VIPS
योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

Vivekananda Institute
of Professional Studies

Linked List

Data structures
Dr. Nivedita Palia

Structures

It is a data type used for handling a group of logically related data items.
For example: record of student : name, roll number, marks

```
struct  personal
{
      char    name[20];
      int     day;
      char    month[10];
      int     year;
      float   salary;
};
main()
{
      struct personal person;

      printf("Input Values\n");
      scanf("%s %d %s %d %f",
                person.name,
                &person.day,
                person.month,
                &person.year,
                &person.salary);
      printf("%s %d %s %d %f\n",
                person.name,
                person.day,
                person.month,
                person.year,
                person.salary);
   }
```

```
#include <stdio.h>
struct employee
                {
int emp_id;
                char emp_name[20];
                } e1;
int main()
{
struct employee *e;
printf("enter the details of employee1\n");

scanf("%d%s",&e1.emp_id,e1.emp_name);
printf("employees details\nemployee id
=%d\nemployee name =
%s",e1.emp_id,e1.emp_name);
e=&e1;
printf("\n%d%s",e->emp_id,e->emp_name);
                return 0;
}
```

Self Referential Structure

A self-referential structure is a structure that can have members which point to a structure variable of the same type

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

Linked List

- A linked list is a data structure which allows to store data dynamically and manage data efficiently.
- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.
- There are three common types of Linked List.

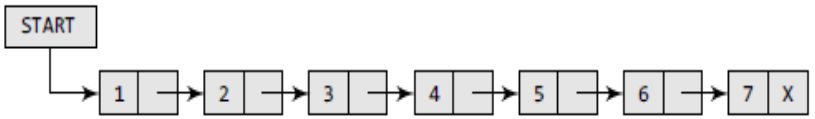


Figure 6.1 Simple linked list

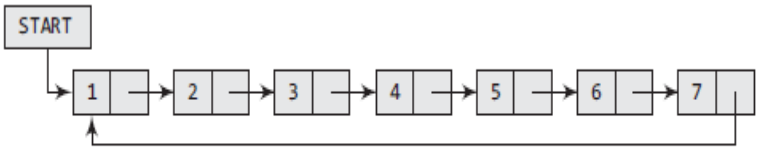


Figure 6.26 Circular linked list

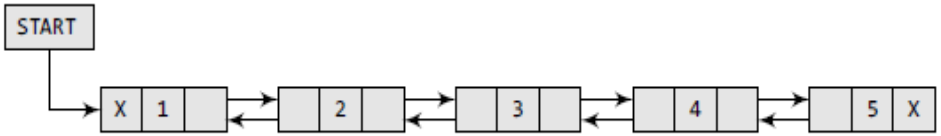


Figure 6.37 Doubly linked list

1.Singly Linked List

2.Doubly Linked List

3.Circular Linked List

Linked list as Abstract data type

- Linked List is an Abstract Data Type (ADT) that holds a collection of Nodes, the nodes can be accessed in a sequential way. Linked List doesn't provide a random access to a Node.

- Usually, those Nodes are connected to the next node and/or with the previous one, this gives the linked effect. When the Nodes are connected with only the next pointer the list is called Singly Linked List and when it's connected by the next and previous the list is called Doubly Linked List.
- Perform following operations

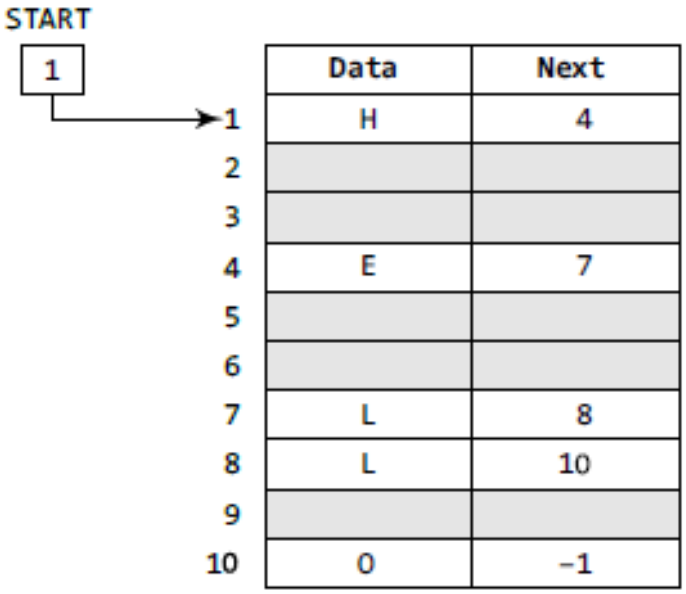
Insertion at the beginning, end, at particular position

Deletion at the beginning, end, at particular position

Traversal

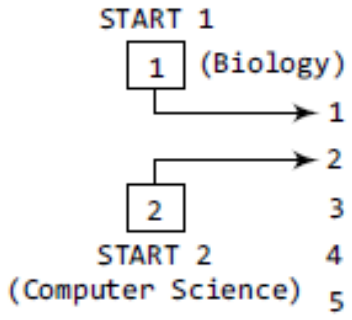
Reversal

Memory Representation of Linked List



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

Figure 6.2 START pointing to the first element of the linked list in the memory



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

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

Arrays Vs Linked List

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Array vs Linked List

Free Pool/ Avail List

list of available space is called the *free pool/ Avail list*

START 1 (Biology)

AVAIL 4

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

Figure 6.6 Linked list with AVAIL and START pointers

START 1 (Biology)

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

(a)

START 1 (Biology)

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

(b)

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

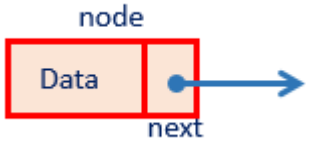
Garbage Collection: The operating system scans through all the memory cells and marks those cells that are being used by some program. Then it collects all the cells which are not being used and adds their address to the free pool, so that these cells can be reused by other programs. This process is called *garbage collection*.

Singly Linked list

Figure 6.7 Singly linked list

- A start pointer stores address of the next node.
- Each node contains two parts information and link(pointer to next node)
- Link of the last node is NULL (indicates end of linked list).

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



Note:
Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Operations on Linked List

- Traversing a list
  - Printing, finding minimum, etc.
- Insertion of a node into a list
  - At front, end and anywhere, etc.
- Deletion of a node from a list
  - At front, end and anywhere, etc.
- Comparing two linked lists
  - Similarity, intersection, etc.
- Merging two linked lists into a larger list
  - Union, concatenation, etc.
- Ordering a list
  - Reversing, sorting, etc.

Traversing a Singly Linked List

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

Figure 6.8 Algorithm for traversing a linked list

Creation and Traversal

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
struct node
{ int info;
struct node *link;
};
struct node *S = NULL; // empty linked list
struct node *one, *two, *ptr;
one = (struct node*)malloc(sizeof(struct node));
two = (struct node*)malloc(sizeof(struct node));
one->info = 1;
one->link=two;
two->info =2;
two->link=NULL;
S=one;
ptr=S;
```

```
while(ptr!=NULL)
{
        printf("%d\n", ptr->info);
        ptr=ptr->link;
}
return 0;
}
```

start

Searching in a Linked List

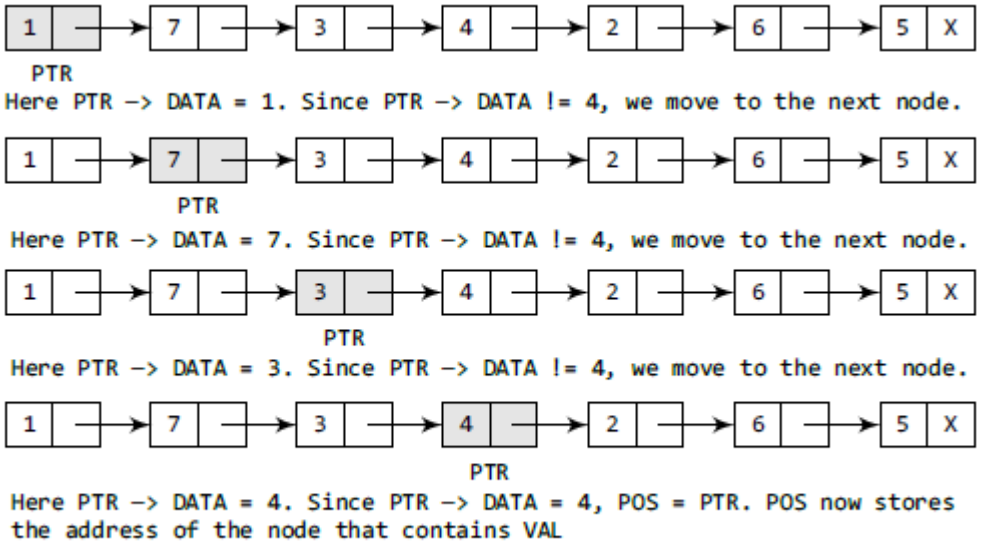


Figure 6.11 Searching a linked list

```
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
```

Figure 6.10 Algorithm to search a linked list

Insertion in linked list

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.

Overflow is a condition that occurs when `AVAIL = NULL` or no free memory cell is present in the system.

Insertion at the beginning

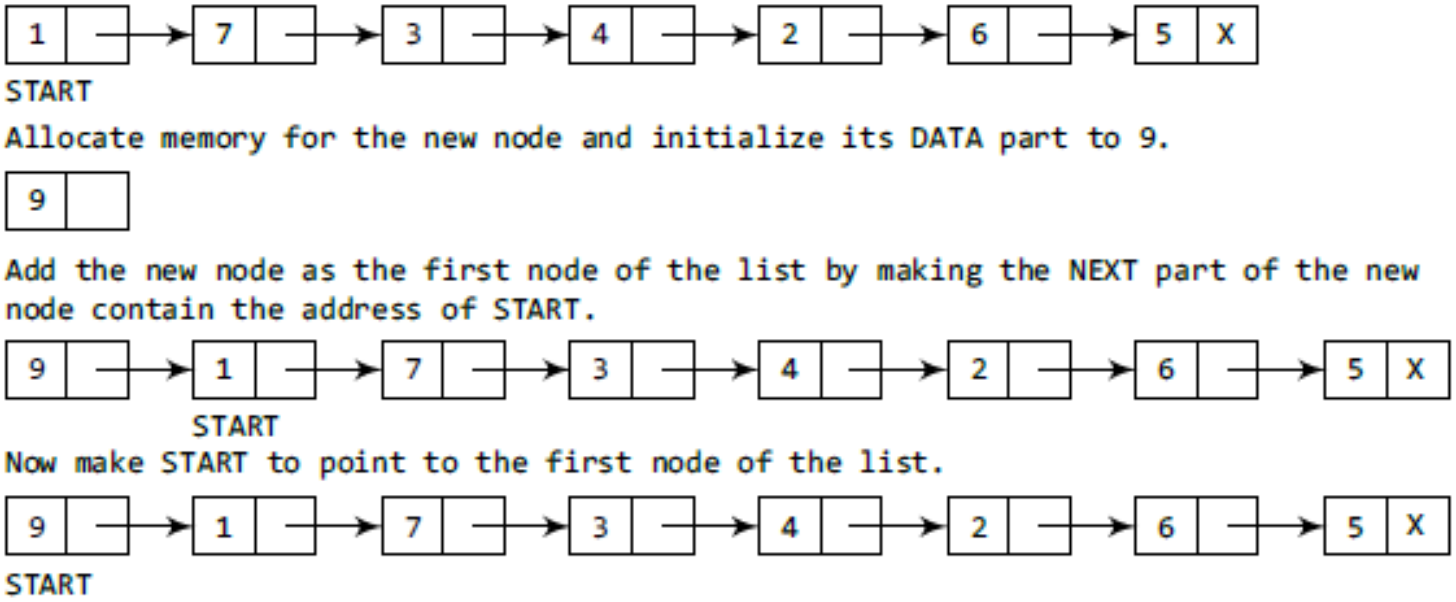


Figure 6.12 Inserting an element at the beginning of a linked list

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

Insertion at the end of the linked list

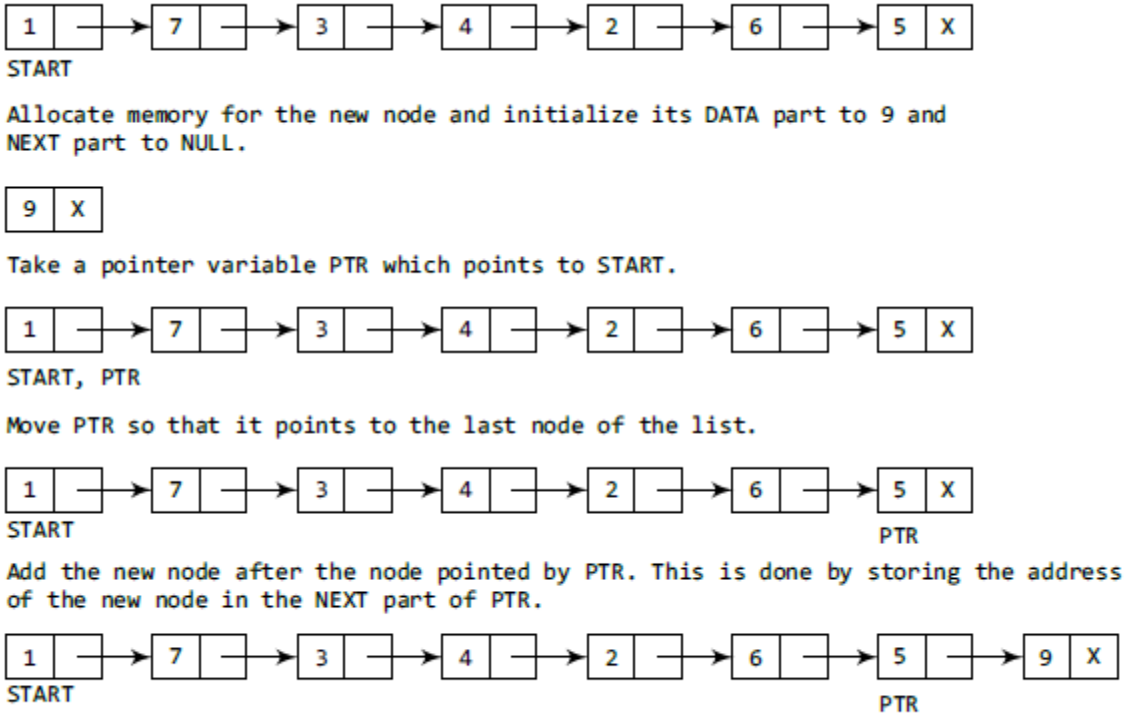


Figure 6.14 Inserting an element at the end of a linked list

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

Insertion after a given node

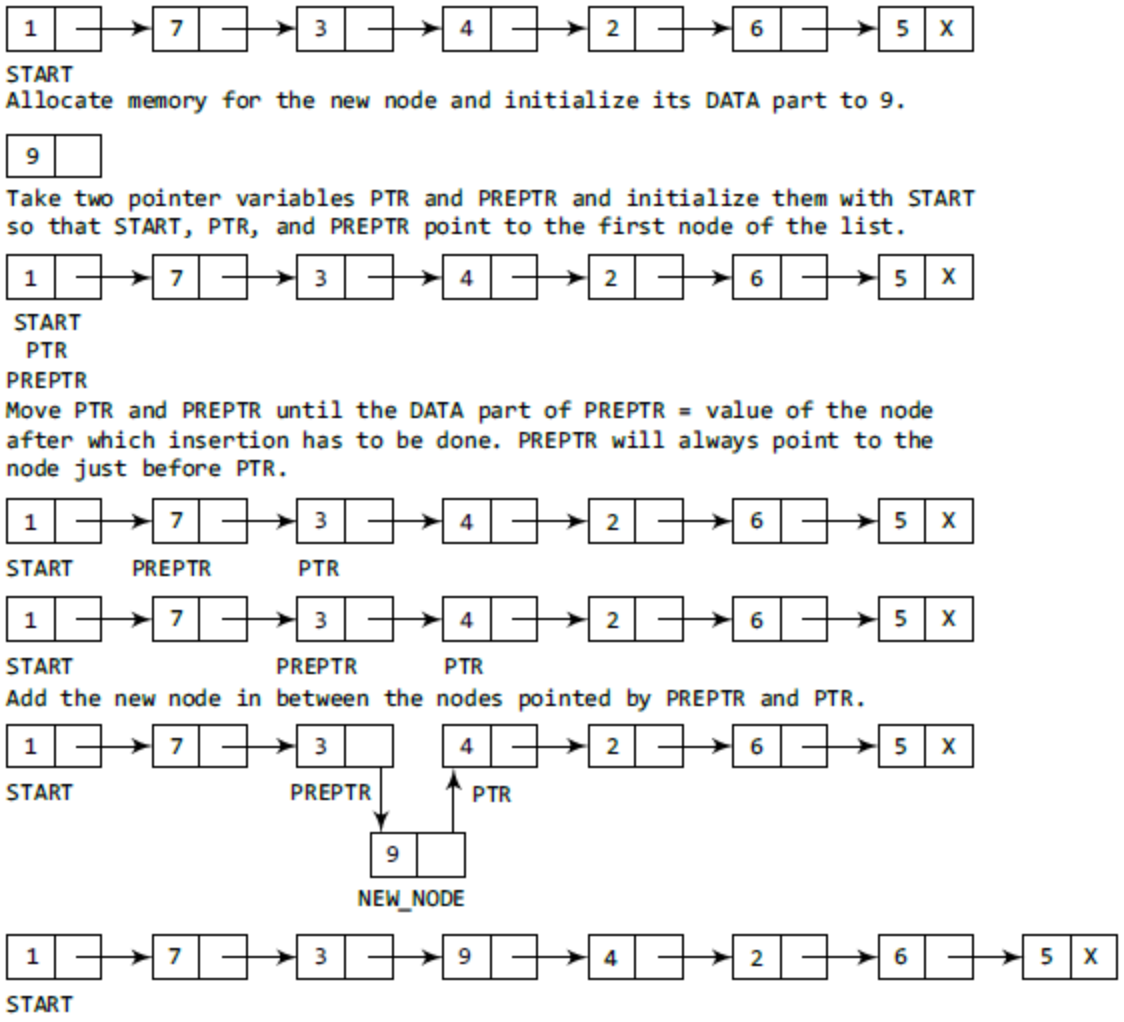


Figure 6.17 Inserting an element after a given node in a linked list

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

Deletion of a node from linked list

Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.

Underflow: when linked list is empty no deletion is possible such condition is known as underflow

Free Memory after deletion

- Do not forget to free() memory location dynamically allocated for a node after deletion of that node.
- It is the programmer's responsibility to free that memory block.
- Failure to do so may create a dangling pointer – a memory, that is not used either by the programmer or by the system.
- The content of a free memory is not erased until it is overwritten.

Deletion from first node

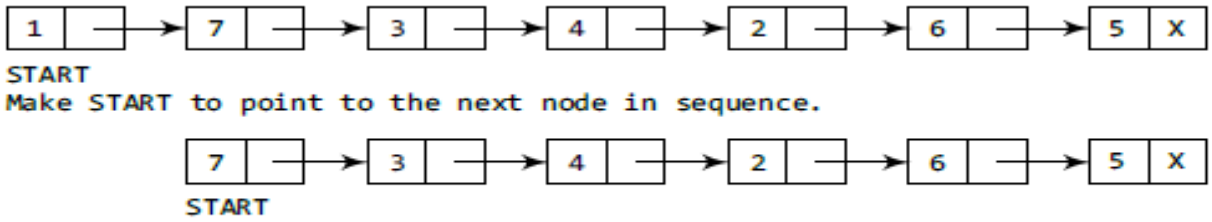


Figure 6.20 Deleting the first node of a linked list

```
struct node *delete_beg(struct
node *start)
{
struct node *ptr;
ptr = start;
if(ptr==NULL)
printf("underflow");
else{
start = start -> next;
free(ptr);
return start;
}}
```

Deletion of last node

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

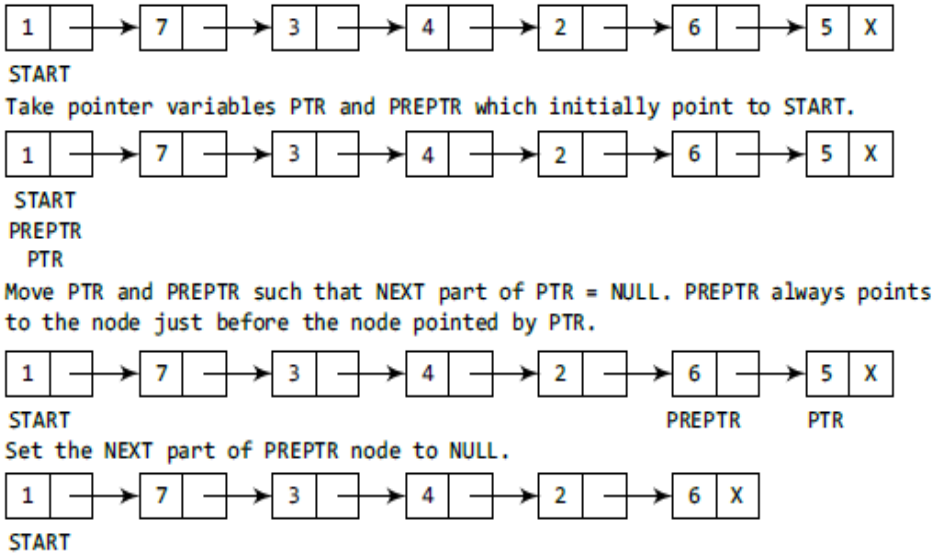


Figure 6.22 Deleting the last node of a linked list

Delete a node after given node

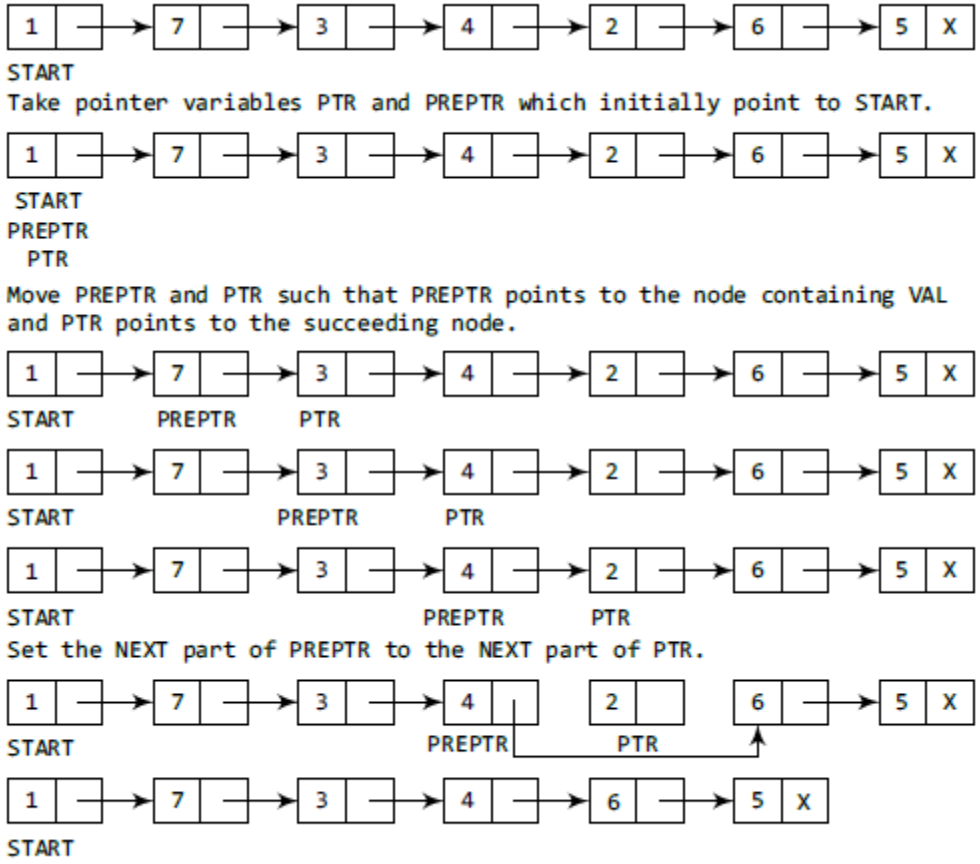


Figure 6.24 Deleting the node after a given node in a linked list

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

Reverse a linked list

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
struct node
{ int info;
struct node *link;
};
struct node *S = NULL; // empty linked list
struct node *one, *two,*three,
*ptr,*current,*prev=NULL,*next=NULL;
one = (struct node*)malloc(sizeof(struct node));
two = (struct node*)malloc(sizeof(struct node));
three=(struct node*)malloc(sizeof(struct node));
one->info = 1;
one->link=two;
two->info =2;
two->link=three;
three->info=3;
three->link=NULL;
S=one;
ptr=S;
```


```
ptr=S;
while(ptr!=NULL)
{
        printf("%d", ptr->info);
        ptr=ptr->link;
}
printf("\n");
current =S;
prev=NULL;
next=NULL;
while(current!=NULL){
next=current->link;
current->link=prev;
prev= current;
current=next;
}
ptr=prev;
while(ptr!=NULL)
{
        printf("%d", ptr->info);
        ptr=ptr->link;
}
return 0;
}
```

Doubly linked list

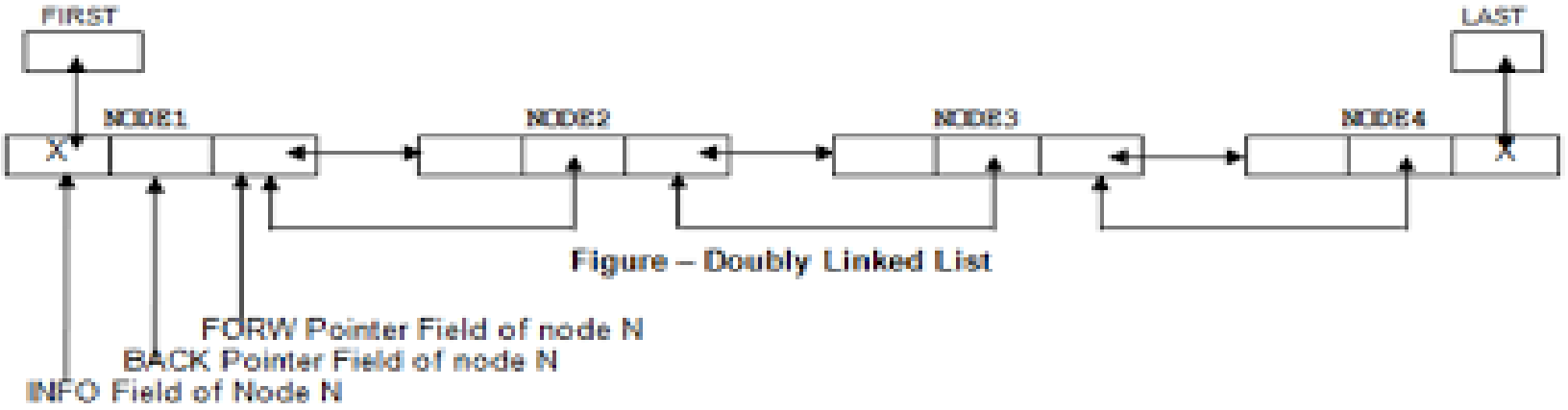


Figure – Doubly Linked List

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

Doubly linked list VS singly linked list

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

Disadvantages over singly linked list

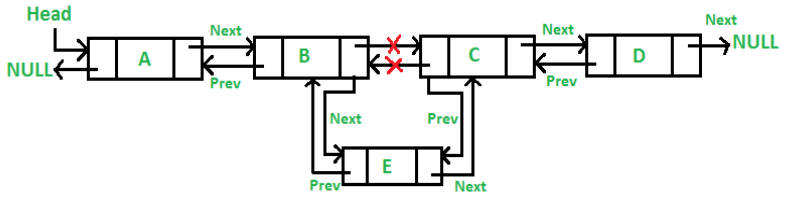
1) Every node of DLL Require extra space for an previous pointer.
2) All operations require an extra pointer previous to be maintained.

Operations on Doubly linked list

Insertion at beginning
Insertion at end
Insertion after given node
Traversal
Deletion at the beginning
Deletion at the end
Delete a node

Insertion after a given node

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



Circular Linked List

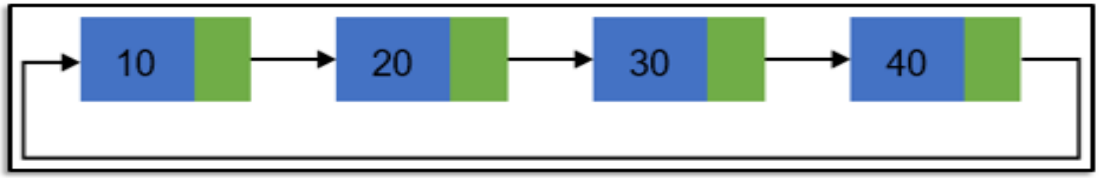
- The pointer from the last element in the list points back to the first element.
- A circular linked list is basically a linear linked list that may be single- or double-linked.

- The only difference is that there is no any NULL value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a circle with no end to stop it is called as **circular linked list**.

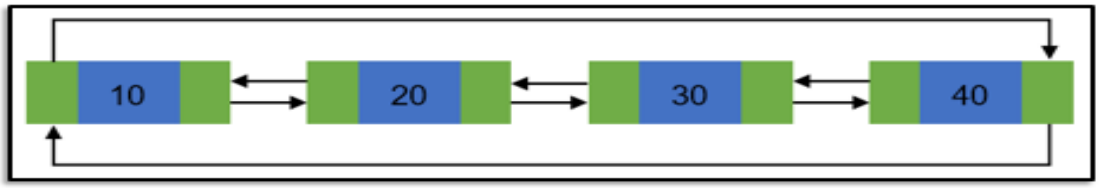
- In circular linked list there can be no starting or ending node, whole node can be traversed from any node.

- In order to traverse the circular linked list, only once we need to traverse entire list until the starting node is not traversed again.

Basic structure of singly circular linked list:



Doubly circular linked list:



Advantages and Disadvantages of Circular Linked List

Advantages of a Circular linked list

- Entire list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- Despite of being singly circular linked list we can easily traverse to its previous node, which is not possible in singly linked list.

Disadvantages of Circular linked list

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.

Creation and Traversal of Circular Linked List

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

/* Basic structure of Node */

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

int main()
{
    int n, data;
    head = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                    // function to create circular linked list
    displayList();                    // function to display the list

    return 0;
}
```

```
void createList(int n)
{
    int i, data;
    struct node *prevNode, *newNode;
    if(n >= 1){                          /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));

        printf("Enter data of 1 node: ");
        scanf("%d", &data);

        head->data = data;
        head->next = NULL;
        prevNode = head;

        for(i=2; i<=n; i++){        /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));

            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->next = NULL;
            prevNode->next = newNode; //Links the previous node with newly created node
            prevNode = newNode;       //Moves the previous node ahead
        }
            prevNode->next = head; //Links the last node with first node
           printf("\nCIRCULAR LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

```
void displayList()
{
    struct node *current;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        current = head;
        printf("DATA IN THE LIST:\n");

        do {
            printf("Data %d = %d\n", n, current->data);

            current = current->next;
            n++;
        }while(current != head);
    }
}
```

Header Linked List

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.

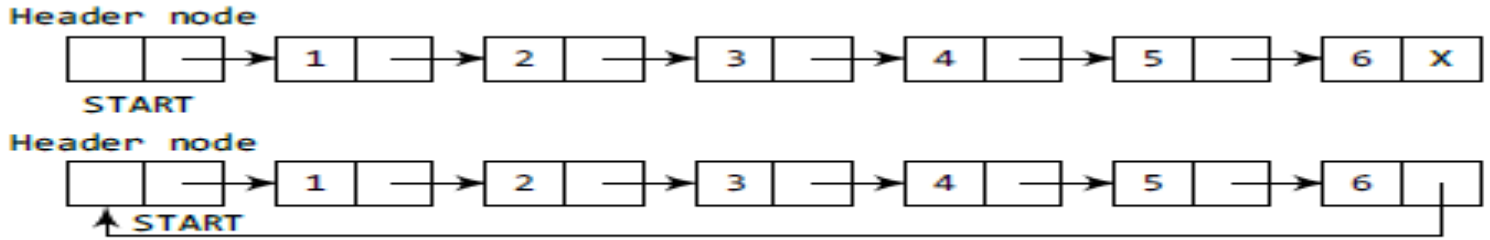


Figure 6.65 Header linked list

VIPS
योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

Vivekananda Institute
of Professional Studies

END