UNIT-I

Algorithm Specification:

An algorithm is defined as a finite set of instructions that, if followed, performs a particular task. All algorithms must satisfy the following criteria

Input. An algorithm has zero or more inputs, taken or collected from a specified set of objects.

Output. An algorithm has one or more outputs having a specific relation to the inputs.

Definiteness. Each step must be clearly defined; Each instruction must be clear and unambiguous.

Finiteness. The algorithm must always finish or terminate after a finite number of steps.

Effectiveness. All operations to be accomplished must be sufficiently basic that they can be done exactly and in finite length.

We can depict an algorithm in many ways.

- Natural language: implement a natural language like English
- Flow charts: Graphic representations denoted flowcharts, only if the algorithm is small and simple.
- Pseudo code: This pseudo code skips most issues of ambiguity; no particularity regarding syntax programming language.

Example 1: Algorithm for calculating factorial value of a number

```
Step 1: a number n is inputted
Step 2: variable final is set as 1
Step 3: final<= final * n
Step 4: decrease n
Step 5: verify if n is equal to 0
Step 6: if n is equal to zero, goto step 8 (break out of loop)
Step 7: else goto step 3
Step 8: the result final is printed
```

Recursive Algorithms:

A recursive algorithm calls itself which generally passes the return value as a parameter to the algorithm again. This parameter indicates the input while the return value indicates the output.

Recursive algorithm is defined as a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is treated as the input for the next recursion. The repletion is in the self-similar fashion manner. The algorithm calls itself with smaller input values and obtains the results by simply accomplishing the operations on these smaller values. Generation of factorial, Fibonacci number series are denoted as the examples of recursive algorithms.

Example: Writing factorial function using recursion

```
intfactorialA(int n)
{
  return n * factorialA(n-1);
}
```

Abstract Data Type in Data Structures:

The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

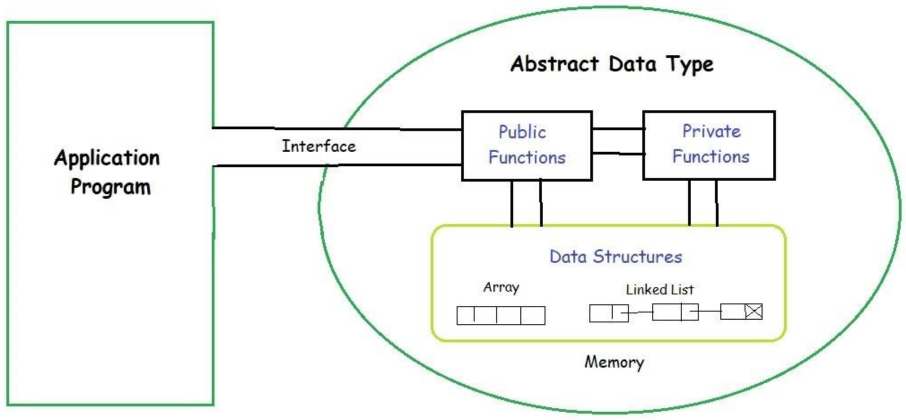
Let us see some operations of those mentioned ADT −

- Stack −
    - isFull(), This is used to check whether stack is full or not
    - isEmpry(), This is used to check whether stack is empty or not
    - push(x), This is used to push x into the stack
    - pop(), This is used to delete one element from top of the stack
    - peek(), This is used to get the top most element of the stack
    - size(), this function is used to get number of elements present into the stack
- Queue −
    - isFull(), This is used to check whether queue is full or not
    - isEmpry(), This is used to check whether queue is empty or not
    - insert(x), This is used to add x into the queue at the rear end
    - delete(), This is used to delete one element from the front end of the queue
    - size(), this function is used to get number of elements present into the queue
- List −
    - size(), this function is used to get number of elements present into the list
    - insert(x), this function is used to insert one element into the list
    - remove(x), this function is used to remove given element from the list
    - get(i), this function is used to get element at position i
    - replace(x, y), this function is used to replace x with y value

Abstract Data Types:

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

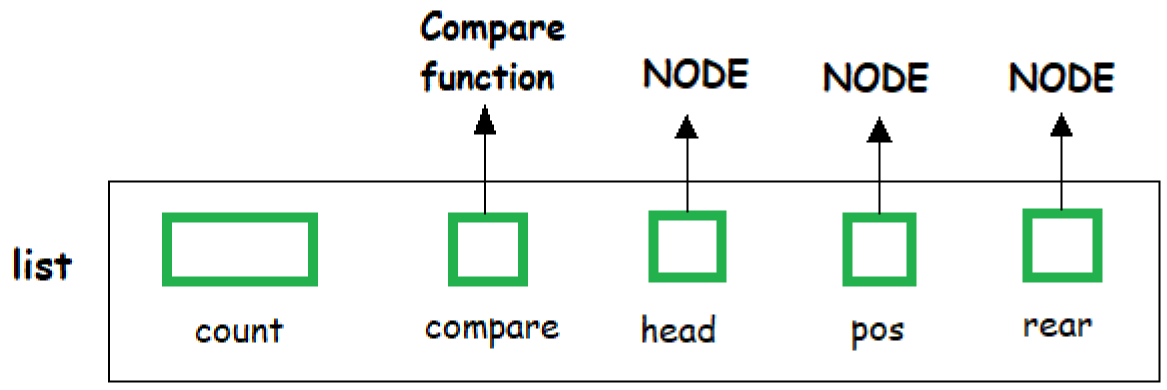
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

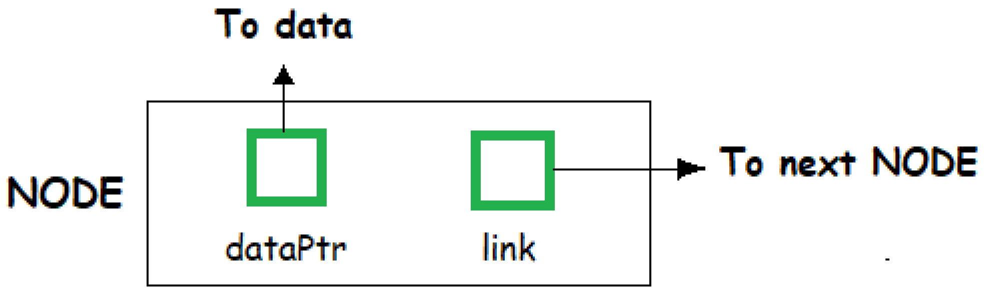


The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

1. **List ADT**

- The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.

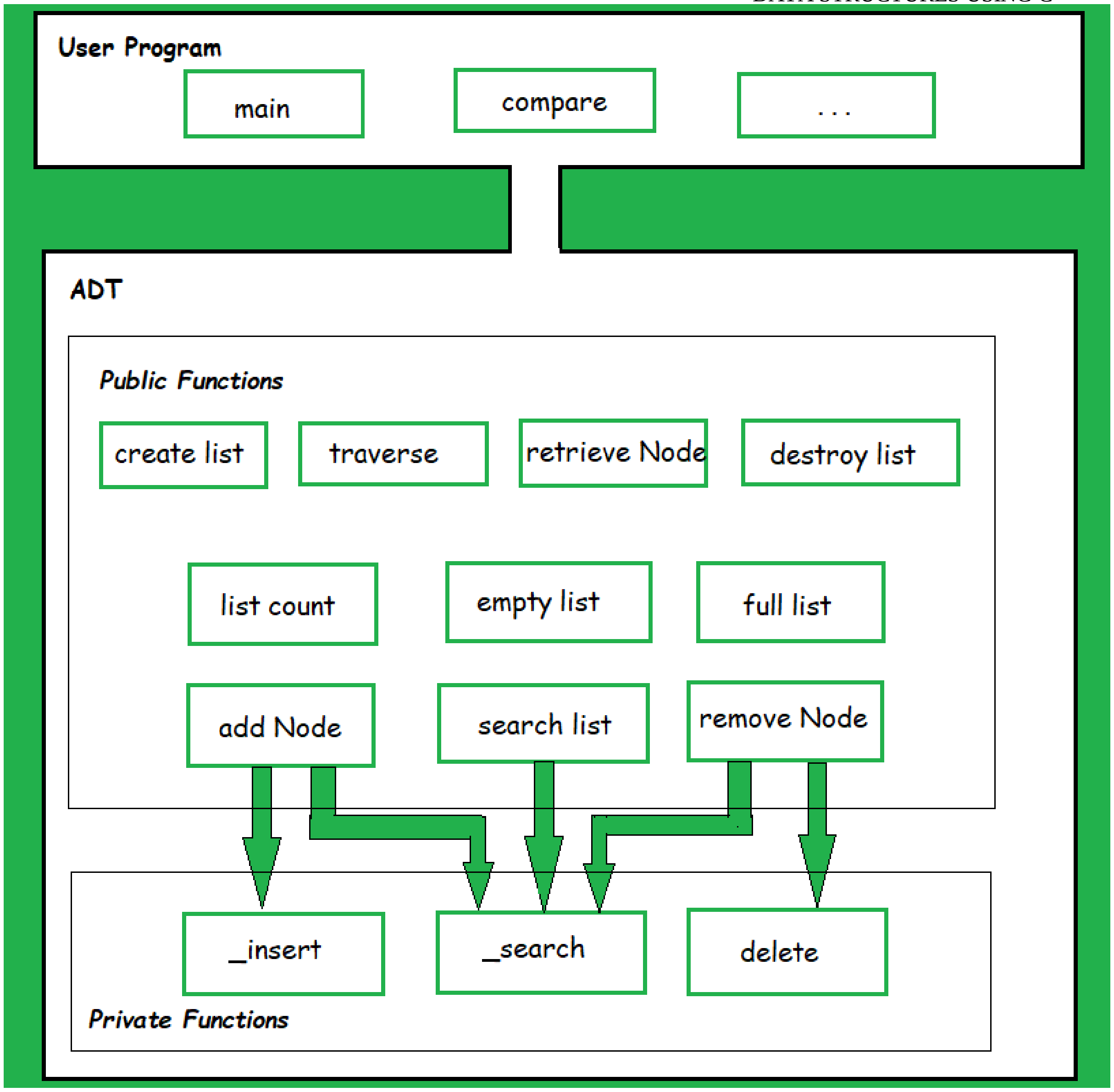




- The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.

```
//List ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} Node;
typedef struct
{
 int count;
 Node *pos;
 Node *head;
 Node *rear;
 int (*compare) (void *argument1, void *argument2)
} LIST;
```

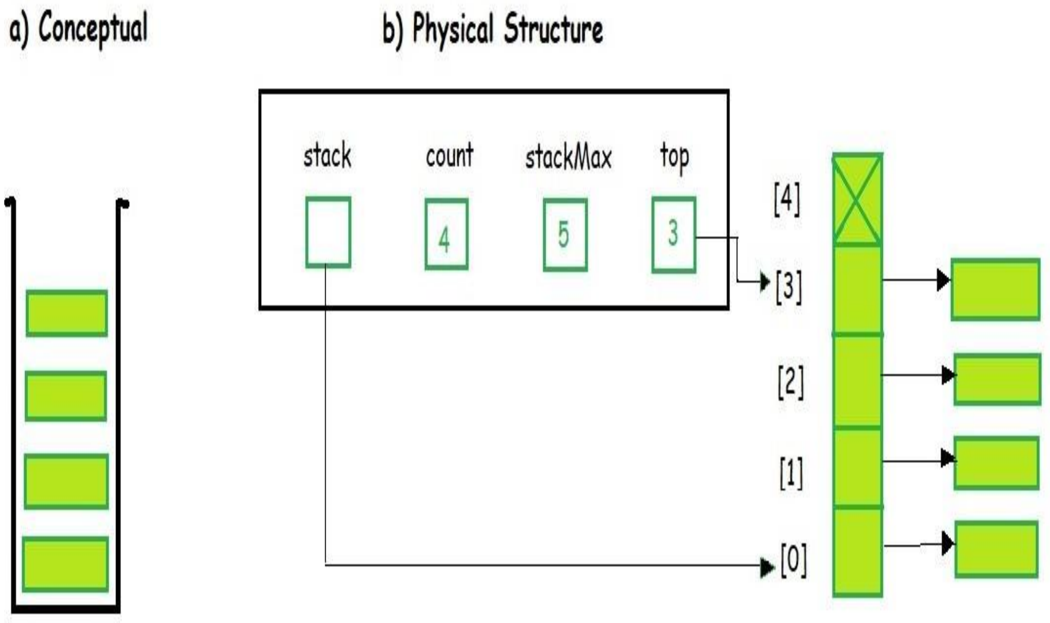
- The **List ADT Functions** is given below:



A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

2. **Stack ADT**
    - In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
    - The program allocates memory for the *data* and *address* is passed to the stack ADT.



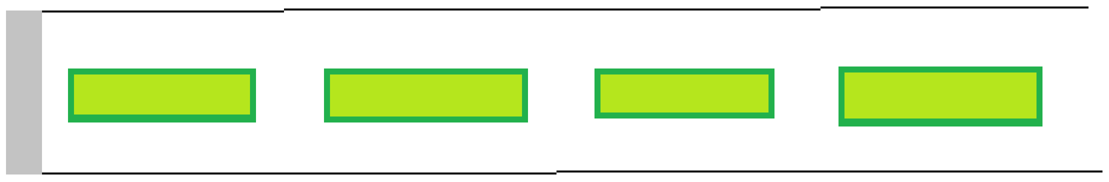
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

```
//Stack ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} StackNode;
typedef struct
{
 int count;
 StackNode *top;
} STACK;
```

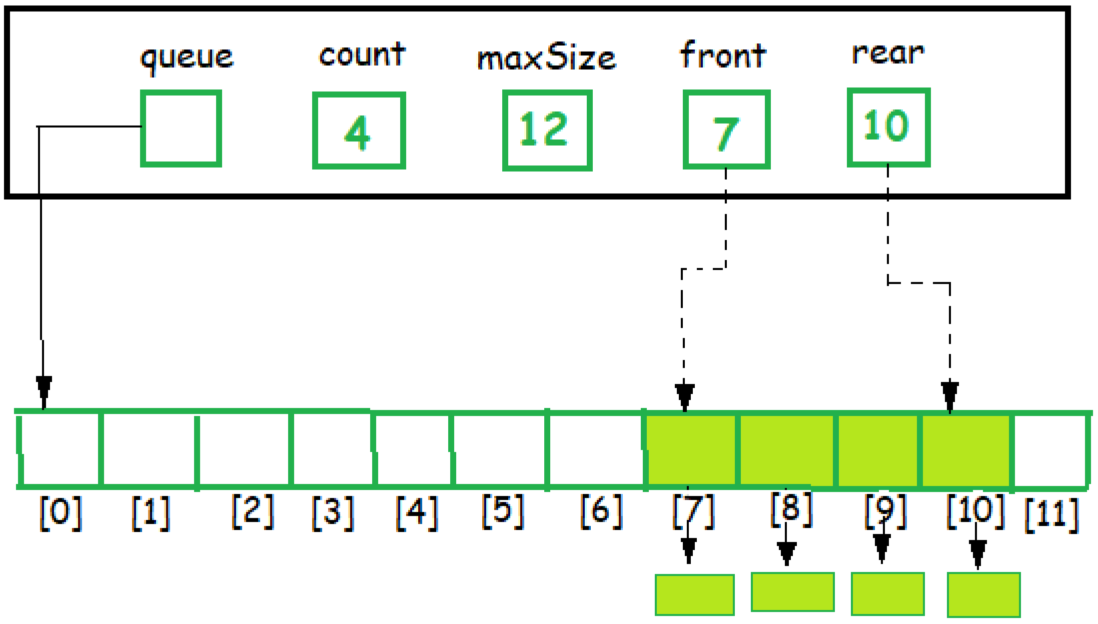
A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

3. **Queue ADT**
    - The queue abstract data type (ADT) follows the basic design of the stack abstract data type.



a) Conceptual



b) Physical Structures

- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

```
//Queue ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *next;
} QueueNode;
typedef struct
{
 QueueNode *front;
 QueueNode *rear;
 int count;
} QUEUE;
```

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

Performance Analysis:

Algorithm Efficiency:

The efficiency of an algorithm is mainly defined by two factors i.e. space and time. A good algorithm is one that is taking less time and less space, but this is not possible all the time. There is a trade-off between time and space. If you want to reduce the time, then space might increase. Similarly, if you want to reduce the space, then the time may increase. So, you have to compromise with either space or time. Let's learn more about space and time complexity of algorithms.

TIME COMPLEXITY

- Time complexity of an algorithm signifies the total time required by the program to run till its completion.
- The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity.
- Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

Example, for the first code the loop will run n number of times, so the time complexity will be n at least and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

Algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

The most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity.

```
statement;
```

Above it's a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)

{

   statement;

}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)

{

   for(j=0; j < N;j++)

   {

   statement;

   }

}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)

{

   mid = (low + high) / 2;

   if (target < list[mid])

      high = mid - 1;

   else if (target > list[mid])

      low = mid + 1;

   else break;

}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)

{
    int pivot = partition(list, left, right);

    quicksort(list, left, pivot - 1);

    quicksort(list, pivot + 1, right);

}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE: In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Suppose calculate an algorithm takes f(n) operations, where,

```
f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

SPACE COMPLEXITY:

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime Auxiliary Space is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = Auxiliary Space + Input space

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

Sometimes an algorithm (function) may be called inside another algorithm (function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm (function) is made.

For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the function A().

3. **Data Space**

Amount of space used by the variables and constants.

But while calculating the Space Complexity of any algorithm, we usually consider only Data Space and we neglect the Instruction Space and Environmental Stack.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of data type variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

To compute space complexity by taking a few examples:

```
{
   int z = a + b + c;
   return(z);
}
```

In the **above expression**, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be (4(4) + 4) = 20 bytes, this additional 4 bytes is for return value. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity.**

Another example, this time a bit complex one,

```
// n is the length of array a[]
```

```
int sum(int a[], int n)
{
        int x = 0;                  // 4 bytes for x
        for(int i = 0; i < n; i++)          // 4 bytes for i
        {
           x  = x + a[i];
        }
        return(x);
}
```

In the above code, 4*n bytes of space is required for the array a[] elements.
4 bytes each for x, n, i and the return value.
Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as **Linear Space Complexity.**
Similarly, quadratic and other complex space complexity as well, as the complexity of an algorithm increases. But always focus on writing algorithm code in such a way keep the space complexity minimum.

Asymptotic Notations:

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required. So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis proce

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 has least significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**.
Here, for larger value of **'n'** the value of most significant terms (**$5n^2$** and **$10n^2$**) is very larger than the value of least significant terms (**2n + 1** and **8n + 3**). So for larger value of **'n'** we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

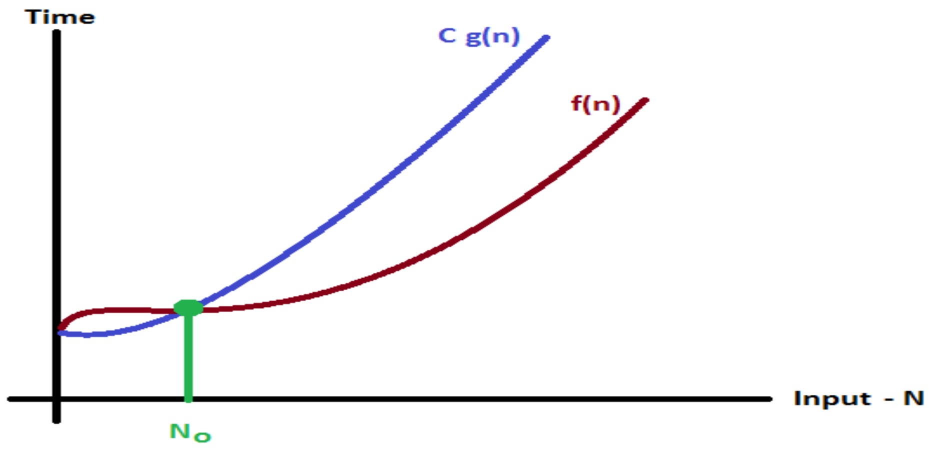
Big - Oh Notation (O):

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity. Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If $f(n) <= C\, g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent f(n) as O(g(n)).

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) <= C g(n)
⇒3n + 2 <= C n
Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

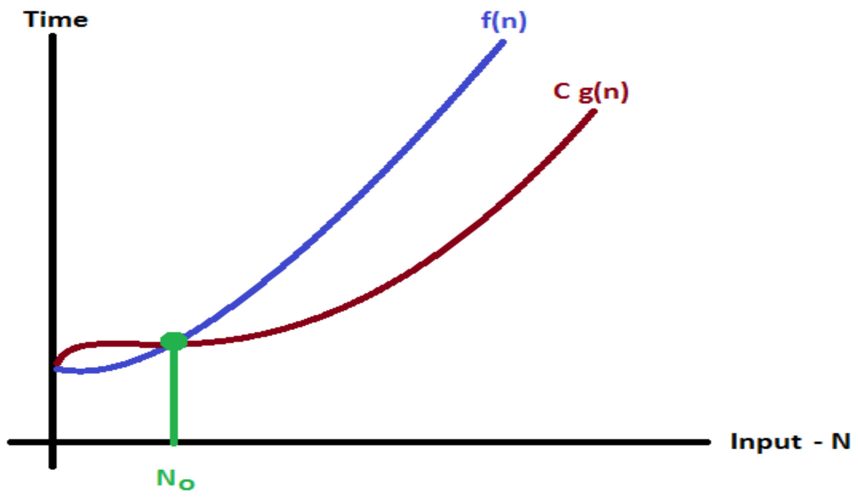
Big - Omege Notation (Ω):

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity. That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) >= C g(n)
⇒3n + 2 >= C n
Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

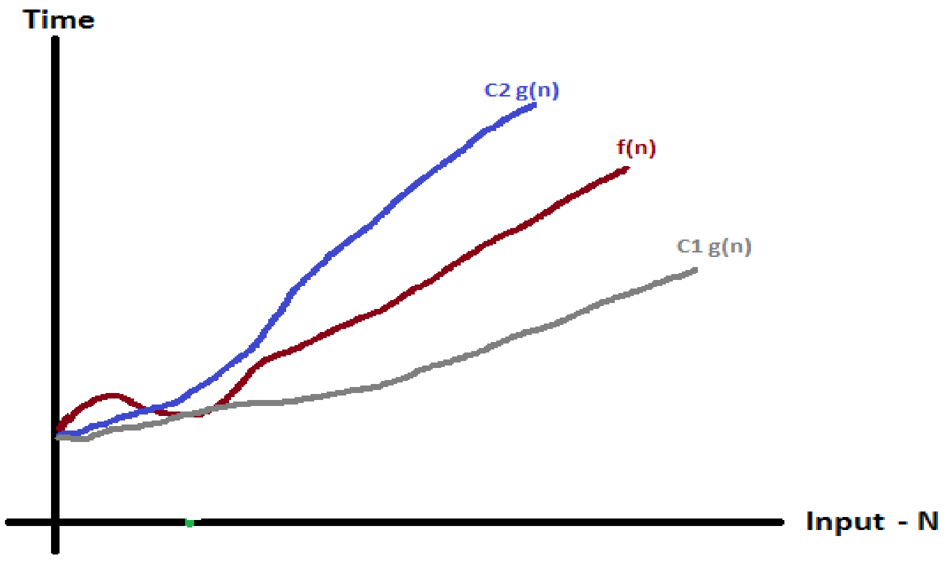
Big - Theta Notation (Θ):

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.
Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**
C_1 g(n) <= f(n) <= C_2 g(n)
⇒C_1 n <= 3n + 2 <= C_2 n
Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.
By using Big - Theta notation we can represent the time compexity as follows...
3n + 2 = Θ(n)

Introduction to Data Structures:

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's **name** "Virat" and **age** 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

If you are aware of Object Oriented programming concepts, then a class also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

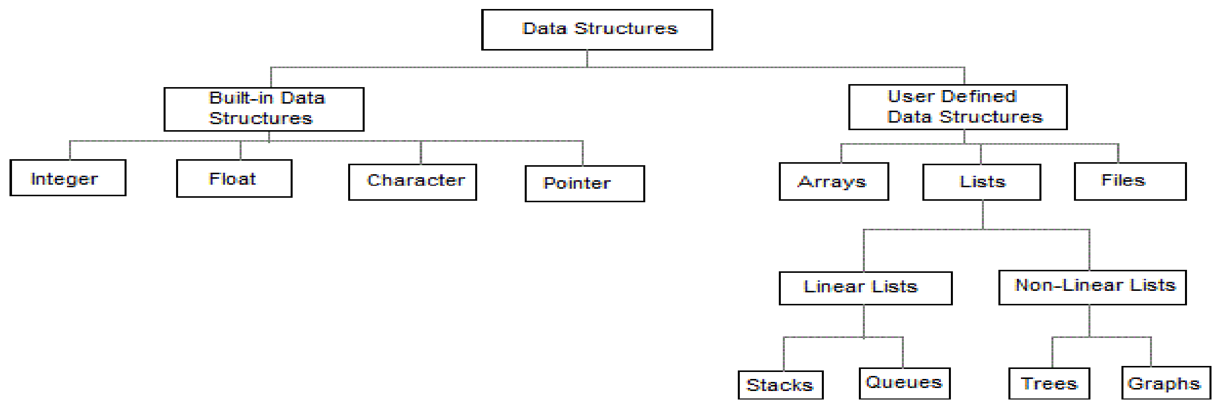
Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

Characterstic	**Description**
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

Linked lists:

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast.

The disadvantages of arrays are:

• The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

• Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

• Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Linked List Concepts:

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer.

It is a dynamic data structure.In linked list, each node consists of its own data and the address of the next node and forms a chain.

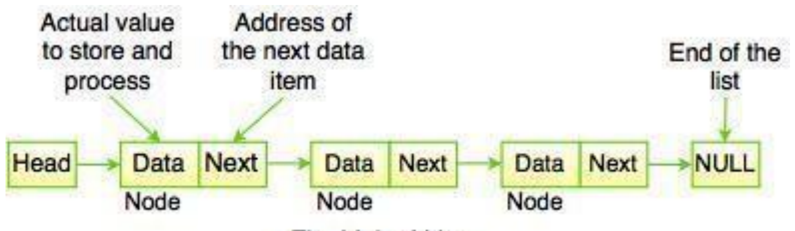

Actual value to store and process
Address of the next data item
End of the list
Head
Data
Next
Data
Next
Data
Next
NULL
Node
Node
Node

Fig. Linked List

The above figure shows the sequence of linked list which contains data items connected together via links. It can be visualized as a chain of nodes, where every node points to the next node.

For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. Entry point into the linked list is called the **head of the list.**

Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of the list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Note:

**Head is not a separate node but it is a reference to the first node. If the list is empty, the head is a null reference.

**Data part of the node is also referred as info and link part of the node is also referred as ptr or next.

Linked list is a dynamic data structure. While accessing a particular item, head at the head and follow the references until you get that data item.

Advantages of linked lists:

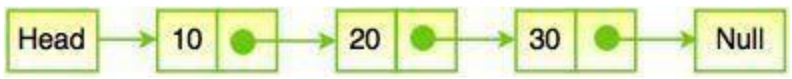
Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre- allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.
5. In linked list, stack and queue can be easily executed.
6. It reduces the access time.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.
3. Reverse traversing is difficult in linked list.
4. Linked list has to access each node sequentially; no element can be accessed randomly.
5. In linked list, the memory is wasted as pointer requires extra memory for storage.

Linked list is used while dealing with an unknown number of objects:



In the above diagram, Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL.

The real life **example of Linked List** is that of Railway Carriage. It heads from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

Creation of linked list:

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request.

It is defined by:

```
void *malloc (number_of_bytes)
```

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
        char *cp;
        cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

```
int *ip;
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function.

The syntax is:

```
free (ptr);
```

The advantage of free() is simply memory management when we no longer need a block.

Structure of a linked list:

The linked list will be created using the following structure

```
        struct node
         {
            int num;
            struct node *ptr;
          };
          typedef struct node NODE;
          NODE *head, *newnode, *temp=0;
```

Each node of the linked list is created and will be allocated memory space during run time as follows.

The above code identifies the basic structure of declaring a node. Struct node contains an int data field and a pointer to another node can be defined as struct node* next.

```
    newnode=(NODE*) malloc(sizeof(NODE));
```

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular single Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

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

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

Sl No.	Array	Linked list
1.	Size of an array is fixed	Size of a list is not fixed
2.	Memory is allocated from stack	Memory is allocated from heap
3.	It is necessary to specify the number of elements during declaration (i.e., during compile time)	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time)
4.	It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
5.	Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
6.	Deleting an element from an array is not possible.	Deleting an element is possible

Trade offs between linked lists and arrays:

Feature	**Array**	**Linked list**
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Following are the operations that can be performed on a Linked List:

1. Create
2. Insert
3. Delete
4. Traverse
5. Search
6. Concatenation
7. Display

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents.

For example:

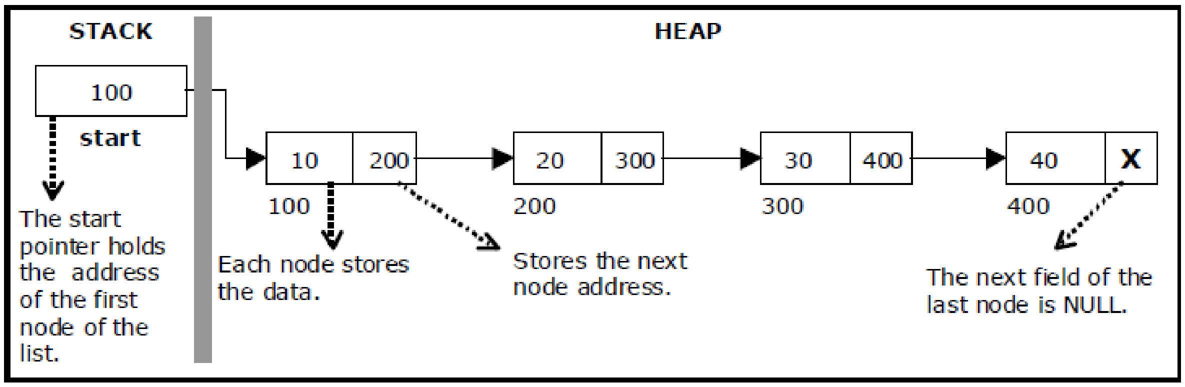
P(x) = a0 Xn + a1 Xn-1 +....... + an-1 X + an

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction

2.1 Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node.

A single linked list is shown in figure.


STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
10
200
100
20
300
200
30
400
300
40
X
400
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Note: Start or head is same.

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to showthat they are allocated in the heap.

Implementation of Single Linked List:

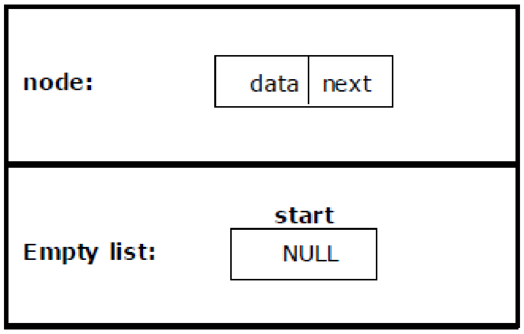
Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see):

Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
Initialise the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```



Structure definition, single link node and empty list

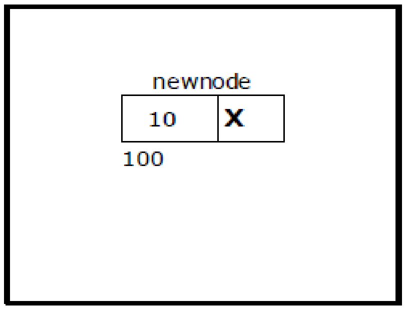
The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 3.2.3 illustrates the

```
node* getnode()
{
    node* newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}
```



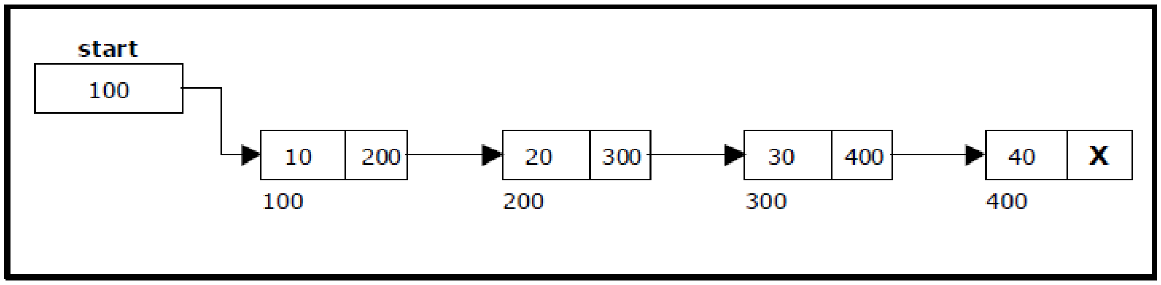
creation of a node for single linked list.

Creating a Singly Linked List with ‘n’ number of nodes:

The following steps are to be followed to create ‘n’ number of nodes:

Get the new node using getnode().
newnode = getnode();
If the list is empty, assign new node as start. start
= newnode;
If the list is not empty, follow the steps given below:
The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.
The start pointer is made to point the new node by assigning the address of the new node.
Repeat the above steps ‘n’ times.

shows 4 items in a single linked list stored at different locations in memory.



```
void createlist(int n)
{
        int i;
        node * newnode;
        node *temp;
        for(i = 0; i < n ; i+ +)
        {
                newnode = getnode();
                if(start = = NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                              while(temp - > next != NULL)
                              temp = temp - > next;
                        temp - > next = newnode;
                }
        }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field.

The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

Inserting a node at the beginning.
Inserting a node at the end.
Inserting a node at intermediate position.

Inserting a node at the beginning:

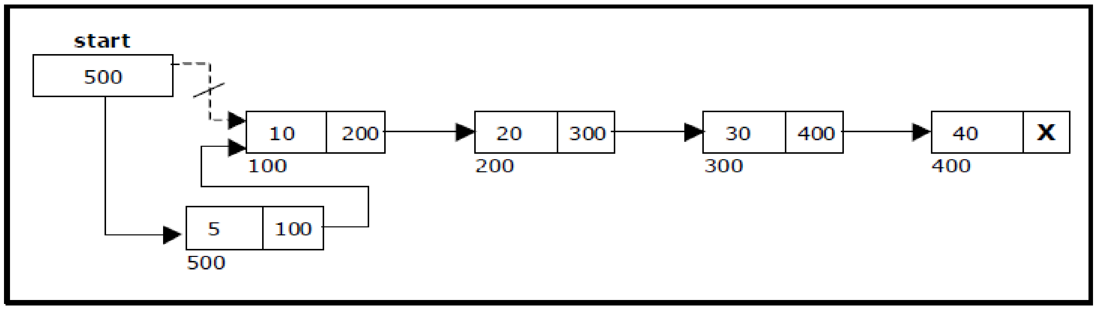
The following steps are to be followed to insert a new node at the beginning of the list:

Get the new node using getnode().
newnode = getnode();

If the list is empty then *start = newnode.*

If the list is not empty, follow the steps given below:
newnode -> next = start;
start = newnode;

Figure shows inserting a node into the single linked list at the beginning.



Inserting a node at the end:

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

Get the new node using getnode()

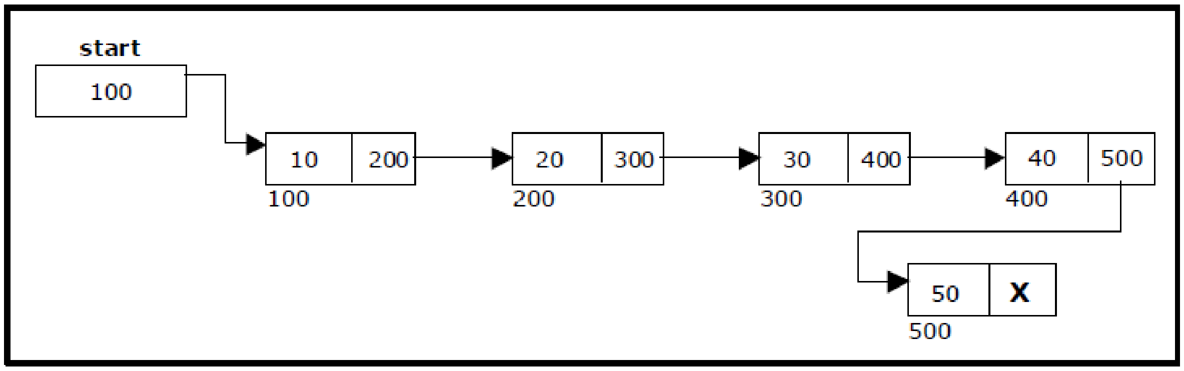
```
newnode = getnode();
```

If the list is empty then *start = newnode*.

If the list is not empty follow the steps given below: temp

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

Figure shows inserting a node into the single linked list at the end.



Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

Get the new node using getnode().

newnode = getnode(

Ensure that the specified position is in between first node and last node If not, specified position is invalid This is done by countnode()function

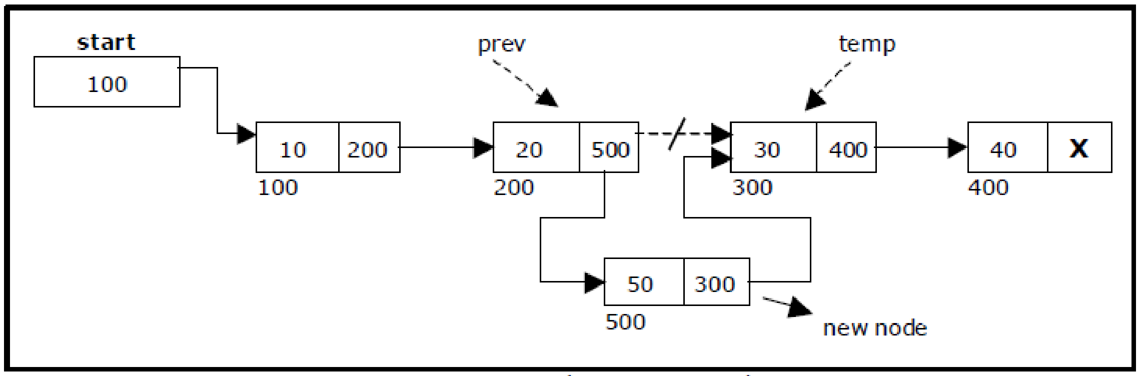
Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode -> next = temp;
```

Let the intermediate position be 3.

Figure shows inserting a node into the single linked list at a specified intermediate position



other than beginning and end.
The function insert_at_mid(), is used for inserting a node in the intermediate position.
Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

Deleting a node at the beginning.
Deleting a node at the end.
Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:
If list is empty then display 'Empty List' message.
If the list is not empty, follow the steps given below: temp
= start;
start = start -> next;
free(temp);

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

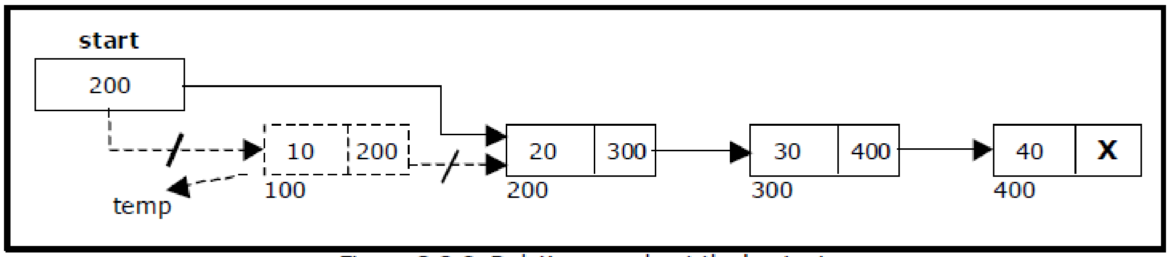


Figure 3.2.8. Deleting a node at the beginning.

The function delete_at_beg(), is used for deleting the first node in the list.

Deleting a node at the end:

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

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

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

```
temp = prev = start;
while(temp -> next != NULL)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```

Figure 3.2.9 shows deleting a node at the end of a single linked list.

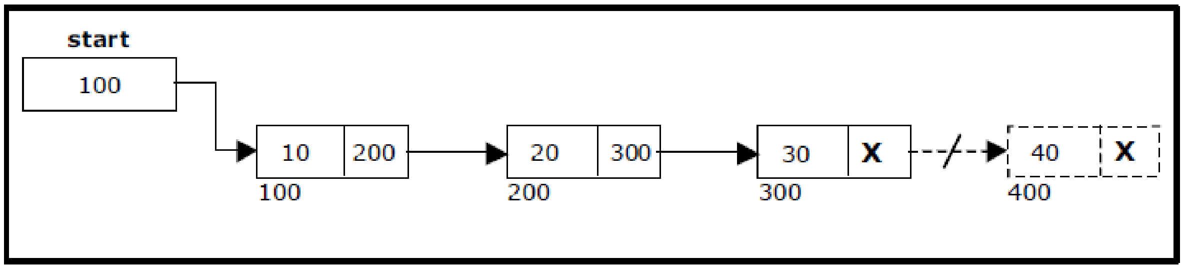


Figure 3.2.9. Deleting a node at the end.

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

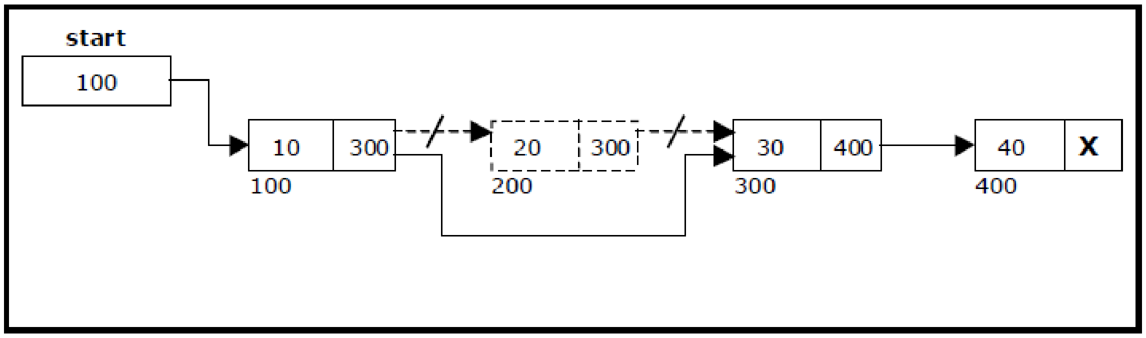
If list is empty then display ‘Empty List’ message

If the list is not empty, follow the steps given below.

```
 temp=start ;
 while (temp□ link!=NULL && temp□ data!=key)
 {
    prev=temp;
    temp=temp□  link;
  }
if(temp□ data==key)
{
  prev□  link=temp□  link;
  temp□  link=NULL;
}
else
{
printf("key element not found \n");
          }
          }
```

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

Figure 3.2.10. Deleting a node at an intermediate position.



Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

Assign the address of start pointer to a temp pointer.
Display the information from the data field of each node.

The function *traverse*() is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right): \n");
        if(start == NULL )
                printf("\n Empty List");
        else
        {
                while (temp != NULL)
                {
                        printf("%d ->", temp -> data);
                        temp = temp -> next;
                }
        }
        printf("X");
}
```

3.2. Double Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

Left link.
Data.
Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

Creation.
Insertion.
Deletion.
Traversing.

A double linked list is shown in figure 3.3.1.

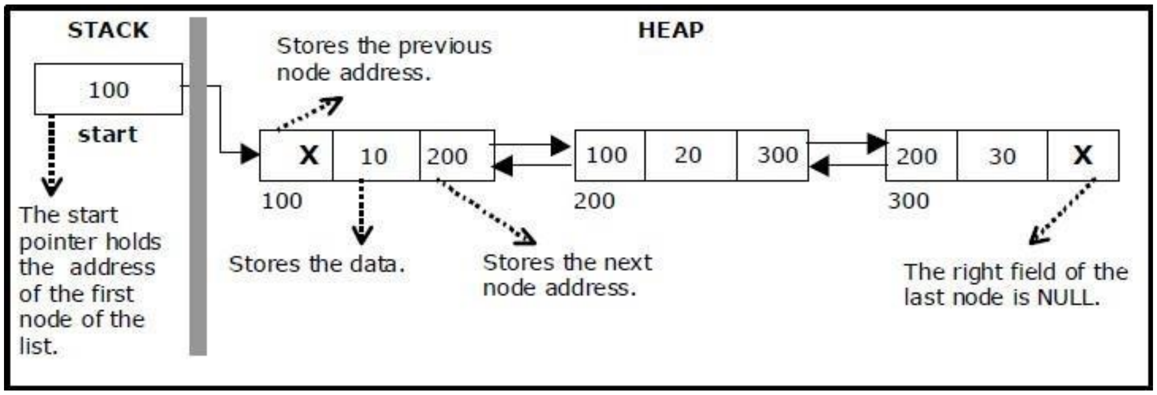
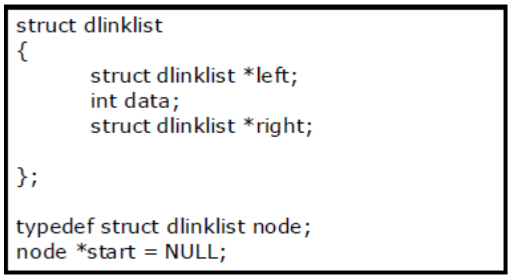

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Figure 3.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL



```
struct dlinklist
{
        struct dlinklist *left;
        int data;
        struct dlinklist *right;

};

typedef struct dlinklist node;
node *start = NULL;
```

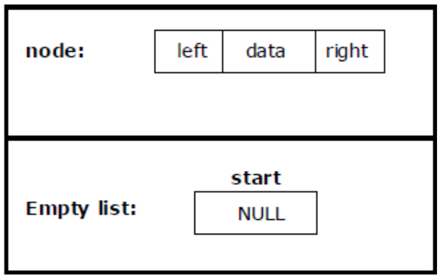
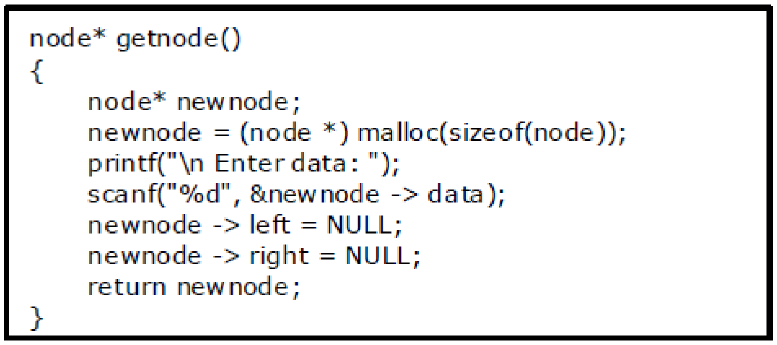


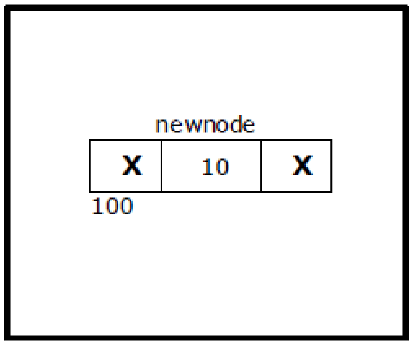
Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL (see figure 3.2.2).



```
node* getnode()
{
     node* newnode;
     newnode = (node *) malloc(sizeof(node));
     printf("\n Enter data: ");
     scanf("%d", &newnode -> data);
     newnode -> left = NULL;
     newnode -> right = NULL;
     return newnode;
}
```



Creating a Double Linked List with ‘n’ number of nodes:

The following steps are to be followed to create ‘n’ number of nodes:

Get the new node using getnode().

newnode =getnode();

If the list is empty then *start* = *newnode*.

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

The left field of the new node is made to point the previous node.

The previous nodes right field must be assigned with address of the new node.

Repeat the above steps ‘n’ times.

The function createlist(), is used to create ‘n’ number of nodes:

```
void createlist(int n)
{
        int i;
        node *newnode;
        node *temp;
        for(i = 0; i < n; i++)
        {
                newnode = getnode();
                if(start == NULL)
                {
                        start = newnode;
                }
                else
                {
                        temp = start;
                        while(temp -> right)
                                temp = temp -> right;
                        temp -> right = newnode;
                        newnode -> left = temp;
                }
        }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

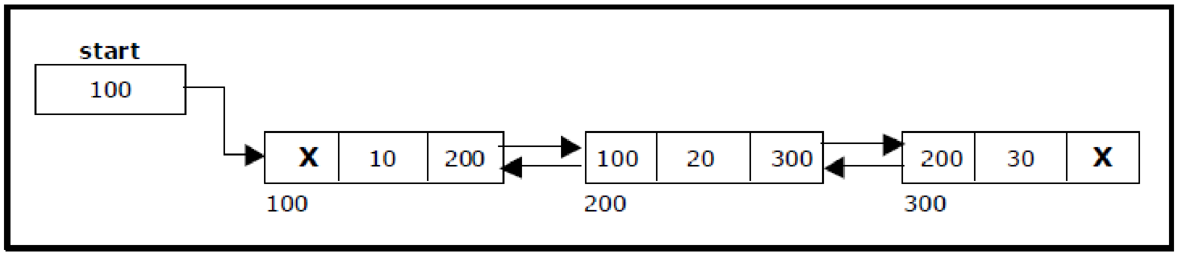


Figure 3.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list: Get the new node using getnode().

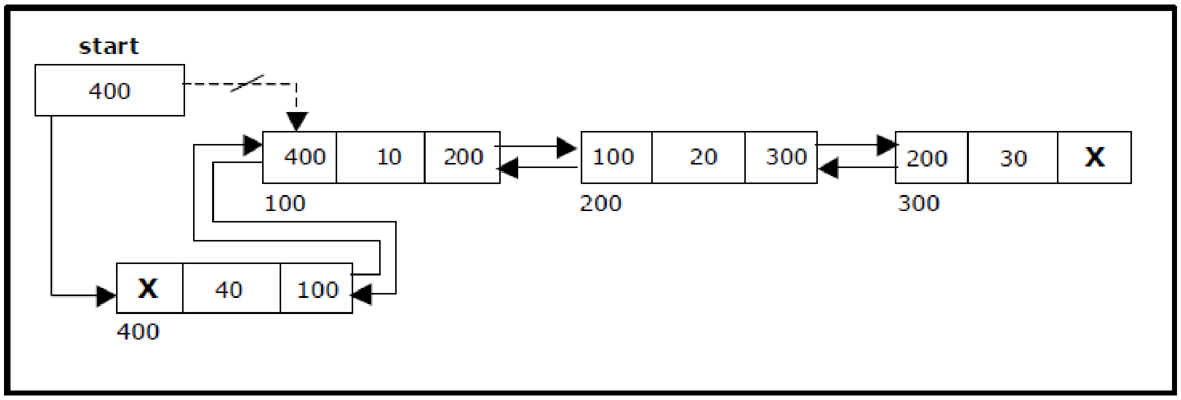
newnode=getnode();

If the list is empty then *start = newnode*.

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

```
newnode -> right = start;
start -> left = newnode;
start = newnode;
```

The function dbl_insert_beg(), is used for inserting a node at the beginning. Figure 3.4.4 shows inserting a node into the double linked list at the beginning.



Inserting a node at the end:

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

Get the new node using getnode()

```
newnode=getnode();
```

If the list is empty then *start = newnode*.

If the list is not empty follow the steps given below:

```
temp = start;
    while(temp -> right != NULL)
            temp = temp -> right;
temp -> right = newnode;
newnode -> left = temp;
```

The function dbl_insert_end(), is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

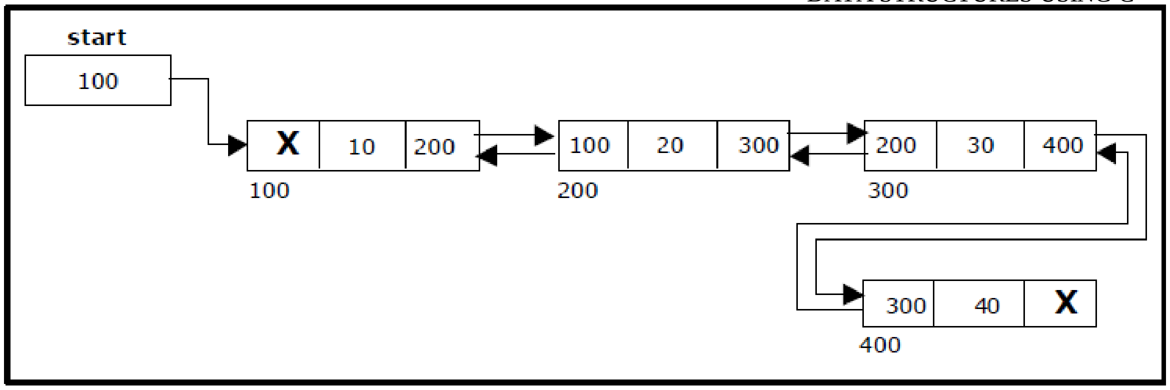


Figure 3.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

Get the new node using getnode().

newnode=getnode();

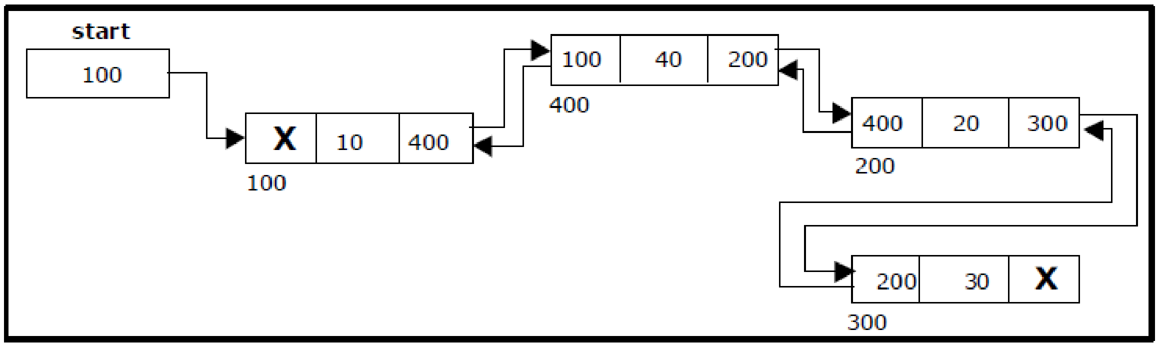
Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

After reading the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
```

The function dbl_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.



Deleting a node at the beginning:

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

If list is empty then display 'Empty List' message.

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

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list. Figure 3.4.6 shows deleting a node at the beginning of a double linked list.

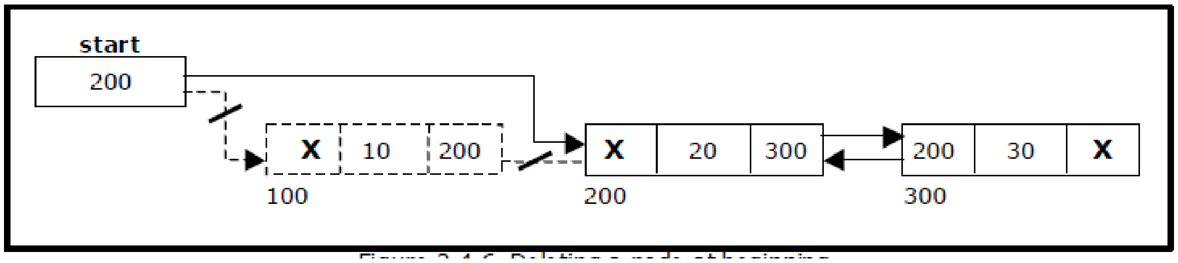


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

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

If list is empty then display 'Empty List' message

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

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp -> right;
}
temp -> left -> right = NULL;
free(temp);
```

The function dbl_delete_last(), is used for deleting the last node in the list. Figure 3.4.7 shows deleting a node at the end of a double linked list.

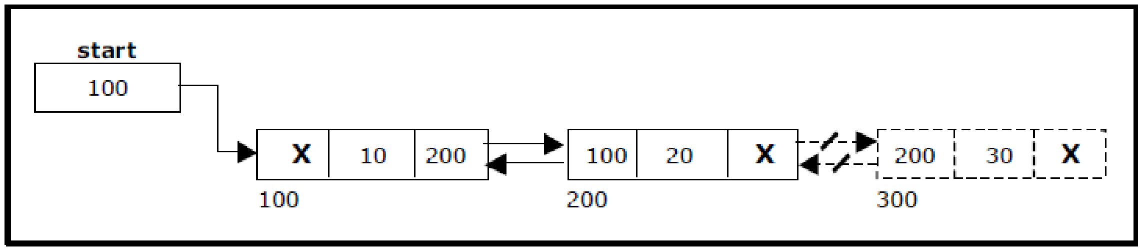


Figure 3.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

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

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

Get the position of the node to delete.

Ensure that the specified node is in between first node and last node. If not, specified key is invalid.

Then perform the following steps:

```
temp=head;
    while(temp data!=key&&temp right!=Null)
    {
        previous=temp;
        temp=temp  right;
}
if(temp data==key)
{
previous right=temp      right;
temp  right  left=temp  left;
temp  left =Null;
temp  right =Null;
}
else
{
    printf(“Invalid key”);
}
}
```

The function delete_at_mid(), is used for deleting the intermediate node in the list. Figure 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

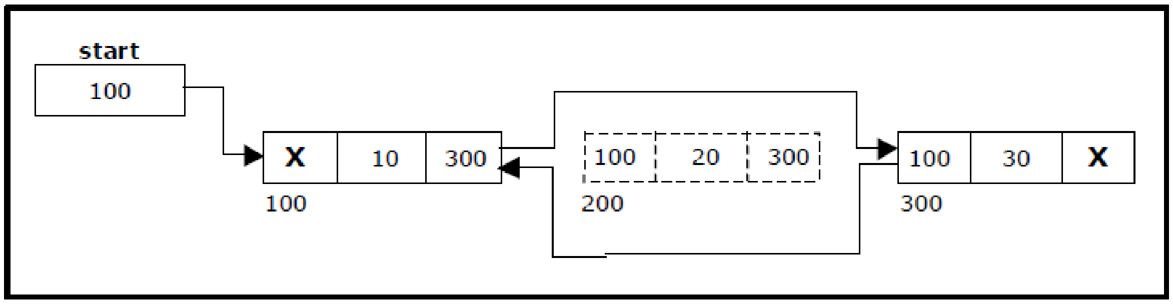


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

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 *traverse_left_right*() 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:

```
If list is empty then display ‘Empty List’ message.
If the list is not empty, follow the steps given below:
    temp = start;
    while(temp != NULL)
    {
            print temp -> data;
            temp = temp -> right;
    }
```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left*() 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:

```
If list is empty then display ‘Empty List’ message.

If the list is not empty, follow the steps given below:
temp= start;
        while(temp -> right != NULL)
                temp = temp -> right;
    while(temp != NULL)
    {
            print temp -> data; temp
            = temp -> left;
    }
```

2.2.1. Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

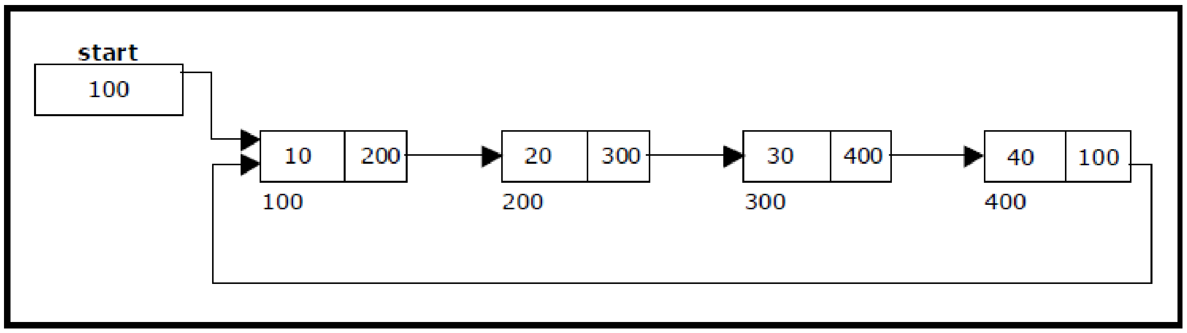


Figure 3.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

Creation.
Insertion.
Deletion.
Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

Get the new node using getnode().

newnode = getnode();

If the list is empty, assign new node as start. start

= newnode;

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

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

Repeat the above steps ‘n’ times.

newnode -> next = start;

The function createlist(), is used to create ‘n’ number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

Get the new node using getnode().

```
newnode = getnode();
```

If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

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

```
last = start;
     while(last -> next != start)
             last = last -> next;
newnode -> next = start;
start = newnode;
last -> next = start;
```

The function cll_insert_beg(), is used for inserting a node at the beginning. Figure 3.6.2 shows inserting a node into the circular single linked list at the beginning.

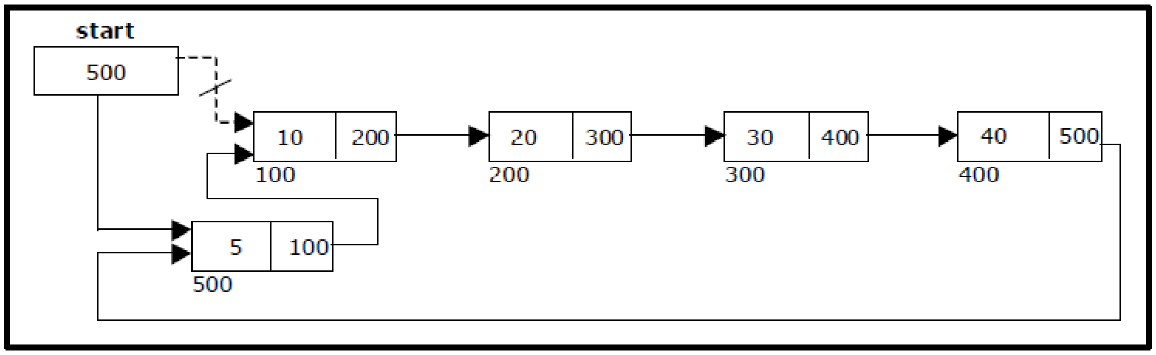


Figure 3.6.2. Inserting a node at the beginning

Inserting a node at the end:

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

```
Get the new node using getnode().
      newnode = getnode();
If the list is empty, assign new node as start.
      start = newnode;
      newnode -> next = start;
If the list is not empty follow the steps given below:
      temp = start;
      while(temp -> next != start)
            temp = temp -> next;
      temp -> next = newnode;
      newnode -> next = start;
```

The function cll_insert_end(), is used for inserting a node at the end.

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

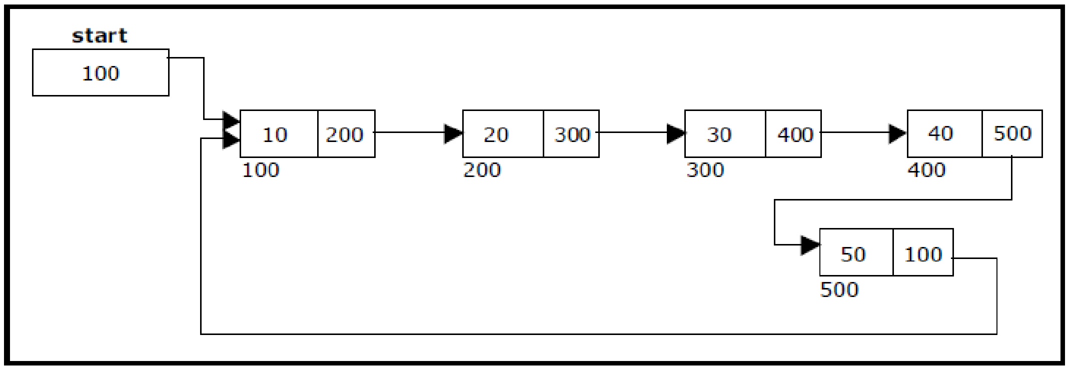


Figure 3.6.3 Inserting a node at the end**Deleting a node at the beginning:**

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

```
If the list is empty, display a message 'Empty List'.
If the list is not empty, follow the steps given below:

      last = temp = start;
      while(last -> next != start)
            last = last -> next;
      start = start -> next;
```

```
        last -> next = start;
```

After deleting the node, if the list is empty then *start = NULL*.

The function cll_delete_beg(), is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

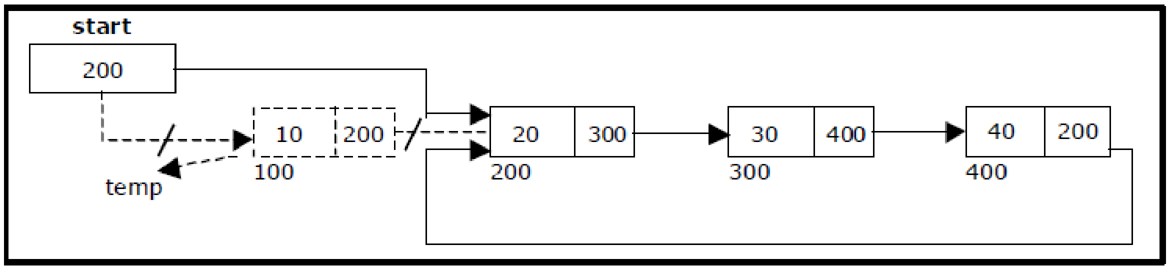


Figure 3.6.4. Deleting a node at beginning.

Deleting a node at the end:

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

If the list is empty, display a message 'Empty List'.

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

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = start;
```

After deleting the node, if the list is empty then *start = NULL*.

The function cll_delete_last(), is used for deleting the last node in the list.

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

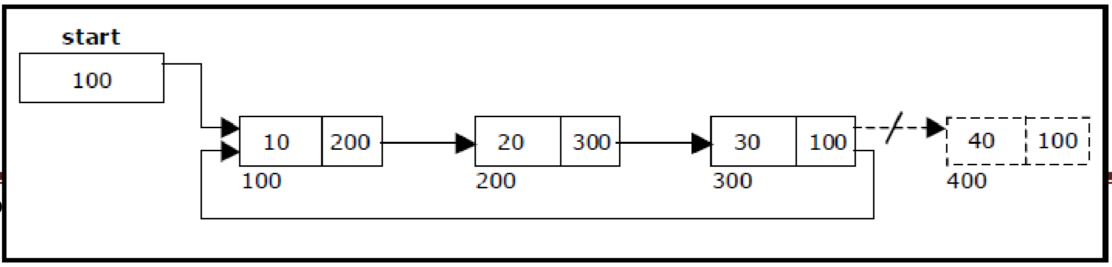


Figure 3.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:

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

If list is empty then display 'Empty List' message.

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

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

2.3.2. Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

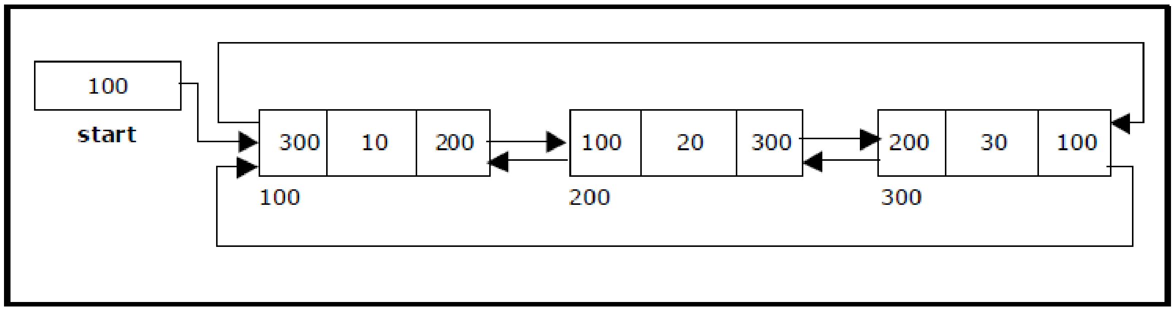


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

Creation.

Insertion.

Deletion.

Traversing.

Creating a Circular Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

Get the new node using getnode().

newnode = getnode();

If the list is empty, then do the following start
= newnode;
newnode -> left = start;
newnode ->right = start;

If the list is not empty, follow the steps given below:
newnode -> left = start -> left;
newnode -> right = start; start ->
left->right = newnode; start -> left =
newnode;

Repeat the above steps 'n' times.

The function cdll_createlist(), is used to create 'n' number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:
Get the new node using getnode().
newnode=getnode();

If the list is empty, then
start = newnode;
newnode -> left = start;
newnode -> right = start;

If the list is not empty, follow the steps given below:
newnode -> left = start -> left;
newnode -> right = start; start -> left -
> right = newnode; start -> left =
newnode;
start = newnode;

The function cdll_insert_beg(), is used for inserting a node at the beginning. Figure 3.8.2 shows inserting a node into the circular double linked list at the beginning.

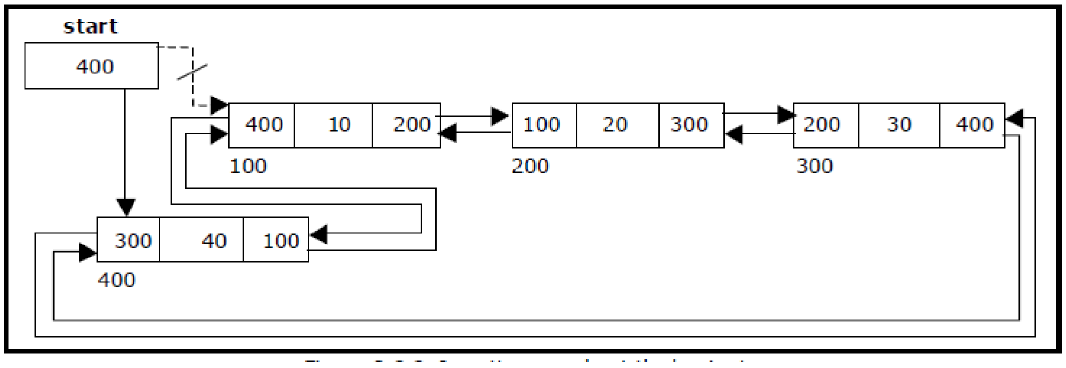


Figure 3.8.2. Inserting a node at the beginning

Inserting a node at the end:

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

Get the new node using getnode()

newnode=getnode();

If the list is empty, then

start = newnode;

newnode -> left = start;

newnode -> right = start;

If the list is not empty follow the steps given below:

newnode -> left = start -> left;

newnode -> right = start; start -> left -

> right = newnode; start -> left =

newnode;

The function cdll_insert_end(), is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

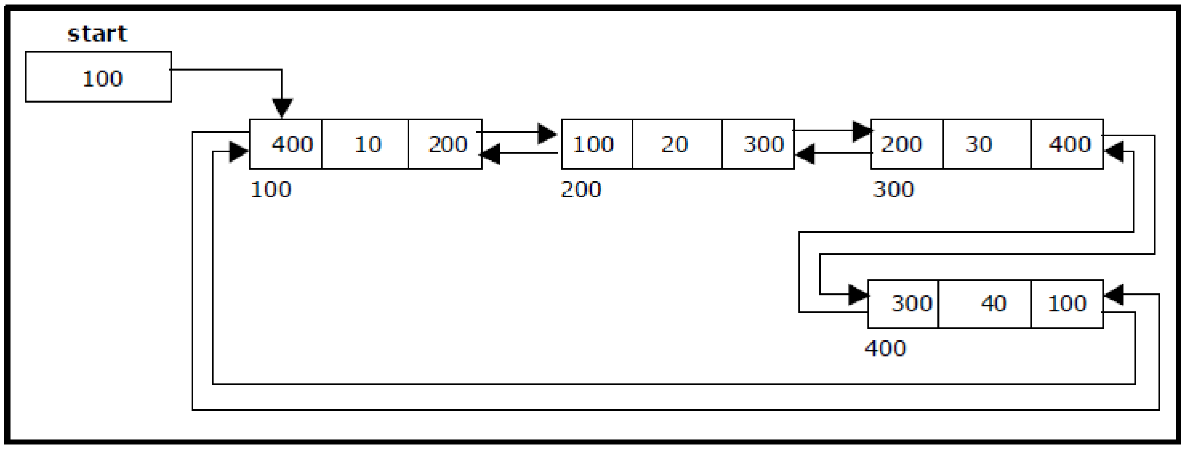


Figure 3.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

Get the new node using getnode().

```
newnode=getnode();
```

Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

Store the starting address (which is in start pointer) in temp. Then traverse the temp pointer upto the specified position.

After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
nodectr++;
```

The function cdll_insert_mid(), is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

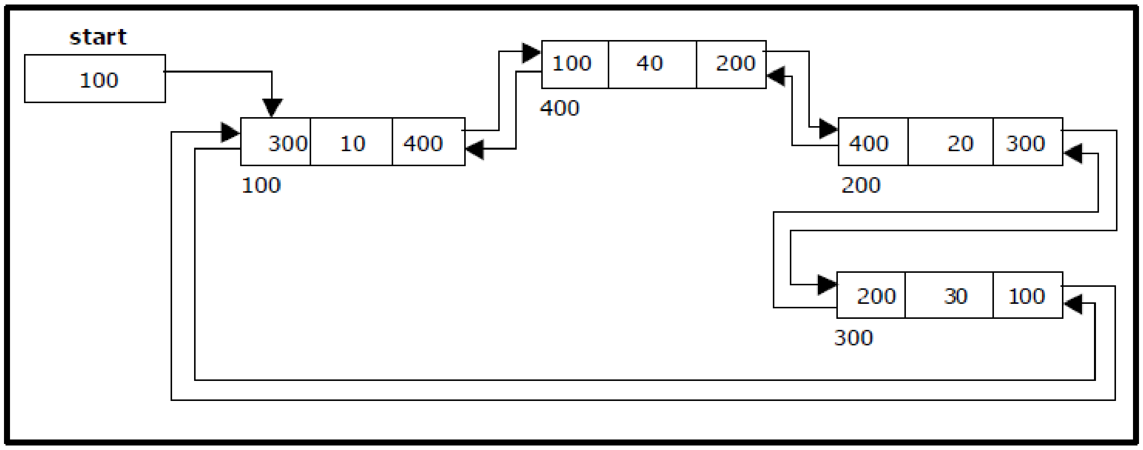


Figure 3.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

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

If list is empty then display 'Empty List' message.

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

```
temp = start;
start = start -> right;
temp -> left -> right = start;
```

```
start -> left = temp -> left;
```

The function cdll_delete_beg(), is used for deleting the first node in the list. Figure 3.8.5 shows deleting a node at the beginning of a circular double linked list.

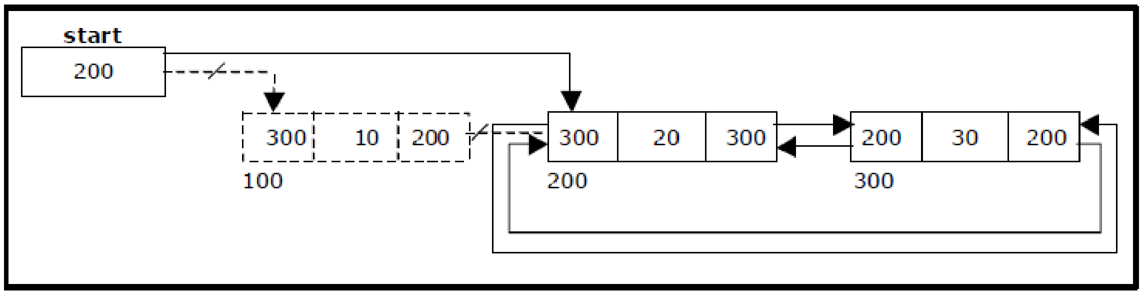


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

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

If list is empty then display 'Empty List' message

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

```
temp = start;
while(temp -> right != start)
{
        temp = temp -> right;
}
temp -> left -> right = temp -> right;
temp -> right -> left = temp -> left;
```

The function cdll_delete_last(), is used for deleting the last node in the list. Figure 3.8.6 shows deleting a node at the end of a circular double linked list.

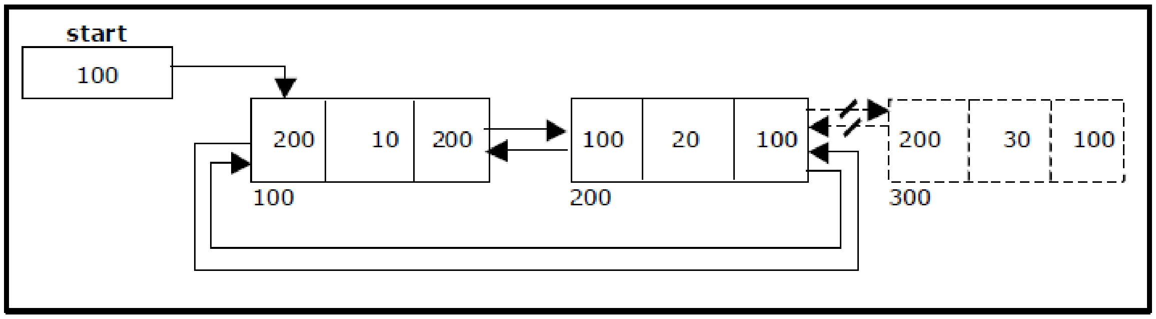


Figure 3.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

```
If list is empty then display 'Empty List' message.
If the list is not empty, follow the steps given below:
    Get the position of the node to delete.

    Ensure that the specified position is in between first node and last node. If
    not, specified position is invalid.
    Then perform the following steps:
        if(pos > 1 && pos < nodectr)
        {

                temp = start;
                i = 1;
                while(i < pos)
                {
                        temp = temp -> right ;
                        i++;
                }
                temp -> right -> left = temp -> left;
                temp -> left -> right = temp -> right;
                free(temp);
                printf("\n node deleted..");
                nodectr--;
        }
```

The function cdll_delete_mid(), is used for deleting the intermediate node in the list. Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

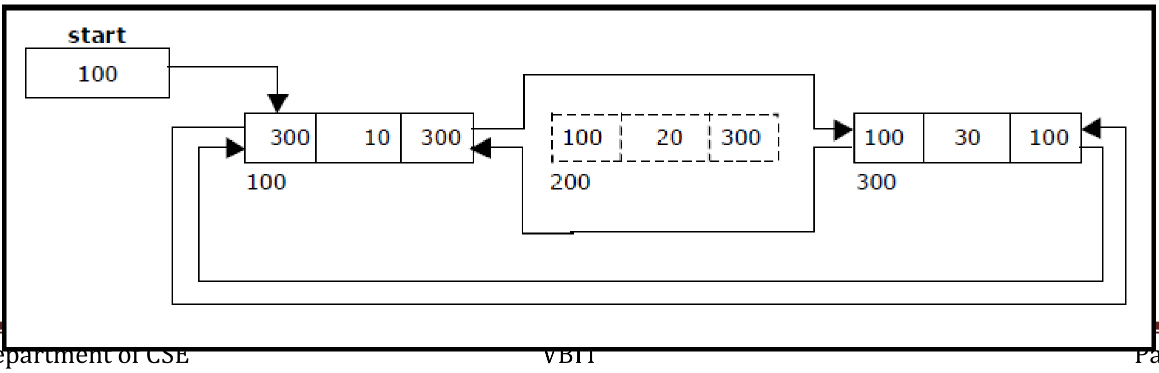


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

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

```
If list is empty then display 'Empty List' message.
If the list is not empty, follow the steps given below: temp
    = start;
    Print temp -> data; temp =
    temp -> right; while(temp
    != start)
    {
        print temp -> data; temp =
        temp -> right;
    }
```

The function cdll_display_left _right(), is used for traversing from left to right.

Traversing a circular double linked list from right to left:

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

```
If list is empty then display 'Empty List' message.

If the list is not empty, follow the steps given below: temp
    = start;
    do
    {
        temp = temp -> left; print
        temp -> data;
    } while(temp != start);
```

The function cdll_display_right_left(), is used for traversing from right to left.

Comparison of Linked List Variations:

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

Sparse Matrix

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Example:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

Sparse Matrix Representations

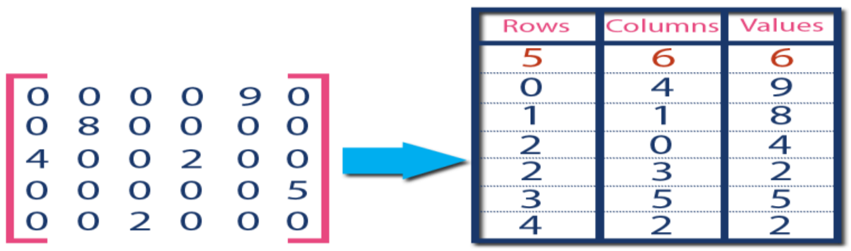
A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

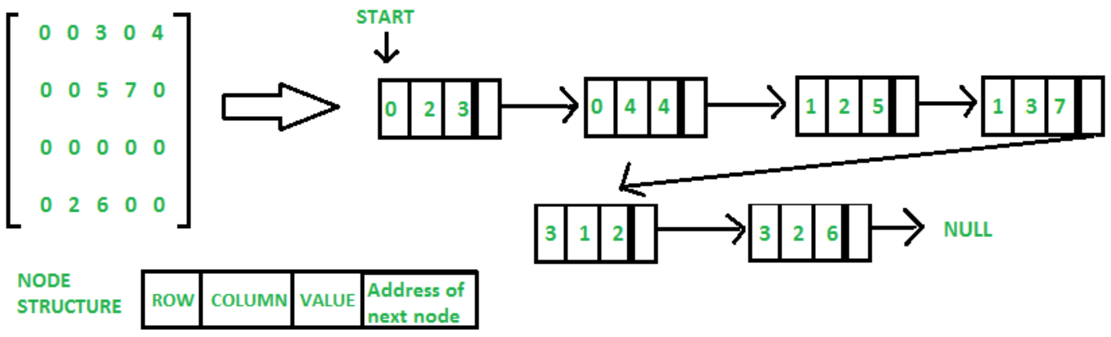


In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

Linked Representation

n 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



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 non-zero elements.
-
- **Computing time:**Computing time can be saved by logically designing a data structure traversing only non-zero elements.