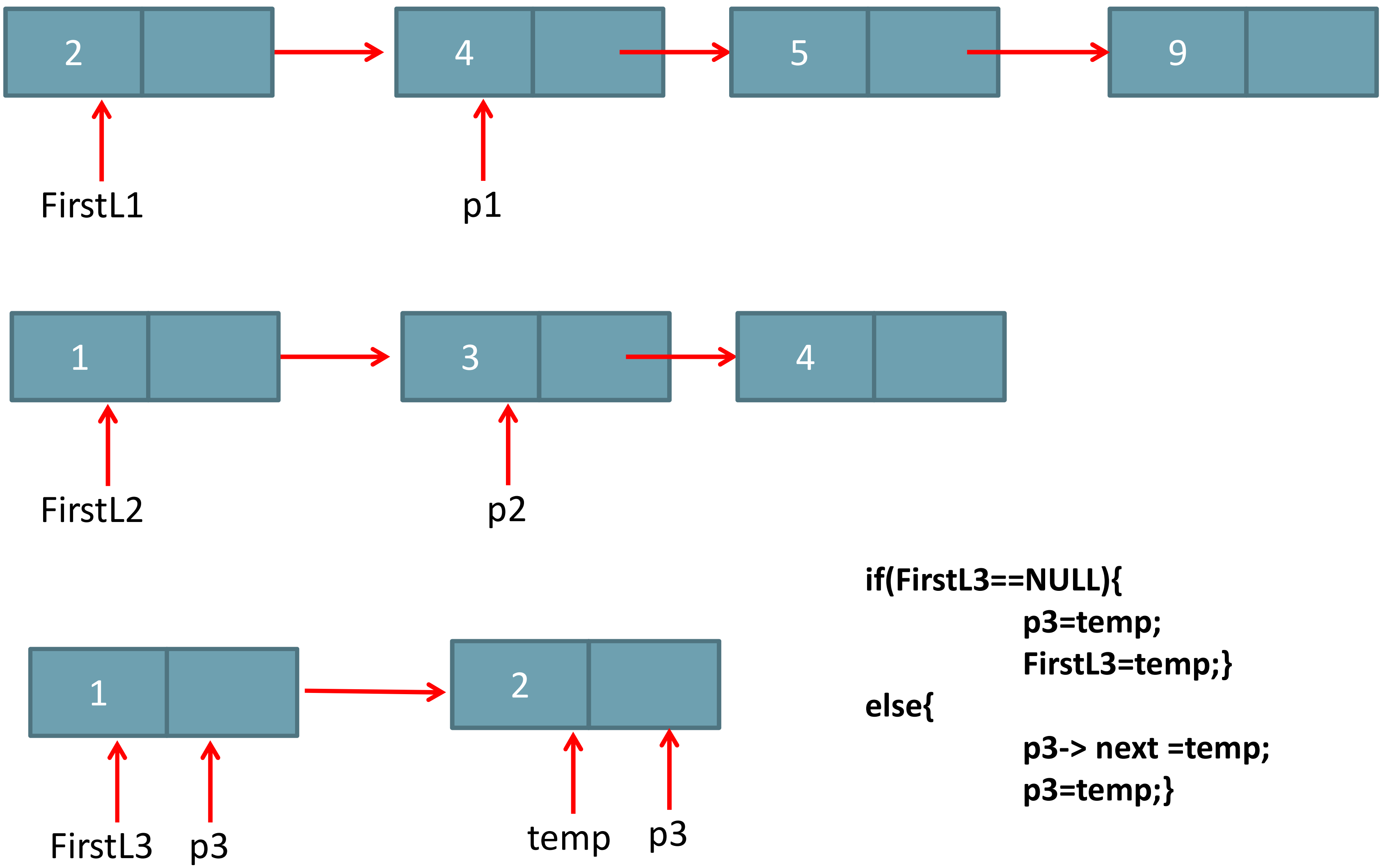
Merge two sorted Linked List





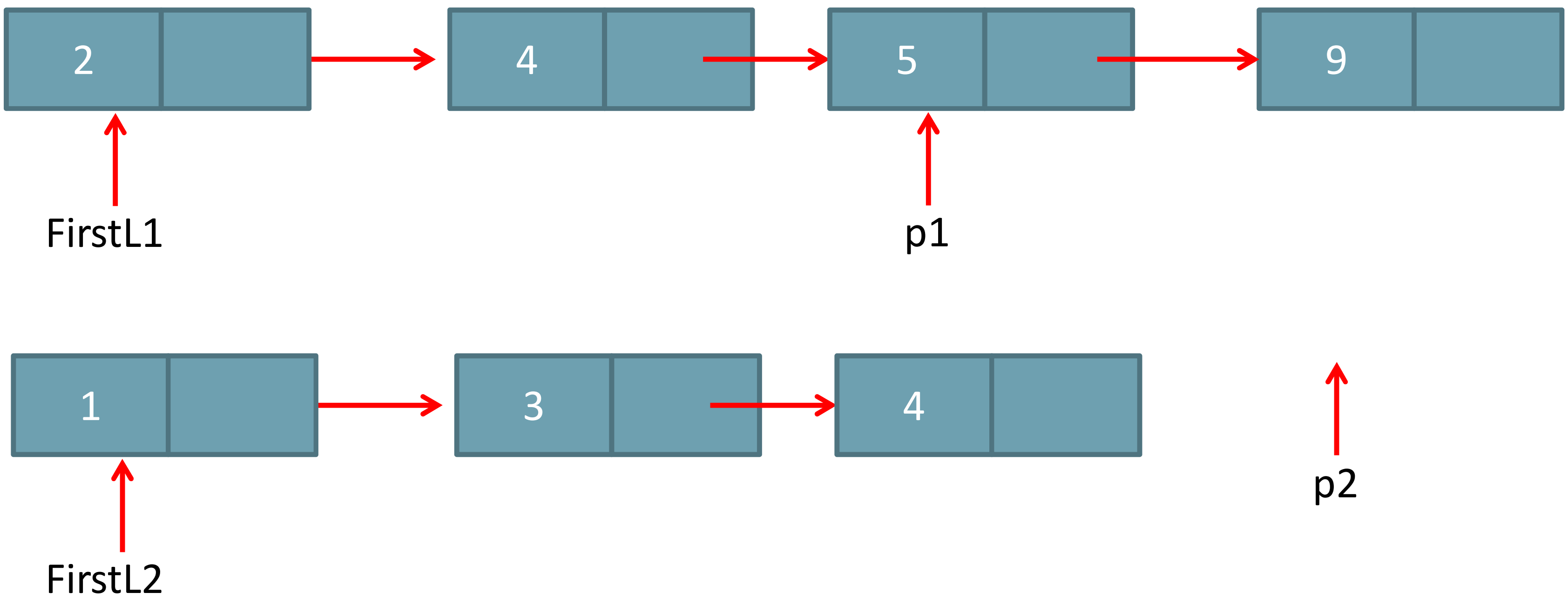

```
if(p1->data<p2->data){
        temp->data=p1->data;
        p1=p1-> next; }
else if(p1->data>p2->data){
        temp->data=p2->data;
        p2=p2-> next; }
else if(p1->data==p2->data){
        temp->data=p1->data;
        p1=p1-> next;
        p2=p2-> next;}
```

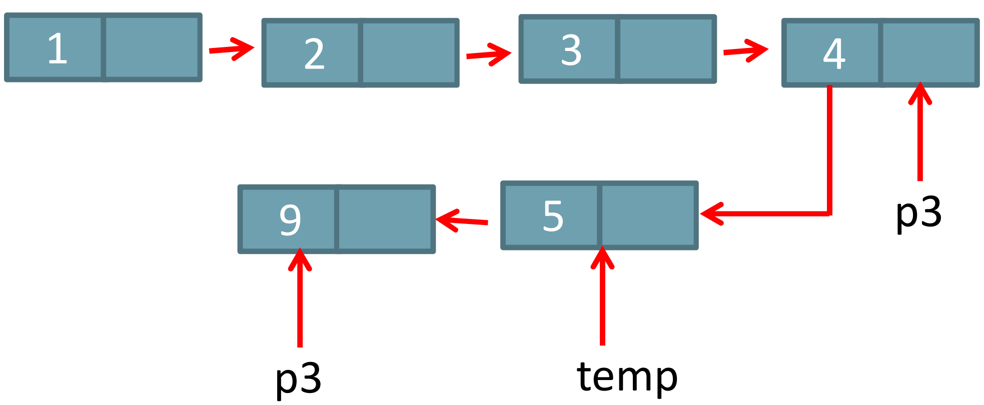
Merge two sorted Linked List




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

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;
}
```

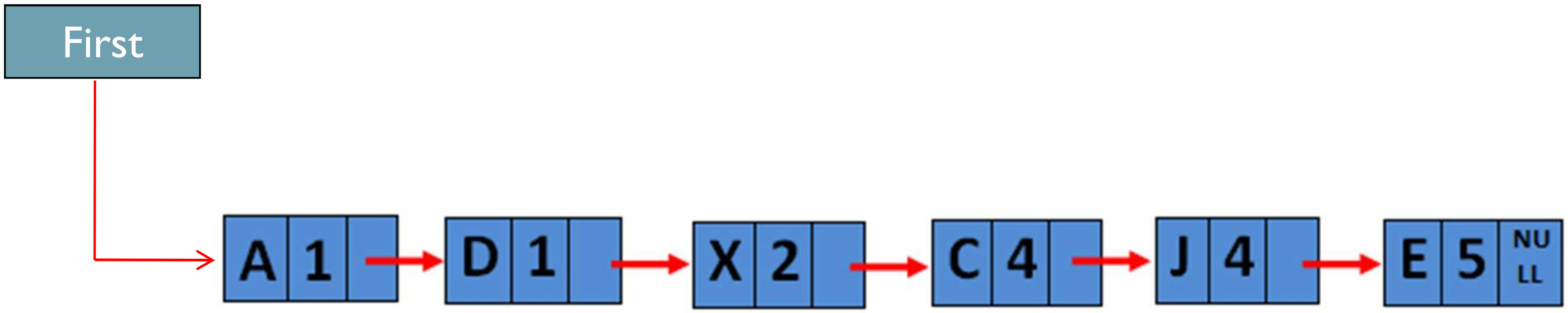
Merge two Linked List

```
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

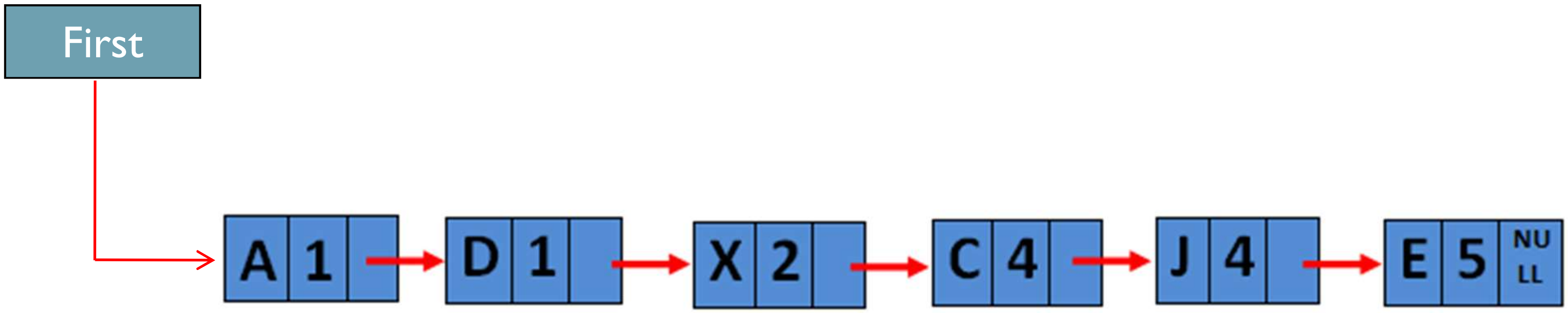
LINKED LIST REPRESENTATION OF A PRIORITY QUEUE



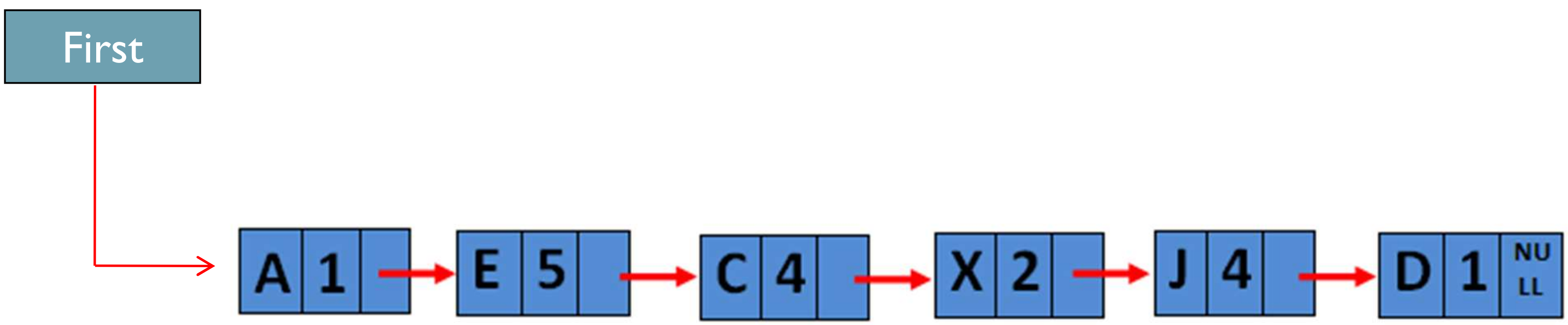

```
typedef struct Node
{
        int data;
        int priority;
        struct Node *next;
} node;
```

VARIANTS OF PRIORITY QUEUE

- Sorted List

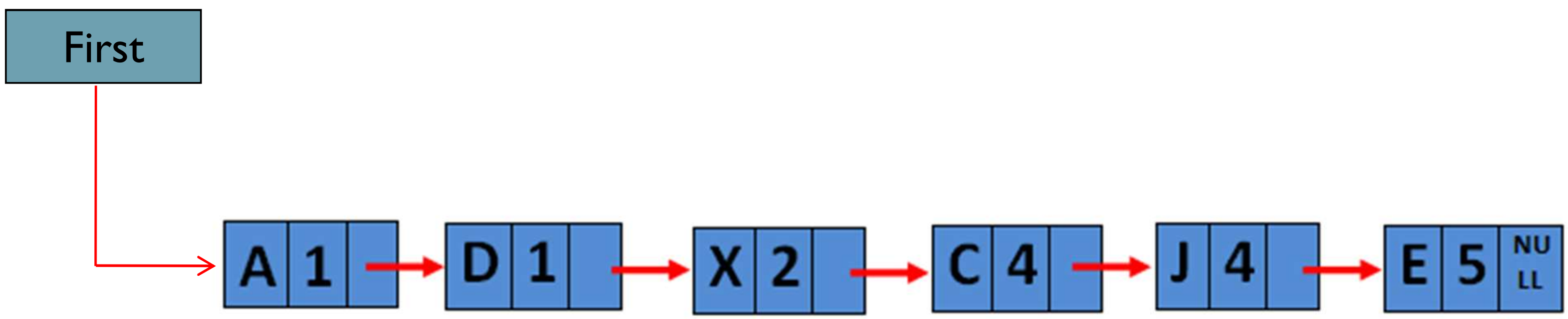


- Unsorted List



VARIANT OF PRIORITY QUEUE

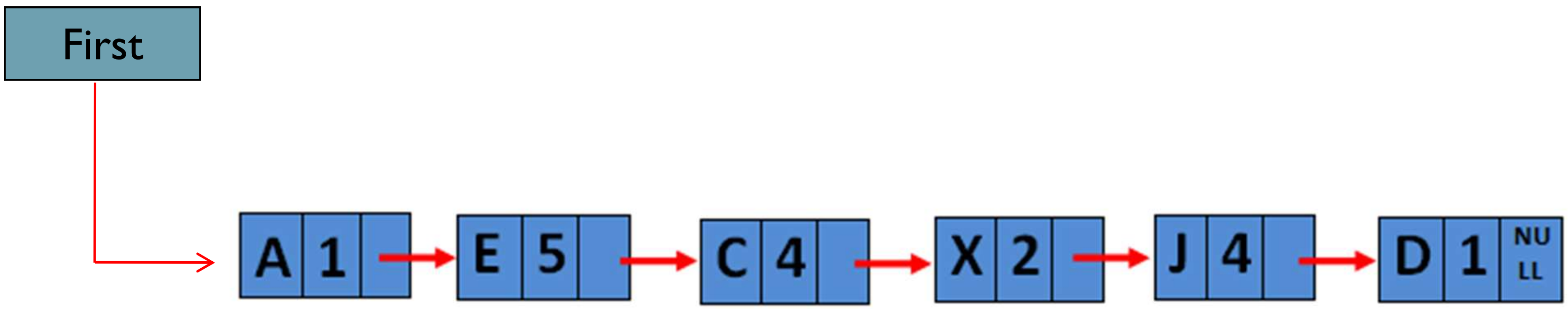
- Sorted List



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

VARIANT OF PRIORITY QUEUE

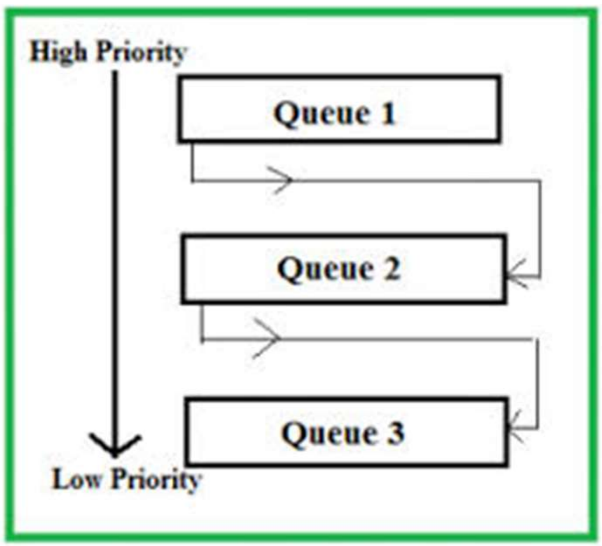
- Unsorted List

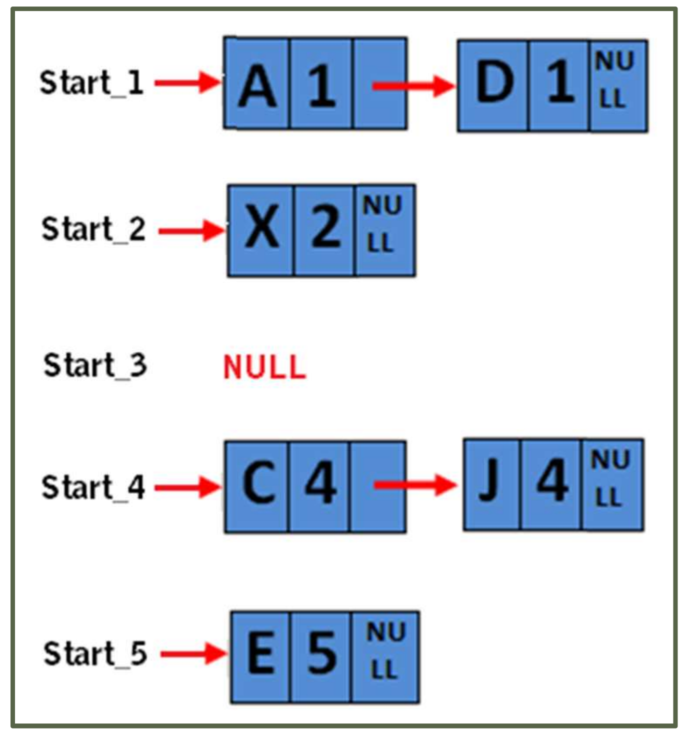


Complexity of Insertion – O(1)
Complexity of Deletion – O(n)

APPLICATION OF PRIORITY QUEUE

- Multilevel Queue Scheduling





ASSIGNMENT

- 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

P1->p2

P1->p3->p2

P1->p4->p3->p2

p1->