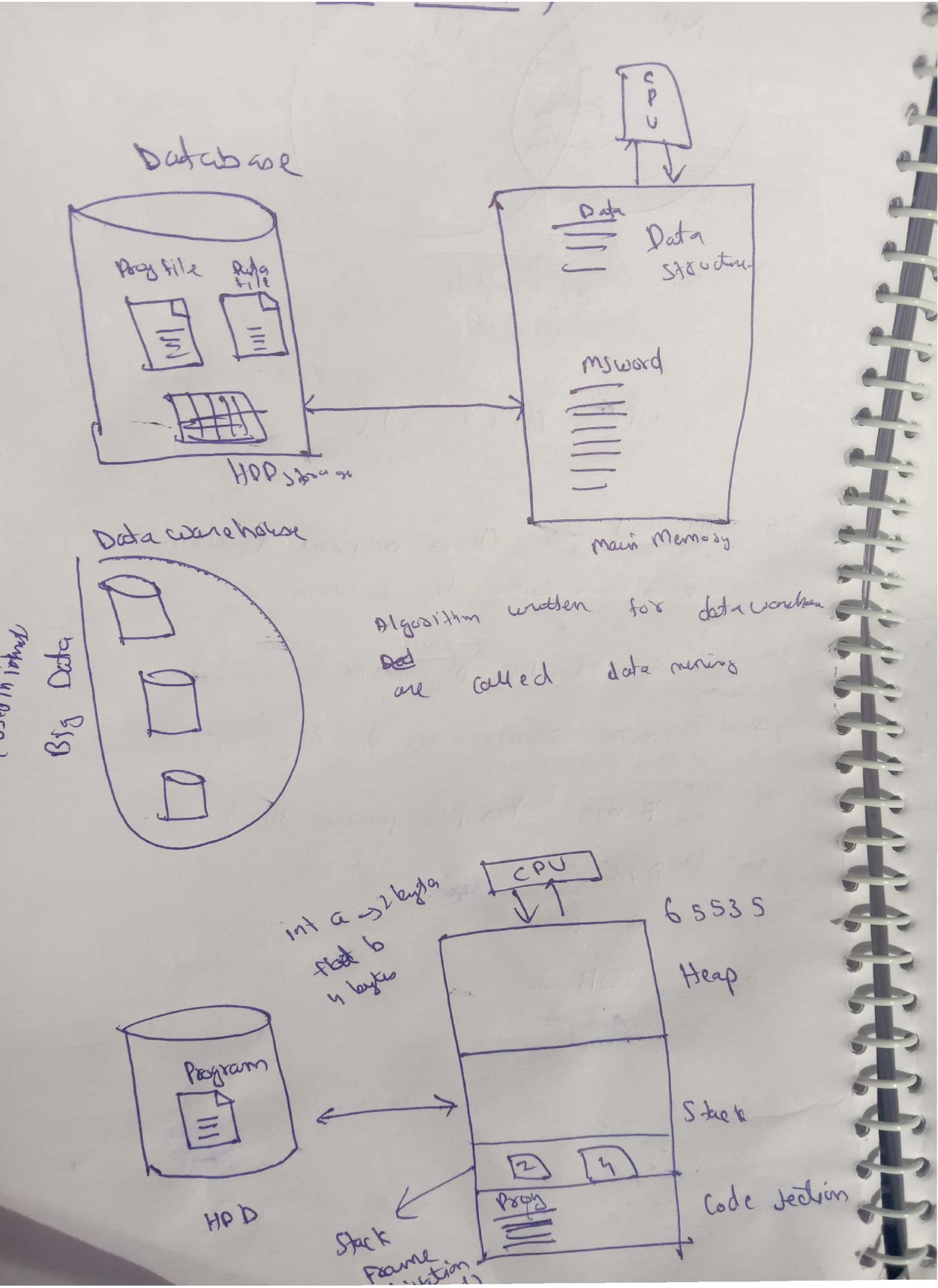
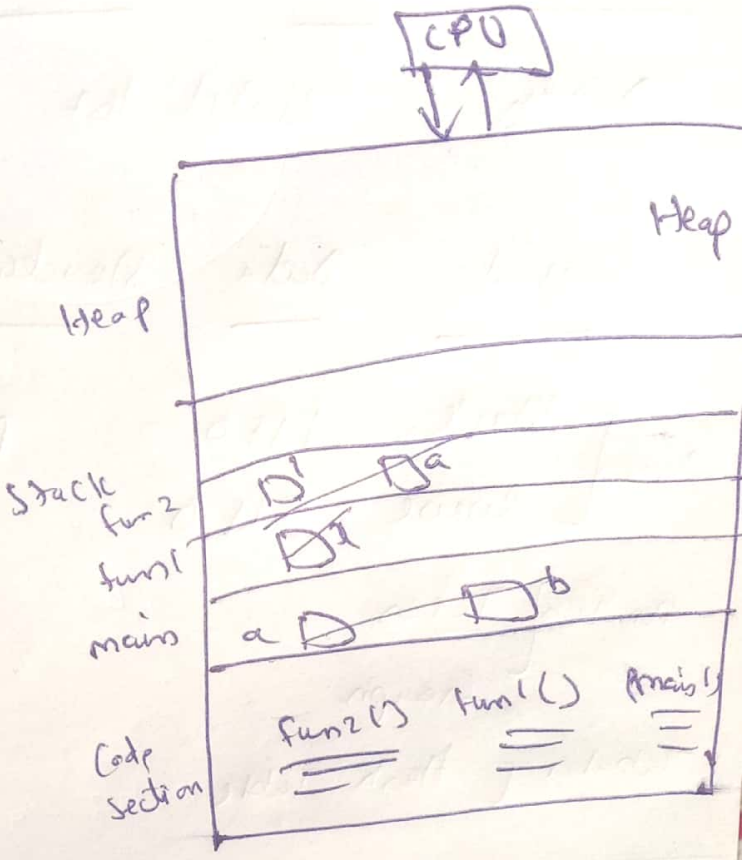

Database
Prog file
Data file
HDD storage
CPU
Data
Data structure
msword
Main Memory
Data warehouse
Big Data
Algorithm written for data warehouse are called data mining
CPU
int a -> 2 bytes
float b
4 bytes
65535
Heap
Stack
Program
HDD
2
4
Prog
Code section
Stack
Scanned by CamScanner

```
void fun2 (int i)
{
  int a;
  ==
}
```

```
void fun1()
{ int x;
  fun2(x);
}
```

```
void main()
{ int a;
  float b;
  ==
  fun1();
}
```



Programs cannot access heap memory directly.
They can access only using pointers.

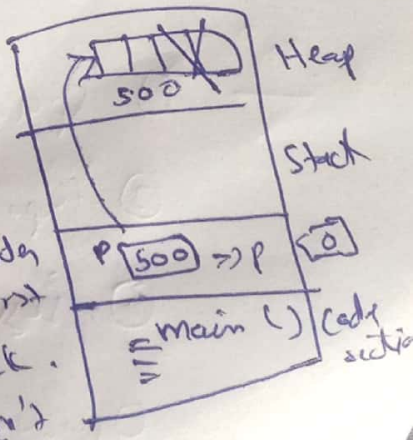
```
void main()
{ int *p;
  p = new int[5];
  p = (int *)malloc(2*5);
  p = NULL;
  delete []p;
}
```

depends on size of integer usually

sizeof(int)

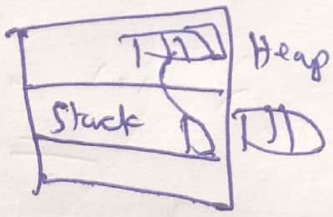
allocates memory & returns pointer to first block.

& doesn't initialize allocated memory to 0



Physical Data Structure → Defines How memory is allocated

Array Linked list



Logical Data Structure →

linear	Stack	LIFO	Discipline to define how physical data structure works
	Queue	FIFO	
non-linear	Trees		
	Graph		
tabular	Hash Table		

ADT →

It is a datatype where only behaviour is defined but not implementation. opposite of ADT is Concrete DT (CDT) where it contains implementation.

1. Representation of data
2. operation on data

List 8 3 9 4 6 10 12

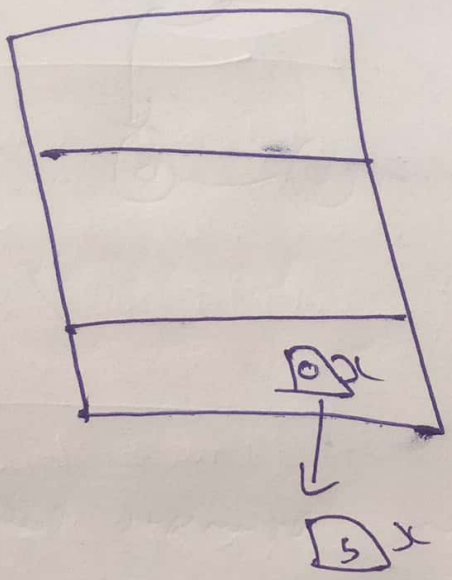
Data required for storing list:

1. Space for storing elements
2. Capacity
3. size

Operations →

Array, List, Map, Queue, Set, Stack, Table, Tree, Vectors are ADTs.

Static int x = 0;




```
int func (int n)
{ static int x = 0;
  if (n > 0)
  { x++;
    return func(n-1)
           + x;
  }
  return 0; }

main ()
{ int a = 5;
  printf ("%d", func(a));
}
```

fun 5

Types of Recursion

1. Tail → Recursive call is in last step. } ≡ fn(n);
2. Head → first
3. Tree
4. Indirect
5. Nested

Linear Recursion

fn calls itself only one time

Tree Recursion

```
{
  fn(n-1)
  fn(n-1)
}
```

when one or more fn calls each other

A()
{ B() }

B()
{ A() }

Nested →

Recursive fn passes recursive call as argument

```
int fn(int n)
{ if (n > 100)
    return n-10;
  else
    fn(fn(n+11));
}
fn(95);
```

"m" →

```
int pow(int m, int n)
{ if(n==0)
    return 1;
  return pow(m, n-1) * m;
}
```

```
int pow(int m, int n)
{
    if(n==0)
        return 1
    if(n%2==0)
        return pow(m*m, n/2);
    else
        return m*pow(m*m, (n-1)/2);
}
```

Q Taylor using Horner's rule

$1+\left[\frac{x}{1}\left[1+\frac{x}{2}\left[1+\frac{x}{3}\left[1+\frac{x}{4}\right]\right]\right]\right]$

```
int e(int x, int n)
{
    static int s=1;
    if(n==0)
        return s;
    s = 1+x/n*s;
    return e(x, n-1);
}
```

Q. ^nC_r →

```
int C(int n, int r)
{
    if(r==0 || n==r)
        return 1;
    else return C(n-1, r-1) + C(n-1, r);
}
```

int **A;

A = new int*[3]; A[3][4].

A[0] = new int[4];

Addr A[i][j] = Lo + i x w

Base index size of data type
Addr.

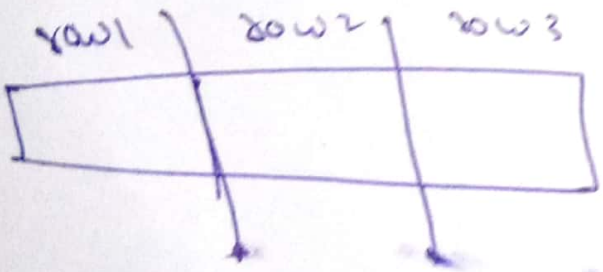
int A[1..5]

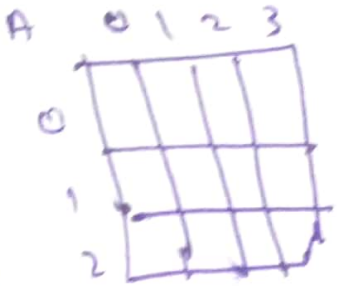
⇒ Addr[3] = 200 + 2 x 2

= 204

Row Major →

int A[3][4];
m n



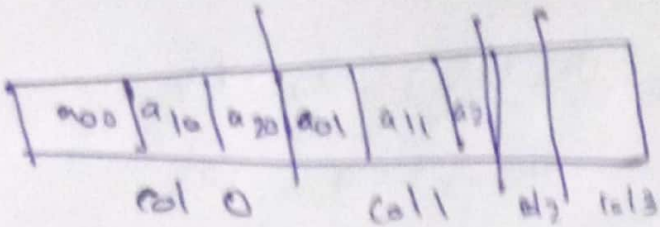


Addr[A[i][j]) = Lo + [i x n + j] * w

int A[1..3][1..4]

Addr [A[i][j]] = Lo + [(i-1)n + (j-1)] x w

Column Major →



Addr (A[i][j]) = lo + [j x m + i] * w

ND

A[d1][d2][d3][d4]

Row Major

Addr (A[i1][i2][i3][i4])

= lo + [i1 * d2 * d3 * d4

+ i2 * d3 * d4 + i3 * d4

+ i4] * w

Column Maj'or

Addr = lo + [i4 * d3 * d2 * d1 + i3 * d2 * d1

+ i2 * d1 + i1] * w

- For finding missing elements in sequence

A[i] - i ≠ diff

- Permutations of a string

void Perm (char S[], int l, int h)

{ int i if(l==h) Pf(S) else { for i=l ; i<h; i++
swap(S[l],S[i]);

Special Matrices (n×n)

1. Diagonal — $m[i,j] = 0,\ i \neq j$
2. Lower Δ — $m[i,j] = 0,\ i > j$
3. Upper Δ — $m[i,j] = 0,\ i < j$
4. Symmetric (3 diagonals) — $m[i,j] = m[j,i]$
5. Tridiagonal — $m[i,j] = 0 \ \ |i-j| > 1$
6. Band (several diagonals along with main diagonal) — other are zero
7. Toeplitz — $m[i,j] = m[i-1, j-1]$
8. Sparse (most no. of 0's)

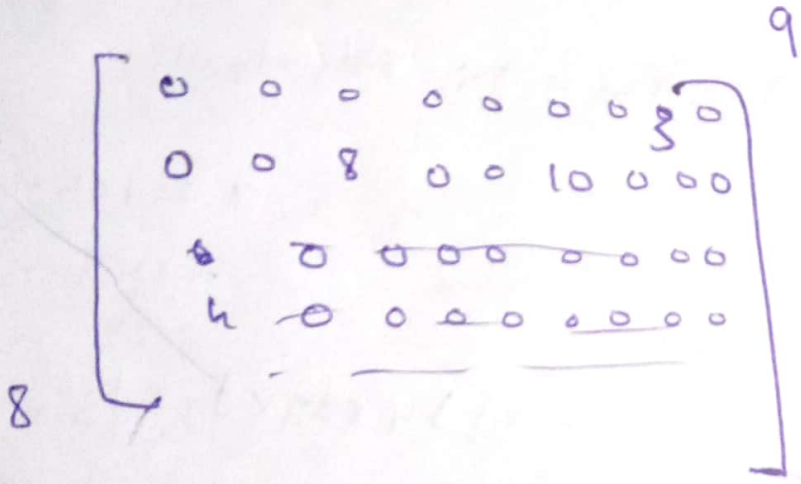
9

8

row	column	element
8	9	8
1	8	3
2	3	8
2	6	10

Special Matrices (n×n)

1. Diagonal — $m[i,j] = 0,\ i \neq j$
2. Lower Δ — $m[i,j] = 0,\ i > j$
3. Upper Δ — $m[i,j] = 0,\ i < j$
4. Symmetric (3 diagonals) — $m[i,j] >= m[j,i]$
5. Tridiagonal — $m[i,j] = 0 \ \ |i-j| > 1$
6. Lower Band — (several diagonals along with main diagonal) other are zero
7. Toeplitz — $m[i,j] = m[i-1,j-1]$
8. Sparse (most no. of 0's)



row	column	element
8	9	8
1	8	3
2	3	8
2	6	10

A [3, 8, 10, ...]

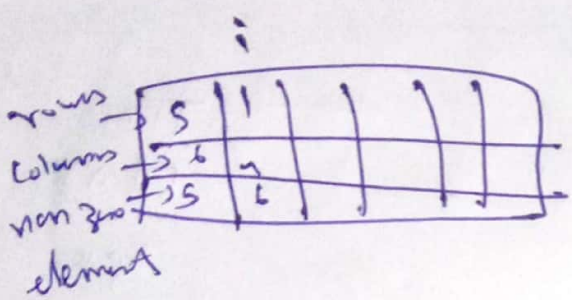
IA [0, 1, 3, 5, 4, ...]

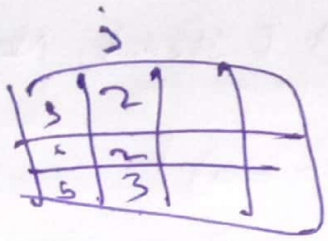
A [3, 8, ...]

JA [8, 3, 6, ...]

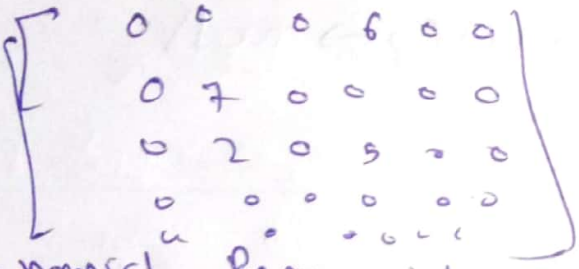
Column

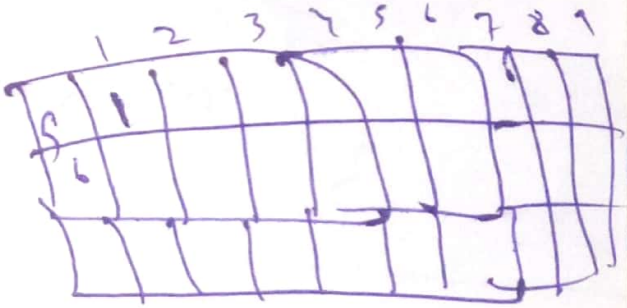
Coordinate list





compare & place rows





Polynomial Representation

$3x^5 + 2x^4 + 5x^2 + 2x + 2$

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

In class, everything by default is private,

in structure everything by default is public.

struct Node { 2 bytes int data; 2 bytes struct node * next;

```
Struct Node *p;
p = (Struct Node *) malloc (sizeof (Struct Node));
p = new Node;
p -> data = 10;
p -> next = 0;
```

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

Time
O(n)
Space
O(n)

Searching →

1. linear
2. binary ✗ (can't be performed)

O(1) O(n)

Insertion

O(1) O(n)

Remove Duplicates in a linked list →

```
Node *p = first;
Node *q = first -> next;
```

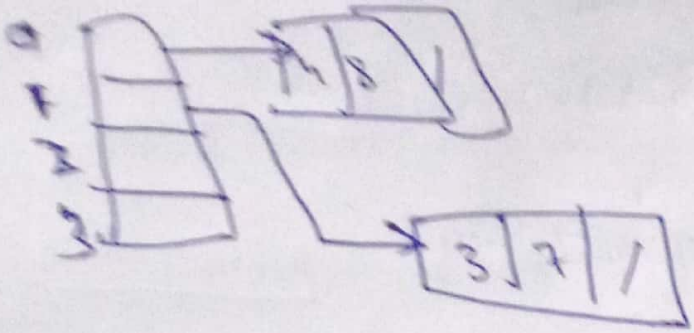
```
while(q != NULL)
{  if(p -> data != q -> data)
   {  p = q;
      q = q -> next; }
   else
   {  p -> next = q -> next;
      delete q;
      q = p -> next;
   }
}
```

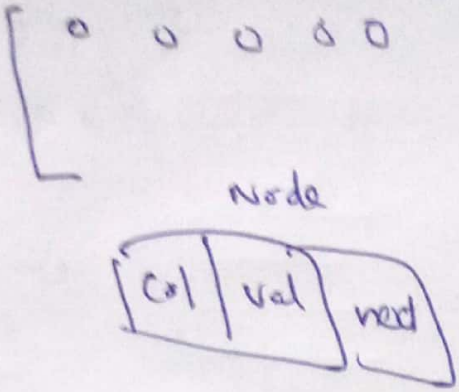
Reverse a linked list →


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

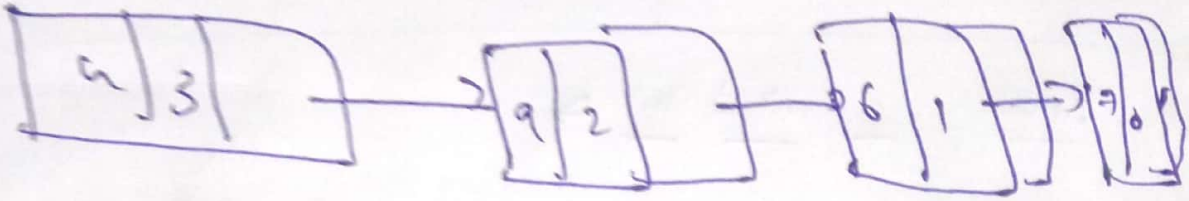
Sparse matrix Representation →

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 \\ 5 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 6 & 0 & 0 & 4 & 0 & 0 \end{bmatrix}$$ 5 x 6





$4x^3 + 9x^2 + 6x + 7$



```
#include <stdio.h>

struct Node {
    int coeff, exp;
    struct Node *next;
} *poly = NULL;

int main() {
    create();
    Display(poly, 1);
    return 0;
}

void Display(struct Node *p)
{
    while(p)
    {
        printf("%d %d", p->coeff, p->exp);
        p = p->next;
        printf("\n");
    }
}
```

```
void create() {
    struct Node *t, *last;
    int num, i;
    printf("Enter no. of terms");
    scanf("%d", &num);
    for (i=0; i<num; i++)
    {
        t = (struct Node *) malloc(sizeof(struct Node));
        scanf("%d %d", &t->coeff, &t->exp);
        t->next = NULL;
        if (poly == NULL) {
            poly = last = t; } else
```

```
long Eval (struct node *p, int x)
{ long val;
  while (p)
  { val t = p -> coeff * pow (x, p -> exp);
    p = p -> next;
  }
  return val;
}
```

Application

Polynomial Addition

Stack →

LIFO

ADT stack

Data

- Space for storing elements
- top pointer

Operations

1. push()
2. pop()
3. peek(index)
4. stackTop()
5. isEmpty()
6. isFull()

```
struct Stack
{ int size;
  int top;
  int *s;
};
```

```
int main() {
  struct Stack st;
  printf("Enter stack size: ");
  scanf("%d", &st.size);
  st.s = new int[st.size];
  st.top = -1;
}
```

Empty

top = -1

underflow

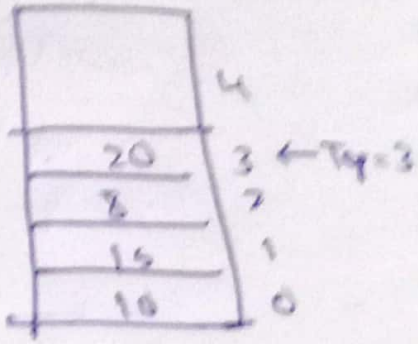
full

top = size - 1

O(1) → pop push

peek ( ) →

pos	Index = top - pos + 1
1	3 = 3 - 1 + 1
2	2 = 3 - 2 + 1
3	1 = 3 - 3 + 1

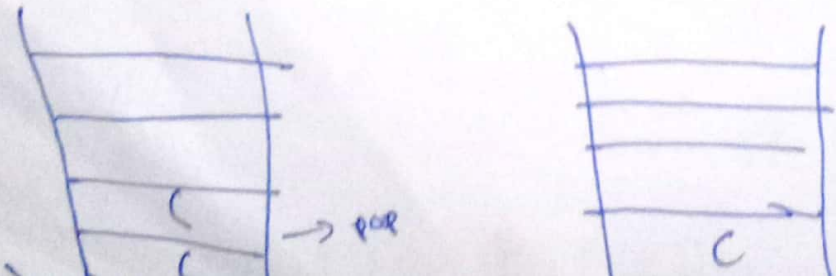


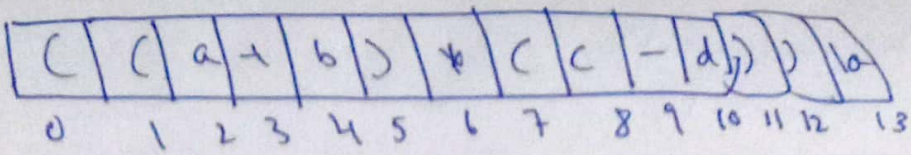
~~peek (stack st~~

```
int peek (stack st, int pos)
{   int x = -1;
    if (st.top - pos +1 < 0)
      printf("Invalid Position");
    else
      x = st.s[st.top - pos +1];
    return x;
}
```

Application →

((a+b) * (c-d))






```
int isBalance (char *exp)
{ struct Stack st;
for (i=0; exp[i] != '\0'; i++)
{ if (exp[i] == '(')
    push ( &st, exp[i]);
  else if [exp[i] == ')' )
  { if (IsEmpty (st)
      return false;
    pop(&st);
  }
}
return IsEmpty (st) ? true : false;
}
```

```
int main()
{ char *exp = "((a+b)*(c-d))*
  printf ("%d", isBalanced (exp));
return 0; }

int isBalanced (char *exp) { int i;
for (i=0; exp[i] != '\0'; i++)
{ if (exp[i] == '(') { push (exp[i]); }
  else if ( exp[i] == ')')
  { if (top == NULL)
    { return 0; } pop();
```

```
if (top == NULL)
    return 1;
else
    return 0;
}
```

Conversions →

Infix to Postfix

	Precedence
+ −	1
* /	2
( )	3

$8 + 3 * (9 - 6) / 2^2 + 6/2$

- $(a + (b * c))$
- $+a*bc$ Prefix

  $abc*+$ Postfix
- $a+b+c*d$

  → $a+b+[cd*]$

  ~~$a+bcd*+$~~ $[ab+]+[cd*]$

  ~~$abcd*++$~~ $[ab+cd*]+$
- $(a+b)*(c-d)$

  $(ab+)*(cd-)$ → $ab+cd-*$

Queue ADT →

Data

① Space for storing elements
② front → deletion
③ Rear → insertion

Operations

enqueue(x) dequeue()
isEmpty() isFull()
first() last()

Insert O(1)

Delete O(n)

initially

front = Rear = -1

enqueue → O(1)

dequeue → O(1)

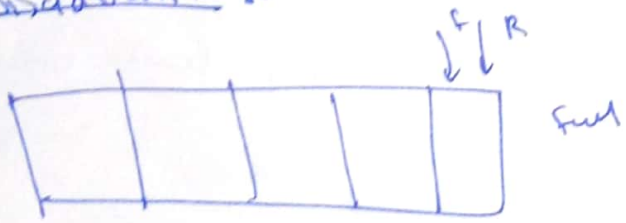
Empty →

front == rear

Full →

Rear == size - 1

Disadvantage



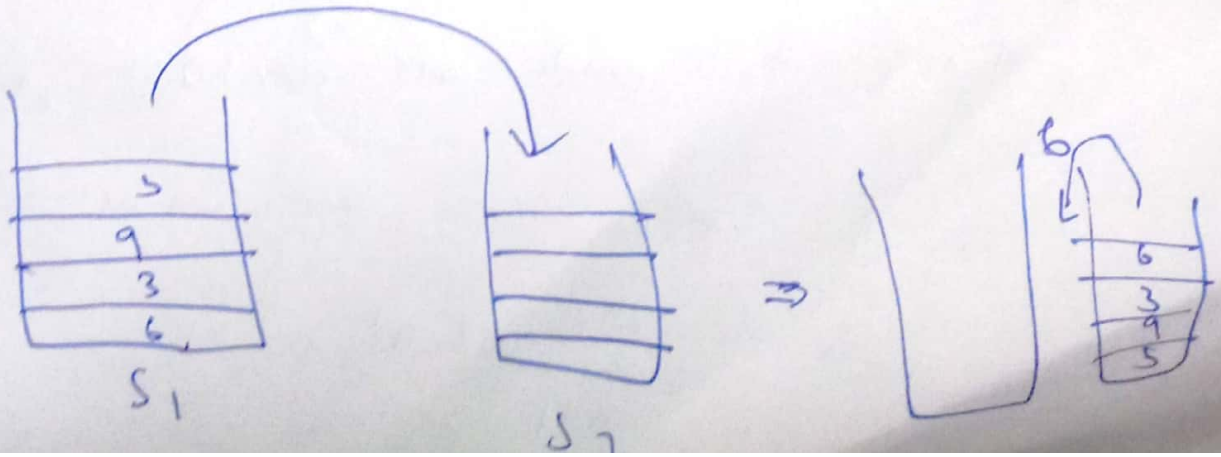
full

Solution

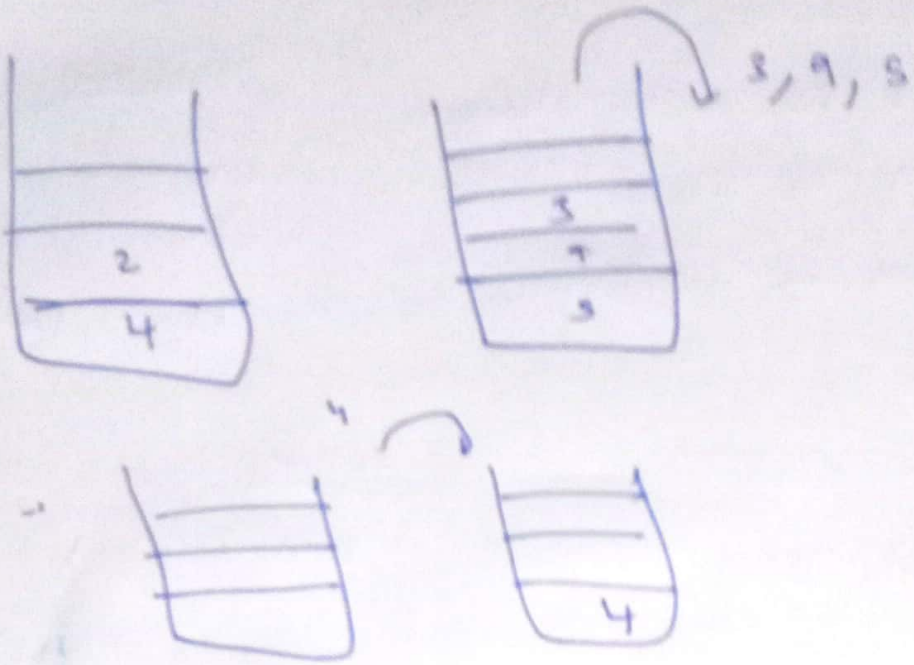
Resetting pointers
circular Queue

Queue using 2 stacks →

elements → 6, 3, 9, 5, 4, 2, 8, 12, 10..



enqueue 4



Trees →

degree → no. of direct children

Degree of tree = $\min$ (max. degree of node)

Forest = Collection of trees obtained from omitting the root

Binary tree $\deg(T) = 2$

Catalan no. : $\frac{2nC_n}{n+1}$ (shapes)

Binary trees possible with max height $= 2^n - 1$

No. of ways binary tree can be filled = $\frac{2nC_n}{n+1} \times n!$

Height Vs Nodes

min nodes = h+1

max nodes = $2^{h+1} - 1$

n-ary tree

degree n

3-ary tree

{0, 1, 2, 3}

$$\boxed{deg(0) = deg(2) + 1}$$

Strict Binary tree

{0, 2}

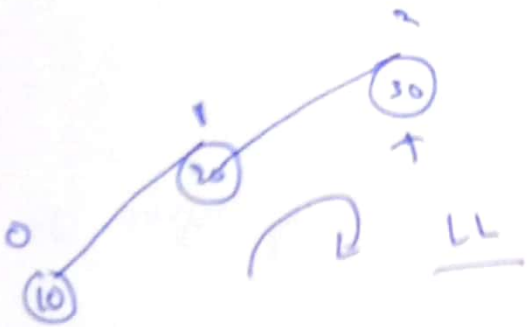
AVL ⇒ height balanced BST

Deletion

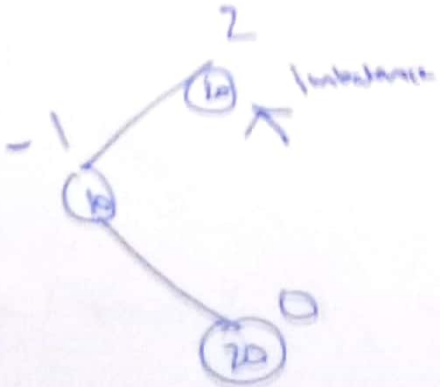
L1, L-1, L0] Rotation

R1, R-1, R0] Rotation

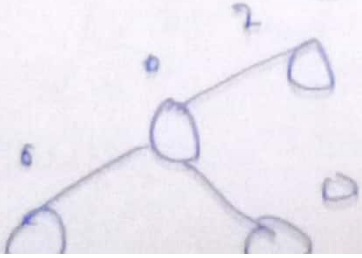
L1 →



L-1 →

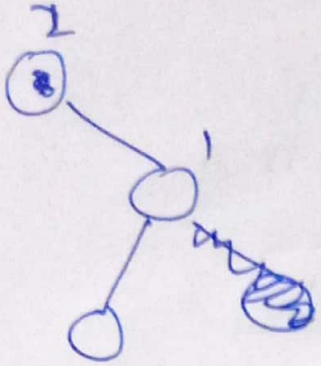


L0 →



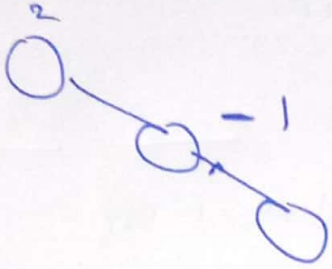
L-1 or L-1

R 1



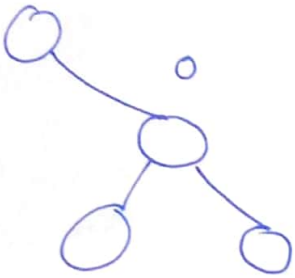
RR L

R -1



RR

R 0

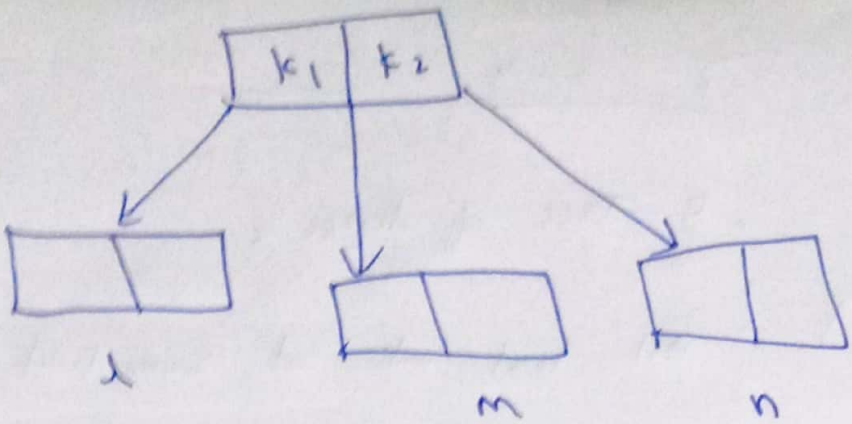


R or R -1

2-3 Trees →

(B trees of degree 3)

- Multiway Search Tree
- Degree
- B - Tree
- All leaf nodes are at same level
- Every node must have $\lceil \frac{n}{2} \rceil = 2$ children

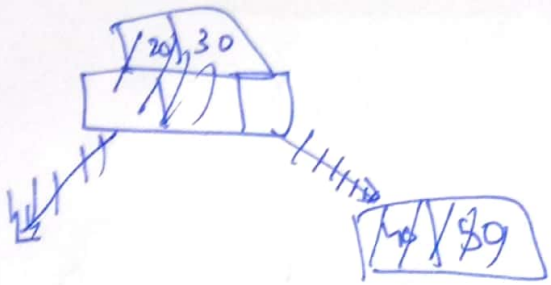


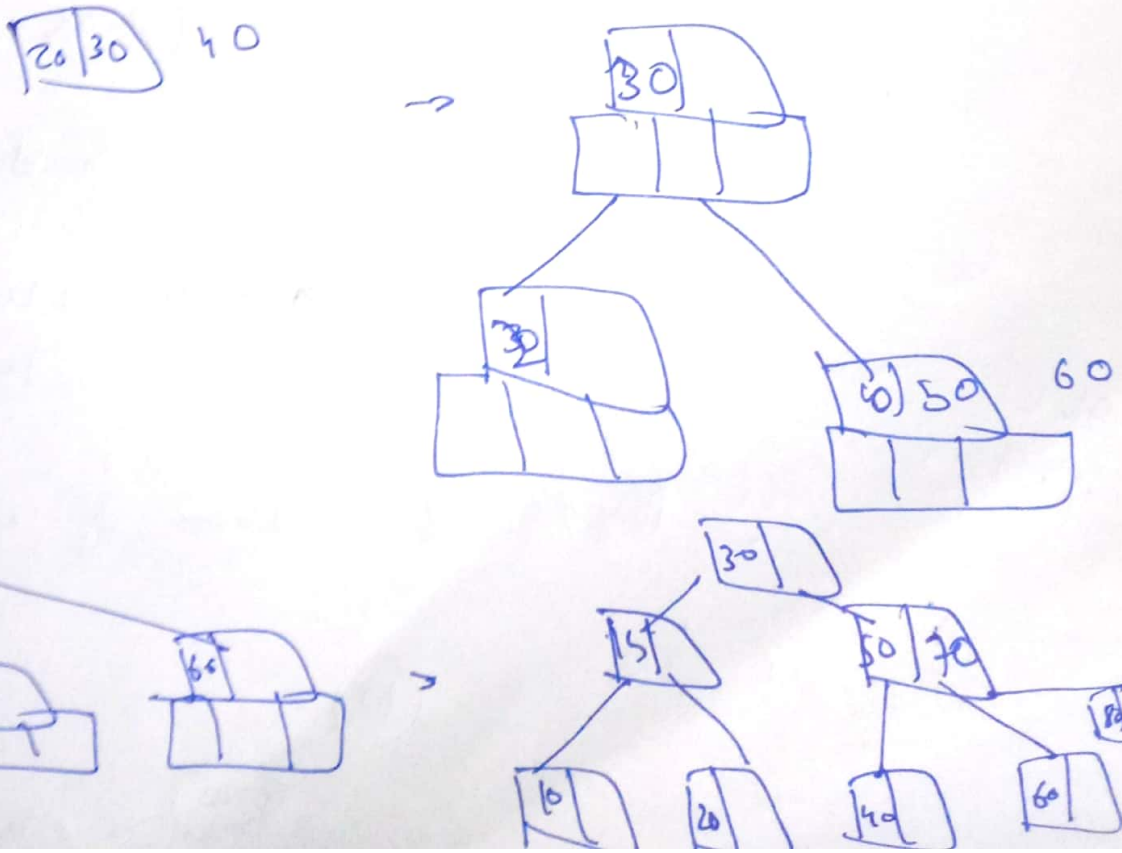
$K_1 < K_2$

$l < K_1 \qquad K_2 < n$

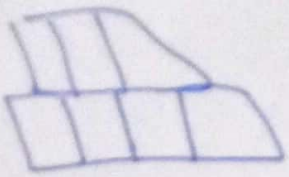
$K_1 < m < K_2$

Q. Keys → 20, 30, 40, 50, 60, 10, 15, 70, 80





2 - 3 - 4



- B-Tree of degree 4
- All leaf node at same level
- Every node must have $\lceil \frac{4}{2} \rceil = 2$

Red - Black Tree

- Height balanced BST similar to 2-3-4 Tree
- Every node is either Red or Black.
- Root → Black
  NULL
- Number of blacks on paths from root to leaf are same
- NO two red nodes should be there
- height $\log n \leq h \leq 2 \log n$
  less height than AVL

AVL
$1.44 \log n$

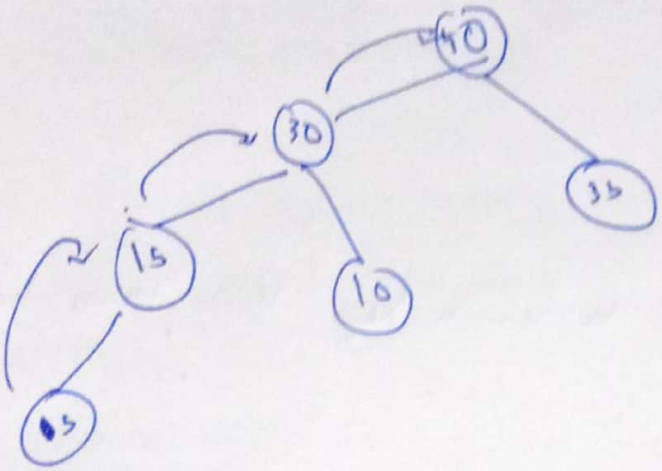
- Parent & children of red are black

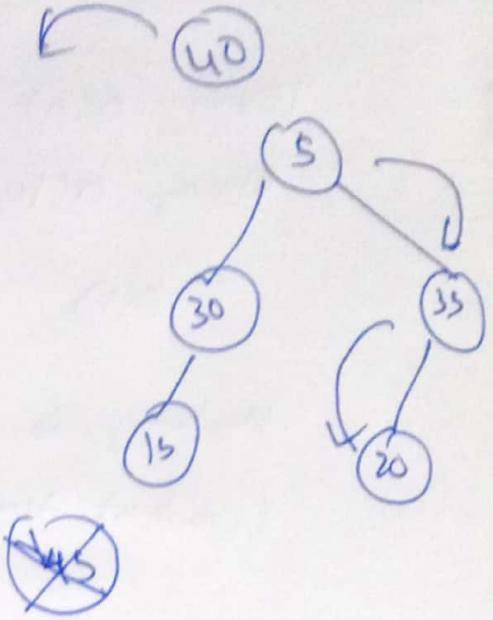
10 20 30

Heap →

Max Heap Min Heap

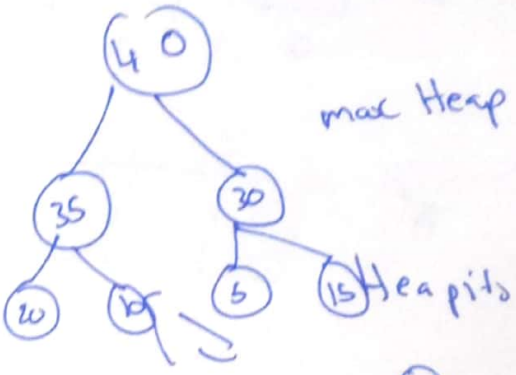
Heapify

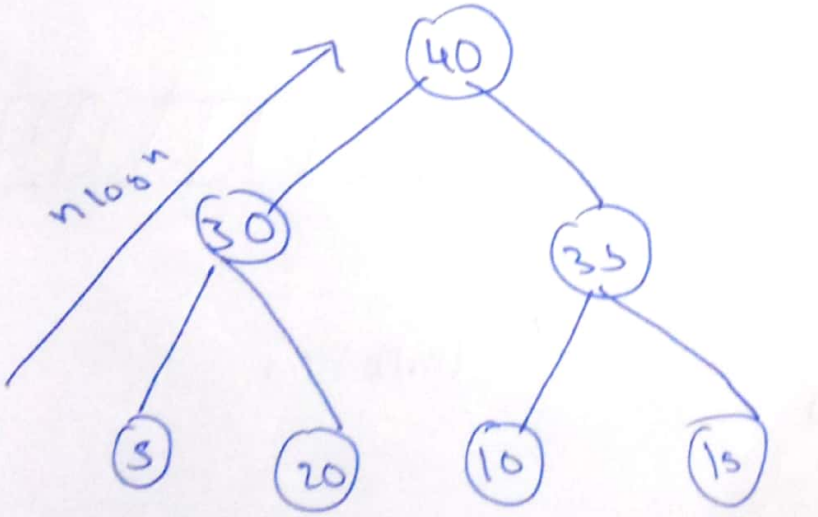


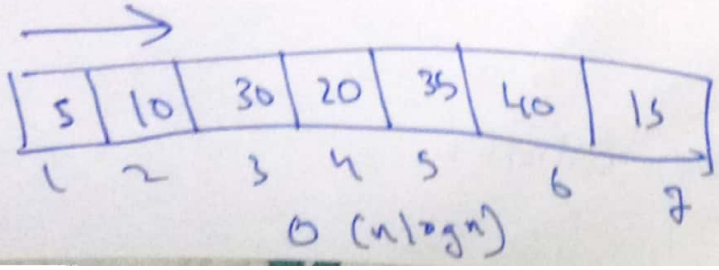


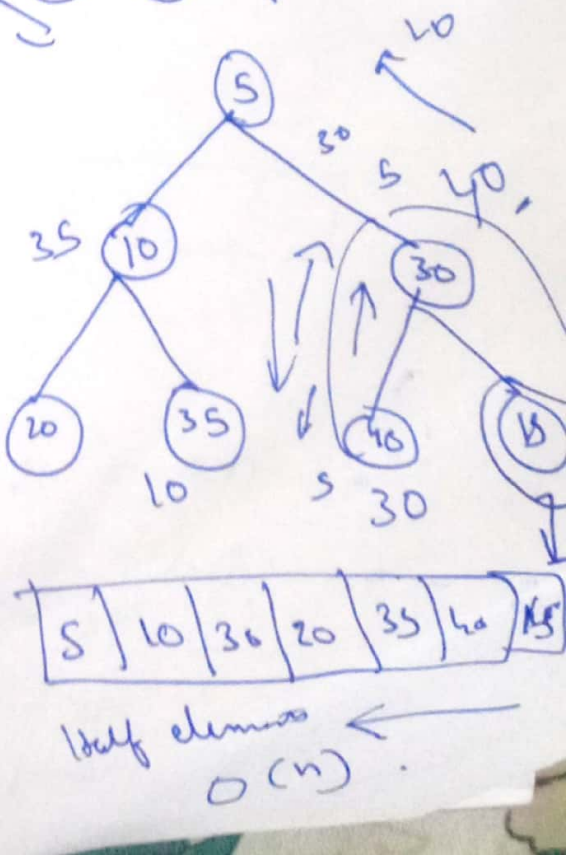
Generally

O(nlogn)









Hashing Technique

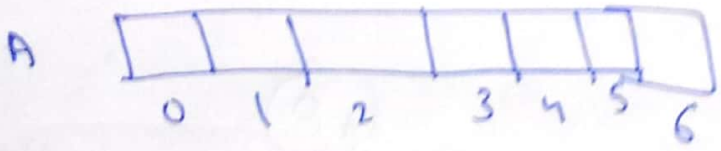
keys : 8, 3, 6, 10, 15, 18, 4

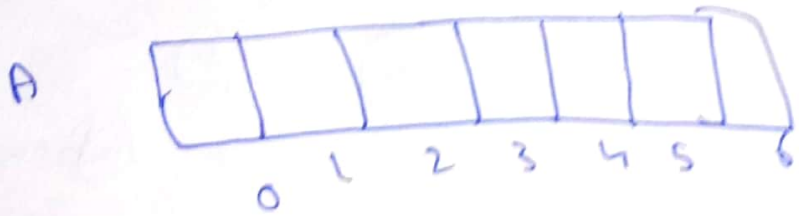
Linear O(n)

Binary O(log n)

O(1)

Hashing is used to search key less than log n
(constant time)





(collision

$h(x) = x \% 10$

Ideal

One - one

Many - one Collision.

* Open Hashing →

- Chaining

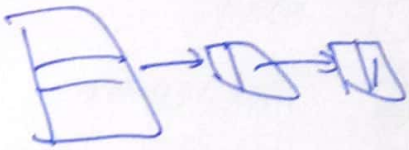
* Closed Hashing →

- Open Addressing
  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing

Chaining →

$h(x) = x \% 10$

λ (Loading factor) $= \frac{n \rightarrow \text{keys}}{\text{size}}$



Successful search →

$O(1)$ $\quad t = 1 + \frac{\lambda}{2}$

Avg. time

Avg. unsuccessful search

$t = 1 + \lambda$

Linear Probing →

$h(x) = (h(x) + i) \% 10$ where $i = 0, 1, 2, 3$

Successful $\frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right)$ $\quad \frac{1}{1-\lambda}$ unsuccessful search

Linear probing has problem of clustering

Quadratic Probing

$h(x) = (h(x) + f(i)) \% 10$

Double hashing →

$h'(x) = (h1(x) + i * h2(x)) \% 10$

$h1(x) = x \% 10$

$h2(x) = 7 - (x \% 7)$

Hash Functions →

1. mod — $h(x) = (x \% \text{size}) + 1$
2. mid square — $h(x) = x^2$ (121)
3. folding

ABC

→ 65+66+67

= 198

Skip carry or take mod.

12
>3
47
92 → 9+2 = 11

if not there we can perform mod

we can design our fn but hash fn should always give same result.

Sorting Techniques →

- Comparison Based
  - $O(n^2)$
    1. Bubble
    2. Insertion
    3. Selection
  - $O(n \log n)$
    4. Heap
    5. Merge
    6. Quick
    7. Tree
  - $O(n^{1.5})$
    8. Shell
- Index Based $O(n)$
  9. Count
  10. Bucket / Bin
  11. Radix

Criteria for Analysis →

1. Number of comparisons
2. Number of swaps
3. Adaptive
4. Stable
5. Extra Memory

Bubble Sort →

A

8	5	7	3	2
0	1	2	3	4

1st Pass

8 5 7 3 2 → 5 8 7 3 2 → 5 7 8 3 2 → 5 7 3 8 2 → 5 7 3 2 (8) one element is sorted.

repeat

4 comp

4 swap

2nd pass	3rd pass	4th pass
3 comp	2 comp	1 comp
3 swap	2 swap	1 swap

No. of passes : 4 = (n − 1) passes

No. of comparisons : 1 + 2 + 3 + 4 → $\frac{n(n-1)}{2}$ $O(n^2)$.

Max. No. of swaps : $\frac{n(n-1)}{2}$ $O(n^2)$

Stone is heavier, will settle at the bottom, bubbles are lighter they will raise up

```
void Bubblesort (int A[], int n) {
                                          int flag;
 for ( i=0; i<n-1; i++) { ~~int flag~~;  flag=0;
  for (j=0 ; j<n-1-i ; j++) {  ~~flag=0~~;
    if (A[j] > A[j+1])
      swap (A[j], A[j+1])  flag=1;        O(n^2)
    } } }
    if (flag == 0) break;
```

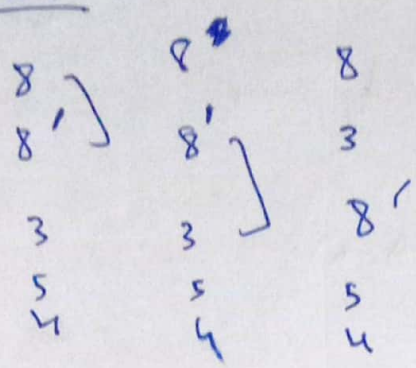
Adaptive → There can be cases when no swaps (passes) are done. (i.e. program is already sorted) (can be used to detect sorted

j → 2
3
O(n)
5
7
8

min $O(n)$

max $O(n^2)$

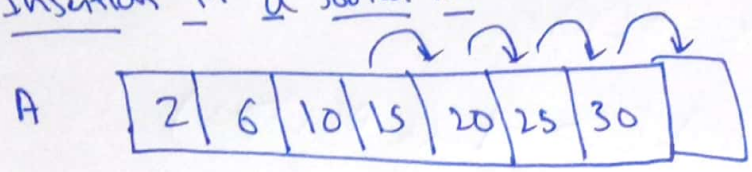
Stable



Insertion sort →

Insertion in a sorted list →

$O(1) - O(n)$

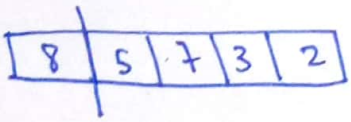


ele = 12

similarly, in linked list

Inserting element in sorted position in an array.

Insertion Sort →



1st pass

1 comp
1 swap

2nd pass

2 comp
2 swap

3rd pass

3 Comp
3 pass

4th pass

4 Comp
4 pass

No. of passes → (n-1) passes

No. of comp. → $\frac{n(n-1)}{2}$

No. of swap → $O(n^2)$

```
void insertionSort (int A[], int n)
{
    for (i=1; i<n; i++)
    {
        j = i-1;
        x = A[i];
        while (j > -1 && A[j] > x)
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = x;
    }
}
```

$O(n^2)$

Analysis →

Adaptive → By nature

Stable.

min time $O(n)$

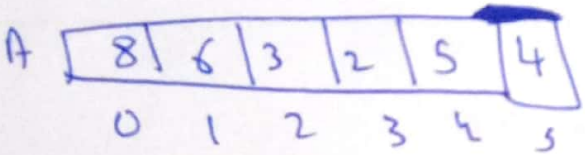
max time $O(n^2)$

min swaps $O(1)$

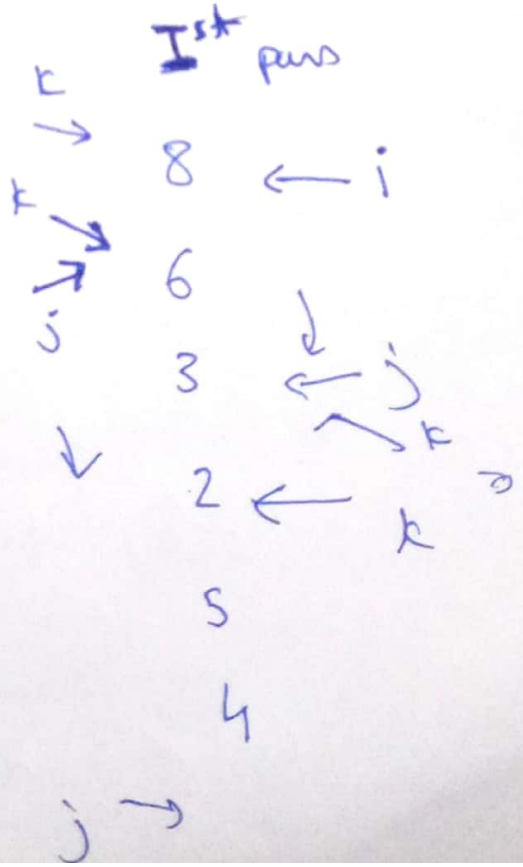
max swaps $O(n^2)$

	Bubble Sort	Insertion Sort
min Comp	$O(n)$	$O(n)$
max Comp	$O(n^2)$	$O(n^2)$
min swap	$O(1)$	$O(1)$
max swap	$O(n^2)$	$O(n^2)$
Adaptive	✓	✓ → these are only adaptive sorts
Stable	✓	✓ only 3 sorts are stable
Linked list	No	Yes
k passes	Yes (2 passes 2 largest element)	No

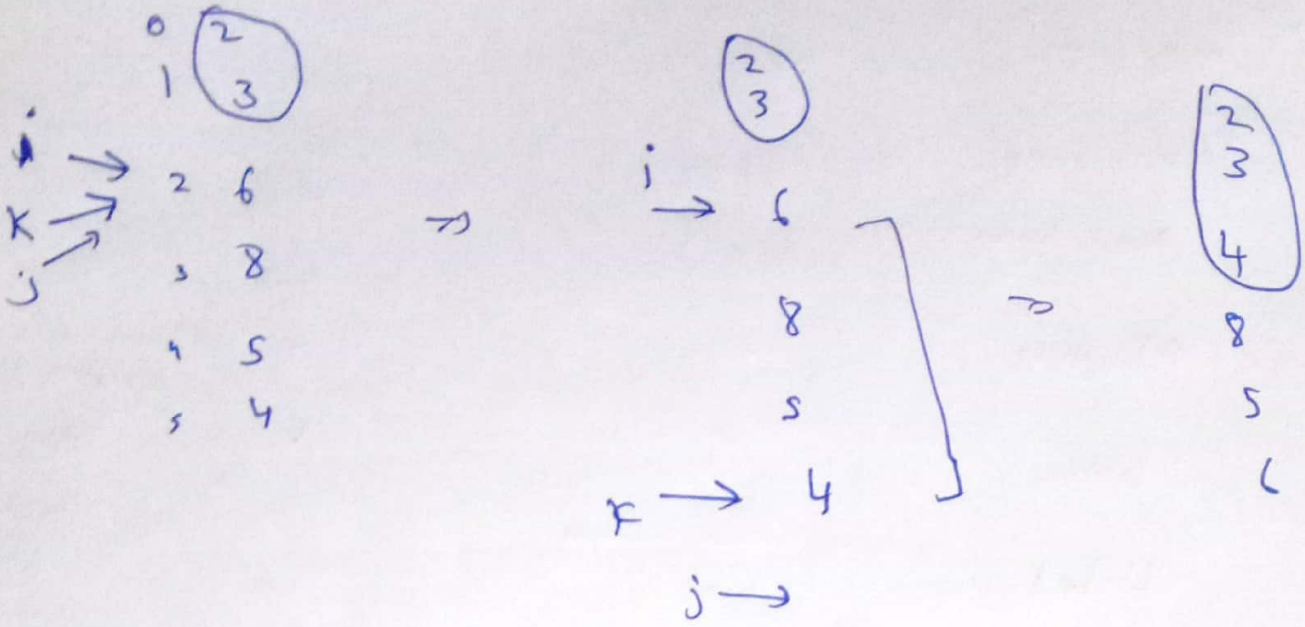
Selection Sort →



j will find minimum element & if any element is found, bring j to that position.



III rd pass



Comp. 5 4 3 2 1

Swaps 1 1 1 1 1 min swaps.

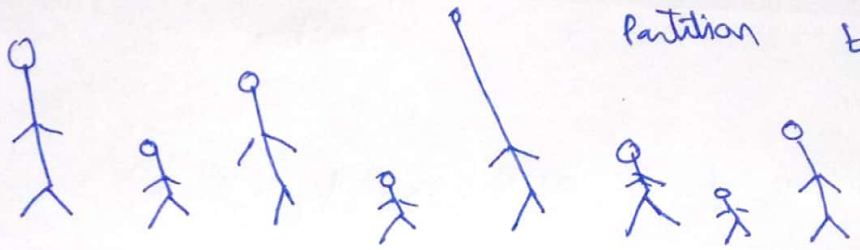
No. of Comp. $= O(n^2)$

```
void selectionSort(int A[], int n)
{
    int i, j, k;
    for (i=0; i<n-1; i++)
    {
        for (j=k=i; j<n; j++)
        {
            if(A[j] < A[k])
                k = j;
        }
        swap(A[i], A[k]);
    }
}
```

Adaptive	Stable.
2 ← i ← k	8
4	8
8	8'
10	4
12	2
5	7
X	X

→ 2 3 5 8' 4 8 7

Quick Sort (Selection Exchange / Partition Exchange sort)



→ (10) 30 20 70 40 90 80

→ 80 70 40 30 20 10 (90)

→ (40 30 20) (50) (90 70 80)

Pivot ↓

(50) 70 60 90 40 80 10 20 30

i i i j j

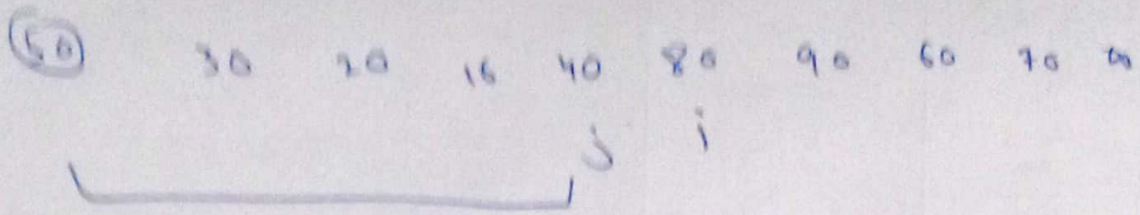
70 > 50

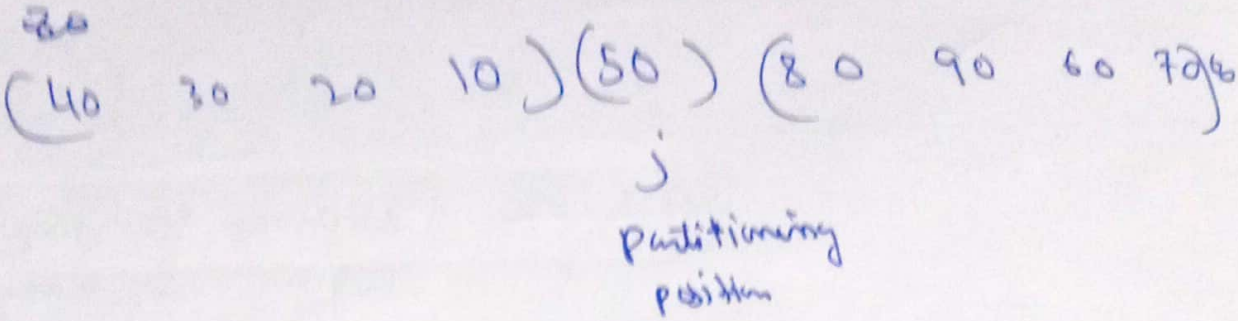
using i find element smaller than i

i should move it element is smaller it should stop for greater no.

interchange

j greater Continue Smaller exist stop, j find element smaller than i






```
int partition (int A[], int l, int h)
{  int pivot = A[l];
   int i=l, j=h;
   do {
      do {i++;} while (A[i] <= pivot);
      do {j--;} while (A[j] > pivot);
      if (i < j)
        swap (A[i], A[j]);
   } while (i<j);
   swap (A[l], A[j]);
   return j; }
```

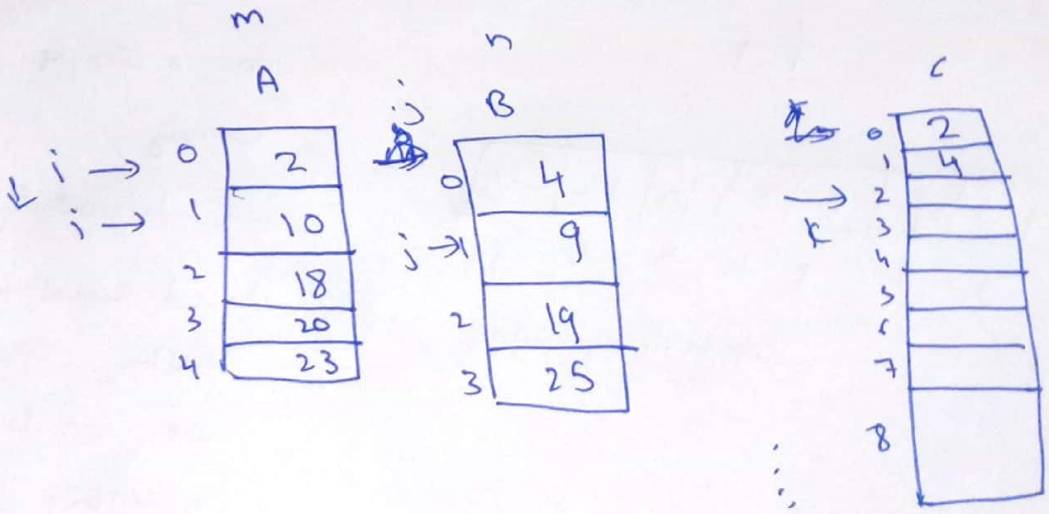
Worst case $O(n^2)$ already sorted

Best case $O(n \log n)$ Partitioning middle

Merging

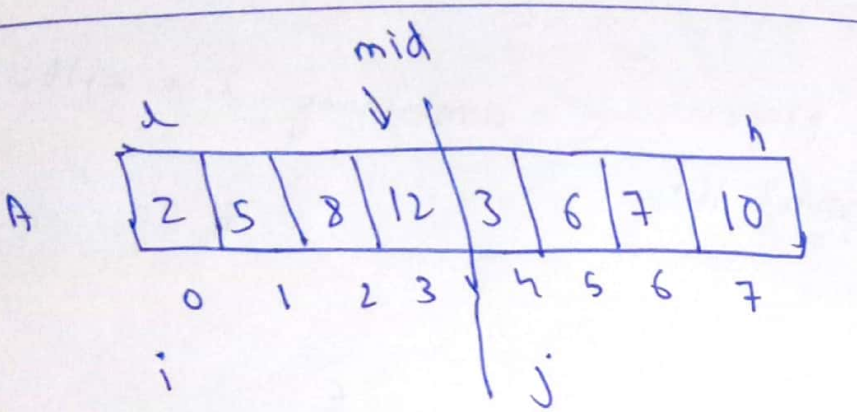
1. Merging 2 list
2. Merging 2 list in single array
3. Merging multiple list

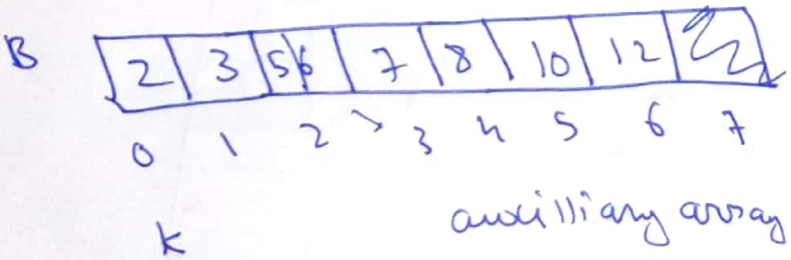
Merging is a process of combining two sorted lists into single sorted list.




```
void merge (int A[], int B[], int m, int n)
{  int i,j,k;
   i=j=k=0;
   while (i<m && j<n)
   { if(A[i]<B[j])
       C[k++] = A[i++];
     else
       C[k++] = B[j++];
```

```
for (; i<m; i++)
    C[k++] = A[i];
for (; j<n; j++)
    C[k++] = B[j];
}
```

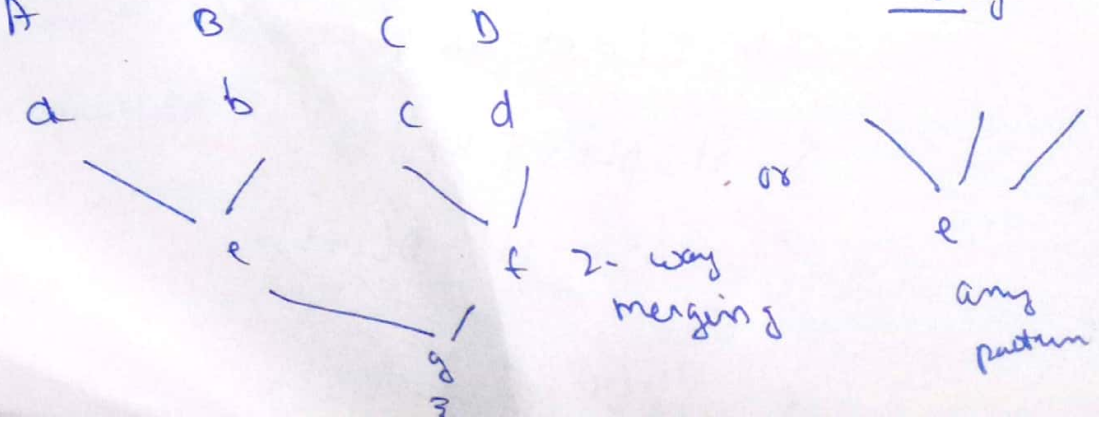




for merging in a single array, we definitely need another array then we can copy elements in earlier array

4- way Merging

m-way merging



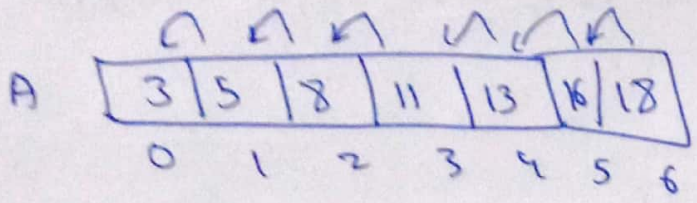
```
void merge (int A[], int l, int mid, int h)
{
    int i, j, k;
    int B[h+1];
    i = l; j = mid+1; k = l;
    while (i <= mid && j <= h)
    {
        if (A[i] < A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    }
    for (; i <= mid; i++)
        B[k++] = A[i];
    for (; j <= h; j++)
        B[k++] = A[j];
}

for (i=l; i<=h; i++)
    A[i] = B[i];

void MergeSort (int A[], int l, int h)
{
    if (l < h)
    {
        mid = ⌊(l+h)/2⌋;
        MergeSort (A, l, mid);
        MergeSort (A, mid+1, h);
        merge (A, l, mid, h);
    }
}
```

n log n

Shell Sort → (Insertion sort used in shell sort)



gap = $\lfloor \frac{n}{2} \rfloor$

Successive division

$n \log n$ each pass

Scan list

no. of pass

Normally gaps are taken from prime numbers

11 elements 7 5 2

$O(n^{3/2})$ $n^{1.5} < n^2$ $n^{5/3}$ $n^{1.66}$

$O(n \log n)$ $O(n^{1.5})$ $O(n^{1.66})$

adaptive

Doesn't take extra space.

Only merge sort in comparison based sorts takes

auxiliary array.

Heap Sort

$(O(n \log n))$

Heapify

$O(n)$

```
# include <stdio.h>
void Insert ( int H[], int n )
{   int i = n, temp;
```

```
temp = A[i];
while (i>1 && temp > A[i/2])
{   A[i] = A[i/2];
    i=i/2;
}
A[i] = temp;
}
void int Delete (int A[], int n);
int main ()
{   int H[] = {0,2,5, 8,9,4,10,73;
    for(i=2; i<=7; i++
    Insert (H, 7); print(Delete (H,7));
                   print (H).
    return 0;
}
int Delete (int A[], int n)
{ int i,j,x;   int temp, val;
    x = A[n] :          val = A[1];
    A[1] = A[n];
    i=1; j=i*2;
    while (j<n)
    { if(A[j+1] > A[j]) j+=1;
      if (A[i] < A[j]) { temp= A[i]; A[i]=A[j]
A[j]= temp; i=j; j=2*j; } return val;}
```

Index Based Sorting →

① Count Sort →

A

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

0	0	0	2	0	0	3	0	1	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

→

3	3	6	6	6	8	9	10	12	15

→

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

O(m+n) O(n)

```
Void CountSort (int A[], int n)
{ int max, i;
  int *C;
  max = findMax(A, n);
  C = new int [max+1];
  for (i=0; i<max+1; i++)
      C[i] = 0;
```

```
for (i=0; i<n; i++)
    c[A[i]]++;
i=0; j=0;
while (i< max+1)
{ if (c[i] > 0)
  { A[j++]=i;
    c[i]--;
  }
  else
      i++;
}
```

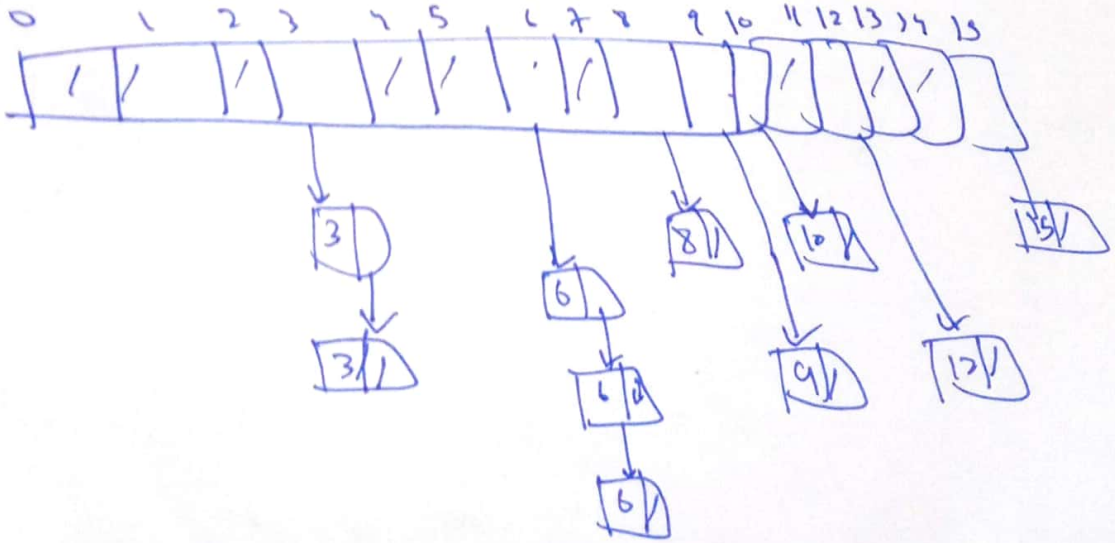
Bucket / Bin sort →

O(n)

A

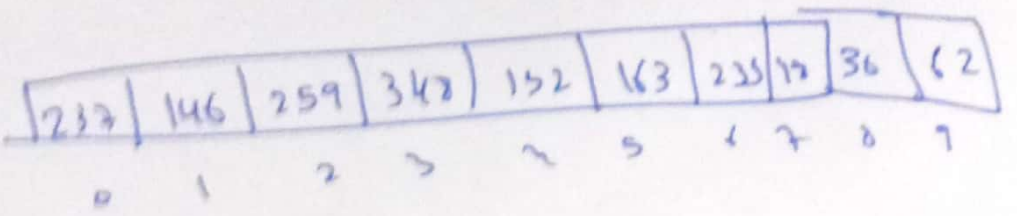
6	8	3	10	15	6	9	12	6	3
0	1	2	3	4	5	6	7	8	9

Bins

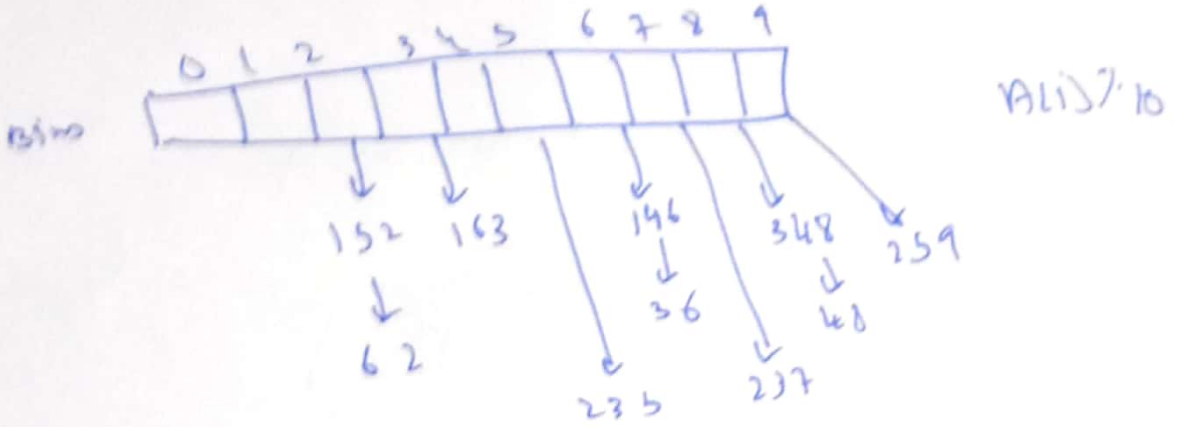


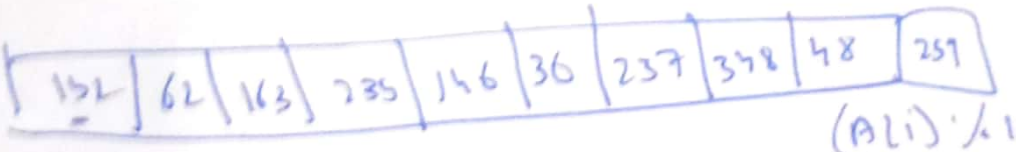
Radix Sort

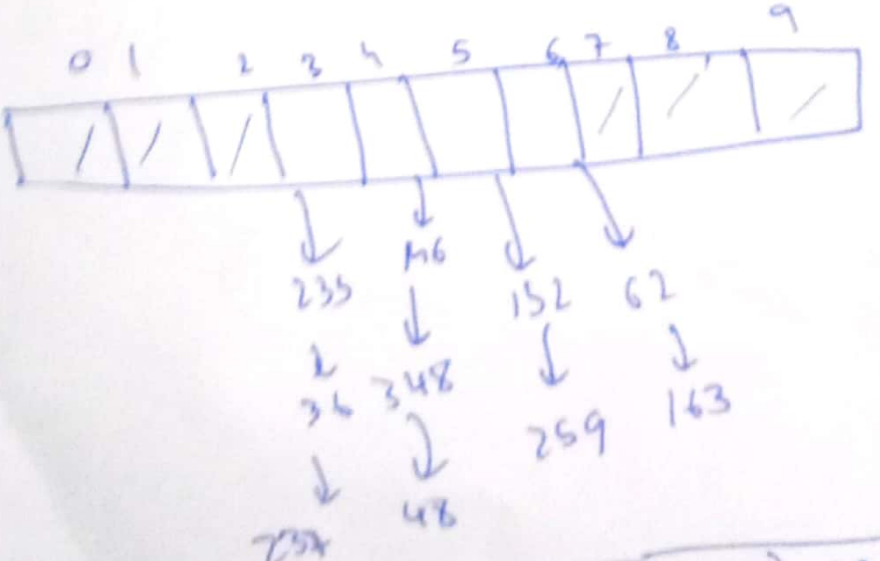
In quick sort, we were taking array of bins of length of largest element

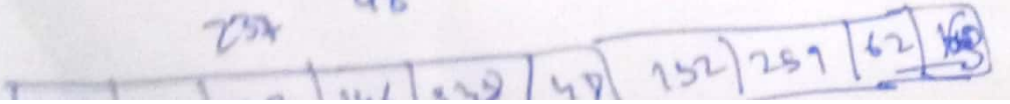


we won't take 348 bins, we'll take 10 bins



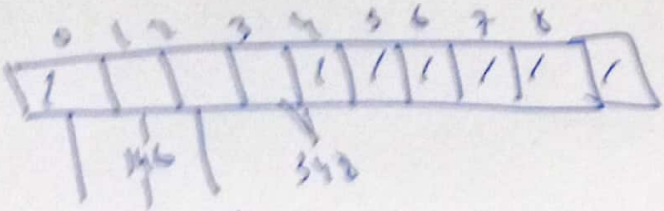






3rd pass

(A[i]/100)%10



36 152 235

152

48 237

62 1

259

d = constant

O(dn)

O(n)

36 48 62 146 152 168 235 237 259 368