ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY

DATA STRUCTURES

UNIT II : Linked List

Branch: I-II IT

T. Srinivasulu

Dept. of Information Technology

Aditya College of Engineering & Technology

Surampalem.

Contents

Linked List:

- Introduction
- Single linked list
- Representation of Linked list in memory,
- Operations on Single Linked list-Insertion, Deletion, Search and Traversal ,Reversing Single Linked list,
- Applications on Single Linked list- Polynomial Expression Representation - Addition and Multiplication
- Sparse Matrix Representation using Linked List,
- Advantages and Disadvantages of Single Linked list,
- Double Linked list-Insertion, Deletion,
- Circular Linked list-Insertion, Deletion.

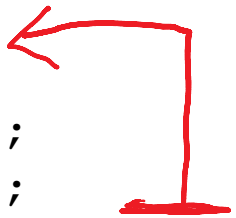
INTRODUCTION

- an array is a linear collection of data elements in which the elements are stored in consecutive memory locations.
- While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store.
- For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that.
- A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.
- Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

Self Referential Structures

- Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.
- structures pointing to the same type of structures are self-referential in nature.

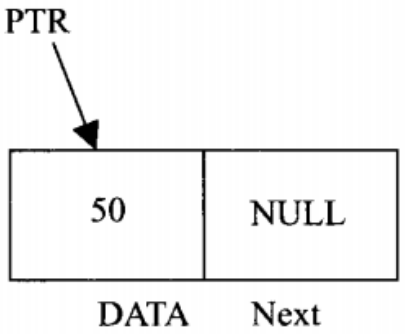
```
struct  node
{
     int data1;
     int data2;
     struct node *link;
}
```



- 'link' is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.
- An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

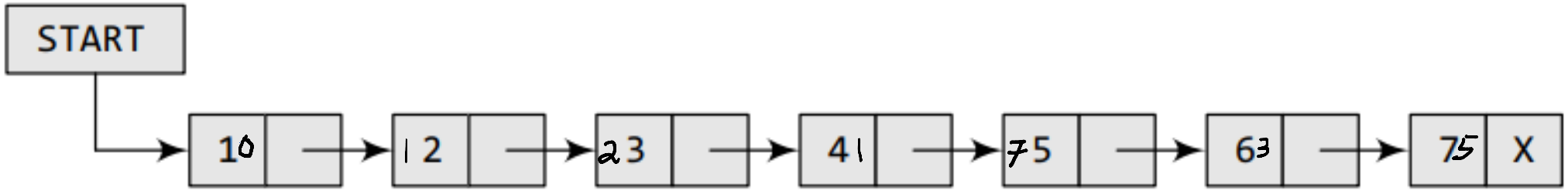
Basic Terminologies

- A linked list, in simple terms, is a linear collection of data elements. These data elements are called **nodes**.
- Each node contains **two parts**, first part contains the information of the element or **value**, and the second part contains the **address on the next node** in the linked list.



PTR ➔ DATA = 50
PTR ➔ Next = NULL

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



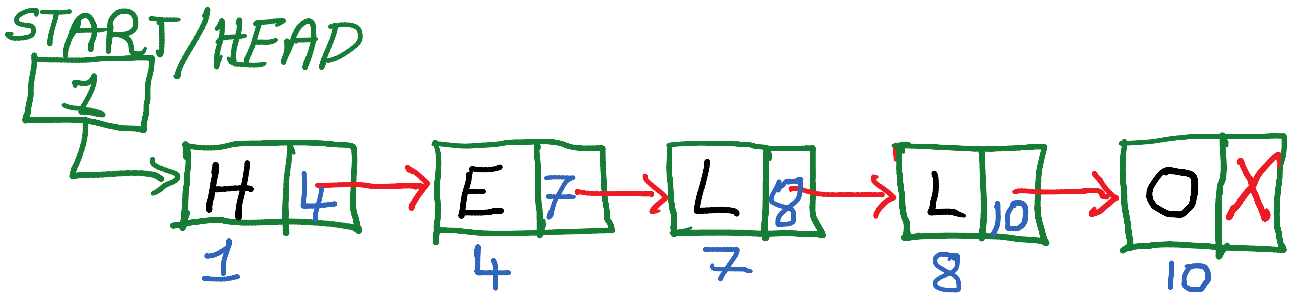
- Linked lists contain a pointer variable START that stores the address of the first node in the list
- The last node will have no next node connected to it, so it will store a special value called NULL.

- Let us see how a linked list is maintained in the **memory.**
- In order to form a linked list, we need a structure called **node** which has two fields, DATA and NEXT.
- DATA will store the information part and NEXT will store the address of the next node in sequence.

START

1

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



- Two linked lists which are simultaneously maintained in the memory

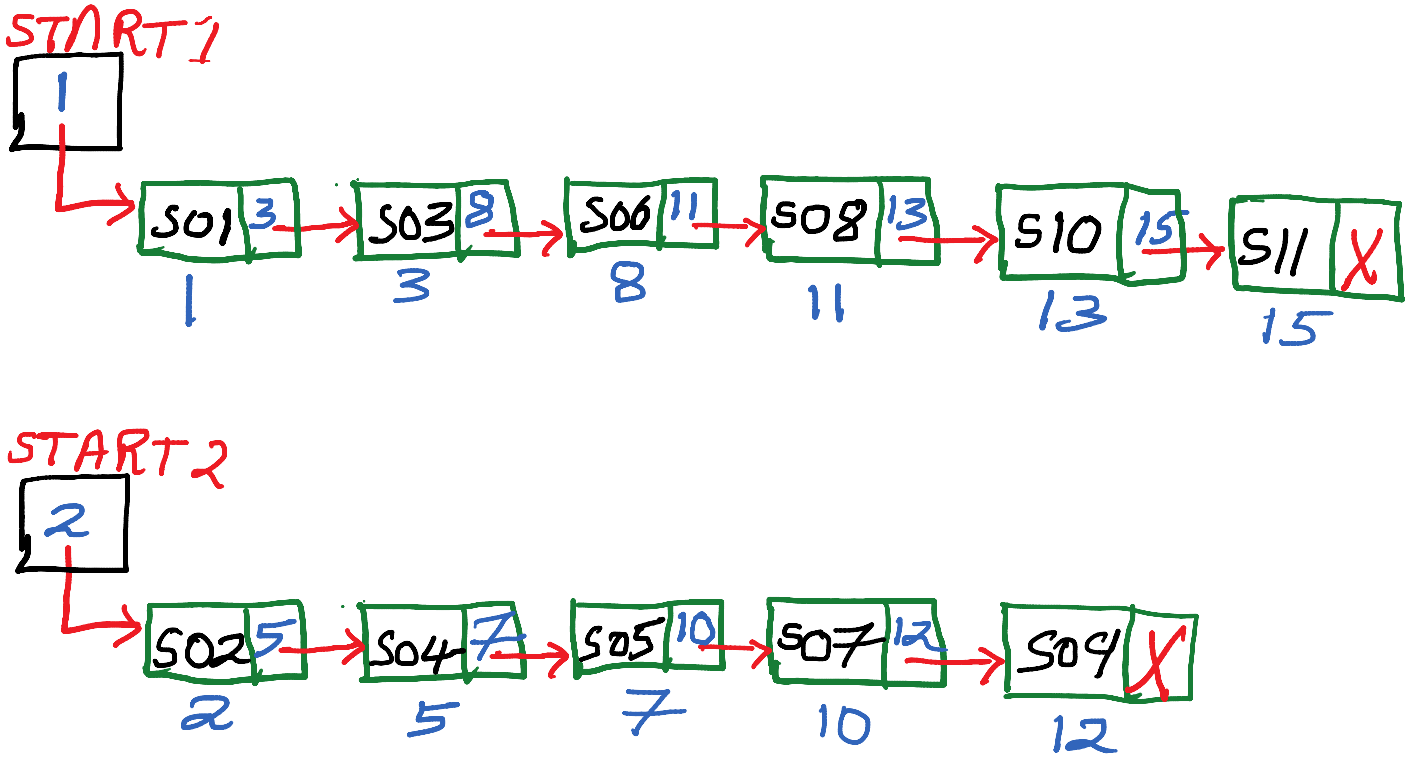
START 1

1 (Biology)

2

START 2
(Computer Science)

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



Linked Lists versus Arrays

- Both arrays and linked lists are a **linear collection** of data elements.
- But unlike an array, a linked list does not store its nodes in **consecutive memory locations**.
- Another point of difference between an array and a linked list is that a linked list **does not allow random access of data**. Nodes in a linked list can be accessed only in a sequential manner.
- But like an array, **insertions** and **deletions** can be done at any point in the list in a **constant time**.
- Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array. For example, if we declare an array as int marks[20], then the array can store a maximum of 20 data elements only. There is no such restriction in case of a linked list.
- Thus, linked lists provide an **efficient way** of storing related data and performing basic operations such as insertion, deletion, and updation of information at the cost of **extra space** required for storing the address of next nodes.

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

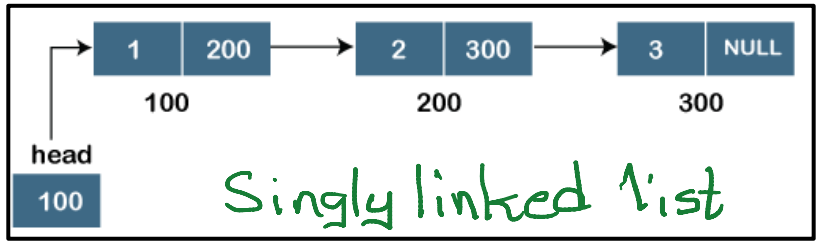
Trade of between linked list and arrays

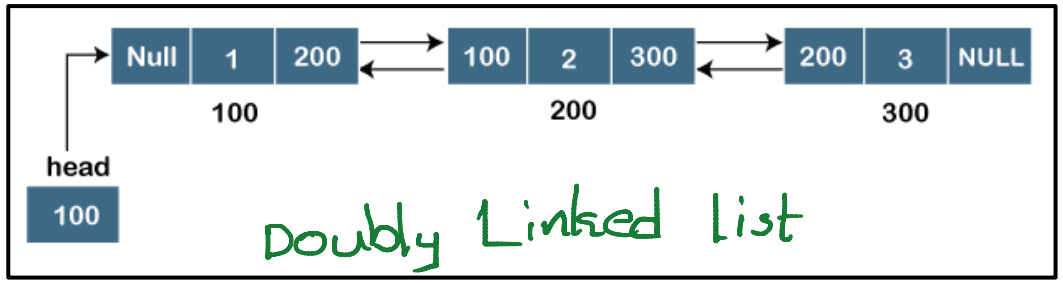
FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

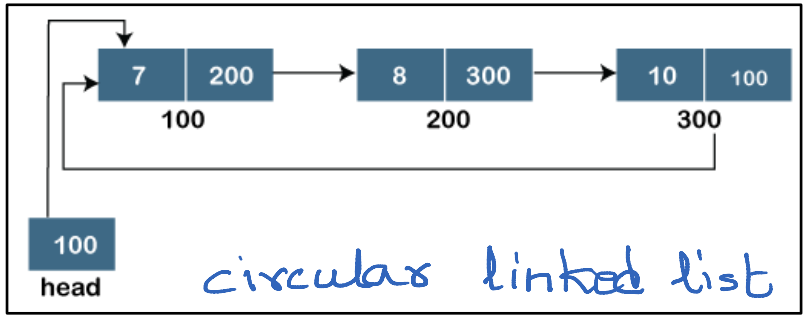
Types of Linked Lists

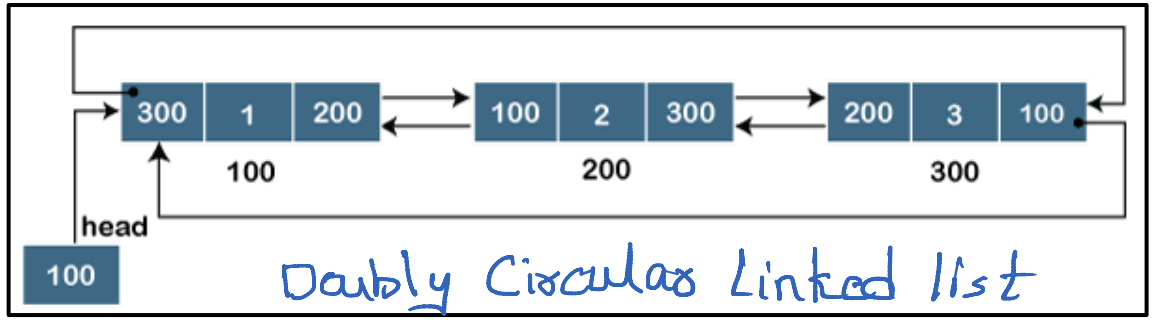
➢ Depending upon the usage of links the linked list is divided into four types, they are

1. **Singly Linked List**: It is the most common. Each node has data and a pointer to the next node.
2. **Doubly Linked List:** We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.
3. **Circular Linked List:** A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.
4. **Doubly Circular linked list:** The doubly circular linked list has the features of both the **circular linked list** and **doubly linked list**.



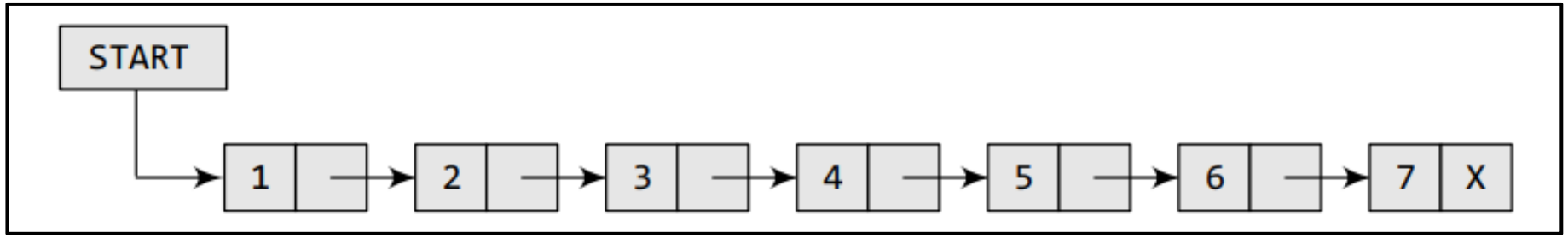


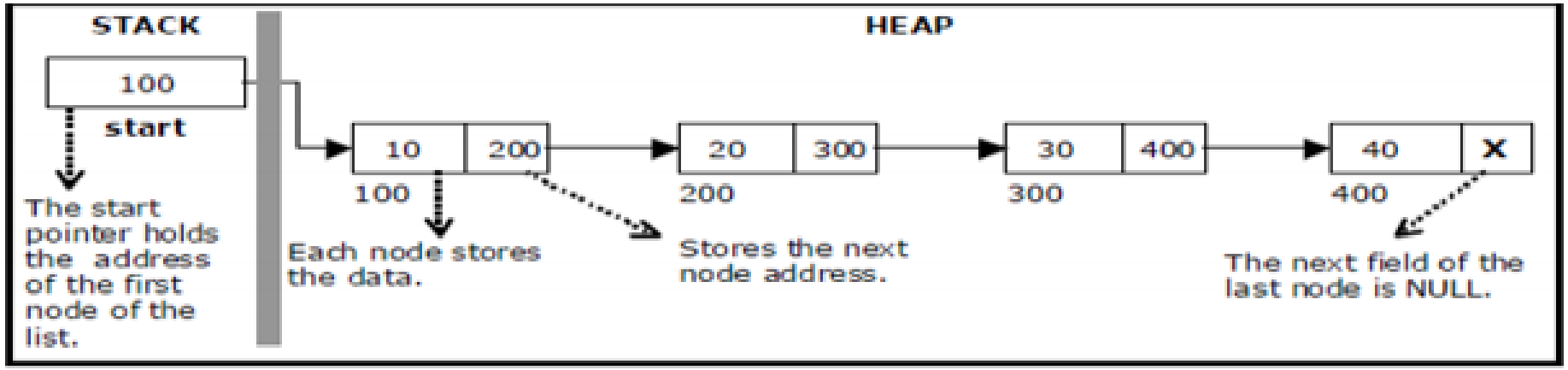




SINGLY LINKED LISTS

- A singly linked list is the simplest type of linked list in which every node contains some **data** and a **pointer** to the next node of the same data type.
- By saying that the node contains a pointer to the next node, we mean that the node stores the **address of the next node** in sequence.
- A singly linked list allows traversal of data only in **one way**.





Implementation of Single Linked List

- Before writing the code to build the above list, we need to create a **start node**, used to create and access other nodes in the linked list.
- The following structure definition will do:
  - ➢Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as **self-referential structure**.
  - ➢ Initialize the start pointer to be NULL.

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

Operations on Single Linked list

➢The following operations are performed on single linked list.

1. **Creating a single linked list**
2. **Traversing a single linked list**
3. **Searching for a value in a single linked list**
4. **Inserting a new node in a single linked list**
5. **Deleting a node from a linked list**
6. **Reversing a single linked list**
7. **Sorting a single linked list**

Creating a single linked list

- Creating a singly linked list starts with **creating a node**. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the **malloc()** function.
- After allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node.
- The following steps are to be followed to create n number of nodes:

➢ Get the **new node** using **malloc().**

➢ If the list is empty, assign new node as start. **start = new node**;

➢ If the list is not empty, follow the steps given below:

➢ The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.

➢ The **start** pointer is made to point the **new node** by assigning the address of the new node.

➢ Repeat the above steps ‘n’ times.

Traversing a single linked list

- Traversing a linked list means **accessing the nodes** of the list in order to perform some processing on them.
- Remember a linked list always contains a pointer variable START which stores the address of the first node of the list.
- End of the list is marked by storing NULL or –1 in the NEXT field of the last node.
- For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR -> DATA
Step 4:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: EXIT
```

Algorithm for traversing a linked list

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:            SET COUNT = COUNT + 1
Step 5:            SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Algorithm to print the number of nodes in a linked list

Code for creating a singly linked list:

```
struct node* createll(struct node* start)
{
        struct node *newnode, *ptr;
        int num;
        printf("\n enter the data or -1 to end: ");
        scanf("%d", & num);
        while(num!= -1)
        {
                newnode = (struct node *)malloc(sizeof (struct node));
                newnode->data=num;
                if(start == NULL)
                {
                        newnode->next=NULL;
                        start=newnode;
                }
                else
                        {
                                ptr = start;
                                while(ptr->next != NULL)
                                        ptr = ptr->next;
                                ptr->next = newnode;
                                newnode->next=NULL;
                        }
                printf("\n enter the data: ");
                scanf("%d",&num);
                }
                return start;
}
```

Searching for a value in a single linked list

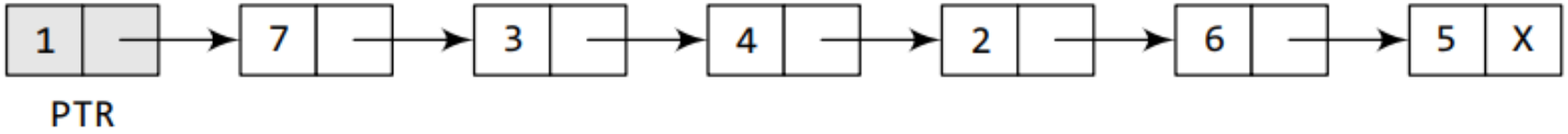
- Searching a linked list means to **find a particular element** in the linked list.
- As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part.
- So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:     IF VAL = PTR -> DATA
                SET POS = PTR
                Go To Step 5
            ELSE
                SET PTR = PTR -> NEXT
            [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

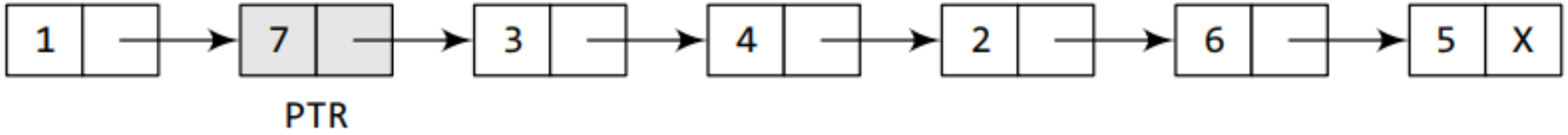
Algorithm to search a linked list

Example:

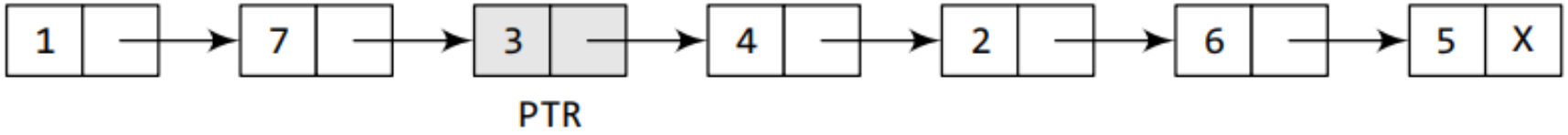
- Consider the linked list shown in below, If we have to search a VAL = 4,



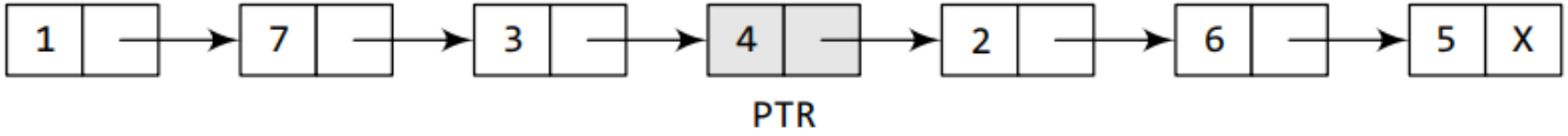
Here PTR –> DATA = 1. Since PTR –> DATA != 4, we move to the next node.



Here PTR –> DATA = 7. Since PTR –> DATA != 4, we move to the next node.



Here PTR –> DATA = 3. Since PTR –> DATA != 4, we move to the next node.



Here PTR –> DATA = 4. Since PTR –> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

Code for Searching a value in a single linked list

```
void search(struct node *start, int val)
{
        int flag = -1;
        struct node *ptr;
        ptr=start;
        while(ptr != NULL && ptr->data != val )
        {
                                flag++;
                ptr=ptr->next;
        }

        if(flag > -1 || ptr->data == val)
                printf("%d is found in the linked list at %d position",val, flag);
        else
                printf("%d is not found in the linked list", val);

        }
```

Inserting a new node in a single linked list

- we will see how a new node is added into an already existing linked list.
- We will take four cases and then see how insertion is done in each case.

  Case 1: The new node is inserted at the beginning.

  Case 2: The new node is inserted at the end.

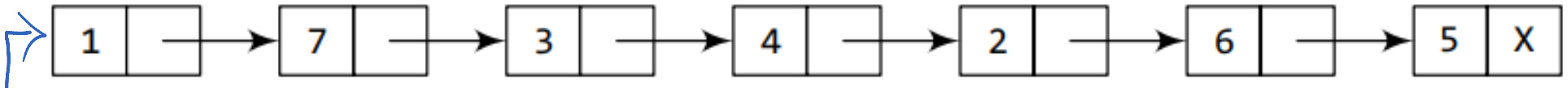
  Case 3: The new node is inserted after a given node.

  Case 4: The new node is inserted before a given node

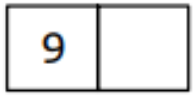
- **Overflow** is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

Inserting a Node at the Beginning of a Linked List

- Consider the linked list shown in below Fig. Suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.



Allocate memory for the new node and initialize its DATA part to 9.



newnode → DATA = num;

Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.

newnode → next = start

Now make START to point to the first node of the list.

start = newnode

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

checking memory

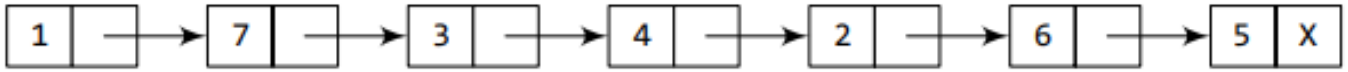
Algorithm to insert a new node at the beginning

Code for Inserting a Node at the Beginning of a Linked List

```
struct node *insert_beg(struct node *start)
{
        struct node *newnode;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        newnode = (struct node *)malloc(sizeof(struct node));
        newnode -> data = num;
        newnode -> next = start;
        start = newnode;
        return start;
}
```

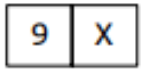
Inserting a Node at the End of a Linked List

- Consider the linked list shown in below Fig. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

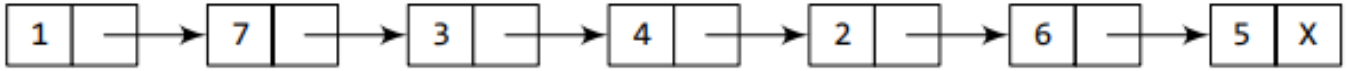


START

Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

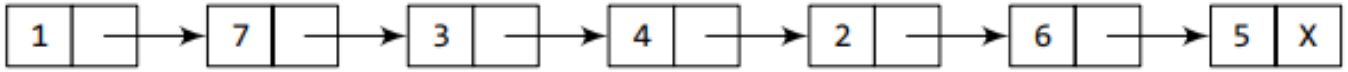


Take a pointer variable PTR which points to START.



START, PTR

Move PTR so that it points to the last node of the list.



START

PTR

Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

START

PTR

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

checking memory

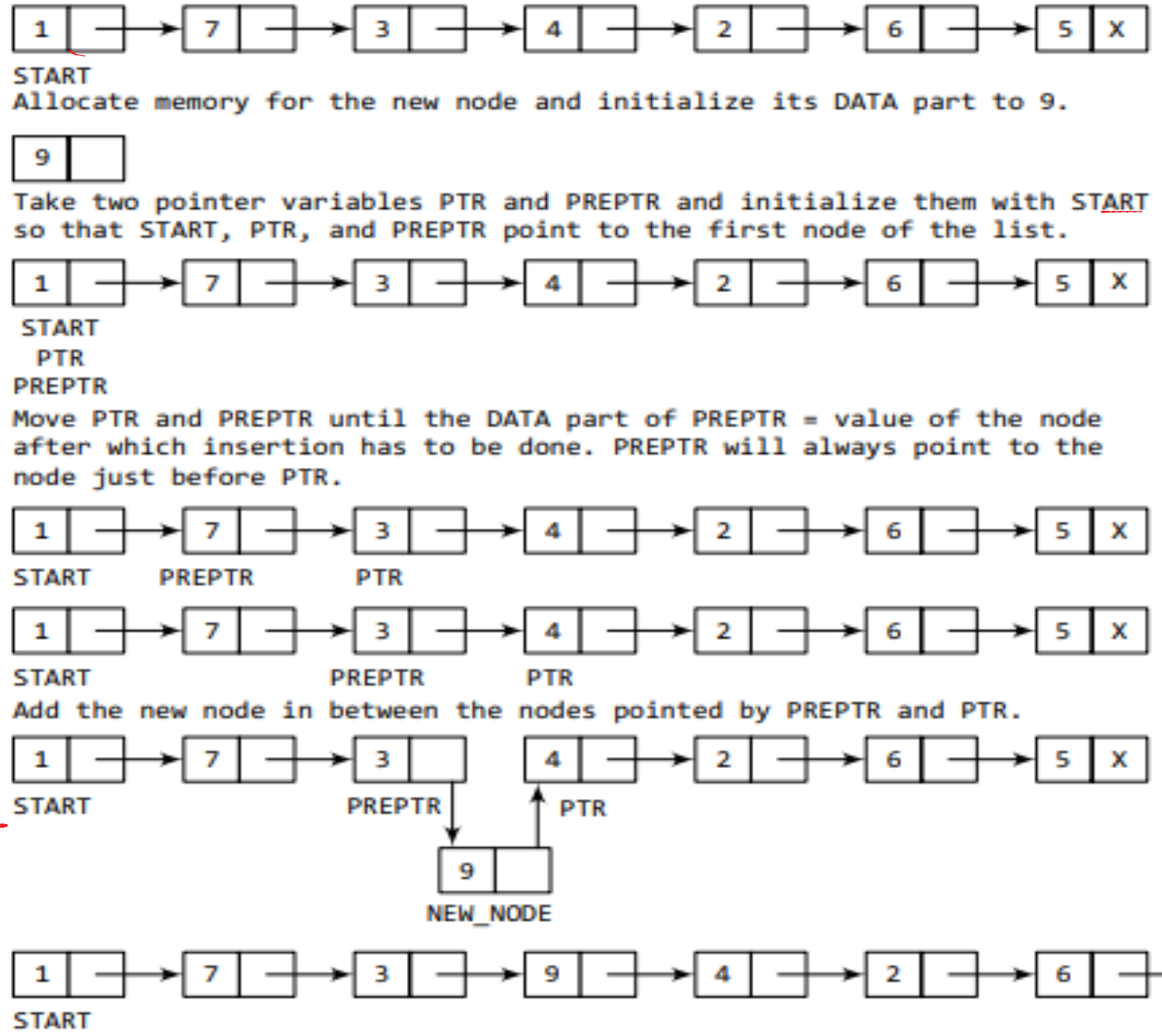
Algorithm to insert a new node at the end

Code Inserting a Node at the End of a Linked List

```
struct node *insert_end(struct node *start)
{
        struct node *ptr, *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        new_node -> next = NULL;
        ptr = start;
        while(ptr -> next != NULL)
                ptr = ptr -> next;
        ptr -> next = new_node;
        return start;
}
```

Inserting a Node after a given node of a Linked List

- Consider the linked list shown in below Fig. Suppose we want to add a new node with value 9 after the node containing data 3.




```
while(preptr->data != 3)
{
  preptr = ptr;
  ptr = ptr->next;
}
preptr->next = newnode
newnode->next = ptr.
```

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
         != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

checking memory

START

Algorithm to insert a new node after a node that has value NUM

Code for Inserting a Node after a given node of a Linked List

```
struct node *insert_after(struct node *start)
{
struct node *new_node, *ptr, *preptr;
int num, val;
printf("\n Enter the data : ");
scanf("%d", &num);
printf("\n Enter the value after which the data has to be inserted : ");
scanf("%d", &val);
new_node = (struct node *)malloc(sizeof(struct node));
new_node -> data = num;
ptr = start;
preptr = ptr;
```

```
while(preptr -> data != val)
{
 preptr = ptr;
 ptr = ptr -> next;
}
preptr -> next=new_node;
new_node -> next = ptr;
return start;
}
```

Deleting a Node from a Linked List

- we will discuss how a node is deleted from an already existing linked list.
- We will consider three cases and then see how deletion is done in each case.

ptr = s1

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted

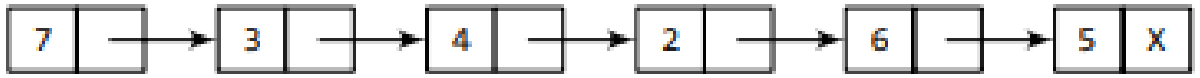
- **Underflow** is a condition that occurs when we try to delete a node from a linked list that is empty.
- This happens when START = NULL or when there are no more nodes to delete.
- Note that when we delete a node from a linked list, we actually have to **free the memory** occupied by that node.

Deleting the First Node from a Linked List

- Consider the linked list in below Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

START

Make START to point to the next node in sequence.



START

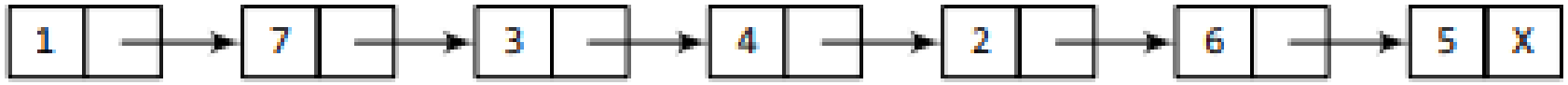
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Algorithm to delete the first node

```
struct node *delete_beg(struct node *start)
{
struct node *ptr;
ptr = start;
start = start -> next;
free(ptr);
return start;
}
```

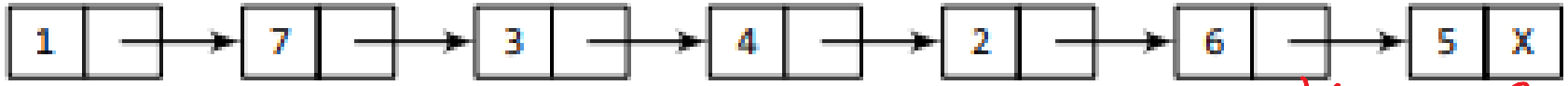
Deleting the Last Node from a Linked List

- Consider the linked list shown in below Fig. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



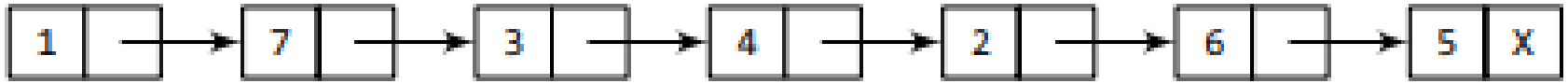
START

Take pointer variables PTR and PREPTR which initially point to START.

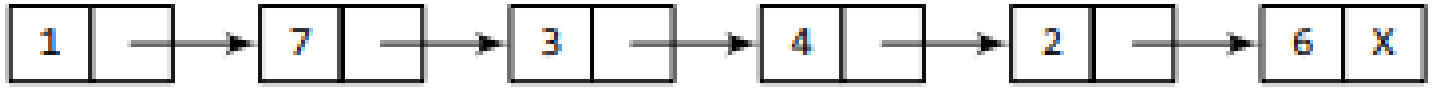


START
PREPTR
PTR

Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.



Set the NEXT part of PREPTR node to NULL.



preptr -> next = NULL
free (ptr);

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

checking nodes available or not

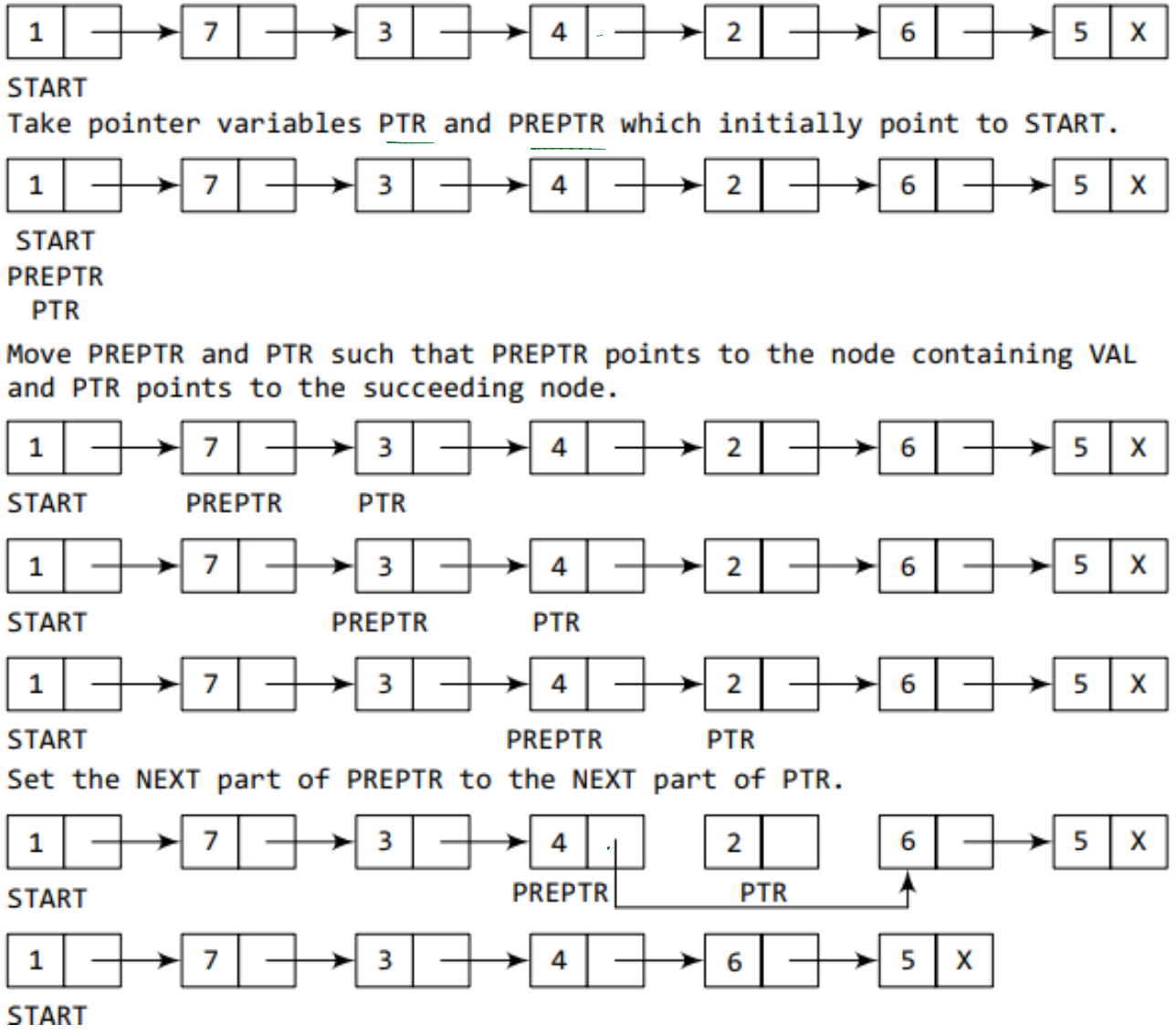
Algorithm to delete the last node

Code to Delete the Last Node from a Linked List

```
struct node *delete_end(struct node *start)
{
    struct node *ptr, *preptr;
    ptr = start;
    while(ptr -> next != NULL)
    {
        preptr = ptr;
        ptr = ptr -> next;
    }
    preptr -> next = NULL;
    free(ptr);
    return start;
}
```

Deleting the Node After a Given Node in a Linked List

- Consider the linked list shown in below Fig. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



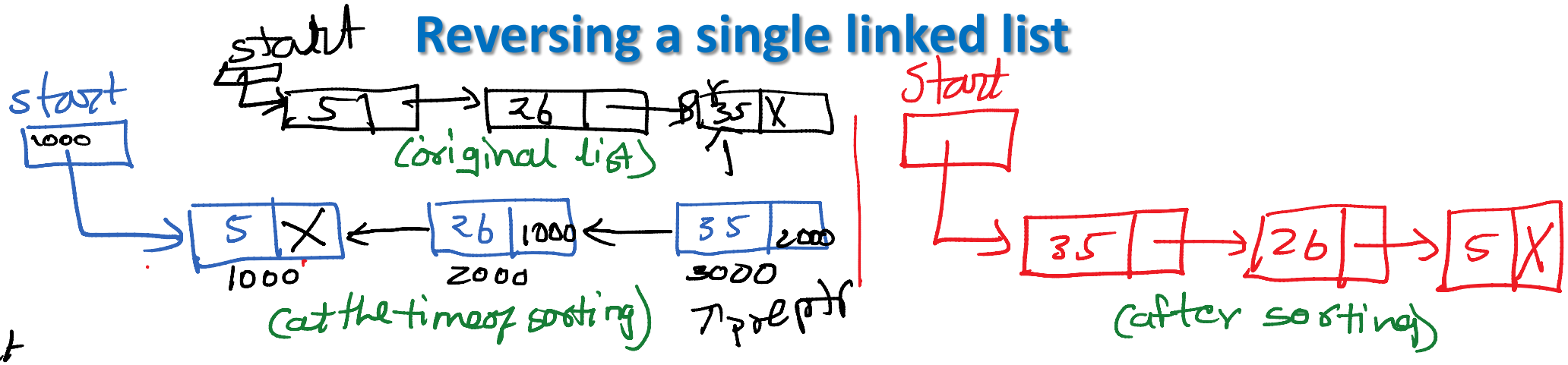
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Algorithm to delete the node after a given node

Code for Deleting the Node After a Given Node in a Linked List

```
struct node *delete_after(struct node *start)
{
        struct node *ptr, *preptr;
        int val;
        printf("\n Enter the value after which the node has to deleted : ");
        scanf("%d", &val);
        ptr = start;
        preptr = ptr;
        while(preptr -> data != val)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next=ptr -> next;
        free(ptr);
        return start;
}
```

Reversing a single linked list

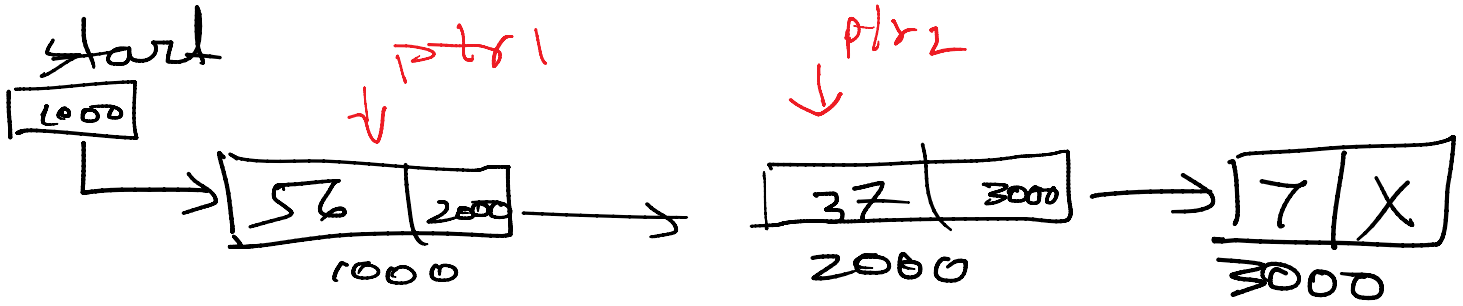



```
ptr = start
preptr = NULL
postptr = NULL

while (ptr != NULL)
{
    postptr = ptr -> next;
    ptr -> next = preptr;
    preptr = ptr
    ptr = postptr;
}
start = preptr;
```

postptr = NULL

ptr = NULL

Sorting a single linked list




```
for(i=0; i<n; i++)
  for(j=i; j<n; j++)
    ptr2
ptr1
```

```
struct node * sort_ll (struct node * start)
{
  struct node *ptr1, ptr2;
  int temp;
  ptr1 = start;
  while (ptr1 -> next != NULL)
  {
    ptr2 = ptr1 -> next;
    while(ptr2 != NULL)
    {
      if (ptr1 -> data > ptr2 -> data)
      {
        temp = ptr1 -> data;
        ptr1 -> data = ptr2 -> data;
        ptr2 -> data = temp;
      }
      ptr2 = ptr2 -> next;
    }
    ptr1 = ptr1 -> next;
  }
  return start
}
```

Time complexity: linked lists

- A linked list can typically only be accessed via its head node. From there you can only **traverse** from node to node until you reach the node you seek. Thus **access is O(n)**.
- **Searching** for a given value in a linked list similarly requires traversing all the elements until you find that value. Thus **search is O(n)**.
- **Inserting** into a linked list requires re-pointing the previous node (the node before the insertion point) to the inserted node, and pointing the newly-inserted node to the next node. Thus **insertion is O(1)**.
- **Deleting** from a linked list requires re-pointing the previous node (the node before the deleted node) to the next node (the node after the deleted node). Thus **deletion is O(1)**.

Applications on Single Linked list

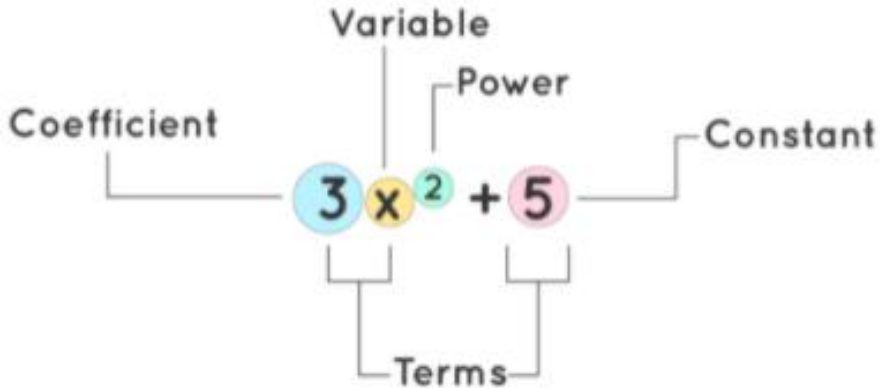
- Manipulation of polynomials and performing various operations like addition, multiplication.
- Representing sparse matrix using linked list.
- Linked list used to implement stacks and queues.
- Linked list used to implement trees and graphs.
- Used to implement dynamic memory allocation.
- Used to maintain directory of names.
- Used to implement hash tables.

Polynomial Expression Representation

- The word polynomial is derived from the Greek words 'poly' means '**many**' and 'nominal' means '**terms**', so altogether it said "many terms". A polynomial can have any number of terms but not infinite.
- Polynomials are algebraic expressions that consist of **variables** and **coefficients**.
- Examples, $4x^2+1x+1$,

  $6x^3+4x^3+3x+1$,

  $6x^4+3x^3+3x^2+2x+1$



- A polynomial equation having one variable which has the largest exponent is called a degree of the polynomial.
- The degree of the polynomial 6s4+ 3x2+ 5x +19 is
- The terms of polynomials are the parts of the equation which are generally separated by "+" or "-" signs.

Polynomial	Terms	Degree
$P(x) = x^3-2x^2+3x+4$	x^3, $-2x^2$, 3x and 4	3

Addition of Polynomials:

Example: Add $2x^2 + 6x + 5$ and $3x^2 - 2x - 1$

Start with: $2x^2 + 6x + 5 + 3x^2 - 2x - 1$

Place like terms together: $2x^2+3x^2 + 6x-2x + 5-1$

Which is: $(2+3)x^2 + (6-2)x + (5-1)$

Add the like terms: $5x^2 + 4x + 4$

Multiplication of Polynomials:

– Multiply: $(3x + 2)(4x^2 - 7x + 5)$

Step 1: Distribute each term of the first polynomial to every term of the second polynomial. In this case, we need to distribute the 3x and the 2.	$(3x+2)(4x^2-7x+5) = 12x^3 - 21x^2 + 15x + 8x^2 - 14x + 10$
Step 2: Combine like terms.	$12x^3 - 13x^2 + x + 10$

Polynomial Representation using Arrays

- Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript (index) of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array.
- Example: $P(x) = 4x^3 + 6x^2 + 7x^1 + 9x^0$

9	7	6	4	(coefficients)
0	1	2	3	(exponents)

→ ascending

$P(x) = 5x^6 - 3x^2 + 1$

array size = degree + 1

6 + 1 = 7

checking memory

1	0	-3	0	0	0	5
0	1	2	3	4	5	6

$P(x) = 15x^{97} + 6x^{20} + 7x^3$

1	0 0 0 . 7	0 0 0 0	6	- 0 0 0	15
0	1 2 3	4 19	20	21 96	97

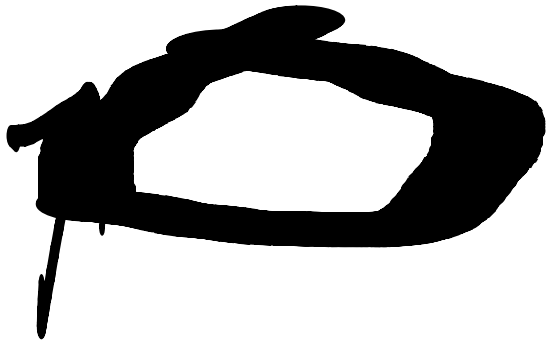
$P(x) = x^{1000} + 1$

- Using array for polynomial operations will consume a **lot of space**.

Polynomial Representation using Linked list

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:



- one is the coefficient
- other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

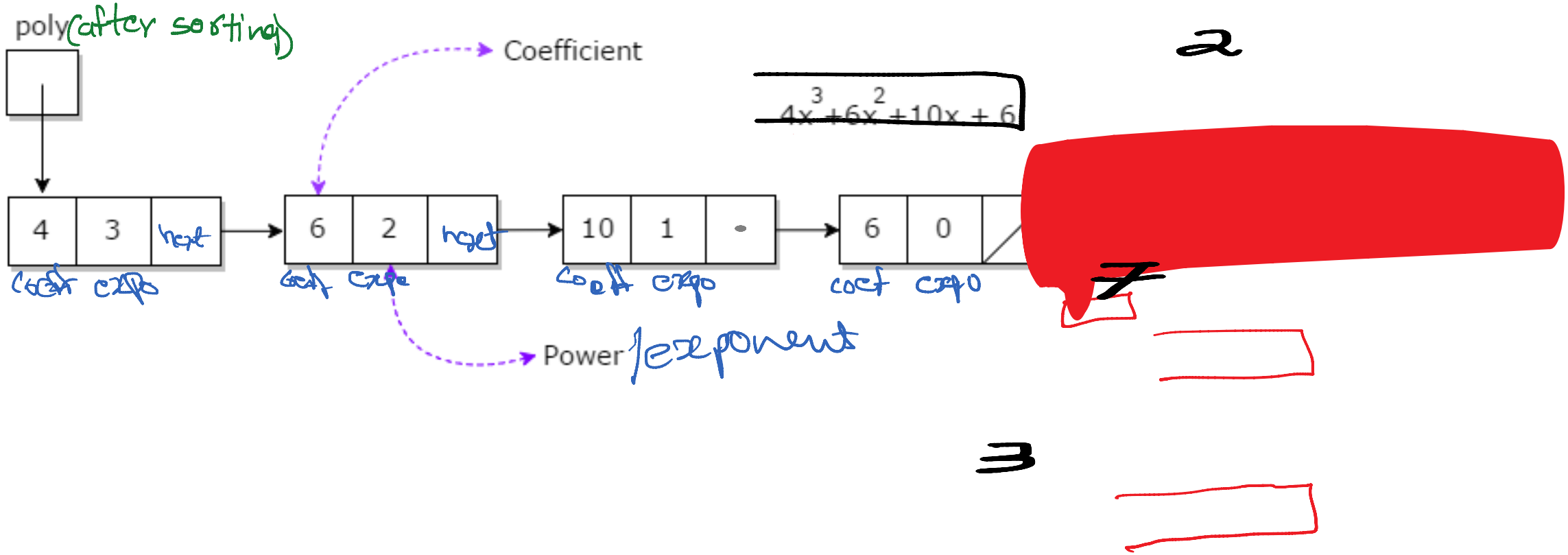
Node Structure:

3 fields

Coefficient	Exponent	Address of next node

Points to keep in Mind while working with Polynomials:

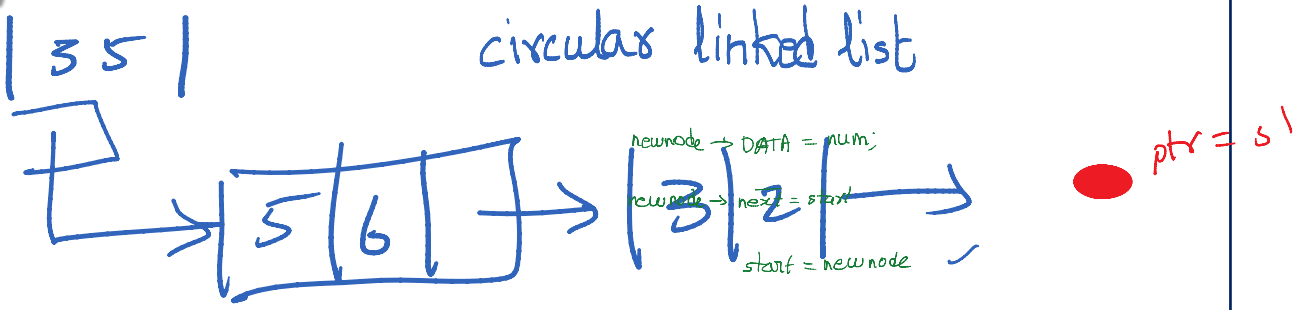
- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Creating a Polynomial linked list

```
struct node *create_poly(struct node *start)
{
struct node *new_node, *ptr;
int n, c,terms,i;
printf("\n enter the number of terms of the polynomial: ");
scanf("%d",&terms);
for(i=0;i<terms;i++)
{
      printf("\n Enter the coefficient number : ");
      scanf("%d", &n);
      printf("\t Enter its exponent : ");
      scanf("%d", &c);
      new_node = (struct node *)malloc(sizeof(struct node));
      new_node -> coeff = n;
      new_node -> expo = c;
      if(start==NULL)
      {
            new_node -> next = NULL;
            start = new_node;
      }
else
      {
            ptr = start;
            while(ptr -> next != NULL)
                  ptr = ptr -> next;
            new_node -> next = NULL;
            ptr -> next = new_node;
      }

}
return start;
}
```

Displaying a Polynomial linked list



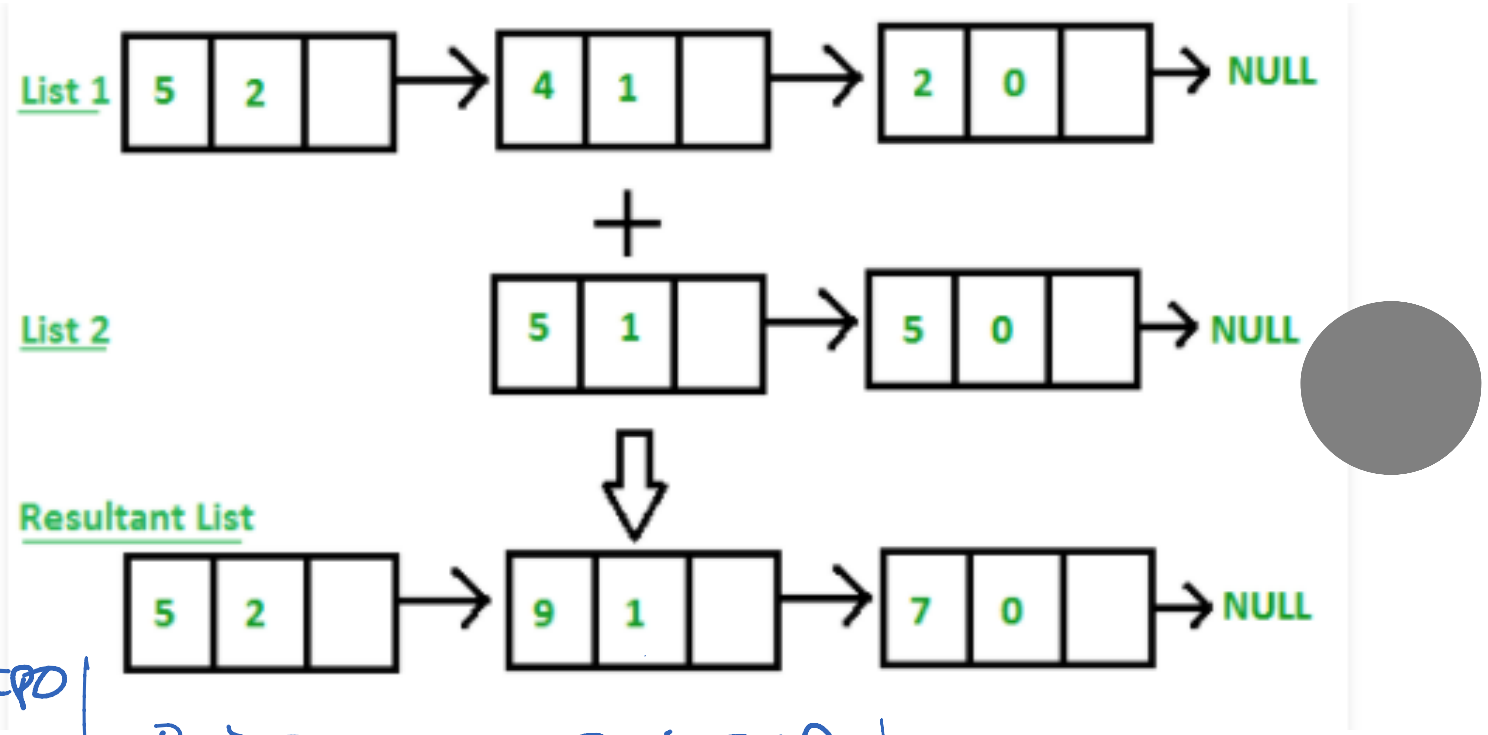

```
void display_poly(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\n%d x^ %d\t", ptr -> coeff, ptr -> expo);
                ptr = ptr -> next;
        }

}
```

Singly linked list
Doubly Linked list

Adding two polynomials using Linked List

List1
$P(x) = 5x^2 + 4x + 2$

List2
$q(x) = 5x + 5$



p->expo == q->expo
add
p -> next
q -> next

p->expo > q->exp
insert p term
p -> next

Procedure for addition of two Polynomials:

Let p and q be the two polynomials represented by linked lists.

Let r represent result list of the addition of two polynomials

1. while p and q are not null, repeat step 2.
2. If exponents of the two terms are equal

then insert the sum of the terms into the sum Polynomial

```
if(p ->expo==q->expo) {
        r->coeff=p->coeff + q->coeff
        r->expo=p->expo
        p=p->next
        q=q->next    }
```

Else if the expo of the polynomial p> expo of polynomial q

Then insert the term from polynomial p into sum polynomial
Advance p

```
        Else if(p ->expo>q->expo) {
                r->coeff = p->coeff
                r->expo=p->expo
                p=p->next    }
```

Else insert the term from q polynomial into sum polynomial Advance q

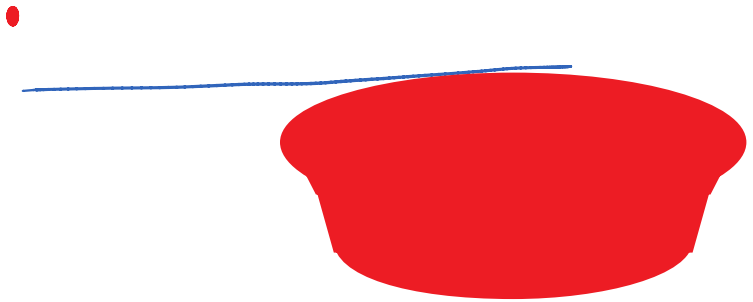
```
        Else {
                r->coeff = q->coeff
                r->expo=q->expo
                q=q->next   }
```

3. copy the remaining terms from the non empty polynomial into the sum polynomial. The third step of the algorithm is to be processed till the end of the polynomials has not been reached.

```
While(p!=NULL) {
        r->coeff=p->coeff
        r->expo=p->expo
        p=p->next     }

While(q!=NULL) {
        r->coeff=q->coeff
        r->expo=q->expo
        q=q->next    }
```

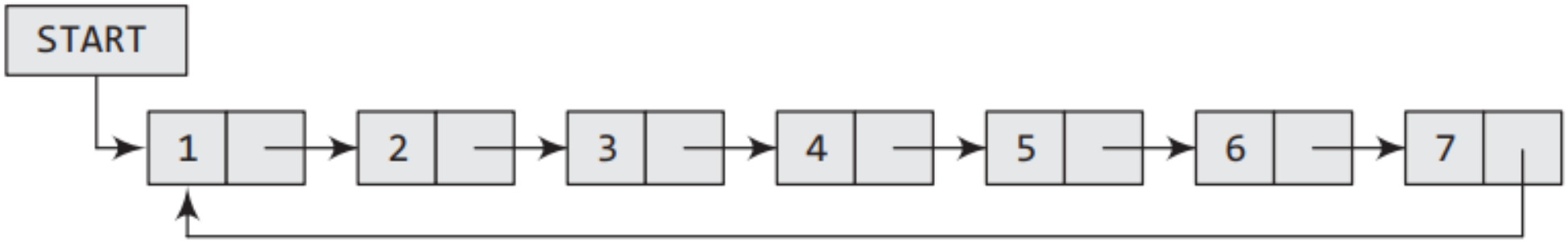
Multiplying two polynomials using Linked List

- Each term of first polynomial is multiplied with the each term of the second polynomial.
- Multiplying each term means multiplying their coefficients and adding their exponents.
- Add this terms to a new polynomial in descending order of the exponents.
- Remove duplicate exponents by adding their coefficients.



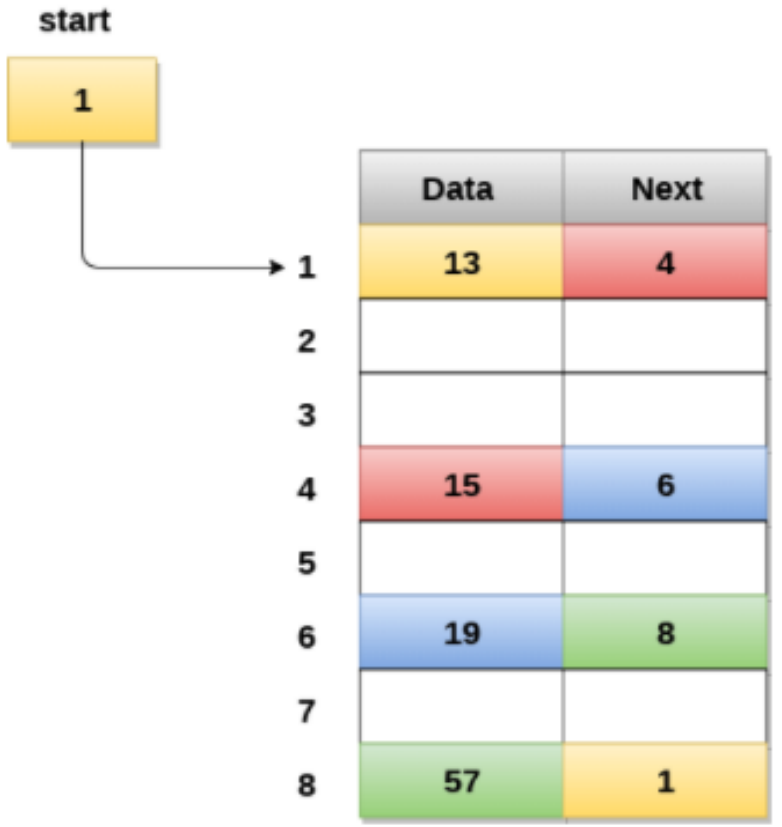
CIRCULAR LINKED LISTS

- Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end.
- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.



Memory Representation of circular linked list:

- In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory.
- The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.
- However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Operations on Circular Linked list

➢The following operations are performed on circular linked list.

1. **Creating a circular linked list**
2. **Inserting a new node in a circular linked list**
3. **Deleting a node from a circular linked list**

displaying circular list

{

printf("%d", ptr -> d

ptr = ptr -> next;

printf("%d", ptr ->

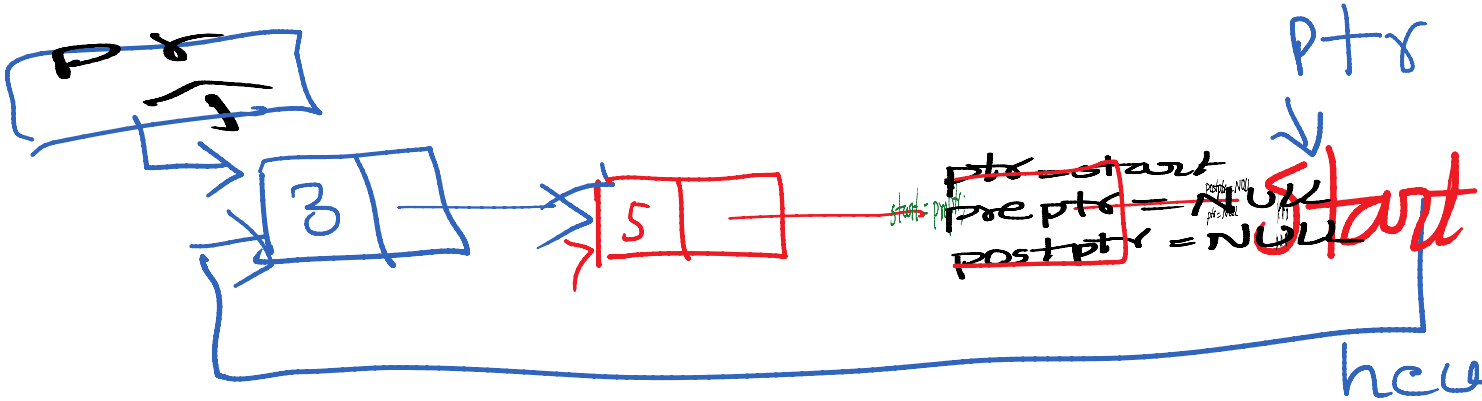
checking whether available or not

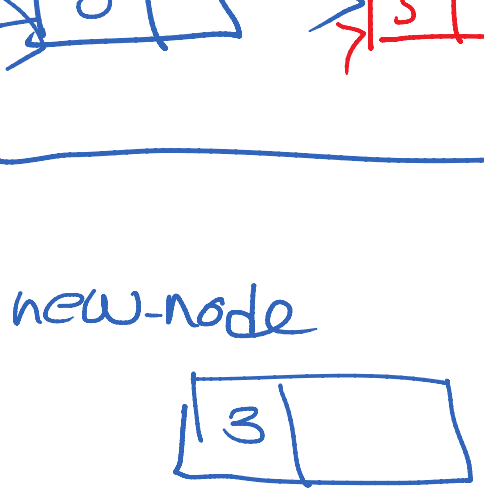
Inserting a New Node in a Circular Linked List

- In this section, we will see how a new node is added into an already existing linked list.
- We will take two cases and then see how insertion is done in each case.

  Case 1: The new node is inserted at the beginning of the circular linked list.

  Case 2: The new node is inserted at the end of the circular linked list.

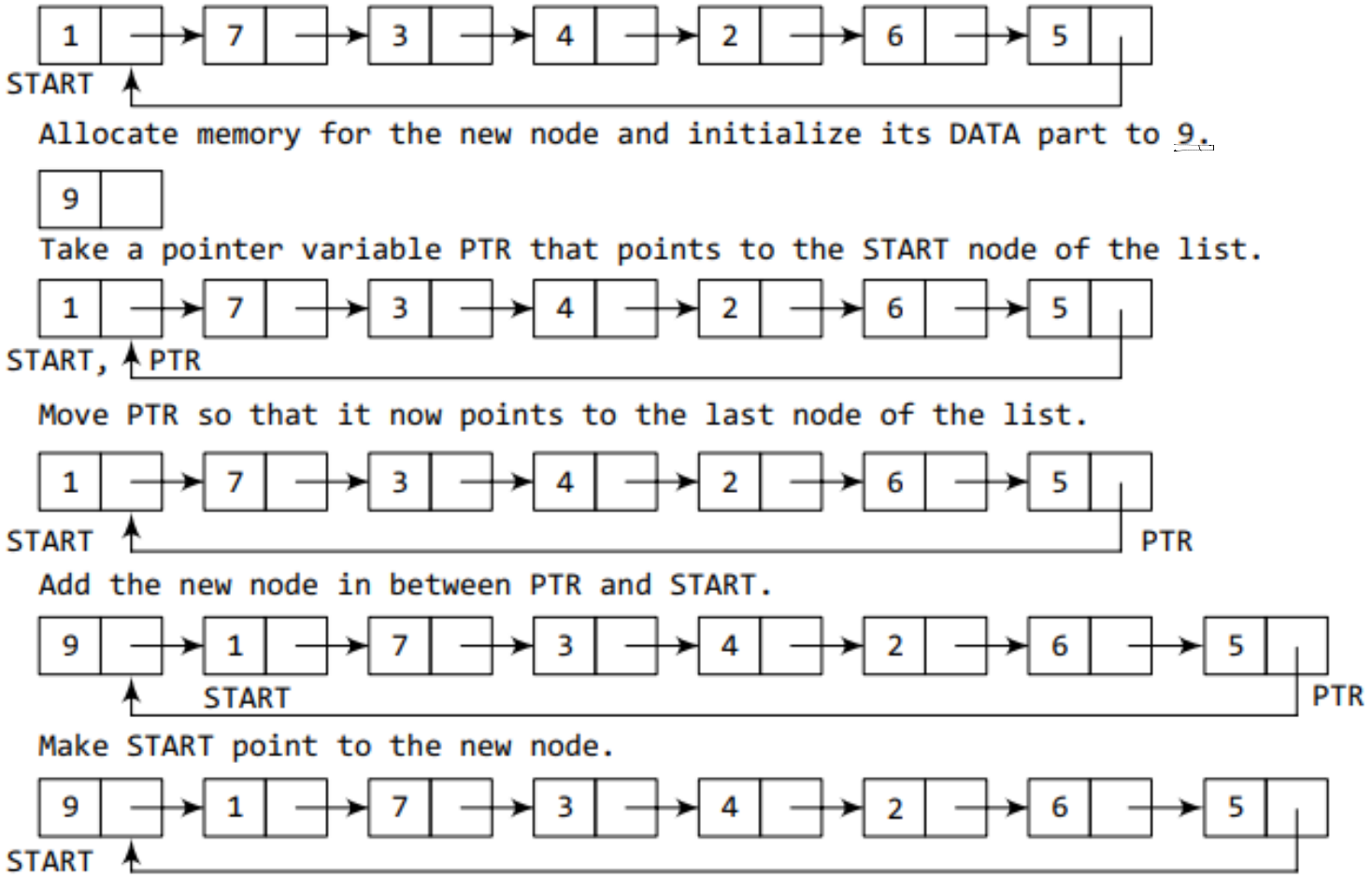





```
new_node -> next = start
ptr = start
while (ptr -> next != start)
    ptr = ptr -> next
ptr -> next = new_node
start = new_node
```

Inserting a Node at the Beginning of a Circular Linked List

- Consider the linked list shown in below Fig. Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.



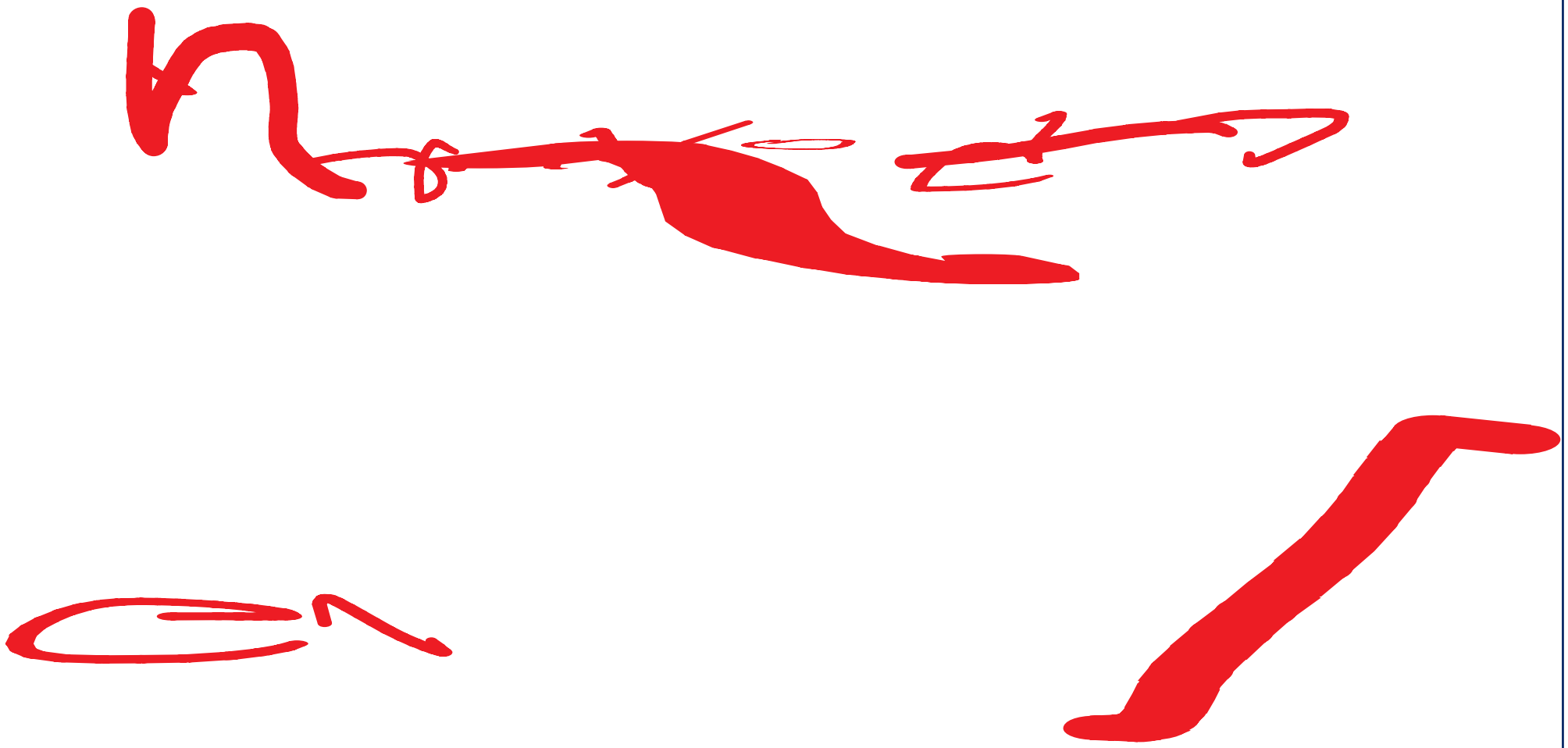
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Algorithm to insert a new node at the beginning

Code for Inserting a Node at the Beginning of a Circular Linked List

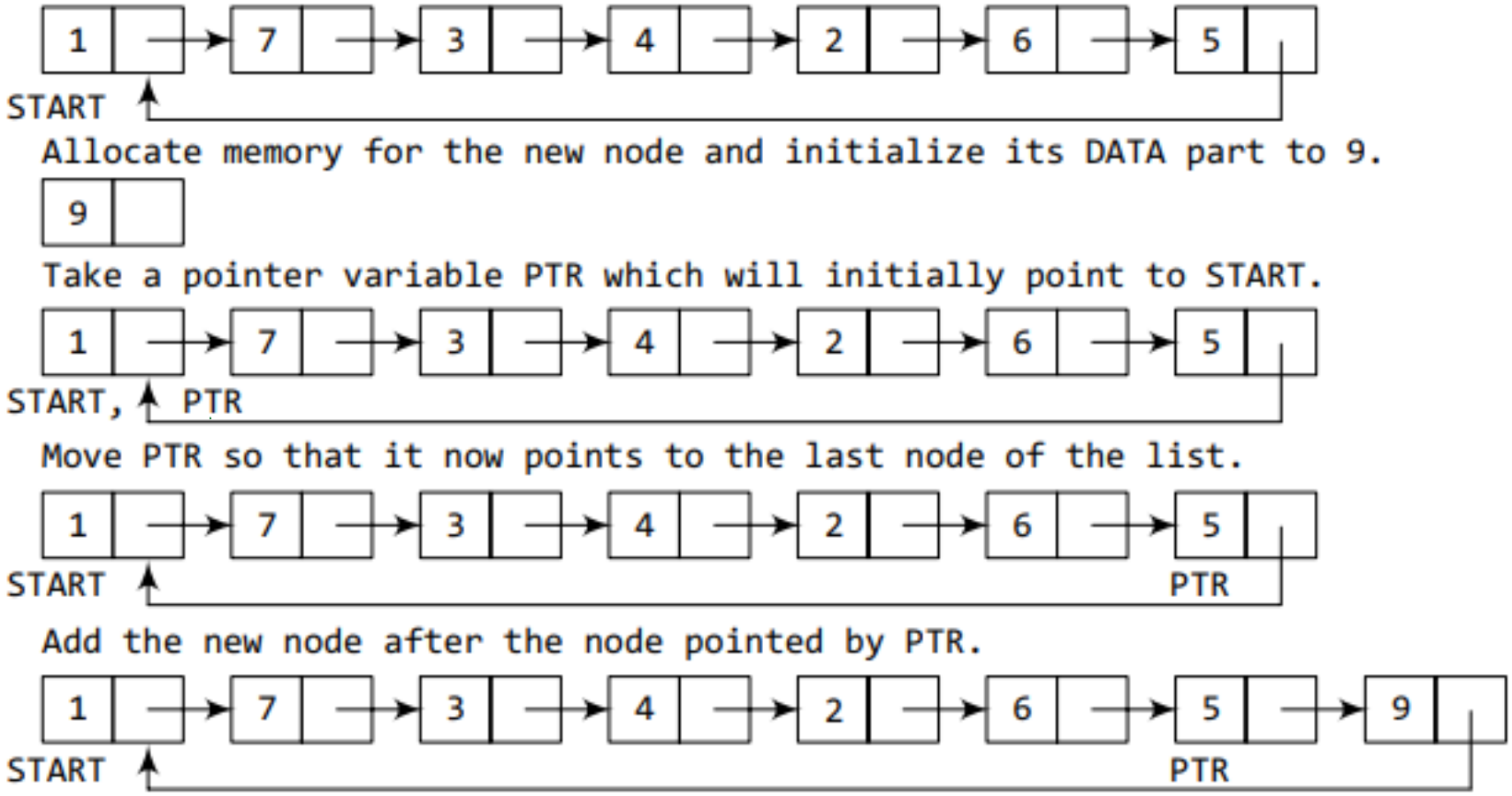
```
struct node *insert_beg(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        while(ptr -> next != start)
                ptr = ptr -> next;
        ptr -> next = new_node;
        new_node -> next = start;
        start = new_node;
        return start;
}
```

Inserting a Node at the End of a Circular Linked List



Inserting a Node at the End of a Circular Linked List

- Consider the linked list shown in below Fig. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

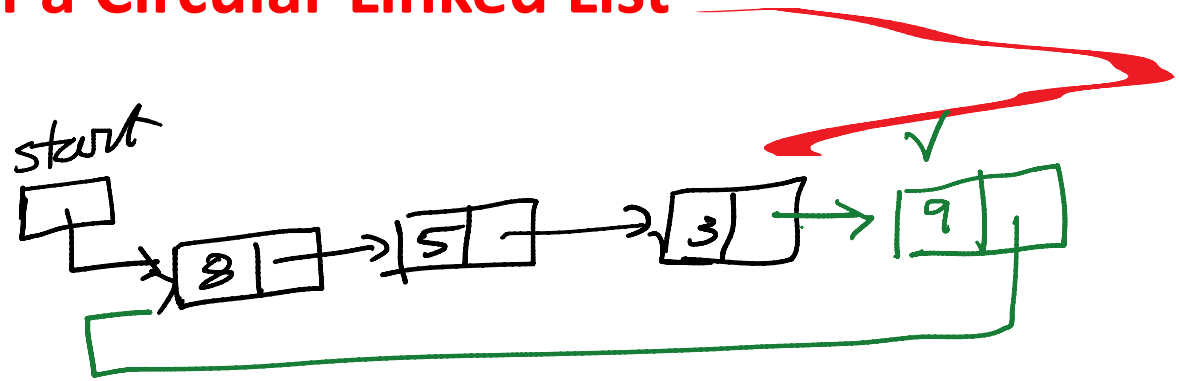


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Algorithm to insert a new node at the end

Code for Inserting a Node at the End of a Circular Linked List

```
struct node *insert_end(struct node *start)
{
        struct node *ptr, *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        while(ptr -> next != start)
                ptr = ptr -> next;
        ptr -> next = new_node;
        new_node -> next = start;
        return start;
}
```

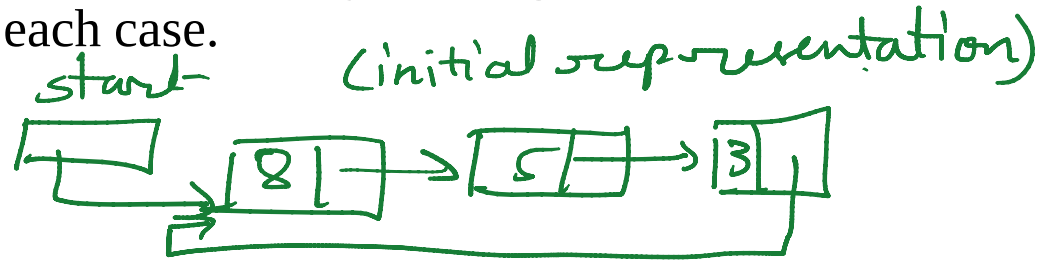


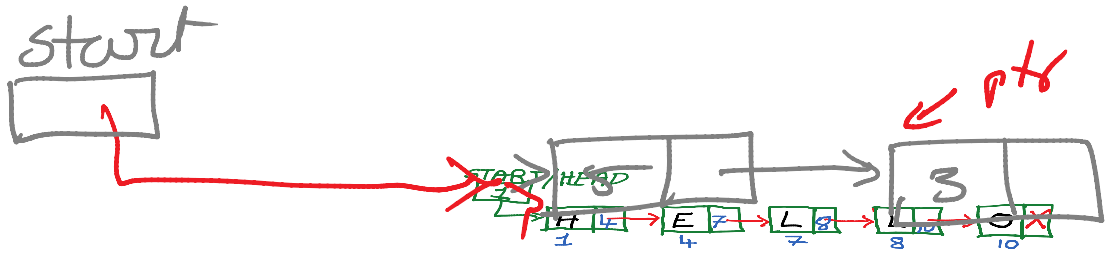
Deleting a Node from a Circular Linked List

- In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case.

  Case 1: The first node is deleted.

  Case 2: The last node is deleted.



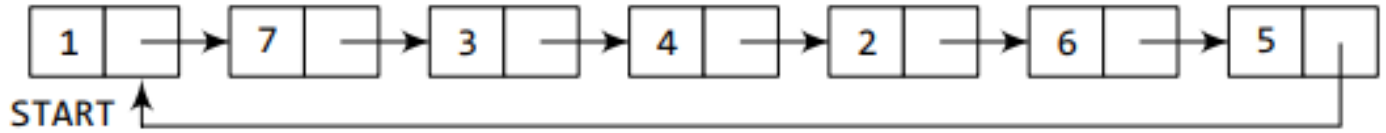


ptr = start;
while (ptr->next !=
ptr = ptr->ne

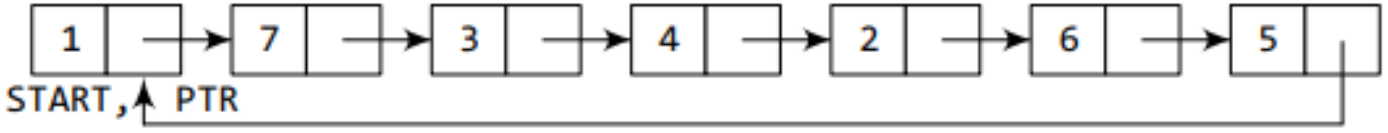
ptr->next = start->next
→ free(start);
start = ptr->next

Deleting the First Node from a Circular Linked List

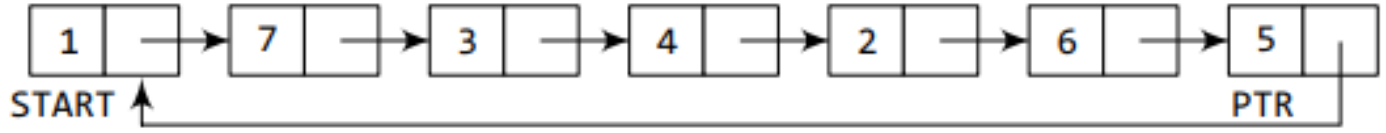
- Consider the circular linked list shown in below Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



Take a variable PTR and make it point to the START node of the list.



Move PTR further so that it now points to the last node of the list.



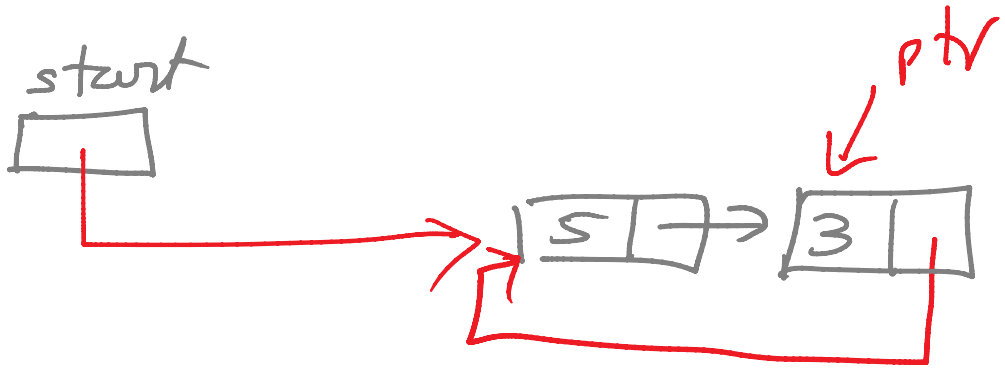
The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

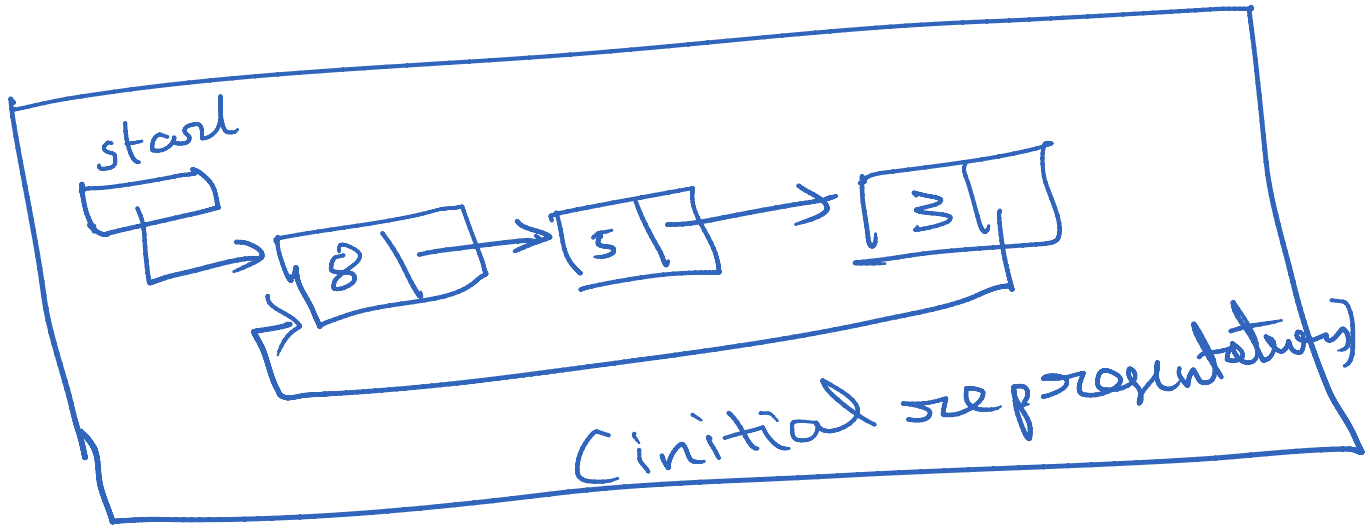
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

Algorithm to delete the first node

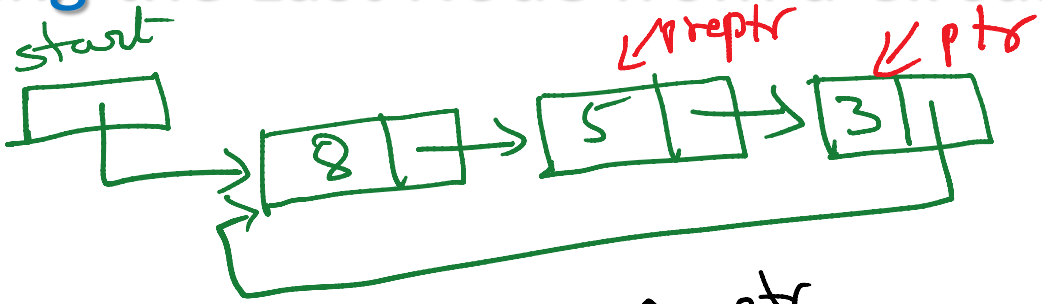
Code for deleting a Node at the Beginning of a Circular Linked List

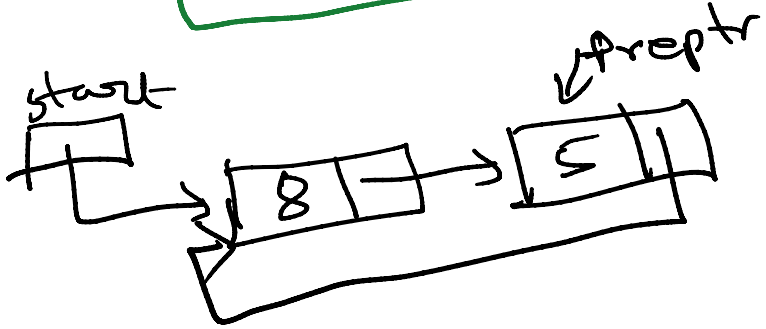
```
struct node* delete_beg(struct node *start)
{
      struct node *ptr;
      ptr = start;
      while(ptr -> next != start)
            ptr = ptr -> next;
      ptr -> next = start -> next;
      free(start);
      start = ptr -> next;
      return start;
}
```





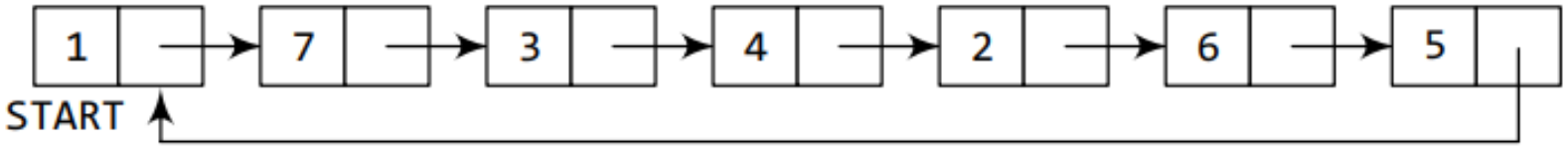
Deleting the Last Node from a Circular Linked List



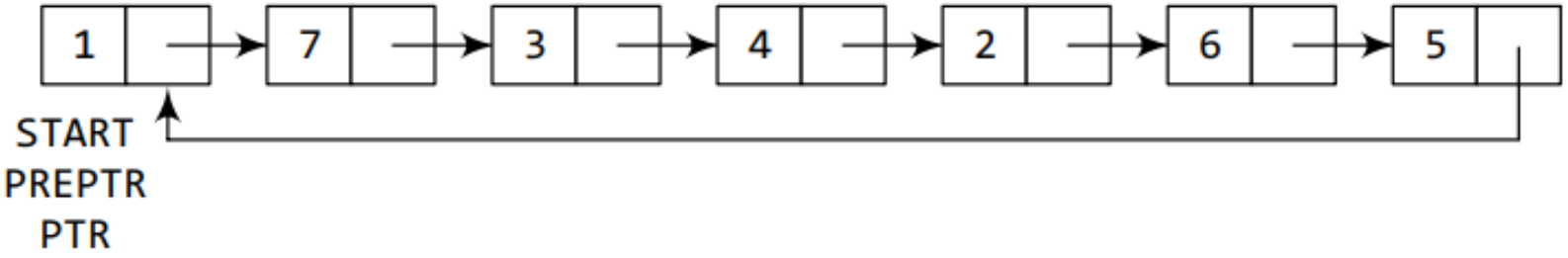


free(ptr)

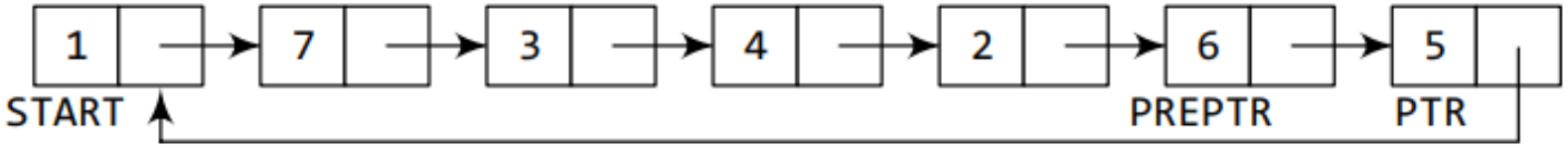
Deleting the Last Node from a Circular Linked List



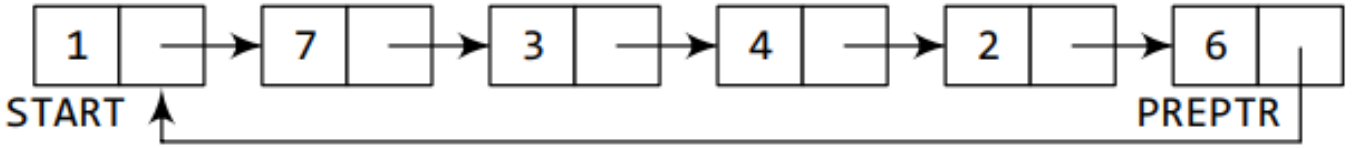
Take two pointers PREPTR and PTR which will initially point to START.



Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.



Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.

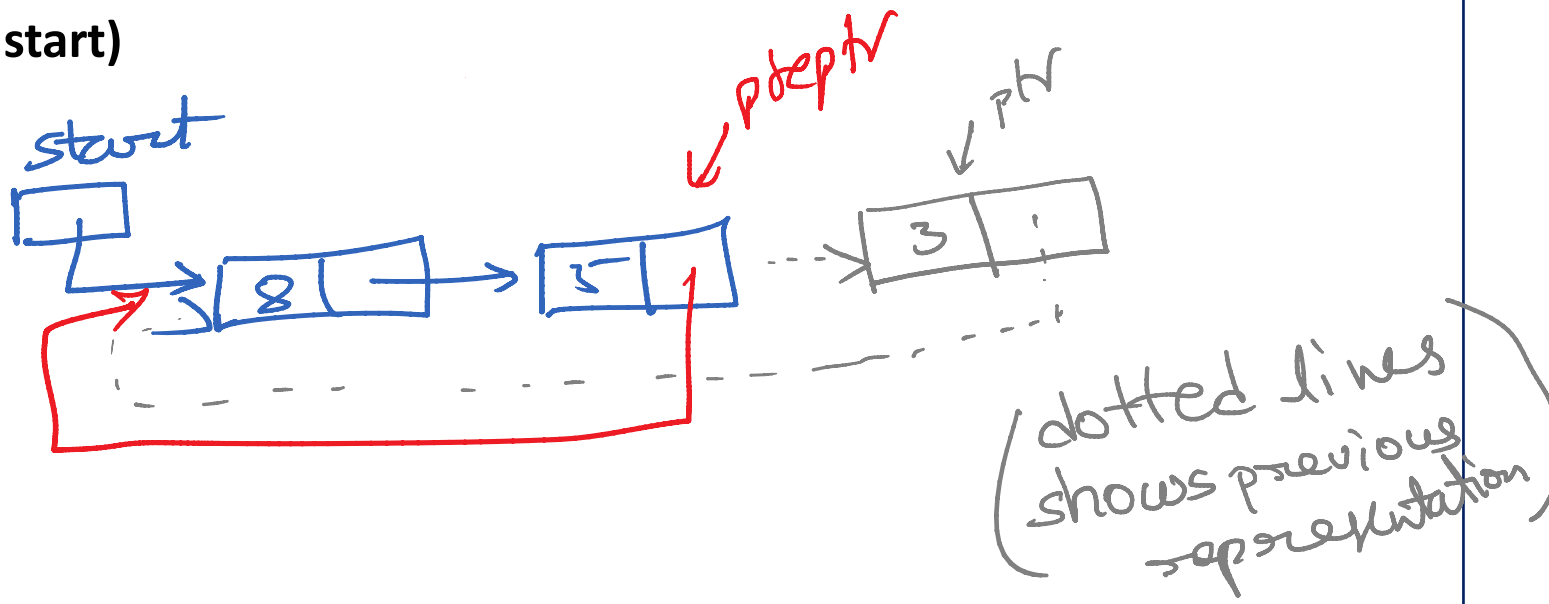


```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

Algorithm to delete the last node

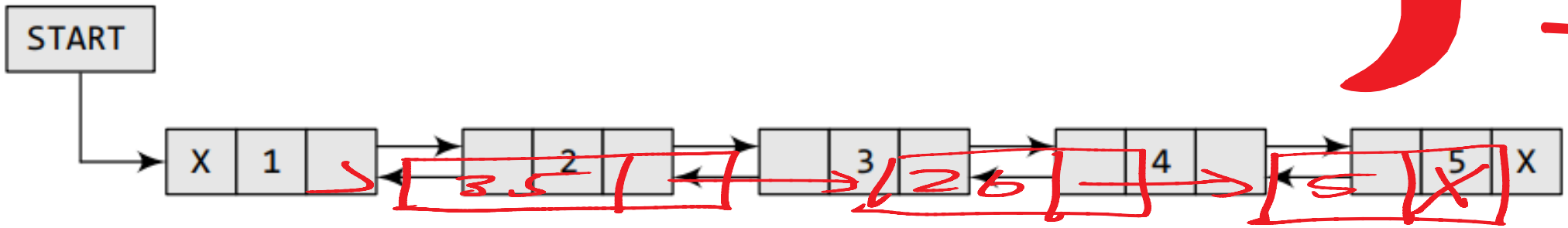
Code for deleting a Node at the End of a Circular Linked List

```
struct node* delete_end(struct node *start)
{
        struct node *ptr, *preptr;
        ptr = start;
        while(ptr -> next != start)
        {
                preptr = ptr;
                ptr = ptr -> next;
        }
        preptr -> next = start;
        free(ptr);
        return start;
}
```



DOUBLY LINKED LISTS

- A doubly linked list or a two-way 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.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.



- The PREV field of the first node and the NEXT field of the last node will contain NULL.
- The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

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

Memory Representation of doubly linked list:

START

1

	DATA	PREV	NEXT
1	H	-1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	-1

Operations on Doubly Linked list

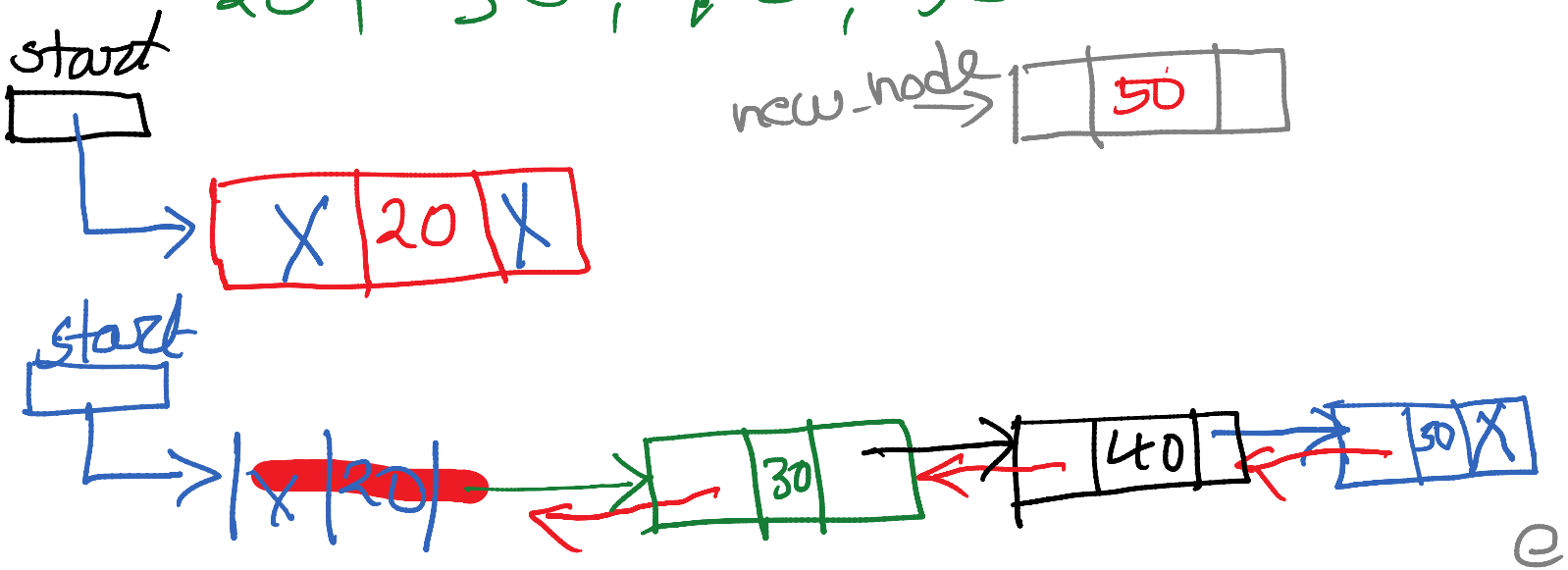
➢The following operations are performed on Doubly linked list.

1. **Creating a Doubly linked list**
2. **Inserting a new node in a Doubly linked list**
3. **Deleting a node from a Doubly linked list**

Creating a Doubly Linked List

creation

20, 30, 40, 50




```
if(start == NULL)
{
new_node -> next = N
new-node -> prev = N
start = new_node
}
else
{ ptr = start;
while(ptr -> next != NU
    ptr = ptr -> next
ptr -> next = new-no
new-node -> next = NU
new-node -> prev = ptr
}
```

display

```
ptr = start;
while(ptr != NULL)
{ printf("%d", ptr -> data)
  ptr = ptr -> next;
}
```

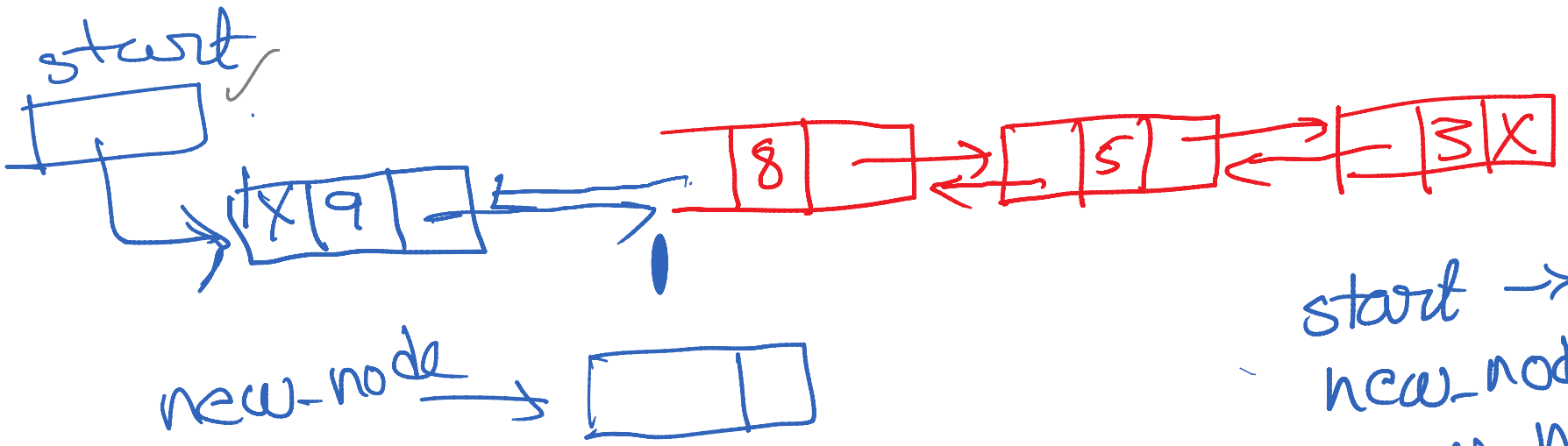
Inserting a New Node in a Doubly Linked List

- In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take three cases and then see how insertion is done in each case.

  Case 1: The new node is inserted at the beginning.

  Case 2: The new node is inserted at the end.

  Case 3: The new node is inserted after a given node.



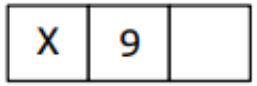
start → prev = new_node
new_node → prev = NULL
new_node → next = start
start = new_node

Inserting a Node at the Beginning of a Doubly Linked List

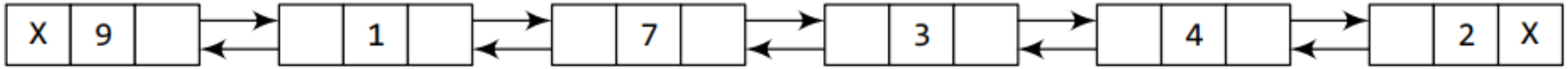
- Consider the doubly linked list shown in below Fig. Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.

START

Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.



Add the new node before the START node. Now the new node becomes the first node of the list.



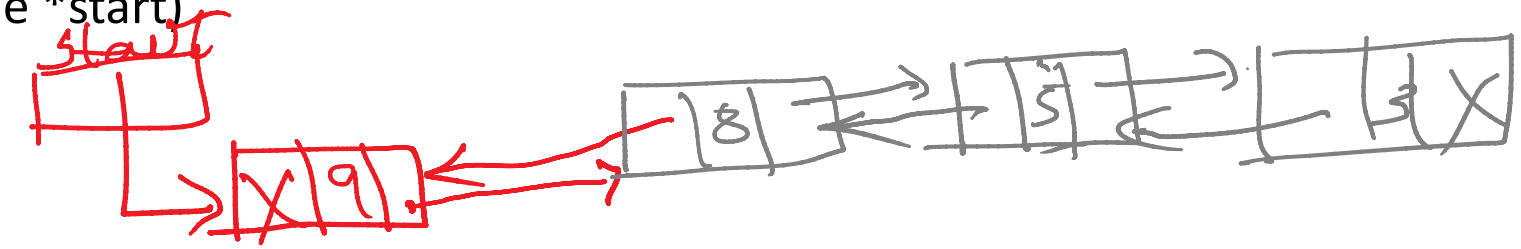
START

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Algorithm to insert a new node at the beginning

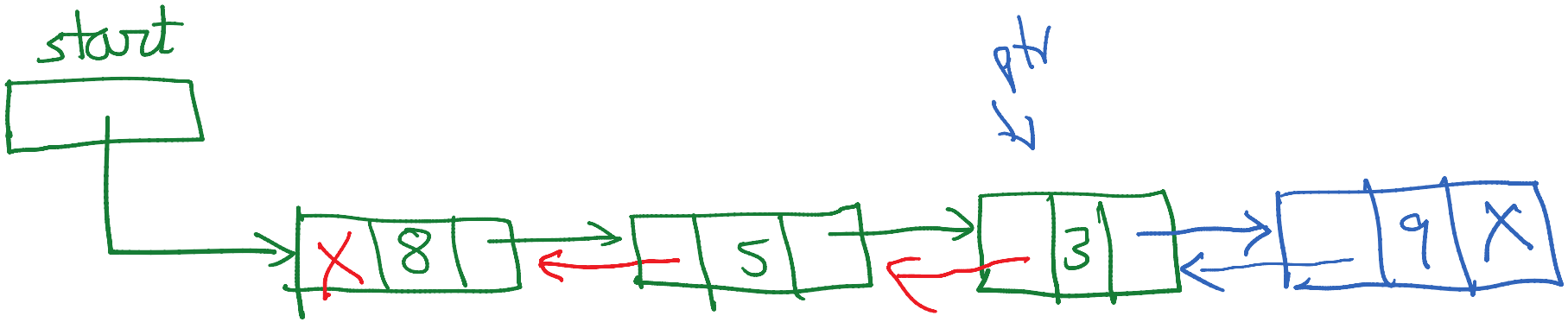
Code for inserting a Node at the Beginning of a Doubly Linked List

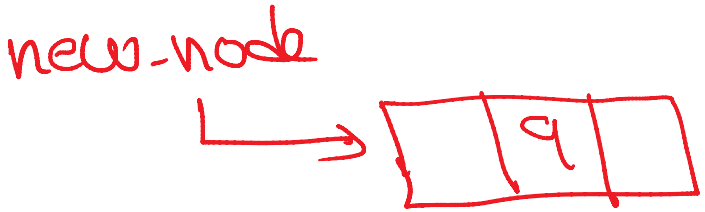
```
struct node *insert_beg(struct node *start)
{
struct node *new_node;
int num;
printf("\n Enter the data : ");
scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));
new_node -> data = num;
start -> prev = new_node;
new_node -> next = start;
new_node -> prev = NULL;
start = new_node;
return start;
}
```



num=9

Inserting a Node at the End of a Doubly Linked List






```
ptr = start
while(ptr -> next != NUL
    ptr = ptr -> nex
ptr -> next = new_n
new_node -> next = NUL
new_node -> prev = ptr
```

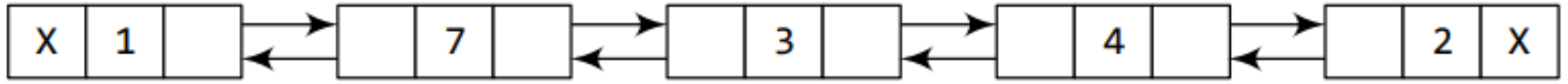
Inserting a Node at the End of a Doubly Linked List

- Consider the doubly linked list shown in below Fig. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

START

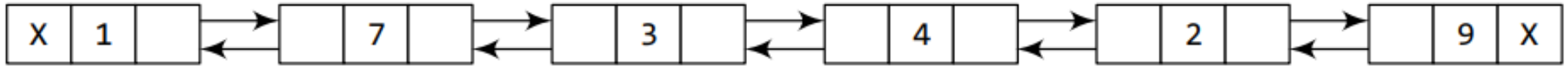
Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

Take a pointer variable PTR and make it point to the first node of the list.



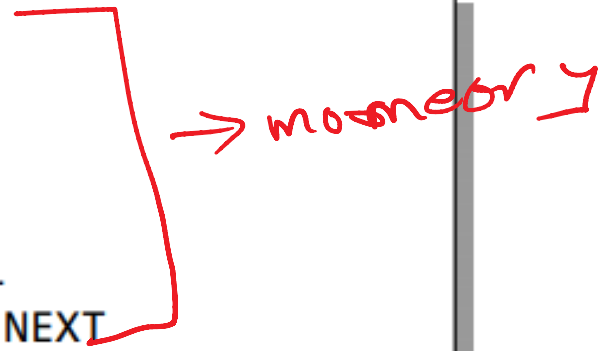
START,PTR

Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.



START PTR

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```



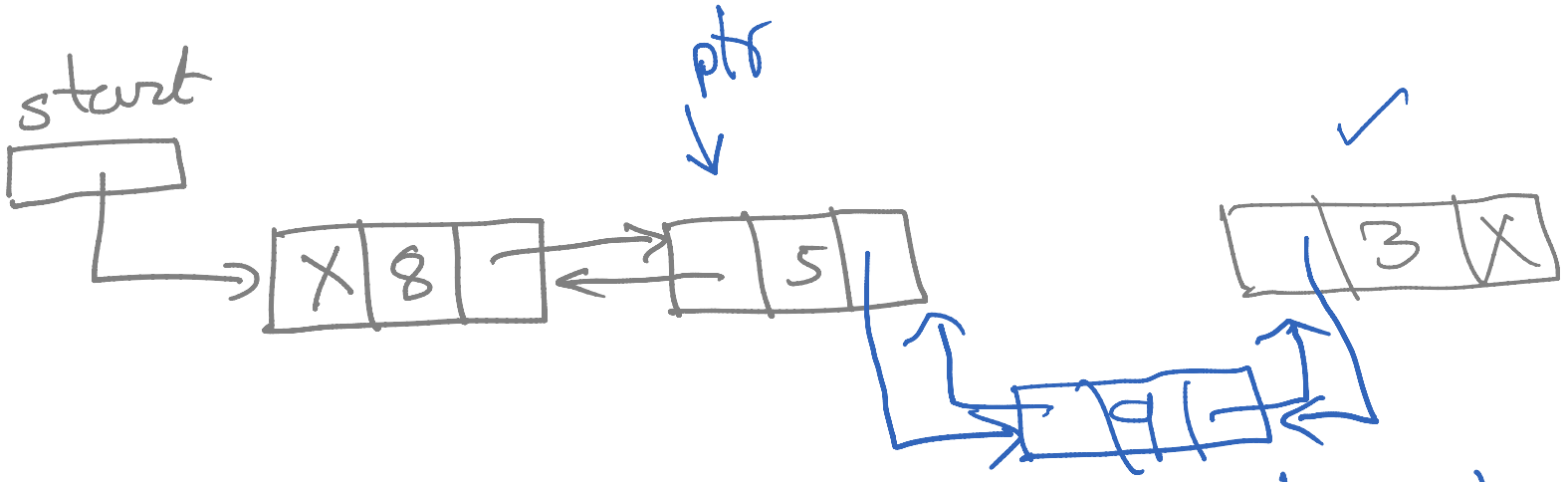
Algorithm to insert a new node at the end

Code for inserting a Node at the End of a Doubly Linked List

```
struct node *insert_end(struct node *start)
{
        struct node *ptr, *new_node;
        int num;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr=start;
        while(ptr -> next != NULL)
                ptr = ptr -> next;
        ptr -> next = new_node;
        new_node -> prev = ptr;
        new_node -> next = NULL;
        return start;
}
```

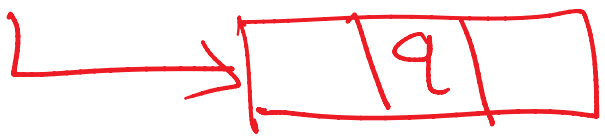
num=9

Inserting a Node After a Given Node in a Doubly Linked List



val = 5

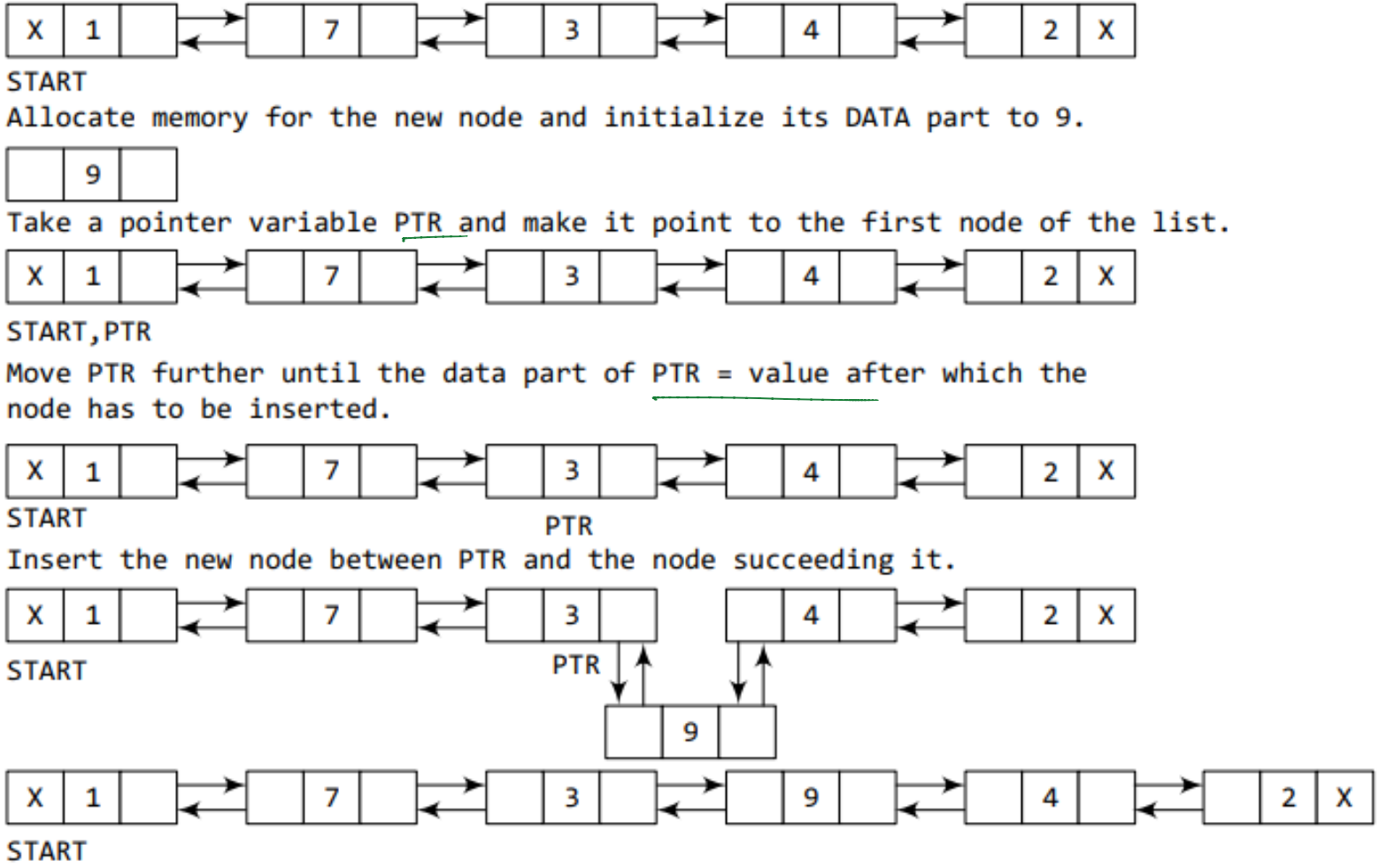
new_node




```
ptr = start;
while(ptr->data != va
    ptr = ptr->next;
new_node->prev = ptr;
new_node->next = ptr->
ptr->next->prev = new
ptr->next = new_no
```

Inserting a Node After a Given Node in a Doubly Linked List

- Consider the doubly linked list shown in below Fig. Suppose we want to add a new node with value 9 after the node containing 3. The following changes will be done in the linked list

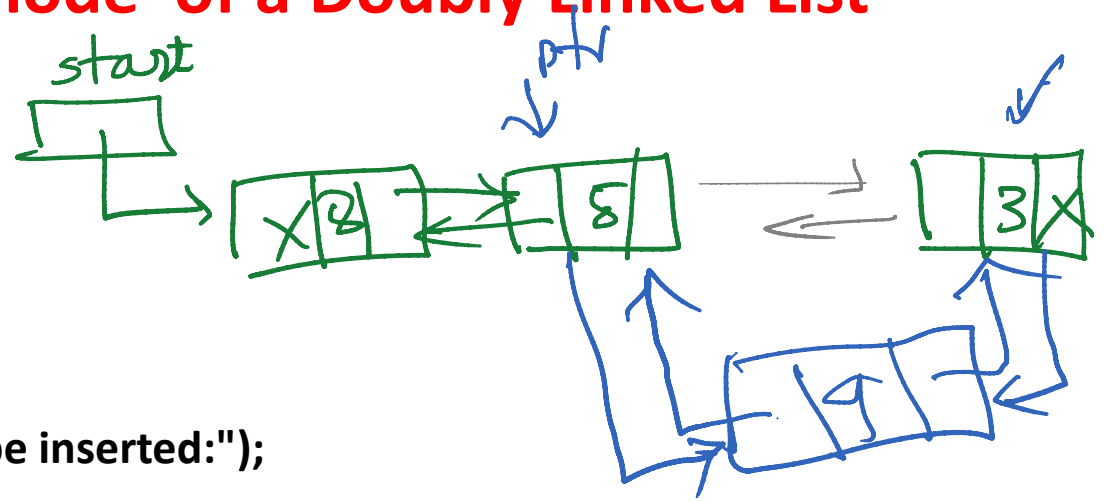


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Algorithm to insert a new node after a given node

Code for inserting a Node after a given node of a Doubly Linked List

```
struct node *insert_after(struct node *start)
{
        struct node *new_node, *ptr;
        int num, val;
        printf("\n Enter the data : ");
        scanf("%d", &num);
        printf("\n Enter the value after which the data has to be inserted:");
        scanf("%d", &val);
        new_node = (struct node *)malloc(sizeof(struct node));
        new_node -> data = num;
        ptr = start;
        while(ptr -> data != val)
                ptr = ptr -> next;
        new_node -> prev = ptr;
        new_node -> next = ptr -> next;
        ptr -> next -> prev = new_node;
        ptr -> next = new_node;
        return start;
}
```



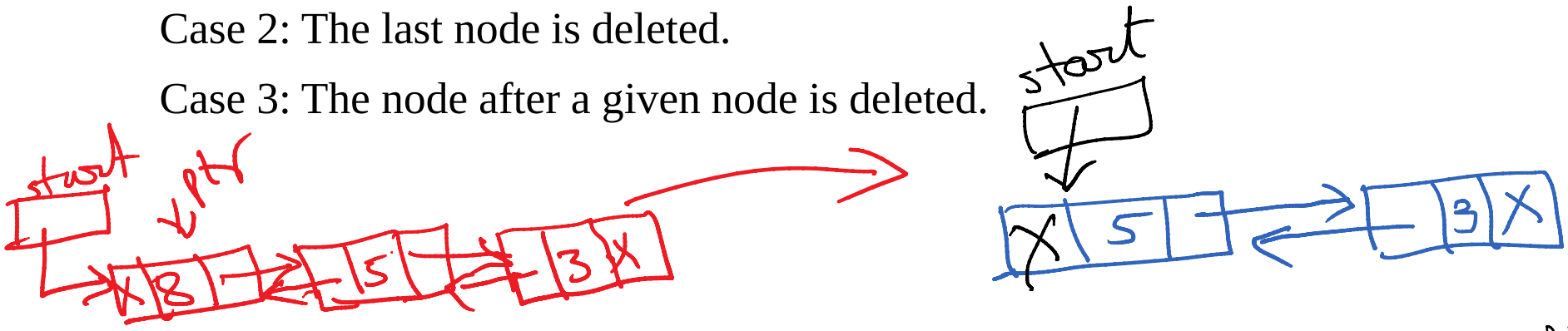
Deleting a Node from a Doubly Linked List

- In this section, we will see how a node is deleted from an already existing doubly linked list. We will take four cases and then see how deletion is done in each case.

  Case 1: The first node is deleted.

  Case 2: The last node is deleted.

  Case 3: The node after a given node is deleted.




```
ptr = start;
start = start -> ne
start -> prev = Nu
free(ptr);
```

Deleting the First Node from a Doubly Linked List

- Consider the doubly linked list shown in below Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

START

Free the memory occupied by the first node of the list and make the second node of the list as the START node.

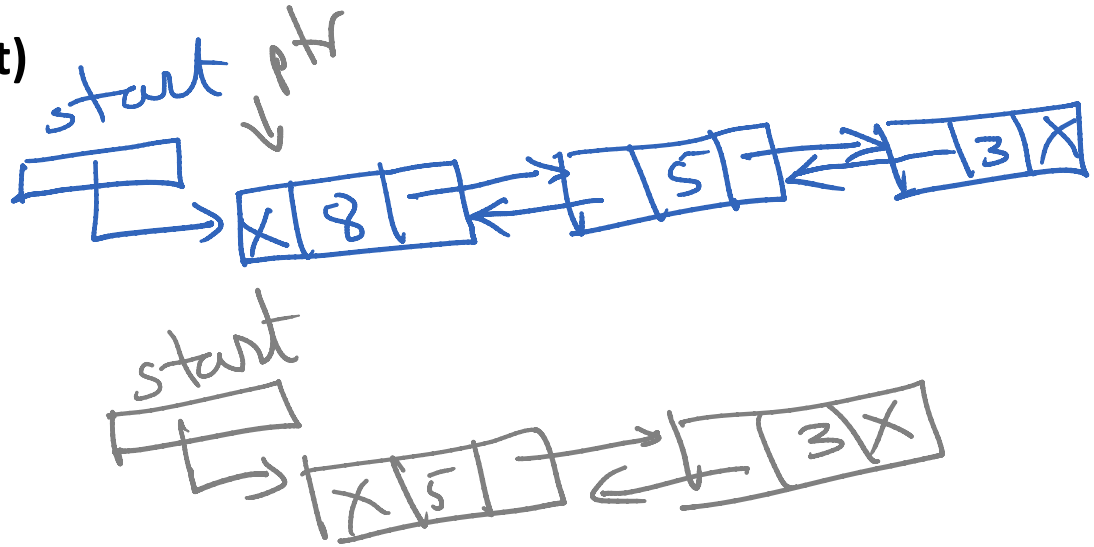
START

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

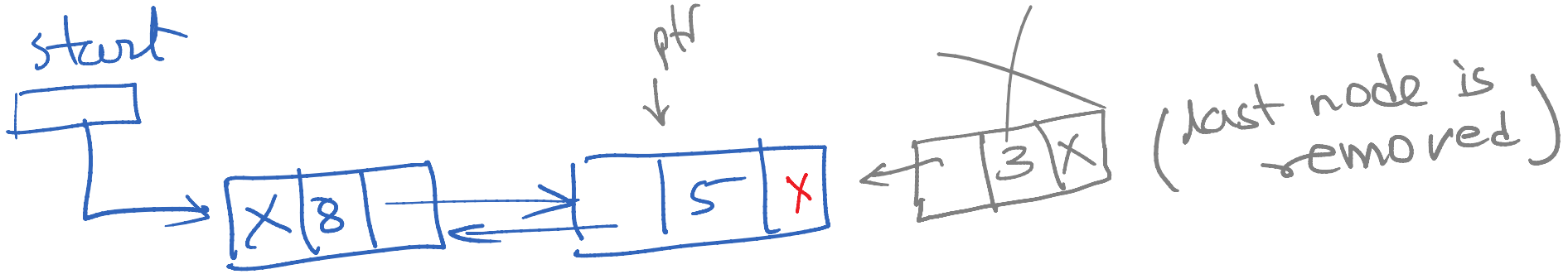
Algorithm to delete the first node

Code for Deleting a Node at the Beginning of a Doubly Linked List

```
struct node *delete_beg(struct node *start)
{
        struct node *ptr;
        ptr = start;
        start = start -> next;
        start -> prev = NULL;
        free(ptr);
        return start;
}
```



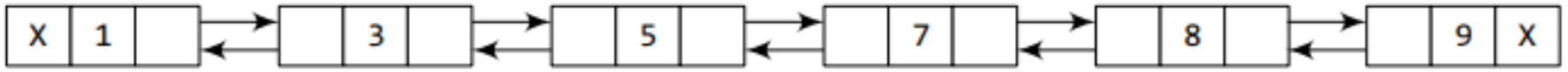
Deleting the Last Node from a Doubly Linked List




```
ptr = start
while(ptr -> next != N
    ptr = ptr -> ne
ptr -> prev -> next =
free (ptr);
```

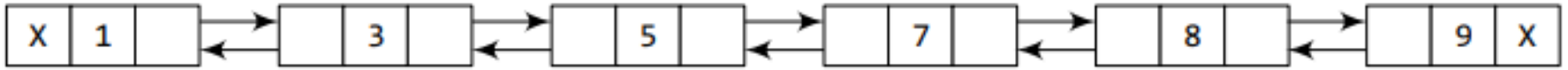
Deleting the Last Node from a Doubly Linked List

- Consider the doubly linked list shown in below Fig. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



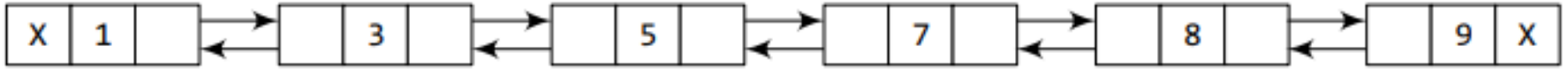
START

Take a pointer variable PTR that points to the first node of the list.



START,PTR

Move PTR so that it now points to the last node of the list.



START PTR

Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.

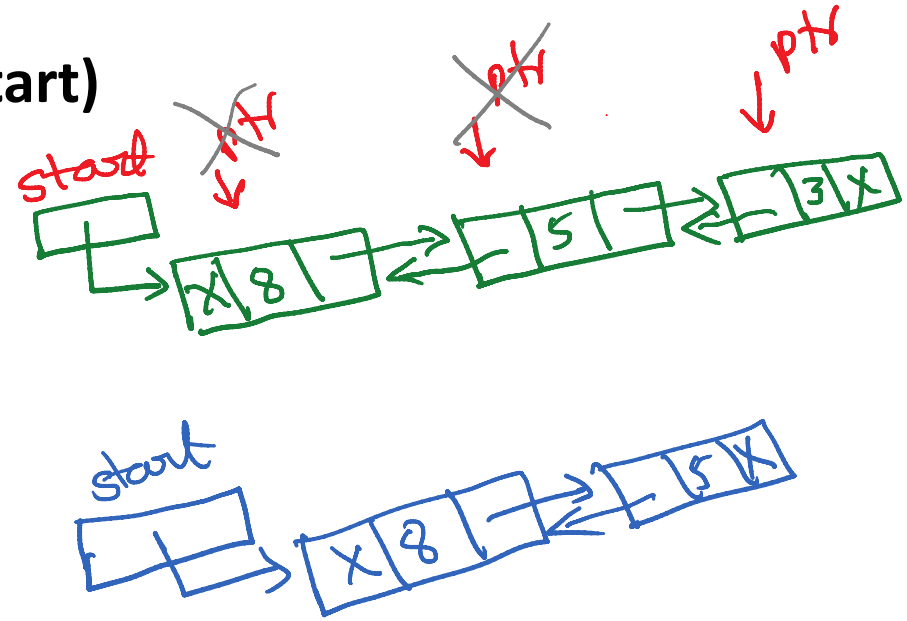
START

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

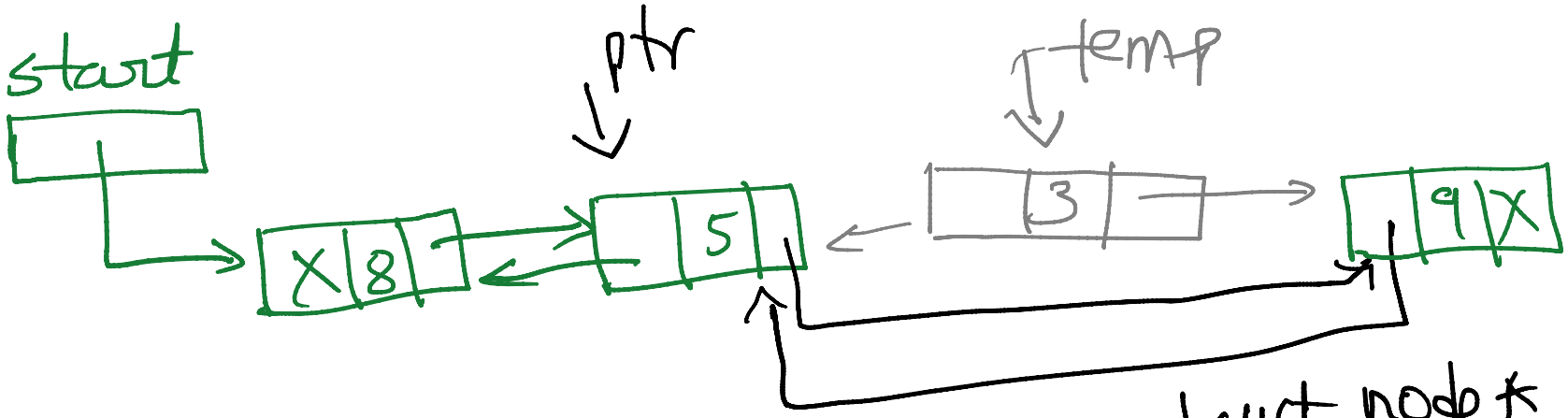
Algorithm to delete the last node

Code for Deleting a Node at the End of a Doubly Linked List

```
struct node *delete_end(struct node *start)
{
      struct node *ptr;
      ptr = start;
      while(ptr -> next != NULL)
            ptr = ptr -> next;
      ptr -> prev -> next = NULL;
      free(ptr);
      return start;
}
```



Deleting the Node After a Given Node in a Doubly Linked List

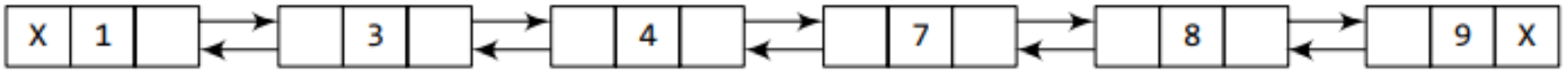


val = 5

```
struct node * temp;
ptr = start;
while(ptr->data != val)
    ptr = ptr -> next;
temp = ptr -> next;
ptr -> next = temp -> next
temp -> next -> prev = ptr;
free(temp);
```

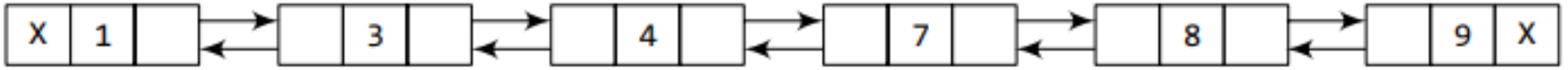
Deleting the Node After a Given Node in a Doubly Linked List

- Consider the doubly linked list shown in below Fig. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



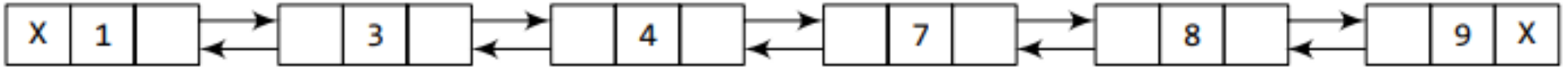
START

Take a pointer variable PTR and make it point to the first node of the list.



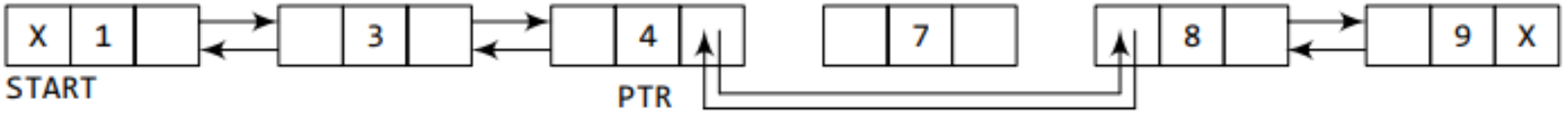
START,PTR

Move PTR further so that its data part is equal to the value after which the node has to be ~~inserted~~. deleted



START PTR

Delete the node succeeding PTR.



START PTR

START

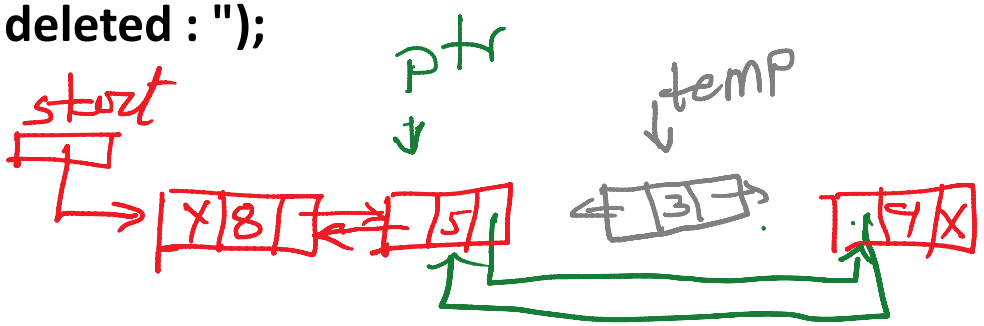
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Algorithm to delete a node after a given node

Code for Deleting a Node after a given node of a Doubly Linked List

```
struct node *delete_after(struct node *start)
{
        struct node *ptr, *temp;
        int val;
        printf("\n Enter the value after which the node has to deleted : ");
        scanf("%d", &val);
        ptr = start;
        while(ptr -> data != val)
                ptr = ptr -> next;
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        return start;
}
```

val = 5



Sparse Matrix Representation using Linked List

What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m x n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as **sparse matrix.**

Sparse matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

Why do we need to use a sparse matrix instead of a simple matrix?

The following are the advantages of using a sparse matrix:

- **Storage:** As we know, a sparse matrix that contains lesser non-zero elements than zero so less memory can be used to store elements. It evaluates only the non-zero elements.
- **Computing time:** In the case of searching n sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Sparse Matrix Representation

The non-zero elements can be stored with triples, i.e., rows, columns, and value. The sparse matrix can be represented in the following ways:

- Array representation
- Linked list representation

Array Representation:

The 2d array can be used to represent a sparse matrix in which there are three rows named as:

- **Row:** It is an index of a row where a non-zero element is located.
- **Column:** It is an index of the column where a non-zero element is located.
- **Value:** The value of the non-zero element is located at the index (row, column).

Sparse matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

→

Row	0	0	1	1	2	3	3
Column	1	3	2	3	2	0	1
Value	4	5	3	6	2	2	3

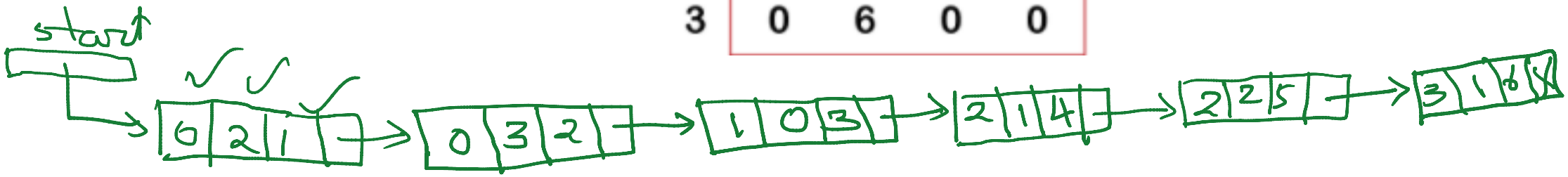
Sparse Matrix Representation using Linked List

In linked list representation, linked list data structure is used to represent a sparse matrix. In linked list representation, each node consists of four fields whereas, in array representation, there are three fields, i.e., row, column, and value. The following are the fields in the linked list:

- **Row:** It is an index of row where a non-zero element is located.
- **Column:** It is an index of column where a non-zero element is located.
- **Value:** It is the value of the non-zero element which is located at the index (row, column).
- **Next node:** It stores the address of the next node.

Sparse matrix →

	0	1	2	3
0	0	0	1	2
1	3	0	0	0
2	0	4	5	0
3	0	6	0	0

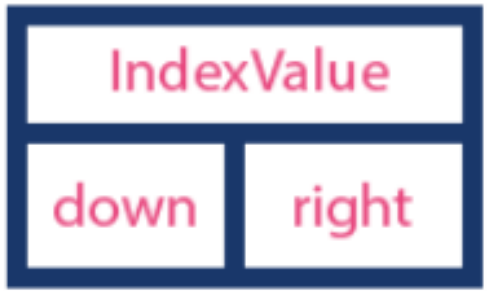


Another Representation using Linked List

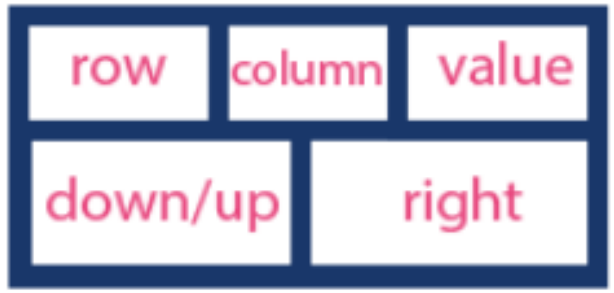
Linked Representation

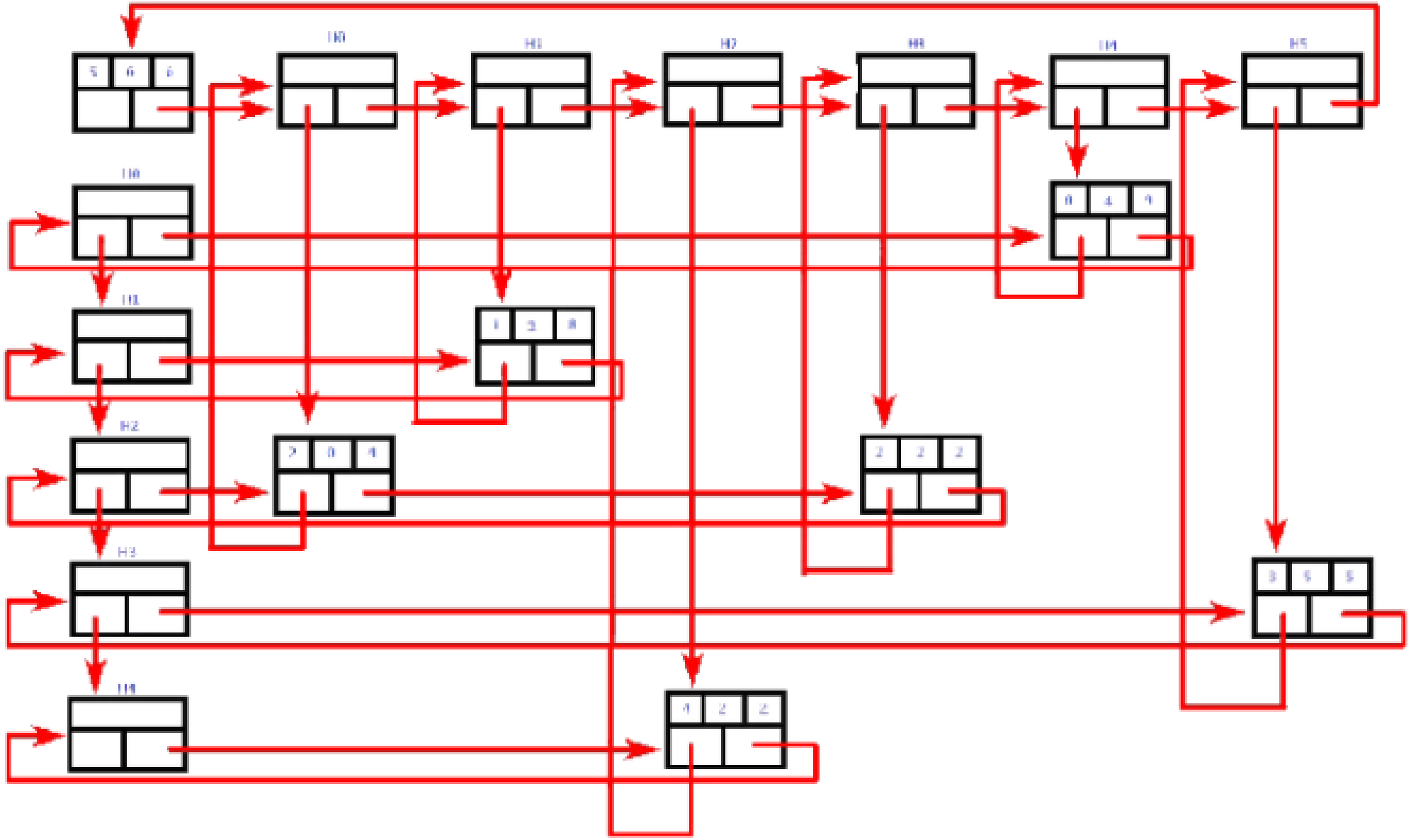
In linked representation, we use a linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node.** Header node consists of three fields and element node consists of five fields as shown in the image...

Header Node



Element Node





Advantages and Disadvantages of Linked list

Advantages Of Linked List:

- **Dynamic data structure:** A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give the initial size of the linked list.
- **No memory wastage:** In the Linked list, efficient memory utilization can be achieved since the size of the linked list increase or decrease at run time so there is no memory wastage and there is no need to pre-allocate the memory.
- **Implementation:** Linear data structures like stack and queues are often easily implemented using a linked list.
- **Insertion and Deletion Operations:** Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element only the address present in the next pointer needs to be updated.

Advantages and Disadvantages of Linked list

Disadvantages Of Linked List:

- **Memory usage:** More memory is required in the linked list as compared to an array. Because in a linked list, a pointer is also required to store the address of the next element.
- **Traversal:** In a Linked list traversal is more time-consuming as compared to an array. Direct access to an element is not possible in a linked list as in an array by index. For example, for accessing a mode at position n, one has to traverse all the nodes before it.
- **Reverse Traversing:** In a singly linked list reverse traversing is not possible, but in the case of a doubly-linked list, it can be possible as it contains a pointer to the previously connected nodes with each node. For performing this extra memory is required for the back pointer hence, there is a wastage of memory.
- **Random Access:** Random access is not as possible in a linked list due to its dynamic memory allocation.