Chapter 04
Linked List

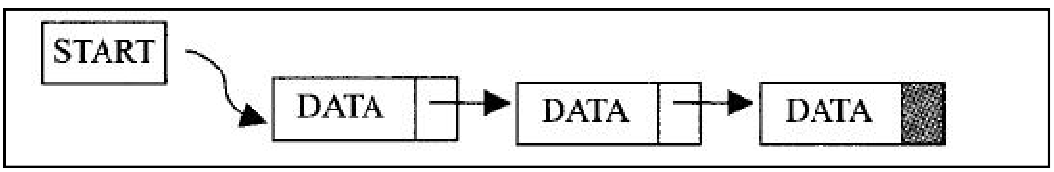
01 Definition:

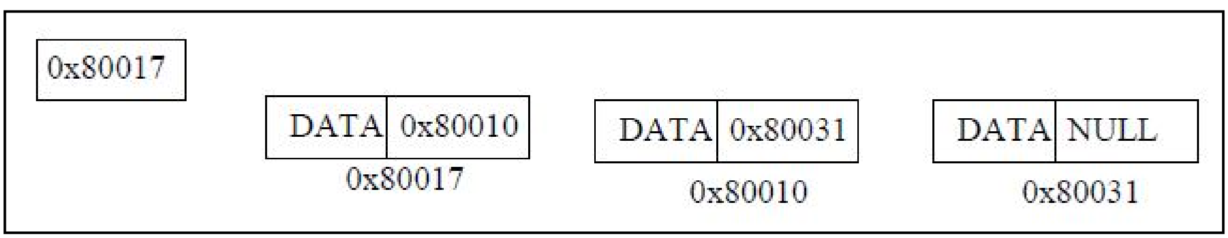
A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node.

A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

02 Linked list representations:

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



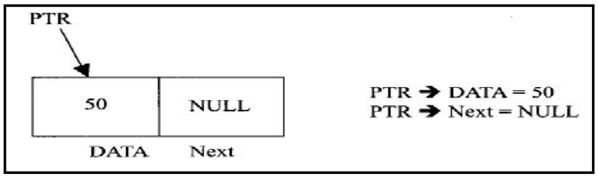


In Fig. 6.1, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The left part of the node which contains data may include a simple data type, an array, or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL. In Fig. 6.1, the NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list. Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

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

In C, we can implement a linked list using the following code:

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

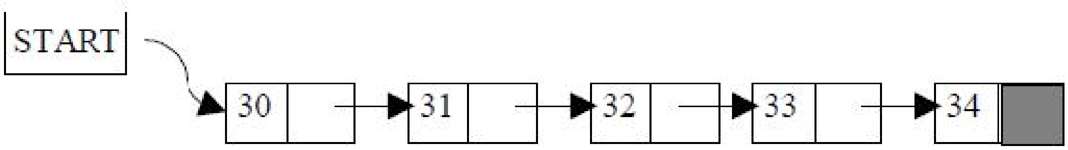


TYPES OF LINKED LIST:

1. Singly linked list
2. Doubly linked list
3. Circular linked list

1) 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.



Singly linked list Operation:

1. **Traversing a Linked List**
2. **Searching for a Value in a Linked List**
3. **Inserting a New Node in a Linked List**
    - **a.** The new node is inserted at the beginning.
    - **b.** The new node is inserted at the end.
    - **c.** The new node is inserted in between node.
4. **Deleting a Node from a Linked List**
    - **a.** The first node is deleted.
    - **b.** The last node is deleted.
    - **c.** The node after a given node is deleted.

A linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

1. **Traversing a Linked List:**

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.

Algorithm: traversing_linked_list
Input: --
Output: visit_each_element

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

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

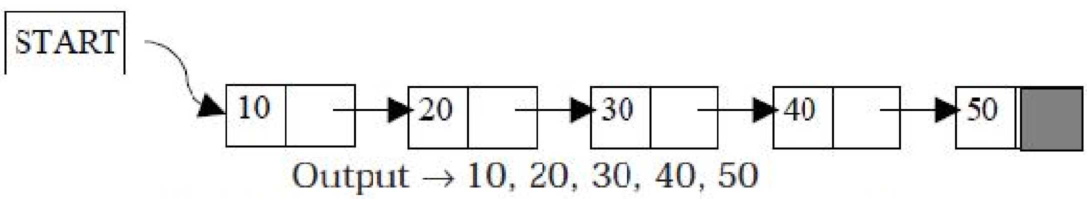


Fig. 5.10. Traversing the nodes from left to right

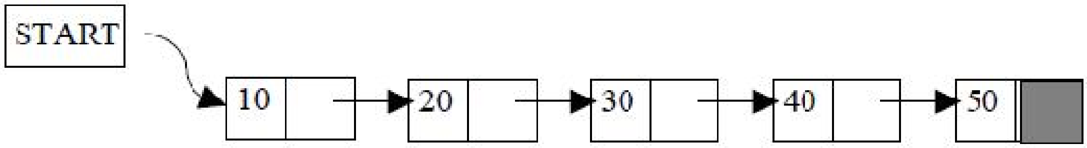
2. **Searching for a Value in a Linked List:**

Searching a linked list means to find a particular element in the linked list.

Algorithm: searching_linked_list
Input: VAL (Searching Element)
Output: POS (Position Searching Element)

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

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



Consider above list search element 30 which give result position 3rd in list. If search element 60 which give result NULL.

3. Inserting a New Node in a Linked List:

A new node is added into an already existing linked list.

Three Types of insertion new node in linked list:

a. The new node is inserted at the beginning:

Algorithm: insertion_at_beginning
Input: Data
Output: --

```
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
```

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

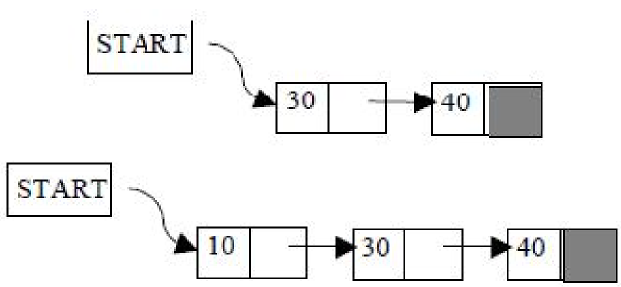


Fig. 5.7. Insert a node with DATA(10) at the beginning

b. The new node is inserted at the end:
Algorithm: insertion_at_end
Input: Data
Output: --

```
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
```

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

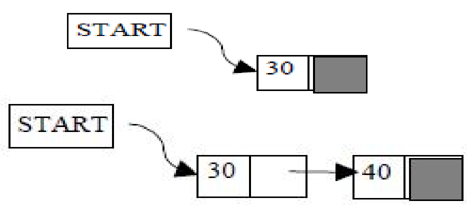
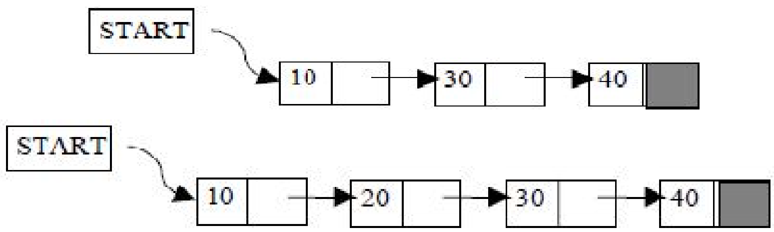


Fig. 5.6. Insert a node with DATA(40) at the end

c. The new node is insert a new node after a node:
Algorithm: insertion_after_node
Input: Data
Output: --

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

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



Insert node with DATA(20) after DATA(10).

4. Deleting a Node from a Linked List:

A node is deleted from an already existing linked list.

Three Types of deletion node from linked list:

a. The first node is deleted:
Algorithm: detetion_first_node
Input: --
Output: --

```
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
```

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

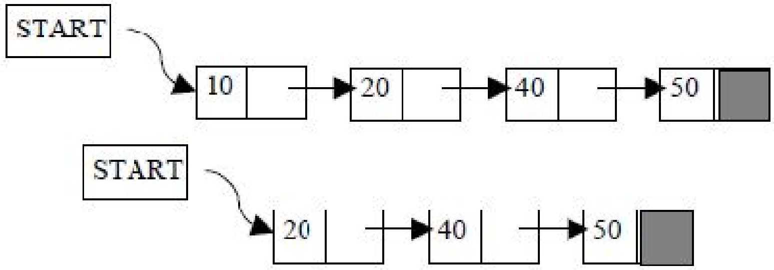


Fig. 5.12. Delete the 1st node

b. The last node is deleted:

Algorithm: deletion_end_node
Input: --
Output: --

```
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
```

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

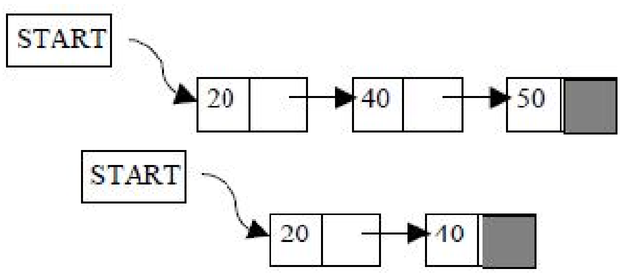


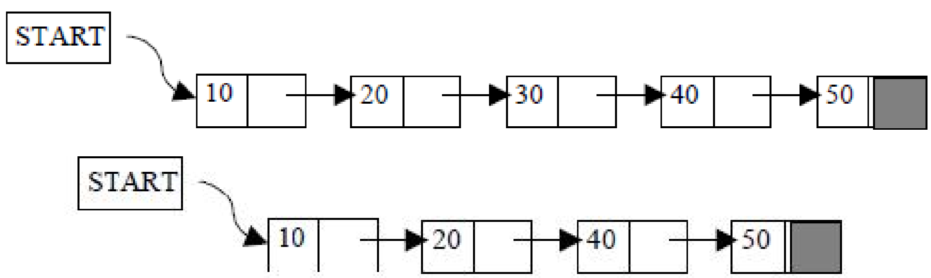
Fig. 5.13. Delete the last node

c. The node after a given node is deleted:

Algorithm: deletion_given_node
Input: Num (Data want to delete)
Output: --

```
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
```

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



Delete node with DATA(30) which present after DATA(20)

Applications of Linked List:

Implementation of linked list:

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


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

```
        }
        preptr -> next=ptr -> next;
        free(ptr);
        return start;
}


struct node *delete_list(struct node *start)
{
        struct node *ptr; // Lines 252-254 were modified from original code to fix
        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
```

02) Doubly linked list:

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every nodes in the doubly linked list has three fields: prev, next and DATA.

prev will point to the node in the left side (or previous node) that is prev will hold the address of the previous node. next will point to the node in the right side (or next node) that is next will hold the address of the next node. DATA will store the information of the node.

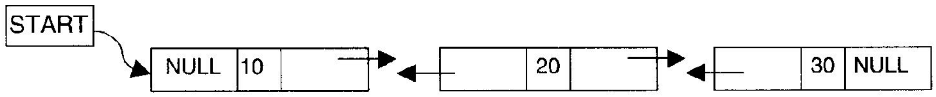


Fig. 5.25. Doubly Linked List

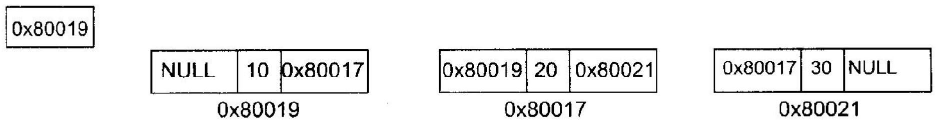


Fig. 5.26. Memory Representation of Doubly Linked List

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

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

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

Doubly linked list Operation:

1. **Traversing a Linked List**
2. **Searching for a Value in a Linked List**
3. **Inserting a New Node in a Linked List**
    - **a.** The new node is inserted at the beginning.
    - **b.** The new node is inserted at the end.
    - **c.** The new node is inserted in between node.
4. **Deleting a Node from a Linked List**
    - **a.** The first node is deleted.
    - **b.** The last node is deleted.
    - **c.** The node after a given node is deleted.

1. **Inserting a New Node in a Linked List:**

    - **a.** The new node is inserted at the beginning:

        Algorithm: doubly_insertion_end_node
        Input: VAL
        Output: --

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

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

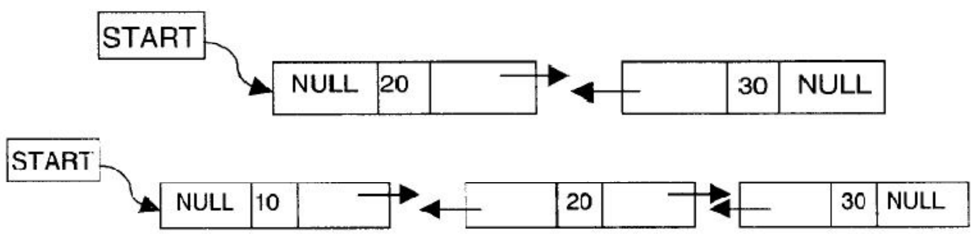


Fig 5.29. Insert (10) at the beginning

b. The new node is inserted at the end:
Algorithm: doubly_insertion_end_node
Input: VAL
Output: --

```
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
```

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).

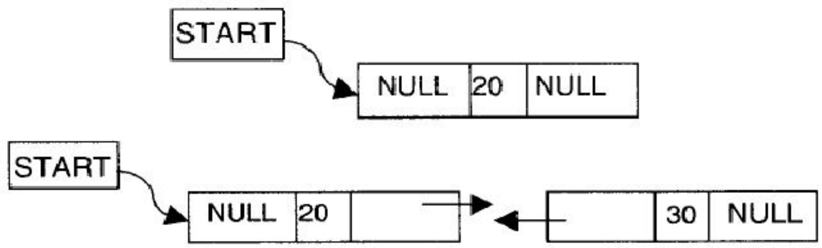


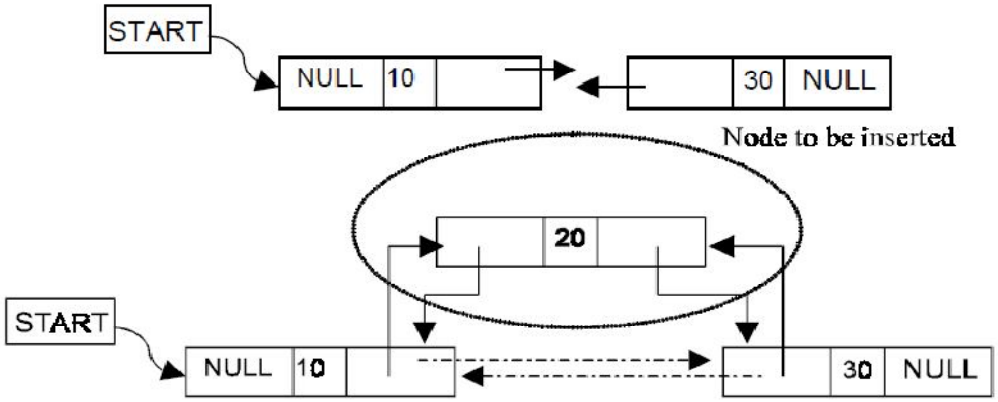
Fig 5.28. Insert (30) at the end

c. The new node is inserted in between node:

Algorithm: doubly_insertion_given_node
Input: VAL
Output: --

```
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
```

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.

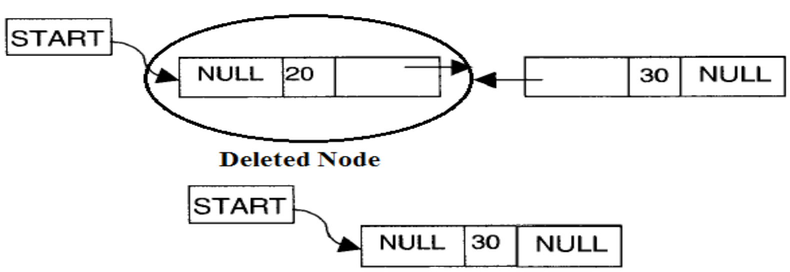


2. Deleting a Node from a Linked List:

a. The first node is deleted:
Algorithm: doubly_deletion_first_node
Input: --
Output: --

```
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
```

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

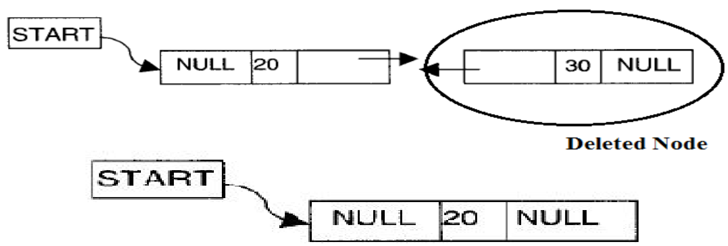


b. The last node is deleted:

Algorithm: doubly_deletion_last_node
Input:--
Output:--

```
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
```

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

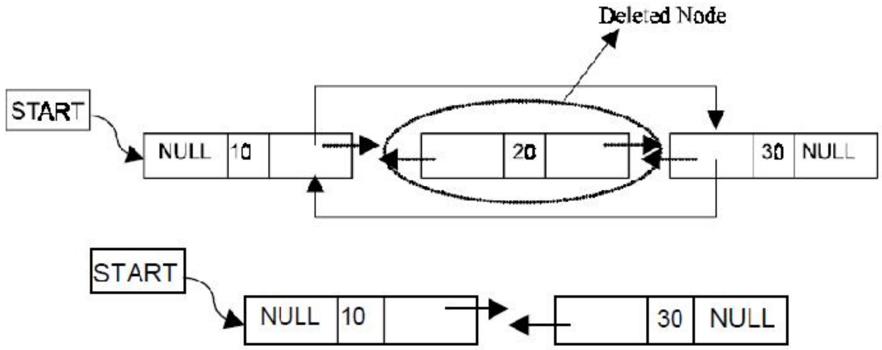


c. The node after a given node is deleted

Algorithm: doubly_deletion_after_node
Input: NUM
Output:--

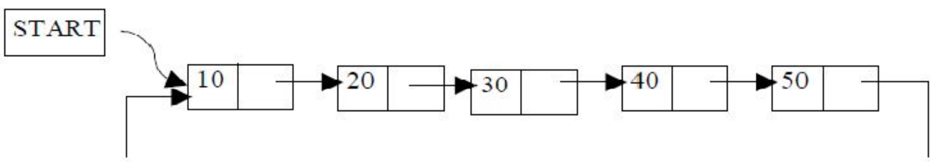
```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT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

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

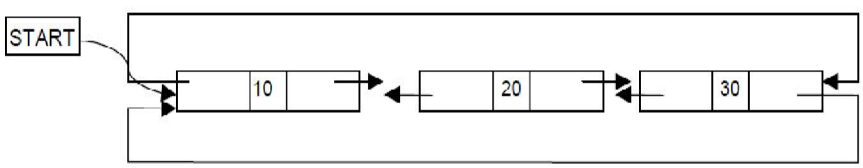


03 Circular linked lists:

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node.



A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner.



Algorithm: circular_list_insertion_beginning
Input: VAL
Output: --

```
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
```

Figure 6.30 Algorithm to insert a new node at the beginning

Algorithm: circular_list_insertion_end
Input: VAL
Output: --

```
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
```

Figure 6.32 Algorithm to insert a new node at the end

Algorithm: circular_list_deletion_first_node
Input: --
Output: --

```
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
```

Figure 6.34 Algorithm to delete the first node

Algorithm: circular_list_deletion_last_node
Input: --
Output: --

```
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
```

Figure 6.36 Algorithm to delete the last node

04 Linked representation of stack:

Implementation issues of the stack (Last In First Out - LIFO) using linked list.

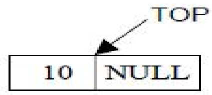


Fig. 5.11. push (10)

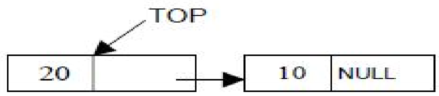


Fig. 5.12. push (20)

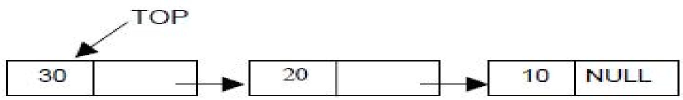


Fig. 5.13. push (30)

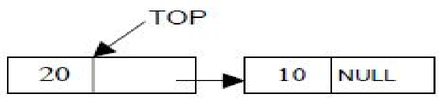


Fig. 5.14. X = pop() (ie; X = 30)

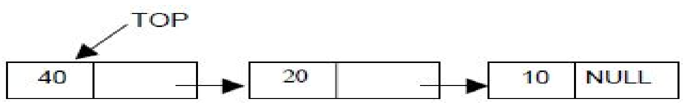


Fig. 5.15. push (40)

Algorithm: push_operation_stack_linked_list
Input: VAL
Output: --

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

Algorithm: pop_operation_stack_linked_list
Input:--
Output:--

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

05 Linked representation of Queue:

Queue is a First In First Out [FIFO] data structure.

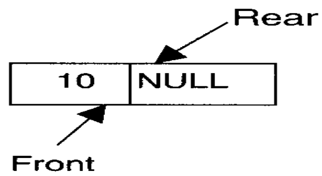


Fig. 5.16. push (10)

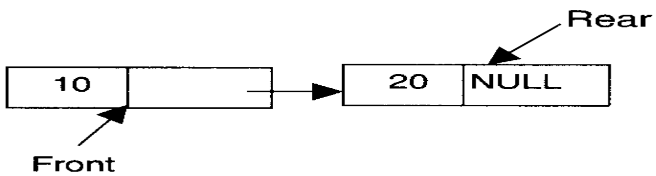


Fig. 5.17. push (20)

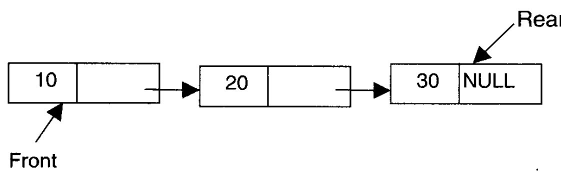


Fig. 5.18. push (30)

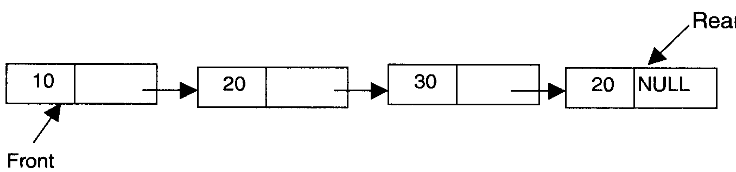


Fig. 5.19. push (40)

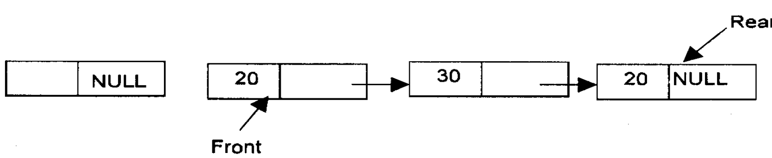


Fig. 5.20. X = pop() (*i.e.*; X = 10)

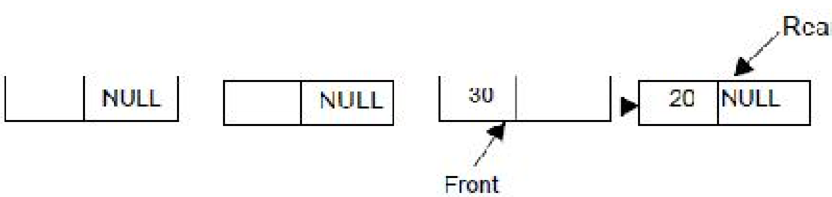


Fig. 5.21. X = pop() (*i.e.*; X = 20)

Algorithm: insert_element_queue_linked_list
Input: VAL
Output: --

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Figure 8.9 Algorithm to insert an element in a linked queue

Algorithm: delete_element_queue_linked_list
Input:--
Output:--

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

Figure 8.12 Algorithm to delete an element from a linked queue