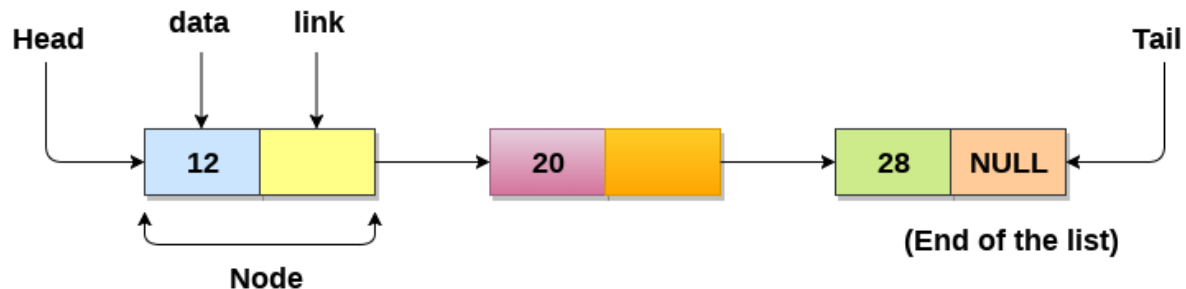


UNIT 3:

Linked Lists

- 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.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and be linked together to make a list. This achieves optimized utilization of space.
- List size is limited to the memory size and doesn't need to be declared in advance.
- Empty nodes cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were 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 which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

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

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing 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.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist 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.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

Data Structure	Time Complexity								Space Complexity
	Average				Worst				
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

1. struct node
2. {
3. int data;
4. struct node *next;
5. };
6. struct node *head, *ptr;
7. ptr = (struct node *)malloc(sizeof(struct node *));

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	<u>Insertion at beginning</u>	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	<u>Insertion at end of the list</u>	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	<u>Insertion after specified node</u>	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	<u>Deletion at beginning</u>	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	<u>Deletion at the end of the list</u>	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.

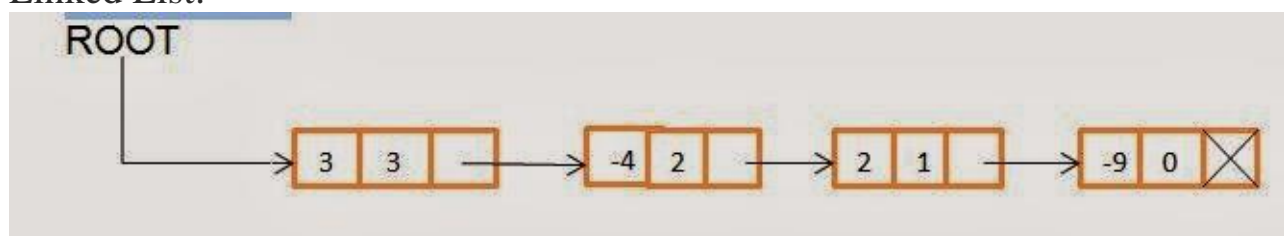
3	<u>Deletion after specified node</u>	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	<u>Traversing</u>	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	<u>Searching</u>	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Polynomial representation using Linked List

The linked list can be used to represent a polynomial of any degree. Simply the information field is changed according to the number of variables used in the polynomial. If a single variable is used in the polynomial the information field of the node contains two parts: one for coefficient of variable and the other for degree of variable. Let us consider an example to represent a polynomial using linked list as follows:

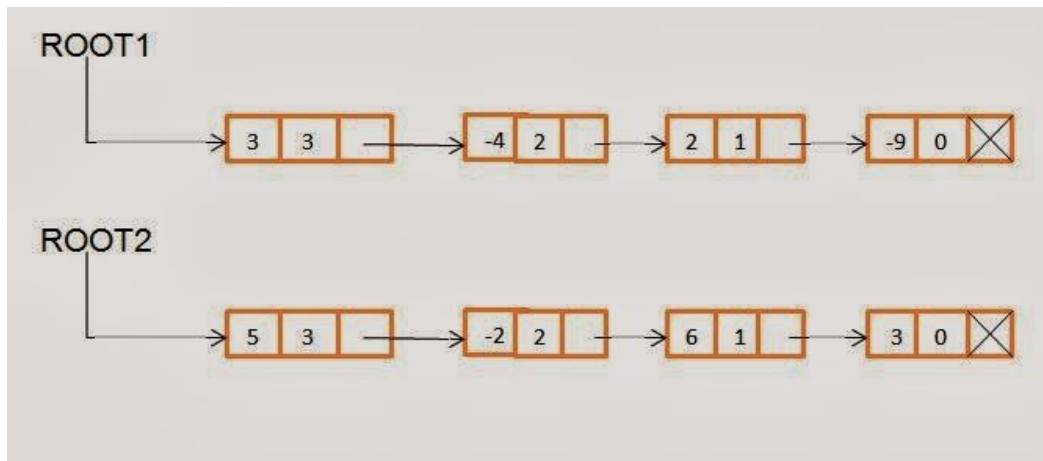
Polynomial: $3x^3-4x^2+2x-9$

Linked List:

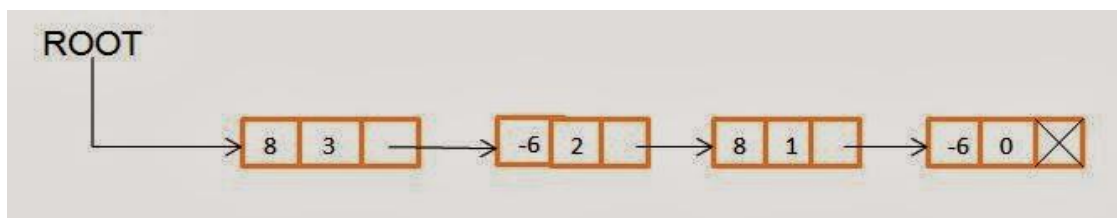


In the above linked list, the external pointer 'ROOT' points to the first node of the linked list. The first node of the linked list contains the information about the variable with the highest degree. The first node points to the next node with next lowest degree of the variable.

Representation of a polynomial using the linked list is beneficial when the operations on the polynomial like addition and subtractions are performed. The resulting polynomial can also be traversed very easily to display the polynomial.



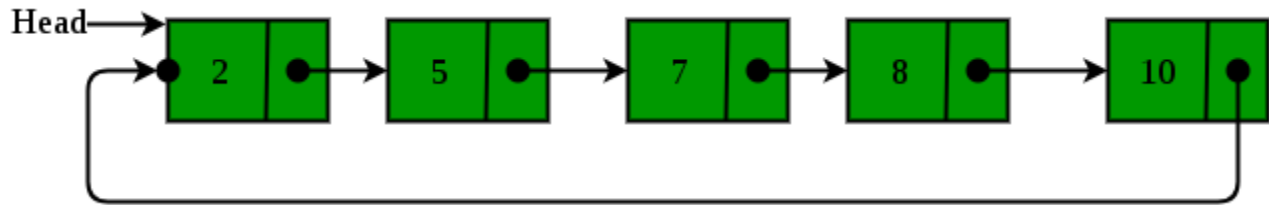
The above two linked lists represent the polynomials, $3x^3 - 4x^2 + 2x - 9$ and $5x^3 - 2x^2 + 6x + 3$ respectively. If both the polynomials are added then the resulting linked will be:



The linked list pointer ROOT gives the representation for polynomial, $8x^3 - 6x^2 + 8x - 6$.

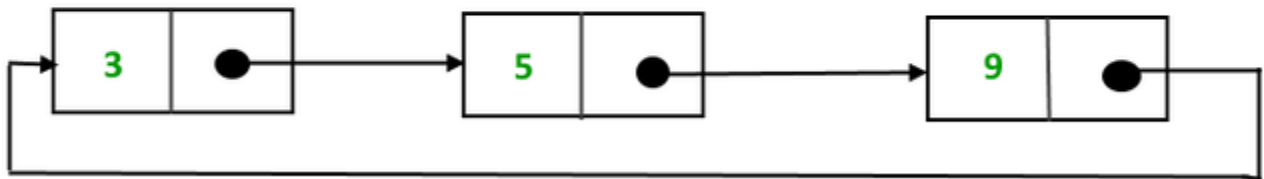
What is Circular linked list?

The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.



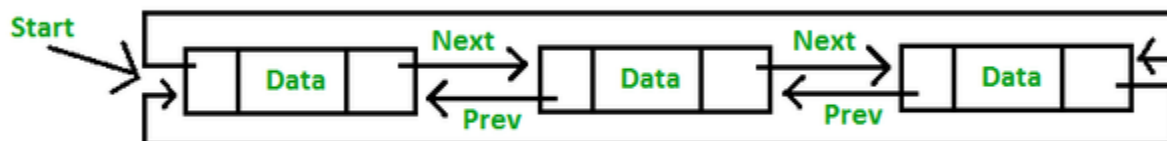
There are generally two types of circular linked lists:

- **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 traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



Representation of Circular singly linked list

- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Representation of circular doubly linked list

Note: We will be using the singly circular linked list to represent the working of the circular linked list.

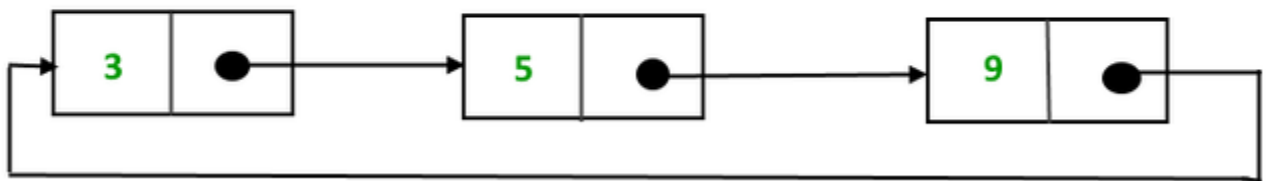
Representation of circular linked list:

Circular linked lists are similar to single Linked Lists with the exception of connecting the last node to the first node.

Node representation of a Circular Linked List:

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

Example of Circular singly linked list:



Example of circular linked list

The above Circular singly linked list can be represented as:

```
Node* one = createNode(3);  
  
Node* two = createNode(5);  
  
Node* three = createNode(9);  
  
  
// Connect nodes  
  
one->next = two;
```



```
two->next = three;  
  
three->next = one;
```

Explanation: In the above program one, two, and three are the node with values 3, 5, and 9 respectively which are connected in a circular manner as:

- **For Node One:** The Next pointer stores the address of Node two.
- **For Node Two:** The Next stores the address of Node three
- **For Node Three:** The Next points to node one.

Operations on the circular linked list:

We can do some operations on the circular linked list similar to the singly linked list which are:

1. Insertion
2. Deletion

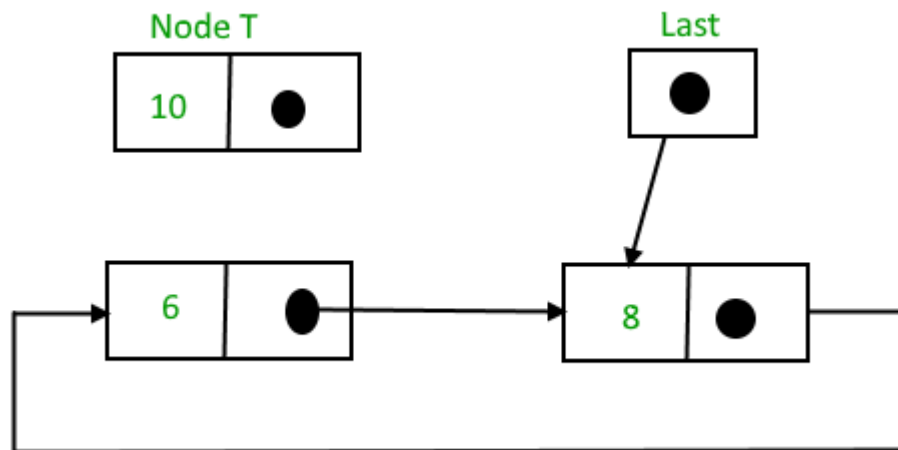
1. Insertion in the circular linked list:

A node can be added in three ways:

1. Insertion at the beginning of the list
2. Insertion at the end of the list
3. Insertion in between the nodes

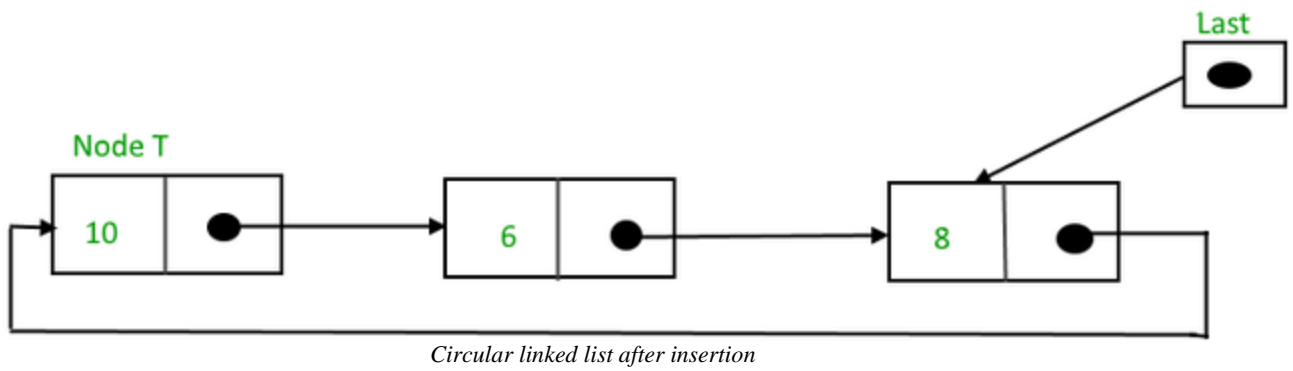
1) **Insertion at the beginning of the list:** To insert a node at the beginning of the list, follow these steps:

- Create a node, say T.
- Make T -> next = last -> next.
- last -> next = T.



Circular linked list before insertion

And then,



Below is the code implementation to insert a node at the beginning of the list:

```
void insertAtBeginning(int data) {  
  
    struct Node *newNode = (struct Node*) malloc(sizeof(struct Node));  
  
    newNode->data = data;  
  
  
    if (!head) {  
  
        head = newNode;  
  
        newNode->next = head;  
  
    } else {  
  
        struct Node *current = head;  
  
  
        while (current->next != head) {  
  
            current = current->next;  
  
        }  
  
    }  
}
```

```

newNode->next = head;

current->next = newNode;

head = newNode;

}

}

```

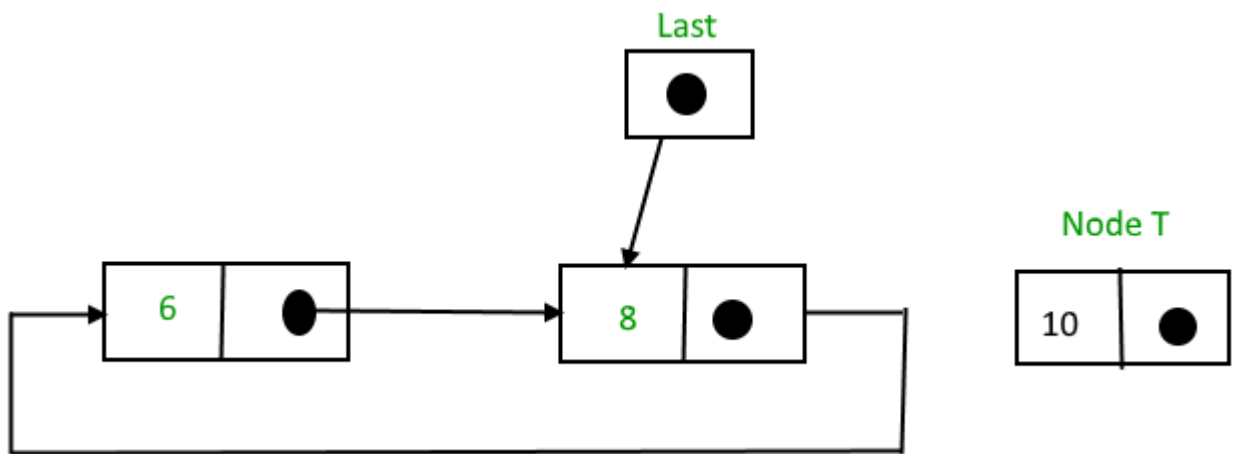
Time complexity: $O(1)$ to insert a node at the beginning no need to traverse list it takes constant time

Auxiliary Space: $O(1)$

2) **Insertion at the end of the list:** To insert a node at the end of the list, follow these steps:

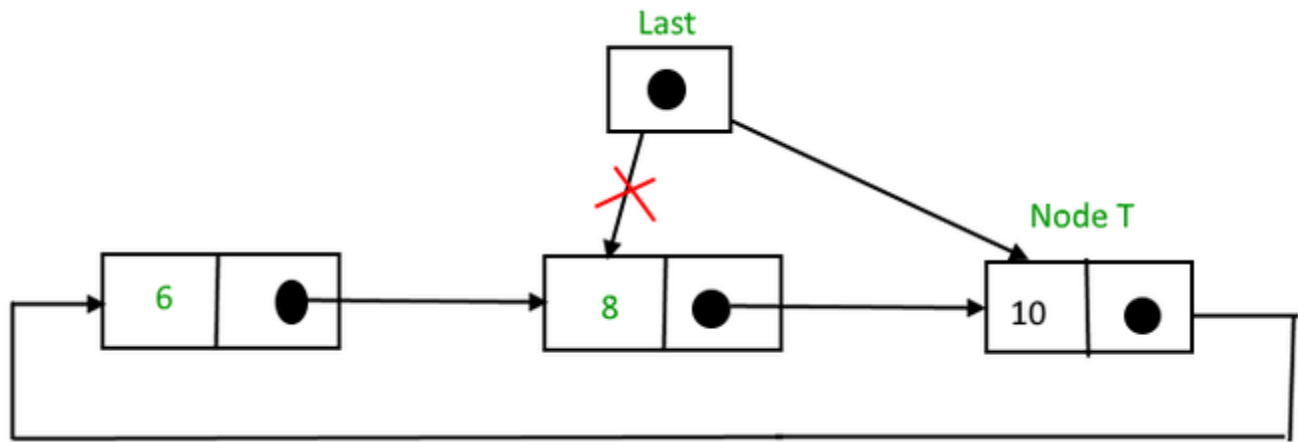
- Create a node, say T.
- Make $T \rightarrow \text{next} = \text{last} \rightarrow \text{next}$;
- $\text{last} \rightarrow \text{next} = T$.
- $\text{last} = T$.

Before insertion,



Circular linked list before insertion of node at the end

After insertion,



Circular linked list after insertion of node at the end

Below is the code implementation to insert a node at the beginning of the list:

```
void insertAtEnd(int newData) {  
  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
  
    newNode->data = newData;  
  
    if (!head) {  
  
        head = newNode;  
  
        newNode->next = head;  
  
    } else {  
  
        struct Node* current = head;  
  
        while (current->next != head) {
```

```

    current = current->next;

}

current->next = newNode;

newNode->next = head;

}

}

```

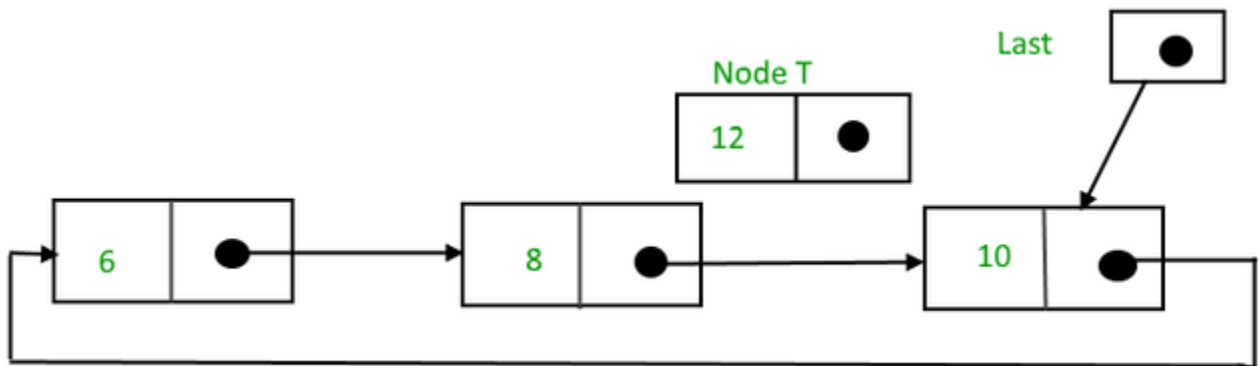
Time Complexity: $O(1)$ to insert a node at the end of the list. No need to traverse the list as we are utilizing the last pointer, hence it takes constant time.

Auxiliary Space: $O(1)$

3) **Insertion in between the nodes:** To insert a node in between the two nodes, follow these steps:

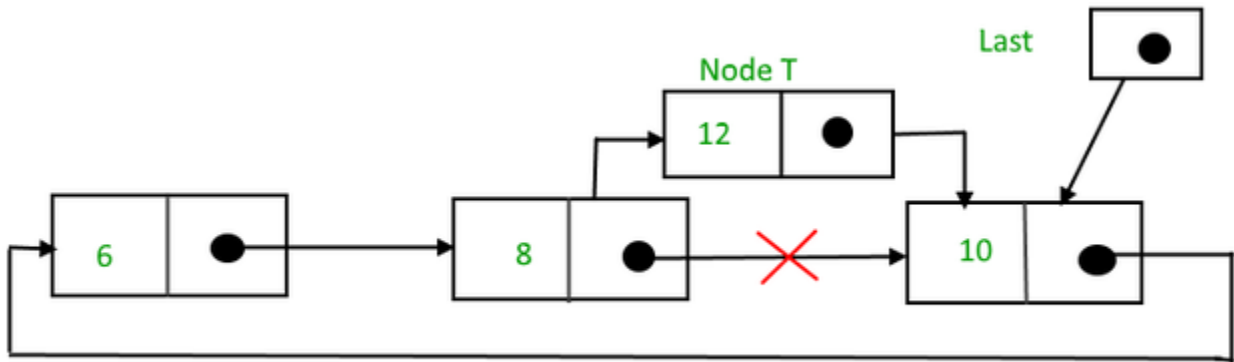
- Create a node, say T.
- Search for the node after which T needs to be inserted, say that node is P.
- Make $T \rightarrow \text{next} = P \rightarrow \text{next}$;
- $P \rightarrow \text{next} = T$.

Suppose 12 needs to be inserted after the node has the value 10,



Circular linked list before insertion

After searching and insertion,



Circular linked list after insertion

Below is the code to insert a node at the specified position of the List:

```

void insertAtPosition(int position, Node *newNode) {

    if (position < 1 || position > getSize() + 1) {

        printf("Invalid position!");

        return;

    }

    if (position == 1) {

        insertAtBeginning(newNode);

    } else if (position == getSize() + 1) {

        insertAtEnd(newNode);

    } else {

        Node *current = head;

```

```

    for (int i = 1; i < position - 1; i++) {

        current = current->next;

    }

    newNode->next = current->next;

    current->next = newNode;

}

}

```

Time Complexity: $O(N)$

Auxiliary Space: $O(1)$

2. Deletion in a circular linked list:

1) Delete the node only if it is the only node in the circular linked list:

- Free the node's memory
- The last value should be NULL A node always points to another node, so NULL assignment is not necessary.

Any node can be set as the starting point.

Nodes are traversed quickly from the first to the last.

2) Deletion of the last node:

- Locate the node before the last node (let it be temp)
- Keep the address of the node next to the last node in temp
- Delete the last memory
- Put temp at the end

3) Delete any node from the circular linked list: We will be given a node and our task is to delete that node from the circular linked list.

Algorithm:

Case 1: List is empty.

- If the list is empty we will simply return.

Case 2: List is not empty

- If the list is not empty then we define two pointers **curr** and **prev** and initialize the pointer **curr** with the **head** node.
- Traverse the list using **curr** to find the node to be deleted and before moving to curr to the next node, every time set $prev = curr$.

- If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr).
- If the list has more than one node, check if it is the first node of the list. Condition to check this (curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr.
- If curr is not the first node, we check if it is the last node in the list. Condition to check this is (curr -> next == head).
- If curr is the last node. Set prev -> next = head and delete the node curr by free(curr).
- If the node to be deleted is neither the first node nor the last node, then set prev -> next = curr -> next and delete curr.
- If the node is not present in the list return head and don't do anything.

Below is the implementation for the above approach:

```
#include <stdio.h>

#include <stdlib.h>

// Structure for a node

struct Node {

    int data;

    struct Node* next;

};

// Function to insert a node at the

// beginning of a Circular linked list

void push(struct Node** head_ref, int data)
```



```
{

    // Create a new node and make head

    // as next of it.

    struct Node* ptr1 = (struct Node*)malloc(sizeof(struct Node));

    ptr1->data = data;

    ptr1->next = *head_ref;


    // If linked list is not NULL then

    // set the next of last node

    if (*head_ref != NULL) {

        // Find the node before head and

        // update next of it.

        struct Node* temp = *head_ref;

        while (temp->next != *head_ref)

            temp = temp->next;

        temp->next = ptr1;

    }

    else
```

```

        // For the first node

        ptr1->next = ptr1;

        *head_ref = ptr1;
    }

// Function to print nodes in a given
// circular linked list
void printList(struct Node* head)
{
    struct Node* temp = head;

    if (head != NULL) {
        do {
            printf("%d ", temp->data);

            temp = temp->next;

        } while (temp != head);
    }

    printf("\n");
}

```

```

}

// Function to delete a given node
// from the list

void deleteNode(struct Node** head, int key)
{

    // If linked list is empty

    if (*head == NULL)

        return;

    // If the list contains only a
    // single node

    if ((*head)->data == key && (*head)->next == *head) {

        free(*head);

        *head = NULL;

        return;

    }

    struct Node *last = *head, *d;

```

```
// If head is to be deleted

if ((*head)->data == key) {

    // Find the last node of the list

    while (last->next != *head)

        last = last->next;

    // Point last node to the next of

    // head i.e. the second node

    // of the list

    last->next = (*head)->next;

    free(*head);

    *head = last->next;

    return;

}

// Either the node to be deleted is

// not found or the end of list

// is not reached
```

```
while (last->next != *head && last->next->data != key) {

    last = last->next;

}

// If node to be deleted was found

if (last->next->data == key) {

    d = last->next;

    last->next = d->next;

    free(d);

}

else

    printf("Given node is not found in the list!!!\n");

}

// Driver code

int main()

{

    // Initialize lists as empty

    struct Node* head = NULL;
```

```
// Created linked list will be  
  
// 2->5->7->8->10  
  
push(&head, 2);  
  
push(&head, 5);  
  
push(&head, 7);  
  
push(&head, 8);  
  
push(&head, 10);  
  
  
printf("List Before Deletion: ");  
  
printList(head);  
  
  
  
deleteNode(&head, 7);  
  
  
printf("List After Deletion: ");  
  
printList(head);  
  
  
  
return 0;  
  
}
```

Output

List Before Deletion: 10 8 7 5 2

List After Deletion: 10 8 5 2

Time Complexity: $O(N)$, Worst case occurs when the element to be deleted is the last element and we need to move through the whole list.

Auxiliary Space: $O(1)$, As constant extra space is used.

Advantages of Circular Linked Lists:

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of a queue. Unlike [this](#) implementation, we don't need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and the front can always be obtained as next of last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like the [Fibonacci Heap](#).
- Implementing a circular linked list can be relatively easy compared to other more complex data structures like trees or graphs.

Disadvantages of circular linked list:

- Compared to singly linked lists, circular lists are more complex.
- Reversing a circular list is more complicated than singly or doubly reversing a circular list.
- It is possible for the code to go into an infinite loop if it is not handled carefully.
- It is harder to find the end of the list and control the loop.
- Although circular linked lists can be efficient in certain applications, their performance can be slower than other data structures in certain cases, such as when the list needs to be sorted or searched.
- Circular linked lists don't provide direct access to individual nodes

Applications of circular linked lists:

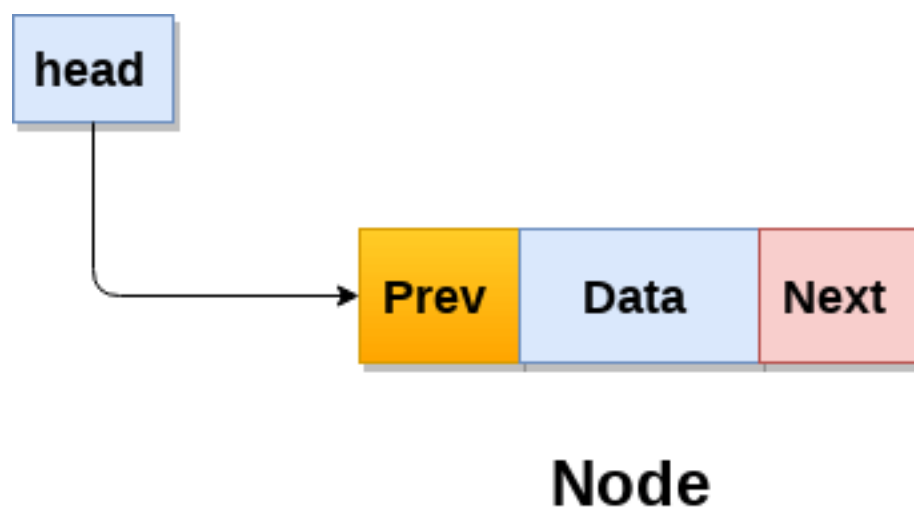
- Multiplayer games use this to give each player a chance to play.
- A circular linked list can be used to organize multiple running applications on an operating system. These applications are iterated over by the OS.
- Circular linked lists can be used in resource allocation problems.
- Circular linked lists are commonly used to implement circular buffers,
- Circular linked lists can be used in simulation and gaming.

Why circular linked list?

- A node always points to another node, so NULL assignment is not necessary.
- Any node can be set as the starting point.
- Nodes are traversed quickly from the first to the last.

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), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

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

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

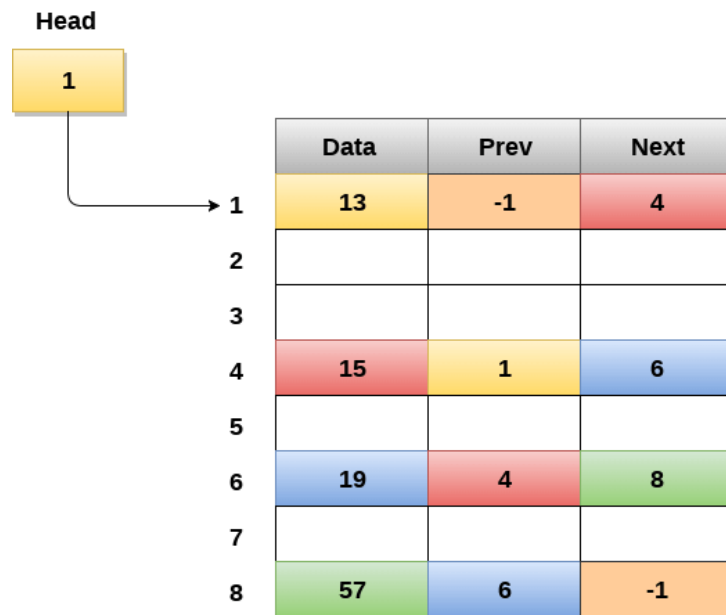
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

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

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	<u>Insertion at beginning</u>	Adding the node into the linked list at beginning.
2	<u>Insertion at end</u>	Adding the node into the linked list to the end.
3	<u>Insertion after specified node</u>	Adding the node into the linked list after the specified node.
4	<u>Deletion at beginning</u>	Removing the node from beginning of the list
5	<u>Deletion at the end</u>	Removing the node from end of the list.
6	<u>Deletion of the node having given data</u>	Removing the node which is present just after the node containing the given data.
7	<u>Searching</u>	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	<u>Traversing</u>	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Sparse Matrix

In this article, we will discuss the sparse matrix.

Let's first see a brief description of the matrix.

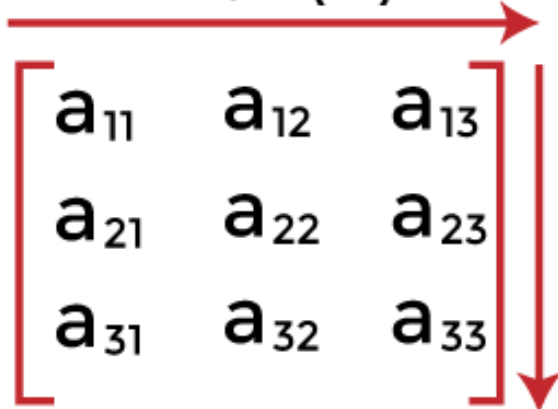
What is a matrix?

A matrix can be defined as a two-dimensional array having 'm' rows and 'n' columns. A matrix with m rows and n columns is called $m \times n$ matrix. It is a set of numbers that are arranged in the horizontal or vertical lines of entries.

For example -

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Row (m) Columns (n)

A diagram showing a 3x3 matrix A enclosed in large square brackets. Above the matrix, a horizontal red arrow points to the right, labeled "Row (m)". To the right of the matrix, a vertical red arrow points downwards, labeled "Columns (n)". The elements of the matrix are labeled a₁₁, a₁₂, a₁₃ in the first row; a₂₁, a₂₂, a₂₃ in the second row; and a₃₁, a₃₂, a₃₃ in the third row.

What is a sparse matrix?

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Now, the question arises: we can also use the simple matrix to store the elements, then why is the sparse matrix required?

Why is a sparse matrix required if we can use the simple matrix to store elements?

There are the following benefits of using the sparse matrix -

Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representation of sparse matrix

Now, let's see the representation of the sparse matrix. The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows -

- Array representation
- Linked list representation

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as -

ROW	COL	VALUE
-----	-----	-------

- **Row** - It is the index of a row where a non-zero element is located in the matrix.
- **Column** - It is the index of the column where a non-zero element is located in the matrix.
- **Value** - It is the value of the non-zero element that is located at the index (row, column).

Example -

Let's understand the array representation of sparse matrix with the help of the example given below -

Consider the sparse matrix -

Sparse Matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	0	0	0
4	1	0	0	0

In the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements. The above matrix occupies $5 \times 4 = 20$ memory space. Increasing the size of matrix will increase the wastage space.

The tabular representation of the above matrix is given below -

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

In the above structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value. The first row of the table represents the triplets. The first triplet represents that the value 4 is stored at 0th row and 1st column. Similarly, the second triplet represents that the value 5 is stored at the 0th row and 3rd column. In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.

The size of the table depends upon the total number of non-zero elements in the given sparse matrix. Above table occupies $8 \times 3 = 24$ memory space which is more than the space occupied by the sparse matrix. So, what's the benefit of using the sparse matrix? Consider the case if the matrix is 8×8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be $8 \times 8 = 64$, whereas the space occupied by the table represented using triplets would be $8 \times 3 = 24$.

Node Structure

Row	Column	Value	Pointer to Next Node
-----	--------	-------	-------------------------