

LINKED LIST

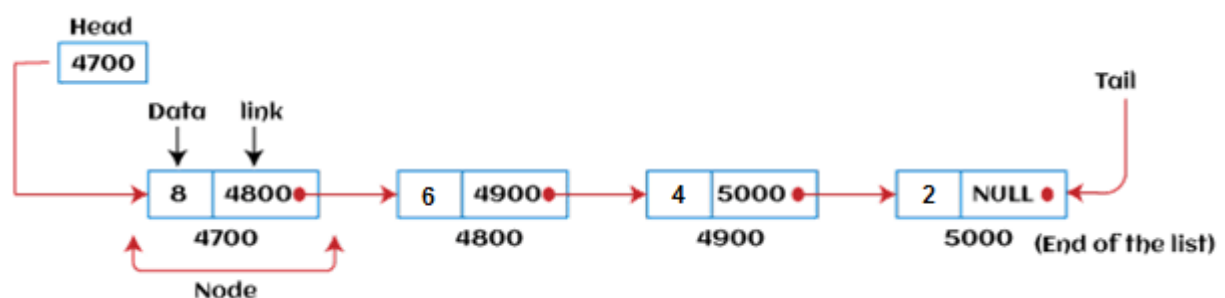
Basic Concepts – Definition and Representation of linked list, Types of linked lists - Singly linked list, doubly linked list, Circular linked list, doubly circular linked list; Representation of Linked list in Memory; Operations on Singly linked lists – Traversing, Searching, Insertion, and Deletion.

Definition and Representation of linked list:

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked List

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages that must be known to decide the data structure that will be used throughout the program.

Why use Linked List over an Array?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
 - In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.
-

How to declare a Linked List?

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type structure.

The declaration of linked list is given as follows -

```
1. struct node
2. {
3. int data;
4. struct node *next;
5. }
```

In the above declaration, we have defined a structure named as node that contains two variables, one is data that is of integer type, and another one is next that is a pointer which contains the address of next node.

Operations performed on Linked List

The basic operations that are supported by a list are mentioned as follows -

- **Insertion** : This operation is performed to add an element into the list.
 - **Deletion** : It is performed to delete an operation from the list.
 - **Traversal** : It is performed to traverse the elements of the list.
 - **Search** : It is performed to search an element from the list using the given key.
-

Advantages of Linked List

The advantages of using the Linked list are given as follows -

- **Dynamic data structure** : The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
 - **Insertion and deletion** : Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
 - **Memory efficient** : The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
 - **Implementation** : We can implement both stacks and queues using linked list.
-

Disadvantages of Linked List

The limitations of using the Linked list are given as follows -

- Memory usage : In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
 - Traversal : Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
 - Reverse traversing : Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.
-

Applications of Linked List

The applications of the Linked list are given as follows:

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
 - A linked list can be used to represent the sparse matrix.
 - The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
 - Using linked list, we can implement stack, queue, tree, and other various data structures.
 - The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
 - A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.
-

Types of Linked list

Linked list is classified into the following types -

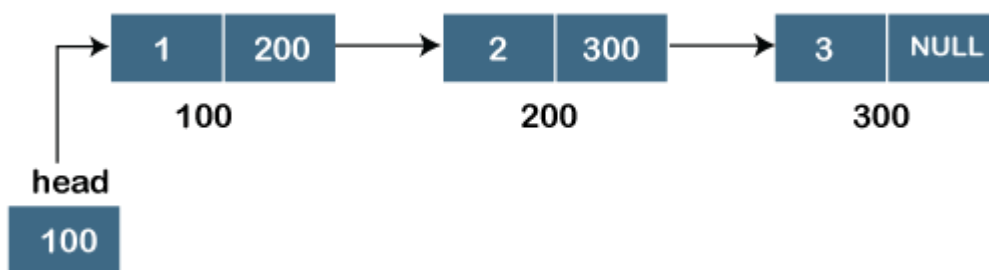
- Singly-linked list : Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- Doubly linked list : Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).
- Circular singly linked list : In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- Circular doubly linked list : Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node.

Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

1) Singly Linked List

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a pointer.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a *head pointer*.

The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. }
```

In the above representation, we have defined a user-defined structure named a node containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Representation of the node in a singly linked list

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. }
```

In the above representation, we have defined a user-defined structure named a node containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Program:

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

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

```
Struct Node {
    Int data;
    Struct Node *next;
};
```

```
Struct Node *head = NULL;
```

```
void insertAtBeginning(int val) {
    Struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = head;
    head = newNode;
}
```

```
void deleteFromBeginning() {
    If(head == NULL) printf("List is empty!\n");
    Else {
```

```

    Struct Node *temp = head;

    Head = head->next;

    Printf("Deleted: %d\n", temp->data);

    Free(temp);

}

}

```

```

Void display() {
    Struct Node *current = head;

    Printf("List: ");

    While(current != NULL) {
        Printf("%d ", current->data);

        Current = current->next;
    }

    Printf("\n");
}

```

```

Int main() {
    insertAtBeginning(30);
    insertAtBeginning(20);
    insertAtBeginning(10);
    display();

    deleteFromBeginning();
    display();

    return 0;
}

```

Output:

List: 10 20 30

Deleted: 10

List: 20 30

Explanation:

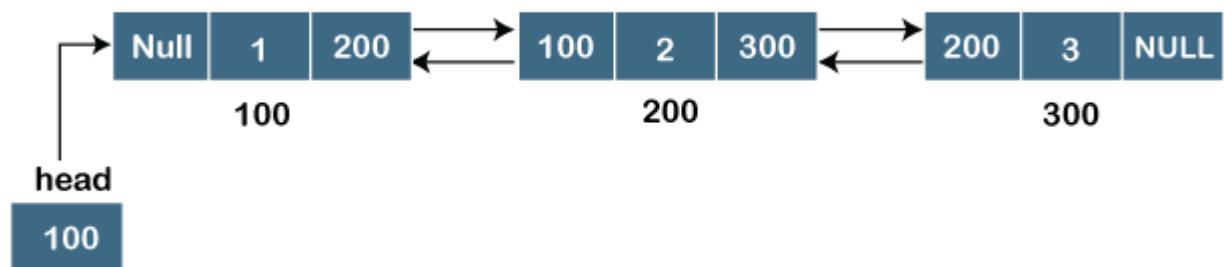
This implements a singly linked list with:

- Insertion at beginning
- Deletion from beginning
- Traversal to display list

2) Doubly Linked List

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the *address of the next* while the other part of the node stores the *previous node's address*. The initial node in the doubly linked list has the NULL value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5.     struct node *prev;
6. }
```

In the above representation, we have defined a user-defined structure named *a node* with three members, one is data of integer type, and the other two are the pointers, i.e., next and prev of the node type. The next pointer variable holds the address of the next node, and the prev pointer holds the address of the previous node. The type of both the pointers, i.e., next and

prev is struct node as both the pointers are storing the address of the node of the *struct node* type.

Program:

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

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

```
// Node structure
```

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

```
struct Node* head = NULL;
```

```
// Function to insert at the beginning
```

```
void insertAtBeginning(int value) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = value;  
    newNode->prev = NULL;  
    newNode->next = head;  
  
    if (head != NULL)  
        head->prev = newNode;  
  
    head = newNode;  
    printf("Inserted %d at the beginning.\n", value);  
}
```

```
// Function to insert at the end
```

```
void insertAtEnd(int value) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    struct Node* temp = head;  
    newNode->data = value;
```



```
newNode->next = NULL;
```

```
if (head == NULL) {  
    newNode->prev = NULL;  
    head = newNode;  
    printf("Inserted %d at the end (first node).\n", value);  
    return;  
}
```

```
while (temp->next != NULL)  
    temp = temp->next;
```

```
temp->next = newNode;  
newNode->prev = temp;  
printf("Inserted %d at the end.\n", value);  
}
```

// Function to delete from the beginning

```
void deleteFromBeginning() {  
    if (head == NULL) {  
        printf("List is empty. Nothing to delete.\n");  
        return;  
    }
```

```
    struct Node* temp = head;  
    head = head->next;
```

```
    if (head != NULL)  
        head->prev = NULL;
```

```
    printf("Deleted %d from the beginning.\n", temp->data);  
    free(temp);  
}
```

```

// Function to delete from the end
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = head;

    if (temp->next == NULL) {
        printf("Deleted %d from the end (only node).\n", temp->data);
        free(temp);
        head = NULL;
        return;
    }

    while (temp->next != NULL)
        temp = temp->next;

    temp->prev->next = NULL;
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}

// Function to display the list
void displayForward() {
    struct Node* temp = head;

    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Doubly Linked List (forward): ");

```

```

while (temp != NULL) {
    printf("%d <-> ", temp->data);
    temp = temp->next;
}
printf("NULL\n");
}

```

// Main function

```

int main() {
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtBeginning(5);
    insertAtBeginning(2);
    displayForward();

    deleteFromBeginning();
    displayForward();

    deleteFromEnd();
    displayForward();

    return 0;
}

```

Output:

Inserted 10 at the end (first node).

Inserted 20 at the end.

Inserted 5 at the beginning.

Inserted 2 at the beginning.

Doubly Linked List (forward): 2 <-> 5 <-> 10 <-> 20 <-> NULL

Deleted 2 from the beginning.

Doubly Linked List (forward): 5 <-> 10 <-> 20 <-> NULL

Deleted 20 from the end.

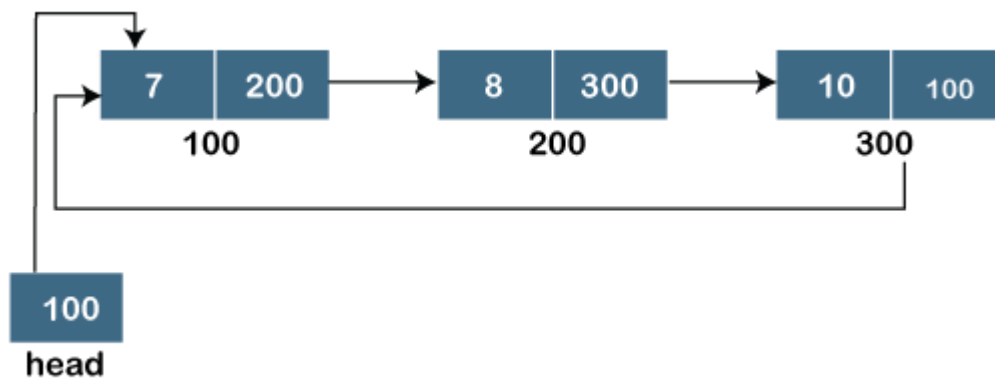
Doubly Linked List (forward): 5 <-> 10 <-> NULL

3) Circular Linked List

A circular linked list is a variation of a singly linked list. The only difference between the *singly linked list* and a *circular linked list* is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. }
```

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



Program:

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

```
// Node structure
```

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

```
struct Node* head = NULL;
```

// Function to insert at the end

```
void insertAtEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }

    printf("Inserted %d at the end.\n", value);
}
```

// Function to insert at the beginning

```
void insertAtBeginning(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head)
            temp = temp->next;

        temp->next = newNode;
    }
}
```

```

        newNode->next = head;
        head = newNode;
    }

    printf("Inserted %d at the beginning.\n", value);
}

// Function to delete from the beginning
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = head;

    if (head->next == head) {
        printf("Deleted %d from the beginning (only node).\n", head->data);
        free(head);
        head = NULL;
        return;
    }

    struct Node* last = head;
    while (last->next != head)
        last = last->next;

    head = head->next;
    last->next = head;

    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

```

```

// Function to display the list
void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("Circular Linked List: ");

    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);

    printf("(head)\n");
}

// Main function
int main() {
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtBeginning(5);
    insertAtBeginning(2);

    display();

    deleteFromBeginning();
    display();

    return 0;
}

```

Output:

Inserted 10 at the end.

Inserted 20 at the end.

Inserted 5 at the beginning.

Inserted 2 at the beginning.

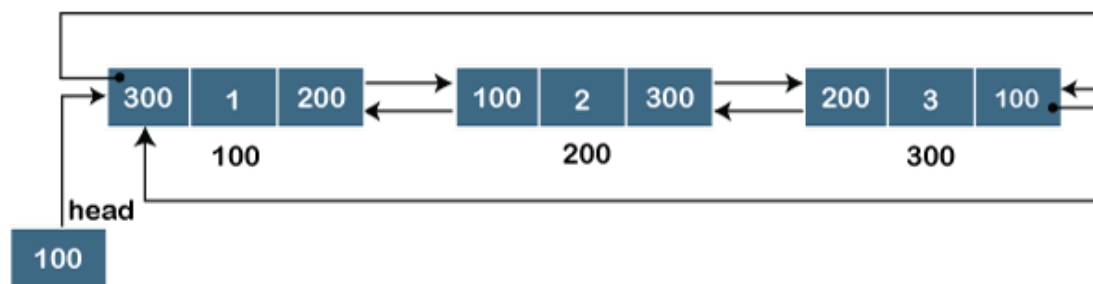
Circular Linked List: 2 -> 5 -> 10 -> 20 -> (head)

Deleted 2 from the beginning.

Circular Linked List: 5 -> 10 -> 20 -> (head)

4) Circular Doubly Linked List

The doubly circular linked list has the features of both the *circular linked list* and *doubly linked list*.



The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5.     struct node *prev;
6. }
```

Program:

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

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

```
// Node structure
```

```
struct Node {
    int data;
```



```

    struct Node* next;
    struct Node* prev;
};

struct Node* head = NULL;

// Function to insert at the end
void insertAtEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;

    if (head == NULL) {
        head = newNode;
        newNode->next = newNode;
        newNode->prev = newNode;
    } else {
        struct Node* tail = head->prev;

        newNode->next = head;
        newNode->prev = tail;

        tail->next = newNode;
        head->prev = newNode;
    }

    printf("Inserted %d at the end.\n", value);
}

// Function to insert at the beginning
void insertAtBeginning(int value) {
    insertAtEnd(value);
    head = head->prev;
    printf("Moved %d to the beginning.\n", head->data);
}

```

// Function to delete from the beginning

```
void deleteFromBeginning() {
```

```
    if (head == NULL) {
```

```
        printf("List is empty. Nothing to delete.\n");
```

```
        return;
```

```
    }
```

```
    if (head->next == head) {
```

```
        printf("Deleted %d (only node).\n", head->data);
```

```
        free(head);
```

```
        head = NULL;
```

```
        return;
```

```
    }
```

```
    struct Node* tail = head->prev;
```

```
    struct Node* temp = head;
```

```
    head = head->next;
```

```
    tail->next = head;
```

```
    head->prev = tail;
```

```
    printf("Deleted %d from the beginning.\n", temp->data);
```

```
    free(temp);
```

```
}
```

// Function to display the list forward

```
void displayForward() {
```

```
    if (head == NULL) {
```

```
        printf("List is empty.\n");
```

```
        return;
```

```
    }
```

```
    struct Node* temp = head;
```

```

printf("Circular Doubly Linked List (forward): ");

do {
    printf("%d <-> ", temp->data);
    temp = temp->next;
} while (temp != head);

printf("(head)\n");
}

// Function to display the list in reverse
void displayBackward() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head->prev;
    printf("Circular Doubly Linked List (backward): ");

    do {
        printf("%d <-> ", temp->data);
        temp = temp->prev;
    } while (temp != head->prev);

    printf("(tail)\n");
}

// Main function
int main() {
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtBeginning(5);
    insertAtBeginning(2);

```

```

displayForward();
displayBackward();

deleteFromBeginning();
displayForward();
displayBackward();

return 0;
}

```

Output:

Inserted 10 at the end.

Inserted 20 at the end.

Inserted 5 at the end.

Moved 5 to the beginning.

Inserted 2 at the end.

Moved 2 to the beginning.

Circular Doubly Linked List (forward): 2 <-> 5 <-> 10 <-> 20 <-> (head)

Circular Doubly Linked List (backward): 20 <-> 10 <-> 5 <-> 2 <-> (tail)

Deleted 2 from the beginning.

Circular Doubly Linked List (forward): 5 <-> 10 <-> 20 <-> (head)

Circular Doubly Linked List (backward): 20 <-> 10 <-> 5 <-> (tail)

Operations on Singly linked lists – Traversing, Searching, Insertion, and Deletion.

Note: For this concept refer the Lab program Part B Program number 6

Importance Questions:

1. **Explain** the structure and working of singly linked list with a neat diagram.
2. **Illustrate** the Deletion from the beginning operation with example program.
3. **Develop** a program to create a singly linked list and perform the following operations: Insertion at the beginning, Deletion from the beginning, and Traversal of the linked list.

4. **Demonstrate** the representation of circular linked list with neat diagram and example program.
5. **Illustrate** the Insertion from the beginning operation with example program.