B.Tech. I - II Sem. (CSE)

T	Tu	C
3	1	3

(15A05201) DATA STRUCTURES

(Common to CSE and IT branches of Engineering)

Unit-1:

Introduction and overview: Asymptotic Notations, One Dimensional array- Multi Dimensional array- pointer arrays.

Linked lists: Definition- Single linked list- Circular linked list- Double linked list- Circular Double linked list- Application of linked lists.

CHAPTER-1
INTRODUCTION & OVERVIEW

1.0 Introduction

Efficiency of algorithm is defined in terms of two parameters i.e time and space. Time complexity refers to running time of an algorithm and space complexity refers to the additional space requirement for an algorithm to be executed. Analysis will be focused on running time complexity as response time and computation time is more important as computer speed and memory size has been improved by many orders of magnitude. Time complexity depends on input size of the problem and type of input. Based on the type of data input to an algorithm complexity will be categorized as worst case, average case and best case analysis.

In the last section, linear, quadratic, polynomial and exponential algorithm efficiency will be discussed. It will help to identify that at what rate run time will grow with respect of size of the input

1.1 Objective

- Meaning of Asymptotic notations
- Computation of Worst case, best case and average case analysis of various algorithms
- Comparative analysis of Constant, Logarithmic, Linear, Quadratic and Exponential growth of an algorithm

1.2 Asymptotic Notations

Def: An ***asymptote*** is a line or curve that a graph approaches but does not intersect. An ***asymptote*** of a curve is a line in such a way that distance between curve and line approaches zero towards large values or infinity.

Symbol	Name
=>	Implies
Θ	Theta
Ω	Big Omega
O	Bigoh notation
ω	Small Omega
o	Smalloh notation
Є	Belongs to

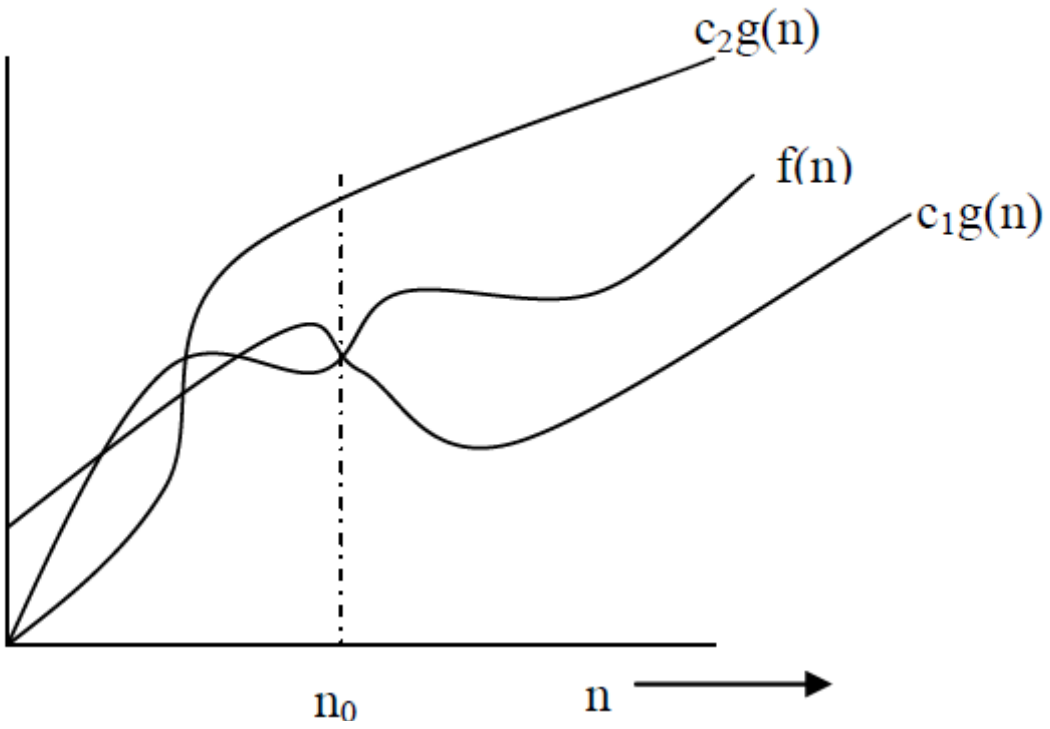
1.2.1 Theta (Θ) Notation :

It provides both upper and lower bounds for a given function.

(Theta) Notation: means `order exactly'. Order exactly implies a function is bounded above and bounded below both. This notation provides both minimum and maximum value for a function. It further gives that an algorithm will take this much of minimum and maximum time that a function can attain for any input size as illustrated in figure 1.

Let g(n) be given function. f(n) be the set of function defined as (g(n)) = {f(n): if there exist positive constant c1,c2 and n0 such that 0≤c1g(n)≤f(n) ≤c2g(n) for all n, n n0} It can be written as f(n)= (g(n)) or f(n) (g(n)), here f(n) is bounded

both above and below by some positive constant multiples of (n) for all large values



of n.

figure 1

Example:

To show that 3n+3 = Θ. (n) we will verify that f(n) g(n) or not with the help of the definition i.e (g(n)) = {f(n): if there exist positive constant c_1,c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, n >=n_0}

In the given problem f(n)= 3n+3 and g(n)=n to prove f(n) g(n) we have to find c_1,c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, n >=n_0

=> to verify f(n) ≤c2g(n)

We can write f(n)=3n+3 as f(n)=3n+3 ≤ 3n+3n (write f(n) in terms of g(n) such that mathematically inequality should be true) ≤6n for all n > 0 c_2=6 for all n > 0 i.e n_0=1 To verify 0≤c_1g(n)≤f(n) We can write f(n)=3n+3 3n (again write f(n) in terms of g(n) such that mathematically inequality should be true) c_1=3 for all n, n_0=1

=> 3n≤3n+3≤6n for all n n_0, n_0=1

i.e we are able to find, c_1=3, c_2=6 n_0=1 such that 0≤c_1g(n)≤f(n) ≤c_2g(n) for all n, n>= n_0 So, f(n)= Θ (g(n)) for all n >=1

1.2.2 Big Oh Notation (O)

This notation provides upper bound for a given function. O(Big Oh) Notation: mean `order at most' i.e bounded above or it will give maximum time required to run the algorithm. For a function having only asymptotic upper bound, Big Oh ‘O’ notation is used.

Let a given function g(n), O(g(n))) is the set of functions f(n) defined as O(g(n))={f(n): if there exist positive constant c and n0 such that 0≤f(n) ≤cg(n) for all n, n n0} f(n)= O(g(n)) or f(n) O(g(n)), f(n) is bounded above by some positive constant multiple of g(n) for all large values of n. The definition is illustrated with the help of figure 2

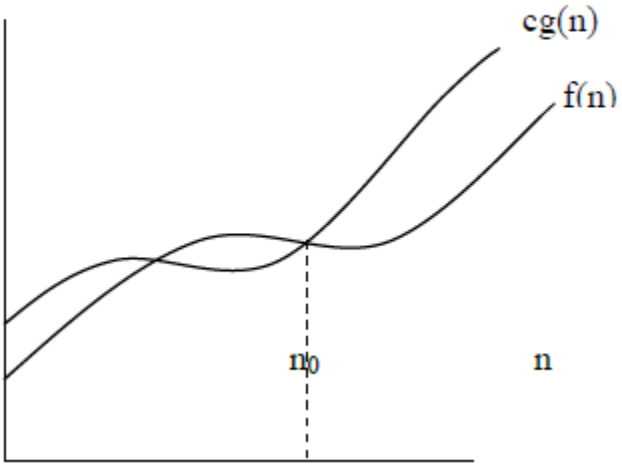


figure 2

Example : To show $3n^2+4n+6=O(n^2)$ we will verify that f(n) g(n) or not with the help of the definition i.e O(g(n))={f(n): if there exist positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all n, n n_0}

In the given problem $f(n)= 3n^2+4n+6$ $g(n)= n^2$ To show $0 \leq f(n) \leq cg(n)$ for all n, $n >= n_0$ $f(n)= 3n^2+4n+6 \leq 3n^2+n^2$ for $n >= 6 \leq 4n^2$

$c=4$ for all $n >= n_0$, $n_0=6$ i.e we can identify , $c=4$, $n_0=6$ So, $f(n)=O(n^2)$

1.2.3 Big Omega Notation (Ω)

This notation provides lower bound for a given function.

(Big Omega): mean 'order at least' i.e minimum time required to execute the algorithm or have lower bound For a function having only asymptotic lower bound, **Ω** notation is used.

Let a given function g(n). **Ω** (g(n))) is the set of functions f(n) defined as **Ω** (g(n)) ={f(n): if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, $n >= n_0$}

f(n)= **Ω** (g(n)) ,f(n) is bounded below by some positive constant multiple of g(n) for all large values of n. It is described in the following figure 3.

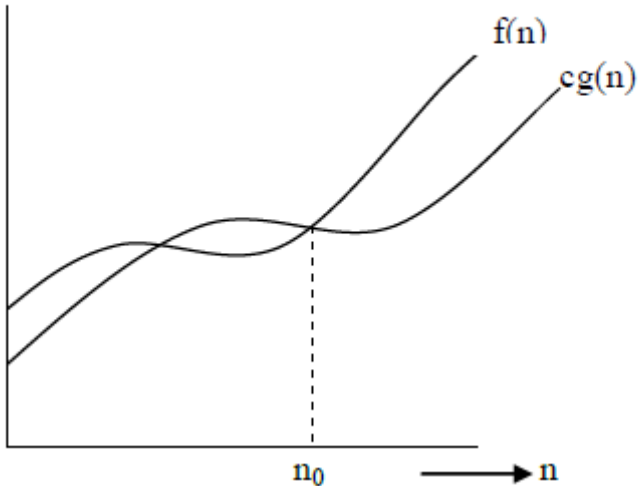


Figure 3

Example :

To show $2n^2+4n+6 = \mathbf{\Omega}(n2)$ we will verify that f(n) Є g(n) or not with the help of the definition i.e $\mathbf{\Omega}(g(n)) = \{f(n)$: if there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n, n $n_0\}$

In the given problem $f(n)= 2n^2+4n+6$ $g(n)= n^2$

To show $0 \leq cg(n) \leq f(n)$ for all n, $n >= n_0$

We can write $f(n)= 2n^2+4n+6$

$0 \leq 2n^2 \leq 2n^2+4n+6$ for $n>=0$

$c=2$ for all $n>=n_0$,

$n_0=0$

i.e we are able to find, $c=2$, $n_0=0$

So, $f(n)= \mathbf{\Omega}(n2)$

1.2.4 Small o Notation (o)

For a function that does not have asymptotic tight upper bound, o (small o) notation is used. i.e. It is used to denote an upper bound that is not asymptotically tight.

Let a given function g(n), o (g(n))) is the set of functions f(n) defined as

o (g(n)) = {f(n): for any positive constant c there exist a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n >= n_0\}$

f(n)= o(g(n)) , f(n) is loosely bounded above by all positive constant multiple of g(n) for all large n. It is illustrated in the following figure 4

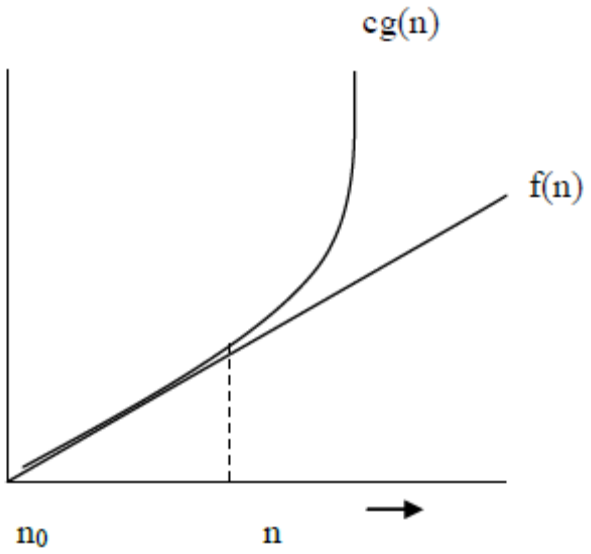


Figure 4

In this figure 4, function f(n) is loosely bounded above by constant c times g(n). We can explain this by following example:

To show $2n+4=o(n^2)$ we will verify that $f(n) \varepsilon g(n)$ or not with the help of the definition i.e $o(g(n)) = \{f(n)$: for any positive constant c there exist a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$

In the given problem

$f(n)=2n+4$, $g(n)= n^2$

To show $0 \leq f(n) < cg(n)$ for all $n \geq n_0$ We can write as
$f(n)=2n+4 < cn^2$
for any $c > 0$, for all $n \geq n_0$, $n_0=1$

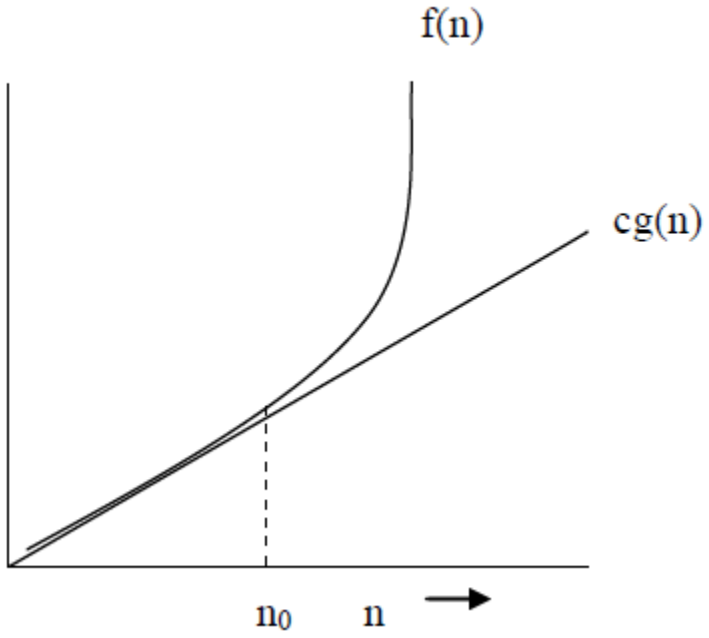
i.e we can find, $c=1$, $n_0=1$

Hence, $f(n)=o(g(n))$

1.2.5 Small Omega Notation (ω)

(Small Omega) Notation: For a function that does not have asymptotic tight lower bound, **ω** notation is used. i.e. It is used to denote a lower bound that is not asymptotically tight.

Let a given function g(n). **ω** (g(n))) is the set of functions f(n) defined as **ω** (g(n)) = {f(n): for any positive constant c > 0 there exist a constant n_0 > 0 such that 0≤ cg(n) < f(n) for all n >=n_0}
f(n)= **ω** (g(n)) ,f(n) is loosely bounded below by all positive constant multiple of g(n) for all large n. It is described in the following figure 5



In this figure function f(n) is loosely bounded below by constant c times g(n). Following example illustrate this notation:

Example:

To show $2n^2+4n+6 = \omega(n)$ we will verify that $f(n) \varepsilon g(n)$ or not with the help of the definition i.e $\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$ there exist a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

In the given problem

$f(n)= 2n^2+4n+6$
$g(n)= n$

To show $0 \leq cg(n) < f(n)$ for all $n \geq n_0$ We can write as

$f(n)= 2n^2+4n+6$

$cn < 2n^2+4n+6$ for any $c > 0$, for all $n \geq n_0$, $n_0=1$
i.e we can find, $c=1$, $n_0=1$

Hence, $f(n)= \omega(g(n))$ i.e $f(n)= \omega(n)$

1.3 Concept of efficiency analysis of algorithm

If we are given an input to an algorithm we can exactly compute the number of steps our algorithm executes. We can also find the count of the processor instructions. Usually, we are interested in identifying the behavior of our program with respect to input supplied to the algorithm. Based on type of input, analysis can be classified as following:

- Worst Case
- Average Case
- Best Case

In the worst case - we need to look at the input data and determine an upper bound on how long it will take to run the program. Analyzing the efficiency of an algorithm in the worst case scenario speaks about how fast the maximum runtime grow when we increase the input size. For example if we would like to sort a list of n

numbers in ascending order and the list is given in descending order. It will lead to worst case scenario for the sorting algorithm.

In average case – we need to look at time required to run the algorithm where all inputs are equally likely. Analyzing the efficiency of an algorithm speaks about probabilistic analysis by which we find expected running time for an algorithm. For example in a list of n numbers to be sorted in ascending order, some numbers may be at their required position and some may be not in order.

In Best case- Input supplied to the algorithm will be almost similar to the format in which output is expected. And we need to compute the running time of an algorithm. This analysis will be referred as best case analysis. For example we would like to sort the list of n numbers in ascending order and the list is already in ascending order.

For example: Consider the linear search algorithm in which we are required to search an element from a given list of elements, let's say size of the list is n. Input: An array of n numbers and an element which is required to be searched in the given list Output: Number exists in the list or not. Algorithm:
1. Input the size of list i.e. n
2. Read the n elements of array A
3. Input the item/element to be searched in the given list.
4. for each element in the array i=1 to n
5. if A[i]==item
6. Search successful, return
7. if i==n+1
8. Search unsuccessful.
9. Stop

Efficiency analysis of the above algorithm in respect of various cases is as follows:
Worst Case: In respect of example under consideration, the worst case is when the element to be searched is either not in the list or found at the end of the list. In this case algorithm runs for longest possible time i.e maximum running time of the algorithm depends on the size of an array so, running time complexity for this case will be O(n).

Average case: In this case expected running time will be computed based on the assumption that probability of occurrence of all possible input is equal i.e array elements could be in any order. This provides average amount of time required to solve a problem of size n. In respect of example under consideration, element could be found at random position in the list. Running time complexity will be O(n).

Best Case: In this the running time will be fastest for given array elements of size n i.e. it gives minimum running time for an algorithm. In respect of example under consideration, element to be searched is found at first position in the list. Running time complexity for this case will be O(1).

1.4 COMPARASION OF EFFICIENCIES OF AN ALGORITHM

(a)	1	Constant Time	When instructions of program are executed once or at most only a few times , then the running time complexity of such algorithm is know as constant time. it is independent of the problem‟s size. It is represented as O(1). For example, linear search best case complexity is O(1)
(b)	log n	Logarithmic	The running time of the algorithm in which large problem is solved by transforming into smaller sizes sub problems is said to be Logarithmic in nature. In this algorithm becomes slightly slower as n grows. It does not process all the data element of input size n. The running time does not double until n increases to n2. It is represented as O(log n). For example binary search algorithm running time complexity is O(log n).
(c)	n	linear	In this the complete set of instruction is executed once for each input i.e input of size n is processed. It is represented as O(n). This is the best option to be used when the whole input has to be processed. In this situation time requirement increases directly with the size of the problem. For example linear search Worst case complexity is O(n).
(d)	n2	Quadratic	Running time of an algorithm is quadratic in nature when it process all pairs of data items. Such algorithm will have two nested loops. For input size n, running time will be O(n2). Practically this is useful for problem with small input size or elementary sorting problems. In this situation time requirement increases fast with the size of the problem. For example insertion sort running time complexity is O(n2).
(e)	2n	Exponential	Running time of an algorithm is exponential in nature if brute force solution is applied to solve a problem. In such algorithm all subset of an n-

			element set is generated. In this situation time requirement increases very fast with the size of the problem. For input size n, running time complexity expression will be O(2n).For example Boolean variable equivalence of n variables running time complexity is O(2n). Another familiar example is Tower of Hanoi problem where running time complexity is O(2n).

For large values of n or as input size n grows, some basic algorithm running time approximation is depicted in following table. As already discussed, worst case analysis is more important hence O Big Oh notation is used to indicate the value of function for analysis of algorithm.

n	**Constant**	**Logarithmic**	**Linear**	**Quadratic**	**Exponential**
	O(1)	**O(log n)**	**O(n)**	**$O(n^2)$**	**$O(2^n)$**
1	1	1	1	1	2
2	1	1	2	4	4
4	1	2	4	16	16
8	1	3	8	64	256
10	1	3	10	10^2	10^3
10^2	1	6	10^2	10^4	10^{30}
10^3	1	9	10^3	10^6	10^{301}
10^4	1	13	10^4	10^8	10^{3010}

The running time of an algorithm is most likely to be some constant multiplied by one of above function plus some smaller terms. Smaller terms will be negligible as input size n grows. Comparison given in above table has great significance for analysis of algorithm.

ONE-DIMENSIONAL ARRAY:

An array is a collection of variables of the same type that are referenced by a common name. In C, all arrays consists of contiguous memory locations. The lowest address corresponds to the first element, and the highest address to the last element. Arrays may have from one to several dimensions. A specific element in an array is accessed by an index.

One Dimensional Array:

The general form of single-dimension array declaration is:

Type variable-name[size];
Here, type declares the base type of the array, size defines how many elements the array will hold.
For example, the following declares as integer array named sample that is ten elements long
int sample[10];
In C, all arrays have zero as the index of their f irst element. This declares an integer array that has ten elements, sample[0] through sample[9]

Example 1:

```
/* To find the average of 10 numbers */
# include <stdio.h>
main()
{
int i, avg, sample[10];
for (i=0; i<10; i++)
{
printf ("\nEnter number: %d ", i);
scanf ("%d", &sample[i]);
}
avg = 0;
for (i=0; i<10; i++)
avg = avg + sample[i];
printf ("\nThe average is: %d\n", avg/10);
}
```

Two-dimensional arrays:

To declare two-dimensional integer array num of size (3,4),we write:

Two dimensional arrays are stored in a row-column matrix where the first index indicates the row and the second indicates the column. This means that the right most index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory.

Num[t][I] → / ↓	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Example:

```
main ()
{
int t, i, num [3][4];
for (t=0; t<3; t++)
for (i=0; i<4; ++i)
num [t][i] = (t * 4) + i + 1;
for (t=0; t<3; t++)
{
for (i=0; i<4; ++i)
printf ("%3d", num[t][i]);
printf ("\n");
}
}
```

the graphic representation of a two-dimensional array in memory is:

byte = sizeof 1st Index * sizeof 2nd Index * sizeof (base type)
Size of the base type can be obtained by using **size of operation**.
returns the size of memory (in terms of bytes) required to store an integer object.
sizeof (unsigned short) = 2
sizeof (int) = 4
sizeof (double) = 8
sizeof (float) = 4
assuming 2 byte integers as integer with dimension 4, 3 would have

4 * 3 * 2 = 24 bytes

Given: char ch[4][3]

Ch [0][0]	Ch [0][1]	Ch [0][2]
Ch [1][0]	Ch [1][1]	Ch [1][2]
Ch [2][0]	Ch [2][1]	Ch [2][2]
Ch [3][0]	Ch [3][1]	Ch [3][2]

N - dimensional array or multi dimensional array:
This type of array has n size of rows, columns and spaces and so on. The syntax used for declaration of this type of array is as follows:
Data type array name[s1] [s 2] … … … [sn];
In this sn is the nth size of the array.

Array Initialization:
The general form of array initialization is:
Type_spec ifier array_name[size 1].... [size N] = { value _list};

The value list is a comma_separated list of constants whose type is compatible with type_specifier.

Example 1:

10 element integer array is initialized with the numbers 1 through 10 as:

int I[10] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

i.e., I[0] will have the value 1 and

…….

…….

I[9] will have the value 10

Character arrays that hold strings allow a shorthand initialization that takes the form:

c har array _name[s ize] = “string” ;

POINTER-ARRAYS:

4.1.1. Pointers and Arrays:

There is a close association between pointers and arrays. Let us consider the following statements:

int x[5] = {11, 22, 33, 44, 55};
int *p = x;

The array initialization statement is familiar to us. The second statement, array name x is the starting address of the array. Let we take a sample memory map as shown in figure 4.2.

From the figure 4.2 we can see that the starting address of the array is 1000. When x is an array, it also represents an address and so there is no need to use the (&) symbol before x. We can write int *p = x in place of writing int *p = &x[0].

The content of p is 1000 (see the memory map given below). To access the value in x[0] by using pointers, the indirection operator * with its pointer variable p by the notation *p can be used

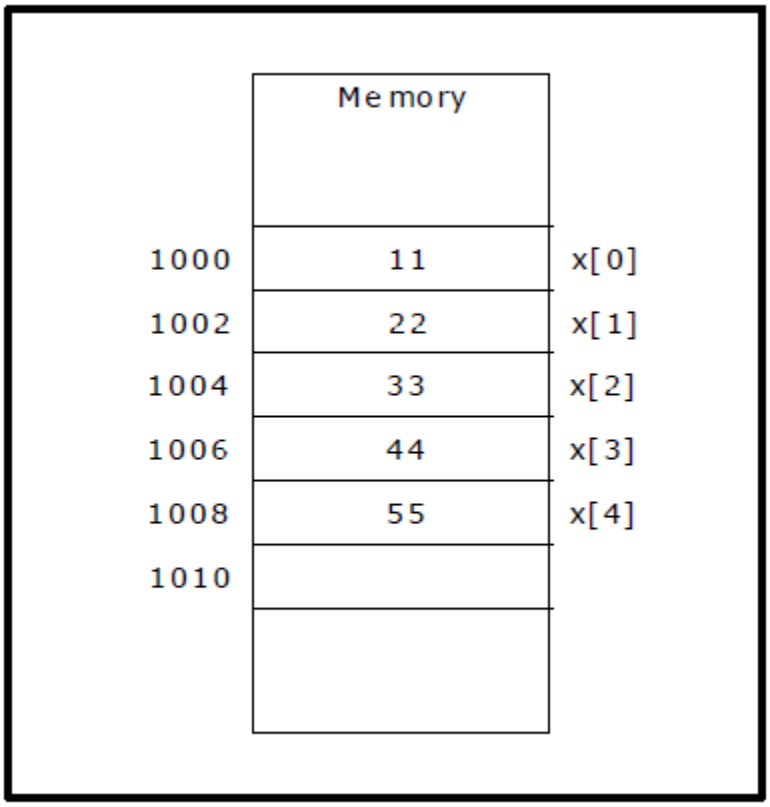


Figure 4.2. Memory map - Arrays

The increment operator ++ helps you to increment the value of the pointer variable by the size of the data type it points to. Therefore, the expression p++ will increment p by 2 bytes (as p points to an integer) and new value in p will be 1000 + 2 = 1002, now *p will get you 22 which is x[1].
Consider the following expressions:
*p++;
*(p++); (*p)++;
How would they be evaluated when the integers 10 & 20 are stored at addresses 1000 and 1002 respectively with p set to 1000.

p++ : The increment ++ operator has a higher priority than the indirection operator * . Therefore p is increment first. The new value in p is then 1002 and the content at this address is 20.

*(p++): is same as *p++.
(*p)++: *p which is content at address 1000 (i.e. 10) is incremented. Therefore (*p) ++ is 11.
Note that, *p++ = content at incremented address.

Example:
#include <stdio.h>

```
main()
{
int x[5] = {11, 22, 33, 44, 55};
int *p = x, i; /* p=&x[0] = address of the first element */
for (i = 0; i < 5; i++)
{
printf (“\n x[%d] = %d”, i, *p); /* increment the address*/
p++;
}
}
```

Output:

x [0] = 11 x [1] = 22 x [2] = 33
x [3] = 44 x [4] = 55

The meanings of the expressions p, p+1, p+2, p+3, p+4 and the expressions *p, *(p+1), *(p+2), *(p+3), *(p+4) are as follows:

P = 1000	*p = content at address 1000 = x[0]
P+1 = 1000 + 1 x 2 = 1002	*(p+1) = content at address 1002 = x[1]
P+2 = 1000 + 2 x 2 = 1004	*(p+2) = content at address 1004 = x[2]
P+3 = 1000 + 3 x 2 = 1006	*(p+3) = content at address 1006 = x[3]
P+4 = 1000 + 4 x 2 = 1008	*(p+4) = content at address 1008 = x[4]

CHAPTER-2:LINKED LISTS

Introduction to Linked List:

A **linked list** is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**.Each **node** is divided into two parts:
1. The first part contains the **information** of the element and
2. The second part contains the address of the next node (**link /next pointer field**) in 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.

REPRESENTATION OF SINGLE LINKED LIST:

In static representation (arrays), the memory is allotted as per the specification and it is fixed. It is not possible to allocate additional memory or to delete memory as needed. The advantage of dynamic representation is allocation and de-allocation of memory can be done as needed using pointers. It is not only used for efficient memory management but also for faster processing of data.

Operations on linked lists

The most common operations performed on Linked list are

- *Checking* whether the list is empty
- *Traversing* the process or visit all elements of list
- Determining the *size* (i.e., the number of elements) of the list;
- *Modifying* the content of the node
- *Inserting a node to list*
- *Removing* a specific node from list

Node creation in single linked list:

A node of single linked list is created using **self-referential structure.** A Self-referential structure is a structure which includes at least one member that is a pointer to the same structure type

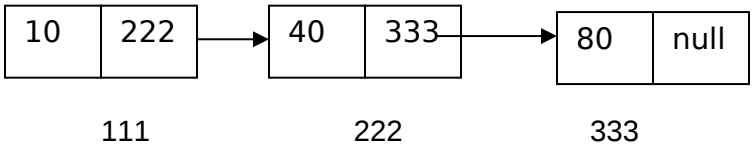
Node declaration in C language:

```
struct node
{
```

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

In the above example **next* is a self referential because it is referencing to struct *node* type.

Linked List with three nodes



checking list is empty or not: If the address of the head node is null, then the list is said to be empty

Algorithm to check node is empty

Step 1: Let r= head
Step 2: if r=NULL then Display List is empty and execute Step 5
Step 3: Confirm List is not empty
Step 4: Stop

TRAVERSING

To display the elements of an existing linked list, opt the following procedure

- If list is empty then return NULL
- Starting from the head node of the list the elements are displayed in sequence one after the other up to last node. The last node of linked list pointer filed (or) address filed contains NULL value.

Algorithm to traverse the list:

Step 1: Let r=head
Step 2: While r <> NULL
Step 3: Display the data of r
Step 4: r= r-> next
Step 5: End while
Step 6:Stop

COUNTING NUMBER OF ELEMENTS

For counting the elements of an existing linked list, opt the following procedure

- If list is empty then NULL is returned
- Counter variable is initially initialized with zero
- Starting from the head node of the list, the elements are counted in sequence one after the other up to last node
- The last node of linked list pointer filed (or) address filed contains NULL value.
- Display the value of counter variable

Algorithm to count the number of elements in the list:

Step 1: Let r=head: Let count = 0
Step 2: While r <> NULL
Step 3: Display the data of r
Step 4: r= r-> next
Step 5: count = count + 1
Step 6: End while
Step 7: Display the count value
Step 8: stop

Modifying the content of node: In order to modify the content of a node, search for the node based on its content and then update it.

Algorithm to Update a node:
Step 1: Accept existing value ‘ e’ and a new value ‘n’ to update node
Step 2: Let r=head
Step 3: Traverse the list, till the value of node = e
Step 4: If found, then update the value of e with n
Step 5: Stop

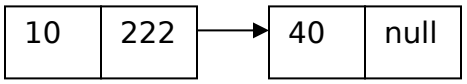
Inserting an Element to Single Linked List:

Appending means adding an element at the end of list. Add new element after the last node, such that the newly added node becomes the last node of the list.

Adding of an element can be done in 3 ways
1. Adding an element before first node
2. Adding an element after last node
3. Adding an element in particular position

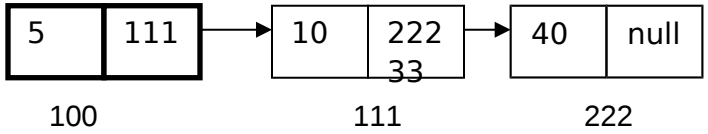
Initially consider the single linked list as below

Linked list

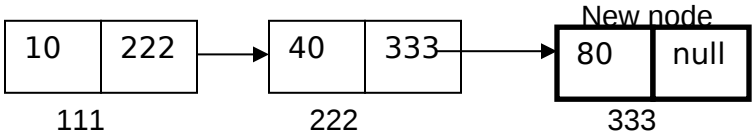


Insertion of node at First

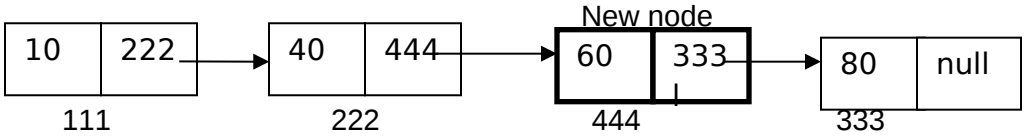
New node



Insertion of node at last:



Inserting node with data 60 in between 40 and 80



Algorithm to Add node before first node

Step 1: Let n be the new node with data
Step 2: Let n->next = first node (header node)
Step 3: let first node be n i.e header node

Algorithm to Add node after last node

Step 1: Let n be the new node with data
Step 2: Let last->next = n (new node)
Step 3: let last node be n i.e new node
Step 4: Stop

Algorithm to Add node after required position

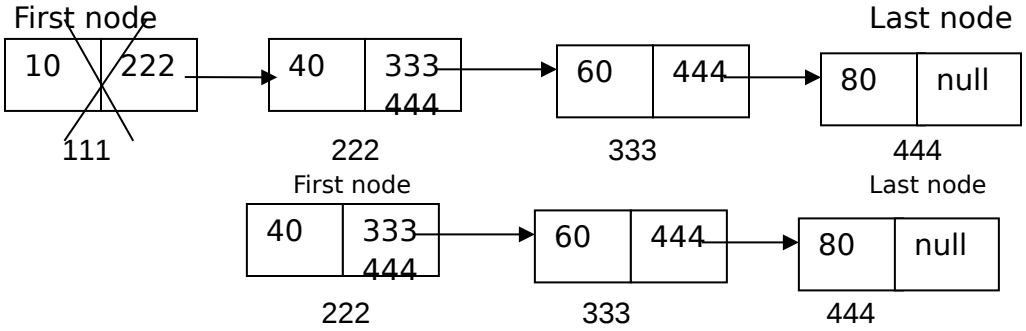
Step 1: Let n be the new node with data
Step 2: Accept the position of insertion into 'pos'
Step 3: let count =1: r= first
Step 4: While count <= pos
Step 5: count=cpunt+1: r= r->next
Step 6: End While
Step 7: n->next = r-> next
Step 8: r->next = n

Deleting an Element from Single Linked List:

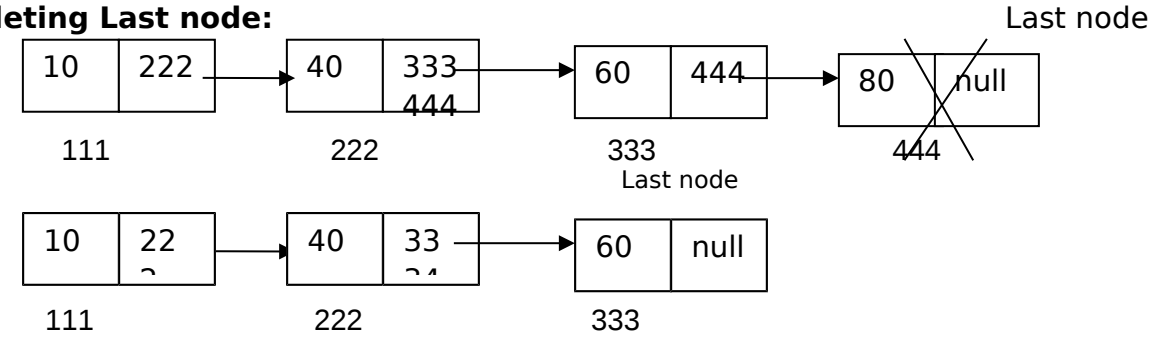
Deletion operation in single linked list is of 3 ways

(a) Deleting header node
(b) Deleting last node
(c) Deleting a particular node

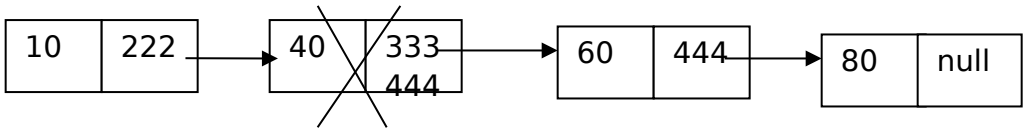
Deleting starting node:

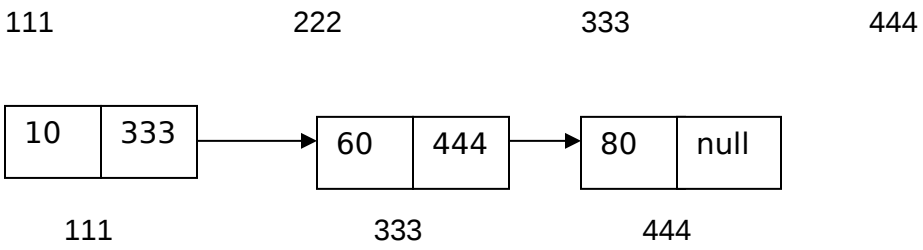


Deleting Last node:



Deleting node with data 40





Algorithm to delete header node

Step 1: Let r=head; /* r contains header node information */
Step 2: Let r1=head->next; /* r1 contains second node information */
Step 3: Delete node r
Step 4: head=r1;
Step 5: Stop

Algorithm to delete last node

Step 1: Let r=head; /* r contains header node information */
Step 2:While r ->next ->next <> last node
Step 3:r=r->next
Step 4:End While
Step 5:r=last;
Step 6:Delete last node
Step 7:r->next=NULL
Step 8:Stop

Algorithm to delete a Particular node

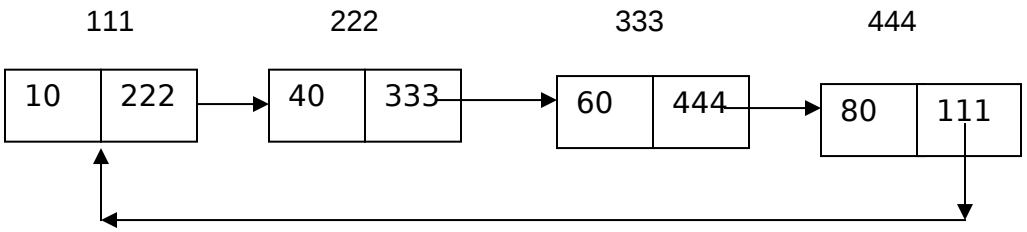
Step 1: Let r=head; /* r contains header node information */
Step 2:Let s the data of a node to delete
Step 3:While r ->next ->datat <> s
Step 4:r=r->next;
Step 5:End While
Step 6:if r==NULL
Step 7:Display node not found
Step 8:else
Step 9:K=r->next
Step 9: r->next=k->next
Step 10: r=r->next ;
Step 11:Delete r
Step 11:Stop

CIRCULAR LINKED LIST

- In linear linked lists if a list is traversed (all the elements visited) an external pointer to the list must be preserved in order to be able to reference the list again.

- A circular list is very similar to the linear list where in the circular list the pointer of the last node points to the first node.
- Circular linked lists can helps to traverse the same list again and again if needed.

Example



Operations on linked lists:

The most common operations performed on Circular Linked list are

- *Checking* whether the list is empty;
- *Traversing* the process or visit all elements of list
- Determining the *size* (i.e., the number of elements) of the list
- *Inserting a node to list*
- *Removing* a specific node from list

Node creation in circular linked list:

The node creation in circular linked list is same as that of single linked list.

Checking list is empty or not: If the address of the head node is null, then the list is said to be empty

Algorithm to check node is empty

Step 1: Let r= head
Step 2: if r=NULL then Display List is empty and execute Step 5
Step 3: Confirm List is not empty
Step 4: Stop

TRAVERSING

To display the elements of an existing circular linked list, opt the following procedure

- If list is empty then return NULL
- Starting from the head node of the list, the elements are displayed in sequence one after the other upto the last node. The last node of linked list points in turn to fist node.

Algorithm to traverse the list:

Step 1: Let r=head
Step 2: do
Step 3: Display the data of r
Step 4: r= r-> next
Step 5: while r<>head
Step 6:Stop

COUNTING NUMBER OF ELEMENTS

For counting the elements of an existing linked list, opt the following procedure

- If list is empty then NULL is returned
- Counter variable is initially initialized with zero

- Starting from the head node of the list, the elements are counted in sequence one after the other up to last node
- The last node of linked list pointer filed (or) address filed contains NULL value.
- Display the value of counter variable

Algorithm to count the number of elements in the list:

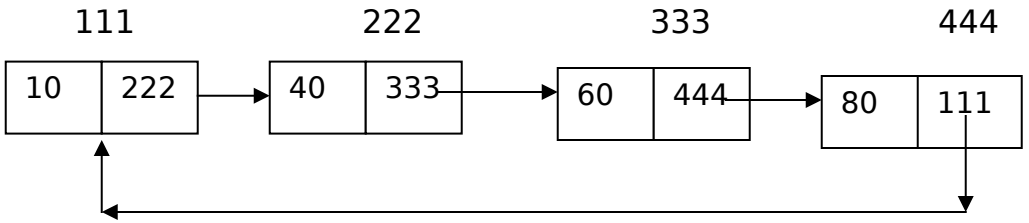
Step 1: Let r=head: Let count = 0
Step 2:do
Step 3: Display the data of r
Step 4: r= r-> next
Step 5: count = count + 1
Step 6: while r <> head
Step 7: Display the count value
Step 8: stop

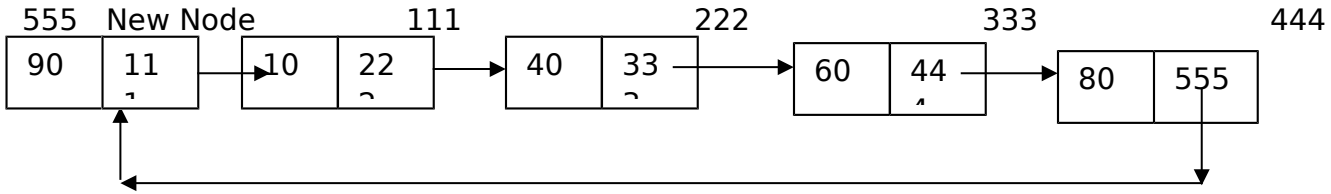
Inserting An Element To A circular Linked List:

Adding of an element can be done in 3 ways

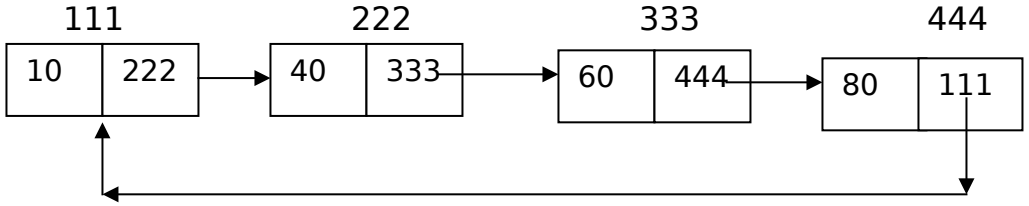
1. Adding an element before first node
2. Adding an element after last node
3. Adding an element in a particular position

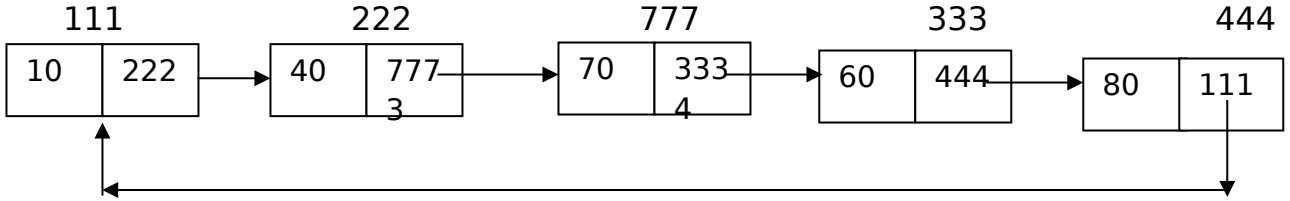
Insertion at the front of single circular linked list:



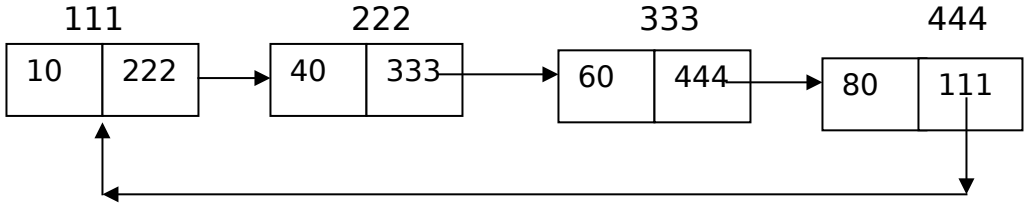


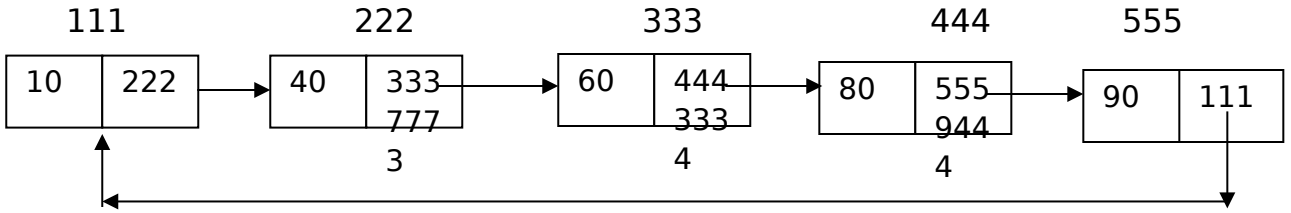
Insertion node with 70 at the middle of single circular linked list:





Insertion 90 at the end of single circular linked list:





Algorithm to Add node before first node

Step 1: Let n be the new node with data
Step 2: Let n->next = first node (header node)
Step 3: let first node be n i.e header node
Step 4:last->next=n;(header node)
Step 4: Stop

Algorithm to Add node after last node

Step 1: Let n be the new node with data
Step 2: Let last->next = n (new node)

Algorithm to Add node after required position

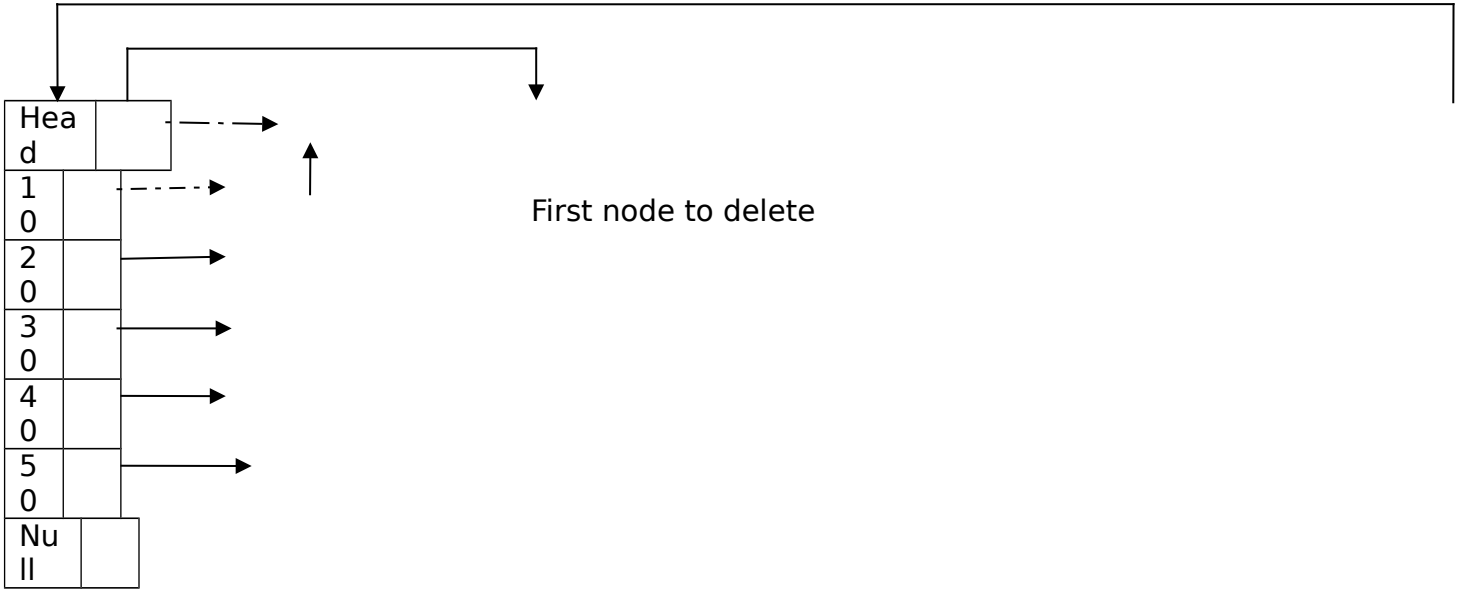
Step 1: Let n be the new node with data
Step 2: Accept the position of insertion into ‘pos’
Step 3: let count =1: r= first
Step 4: While count <= pos
Step 5: count=count + 1: r= r->next
Step 6: End While
Step 7:n->next=r->next
Step 8: r->next = n

DELETING AN ELEMENT FROM A CIRCULAR LINKED LIST:

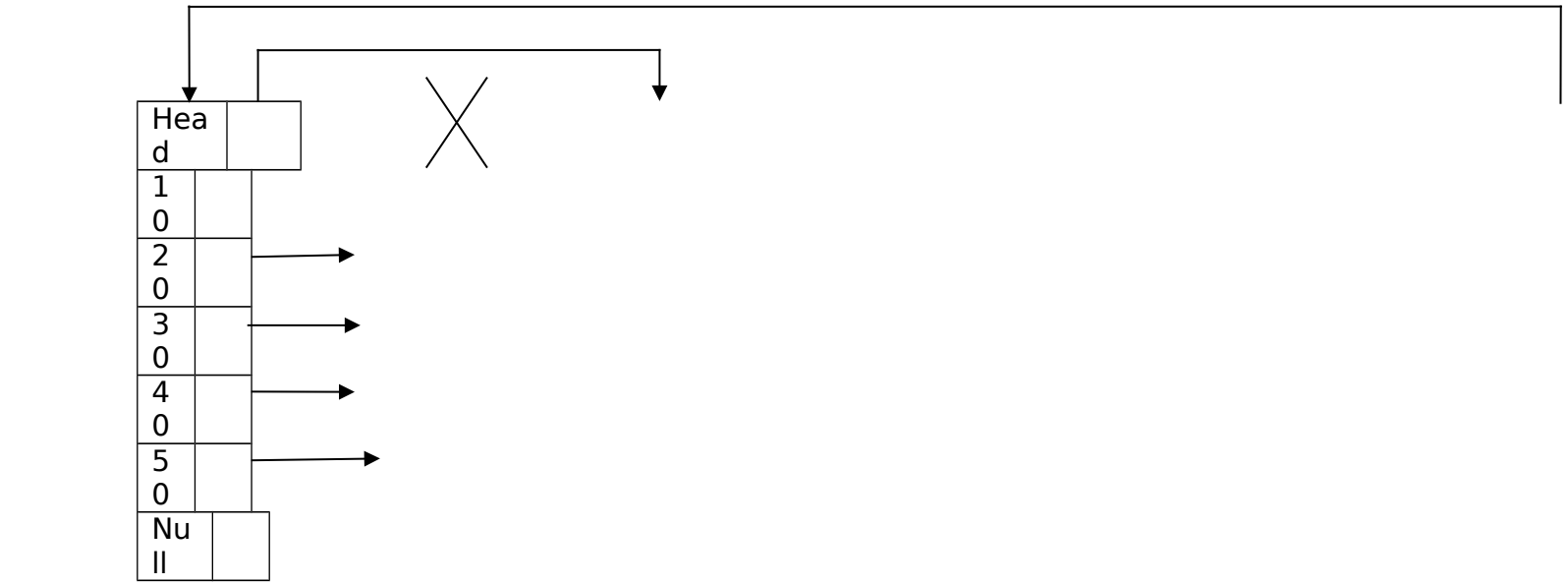
Deletion operation in circular linked list is of 3 ways

(a) Deleting header node
(b) Deleting last node
(c) Deleting a particular node

Deletion at the first node of single circular linked list:

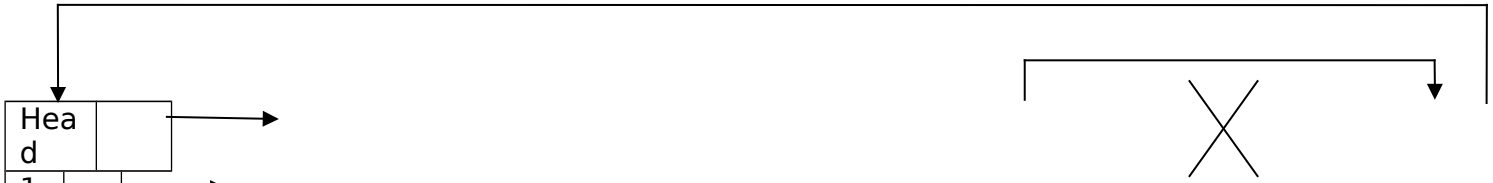


After deletion:



Deletion at the end of single circular linked list:

After deletion:



Algorithm to delete header node

Step 1: Let r=head; /* r contains header node information */
Step 2: Let r1=head->next; /* r1 contains second node information */
Step 3: Delete node r
Step 4:last->next=r1; /* Second node of list */
Step 5: head=r1;
Step 6: Stop

Algorithm to delete last node

Step 1: Let r=head; /* r contains header node information */
Step 2:While r ->next ->next <> last node
Step 3:r=r->next
Step 4:End While
Step 5:r=last;
Step 6:Delete last node
Step 7:r->next=first /* last but one node becomes last and it points to first node such that list becomes circular */
Step 8:Stop

Algorithm to delete a Particular node

Step 1: Let r=head; /* r contains header node information */
Step 2:Let s the data of a node to delete
Step 3:While r ->next ->datat <> s
Step 4:r=r->next;
Step 5:End While
Step 6:if r==NULL
Step 7:Display node not found
Step 8:else
Step 9:K=r->next
Step 9: r->next=k->next
Step 10: r=r->next ;
Step 11:Delete r
Step 11:Stop

DOUBLE LINKED LIST:

A doubly linked list is a list that contains links to next and previous nodes. In single linked list traversing can be done in one way, whereas double linked list allows traversing in both ways, that is in forward direction as well as backward direction.

Operations on linked lists:

The most common operations performed on Double Linked list are

- *Checking whether the list is empty*
- *Traversing i.e., the process of visiting all elements of list(in both directions)*
- *Determining the size (i.e., the number of elements) of the list;*
- *Inserting a node to list*
- *Removing* a specific node from list

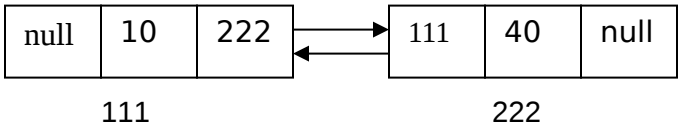
Node creation in Double linked list:

A node of double linked list is created using **self-referential structure.** A Self-referential structure is a structure which includes at least two members that are pointers to the same structure type. One pointer pointing to the previous node and another pointer pointing to the next node.

Node declaration in C language

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

Double linked list



checking list is empty or not: If the address of the head node is null, then the list is said to be empty

Algorithm to check list is empty

Step 1: Let r= head->next and s=head->prev
Step 2: if r== NULL and s==NULL then Display list is empty and execute Step 4
Step 3: Confirm List is not empty
Step 4: Stop

TRAVERSING

To display the elements of an existing double linked list, opt the following procedure

- If list is empty then return NULL
- Starting from the head node of list the elements are displayed in sequence one after the other up to last node.

TRAVERSING THE LIST THROUGH FORWARD POINTER

Algorithm to traverse the list:

Step 1: Let r=head->next
Step 2: While r <> NULL
Step 3: Display the data of r
Step 4: r= r-> next
Step 5: End while
Step 6:Stop

TRAVERSING THE LIST THROUGH BACKWARD POINTER

Algorithm to traverse the list:

Step 1:let r=last node;
Step 2:While r->prev<> NULL
Step 3:Display the data of r
Step 4:r=r->prev

COUNTING NUMBER OF ELEMENTS

For counting the elements of an existing linked list, opt the following procedure

- If list is empty then NULL is returned
- Counter variable is initialized to zero
- Starting from the head node of the list, the elements are counted in sequence one after the other up to last node
- Display the value of counter variable

COUNTING NUMBER OF ELEMENTS THROUGH FORWARD POINTER

Algorithm to count the number of elements in the list:

Step 1: Let r=head: Let count = 0
Step 2: While r ->next<> NULL
Step 3: Display the data of r
Step 4: r= r-> next
Step 5: count = count + 1
Step 6: End while
Step 7: Display the count value
Step 8: stop

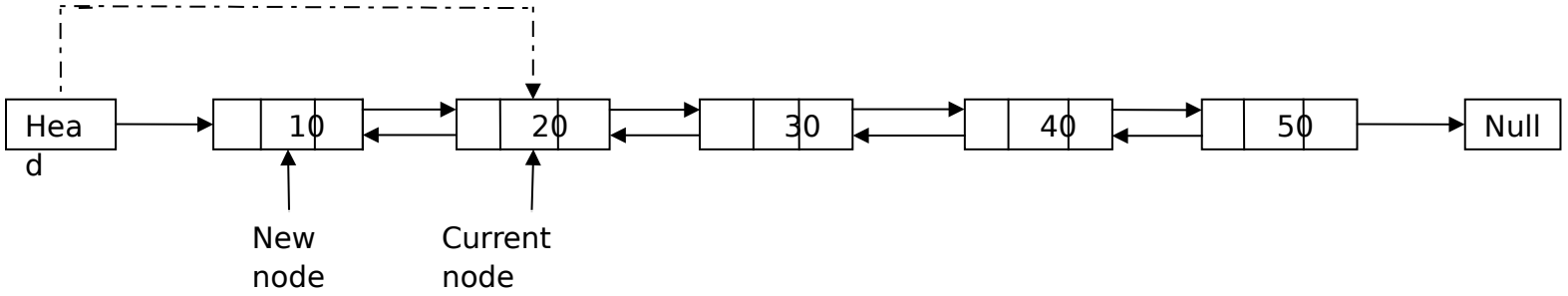
Note: In the Similar manner counting of elements can also be done through backward pointer

Inserting an Element to Double Linked List:

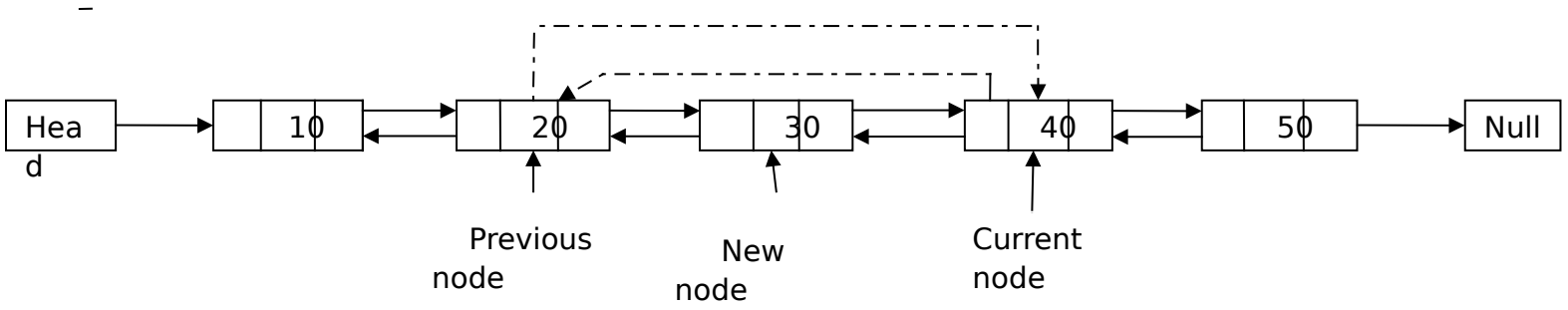
Adding of an element can be done in 3 ways

1. Adding an element before first node
2. Adding an element after last node
3. Adding an element in a particular position

Insertion at the front of double linked list:

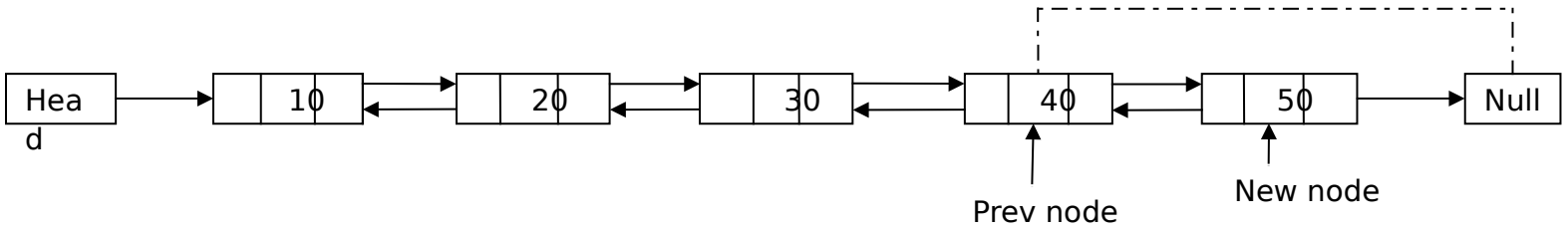


Insertion at the middle of double linked list:



Insertion at the end of double linked list:

–



Algorithm to Add node before first node

Step 1: Let n be the new node with data
Step 2: Let n->next = first node (header node)
Step 3:n->prev=NULL;
Step 4: let first node be n i.e header node
Step 5:Stop

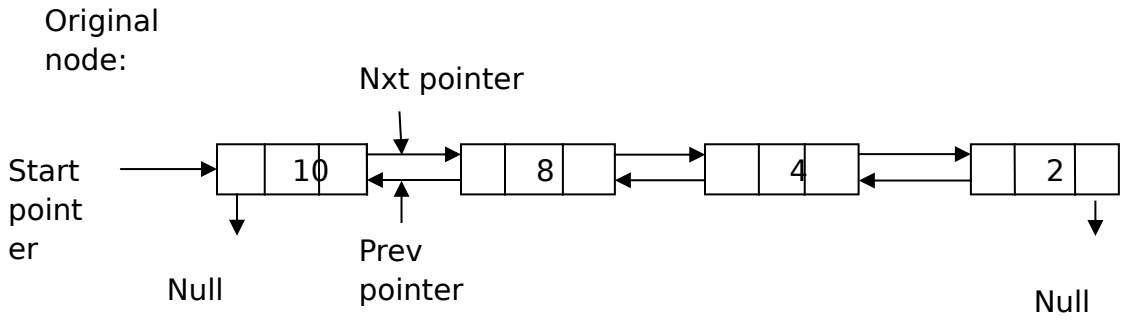
Algorithm to Add node at required position

Step 1: Let n be the new node with data
Step 2: Accept the position of insertion into ‘pos’
Step 3: let count =1: r= first
Step 4: While i <= pos
Step 5: i = i + 1:   r= r->next
Step 6: EndWhile
Step 7: n->next = r;
Step 8:n->prev=r->prev
Step 9: Stop

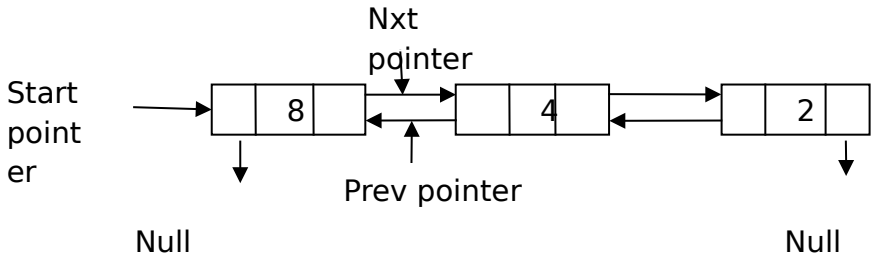
DELETING AN ELEMENT FROM A DOUBLE LINKED LIST:

Deletion operation in Double linked list is of 3 ways

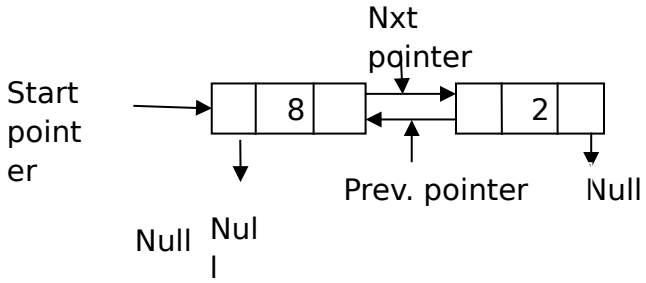
(a) Deleting header node
(b) Deleting last node
(c) Deleting a particular node



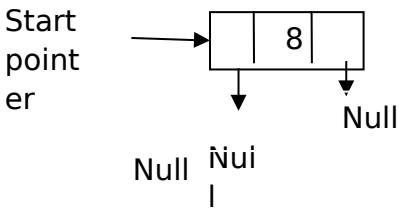
Deleting the head node in double linked list:



Deleting the middle node in double linked list:



Deleting the last node in double linked list:



Algorithm to delete header node
Step 1: Let r=head; /* r contains header node information */ Step 2: Let r1=head->next; /* r1 contains second node information */ Step 3:r1->prev=NULL Step 4: Delete node r Step 5: head=r1;
Algorithm to delete last node
Step 1: Let r=head; /* r contains header node information */ Step 2:While r ->next ->next <> last node Step 3:r=r->next Step 4:End While Step 5:Delete the last node Step 6:last=r; Step 7:last->next=NULL Step 8:Stop

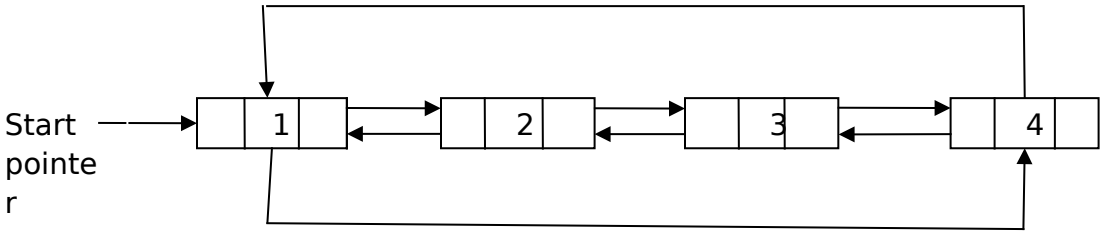
Algorithm to delete a Particular node

Step 1: Let r=head; /* r contains header node information */
Step 2:Let s the data of a node to delete
Step 3:While r ->data <> s
Step 4:r=r->next;
Step 5:End While
Step 6:if r==NULL
Step 7:Display node not found
Step 8:else
Step 9:r->prev->next=r->next
Step 10:r->next->prev=r->prev
Step 11:Delete r
Step 12:Stop

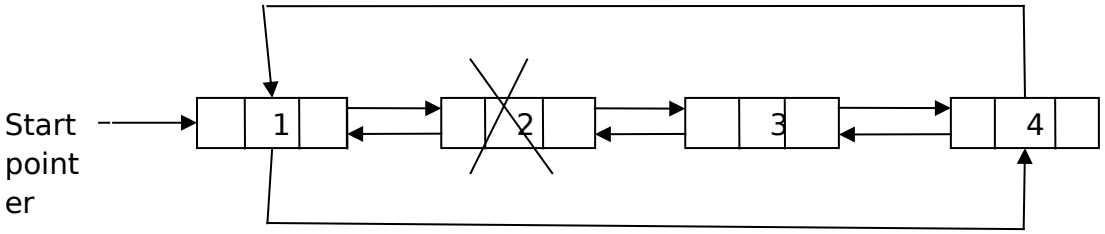
CIRCULAR DOUBLE LINKED LIST:

Doubly Circular linked list has both the properties of doubly linked list and circular linked list. Two consecutive elements are linked by previous and next pointer and the last node points to first node by next pointer and also the previous pointer of the head node points to the tail node

Double circular linked list:



Deleting Double circular linked list:



APPLICATIONS OF LINKED LISTS:

- Linear data structures such as stacks and queues are easily executed with a linked list.
- They reduce access time and may expand in real time without memory overhead.

- Circular linked list are used is a timesharing problem solved by the operating system.
  In a timesharing environment, the operating system must maintain a list of present users and must alternately allow each user to use a small slice of CPU time, one user at a time. The operating system will pick a user, allots CPU time and then move on to the next user.

- Applications of Doubly linked list can be
  - A great way to represent a deck of cards in a game.
  - The browser cache which allows you to hit the BACK button (a linked list of URLs)
  - Applications that have a Most Recently Used (MRU) list
  - A stack, hash table, and binary tree can be implemented using a doubly linked list.

ASSIGNMENT QUESTIONS:

- Given a circular linked list, how to find the longest sequence of no-repeated-value nodes?
- What is the difference between a de-que and a doubly linked list?

- Defend that pointer based linked list is better than Array based linked list.

- How to convert a doubly circular linked list into a singly circular linked list?