3rd Sem

35/-

Data Structures

Big 'O' notations - GARIMA / [redacted]

Algorithm - An algorithm is a well defined computational procedure that takes some value, or set of values as input and produces some value, or set of values as output.

An algorithm is a finite set of instructions which accomplish a particular task. Every algorithm must satisfy the following criteria:

i) Input - There are some (possibly empty) input data which are externally supplied to the algorithm.

ii) Output - There will be atleast one output.

iii) Definiteness - Each instruction of the algorithm must be clear and unambiguous.

iv) Finiteness - If we trace out the instruction of the algorithm then for all cases, the algorithm will terminate after a finite number of steps.

v) Effectiveness - The steps of an algorithm must be sufficiently basic that it can be carried out by a person using pen & paper.

Algorithm Design and Data Structure -

For solving a problem we must start with the following four tasks -

1. Identify the data ~~structure~~ items and their relationship.
2. Find the operations that must be performed on these data items.
3. Determine the best method to represent these data items in computer's memory.
4. Select a suitable programming language which can efficiently code the identified data representation.

Arrangement of data items is called data structures. Data structure can be defined as a way of organizing data items along with the logical relationships between them.

Complexity of Algorithm –

The complexity of the algorithm is determined in terms of Time & Space. When a program is executed, it uses resources of a computing system such as CPU, memory etc. Important thing is to determine the amount of time and storage it may require for execution.

Time and Space Complexity –

When we run a program on a computer, two most important consideration are:

→ Time Complexity
→ Space Complexity

Time Complexity – The time complexity of a program or algorithm is the amount of computational time it needs to completion. The time complexity is measured by counting the number of key operations in sorting and searching algorithms. The time complexity can be expressed as a function of number of key operations performed.

Space Complexity – The space complexity of program is the amount of memory that it needs to run to completion. The space is measured by counting the maximum memory needed by the algorithm in terms of variables used by algorithm.

There are three cases of complexity of an algorithm –

a) Worst case – The worst case of an algorithm is the function which defines the maximum number of steps taken for a data of size n.

b) Average case – The average case of the algorithm is the function which defines the average number of steps taken for a data of size n.

c) Best case – The best case of an algorithm is the function which defines the minimum number of steps taken for a data of size n.

Asymptotic Notation –

While doing analysis of algorithm more than the exact number of operations performed by the algorithm, it is the rate of increase in operations as the size of the problem increases that is of more importance.

We can observe that there isn't a significant difference in values when the input is small, but once the input value get large, there are big differences.

Algorithms can be grouped into three categories:

a) Algorithms that grow at least as fast as some function.

b) Algorithms that grow at the same rate.

c) Algorithms that grow no faster. $O(f)$

Big oh $O(f)$ category represents the class of functions that grow no faster than f. This means that for all values of n greater than some threshold n_0, all the functions in $O(f)$ have values that are no greater than f. Thus the class $O(f)$ has f as an upper bound, so none of the functions in this class grow faster than f. This means that, if $g(x) \in O(f)$, $g(n) < cf(n)$ for all $n > n_0$.

Algorithm & Analysis

An algorithm is a well-defined computational procedure that takes some value, or set of values as input and produces some value, or set of values as output. It is a sequence of computational steps that transform the input into the output.

Every algorithm must satisfy the following criteria –

1) Input – There are some (possibly empty) input data which are externally supplied to the algorithm.

2) Output – There will be atleast one output.

3) Definiteness – Each instruction / step of the algorithm must be clear and unambiguous.

4) Finiteness – If we trace out the instruction/step of an algorithm, then for all cases, the algorithm will terminate after ~~a finite~~ number of steps.

5) Effectiveness – The steps of an algorithm must be sufficiently basic that it can be carried out by a person mechanically using pen & paper.

Algorithm Design & Data Structure

For solving a problem we must start with the following four tasks:

1. Identify the data items and their relationship
2. Find the operations that must be performed on these data items.
3. Determine the best method to represent these data items in computer's memory.
4. Select a suitable programming language which can efficiently code the identified data representation

Arrangement of these data items is called data structure.

Data structure can be defined as a way of organizing data items along with these logical relationships between them.

Introduction to Data Structure -

A computer is a machine that manipulates information. The study of computer science includes the study of how information is organized in a computer, how it can be manipulated, and how it can be utilized.

Data + Structure
+ container

Main Objectives -

1) Identify and create useful mathematical entities and operations for the problem which need to be solved.
2) Determine the representation of entities and the implementation of operations on these problems.

Data Structure represents the knowledge of data to be organized in memory.

Data Type -

* A method of interpreting a bit pattern is often called a data type.
* A set of values on which some operations are performed is called a data type.

Primitive Data Types – These are the basic data types of any programming language that form the basic unit for the data structure. They define how the data will be internally represented in, stored and retrieved from the memory.

examples –

i) Integer (int)
ii) Character (char)
iii) Floating point (float)
iv) Double precision Number (double)

Abstract Data Types (ADT) – ADT is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.

* Description of the way in which components are related to each other.
* The operations that can be performed on that data type.

example - Stack –

Classification of Data Structure –

Computer can manipulate only primitive data in term of 0's and 1's, which is inherent within the computer. In real life applications, various kinds of data other than the primitive data are involved. Manipulation requires the following essential tasks:-

1) Storage representation of user data – should be in understandable form
2) Retrieval of storage data – should be in understandable form.
3) Transformation of user data –

10/1/08

GROWTH OF FUNCTIONS

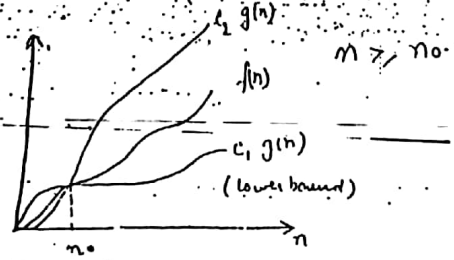
→ The order of growth of runnig time of an algorithm w.r.t the size of input gives a simple characterisation of the algorithm efficiency. In this computations, precision is not req^{d}. For large value of n, the ~~most~~ multiplicative constant and lower order terms are dominated by the effects of the input size itself.

→ To represent the order of growth of an algo for sufficiently large value of n, ~~Asymptotic~~ asymptotic notations are used.

Θ notation

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 s.t.

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$



(9)

eg 1: Let $f(n) = 3n + 2$

now we have to find out the value of c_1, c_2 & n_0 s.t.

$0 \leq c_1 g(n) \leq (3n+2) \leq c_2 g(n)$

$\Rightarrow$ Let $g(n) = n$

then $(3n+2) \leq 4n$

for $n_0 = 2$

and $(3n+2) \geq 3n$

for $n_0 = 1$

After combining these two we get

$c_1 = 3$

$c_2 = 4$

$n_0 = 2$

& $f(n) = \Theta(n)$.

eg 2: $f(n) = 100n + 6$

$f(n) = O(n)$ for $c = 101$, $n_0 = 6$

$f(n) = \Omega(n)$ for $c = 100$, $n = 1$

$\therefore f(n) = O(n)$ for $n_0 = 6$

eg 3: $f(n) = 10n^2 + 4n + 2$

$f(n) = O(n^2)$ for $c = 11$, $n_0 = 5$

$f(n) = \Omega(n^2)$ for $c = 10$, $n_0 = 1$

$\therefore f(n) = \Theta(n^2)$ for $c_1 = 10$, $c_2 = 11$, $n_0 = 5$

eg 4: $f(n) = 6 * 2^n + n^2$

$f(n) = O(2^n)$ for $c = 7$, $n_0 = 4$

$f(n) = \Omega(2^n)$ for $c = 1$, $n_0 = 1$

$f(n) = \Theta(2^n)$ for $c_1 = 1$, $c_2 = 7$, $n_0 = 4$

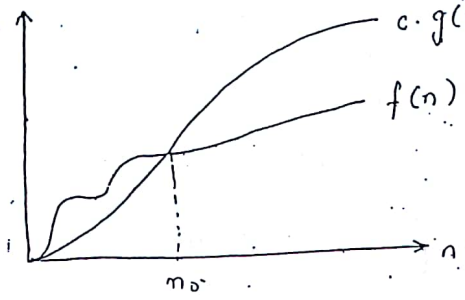
Theorem: If $f(n)$ & $g(n)$ be two fn s.t. $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then $f \in \Theta(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

Big Oh – it gives upper bound or worst case complexity of an algorithm. (11)

It can be defined as

$$O(g(n)) = \{ f(n): \text{there exist +ve const } c \text{ \& } n_0 \text{ s.t. } 0 \le f(n) \le c \cdot g(n) \text{ for all } n \ge n_0 \}$$



eg 1:- $f(n) = 3n + 2$

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

if $f(n) \le c \cdot g(n)$ for $n \ge n_0$

i.e. $(3n+2) \le 4 \cdot g(n)$ for $n \ge 2$

hence $f(n) = O(n)$ $3n+2 < cn$

eg 2:- $f(n) = 100n + 6$

Now $(100n + 6) \le c \cdot n$ for $n \ge n_0$

where $c = 101$

& $n_0 = 6$

eg3:- $f(n) = 10n^2 + 4n + 2$

$(10n^2 + 4n + 2) \leq 11n^2$ for $n \geq n_0$

here $n_0 = 5$

eg4:- $f(n) = 6 * 2^n + n^2$

$(6 * 2^n + n^2) \leq 7 * 2^n$ for $n \geq n_0$

here $n_0 = 4$

eg-5:- if $f(n) = a_m n^m + \ldots\ldots + a_1 n + a_0$

then $f(n) = O(n^m)$.

Proof:- $f(n) = \sum_0^m a_i n^i$

$= n^m \sum_{i=0}^m a_i n^{i-m}$

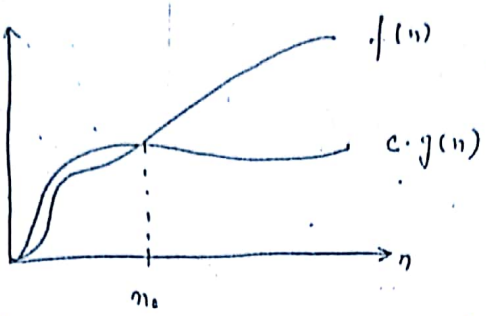
$\leq n^m \sum_{i=0}^m a_i$ for $n \geq 1$

$= O(n^m)$.

Big Omega - it gives lower bound of an algo & can be defined as

$\Omega(g(n)) = \{ f(n)$: There exist some +ve const c & n_0 s.t. $0 \leq c\,g(n) \leq f(n)$ for $n \geq n_0$.

(13)



eg1 : $f(n) = 3n+2$

$0 \leq 3n \leq (3n+2)$ for $n \geq 1$

$\therefore f(n) = \Omega(n)$

eg2 : $f(n) = 100n+6$

$100n \leq (100n+6)$ for $n \geq 1$

$\therefore f(n) = \Omega(n)$

eg3 : $f(n) = 10n^2 + 4n + 2$

$10n^2 \leq (10n^2+4n+2)$ for $n \geq 1$

$\therefore f(n) = \Omega(n^2)$

eg4 : $f(n) = 6 * 2^n + n^2$

$2^n \leq (6 * 2^n + n^2)$ for $n \geq 1$

$\therefore f(n) = \Omega(2^n)$

POINTERS

i &i

& → address of

* → value at address.

```
int i = 3
    &i  → 200
    i   → 3
   *(&i)
   *(200) → 3.
```

	i	j	k
	3	200	400
	200	400	600

j = &i;

```
int *j;
```

j = &i;

```
&i  → 6485
j   → 6485
&j  → 3276
j   → 6485
i   → 3
(&i) → 3
*j  → 3
```

```
&i  → 200
j   → 200
*k  → 200
&j  → 400
k   → 400
&k  → 600
j   → 200
k   → 400
i   → 3
*(&i) → 3
*j  → 3
**k → 3
```

& * Operator C → Row Major

Multidimensional Array a[x][y][z]

Row Major

a[i][j][k] = Base Addres + w(i*y*z + j*z + k).

= Base Addres + w(i*y*z + j*z + k).

a[x][y][z][m]

a[i][j][k][l] = b.A + w(i*y*z*m + j*z*m + k*m + l)

R.M

a[m][n]

a[i][j] = b.A + w(i*n + j).

CM

a[m][n].

a[i][j] = b.A + w(j*m + i).

a[m][n][o]

a[i][j][k] = b.A + ~~w(j*m*n + k*n~~

+ w(k*m*n + j*m + k).

① Column Major Order

② Row Major order.

$$Loc[LA[k]] = Base(LA) + w(k-1)$$

A[10]

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
200	202	204	206	208	210	212

A[7]	A[8]	A[9]
214	216	218

$= 200 + 2(5-0)$

$= 200 + 2(5)$

$= 200 + 10$

$A[5] = 210$

$m \times n$

$$Loc(A[J,K]) = Base(A) + w[M(K-1) + (J-1)]$$

$w \rightarrow$ no. of words per memory location

$A\left[\overset{m}{25} \times \overset{n}{4}\right]$ $w = 4$ $A(\overset{J}{12}, \overset{K}{3})$ $= Ba(A) + w[M(K$

$= 200 + 4[25(3-1) + (12-1)]$

$= 200 + 4(25 \times 2 + 11)$

$= 200 + 4(50 + 11)$

$= 200 + 4(61)$

$= 200 + 244 \Rightarrow 444$

$A[\overset{M}{2}, \overset{N}{2}]$ 36 $A[\overset{I}{1}, \overset{K}{2}]$

200 202

A[0,0] A[0,1] A[1,1] - A[1,2]

A[1,1] A[1,2] A[2,1] $A[\overset{I}{2}\ \overset{K}{2}]$

204 206

$= \boxed{200 + 2\,(2\,(1-0) + (2-0))} =$

$= 200 + 2\,(2+2)$ $= 200 + 2($

$= 200 + 2\,(4)$

$= 200 + 8$ $= 200 + 2\,(2(2-1) + 2-1)$

$= 208$ $= 200 + 2\,(2+1)$

$= 200 + 2\,(3)$

$= 200 + 6$

$= \boxed{206}$

L_i = upper bound − lower bound + 1

(2:8, −4:1, 6:10)

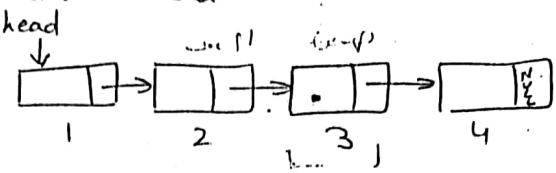
$L_1 = 8 - 2 + 1 = 7$

$L_2 = 1 + 4 + 1 = 6$ $L_1 \cdot L_2 \cdot L_3 = 210$ elements

$L_3 = 10 - 6 + 1 = 5$

Traversing a Linked List -

Step1 - Set pointer to head




```
Void delbet (int pos)
{
    struct node *temp;
    int count = 1;
    temp = head;
    if (head == NULL)
        printf ("No Link List");
    else
    {
        while (count != pos-1)
        {
            temp1 = temp;
            temp = temp->link;
            count++;
        }
        if (pos == 1)
        {
            head = temp->link;
            free (temp);
        }
        else
        {
            temp1->link = temp->link;
            free (temp);
        }
    }
}
```

SINGLY LINKED LIST (ADDITION)


```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>

struct node
{
    int data;
    int node *link;
} *head;

void addbeg(int data)
{
    struct node *temp;
    temp = (struct node *) malloc (sizeof(struct node));
    if (temp == NULL)
        printf("Not enough memory");
    else
    {
        temp -> data = data;
        temp -> link = head;
        head = temp;
    }
}

void addend(int data)
{
    struct node *temp, *temp1;
    temp = (struct node *) malloc (sizeof(struct node));
    if (temp == NULL)
        printf("Not enough memory");
    else
    {
        temp -> data = data;
        temp -> link = NULL;
        if (head == NULL)
            head = temp;
        else
        {
            if (head == NULL)
                head = temp;
```

```
void main()
{
    clrscr();
    head = NULL;
    addbeg(4);
    addbeg(3);
    addend(5);
    addend(6);
    display();
    getch();
}

void display()
{
    struct node *temp;
    temp = head;
    if (head == NULL)
        printf("No Linked list");
    else
    {
        while (temp != NULL)
        {
            printf("%d", temp->data);
            temp = temp -> link;
        }
    }
    else
    {
        temp1 = head;
        while (temp1->link != NU
            temp1 = temp1->link;
        temp1->link = temp;
    }
}
```

```
else
  {
    temp1 = head;
    while (count != pos-1)
    {
      count++;
      temp1 = temp1->next;
    }
    temp->prev = temp1;
    temp->next = temp1->next;
    temp1->next->prev = temp;
    temp1->next = temp;
  }
}
}

void delbeg()
{
  struct node *temp;
  if (head == NULL)
    printf("No L.L");
  else
  {
    temp=head;
    if(temp->next == head)
    {
      head = NULL;
      free(temp);
    }
    else
    {
      head = temp->next;
      temp->next->prev = NULL;
      free(temp);
    }
  }
}

void delend()
{
  struct node *temp, *temp1;
  if (head == NULL)
  {
    printf("No Linked");
  }
  else
  {
    temp = head;
    if (temp->next == head)
    {
      head = NULL;
      free(temp);
    }
    else
    {
      while (temp->next != NULL)
        temp = temp->next;
      temp->prev->next = NULL;
      free(temp);
    }
  }
}
```

```
void delbet (int pos)
{
  struct node *temp;
  int i=0, count=1;
  temp = head;
  while (temp != NULL)
  {
    i++;
    temp = temp->next;
  }
  if (pos > i)
    printf("Not Possible");
  else
  {
    temp = head;
    if (pos == 1)
    {
      head = temp->next;
      temp->next->prev = NULL;
      free(temp);
    }
    else
    {
      while (count != pos)
      {
        count++;
        temp = temp->next;
      }
      temp->prev->next = temp->next;
      temp->next->prev = temp->prev;
      free(temp);
    }
  }
}
```

INSERTION


```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
struct node
{
  int data;
  struct node *prev;
  struct node *next;
} *head;

void addbeg(int data)
{
  struct node *temp;
  temp = (struct node *) malloc (sizeof (struct node));
  if (temp == NULL)
      printf ("Not enough m/m");
  else
  {
    temp->data = data;
    temp->prev = NULL;
    if (head == NULL)
    {
      temp->next = NULL;
      head = temp;
    }
    else
    {
      temp->next = head;
      head = temp;
      temp->next->prev = head;
    }
  }
}

void display()
{
  struct node *temp;
  if (head == NULL)
      printf ("No L L");
  else
  {
    temp = head;
    while (temp != NULL)
    {
      printf ("%d", temp->data);
      temp = temp->next;
```

```
void addend(int data)
{
  struct node *temp, *templ;
  temp = (struct node *) malloc (sizeof (struct node));
  if (temp == NULL)
      printf ("Not enough space");
  else
  {
    temp->data = data;
    temp->next = NULL;
    if (head == NULL)
    {
      temp->prev = NULL;
      head = temp;
    }
    else
    {
      templ = head;
      while (templ->next != NULL)
        templ = templ->next;
      temp->prev = templ->next;
      templ->next = temp;
    }
  }
}

void addbet (int data, int pos)
{
  struct node *temp, *templ;
  int tnode = 0, count = 1;
  temp = (struct node *) malloc ( );
  if (temp == NULL)
      printf ("Not m/m");
  else
  {
    temp->data = data;
    templ = head;
    while (templ != NULL)
    {
      tnode++;
      templ = templ->next;
    }
    if (pos > tnode)
      printf ("Not Possible");
    else
    {
      if (pos == 1)
      {
        temp->prev = NULL;
        temp->next =
```

Circular Linked List

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
struct node
{
    int data;
    struct node *link;
} *head;

void addbeg(int data)
{
  struct node *temp, *temp1;
  temp = (struct node *) malloc(sizeof(struct node));
  if (temp == NULL)
      printf("Not enough space");
  else
  {
      temp->data = data;
      if (head == NULL)
      {
          temp->link = temp;
          head = temp;
      }
      else
      {
          temp->link = head;
          temp1 = head;
          while (temp1->link != head)
          {
              temp1 = temp1->link;
          }
          head = temp;
          temp1->link = head;
      }
  }
}
void addend(int data)
{
  struct node *temp, *temp1;
  temp = (struct node *) malloc(sizeof(struct node));
  if (temp == NULL)
      printf("Not enough mem");
  else
  {
      temp->data = data;
      if (head == NULL)
      {
          temp->link = temp;
          head = temp;
      }
      else
      {
          temp->link = head;
          temp1 = head;
          while (temp1->link != head)
          {
              temp1 = temp1->link;
          }
          temp1->link = temp;
      }
  }
}
void addbet(int data, int pos)
{
  struct node *temp, *temp1;
  int count = 1, tnode = 1;
  temp = (struct node *) malloc(sizeof(struct node));
  if (temp == NULL)
      printf("Not enough space");
  else
  {
      temp->data = data;
      temp1 = head;
      while (temp1->link != head)
      {
          tnode++;
          temp1 = temp1->link;
      }
      if (pos == 1)
      {
          temp->link = head;
          head = temp;
          temp1->link = head;
      }
      else
      {
          if (tnode < pos)
              printf("Position does not exists");
          else
          {
              temp1 = head;
              while (count != pos-1)
              {
                  count++;
                  temp1 = temp1->link;
              }
              temp->link = temp1->link;
              temp1->link = temp;
          }
      }
  }
}
void display(C)
{
  struct node *temp;
  temp = head;
  if (head == NULL)
      printf("Link list does not exists");
  else
  {
      while (temp->link != head)
      {
          printf("%d", temp->data);
          temp = temp->link;
      }
      printf("%d", temp->data);
```

```
void delbeg()
{
  struct node *temp, *temp1;
  temp=head;
  if(head==NULL)
      printf("Link list does not exists");
  else
  {  temp=head;
     if(temp->link==head)
     {
        head=NULL;
        free(temp);
     }
     else
     {
        temp1=head;
        while(temp1->link!=head)
            temp1=temp1->link;
        head=temp->link;
        temp1->link=head;
        free(temp);
     }
  }
}
void delend()
{
  struct node *temp, *temp1;
  if(head==NULL)
     printf("No Link List");
  else
  {  temp=head;
     if(temp->link==head)
     {
        head=NULL;
        free(temp);
     }
     else
     {  while(temp->link!=head)
        {
           temp1=temp;
           temp=temp->link;
        }
        temp1->link=head;
        free(temp);
     }
  }
}
```

Circular Linklist

DEL

(23)

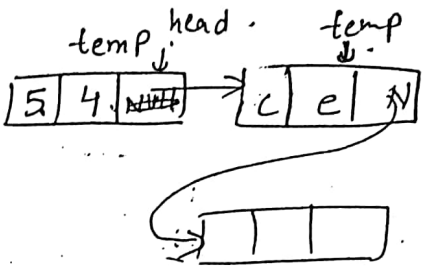
```
void delpos(int pos)
{
  struct node *temp, *temp1;
  int count=1, tnode=1;
  if(head==NULL)
     printf("No. L.L");
  else
  {
     temp=head;
     while(temp->link!=head)
     {
        tnode++;
        temp=temp->link;
     }
     if(tnode<pos)
        printf("position not exists");
     else
     {
        temp1=head;
        if(pos==1)
        {  head=temp1->link;
           temp->link=head;
           free(temp1);
        }
        else
        {
           temp=head;
           while(count!=pos)
           {
              count++;
              temp1=temp;
              temp=temp->link;
           }
           temp1->link=temp->link;
           free(temp);
        }
     }
  }
}
```

Linked lists and Polynomials —

$5x^4 + 2x^3 + 7x^2 + 10x - 8$.

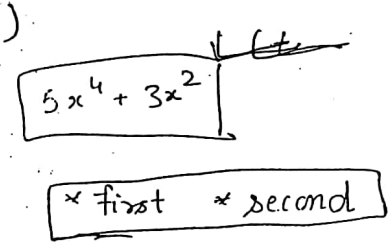
```
struct poly
{
    int coeff;
    int exp;
    struct poly * link;
} * head;
```

c = 5 e = 4.

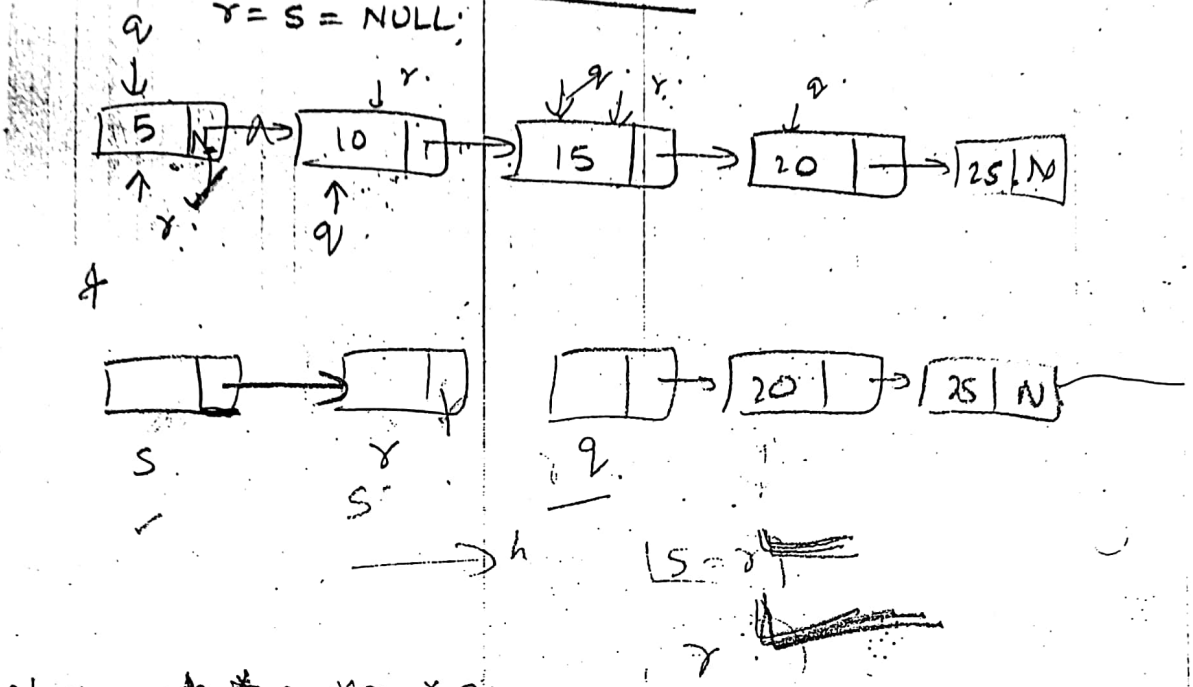



```
void display ( )
```

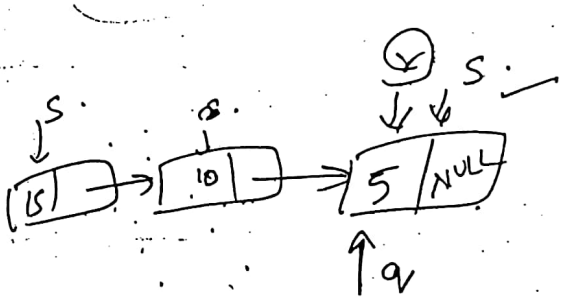
```
void poly_append ( int c, int e)
{
    struct poly * temp;
    temp = head;
    if (temp == NULL)
    {
        temp = malloc (sizeof (struct poly));
        temp = head;
    }
    else
    {
        while( temp -> link != NULL)
            temp = temp -> link;

        temp -> link = malloc (sizeof (struct poly));
        temp = temp -> link;
    }
    temp -> coeff = c;
    temp -> exp = e;
    temp -> link = NULL;
}
```

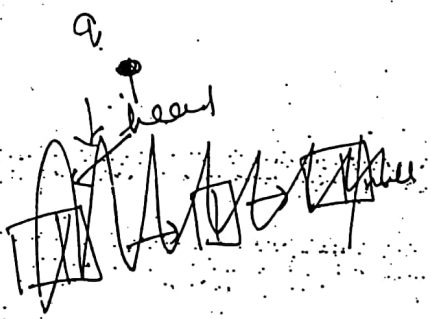


Reverse of linked list




```
struct node *q, *r, *s;
q = head;
r = NULL;
while (q != NULL)
{
    s = r;
    r = q;
    q = q -> link;
    r -> link = s;
}
head = r;
}
```



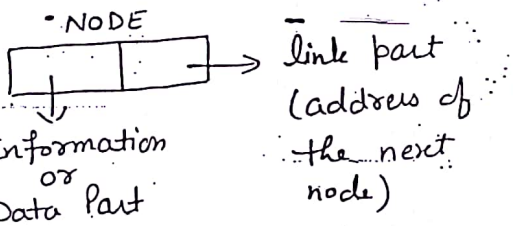


Linked List

Another way of storing a list in memory is to have each element in the list contain a field, called a link or pointer, which contains the address of the next element in the list. Successive elements in the list need not occupy adjacent space in memory.

A linked list, or one way 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: the first part contains the information of the element and the second part called the link field or nextpointer field, contains the address of the next node in the list.



Link Part of the last node in the link list always contain NULL (means not pointing to any other node)

Address of the first node is always contain in a special pointer variable called the 'head' node.

Search a node in the linked list

when list is unsorted.

Step 1 - Create two temporary (temp) & loc pointer variable.

Step 2 - Set temp = head (initialize it to the first node in the link list) & loc = NULL.

Step 3 - Repeat steps 4. while temp ≠ NULL.

Step 4 - If data = temp → data then
set loc = temp and exit.

Else
temp = temp → link (point the temp to the next node in the linked list).

Step 5 - Set loc = NULL & print Search is unsuccessful.

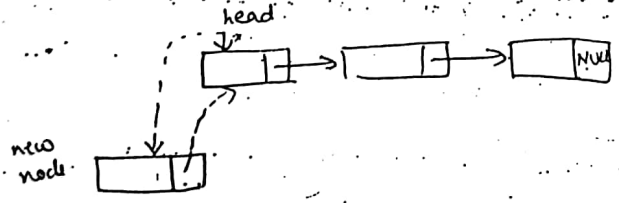
Step 6 - Exit.

Insertion of a Node in a Linked List

Linked List is a dynamic way to inserting an element one after the other and we can insert the elements in 3 major ways -

1) Insertion of a node in the begining of the linked list.
2) Insertion of a node at the end of the linked list.
3) Insertion of a node in between the nodes of the linked list.

1) Insertion in the begining



Step1 - Create a temporary (temp) pointer variable & allocate memory. if temp = NULL. then print no space available and Exit.

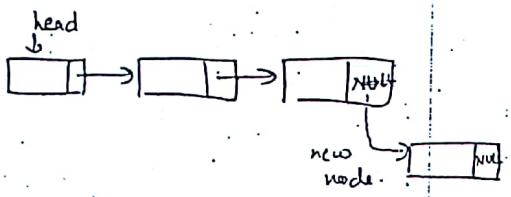
Step2 - Set temp -> data = num (copy the new data to the new node).

Step3 - Set temp -> link = head (copy the address of first node to the link part of new node.)

Step4 - Set head = temp (copy the address of ~~temp~~ new node to the head node (i.e. make the new node as the first node of the linked list)).

Step5 - Exit.

2) Insertion at the end.



Step1 - Create a temporary (temp) pointer variable & allocate memory of temp = NULL then print no space available and Exit.

Step2 - Create another temporary (trav) pointer variable and point it to the first node of the link list
trav = head.

Step3 - Repeat step4 while trav -> link ≠ NULL

step4 - trav = trav -> link i.e. point to the next node.

Step5 - Set temp -> data = num. (copy the new data to the new node)

Step6 - Set temp -> link = NULL (last node's link part should always contain NULL).

Step7 - Set trav -> link = temp (copy the address of new node to the last node's link part).

Step8 - Exit.

Traversing a Linked List –

Step1 – Create a temporary (temp) pointer variable.

Step2 – Set temp to the head of the linked list. i.e. first node of the linked list (head).

Step3 – Repeat steps 4 & 5 while temp ≠ NULL

Step4 – Do whatever you want to do on the data part of the node. e.g print the data part.

Step5 – set temp = temp → link i.e. set temp pointer to the next node in the linked list.

Step6 – Exit.

Counting the no. of nodes in the link list

Step1 – Create a integer variable (count) & initialize it to zero count = 0.

Step2 – Create a temporary (temp) pointer variable.

Step3 – Set temp = head (i.e. initialize temp to the first node in the link list).

Step4 – Repeat steps 5 & 6 while temp ≠ NULL

Step5 – count = count + 1

Step6 – Set temp = temp → link i.e. set temp to the next node in the link list.

Step7 – Exit.

(5) SINGELY LINK LIST (30)

Insertion of a node in the begining -

Step1 - Allocate free space
temp = create new node

Step2 - Check for free space
if temp == NULL print Not enough memory and exit

Step3 - ~~Read~~ Insert the data part into new node.
temp -> data = data

Step4 - Link address part of the new node with address of head
temp -> link = head

Step5 - Now assign address of newly created node to the head.
head = temp

Step6 - Exit

Insertion of a node in the end

Step1 - Allocate free space
temp = create new node

Step2 - Check for free space
if temp == NULL print Not enough memory and exit.

Step3 - Insert the data into new node i.e link part to NULL.
temp -> data = data , temp -> link = NULL

Step4 - Move the pointer to the end of the list
temp1 = head
Repeat while temp1 -> link ≠ NULL
temp1 = temp1 -> link

Step5 - Link ~~we~~ last node with the new node
temp1 -> link = temp

Step6 - Stop

Insertion of a Node at a specific location.

Step1 - Create a new node and check for availability of space

temp = create a new node

if temp == NULL print Not enough memory and Exit

Step2 - Insert the data

temp → data = data.

Step3 - Count the total number of node in tnode.

Step4 - If tnode < pos print position does not exists & exit.

Step5 - If pos == 1

temp → link = head

head = temp

Step6 - Traverse till that position. & change the link part

temp → link = temp1 → link.

temp1 → link = temp

Step7 - Exit.

Deletion of a Node from begining -

Step1 - Initialize a structure points which points to the first node in the list

temp = head

Step2 - Perform deletion operation

if temp == NULL print No Link list and Exit

Else

head = temp -> link

free the space associated with temp

free (temp)

Step3 - Exit.

Deletion of a Node from last -

Step1 - Initialize two structure pointes to the first node

temp = temp1 = head

Step2 - Check whether list is empty or not.

If temp = NULL print No Link list and Exit

Step3 - Reach to the last and second last node in list

while (temp -> link != NULL)

temp1 = temp

temp = temp -> link

Step4 - Perform deletion operation

temp1 -> link = NULL

free (temp)

Step5 - Exit

Deletion of a Node from a specific position

Step1 - Check whether the link list exists or not
if (head == NULL) print No Link list & exit.

Step2 - Initialize pointer to first node
temp = head

Step3 - Count the total number of nodes in list (tnode)

Step4 - Traverse till the position (temp) & position -1 (temp1)
while (count != pos)
temp1 = temp
temp = temp->link
count++

Step5 - Change the link part of position-1 node
temp1->link = temp->link.

Step6 - Free the node free(temp);

Step7 - If pos == -1
head = temp->link
& free(temp)

Step8 - Exit

CIRCULAR LINKED LIST

Insertion in the begining.

Step1 - Allocate free space & check for availability.
temp = create new node
if temp == NULL print Not enough memory & exit

Step2 - Insert the data to the new node.
temp -> data = data

Step3 - Link address part of the new node with the head
temp -> link = head

Step4 - Change the link of the last node with new node.
temp1 = head
while (temp1 -> link != head)
temp1 = temp1 -> link
temp1 -> link = temp

Step5 - Change the head node to point to the new node now
head = temp

Step6 - Exit

Insertion in the end.

Step1 - Allocate free space and check for availability
temp = create new node
if temp == NULL print Not enough memory & exit

Step2 - Insert the data ~~part~~ to the new node
temp -> data = data

Step3 - Link the address part of the new node with head
temp -> link = head.

Step4 - Move the pointer to the end of the list
temp1 = head
while (temp1 -> link != head)
temp1 = temp1 -> link

Step5 - Point the link part of the last node to the new node
temp1 -> link = temp

Step6 - Exit

Insertion of a node at a particular position

Step1 - Allocate free space and check for availability.

temp = create new node

if temp == NULL print Not enough memory & Exit.

Step2 - Insert the data to the data part.

temp -> data = data

Step3 - Count the total number of node and check for insertion.

if (tnode < pos) print Position does not exists

Step4 - Traverse to that position and add the new node

temp1 = head & count = 1

while (count != pos-1)

count++

temp1 = temp1 -> link

Add the new node

temp -> link = temp1 -> link

temp1 -> link = temp

Step5 - Exit

Deletion from begining.

Step1 - Initialize a pointer which points to the first node in the list
temp = head.

Step2 - Check whether ~~first node~~ link list exists or not
if (head == NULL) print No Link list and exit.

Step3 - Change the link part of the last node
temp1 = head
while (temp1 -> link != head)
temp1 = temp1 -> link

~~Step~~ - Change the head node
~~to~~ head = temp -> link
temp1 -> link = head
free the first node now
free (temp)

Step4 - Exit

Deletion from end.

Step1 - Initialize a pointer to point to first node in the list
temp = head

Step2 - Check whether link list exists or not
if (head == NULL) print No Link list and exit.

Step3 - Traverse the last and second last node of the list
~~temp1 =~~ while (temp -> link != NULL)
temp1 = temp
temp = temp -> link

Step4 - Change the link part of the second last node
temp1 -> link = head

Step5 - Free the last node.
free (temp)

Step6 - Exit.

Deletion from a particular position:

Step1 - Initialize a pointer to point to the first node of list
temp = head

Step2 - Check for the availability of link list
if (head == NULL) print No Link list & Exit

Step3 - Count the total number of nodes (tnode)

Step4 - Check whether position given exists or not
if (tnode < pos) print: Position does not exists

Step5 - Traverse till that position temp1 = head
if (pos == 1)
head = temp1 -> link;
temp -> link = head
free(temp)
else temp = head
while (count != pos)
count++
temp1 = temp
temp = temp -> link

Step6 - Change the link part of the previous node
temp1 -> link = temp -> link;
free(temp)

Step7 - Exit

DOUBLY LINK LIST (6)

Insertion in the begining

Step1 - Allocate free space and check for availability.
temp = create new node
if temp == NULL print Not enough memory & exit

Step2 - Insert the data and set the previous pointer to NULL
temp → data = data
temp → prev = NULL

Step3 - Set the next pointer of new node to the head
temp → next = head

Step4 - Change the head to the new node now
head = temp

Step5 - Change the prev pointer of second node to the new node now
temp → next → prev = head

Step6 - Exit

Insertion in the end

Step1 - Allocate free space and check for availability
temp = create new node
if temp == NULL print Not enough memory & exit

Step2 - Insert the data & set the next node to NULL
temp → data = data
temp → next = NULL

Step3 - Traverse till the last node and ~~change its next address part to the new node~~
temp1 = head
while (temp1 → next != NULL)
temp1 = temp1 → next

Step4 - ~~temp → nex~~ Point the prev part of new node to the last node & change the next pad of last node to

(b)

Insertion at a particular location

Step1 – Allocate free space & check for availability
temp = create new node
if temp == NULL print Not enough memory & exit.

Step2 – Insert the data part
temp → data = data.

Step3 – Count the number of node and (tnode)

Step4 – Check whether position is available or not
if (pos > tnode) print Not available

Step5 – If pos == 1 then set the prev to NULL & next to head
temp → prev = NULL
temp → next = head
head = temp.

Step6 – Traverse till the position templ=head while(count != pos-1)
count++
templ = templ → next

Step7 – Change the prev part & next part
temp → prev = templ
temp → next = templ → next
templ → next → prev = temp
templ → next = temp.

Step8 – Exit

(48)

Deletion from begining

Step1 - Check whether link list exists or not
if head == NULL print No Link list & exit.

Step2 - Initialize the pointer to first node
temp = head

Step3 - Change the head & change the prev part of ~~first~~ second node to NULL
head = temp → next
temp → next → prev = NULL

Step4 - Delete the first node
free (temp)

Step5 - Exit

Deletion from end

Step1 - check whether link list exists or not.
if head == NULL print No link list & exit.

Step2 - Initialize the pointer to first node
temp = head

Step3 - Traverse temp to the last node of link list

Step4 - Change the prev & next part of concerned nodes.
temp → prev → next = NULL

Step5 - Free the last node
free (temp)

Step6 - Exit

(10) 41

Deletion of a node from particular position.

Step1 - Initialize the pointer to first node
temp = head

Step2 - Count the total number of nodes in link list (tnode)

Step3 - Check whether position exists or not if not print Not possible & exit.

Step4 - If pos == 1
head = temp -> next
temp -> next -> prev = NULL

Step5 - Traverse till the position & change the prev & next part
temp -> prev -> next = temp -> next
temp -> next -> prev = temp -> prev.

Step6 - Free the node
free (temp)

Step7 - Exit

Name _____

Branch _____ Roll No. _____

Stack using Array 59 Signature of Invigilator

```
#include <stdio.h>
#include <conio.h>
#define MAX 10

struct stack
{
    int tos;
    int st[20];
} s;

void push (int data)
{
    if (s.tos == MAX-1)
        printf ("stack overflow");
    else
    {
        s.tos++;
        s.st[s.tos] = data;
    }
}

void pop( )
{
    if (s.tos == -1)
        printf ("stack underflow");
    else
    {
        printf("%d", s.st[s.tos]);
        s.tos--;
    }
}

void display
{
    int i;
    if (s.tos == -1)
        printf ("stack is empty");
    else
    {
        for (i=s.tos; i>=0; i--)
            printf ("%d ", s.st[i]);
    }
}

for (i=0; i<=TOS; i++)

void main()
{
    s.tos = -1;
    clrscr();
    push(5);
    push(6);
    display();
    pop();
    display();
    getch();
}
```

Stack using link list

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
    int data;
    struct node *link;
} *tos;

void push(int data)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    if(temp==NULL)
        printf("Stack overflow");
    else
    {
        temp->data = data;
        if (tos == NULL)
        {
            temp->link = NULL;
            tos = temp;
        }
        else
        {
            temp->link = tos;
            tos = temp;
        }
    }
}

void pop()
{
    struct node *temp;
    if(tos==NULL)
    {
        printf("Stack underflow");
    }
    else
    {
        temp = tos;
        if(temp->link==NULL)
        {
            tos = NULL;
            free(temp);
        }
        else
        {
            tos = temp->link;
            free(temp);
        }
    }
}

void display()
{
    struct node *temp;
    if (tos==NULL)
        printf("No stack yet");
    else
    {
        temp = tos;
        while(temp!=NULL)
        {
            printf("%d", temp->data);
            temp = temp->link;
        }
    }
}

void main()
{
    tos = NULL;
    clrscr();
    push(5);
    pop();
    display();
    getch();
}
```

Using Array

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
int queue[MAX];
int front=-1;
int rear=-1;
void insert(int data)
{
    if (rear == MAX-1)
        printf("Queue is full");
    else
    {
        if (rear==-1)
        {
            rear++;
            front++;
            queue[rear]=data;
        }
        else
        {
            rear++;
            queue[rear]=data;
        }
    }
}

void deleteq()
{
    if (front==-1)
        printf("Queue is empty");
    else
    {
        if(front==rear)
        {
            queue[front]=-999;
            front=-1;
            rear=-1;
        }
        else
        {
            queue[front]=-999;
            front++;
}}}

void display()
{
    int i;
    if(front==-1 && rear==-1)
        printf("Queue is empty");
    else
    {
        for(i=front; i<=rear ;i++)
            printf("%d", queue[i]);
    }
}

void main()
{
    clrscr();
    insert(2);
    deleteq();
    display();
    getch();
}
```

using linked list

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
    int data;
    struct node *link;
} *front, *rear;

void insert(int data)
{
    struct node *temp;
    temp = malloc();
    if (temp == NULL)
        printf("Queue is full");
    else
    {
        temp->data = data;
        temp->link = NULL;
        if (rear == NULL)
        {
            rear = temp;
            front = temp;
        }
        else
        {
            rear->link = temp;
            rear = temp;
        }
    }
}
void deleteq()
{
    struct node *temp;
    if (front == NULL)
        printf("Queue is empty");
    else
    {
        temp = front;
        if (front == rear)
        {
            front = NULL;
            rear = NULL;
            free(temp);
        }
        else
        {
            front = temp->link;
            free(temp);
        }
    }
}
void display()
{
    struct node *temp;
    temp = front;
    if (front == NULL)
        printf("Queue is empty");
    else
    {
        while (temp != NULL)
        {
            printf("%d", temp->data);
            temp = temp->link;
        }
    }
}
.....

void main()
{
    clrscr();
    front = NULL;
    rear = NULL;
    insert(1);
    insert(2);
    deleteq();
    display();
    getch();
}
```

Circular Queue using Array

```
#define MAX 10
#include <stdio.h>
#include <conio.h>

int queue[MAX];
int front = -1;
int rear = -1;

void insert(int data)
{
    if ((rear==MAX-1) && (front==0)) || (rear==front-1))
            printf("\n Queue is full");
    else
    {
        if (rear == -1)
        {
            rear++;
            front++;
            queue[rear] = data;
        }
        else
        {
            if ((rear== MAX-1) && front != 0))
            {
                rear = 0;
                queue[rear] = data;
            }
            else
            {
                rear++;
                queue[rear] = data;
            }
        }
    }
}

void display()
{
    int i;
    if (rear == -1 && front == -1)
        printf("Queue is empty");
    else
    {
        if (rear < front)
        {
            for(i= front; i<=MAX-1 ; i++)
                printf("%d", queue[i]);
            for (i=0; i<= rear; i++)
                printf("%d", queue[i]);
        }
        else
        {
            for(i=front; i<=rear; i++)
                printf("%d", queue[i]);
        }
    }
}

void deleteq()
{
    if (front == -1)
        printf("Queue is empty");
    else
    {   if(front == rear)
        {   queue[front] = -999;
            front = -1;
            rear = -1;
        }
        else
        {   if(front == MAX-1 && rear<front)
            {   queue[front] = -999;
                front = 0;
            }
            else
            {   queue[front] = -999;
                front++;
            }
        }
    }
}
```

Conversion of Infix to Postfix Expression

Step1 - Add "(" to the starting of expression and ")" to the end of expression.

Step2 - Start scanning from left to right and repeat steps 3 to 6 for each element of the expression until stack is empty.

Step3 - If an operand is encountered add it to the target string.

Step4 - If a left parenthesis is encountered, push it onto stack.

Step5 - If an operator is encountered then

a) Add the operator on to the stack.

b) If there is another operator onto the top of the stack and new operator has less or equal precedence then POP then POP the stack and add it on to target string & push the new operator on to the top of stack.

Step6 - If a right parenthesis is encountered, then

a) POP from stack and add to target string till a left parathesis encounters.

b) Remove the left parthesis

Step7 - Exit

Conversion of Infix to Prefix Expression. 63

Step1 – Add "(" to the starting of expression and ")" to the end to expression.

Step2 – Scan the expression from right to left and repeat steps 3 to 6 for each elemnt until stack is empty.

Step3 – If an operand is encountered add it to the target string.

Step4 – If a right parenthesis is encountered, push it onto stack.

Step5 – If an operator is encountered then:

a) If there is another operator onto the top of stack and the new operator has less precedence then POP the stack and add it on to target string & Push the new operator onto the top of stack.

b) else Push the operator onto the stack.

Step6 – If a left parenthesis is encountered then

a) Remove the left parenthesis

b) ~~If the encountered operator has less precedence~~ Repeatedly Pop from the stack and add it to the target string until a right paretheis encounters.

c) Remove the right parentheis.

Evaluation of Postfix Expression–

Step1. Scan the expression from left to right and repeat steps 2 & 3 for each elemnt ~~of R~~ until the stack is empty.

Step2 – If an operand is encountered, put it onto stack.

Step3 – If an operator is encountered, then

a) Pop two top elents from the stack.

b) Evaluate the expression formed by the two operands and the opirator.

c) Push the result of b) back on stack.

Step4 Exit

Conversion of Infix to Postfix Expression

Step1 - Add "(" to the starting of expression and ")" to the end of expression.

Step2 - Start scanning from left to right and repeat steps 3 to 6 for each element of the expression until stack is empty.

Step3 - If an operand is encountered add it to the target string.

Step4 - If a left parenthesis is encountered, push it onto stack.

Step5 - If an operator is encountered then

a) Add the operator on to the stack.

b) If there is another operator onto the top of the stack and new operator has less or equal precedence then POP then POP the stack and add it on to target string & push the new operator onto the top of stack.

Step6 - If a right parenthesis is encountered, then

a) POP from stack and add to target string till a left parenthesis encounters.

b) Remove the left parthesis

Step7 - Exit

INFIX TO POSTFIX

CSE

HIGHER PRIORITY :- *, /, % expo → ^(↑)

LOWER PRIORITY :- +, -

A - (B / C + (D % E * F) / G) * H

(A - (B / C + (D % E * F) / G) * H))

Infix character Scanned	STACK	Postfix Expression
(	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	AB
/	(- (/	AB
C	(- (/	ABC
+	(- (+	ABC/
(	(- (+ (	ABC/
D	(- (+ (	ABC/D
%	(- (+ (%	ABC/D
E	(- (+ (%	ABC/DE
*	(- (+ (% *	ABC/DE
F	(- (+ (% *	ABC/DEF
)	(- (+	ABC/DEF*%
/	(- (+ /	ABC/DEF*%
G	(- (+ /	ABC/DEF*%G
)	(-	ABC/DEF*%G/+
*	(- *	ABC/DEF*%G/+
H	(- *	ABC/DEF*%G/+H
)		ABC/DEF*%G/+H*-

INFIX TO PREFIX

CSE

(A-(B/C+(D%E*F)/G)*H)

)H)G/)F*E%D(+C/B(-AC.

Character Scanned	Stack	Prefix Exp.
)	)	
H	)	H
*	)*	H
)	)*)	GH
G	)*)	GH
/	)*)/	GH
)	)*)/)	FGH
F	)*)/)	FGH
*	)*)/)*	EFGH
E	)*)/)*	*EFGH
%	)*)/)%	D*EFGH
D	)*)/)%	%D*EFGH
(	)*)/	/%D*EFGH
+	)*)+	C/%D*EFGH
C	)*)+	C/%D*EFGH
/	)*)+/	BC/%D*EFGH
B	)*)+/	+/BC/%D*EFGH
C	)*	*+/BC/%D*EFGH
-	)-	A*+/BC/%D*EFGH
A	)-	-A*+/BC/%D*EFGH
(	NULL	

Evaluate Postfix Expression :→

$9-((3*4)+8)/4$

The infix expression $9-((3*4)+8)/4$ can be written as $9\ 3\ *\ 8\ +\ 4\ /\ -$ using Postfix notation.

Evaluation of Postfix Expression

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4.

i) ^($) exponential ⇒ highest Priority

ii) *, /, % ⇒ higher Priority

iii) +, - ⇒ Lower Priority

In case of infix to prefix expression if there is any operator of higher priority or same priority on the stack, then it will first add to the prefix expression.

In case of infix to postfix expression if there is any operator of any higher priority on the stack, then it will first add to the postfix expression. If any operator is of same priority operators then both the operator will be on the stack.

Que:- 4 $ 2 * 3 - 3 + 8 | 4 | - (1 + 1) - Infix to Prefix
Infix to Postfix

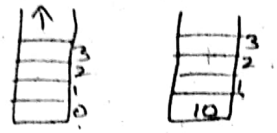
Que: 5 6 2 + * 1 2 4 | - Evaluate Postfix Expression.

4$2*3-3+8/4/(1+1) 67

Character Scanned	Stack	Postfix
4		4
$	$	4
2	$	42
*	*	42$
3	*	42$3
-	-	42$3*
3	-	42$3*3
+	+	42$3*3-
8	+	42$3*3-8
/	+/	42$3*3-8
4	+/	42$3*3-84
/	+/	42$3*3-84/
(	+/(	42$3*3-84/
1	+/(	42$3*3-84/1
+	+/(	42$3*3-84/1
1	+/(	42$3*3-84/1
)	+/	42$3*3-84/1+
		42$3*3-84/1+/+

49

Stacks



top = -1
stack is empty

10 pushed
top = 0

Push & Pop Operation

Representation of Stack

1) Using Arrays. 2) Using Link list.

$A - B / (C * D \wedge E)$

$A + (B * C - (D / E \wedge F) * G) * H$

5 6 2 + * 12 4 / −

Symbol Scanned	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
	37 Result.