UNIT I LINEAR DATA STRUCTURES – LIST

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation –singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

INTRODUCTION TO DATA STRUCTURE

A **program** is said to be efficient when it executes in minimum time and with minimum memory space. In order to write efficient programs, we need to apply certain data management concepts. The concept of data management is a complex task that includes activities like data collection, organization of data into appropriate structures, and developing and maintaining routines for quality assurance.

Good program I defined as a program that

- it runs correctly
- is easy to read and understand
- is easy to debug and
- is easy to modify.

DATA STRUCTURE

Definition

A **data structure** is a particular way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently. Data structures are used in almost every program or software system. Common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables etc.

Applications of Data Structures

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial Intelligence

The major data structures used in the Network data model is graphs, Hierarchical data model is trees, and RDBMS is arrays. Specific data structures are essential ingredients of many efficient algorithms as they enable the programmers to manage huge amounts of data easily and efficiently. Some formal design methods and programming languages emphasize data structures and the algorithms as the key organizing factor in software design. This is because representing information is fundamental to computer science. The primary goal of a program or software is not to perform calculations or operations but to store and retrieve information as fast as possible.

When selecting a data structure to solve a problem, the following steps must be performed.

1.Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.

2. Quantify the resource constraints for each operation.

3. Select the data structure that best meets these requirements.

This three-step approach to select an appropriate data structure for the problem at hand supports a data-centered view of the design process.

In this approach, there are three concern

- The first concern is data and the operations that are to be performed on them.
- The second concern about representation of the data.
- The third concern about the implementation of that representation.

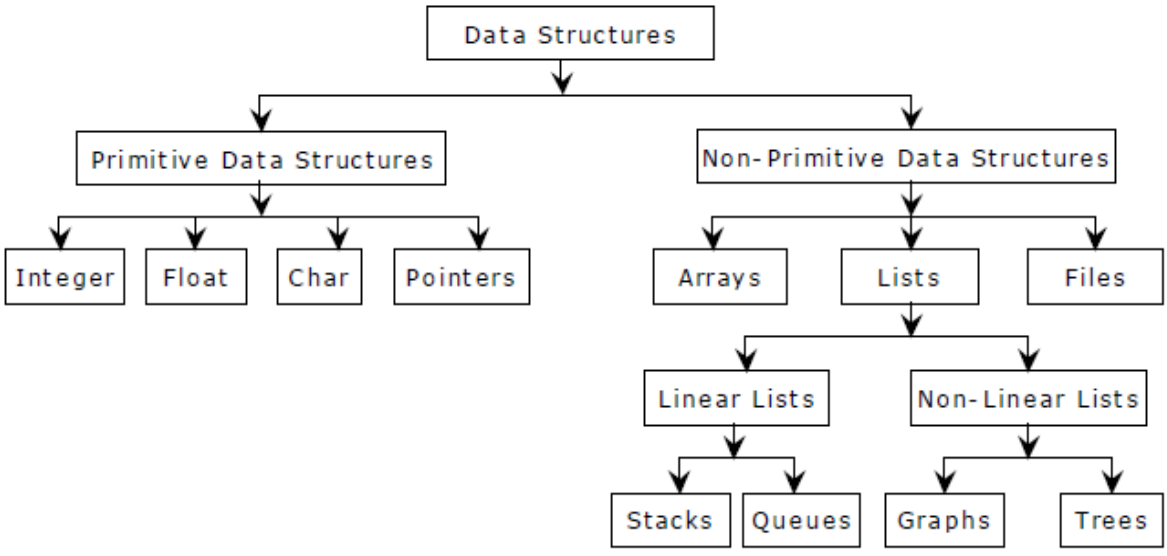
While one data structure may allow accessing data items sequentially, the other may allow random access of data. So, selection of an appropriate data structure for the problem is a crucial decision and may have a major impact on the performance of the program.

ELEMENTARY DATA STRUCTURE ORGANIZATION

Data structures are building blocks of a program. The term data means a value or set of values. It specifies either the value of a variable or a constant. A record is a collection of data items. A file is a collection of related records. Each record in a file may consist of multiple data items but the value of a certain data item uniquely identifies the record in the file. Such a data item K is called a primary key, and the values K1, K2 ... in such field are called keys or key values.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally categorized into two classes: primitive and non-primitive data structures.



- **Primitive data structures** are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean.
- **Non-primitive data structures** are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

LINEAR AND NON-LINEAR STRUCTURES

- **LINEAR DATA STRUCTURES**

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures can be represented in memory in two different ways. One way is to have to a linear relationship between elements by means of sequential memory locations. The other way is to have a linear relationship between elements by means of links.

- **NON-LINEAR DATA STRUCTURES**

If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

OPERATIONS ON DATA STRUCTURES

- **Traversal:** Visit every part of the data structure.
- **Search:** Traversal through the data structure for a given element.
- **Insertion:** Adding new elements to the data structure.
- **Deletion:** Removing an element from the data structure.
- **Sorting:** Rearranging the elements in some type of order (e.g Increasing or Decreasing).
- **Merging:** Combining two similar data structures into one.

ABSTRACT DATA TYPE

- An **abstract data type (ADT)** is the way we look at a data structure, focusing on what it does and ignoring how it does its job. The abstract data type is a triple of D-set of Domains, F-Set of functions, A-Axioms in which only what is to be done is mentioned but how is to be done is not mentioned.

ADT = Type + Function Names + Behavior of each function.

Examples:

- Stacks
- Queues
- Linked List

ADT Operations

- While modeling the problems the necessary details are separated out from the unnecessary details. This process of modeling the problem is called abstraction.
- The model defines an abstract view to the problem. It focuses only on problem related stuff and that you try to define properties of the problem.
- These properties include
  - ✓ The data which are affected
  - ✓ The operations which are identified.

Abstract data type operations are

- **Create:** create the database.
- **Display:** displaying all the elements of the data structure.
- **Insertion:** elements can be inserted at any desired position.
- **Deletion:** desired element can be deleted from the data structure.
- **Modification:** any desired element can be deleted from the data structure.

Advantage of using ADTs

- It is reusable, robust
- It can be re-used at several places and it reduces coding efforts
- Encapsulation ensures that data cannot be corrupted
- The Working of various integrated operation cannot be tampered with by the application program
- ADT ensures a robust data structure

List ADT

List is the collection of elements in sequential order. In memory we can store the list in two ways.

- **Sequential Memory Location - Array**
- **Pointer or links to associate the elements sequential – Linked List.**

THE LIST ADT

List is an ordered set of elements.

The general form of the list is

$A_1, A_2, A_3, \ldots\ldots, A_N$

A_1 - First element of the list

A_N - Last element of the list

N - Size of the list

If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.

Various operations performed on List

create an empty list
printList() – prints all elements in the list
construct a (deep) copy of a list
find(x) – returns the position of the first occurrence of x
remove(x) – removes x from the list if present
insert(x, position) – inserts x into the list at the specified position
isEmpty( ) – returns true if the list has no elements
makeEmpty( ) – removes all elements from the list
findKth(int k) – returns the element in the specified position

LINKED LIST

Definition

A **linked list** is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node.

Reason for Linked List

- 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.
- But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations.
- So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

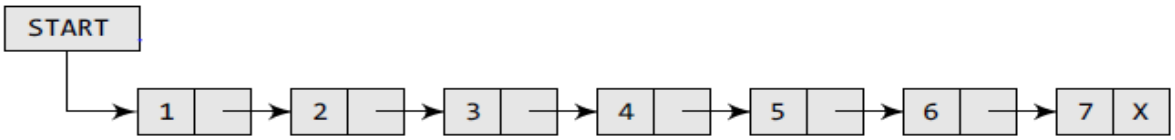
Advantages of using linked list

- Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.
- However, unlike an array, a linked list does not allow random access of data. 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.

Basic Terminologies

- A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes.
- Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

- Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.



- In above figure, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node.
- The left part of the node which contains data may include a simple data type, an array, or a structure.
- The right part of the node contains a pointer to the next node (or address of the next node in sequence).
- The last node will have no next node connected to it, so it will store a special value called NULL. In above figure, the NULL pointer is represented by X.
- While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list.
- Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.

Use of START pointer variable

- Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node.
- Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

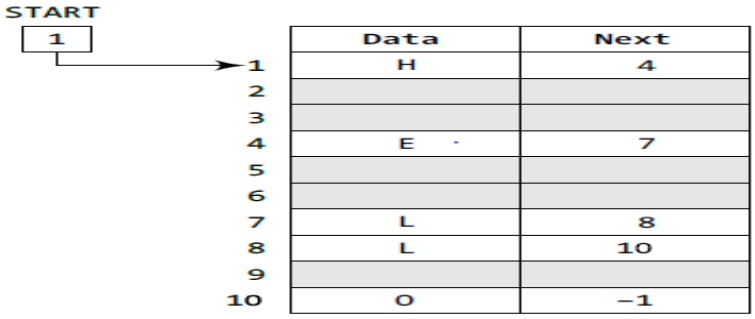
Sample Linked List code

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

Memory representation of Linked List

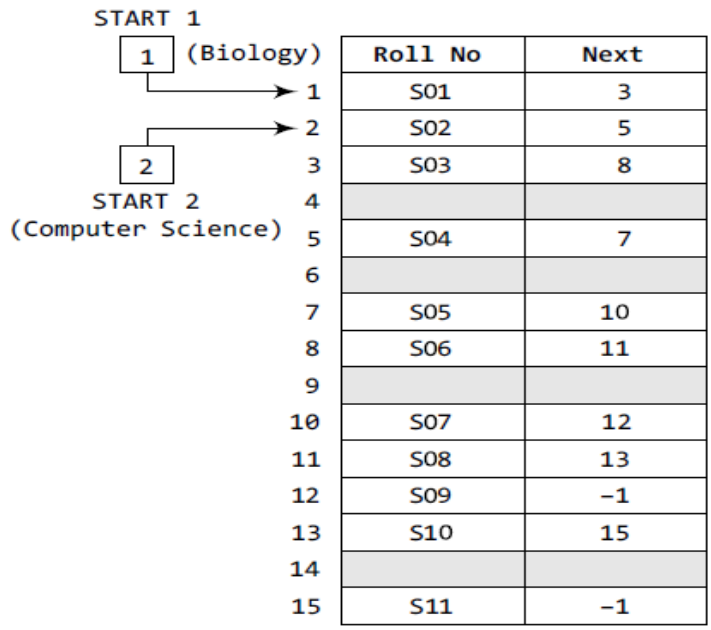
- Let us see how a linked list is maintained in the memory. In order to form a linked list, we need a structure called node which has two fields, DATA and NEXT.
- DATA will store the information part and NEXT will store the address of the next node in sequence.
- Consider the below figure, we can see that the variable START is used to store the address of the first node.
- Here, in this example, START = 1, so the first data is stored at address 1, which is H. The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.
- The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node.

- From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.
- When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO. Note that in the below figure shows a chunk of memory locations which range from 1 to 10. The shaded portion contains data for other applications.
- Remember that the nodes of a linked list need not be in consecutive memory locations.
- In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.



Example 1: Let us take another example to see how two linked lists are maintained together in the computer's memory.

- Let us consider the students of Class XI of Science group are asked to choose between Biology and Computer Science. Now, we will maintain two linked lists, one for each subject. That is, the first linked list will contain the roll numbers of all the students who have opted for Biology and the second list will contain the roll numbers of students who have chosen Computer Science.
- Now, look at the above Figure, two different linked lists are simultaneously maintained in the memory.
- There is no ambiguity in traversing through the list because each list maintains a separate Start pointer, which gives the address of the first node of their respective linked lists.
- The rest of the nodes are reached by looking at the value stored in the NEXT. By looking at the figure, we can conclude that roll numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11.
- Similarly, roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.We have already said that the DATA part of a node may contain just a single data item, an array, or a structure.



	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

Example 2: Let us take an example to see how a structure is maintained in a linked list that is stored in the memory.

Consider a scenario in which the roll number, name, aggregate, and grade of students are stored using linked lists. Now, we will see how the NEXT pointer is used to store the data alphabetically.

START 18

	Roll No	Name	Aggregate	Grade	Next
1	S01	Ram	78	Distinction	6
2	S02	Shyam	64	First division	14
3					
4	S03	Mohit	89	Outstanding	17
5					
6	S04	Rohit	77	Distinction	2
7	S05	Varun	86	Outstanding	10
8	S06	Karan	65	First division	12
9					
10	S07	Veena	54	Second division	-1
11	S08	Meera	67	First division	4
12	S09	Krish	45	Third division	13
13	S10	Kusum	91	Outstanding	11
14	S11	Silky	72	First division	7
15					
16					
17	S12	Monica	75	Distinction	1
18	S13	Ashish	63	First division	19
19	S14	Gaurav	61	First division	8

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.

Linked Lists Versus Arrays

arrays and linked lists are a linear collection of data elements

Array	Linked Lists
It stores its nodes in consecutive memory locations	It does not store its nodes in consecutive memory locations
It allows random access of data	It does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner
It cannot add any number of elements in the array	It can add any number of elements in the list

Memory Allocation and De-allocation for a Linked List

We have seen how a linked list is represented in the memory. If we want to add a node to an already existing linked list in the memory, we first find free space in the memory and then use it to store the information. For example, consider the linked list shown in Figure.

START = 1 (Biology)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	-1

(a)

START = 1 (Biology)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4	S12	75	-1
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	4

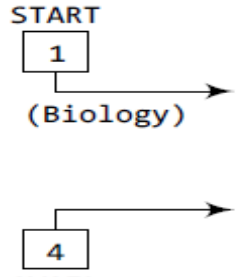
(b)

The linked list contains the roll number of students, marks obtained by them in Biology, and finally a NEXT field which stores the address of the next node in sequence. Now, if a new student joins the class and is asked to appear for the same test that the other students had taken, then the new student's marks should also be recorded in the linked list. For this purpose, we find a free space and store the information there. In above Figure, the grey shaded portion shows free space, and thus we have 4 memory locations available. We can use any one of them to store our data.

Now, the question is which part of the memory is available and which part is occupied? When we delete a node from a linked list, then who changes the status of the memory occupied by it from occupied to available? The answer is the operating system. We can say that the computer does it on its own without any intervention from the user or the programmer.

FREE POOL AND AVAIL POINTER VARIABLE

- The computer maintains a list of all free memory cells. This list of available space is called the free pool.
- We have seen that every linked list has a pointer variable START which stores the address of the first node of the list. Likewise, for the free pool (which is a linked list of all free memory cells), we have a pointer variable AVAIL which stores the address of the first free space.
- Let us revisit the memory representation of the linked list storing all the students' marks in Biology.
- Now, when a new student's record has to be added, the memory address pointed by AVAIL will be taken and used to store the desired information.
- After the insertion, the next available free space's address will be stored in AVAIL.



	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			6
5	S04	98	7
6			9
7	S05	55	8
8	S06	34	10
9			14
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			-1
15	S11	56	-1

- For example, in the above Figure when the first free memory space is utilized for inserting the new node, AVAIL will be set to contain address 6.
- This was all about inserting a new node in an already existing linked list. Now, we will discuss deleting a node or the entire linked list.
- When we delete a particular node from an existing linked list or delete the entire linked list, the space occupied by it must be given back to the free pool so that the memory can be reused by some other program that needs memory space.
- The operating system does this task of adding the freed memory to the free pool. The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space.
- The operating system scans through all the memory cells and marks those cells that are being used by some program.
- Then it collects all the cells which are not being used and adds their address to the free pool, so that these cells can be reused by other programs. This process is called garbage collection.

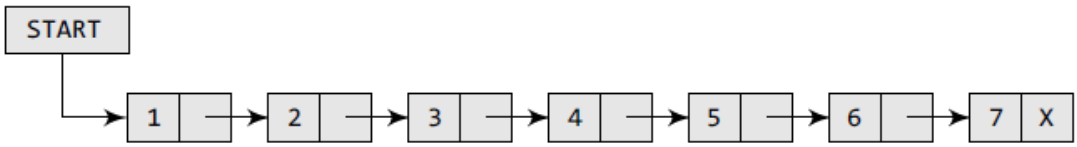
TYPES OF LINKED LISTS

There are different types of linked lists. They are
1.Singly Linked List
2.Doubly Linked list
3.Circular Linked List
4.Circular Doubly Linked List

SINGLY LINKED LISTS

Definition

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



Traversing a Linked List

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.

Algorithm for traversing a linked list

```
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
```

- In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list.
- Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL.
- In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR.
- In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

Algorithm to print the number of nodes in 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
```

We will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed.

Searching for a Value in a Linked List

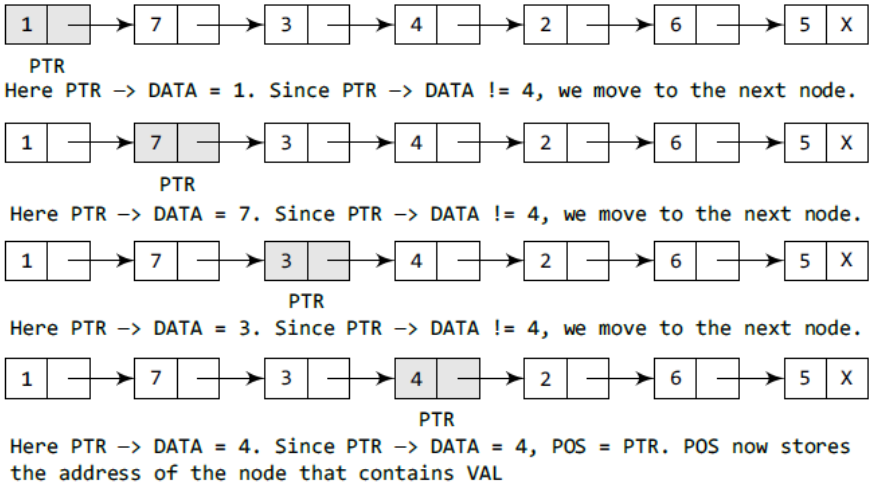
Searching a linked list means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
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
```

- In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node.
- In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made.
- If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm.
- However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Example : Illustration of Searching algorithm

Consider the linked list shown in Figure. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.



Inserting a New Node in a Linked List

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.

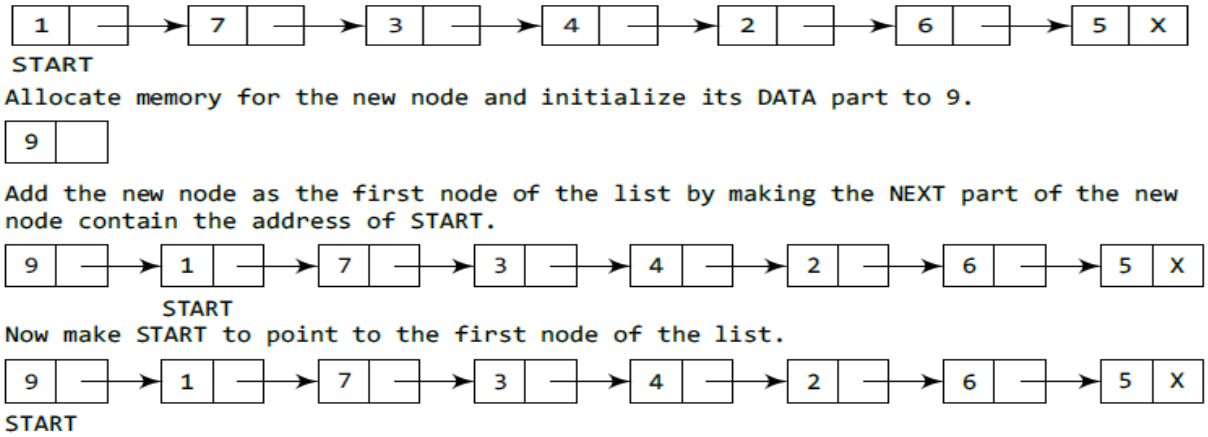
Case 4: The new node is inserted before a given node.

OVERFLOW

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.

Case 1: Inserting a Node at the Beginning of a Linked List

add a new node with data 9 and add it as the first node of the list.



Algorithm to insert a new node at the beginning

```
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
```

- In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if a free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.

Note the following two steps:

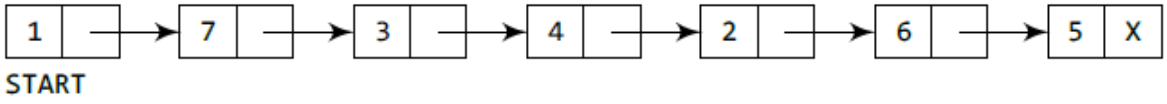
Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

These steps allocate memory for the new node. In C, there are functions like malloc(), alloc, and calloc() which automatically do the memory allocation on behalf of the user.

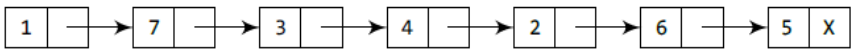
Case 2: Inserting a Node at the End of a Linked List

add a new node with data 9 as the last node of the list

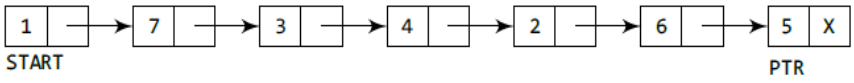


Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

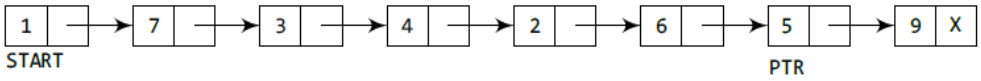
Take a pointer variable PTR which points to START.



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



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.



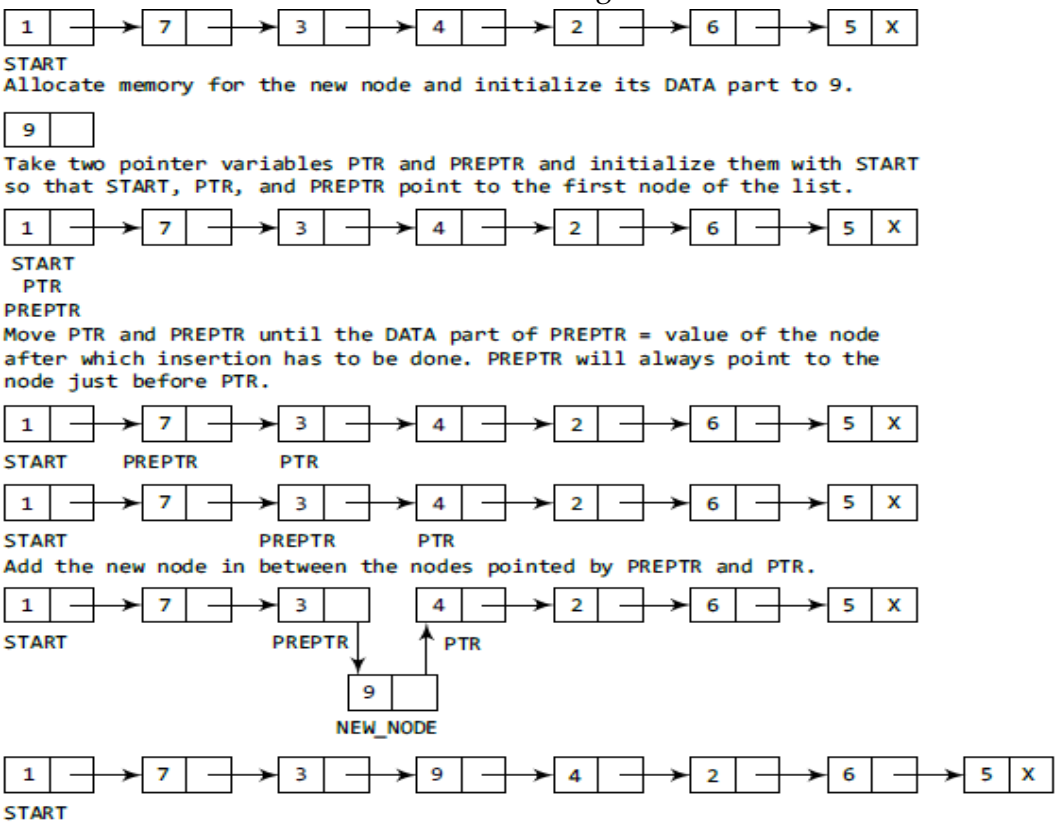
Algorithm to insert a new node at the end

```
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
```

This algorithm to insert a new node at the end of a linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

Case 3: Inserting a Node After a Given Node in a Linked List

Add a new node with value 9 after the node containing data 3.



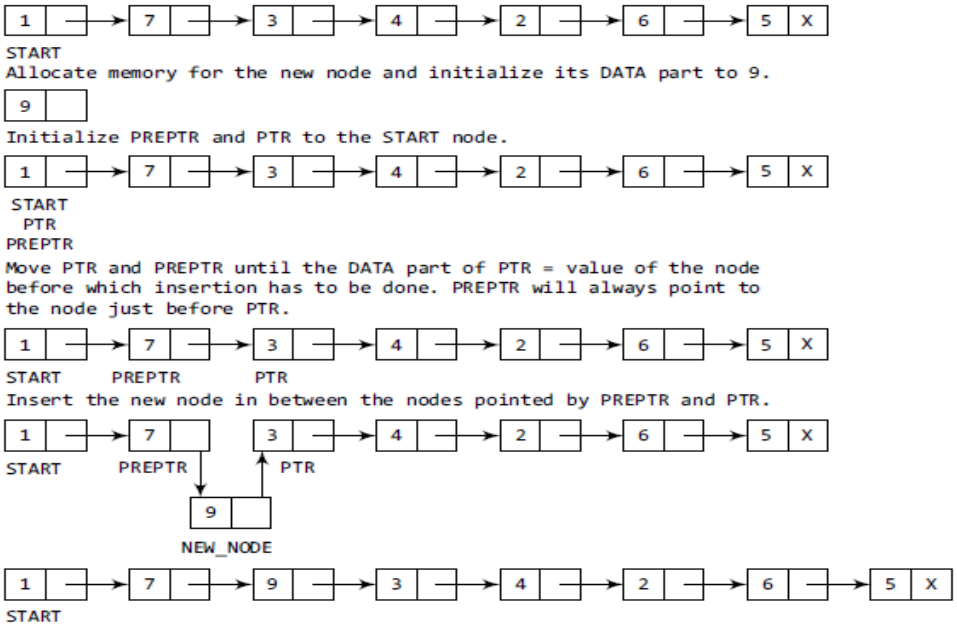
Algorithm to insert a new node after a node that has value NUM

```
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
```

- In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR.
- So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

Case 4: Inserting a Node Before a Given Node in a Linked List

Add a new node with value 9 before the node containing 3.



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

```
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 PTR->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
```

- In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted before this node.
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.

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.

Case 1: The first node is deleted.

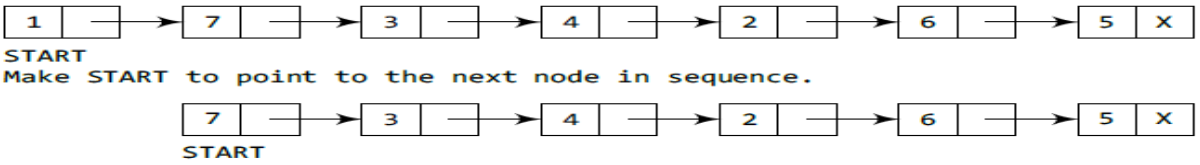
Case 2: The last node is deleted.

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.
- The memory is returned to the free pool so that it can be used to store other programs and data.
- Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Case 1: The first node is deleted.

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



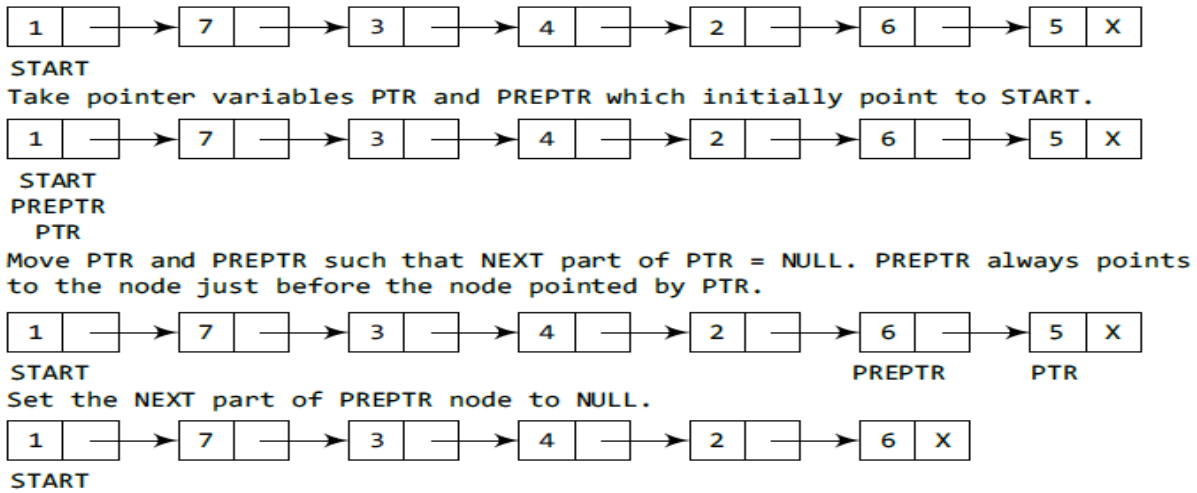
Algorithm to delete the first node

```
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
```

- In Step 1, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Case 2: The last node is deleted.

We want to delete the last node from the linked list, then the following changes will be done in the linked list.



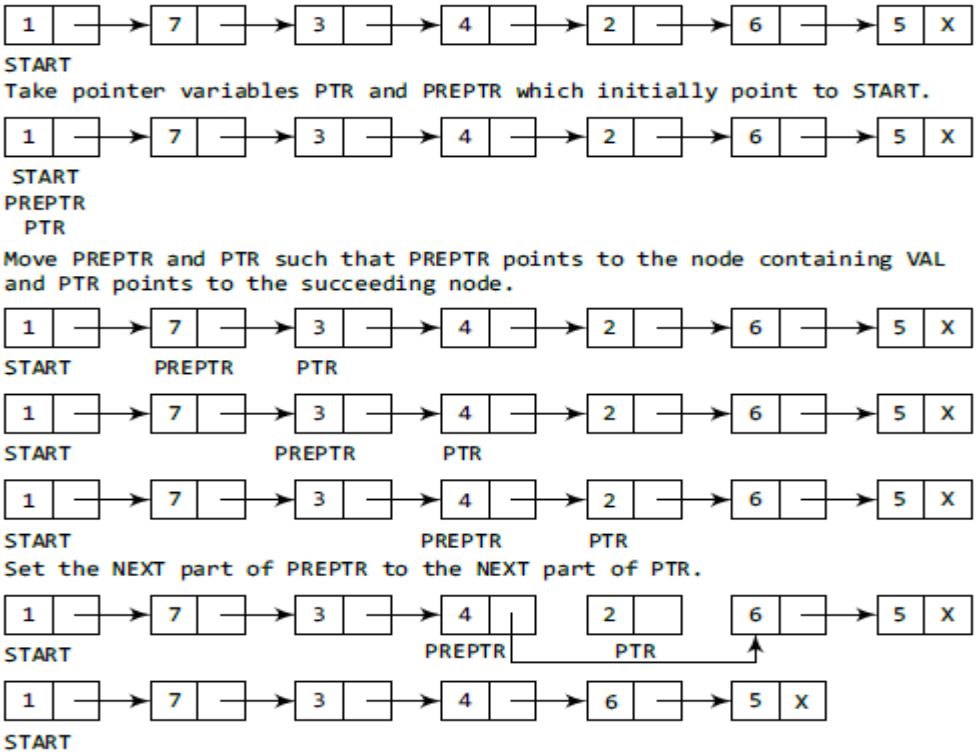
Algorithm to delete the last node

```
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
```

- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned back to the free pool.

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

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.



Algorithm to delete the node after a given node

```
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
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it.
- The memory of the node succeeding the given node is freed and returned back to the free pool.

CIRCULAR LINKED LISTs

Definition

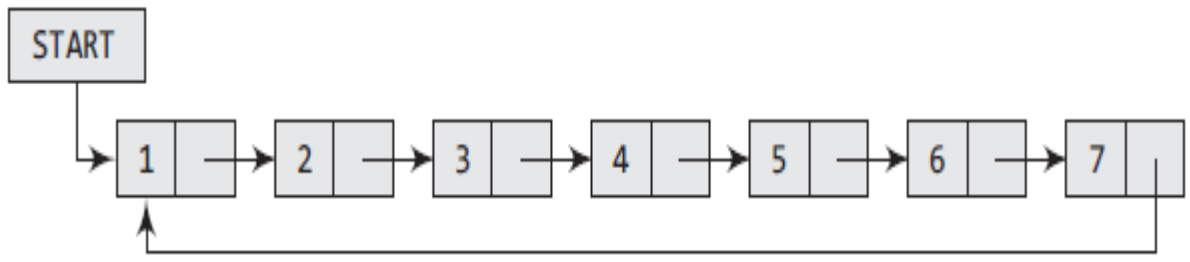
In a circular linked list is similar to singly linked list except that the last node contains a pointer to the first node of the list.

Types

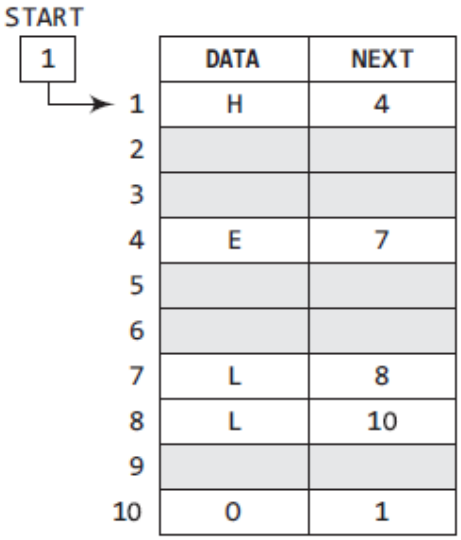
- Circular singly linked list
- Circular doubly linked list.

Traversing in circular linked list

While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.



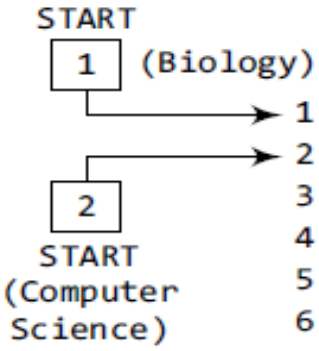
Memory representation of a circular linked list



	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

- We can traverse the list until we find the NEXT entry that contains the address of the first node of the list.
- This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.
- When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above figure stores characters that when put together form the word HELLO.

Memory representation of two circular linked lists stored in the memory



	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	2

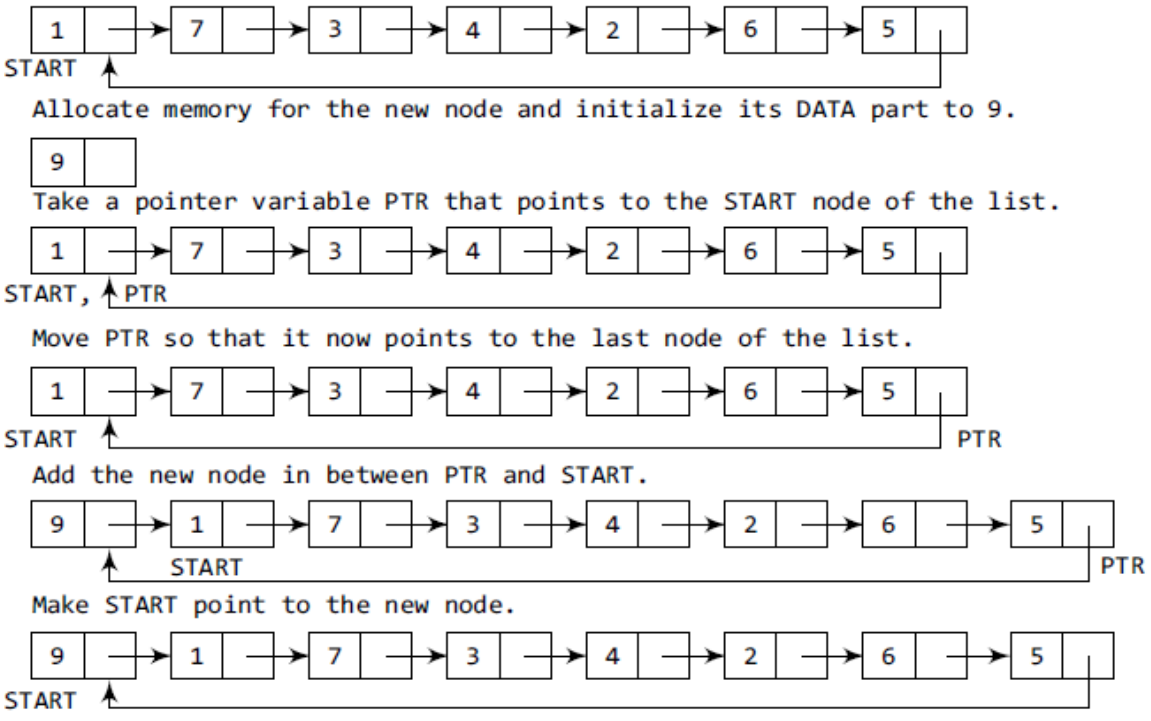
- Two different linked lists are simultaneously maintained in the memory.
- There is no ambiguity in traversing through the list because each list maintains a separate START pointer which gives the address of the first node of the respective linked list.
- The remaining nodes are reached by looking at the value stored in NEXT.
- By looking at the figure, we can conclude that the roll numbers of the students who have opted for Biology are S01, S03, S06, S08, S10, and S11.
- Similarly, the roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.

Inserting a New Node in a Circular Linked List

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.

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

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.



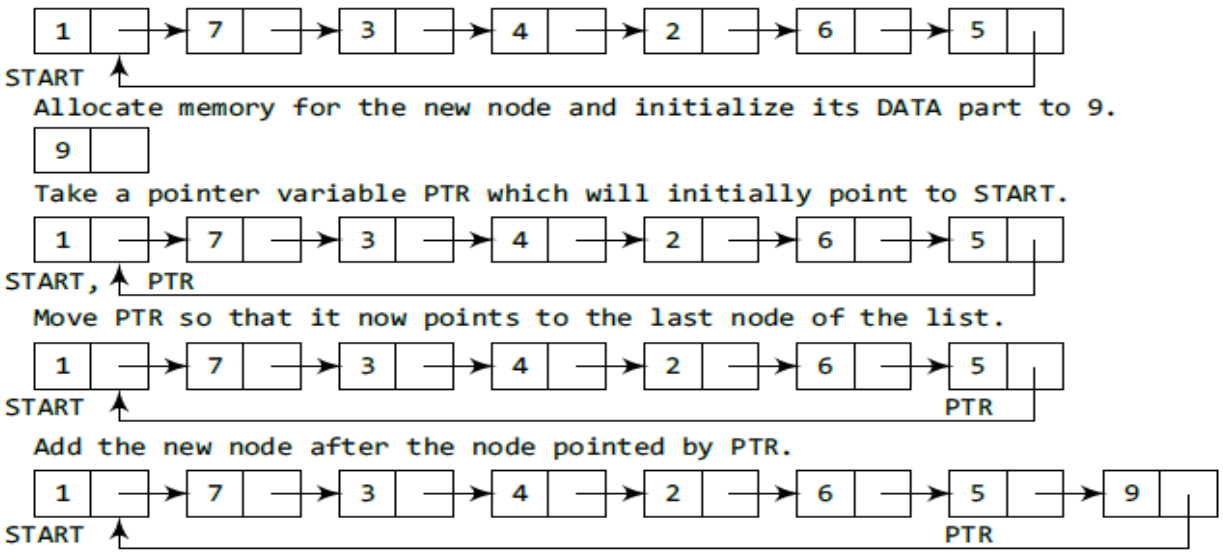
Algorithm to insert a new node at the beginning

```
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
```

- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.
- While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list.
- Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

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

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.



Algorithm to insert a new node at the end

```
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
```

In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.

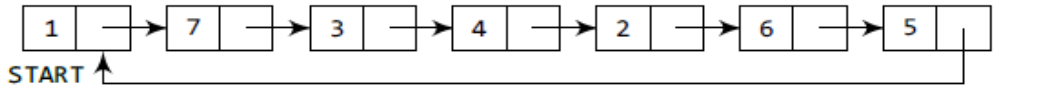
Deleting a Node from a Circular Linked List

Case 1: The first node is deleted.

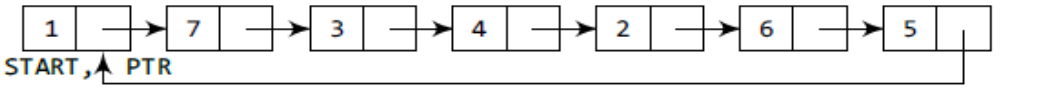
Case 2: The last node is deleted.

Case 1: The first node is deleted.

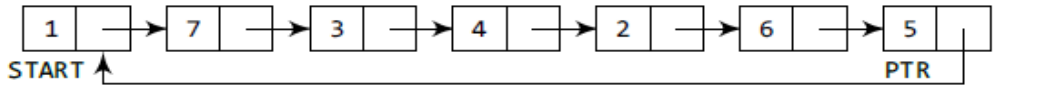
When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



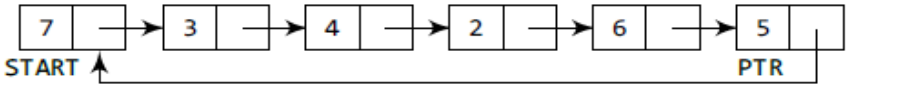
Take a variable PTR and make it point to the START node of the list.



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.

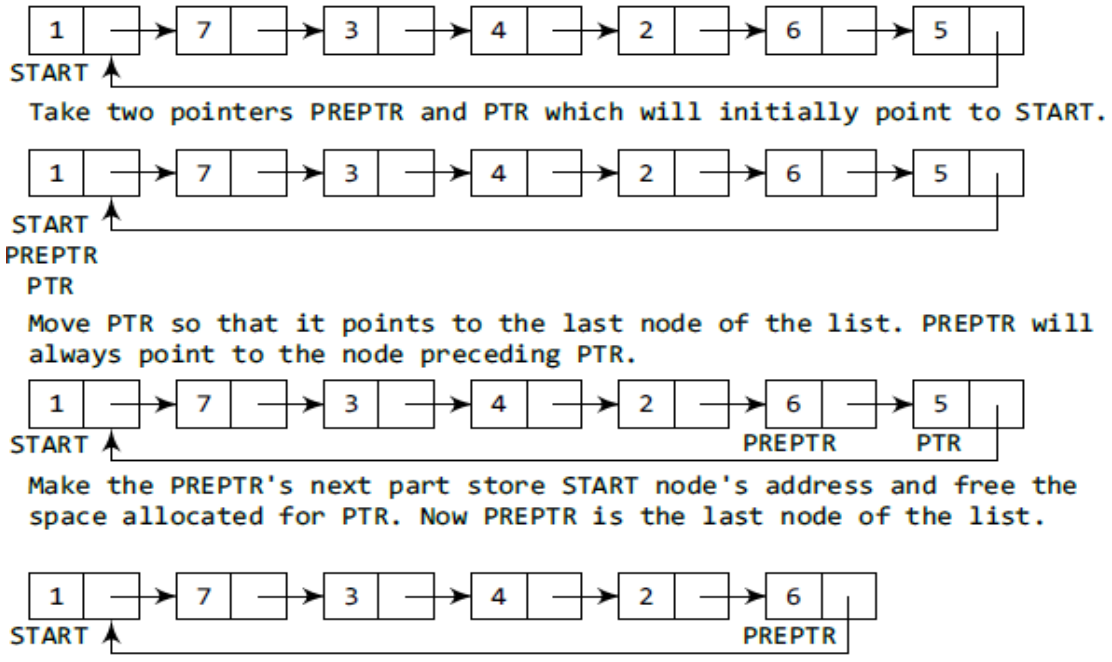


Algorithm to delete the first node

```
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
```

- In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node.
- In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list.
- In Step 6, the memory occupied by the first node is freed.
- Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

Case 2: The last node is deleted.



we want to delete the last node from the linked list, then the following changes will be done in the linked list.

Algorithm to delete the last node

```
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
```

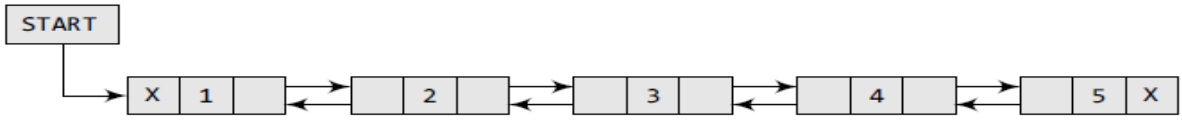
- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR.
- Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.

DOUBLY LINKED LISTS

Definition

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 as shown in Figure



The structure of a doubly linked list can be given as,

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

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.

Doubly linked list calls for more space per node and more expensive basic operations. However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a doubly linked list is that it makes searching twice as efficient.

Memory representation of a 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

- Variable START is used to store the address of the first node.
- In this example, START = 1, so the first data is stored at address 1, which is H.
- Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field.
- We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list.
- When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Inserting a New Node in a Doubly Linked List

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

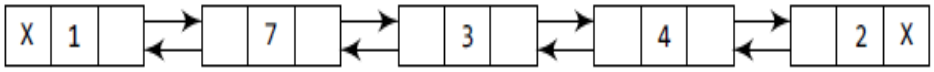
Case 2: The new node is inserted at the end.

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

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

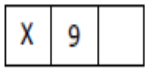
Case 1: The new node is inserted at the beginning.

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.

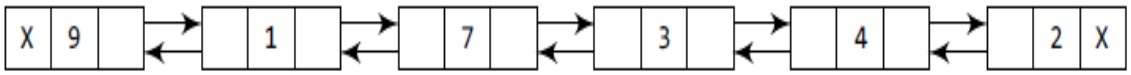


START

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

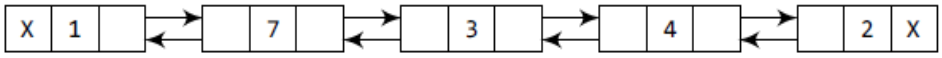
Algorithm to insert a new node at the beginning

```
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
```

- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

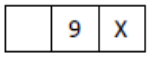
Case 2: The new node is inserted at the end.

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.

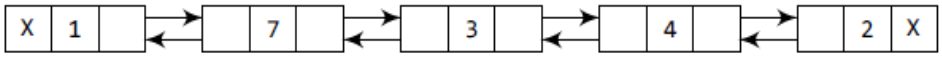


START

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

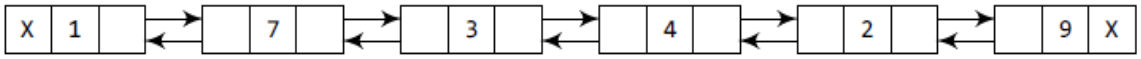


Take a pointer variable PTR and make it point to the first node of the list.



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

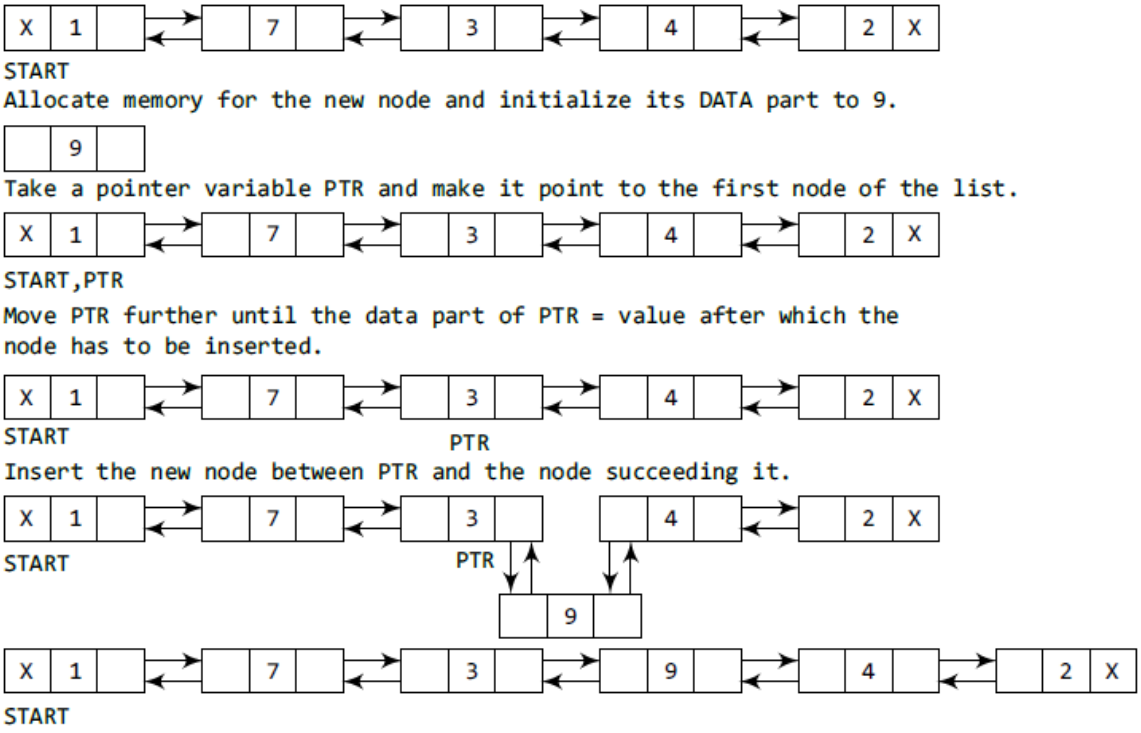
Algorithm to insert a new node at the end

```
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
```

- In Step 6, we take a pointer variable PTR and initialize it with START.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

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

we want to add a new node with value 9 after the node containing 3.



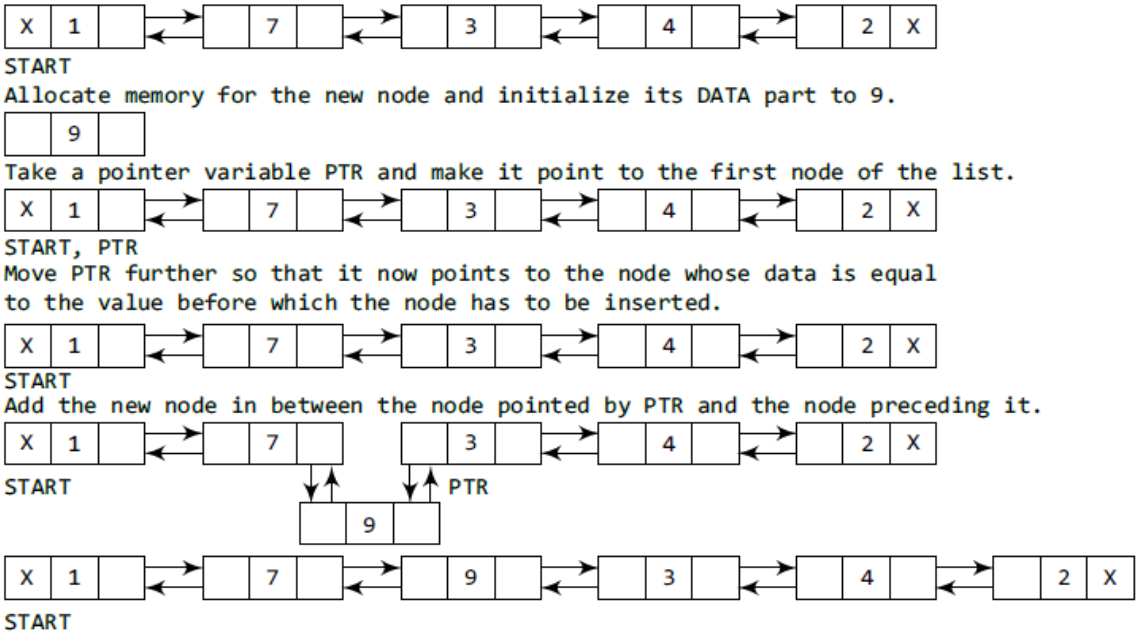
Algorithm to insert a new node after a given node

```
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
```

- In Step 5, we take a pointer PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

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

we want to add a new node with value 9 before the node containing 3.



Algorithm to insert a new node before a given node

```
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
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted before this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

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

Case 1: The first node is deleted.

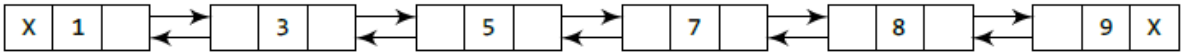
Case 2: The last node is deleted.

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

Case 4: The node before a given node is deleted.

Case 1: The first node is deleted.

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

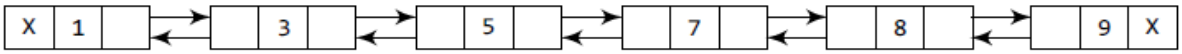
Algorithm to delete the first node

```
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
```

- In Step 1 of the algorithm, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

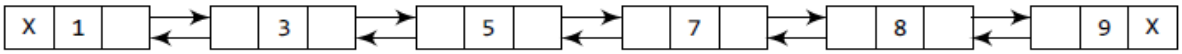
Case 2: The last node is deleted.

we want to delete the last node from the linked list, then the following changes will be done in the linked list.



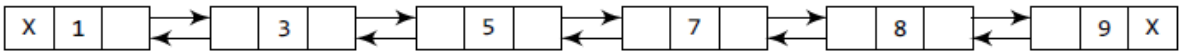
START

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



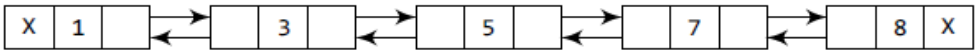
START,PTR

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

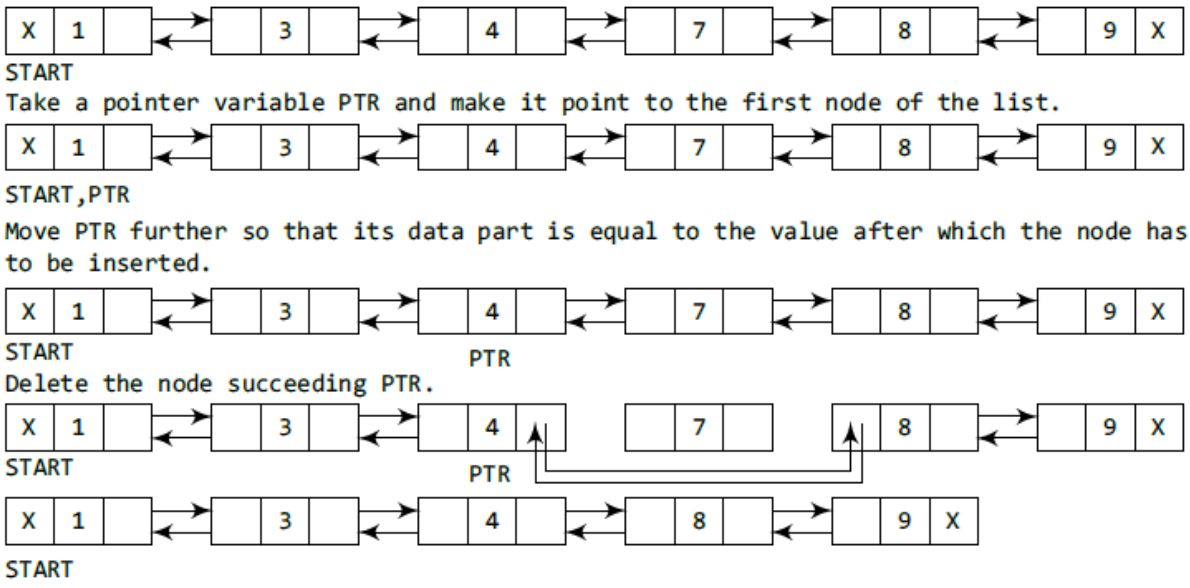
Algorithm to delete the last node

```
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
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node.
- Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node.
- To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

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

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.



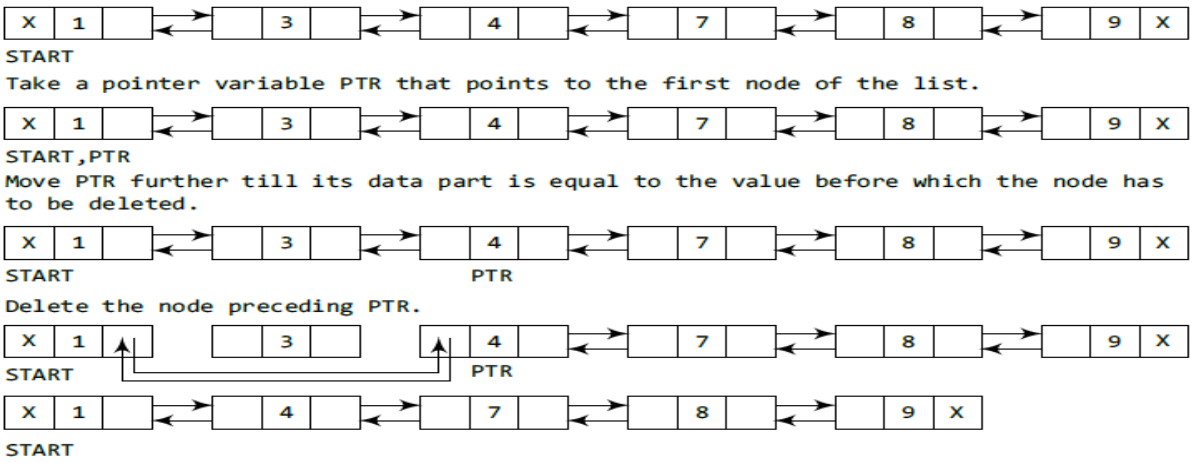
Algorithm to delete a node after a given node

```
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
```

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node. Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

Case 4: The node before a given node is deleted.

Suppose we want to delete the node preceding the node with value 4



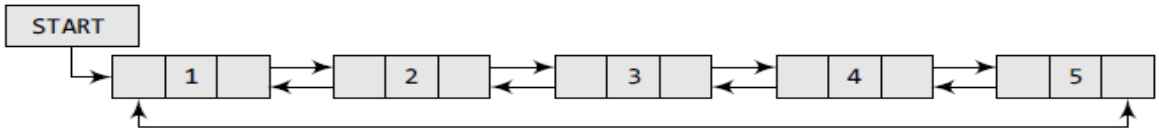
Algorithm to delete a node before a given node

```
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 -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- The while loop traverses through the linked list to reach the desired node.
- Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node preceding the node which comes before PTR.
- The memory of the node preceding PTR is freed and returned to the free pool.

CIRCULAR DOUBLY LINKED LISTs

A circular doubly linked list or a circular 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.
The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node. Rather, the next field of the last node stores the address of the first node of the list, i.e., START. Similarly, the previous field of the first field stores the address of the last node.



- Since a circular doubly linked list contains three parts in its structure, it calls for more space per node and more expensive basic operations.
- However, a circular doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The main advantage of using a circular doubly linked list is that it makes search operation twice as efficient.

Memory representation of a circular doubly linked list

START = 1

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

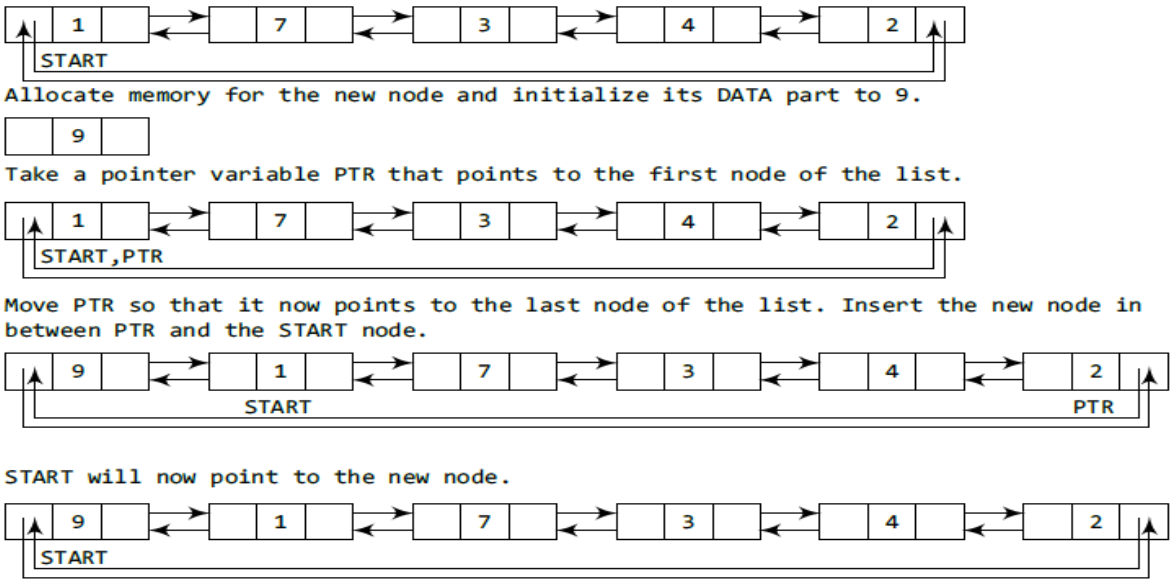
- In the above figure, we see that a variable START is used to store the address of the first node. Here in this example, START = 1, so the first data is stored at address 1, which is H.
- Since this is the first node, it stores the address of the last node of the list in its previous field. The corresponding NEXT stores the address of the next node, which is 3.
- So, we will look at address 3 to fetch the next data item. The previous field will contain the address of the first node.
- The second data element obtained from address 3 is E. We repeat this procedure until we reach a position where the NEXT entry stores the address of the first element of the list.
- This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.

Inserting a New Node in a Circular Doubly Linked List

how a new node is added into an already existing circular doubly linked list.
Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.

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

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.



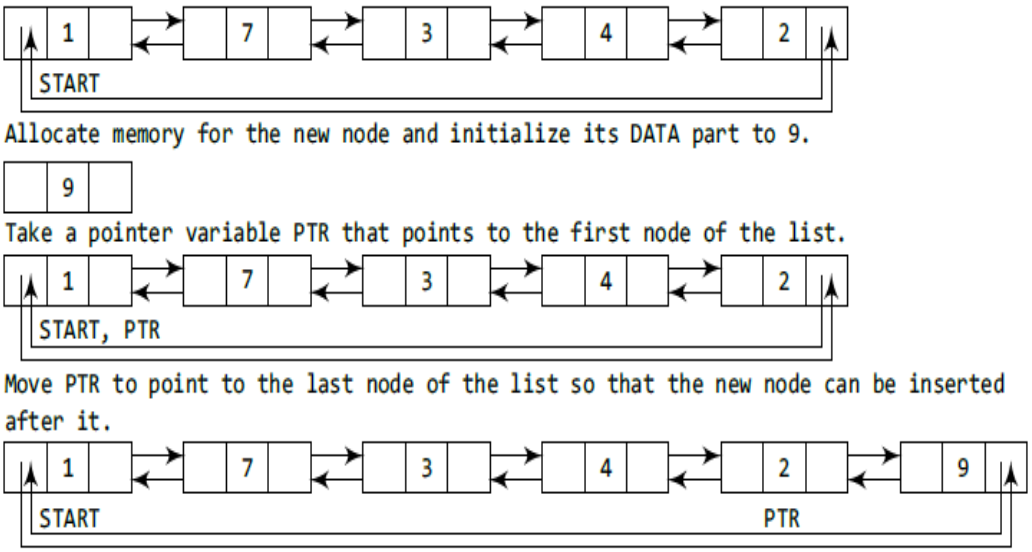
Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 13
        [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:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET PTR -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET NEW_NODE -> NEXT = START
Step 11: SET START -> PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, we allocate space for the new node.
- Set its data part with the given VAL and its next part is initialized with the address of the first node of the list, which is stored in START.
- Now since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE. Since it is a circular doubly linked list, the PREV field of the NEW_NODE is set to contain the address of the last node.

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

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.



Algorithm to insert a new node at the end

```
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 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: SET NEW_NODE -> PREV = PTR
Step 11: SET START -> PREV = NEW_NODE
Step 12: EXIT
```

- In Step 6, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

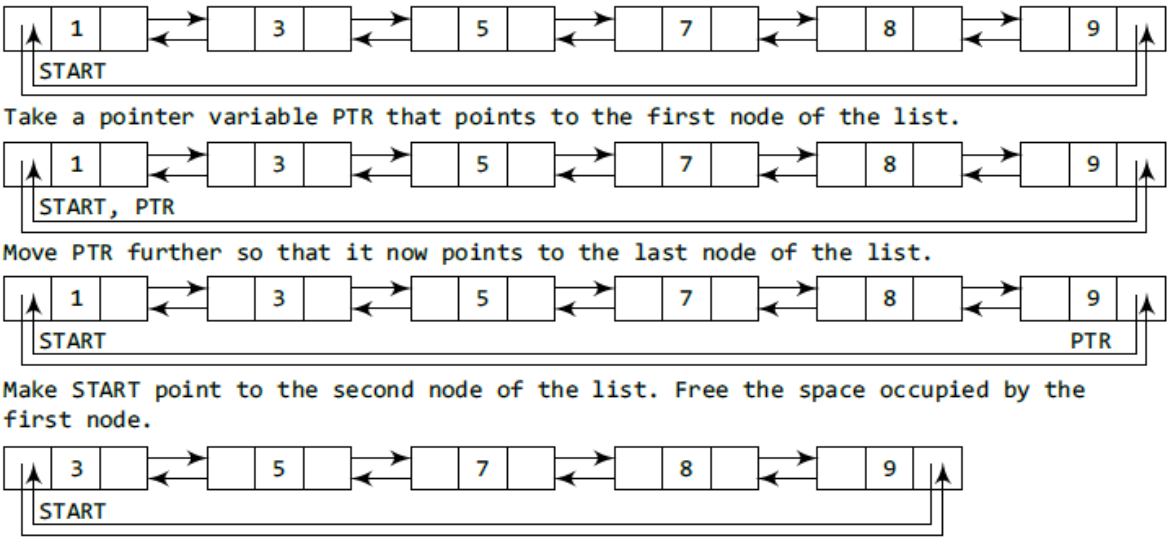
Deleting a Node from a Circular Doubly Linked List

we will see how a node is deleted from an already existing circular doubly linked list.

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

Case 1: The first node is deleted.

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



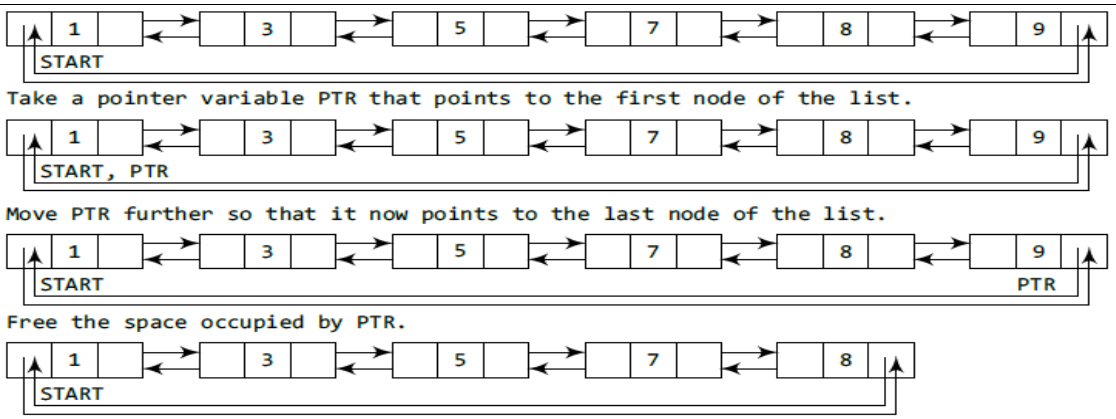
Algorithm to delete the first node

```
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: SET START -> NEXT -> PREV = PTR
Step 7: FREE START
Step 8: SET START = PTR -> NEXT
```

- In Step 1 of the algorithm, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- The while loop traverses through the list to reach the last node. Once we reach the last node, the NEXT pointer of PTR is set to contain the address of the node that succeeds START.
- Finally, START is made to point to the next node in the sequence and the memory occupied by the first node of the list is freed and returned to the free pool.

Case 2: The last node is deleted.

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



Algorithm to delete the last node

```
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 -> PREV -> NEXT = START
Step 6: SET START -> PREV = PTR -> PREV
Step 7: FREE PTR
Step 8: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node.
- Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node.
- To delete the last node, we simply have to set the next field of the second last node to contain the address of START, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.

APPLICATIONS OF LINKED LISTS

Linked lists can be used to represent polynomials and the different operations that can be performed on them

Polynomial Representation

Consider a polynomial $6x^3 + 9x^2 + 7x + 1$. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively. Every term of a polynomial can be represented as a node of the linked list

Linked representation of a polynomial