INTRODUCTION TO DATA STURUCTURE

Data structure is a way of storing and organizing data efficiently such that the required operations on them can be performed efficiently with respect to time as well as memory. Simply, Data Structure are used to reduce complexity (mostly the time complexity) of the code.

or

Data Structure:

A data structure is a collection of data type 'values' which are stored and organized in such a way that it allows for efficient access and modification.

Or

Data structure:

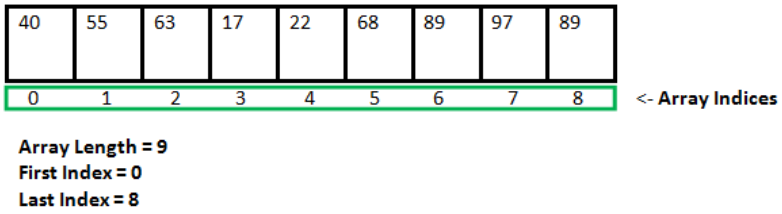
A data structure is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner. In computer science , a data structure is designed in such a way that it can work with various algorithms.

Data structures can be two types:

1. Static Data Structure
2. Dynamic Data Structure

What is a Static Data structure?

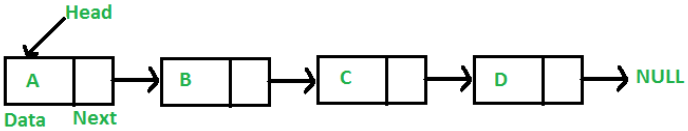
In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.



Example of Static Data Structures: Array

What is Dynamic Data Structure?

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.



Example of Dynamic Data Structures: Linked List

Static Data Structure vs Dynamic Data Structure

Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time. Thus dynamic data structure considered efficient with respect to memory complexity of the code. Static Data Structure provides easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

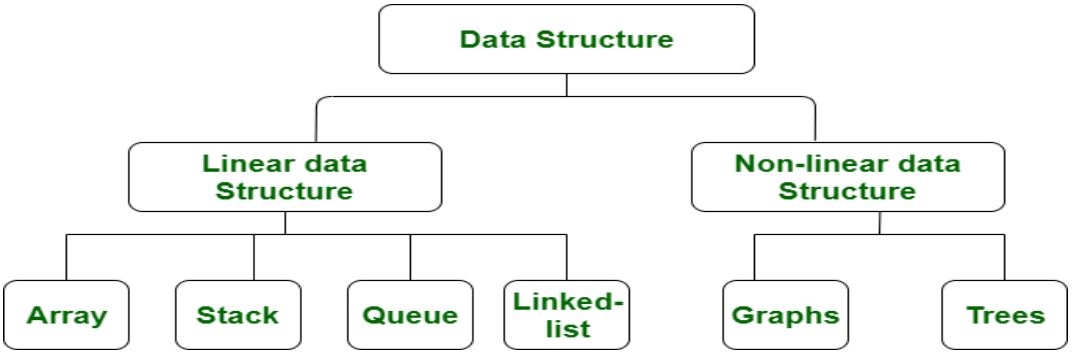
Use of Dynamic Data Structure in Competitive Programming

In competitive programming the constraints on memory limit is not much high and we cannot exceed the memory limit. Given higher value of the constraints we cannot allocate a static data structure of that size so Dynamic Data Structures can be useful.

Classification of Data Structure:

A data structure is classified into two categories:

- Linear data structure
- Non-linear data structure



Linear data structure

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

- **Array:** An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size 10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.
- **Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.
- **Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear** The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue.
- **Linked list:** It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.
-

Non-linear data structure

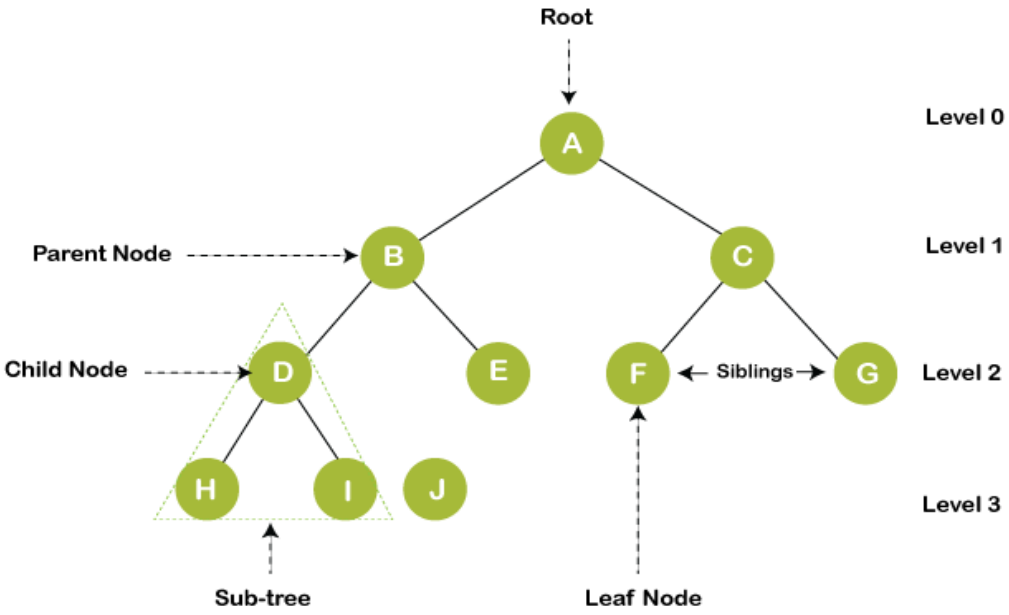
A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and

the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and **Graphs** are the types of non-linear data structure.

- **Tree**

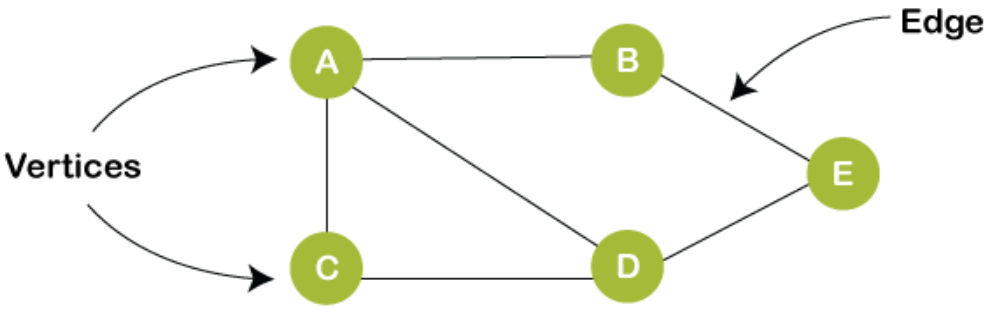
It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:



For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- **Graph**

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



Difference between Linear and Non-linear Data Structures:

NO	inear Data Structure	on-linear Data Structure
	a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	a non-linear data structure, data elements are attached in hierarchically manner.
	linear data structure, single level is involved.	hereas in non-linear data structure, multiple levels are involved.
	s implementation is easy in comparison to non-linear data structure.	hile its implementation is complex in comparison to linear data structure.
	linear data structure, data elements can be traversed in a single run only.	hile in non-linear data structure, data elements can't be traversed in a single run only.
	a linear data structure, memory is not utilized in an efficient way.	hile in a non-linear data structure, memory is utilized in an efficient way.
	s examples are: array, stack, queue, linked list, etc.	hile its examples are: trees and graphs.
	pplications of linear data structures are mainly in application software development.	pplications of non-linear data structures are in Artificial Intelligence and image processing.

Abstract data types:

An abstract data type (ADT) is an abstraction of a data structure .

An ADT specifies:

- Data stored
- Operations on the data
- Error conditions associated with operations

E.g : Lists, Stack, Queue, BinaryTree etc

LINKED LISTS

- **Disadvantages of Arrays**
  1. Static memory allocation
  2. Wastage of memory
  3. Insufficiency of memory
  4. Remove /Inserting element from/in middle of a collection is not easy.
- **Advantages of pointers**
  1. Dynamic memory allocation
  2. Effective usage of memory

3. Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting.

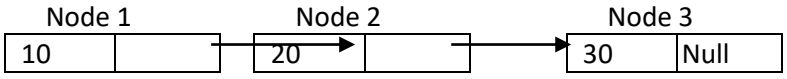
* Linked lists use dynamic memory allocation i.e. they grow and shrink accordingly

TYPES OF LINKED LIST:

1. Single / Singly linked lists
2. Double / Doubly linked lists
3. Circular singly linked list
4. Circular doubly linked list

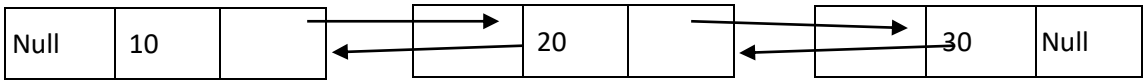
1)Singly linked list : **Singly linked list is a collection of nodes** with each node having a data part and a pointer pointing to the next node.

Representation



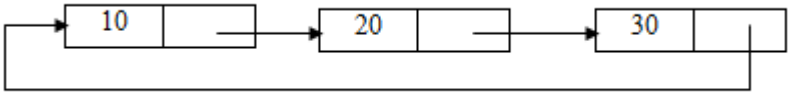
- Node consists of 2 parts
  - → Data → Link
- The link field always point to the next node in the list
- The link field of the last node is null

2) Double linked list: **Doubly linked list is a collection of nodes** with each node having a data part and a left pointer pointing to the next node and right pointer pointing to the right node

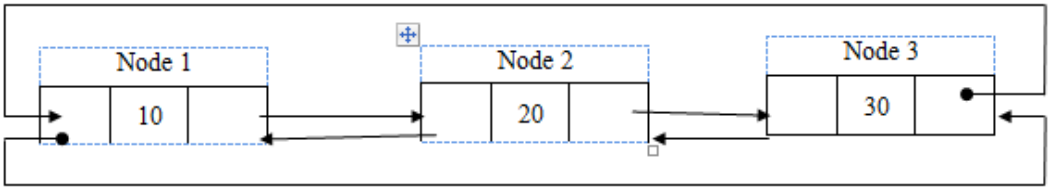


- Node consists of 3 parts namely
  - → Data part →left link →Right link
- The left link always points to the left node in the list and the right link always points to the right node in the list.
- The left link of the first node and the right link of the last node must be Null

3) Circular singly linked list : Last node point of first node

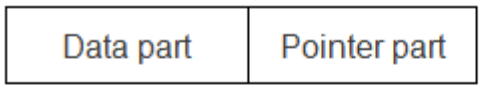


4. Circular double linked list: Last node next contain address of first node and first node prev contain address of last node

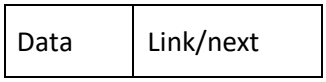


SINGLY LINKED LIST

Singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.



Representation of node



Node

- Node consists of 2 parts
  → Data → Link
- The link field always point to the next node in the list
- The link field of the last node is null

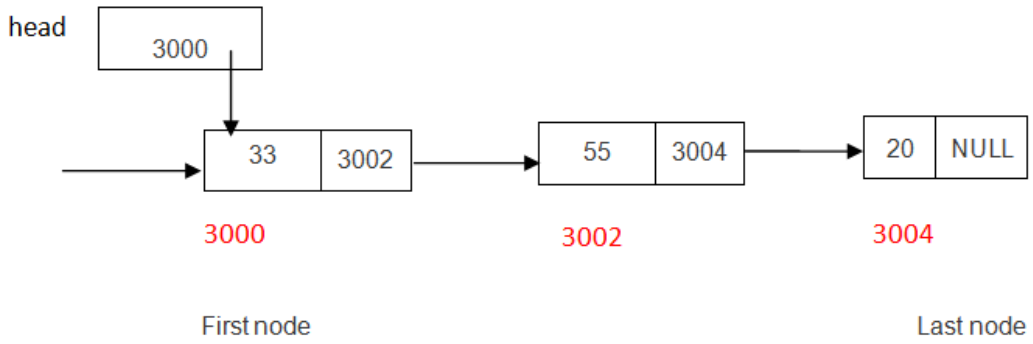
C++ NODE REPRESENTATION

```
class Node
{
    public:
            int data;
            Node *next;

            Node(int d)
    {
                    data=d;
            next=NULL;
    }
 };
```

Singly linked list representation

```
class Linkedlist
 {
     public:
    Node *hptr=NULL;
    // add insert operation/ functions
// add delete operation
// add any other operations
```

};



In a the above singly-linked list, every element contains some data and a link to the next element. The elements of a linked list are called the **nodes**.

A node has two fields i.e. **data** and **next**. The data field contains the data being stored in that specific node. It cannot just be a single variable. There may be many variables presenting the **data** section of a node.

The **next** field contains the address of the next node. So this is the place where the link between nodes is established.

Basic operations performed on SLL are

1. Insertion
2. Deletion

Insertion

Inserting a new node in the linked list is called insertion.

A new node is created and inserted in the linked list.

There are three cases considered while inserting a node:

1. Insertion at the start/beginning
2. Insertion at the end
3. Insertion at a particular position

Insertion at the Start/Beginning

1. New node should be connected to the first node, which means the head. This can be achieved by putting the address of the head in the next field of the new node.
2. New node should be considered as a head. It can be achieved by declaring head equals to a new node.

Algorithm

a. Get the new node

```
Node *nn = new Node(d); //nn means new node
```

b. Put the address of the head in the next field of the new node.

```
nn->next=hptr;
```

c. Make head point to new node

```
hptr = nn;
```

```
// C++ Code
void addatbeg(int x)
{
    Node *nn=new Node(x);
    if(nn==NULL)
```

```
    {
            cout<<"Unable to create node\n";
            return;
    }
     nn->next=hptr;
     hptr=nn;

}
```

Insertion at the End/create

The insertion of a node at the end of a linked list is like inserting the newly created node at the end of the linked list. Same code is used for creation of list.

Algorithm

Step 1: Get the new node

Node *temp= new Node(d);

Step 2: If the list is empty then head point to new node.

hptr = temp;

Step 3: If the list is not empty then traverse the list till last node found

a. Get first node address in temp
b. Move to last node till last node next equal to NULL
c. Make last node next hold address of new node

```
// C++ Code
 void create(int d)
{
    if(hptr==NULL)
    {
        Node *temp=new Node(d);
        hptr=temp;
    }
    else
    {
     Node *temp=hptr;
     while(temp->next!=NULL)
     {
            temp=temp->next;

          }

     Node *nptr=new Node(d);
     temp->next=nptr;
    }
}
```

Insertion at Particular Position

The insertion of a new node at a particular position is a bit difficult to understand. In this case, we don't disturb the head. Rather, a new node is inserted between two consecutive nodes. So, these two nodes should be accessible by our code. We call

one node as current and the other as previous, and the new node is placed between them. This process is shown in a diagram below:

Now the new node can be inserted between the previous and current node by just performing two steps:

1. Pass the address of the new node in the next field of the previous node.
2. Pass the address of the current node in the next field of the new node.

We will access these nodes by asking the user at what position he wants to insert the new node. Now, we will start a loop to reach those specific nodes. We initialized our current node by the head and move through the linked list. At the end, we would find two consecutive nodes.

Algorithm

Step 1: Get first node address in temp

temp=hptr

Step 2: Get the new node

Node *nn= new Node(d);

Step 3: Repeat step 4 and 5 till node at specified position found

Step 4: Move to next node

temp=temp->next;

Step 5: If the list is empty then create node using above create function

Step 6: Make new node point to temp next node and add new node at end of temp

nn->next=temp->next;

temp->next=nn;

```
// C++ code for insertion of node at Particular Position
void addatpos(int x,int p)
{
    Node *temp=hptr;
    Node *nn=new Node(x);
    for(int i=1;i<p;i++)
    {
            temp=temp->next;
            if(temp==NULL)
            {
                    create(x);
                    return;
            }
    }
    nn->next=temp->next;
    temp->next=nn;
}
```

Deletion:

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning

- Point head to the second node

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

2. Delete from end

- Traverse to second last element
- Change its next pointer to null

```
Node* temp = head;

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

{

temp = temp->next;

}

temp->next = NULL;
```

3. Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++)
{
if(temp->next!=NULL) {
```

temp = temp->next;
}
}

temp->next = temp->next->next;

Code for deletion operation

```
void  deletenode(int x)
{

    if(hptr==NULL)
    {
            cout<<"List is empty\n";
            return;
    }
    Node *temp=hptr,*prev;
    while(temp!=NULL)
    {
            if(temp->data== x)
            {
                    if(temp == hptr)
                    {
                            hptr = temp->next;
                    }
                    else
                    {
                            prev->next=temp->next;
                    }
                    cout<<"Node deleted\n";
                    delete temp;
                    return;
            }
            prev=temp;
            temp=temp->next;
    }
    cout<<"Element not found\n";
}
```

TRAVERSAL(DISPLAY METHOD) :

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When `temp` is `NULL`, we know that we have reached the end of the linked list so we get out of the while loop.

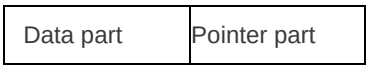
```
void display()
{
    Node *t=hptr;
    while(t!=NULL)

     {
      cout<<t->data<<"->";
      t=t->next;
    }
}
```

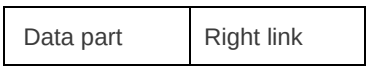
Applications of linked list in computer science –

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices.

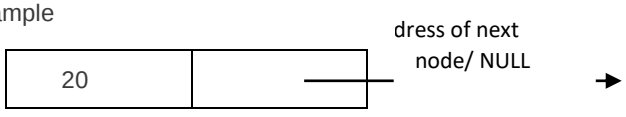
DOUBLY LINKED LIST DATA STRUCTURE IN C++

Singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.

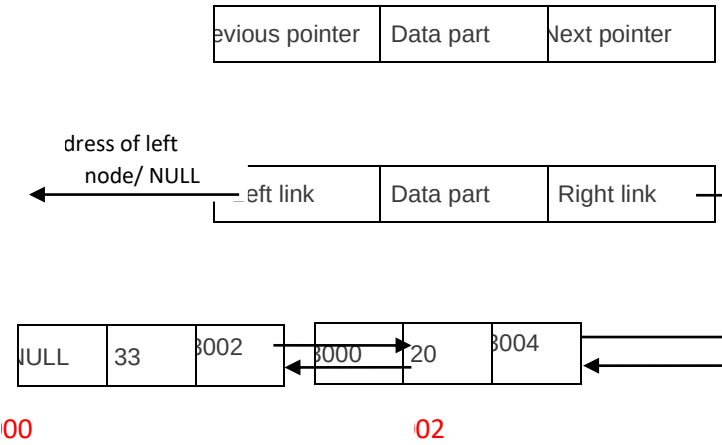




Example



A doubly linked list is also a collection of nodes. Each node here consists of a data part and two pointers. One pointer points to the previous node while the second pointer points to the next node. We can traverse the list is forward or backward directions.

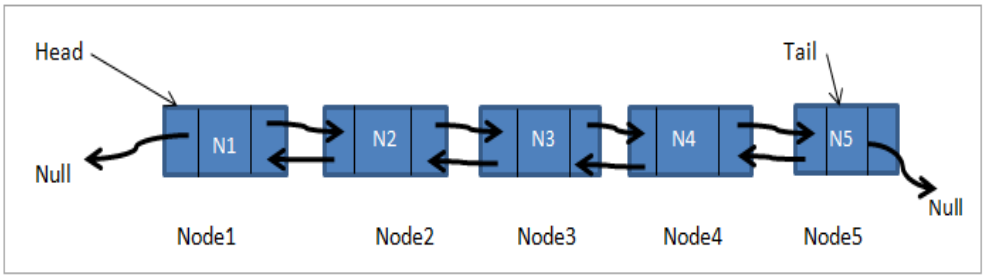


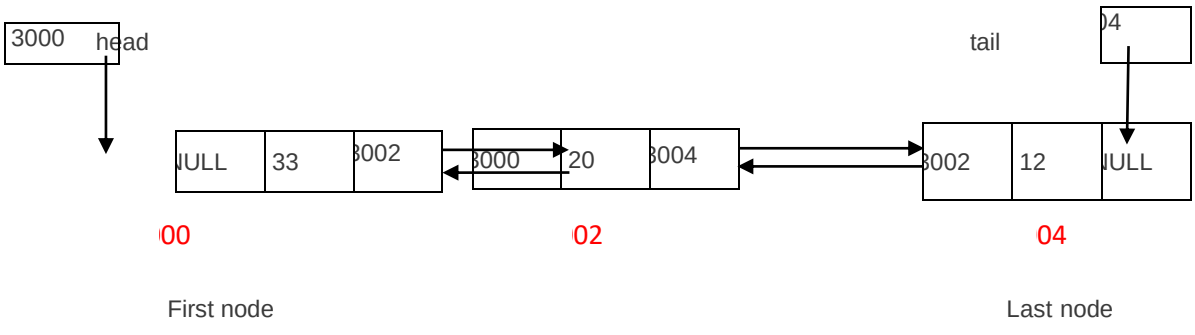
As in the singly linked list, the doubly linked list also has a head and a tail.

The previous pointer of the head is set to NULL as this is the first node.

The next pointer of the tail node is set to NULL as this is the last node.

Layout of the doubly linked list with head is shown below





Each node has two pointers, one pointing to the previous node and the other pointing to the next node. Only the first node (head) has its previous node set to null and the last node (tail) has its next pointer set to null.

The advantage of doubly linked list over the singly linked list is as the doubly linked list contains two pointers i.e. previous and next, we can traverse it into the directions forward and backward.

C++ node representation

```
Class Node
{
     Public:
              int data;
         Node *next;
              Node *prev;
              Node(int d)
              {
                    int data =d;
               next = NULL;
                     prev= NULL;
              }
              void display()
              {
                  cout<< data<<endl;
               }
  };
```

Doubly linked list representation

```
class DLL
{
    public:
    Node *hptr=NULL;    // head pointer
    Node *tptr=NULL;    // tail pointer
};
```

Here hptr and tptr act as pointer to first and last nodel.

Basic Operations

Insertion

This operation inserts a new node in the linked list in the following way

- **Insertion at front –** Inserts a new node as the first node.
- **Insertion at the end –** Inserts a new node at the end as the last node.
- **Insertion before a node/position –** Given a node, inserts a new node before this node/position.
- **Insertion after a node/position –** Given a node, inserts a new node after this node/position.

Deletion

Deletion operation deletes a node from a given position in the doubly linked list.

- **Deletion of the first node –** Deletes the first node in the list
- **Deletion of the last node –** Deletes the last node in the list.
- **Deletion of a node given the data –** Given the data, the operation matches the data with the node data in the linked list and deletes that node.

Traversal

Traversal is a technique of visiting each node in the linked list. In a doubly linked list, we have two types of traversals as we have two pointers with different directions in the doubly linked list.

- **Forward traversal –** Traversal is done using the next pointer which is in the forward direction.
- **Backward traversal –** Traversal is done using the previous pointer which is the backward direction.

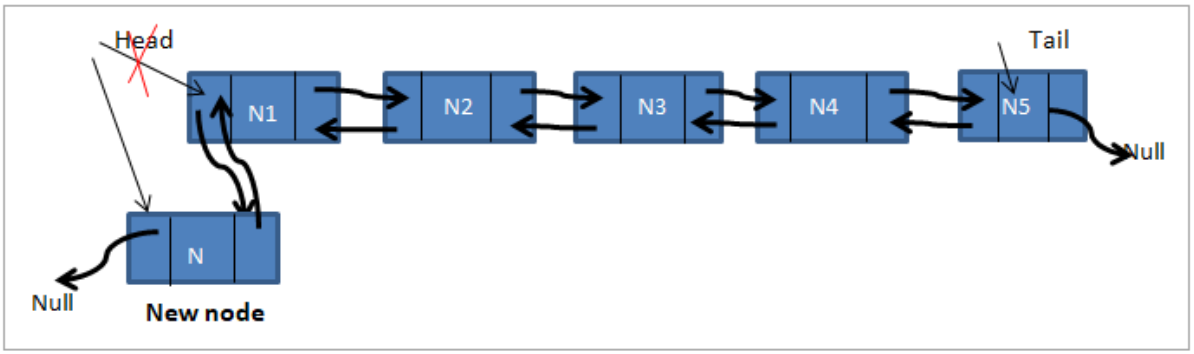
Search

Search operation in the doubly linked list is used to search for a particular node in the linked list. For this purpose, we need to traverse the list until a matching data is found.

Insertion

Insert a node at the front

Diagrammatic presentation and Explanation



Insertion of a new node at the front of the list is shown above. As seen, the previous (prev) of new node N is set to null. Head points to the new node. The next pointer of N now points to N1 and previous of N1 that was earlier pointing to Null now points to N.

Algorithm

Step 1: If the list is empty then head point to new node.

d. Get the new node

Node *nptr = new Node(d);

e. Make head point to new node

hptr=temp;

Step 2: If the list is not empty then make previous first node prev point to new node and new node

next point to previous first node whose address is in head. Now make tail point to new node

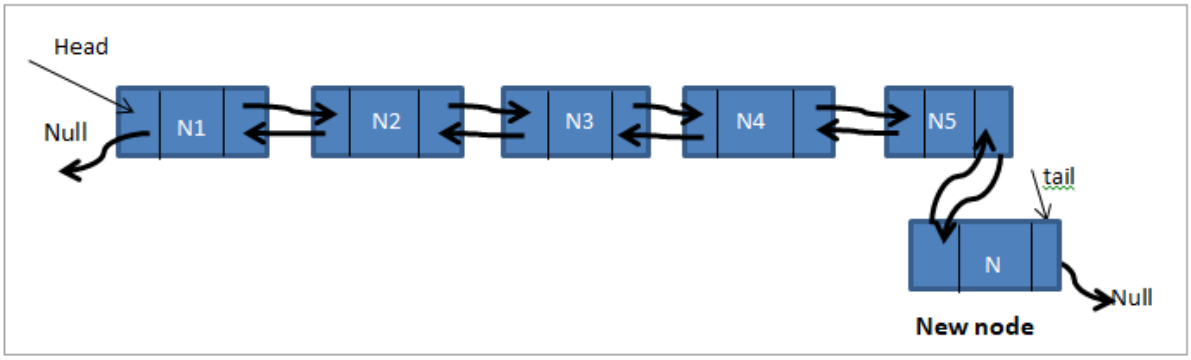
```
hptr->prev = nptr;      // previous first node point to new node
nptr->next = hptr;      // new node next point to previous first node whose address is in head
hptr = nptr;            // head point to new node
```

```
// add node at front / Beginning function
void addatfront(int d)
  {
```

```
        Node *nptr = new Node(d); // new allocate new node of type Node and return address in
     nptr
        if(hptr == NULL){
            hptr = tptr = nptr;
    }
    else
    {
           hptr->prev = nptr;       // previous first node point to new node
          nptr->next = hptr;       // new node next point to previous first node whose address is in
     head
          hptr = nptr;            // head point to new node
    }
  }
```

Insert node at the end

Diagrammatic presentation and Explanation



Inserting node at the end of the doubly linked list is achieved by pointing the next pointer of new node N to null. The previous pointer of N is pointed to N5. The 'Next' pointer of N5 is pointed to N.

Algorithm

Step 1: Get the new node

```
Node *nptr = new Node(d);
```

Step 2: If the list is empty then head and tail point to new node.

```
hptr = tptr = nptr;
```

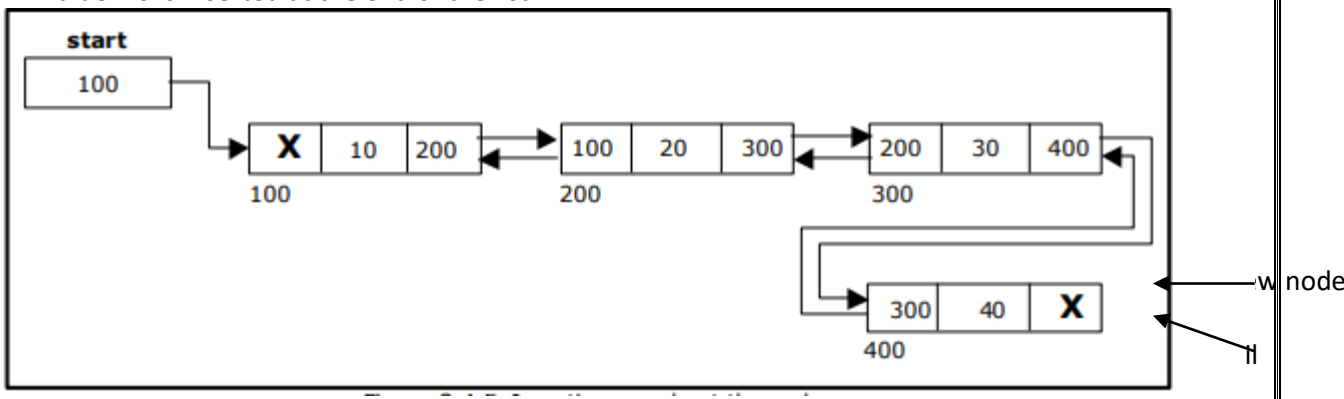
Step 3: If the list is not empty then make previous last node point to new node and new node prev
point to previous last node whose address is in tail. Now make tail point to new node

```
tptr->next = nptr;    // previous last node point to new node
nptr->prev = tptr;   // new node prev point to previous last node whose address is in tail
tptr = nptr;           // tail point to new node
```

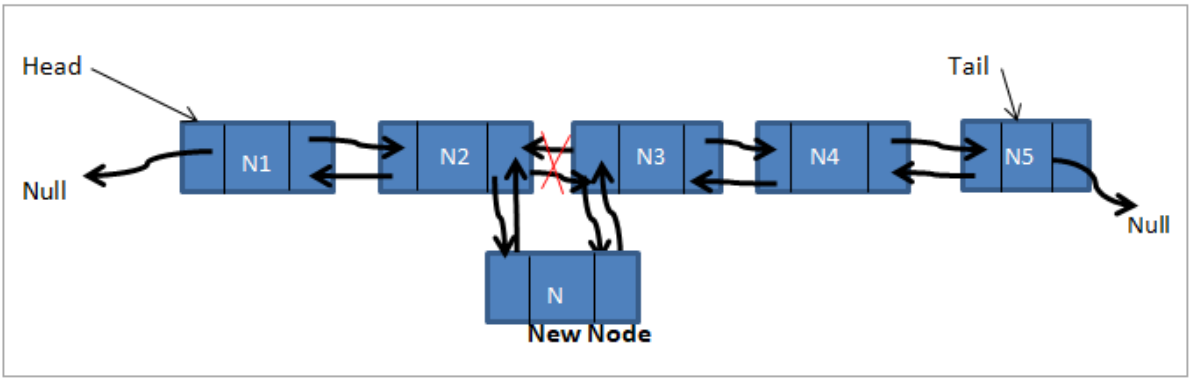
```
// add at end function or use same code for creating DLL
void add_at_end(int d)
   {
       Node *nptr = new Node(d);
       if(hptr == NULL){
           hptr = tptr = nptr;
   }
   else
   {
```

```
        tptr->next = nptr;
        nptr->prev = tptr;
        tptr = nptr;
    }
  }
```

Inserting a node at the end: The following example shows that new node with address 400 and value 40 is inserted at the end of the list



Insert node before/after given node



As shown in the above diagram, when we have to add a node before or after a particular node, we change the previous and next pointers of the before and after nodes so as to appropriately point to the new node. Also, the new node pointers are appropriately pointed to the existing nodes.

Algorithm

Step 1: If the list is empty then print "List empty"

Step 2: If the list is not empty then make p hold address of first node

p= hptr

Step 3: repeat step 4 until end of list

Step 4: If node data equal to specified data // specified node becomes previous node

Step 4.1: Get the new node

Node *temp = new Node(d);

Step 4.2: Make new node temp point previous (p) and next node

temp -> next = p -> next;

temp ->prev = p;

Step 4.3: Make next node of previous(p) node to point to temp by its prev

p->next->prev =temp;

Step 4.4: Make previous node to point to temp by its next

p -> next = temp

Step 4.5: Go to step 6

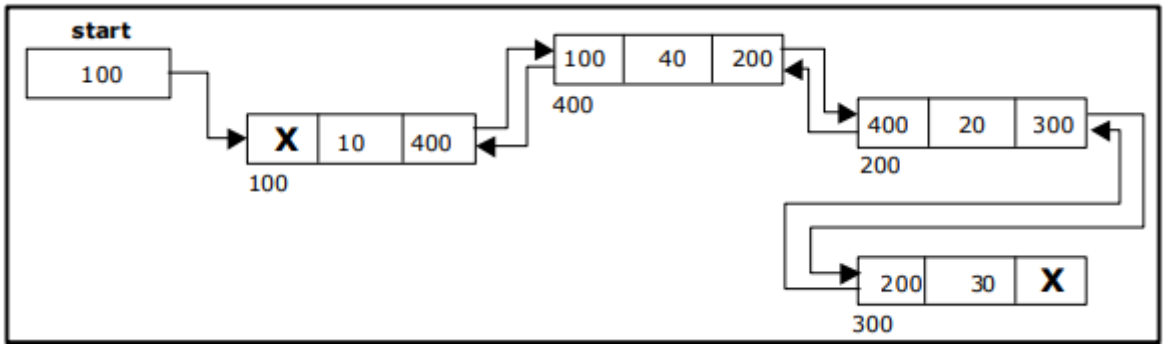
Step 4.5: make p point to next node

p = p -> next;

Step 4.6: Go to step 3
Step 5: Print specified node not found
Step 6: Stop

//Function to add node after specified node data

```
void addAfter(int ndata, int afteritem)
        {
            if (hptr == NULL)
              cout<<"List is empty";
            else
            {
              Node *temp, *p;
              p = hptr;
              do
              {
                if (p ->data == afteritem)
                 {
                   Node *temp = new Node(ndata);
                   temp -> next = p -> next;
                   temp ->prev = p;
                   p->next->prev =temp;

                   p -> next = temp;

                    /*if (p->next == NULL)
                        last = temp;*/
                        return;
                 }
                p = p -> next;
              }while(p!=NULL);

              cout <<"\n"<< afteritem << " not present in the list." << endl;
              return;

            }
        }
```



Here node with address 400 and data 40 has been added after node containing data value 10

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

Algorithm:

Step 1: If list is empty then display 'Empty List' message.

Step 2: If the list is not empty, follow the following steps

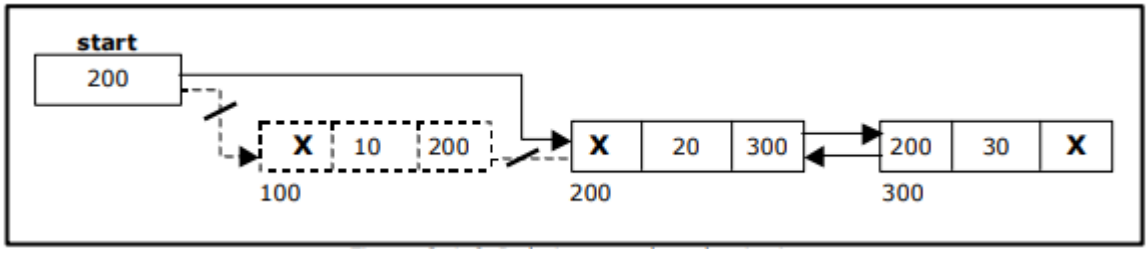
Step 2.1: If hptr->data == d // Check given data matches first node

Step 2.1.1: Make second node previous contain NULL

Step 2.1.1: Make head point to second node to make it first node in first node delete list

```
// Function to delete matched front node
void delete_at_front(int d)  // d is specified data to delete
{
   if(hptr == NULL)
   {
      cout << "List is empty";
   }
   else
     if(hptr->data == d)
    {
         Node *temp = hptr;
         temp->next->prev = NULL;
         hptr = temp->next;
     }
}
```

Figure shows deleting a node at the beginning of a double linked list.



Here first node with address 100 and data 10 is deleted by making head ((start) point to second node with address 200. Make second node prev contain NULL i.e. X here in diagram.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

Algorithm:

Step 1: If list is empty then display 'Empty List' message.

Step 2: If the list is not empty, follow the following steps

Step 2.1: If tptr->data == d // Check given data matches last node data

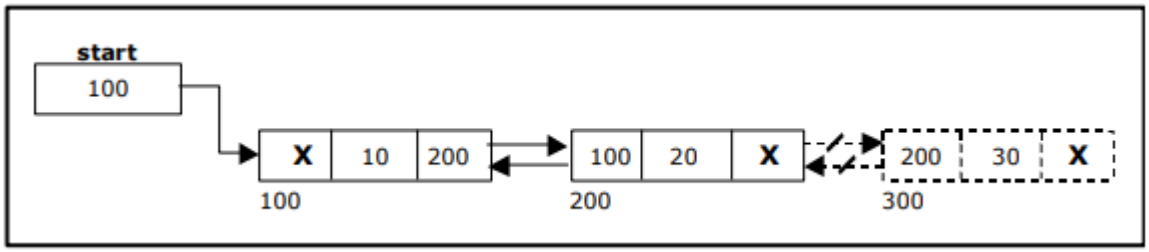
Step 2.1.1: Make last second node next contain NULL

Step 2.1.1: Make tail point to last second node, to make it last node in last node delete list

```
// Function to delete matched last node
void delete_at_end(int d)  // d is specified data to delete
```

```
{
      if(hptr == NULL)
      {
          cout << “List is empty”;
      }
     else
         if(tptr->data == d)
        {
             Node *temp = tptr;
             temp->prev->next = NULL;
             tptr = temp->prev;
      }
}
```

Figure shows deleting a node at the end of a double linked list.



Here last node with address 300 and data 30 is deleted by making tail point to last second node with address 200. Make second node next from last contain NULL i.e. X

Deleting a node at Intermediate position:

```
void delete_intermediadte_node(int d)  // d is specified data to delete
{
        Node* temp = hptr;
        while(temp!=NULL){
          if(temp->next!=NULL && temp->next->data == d){
             temp->next = temp->next->next;
             temp->next->prev = temp;
             return;
          }
          temp = temp->next;
        }
          cout<<"Element Not Found!"<<endl;

      }
```

Algorithm:

The following steps are followed, to delete a node from an intermediate position whose data matches the specified data /search data in the list. Note we go for intermediate node only after if list is not empty, or specified data not matches first or last node.

Step 1: Make temp hold address of first node

Step 2: Repeat step 3 till temp not equal to NULL // search till end of list

Step 3: if temp next node present and next node data matches specified data then

Step 3.1: make temp point to next to next node

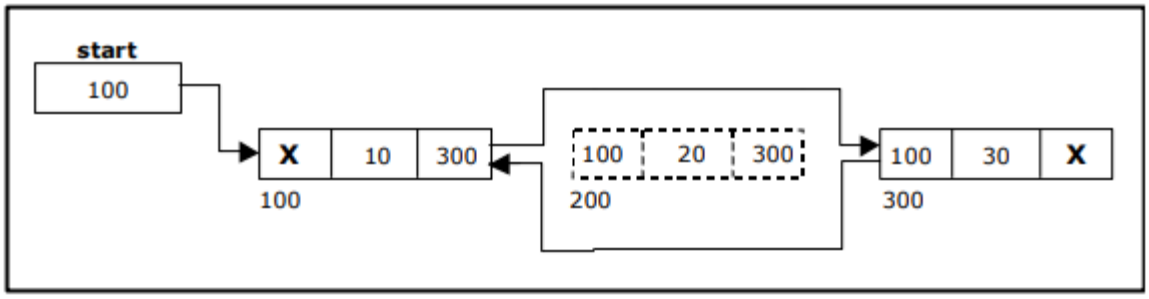
Step 3.2: make temp next node prev point to temp

Step 3.3: go to step 5

Step 4: print “Element Not Found!”

Step 5: Stop

Figure shows deleting a node that matches specified data at intermediate position other than beginning and end from a double linked list



Here node with address 200 and data 20 is deleted

Traversal and displaying a list (Left to Right) / forward traversal:

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function forward_display is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

Algorithm:

Step 1: If list is empty then display ‘Empty List’ message.

Step 2: If the list is not empty, follow the steps given below:

Step 2.1: Make temp hold address of first node

Node *temp = hptr;

Step 3: Repeat step 4 an 5 till temp not equal to NULL

while(temp!=NULL)

Step 4: display temp data

Step 5: Make temp point to next node

temp = temp -> next;

// Function forward display

```
void forward_display(){
      Node *temp = hptr;
      while(temp!=NULL){
         temp->display();
         temp = temp->next;
```

```
    }
  }
```

Traversal and displaying a list (Right to left) / reverse traversal:

The function reverse_display is used for traversing and displaying the information stored in the list from right to left.

The following steps are followed, to traverse a list from right to left

Algorithm:

Step 1: If list is empty then display 'Empty List' message.

Step 2: If the list is not empty, follow the steps given below:

Step 2.1: Make temp hold address of last node

```
Node *temp = tptr;
```

Step 3: Repeat step 4 and 5 till temp not equal to NULL

```
while(temp!=NULL)
```

Step 4: display temp data

Step 5: Make temp point to previous node

```
temp = temp -> prev;
```

```
// Function reverse display

void reverse_display(){
   Node *temp = tptr;
   while(temp!=NULL){
      temp->display();
      temp = temp->prev;
   }
}
```

Doubly Linked list Advantages

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting. 2. It is easy to reverse the linked list.

3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Disadvantages

1. It requires more space per space per node because one extra field is required for pointer to previous node.

2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.

1. What is Double Linked List?

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

2. Difference

● single linked list ● every node has a link to its next node in the sequence. ● we can traverse from one node to another node only in one direction ● we can not traverse back ● double linked list ● every node has a link to its previous node and next node. ● So, we can traverse forward by using the next field and can traverse backward by using the previous field

Applications/Uses of doubly linked list in real life

● Doubly linked list can be used in navigation systems where both front and back navigation is required.

● It is used by browsers to implement backward and forward navigation of visited web pages i.e. back and forward button.

● It is also used by various application to implement Undo and Redo functionality.

● It can also be used to represent deck of cards in games.

● It is also used to represent various states of a game.

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.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

to implement a circular singly linked list:

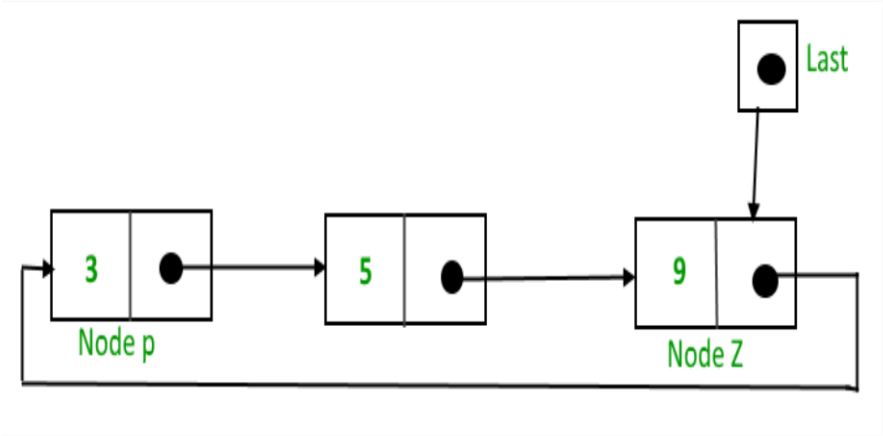
we take an external pointer that points to the last node of the list.

If we have a pointer last pointing to the last node, then last -> next will point to the first node.

Why Circular

- In a singly linked list, for accessing any node of the linked list,
- we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node.
- This problem can be solved by slightly altering the structure of a singly linked list.
- In a singly linked list, the next part (pointer to next node) is NULL. If we utilize this link to point to the first node, then we can reach the preceding nodes.

Example:



The pointer *last* points to node Z and last -> next points to node P.

Why have we taken a pointer that points to the last node instead of the first node?

For the insertion of a node at the beginning, we need to traverse the whole list.

Also, for insertion at the end, the whole list has to be traversed.

If instead of *start* pointer, we take a pointer to the last node,

then in both cases there won't be any need to traverse the whole list.

So insertion at the beginning or at the end takes constant time, irrespective of the length of the list.

Two operations are performed on linked lists:

1. Insertion
2. Deletion

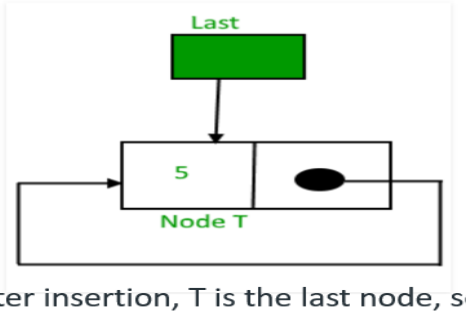
INSERTION

A node can be added in three ways:

- Insertion in an empty list
- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Creating the first node

After inserting node T,



After insertion, T is the last node, so the pointer *last* points to node T. And Node T is the first and the last node, so T points to itself.

Function to insert a node into an empty list,

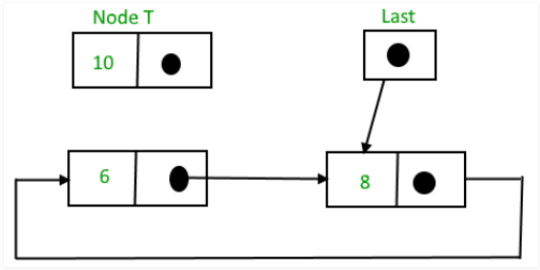
•C++

Insertion at the beginning of the list

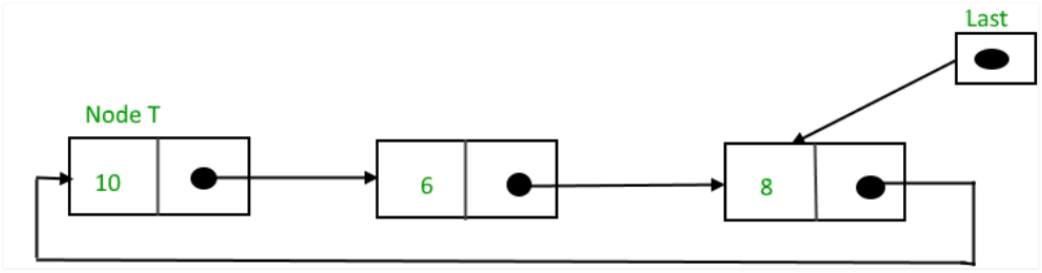
To insert a node at the beginning of the list, follow these steps:

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

Ex:



After insertion,



Code:

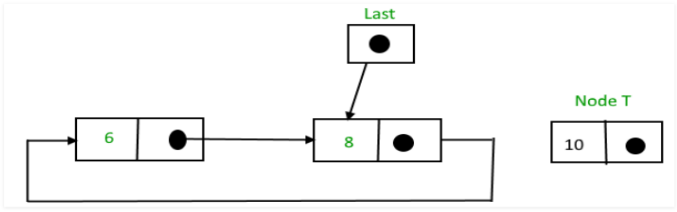
```
void InsertAtBegining(int d)
        {
            Node *temp = new Node(d);

            if ( last == NULL)        // for first node insertion
            {
```

```
        last =temp;         // last point to temp node as last node
        last -> next = last;  // last next point to temp node as first node
      }
     else                  // for another node insertion
     {
       temp -> next = last -> next;
       // temp as begining node point previous node first node to make it
          second node in CSLL
       last -> next = temp;   // last next point to temp node as first node
     }

   }
```

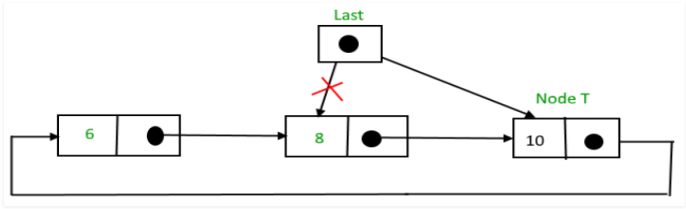
Insertion at the end of the list

- **To insert a node at the end of the list, follow these steps:**
  1. **Create a node, say T.**
  2. **Make T -> next = last -> next;**
  3. **last -> next = T.**
  4. **last = T.**

EX:



After insertion,



Code:

```
void InsertAtEnd(int d)
      {
        Node *temp = new Node(d);

        if ( last == NULL)       // for first node insertion
        {
          last =temp;         // last point to temp node as last node
          last -> next = last;  // last next point to temp node as first node
        }
       else                  // for another node insertion
```

```
        {
          temp -> next = last -> next;
          // temp as last node holds address of first node in CSLL
          last -> next = temp;
         // previous last node next point to temp node as now temp is Circular Linked
List node

          last = temp; // make last to temp as it is now last node
        }

      }
```

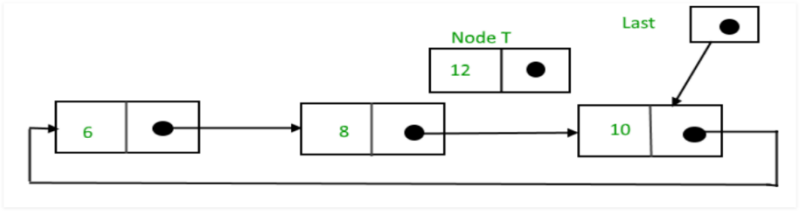
Insertion in between the nodes

To insert a node in between the two nodes, follow these steps:

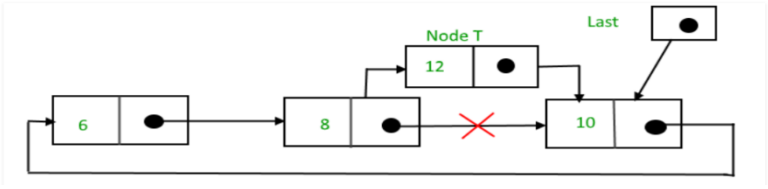
1. Create a node, say T.
2. Search for the node after which T needs to be inserted, say that node is P.
3. Make T -> next = P -> next;
4. P -> next = T.

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

Ex:



After searching and insertion,

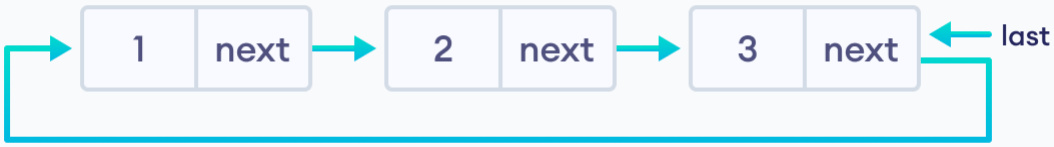



```
void addAfter(int ndata, int afteritem)
    {
     if (last == NULL)
     cout<<"List is empty";
    else
    {
      Node *temp, *p;
      p = last -> next;
      do
      {
       if (p ->data == afteritem)
        {
          Node *temp = new Node(ndata);
          temp -> next = p -> next;
```

```
            p -> next = temp;
              if (p == last)
                 last = temp;
                 return;  }
           p = p -> next;
         }while(p != last -> next);
          cout <<"\n"<< afteritem << " not present in the list." << endl;
         return;  }
       }
```

Deletion on a Circular Linked List

Suppose we have a double-linked list with elements 1, 2, and 3.



Initial circular linked list

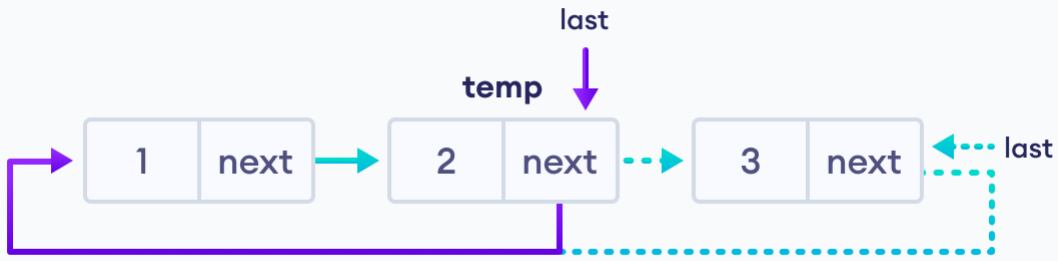
---

1. If the node to be deleted is the only node

- free the memory occupied by the node
- store NULL in `last`

2. If last node is to be deleted

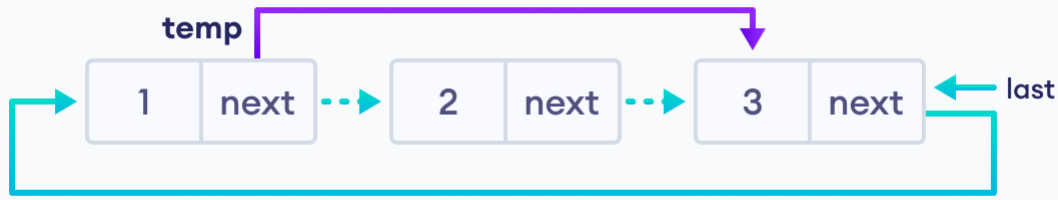
- find the node before the last node (let it be `temp`)
- store the address of the node next to the last node in `temp`
- free the memory of last
- make `temp` as the last node



Delete the last node

3. If any other nodes are to be deleted

- travel to the node to be deleted (here we are deleting node 2)
- let the node before node 2 be `temp`
- store the address of the node next to 2 in `temp`
- free the memory of 2



Delete a specific node

Code for Deletion operation:

```
void deletenode(int value)
    {
            Node *temp=last->next;
            if(temp->data==value)  //To delete first node
            {
                    last->next=temp->next;
            }
            else
            {
                    if(last->data==value) //To delete last node
                    {
                            while(temp->next!=last)
                            {
```

```
                        temp=temp->next;
                }
                temp->next=last->next;
                last=temp;
        }
        else //To delete a node between first and last
        {
                while(temp->next->data!=value)
                {

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

Circular Linked List Applications

- It is used in multiplayer games to give a chance to each player to play the game.
- Multiple running applications can be placed in a circular linked list on an operating system. The os keeps on iterating over these applications.

Recursion *is defined as a process which calls itself directly or indirectly and the corresponding function is called a recursive function.*

Properties of Recursion:

Recursion has some important properties. Some of which are mentioned below:

The primary property of recursion is the ability to solve a problem by breaking it down into smaller sub-problems, each of which can be solved in the same way.
A recursive function must have a base case or stopping criteria to avoid infinite recursion.
Recursion involves calling the same function within itself, which leads to a call stack.
Recursive functions may be less efficient than iterative solutions in terms of memory and performance.

Types of Recursion:

Direct recursion: When a function is called within itself directly it is called direct recursion.

This can be further categorised into four types:

Tail recursion,
Head recursion,
Tree recursion and
Nested recursion.
Indirect recursion: Indirect recursion occurs when a function calls another function that eventually calls the original function and it forms a cycle.

Applications of Recursion:

Recursion is used in many fields of computer science and mathematics, which includes:

Searching and sorting algorithms: Recursive algorithms are used to search and sort data structures like trees and graphs.
Mathematical calculations: Recursive algorithms are used to solve problems such as factorial, Fibonacci sequence, etc.
Compiler design: Recursion is used in the design of compilers to parse and analyze programming languages.
Graphics: many computer graphics algorithms, such as fractals and the Mandelbrot set, use recursion to generate complex patterns.
Artificial intelligence: recursive neural networks are used in natural language processing, computer vision, and other AI applications.

Advantages of Recursion:

Recursion can simplify complex problems by breaking them down into smaller, more manageable pieces.
Recursive code can be more readable and easier to understand than iterative code.
Recursion is essential for some algorithms and data structures.
Also with recursion, we can reduce the length of code and become more readable and understandable to the user/ programmer.

Disadvantages of Recursion:

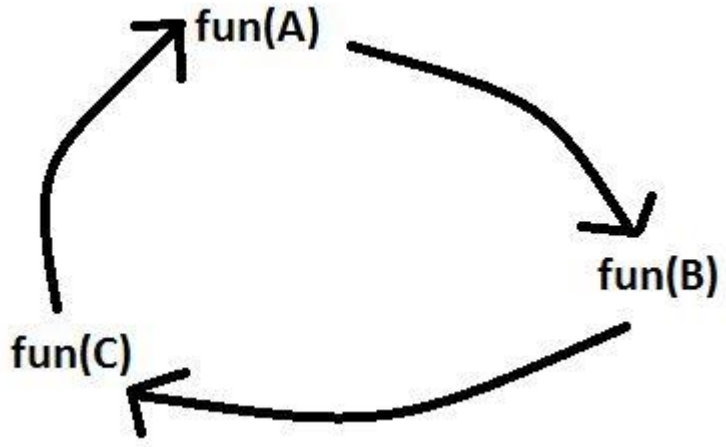
Recursion can be less efficient than iterative solutions in terms of memory and performance.
Recursive functions can be more challenging to debug and understand than iterative solutions.
Recursion can lead to stack overflow errors if the recursion depth is too high.

Direct Recursion:

T**ail Recursion**: If a recursive function calling itself and that recursive call is the last statement in the function then it's known as **Tail Recursion.** After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.
Example:

```
// Code Showing Tail Recursion
#include <iostream>
using namespace std;

// Recursion function
void fun(int n)
{
	if (n > 0) {
		cout << n << " ";

		// Last statement in the function
		fun(n - 1);
	}
}

// Driver Code
int main()
{
	int x = 3;
	fun(x);
	return 0;
}
```

Indirect Recursion: In this recursion, there may be more than one functions and they are calling one another in a circular manner.



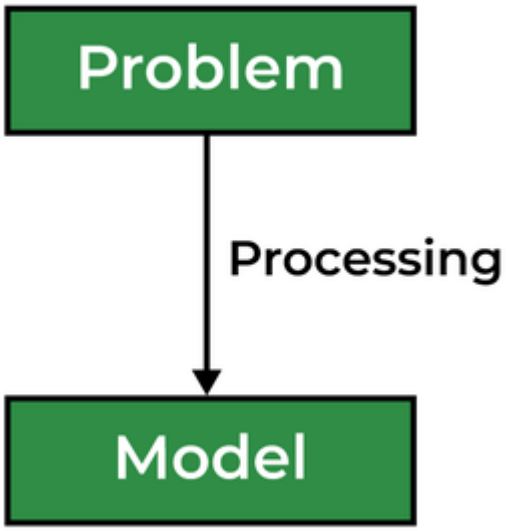
```
// C++ program to show Indirect Recursion
#include <iostream>
using namespace std;
void funB(int n);
void funA(int n)
{
	if (n > 0) {
		cout <<" "<< n;
		// fun(A) is calling fun(B)
		funB(n - 1);
	}
}

void funB(int n)
{
	if (n > 1) {
		cout <<" "<< n;

		// fun(B) is calling fun(A)
		funA(n / 2);
	}
}
int main()
{
	funA(20);
	return 0;
}
```

Data abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is

Types of Abstraction:
Data abstraction – This type only shows the required information about the data and hides the unnecessary data.
Control Abstraction – This type only shows the required information about the implementation and hides unnecessary information.



Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

Members declared as **public** in a class can be accessed from anywhere in the program.
Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

```
// C++ Program to Demonstrate the
// working of Abstraction
#include <iostream>
using namespace std;

class implementAbstraction {
private:
	int a, b;
public:
	// method to set values of
	// private members
	void set(int x, int y)
	{
		a = x;
		b = y;
	}
	void display()
```

```
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Advantages of Data Abstraction

Helps the user to avoid writing the low-level code
Avoids code duplication and increases reusability.
Can change the internal implementation of the class independently without affecting the user.
Helps to increase the security of an application or program as only important details are provided to the user.
It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

An array is a collection of data items, all of the same type, accessed using a common name.

A one-dimensional array is like a list; A two dimensional array is like a table; The C/C++ language places no limits on the number of dimensions in an array, though specific implementations may.

Some texts refer to one-dimensional arrays as ***vectors***, two-dimensional arrays as ***matrices***, and use the general term ***arrays*** when the number of dimensions is unspecified or unimportant.

Declaring Arrays

Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array.

Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.

Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.

In C++, dimensions must still be positive integers, but variables can be used, so long as the variable has a positive value at the time the array is declared.

C++ Example:

```
int numElements;
cout << "How big an array do you want? ";
cin >> numElements;
if( numElements <= 0 ) {
   cerr << "Error - Quitting\n";
   exit( 0 );
}
double data[ numElements ];
```

Initializing Arrays

Arrays may be initialized when they are declared, just as any other variables.

Place the initialization data in curly {} braces following the equals sign. Note the use of commas in the examples below.

An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.

If an array is to be completely initialized, the dimension of the array is not required. The compiler will automatically size the array to fit the initialized data. (Variation: Multidimensional arrays - see below.)

Examples:

```
int i =  5, intArray[ 6 ] = { 1, 2, 3, 4, 5, 6 }, k;
float sum  = 0.0, floatArray[ 100 ] = { 1.0, 5.0, 20.0 };
double  piFractions[ ] = { 3.141592654, 1.570796327, 0.785398163 };
```

Sample Program Using 1-D Arrays

The first sample program uses loops and arrays to calculate the first twenty Fibonacci numbers. Fibonacci numbers are used to determine the sample points used in certain optimization methods.

```
/* Program to calculate the first 20 Fibonacci numbers. */

#include <iostream>
using namespace std;

int main( void ) {

   int i, fibonacci[ 20 ];

   fibonacci[ 0 ] = 0;
   fibonacci[ 1 ] = 1;

   for( i = 2; i < 20; i++ )
      fibonacci[ i ] = fibonacci[ i - 2 ] + fibonacci[ i - 1 ];

   for( i = 0; i < 20; i++ )
      cout << "Fibonacci[ " << i << " ] = " <<  fibonacci[ i ] << endl;

} /* End of sample program to calculate Fibonacci numbers */
```

Multidimensional Arrays

Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.

One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.

Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.

It is better programming practice to enclose each row within a separate subset of curly {} braces, to make the program more readable. This is required if any row other than the last is to be partially initialized. When subsets of braces are used, the last item within braces is not followed by a comma, but the subsets are themselves separated by commas.

Multidimensional arrays may be partially initialized by not providing complete initialization data. Individual rows of a multidimensional array may be partially initialized, provided that subset braces are used.

Sample Program Using 2-D Arrays

```
/* Sample program Using 2-D Arrays */

#include <iostream>
using namespace std;

int main( void ) {

   /* Program to add two multidimensional arrays */
   /* Written May 1995 by George P. Burdell */

   int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
   int b[ 2 ][ 3 ] = { { 1, 2, 3 }, {  3,  2,  1 } };
   int sum[ 2 ][ 3 ], row, column;

   /* First the addition */

   for( row = 0; row < 2; row++ )
      for( column = 0; column < 3; column++ )
         sum[ row ][ column ] =
            a[ row ][ column ] + b[ row ][ column ];

   /* Then print the results */

   cout <<  "The sum is: \n\n" ;

   for( row = 0; row < 2; row++ ) {
      for( column = 0; column < 3; column++ )
         cout << "\t" <<  sum[ row ][ column ];
      cout << endl;   /* at end of each row */
   }

   return 0;
}
```

Calculating the address of any element In the 1-D array:

A 1-dimensional array (or single-dimension array) is a type of linear array. Accessing its elements involves a single subscript that can either represent a row or column index.

Example:

1 Dimensional Array
with 6 Elements

Address of A[I] = B + W * (I – LB)
I = Subset of element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LB = Lower Limit/Lower Bound of subscript(If not specified assume zero).

Example: Given the base address of an array **A[1300 …… 1900]** as **1020** and the size of each element is 2 bytes in the memory, find the address of **A[1700].**
Solution:
Given:
Base address B = 1020
Lower Limit/Lower Bound of subscript LB = 1300
Storage size of one element store in any array W = 2 Byte
Subset of element whose address to be found I = 1700
Formula used:
*Address of A[I] = B + W * (I – LB)*
Solution:
*Address of A[1700] = 1020 + 2 * (1700 – 1300)*
*= 1020 + 2 * (400)*
= 1020 + 800
Address of A[1700] = 1820

Calculate the address of any element in the 2-D array:
The 2-dimensional array can be defined as an array of arrays. The 2-Dimensional arrays are organized as matrices which can be represented as the collection of rows and columns as array[M][N] where M is the number of rows and N is the number of columns.

Columns

Rows

2D Array	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

To find the address of any element in a **2-Dimensional** array there are the following two ways-
Row Major Order
Column Major Order

Row Major Order:

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.
To find the address of the element using row-major order uses the following formula:
Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))
I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

Example: Given an array, **arr[1……10][1……15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.
Solution:
Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1
= 15 – 1 + 1
= 15
Formula:
*Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))*
Solution:
*Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))*
*= 100 + 1 * ((7) * 15 + (5))*

*= 100 + 1 * (110)*
Address of A[I][J] = 210

Column Major Order:

If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in column-major order. To find the address of the element using column-major order use the following formula:

Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))
I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
M = Number of rows given in the matrix.
Example: Given an array **arr[1……10][1……15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.
Solution:
Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10
Formula: used
*Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*
*Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*
*= 100 + 1 * ((5) * 10 + (7))*
*= 100 + 1 * (57)*
Address of A[I][J] = 157

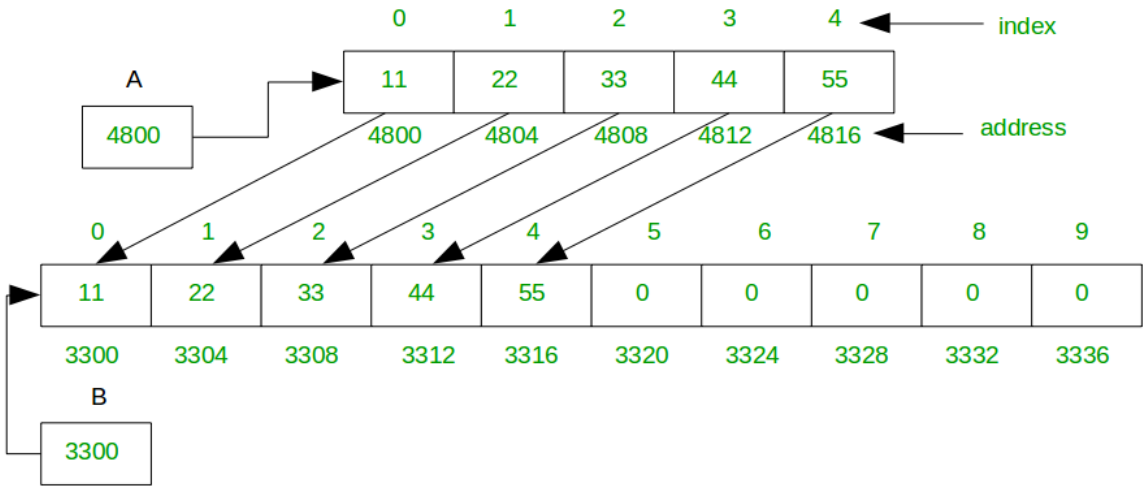
How do Dynamic arrays work?

A Dynamic array automatically grows when we try to make an insertion and there is no more space left for the new item. Usually the area doubles in size. A simple dynamic array can be constructed by allocating an array of fixed-size, typically larger than the number of elements immediately required. The elements of the dynamic array are stored contiguously at the start of the underlying array, and the remaining positions towards the end of the underlying array are reserved, or unused. Elements can be added at the end of a dynamic array in constant time by using the reserved space until this space is completely consumed. When all space is consumed, and an additional element is to be added, the underlying fixed-sized array needs to be increased in size. Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have overgrow before we can finally append our item.

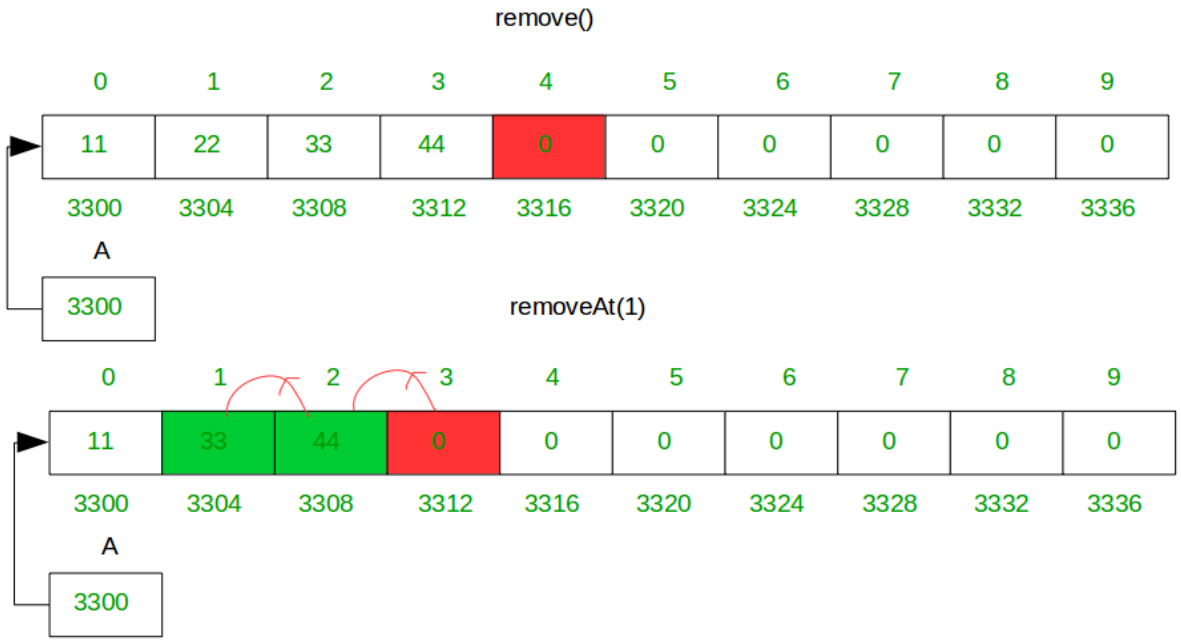
Approach: When we enter an element in array but array is full then you create a function, this function creates a new array double size or as you wish and copy all element from the previous array to a new array and return this new array. Also, we can reduce the size of the array. and add an element at a given position, remove the element at the end default and at the position also.

Key Features of Dynamic Array

Add Element: Add element at the end if the array size is not enough then extend the size of the array and add an element at the end of the original array as well as given index. Doing all that copying takes O(n) time, where n is the number of elements in our array. That's an expensive cost for an append. In a fixed-length array, appends only take O(1) time. But appends take O(n) time only when we insert into a full array. And that is pretty rare, especially if we double the size of the array every time we run out of space. So in most cases appending is still O(1) time, and sometimes it's O(n) time. In dynamic array you can create fixed-size array when required added some more element in array then use this approach:

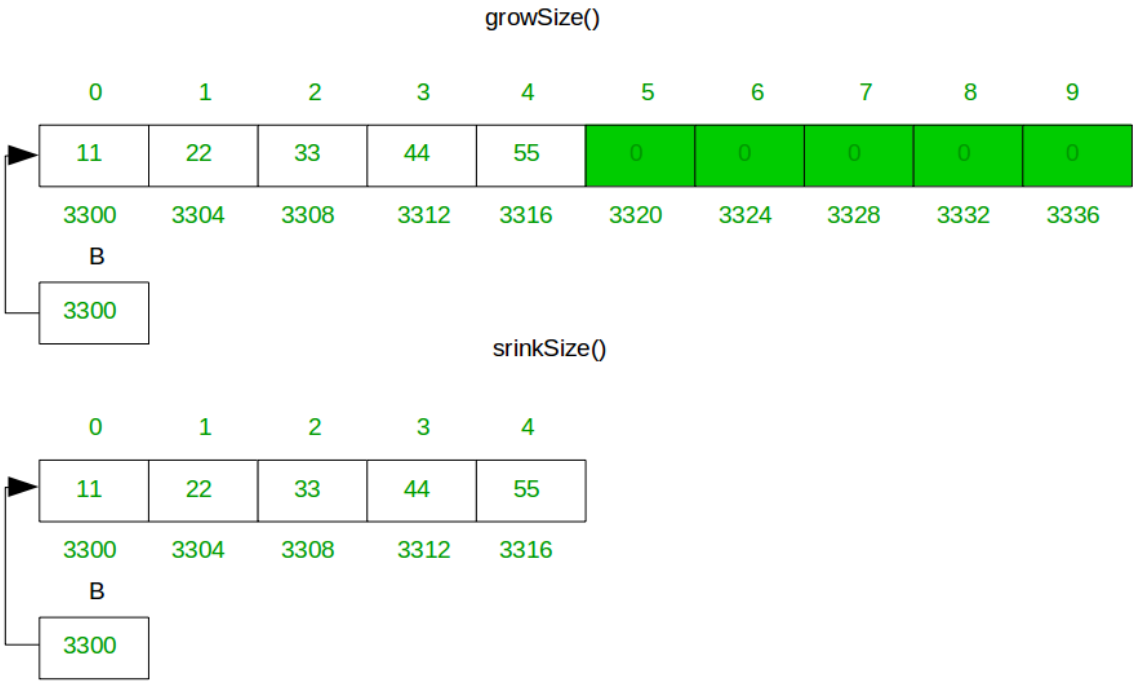


Delete Element: Delete an element from array, default remove() method delete an element from end, simply store zero at last index and you also delete element at specific index by calling removeAt(i) method where i is index. removeAt(i) method shift all right element in the left side from the given index.



Resize of Array Size: When the array has null/zero data (aside from an element added by you) at the right side of the array, meaning it has unused memory, the method shrinkSize() can free up the extra memory. When all space is consumed, and an additional element is to be added, then the underlying fixed-size array

needs to increase in size. Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have outgrown before we can finally append our item.




```
#include <iostream>
using namespace std;

class Polynomial {
  int degree;
  double* coeff;

public:
  Polynomial(int degree, double* coeff);
  Polynomial(const Polynomial& p);
  ~Polynomial();

  Polynomial operator+(const Polynomial& p);
  Polynomial operator-(const Polynomial& p);
  Polynomial operator*(const Polynomial& p);

  double evaluate(double x);
};

Polynomial::Polynomial(int degree, double* coeff) {
  this->degree = degree;
  this->coeff = new double[degree+1];
  for(int i=0; i<=degree; i++) {
  this->coeff[i] = coeff[i];
  }
}

Polynomial::Polynomial(const Polynomial& p) {
  degree = p.degree;
  coeff = new double[degree+1];
  for(int i=0; i<=degree; i++) {
```

```
  coeff[i] = p.coeff[i];
  }
}

Polynomial::~Polynomial() {
  delete[] coeff;
}

Polynomial Polynomial::operator+(const Polynomial& p) {
  // Implementation of addition operation
}

Polynomial Polynomial::operator-(const Polynomial& p) {
  // Implementation of subtraction operation
}

Polynomial Polynomial::operator*(const Polynomial& p) {
  // Implementation of multiplication operation
}

double Polynomial::evaluate(double x) {
  // Implementation of evaluation function
}
```

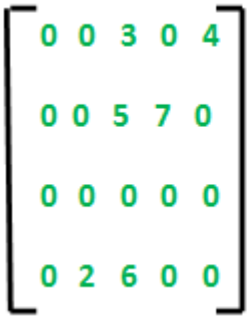
A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:
Array representation
Linked list representation
Method 1: Using Arrays:
2D array is used to represent a sparse matrix in which there are three rows named as
Row: Index of row, where non-zero element is located
Column: Index of column, where non-zero element is located
Value: Value of the non zero element located at index – (row,column)



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

```
// C++ program for Sparse Matrix Representation
// using Array
#include <iostream>
using namespace std;

int main()
{
	// Assume 4x5 sparse matrix
	int sparseMatrix[4][5] =
	{
		{0 , 0 , 3 , 0 , 4 },
		{0 , 0 , 5 , 7 , 0 },
		{0 , 0 , 0 , 0 , 0 },
		{0 , 2 , 6 , 0 , 0 }
	};

	int size = 0;
	for (int i = 0; i < 4; i++)
		for (int j = 0; j < 5; j++)
			if (sparseMatrix[i][j] != 0)
				size++;

	// number of columns in compactMatrix (size) must be
	// equal to number of non - zero elements in
	// sparseMatrix
	int compactMatrix[3][size];

	// Making of new matrix
	int k = 0;
	for (int i = 0; i < 4; i++)
		for (int j = 0; j < 5; j++)
			if (sparseMatrix[i][j] != 0)
			{
				compactMatrix[0][k] = i;
				compactMatrix[1][k] = j;
				compactMatrix[2][k] = sparseMatrix[i][j];
				k++;
			}

	for (int i=0; i<3; i++)
	{
		for (int j=0; j<size; j++)
			cout <<" "<< compactMatrix[i][j];

		cout <<"\n";
```

```
        }
        return 0;
}
```

Output

0 0 1 1 3 3

2 4 2 3 1 2

3 4 5 7 2 6

Method 2: Using Linked Lists

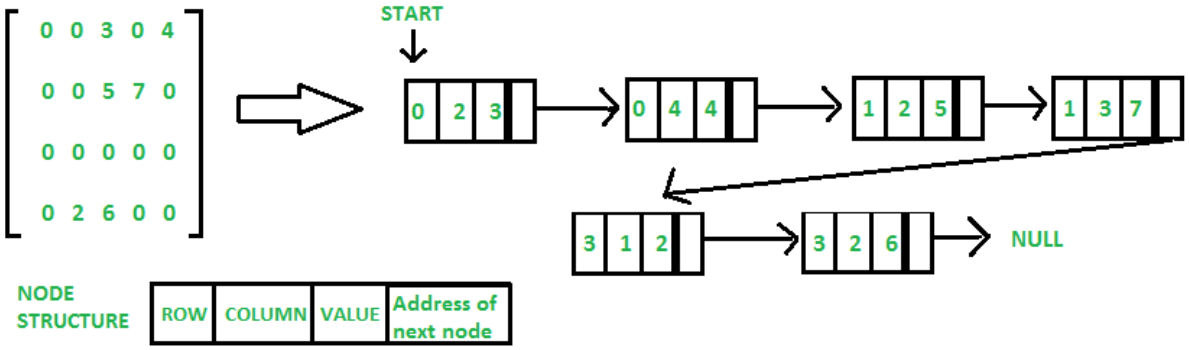
In linked list, each node has four fields. These four fields are defined as:

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row,column)

Next node: Address of the next node




```
// C++ program for sparse matrix representation Using Link list
#include<iostream>
using namespace std;

// Node class to represent link list
class Node
{
        public:
        int row;
        int col;
        int data;
        Node *next;
};

// Function to create new node
void create_new_node(Node **p, int row_index,
                                        int col_index, int x)
{
        Node *temp = *p;
        Node *r;

        // If link list is empty then
        // create first node and assign value.
        if (temp == NULL)
        {
                temp = new Node();
```

```
                temp->row = row_index;
                temp->col = col_index;
                temp->data = x;
                temp->next = NULL;
                *p = temp;
        }

        // If link list is already created
        // then append newly created node
        else
        {
                while (temp->next != NULL)
                        temp = temp->next;

                r = new Node();
                r->row = row_index;
                r->col = col_index;
                r->data = x;
                r->next = NULL;
                temp->next = r;
        }
}

// Function prints contents of linked list
// starting from start
void printList(Node *start)
{
        Node *ptr = start;
        cout << "row_position:";
        while (ptr != NULL)
        {
                cout << ptr->row << " ";
                ptr = ptr->next;
        }
        cout << endl;
        cout << "column_position:";

        ptr = start;
        while (ptr != NULL)
        {
                cout << ptr->col << " ";
                ptr = ptr->next;
        }
        cout << endl;
        cout << "Value:";
        ptr = start;

        while (ptr != NULL)
        {
                cout << ptr->data << " ";
                ptr = ptr->next;
        }
}

// Driver Code
```

```
int main()
{

	// 4x5 sparse matrix
	int sparseMatrix[4][5] = { { 0 , 0 , 3 , 0 , 4 },
	                           { 0 , 0 , 5 , 7 , 0 },
	                           { 0 , 0 , 0 , 0 , 0 },
	                           { 0 , 2 , 6 , 0 , 0 } };

	// Creating head/first node of list as NULL
	Node *first = NULL;
	for(int i = 0; i < 4; i++)
	{
		for(int j = 0; j < 5; j++)
		{

			// Pass only those values which
			// are non - zero
			if (sparseMatrix[i][j] != 0)
				create_new_node(&first, i, j,
				                sparseMatrix[i][j]);
		}
	}
	printList(first);

	return 0;
}
```

Output

row_position:0 0 1 1 3 3

column_position:2 4 2 3 1 2

Value:3 4 5 7 2 6

Arrays and Dynamic Arrays

Types of Arrays:

There are basically two types of arrays:

Static Array: In this type of array, memory is allocated at compile time having a fixed size of it. We cannot alter or update the size of this array.

Dynamic Array: In this type of array, memory is allocated at run time but not having a fixed size. Suppose, a user wants to declare any random size of an array, then we will not use a static array, instead of that a dynamic array is used in hand. It is used to specify the size of it during the run time of any program.

Example:

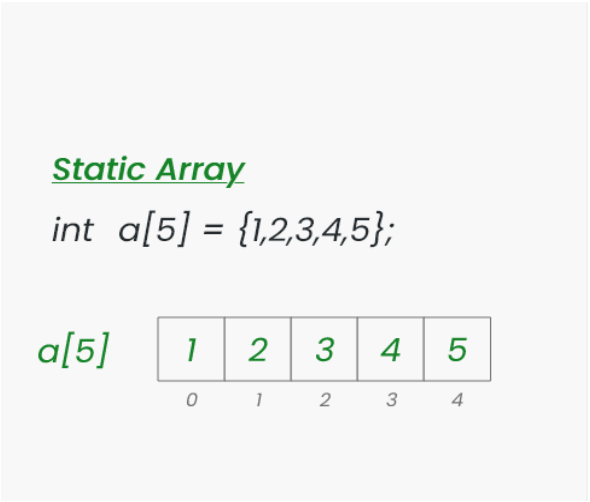
Let us take an example, **int a[5]** creates an array of size 5 which means that we can insert only 5 elements; we will not be able to add ***6th element*** because the size of the array is fixed above.

```
int a[5] = {1, 2, 3, 4, 5}; //Static Integer Array

int *a = new int[5]; //Dynamic Integer Array
```

Example of Static Array: int a[5] = {1, 2, 3, 4, 5};
A static integer array named a and initializes with the values 1, 2, 3, 4, and 5.
The size of this array is determined automatically based on the number of values provided in the initialization list.
Thus the array a is allocated on the stack, and its size cannot be changed once defined.

Example of Dynamic Array: int *a = new int[5];

In this code, we declare a pointer to an integer named a and it allocates memory in a dynamic fashion for an integer array of size 5.
The new keyword here is used for dynamic memory allocation, and int[5] specifies the size of this dynamic array.
The new operator is used to return the address of the dynamically allocated memory, which is already stored in the pointer a.
This array a is allocated on the Heap, and its size can be modified later if needed.



Key Difference between Static and Dynamic Arrays:

Static Array	Dynamic Array
1. The memory allocation occurs during compile time.	1. The memory allocation occurs during run time.
2. The array size is fixed and cannot be changed.	2. The array size is not fixed and can be changed.
3. The location is in Stack Memory Space.	3. The location is in Heap Memory Space.
4. The array elements are set to 0 or to empty strings.	4. The array elements can be destroyed during erase statement and the memory is then released.
5. This array can be Initialized but not erased.	5. This array cannot be read or written after destroying.

INTRODUCTION TO DATA STURUCTURE

Data structure is a way of storing and organizing data efficiently such that the required operations on them can be performed efficiently with respect to time as well as memory. Simply, Data Structure are used to reduce complexity (mostly the time complexity) of the code.

or

Data Structure:

A data structure is a collection of data type 'values' which are stored and organizedin such a way that it allows for efficient access and modification.

Or

Data structure:

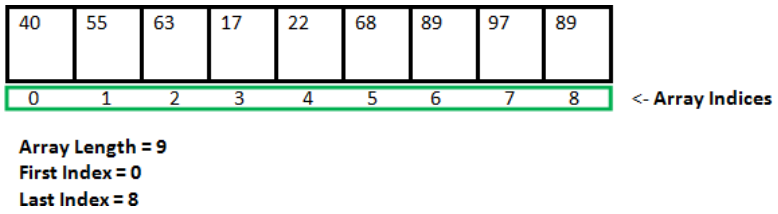
A data structure is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner. In computer science , a data structure is designed in such a way that it can work with various algorithms.

Data structures can be two types:

1. Static Data Structure
2. Dynamic Data Structure

What is a Static Data structure?

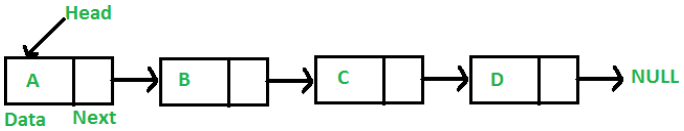
In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.



Example of Static Data Structures: Array

What is Dynamic Data Structure?

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.



Example of Dynamic Data Structures: Linked List

Static Data Structure vs Dynamic Data Structure

Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time. Thus dynamic data structure considered efficient with respect to memory complexity of the code. Static Data Structure provides easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

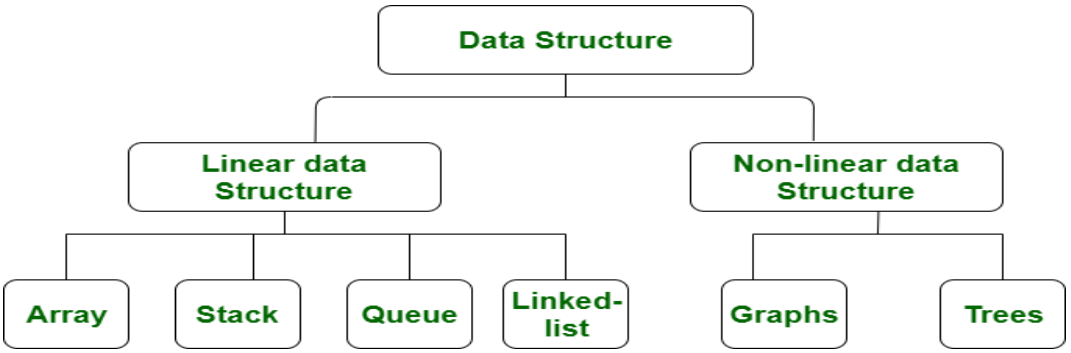
Use of Dynamic Data Structure in Competitive Programming

In competitive programming the constraints on memory limit is not much high and we cannot exceed the memory limit. Given higher value of the constraints we cannot allocate a static data structure of that size so Dynamic Data Structures can be useful.

Classification of Data Structure:

A data structure is classified into two categories:

- Linear data structure
- Non-linear data structure



Linear data structure

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

- **Array:** An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size 10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.
- **Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.
- **Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear** The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue.
- **Linked list:** It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.
-

Non-linear data structure

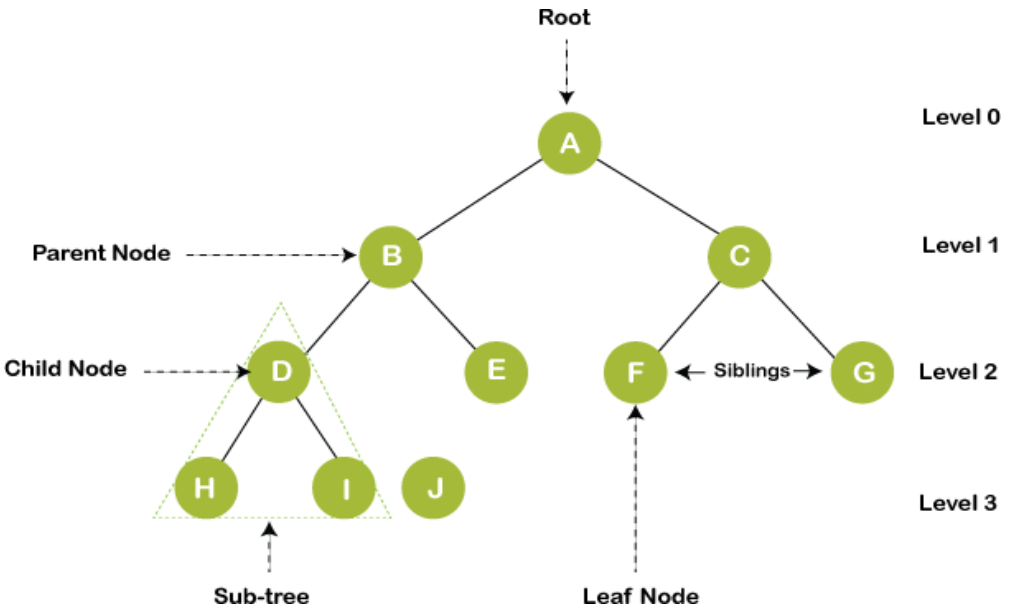
A non-linear data structure is also another type of data structure in which the dataelements are not arranged in a contiguous manner. As the arrangement isnonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and

the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and **Graphs** are the types of non-linear data structure.

- **Tree**

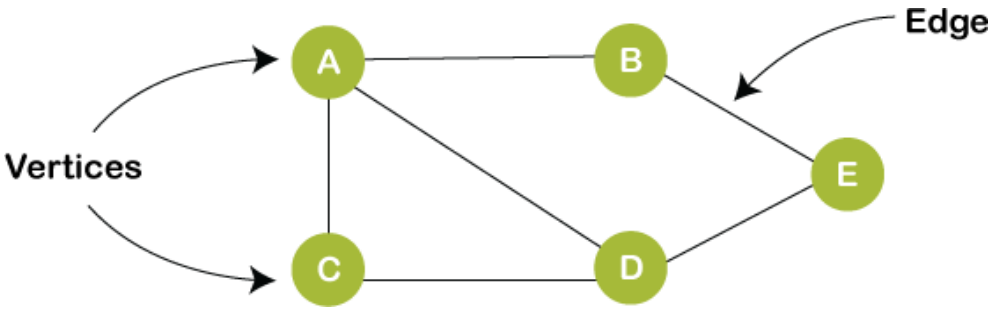
It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:



For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- **Graph**

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



Difference between Linear and Non-linear Data Structures:

-

NO	near Data Structure	on-linear Data Structure
	a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	a non-linear data structure, data elements are attached in hierarchically manner.
	linear data structure, single level is involved.	hereas in non-linear data structure, multiple levels are involved.
	implementation is easy in comparison to non-linear data structure.	hile its implementation is complex in comparison to linear data structure.
	linear data structure, data elements can be traversed in a single run only.	hile in non-linear data structure, data elements can't be traversed in a single run only.
	a linear data structure, memory is not utilized in an efficient way.	hile in a non-linear data structure, memory is utilized in an efficient way.
	examples are: array, stack, queue, linked list, etc.	hile its examples are: trees and graphs.
	pplications of linear data structures are mainly in application software development.	pplications of non-linear data structures are in Artificial Intelligence and image processing.

Abstract data types:

An abstract data type (ADT) is an abstraction of a data structure .

An ADT specifies:

- Data stored
- Operations on the data
- Error conditions associated with operations

E.g : Lists, Stack, Queue, BinaryTree etc

LINKED LISTS

- **Disadvantages of Arrays**
  1. Static memory allocation
  2. Wastage of memory
  3. Insufficiency of memory
  4. Remove /Inserting element from/in middle of a collection is not easy.
- **Advantages of pointers**
  1. Dynamic memory allocation
  2. Effective usage of memory

3. Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting.

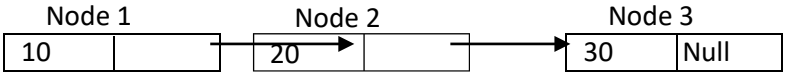
* Linked lists use dynamic memory allocation i.e. they grow and shrink accordingly

TYPES OF LINKED LIST:

1. Single / Singly linked lists
2. Double / Doubly linked lists
3. Circular singly linked list
4. Circular doubly linked list

1) Singly linked list : Singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.

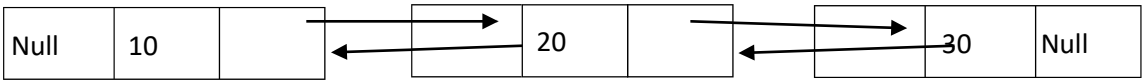
Representation



- Node consists of 2 parts
  - Data
  - Link

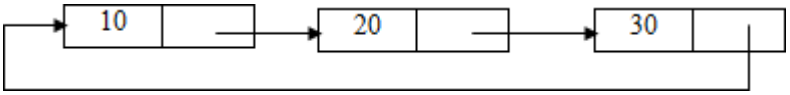
- The link field always point to the next node in the list
- The link field of the last node is null

2) Double linked list: Doubly linked list is a collection of nodes with each node having a data part and a left pointer pointing to the next node and right pointer pointing to the right node

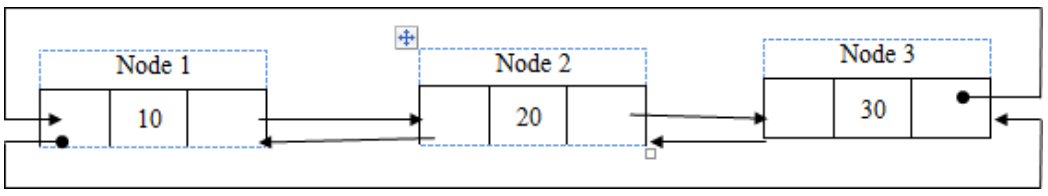


- Node consists of 3 parts namely
  - Data part
  - left link
  - Right link
- The left link always points to the left node in the list and the right link always points to the right node in the list.
- The left link of the first node and the right link of the last node must be Null

3) Circular singly linked list : Last node point of first node

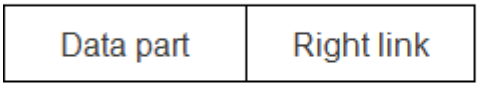


4. Circular double linked list: Last node next contain address of first node and first node prev contain address of last node

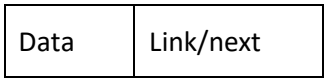


SINGLY LINKED LIST

Singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.



Representation of node



Node

- Node consists of 2 parts
  - Data
  - Link
- The link field always point to the next node in the list
- The link field of the last node is null

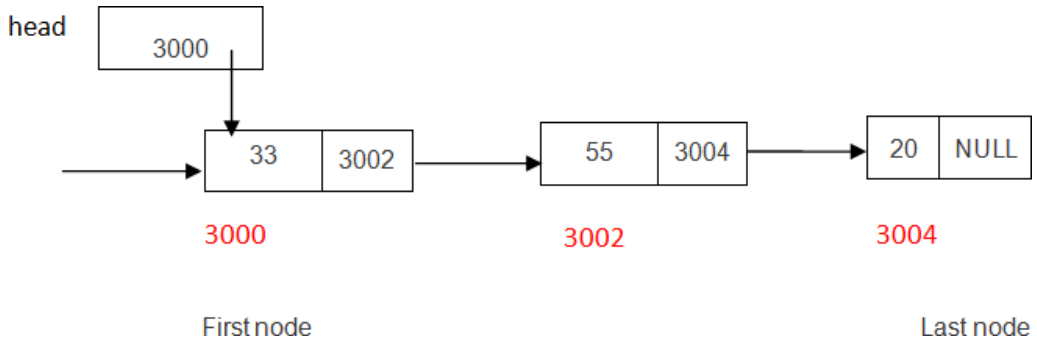
C++ NODE REPRESENTATION

```
class Node
{
    public:
            int data;
            Node *next;

            Node(int d)
    {
                    data=d;
            next=NULL;
    }
 };
```

Singly linked list representation

```
class Linkedlist
 {
     public:
    Node *hptr=NULL;
    // add insert operation/ functions
// add delete operation
// add any other operations
```

};



In a the above singly-linked list, every element contains some data and a link to the next element. The elements of a linked list are called the **nodes**.

A node has two fields i.e. **data** and **next**. The data field contains the data being stored in that specific node. It cannot just be a single variable. There may be many variables presenting the **data** section of a node.

The **next** field contains the address of the next node. So this is the place where the link between nodes is established.

Basic operations performed on SLL are

1. Insertion
2. Deletion

Insertion

Inserting a new node in the linked list is called insertion.

A new node is created and inserted in the linked list.

There are three cases considered while inserting a node:

1. Insertion at the start/beginning
2. Insertion at the end
3. Insertion at a particular position

Insertion at the Start/Beginning

1. New node should be connected to the first node, which means the head. This can be achieved by putting the address of the head in the next field of the new node.
2. New node should be considered as a head. It can be achieved by declaring head equals to a new node.

Algorithm

a. Get the new node

```
Node *nn = new Node(d); //nn means new node
```

b. Put the address of the head in the next field of the new node.

```
nn->next=hptr;
```

c. Make head point to new node

```
hptr = nn;
```

```
// C++ Code
void addatbeg(int x)
{
    Node *nn=new Node(x);
    if(nn==NULL)
```

```
    {
            cout<<"Unable to create node\n";
            return;
    }
     nn->next=hptr;
     hptr=nn;

}
```

Insertion at the End/create

The insertion of a node at the end of a linked list is like inserting the newly created node at the end of the linked list. Same code is used for creation of list.

Algorithm

Step 1: Get the new node

Node *temp= new Node(d);

Step 2: If the list is empty then head point to new node.

hptr = temp;

Step 3: If the list is not empty then traverse the list till last node found

a. Get first node address in temp
b. Move to last node till last node next equal to NULL
c. Make last node next hold address of new node

```
// C++ Code
 void create(int d)
{
    if(hptr==NULL)
    {
        Node *temp=new Node(d);
        hptr=temp;
    }
    else
    {

     Node *temp=hptr;
     while(temp->next!=NULL)
     {
            temp=temp->next;

          }

     Node *nptr=new Node(d);
     temp->next=nptr;
    }
}
```

Insertion at Particular Position

The insertion of a new node at a particular position is a bit difficult to understand. In this case, we don't disturb the head. Rather, a new node is inserted between two consecutive nodes. So, these two nodes should be accessible by our code. We call

one node as current and the other as previous, and the new node is placed between them. This process is shown in a diagram below:

Now the new node can be inserted between the previous and current node by just performing two steps:

1. Pass the address of the new node in the next field of the previous node.
2. Pass the address of the current node in the next field of the new node.

We will access these nodes by asking the user at what position he wants to insert the new node. Now, we will start a loop to reach those specific nodes. We initialized our current node by the head and move through the linked list. At the end, we would find two consecutive nodes.

Algorithm

Step 1: Get first node address in temp

temp=hptr

Step 2: Get the new node

Node *nn= new Node(d);

Step 3: Repeat step 4 and 5 till node at specified position found

Step 4: Move to next node

temp=temp->next;

Step 5: If the list is empty then create node using above create function

Step 6: Make new node point to temp next node and add new node at end of temp

nn->next=temp->next;

temp->next=nn;

```
// C++ code for insertion of node at Particular Position
void addatpos(int x,int p)
{
    Node   *temp=hptr;
    Node *nn=new Node(x);
    for(int i=1;i<p;i++)
    {
            temp=temp->next;
            if(temp==NULL)
            {
                    create(x);
                    return;
            }
    }
    nn->next=temp->next;
    temp->next=nn;
}
```

Deletion:

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning

- Point head to the second node

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

2. Delete from end

- Traverse to second last element
- Change its next pointer to null

```
Node* temp = head;

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

{

temp = temp->next;

}

temp->next = NULL;
```

3. Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++)
{
if(temp->next!=NULL) {
```

temp = temp->next;
}
}

temp->next = temp->next->next;

Code for deletion operation

```
void deletenode(int x)
{

    if(hptr==NULL)
    {
            cout<<"List is empty\n";
            return;
    }
    Node *temp=hptr,*prev;
    while(temp!=NULL)
    {
            if(temp->data== x)
            {
                    if(temp == hptr)
                    {
                            hptr = temp->next;
                    }
                    else
                    {
                            prev->next=temp->next;
                    }
                    cout<<"Node deleted\n";
                    delete temp;
                    return;
            }
            prev=temp;
            temp=temp->next;
    }
    cout<<"Element not found\n";
}
```

TRAVERSAL(DISPLAY METHOD) :

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When `temp` is `NULL`, we know that we have reached the end of the linked list so we get out of the while loop.

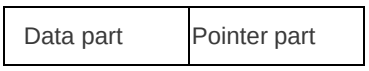
```
void display()
{
    Node *t=hptr;
    while(t!=NULL)

     {
      cout<<t->data<<"->";
      t=t->next;
    }
}
```

Applications of linked list in computer science –

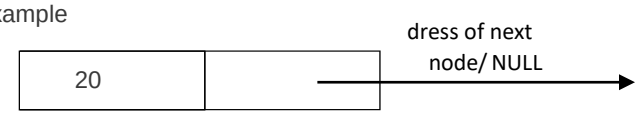
1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices.

DOUBLY LINKED LIST DATA STRUCTURE IN C++

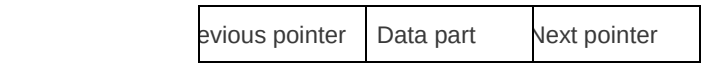
Singly linked list is a collection of nodes with each node having a data part and a pointer pointing to the next node.

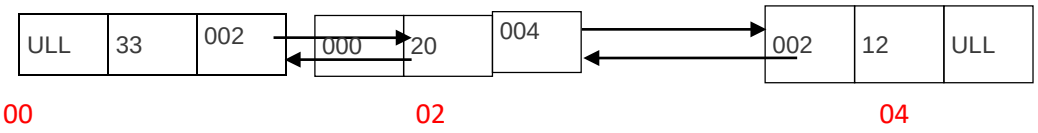


Example



A doubly linked list is also a collection of nodes. Each node here consists of a data part and two pointers. One pointer points to the previous node while the second pointer points to the next node. We can traverse the list is forward or backward directions.





00 02 04

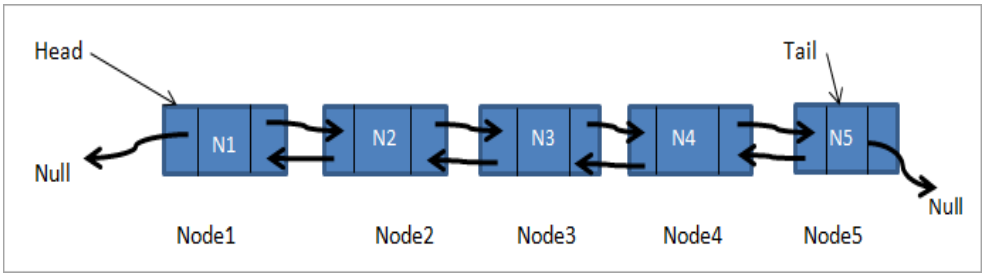
First node Last node

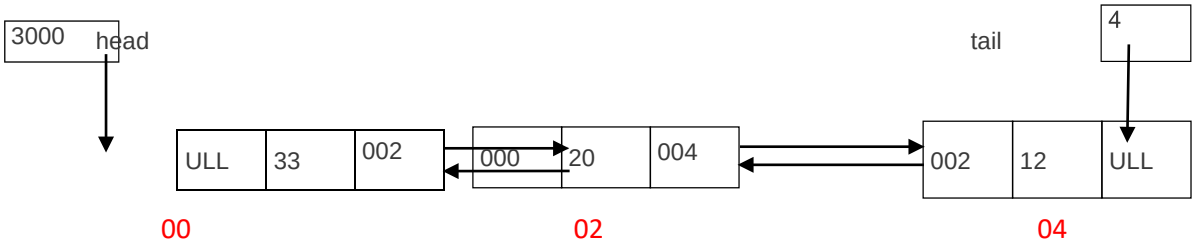
As in the singly linked list, the doubly linked list also has a head and a tail.

The previous pointer of the head is set to NULL as this is the first node.

The next pointer of the tail node is set to NULL as this is the last node.

Layout of the doubly linked list with head is shown below





00 02 04

First node Last node

Each node has two pointers, one pointing to the previous node and the other pointing to the next node. Only the first node (head) has its previous node set to null and the last node (tail) has its next pointer set to null.

The advantage of doubly linked list over the singly linked list is as the doubly linked list contains two pointers i.e. previous and next, we can traverse it into the directions forward and backward.

C++ node representation

```
Class Node
{
      Public:
                int data;
           Node *next;
                Node *prev;
                Node(int d)
                {
                      int data =d;
                 next = NULL;
                       prev= NULL;
                }
                void display()
                {
                    cout<< data<<endl;
                 }
  };
```

Doubly linked list representation

```
class DLL
{
    public:
    Node *hptr=NULL; // head pointer
    Node *tptr=NULL;    // tail pointer
};
```

Here hptr and tptr act as pointer to first and last nodel.

Basic Operations

Insertion

This operation inserts a new node in the linked list in the following way

- **Insertion at front –** Inserts a new node as the first node.
- **Insertion at the end –** Inserts a new node at the end as the last node.
- **Insertion before a node/position –** Given a node, inserts a new node before this node/position.
- **Insertion after a node/position –** Given a node, inserts a new node after this node/position.

Deletion

Deletion operation deletes a node from a given position in the doubly linked list.

- **Deletion of the first node –** Deletes the first node in the list
- **Deletion of the last node –** Deletes the last node in the list.
- **Deletion of a node given the data –** Given the data, the operation matches the data with the node data in the linked list and deletes that node.

Traversal

Traversal is a technique of visiting each node in the linked list. In a doubly linked list, we have two types of traversals as we have two pointers with different directions in the doubly linked list.

- **Forward traversal –** Traversal is done using the next pointer which is in the forward direction.
- **Backward traversal –** Traversal is done using the previous pointer which is the backward direction.

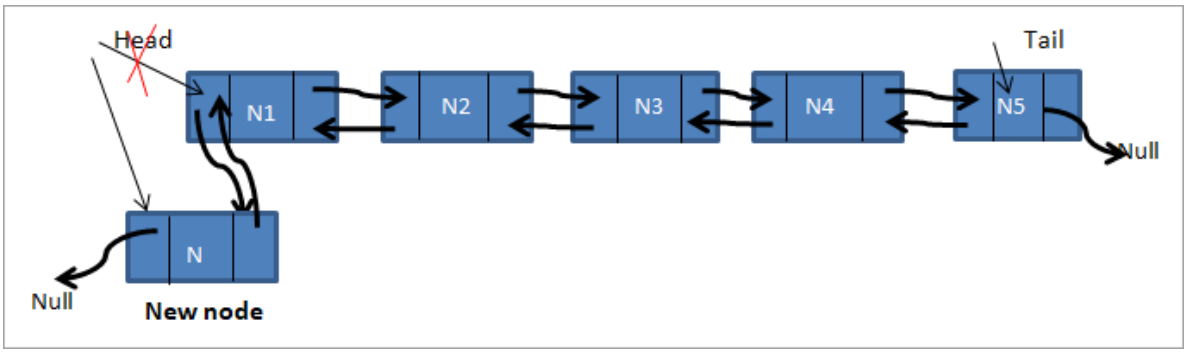
Search

Search operation in the doubly linked list is used to search for a particular node in the linked list. For this purpose, we need to traverse the list until a matching data is found.

Insertion

Insert a node at the front

Diagrammatic presentation and Explanation



Insertion of a new node at the front of the list is shown above. As seen, the previous (prev) of new node N is set to null. Head points to the new node. The next pointer of N now points to N1 and previous of N1 that was earlier pointing to Null now points to N.

Algorithm

Step 1: If the list is empty then head point to new node.

d. Get the new node

Node *nptr = new Node(d);

e. Make head point to new node

hptr=temp;

Step 2: If the list is not empty then make previous first node prev point to new node and new node

next point to previous first node whose address is in head. Now make tail point to new node

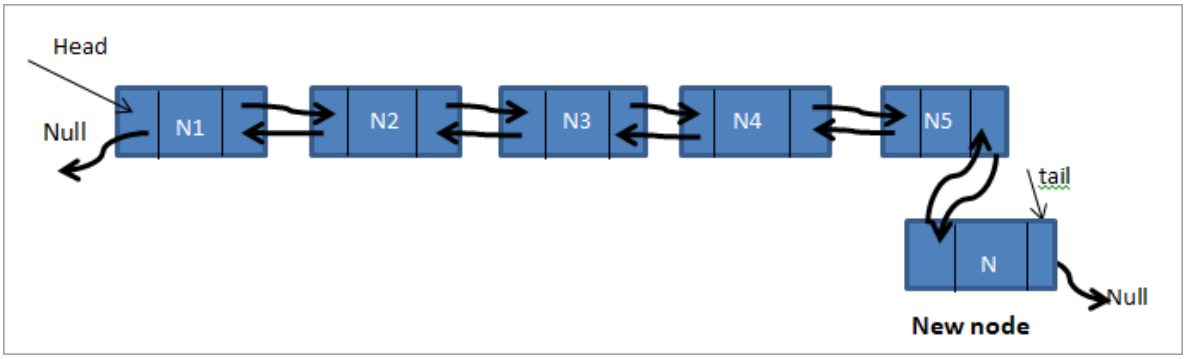
```
hptr->prev = nptr;      // previous first node point to new node
nptr->next = hptr;      // new node next point to previous first node whose address is in head
hptr = nptr;            // head point to new node
```

```
// add node at front / Beginning function
void addatfront(int d)
  {
```

```
        Node *nptr = new Node(d); // new allocate new node of type Node and return address in
     nptr
        if(hptr == NULL){
            hptr = tptr = nptr;
    }
    else
    {
           hptr->prev = nptr;       // previous first node point to new node
           nptr->next = hptr;       // new node next point to previous first node whose address is in
     head
           hptr = nptr;             // head point to new node
    }
  }
```

Insert node at the end

Diagrammatic presentation and Explanation



Inserting node at the end of the doubly linked list is achieved by pointing the next pointer of new node N to null. The previous pointer of N is pointed to N5. The 'Next' pointer of N5 is pointed to N.

Algorithm

Step 1: Get the new node

```
Node *nptr = new Node(d);
```

Step 2: If the list is empty then head and tail point to new node.

```
hptr = tptr = nptr;
```

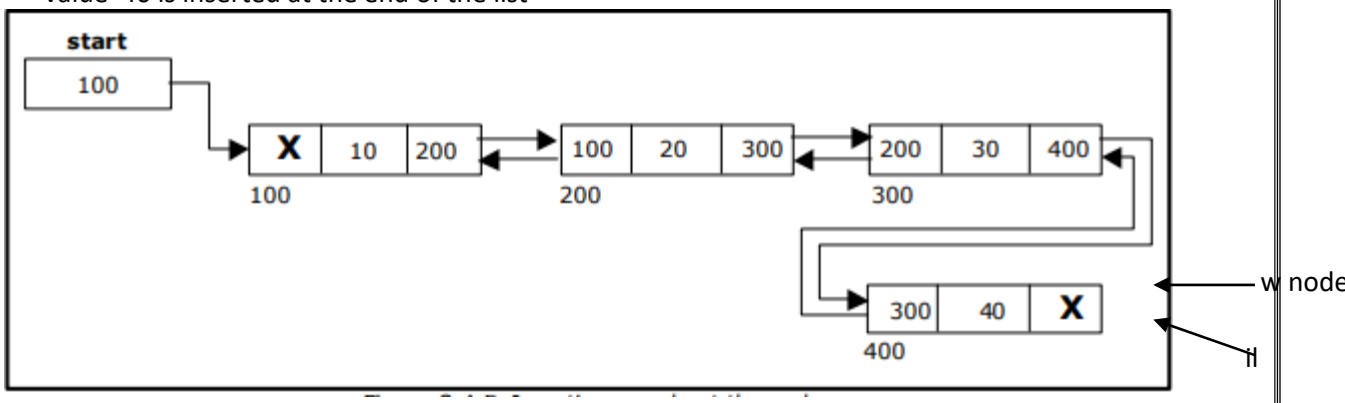
Step 3: If the list is not empty then make previous last node point to new node and new node prev
point to previous last node whose address is in tail. Now make tail point to new node

```
tptr->next = nptr;   // previous last node point to new node
nptr->prev = tptr; // new node prev point to previous last node whose address is in tail
tptr = nptr;           // tail point to new node
```

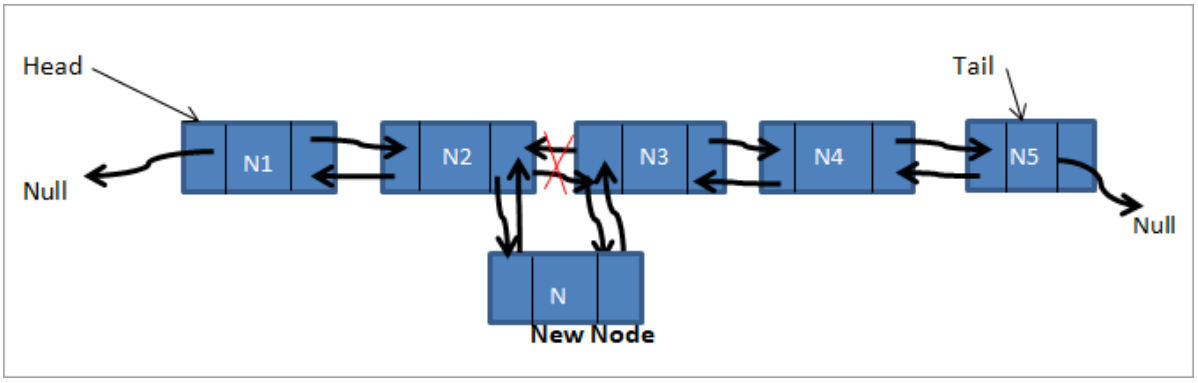
```
// add at end function or use same code for creating DLL
void add_at_end(int d)
   {
       Node *nptr = new Node(d);
       if(hptr == NULL){
           hptr = tptr = nptr;
   }
   else
   {
```

```
        tptr->next = nptr;
        nptr->prev = tptr;
        tptr = nptr;
    }
  }
```

Inserting a node at the end: The following example shows that new node with address 400 and value 40 is inserted at the end of the list



Insert node before/after given node



As shown in the above diagram, when we have to add a node before or after a particular node, we change the previous and next pointers of the before and after nodes so as to appropriately point to the new node. Also, the new node pointers are appropriately pointed to the existing nodes.

Algorithm

Step 1: If the list is empty then print "List empty"

Step 2: If the list is not empty then make p hold address of first node

p= hptr

Step 3: repeat step 4 until end of list

Step 4: If node data equal to specified data // specified node becomes previous node

Step 4.1: Get the new node

Node *temp = new Node(d);

Step 4.2: Make new node temp point previous (p) and next node

temp -> next = p -> next;

temp ->prev = p;

Step 4.3: Make next node of previous(p) node to point to temp by its prev

p->next->prev =temp;

Step 4.4: Make previous node to point to temp by its next

p -> next = temp

Step 4.5: Go to step 6

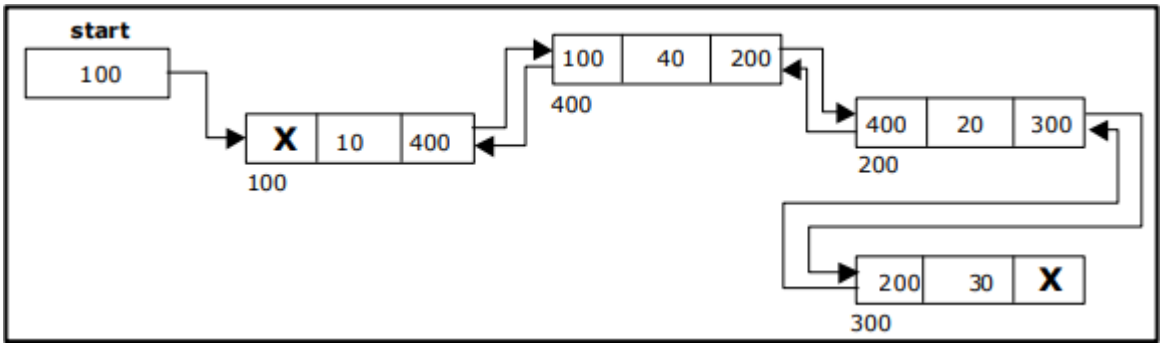
Step 4.5: make p point to next node

p = p -> next;

Step 4.6: Go to step 3
Step 5: Print specified node not found
Step 6: Stop

//Function to add node after specified node data

```
void addAfter(int ndata, int afteritem)
        {
            if (hptr == NULL)
            cout<<"List is empty";
            else
            {
              Node *temp, *p;
              p = hptr;
              do
              {
               if (p ->data == afteritem)
                {
                  Node *temp = new Node(ndata);
                  temp -> next = p -> next;
                  temp ->prev = p;
                  p->next->prev =temp;

                  p -> next = temp;

                   /*if (p->next == NULL)
                       last = temp;*/
                       return;
                }
               p = p -> next;
              }while(p!=NULL);

              cout <<"\n"<< afteritem << " not present in the list." << endl;
              return;

            }
        }
```



Here node with address 400 and data 40 has been added after node containing data value 10

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

Algorithm:

Step 1: If list is empty then display 'Empty List' message.

Step 2: If the list is not empty, follow the following steps

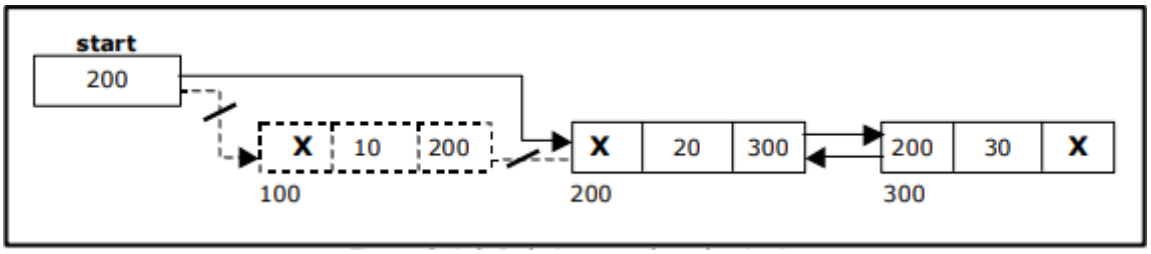
Step 2.1: If hptr->data == d // Check given data matches first node

Step 2.1.1: Make second node previous contain NULL

Step 2.1.1: Make head point to second node to make it first node in first node delete list

```
// Function to delete matched front node
void delete_at_front(int d)  // d is specified data to delete
{
   if(hptr == NULL)
   {
      cout << "List is empty";
   }
   else
     if(hptr->data == d)
    {
         Node *temp = hptr;
         temp->next->prev = NULL;
         hptr = temp->next;
     }
}
```

Figure shows deleting a node at the beginning of a double linked list.



Here first node with address 100 and data 10 is deleted by making head ((start) point to second node with address 200. Make second node prev contain NULL i.e. X here in diagram.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

Algorithm:

Step 1: If list is empty then display 'Empty List' message.

Step 2: If the list is not empty, follow the following steps

Step 2.1: If tptr->data == d // Check given data matches last node data

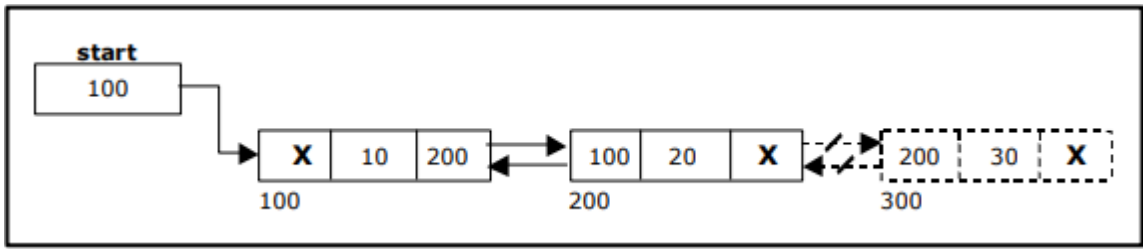
Step 2.1.1: Make last second node next contain NULL

Step 2.1.1: Make tail point to last second node, to make it last node in last node delete list

```
// Function to delete matched last node
void delete_at_end(int d)  // d is specified data to delete
```

```
{
      if(hptr == NULL)
      {
          cout << “List is empty”;
      }
      else
          if(tptr->data == d)
          {
               Node *temp = tptr;
               temp->prev->next = NULL;
               tptr = temp->prev;
      }
}
```

Figure shows deleting a node at the end of a double linked list.



Here last node with address 300 and data 30 is deleted by making tail point to last second node with address 200. Make second node next from last contain NULL i.e. X

Deleting a node at Intermediate position:

```
void delete_intermediadte_node(int d)  // d is specified data to delete
{
        Node* temp = hptr;
        while(temp!=NULL){
          if(temp->next!=NULL && temp->next->data == d){
            temp->next = temp->next->next;
            temp->next->prev = temp;
            return;
          }
          temp = temp->next;
        }
          cout<<"Element Not Found!"<<endl;

      }
```

Algorithm:

The following steps are followed, to delete a node from an intermediate position whose data matches the specified data /search data in the list. Note we go for intermediate node only after if list is not empty, or specified data not matches first or last node.

Step 1: Make temp hold address of first node

Step 2: Repeat step 3 till temp not equal to NULL // search till end of list

Step 3: if temp next node present and next node data matches specified data then

Step 3.1: make temp point to next to next node

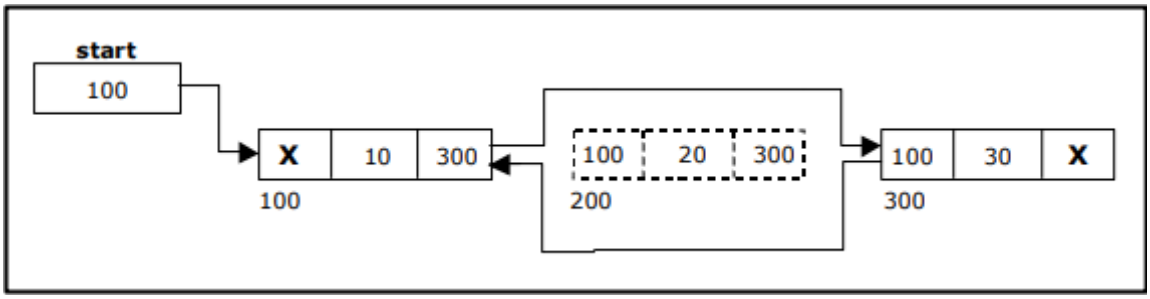
Step 3.2: make temp next node prev point to temp

Step 3.3: go to step 5

Step 4: print “Element Not Found!”

Step 5: Stop

Figure shows deleting a node that matches specified data at intermediate position other than beginning and end from a double linked list



Here node with address 200 and data 20 is deleted

Traversal and displaying a list (Left to Right) / forward traversal:

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function forward_display is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

Algorithm:

Step 1: If list is empty then display ‘Empty List’ message.

Step 2: If the list is not empty, follow the steps given below:

Step 2.1: Make temp hold address of first node

Node *temp = hptr;

Step 3: Repeat step 4 an 5 till temp not equal to NULL

while(temp!=NULL)

Step 4: display temp data

Step 5: Make temp point to next node

temp = temp -> next;

// Function forward display

```
void forward_display(){
      Node *temp = hptr;
      while(temp!=NULL){
         temp->display();
         temp = temp->next;
```

```
    }
  }
```

Traversal and displaying a list (Right to left) / reverse traversal:

The function reverse_display is used for traversing and displaying the information stored in the list from right to left.

The following steps are followed, to traverse a list from right to left

Algorithm:

Step 1: If list is empty then display 'Empty List' message.

Step 2: If the list is not empty, follow the steps given below:

Step 2.1: Make temp hold address of last node

```
Node *temp = tptr;
```

Step 3: Repeat step 4 and 5 till temp not equal to NULL

```
while(temp!=NULL)
```

Step 4: display temp data

Step 5: Make temp point to previous node

```
temp = temp -> prev;
```

// Function reverse display

```
void reverse_display(){
   Node *temp = tptr;
   while(temp!=NULL){
      temp->display();
      temp = temp->prev;
   }
}
```

Doubly Linked list Advantages

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting. 2. It is easy to reverse the linked list.

3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Disadvantages

1. It requires more space per space per node because one extra field is required for pointer to previous node.

2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.

1. What is Double Linked List?

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

2. Difference

● single linked list ● every node has a link to its next node in the sequence. ● we can traverse from one node to another node only in one direction ● we can not traverse back ● double linked list ● every node has a link to its previous node and next node. ● So, we can traverse forward by using the next field and can traverse backward by using the previous field

Applications/Uses of doubly linked list in real life

- Doubly linked list can be used in navigation systems where both front and back navigation is required.
- It is used by browsers to implement backward and forward navigation of visited web pages i.e. back and forward button.
- It is also used by various application to implement Undo and Redo functionality.
- It can also be used to represent deck of cards in games.
- It is also used to represent various states of a game.