12-04-2023.

UNIT - 3 : SEARCHING AND SORTING.

sorting

① * Insertion set Algorithm :-

InsertionSort(a, n):

step-1 :- Declare i, j, key:

step-2 :- for (i=1; i<n; i++)

ⓘ Set key=a[i]

ⓘⓘ for (j=i-1; j>=0 && a[j]>key; j=j-1)

ⓐ set a[j+1]=a[j]

ⓘⓘⓘ set a[j+1]=key

step-3 :- Exit.

② Selection sort Algorithm :-

Selection_sort (a, n)

step-1 :- Declare i, j, min index.

step-2 :- for (i=0 ; i<n ; i++)

ⓘ Set min index = i.

ⓘⓘ for (j=i+1 ; j<n ; j++)

ⓐ if (a[j] < a[min index])

Ⓐ set min index = j.

ⓘⓘⓘ swap (&a[i], & a [min index]);

step-3 :- Exit.

Scanned with CamScanner

③ **Quick sort Algorithm :-**

int Partition
~~Quick sort~~ (a, l, h) → Higher index

array, lower index

Step-1 :- Declare i, j, pivot ;

Step-2 :- Set i = l-1

Step-3 :- set pivot = a[h]

Step-4 :- for (j = l ; j <= h-1 ; j++)

(i) if (a[j] < pivot)

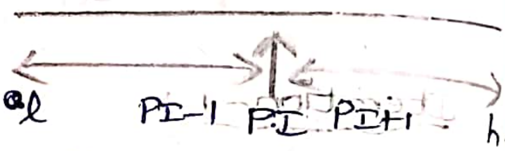
ⓐ i++

ⓑ Swap (&a[i], &a[j]);

Step-5 :- Swap (&a[i+1], &a[h]);

Step-6 :- return i+1 .

Quicksort (a, l, h) .



Step-1 :- Declare PI [Partition Index] ;

Step-2 :- if (l<h):

Step-(i) :- PI = Partition (a, l, h);

Step-(ii) :- Quicksort (a, l, PI-1) ;

Step-(iii) :- Quicksort (a, PI+1, h);

Step-3 :- EXIT.

④ Merge sort Algorithm :-

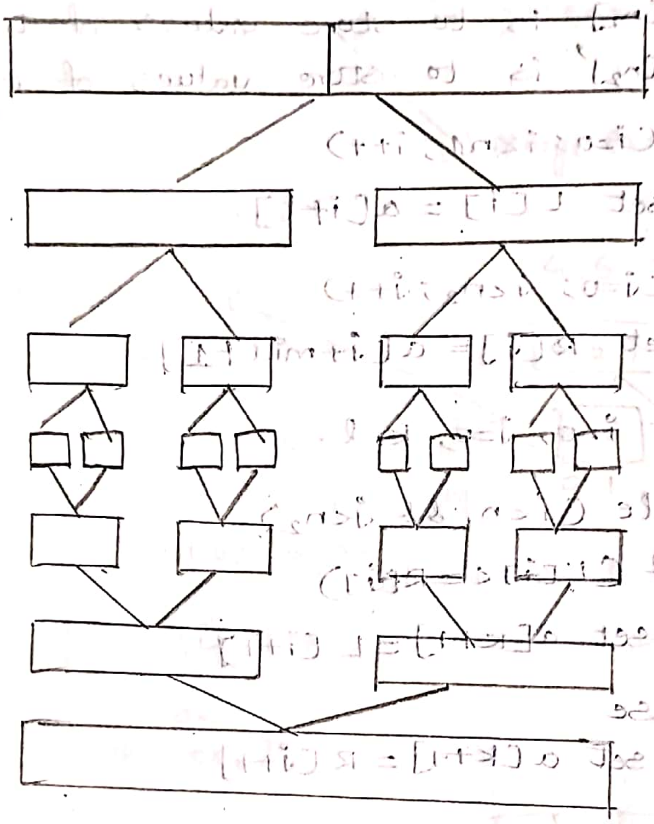
→ It comes under External sorting.

→ External sorting needs Additional space to store the bulk data.

→ The time complexity of Merge sort is n log N.

→ The time complexity of quick sort is order n^2. and the heart of quick sort is Partition.

→ The Heart of merge sort is merging. and Merge sort follows Divide and conquere rule as follows.



Algorithm :- Mergesort (a, l, h).

step-1 :- if (l<h)

① compute mid = (l+h)/2

(ii) mergesort (a, l, mid).

(iii) mergesort (a, mid+1, h)

(iv) merge ~~sort~~ (a, l, mid, h).

step-2 :- EXIT.

→ The key process in Mergsort is Merging.

Algorithm for merge :-

merge (a, l, mid, h).

Step-1 :- Declare i, j, k, n_1, n_2.

Here i, j, k are 'control variables'.

n_1 is 'size of Left half',

n_2 is 'size of Right half';

step-2 :- set n_1 = mid - l + 1. and n_2 = h - mid.

step-3 :- Declare L[n_1] , R[n_2].

'L[n_1]' is to store values of Left half

'R[n_2]' is to store values of Right half.

step-4 :- for (i=0; i<n1; i++)

(i) set L[i] = a[i+l].

step-5 :- for (j=0; j<n_2; j++)

(i) set R[j] = a[j+mid+1].

Step-6 :- set i=0, j=0, k=l.

step-7 :- while (i<n_1 && j<n_2)

(i) if (L[i] <= R[j])

(a) set a[k++] = L [i++] .

(ii) else

(a) set a[k++] = R [j++] .

Step-8 :- while (i<n_1)

(i) a[k++] = L[i++].

step-9 :- while (j<n_2)

(i) a[k++] = R [j++].

step-10 :- Display the elements of array

Step-11 :- EXIT.

⑤ Counting sort :-

4	1	2	1
0	1	2	3

Counting array :-
Count [max+1]

0	~~0~~ 2	~~0~~ 1	0	~~0~~ 1
0	1	2	3	4

for (i=0; i<n; i++)
count [a[i]]++.

cummulative frequency.

0	2	3	3	4
0	1	2	3	4

for(i=1; i<=max; i++)
count[i] + = count [i-1].

~~0~~	~~2~~ 1	~~3~~ 2	3	~~4~~ 3

output array :-
output [count [a[i]] -1] = a[i].

1	1	2	4
0	1	2	3

Algorithm :-

counting sort(a,n);

Step-1 :- Declare output[n], max, i;

Step-2 :- set Max = a[0].

step-3 :- for (i=0 ; i<n ; i++) // Finding max element.
① if (a[i] > max)
ⓐ set Max = a[i].

step-4 :- Declare count [max+1].

step-5 :- for(i=0; i<=max; i++)
ⓘ set count [i] = 0.

step-6 :- for (i=0; i<n; i++) // counting array
ⓘ set count [a [i]] ++

step-7 :- for (i=1; i<=max; i++) // cummulative frequency
ⓘ set count [i] + = count [i-1]

step-8 :- for (i=n-1; i>=0; i--) // mapping the elements.
ⓘ output [count [a[i]] -1] = a[i]
ⓘⓘ count [a[i]] --;

step-9 :- for (i=0; i<n; i++)
ⓘ Display "output[i]"

Step-10 :- EXIT.

Example :-

4	1	3	1	4	3

Max element = 4

count [max+1] = 5.

0	1	2	3	4
0	~~0~~ 2	0	~~0~~ 2	~~0~~ 2

cummulative frequency.

0	1	2	3	4
0	~~1~~ ~~2~~ 0	2	~~3~~ ~~4~~ 2	~~5~~ ~~6~~ 4

output array :-

0	1	2	3	4	5
1	1	3	3	4	4

6 Radix sorting :-

		One's	Ten's	Hundred's
437	437	123	123	123
689	689	483	437	145
123	123	645	645	437
483	483	145	145	483
645	645	437	483	645
145	145	689	689	689.

Algorithm :-

~~step-1 :- for (place = 1;~~ Radixsort (a, n) → Module.

step-1 :- set max = getmax (a, n)

step-2 :- for (place = 1 ; max/place > 0 ; place = place × 10)

(i) countingsort (a, n, place);

→ change a[i] as (a[i] / place) % 10.

Step-3 :- EXIT.

→ The limitation of counting sort is overcomed by Radix sorting [Sorting for large numbers].

Note :-

→ In step-~~8~~9 of counting sort, Don't display the elements & store them [a[i] = output [i]] in case of radix sort.

7 **Shell sorting Algorithm:-**

shellsort (a, n).

step-1 :- Declare interval, i, j, temp.

step-2 :- for (interval $= \frac{n}{2}$; interval > 0; interval /= 2)

(i) for (i = interval; i < n; i++)

(a) temp = a[i]

(b) for (j = i; j >= interval && a[j-interval] > temp; j = j-interval);

(A) a[j] = a[j-interval].

(c) a[j] = temp;

step-3 :- EXIT.

* Hashing :-

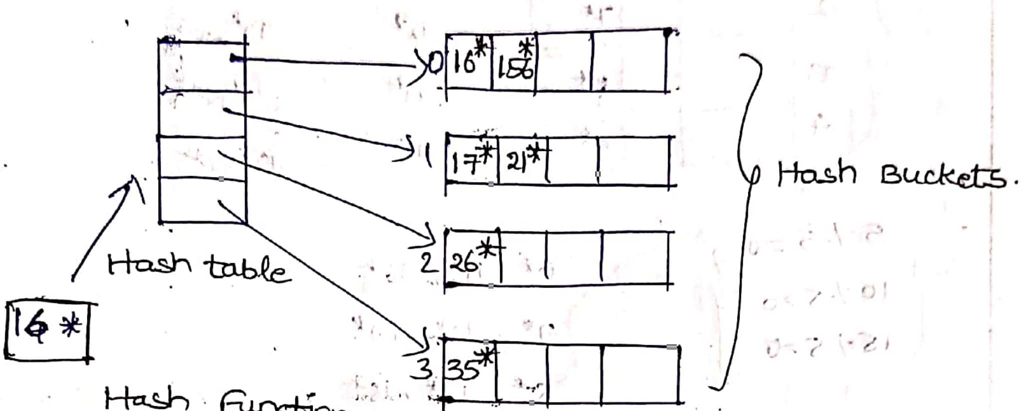
→ Hashing is used to store, and retrieve the data quickly.

→ It has Best time complexity.

→ Before Hashing, we have Direct Addressing [by using arrays] concept.

→ In Direct Addressing key value is Index value.

Hash Table & Hash Function.



Hash Function.

$\boxed{h(k) = k \% M}$

k is key value.

M is Hash values.

16* ; $h(k) = 16 \% 4 = 0$

156* ; $h(k) = 156 \% 4 = 0$

17* ; $h(k) = 17 \% 4 = 1$

→ To overcome collisions, we have two Hashings.

① open Hashing.

② closed Hashing.

① open Hashing:-

→ Under open Hashing, we have a concept 'separate chaining'.

→ By using single linked lists we separates the chain.

② closed Hashing:-

→ Also known as open Addressing.

→ We have 3 Technis under this.

(i) Linear probing

(ii) Quadratic probing.

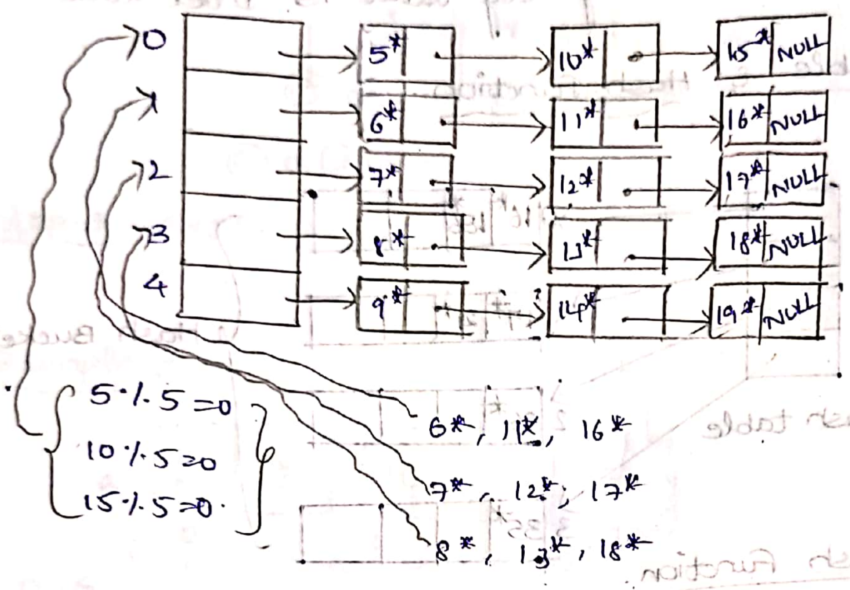
(iii) Double Hashing.

④ Open Hashing :-

→ Set of Records are going to store at same entry, then collision occurs.

→ To resolve collisions, 'Open Hashing' is one of the solution.

Example :-



→ We use single Linked List concept to overcome collisions in open Hashing.

structure :-

```
SIZE 5.

struct node
{
    int data;
    struct node *next;
}* temp, *p;

struct node *a[SIZE];

void Initialize ( ); //Initialize array elements.
```

Algorithm

step-1 :- Declare i

step-2 :- for (i=0; i<SIZE; i++
    (i) set a[i]=NULL

step-3 :- EXIT.

Scanned with CamScanner

* Algorithm for Insertion:-

void insert (int value).

step-1 :- Declare hkey

step-2 :- create a new node i.e temp

step-3 :- set temp→data = value, temp→next = NULL;

step-4 :- compute hkey = value % SIZE.

step-5 :- if (a [hkey] == NULL)
   (i) set a[hkey] = temp.

step-6 :- else
   (i) set P = a[hkey]
   (ii) while (P→next != NULL)
      (A) set P = P→next
   (iii) set P→next = temp

step-7 :- EXIT.

* Algorithm for Deletion:-

void del (int value)

step-1 :- Declare hkey

step-2 :- compute hkey = value % SIZE

step-3 :- Set P = a[hkey].

step-4 :- if (P→next == NULL)
   (i) if (P→data = value)
      (a) Set temp = p.
      (b) P = NULL
      (c) Free (temp)

step-5 :- else
   (i) if (P→data == value)
      (a) set temp = p
      (b) set a[hkey] = p→next, temp→next = NULL
      (c) Free (temp)
      (d) EXIT
   (ii) while (P→next != NULL)
      (a) if (P→next→data == value)
         (A) set temp = p→next
         (B) set p→next = temp→next
         (C) set temp→next = NULL
         (D) Free (temp)

step-6 :- EXIT.

* Algorithm for searching:-

void search (int value)

Step-1:- Declare hkey.

step-2:- hkey=value % SIZE.

step-3:- set P=a[hkey]

step-4:- while (P!=NULL)

(i) if (P->data==value)

(a) return 1

(ii) set P=P->next.

step-5:- return 0.

step-6 :- EXIT.

* closed Hashing [open Addressing] :-

→ There are three types of technics in closed Hashing.

① Linear probing.

② Quadratic probing.

③ Double Hashing.

① Linear probing :-

* Algorithm for Linear probing Insertion:-

#define SIZE 5

int a[SIZE];

void initialize ( );

step-1:- for (i=0; i<SIZE; i++) // Initialization.

(i) set a[i]=-1

Algorithm for Insertion:-

insert (value); // called function.

void insert (int value); // calling function.

Step-1:- Declare i, hkey;

step-2:- compute hkey=value % SIZE.

step-3 :- for (i=0; i<SIZE; i++)

① if (a[(hkey+i)%SIZE] == -1)

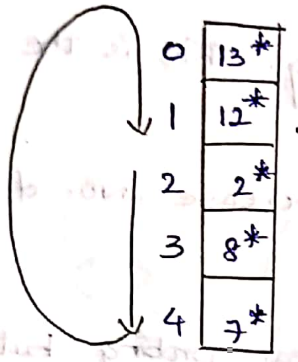
ⓐ set a[(hkey+i)%SIZE] = value.

ⓑ break;

step-4 :- if (i==SIZE)

① Display " NO Free slot ".

step-5 :- EXIT.



2*. a[2] is free

8* a[2] is not free, so a[3]

7* a[2], a[3] are not free, so a[4]

13* a[2], a[3], a[4] are not free, so a[0]

12* a[2], a[3], a[4], a[0] are not free.

*Algorithm for deletion :- Deletion :-

step-1 :- Declare. i, hkey.

step-2 :- Compute hkey = value % SIZE

step-3 :- for (i=0; i<SIZE; i++)

① if a[(hkey+i)%SIZE] == value)

ⓐ Set a[(hkey+i)%SIZE] = -1

ⓑ break.

step-4 :- if (i==SIZE)

① Display " Element to be deleted not found."

(or)

No Input Record found.

step-5 :- EXIT

*Algorithm for searching :-

step-1 :- Declare i, hkey, flag.

step-2 :- compute hkey = value % SIZE.

step-3 :- for (i=0; i<SIZE; i++)

① if a[(hkey+i)%SIZE] == value)

ⓐ set flag = 1

ⓑ break.

step-4 :- if (flag == 1)

① Display " Element found

step-5 :- EXIT.

(2) <u>Quadratic probing</u> :-

0	
1	7*
2	2*
3	8*
4	13*

2*; a[2] is free

8*; a[3] is free

7*; a[2] is not free; a[1] is free.

13*; a[4] is free.

12*;

→ Here one free slot is skipping. This is the Drawback of Quadratic probing.

→ To overcome this we need to increase No. of iterations.

→ The Insertion is same as Linear probing but replace 'i' by 'i×i'.

→ But Deletion & searching same as Linear probing no need to replace anything.

(3) <u>Double Hashing</u> :-

* <u>Algorithm for Insertion</u> :-

Insertion(value);

<u>step-1</u> :- Declare i, $h_1$key, $h_2$key.

<u>step-2</u> :- compute $h_1key = value \% SIZE$

<u>step-3</u> :- compute $h_2key = PRIME - value \% PRIME$.

<u>step-4</u> :- for (i=0; i<n; i++)

(i) if (a[(($h_1$key + i) × ($h_2$key)) % SIZE] == -1)

(a) set a[($h_1$key + i × $h_2$key) % SIZE] = value.

(b) break;

<u>step-5</u> :- EXIT.

* <u>Note</u> :-

→ To get Distinct values take the size of Hashtable as prime numbers.

→ <u>Hashing Applications</u> :- DBMS, Password storage, Data compression, cryptography, Image processing.