Linked Lists

Divyashikha Sethia
Department of CSE, DTU
divyashikha@dtu.ac.in

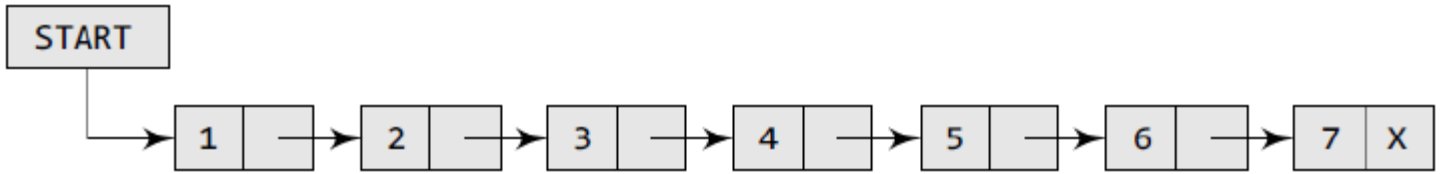
6.1 Basic Terminologies

Linked list- linear collection of data elements

- Data elements are called nodes
- A linked list is collection of data elements called nodes in which linear representation is given by links from one node to next node.

Basic Terminologies

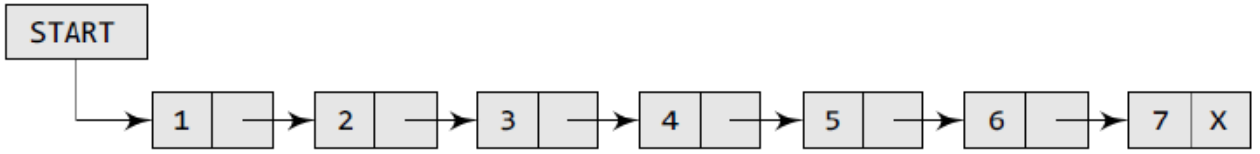
- Acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



Simple linked list

Basic Terminologies

- Every node contains two parts: Data and pointer to the next node of the same type
- Last node will have no next node connected to it, so it will store a special value called NULL.
- Pointer variable START: stores address of first node in list.



Simple linked list

Basic Terminologies

- Traverse entire list using START; the next part of the first node stores address of its succeeding node….
- If START = NULL => linked list is empty (no nodes).
- C implementation for linked list: (integer data)

Basic Terminologies

- struct node

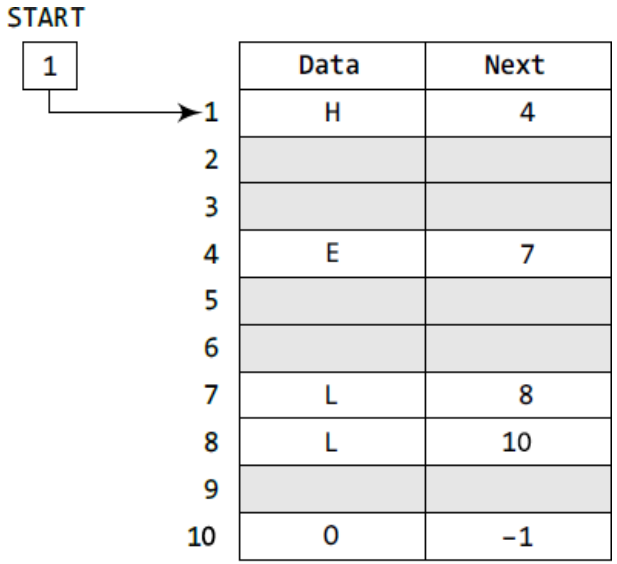
int data;

struct node *next;

-

Basic Terminologies

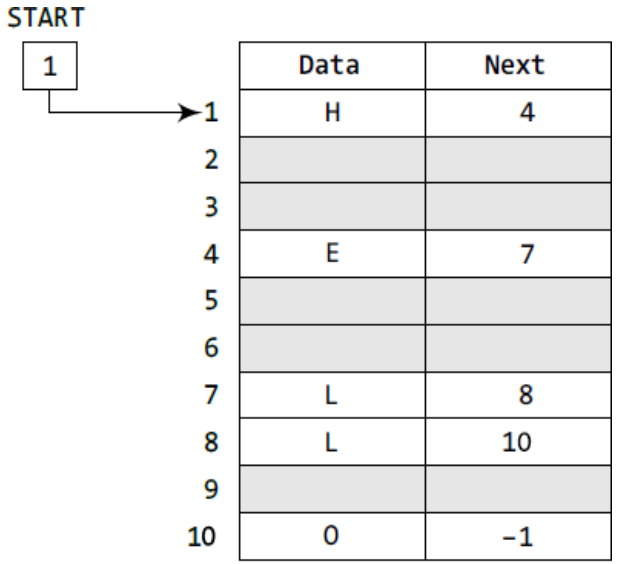
- Need structure called *node* with two fields, DATA and NEXT.
- DATA stores information part
- NEXT stores the address of the next node in sequence.
- START stores address of first node.
- START = 1, => first data (H) stored at address 1
- NEXT stores address of next node, which is 4.
- Look at address 4 to fetch next data item.



START pointing to the first element of the linked list in the memory

Basic Terminologies

- Second data element obtained from address 4 is E.
- Corresponding NEXT is 7
- Fetch data 7
- ::::
- Last node NEXT entry contains –1 or NULL, end of the linked list.
- Characters put together form the word HELLO.



START pointing to the first element of the linked list in the memory

Basic Terminologies

- example :two linked lists are maintained together in the computer's memory. For example, the students of Class XI of Science group are asked to choose between Biology and Computer Science.
- Maintain two linked lists, one for each subject. 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.

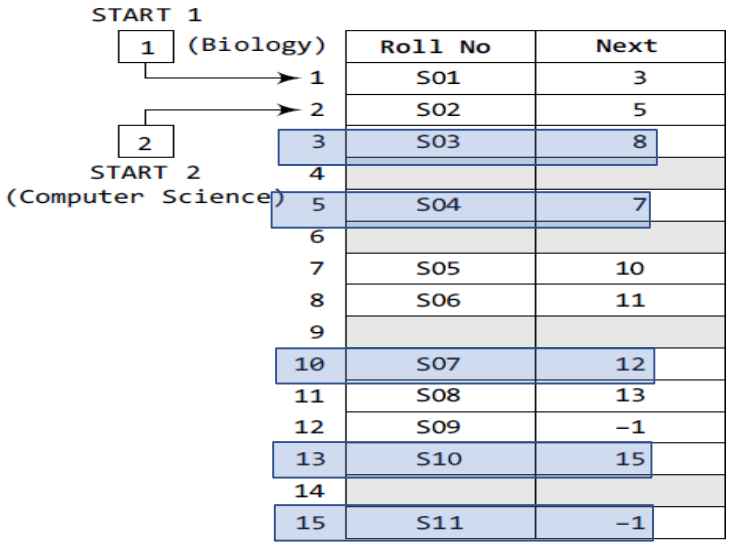
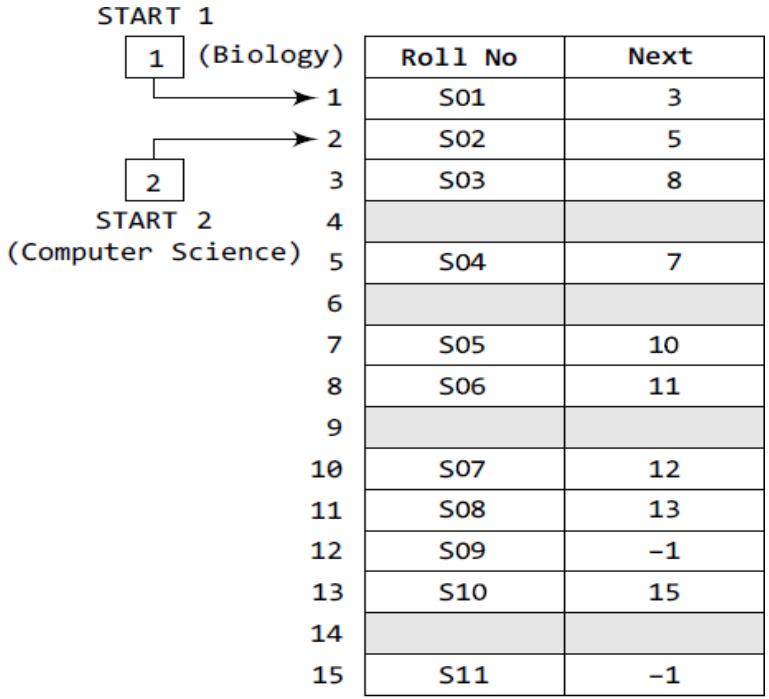


Figure 6.3 Two linked lists which are simultaneously maintained in the memory

Basic Terminologies

- 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.
- 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.



	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

Two linked lists which are simultaneously maintained in the memory

6.1.2 Linked Lists vs Arrays

- Both arrays and linked lists are linear collection of data elements.
- Unlike an array, a linked list does not store its nodes in consecutive memory locations.

Linked Lists vs Arrays

- Linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner. While array, insertions and deletions can be done at any point in a constant time.

Linked Lists vs Arrays

- Advantage of a linked list over an array: we can add any number of elements in the list.
- Linked lists provide an efficient way of storing related data and performing basic operation such as insertion, deletion, and updation of information at the cost of extra space required for storing the address of next nodes

6.1.3 Memory Allocation and Deallocation for a Linked List

- To add a node to an already existing linked list in the memory: -find free space in the memory and store the information.
- Find free space and store information there. Grey shaded portion: free space (4 memory locations)
- List of available space is called the *free pool*.

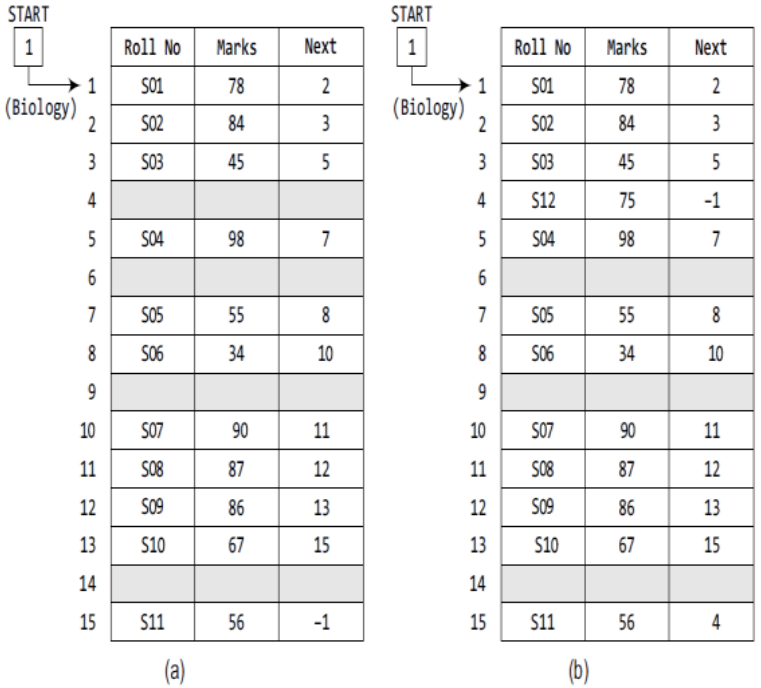


Figure 6.5 (a) Students' linked list and (b) linked list after the insertion of new student's record

Memory Allocation and Deallocation for a Linked List

A pointer variable AVAIL which stores the address of the first free space.

- The memory representation of the linked list storing all the students'marks in Biology.

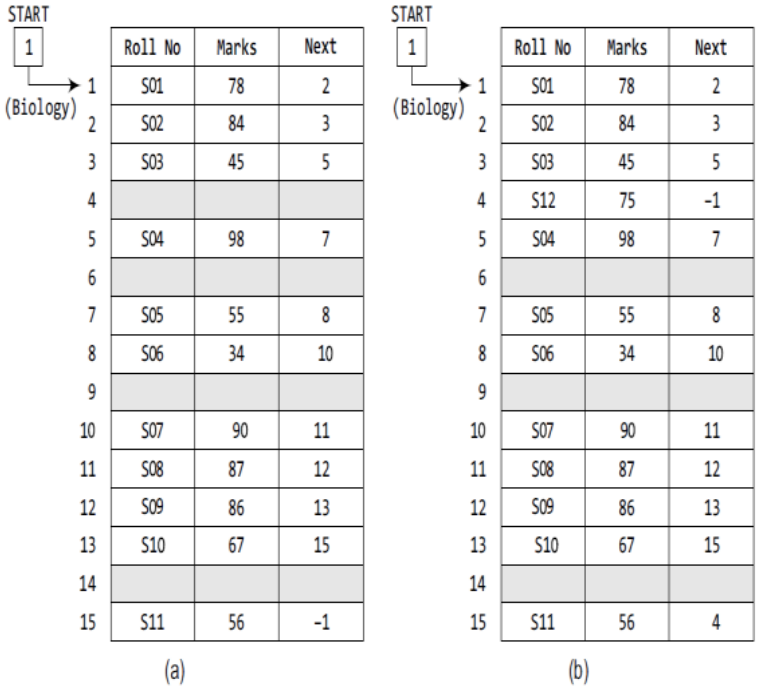


Figure 6.5 (a) Students' linked list and (b) linked list after the insertion of new student's record

Memory Allocation and Deallocation for a Linked List

- Now, when A new student's record has to added, the memory address pointed by AVAIL will be taken and used to store the desired information. After the insertion, th next available free space's address will b stored in AVAIL.
- when the first free memory space is utilize for inserting the new node, AVAIL will be s to contain address 6.

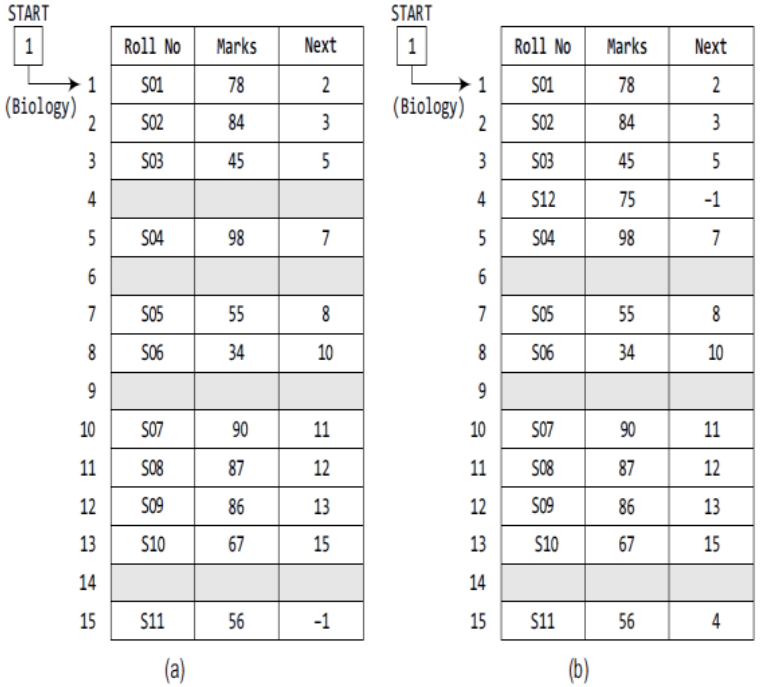


Figure 6.5 (a) Students' linked list and (b) linked list after the insertion of new student's record

Memory Allocation and Deallocation for a Linked List

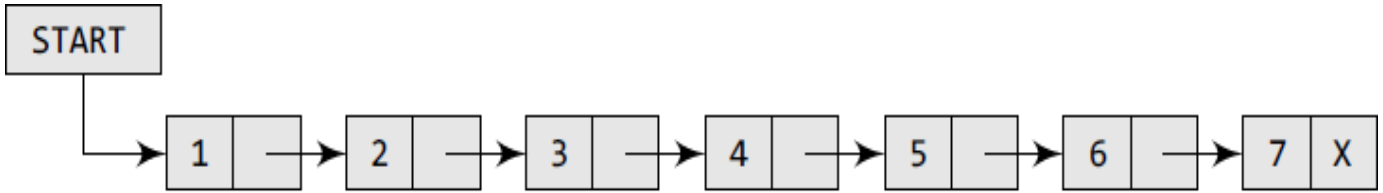
- Delete a particular node from existing linked list/ delete entire linked list: release space occupied by it back to free pool for memory reuse by another program
- OS adds freed memory to free pool when it finds CPU idle or whenever programs are falling short of memory space.
- OS scans through all memory cells and marks those cells that are being used by some program.

  - Collects all cells which are not being used and adds their address to the free pool: process is called *garbage collection*

6.2 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.

- A singly linked list allows traversal of data only in one way.



Simple linked list

Traversing a Linked List

- Accessing nodes of list in order to perform some processing.
- START stores address of first node of list.
- End of list marked by NULL /–1 in NEXT field of last node.
- For traversing linked list, use pointer variable PTR which points to node that is currently being accessed.
- The algorithm to traverse a linked list is shown:

```
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
```

Traversing a Linked List…

1. Initialize PTR with address of START. PTR points to first node of linked list.
2. Start while loop which is repeated till PTR processes the last node, that is until it encounters NULL.
3. Apply process (e.g., print) to current node, that is, the node pointed by PTR.
4. Move to next node by making the PTR variable point to the node whose address is stored in the NEXT field.

```
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
```

Traversing a Linked List…

- **An algorithm to count the number of nodes in a linked list.**
- 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. 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
```

6.2.2 Searching for a Value in a Linked List

- Find particular element in linked list.
- Algorithm returns address of node that contains the value.

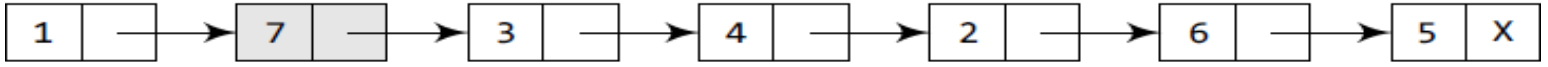
```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:       IF VAL = PTR -> DATA
                    SET POS = PTR
                    Go To Step 5
              ELSE
                    SET PTR = PTR -> NEXT
              [END OF IF]
        [END OF LOOP]
Step 4: SET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

Searching for a Value in a Linked List

- If we have VAL = 4, then the flow of the algorithm is :

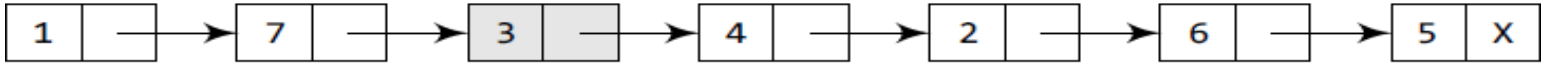
PTR

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



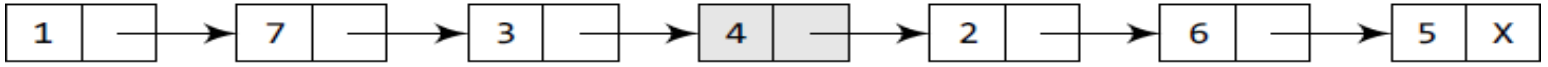
PTR

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



PTR

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



PTR

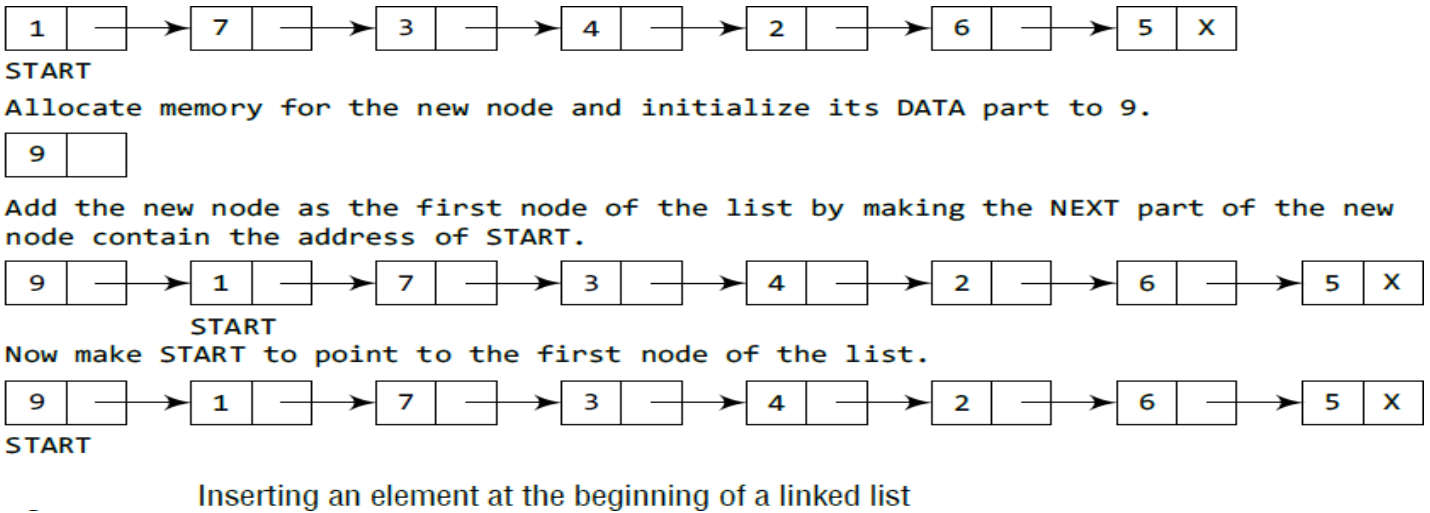
Here PTR –> DATA = 4. Since PTR –> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

6.2.3 Inserting a New Node in a Linked List

- 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: AVAIL = NULL or no free memory cell is present in system.
  - For this condition program must give appropriate message.

Inserting New Node in Link List

- ***Inserting a Node at the Beginning of a Linked List***
- 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.



Inserting an element at the beginning of a linked list

Inserting New Node in Link List

In Step 1, 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.

```
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
```

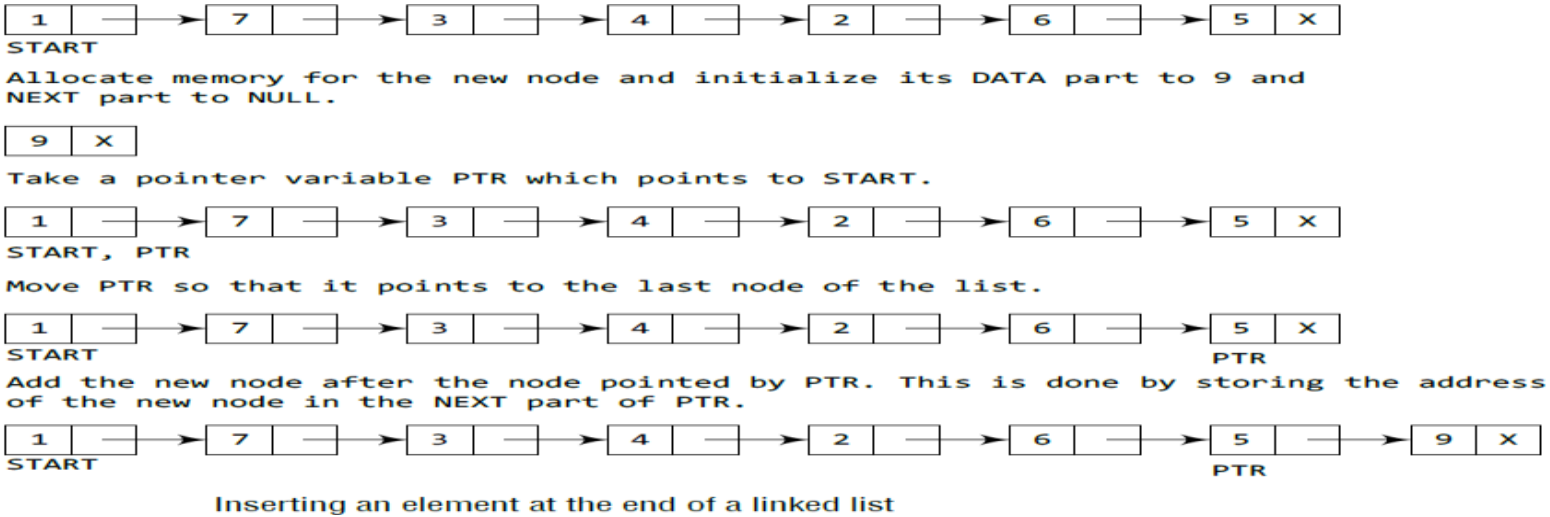
Inserting New Node in Link List

- 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.
- 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.

Inserting New Node in Link List

Inserting a Node at the End of a Linked List

•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.



Inserting an element at the end of a linked list

Inserting New Node in Link 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. The NEXT field of the new node contains NULL, which signifies the end of 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 = 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
```

Algorithm to insert a new node at the end

Inserting New Node in Link List

- **Inserting a Node After a Given Node in a Linked List**
- To add a new node with value 9 after the node containing data 3.
- Step 5, Initialize PTR: START (first node)
- PREPTR: store address of node preceding PTR.
- Initially, PREPTR = PTR.
- while loop, traverse through linked list to reach node with value equal to NUM.

Once we reach this node, in Steps 10 and 11, change NEXT pointers so that new node is inserted after desired 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: 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
```

Insert new node after node with value NUM

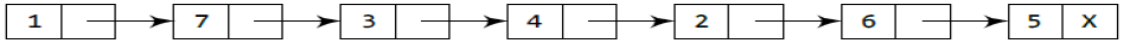
Inserting New Node in Link List

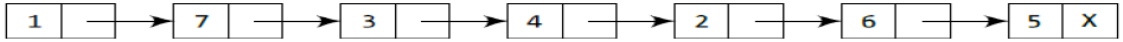
START
Allocate memory for the new node and initialize its DATA part to 9.

Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.

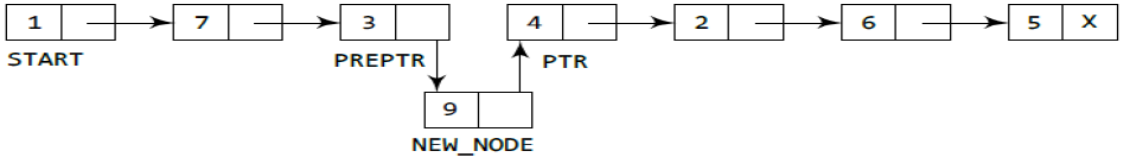
START
PTR
PREPTR

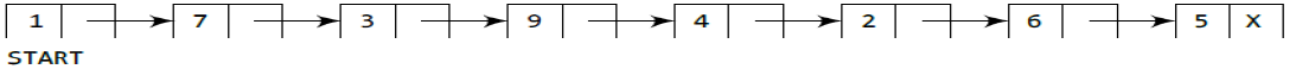
Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.





Add the new node in between the nodes pointed by PREPTR and PTR.





Insert new node 9 after node with value NUM=3

Inserting New Node in Link List

- **Inserting a Node Before a Given Node in a Linked List**
- To add a new node with value 9 before the node containing 3
- 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. 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.

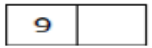
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR->DATA != NUM
Step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

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

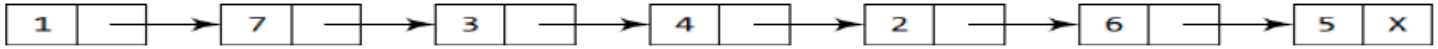
Inserting New Node in Link List

START

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



Initialize PREPTR and PTR to the START node.

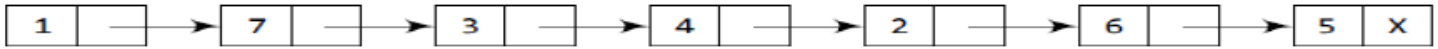


START

PTR

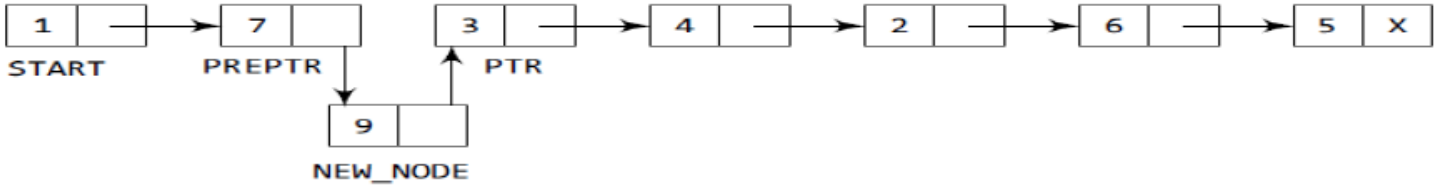
PREPTR

Move PTR and PREPTR until the DATA part of PTR = value of the node before which insertion has to be done. PREPTR will always point to the node just before PTR.



START PREPTR PTR

Insert the new node in between the nodes pointed by PREPTR and PTR.



START

lement before a given node in a linked list

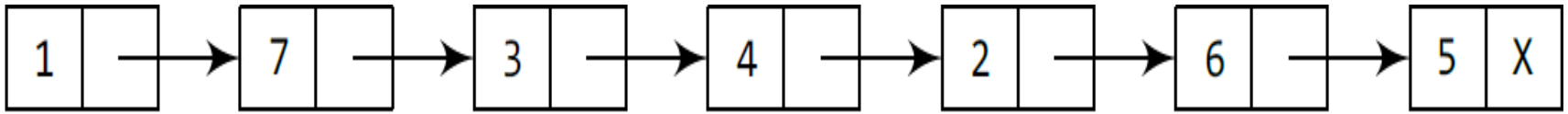
6.2.4 Deleting a Node from a Linked List

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.

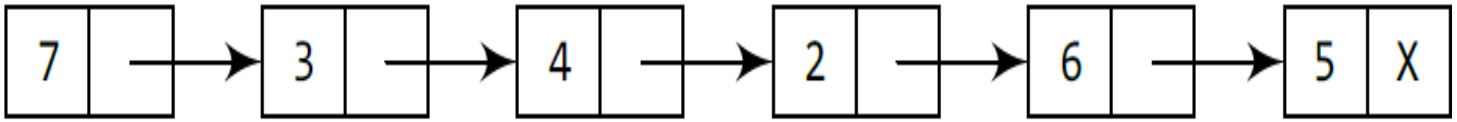
- Underflow condition: delete node from linked list that is empty or START = NULL
- Deleting node from linked list, to free the memory occupied by that node.
- Change AVAIL pointer to points to address that has been recently vacated

- ***Deleting First Node from Linked List***
- 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

Deleting the first node of a linked list

Deleting a Node from a Linked List

The algorithm to delete the first node from a linked list.

- Step 1, check UNDERFLOW Condition
- Initialize PTR with START to point to first node of the list.

Step 3, START made to point to next node in sequence; free memory pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

```
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

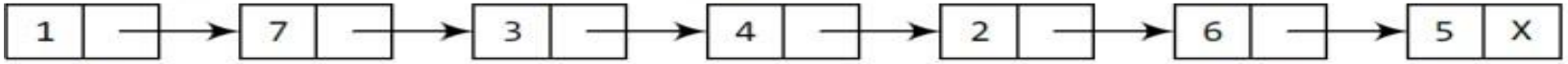
Deleting the Last Node from a Linked List

START

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

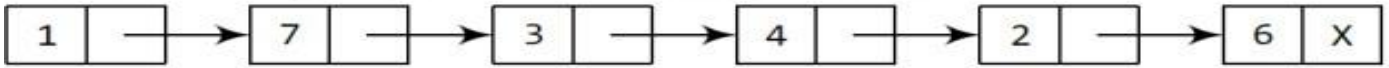
START
PREPTR
PTR

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



START PREPTR PTR

Set the NEXT part of PREPTR node to NULL.



START

Deleting the last node of a linked list

Deleting a Node from a Linked List

- The algorithm to delete the last node from a linked list.
- In Step 2, 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, 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.

```
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
```

Algorithm to delete the last node

Deleting a Node from a Linked List

- ***Deleting the Node After a Given Node in a Linked List***
- To delete the node that succeeds the node which contains data value 4.

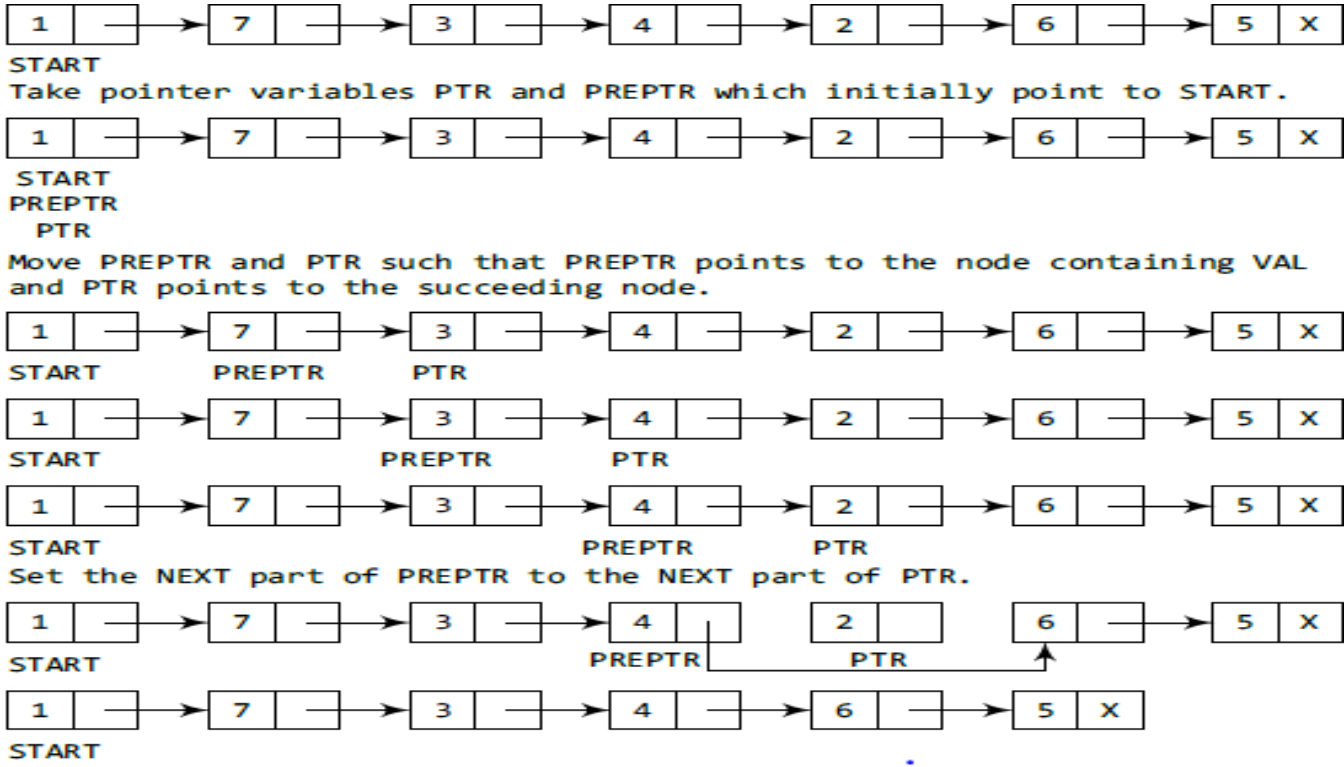


Figure 6.24 Deleting the node after a given node in a linked list

Deleting a Node from a Linked List

- Sep 2: PTR = START.
- In the while loop, another pointer PREPTR points to one node before the PTR.
- Once we reach node containing VAL and node succeeding it, set next pointer of node containing VAL to the address contained in next field of node succeeding it.
- Memory of node succeeding given node is freed and returned back to free pool.

```
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

1. Write a program to create a linked list and perform insertions and deletions of all cases. Write functions to sort and finally delete the entire list at once.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int data;
        struct node *next;
};
struct node *start = NULL;
struct node *create_ll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *insert_before(struct node *);
struct node *insert_after(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_node(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
struct node *sort_list(struct node *);


int main(int argc, char *argv[]) {
        int option;
        do
        {
                printf("\n\n *****MAIN MENU *****");
                printf("\n 1:  Create a list");
                printf("\n 2:  Display the list");
                printf("\n 3:  Add a node at the beginning");
                printf("\n 4:  Add a node at the end");
                printf("\n 5:  Add a node before a given node");
                printf("\n 6:  Add a node after a given node");
                printf("\n 7:  Delete a node from the beginning");
```

```
                printf("\n 8:  Delete a node from the end");
                printf("\n 9:  Delete a given node");
                printf("\n 10: Delete a node after a given node");
                printf("\n 11: Delete the entire list");
                printf("\n 12: Sort the list");
                printf("\n 13: EXIT");
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                case 1: start = create_ll(start);
                        printf("\n LINKED LIST CREATED");
                        break;
                case 2: start = display(start);
                        break;
                case 3: start = insert_beg(start);
                        break;
                case 4: start = insert_end(start);
                        break;
                case 5: start = insert_before(start);
                        break;
                case 6: start = insert_after(start);
                        break;
                case 7: start = delete_beg(start);
                        break;
                case 8: start = delete_end(start);
                        break;
                case 9: start = delete_node(start);
                        break;
                case 10: start = delete_after(start);
                        break;
                case 11: start = delete_list(start);
                        printf("\n LINKED LIST DELETED");
                        break;
                case 12: start = sort_list(start);
                        break;
                }
        }while(option !=13);
        getch();
        return 0;
}
```

```
struct node *create_ll(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter -1 to end");
        printf("\n Enter the data : ");
        scanf("%d", &num);
        while(num!=-1)
        {
                new_node = (struct node*)malloc(sizeof(struct node));
                new_node -> data=num;
                if(start==NULL)
                {
                        new_node -> next = NULL;
                        start = new_node;
                }
                else
                {
                        ptr=start;
                        while(ptr->next!=NULL)
                        ptr=ptr->next;
                        ptr->next = new_node;
                        new_node->next=NULL;
                }
                printf("\n Enter the data : ");
                scanf("%d", &num);
        }
        return start;
}
struct node *display(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\t %d", ptr -> data);
                ptr = ptr -> next;
        }
        return 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;
        new_node -> next = start;
        start = new_node;
        return start;
}
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;
}
struct node *insert_before(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 before 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)
        {
```

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

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

```
                while(ptr -> data != val)
                {
                        preptr = ptr;
                        ptr = ptr -> next;
                }
                preptr -> next = ptr -> next;
                free(ptr);
                return 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;
}
struct node *delete_list(struct node *start)
{
        struct node *ptr; // Lines 252-254 were modified from original code to fi
unresposiveness in output window
        if(start!=NULL){
                ptr=start;
                while(ptr != NULL)
                {
                        printf("\n %d is to be deleted next", ptr -> data);
                        start = delete_beg(ptr);
                        ptr = start;
                }
        }

        return start;
}
struct node *sort_list(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;
```

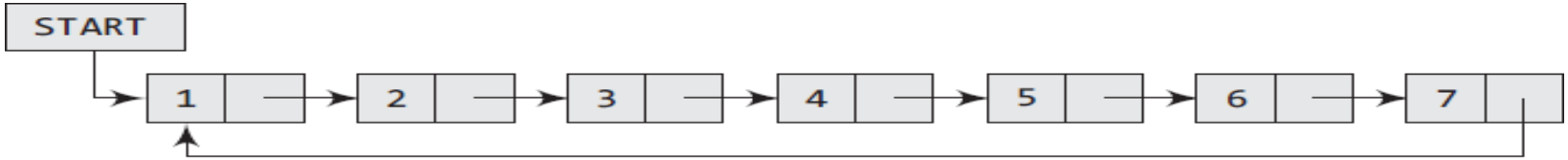
```
                }
        return start;  // Had to be added
}
```

Output

```
*****MAIN MENU *****
1: Create a list
2: Display the list
3: Add a node at the beginning
4: Add the node at the end
5: Add the node before a given node
6: Add the node after a given node
7: Delete a node from the beginning
8: Delete a node from the end
9: Delete a given node
10: Delete a node after a given node
11: Delete the entire list
12: Sort the list
13: Exit
Enter your option : 3
Enter your option : 73
```

CIRCULAR LINKED LISTs

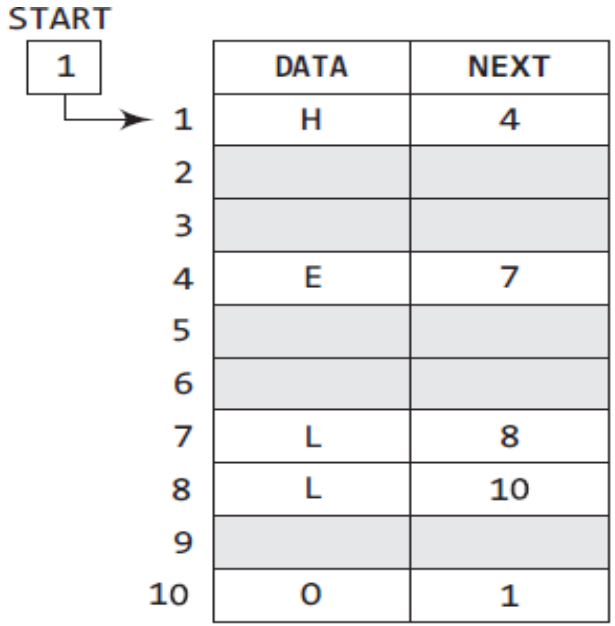
- Last node contains pointer to first node of list.
- It is actually circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.
- Has no beginning and no ending.
- Downside of a circular linked list is the complexity of iteration.
- There are no NULL values in NEXT part of any of the nodes of list.



Circular linked list

CIRCULAR LINKED LISTs

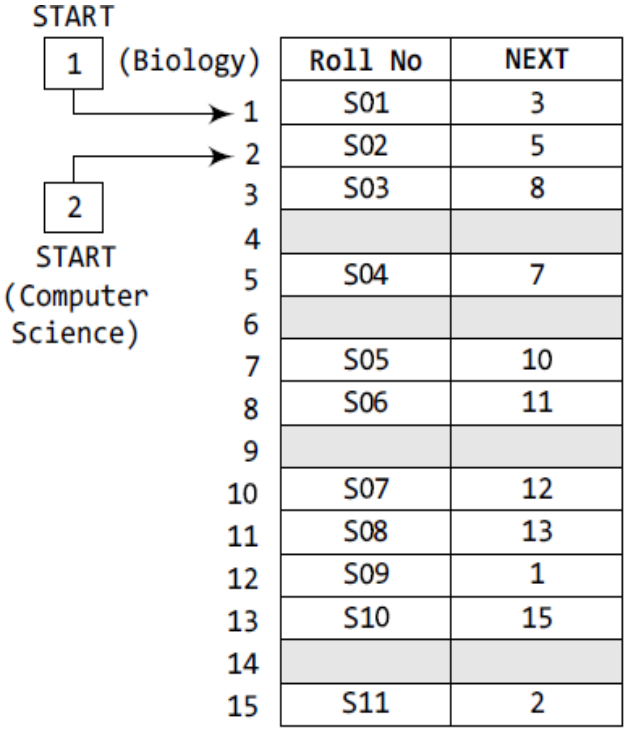
- When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited. How is this done?
- The answer is simple. A circular linked list is used to maintain the sequence of the Web pages visited
- . Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons. Actually, this is done using either the circular stack or the circular queue.
- Traverse the DATA and NEXT in the figure, st characters that when put together form the word HELLO.



Memory representation of a circular linked list

CIRCULAR LINKED LISTs

- 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.
- 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.



	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

Memory representation of two circular linked lists stored in the memory

CIRCULAR LINKED LISTs

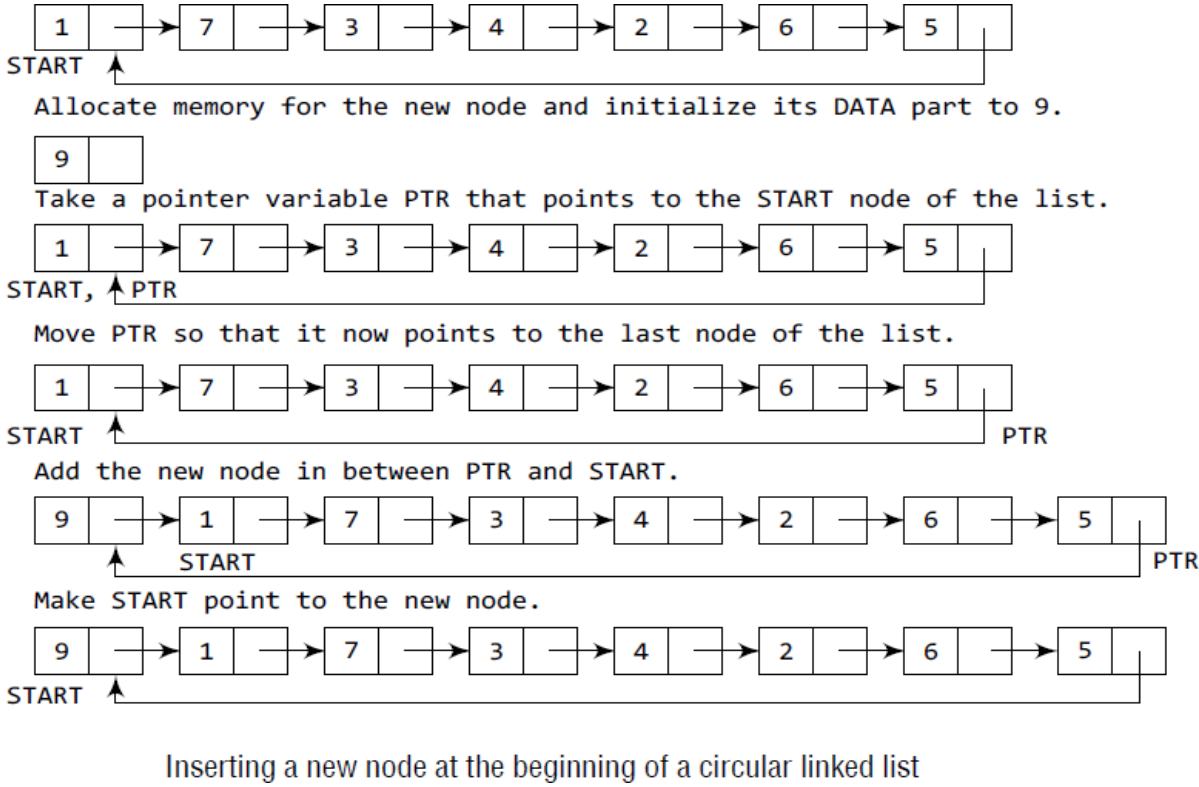
Inserting a New Node in a Circular Linked List

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.

CIRCULAR LINKED LISTs

- ***Inserting a Node at the Beginning of a Circular Linked List***
- 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.



Inserting a new node at the beginning of a circular linked list

CIRCULAR LINKED LISTs

The algorithm to insert a new node at the beginning of a linked list.

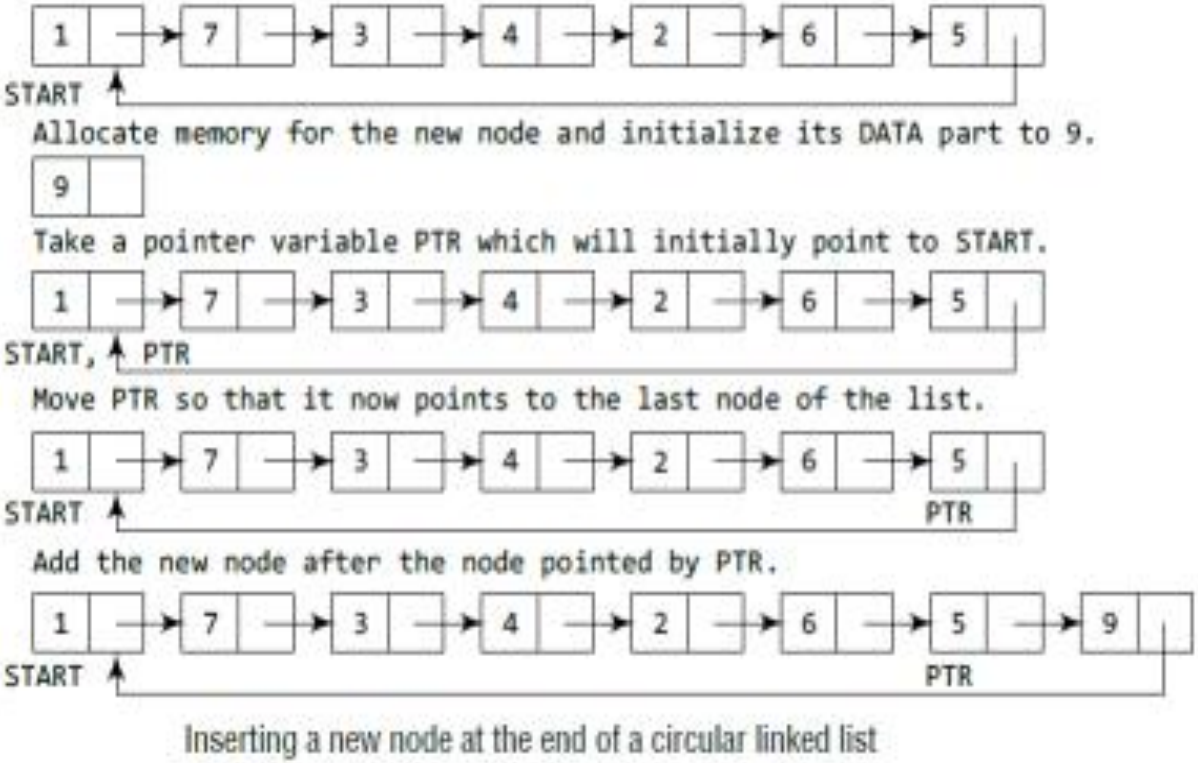
- Step 1: check OVERFLOW message is printed. If free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part to START.
- The new node is added as the first node of the list, hence START will
- be NEW_NODE.
- While inserting node in circular linked list, traverse to last node. last node points to original START, update its NEXT field so that after insertion it points to new node (now known as START).

```
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

CIRCULAR LINKED LISTs

- ***Inserting a Node at the End of a Circular Linked List***
- 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.



Inserting a new node at the end of a circular linked list

CIRCULAR LINKED LISTs

```
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

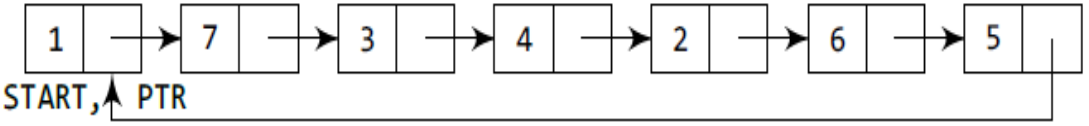
CIRCULAR LINKED LISTs

- **Deleting a Node from a Circular Linked List**
- Take two cases and then see how deletion is done in each case.
- Rest of the cases of deletion are same as that given for singly linked lists.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

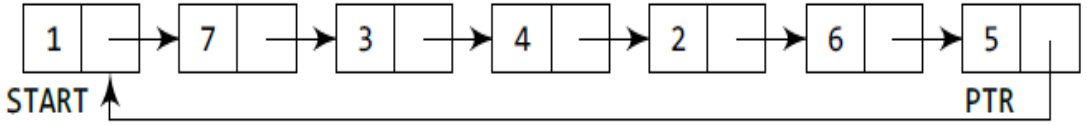
CIRCULAR LINKED LISTs

- ***Deleting the First Node from a Circular Linked List***
- To delete a node from the beginning of the list, then the following changes will be done in the linked list.

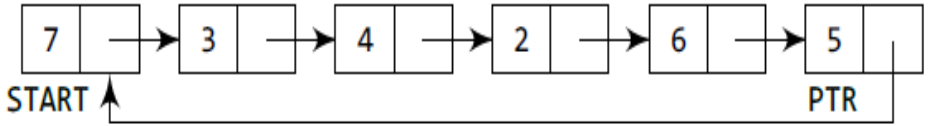
Take a variable PTR and make it point to the START node of the list.



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.



eleting the first node from a circular linked list

CIRCULAR LINKED LISTs

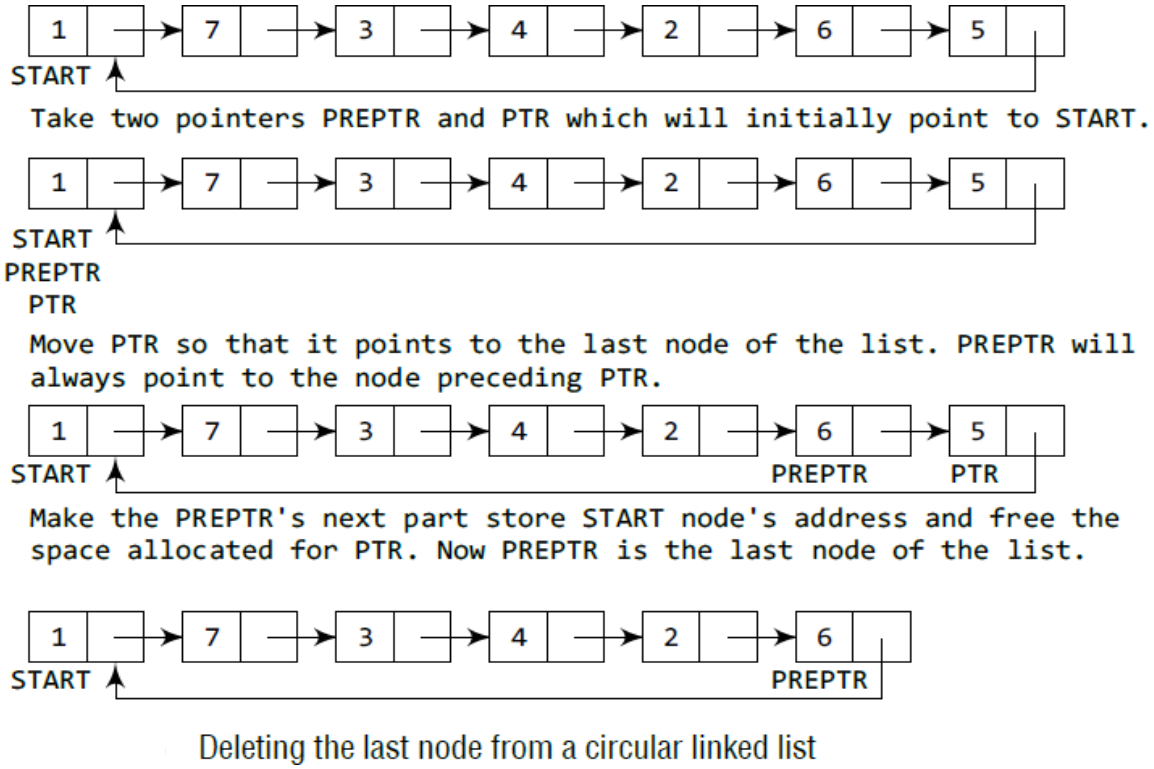
- The algorithm to delete the first node from a circular linked list.
- Step 1: check UNDERFLOW
- Use PTR to traverse list to reach last node.
- Step 5: change next pointer of last node to point to second node of circular linked list.
- Step 6: Free memory occupied by first node.
- Step 7: START = second 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
```

Algorithm to delete the first node

CIRCULAR LINKED LISTs

- ***Deleting the Last Node from a Circular Linked List***
- To delete the last node from the linked list, then the following changes will be done in the linked list.



Deleting the last node from a circular linked list

CIRCULAR LINKED LISTs

- The algorithm to delete the last node from a circular linked list.
- Step 2: PTR = START.
- While loop, another pointer variable PREPTR always points to one node before PTR.
- Once we reach the last node and the second last node, we set the next pointer o 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.

```
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

CIRCULAR LINKED LISTs

2. Write a program to create a circular linked list. Perform insertion and deletion at the beginning and end of the list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
    int data;
    struct node *next;
};
struct node *start = NULL;
struct node *create_cll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
```

```
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
int main()
{
    int option;
    clrscr();
    do
    {
         printf("\n\n *****MAIN MENU *****");
         printf("\n 1: Create a list");
         printf("\n 2: Display the list");
         printf("\n 3: Add a node at the beginning");
         printf("\n 4: Add a node at the end");
         printf("\n 5: Delete a node from the beginning");
         printf("\n 6: Delete a node from the end");
         printf("\n 7: Delete a node after a given node");
         printf("\n 8: Delete the entire list");
         printf("\n 9: EXIT");
         printf("\n\n Enter your option : ");
         scanf("%d", &option);
         switch(option)
         {
```

```
                    case 1: start = create_cll(start);
                            printf("\n CIRCULAR LINKED LIST CREATED");
                            break;
                    case 2: start = display(start);
                            break;
                    case 3: start = insert_beg(start);
                            break;
                    case 4: start = insert_end(start);
                            break;
                    case 5: start = delete_beg(start);
                            break;
                    case 6: start = delete_end(start);
                            break;
                    case 7: start = delete_after(start);
                            break;
                    case 8: start = delete_list(start);
                            printf("\n CIRCULAR LINKED LIST DELETED");
                            break;
            }
        }while(option !=9);
        getch();
        return 0;
    }
```

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

```
struct node *display(struct node *start)
{
        struct node *ptr;
        ptr=start;
        while(ptr -> next != start)
        {
                printf("\t %d", ptr -> data);
                ptr = ptr -> next;
        }
        printf("\t %d", ptr -> data);
        return start;
}
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;
}
```

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

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

Output

```
*****MAIN MENU *****
1: Create a list
2: Display the list
3: Add a node at the beginning
-------------------------
8: Delete the entire list
9: EXIT
Enter your option : 1
Enter -1 to end
Enter the data: 1
Enter the data: 2
Enter the data: 4
Enter the data: -1
CIRCULAR LINKED LIST CREATED
Enter your option : 3
Enter your option : 5
Enter your option : 2
5    1    2    4
Enter your option : 9
```

DOUBLY LINKED LISTS

- A doubly linked list or a twoway linked list whicah contains pointer to next as well as previous node in sequence.
- It consists of three parts—data, a pointer to the next node, and a pointer to the
- previous node.

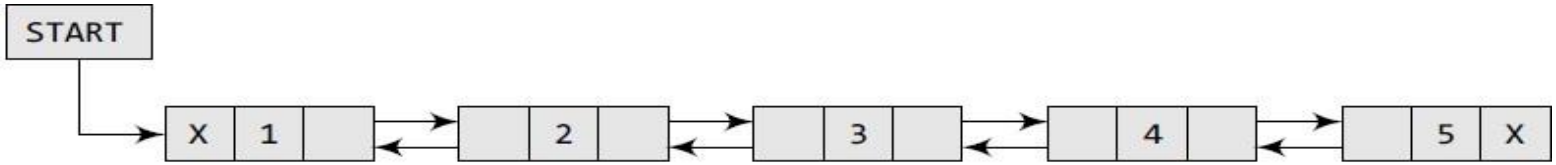


Figure 6. st

In C, the structure of a doubly linked list can be given as,

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

DOUBLY LINKED LISTS

- PREV of first node and NEXT of last node : NULL
- The PREV field is used to store the address of the prec node, which enables us to traverse the list in the backv direction.
- Requires more space per node and more expensive ba operations.
- Provides ease to manipulate elements of list as it maint pointers to nodes in both the directions (forward and b
- Makes searching twice as efficient.
- Example, START = 1, so the first data is stored at addr which is H. Since this is the first node, it has no previou hence stores NULL or–1 in the PREV field.
- We will traverse the list until we reach a position where entry contains–1 or NULL. This denotes the end of the
- traverse the DATA and NEXT , stores characters that .. together form the word HELLO.

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

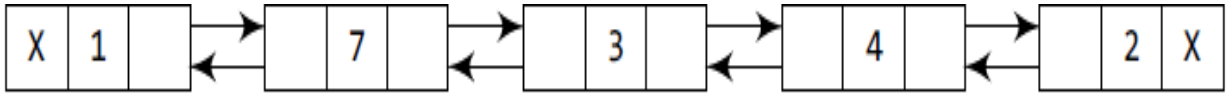
Memory representation of a doubly linked list

DOUBLY LINKED LISTS

- **Inserting a New Node in a Doubly Linked List**
- 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.

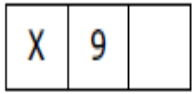
DOUBLY LINKED LISTS

- ***Inserting a Node at the Beginning of a Doubly Linked List***
- 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.

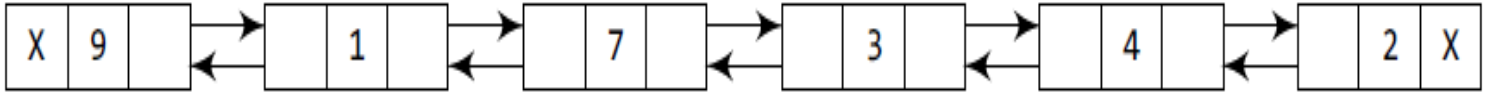


START

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

Inserting a new node at the beginning of a doubly linked list

DOUBLY LINKED LISTS

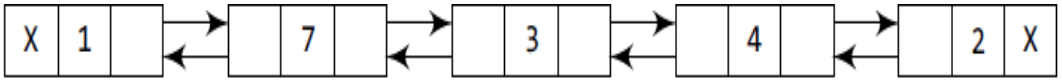
- The algorithm to insert a new node at the beginning of a doubly linked list.
- In Step , check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node. 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.
- 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.

```
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_NOD
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Algorithm to insert a new node
the beginning

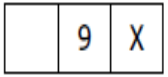
DOUBLY LINKED LISTS

- ***Inserting a Node at the End of a Doubly Linked List***
- 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.

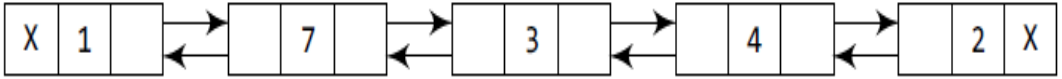


START

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

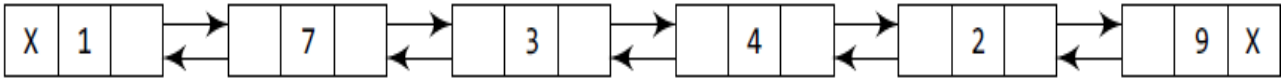


Take a pointer variable PTR and make it point to the first node of the list.



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

Inserting a new node at the end of a doubly linked list

DOUBLY LINKED LISTS

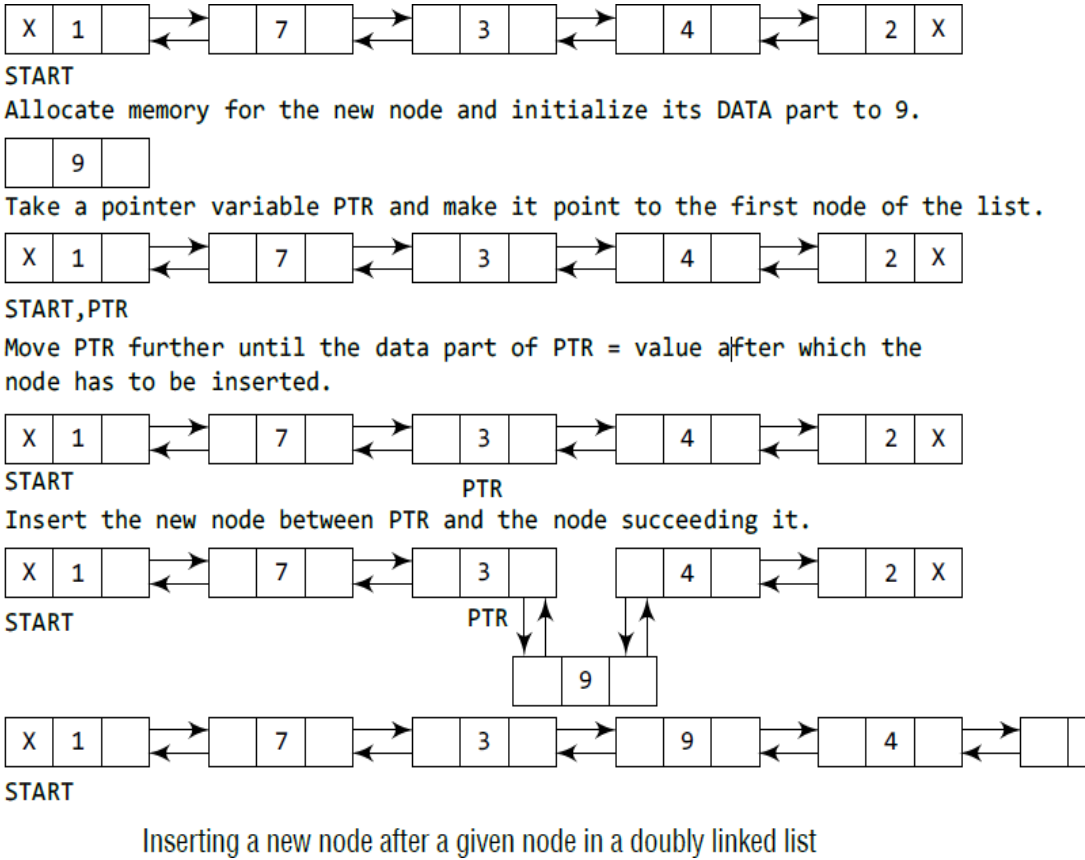
- The algorithm to insert a new node at the end of a doubly linked list. In Step 6, we take a pointer variable PTR and initialize it with START. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node.
- in Step 9, we change the NEXT pointer of the last 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).

```
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
```

Figure 6.42 Algorithm to insert a new node at the end

DOUBLY LINKED LISTS

- ***Inserting a Node After a Given Node in a Doubly Linked List***
- To add a new node with value 9 after the node containing 3.



Inserting a new node after a given node in a doubly linked list

DOUBLY LINKED LISTS

- The algorithm to insert a new node after a given node in a doubly linked list. 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.

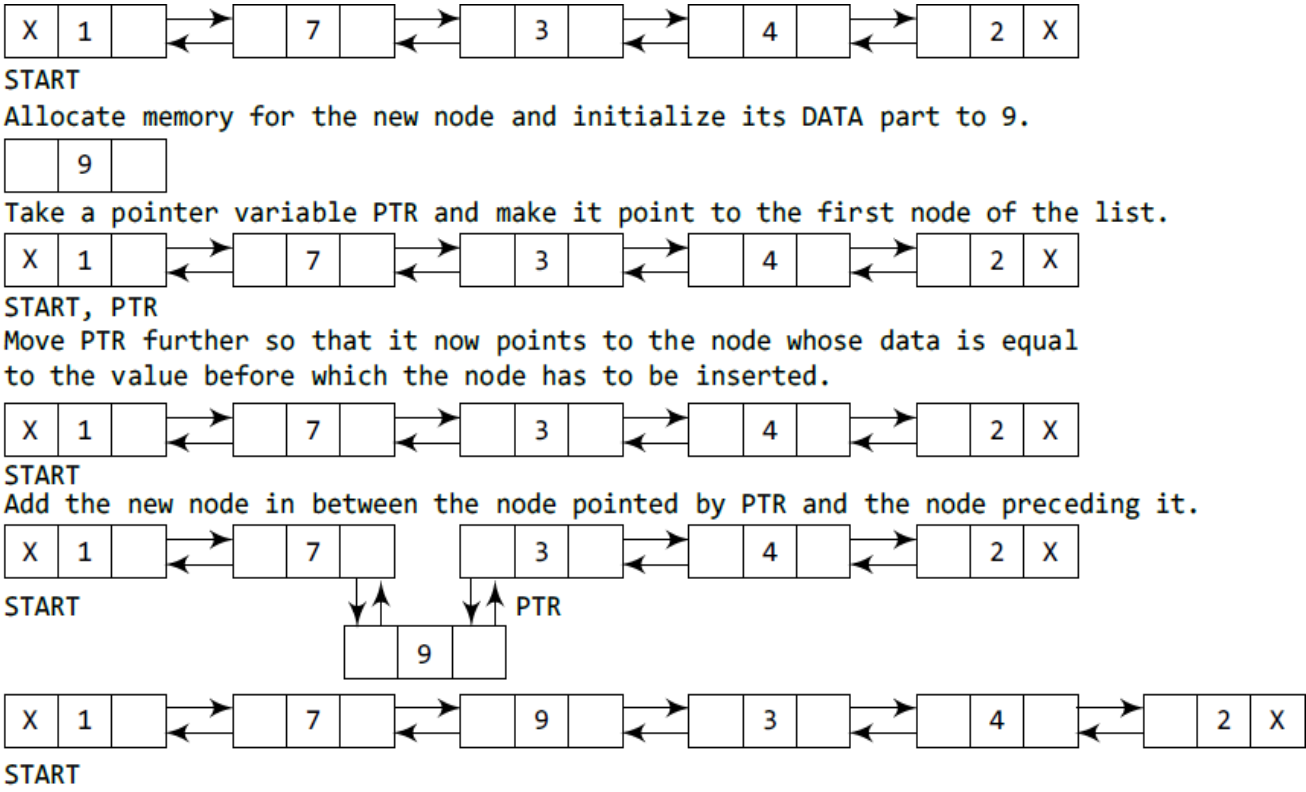
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT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

DOUBLY LINKED LISTS

Inserting a Node before a given node in a Doubly linked list

To add a new node with value 9 before the node containing 3.



Inserting a new node before a given node in a doubly linked list

DOUBLY LINKED LISTS

- Step 1: check for OVERFLOW condition
- Step 5, PTR: START
- while loop, traverse through the linked list to reach the node with value = NUM.
- Change the NEXT and PREV fields in such a way that the new node is inserted before the desired 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
```

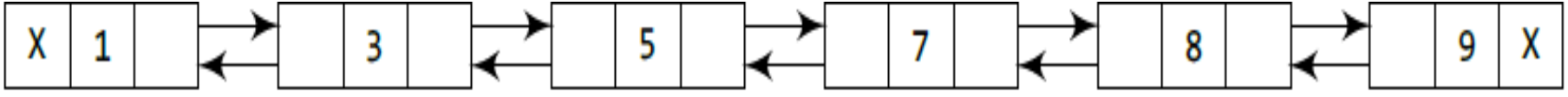
Algorithm to insert a new node before a given node

DOUBLY LINKED LISTS

- **Deleting a Node from a Doubly Linked List**
- 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.
- Case 4: The node before a given node is deleted.

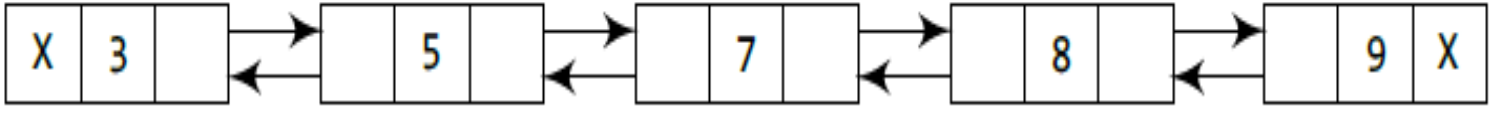
DOUBLY LINKED LISTS

- ***Deleting the First Node from a Doubly Linked List***
- 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

Deleting the first node from a doubly linked list

DOUBLY LINKED LISTS

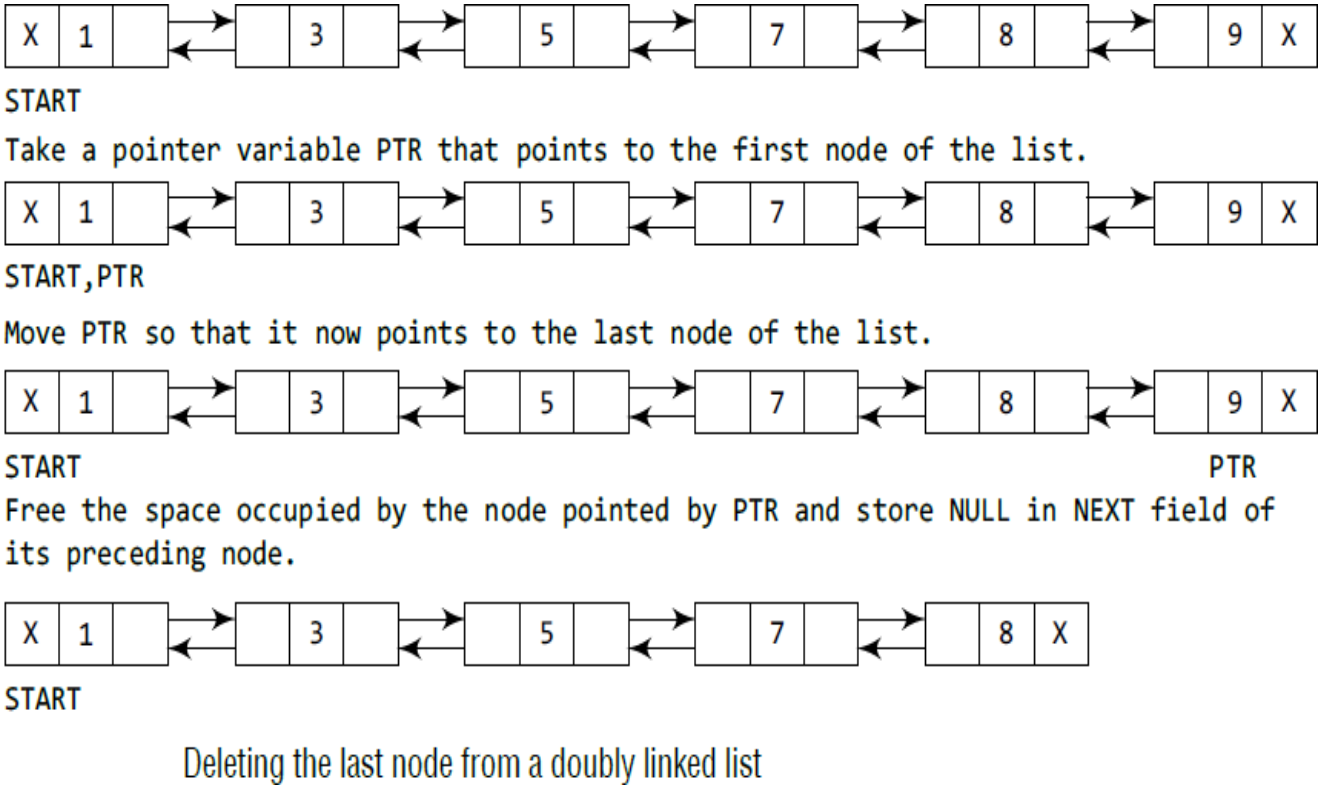
- Step 1: Check for UNDERFLOW
- Set PTR: START
- Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

```
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

DOUBLY LINKED LISTS

- ***Deleting the Last Node from a Doubly Linked List***
- To delete the last node from the linked list, then the following changes will be done in the linked list.



Deleting the last node from a doubly linked list

DOUBLY LINKED LISTS

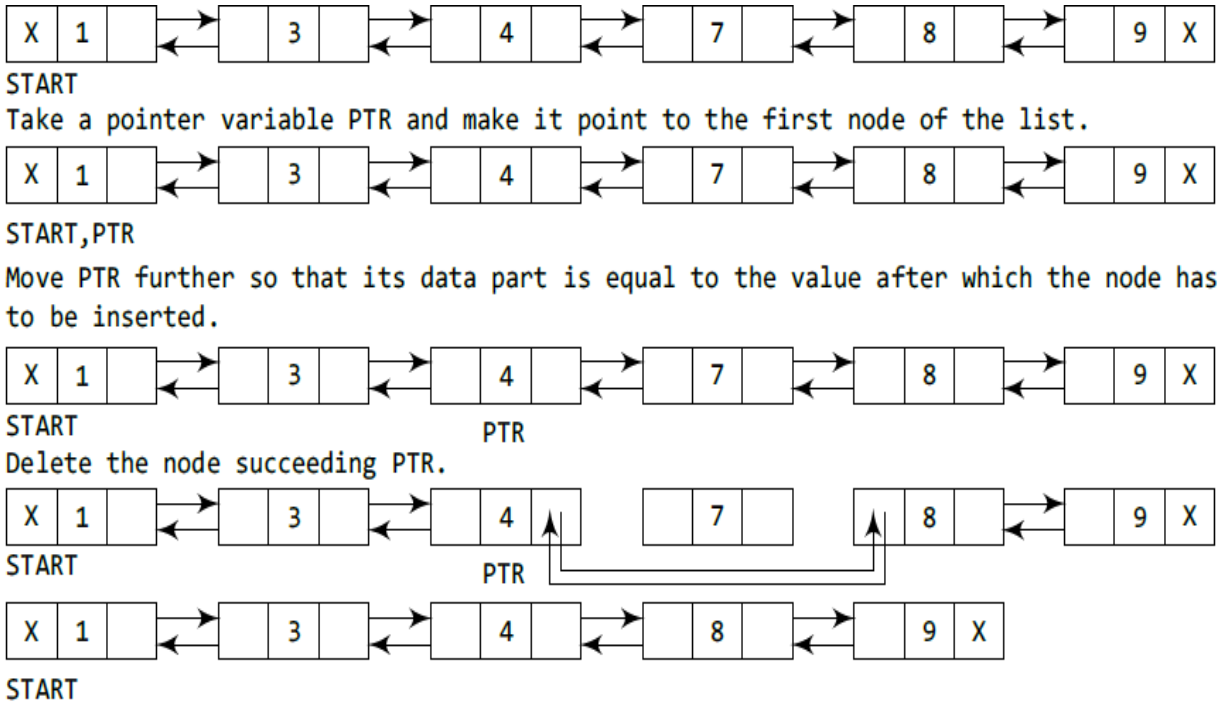
- Step 2: PTR: START
- Traverse the list to reach last node
- To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

```
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
```

Figure 6.50 Algorithm to delete the last node

DOUBLY LINKED LISTS

- ***Deleting the Node After a Given Node in a Doubly Linked List***
- To delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



Deleting the node after a given node in a doubly linked list

DOUBLY LINKED LISTS

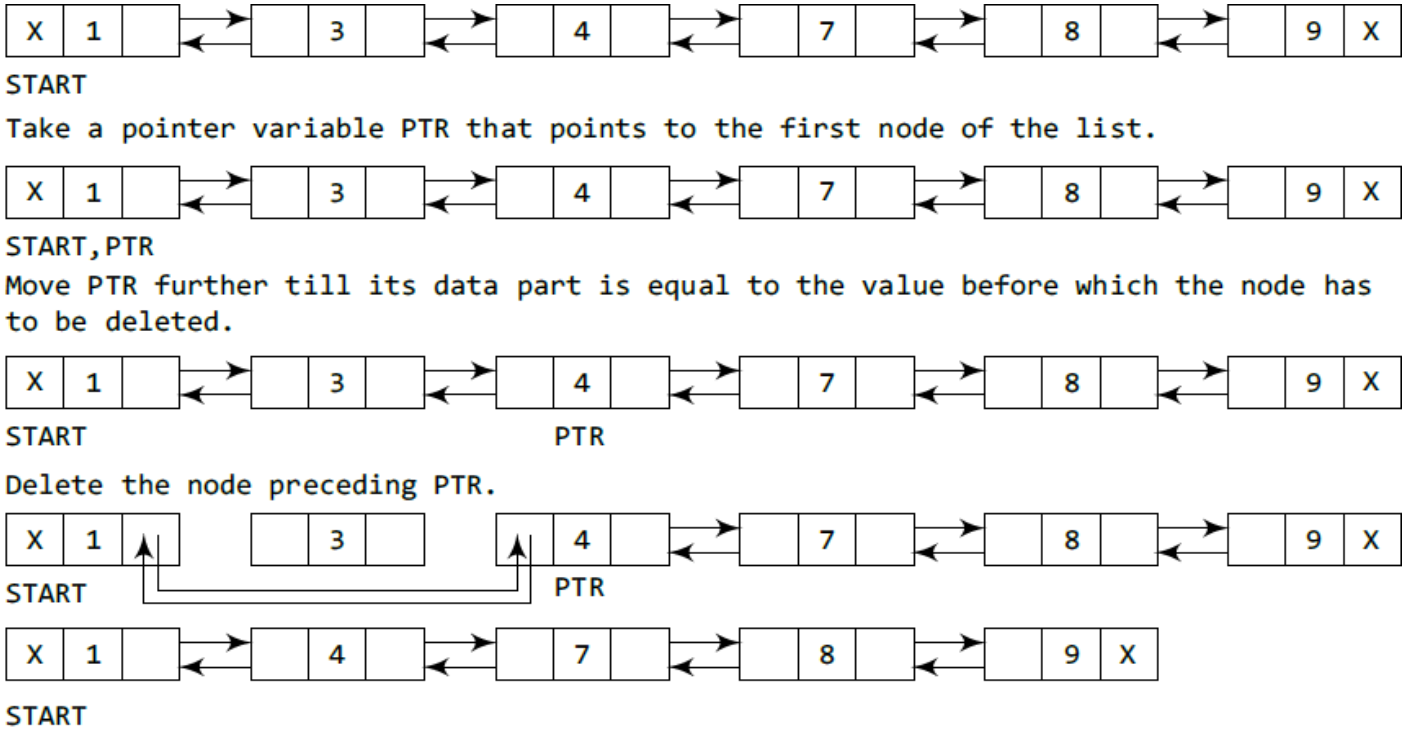
- Step 2: Set PTR: START
- Traverses through the linked list to reach the given node.
- Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

```
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 != N
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

DOUBLY LINKED LISTS

- ***Deleting the Node Before a Given Node in Doubly Linked List***
- To delete the node preceding the node with value 4..



Deleting a node before a given node in a doubly linked list

DOUBLY LINKED LISTS

- Step 2: PTR = START
- Traverses through linked list to reach desired node. Once we reach the node containing VAL, the PREV field of PTR set to contain the address of the node preceding the node which comes before PTR.
- Free memory of the node preceding PTR
- We can insert or delete a node in a constant number of operations given only thatnode's address
- A singly linked list which requires the previous node'saddress also to perform the same operation.

```
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
```

Algorithm to delete a node before a given node

PROGRAMMING EXAMPLE

3. Write a program to create a doubly linked list and perform insertions and deletions in all cases.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
```

```
struct node
{
        struct node *next;
        int data;
        struct node *prev;
};
struct node *start = NULL;
struct node *create_ll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *insert_before(struct node *);
struct node *insert_after(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_before(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
int main()
{
        int option;
        clrscr();
        do
        {
```

```
        printf("\n\n *****MAIN MENU *****");
        printf("\n 1: Create a list");
        printf("\n 2: Display the list");
        printf("\n 3: Add a node at the beginning");
        printf("\n 4: Add a node at the end");
        printf("\n 5: Add a node before a given node");
        printf("\n 6: Add a node after a given node");
        printf("\n 7: Delete a node from the beginning");
        printf("\n 8: Delete a node from the end");
        printf("\n 9: Delete a node before a given node");
        printf("\n 10: Delete a node after a given node");
        printf("\n 11: Delete the entire list");
        printf("\n 12: EXIT");
        printf("\n\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
                case 1:  start = create_ll(start);
                         printf("\n DOUBLY LINKED LIST CREATED");
                         break;
                case 2:  start = display(start);
                         break;
                case 3:  start = insert_beg(start);
                         break;
                case 4:  start = insert_end(start);
                         break;
                case 5:  start = insert_before(start);
                         break;
                case 6:  start = insert_after(start);
                         break;
                case 7:  start = delete_beg(start);
                         break;
                case 8:  start = delete_end(start);
                         break;
                case 9:  start = delete_before(start);
                         break;
                case 10: start = delete_after(start);
```

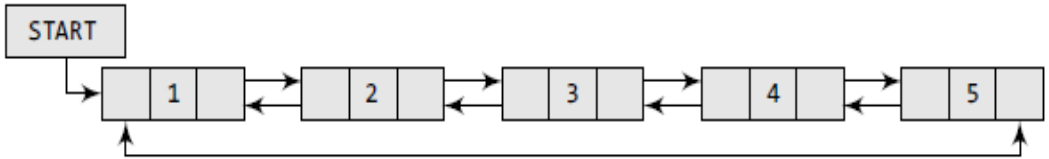
Output

```
*****MAIN MENU *****
1: Create a list
2: Display the list
-------------------------
11: Delete the entire list
12: EXIT
Enter your option : 1
Enter -1 to end
Enter the data: 1
Enter the data: 3
Enter the data: 4
Enter the data: -1
DOUBLY LINKED LIST CREATED
Enter your option : 12
```

CIRCULAR DOUBLY LINKED LISTs

Contains a pointer to the next as well as the previous node in the sequence.

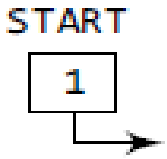
- The difference between a doubly linked and a circular doubly linked list is same as that exists between a singly linked list and a circular linked list.
- The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node.
- The next field 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



Circular doubly linked list

CIRCULAR DOUBLY LINKED LISTs

- Advantage: makes search operation twice as efficient.
- START stores address of 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, take 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.
- 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, tha the node that contains the address of the first node is actually the last node of the list.



	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

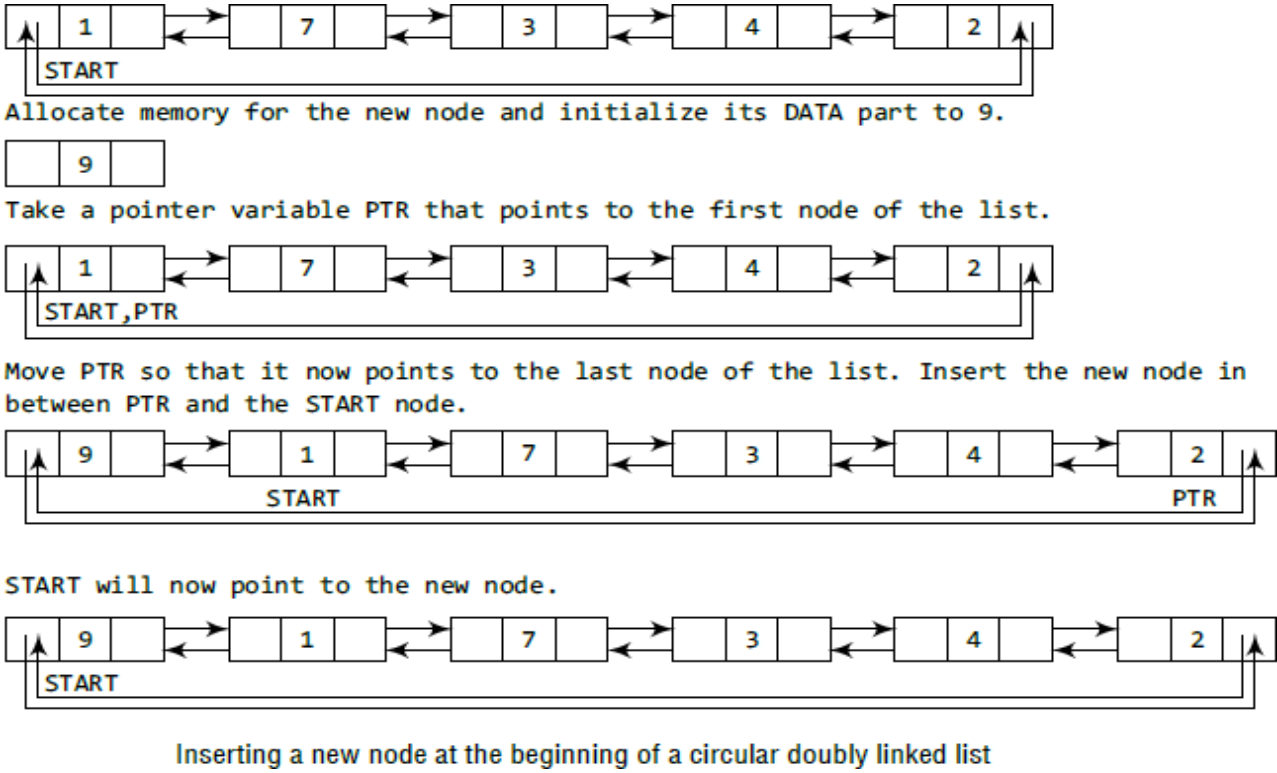
Memory representation of a circular doubly linked list

CIRCULAR DOUBLY LINKED LISTs

- **Inserting a New Node in a Circular Doubly Linked List**
- A new node is added into an already existing circular doubly linked list.
- Take two cases and rest of the cases are similar to that given for doubly linked lists.
- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.

CIRCULAR DOUBLY LINKED LISTs

- ***Inserting a Node at the Beginning of a Circular Doubly Linked List***
- 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.



Inserting a new node at the beginning of a circular doubly linked list

CIRCULAR DOUBLY LINKED LISTs

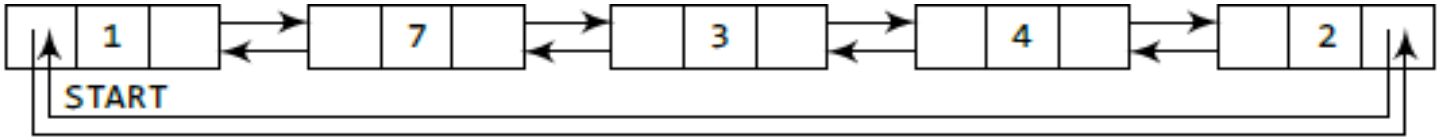
- Step 1: Check OVERFLOW
- Set its data part with the given VAL and its next part is initialized with the address of the first node of the list, which is stored in START.
- The new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.
- It is a circular doubly linked list, the PREV field of the NEW_NODE is set to contain the address of the last node.

```
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
```

Algorithm to insert a new node at the beginning

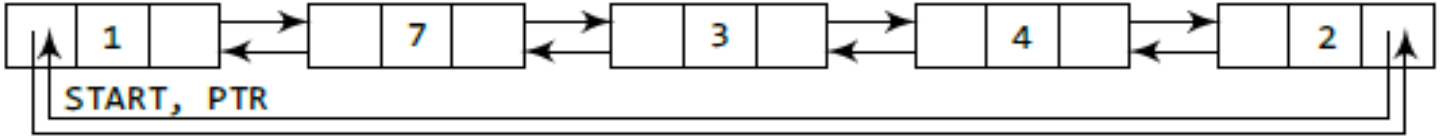
CIRCULAR DOUBLY LINKED LISTs

- ***Inserting a Node at the End of a Circular Doubly Linked List***
- 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.

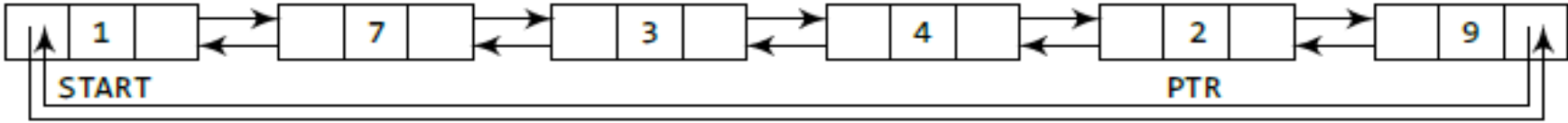


Allocate memory for the new node and initialize its DATA part to 9.

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



Move PTR to point to the last node of the list so that the new node can be inserted after it.



CIRCULAR DOUBLY LINKED LISTs

- Step 6: PTR= START
- traverse through the linked list to reach the last node. Once we reach the last node
- Step 9: 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).

```
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
```

) Algorithm to insert a new node at the end

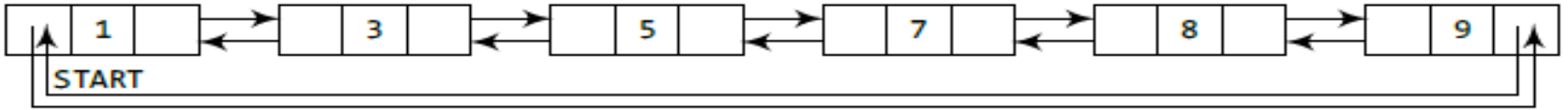
CIRCULAR DOUBLY LINKED LISTs

- **Deleting a Node from a Circular Doubly Linked List**
- Take two cases and rest of the cases are same as that given for doubly linked lists.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

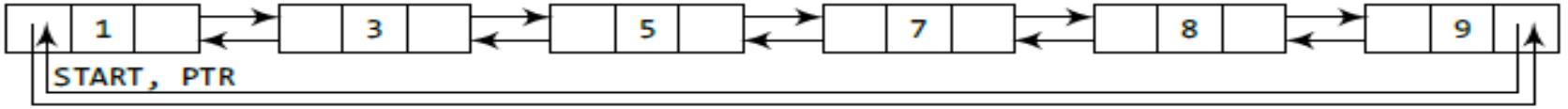
CIRCULAR DOUBLY LINKED LISTs

Deleting the First Node from a Circular Doubly Linked List

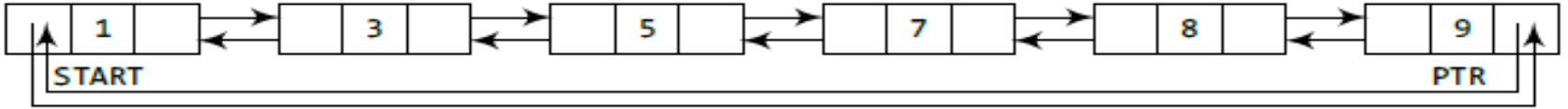
To delete a node from the beginning of the list, then the following changes will be done in the linked list.



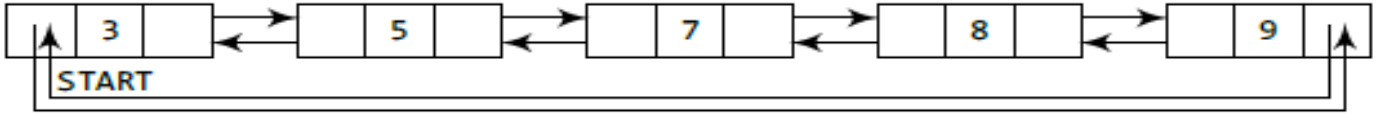
Take a pointer variable PTR that points to the first node of the list.



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



Make START point to the second node of the list. Free the space occupied by the first node.



Deleting the first node from a circular doubly linked list

CIRCULAR DOUBLY LINKED LISTs

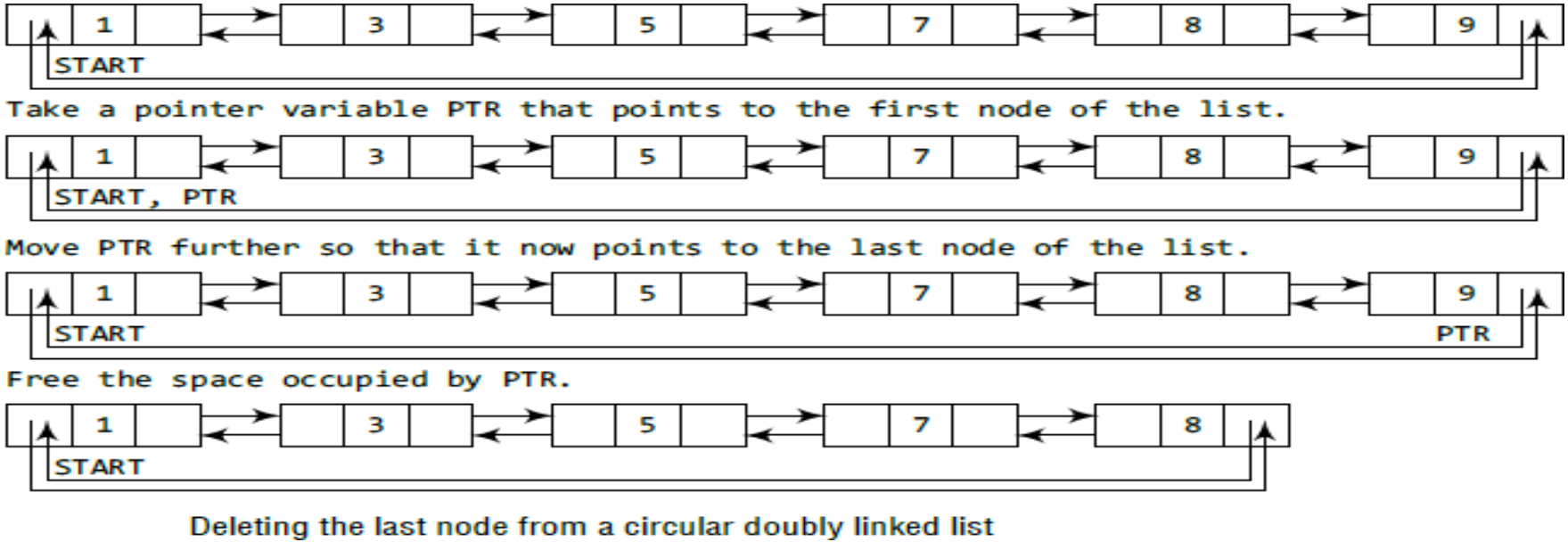
- Step1: check UNDERFLOW
- PTR=START
- Traverses through the list to reach the last node.
- NEXT pointer of PTR is set to contain the address of the node that succeeds START. Finally, START is made to point to the next node in the sequence and the memory occupied by the first node of the list is freed and returnedto the free pool.

```
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
```

Algorithm to delete the first node

CIRCULAR DOUBLY LINKED LISTs

- ***Deleting the Last Node from a Circular Doubly Linked List***
- To delete the last node from the linked list, then the following changes will be done in the linked list.



Deleting the last node from a circular doubly linked list

CIRCULAR DOUBLY LINKED LISTs

- Step 2: PTR = START
- Loop traverses through the list to reach 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, 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.

```
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
```

Algorithm to delete the last node

4. Write a program to create a circular doubly linked list and perform insertions and deletions at the beginning and end of the list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        struct node *next;
        int data;
        struct node *prev;
};
struct node *start = NULL;
struct node *create_ll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_node(struct node *);
struct node *delete_list(struct node *);
int main()
{
        int option;
        clrscr();
        do
        {
                printf("\n\n *****MAIN MENU *****");
                printf("\n 1: Create a list");
                printf("\n 2: Display the list");
                printf("\n 3: Add a node at the beginning");
                printf("\n 4: Add a node at the end");
                printf("\n 5: Delete a node from the beginning");
                printf("\n 6: Delete a node from the end");
                printf("\n 7: Delete a given node");
                printf("\n 8: Delete the entire list");
                printf("\n 9: EXIT");
                printf("\n\n Enter your option : ");
```

```
                {
                        case 1: start = create_ll(start);
                                printf("\n CIRCULAR DOUBLY LINKED LIST CREATED");
                                break;
                        case 2: start = display(start);
                                break;
                        case 3: start = insert_beg(start);
                                break;
                        case 4: start = insert_end(start);
                                break;
                        case 5: start = delete_beg(start);
                                break;
                        case 6: start = delete_end(start);
                                break;
                        case 7: start = delete_node(start);
                                break;
                        case 8: start = delete_list(start);
                                printf("\n CIRCULAR DOUBLY LINKED LIST DELETED");
                                break;
                }
        }while(option != 9);
        getch();
        return 0;
}
struct node *create_ll(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter -1 to end");
        printf("\n Enter the data : ");
        scanf("%d", &num);
        while(num != -1)
        {
                if(start == NULL)
                {
                        new_node = (struct node*)malloc(sizeof(struct node));
                        new_node->prev = NULL;
                        new_node->data = num;
                        new_node->next = start;
                        start = new_node;
                }
```

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

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

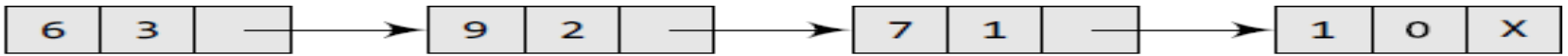
```
struct node *delete_node(struct node *start)
{
        struct node *ptr;
        int val;
        printf("\n Enter the value of the node which has to be deleted : ");
        scanf("%d", &val);
        ptr = start;
        if(ptr->data == val)
        {
                start = delete_beg(start);
                return start;
        }
        else
        {
                while(ptr->data != val)
                        ptr = ptr->next;
                ptr->prev->next = ptr->next;
                ptr->next->prev = ptr->prev;
                free(ptr);
                return start;
        }
}
struct node *delete_list(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr->next != start)
                start = delete_end(start);
        free(start);
        return start;
}
```

Output

```
*****MAIN MENU *****
1: Create a list
2: Display the list
------------------------
8: Delete the entire list
9: EXIT
Enter your option : 1
Enter -1 to end
Enter the data: 2
Enter the data: 3
Enter the data: 4
Enter the data: -1
CIRCULAR DOUBLY LINKED LIST CREATED
Enter your option : 8
CIRCULAR DOUBLY LINKED LIST DELETED
Enter your option : 9
```

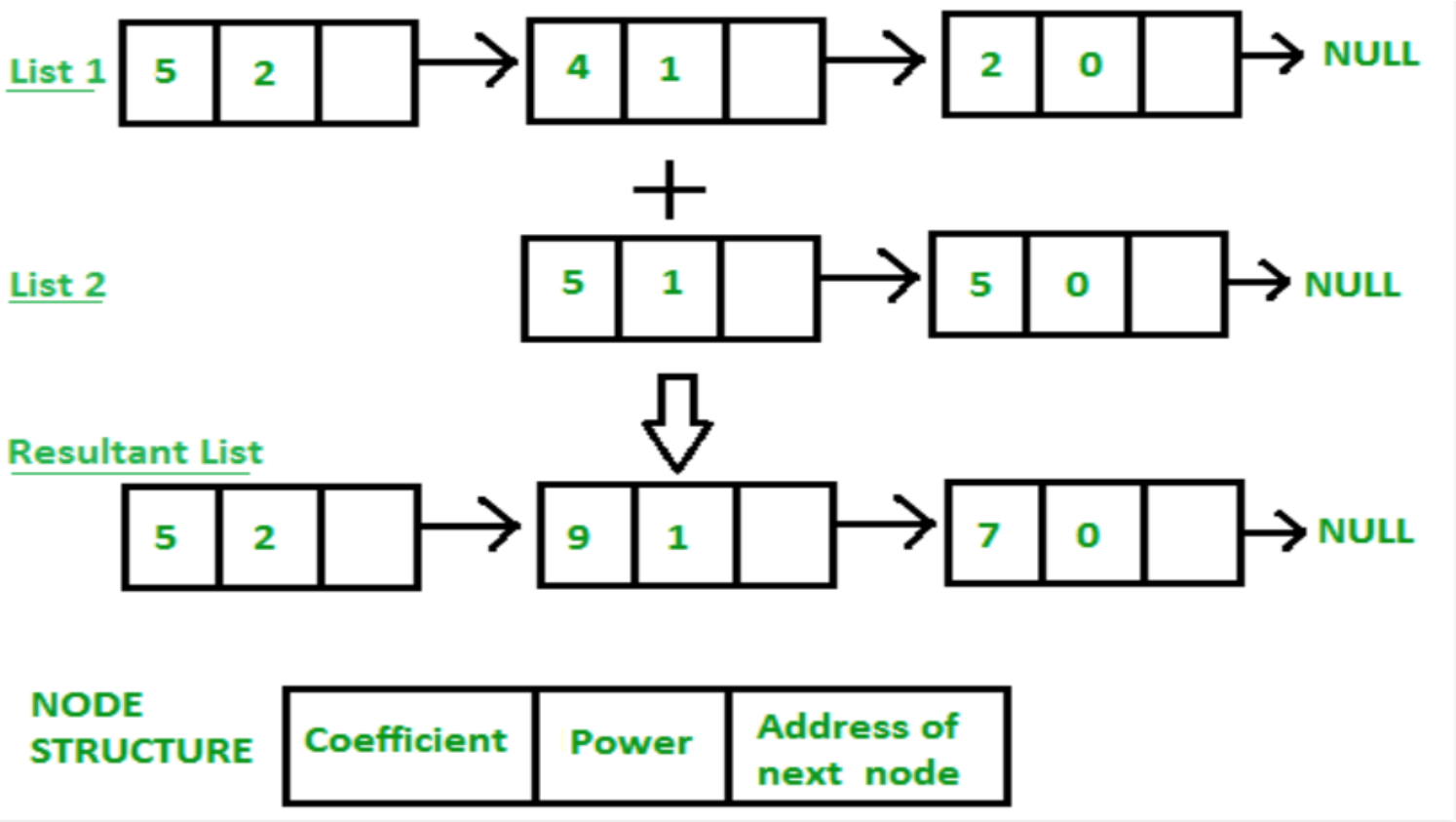
APPLICATIONS OF LINKED LISTS

- **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

- **Polynomial Addition**



6. Write a program to store a polynomial using linked list. Also, perform addition and subtraction on two polynomials.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int num;
        int coeff;
        struct node *next;
};
struct node *start1 = NULL;
struct node *start2 = NULL;
struct node *start3 = NULL;
struct node *start4 = NULL;
struct node *last3 = NULL;
struct node *create_poly(struct node *);
struct node *display_poly(struct node *);
struct node *add_poly(struct node *, struct node *, struct node *);
struct node *sub_poly(struct node *, struct node *, struct node *);
struct node *add_node(struct node *, int, int);
int main()
{
        int option;
        clrscr();
        do
        {
```

```
        printf("\n******* MAIN MENU *******");
        printf("\n 1. Enter the first polynomial");
        printf("\n 2. Display the first polynomial");
        printf("\n 3. Enter the second polynomial");
        printf("\n 4. Display the second polynomial");
        printf("\n 5. Add the polynomials");
        printf("\n 6. Display the result");
        printf("\n 7. Subtract the polynomials");
        printf("\n 8. Display the result");
        printf("\n 9. EXIT");
        printf("\n\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
                case 1: start1 = create_poly(start1);
                        break;
                case 2: start1 = display_poly(start1);
                        break;
                case 3: start2 = create_poly(start2);
                        break;
                case 4: start2 = display_poly(start2);
                        break;
                case 5: start3 = add_poly(start1, start2, start3);
                        break;
                case 6: start3 = display_poly(start3);
                        break;
                case 7: start4 = sub_poly(start1, start2, start4);
                        break;
                case 8: start4 = display_poly(start4);
```

```
                                        break;
                        }
                }while(option!=9);
                getch();
                return 0;
}
struct node *create_poly(struct node *start)
{
        struct node *new_node, *ptr;
        int n, c;
        printf("\n Enter the number : ");
        scanf("%d", &n);
        printf("\t Enter its coefficient : ");
        scanf("%d", &c);
        while(n != -1)
        {
                if(start==NULL)
                {
                        new_node = (struct node *)malloc(sizeof(struct node));
                        new_node->num = n;
                        new_node->coeff = c;
                        new_node->next = NULL;
                        start = new_node;
                }
                else
                {
                        ptr = start;
                        while(ptr->next != NULL)
                                ptr = ptr->next;
                        new_node = (struct node *)malloc(sizeof(struct node));
                        new_node->num = n;
                        new_node->coeff = c;
                        new_node->next = NULL;
                        ptr->next = new_node;
                }
```

```
                printf("\n Enter the number : ");
                scanf("%d", &n);
                if(n == -1)
                        break;
                printf("\t Enter its coefficient : ");
                scanf("%d", &c);
        }
        return start;
}
struct node *display_poly(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\n%d x %d\t", ptr->num, ptr->coeff);
                ptr = ptr->next;
        }
        return start;
}
struct node *add_poly(struct node *start1, struct node *start2, struct node *start3)
{
        struct node *ptr1, *ptr2;
        int sum_num, c;
```

```
ptr1 = start1, ptr2 = start2;
while(ptr1 != NULL && ptr2 != NULL)
{
        if(ptr1->coeff == ptr2->coeff)
        {
                sum_num = ptr1->num + ptr2->num;
                start3 = add_node(start3, sum_num, ptr1->coeff);
                ptr1 = ptr1->next;
                ptr2 = ptr2->next;
        }
        else if(ptr1->coeff > ptr2->coeff)
        {
                start3 = add_node(start3, ptr1->num, ptr1->coeff);
                ptr1 = ptr1->next;
        }
        else if(ptr1->coeff < ptr2->coeff)
        {
                start3 = add_node(start3, ptr2->num, ptr2->coeff);
                ptr2 = ptr2->next;
        }
}
if(ptr1 == NULL)
{
        while(ptr2 != NULL)
        {
                start3 = add_node(start3, ptr2->num, ptr2->coeff);
                ptr2 = ptr2->next;
        }
}
if(ptr2 == NULL)
{
        while(ptr1 != NULL)
        {
                start3 = add_node(start3, ptr1->num, ptr1->coeff);
                ptr1 = ptr1->next;
```

```
                }
        }
        return start3;
}
struct node *sub_poly(struct node *start1, struct node *start2, struct node *start4)
{
        struct node *ptr1, *ptr2;
        int sub_num, c;
        ptr1 = start1, ptr2 = start2;
        do
        {
                if(ptr1->coeff == ptr2->coeff)
                {
                        sub_num = ptr1->num - ptr2->num;
                        start4 = add_node(start4, sub_num, ptr1->coeff);
                        ptr1 = ptr1->next;
                        ptr2 = ptr2->next;
                }
                else if(ptr1->coeff > ptr2->coeff)
                {
                        start4 = add_node(start4, ptr1->num, ptr1->coeff);
                        ptr1 = ptr1->next;
                }
                else if(ptr1->coeff < ptr2->coeff)
```

```
                {
                        start4 = add_node(start4, ptr2->num, ptr2->coeff);
                        ptr2 = ptr2->next;
                }
        }while(ptr1 != NULL || ptr2 != NULL);
        if(ptr1 == NULL)
        {
                while(ptr2 != NULL)
                {
                        start4 = add_node(start4, ptr2->num, ptr2->coeff);
                        ptr2 = ptr2->next;
                }
        }
        if(ptr2 == NULL)
        {
                while(ptr1 != NULL)
                {
                        start4 = add_node(start4, ptr1->num, ptr1->coeff);
                        ptr1 = ptr1->next;
                }
        }
        return start4;
}
struct node *add_node(struct node *start, int n, int c)
{
        struct node *ptr, *new_node;
        if(start == NULL)
        {
                new_node = (struct node *)malloc(sizeof(struct node));
                new_node->num = n;
                new_node->coeff = c;
                new_node->next = NULL;
                start = new_node;
        }
        else
        {
```

```
                ptr = start;
                while(ptr -> next != NULL)
                        ptr = ptr -> next;
                new_node = (struct node *)malloc(sizeof(struct node));
                new_node -> num = n;
                new_node -> coeff = c;
                new_node -> next = NULL;
                ptr -> next = new_node;
        }
        return start;
}
```

Output

```
******* MAIN MENU *******
1. Enter the first polynomial
2. Display the first polynomial
------------------------------
9. EXIT
Enter your option : 1
Enter the number : 6      Enter its coefficient : 2
Enter the number : 5      Enter its coefficient : 1
Enter the number : -1
Enter your option : 2
6 x 2    5 x 1
Enter your option : 9
```

- Polynomial Multiplication

```
#include <stdio.h>
#include <conio.h>

#define MAX 10

struct term
{
     int coeff ;
     int exp ;
} ;

struct poly
{
     struct term t [10] ;
     int noofterms ;
} ;

void initpoly ( struct poly *) ;
void polyappend ( struct poly *, int, int ) ;
struct poly polyadd ( struct poly, struct poly ) ;
struct poly polymul ( struct poly, struct poly ) ;
void display ( struct poly ) ;
```

```
void main( )
{
    struct poly p1, p2, p3 ;

    clrscr( ) ;

    initpoly ( &p1 ) ;
    initpoly ( &p2 ) ;
    initpoly ( &p3 ) ;

    polyappend ( &p1, 1, 4 ) ;
    polyappend ( &p1, 2, 3 ) ;
    polyappend ( &p1, 2, 2 ) ;
    polyappend ( &p1, 2, 1 ) ;

    polyappend ( &p2, 2, 3 ) ;
    polyappend ( &p2, 3, 2 ) ;
    polyappend ( &p2, 4, 1 ) ;

    p3 = polymul ( p1, p2 ) ;

    printf ( "\nFirst polynomial:\n" ) ;
    display ( p1 ) ;

    printf ( "\n\nSecond polynomial:\n" ) ;
    display ( p2 ) ;

    printf ( "\n\nResultant polynomial:\n" ) ;
    display ( p3 ) ;

    getch( ) ;
}
```

```c
/* initializes elements of struct poly */
void initpoly ( struct poly *p )
{
     int i ;
     p -> noofterms = 0 ;
     for ( i = 0 ; i < MAX ; i++ )
     {
          p -> t[i].coeff = 0 ;
          p -> t[i].exp = 0 ;
     }
}

/* adds the term of polynomial to the array t */
void polyappend ( struct poly *p, int c, int e )
{
     p -> t[p -> noofterms].coeff = c ;
     p -> t[p -> noofterms].exp =  e ;
     ( p -> noofterms ) ++ ;
}

/* displays the polynomial equation */
void display ( struct poly p )
{
     int flag = 0, i ;
     for ( i = 0 ; i < p.noofterms ; i++ )
     {
          if ( p.t[i].exp != 0 )
               printf ( "%d x^%d + ", p.t[i].coeff, p.t[i].exp ) ;
          else
          {
               printf ( "%d", p.t[i].coeff ) ;
               flag = 1 ;
          }
     }
     if ( !flag )
          printf ( "\b\b  " ) ;

}
```

```
/* adds two polynomials p1 and p2 */
struct poly polyadd ( struct poly p1, struct poly p2 )
{
    int i, j, c ;
    struct poly p3 ;
    initpoly ( &p3 ) ;

    if ( p1.noofterms > p2.noofterms )
        c = p1.noofterms ;
    else
        c = p2.noofterms ;

    for ( i = 0, j = 0 ; i <= c ; p3.noofterms++ )
    {
        if ( p1.t[i].coeff == 0 && p2.t[j].coeff == 0 )
            break ;
        if ( p1.t[i].exp >= p2.t[j].exp )
        {
            if ( p1.t[i].exp == p2.t[j].exp )
            {
                p3.t[p3.noofterms].coeff = p1.t[i].coeff + p2.t[j].coeff ;
                p3.t[p3.noofterms].exp = p1.t[i].exp ;
                i++ ;
                j++ ;
            }
            else
            {
                p3.t[p3.noofterms].coeff = p1.t[i].coeff ;
                p3.t[p3.noofterms].exp = p1.t[i].exp ;
                i++ ;
            }
        }
        else
        {
            p3.t[p3.noofterms].coeff = p2.t[j].coeff ;
            p3.t[p3.noofterms].exp = p2.t[j].exp ;
            j++ ;
        }
    }
    return p3 ;
}
```

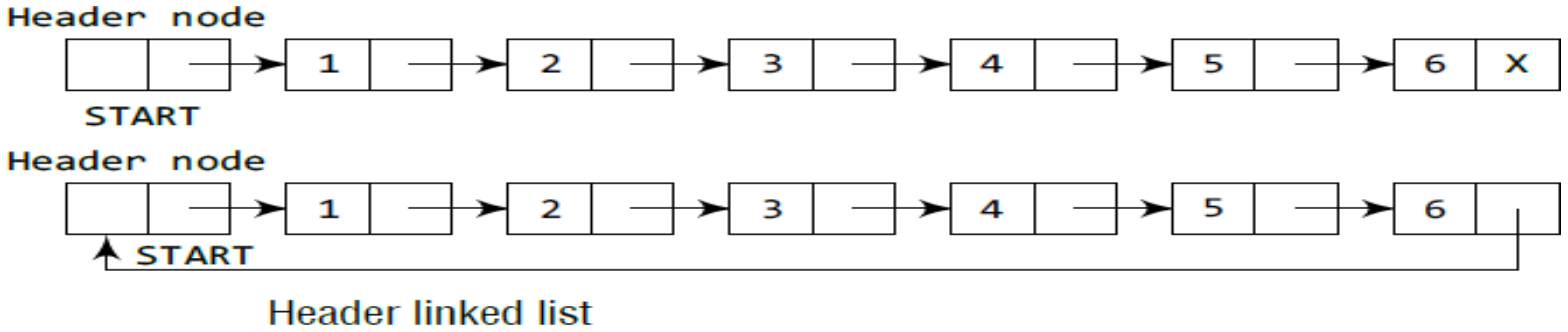
```
/* multiplies two polynomials p1 and p2 */
struct poly polymul ( struct poly p1, struct poly p2 )
{
     int coeff, exp ;
     struct poly temp, p3 ;

     initpoly ( &temp ) ;
     initpoly ( &p3 ) ;

     if ( p1.noofterms != 0 && p2.noofterms != 0 )
     {
          int i ;
          for ( i = 0 ; i < p1.noofterms ; i++ )
          {
               int j ;

               struct poly p ;
               initpoly ( &p ) ;

               for ( j = 0 ; j < p2.noofterms ; j++ )
               {
                    coeff = p1.t[i].coeff * p2.t[j].coeff ;
                    exp = p1.t[i].exp + p2.t[j].exp ;
                    polyappend ( &p, coeff, exp ) ;
               }

               if ( i != 0 )
               {
                    p3 = polyadd ( temp, p ) ;
                    temp = p3  ;
               }
               else
                    temp = p ;
          }
     }
     return p3 ;
}
```

Points to Remember

- A linked list is a linear collection of data elements called as nodes in which linear representation is given by links from one node to another.
- Linked list is a data structure which can be used to implement other data structures such as stacks, queues, and their variations.
- Before we insert a new node in linked lists, we need to check for OVERFLOW condition, which occurs when no free memory cell is present in the system.
- Before we delete a node from a linked list, we must first check for UNDERFLOW condition which occurs when we try to delete a node from a linked list that is empty.
- When we delete a node from a linked list, we have to actually free the memory occupied by that node. The memory is returned back to the free pool so that it can be used to store other programs and data.
- In a circular linked list, the last node contains a pointer to the first node of the 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 had started.
- A doubly linked list or a two-way linked list is a 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 contain NULL. This enables to traverse the list in the backward direction as well.
- Thus, a 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 search operation twice as efficient.
- 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 previous node in the sequence. The difference between a doubly linked and a circular doubly linked list is that 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. Similarly, the previous field of the first field stores the address of the last node.
- A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, in a header linked list START will not point to the first node of the list but START will contain the address of the header node.
- Multi-linked lists are generally used to organize multiple orders of one set of elements. In a multi-linked list, each node can have n number of pointers to other nodes.

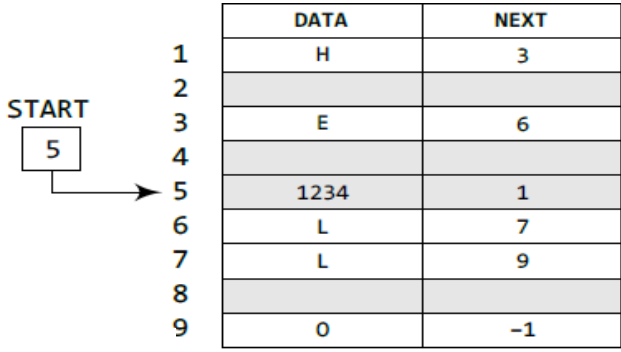
Header Linked Lists

- A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, in a header linked list, START will not point to the first node of the list but START will contain the address of the header node. The following are the two variants of a header linked list:
- Grounded header linked list which stores NULL in the next field of the last node.
- Circular header linked list which stores the address of the header node in the next field of the last node. Here, the header node will denote the end of the list.

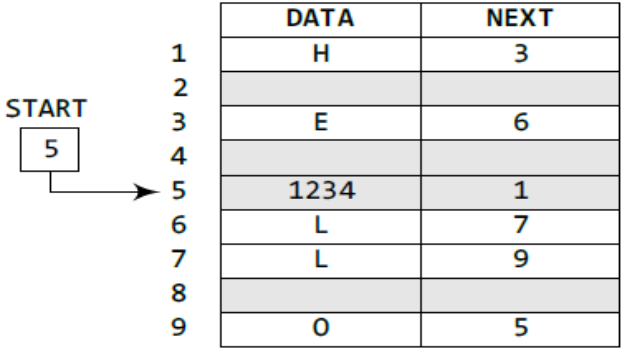


Header Linked Lists

- In a grounded header linked list, a node has two fields, DATA and NEXT. The DATA field will store the information part and the NEXT field will store the address of the node in sequence.
- START stores the address of the header node. The shaded row denotes a header node.
- The NEXT field of the header node stores the address of the first node of the list. This node stores H. The corresponding NEXT field stores the address of the next node, which is 3. So, we will look at address 3 to fetch the next data item.
- The first node can be accessed by writing FIRST_NODE = START> NEXT and not by writing START = FIRST_ NODE. This is because START points to the header node and the header node points to the first node of the header linked list.
- Example : The last node in this case stores the address of the header node (instead of–1).
- The first node can be accessed by writing FIRST_NODE = START> NEXT and not writing START = FIRST_NODE. This is because START points to the header node and the header node points to the first node of the header linked list.



Memory representation of a header linked list



Memory representation of a circular header linked list

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

Algorithm to traverse a circular header linked list

```
Step 1: SET PTR = START->NEXT
Step 2: Repeat Steps 3 and 4 while
        PTR -> DATA != VAL
Step 3:     SET PREPTR = PTR
Step 4:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 5: SET PREPTR -> NEXT = PTR -> NEXT
Step 6: FREE PTR
Step 7: EXIT
```

Algorithm to delete a node from a circular header 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 PTR = START -> NEXT
Step 5: SET NEW_NODE -> DATA = VAL
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
       [END OF LOOP]
Step 8: NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Algorithm to insert a new node in a circular header linked list

5. Write a program to implement a header linked list.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int data;
        struct node *next;
};
struct node *start = NULL;
struct node *create_hll(struct node *);
struct node *display(struct node *);
int main()
{
        int option;
        clrscr();
        do
        {
                printf("\n\n *****MAIN MENU *****");
                printf("\n 1: Create a list");
                printf("\n 2: Display the list");
                printf("\n 3: EXIT");
                printf("\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1: start = create_hll(start);
                                printf("\n HEADER LINKED LIST CREATED");
                                break;
```

```
                        case 2: start = display(start);
                                break;
                }
        }while(option !=3);
        getch();
        return 0;
}
struct node *create_hll(struct node *start)
{
        struct node *new_node, *ptr;
        int num;
        printf("\n Enter -1 to end");
        printf("\n Enter the data : ");
        scanf("%d", &num);
        while(num!=-1)
        {
                new_node = (struct node*)malloc(sizeof(struct node));
                new_node->data=num;
                new_node->next=NULL;
                if(start==NULL)
                {
                        start = (struct node*)malloc(sizeof(struct node));
                        start->next=new_node;
                }
                else
                {
                        ptr=start;
                        while(ptr->next!=NULL)
                                ptr=ptr->next;
                        ptr->next=new_node;
                }
                printf("\n Enter the data : ");
                scanf("%d", &num);
        }
        return start;
}
```

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

Output

```
*****MAIN MENU *****
1: Create a list
2: Display the list
3: EXIT
Enter your option : 1
Enter -1 to end
Enter the data: 1
Enter the data: 2
Enter the data: 4
Enter the data: -1
HEADER LINKED LIST CREATED
Enter your option : 3
```

MultiLinked Lists

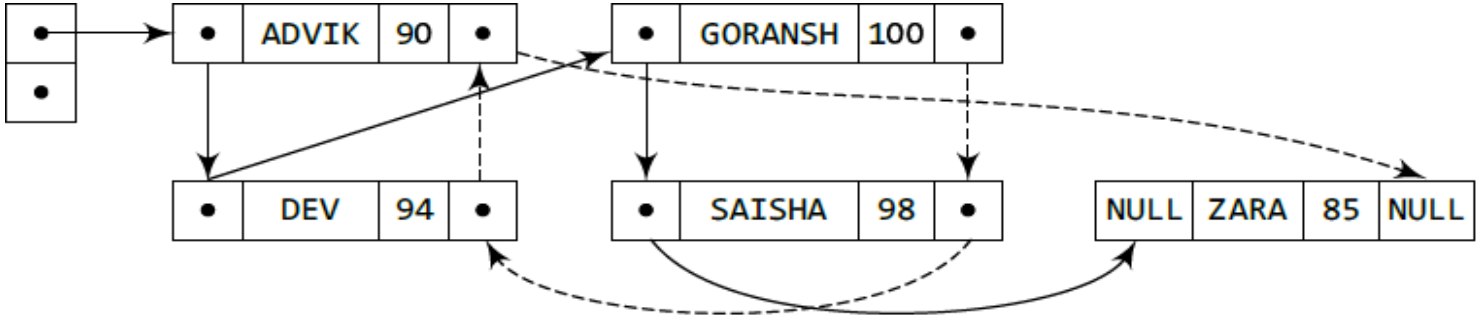
- In a multilinked list, each node can have n number of pointers to other nodes. A doubly linked list is a special case of multilinked lists.
- Nodes in a multilinked list may or may not have inverses for each pointer.
- Differentiate a doubly linked list from a multilinked list in two ways:
- (a) A doubly linked list has exactly two pointers. One pointer points to the previous node and the other points to the next node. But a node in the multilinked list can have any number of pointers.
- (b) In a doubly linked list, pointers are exact inverses of each other, i.e., for every pointer which points to a previous node there is a pointer which points to the next node. This is not true for a multilinked list.

MultiLinked Lists

- Multilinked lists are generally used to organize multiple orders of one set of elements. For example, if we have a linked list that stores name and marks obtained by students in a class, then we can organize the nodes of the list in two ways:

(i) Organize the nodes alphabetically (according to the name)

(ii) Organize the nodes according to decreasing order of marks so that the information of student who got highest marks comes before other students.



Multi-linked list that stores names alphabetically as well as according to decreasing order of marks

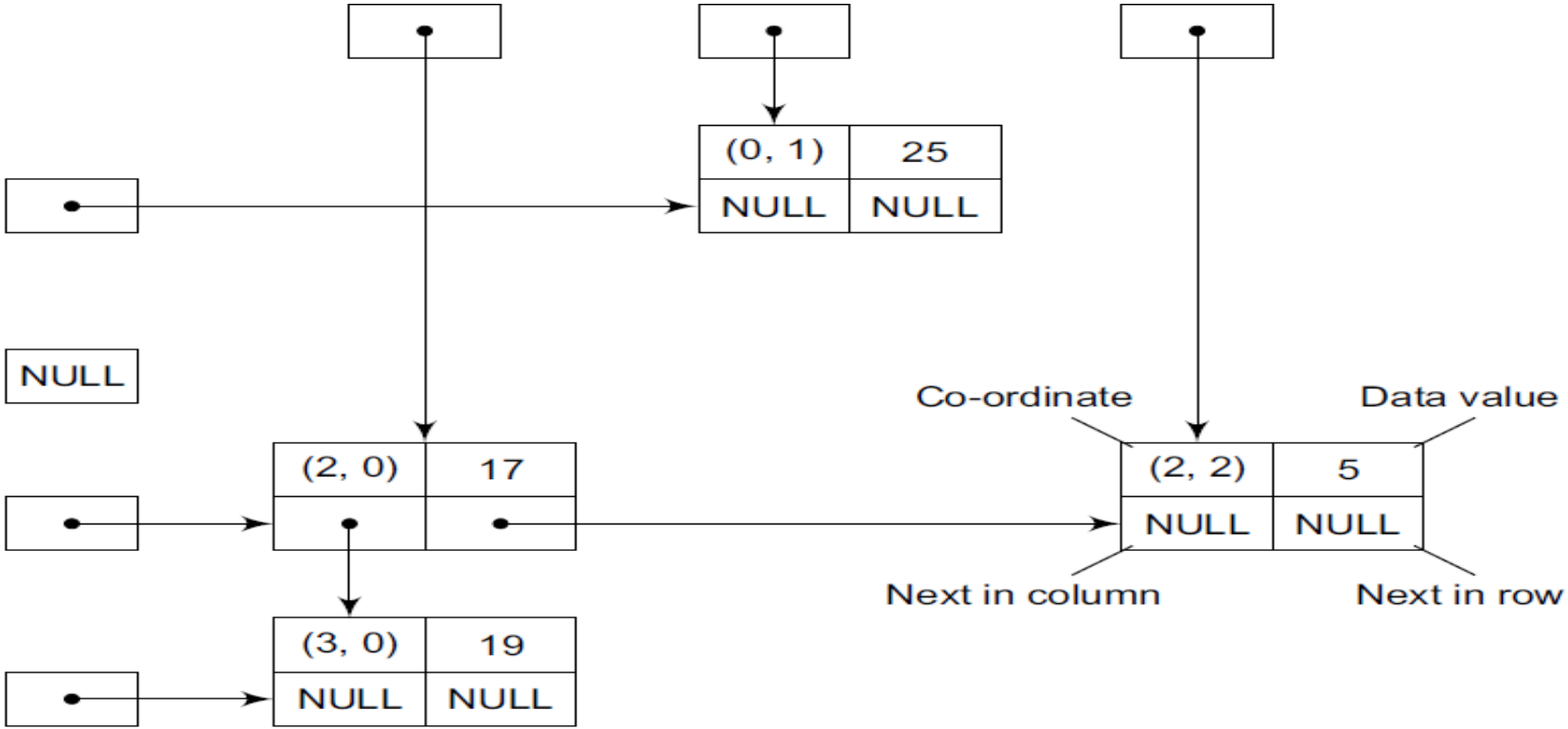
MultiLinked Lists

- The sparse matrix can be represented using a linked list for every row and column.
- A value is in exactly one row and one column, it will appear in both lists exactly once.
- A node in the multilinked will have four parts. First stores the data, second stores a pointer to the next node in the row, third stores a pointer to the next node in the column, and the fourth stores the coordinates or the row and column number in which the data appears in the matrix.
- In doubly linked lists, A corresponding inverse pointer for every pointer in the multilinked list representation of a sparse matrix.

y \ x	0	1	2
0	0	25	0
1	0	0	0
2	17	0	5
3	19	0	0

Sparse matrix

MultiLinked Lists



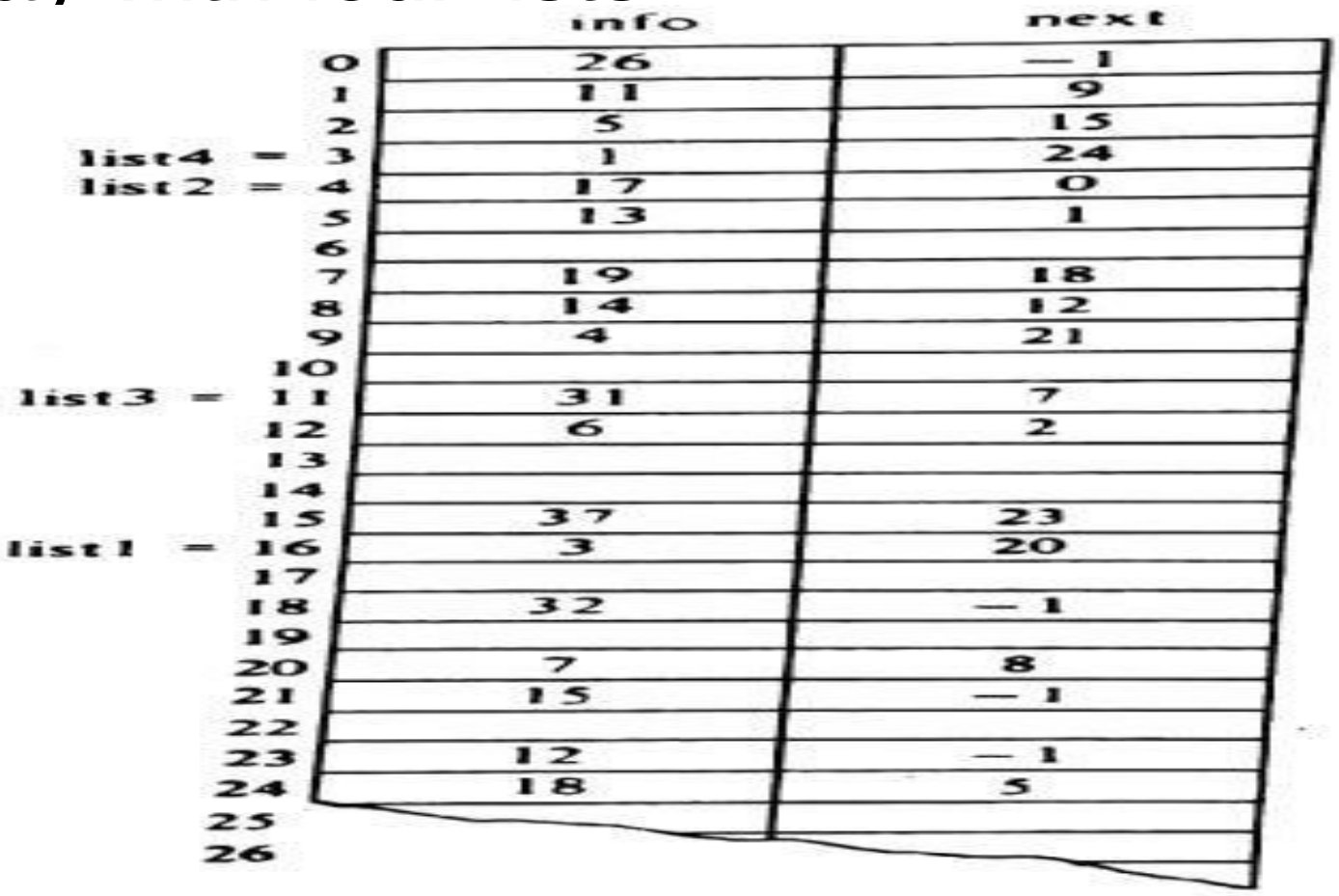
Multi-linked representation of sparse matrix shown in Fig. 6.72

Array Implementation of List

- Array of 500 nodes
- Pointer values 1 to NUMNODES1

```
#define NUMNODES 500
struct nodetype  {
    int info, next;
};
struct nodetype node[NUMNODES];
```

Array with four lists



Initial array

- Linked in natural order

```
avail = 0;
for (i = 0; i < NUMNODES-1; i++)
    node[i].next = i + 1;
node[NUMNODES-1].next = -1;
```

Allocate node

```
int getnode(void)
{
    int p;
    if (avail == -1)  {
        printf("overflow\n");
        exit(1);
    }
    p = avail;
    avail = node[avail].next;
    return(p);
} /* end getnode */
```

Free node

- Avail global

```
void freenode(int p)
{
    node[p].next = avail;
    avail = p;
    return;
} /* end freenode */
```

Insertion

```
void insafter(int p, int x)
{
    int q;
    if (p == -1)   {
        printf("void insertion\n");
        return;
    }
    q = getnode();
    node[q].info = x;
    node[q].next = node[p].next;
    node[p].next = q;
    return;
} /* end insafter */
```

Deletion

```
void delafter(int p, int *px)
{
    int q;
    if ((p == -1) || (node[p].next == -1))  {
        printf ("void deletion\n");
        return;
    }
    q = node[p].next;
    *px = node[q].info;
    node[p].next = node[q].next;
    freenode(q);
    return;
} /* end delafter */
```

Disadvantage

- Difficult to predict number of elements
- Number of nodes allocated is static cannot be released

THANKS