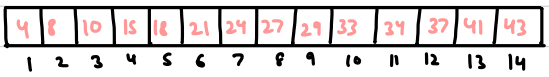
BINARY SEARCH

Size = 15
Length = 15

A



Sorted Array

ITERATIVE VERSION

```
Algorithm BinSearch (l,h, key)
{
    while ( l<= h)
    {
        mid = [(l+h) /2] ;

        if (key == A[mid])
            return mid;

        else if (key < A[mid])
            h = mid -1;
        else
            l = mid +1;
    }
    return -1;      # key not found
}
```

Binary search is faster than linear search

RECURSIVE VERSION

Tail Recursion

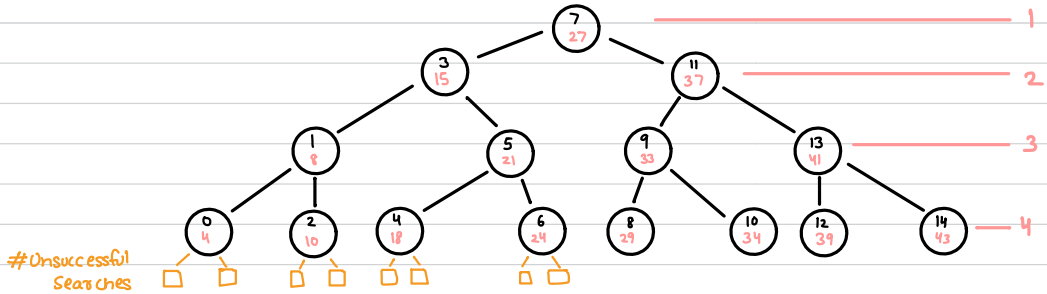
```
Algorithm RBinSearch (l,h, key)
{
    if ( l<= h)
    {
        mid = [(l+h) /2] ;

        if (key == A[mid])
            return mid;

        else if (key < A[mid])
            return   RBinsearch (l,mid -1, key);
        else
            return   RBinsearch (mid +1, h, key);
    }
    return -1;      # key not found
}
```

ANALYSIS OF BINARY SEARCH

Size = 15
Length = 15

A

1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	8	10	15	18	21	24	27	29	33	34	37	41	43



Best min – O(1)
Worst max – O (logn)
Unsuccesful max – O (logn)

Why logn ?

logn
Let size of array be 16

16 / 2 / 2 / 2 / 2

$2^4 = 16$
$4 = \log_2 16$

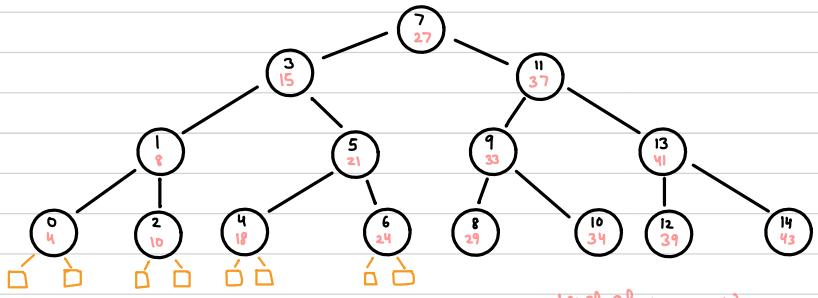
$\log_2 n$

Inverse of power is log.

16 is divided by 2 multiple times

AVERAGE CASE ANALYSIS OF BINARY SEARCH

$$= \frac{\text{Total time taken in all possible cases}}{\text{Number of cases}}$$



$1 + 1 \times 2 + 2 \times 4 + 3 \times 8$

No of comparisons ↵ ↳ No of elements

$1 + 1 \times 2^1 + 2 \times 2^2 + 3 \times 2^3$

logn → Level of tree (Here 4)

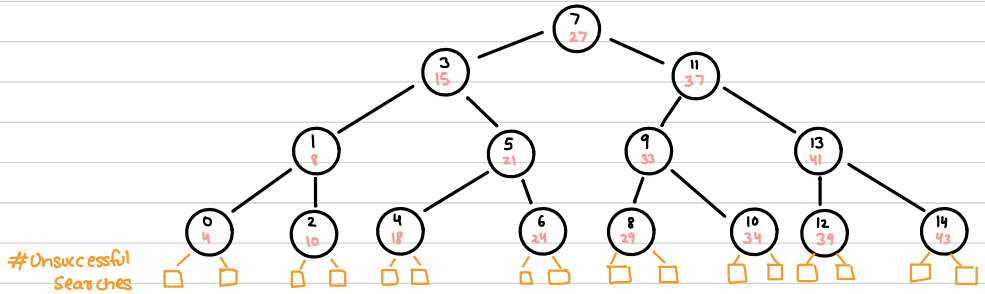
$$\sum_{i=1}^{\log n} \frac{i \times 2^i}{n} = \frac{\log n \times 2^{\log n}}{n}$$

i is replaced with logn

Same as formula *

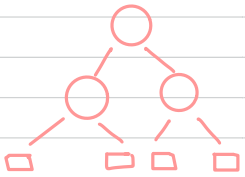
$$= \frac{\log n \times n^{\log 2}}{n}$$

$$= \log n$$



I = sum of all internal nodes
E = sum of all external nodes

$E = I + 2n$



n = 3
I = 2
E = 2x4 = 8

$E = I + 2n$

i = num of internal nodes
e = num of external nodes

n = num of nodes

$e = i + 1$

Average successful time for n elements

$$A_S(n) = 1 + \frac{I}{n} \Rightarrow 1 + \frac{E-2n}{n} \Rightarrow 1 + \frac{n\log n}{n} - 2$$

$$\Rightarrow 1 + \frac{E}{n} - 2 \Rightarrow \log n$$

Average unsuccessful time

$$A_U(n) = \frac{E}{n+1} = \frac{n\log n}{n+1} = \log n$$

$E = n\log n$
$E = I + 2n$
$I = E - 2n$

6. **Get (index) :**

```
if (index >= 0 && index < length)        O(1)
    return A[index];
```

7. **Set (index, x)**

```
if (index >= 0 && index < length)        O(1)
    A[index] = x;
```

8. Max()

```
max = A[0];                          ——— 1
for (i = 1; i < length; i++)         ——— n
{
        if (A[i] > max)              ——— n-1
            max = A[i];
}
return max;                          ——— 1
                                     2n+1   O(n)
```

9. Min()

```
min = A[0];
for (i = 1; i < length; i++)
{
        if (A[i] < min )
            min = A[i];
}
return min;
```

10. Sum()

```
Total = 0;                           ——— 1
for (i = 0; i < length; i++)         ——— n+1
        Total += A[i]                ——— n
return total                         ——— 1
                                     2n+3
                                     O(n)
```

$$Sum(A,n) = \begin{cases} 0 & n < 0 \\ Sum(A,(n-1)) + A[n] & n >= 0 \end{cases}$$

11. Avg()

```
Total = 0;
for (i = 0; i < length; i++)
        Total += A[i]
return total/n;
```

RECURSIVE CALL

```
int Sum(A,n)
{
    if (n < 0)
        return 0;
    else
        return Sum(A,n-1) + A[n];
}
```

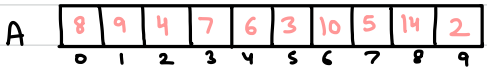
Sum(A, length-1) — call

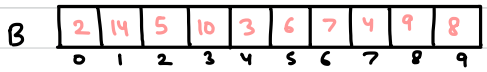
REVERSE AND SHIFT AN ARRAY

1. Reverse
2. Left Shift
3. Left Rotate
4. Right Shift
5. Right Rotate

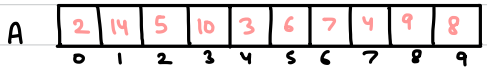
1. Reversing an array

Reversing using auxillary array B.





A → B n



B → A n

2n

O(n)

```
for (i = length-1; j=0; i>=0 ; i-- , j++)
        B[j] = A[i];
```

] Reverse copying array

```
for (i=0 ; i < length ; i++)
        A[i] = B[i];
```

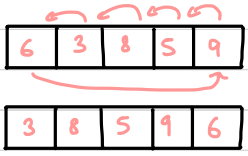
ANOTHER METHOD OF REVERSING ARRAY

```
for ( i=0 ; j = length-1 ; i<j ; i++ ; j--)
{
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
}
```

O(n)

2. Left Shift

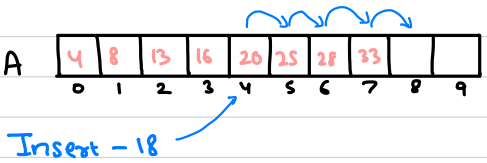
3. Left Rotate



CHECK IF ARRAY IS SORTED

1. Inserting in an sorted array
2. Checking if array is sorted
3. Arranging -ve elements on left side and +ve on right side.

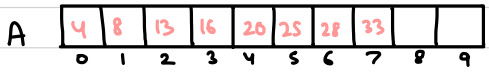
1. Inserting in an sorted array




```
x = 18 ;
i = length - 1 ;
while ( A[i] > x )
{
      A[i+] = A[i];
      i-- ;
}
A[i+1] = x ;
```

Starting from last element, loop will run until the element is greater than the element to be inserted

2. Checking if array is sorted




```
Algorithm isSorted ( A, n)
{
      for (i=0 ; i<n-1 ; i++)
      {
            if ( A[i] > A[i+1])
                  return false;
      }
      return true;
}
```

We will check false condition first by checking if any element is greater than its next element.

O(n) - Max Worst
O (1) - Min Best

3. Arranging -ve elements on left side and +ve on right side.

```
i=0;
j= length-1;

while (i<j)
{
        while (A[i]<0)
            i++;
        while (A[j]≥0)
            j--

        if (i<j)
            swap(A[i], A[j])
}
```

A

-6	3	-8	10	5	-7	-9	12	-4	2
0	1	2	3	4	5	6	7	8	9

i → 0, j → 9

A

-6	-4	-8	-9	-7	5	10	12	3	2
0	1	2	3	4	5	6	7	8	9

j → 4, i → 5

$O(n)$ n+2 comparisons are made

MERGING ARRAYS

1. Append / Concat
2. Compare
3. Copy

```
i=0;
j=0;
k=0;
while (i<m && j<n) ⟶ # When either of i or j have reached end of array.
{
        if (A[i]<B[j])
        {
            C[k]=A[i];
            k++;
            i++;
        }
```

Can also be written as `C[k++]=A[i++];`

m

A

3	8	16	20	25
0	1	2	3	4

i

n

B

4	10	12	22	23
0	1	2	3	4

j

C

3	4	8	10	12	16	20	22	23	25
0	1	2	3	4	5	6	7	8	9

k

```
        else
        {
            C[k]=B[j];
            k++;
            j++;
        }
```

`C[k++]=B[j++]`

$\Theta = (m+n)$ ↳ Time is known

```
        }
        for(; i<m; i++) ⟶ # When some elements in either of array are remaining to be copied.
            C[k++]=A[i];
        for(; j<n; j++)
            C[k++]=B[j];
}
```

SET OPERATIONS

1. UNION

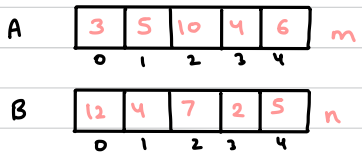
- Copy all elements of A in C
- Then copy elements of B which are not there in C.

$m + m*n$
$n + n*n$
$n + n^2$
$O(n^2)$

- Use merge procedure
- Copy the smaller element to C
- If same element, copy once
- Use i, j, k method

$\Theta(m+n)$
$\Theta(n+n)$
$\Theta(n)$

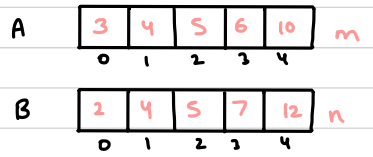
2. INTERSECTION



Copy the common elements of A and B to C

- One by one, take each element of A
  If it is present in B, copy it to C
  otherwise move on to next element

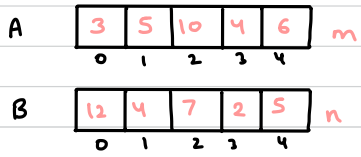
$n*m$
$n*n$
$O(n^2)$

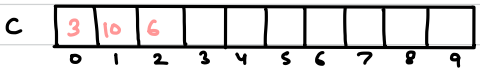


- Use merge method
- If element is smaller, don't copy.
- If element is same, copy
- Not merging but similar to it.

$\Theta(m+n)$
$\Theta(n)$

3. DIFFERENCE





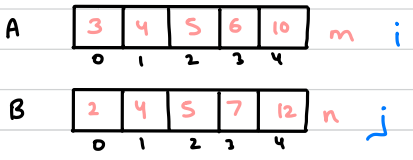
A-B

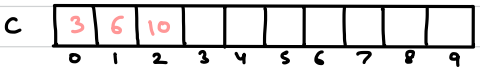
- We want elements of A which are not there in B.
- One by one, take each element of A If it is not present in B, copy it to C, otherwise move to next element

$m * n$

m^2

$O(n^2)$





- Use merge procedure.
- Compare A[i] and B[j], if A[i] is small, copy it to C otherwise increment j.
- If same, don't copy.

$\theta(m+n)$

$\theta(n)$

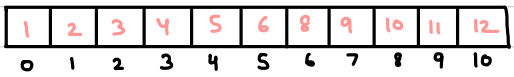
FIND MISSING ELEMENT

1. Single missing element in an sorted array.
2. Multiple missing element in an sorted array.
3. Missing element in unsorted array.

1. Single missing element in an sorted array.

METHOD 1

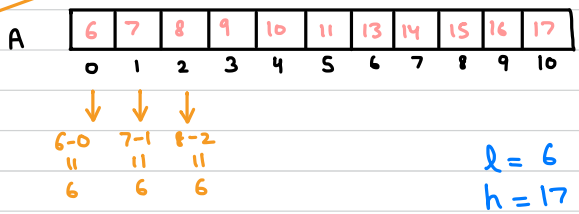
ITERATIVE METHOD




```
for (i = 0; i < 11; i++)
    sum += A[i];
s = n*(n+1)/2;      # Formula for sum of n natural numbers
s - sum
```

78 − 71 = 7 → Missing num

METHOD 2



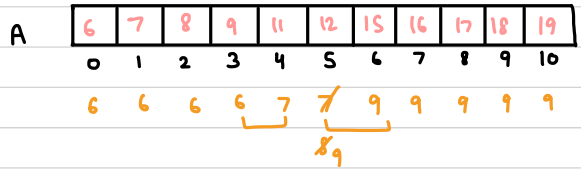
l = 6
h = 17
n = 11
To be known

```
diff = l-0;
for(i=0;i<n;i++)
{
    if(A[i]-i!=diff)
    {
        printf("missing element: %d", i+diff);
        break;
    }
}
```

O(n)

2. Multiple missing element in an sorted array.

METHOD 1




```
diff = 6-0;
for(i=0;i<n;i++)
{
    if(A[i]-i!=diff)
    {
        while(diff<A[i]-i)
        {
            printf("%d", i+diff);
            diff++;
        }
    }
}
```

negligible (few elements)

O(n)

METHOD 2

A

6	7	8	9	11	12	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9	10

0	0	0	0	0	1	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12

Hash Table / Bit Set

A table is created of size equal to the largest element of array A. The array is initialized with 0 One by one, each element present in array A, the index of hash array is initialized by 1 corresponding to it.

```
for (i=0; i<n; i++)                    ---------- n
    H[A[i]]++;

for (i= l; i<= h; i++)                 ---------- n
    if (H[i]==0)                       ----------
        printf("%d", i);               2n
                                       O(n)
```

FINDING DUPLICATE IN A SORTED ARRAY

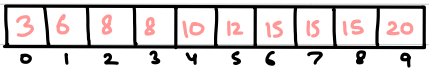
3	6	8	8	10	12	15	15	15	20
0	1	2	3	4	5	6	7	8	9

```
lastDuplicate = 0;

for (i=0; i<n; i++)
{
    if (A[i] = A[i+1] && A[i]!= lastDuplicate)
    {
        printf("%d\n", A[i]);
        lastDuplicate = A[i];
    }
}
```

So that if there are more than 1 duplicate, it will print only one time.

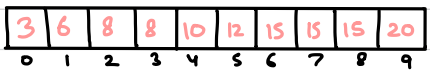
COUNTING NO OF TIMES OF DUPLICATE ELEMENT

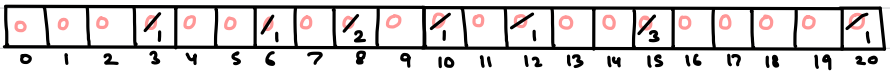
```
for (i=0; i<n-1; i++)
{
    if (A[i] == A[i+1])
    {
        j = i+1;

        while (A[j] == A[i])
            j++;

        printf("%d is appearing %d times", A[i], j-1);
        i = j-1;
    }
}
```



O(n)

FINDING DUPLICATES IN SORTED ARRAY USING HASHING





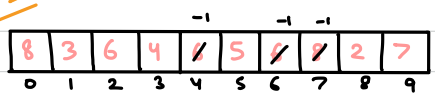

```
for (i=0; i<n; i++)
    H[A[i]]++;

for (i=0; i<=max; i++)
    if (H[i]>1)
        printf("%d %d", i, H[i]);
```

n+n = 2n

O(n)

FINDING DUPLICATES IN AN UNSORTED ARRAY

METHOD 1

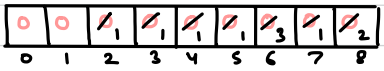



```
for (i=0; i<n-1; i++)
{
        count = 1;

        if( A[i] != -1)
        {
              for (j = i+1; i<n; j++)
              {
                    if (A[i] == A[j])
                    {
                          count ++;
                          A[j] = -1;
                    }
              }

              if ( count > 1 )
                    printf("%d %d ", A[i], count );
        }
}
```

$O(n^2)$

METHOD 2

USING HASH TABLE

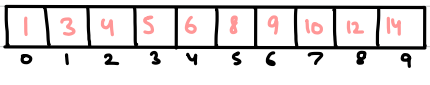


$n + n$

$O(n)$

FIND PAIR WITH SUM K (a+b=k)

SORTED



Let a+b = 10 (to be searched)

i+j

If i+j > 10
decrement j

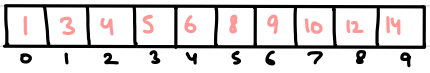
i+j < 10
increment i

i+j = 10
increment i
decrement j

```
i = 0, j = n-1
while (i<j)
{
    if (A[i] + A[j] == k)
    {
        printf("%d+ %d = %d", A[i], A[j], k);
        i++;
        j--;
    }
    else if (A[i] + A[j] < k)
        i++;
    else                                  O(n)
        j--;
}
```

FIND PAIR WITH SUM K (a+b=k)

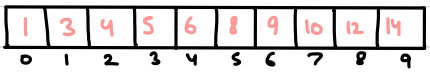
UNSORTED




```
for (i=0; i<n-1; i++)
{                                         O(n²)
    for (j = i+1; j<n; j++)
    {
        if (A[i] + A[j] == k)
            printf("%d+ %d = %d", A[i], A[j], k);
    }
}
```

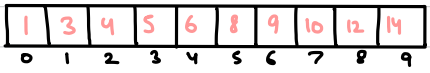
FIND PAIR WITH SUM K (a+b=k)

USING HASHING




```
for (i=0; i<n; i++)
{
        if (H[k-A[i]] != 0)
                printF ("%d + %d = %d", A[i], k-A[i], k);
        H[A[i]]++;
}
```

FIND MAX AND MIN IN SINGLE SCAN




```
min = A[0];
max = A[0];

for (i=1; i<n; i++)
{
        if (A[i]< min)
                min = A[i];
        else if (A[i] > max)
                max = A[i];
}
```

n=10

O(n)

Best = ← 10, 9, 8, 7, 2, 1

comp = n-1

worst = → 1, 2, 3, 5, 8, 9

comp = 2(n-1)

STRINGS

1. Character Sets / ASCII codes
2. Character Array
3. String
4. Creating a string

1. Character Sets / ASCII codes

→ For English language

American Standard Code for Information Interchange

A - 65
B - 66
⋮
Z - 90

a - 97
b - 98
⋮
z - 122

0 - 48
1 - 49
⋮
9 - 57

UNICODES
2 byte
16 bits
4x4 bits
Represented in form of hexadecimal

↵ enter - 10
Space - 13
esc - 27

0 - 127 Total 128 1 byte
$2^7 = 128$
7 bits

2. Character Array

```
char temp;
temp = 'A';
```

65 (Stored as 65)

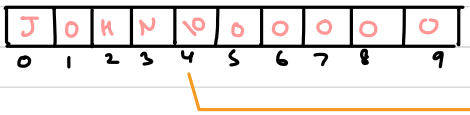
A

```
printf("%c", temp);
```

Displays 'A' and not 65

```
char X[5];
char X[5] = {'A','B','C','D','E'};

char X[5] = {65,66,67,68,69};
```

```
char name[10] = { 'J', 'O', 'H', 'N', '\0' };
```



String delimeter
End of string char
NULL char

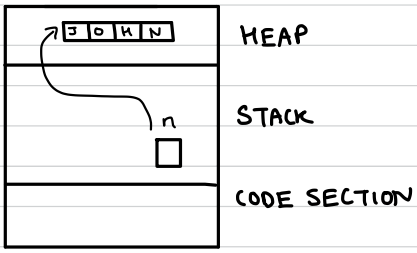
CREATING A STRING

(1) `char name[10] = { 'J', 'O', 'H', 'N', '\0' };`

(2) `char name[] = { 'J', 'O', 'H', 'N', '\0' };` Size of array = 5

(3) `char name[] = "John";` compiler takes null character on its own

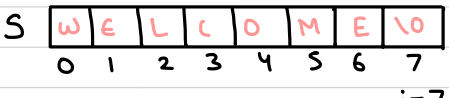
(4) `char *n = "John";`



DISPLAYING A STRING

```
char name[10] = "David";
printf("%s", name);
scanf("%s", name);   → cannot read after space
gets(name)           → can read until you hit enter
```

FINDING LENGTH OF STRING

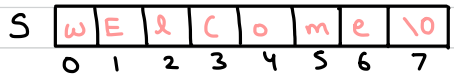



```
int main()
{
    char *s = "welcome";   // No size is required
    int i;
    for (i=0; s[i]!='\0'; i++)
    {
        // This will remain empty
    }
    printf("Length is %d", i);
}
```

CHANGING CASE OF A STRING

```
int main()
{
    char A[] = " WELCOME";
    int i;
    for (i=0; A[i] != '\0'; i++)
        A[i] = A[i] + 32;
    printf (" %s", A);
}
```

TOGGLE CASES




```
int main()
{
    char A[] = " WElCome";
    int i;

    for (i=0; A[i] != '\0'; i++)
    {
        if ( A[i] >= 65 && A[i] <=90)
            A[i] += 32;
        else if ( A[i]>= 'a' && A[i] <= 122)
            A[i] -= 32;
    }
}
```

COUNTING NO OF VOWELS AND CONSONANTS

```
for (i=0; A[i] != '\0'; i++)
{
    if ( A[i] == 'a' || A[i] == 'e'..... )
        vcount ++;

    else if ((A[i] >= 65 && A[i] <=90) || (A[i]>=97 && A[i]<=122))
        ccount ++;
}
```

COUNTING NO OF WORDS

```
int main()
{
    char A[] = " How are    you ";
    int i, word = 1;

    for (i=0; A[i] != '\0'; i++)
        if (A[i] == ' ' && A[i-1] != ' ')
            word ++;
}
```

white space

when there are more than one space

VALIDATING A STRING

```
int valid ( char * name)
{
    int i;
    for (i=0; name[i] != '\0'; i++)
    {
        if ( ! ( name[i] >= 65 && name[i] <= 90 ) &&
             ! ( name[i] >= 97 && name[i] <= 122) &&
             ! ( name[i] >= 48 && name[i] <= 57))
                return 0;
        else
                return 1;
    }
}

int main()
{
    char * name = " Anil 321";
    if ( validate (name))
            printf (" Valid String");
    else
            printf (" Invalid String");
}
```

This type of string is not modifiable

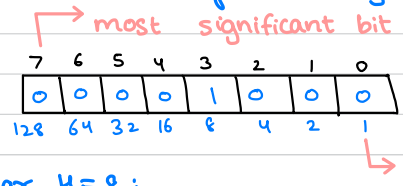
FINDING DUPLICATES IN A STRING

```
int main()
{
        char A[] = "finding";
        int H[26], i;

        for(i=0; A[i]!='\0'; i++)
                H[A[i]-97]+=1;

        for(i=0; i<26; i++)
                if(H[i]>1)
                        printf("%c", i+97);
}
```

FINDING DUPLICATES IN A STRING USING BITWISE OPERATIONS

1. Left Shift <<
2. Bits ORing (Merging)
3. Bits ANDing (Masking)

Manipulating bits of a byte



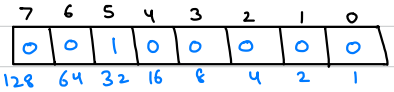
(Represents how 8 is stored)

1 BYTE = 8 BITS

```
char H=8;
```

(1) LEFT SHIFT

H = H << 2



(2) BITS ORing

a = 10 ⟶ 1010
b = 6 ⟶ 0110
1110 ⟶ 14

(3) BITS ANDing

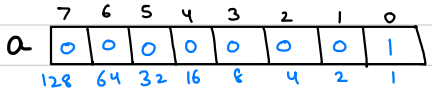
a = 10 ⟶ 1010
b = 6 ⟶ 0110
0010 ⟶ 2

MASKING

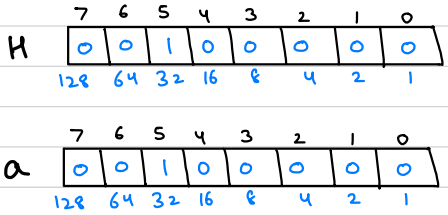
Q We want to know whether 4th bit in H is on or off.

Sol

a = 1
a = a<<4
a & H (perform ANDing)



MERGING



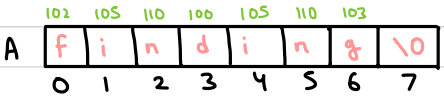
Q we want to set 3rd bit in H as on.

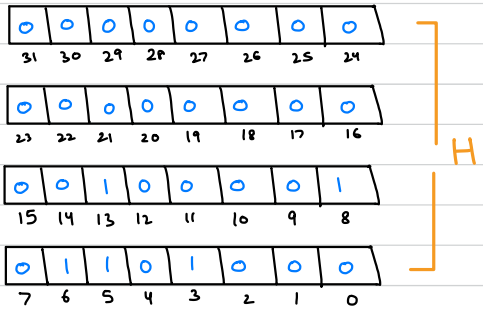
Sol

a = 1
a = a<<2
H = a | H

FINDING DUPLICATES

ASCII Codes





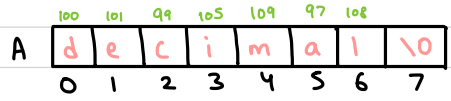
32 bits created for 26 alphabets

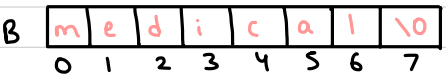
X also consists of 32 bits

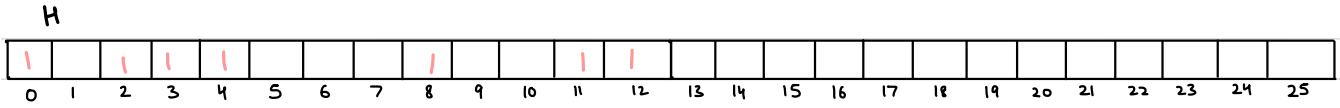
```
int main()
{
    char A[] = "finding";
    long int H=0, x=0;
    for(i=0; A[i]!='\0';i++)
    {
        x=1;
        x = x << A[i]-97;
        if (x & H >0)
            printf(" %c is duplicate", A[i]);
        else
            H = x | H;
    }
}
```

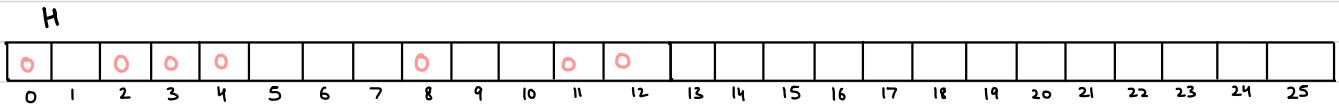
CHECK FOR ANAGRAM

When two strings are made up of same alphabets










```
int main
{
    char A[] = "decimal";
    char B[] = "medical";
    int i, H[26] = {0};

    for (i=0; A[i]!='\0'; i++)          // Making it 1
        H[A[i]-97] += 1;

    for (i=0; B[i]!='\0'; i++)          // Making it 0
        if (H[A[i]-97] < 0)
        {
            printf("Not Anagram");
            break;
        }

    if (B[i]=='\0')
        printf("Anagram");
```

First check if the strings are of same length.

PERMUTATION OF A STRING

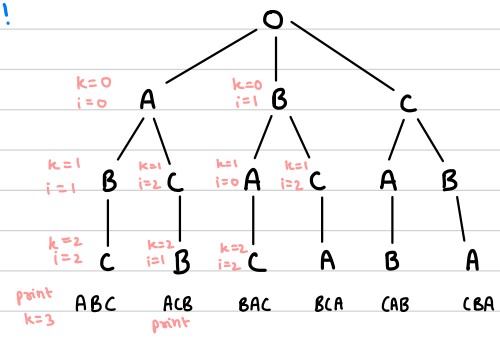
A	B	C	\0
0	1	2	3

1st Method

3!

n!

State Space Tree



Brute Force: Finding all possible permutations

Back Tracking

- Implemented using recursion
- To achieve brute force.

```
void perm(char s[], int k)
{
    Static int A[10] = {0};
    Static char Res[10];
    int i;

    if (s[k] == '\0')
    {
        Res[k] = '\0'
        printf("%s", Res);
    }
    else
    {
        for (i=0; s[i] != '\0'; i++)
        {
            if (A[i] == 0)
            {
                Res[k] = s[i];
                A[i] = 1;
                perm(s, k+1);
                A[i] = 0;
            }
        }
    }
}
```

S	A	B	C	\0	
	0	1	2	3	
A	0	0	0	0	
	0	1	2	3	
Res	A	B	C	\0	
	0	1	2	3	

```
main()
{
    char s[] = "ABC";
    perm(S, 0);
}
```

2nd Method

```
void perm(char s[], int l, int h)
{
    int i;
    if (l == h)
        printf("%s", s);
    else
    {
        for (i = l; i <= h; i++)
        {
            swap(s[l], s[i]);
            perm(s, l+1, h);
            swap(s[l], s[i]);
        }
    }
}
```

MATRICES

SPECIAL MATRICES

↳ Only square matrix
n x n

1. Diagonal Matrix
2. Lower Triangular
3. Upper Triangular
4. Symmetric Matrix
5. Tridiagonal Matrix
6. Band Matrix
7. Toeplitz Matrix
8. Sparse Matrix

1. DIAGONAL MATRIX

	1	2	3	4	5
1	3	0	0	0	0
2	0	7	0	0	0
3	0	0	4	0	0
4	0	0	0	9	0
5	0	0	0	0	6

A

3	7	4	9	6
0	1	2	3	4

ROWS COLUMNS ELEMENT TO BE INSERTED

```
int A[5];
void set ( int A[], int i, int j, int x)
{
        if (i==j)
                A[i-1] = x;
}
void get ( int A[], int i, int j)
{
        if (i==j)
                return A[i-1];
        else
                return 0;
}
```

C++ Class For Diagonal Matrix

```
Class Diagonal()
{
    private :
        int n;
        int *A;

    public :
        Diagonal(int n)
        {
            This -> n = n;
            A = new int(n);
        }

        void set(int i, int j, int k);
        void get(int i, int j);
        void Display();
        ~ Diagonal()
        {
            delete []A;
        }
};
```

```
void Diagonal :: set(int i, int j, int x);
{
    if (i == j)
        A[i-1] = x;
}
```

```
int Diagonal :: get(int i, int j)
{
    if (i==j)
        return A[i-1];
    else
        return 0;
}
```

```
void Diagonal :: Display()
{
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            if(i==j)
                cout << A[i-1];
            else
                cout << " 0 ";
        }
        Cout << endl;
    }
}
```

LOWER TRIANGULAR MATRIX

$$M = \begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

$M[i,j] = 0$ if $i < j$

$M[i,j]$ = Non Zero if $i >= j$

Non Zero $= 1+2+3+4+5$

$= 1+2+3+4 \ldots\ldots n$

$= \frac{n(n+1)}{2}$

Zero $= n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$

ROW MAJOR

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
row 1	row 2		row 3			row 4				row 5				

Index (A[4][3]) = [1+2+3] +2 = 8

Index (A[5][4]) = [1+2+3+4] +3 = 13

Index (A[i][j]) = $\left[\frac{i(i-1)}{2}\right] + j - 1$

COLUMN MAJOR

a_{11}	a_{21}	a_{31}	a_{41}	a_{51}	a_{22}	a_{32}	a_{42}	a_{52}	a_{33}	a_{43}	a_{53}	a_{44}	a_{54}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
col 1					col 2				col 3			col 4		col 5

Index (A[4][4]) = [5 + 4 + 3] + 0 = 12

Index (A[5][4]) = [5 + 4 + 3] + 1 = 13

Index (A[5][3]) = [5 + 4] + 2 = 11

Index (A[i][j]) = [n + n-1 + n-2 + ... n - (j-2)] + (i - j)

$= [n(j-1) - [1+2+3+\ldots\ldots j-2]] + (i-j)$

$= \left[n(j-1) - \frac{(j-2)(j-1)}{2}\right] + (i-j)$

UPPER TRIANGULAR MATRIX

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} \\ 0 & 0 & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{bmatrix}$$

$M[i,j] = 0 \quad \text{if} \quad i > j$

$M[i,j] = \text{Non Zero} \quad \text{if} \quad i <= j$

$$\text{Non Zero} = 5+4+3+2+1 = \frac{n(n+1)}{2}$$

$$\text{Zero} = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

ROW MAJOR

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{22}	a_{23}	a_{24}	a_{25}	a_{33}	a_{34}	a_{35}	a_{44}	a_{45}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
row1					row2				row3			row4		row5

INDEX (A[4][5]) = [5+4+3]+1 = 13

INDEX (A[i][j]) = [n+ n-1+n-2 +.... + n- (i-2)] + (j-i)

$$= \left[(i-1)n - \frac{(i-2)(i-1)}{2}\right] + (j-i)$$

COLUMN MAJOR

a_{11}	a_{12}	a_{22}	a_{13}	a_{23}	a_{33}	a_{14}	a_{24}	a_{34}	a_{44}	a_{15}	a_{25}	a_{35}	a_{45}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

INDEX (A[4][5]) = [1+2+ 3+4] +3 =13

INDEX (A[i][j]) = [1+2+3+.... j-1] + i-1 $= \left[\frac{j(j-1)}{2}\right] + i - 1$

SYMMETRIC MATRIX

$$M = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 3 & 3 & 3 & 3 \\ 2 & 3 & 4 & 4 & 4 \\ 2 & 3 & 4 & 5 & 5 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

if $M[i,j] = M[j,i]$

Either we can store lower triangular matrix, or we can store upper triangular matrix

TRI DIAGONAL MATRIX

$$M = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

a_{21}	a_{32}	a_{43}	a_{54}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{12}	a_{23}	a_{34}	a_{45}
0	1	2	3	4	5	6	7	8	9	10	11	12

lower diagonal | main diagonal | upper diagonal

Main diagonal $i-j=0$
Lower diagonal $i-j=1$
Upper diagonal $i-j=-1$

$|i-j| \leq 1$

$M[i,j]$ = Non Zero if $|i-j| \leq 1$
$M[i,j]$ = Zero if $|i-j| > 1$

5 + 4 + 4
n + n-1 + n-1
3n−2

Index (A[i][j])

case 1 : if $i-j=1$ index = i-1

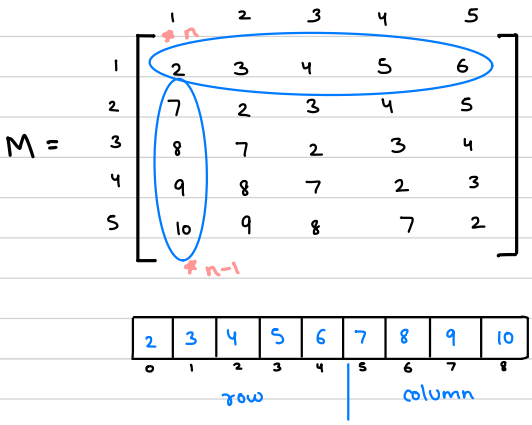
case 2 : if $i-j=0$ index = n-1 + i-1

case 3 : if $i-j=-1$ index = 2n-1 + i-1

SQUARE BAND MATRIX (Same as TRI DIAGONAL matrix)

When there are more than one diagonals below the main diagonal and the number of lower and upper diagonal is equal.

TOEPLITZ MATRIX



$M[i,j] = M[i-1, j-1]$

No of elements we want to store : $n + n-1$ *

Index A[i][j]

case 1 : if $i <= j$ Index = $j-1$

case 2 : if $i > j$ Index = $n + i - j - 1$

CREATING A DYNAMICALLY ALLOCATED ARRAY

```
int *A, n;
printf("Enter dimension");
scanf("%d", &n);
A = (int*)malloc(n*sizeof(int));
A = new int[n]; C++
```

SPARSE MATRIX

↳ Having many number of non-zero elements

Methods for storing sparse matrix

1. Coordinate list / Three column representation
2. Compressed sparse row

1. Coordinate list / Three column representation

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	3	0
2	0	0	8	0	0	10	0	0	0
3	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	2	0	0	0	0	0	0
7	0	0	0	6	0	0	0	0	0
8	0	9	0	0	5	0	0	0	0

8 x 9 — 72 elements

72 x 2 = 144 bytes

row	column	element
8	9	8
1	8	3
2	3	8
2	6	10
4	1	4
6	3	2
7	4	6
8	2	9
8	5	5

2. Compressed sparse row

A[3, 8, 10, 4, 2, 6, 9, 5] → Non Zero elements

IA[0, 1, 3, 3, 4, 4, 5, 6, 8] (0 1 2 3 4 5 6 7 8 → Row Number) → Cummulative sum of number of non zero elements in each row

JA[8, 3, 6, 1, 3, 4, 2, 5] → No of column in which each non zero element is located

8 + 9 + 8 = 25 x 2 = 50 bytes

1. Coordinate List / Three column representation (ADDITION)

$A =$

	1	2	3	4	5	6
1	0	0	0	6	0	0
2	0	7	0	0	0	0
3	0	2	0	5	0	0
4	0	0	0	0	0	0
5	4	0	0	0	0	0

0	1	2	3	4	5
5	1	2	3	3	5
6	4	2	2	4	1
5	6	7	2	5	4

$B =$

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	0	3	0	0	5	0
3	0	0	2	0	0	7
4	0	0	0	9	0	0
5	8	0	0	0	0	0

0	1	2	3	4	5	6
5	2	2	3	3	4	5
6	2	5	3	6	4	1
6	3	5	2	7	9	8

$C =$

	1	2	3	4	5	6
1	0	0	0	6	0	0
2	0	10	0	0	5	0
3	0	2	2	5	0	7
4	0	0	0	9	0	0
5	12	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9
5	1	2	2	3	3	3	3	4	5
6	4	2	5	2	3	4	6	4	1
	6	10	5	2	2	5	7	9	12

Q

	1	2	3	4	5
1	0	0	7	0	0
2	2	0	0	5	0
3	9	0	0	0	0
4	0	0	0	0	4

4 x 5 mxn

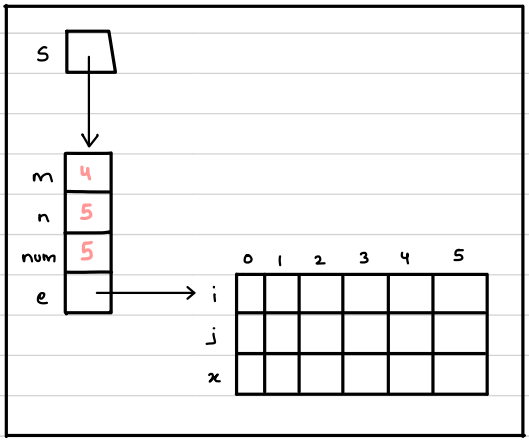
	0	1	2	3	4	5
i	4	1	2	2	3	4
j	5	3	1	4	1	5
k	5	7	2	5	9	4

CREATING SPARSE MATRIX PROGRAM

```
Struct Element
{
        int i;
        int j;
        int x;
};
Struct sparse
{
        int m;
        int n;
        int num;
        Struct Element *e;
};
void main()
{
        Struct sparse s;
        create (&s);
}
```

```
void create ( struct sparse *s)
{
    int i;
    printf(" Enter dimensions");
    scanf (" %d %d", &s->m, &s->n);
    printf(" Enter number of non-zero elements");
    scanf (" %d", &s->num);
    s->e = new Elements [s->num];
    printf(" Enter all elements ");

    for (i=0; i< s->num; i++)
        scanf(" %d %d %d", &s->e[i].i,
                           &s->e[i].j,
                           &s->e[i].x);
}
```

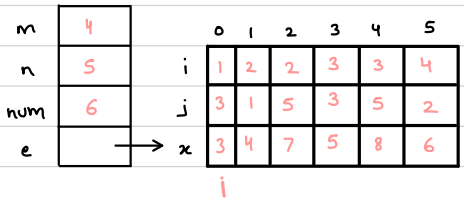
```
s->e = (Struct Element *) malloc
(s->num* sizeof(Struct Element))
```



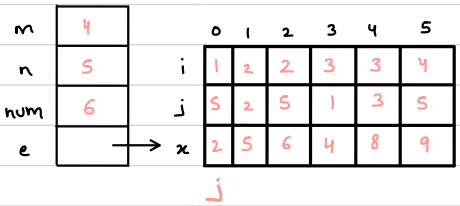
ADDING SPARSE MATRIX PROGRAM

	1	2	3	4	5
1	0	0	3	0	0
2	4	0	0	0	7
3	0	0	5	0	8
4	0	6	0	0	0

	1	2	3	4	5
1	0	0	0	0	2
2	0	5	0	0	6
3	4	0	8	0	0
4	0	0	0	0	9

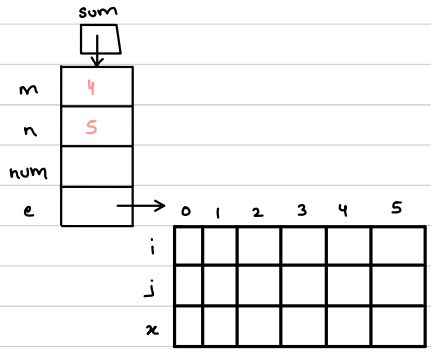


i



j

```
add ( struct sparse *s1 , struct sparse s2)
{
        struct Sparse *sum;

        if ( s1->m != s2->m || s1->n != s2->n)
                return 0;
        sum = new sparse;
        sum->m = s1->m;
        sum->n = s1->n;
        sum->e = new Element[s1->num + s2->num];

        while (i< s1->num && j< s2->num)
        {
                if ( s1.e[i].i < s2.e[j].i )
                        sum->e[k++] = s1->e[i++];

                else if ( s1->e[i].i > s2->e[j].i)
                        sum->e[k++] = s2->e[j++];

                else
                {
                        if ( s1->e[i].j < s2->e[j].j)
                                sum->e[k++] = s1->e[i++];
                        else if ( s1->e[i].j > s2->e[j].j )
                                sum->e[k++] = s2->e[j++];

                        else
                        {
                                sum->e[k] = s1->e[i++];
                                sum->e[k++].x += s2->e[j++].x;
                        }
                }
        }
```



k

Rows are Compared

Column compared

POLYNOMIAL REPRESENTATION

1. Polynomial Representation
2. Evaluation of Polynomial
3. Addition of two Polynomials

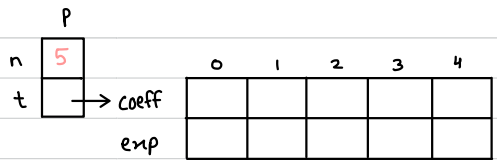
$p(x) = 3x^5 + 2x^4 + 5x^2 + 2x + 7$

coeff	3	2	5	2	7
exp	5	4	2	1	0

$n = 5$

```
Struct Term
{
    int coeff;
    int Exp;
}

Struct Poly
{
    int n;
    Struct Term *t;
}
```

```
Struct Poly P;
printf(" No of non-zero terms ");
scanf (" %d", &p.n);
p.t = new Term[p.n];
printf(" Enter polynomial terms ");
for(i=0 ; i<p.n ; i++)
{
    printf(" Term No : %d ", i+1);
    scanf("%d %d", &p.t[i].coeff, &p.t[i].Exp);
}
```



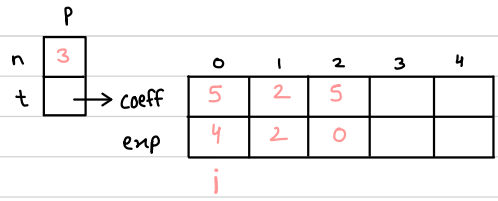
EVALUATION

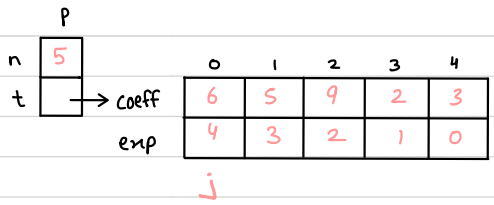
```
Struct Poly P;
x = 5 ; sum = 0 ;

for( i=0 ; i<p.n ; i++)
    sum += p.t[i].coeff * pow (x, p.t[i].Exp);
```

POLYNOMIAL ADDITION

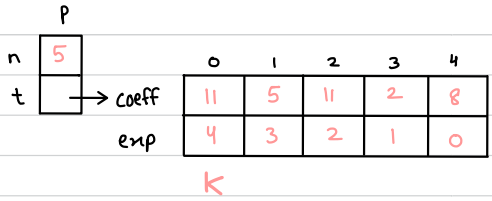
$p1(x) = 5x^4 + 2x^2 + 5$

$p2(x) = 6x^4 + 5x^3 + 9x^2 + 2x + 3$






```
while ( i< p1.n && j < p2.n)
{
        if ( p1.t[i].Exp > p2.t[j].Exp)
                p3.t[k++] = p1.t[i++];

        else if ( p2.t[j].Exp > p1.t[i].Exp)
                p3.t[k++] = p2.t[j++];

        else
        {
                p3.t[k].Exp = p1.t[i].Exp;
                p3 t[k++].coeff = p1.t[i++].coeff + p2.t[j++].coeff;
        }
}
```



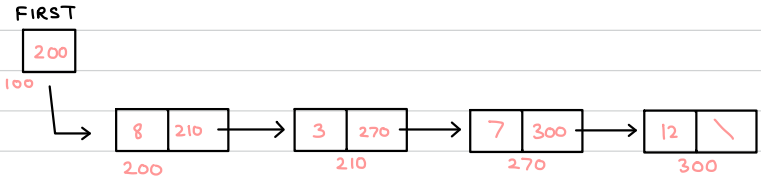
LINKED LIST

1. Problem with arrays : Fixed Size

Q What is a linked list?

Ans Linked list is a collection of nodes where each node contain data and pointer to next node.

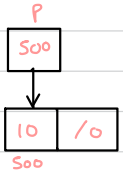
NODE




```
Struct Node                    SELF REFERENTIAL STRUCTURE
{
    int data;          ———— 2
    Struct Node * next; ———— 2   (same as datatype)
};                             4 bytes
```

```
Struct Node *p;
p = (struct Node *)malloc(sizeof(struct Node));
p = new Node;        C++
```

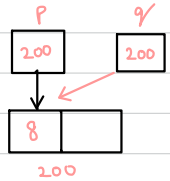
```
p → data = 10;
p → next = 0;
```



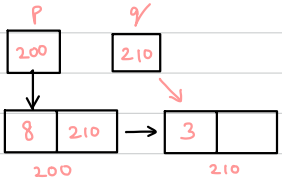
X X

```
Struct Node *p, *q;
```

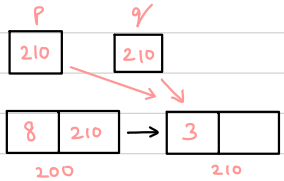
1. q=p



2. q = p → next



3. p = p → next



```
Struct Node *p = NULL;

if (p == NULL)
if (p == 0)
if (!p)
if (p->next == NULL)
```

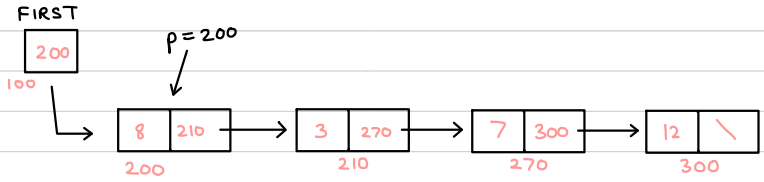
To check if pointer is not pointing anywhere

```
if (p != NULL)
if (p != 0)
if (p)
if (p -> next != NULL)
```

To check if pointer is not NULL.

TRAVERSING THROUGH LINKED LIST

```
Struct Node * p = first;
while (p != 0)
{
        p = p->next;
}
```



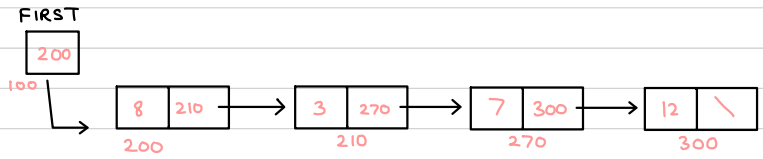

```
#include<stdio.h>
#include <stlib.h>

Struct Node
{
      int data;
      Struct Node *next;
} * first = NULL;
```

Check Struct Node * first = NULL in main

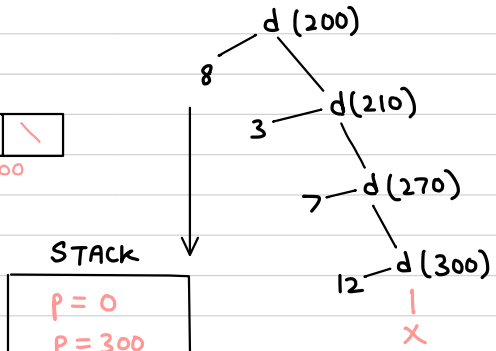
```
void create (int A[], int n)
{
     int i;
     Struct Node *t, *last;
     first = (Struct Node *)malloc (sizeof (Struct Node));
     first ->data = A[0]
     first -> next = NULL;
     last = first;
```

```
    for( i=1; i<n; i++)
    {
        t = (struct node *)malloc( sizeof ( struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}

void Display ( struct Node *p)
{
    while (p != NULL)
    {
        printf (" %d ", p -> data);
        p = p -> next;
    }
}

Void main()
{
    struct Node * temp;
    int A[] = { 3, 5, 7, 10, 25, 8, 32, 2};
    Create (A, 8);
    Display (first);
}
```

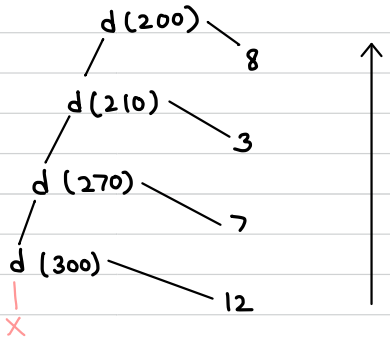
RECURSIVE DISPLAY OF LINKED LIST




```
void Display (struct Node *p)
{
    if (p != NULL)
    {
        printf (" %d", p -> data);
        Display ( p -> next);
    }
}
    Display ( first);
```



```
void Display (struct Node *p)
{
    if (p != NULL)
    {
        Display (p -> next);
        printf(" %d", p -> data);
    }
}                                   O(n)
```

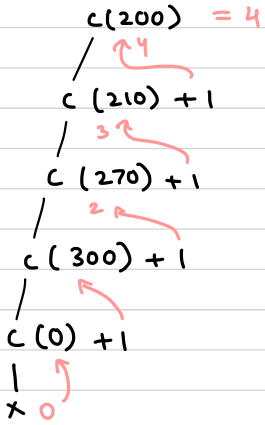


COUNTING NODES IN LINKED LIST

```
int count ( struct Node *p)
{
    int c=0;                        O(n)
    while (p!=0)
    {
        c++;
        p=p->next;
    }
    return(c);
}
```

RECURSIVE FUNCTION FOR COUNTING NUMBER OF NODES

```
int count( struct Node *p)
{                                   O(n)
    if (p==0)
        return 0;
    else
        return count(p->next)+1;
}
```



SUM OF ALL ELEMENTS IN A LINKED LIST

```
int add ( Struct Node *)
{
    int sum = 0;
    while (p)
    {
        sum = sum + p -> data;
        p = p -> next;
    }
    return sum;
}
```

USING RECURSION

```
int Add ( struct Node *p)
{                                   O(n)
    if (p==0)
        return 0;
    else
        return Add (p->next) + p->data;
}
```

MAXIMUM ELEMENT IN A LINKED LIST

```
int max ( struct Node *p)
{
    int m = -32768;
    while (p)
    {
        if (p->data > m)
            m = p->data;
        p = p->next;
    }
    return (m);
}
```

RECURSION

```
int max ( Node *p)
{
    int x = 0;
    if ( p == 0)
        return MIN_INT;
    else
    {
        x = max (p->next);
        if ( x > p->data)
            return x;
        else
            return p->data;
    }
}
```

SEARCHING IN A LINKED LIST

Binary search is not suitable for linked list as we cannot go directly in the middle of the list

LINEAR SEARCH

```
Node* Search ( Struct Node *p , int key)
{
    while ( p != NULL)
    {
        if ( key = p->data)
            return (p);

        p = p->next;
    }
    return NULL;
}
```

RECURSIVE

```
Node * Search ( Node *p , int key)
{
    if (p == NULL)
        return NULL;
    if ( key == p->data)
        return (p);
    return Search (p->next , key);
}
```

IMPROVING SEARCHING

```
Node * search ( Node *p, int key)
{
        Node *q = NULL;

        while ( p != NULL)
        {
                if ( key == p->data)
                {
                        q->next = p->next;
                        p->next = first;
                        first = p;
                }
                q = p;
                p = p->next;
        }
}
```

INSERTING IN A LINKED LIST

Before first Node

```
Node *t = new Node;
t->data = x;
t->next = first;
first = t
```

Pos == 4

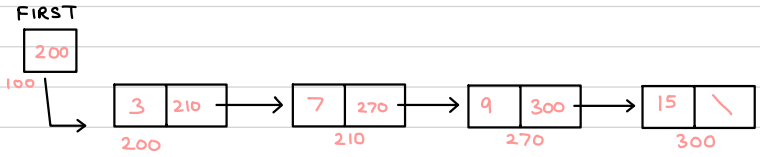
```
Node *t = new Node;
t->data = x;
p = first;
for(i=0; i < pos-1; i++)
        p = p->next;
t->next = p->next;
p->next = t;
```

```
void Insert ( int pos, int x)
{
    Node *t, *p;
    if (pos == 0)
    {
        t = new Node;
        t->data = x;
        t->next = first;
        first = t;
    }

    else if (pos > 0)
    {
        p = first;
        for (i=0; i< pos-1 && p; i++)
            p = p->next;
        if (p)
        {
            t = new Node;
            t->data = x;
            t->next = p->next;
            p->next = t;
        }
    }
}
```

INSERTING AT LAST

```
void Insertlast (int x)
{
        Node *t = new Node;
        t -> data = x;
        t -> next = NULL;

        if ( first == NULL)
        {
                first = last = t;
        }
        else
        {
                last -> next = t;
                last = t;
        }
}
```

INSERTING IN A SORTED LIST




```
p = first;
q = NULL        (Tailing Pointer)

while ( p && p -> data < x)
{
        q = p;
        p = p -> next;
}
t = new Node;
t -> data = x;
t -> next = q -> next;
q -> next = t;
```

DELETION FROM LINKED LIST

(1) Deletion of first Node

```
Node *p = first;          O(1)
first = first -> next;
x = p -> data;
free (p);
```

(2) Deletion from a given position

```
Node *p = first;          min O(1)
Node *q = NULL;           max O(n)

for (i = 0 ; i < pos-1 ; i++)
{
    q = p;
    p = p -> next;
}

q -> next = p -> next;
x = p -> data;
free (p);
```

CHECK IF LIST IS SORTED

```
int x = -32768;           O(n)
Node *p = first;
while (p != NULL)
{
    if (p -> data < x)
        return -1;
    x = p -> data;
    p = p -> next;
}
return 0;
```

REMOVE DUPLICATES FROM LIST

```
Node *p = first;                    O(n)
Node *q = first->next;

while (q != NULL)
{
    if (p->data != q->data)
    {
        p = q;
        q = q->next;
    }
    else
    {
        p->next = q->next;
        free(q);
        q = p->next;
    }
}
```

REVERSING A LINKED LIST

(1) Reversing Elements
(2) Reversing Links

Reversing links is preferred over reversing elements because maybe there are lots of values in a single node.

(1) Reversing Elements

First create an array A equal to size of length of linked list.

```
p = first;                          O(n)
i = 0;

while (p != NULL)
{
    A[i] = p->data;
    p = p->next;
    i++;
}
```

```
p = first; i--;

while (p != NULL)
{
    p->data = A[i--];
    p = p->next;
}
```

(2) Reversing Links TRACE IT!

```
p = first;
q = NULL;
r = NULL;
while (p != NULL)
{
    r = q;
    q = p;
    p = p->next;
    q->next = r;
}
first = q;
```

Sliding pointers

REVERSING LINKED LIST USING RECURSION

```
void Reverse (Node *q, Node *p)
{
    if (p != NULL)
    {
        Reverse (p, p->next)
        p->next = q;
    }
    else
        first = q;
}
```

CONCATENATING TWO LINKED LIST

```
p = first;                    O(n)
while (p->next != NULL)
    p = p->next
p->next = second;
second = NULL;
```