UNIT – I : HandOut

Algorithm: A step by step process to perform a particular task.

Algorithm Features: Input/Output, Finiteness, definiteness, Effectiveness, Generality

Performance Analysis: Determining an estimate of the time and memory requirement of the algorithm.

- Time estimation is called time complexity analysis
- Memory size estimation is called space complexity analysis.

Notations used to represent Time Complexity:

- **Big-O Notation** : f(n)=**O(g(n))** iff there exist a positive constant C and non-negative integer n_0 such that f(n) □ Cg(n) for all n□n0.
- **Omega Notation:** f(n) = □ **(g(n))** iff there exist a positive constant C and non-negative integer n_0 such that f(n) ≥ Cg(n) for all n> n_0.
- **Theta Notation:** □*(g) = O(g)* □□□□*(g) :* function *f* is bounded both from above and below by the same function *g*
  - *F(n) =* □*(g) iff* $c_1 g(n) < f(n) < c_2 g(n)$ for all *n>N*

Recursion: Process of calling a function by itself.

Algorithm fact(n)	Recursive Alg.: gcd(a,b)	Recursive Alg.: fib(n)
{	begin	begin
if((n==0) || (n==1)	if(b==0)	if (n <= 1)
return 1;	return a;	return n;
else	else	else
return(n*fact(n-1));	return gcd(b, a mod b)	return (fib(n-1) + fib(n-2));
}	end	end

Recursion Types: Direct recursion, Indirect recursion.

Linear Recursion, Binary Recursion, Tail Recursion

UNIT - I

Algorithm: Algorithm is a step by step process used to perform a particular task. Algorithm can be writen in General English i.e not by using any specific computer programming language.

Eg: Algorithm: To perform the addition of two numbers

Step 1: Start

Step 2: Read the values of a,b

Step 3: c = a + b

Step 4: print c

Step 5: Stop

Algorithm Features: The various characteristics of an algorithm are,

Input/Output, Finiteness, definiteness, Effectiveness, Generality

Input/Output : Algorithm should consist zero or more but only finite number of inputs and atleast one output.

Finiteness: An algorithm must terminate after a finite number of steps and further each step must be executable in finite amount of time that it terminates (in finite number of steps) on all allowed inputs.

Definiteness: Every step should be precisely defined means each and every step of the algorithm should be simple and clear i.e. easy to understand without any ambiguity.

Effectiveness: Each step of the Algorithm must be feasible i.e., it should be practically possible to perform the action. In other words, algorithm can be converted into code.

Performance Analysis: Determining an estimate of the time and memory requirement of the algorithm.

- Time estimation is called time complexity analysis
- Memory size estimation is called space complexity analysis.

Space Complexity:

Space Complexity of an algorithm is the amount of memory it needs to run to completion.
The space needed by any algorithm is the sum of the following components.

1. A **fixed part**, that is independent of the characteristics of inputs and outputs. This part includes space for instructions(code), space for simple variables, & fixed size component variables, space for constants etc.
2. A **variable part**, which consists of space needed by component variables whose size, is dependent on the particular problem instance being solved, space for reference variables and recursion stack space etc.

The Space requirement *S(P)* of an algorithm *P* may be written as
$S(P) = c + S_P(\text{instance characteristics})$ where *'c'* is a constant.

When analyzing the space complexity of an algorithm first we estimate $S_P(\text{instance characteristics})$. For any given problem, we need to determine which instance characteristics to use to measure the space requirements.

Example 1:

```
1    Algorithm abc(a, b, c)
2    {
3       return a+b+b*c + (a+b –c) / ( a+b)
4    + 4.0;
     }
```

It is characterized by values of *a, b, c*. If we assure that one word is needed to store the values of each *a, b, c , result* and also we see $S_P(instance\ characteristics)=0$ as space needed by *abc* is independent of instance characteristics; So 4 words of space is needed by this algorithm.

Example 2:

1
2
3
4
5
6
7
	Algorithm sum(*a, n*)
	{
	s := 0.0;
	
	for *i* := 1 **to** *n* **do**
	s := s+a[i];
	return *s*;
	}
		This algorithm is characterized by *'n'* (number of elements to be summed). The space required for *'n'* is 1 word. The array *a[]* of float values require atleast *'n'* words. So, we obtain
		$\boldsymbol{S_{sum}(n) \geq n+3}$
		(*n* words for *a*, 1 word for each of *n, i, s*)

Time Complexity:

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

- The time *T(P)* taken by a program *P* is the sum of the *compile time* and *Run time*. Compile time does not depend on instance characteristics and a compiled program will be run several times without recompilation, so we concern with just run time of the program. Run time is denoted by $\boldsymbol{T_P(instance\ characteristics)}$.
- The time complexity of an algorithm is given by the number of steps taken by the algorithm to compute the function. The number of steps is computed as a function of some subset of number of inputs and outputs and magnitudes of inputs and outputs.
- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.
- The number of steps any problem statement is assigned depends on the kind of statement.
  - For example, comments 0 steps.
  - Assignment statements 1 step
  - Iterative statements such as for, while & repeat-until 1 step for control part of the statement.
- Operation Counts
  - Based on compiler used: number of adds, multiply, compares etc.
  - Success depends on our ability to identify the operations that contribute most to the time complexity
- Step Counts
  - Account for all time spent in all parts of the program/function
  - Function of instance characteristics
  - Definition: A **program step** is syntactically or semantically meaningful segment of a program for which the execution time is independent of the instance characteristics
    - Ex: 10 additions can be one step, 100 multiplications – one step
- Instance characteristics
  - *Number of inputs, outputs etc.*

- *More complex example:*
  - Number of swaps performed by a Bubble sort
  - Depends on the instance characteristic array size (n), but also on the values: the numbers of swaps varies from *0* to *n-1*
- Operation count is not uniquely determinable by the chosen instance characteristics
- Need best, worst, and average counts

- Number of steps needed by a program to solve a particular problem is determined in two ways.

Method 1 : (Step count calculation using a variable)

- In this method, a new global variable count with initial value '0' is introduced into the program.
- Statements to increment count are also added into the program.
- Each time a statement in the original program is executed, count is incremented by the step count of the statement.

Example:

```
1     Algorithm Sum (a, n)
2     {
3        s:=0.0;
4        count :=count + 1;  //count is global;
5     initially 0
6        for i := 1 to n do
7        {
8           count:=count+1;  //for for
9           s:=s+a[i]; count:=count+1; //for
10    assignment
11       }
12       count :=count + 1;  //for last time of for
13       count :=count + 1;  //for the return
14       return s;
      }
```

- The change in the value of *count* by the time this program terminates is the number of steps executed by the algorithm.
- The value of count is increment by ***2n*** in the for loop.
- At the time of termination the value of count is ***2n+3***.
- So invocation of Sum executes a total of **2n+3** steps.

Example 2:

```
Algorithm RSum(a, n)
{
   count:=count+1; //for the if
conditional
   if ( n ≤ 0) then
   {
        count:=count+1;  //for the return
        return 0.0;
   }
   else
   {
        count:=count+1; //for addition,
function call, return
        return RSum(a,n-1)+a[n];
   }
}
```

- Let $t_{RSum}(n)$ be the increase in the value of *count* when the algorithm terminates.
- When n=0, $t_{RSum}(0) = 2$
- When n>0,*count* increases by 2 plus $t_{RSum}(n-1)$
- When analyzing a recursive program for its step count, we often obtain a recursive formula.
- These Recursive formulas are referred to as *recurrence relations*

To solve it, use repeated substitutions

$$
\begin{aligned}
t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\
&= 2 + 2 + t_{RSum}(n-2) \\
&= 2(2) + t_{RSum}(n-2) \\
&: \\
&= n(2) + t_{RSum}(0) \\
&= 2n + 2, \qquad n \geq 0
\end{aligned}
$$

So the step count of *RSum* is $2n+2$.
This step count is telling *the run time for a program with the change in instance characteristics*.

Method 2 : (Step count calculation by building a table)

- In this method, the step count is determined by building a table. We list total number of steps contributed by each statement in the table. The table is build in this order.
- Determine the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed.
- (*The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.)*
- The total contribution of the statement is obtained by combining these two quantities.
- Step count of the algorithm is obtained by adding the contribution of all statements.

Asymoptotic Notations used for Time Complexity:

The various notations used to measure time complexity are:

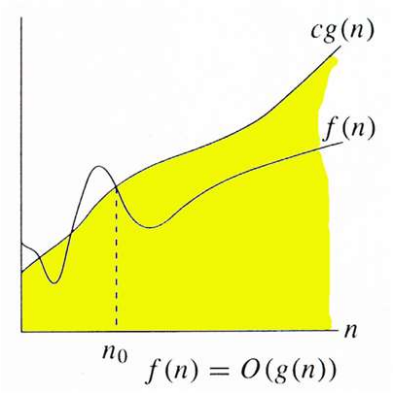
- Big-Oh notation(O)
- Omega Notation( )
- Theta Notation( )

Big-O Notation :

- Let n be a non-negative integer representing the size of the input to an algorithm
- The Big O notation defines an upper bound of an algorithm; it bounds a function only from above.
- Big-Oh notation is used widely to characterize running time and space bounds in terms of some parameter *n*, which varies from problem to problem.
- Constant factors and lower order terms are not included in the big-Oh notation.
- For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.

- **Def:** The function $f(n)=O(g(n))$ iff there exists positive constants c and n_0 such that
  - $f(n) \leq c*g(n)$ for all n, $n \geq n_0$.

- Figure shows that, for all values n at and to the right of n_0, the value of function $f(n)$ is on or below $c.g(n)$.



- When computing the complexity,
  - *f(n) is the actual time formula*
  - *g(n) is the simplified version of f*
  - Since f(n) stands often for time, we use T(n) instead of f(n)
  - In practice, the simplification of T(n) occurs while it is being computed by the designer
- If T(n) is the sum of a constant number of terms, drop all the terms except for the most dominant (biggest) term;
- Simplification Methods:
  - Drop any multiplicative factor of that term
  - What remains is the simplified g(n).
  - $a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 = O(n^m)$.
  - $n^2 - n + \log n = O(n^2)$

Example : *7n – 2* is *O(n)*

Proof: By the big-Oh definition, we need to find a real constant $c>0$ and an integer constant $n_0 \geq 1$ such that $7n - 2 \leq cn$ for every integer $n \geq n_0$.

Possible choice is $c=7$ and $n_0=1$.

Example: $20n^3+10n \log n+5$ *is* $O(n^3)$

Proof: $20n^3+10n \log n+5 \leq 35n^3$, for $n \geq 1$

In fact, any polynomial $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ will always be $O(n^k)$.

Here is a list of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first. k is some arbitrary constant.

Notation	• Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Polylogarithmic
O(n^2)	Quadratic
O(n^3)	Cubic
O(n^k) (k≥1)	Polynomial
O(k^n) (k>1)	exponential

- Instead of always applying the big-Oh definition directly to obtain a big-Oh characterization, we can use the following rules to simplify notation.

- Example:

  $f(x) = x^2 + 2x + 1$.

  For $x > 1$ we have:

$x^2 + 2x + 1 <= x^2 + 2x^2 + x^2$

$x^2 + 2x + 1 <= 4x^2$

Therefore, for C = 4 and k = 1:

$f(x) <= Cx^2$ whenever $x > k$.

So, $f(x)$ is $O(x^2)$.

- If $f(x)$ is $O(x^2)$, is it also $O(x^3)$? **Yes.** x^3 grows faster than x^2, so x^3 grows also faster than $f(x)$. Therefore, we always have to find the **smallest** simple function $g(x)$ for which $f(x)$ is $O(g(x))$.
- "Popular" functions g(n) are n log n, 1, 2^n, n^2, n!, n, n^3, log n
- Listed from slowest to fastest growth: 1 log n n n log n n^2 n^3 2^n n!
- **Useful Rules for Big-O**
  - For any **polynomial** $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers, $f(x)$ is $O(x^n)$.
  - If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then
    $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$
  - If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $O(g(x))$, then
    $(f_1 + f_2)(x)$ is $O(g(x))$.
  - If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then
    $(f_1 f_2)(x)$ is $O(g_1(x)\, g_2(x))$.

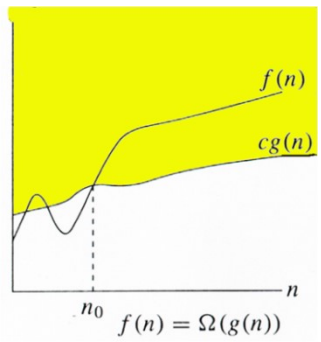
- **[Omega] Ω-notation:**

  Ω-notation provides an asymptotic upper bound.

  Def: The function $f(n)=\Omega(g(n))$ iff there exists positive constants c and n_0 such that

  $f(n) \geq c*g(n)$ for all n, $n \geq n_0$

  Figure shows the intuition behind Ω- notation for all values n at or to the right of n_0, the value of $f(n)$ is on or above $c.g(n)$.



- If the running time of an algorithm is $\Omega(g(n))$, then the meaning is, "the running time on that input is atleast a constant times $g(n)$, for sufficiently large n".

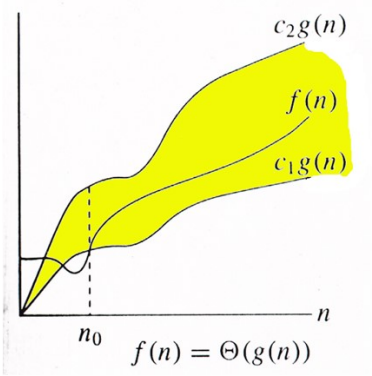
- It says that, $\Omega(n)$ gives a lower-bound on the best-case running time of an algorithm.

[Theta] - notation :

Def: The function f(n) = $\Theta(g(n))$ iff there exists positive constants c_1, c_2 and n_0 such that

$c1*g(n) \leq f(n) \leq c2*g(n)$ for all n, $n \geq n_0$

Following figure gives an intuitive picture of functions $f(n)$ and $g(n)$, where .



- For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c1.g(n)$ and at or below $c2.g(n)$.
- For all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within constant factors.
- We can say that $g(n)$ is an asymptotically tight bound for $f(n)$.

SEARCHING

Searching is the (concept) mechanism of searching an element in the list of da ta stored. Basically it is used to find the location or to see whether element is available or not in the list. The application of search occurs when a particular element is required from a set of stored elements. For example, consider a database of telephone directory. It is required to find the address when the telephone number is known.

The efficiency of a search tends to depend upon three things.

- The size and organization of the collection of data we are searching
- the search algorithm being used
- The efficiency of the test used to determine if the search is successful.

The popular searching methods are:

- Linear search
- Binary Search
- Fibonacci search

Any search method requires three components. They are,

- **Search List:** The search list is simply a general term for the collection of elements that we are searching.
- **Key element/ Target element:** This is the term we use to refer to the element that we are searching for. When ever a search is conducted, there are two possible outcomes:
  - The element is found in the list (successful search)
  - The element is not found (unsuccessful search)
- **Test function (or) operator :** These are used to determine how an element in the search list matched the target element. For numbers ==, !=, >= etc and for strings strcmp(), strncmp(), strcmpi() etc are used.

Hence, searching is a terchnique of finding accurate location of an element in the given data list. If the key element is present in the search list then search process is said to be successful and the search is said to be unsuccessful if the given element does not exist in the list.

Linear Search

Linear search, also referred as sequential search is the simplest searching technique. In this, the required element (key element) is searched linear through out the search list.

The search begins atone end of the list and searches for the required element one by one until the element is found or till the end of the list is reached. i.e. from the given list, start the search process by comparing the first element with the key element. If the match is found, then stop the search. Otherwise compare the second element and so on. Continue this process till the key element is

found or there are no more elements in the list (in this case the element is not there in the list). The search is said to be successful if the search element is found and unsuccessful if the search element is not found.

For example, consider the list of 5 elements as, 23 45 67 32 86

- if the key element is 67 then, Start searching by comparing the first element (23) with the key element(67) . As two elements are not matched, move to second(45) and so on. The element is found at position 3.
- If the key element is 58 then search linearly through out the list. In this case, the element is not present in the list.

Algorithm: Linear_Search(A, key,n)

```
// A is the linear array with n elements and key is the key element
{
        i <-- 0;  // initialize linear search
        while ( a[i] ≠key) do
                i <--  i+1;
        if ( i=n) then
          print ' unsuccessful search'
        else
           print ' element found at ', i
}
```

Example:

```
/* program for linear serach using non-resursive function*/
#include<stdio.h>
#include<conio.h>
int l_search(int[],int,int);
void main()
{
 int a[10],i,n,key,x;
 clrscr();
 printf("Enter Integer Array size :");
 scanf("%d",&n);
 printf("\n Enter elements \n");
 for(i=0;i<n;i++)
   scanf("%d",&a[i]);
 printf("\n Enter Key element to search :");
   scanf("%d",&key);
 x=l_search(a,key,n);
 if(x==-1)
   printf("unsuccessful search");
 else
  printf("key element found at:%d",x);
 getch();
}
int l_search(int a[],int key,int n)
{
 int i;
 for(i=0;i<n;i++)
 {
  if(key==a[i])
    return i;
 }
 return -1;
}
```

```
/* program for linear search using recursive function */

#include<stdio.h>
#include<conio.h>
int l_search(int[],int,int);
void main()
{
 int a[10],i,n,key,x;
 clrscr();
 printf("Enter Integer Array size :");
 scanf("%d",&n);
 printf("\n Enter elements \n");
 for(i=0;i<n;i++)
   scanf("%d",&a[i]);
 printf("\n Enter Key element to search :");
   scanf("%d",&key);
 x=l_search(a,key,n-1);
 if(x==-1)
   printf("unsuccessful search");
 else
  printf("key element found at:%d",x);
 getch();
}
int l_search(int a[],int key,int i)
{

  if(i<0)
    return -1;
  if(key==a[i])
    return i;
  return l_search(a,key,i-1);

}
```

- The linear search can be applied on sorted or unsorted list of elements i.e. this method required no ordering of elements in the list that's why it is some times referred to as ' brute force' method.
- For large size lists, the performance of linear search, how ever makes it a poor search strategy . The time it takes to perform linear search grows proportationally with the size of the list.

Analysis:

- If there are N items in the list, then in the 'worst case' (i.e. where there is no key element in the list or key element is the last element in the list) N comparisons are required.
- The 'best case' , in which the first comparison returns a match, it requires a single(1)

comparison.

- The average case roughly requires 'N/1' comparisons to search the element.

Time Complexity	Best Case	Best Case
	O(1)	O(N)

Binary Search

The binary search method is a classical method and it is one of the most efficient searching techniques, which requires the list to be sorted.

To search for an element in the list the binary search procedure splits the list and locates the moddle element of the list. It is then compared with the search element. If the search element is matched with the middle element, then the search is complete. Otherwise, the key element may be in the upperhalf or lower half. If the key element is less than the middle element, the first part(upperhalf/ left half) of the list is searched else the second part(lower half/ right half) of the list is searched. The process is continued until the key element is found or the portion of the sublist to be searched is empty.

In this method,

- the terms ub and lb are used to represent last and first positions of an array. The mid position is found out as , mid = (lb+ub)/2.
- Now, compare the key element with the element in the mid position then, any one the following cases may arise:

case 1: If the key element is equal to the element present in the mid position, then the search option is complete.

Case 2: if the key element is less than the element present in the mid position, then there is no need to check elements in the second(right) half of array i.e. right half can be excluded. Now assign mid-1 to ub.

Case 3: If the key element is greater than the element present in the mid position, then there is no need to check the elements in the first(left) half of array i.e. left half can be excluded. Now assign mid+1 to lb.

- This process is executed repeatedly on the non-excluded portion of the array until the key element is found or if we eliminate all elements from search then we can say the element is not present in the given list.

Note: Since, in this method, we eliminate elements by half each time, this method is faster when compared to sequential search.

Algorithm: Binary_Search(A,N,Key)

```
//  A is an array consisting of N elements in ascending order.
// Key is the element to be searched
{
      lb <-- 0;  ub <-- N-1;
      while ( lb<=ub)   do
      {
          mid <-- (lb + ub)/2;
          if(key = A[mid]) then
               break;
          else
               if( A[mid] > key) then
                     ub <-- mid-1;
             else
                     lb <-- mid+1;
          }
          if(lb>ub)
            print ' unsuccessful search';
          else
             print 'element found at', mid;
}
```

/* Program for Binary Search (non - recursive) */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
int binarysearch(int a[],int key,int low,int high)
{
  int mid;
  while(low<=high)
  {
    mid=(low+high)/2;
    if(a[mid]==key)
       return mid;
    if(a[mid]>key)
      high=mid-1;
    else
      low=mid+1;
  }
  return -1;
}
void main()
{
 int a[10],i,n,key,x;
 clrscr();
 printf ("Enter Integer Array size :");
 scanf("%d",&n);
 printf("\n Enter elements in Sorted order\n");
 for(i=0;i<n;i++)
   scanf("%d",&a[i]);
 printf("\n Enter Key element to search :");
 scanf("%d",&key);
 x=binarysearch(a,key,0,n-1);
 if(x==-1)
   printf("the key element not found");
 else
   printf("the key element found at:%d",x);
 getch();
}
```

/* Program for Binary Search (recursive) */

```
#include<stdio.h>
#include<conio.h>
int binarysearch(int a[],int key,int low,int high)
{
  int mid;
  if (low>high)
        return -1;
  else
  {
    mid=(low+high)/2;
    if(a[mid==key)
      return mid;
    else
      if(a[mid]>key)
        return  binarysearch(a,key,lb,mid-1);
    else
      return binarysearch(a,key,mid+1,ub);
  }
}
void main()
{
 int a[10],i,n,key,x;
 clrscr();
 printf ("Enter Integer Array size :");
 scanf("%d",&n);
 printf("\n Enter elements in Sorted order\n");

 for(i=0;i<n;i++)
   scanf("%d",&a[i]);

 printf("\n Enter Key element to search :");
   scanf("%d",&key);

 x=binarysearch(a,key,0,n-1);
 if(x==-1)
   printf("the key element not found");
 else
   printf("the key element found at:%d",x);
 getch();
}
```

SORTING

"Sorting refers to the operation of rearranging data in either in ascending or descending order".

* the data may be numerical data or character data.

The sorting methods are classified into two types. They are ' Internal Sorting' and 'External Sorting'. In internal sorting, all the data elements to be sorted are present in the main memory. External sorting techniques are applied to large data sets which reside on secondary storage devices and cannot be completely fit in main memory.

There are many sorting methods available. But no method for sorting is best in all cases. The factors to be considered while choosing a sorting technique are,

- Programming time of the sorting technique
- Execution time of the sorting technique
- No. of comparisons required for sorting the list.
- Main or secondary memory space needed for sorting technique.

Different applications require different sorting methods. The different sorting methods are,

- Bubble Sort (or) Exchange Sort
- Selection Sort
- Quick Sort (or) Partition Exchange Sort
- Insertion Sort
- Merge Sort

Note: A sorting method is said to be stable when it has the minimum no. of swaps.

- Stable sorting methods are - Bubble Sort, Selection Sort & Quick Sort
- Unstable Sorting methods are – Insertion Sort, Merge Sort

Bubble Sort (or) Exchange Sort

The simplest and the most widely used sorting technique is ' Bubble Sort'.

This method compares the two adjacent(consecutive) elements of the list. If thesy are not in order, th two elemtns will be interchanged. If they are in order, the two elements remains the same. This process continues (n-1) times for sorting an array of size n. There ends one pass. During the first pass the largest/smallest will be moved to Nth position. The second pass will be continued with first (N-1) elements. Repeat this process till all the elemtns are in sorted order.

* Since, each pass places one element into its proper position, a list of N elements requires (N-1) passes.

The procedure for bubble sort to sort 'N' elements in ascending order is,

Step 1: First compare two elements at time starting with the first tow elements.

Step 2: If the first element is larger than second element then exchange the two elements

Step 3: Go down one element and compare that element to the element that follows it.

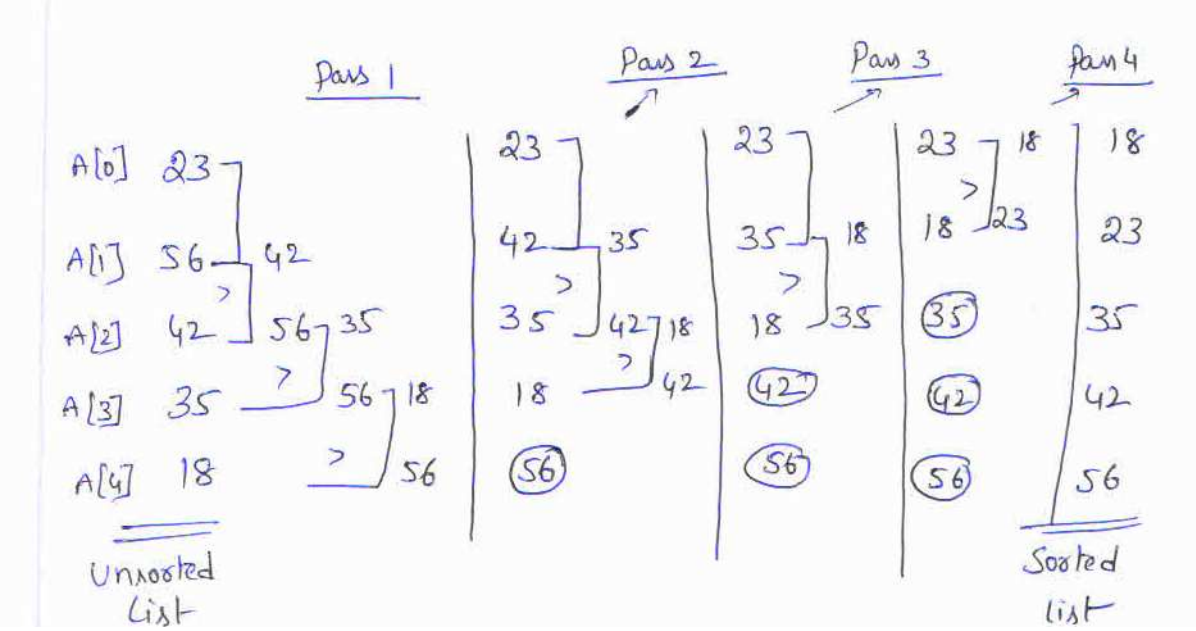
Step 4: Continue this process till the end of list. (During this process the largest element moved to Nth position)

Step 5: Repeat steps 1 thru 4 (N-1) times by leaving the last element of sorting list each time.

For example, consider the list of 5 elements,

A[0]	A[1]	A[2]	A[3]	a[4]
23	56	42	35	18

The following comparisons are made in each pass:



" The first largest element is placed in proper position i.e. at A[N-1] after the first pass" . In general " Kth largest element is placed in its proper position i.e. at A[n-k] after pass 'k' ".

The complete set of passes in bubble sort method is,

Algorithm: Bubble_Sort(A,N)

```
// A is an array of N elements
{
        for i:= 1 to N-1 do
        {
          for j:=0 to N-i-1  do
             if( A[j] > A[j+1] ) then
                  swap( A[j],A[j+1]);
        }
}
```

```
/*  Program for BUBBLE SORT   */
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void Bubble_Sort( int[], int)
void main()
{
        int a[10],n,i;
        clrscr();
        printf("\n Enter Array size :");
        scanf("%d",&n);
        printf("\n Enter elements \n");
        for (i=0;i<n;i++)
        scanf("%d",&a[i]);
        bubblesort(a,n);
        printf("\n Array elements after sorting:
        \n");
        for (i=0;i<n;i++)
        printf("%d  ",a[i]);
        getch();
}
```

```
void Bubble_Sort(int a[],int n)
{
   int i,j,temp;
   for(i=1;i<n;i++)
   {
    for(j=0;j<n-i;j++)
    {
     if(a[j]>a[j+1])
     {
       temp=a[j];
       a[j]=a[j+1];
       a[j+1]=temp;
     }
    }
  }
}
```

Analysis (or) Time Complexity for Bubble Sort:

- To sort N elements Bubble Sort requires (N-1) passes.
  - Pass 1 requires (n-1) comparisons
  - Pass 2 requires (n-2) comparisons
  - ..
  - ..
  - Pass k requires (n-k) comparisons
  - ..
  - ..
  - Pass (n-1) performs 1 comparison
- The total no. of comparisons = 1+2+3+ ..... + n-1

= n(n-1)/2 = (n2-n)/2
= O(n2)

Selection Sort

Selection Sort is the easiest method to sort list of elements.

In selection sort method first find the smallest(largest) element in the list and swap it with the first element. Then find the second smallest(largest) element in the list and swap it with the second element of the list. The process of searching and swapping the next smallest is repeated until all the elements in the list have been sorted in ascending (descending) order.

i.e the Selection Sort searches all of the elements in a list until it finds the smallest element. It swaps

this with the first element in the list.

Next it finds the smallest of the remaining elements and swaps it with the second element and so on. The procedure to sort 'n' elements in ascending order is,

Pass 1: Find the position (minpos) of the smallest element in the list of 'n' elements. Then interchange A[minpos] & A[0]. i.e. A[0] is sorted means it is in proper place.

Pass 2: Find position (minpos) of the smallest element in the sublist of (n-1) elements then interchange A[minpos] & A[1]. Now A[0] & A[1] are sorted i.e. A[0] ≤ A[1].

..

..

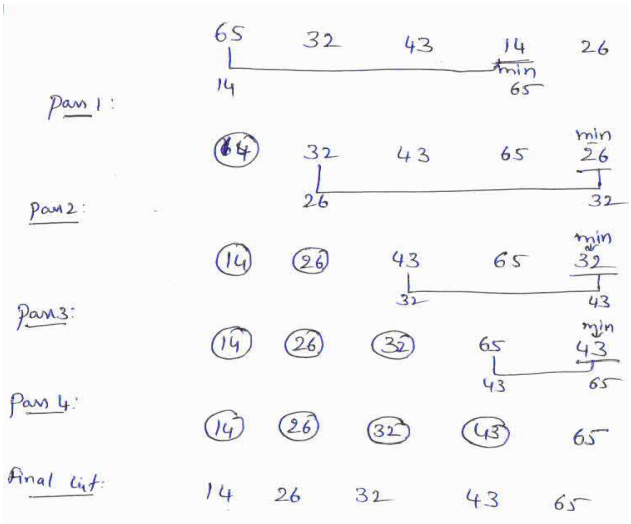
pass (n-1) : Find the position (minpos) of the smallest element in A[n-2] & A[n-1] then interchange A[minpos] & A[n-2]. Now A[0], A[1],...., A[n-2] are sorted means A[n-1] is also in proper place.

Thus " the list of n elements are sorted after (n-1) passes".

For example, suppose an array A contains 5 elements: 65 32 43 14 26

In first pass, the smallest element is searched by making of 4 comparisons and the smallest element is 14. Then it is swapped with the first element i.e. 65. After the first pass the elements are : 14 32 43 65 26.

The complete set of passes in selection sort are:



Algorithm: Selection_Sort(A, n)

```
// A is an array of n elements
{
        for i:= 0 to n-2 do
        {
          min := A[i];  minpos := i;
          for j:= i+1 to n-1 do
                if(a[j]< min) then
                {   min := a[j] ;  minpos := j;    }
          swap(A[i], A[minpos]);
```

```
        }
}
```

/* Program for Selection Sort */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void Selection_Sort(int[], int);
void main()
{
  int a[10],n,i;
  clrscr();
  printf("\n Enter Array size :");
  scanf("%d",&n);
  printf("\n Enter elements \n");
  for (i=0;i<n;i++)
  scanf("%d",&a[i]);
  Selection_Sort(a,n);
  printf("\n Array elements after sorting: \n");
  for (i=0;i<n;i++)
  printf("%d  ",a[i]);
 }
```

```
void Selection_Sort(int a[], int n)
{
 int i,j,min,temp;
 for(i=0;i<n-1; i++)
 {
  min=i;
  for(j=i+1; j<n; j++)
  {
   if(a[j]<a[min])
        min=j;
  }
  temp=a[i];
  a[i]=a[min];
  a[min]=temp;
 }
}
```

Analysis of Selection Sort:

Analysis (or) Time Complexity for Bubble Sort:

- To sort N elements Bubble Sort requires (N-1) passes.
  - Pass 1 requires (n-1) comparisons
  - Pass 2 requires (n-2) comparisons
  - ..
  - ..
  - Pass k requires (n-k) comparisons
  - ..
  - ..
  - Pass (n-1) performs 1 comparison
- The total no. of comparisons = 1+2+3+ ..... + n-1

= n(n-1)/2 = (n2-n)/2

= O(n2)

Insertion Sort

The main idea of Insertion Sort is to consider each element at a time, into the appropriate position relative to the sequence of previously ordered elements, such that the resulting sequence is also ordered.

In each pass an element is compared with its predecessors and if it is not in the right position, it is

placed in the right place among the elements being compared.

Suppose an array 'A' with 'n' elements A[0],A[1],....A[n-1]. The insertion sort algorithm scans A from A[0] to A[n-1], inserting each element A[k] into proper position in the previously sorted subarray A[0], A[1], ... , A[k-1].

The procedure is, consider the first element, for only one element no sort is required because it is trivially sorted. So the procedure starts with second element.

Pass 1: A[1] is inserted either before or after A[0] so that A[0] & A[1] are sorted.

Pass 2: A[2] is inserted into proper position by comparing with A[0],A[1] so that A[0],A[1] & A[2] are sorted.

..

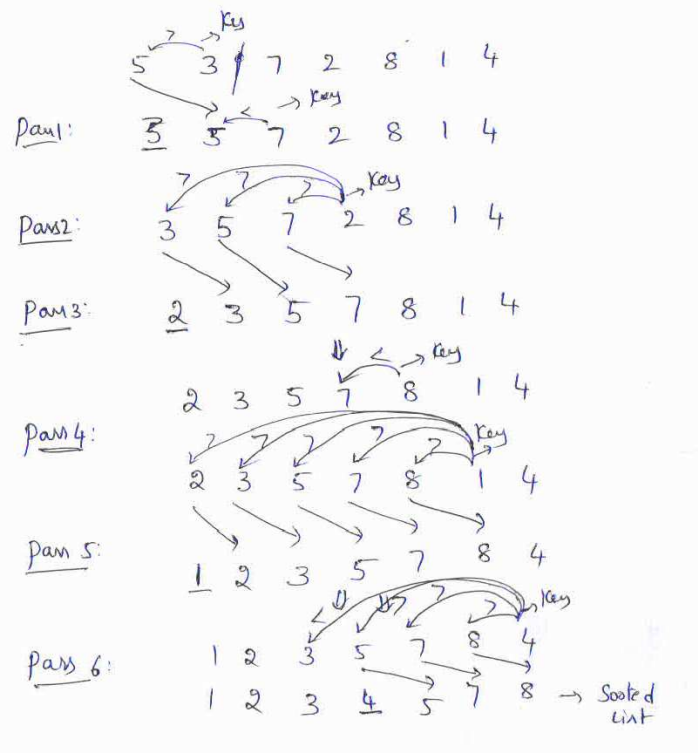
..

Pass n-1: A[n-1] is inserted into its proper positioning A[0], A[1],...., A[n-2] so that all the n elements are sorted

* Simply, in Pass k, A[k] is compared with its previous elements (A[0],A[1],...., A[k-1]) and inserted after the element which is less than or equal to the A[k].

Example: consider the list of elements - 5 3 7 2 8 1 4

To sort this list in ascending order using insertion method, the passes are:



Algorithm: Insertion_Sort(A,n)

```
// A is an array of n elements
{
        for i:= 1 to n-1 do
        {
           key:=A[i];  j := i-1;
           while ((j ≥ 0) and (A[j] > key)) do
           {
                 A[j+1] := A[j];  j := j-1;
           }
           a[j+1] := key;
        }
}
```

/* Program for INSERTION SORT */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
void Insertion_Sort(int[], int);
void main()
{
int a[10],n,i;
clrscr();
printf("\n Enter Array size :");
scanf("%d",&n);
printf("\n Enter elements \n");
for (i=0;i<n;i++)
scanf("%d",&a[i]);
Insertion_Sort(a,n);
printf("\n Array elements after sorting: \n");
for (i=0;i<n;i++)
printf("%d  ",a[i]);
getch();
}
```

```
void Insertion_Sort(int a[],int n)
{
  int i,j,key;
  for(i=1;i<n;i++)
  {
        key=a[i];j=i-1;
        while((j>=0)&&(a[j]>key))
        {
           a[j+1]=a[j];
           j--;
        }
        a[j+1]=key;
  }
}
```

Analysis:

It is easiest sorting method. If list is already sorted, only one comparison is made on each pass so that (n-1) passes requires (n-1) comparisons. Hence the time complexity is O(n) i.e. Best case.

In worst case i.e if the list is sorted in reverse order (unsorted) then the total no. Of comparisons are ,

$1 + 2 + 3 + ... + (n-1) = n(n-1)/2 = (n2-n)/2 = O(n^2)$.

So, the Worst Case time complexity is : $O(n^2)$

and the Best Case time complexity is : O(n)

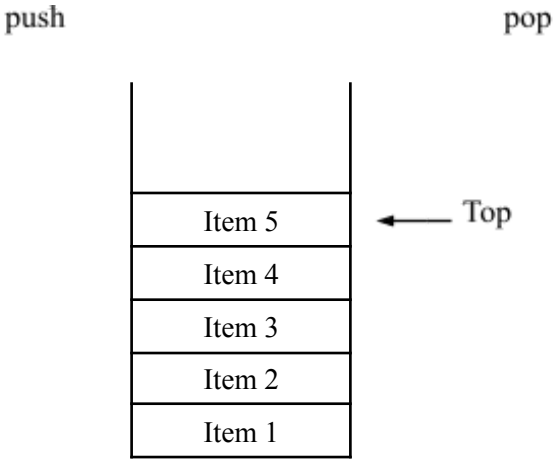
Unit - III

Stacks and Queues

Learning Material

Stacks

- Stack is a linear data structure.
- Stack is an ordered collection of homogeneous data elements, where insertion and deletion operations takes place at only one end called **TOP**.
- The insertion operation is termed as **PUSH** and deletion operation is termed as **POP** operation.
- The **PUSH** and **POP** operations are performed at **TOP** of the stack.
- An element in a stack is termed as **ITEM**.



Schematic diagram of a stack

- The maximum number of elements that stack can accommodate is termed as **SIZE** of the stack.
- Stack follows **LIFO** principle. i.e. **L**ast **I**n **F**irst **O**ut.

Representation of stack

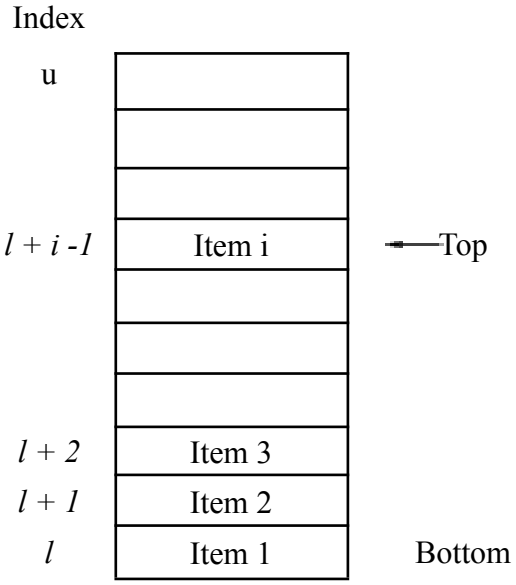
There are two ways of representation of a stack.

1. Array representation of a stack.
2. Linked List representation of a stack.

1. Array representation of a stack

- First we have to allocate memory for array.

- Starting from the first location of the memory block, items of the stack can be stored in sequential fashion.



Array representation of stack

In the above figure item i denotes the i^{th} item in stack.

l and u denotes the index ranges.

Usually *l* value is **0** and *u* value is **size-1**.

From the above representation the following two statuses can be stated.

Empty Stack: top < *l* i.e. **top < 0**

Stack is full: top >=*u* i.e. **top >= size-1**

Stack overflow

Trying to PUSH an item into full stack is known as stack overflow.

Stack overflow condition is top >= size-1

Stack underflow

Trying to POP an item from empty stack is known as Stack underflow.

Stack underflow condition is top < 0 or top = = -1

Operations on Stack

PUSH	:	To insert element in to stack
POP	:	To delete element from stack
Status	:	To know present status of the stack

Algorithm Stack_PUSH(item)

Input: *item* is new item to push into stack.

Output: pushing new item into stack at top whenever stack is not full.

1. if(top >= size-1)
    a) print "stack is full, not possible to perform push operation"
2. else
    a) top = top+1
    b) s[top] = item
3. end if

End Stack_PUSH

Algorithm Stack_POP( )

Input: Stack with some elements.

Output: item deleted at top most end.

1. if(top < 0)
    a) print "stack is empty not possible to pop"
2. else
    a) item = s[top]
    b) top = top - 1
    c) print(item)
3. end if

End Stack_POP

Algorithm Stack_Status( )

Input: Stack with some elements.

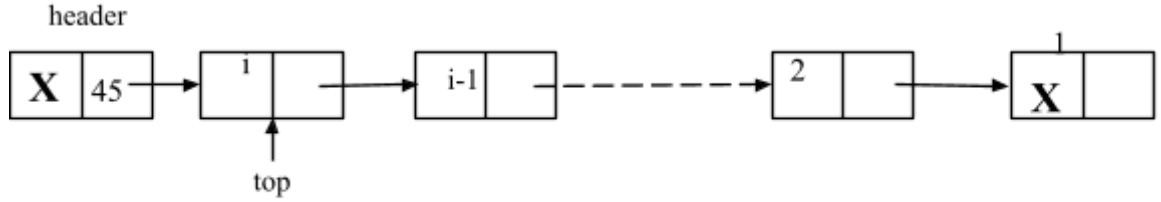
Output: Status of stack. i.e. Stack is empty or not, full or not, top most element in Stack.

1. if(top > = size-1)
    a) print "stack is full"
2. else if(top < 0)
    a) print "stack is empty"
3. else
    a) print "top most item in stack is" s[top]
4. end if

End Stack_Status

2. Linked List representation of a stack

- The array representation of stack allows only fixed size of stack. i.e. static memory allocation only.
- To overcome the static memory allocation problem, linked list representation of stack is preferred.
- In linked list representation of stack, each node has two parts. One is data field is for the item and link field points to next node.



Linked List representation of stack

- Empty stack condition is
    top = = NULL (or) header□link = = NULL
- Full condition is *not applicable* for Linked List representation of stack. Because here memory is dynamically allocated**.**
- In linked List representation of stack, top pointer always points to top most node only. i.e. first node in the list.

Operations on Stack with linked list representation

PUSH : To insert element in to stack

POP : To delete element from stack

Status : To know present status of the stack

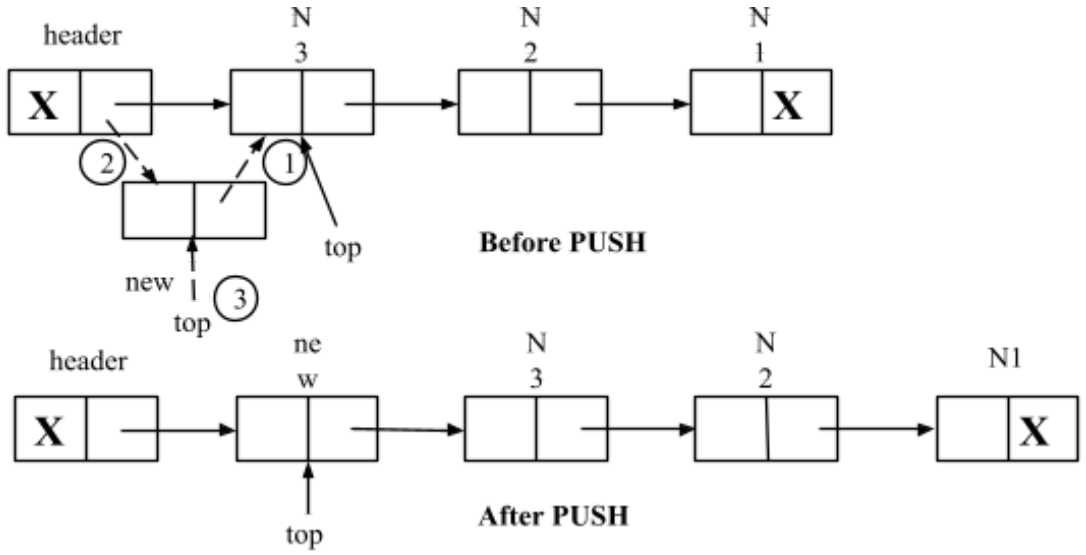
Algorithm Stack_PUSH_LL(item)

Input: *item* is new item to push into stack.

Output: pushing new item into stack at top.

1. new = getnewnode( )
2. if(new = = NULL)
    a) print "Required node is not available in memory"
3. else
    a) new□link = header□link
    b) header□link = new
    c) top = new
    d) new□data = item

End Stack_PUSH_LL



1. The link part of the new node is re placed with address of the previous top most node.
2. The link part of the header node is replaced with address of the new node.
3. Now the new node becomes top most node. So top is points to new node.

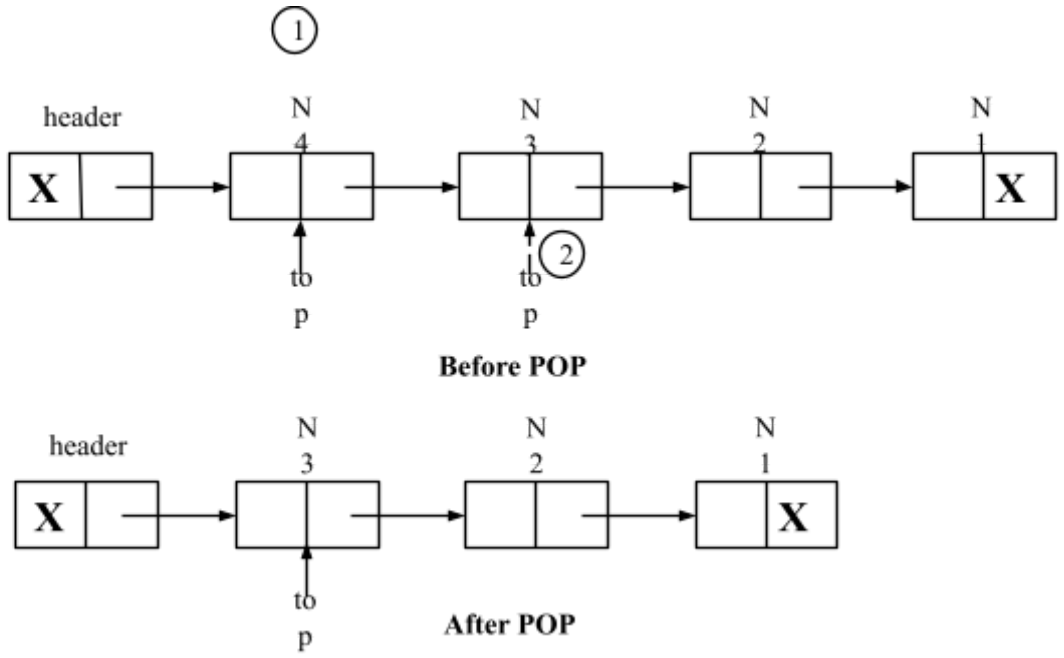
Algorithm Stack_POP_LL( )

Input: Stack with some elements.

Output: item deleted at top most end

1. if(header□link = = NULL)
    a) print "Stack is empty, unable to perform POP operation"
2. else
    a) header□link = top□link
    b) item = top□data
    c) top = header□link

End Stack_POP_LL



1. Link part of the header node is replaced with the address of the second node in the list.
2. After deletion of top most node from list, the second node becomes the top most node in the list. So top points to the second node.

Algorithm Stack_Status_LL( )

Input: Stack with some elements.

Output: Status of stack. i.e. Stack is empty or not, top most element in Stack.

1. if(header□link = = NULL || top = = NULL)
    a) print "Stack is empty"
2. else
    a) print "Element present at top of stack is" top□data
3. end if

End Stack_Status_LL

Applications of stack

1. Infix to postfix conversion
2. Evaluation of postfix expression
3. Reversing list

1. Infix to postfix conversion

- An expression is a combination of operands and operators.

  Eg: c = a + b

- In the above expression a, b, c are operands and +, = are called as operators.
- We have 3 notations for the expressions.
  i. Infix notation
  ii. Prefix notation
  iii. Postfix notation

Infix notation: Here operator is present between two operands.

eg. a + b

The format for Infix notation as follows:

<operand> ***<operator>*** <operand>

Prefix notation: Here operator is present before two operands.

eg. + a b

The format for Prefix notation as follows:

<operator> <operand> <operand>

Postfix notation: Here operator is present after two operands.

eg. a b +

The format for Prefix notation as follows:

<operand> <operand> ***<operator>***

- While conversion of infix expression to postfix expression, we must follow the precedence and associativity of the operators.

Operator	**Precedence**	**Associativity**
^ or $ or ↑ (exponential)	3	Right to Left
* / %	2	Left to Right
+ -	1	Left to Right

- In the above table *, / and % have same precedence. So then go for associativity rule, i.e. from Left to Right.
- Similarly + and - same precedence. So then go for associativity rule, i.e. from Left to Right.

Eg: 1(Conversion of infix expression to postfix expression without using STACK)

(A + B) * (C – D)

A B + * (C – D)

A B + * C D –

A B + C D - *

Eg: 2 (Conversion of infix expression to postfix expression without using STACK)

([(A - { B + C }) * D] $ E + F)

([(A - BC+) * D] $ E + F)

([ABC+- * D] $ E + F)

(ABC+-D* $ E + F)

(ABC+-D*E$ + F)

ABC=-D*E$F+

- To convert an infix expression to postfix expression, we can use one stack.
- Within the stack, we place only operators and left parenthesis only. So stack used in conversion of infix expression to postfix expression is called as operator stack.

Ex: Convert the given infix expression to postfix expression using STACK

Input Character	**Operations on Stack**	**Operator Stack**	**Postfix Expression**
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and Append to Postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End	Pop till Empty		**AB*CD+-E+**

A * B- (C + D) + E

Ex: Convert the given infix expression to postfix expression using STACK

(A + B * (C - D)) / E

Input Character	**Operator Stack**	**Postfix Expression**
(	(	
A	(	A
+	(+	A
B	(+	AB
*	(+*	AB
(	(+*(	AB
C	(+*(	ABC
-	(+*(-	ABC
D	(+*(-	ABCD
)	(+*	ABCD-
)		ABCD-*+
/	/	ABCD-*+
E	/	ABCD-*+E
End		ABCD-*+E/

Algorithm Conversion of infix to postfix

Input: Infix expression.

Output: Postfix expression.

1. Perform the following steps while reading of infix expression is not over
    a) if symbol is left parenthesis then push symbol into stack.
    b) if symbol is operand then add symbol to postfix expression.
    c) if symbol is operator then check stack is empty or not.
        i) if stack is empty then push the operator onto stack.
        ii) if stack is not empty then check priority of the operators.
            (I) if priority of current operator > priority of operator present at top of stack then push operator into stack.
            (II) else if priority of operator present at top of stack >= priority of current operator then pop the operator present at top of stack and add popped operator to postfix expression (go to step i)
    d) if symbol is right parenthesis then pop every element form stack up corresponding left parenthesis and add the poped elements to postfix expression.
2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.

End conversion of infix to postfix

2. Evaluation of postfix expression

- To evaluate a postfix expression we use one stack.
- For Evaluation of postfix expression, in the stack we can store only operand. So stack used in Evaluation of postfix expression is called as operand stack.

Algorithm PostfixExpressionEvaluation

Input: Postfix expression

Output: Result of Expression

1. Repeat the following steps while reading the postfix expression.
    a) if the read symbol is operand, then push the value onto stack.
    b) if the read symbol is operator then pop the top most two items of the stack and apply the operator on them, and then push back the result to the stack.
2. Finally stack has only one item, after completion of reading the postfix expression. That item is the result of expression.

End PostfixExpressionEvaluation

Ex: Evaluate the given postfix epression using stack: 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

3. Reversing List of elements

- A list of numbers can be reversed by reading each number from an array starting from 1st index and pushing into stack.
- Once all the numbers have been push into stack, the numbers can be poped one by one from stack and store into array from the 1st index.

Algorithm Reverse_List_Stack(a <array>, n <intetger>)

Input : Array a with n elements

Output: Reversed List of elements

1. i=1 , top =0

2. while (i <=n)

    a) top = top + 1

    b) s[top] = a[i]

    c) i = i + 1

3. end while loop

4. i = 1

5. while(i <= n)

    a) a[i] = s[top]

    b) top = top – 1

    c) i = i + 1

6. end while loop

End Reverse_List_Stack

UNIT –II

Linked Lists

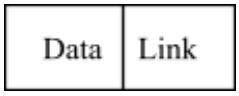
Learning Material

2.1 LINKED LISTS-BASIC CONCEPTS

- In arrays once memory is allocated, it can't extended any more. So array is known as *static Data Structure*.
- Linked List is *dynamic* Data Structure, where amount of memory required can be vary during it use.

Definition: A Linked List is an ordered collection of homogeneous data elements called nodes. Where the linear order is maintained by means of *links* (or) *pointers*.

- The representation of node is as follows:



Node: an element in Linked List

- A node consists of two parts. i.e. **Data part** and **Link part**.
- The data part contains actual data to be represented.
- The link part is also referred as address field, which contains address of the next node.

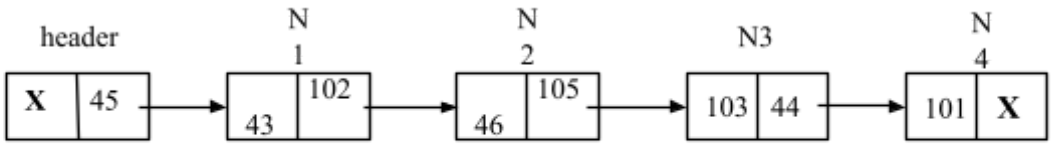
2.1.1 Representation of Linked List in memory

There are two ways to represent a Linked List in memory.

1. Static memory allocation using arrays.
2. Dynamic allocations using pool of storage.

2.1.1.1 Static memory allocation using arrays

- This representation maintains two arrays.
- One array used for Data and another array is used for link.



SLL with 4 nodes

- The static representation for a linked list shown in the above figure is given in the following figure.

Address	Array 1 (Data)	Array 2 (Link)
41		
42		
43	105	46
44	101	**X**
45	102	43
46	103	44
47		

Static Representation of SLL using arrays

2.1.1.2 Dynamic allocations using pool of storage

- The efficient way of representation of linked list is using pool of storage.
- In this method memory bank, memory manager and garbage collector is available.

Memory bank: is a collection of free memory spaces.

Memory manager: is a program.

- Whenever a linked list requires a node, then request is placed to memory manager.
- If the required node is available in the memory bank, then that node is send to caller.
- If the required node is not available in the memory bank, then memory manager send NULL to caller.

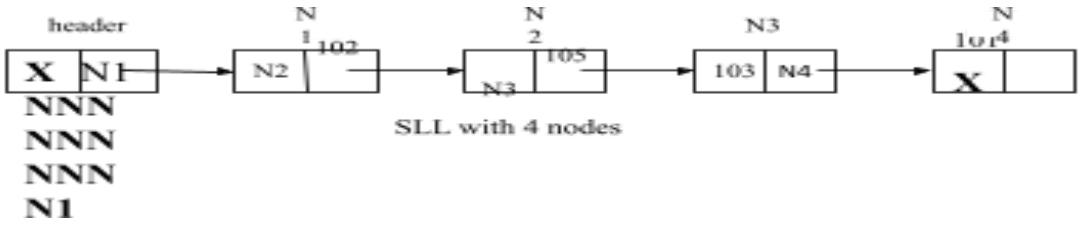
Garbage collector: collect the unused nodes in the linked list and send back to memory bank.

Types of linked lists

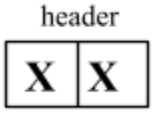
1. Single Linked List (SLL)
2. Double Linked List (DLL)
3. Circular Linked List (CLL)

2.2 SINGLE LINKED LIST

- In SLL, each node contains only one link, which points to the next node in the list.
- The pictorial representation of SLL is as follows.



- Here header is an empty node, i.e. data part is NULL, represented by X mark.
- The link part of the header node contains address of the first node in the list.
- In SLL, the last node link part contains NULL.
- In SLL we can move from left to right only. So SLL is called as one way list.
- If a SLL is empty, then the link part of the header node is NULL.



Empty Single Linked List

Operations On Single Linked List

1.Traversing a list

2. Insertion of a node in to SLL

3. Deletion of a node from SLL

2.2.1 Traversing a list

- Traversing a SLL means, visit every node in the list starting from first node to the last node.

Algorithm SLL_Traverse(header)

Input: header is the pointer to header node.

Output: Visiting of every node in SLL.

1. ptr = header
2. if(ptr->link==NULL)
   a) print “SLL is empty”
3. else
   a) while(ptr□link != NULL)
      i) ptr = ptr□link
      ii) print "ptr□data"
   b) end while
3. end if

End SLL_Traverse

2.2.2 Insertion of a node into SLL

The Insertion of a node in to SLL can be done in various positions.

i) Insertion of a node into SLL at beginning.

ii) Insertion of a node into SLL at ending.

iii) Insertion of a node into SLL at any position.

- For insertion of a node into SLL, we must get node from memory bank.
- The procedure for getting node from memory bank is as follows:

Procedure for getnewnode( )

1. Check for availability of node in memory bank
2. if (AVAIL = = NULL)
   a) print "Required node is not available in memory"
   b) return NULL
3. else
   a) return address of node to the caller
4. end if

End Procedure for getnewnode

2.2.2.1 Insertion of a node into SLL at beginning

- Insertion of a node into SLL at beginning means insert new node after header node.

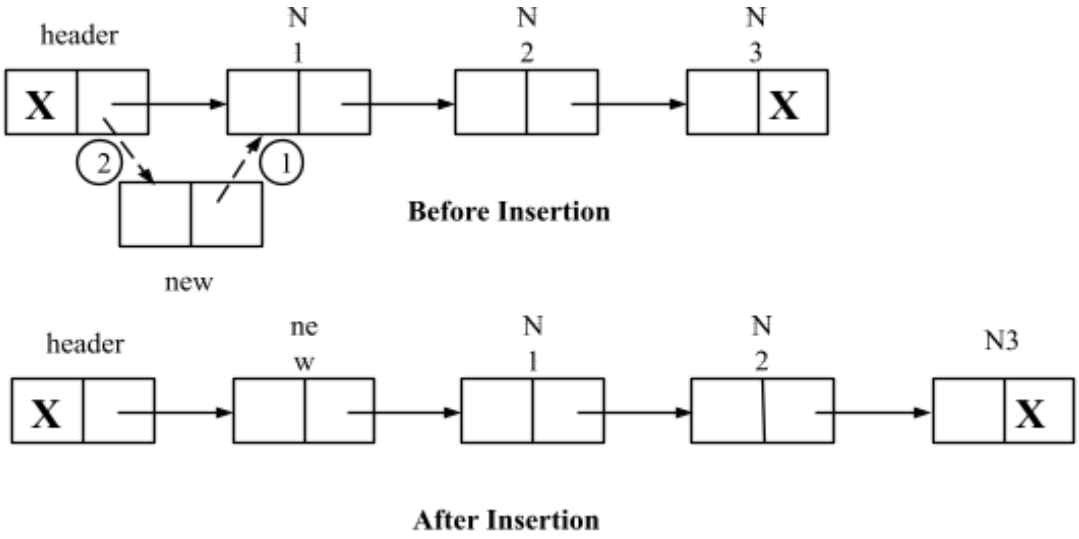
Algorithm SLL_Insert_Begin(header,x)

Input: *header* is a pointer to the header node, *x* is data part of new node to be insert.

Output: SLL with new node inserted at beginning.

1. new = getnewnode( )
2. if(new==NULL)
   a) print “node not creates”
3. else
   a) new□link = header□link /* 1 */
   b) header□link = new /* 2 */
   c) new□data = x
4. end if

End SLL_Insert_Begin



1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address.

2.2.2.2 Insertion of a node into SLL at ending

- To insert a node into SLL at ending first we need to traverse to last node, then insert as new node as last node.

Algorithm SLL_Insert_Ending(header,x)

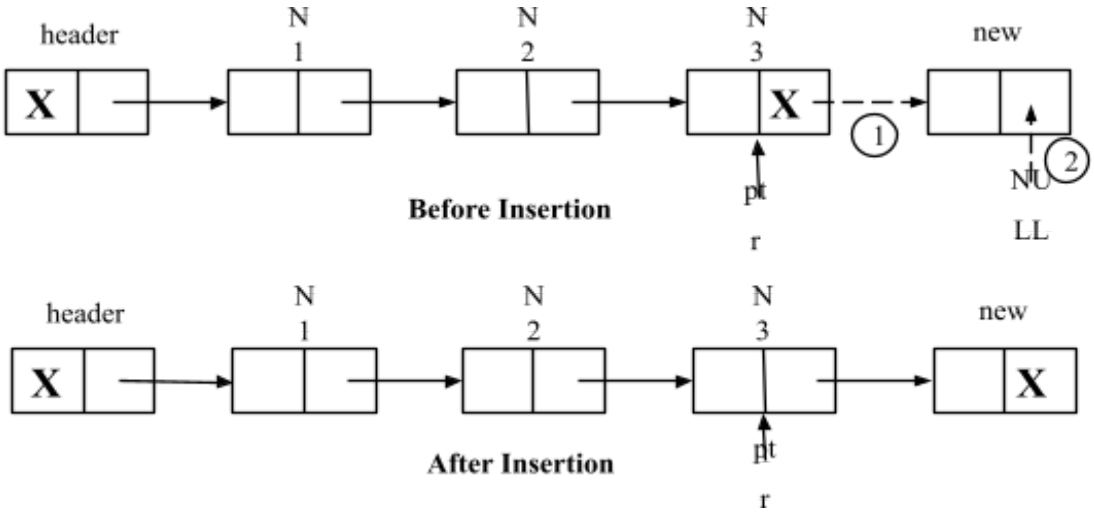
Input: *header* is a pointer to the header node, *x* is data part of new node to be insert.

Output: SLL with new node at ending.

1. new = getnewnode()
3. if(new==NULL)
   a) print "node not creates"

3. else
   a) ptr = header
   b) while(ptr□link != NULL)
      i) ptr = ptr□link
   c) end loop
   d) ptr□link = new /* 1 */
   e) new□link = NULL /* 2 */
   f) new□data = x
3. end if

End SLL_Insert_Ending



1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new node becomes the last node.

2.2.2.3 Insertion of a node into SLL at any position.

- For insertion of a node at any position in SLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

Algorithm SLL_Insert_ANY(header,x,key)

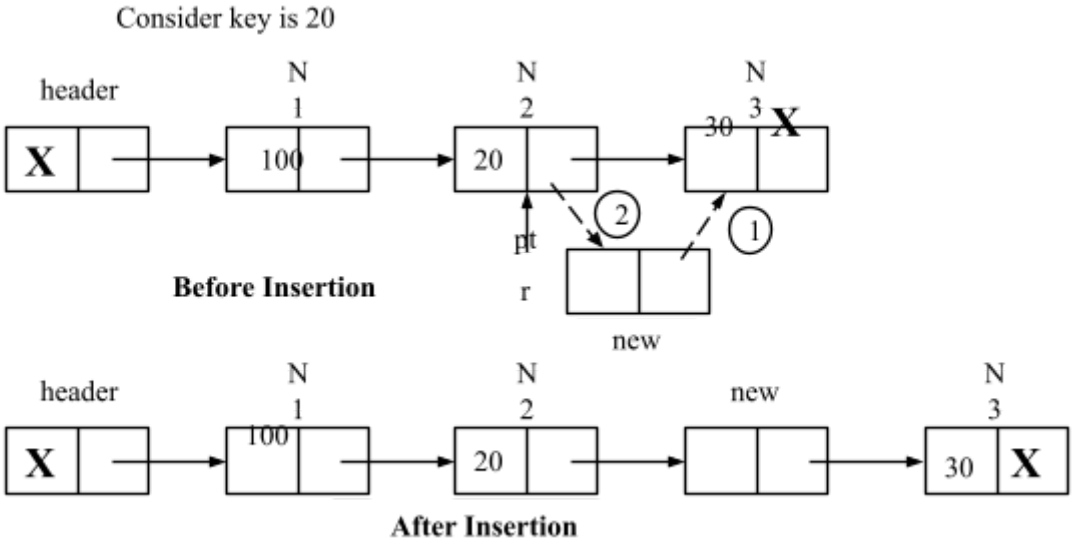
Input: *header* is a pointer to the header node, *x* is data part of new node to be inserting, *key* is the data part of a node, after this node we want to insert new node.

Output: SLL with new node inserted after the node with data part as specified key value.

1. new = getnewnode( )
4. if(new==NULL)
   a) print “node not creates”

3. else
   a) ptr = header
   b) while(ptr□link != NULL && ptr□data != key)
      i) ptr = ptr□link
   c) end loop
   d) if(ptr□link = = NULL && ptr□data != key)
      i) print"Required node with data part as key value is not available, so unable to process"
   e) else
      i) new□link = ptr□link /* 1 */
      ii) ptr□link = new /* 2 */
      iii) new□data = x
   f) end if
4. end if

End SLL_insert_ANY



1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

2.2.3 Deletion of a node from SLL

- The deletion of a node in from SLL can be done in various positions.

  i) Deletion of a node from SLL at beginning.

  ii) Deletion of a node from SLL at ending.

  iii) Deletion of a node from SLL at any position.

2.2.3.1 Deletion of a node from SLL at beginning

- Deletion of a node from SLL at beginning means, delete the node which is after the header node.

Algorithm SLL_Delete_Begin(header)

Input: *header* is a pointer to the header node.

Output: SLL with node deleted at Beginning.

1. if(header□link = = NULL)

   a) print "SLL is empty, so unable to delete node from list"

2. else /*SLL is not empty*/

   a) ptr = header□link /* ptr points to first node into list*/

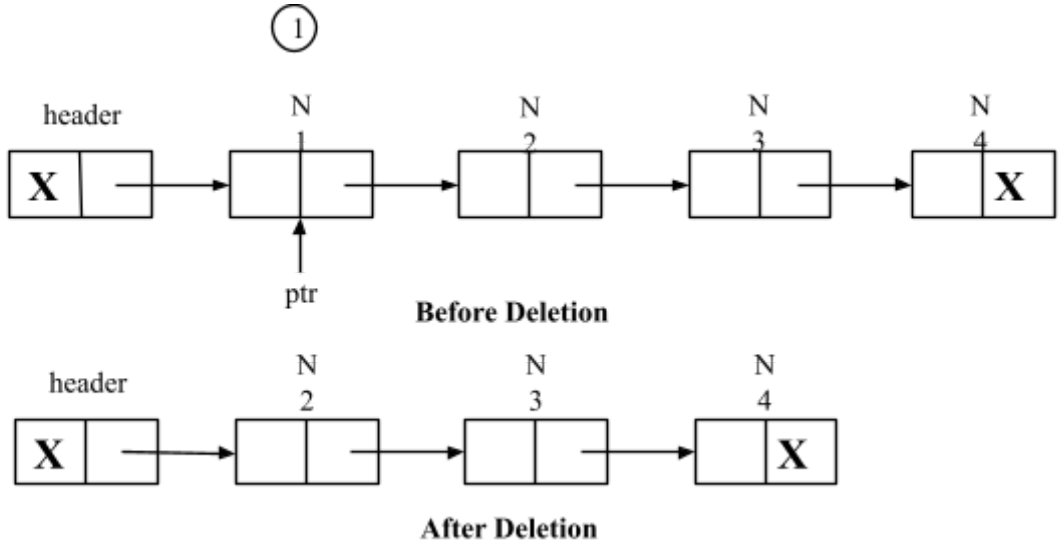
   b) header□link=ptr□link /* 1 */

   c) print "Deleted node is" ptr□data

   d) free(ptr) /*send back deleted node to memory bank*/

3. end if

End SLL_Delete_Begin



1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

2.2.3.2 Deletion of a node from SLL at ending

- To delete a node from SLL at ending, first we need to traverse to last node in the list. After reach the last node in the list, last but one node link part is replaced with NULL.

Algorithm SLL_Delete_End (header)

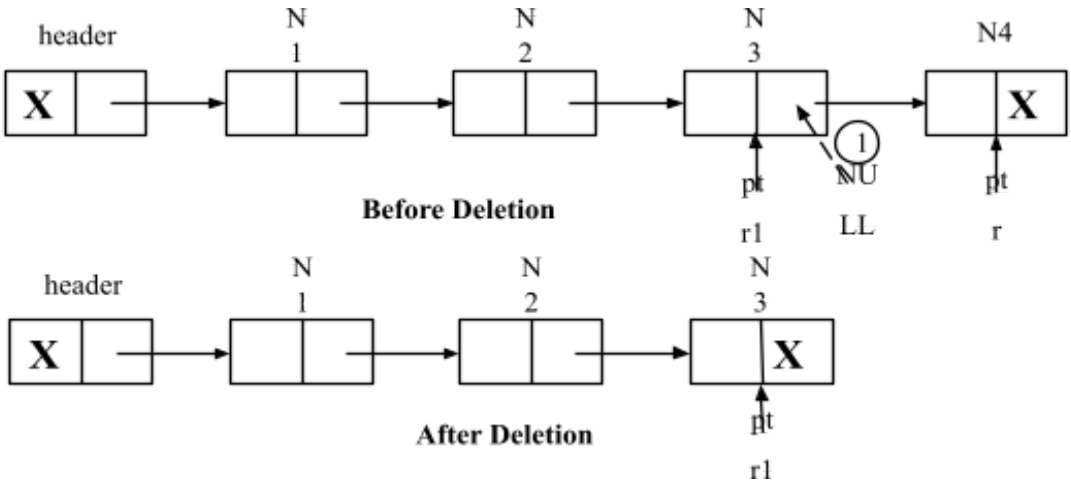
Input: *header* is a pointer to the header node.

Output: SLL with node deleted at ending.

```
1. if(header□link == NULL)
        a) print"SLL is empty, so unable to delete the node from list"
2. else                                  /*SLL is not empty*/
        a) ptr = header                  /*ptr  initially points to header node*/
        b) while(ptr□link!=NULL)
            i) ptr1 = ptr
            ii) ptr = ptr□link
        c) end loop
        d) ptr1□link = NULL              /* 1 */
    e) print "Deleted node is" ptr□data
        f) free(ptr)                     /*send back deleted node to memory bank*/
```

3. end if

End SLL_Delete_End



1. Link part of last but one node is replaced with NULL. Because after deletion of last node in the list, last but one node become the last node.

2.2.3.3 Deletion of a node from SLL at any position

- For deletion of a node from SLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

Algorithm SLL_Delete_ANY (header,key)

Input: *header* is a pointer to the header node, key is the data part of the node to be delete.

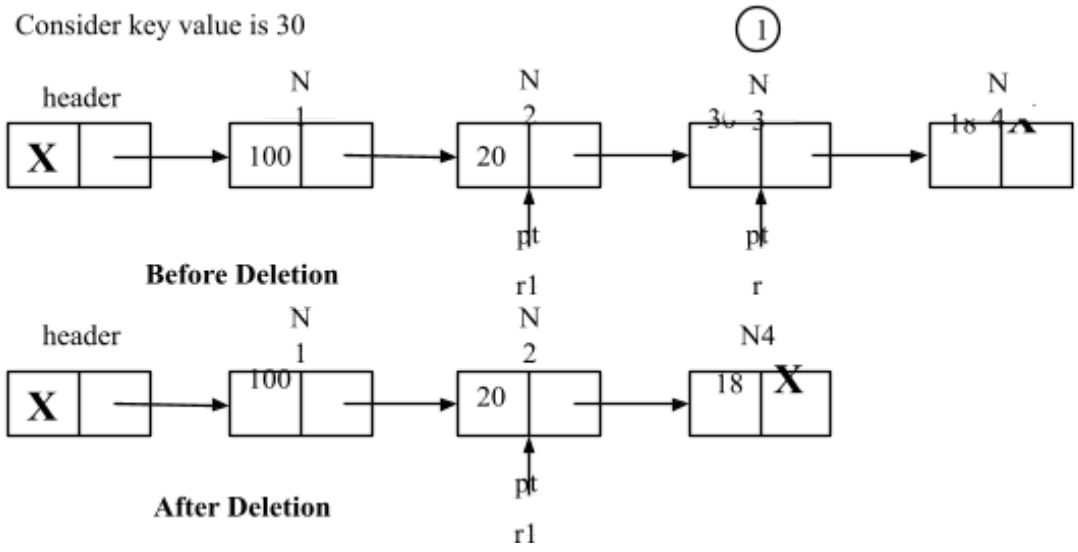
Output: SLL with node deleted at Any position. i.e. Required element.

1. if(header□link = = NULL)
   a) print "SLL is empty, so unable to delete the node from list"
2. else /*SLL is not empty*/
   a) ptr =header /*ptr initially points to header node*/
   b) while(ptr□link != NULL && ptr□data != key)
      i) ptr1 = ptr
      ii) ptr=ptr□link
   c) end loop
   d) if(ptr□link = = NULL && ptr□data != key)
      i) print "Required node with data part as key value is not available"
   e) else /* node with data part as key value available */
      i) ptr1□link = ptr□link /* 1 */
      ii) print "Deleted node is" ptr□data
      iii) free(ptr)

f) end if

3. end if

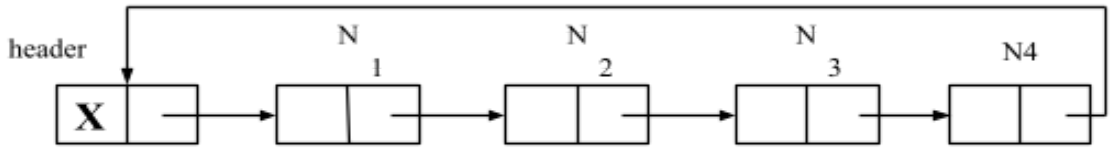
End SLL_Delete_ANY



1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.

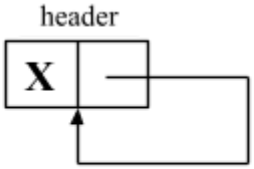
2.3 CIRCULAR LINKED LIST

- A linked list where last node link part points to *header* node is called as Circular Linked List.



Circular Linked List

- If a CLL is empty, then the link part of the header node points to itself.



Empty Circular Linked List

Operations on Circular Linked List

1. Traversing list
2. Insertion of a node in to CLL

3. Deletion of a node from CLL

2.3.1 Insertion of a node in to CLL

- The Insertion of a node in to CLL can be done in various positions.

  i) Insertion of a node into CLL at beginning.

  ii) Insertion of a node into CLL at ending.

iii) Insertion of a node into CLL at any position.

2.3.1.1 Insertion of a node into CLL at beginning

- Insertion of a node into CLL at beginning means insert new node after header node.

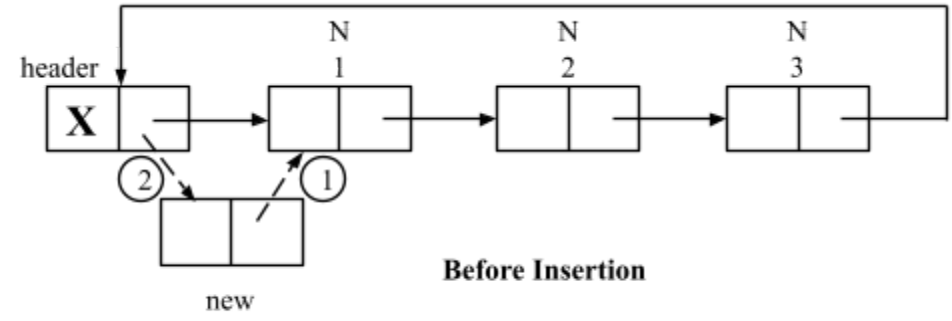
Algorithm CLL_Insert_Begin(header,x)

Input: *header* is pointer to header node, x is data part of new node to be inserting.

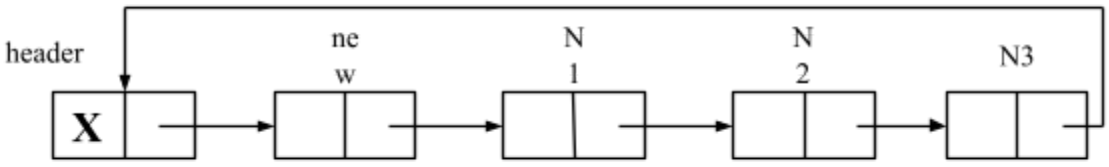
Output: CLL with new node inserted at beginning.

1. new = getnewnode( )
2. new□link = header□link /* 1 */
3. header□link = new /* 2 */
4. new□data = x

End CLL_Insert_Begin



Before Insertion



After Insertion

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.
2. Link part of header node is replaced with new node address.

2.3.1.2 Insertion of a node into CLL at ending

- To insert a node into CLL at ending first we need to traverse to last node, then insert as new node as last node.

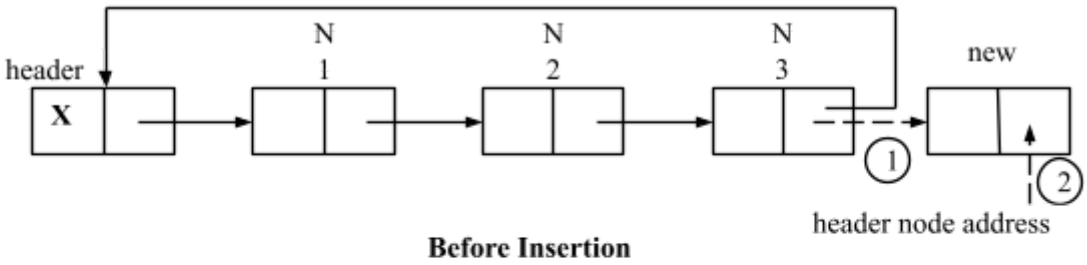
Algorithm CLL_ Insert_End(header,x)

Input: *header* is pointer to header node, *x* is data part of new node to be inserting.

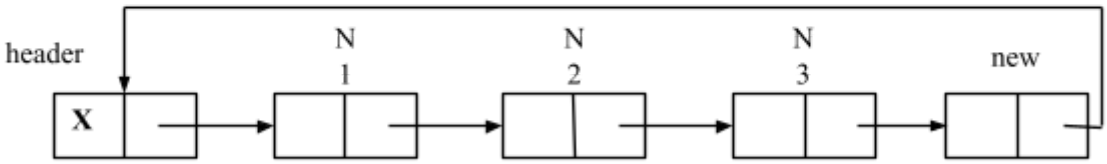
Output: CLL with new node inserted at ending.

1. new = getnewnode()
2. ptr = header
3. while(ptr□link != header)
   a) ptr = ptr□link
4. end loop
5. ptr□link = new
6. new□link = header
7. new□data = x

End CLL_Insert_End



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with address of header node, because new node becomes the last node.

2.3.1.3 Insertion of a node into CLL at any position

- For insertion of a node at any position in CLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

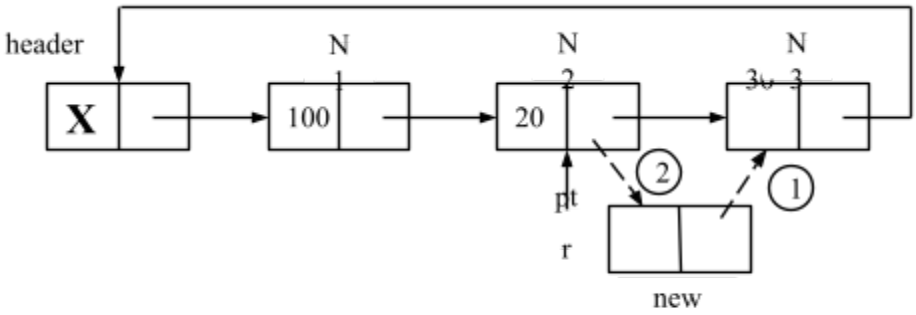
Algorithm CLL_Insert_ANY(header,x,key)

Input: header is pointer to header node, x is data pat of new node to be insert, key is the data part of a node, after this node we want to insert new node.

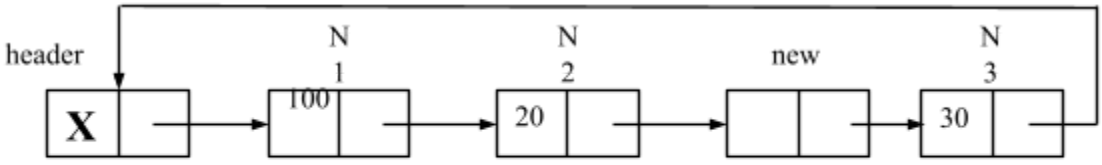
Output: CLL with new node inserted after the node with data part as specified key value.

1. new = getnewnode()
2. ptr=header
3. while(ptr□link != header && ptr□data != key)
    a) ptr = ptr□link
4. end loop
5. if(ptr□link = = header && ptr□data != key)
    a) print "required node with data part as key value is not available, so unable to process"
6. else
    a) new□link = ptr□link /* 1*/
    b) ptr□link = new /* 2 */
    c) new□data = x
7. end if

End CLL_Insert_ANY



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

2.3.2 Deletion of a node from CLL

- The Deletion of a node from CLL can be done in various positions.

  i) Deletion of a node from CLL at beginning

ii) Deletion of a node from CLL at ending

iii) Deletion of a node from CLL at any position

2.3.2.1 Deletion of a node from CLL at beginning

- Deletion of a node from CLL at beginning means, delete the node which is after the header node.

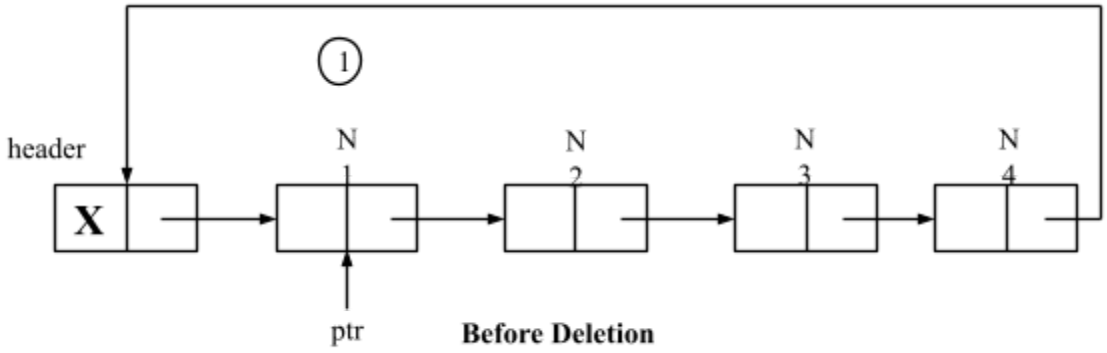
Algorithm CLL_Delete_Begin(header)

Input: *header* is pointer to header node.

Output: CLL with node deleted at Beginning.

1. if(header□link = = header)
   a) print "CLL is empty, so unable to delete node from list"
2. else /*DLL is not empty*/
   a) ptr = header□link /* ptr points to first node into list*/
   b) header□link = ptr□link /* 1 */
   c) print "deleted node is " ptr□data
   d) free(ptr) /*send back deleted node to memory bank*/
3. end if

End CLL_Delete_Begin



1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

2.3.2.2 Deletion of a node from CLL at ending

- To delete a node from CLL at ending, first we need to traverse to last node in the list. After reach the last node in the list, last but one node link part is replaced with header node address.

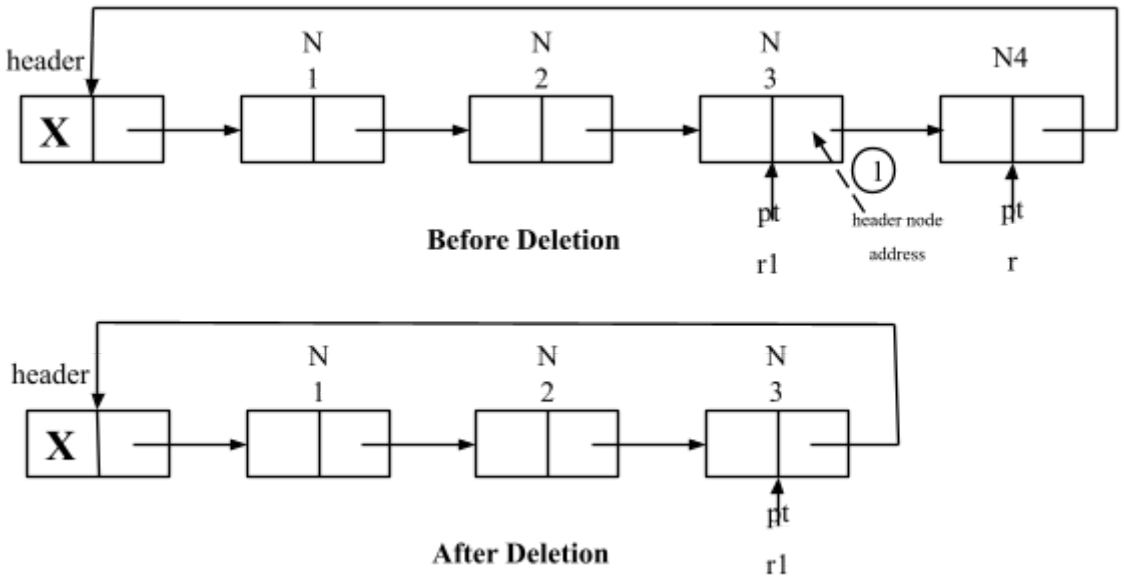
Algorithm CLL_Delete_End (header)

Input: *header* is pointer to header node.

Output: CLL with node deleted at ending.

```
1. if(header□link = = header)
        a) print "CLL is empty, so unable to delete the node from list"
2. else                                  /* CLL is not empty*/
        a) ptr=header                    /*ptr  initially points to header node*/
        b) while(ptr□link!=header)
              i) ptr1=ptr
              ii) ptr=ptr□link
        c) end loop
        d) ptr1□link=header              /* 1 */
        e) print "deleted node is" ptr□data
        f) free(ptr)                     /*send back deleted node to memory bank*/
3. end if
```

End CLL_Delete_End



1. Link part of last but one node is replaced with address of header node. Because after deletion of last node in the list, last but one node become the last node.

2.3.2.3 Deletion of a node from CLL at any position

- For deletion of a node from CLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

Algorithm CLL_Delete_ANY(header,key)

Input: *header* is pointer to header node, key is the data part of the node to be deleting.

Output: CLL with node deleted at Any position. i.e. Required element.

1. if(header□link = = header)
   a) print "CLL is empty, so unable to delete the node from list"
2. else /*CLL is not empty*/
   a) ptr=header /*ptr initially points to header node*/
   b) while(ptr□link != header && ptr□data != key)
      i) ptr1 = ptr
      ii) ptr = ptr□link
   c) end loop
   d) if(ptr□link = = header && ptr□data!=key)
      i) print "Required node with data part as key value is not available"
   e) else
      i) ptr1□link = ptr□link /* 1 */

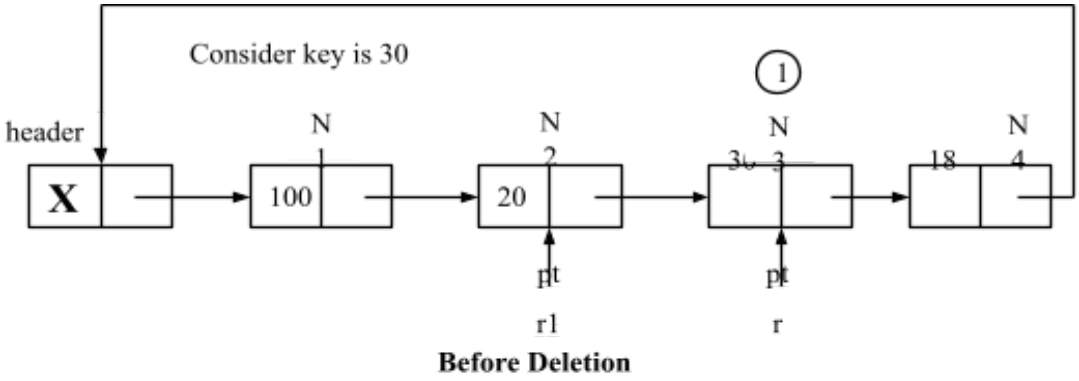
ii) print "deleted node is " ptr□data

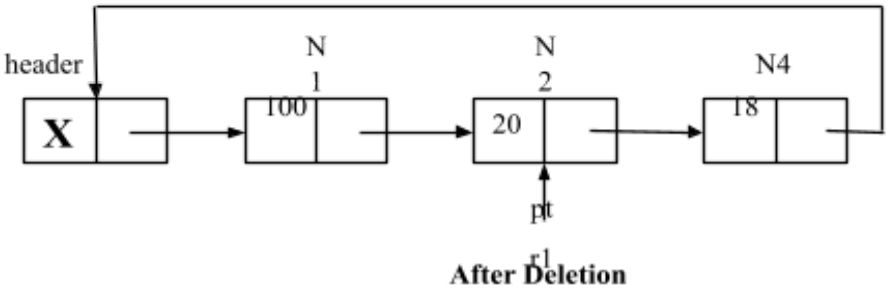
iii) free(ptr)

f) end if

3. end if

End CLL_Delete_ANY

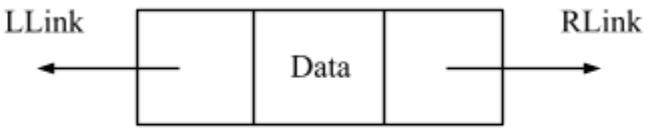




1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and *N4* becomes the next node for the node to be deleting.

2.4 DOUBLE LINKED LIST

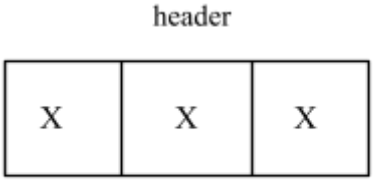
- In a SLL one can move from the header node t o any node in one direction only. i.e. from left to right.
- A DLL is a two way list. Because one can move either from left to right or right to left.
- In DLL, each node maintains two links.



Structure of a node in DLL

- Here LLink refers *Left Link* and RLink refers *Right Link*.
- The LLink part of a node in DLL always points to the previous node. i.e. LLink part of a node Consists address of previous node.

- The RLink part of a node in DLL always points to the next node. i.e. RLink part of a node Consists address of next node.
- If a DLL is empty, then the llink part and rlink of the header node is NULL.



Empty Double Linked List

Operations on Double Linked List

1. Traversing list
2. Insertion of a node in to DLL
3. Deletion of a node from DLL

2.4.1 Traversing a list

- Traversing a DLL means, visit every node in the list starting from first node to the last node.

Algorithm DLL_Traverse(header)

Input: header is the pointer to header node.

Output: Visiting of every node in DLL.

1. ptr = header
2. while(ptr□rlink != NULL)
   a) ptr = ptr□rlink
   b) print "ptr□data"
3. end loop

End DLL_Traverse

2.4.2 Insertion of a node in to DLL

- The Insertion of a node in to DLL can be done in various positions.
  i) Insertion of a node into DLL at beginning
  ii) Insertion of a node into DLL at ending
  iii) Insertion of a node into DLL at any position
- For insertion of a node into DLL, we must get node from memory bank. The procedure for getting node from memory bank is same as getting node for SLL from memory bank.

2.4.2.1 Insertion of a node into DLL at beginning

- Insertion of a node into DLL at beginning means insert new node after header node.

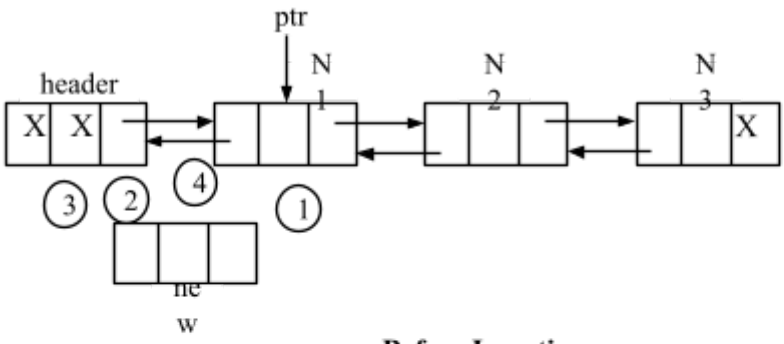
Algorithm DLL_Insertion_Begin(header,x)

Input: *header* is a pointer to the header node, *x* is data part of new node to be inserting.

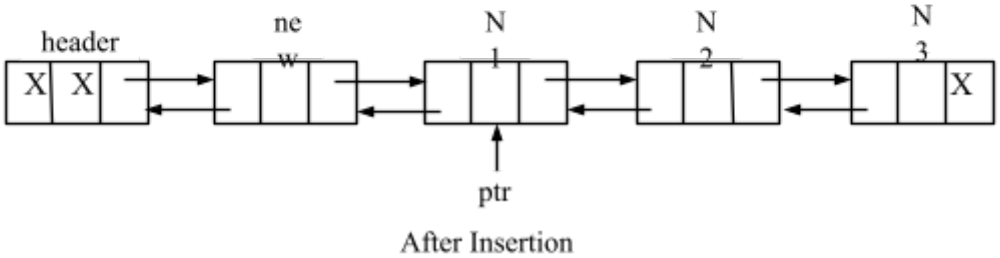
Output: DLL with new node at begin.

1. new = getnewnode()
2. if(header□rlink == NULL)
    a) header□rlink = new
    b) new□llink = header
    c) new□rlink = NULL
    d) new□data = x
3. else
    a) ptr = header□rlink
    b) new□rlink = ptr /* 1 */
    c) new□llink = header /* 2 */
    d) header□rlink = new /* 3 */
    e) ptr□llink = new /* 4 */
    f) new□data = x
4. end if

End DLL_Insertion_Begin



Before Insertion



After Insertion

1. *Rlink* part of new node is replaced with the address of first node in the DLL. i.e. address of first node is available in *Rlink* part of header node.
2. *Llink* part of new node is replaced with the address of header.
3. *Rlink* part of header node is replaced with the address of new node.
4. *Llink* part of previous first node is replaced with the address of new node.

2.4.2.2 Insertion of a node into DLL at ending

- To insert a node into DLL at ending, first we need to traverse to last node, then insert as new node as last node.

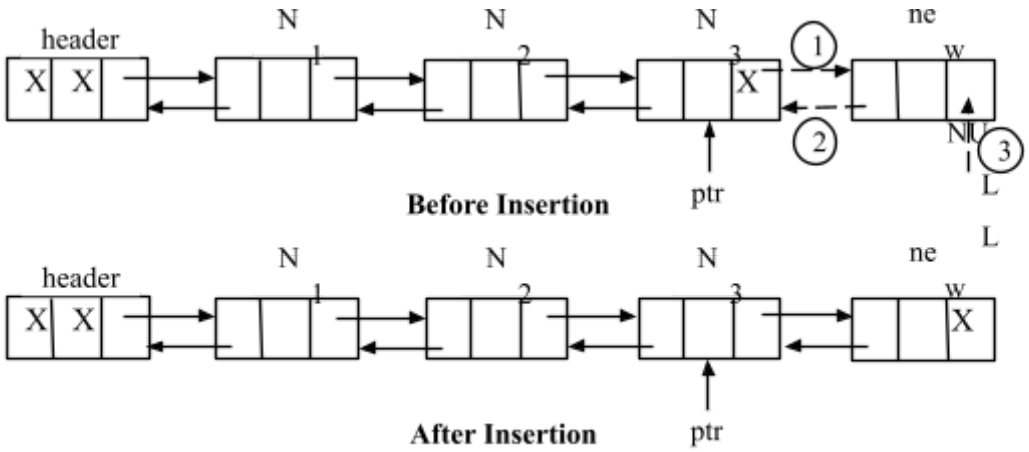
Algorithm DLL_ Insertion_Ending(header,x)

Input: *header* is a pointer to the header node,*x* is data part of new node to be inserting.

Output: DLL with new node inserted at the ending.

1. new = getnewnode()
   2. ptr = header
   3. while(ptr□rlink != NULL)
      a) ptr = ptr□rlink
   4. end loop
   5. ptr□rlink = new /* 1 */
   6. new□llink = ptr /* 2 */
   7. new□rlink = NULL /* 3 */
   8. new□data = x

End DLL_Insertion_Ending



1. *RLink* part of last node in the DLL is replaced with address of new node.
2. *LLink* part of new node is replaced with address of previous last node.
3. *RLink* part of new node is replaced with NULL. Because newly inserted node becomes the last node in the list.

2.4.2.3 Insertion of a node into DLL at any position

- For insertion of a node at any position in DLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

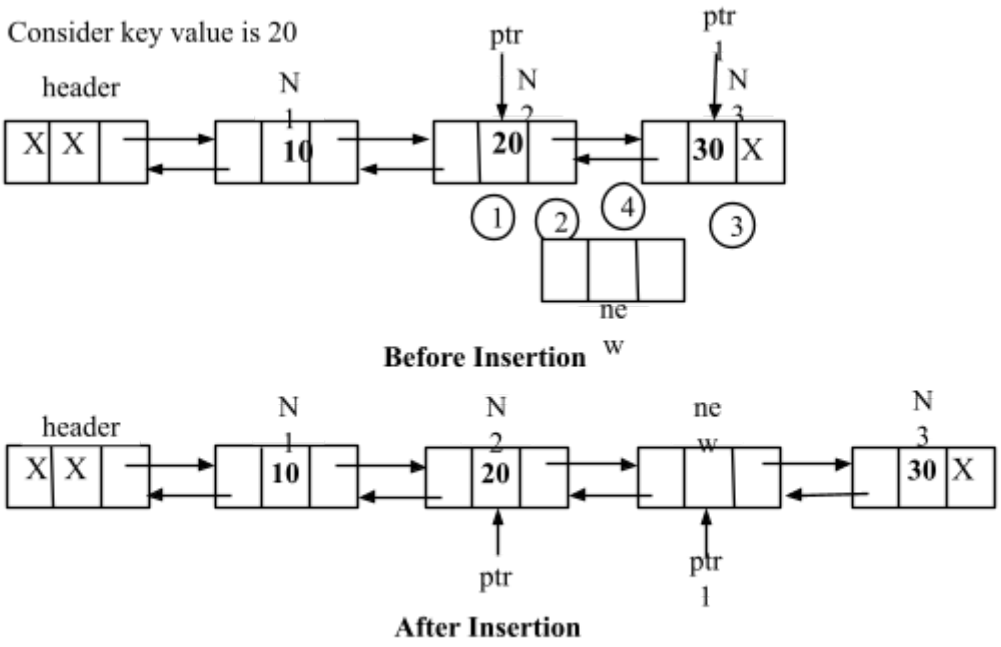
Algorithm DLL_Insertion_ANY(header,x,key)

Input: *header* is a pointer to the header node, *x* is data pat of new node to be inserting, key is the data part of a node, after that node new node is inserted.

Output: DLL with new node inserted after the node with data part as specified key value.

1. new = getnewnode()
2. ptr = header
3. while(ptr□rlink != NULL && ptr□data != key)
   a) ptr = ptr□rlink
4. end loop
5. if(ptr□rlink = = NULL && ptr□data != key)
   a) print "required node with key value was not available"
6. else if(ptr□rlink = = NULL && ptr□data == key)
   a) ptr□rlink = new
   b) new□llink = ptr
   c) new□rlink = NULL
   d) new□data = x
7. else
   a) ptr1 = ptr□rlink
   b) ptr□rlink = new /* 1 */
   c) new□llink = ptr /* 2 */
   d) new□rlink = ptr1 /* 3 */
   e) ptr1□llink = new /* 4 */
   f) new□data = x
8. end if

End DLL_Insertion_ANY



1. *RLink* part of previous node is replaced with address of new node.
2. *LLink* part of new node is replaced with the address of previous node. i.e. in the above example N2 becomes the previous node for newly inserting node.
3. *RLink* part of new node is replaced with the address of next node. i.e. in the above example N3 becomes the next node for newly inserting node.
4. *LLink* part of next node is replaced with address of new node.

2.4.3 Deletion of a node from DLL

- The deletion of a node in from DLL can be done in various positions.

  i) Deletion of a node from DLL at beginning.

  ii) Deletion of a node from DLL at ending.

  iii) Deletion of a node from DLL at any position.

2.4.3.1 Deletion of a node from DLL at beginning

- Deletion of a node from DLL at beginning means, delete the node which is after the header node.

Algorithm DLL_Deletion_Begin(header)

Input: header is pointer to header node

Output: DLL with node deleted at begin

1. if(header□rlink = = NULL)

   a) print "DLL is empty, not possible to perform deletion operation"

2. else

a) ptr = header□rlink

b) ptr1= ptr□rlink

c) header□rlink = ptr1 /* 1 */

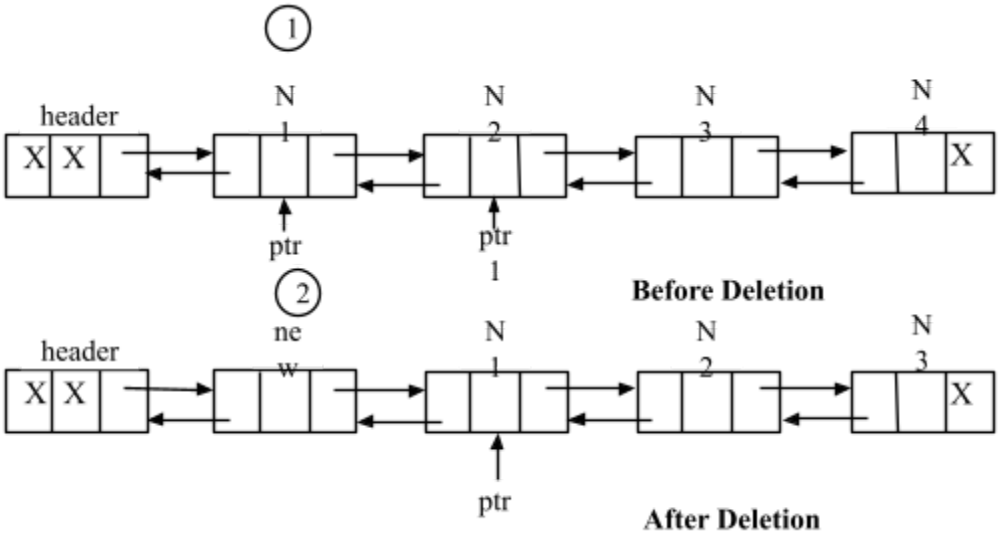
d) ptr1□llink = header /* 2 */

e) print "Deleted node is" ptr□data

f) free(ptr)

3. end if

End DLL_Deletion_Begin



1. RLink part of header node is replaced with the address of second node. i.e. address of second node is available RLink part of first node.
2. LLink part of second node is replaced with the address of header node.

2.4.3.2 Deletion of a node from DLL at ending

- To delete a node from DLL at ending, first we need to traverse to last node in the list. After reach the last node in the list, last but one node rlink part is replaced with NULL.

Algorithm DLL_Deletion_End(header)

Input: header is pointer to header node.

Output: DLL with deleted node at ending.

1. if(header□rlink = = NULL)

a) Print"DLL is empty, not possible to perform deletion operation"

2. else

a) ptr = header

b) while(ptr□rlink != NULL)

i) ptr = ptr□rlink

c) end loop

d) ptr1 = ptr□llink

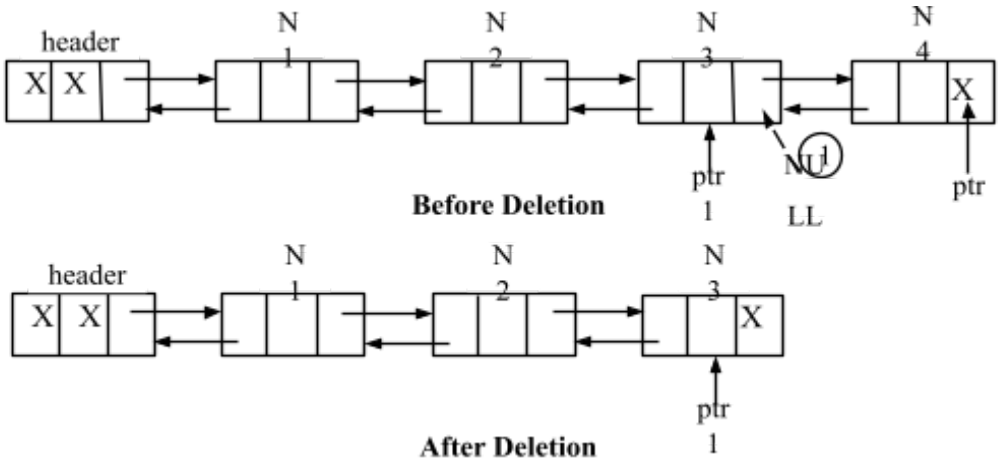
e) ptr1□rlink = NULL /* 1 */

f) print "Deleted node is" ptr□data

g) free(ptr)

3. end if

End DLL_Deletion_Ending



1. RLink part of last but one node in DLL is replaced with NULL. Because last but one node becomes last node.

2.4.3.3 Deletion of a node from DLL at any position

- For deletion of a node from DLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

Algorithm DLL_Deletion_Any(header,key)

Input: *header* is pointer to header node, key is the data part of a node to be delete.

Output: DLL without node as data part is key value.

1. if(header□rlink = = NULL)

    a) print "DLL is empty, not possible for deletion operation"

2. else

    a) ptr=header

    b) while(ptr□rlink != NULL && ptr□data != key)

        i) ptr = ptr□rlink

    c) end loop

    d) if(ptr□rlink = =NULL && ptr□data != key)

        i) print "required node was not available in list"

    e) else if(ptr□rlink = = NULL && ptr□data == key)

        i) ptr1 = ptr□llink

ii) ptr1□rlink = NULL

iii) print "Deleted node is" ptr□data

iv) free(ptr)

e) else

i) ptr1 = ptr□llink

ii) ptr2 = ptr□rlink

iii) ptr1□rlink = ptr2 /* 1 */

iv) ptr2□llink = ptr1 /* 2 */

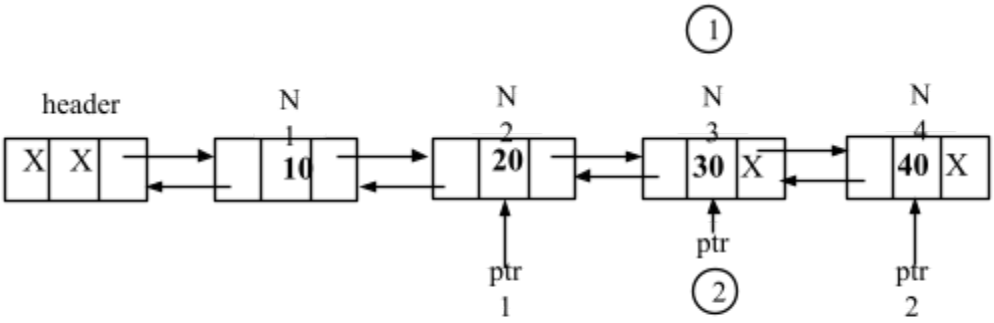
v) print "Deleted node is" ptr□data

vi) free(ptr)

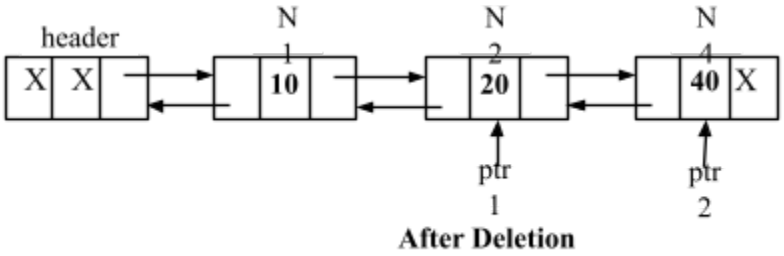
f) end if

3. end if

End DLL_Deletion_Any



Before Deletion



After Deletion

1. *RLink* part of previous node is replaced with the address of next node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.
2. *LLink* part of next node is replaced with the address of previous node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.

Comparison between SLL, CLL and DLL

Single Linked List	Circular Linked List	Double Linked List
1. in SLL, each node has one data part and one link part.	1. in CLL, each node has one data part and one link part.	1. in DLL, each node has one data part and two link parts.
2. Traverse from Left to Right.	2. Traverse from Left to Right in circular fashion.	2. Traverse from Left to Right and Right to Left.
3. Last node link part is NULL.	3. Last node link part points to address of header node..	3. Last node rlink part is NULL.
4. it require less memory when compared with DLL, because SLL has only one link part.	4. it require less memory when compared with DLL, because CLL has only one link part.	4. it require more memory when compared with SLL and CLL, because DLL has two link parts.