Outline : Lecture 5

Sorting Algorithms

- Insertion Sorting algorithm
- Analysis of Insertion Sorting Algorithm
- Selection Sorting algorithm
- Analysis of Selection Sorting Algorithm
- Bubble Sorting Algorithm
- Analysis of Bubble Sorting Algorithm
- Modified Bubble Sorting Algorithm
- Advantage of Modified Bubble Sort Algorithm

Linked List

- **Classification of Linked List**
- **Classification of Linked List with respect to Implementation**
  - *Static Linked List*
  - *Dynamic Linked List*
- **Operations on Single Linked List:**
  - *Creation of Single Dynamic Linked List*
  - *Display the Linked List (Iterative and recursive Algorithms)*
  - *Searching Operation*
  - *Insertion Operation*
  - *Deletion Operation*
  - *Reverse Print the Linked List (Iterative and Recursive Method)*
  - *Reverse the Linked List*

Insertion Sorting algorithm

- This Sorting Algorithm is very popular with bridge players when they are first sorting their cards.

Pass	A[0]	A[1]	A[2]	A[3]	A[4]
Initially	77	33	44	11	55
i=1	77	33			
i=2	33	77	44		
i=3	33	44	77	11	
i=4	11	33	44	77	55
Sorted	11	33	44	55	77

Analysis of Insertion Sort Algorithm:

Best case: When the list is already sorted, only one comparison is made on each pass.

Therefore, $f(n) = n - 1 = O(n)$

Worst case: If the list is initially sorted in reverse order, then

$f(n) = 1 + 2 + 3 + \cdots + (n-1) = O(n^2)$

```
void insertion_sort(int a[ ], int n)
{
 int i, j, y;
 /* initially a[0] may be thought of as a sorted file of one element.
   After each pass, the elements a[0] through a[k]  are in order */
 for ( i = 1; i < n ; i++)
 {
   y = a[i];
    /* move right one position all elements greater than y*/
   for ( j=i-1 ; j>=0 && y<a[j] ; j--)
      a[j + 1] = a[j];
   /* insert y at proper position */
   a[j + 1] = y;
 }
}
```

Selection Sorting algorithm: Example

Pass	position	Smallest	0	1	2	3	4	5
1	0	55	55	33	88	44	22	11
1	1	33	55	33	88	44	22	11
1	1	33	55	33	88	44	22	11
1	1	33	55	33	88	44	22	11
1	4	22	55	33	88	44	22	11
	5	11	55	33	88	44	22	11
Steps	5		11	33	88	44	22	55
2	1	33	11	33	88	44	22	55
2	1	33	11	33	88	44	22	55
2	1	33	11	33	88	44	22	55
2	4	22	11	33	88	44	22	55
	4	22	11	33	88	44	22	55
Steps	4		11	22	88	44	33	55

Pass	position	Smallest	0	1	2	3	4	5
3	2	88	11	22	88	44	33	55
3	3	44	11	22	88	44	33	55
3	4	33	11	22	88	44	33	55
	4	33	11	22	88	44	33	55
Steps	3		11	22	33	44	88	55
4	3	44	11	22	33	44	88	55
4	3	44	11	22	33	44	88	55
	3	44	11	22	33	44	88	55
Steps	2		11	22	33	44	88	55
5	4	88	11	22	33	44	88	55
	5	55	11	22	33	44	88	55
Steps	1		11	22	33	44	55	88
Sorted			11	22	33	44	55	88

Selection Sorting algorithm (Cont..)

Pass No.	No. of Steps	Total

Selection Sorting algorithm (Cont..)

```
void selection_sort(int a[ ], int n)
{
  int i, j, smallest, position;
  /* we select the position one by one starting from 0 to (n − 2). When we select a particular position, the
smallest element at that position is a[position] at the beginning*/
 for ( i = 0; i < (n − 1) ; i ++)
 {
    position = i; smallest = a[i];
    for ( j = i + 1 ; j < n ; j ++)
      {
       if (smallest > a[j])
          {
                          position = j; smallest = a[j];
          }
      }
     if (i < position)
      {
       a[position] = a[i]; a[i] = smallest;
      }
  }
}
```

Bubble Sorting algorithm :: Example

Pass	Swapping Status (1/0)	0	1	2	3	4	5
1	1	55	33	88	44	22	11
1	0	33	55	88	44	22	11
1	1	33	55	88	44	22	11
1	1	33	55	44	88	22	11
1	1	33	55	44	22	88	11
Steps	5	33	55	44	22	11	88
Pass	Swapping Status (1/0)	0	1	2	3	4	5
2	0	33	55	44	22	11	88
2	1	33	55	44	22	11	88
2	1	33	44	55	22	11	88
2	1	33	44	22	55	11	88
Steps	4	33	44	22	11	55	88

Pass	Swapping Status (1/0)	0	1	2	3	4	5
3	0	33	44	22	11	55	88
3	1	33	44	22	11	55	88
3	1	33	22	44	11	55	88
Steps	3	33	22	11	44	55	88
Pass	Swapping Status (1/0)	0	1	2	3	4	5
4	1	33	22	11	44	55	88
4	1	22	33	11	44	55	88
Steps	2	22	11	33	44	55	88
Pass	Swapping Status (1/0)	0	1	2	3	4	5
5	1	22	11	33	44	55	88
Steps	1	11	22	33	44	55	88
Sorted		11	22	33	44	55	88
Total No. of Passes							

Bubble Sorting algorithm (Cont..)

Pass No.	No. of Steps	Total

Bubble Sorting algorithm (Cont..)

```
void bubble_sort(int a[ ], int n)
{
 int i, j, hold;
 for ( i = 1; i < n ; i ++) // outer loop control the number of passes
 {
   for ( j = 0 ; j < n − i ; j ++)
     {
       if (a[j] > a[j + 1])
         {
             hold = a[j];
             a[j] = a[j + 1];
             a[j + 1] = hold;
         }
     }
 }
}
```

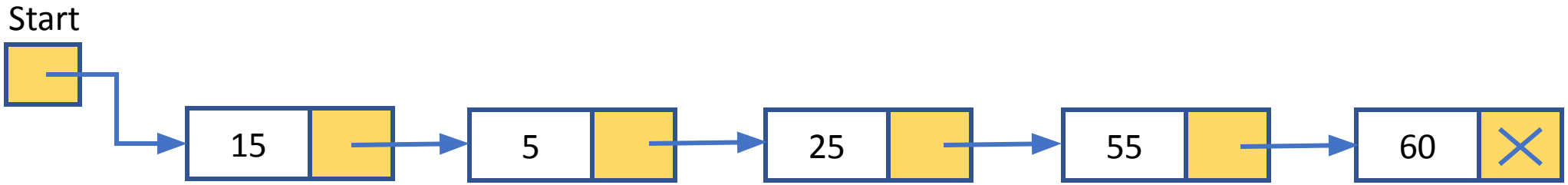
Modified Bubble Sorting algorithm :: Example

Pass	Swapping Status (1/0)	0	1	2	3	4	5
1	0	11	22	33	44	55	66
1	0	11	22	33	44	55	66
1	0	11	22	33	44	55	66
1	0	11	22	33	44	55	66
1	0	11	22	33	44	55	66
Steps	5	11	22	33	44	55	66
No swapping is done in pass 1. So, the question is whether we will continue the nest pass or not?		So, we don't need to continue the next pass as because the current pass indicates the list is sorted.					

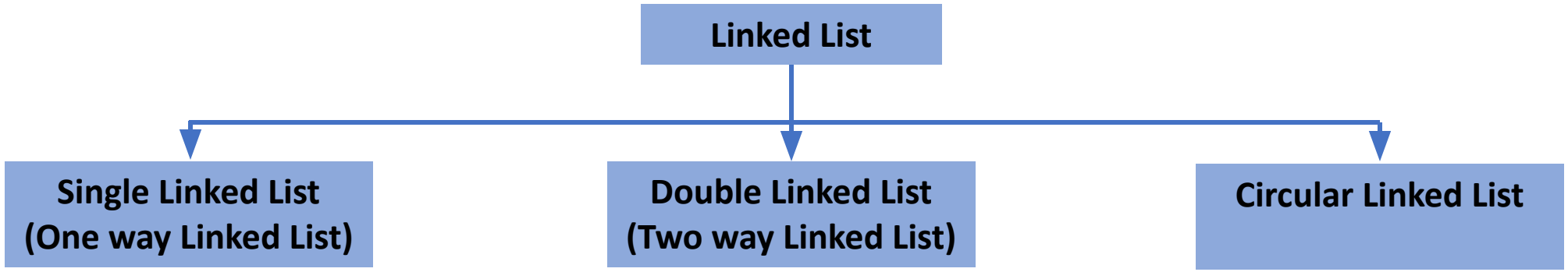
Best Case: If the list is sorted, then $f(n) = (n-1) = O(n)$
Worst Case: $f(n) = O(n^2)$.

```
void bubble_sort(int a[ ], int n)
{
 int i, j, hold, flag = 1;
 for ( i = 1; (i < n)&& (flag) ; i ++)
  // outer loop control the number of passes
 {
   flag = 0;
   for ( j = 0 ; j < n – i ; j ++)
     {
       if (a[j] > a[j + 1])
         {
             hold = a[j];
             a[j] = a[j + 1];
             a[j + 1] = hold;
             flag = 1;
         }
     }
 }
}
```

Linked List

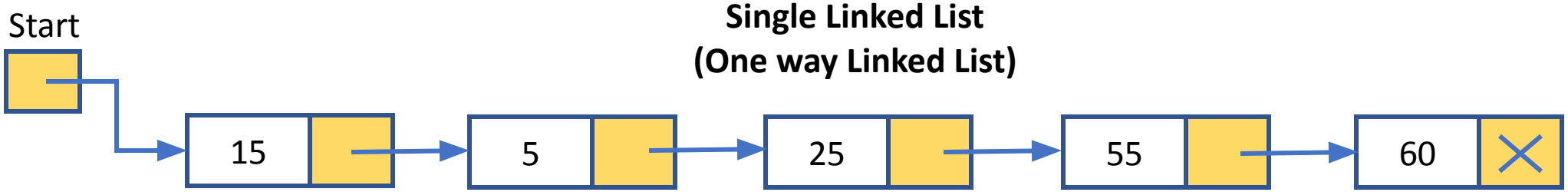


- **A linked list is a linear data structure where the linearity is maintained by means of pointers.**
- **The linked list is a collection of nodes, where each node has at least two parts: first part contains the information / data and the second part, called the link part, contains the address of the next node if exist, otherwise contains null pointer.**

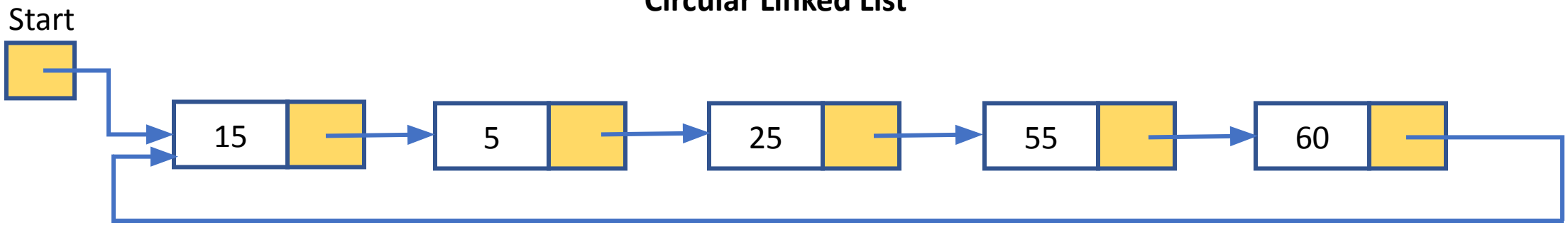


Linked List Classification

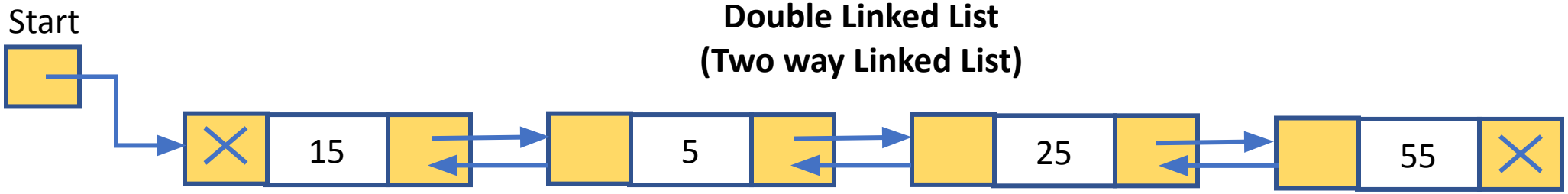
Single Linked List
(One way Linked List)



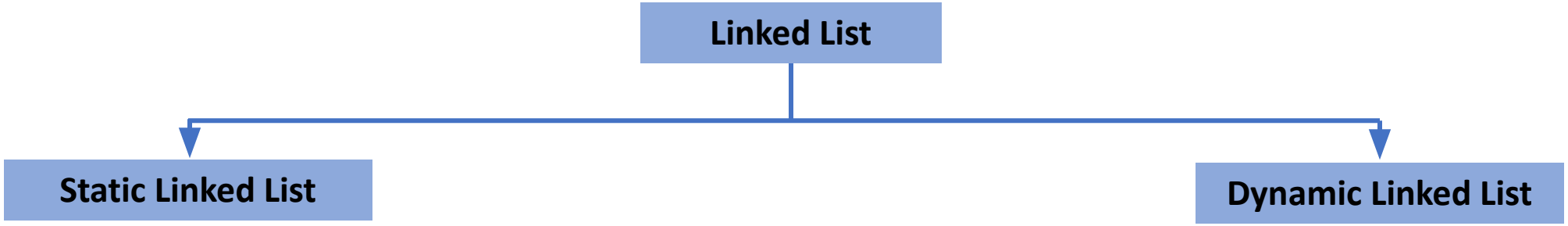
Circular Linked List



Double Linked List
(Two way Linked List)

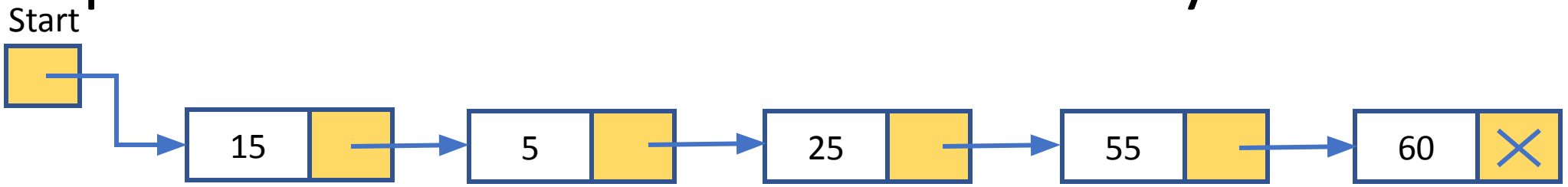


Linked List Classification with respect to Implementation



- **Static Linked List:** A linked list, which is implemented using array data structure, is called static linked list.
- **Dynamic Linked List:** A linked list, which is implemented using dynamic memory allocation concept (memory space will be allocated / deallocated during run time of the program)

Representation of Linked List in Memory:



Start

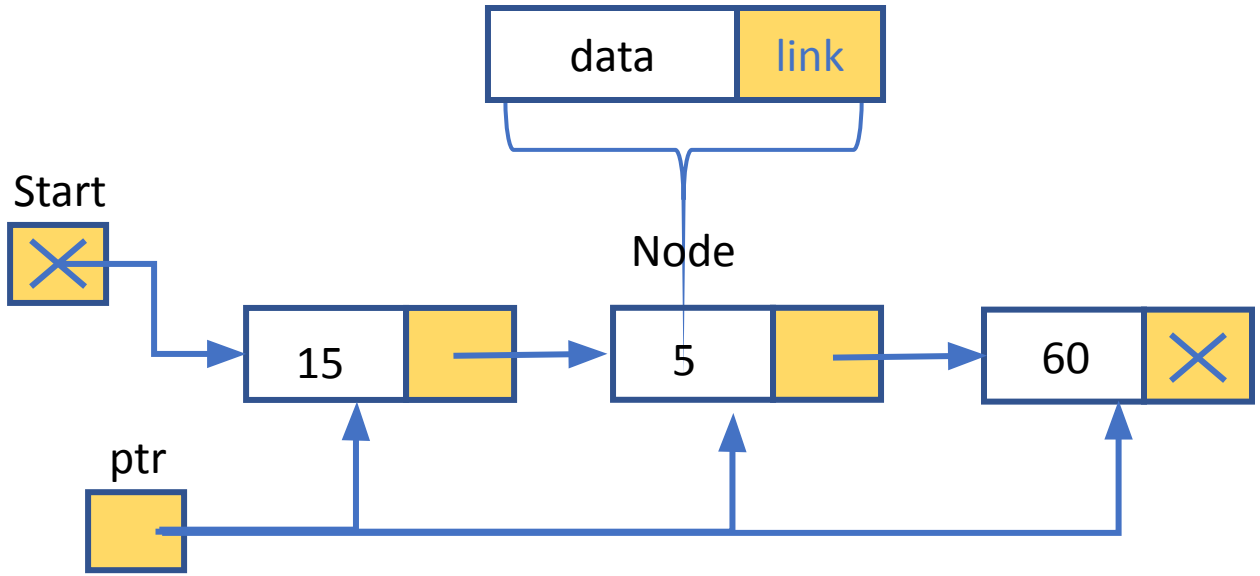
3

	INFO
0	5
1	55
2	
3	15
4	
5	60
6	
7	25
8	

	LINK
0	7
1	5
2	
3	0
4	
5	-1
6	
7	1
8	

Implementation of Dynamic Linked List:

```
typedef struct node
{
   int data ;
   struct node * link;
 }  nd;
```



Step 1: nd * start=NULL;

Step 2: ptr=(nd*) malloc(sizeof(nd));

Step 3: ptr->data=15;
start=ptr;

Step 4: Do you want to continue?(y/n)
If no, then go to Step 5;

Step 4.1: ptr->link=(nd*) malloc(sizeof(nd));

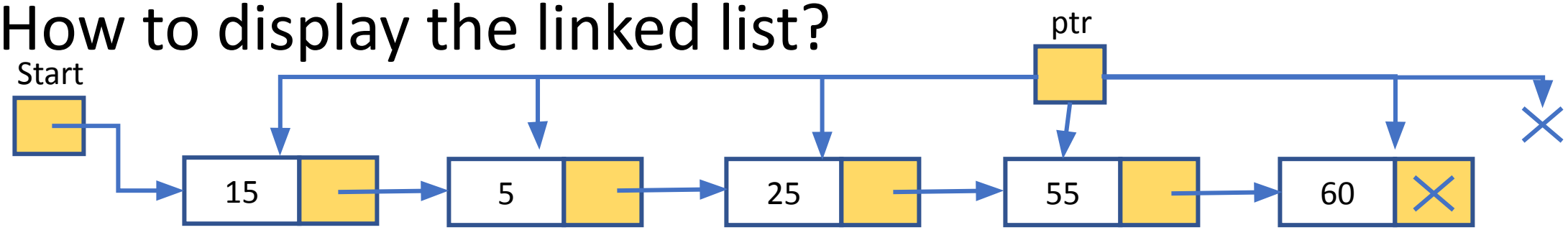
Step 4.2: ptr=ptr->link;
printf("\nEnter the data:");
scanf("%d",&ptr->data);

Step 5: ptr->link=NULL;

Creation of Dynamic Linked List:

```
nd * start=NULL;
void create_linkedlist()
{
  nd *ptr;
  char ch;
  ptr=(nd*)malloc(sizeof(nd));
  printf("\nEnter the data:");
  scanf("%d",&ptr->data);
  start=ptr;
  printf("\n Do you want to continue?(y/n)");
  fflush(stdin);
  ch=getchar();
  while(ch=='y')
  {
    ptr->link=(nd*)malloc(sizeof(nd));
    ptr=ptr->link;
    printf("\nEnter the data:");
    scanf("%d",&ptr->data);
    printf("\n Do you want to continue?(y/n)");
    fflush(stdin);
    ch=getchar();
  }
ptr->link=NULL;
} // end of function
```

How to display the linked list?




```
void display()
{
  nd *ptr;
  ptr=start;
  printf("\nStart");
  while(ptr)
   {
     printf("->%d",ptr->data);
     ptr=ptr->link;
   }
  } // end of display function
```

Output::

Start ->15 ->5 ->25 ->55->60

```
void main()
{
  display(start);
}
void display(nd * ptr)
{
   if (ptr)
    {
      printf("->%d",ptr->data);
      ptr=ptr->link;
      display(ptr);
    }
 } //end of function
```

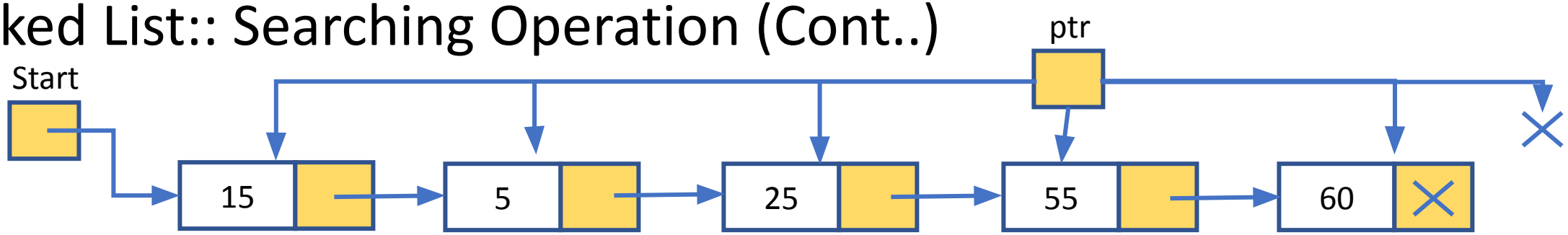
Linked List:: Searching Operation

- Linear searching and Binary Searching
- Limitations of Binary Searching Algorithms:

1. The list must be sorted.
2. The data structure, where the list is stored, should have the facility to direct access the element without accessing the other elements.

- The second limitation fails in case of linked list.
- So, we can not apply binary search algorithm even if the list is sorted in the linked list.
- Therefore, we can implement the only linear searching algorithm on the linked list.

Linked List:: Searching Operation (Cont..)




```
void main()
{
  nd *temp;
  int x;
  printf("\n Enter the searching element:");
  scanf("%d",&x);
  temp=search(x)
  if (temp)
   printf("\n Search is successful");
 else
   printf("\n Search is unsuccessful");
 }
```

```
nd *search(int x)
{
  nd *ptr;
  ptr=start;
  while ((ptr)&&(ptr->data!=x))
    ptr=ptr->link;
  return(ptr)
  } // end of search function
```

Input/Output::

Enter the searching element:
x=25

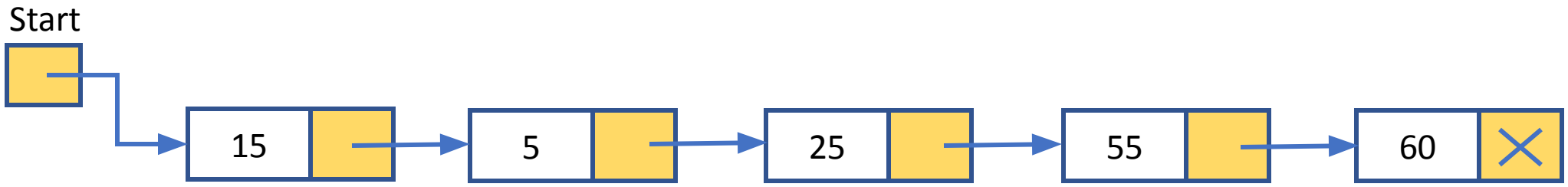
Search is successful

Enter the searching element:
x=85

Search is unsuccessful

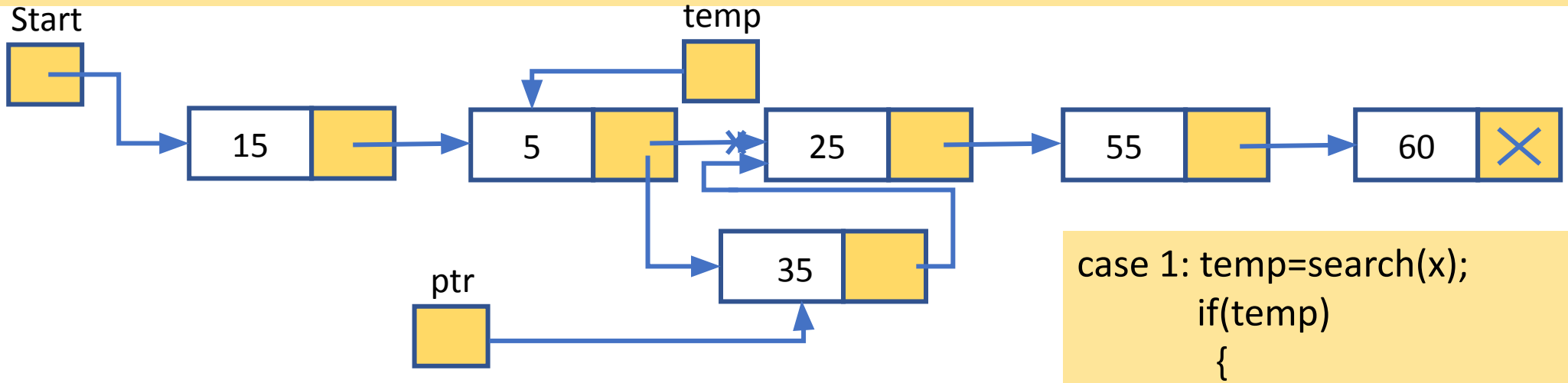
Linked List:: Insertion Operation

- Different cases of Insertion operations:
  1. After a specified element
  2. Before a specified element
  3. At a particular given position



Linked List:: Insertion Operation (Cont..)

Case 1: After a specified element



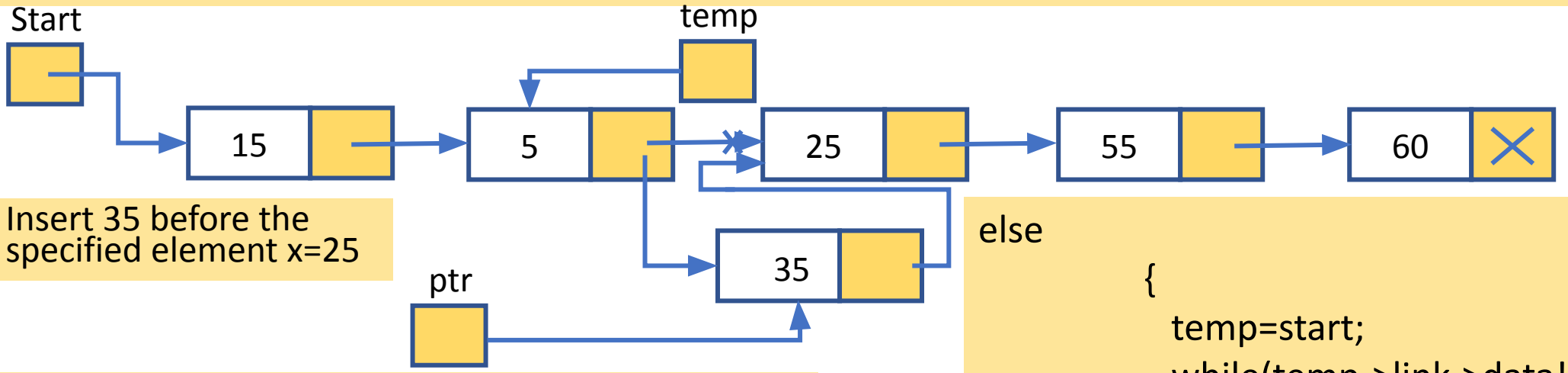
Insert 35 after the specified element x=5

```
temp=search(5);
ptr->link=temp->link;
temp->link=ptr;
```

```
case 1: temp=search(x);
          if(temp)
           {
           ptr=(nd*)malloc(sizeof(nd));
           printf("\nEnter the data");
           scanf("%d",&ptr->data);
           ptr->link=temp->link;
           temp->link=ptr;
           }
          else
            printf("\n Specified element is
not present in the list");
           break
```

Linked List:: Insertion Operation (Cont..)

Case 2: Before a specified element



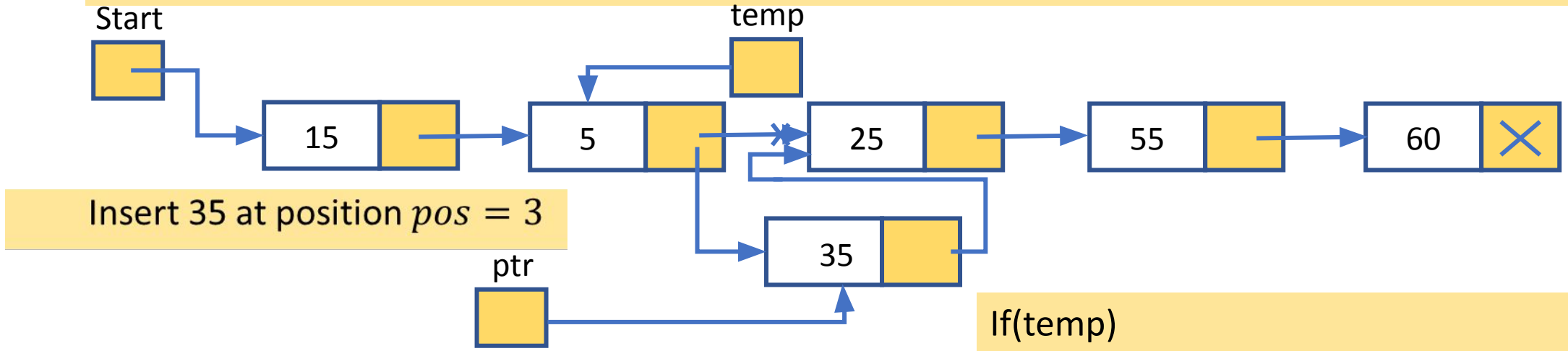
Insert 35 before the specified element x=25

```
case 2: ptr=(nd*)malloc(sizeof(nd));
          printf("\nEnter the data");
          scanf("%d",&ptr->data);
         if(start->data==x )
            {
               ptr->link=start;
               start=ptr;
             }
```

```
else
            {
              temp=start;
              while(temp->link->data!=x)
                 temp=temp->link;
              if(temp)
               {
              ptr->link=temp->link;
              temp->link=ptr;
               }
             else
printf("\n Specified element is not present in the list");
             }
              break;
```

Linked List:: Insertion Operation (Cont..)

Case 3: At a particular given position



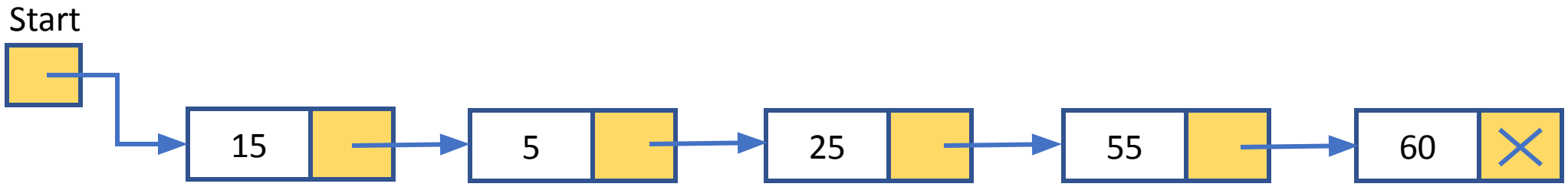
Insert 35 at position $pos = 3$

```
case 3: ptr=(nd*)malloc(sizeof(nd));
        printf("\nEnter the data");
        scanf("%d",&ptr->data);
        temp=start; count=1;
        while((count!=(pos-1))&&(temp))
              {
                temp=temp->link;
                 count++; }
```

```
If(temp)
     {
       ptr->link=temp->link;
       temp->link=ptr;
      }
  else
     printf("\n Invalid Position supplied");
break;
```

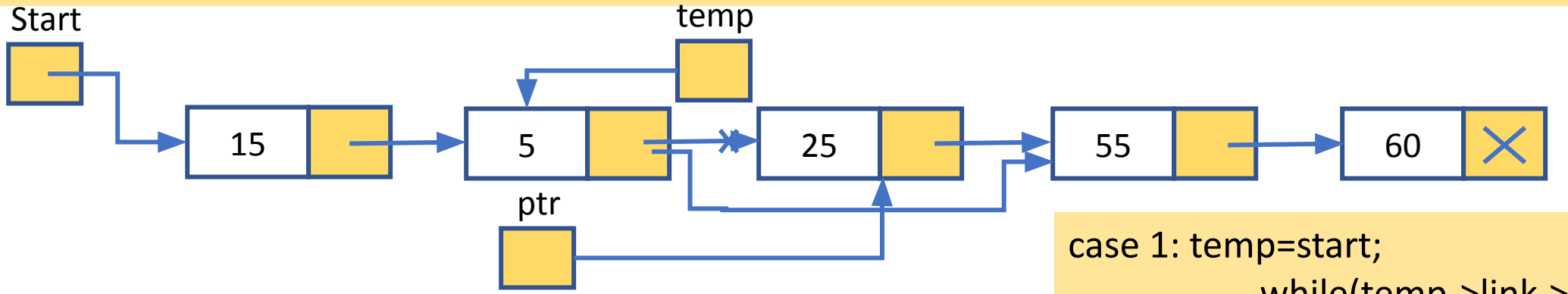
Linked List:: Deletion Operation

- Different cases of Deletion operations:
  1. Delete a specified element
  2. Delete an element whose position is given



Linked List:: Deletion Operation (Cont..)

Case 1: Delete a specified element



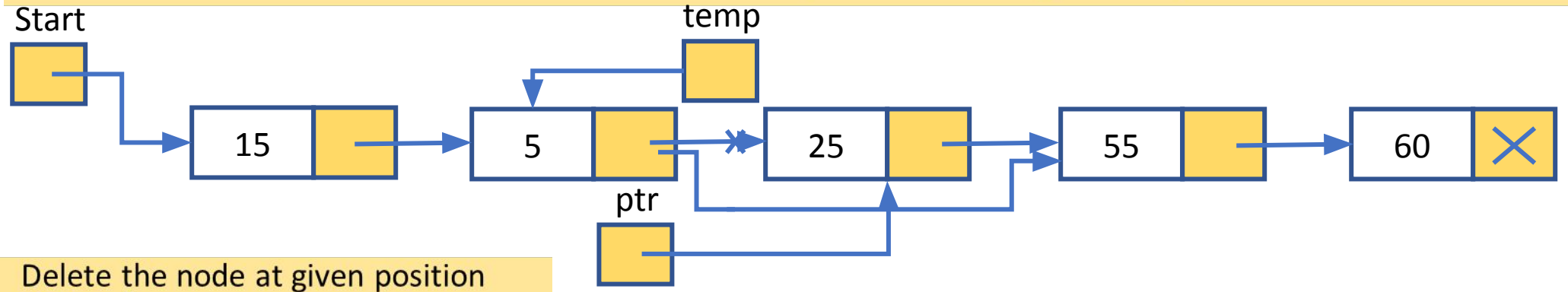
Delete the specified element 25

```
temp->link=temp->link->link;
ptr=temp->link;
free(ptr); // for deallocating the node (25)
```

```
case 1: temp=start;
                while(temp->link->data!=x)
                   temp=temp->link;
                if(temp)
                 {
                   ptr=temp->link;
                   temp->link=ptr->link;
                   free(ptr);
                 }
                else
                   printf("\n Specified element is
                              not present in the list");
              break;
```

Linked List:: Deletion Operation (Cont..)

- Case 2: Delete an element at a particular given position, pos

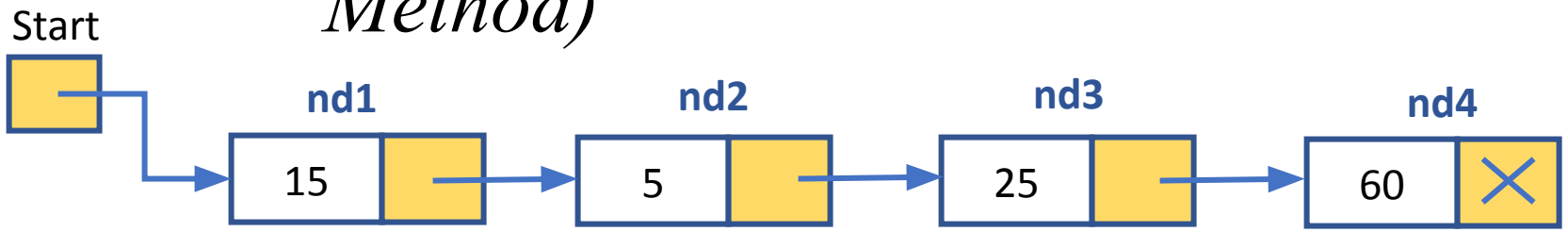


Delete the node at given position $pos = 3$

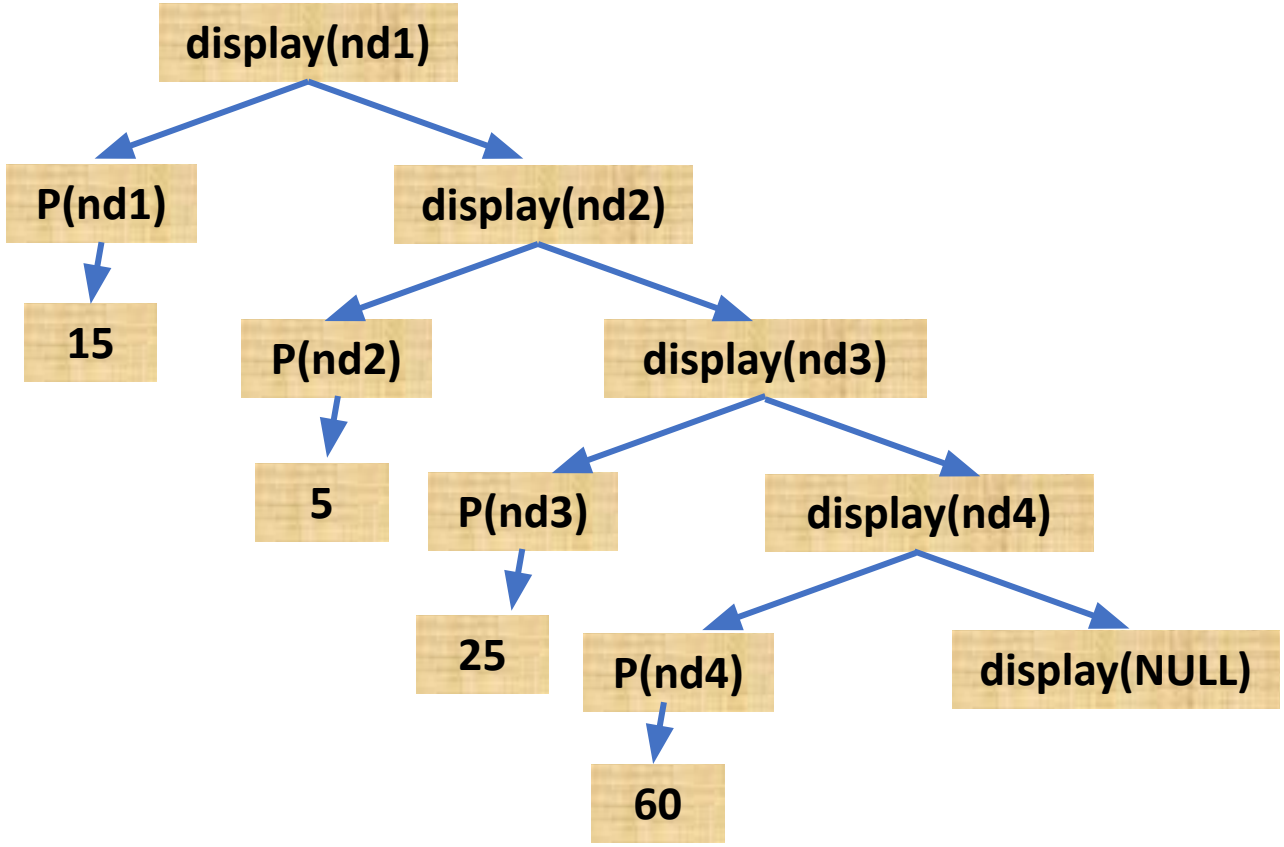
```
case 3: printf("\n Enter the position");
        scanf("%d",&pos);
        temp=start; count=1;
        while((count!=(pos-1))&&(temp))
            {
              temp=temp->link;
               count++; }
```

```
If(temp)
    {
      ptr=temp->link;
      temp->link=ptr->link;
      free(ptr);
    }
  else
     printf("\n Invalid Position supplied");
break;
```

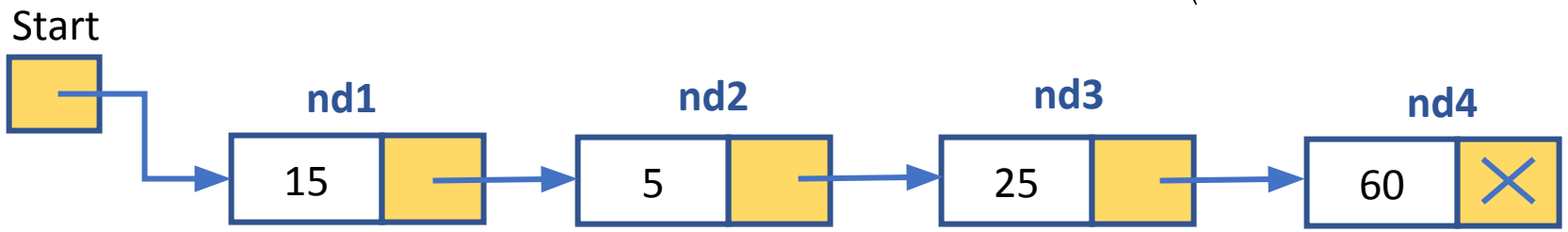
Linked List:: Reverse Print *(Recursive Method)*



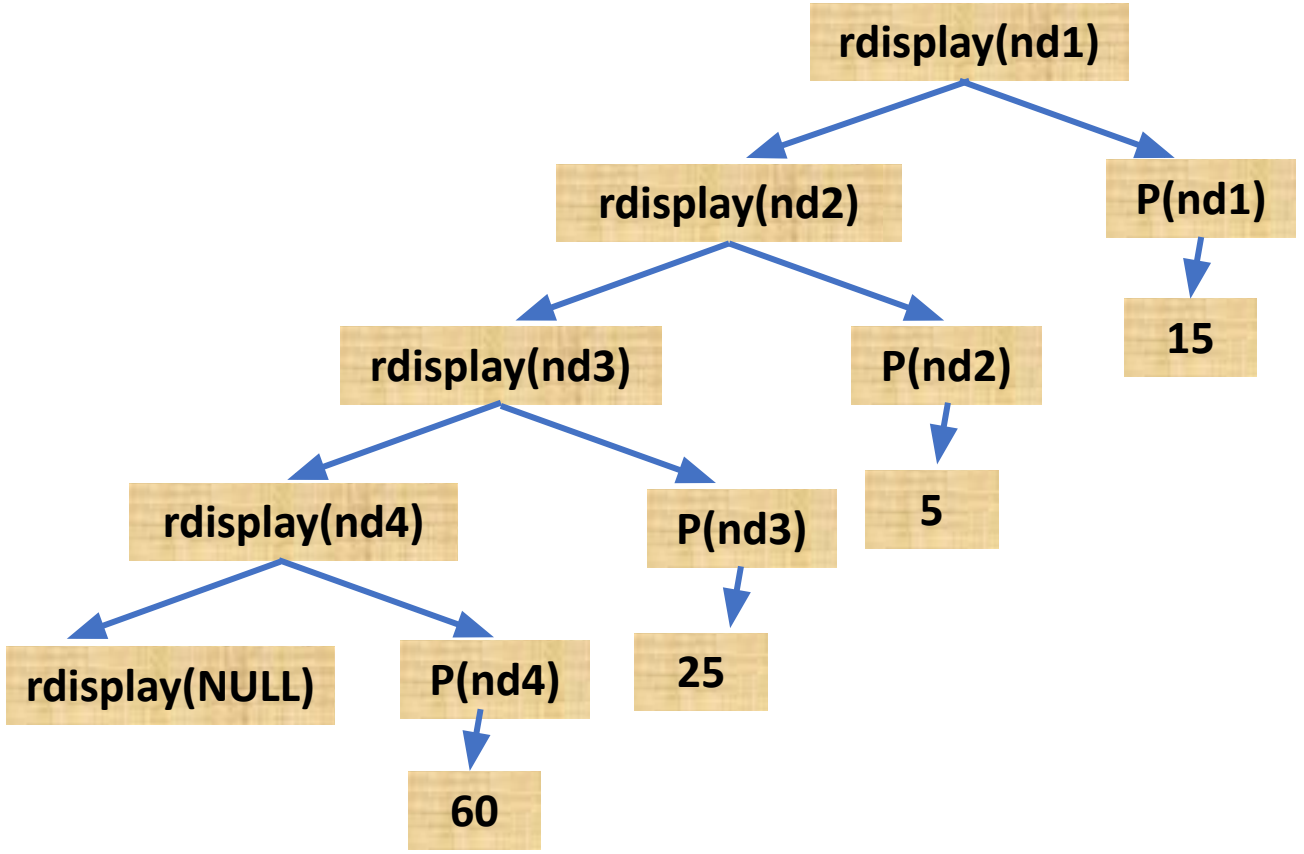

```
void display(nd *ptr)
{
  if(ptr)
    {
       printf("%d",ptr->data);
       display(ptr->link);
    }
  } // end of display function
```



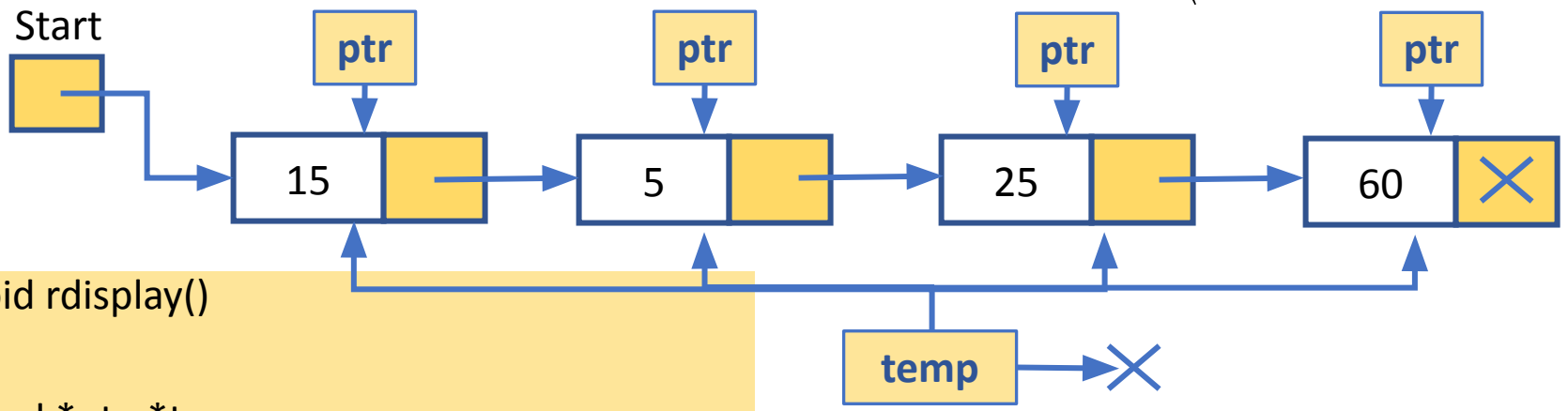
Linked List:: Reverse Print *(Recursive Method)*




```
void rdisplay(nd *ptr)
{
  if(ptr)
    {
     rdisplay(ptr->link);
     printf("%d",ptr->data);
    }
  } // end of rdisplay function
```



Linked List:: Reverse Print *(Iterative Method)*



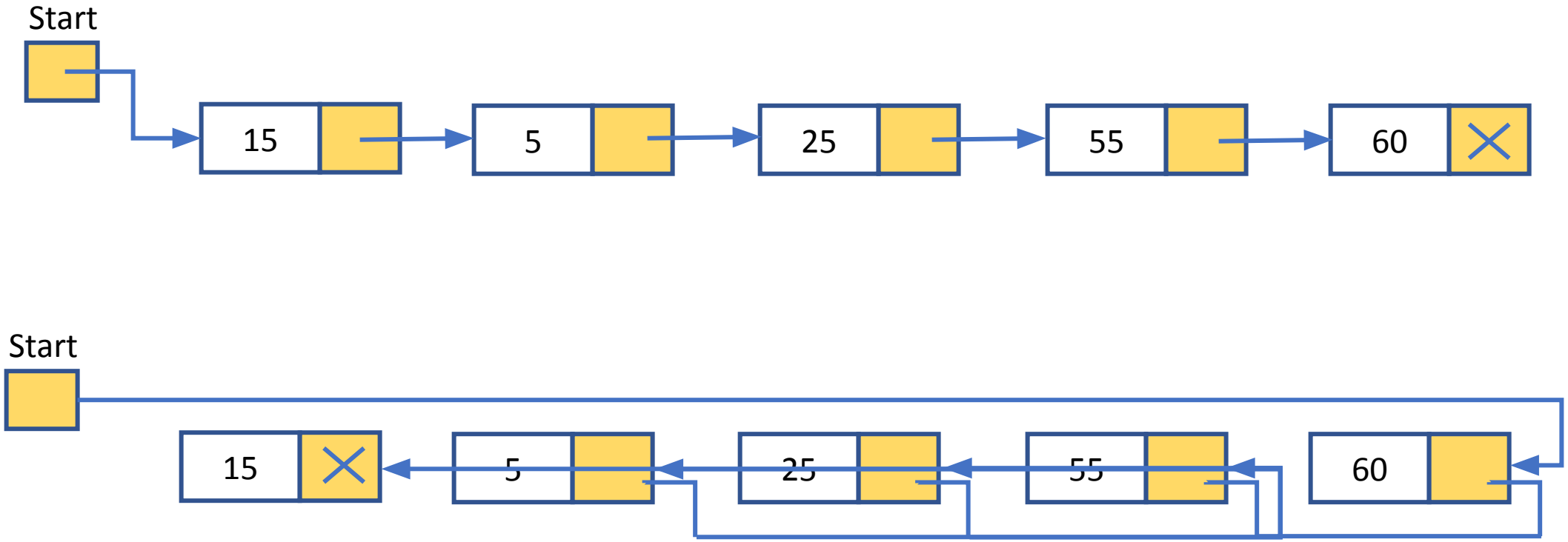

```
void rdisplay()
{
   nd *ptr, *temp;
   temp=NULL;
   while(temp!=start)
    {
      ptr=start;
       while(ptr->link!=temp)
           ptr=ptr->link;
       printf("%d",ptr->data);
       temp=ptr;
    }
  } // end of function
```

Output:

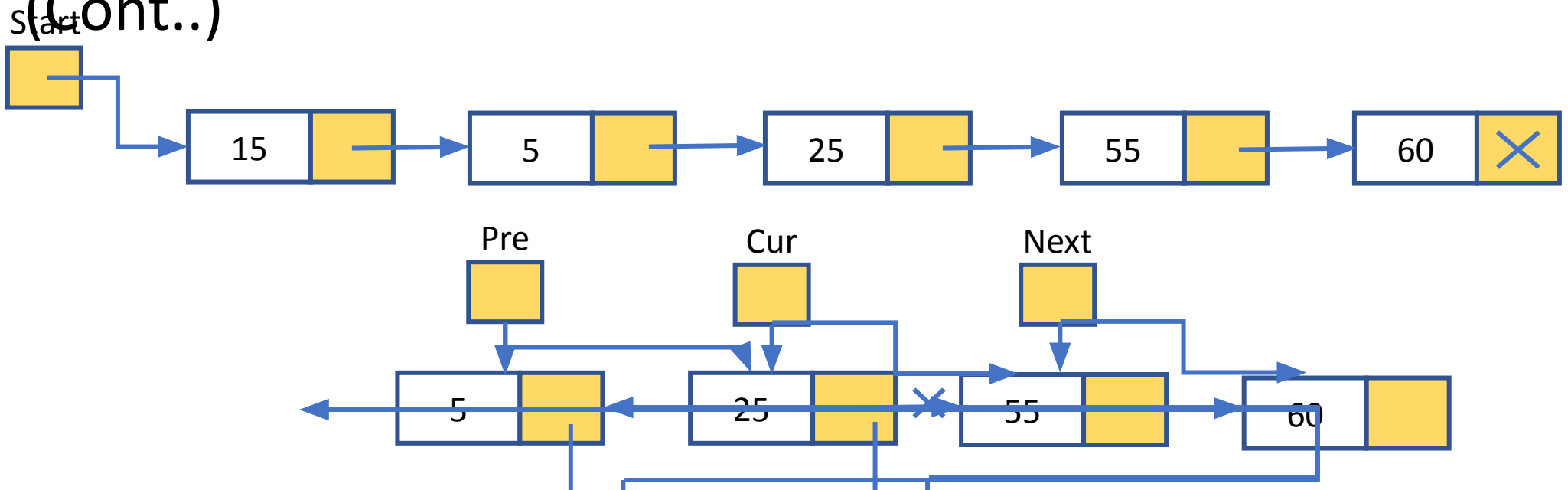
60 25 5 15

Linked List:: Reverse Linked List

How?



Linked List:: Reverse Linked List (Cont..)



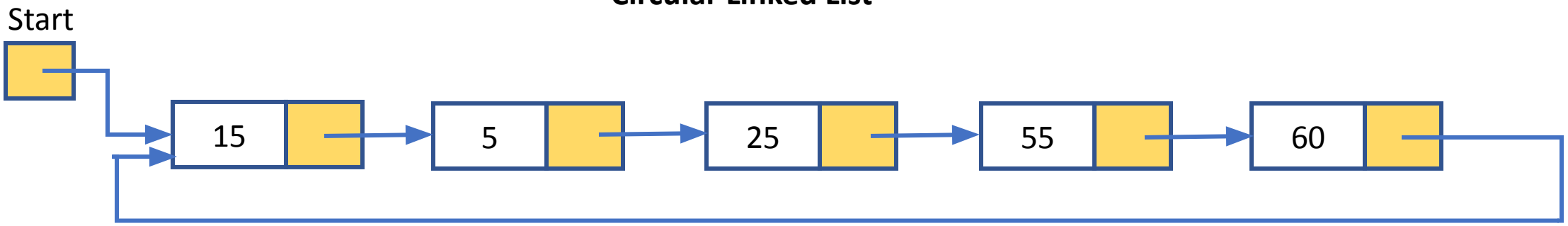

```
Cur->link=Pre;
Pre=Cur;
Cur=Next;
Next=Next->link;
```

We will continue to update the link part of the node pointed by Cur. What will be the stopping criteria?

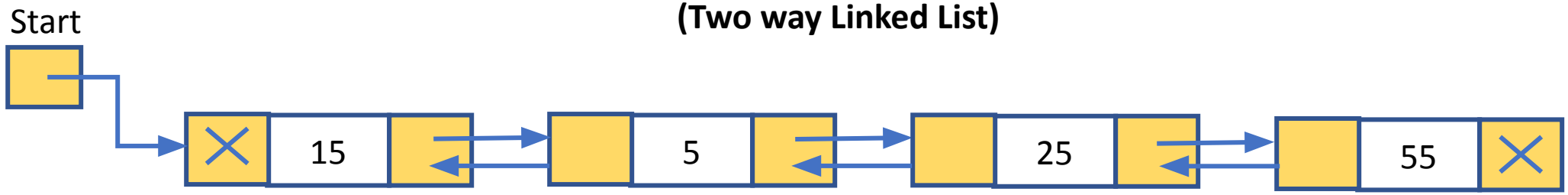
```
while (Cur!=NULL)
```

Circular Linked List & Double Linked List

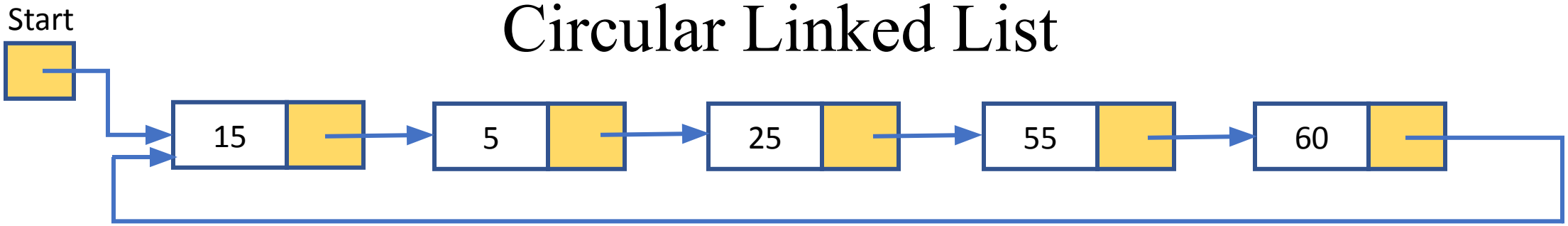
Circular Linked List



Double Linked List
(Two way Linked List)



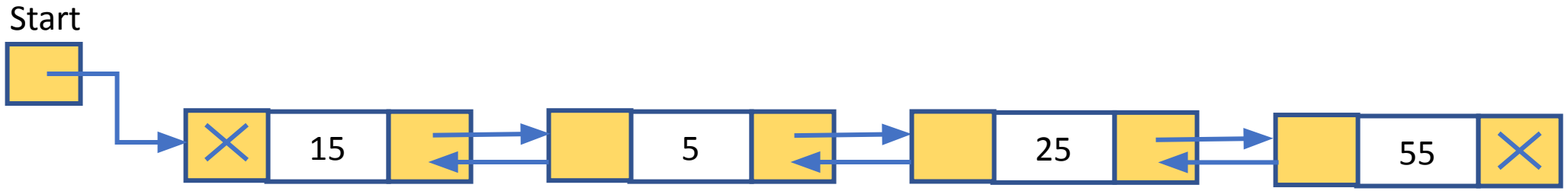
Circular Linked List




```
void display()
{
  nd *ptr;
  ptr=start;
  printf("\nStart");
  do{
      printf("->%d",ptr->data);
      ptr=ptr->link;
     }while(ptr!=start);
  } // end of display function
```

Advantage and Disadvantage of Circular Linked List:

Double Linked List




```
typedef dnode
{
   int data;
   struct dnode * left;
   struct dnode * right;
 } dnd;
```

Advantage and Disadvantage of Double Linked List: