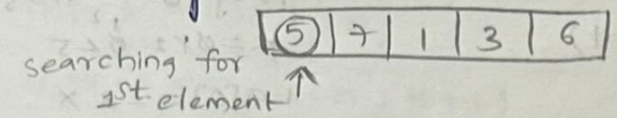
Unit - 5 (Searching - Sorting)

⇒ To understand how algorithms perform in different conditions, we use best, average and worst case analysis.

1) Best case analysis: The best case is when an algorithm performs the minimum number of steps on input data.

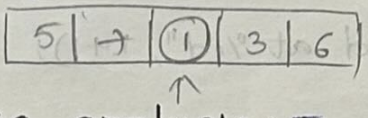
ex: In a linear search algorithm, the best case is when the target element is the first element in the array.



* O(1) - order of 1

2) Average case analysis: The average case is when an algorithm performs an average number of steps on input data.

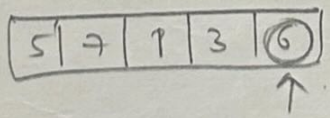
ex: The average case is searching middle element or nearly.



* O(n) - order of n

3) Worst case analysis: The worst case is when an algorithm performs the maximum number of steps on input data.

ex: In linear search algorithm, worst case is when the target element is at the end of the array or not present at all.



* O(n) - order of n

Searching

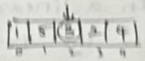
There are two popular methods for searching the array elements.

1) linear search

2) binary search

1) Linear search: Linear search, also called as sequential search, used for searching an array for a particular value.
- It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.
- mostly used for unordered list of elements (not sorted)

ex:
```
void lsearch(int el)
{
 int k, flag=0;
 for(k=0; k<n; k++)
 {
  if(a[k]==el)
  {
   flag=1;
   break;
  }
 }
 if(flag)
 printf("element found at %d", k+1);
 else
 printf("not found");
}
```



flag=0 n=5
k=0 1 2 3 4 5
1 a[0]==3 ✗
5 a[1]==3 ✗
3 a[2]==3 ✓

//wap to implement linear searching in an array
```
#include<stdio.h>
#include<conio.h>
void lsearch(int);
int a[10], n;
void main()
{
 int ele, i;
 clrscr();
 printf("enter no of elements in an array\n");
 scanf("%d", &n);
 printf("enter array elements\n");
 for(i=0; i<n; i++)
 scanf("%d", &a[i]);
 printf("array elements are\n");
 for(i=0; i<n; i++)
 printf("%d\t", a[i]);
 printf("\n enter ele to search\n");
 scanf("%d", &ele);
 lsearch(ele);
 getch();
}
void lsearch(int el)
{
 int k, flag=0;
 for(k=0; k<n; k++)
 {
  if(a[k]==el)
  {
   flag=1;
   break;
  }
 }
 if(flag)
 printf("element %d is found at %d\n", el, k+1);
 else
 printf("element not found");
}
```

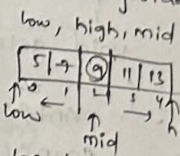
//wap to implement linear searching in an array (using functions).
```
#include<stdio.h>
#include<conio.h>
void lsearch(int);
void getdata();
void putdata();
int a[10], n, i;
void main()
{
```

```
    int ele;
    clrscr();
    getdata();
    putdata();
    printf("\nenter ele to search\n");
    scanf("%d", &ele);
    lsearch(ele);
    getch();
}
void getdata()
{
  printf("enter no of elements in array\n");
  scanf("%d", &n);
  printf("enter array elements\n");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);
}
void putdata()
{
  printf("array elements are\n");
  for(i=0; i<n; i++)
  printf("%d\t", a[i]);
}
void lsearch(int el)
{
  int k, flag=0;
  for(k=0; k<n; k++)
  {
    if(a[k]==el)
    {
      flag=1;
      break;
    }
  }
  if(flag)
  printf("element %d is found at %d\n", el, k+1);
  else
} printf("element not found");
```

2) Binary search: Binary search is a searching algorithm that works efficiently with a sorted list like a dictionary is in lexicographic order.

Ex:
```
void bsearch(int el)
{
  int low, high, mid, flag=0;
  low=0;
  high=n-1;
  while(low<=high)
  {
    mid = (low+high)/2;
    if(el<a[mid])
      high = mid-1;
    else if(el>a[mid])
      low = mid+1;
    else if(el==a[mid])
    {
      flag=1;
      break;
    }
  }
  if(flag)
  pf("element found at %d\n", mid+1);
}
```

* ascending order



l=0, h=4
0<=4
m = 0+4/2 = 2
9<a[2]=9
9<9 x
9>9 x
9==9 ✓
flag=1

//wap to implement binary search in an array. (using functions)

```
#include<stdio.h>
#include<conio.h>
void bsearch(int);
void getdata();
void putdata();
int a[10], n, i, ele;
int main()
{
  int p;
  clrscr();
  getdata();
  putdata();
```

```
    printf("\n enter ele to search\n");
    scanf("%d", &ele);
    bsearch(ele);
    getch();
    return 0;
}
void getdata()
{
    printf("enter no of elements in an array\n");
    scanf("%d", &n);
    printf("enter array elements\n");
    for(i=0; i<n; i++)
    scanf("%d", a[i]);
}
void putdata()
{
    printf("array elements are\n");
    for(i=0; i<n; i++)
    printf("%d\t", a[i]);
}
void bsearch(int el)
{
    int low, high, mid, flag=0;
    low=0;
    high=n-1;
    while(low<=high)
    {
        mid = (low+high)/2;
        if(el < a[mid])
            high = mid-1;
        else if(el > a[mid])
            low = mid+1;
        else if(el == a[mid])
        {
            flag=1;
            break;
        }
    }
    if(flag)
    printf("element %d is found at %d position\n", ele, mid+1);
}
```

⇒ Complexity (Time taken for algorithm)

1) Algorithm sum(a, x)

```
{
    sum=0;          ____ 0
    for(i=0; i<n; i++)  ____ n+1
    {
        sum = sum+a[i];  ____ n
    }                    ____ 0
}                        ____ 0
```

$T(n) = 2n+2$

$T(n) = O(n)$

Ignore constants, coefficient for time complexity. Consider high order

2) Alg sort()

```
{
    for(i=0; i<n-1; i++)   ____ n+1
    {
        for(j=0; j<n-1-i; j++)  ____ n(n+1)
        {
            if(a[j] > a[j+1])   ____ n*n
                swap(a[j], a[j+1]);  ____ n*n
        }
    }
}
```

$T(n) = n+1+n^2+n+n^2$

$= 3n^2+2n+1$

$T(n) = O(n^2)$

No. of for loops

1 for loop = O(n)

2 for loops = O(n²)

3 for loops = O(n³)

→ Binary Search

Time complexity

$T(n) = 1 + T(n/2)$

$T(n) = 1+1+T(n/4)$

$T(n) = 1+1+1+T(n/8)$

$T(n) = 1+1+1+1+\cdots+T(n/6)$

$= 1+1+1+1+T(n/2^4)$

$= k + \cdots + T\left(\frac{n}{2^k}\right)$

$= \log n + T(1)$

$T(n) = O(\log n)$

Assume $\frac{n}{2^k} = 1$

$n = 2^k$

Apply 'log' on both sides

$\log n = \log(2^k)$

$\log n = k \log_2 2$

$k = \log n$

→ Binary Search algorithm using Recursion.

```
int bsearch(int l, int h, int k)
{ int mid;
  if(l>h)
  {
    pf("element not found\n");
     return -1;
  }
  mid = (l+h)/2;
  if(a[mid]==k)
     return mid;
  else if(a[mid]<k)
     bsearch(mid+1, h, k);
  else if(a[mid]>k)
     bsearch(l, mid-1, k);
```

//wap for binary search using Recursion.

```
#include<stdio.h>
#include<conio.h>
int a[10], n, ele, i;
int bsearch(int, int, int);
void main()
{ int x;
  clrscr();
  printf("enter no of elements\n");
  scanf("%d", &n);
  printf("\nenter array elements\n");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);
  printf("\n array elements are\n");
  for(i=0; i<n; i++)
  printf("%d\t", a[i]);
  printf("\nenter ele to search\n");
  scanf("%d", &ele);
  x=bsearch(0, n-1, ele);
  if(x==-1)
    printf("element not found");
  else
     printf("element found");
}
int bsearch(int l, int h, int k)
{ int mid;
  if(l>h)
    { return -1;
    }
    mid=(l+h)/2;
    if(a[mid]==k)
        return mid;
    else if(a[mid]<k)
      bsearch(mid+1, h, k);
    else if(a[mid]>k)
      bsearch(l, mid-1, k);
}
```

⇒ Sorting

1) **Bubble sort**: It is the sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

- Start at the first element of the array.
- Compare the current element with the next element.
- If the current element is greater than the next element, swap them.
- Move to the next pair of elements and repeat the comparison and swap if needed.
- After each complete pass through the array, the largest unsorted element is placed at its correct position at the end of the array.
- Repeat the above process of a pass for the remaining unsorted elements until the entire array is sorted.

Ex:

0	1	2	3	4
16	15	8	6	5

pass=0
16 15 8 6 5
15 16 8 6 5
15 8 16 6 5
15 8 6 16 5
15 8 6 5 16

pass=1
15 8 6 5 16
8 15 6 5 16
8 6 15 5 16
8 6 5 15 16
8 6 5 15 16

pass=2
8 6 5 15 16
6 8 5 15 16
6 5 8 15 16
6 5 8 15 16
6 5 8 15 16

pass=3
6 5 8 15 16
5 6 8 15 16
5 6 8 15 16
5 6 8 15 16
5 6 8 15 16

Time Complexity:

$T(n) = O(n^2)$

algorithm:

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

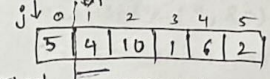
// wap for bubble sort.

```
#include<stdio.h>
#include<conio.h>
int a[10], n, i;
void getdata();
void putdata();
void bubblesort();
void main()
{
  clrscr();
  getdata();
  printf("\n before sorting array elements are\n");
  putdata();
  bubblesort();
  printf("\n after sorting array elements are\n");
  putdata();
  getch();
}
void getdata()
{
  printf("enter no of elements in an array\n");
  scanf("%d", &n);
  printf("\n enter array elements\n");
```

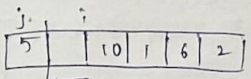
```
    for(i=0; i<n; i++)
    scanf("%d", &a[i]);
}
void putdata()
{
    for(i=0; i<n; i++)
    printf("%d", a[i]);
}
void bubblesort()
{
    int temp, j;
    for(i=0; i<n-1; i++)
    { for(j=0; j<n-1-i; j++)
      { if(a[j]>a[j+1])
        { temp=a[j];
          a[j]=a[j+1];
          a[j+1]=temp;
        }
      }
    }
}
```

2) Insertion sort: It is a sorting algorithm that builds a sorted array by moving each element one at a time to its correct position

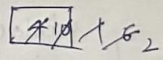
- It splits into two: sorted and unsorted.
- We start with second element of the array as first element in the array is assumed to be sorted.
- compare 2nd element with the first element and check if the 2nd element is smaller then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
- continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
- Repeat until the entire array is sorted.



[∵ consider one element from unsorted & place in temp]



temp



⇒ 5 > 4 ✓

4 5 | 10 1 6 2
sorted unsorted

move 'j' position to unsorted index position

4 5 | — 1 6 2

j++

⇒ 4 5 10 | 1 6 2 5 > 10 ✗ no need to move 1 pos.
sorted unsorted

⇒ 4 5 ? 10 6 2 10 > 1 ✓

4 ? 5 10 6 2 5 > 1 ✓

1 ? 4 5 10 6 2 4 > 1 ✓

1 4 5 10 | 6 2
sorted unsorted

⇒ 1 4 5 10 | — 2 10 > 6 ✓

1 4 5 ? 10 2 5 > 6 ✗

1 4 5 6 10 | 2
sorted unsorted

1 4 5 6 10 — 1072 ✓
1 4 5 6 7 10 672 ✓
1 4 5 7 6 10 572 ✓
1 4 7 5 6 10 472 ✓
1 7 4 5 6 10 172 ✗
1 2 4 5 6 10

1 2 4 5 6 10

Time complexity:

$T(n) = O(n^2)$

algorithm:

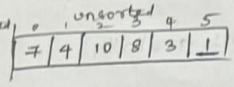
```
void insertionsort()
{
  for(i=1; i<n; i++)
  {
    temp=a[i];
    j=i-1;
    while(j>=0 && a[j]>temp)
    {
      a[j+1]=a[j];
      j--;
    }
    a[j+1]=temp;
  }
}
```

// wap for Insertion sort.

```
#include<stdio.h>
#include<conio.h>
int a[10], n, i, j;
void getdata();
void putdata();
void insertionsort();
void main()
{
  clrscr();
  getdata();
  printf("\n before sorting array elements \n");
  putdata();
  insertionsort();
  printf("\n after sorting array elements\n");
  putdata();
  getch();
}
void getdata()
{
  printf("enter no of elements in an array\n");
  scanf("%d", &n);
  printf("enter array elements are\n");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);
}
void putdata()
{
  for(i=0; i<n; i++)
  printf("%d\t", a[i]);
}
void insertionsort()
{
  int temp;
  for(i=1; i<n; i++)
  {
    temp=a[i];
    j=i-1;
    while(j>=0 && a[j]>temp)
    {
      a[j+1]=a[j];
      j--;
    }
    a[j+1]=temp;
  }
}
```

3) Selection sort: It is a sorting algorithm that arranges a list of elements in order by finding the smallest element at each step and adding it to a sorted list.

- first we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
- Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
- we keep doing this until we get all elements moved to correct position.



pass=0	1	4	10	8	3	7
	sorted		Unsorted			
pass=1	1	3	10	8	4	7
pass=2	1	3	4	8	10	7
pass=3	1	3	4	7	10	8
pass=4	1	3	4	7	8	10
	sorted					

Time complexity: $T(n) = O(n^2)$ best, worst, average case $O(n^2)$

algorithm,

```
void selectionsort()
{
  for(i=0; i<n-1; i++)
  {
    min=i;
    for(j=i+1; j<n; j++)
    {
      if(a[j]<a[min])
      {
        min=j;
      }
    }
    t=a[j];
    a[j]=a[i];
```

n=6 i=0; 0<5 min=0

j=1, 1<6

a[1]<a[0] 4<7 ✓

min=1

a[2]<a[1] 10<4 ✗

a[3]<a[1] 8<4 ✗

a[4]<a[1] 3<4 ✓

min=4

a[5]<a[4] 1<3 ✓

min=5

t=a[j]=a[5]=1

a[5]=a[j]=a[i]=a[0]=7

a[0]=1

---

i=1, 1<5 min=1

j=2, 2<6

a[2]<a[1] 10<4 ✗

a[3]<a[1] 8<4 ✗

a[4]<a[1] 3<4 ✓

min=4

a[5]<a[4] 7<3 ✗

t=a[j]=a[4]=3

a[4]=a[j]=a[i]=a[1]=4

a[1]=3

---

i=2, 2<5 min=2

j=3, 3<6

min=3

a[3]<a[2] 8<10 ✓

min=3

a[4]<a[3] 4<8 ✓

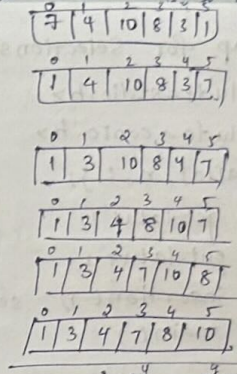
min=4

a[5]<a[4] 7<4 ✗

t=a[j]=a[4]=4

a[4]=a[j]=a[i]=a[2]=10

a[2]=4



i=3, 3<5 min=3

j=4, 4<6

min=3

a[4]<a[3] 10<8 ✗

a[5]<a[3] 7<8 ✓

min=5

t=a[j]=a[5]=7

a[5]=a[j]=a[i]=a[3]=8

a[3]=7

---

i=4 4<5 min=4

j=5 5<6

min=4

a[5]<a[4] 8<10 ✓

min=5

t=a[j]=a[4]=8

a[j]=a[i]=a[8]=10

a[4]=8

// wap for Selectionsort.

```
#include<stdio.h>
#include<conio.h>
int a[10], n, i, j;
void getdata();
void putdata();
void insertion();  selectionsort();
void main()
{
   clrscr();
   getdata();
   pf("\n before sorting array elements\n");
   putdata();
   selectionsort();
   pf("\n after sorting array elements\n");
   putdata();
   getch();
}
void getdata()
{
   pf("enter no of elements in an array\n");
   sf("%d", &n);
   pf("\n enter array elements\n");
   for(i=0; i<n; i++)
   sf("%d", &a[i]);
}
void putdata()
{
   for(i=0; i<n; i++)
   pf("%d\t", a[i]);
}
```

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

```
// wap for Selectionsort.
#include<stdio.h>
#include<conio.h>
int a[10], n, i, j;
void getdata();
void putdata();
void insertion(); selectionsort();
void main()
{
   clrscr();
   getdata();
   pf("\n before sorting array elements\n");
   putdata();
   selectionsort();
   pf("\n after sorting array elements\n");
   putdata();
   getch();
}
void getdata()
{
   pf("enter no of elements in an array\n");
   sf("%d", &n);
   pf("\n enter array elements\n");
   for(i=0; i<n; i++)
   sf("%d", &a[i]);
}
void putdata()
{
   for(i=0; i<n; i++)
   pf("%d\t", a[i]);
}
```

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

Unit - III

Stacks

A stack is known as LIFO (Last in first out) structure because the element which is inserted last will be taken out first.

Stack is a linear data structure and an ordered collection of homogeneous data elements where the insertion and deletion operations done at one end only.

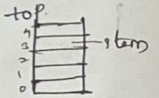
Insertion operation is known as push and deletion operation is known as pop. Element of the stack is known as item and end of the stack is known as top.

- top always indicate top of the stack (or) current position of the stack.

Initially, stack is empty ⇒ top < 0

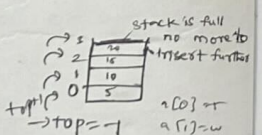
top = -1.

stack is full ⇒ top > n-1



- Using single-dimensional array and a top variable which is initialized to -1. To implement stack, the following operations should be implemented.

1. push 2. pop



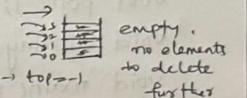
1. push()

```
top+1
a[top] = el;
```

```
void push()
{ int el;
  //a[n] is stackhome
  //top = -1;
  if(top
  pf("enter element to be pushed\n");
  sf("%d", &el);
  if(top > n-1)
    pf("stack is full\n");
  else
  {
    top = top+1;
    a[top] = el;
  }
}
```

2. pop()

```
el = a[top];
top = top-1;
```




```
void pop()
{
  //a[n] is stackhome
  //top = -1;
  if(top < 0)
    pf("stack is empty");
  else
  { el = a[top];
    top = top-1;
    pf("popped element is %d\n", el);
  }
}
```

⇒ displaying the elements of stack

```
void display()
{
  if(top < 0)
    pf("stack is empty\n");
  else
  {
    pf("stack elements are\n");
    for(i = top; i >= 0; i--)
    {
      pf("%d\t", a[i]);
    }
  }
}
```

```
//wap for stack.
#include<stdio.h>
#include<conio.h>
int a[30], n, i;
int top=-1;
void push();
void pop();
void display();
void main()
{
  int ch;
  printf("enter no. of elements in stack\n");
  scanf("%d",&n);
  do
  {
    pf("1:push(), 2:pop(), 3:display(),
                    4:exit()\n");
    pf("enter choice\n");
    sf("%d", &ch);
    switch (ch)
    {
      case 1: push();
              break;
      case 2: pop();
              break;
      case 3: display();
              break;
      case 4: exit();
      default: pf("Invalid choice\n");
    }
  }while(ch!=4);
  getch();
}
void push()
{
  int el;
  pf("enter element to be pushed\n");
  if(top>=n-1)
    pf("stack is full\n");
  else
  {
    top=top+1;
    a[top]=el;
  }
}
void pop()
{
  // a[n] is stackhome
  // top=-1;
  if(top<0)
    pf("stack is empty");
  else
  {
    el=a[top];
    top=top-1;
    pf("popped element is %d\n", el);
  }
}
void display()
{
  if(top<0)
    pf("stack is empty\n");
  else
  {
    pf("stack elements are\n");
    for(i=top; i>=0; i--)
    {
      pf("%d\t", a[i]);
    }
  }
}
```

⇒ Stack applications:

* Infix, Postfix, Prefix Notations

To represent expressions, we use different notations.

- Infix expression

Infix expressions are expressions where the operator is placed between its operands.

ex: a+b

- Prefix expression

Prefix expressions also known as Polish notation, where the operator precedes its operands.

ex: a+b ⟶ +ab

- Postfix expression

Postfix expressions also known as Reverse Polish Notation, where the operator follows (after) its operands.

ex: a+b ⟶ ab+

Ex: (A+B)*C/D ⇒ Infix expression

Infix to prefix	Infix to postfix
+AB*C/D	AB+*C/D
*+ABC/D	AB+C*/D
/*+ABCD	AB+C*D/

⇒ Algorithm for evaluation of postfix notation. (conversion from postfix to infix)

Step 1: Scan postfix expression from left to right.

Step 2: a) if scanned character is operand, read its value and push onto the stack.

b) Otherwise, if it is operator, pop top two elements and perform operation with scanned operator and push the result onto the stack.

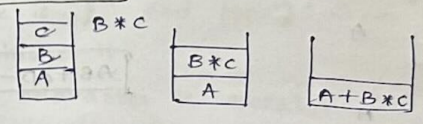
Step 3: Repeat step 2, until no character to be scanned.

Step 4: Pop the stack and print the result.

Ex: A+B*C

A+BC*

postfix ABC*+



⇒ Algorithm for Conversion from infix to postfix expression

Step 1: push '(' onto the stack and add ')' to infix string

Step 2: Scan infix string from left to right

a) if character is operand, add to postfix string.

b) if it is '(' push onto stack.

c) if it is ')', pop all elements from the stack and add to postfix string until top of stack is '(' and pop '(' from the stack and eliminate it

d) if character is operator

i) if precedence of stack top is higher or equivalent to precedence of scanned operator, then pop the elements from the stack and add to postfix string.

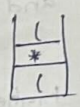
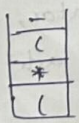
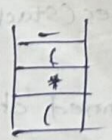
Repeat this process until prec(Stack top) < prec(scanned character)

ii) if prec(Stack top) < prec(scanned character) push the operator onto the stack.

Step 3: Print the postfix string

Ex: $(A+B)*(C-D)$ AB+CD-*

$((A+B)*(C-D))$

i/p	stack	o/p
		-
'('		-
A		A
+		A
B		AB
)		AB+
*		AB+
(		AB+
C		AB+C
-		AB+C
D		AB+CD
)		AB+CD-

AB+CD-*

i/p	stack	o/p
)		AB+CD-*
)		AB+CD-*

AB+CD-*

⇒ Algorithm for conversion of infix to prefix

Step 1: scan infix expression from right to left

Step 2: a) if character is operand, add to o/p string.

b) if character is ")", then push onto the stack.

c) if character is "(", then pop all elements from the stack and to prefix string until top of stack is ")" and pop ")" from the stack and eliminate it.

d) if character is operator

(i) if prec (stack top) >= prec (scanned character) then pop elements from the stack and add to o/p string.

Repeat this process until prec (stack top) < prec (scanned character)

(ii) if prec (stack top) < prec (scanned character) then push the operator onto the stack.

Step-3: Write the o/p string in reverse order to form the prefix string and print the result.

Ex: $(A+B)*(C-D)$ *+AB-CD

i/p	stack	o/p
)		-
D		D
-		D
C		DC
(		DC-

i/p	Stack	o/p
*		DC-
)		DC-
B		DC-B
+		DC-B
A		DC-BA
(		DC-BA+
		DC-BA+*

Reverse it = *+AB-CD

Assignment:
A+B*C-D

⇒ Algorithm for conversion of prefix to infix

Write the prefix expression in reverse order and consider that as i/p string.

Step 1: Scan prefix expression from left to right.

Step 2: a) if character is operand, read its value and push onto the stack.

b) if it is operator, pop top two elements and perform operation with scanned operator and push the result value onto the stack.

Step 3: Repeat step 2, until no character to be scanned.

Step 4: pop the stack and print the result.

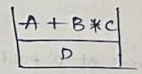
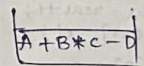
Ex: A+B*C-D

Prefix expression: A+*BC-D

+A*BC-D

-+A*BCD

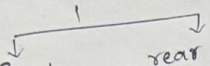
Reverse order :- DCB*A+-

i/p	Stack
D	
C	
B	
*	
A	
+	
-	

Result: A+B*C-D

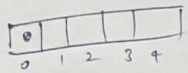
Queues

- Queue is non-primitive data type.
- FIFO (First In First Out)
- Two variables



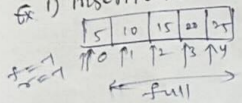
- used for deletion
- used for insertion

Initially, front = -1 , rear = -1



- Queue is represented in 2 ways.
  1) Single dimensional array
  2) linked list
- Queue has 2 operations.
  1) insertion
  2) deletion

fx 1) insertion :- first front & rear at '0' position.

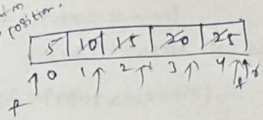



```
rear = rear+1;
a[rear] = el;
a[0] = 5;
```

```
⇒ void insert()                  // f=-1, r=-1;
  { int el;                      // a[max] is an array;
    if(rear==max-1)              // 'max' is the size of an array;
      printf("Queue is full\n");
    if(rear==-1)
    {
      front=0;
      rear=0;
    }
    else
      rear=rear+1;
      a[rear]=el;
  }
```

2) deletion :-

after inserting 5 front at '0' position.




```
el = a[front];
el = a[0]
el = 5;
```

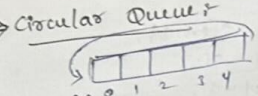
r=-1 empty.

```
⇒ void delete()
  {
    if(front==-1)
      printf("Queue is empty\n");
    el=a[front];
    if(front==rear)
    {
      front=-1;
      rear=-1;
    }
    else
      front=front+1;
      return el;   printf("deleted element is %d\n", el);
  }
```

⇒ display algorithm :-

```
  void display()
  {
    if(front==-1)
      printf("Queue is empty");
    if(front<=rear)
    {
      for(i=front; i<=rear; i++)
      {
        printf("%d\t", a[i]);
      }
    }
  }
```

⇒ Circular Queue :



f=-1
r=-1

1) insertion algorithm :

```
void insert( )
{ int el;
  if (front==(rear+1)%max)
     printf("Queue is full\n");
  if (rear==-1)
  {
    front=0;
    rear=0;
  }
  else
    rear=(rear+1)%max;
    a[rear]=el;
}
```

2) deletion algorithm :

```
void delete( )
{ if(front==-1)
    printf("Queue is empty\n");
  el=a[front];
  if(front==rear)
  {
    front=-1;
    rear=-1;
  }
  else
    front=(front+1)%max;
    printf("deleted element is %d\n", el);
}
```

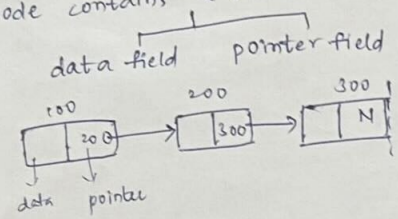
3) display .

```
void display
{ if(front==-1)
    printf("Queue is empty");
  if(front<=rear)
  {
    for(i=front; i<=rear; i++)
    { printf("%d\t", a[i]);
    }
  }
}
```

II unit

Linked Lists

- It is a self-referential structure that consists of collection of nodes.
- Node contains 2 fields.



2M ⇒ Node which is a self-referential structure that contains 2 fields.

format :-

⇒
```
struct node
{
    int data;
    struct node *next;
};
```

⇒ linked list types.

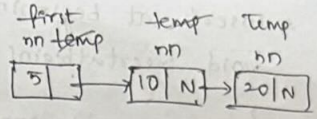
1) Single linked list
2) Double linked list
3) Circular linked list

1) Single linked list operations :-

- create node

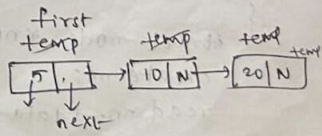
- insert at beginning
- insert at end
- insert at specified position
- delete from beginning
- delete from end
- delete at specified position
- traverse()

⇒ Creation of new node :-




```
void createnode()
{
    struct node *nn, *temp;
    int ch=1;
    while(ch==1)
    {
        create nn;
        read nn→data;
        nn→next=NULL;
        if(first==NULL)
        {
            first=nn;
            temp=nn;
        }
        else
        {   temp→next=nn;
            temp=nn;
        }
        read choice;
    }
}
```

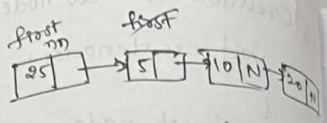
⇒ traverse()




```
void traverse()
{
    struct node *temp;
    temp=first;
    while(temp!=NULL)
    {
        printf("%d\t", temp→data);
        temp=temp→next;
    }
}
```

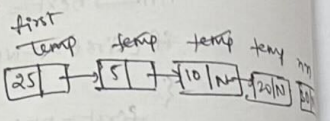
5 10 20.

⇒ insert at beginning :-



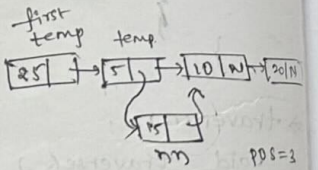

```
void insertatbegin()
{
  struct node *nn;
  create nn;
  read nn->data;
  nn->next = first;
  first = nn;
}
```

⇒ insert at end :-



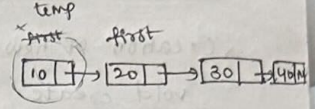

```
void insertatend()
{
  struct node *nn, *temp;
  temp = first;
  while (temp->next != NULL)
      temp = temp->next;
  temp->next = nn;
}
```

⇒ insert at position :-



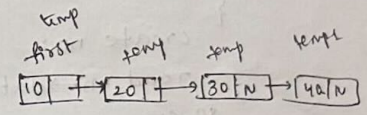

```
void insertatpos()
{
  struct node *nn, *temp;
  create nn;
  read nn->data;
  temp = first;
  for(i=1; i<pos-1; i++)
      temp = temp->next;
  nn->next = temp->next;
  temp->next = nn;
}
```

⇒ delete from beginning




```
void deletefrombeg()
{
  struct node *temp;
  temp = first;
  first = first->next;
  delete temp;
}
```

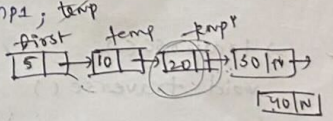
⇒ delete at end



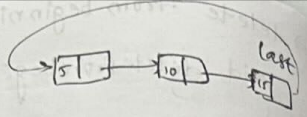

```
void deleteatend()
{
  struct node *temp, *temp1;
  temp = first;
  while (temp->next->next != NULL)
  temp = temp->next;
  temp1 = temp->next;
  temp->next = NULL;
  delete temp1;
}
```

⇒ delete at position

```
void deleteatpos()
{
  struct node *temp, *temp1;
  temp = first;
  for(i=1; i<pos-1; i++)
      temp = temp->next;
  temp1 = temp->next;
  temp->next = temp1->next;
  delete temp1;
}
```



⇒ circular linked list :

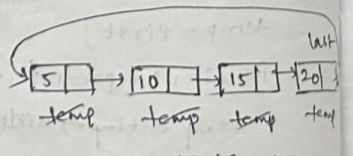


⇒ creation of new node :-

```
void create
{
  struct node *nn;
  int ch=1;
  while(ch==1)
  {
    create nn;
    read nn→data;
    if(last==NULL)
    {
      last=nn;
      last→next=nn;
    }
    else
    {
      nn→next=last→next;
      last→next=nn;
      last=last→next;
    }
    read ch;
  }
}
```

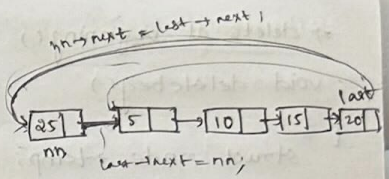
⇒ traverse()

```
void traverse()
{
  struct node *temp;
  temp=last→next;
  while(temp!=last)
  {
    printf("%d\t", temp→data);
    temp=temp→next;
  }
```



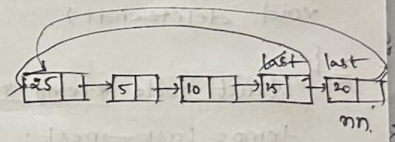
⇒ insert at beginning()

```
void insertatbeg()
{
  struct node *nn;
  create nn;
  read nn→data;
  nn→next=last→next;
  last→next=nn;
}
```



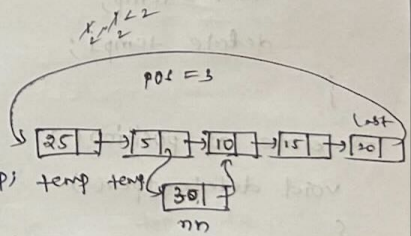
⇒ insert at end()

```
void insertatend()
{
  struct node *nn;
  create nn;
  read nn→data;
  nn→next=last→next;
  last→next=nn;
  last=last→next;
}
```

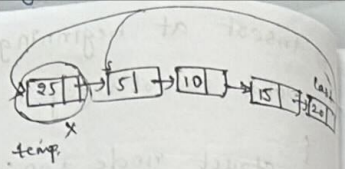


⇒ insert at position()

```
void insertatpos()
{
  struct node *nn, *temp;
  temp=last→next;
  create nn;
  read nn→data;
  for(i=1; i<=pos-1; i++)
    temp=temp→next;
  nn→next=temp→next;
  temp→next=nn;
}
```

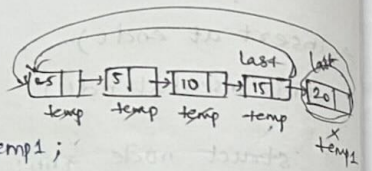


⇒ delete at beginning()



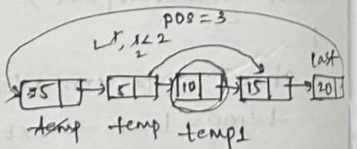

```
void deletebeg()
{
  struct node *temp;
  temp=last->next;
  last->next=temp->next;
  delete temp;
}
```

⇒ delete at end()




```
void deleteend()
{
  struct node *temp, *temp1;
  temp=last->next;
  while(temp->next!=last)
  temp=temp->next;
  temp1=temp->next; (or) last
  temp->next=last->next;
    last=temp;
    delete temp1;
}
```

⇒ delete at position()

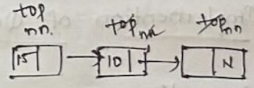



```
void deleteatpos()
{
  struct node *temp, *temp1;
  temp=last->next;
  for(i=1; i<pos-1; i++)
  temp=temp->next;
  temp1=temp->next;
  temp->next=temp1->next;
  delete temp1;
}
```

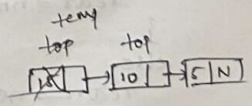
⇒ Implementing Stack using Single Linked list

⇒ push operation

```
void push()
{
  struct node *nn;
  int el, ch=1;
  while(ch==1)
  {
    pf("enter data to be pushed\n");
    sf("%d", &el);
    create nn;
    nn->data=el;
    nn->next=NULL;
    if(top==NULL)
    {
      top=nn;
    }
    else
      nn->next=top;
      top=nn;
  }
  read choice;
}
```

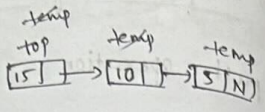


⇒ pop operation




```
void pop()
{
  struct node *temp;
  temp=top;
  el=top->data; (or) temp->data
  top=top->next;
  delete temp;
  pf("popped element is %d\n", el);
}
```

⇒ display

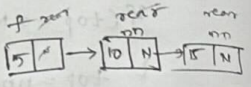



```
void traverse()
{
  struct node *temp;
   temp=top;
   while(temp!=NULL)
   {
    printf("%d\t", temp->data);
    temp = temp->next;
   }
}
```

15 10 5

⇒ Implemention of Queue using Single linked list

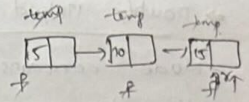
⇒ insertion :-




```
void insert()
{
   int el;
   struct node *nn;
   create nn;
   nn->data = el;
   nn->next = NULL;
   if(rear==NULL)
   {
     front = nn;
     rear = nn;
   }
   else
     rear->next = nn;
      rear = nn;
}
```

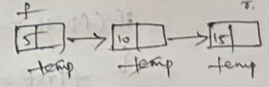
⇒ deletion :-

```
void delete()
{
   int el;
   struct node *temp;
   if(front==NULL)
     printf("Queue is empty\n");
el = front->data;
temp = front;
if(front==rear)
{
    front = NULL;
    rear = NULL;
}
else
    front = front->next;
    delete temp;
    pf("deleted element is %d\n", el);
}
```



⇒ traverse :-

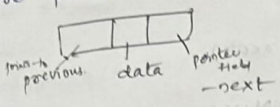
```
void traverse()
{
   struct node *temp;
   temp = front;
   while(temp!=NULL)
   {
     printf("%d\t", temp->data);
     temp = temp->next;
   }
}
```



5 10 15

⇒ Double linked list :-

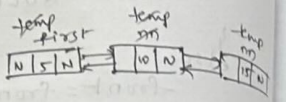
- Node contains 3 fields. One is data field & other two are pointer fields.




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

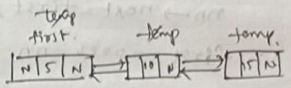
Operations :-

1) creation of node :-




```
void create()
{
  int ch=1;
  struct node *nn, *temp;
  while(ch==1)
  {
    create nn;
    read nn->data;
    nn->next->NULL;
    if(first==NULL)
    {
      first=nn;
      nn->prev=NULL;
      temp=nn;
    }
    else
    {
      temp->next=nn;
      nn->prev=temp;
      temp=nn;    temp=temp->next;
    }
    read choice;
  }
}
```

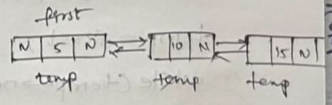
2) display forward :-



5 10 15

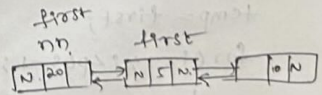
```
void displayforward()
{
  struct node *temp;
  temp=first;
  while(temp!=NULL)
  {
    printf("%d\t", temp->data);
    temp=temp->next;
  }
}
```

3) display backward :-




```
void displaybackward()
{
  struct node *temp;
  temp=first;
  while(temp->next!=NULL)
    temp=temp->next;
  while(temp!=NULL)
  {
    printf("%d\t", temp->data);
    temp=temp->prev;
  }
}
```

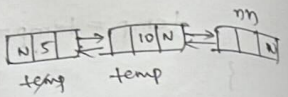
4) insert at beginning :-




```
void insertbeg()
{
  struct node *nn;
  create nn;
  read nn->data;
```

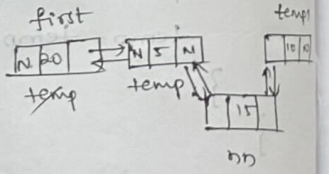
```
nn->prev = NULL;
nn->next = first;
first->prev = nn;
first = nn;
}
```

5) insert at end :-



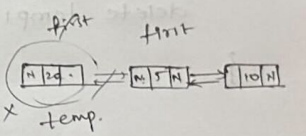

```
void insertend()
{
  struct node *nn;
  create nn;
  read nn->data;
  nn->next = NULL;
  temp = first;
  while (temp->next != NULL)
    temp = temp->next;
  temp->next = nn;
  nn->prev = temp;
}
```

6) insert at position :-



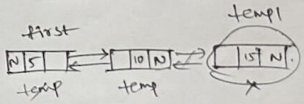

```
void insertpos()
{
  struct node *nn;
  create nn;
  read nn->data;
  temp = first;
  for(i=1; i<pos-1; i++)
    temp = temp->next;
  temp1 = temp->next;
  temp->next = nn;
  nn->prev = temp;
  nn->next = temp1;
  temp1->prev = nn;
}
```

7) delete at beginning :-




```
void deletebeg()
{
  struct node *temp;
  temp = first;
  first = first->next;
  first->prev = NULL;
  delete temp;
}
```

8) delete end :-




```
void deleteend()
{
  struct node *temp, *temp1;
  temp = first;
  while(temp->next->next != NULL)
    temp = temp->next;
  temp1 = temp->next;
  temp->next = NULL;
  delete temp1;
}
```

9) delete position :-

```
void deletepos()
{
  struct node *temp, *temp1;
  temp = first;
  for(i=1; i<pos-1; i++)
  {
```

```
        temp = temp -> next;
    }
    temp1 = temp -> next;
    temp -> next = temp1 -> next;
    temp -> next -> prev = temp;
    delete temp1;
}
```