Searching & Hashing

Searching :—
Searching is a process of finding an element within the list of elements (sorted in any order / randomly)
It is of 2 types —

① Linear search / Sequential search :—

Elements are searched one by one sequentially

```
Procedure SEQSEARCH (i, n, k)
//  i = variable
//  n = no of elements in list
//  k = we have to search it
```

- $k_0 \leftarrow k$
  $i \leftarrow n$

```
while ki ≠ K do
        i ← i-1
end
end SEQSEARCH
```

complexity :— O(n)

~~for i ← 1 to n~~
~~if ai = k~~

② Binary search

complexity :— O(log n)

Searching is done in ordered / sorted sequential file

```
Procedure BINSEARCH (i, n, k)
    l ← 1
    u ← n
    while (l ≤ u) do
        m ← [(l+u)/2]   // index of middle record
        case
            : k > km ; l ← m+1     // look in upper half
            : k = km ; i ← m ; return
            : k < km ; u ← m-1     // look in lower half
        end
    end
    i ← 0                          // no record is present with key k
end BINSEARCH
```

...sing of data items is possible in linear ... one by one sequentially ... not ...

Hashing :- Sorting

① Bubble :-

→ 11 15 2 13 6

swap

11 2 15 13 6

swap

11 2 13 15 6

swap

11 2 13 6 15

→ 2 11 13 6 15

2 11 6 13 15

→ 2 11 6 13 15

2 6 11 13 15

Procedure Bubble_sort

① $i \leftarrow 0$
② Repeat step 3 to 5 until $i < (n-1)$
③ $j \leftarrow 0$
④ Repeat step ⑤ until $j < (n-i-1)$
⑤ if $A[j] > A[j+1]$
  temp $\leftarrow A[j]$
  $A[j] \leftarrow A[j+1]$
  $A[j+1] \leftarrow$ temp
  end if
⑥ exit

Insertion sort

	16	15	2	13	6
I	15	16	2	13	6
II	2	15	16	13	6
III	2	13	15	16	6
IV	2	6	13	15	16

Insertion sort

```
for (i ← 1        to   (n-1))
        temp  ← A[i]
        j ← i-1
     while  ( temp < A[j] )
            A[j+1] ← A[j]
            j ← j-1
     end while
          A [j+1 ] ← temp
end
```

4 5

separate chaining
open Addressing
Linear Probing
Quadratic "
Double Hashing

0,5
5,6
7,4

10 series of list, Binary heap

0to7
① 0,5
② 2,5,6,1,6,9
③ 16
13

1,6
1,3

7
4 (Greatest is ignored while E)

Sorting

(1) Insertion sort

(77)	33	44	11	88	22	66	55
77	(33)	44	11	88	22	66	55
33	77	(44)	11	88	22	66	55
33	44	77	(11)	88	22	66	55
11	33	44	77	(88)	22	66	55
11	33	44	77	88	(22)	66	55
11	22	33	44	77	88	(66)	55
11	22	33	44	66	77	88	(55)
11	22	33	44	55	66	77	88

```
Procedure InsertionSort
    Set k ← 1
    for k ← 1  to (n-1)
        set temp ← a[k]
        set j ← k-1
        while temp < a[j] and (j >= 0)
            Set a[j+1] ← a[j]
        end while;
        Assign the value of temp to a[j+1]
    end for;
end;
```

```
k=1
for(k=1 ; k<=(n-1); k++)
    {
        temp = a[k];
        for (j = k-1; temp<a[j] && j>=0; j--)
            Insert temp at j
    }
```

2. Non-Linear DS - Tree, graph

...ing of data items is possible in linear fashion ...ced one by one sequentially ... is not

(2) Selection sort -

(77)	33	44	(11)	88	22	66	55
11	(33)	44	77	88	(22)	66	55
11	22	(44)	77	88	(33)	66	55
11	22	33	(77)	88	(44)	66	55
11	22	33	44	(88)	77	66	(55)
11	22	33	44	55	(77)	(66)	88
11	22	33	44	55	66	(77)	88
11	22	33	44	55	66	77	88

```
Procedure MIN (a, k, n, loc)        loc^n of smallest elmt
    set min ← a[k], loc ← k
    for j ← k+1 to n
        if min > a[j]
            set min ← a[j], loc ← j
        end if
    end for;
Return;
```

```
Procedure Selection_sort
    for k ← 1 to n-1
        call MIN(a, k, n, loc)
        // Now interchange a[k] and a[loc]
        temp ← a[k]
        a[k] ← a[loc]
        a[loc] ← temp
    end for;
end;
```

③ **Bubble sort -**

unsorted list

11, 15, 2, 13, 6

First Pass (i=0)

j=0 j=1 j=2 j=3

11 15 2 13 6

no swapping

11 2 15 13 6

11 2 13 15 6

11 2 13 6 15

Second Pass (i=1)

j=0 j=1 j=2

11 2 13 6 15

2 11 13 6 15

no swapping

2 11 6 13 15

no swapping (sorted)

Third Pass (i=2)

j=0 j=1

2 11 6 13 15

no swapping

2 6 11 13 15

no swapping (sorted)

Fourth Pass (i=3)

j=0

2 6 11 13 15

no swapping (sorted)

Algorithm bubble_sort

```
for i ← 0 to n-1
    for j ← 0 to (n-1-i)
        if A[j] > A[j+1]
        // swap A[j] with A[j+1]
            temp ← A[j]
            A[j] ← A[j+1]
            A[j+1] ← temp
        end if
    end for
end for
end;
```

i	n-1-i	times
0	n-1-0 = n-1	n-1
1	n-1-1 = n-2	n-2
2	n-1-2 = n-3	n-3
:	:	:
(n-1)-(n-1) = 0		0

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

2. Non-linear ... data items, is possible in linear fashion ... one by one sequentially ... not

(4) Merge sort

unsorted list – 85, 76, 46, 92, 30, 41, 12, 19, 93, 3, 50, 11

merge each pair of elements to obtain the list of sorted pairs

76	85		46	92		30	41		12	19		3	93		11	50

Merge each pair of pairs to obtain the lists of sorted elements

46	76	85	92		12	19	30	41		3	11	50	93

again merge the two subarrays to get two lists

12	19	30	41	46	76	85	92		3	11	50	93

merging the above two lists, we get

3	11	12	19	30	41	46	50	76	85	92	93

Procedure Merge sort

loop until last merge

merge each pair of elements by getting it ~~to obtain the list of sorted elements~~ subarrays

end loop;

end;

Heap sort

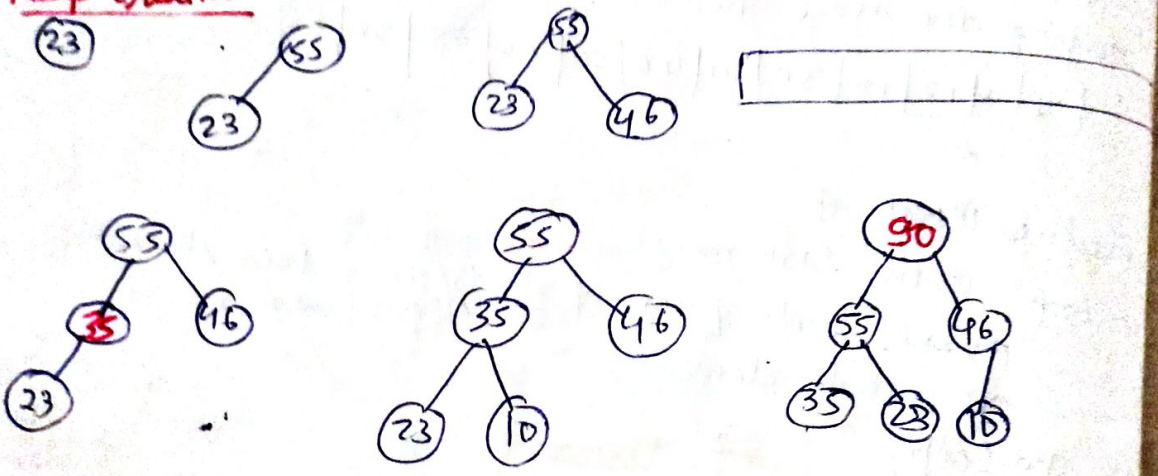
min heap (root – min) Method
Ascendy order list is obtained

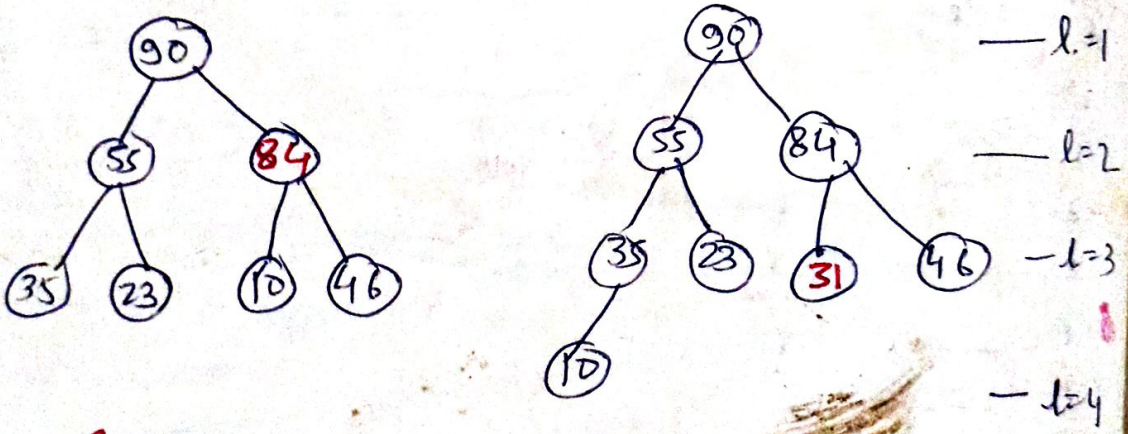
max heap (root – max) Method
Deccendy order list is obtained

Ex. max heap method

23, 55, 46, 35, 10, 90, 84, 31

Heap creation





Heap sort. (Delete max of level i)

90	84	55	46	35	31	23	10

Procedure Heap_sort–

(create heap (max for descendy / min for ascendy)
loop i ← 1 to l // l is the levels
~~Delete max of level i~~
loop j = 1 to last node at level i
Delete max of level i
end loop j;
end; end lp 2;
end DFS

Radix

348,
max digit

Pass I

I/P
341
14
36
42
5
1
3
5
3

Pass II

I/P
361
321
143
423
543
366
348
538
128

Pass III

I/P
321
423
128
538
143
543
348
361
366

Radix sort / Bucket sort

348, 143, 361, 423, 538, 128, 321, 543, 366

max digits = 3, so 3 passes (so that in each pass we get sorted list at a particular place value)

Pass I (for ones places)

I/P	0	1	2	3	4	5	6	7	8	9
348									348	
143		361		143						
361				423						
423									538	
538									128	
128		321								
321				543						
543							366			
366										

Pass II (for tens places)

I/P	0	1	2	3	4	5	6	7	8	9
361	348						361			
321	143		321							
143	361				143					
423	423		423							
543	538				543					
366	128						366			
348	321				348					
538	543			538						
128	366		128							

Pass III (for hundreds place)

I/P	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128				538				
538										
143		143				543				
543										
348				348						
361				361						
366				366						

Sorted list - 120, 143, 321, 348, 361, 366, 423, 538, 543

Procedure Radix_sort

Loop, for pass until max no of digits of the no in the list

In Pass I, unit digits are sorted, then collect the nos & reinput and reinput for next pass

In Pass II, tens digits are sorted, then collect the nos & reinput and reinput for next pass

In Pass III, Hundreds digits are sorted, then collect the nos & reinput and reinput for next pass

and so-on.

end loop;

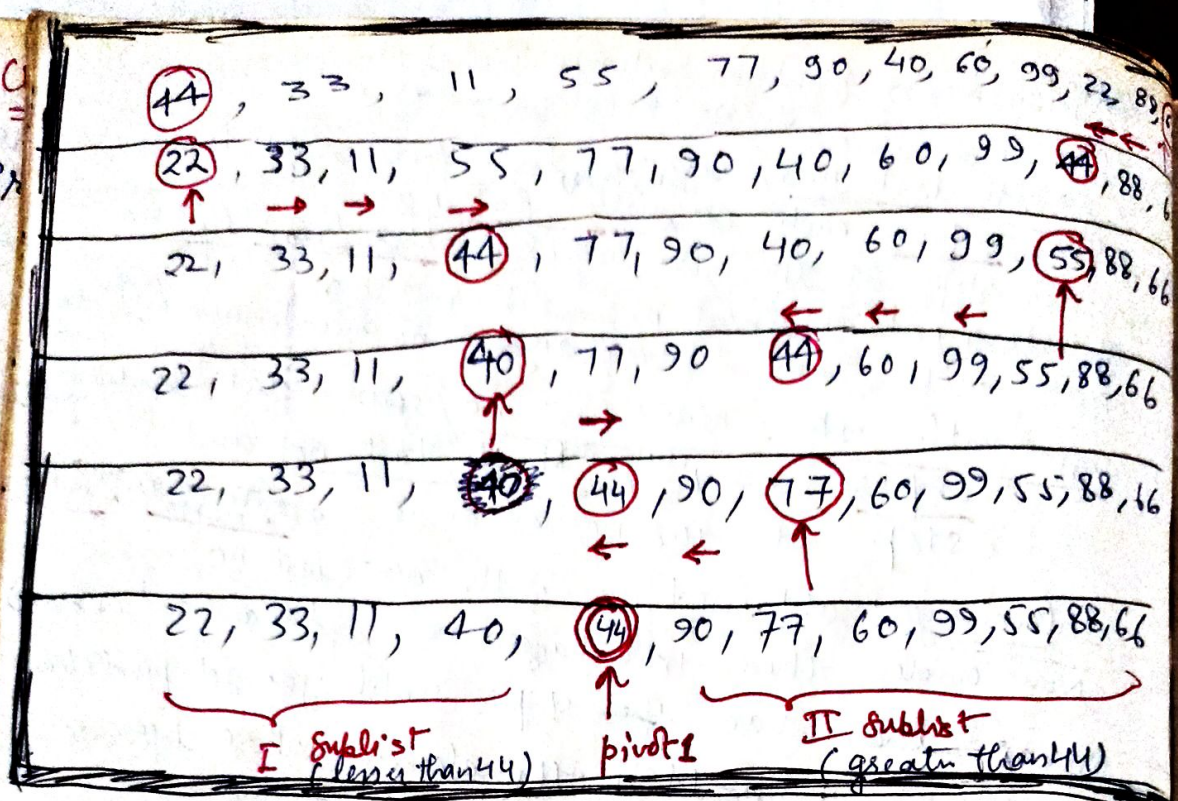
end;

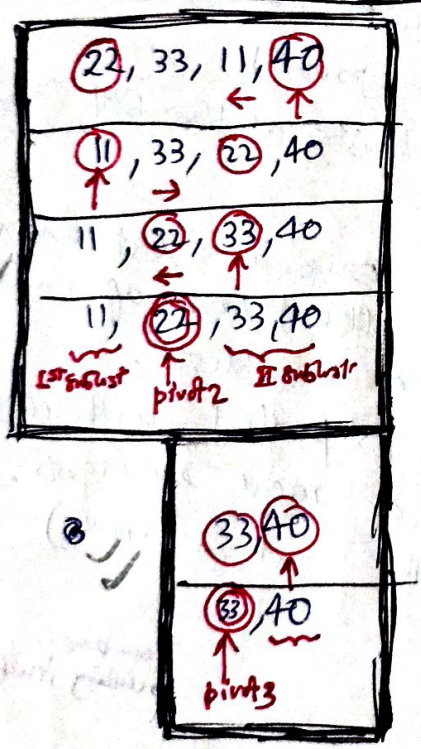
Quick sort (Partition Exchange Techn) (an appn of stacks)
(divide & conquer method) ↳ Recursion

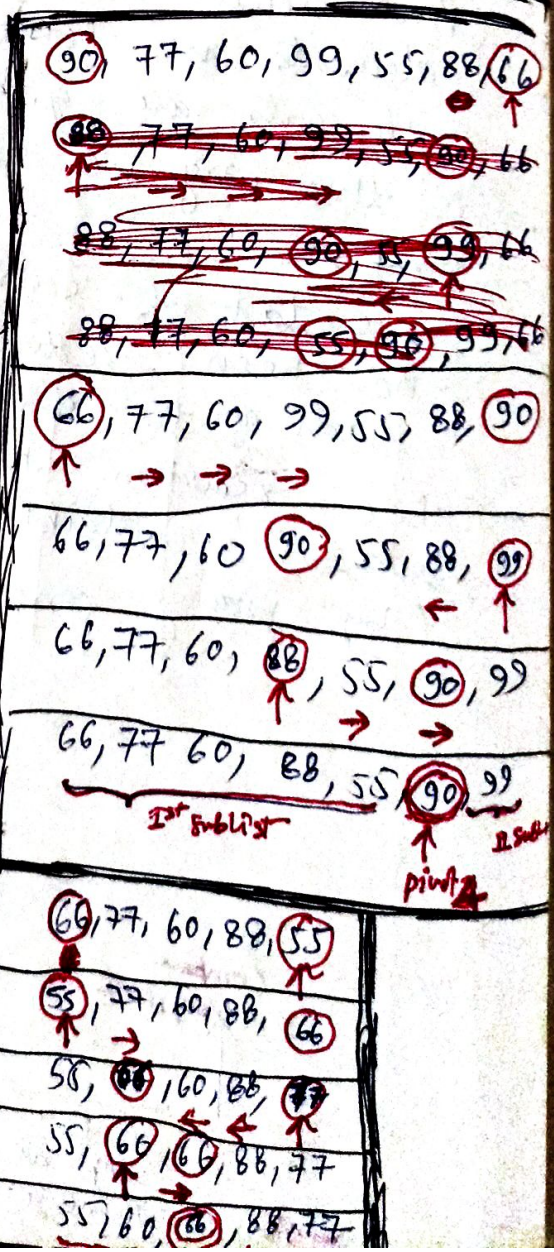
→ circle first and last no
→ Begin with last no, scan the (circling itself) list from right to left, compare each no with first no and stop at the no which is less than or equal to first no
Interchange the obtained no with first no
Now circle these interchanged nos and name as new 1st & last
→ Begin with this new first no, scan the list from left to right, compare each no (circling itself) with new last no and stop at the no which is greater than or equal to new last no
Interchange the obtained no. with new last no
Now circle these interchanged nos and name as new 1st & last
→ Repeat the above steps until you get pivot element
(Pivot element would be the element whose left-side elements are lesser than pivot and right-side elements are greater than pivot)
→ In this way we get 2 sublists at both the sides of pivot.
Thus task of sorting the original list has now been reduced to the task of sorting each of the abovementioned sublists.
→ Above reduction steps are repeated with each sublist containing 2 or more elements.
(by proving one sublist at a time)

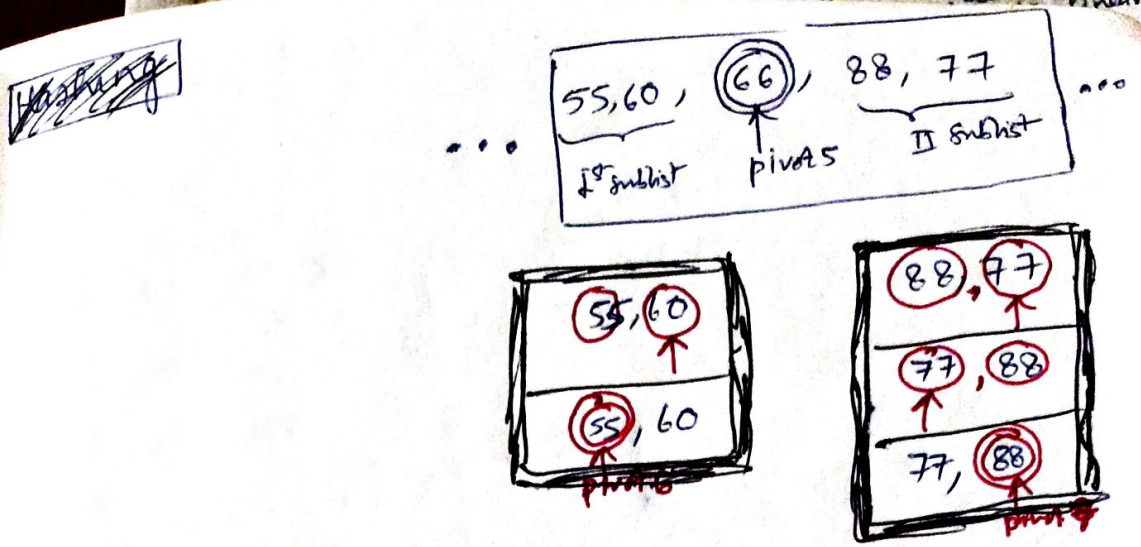
compare including itself
↑
compare with I st no
stop at no < I st no
Interchange & circle & rename I st & last no
obtained no with I st no

compare including itself
↑
compare with last no
stop at no > last no
Interchg & circle & rename I st & last no
obtained no with last no


44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66
22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66
22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66
22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66
22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66
I Sublist (less than 44)
pivot1
II Sublist (greater than 44)


22, 33, 11, 40
11, 33, 22, 40
11, 22, 33, 40
11, 22, 33, 40
pivot2
33, 40
33, 40
pivot3


90, 77, 60, 99, 55, 88, 66
66, 77, 60, 99, 55, 88, 90
66, 77, 60, 90, 55, 88, 99
66, 77, 60, 88, 55, 90, 99
I Sublist
pivot4
66, 77, 60, 88, 55
55, 77, 60, 88, 66



sorted list ⇒

11, 22, 33, 40, 44, 55, 60, 66, 77, 88, 90, 99

Ref: Schaum's Series
Page 6.14
Section 6.6

2. Non-linear DS ... items. is possible in linear fashion ... one sequentially ... is not

Hashing **Searching** — process of finding element in the list

Linear / Sequential search — $O(n)$

⇒ elements are searched one by one sequentially.

```
Procedure SEQ_SEARCH
        c ← 0
    loop i ← l to u
            if Item = k[i]
                c ← 1
                Print 'Item found at i position' & exit
            end if;
    end loop;
    if c = 0
        Print 'Item not found'
    end if;
end;
```

Binary search — $O(\log_2 n)$

⇒ elements are searched in a sorted/ordered list.

```
Procedure BIN_SEARCH
        c ← 0
    loop i ← l to u
            m ← (l+u)/2
            if (Item = k[m])
                c ← 1
                Print 'Item found at m position' & exit
            else if (Item > k[m])
                l ← m+1
            else
                u ← m-1
    end loop;
    if c = 0
        Print 'Item not found'
    end if;
end;
```

Hashing O(1)

Linear & Binary search are based on comparison of keys.

Hashing is a search tech in which there are no unnecessary comparisons of keys. Thus, in hashing no. of comparisons to find the desired record are minimized.

Hashing is an approach in which location of desired record is retrieved in single access. This location depends only on the given key (not on other keys)

Hash Table

It is a data structure where we store a key value after applying the hash fn.

Hash function

It is a fn that take key as i/p and transforms it into a hash index.

$H : K \rightarrow M$

H: Hash fn, K: Set of keys, M: set of memory addresses

Hash collision

Sometimes, Hash fn H may not yield distinct values. It is possible that two different keys K_1 and K_2 will yield the same hash address. This situation is called Hash collision

Types of Hash fn

- Division Remainder Method
- Mid Square method
- Folding method

We should consider following things while solving a hash fn.

1. H should be very easy and quick to compute
2. H should result in less no of collisions

1. Linear DS - Array, Linked list, Stack, Queue

2. Non-linear DS - Tree, graph ... linear fashion ... tially ... not

① Division Remainder Method

$h(k) = k \bmod m$

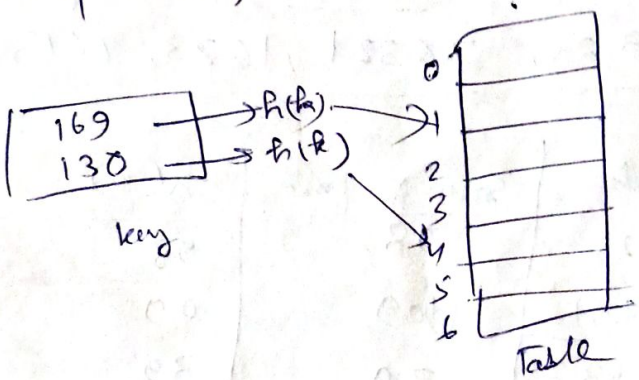
h → hash function, k → key, m → slots / position no (minimizes the no of collisions)

Ex. Consider a hash table with 7 slots i.e $m = 7$

Then hash fn $h(k) = k \bmod m$ will map the key 169 to slot 1 and key 130 will mapped to slot 4.

$h(169) = 169 \bmod 7 = 1$

Similarly $h(130) = 130 \bmod 7 = 4$



② Mid square Method

$h(k) = p$

↓

obtained by deleting digits from both sides of k^2

Ex. Consider a hash table with 50 slots i.e. $m = 50$ and key values $k = 1632, 1739, 3123$

k =	1632	1739	3123
k^2 =	2663424	3024121	9753129
$h(k)$ =	34	41	31

To properly implement this, the same ... of k^2 must be used for all the keys

③ Folding Method

key k is partitioned into a no. of parts $k_1, k_2 \cdots k_r$ where each part (except possibly the last) has the same no of digits as the required address. Then the parts are added together, ignoring the last carry

$$H(k) = k_1 + k_2 + \cdots + k_r$$

Eg. consider a hash table with 100 slots i.e. $m = 100$ and key values $k = 7325, 76321, 1623, 7613$

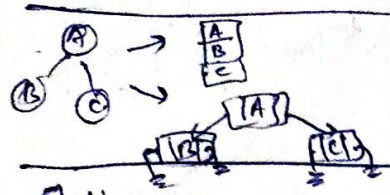
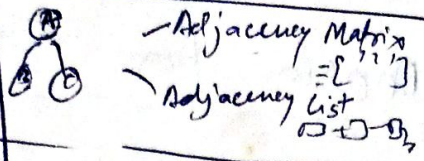
k	Parts	Sum of parts	h(k)
7325	73, 25	98	98
76321	76, 32, 1	109	09
1623	16, 23	39	39
7613	76, 13	89	89

complexity of sorting Algo's

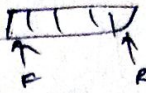
Algo	worst case	Avg case	Extra Memory
Bubble	$\frac{n(n+1)}{2} = O(n^2)$	$\frac{n(n+1)}{2} = O(n^2)$	
Quick	$\frac{n(n+3)}{2} = O(n^2)$	$1.4 n \log n = O(n \log n)$	
~~Heap~~			
Heap	$3n \log n = O(n \log n)$	$3n \log n = O(n \log n)$	
Insertion	$\frac{n(n+1)}{2} = O(n^2)$	$\frac{n(n+1)}{4} = O(n^2)$	
selection	$\frac{n(n+1)}{2}$ $O(n^2)$	$\frac{n(n+1)}{2} = O(n^2)$	
merge	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$
Radix (bucket)	$O(n^2)$	$O(n \log n)$	

Linear searching = $O(n)$

Binary searching = $O(\log n)$

Tree	Graph	Array
		
Static Dynamic	Static Dynamic	
Traversals: Inorder, Preorder, Postorder	Traversals: DFS, BFS	Insert Del^n
Heapsort etc.	colorization effect Airline Path Access Matrix (O.S.)	Tables Mathematics matrix
Thread B.T. Huffman Algo AVL B Tree / B+ Tree / BST	MST (Minimum Tree) Kruskal Algo	Sparse matrix (1, 2)
A Tree may be graph	but a graph is not necessarily a tree	

particular org^n of data is called DS.

Linked list	Stack (LIFO)	Queue (FIFO)	
			Representation
	static Dynamic	static Dynamic	Implementation
Insert^n Del^n	PUSH POP	Insert^n Del^n	Operations
Polynomial Representat^n & Implementation	① Conversion of Infix to Prefix & Postfix ② Evaluation of Postfix	CPU scheduling (OS) Railway Res^n Printer	Applns
	top increases on insert^n & decreases on delet^n	R increases on insert^n (no effect on F) F increases on del^n (no effect on R)	(Extra)
malloc() calloc() realloc() free() structure	PUSH POP	circular DE-Queue Priority Q	

greedy