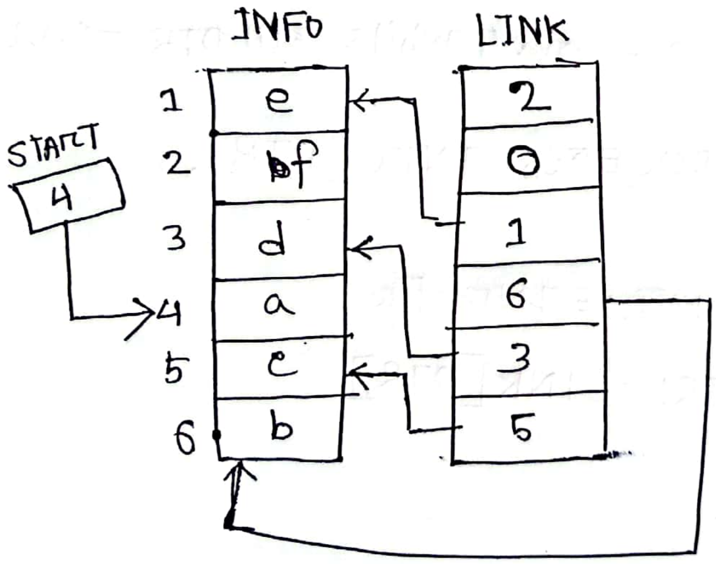
Linked List: A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. That is, each node is divided into two parts: the first part contains the ~~element~~ information of the element and the second part, called the link field, contains the address of the next node in the list.



INFO [4] = a   LINK[4] = 6

INFO [6] = b   LINK[6] = 5

INFO [5] = c   LINK [5] = 3

INFO [3] = d   LINK [3] = 1

INFO [1] = e   LINK[1] = 2

INFO[2] = f   LINK[2] = NULL

Traversing a Linked list:->

Algorithm:- Let LIST be linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

count (START, INFO, LINK, PTR)

Step-1: Set PTR := START.

Step-2: Repeat step 3 and 4 while PTR ≠ NULL.

Step-3: Apply PROCESS INFO[PTR].

Step-4: Set ~~PTR := INFO[5]~~

PTR := LINK[PTR]

Step-5: Exit.

Print Information at each node of the list.

Algorithm:-

PRINT(INFO, LINK, START)

Step-1:- Set PTR := START.

Step-2:- Repeat step 3 and 4 while PTR $\neq$ NULL.

Step-3:- PRINT: INFO [PTR].

Step-4:- Set PTR := LINK [PTR].

Step-5:- Exit.

Finds the number NUM of elements in a linked list.

Algorithm:

COUNT(START, INFU, LINK, NUM).

Step-1:- Set NUM := 0

Step-2:- set PTR := START

Step-3:- Repeat step-4 and 5 while PTR ≠ NULL.

Step-4:- set NUM := NUM + 1.

Step-5:- set PTR := LINK[PTR]

Step-6:- Exit.

Array: List of elements of the same data type placed in a contiguous memory location.

Traversing Linear Arrays:-

	0	1	2	3	4
LA	10	11	12	13	14
	LB				UB

Algorithm:- Here LA is linear array with lower bound(LB) and upper bound (UB). This algorithm traverse LA applying an operation PROCESS to each element of LA.

Step-1:- Set $K := LB$.

Step-2: Repeat step-3 and 4 while $K \le UB$.

Step-3: Apply PROCESS to LA[K]

Step-4:- Set $K := K+1$.

Step-5: Exit.

CamScanner

Inserting into a Linear Array:

Algorithm: INSERT (LA, N, K, ITEM), LB)

Here, LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm insert an element ITEM into the K-th position.

Step-1: Set $j := N$. / $j := LB$

Step-2: Repeat step-3 and 4 while $j \geq K$.

Step-3: Set $LA[j+1] = LA[j]$

Step-4: Set $j := j-1$.

Step-5: Set $LA[K] := ITEM$

Step-6: Set $N := N+1$

Step-7: Exit.

Deleting from a linear array:-

Algorithm:- DELETE (LA, N, K, ITEM

Here, LA is linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm delets the K-th elements from LA.

Step-1: Set ITEM := LA[K].

Step-2: Repeat for j=k to N-1:

Set LA[j] := LA[j+1]

Step-3: Set N := N-1

Step-4: Exit.

LA=

0	1	2	3	4	5	6	7
10	20	35	40	45	50	60	

K=4 ITEM

LA[4] = 45

for K=4 to 6

j=4 LA[4] = LA[5]

Bubble sort:-

Algorithm: BUBBLE(DATA,N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

Step-1: Repeat step 2 and 3 for K=1 to N-1.

Step-2:- set PTR:=1

Step-3:- Repeat while PTR ≤ N-K

a) If DATA[PTR] > DATA[PTR+1]

DATA[PTR] and DATA[PTR+1] interchange

b) set PTR:= PTR+1.

Step-4:- Exit.

Linear search:-

Algorithm: LINEAR(DATA, N, ITEM, LOC)

Here, DATA is a linear array with N elements and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA or set LOC:=0 if the search is unsuccessfull.

Step-1: Set DATA[N+1]:=ITEM.

Step-2:- Set LOC:=1.

Step-3:- Repeat while DATA[LOC] ≠ ITEM.

Set LOC:=LOC+1.

Step-4:- If LOC:=N+1

then: Set LOC:=0

Step-5: Exit.

Binary Search:

Algorithm:- BINARY(DATA, LB, UB, ITEM, LOC)

Here, DATA is sorted array with lower bound (LB) and upper bound (UB), and ITEM is a given item of information. The variable. The variable START, END and MID denote, respectively, the starting, ending and middle location of a segment of elements of DATA.

Step-1:- Set START := LB, END := UB and MID = INT((START+END)/2)

Step-2:- Repeat step-3 and 4 while START $\leq$ END and DATA[MID] $\neq$ ITEM.

Step-3: If ITEM < DATA[MID]

Set END := MID-1.

Else :

set START := MID+1.

Step-4:- Set MID := INT((START+END)/2).

Step-5:- If DATA[MID] = ITEM. then

set LOC := MID.

Else, set LOC := NULL.

Step-6:- Exit.

First Pattern Matching Algorithm:

Here, P and T are strings with lengths R and S, respectively and are sorted as arrays with one characters per element. This algorithm find the INDEX of P in T.

T = abcde Length(T) = S

P = cd Length(P) = R

$\therefore MAX = S - R + 1$

$= 5 - 2 + 1$

$= 4$

Step-1: set $K := 1$ and $MAX := S - R + 1$.

Step-2: Repeat step-3 to 5 while $K \leq MAX$.

Step-3: Repeat for $L = 1$ to R: [Test each character of P]

if $P[L] \neq T[K+L-1]$ then go to step-5

Step-4: [Success] set $INDEX := K$ and Exit.

Step-5: set $K := K+1$.

Step-6: [Failure] Set $INDEX := 0$

Step-7: Exit.

— o —

$K = 1$ $MAX = 4$

$K \leq MAX$ ✓

$P[1] \neq T[1]$ $P[2] =$

c

1 a

2 – b

3 – c

4 – d

INDEX = 3

Word / String Processing:→

i) <u>Insertion</u>: Inserting a string in the middle of the text.

INSERT (text, position, string)

EX→ INSERT ("ABCDEFG", 3, "XYZ")

= "AB" || "XYZ" || "CDEFG"

= "ABXYZCDEFG"

INSERT (T, K, S) = Substring (T, 1, K-1) || S || Substring (T, K, Length(T) - K+1)

This is, the initial substring of T before the position K, which has length K-1, is concatenated with the string S, and the result is concatenated with the remaining part of T, which begins in position K and has length = Length(T) - K+1.

ii) Deletion: Deleting a string from the text.

DELETE(text, position, length)

EX→

DELETE("ABC~~DE~~FGHI", 4, 2)

= "ABC" || "FGHI"

DELETE(T, K, L) = Substring(T, 1, K-1) ||

Substring(T, K+L, Length(T) - (K+L) + 1)

That is, the initial substring of T before position K is concatenated with the terminal substring of T begining in position K+L. The length of the initial substring is K-1 and the length of the terminal substring is:

(Length(T) - (K+L) + 1)

Algorithm:-

Step-1: Set K := INDEX(T, P)

Step-2: Repeat while K ≠ 0

a) [Delete P from T]

Set T := DELETE(T, INDEX(T, P), LENGTH(P))

b) [update index].

Set K := INDEX(T, P).

Step-3: Write: T

Step-4: Exit.

iii) **Replacement:** Replacing one string in the text by another.

REPLACE(text, pattern1, pattern2)

Ex→ REPLACE("ABCDEFGH", "CD", "WXYZ")

= "AB" || "WXYZ" || "EFGH"

= "ABWXYZEFGH"

REPLACE(T, P_1, P_2)

K := INDEX(T, P_1)

T := DELETE(T, K, Length(P_1))

INSERT(T, K, P_2)

The first two steps delete P_1 from T, and the third step insert P_2 in the position K from which P_1 was deleted.

Step-1: Set K := INDEX(T, P_1)

Step-2: Repeat while $K \neq 0$

a) Set T := REPLACE(T, P_1, P_2)

b) Set K := INDEX(T, P_1)

Step-3: Write: T

Step-4: Exit.

String Operations:-

i) Length:-> The number of characters in a string is called its length. we will write,

LENGTH(string)

LENGTH('COMPUTER') = 8

ii) Substring:-> Accessing a substring from a given string requires three pieces of information i) The string itself, ii) the position of the first character of the substring in the given string and iii) the length of the substring. we can write,

SUBSTRING (string, initial, length)

EX-> string = "ABCDEFGH"

SUBSTRING (string, 4,3)

= "DEF"

iii) Concatenation: Let, S_1 and S_2 be strings. The concatenation of S_1 and S_2, which we denote by $S_1 || S_2$, is the string consisting of characters S_1 followed by the characters of S_2.

EX-> S_1 = "MARK" S_2 = "TWIN"

$\therefore$ $S_1 || S_2$ = "MARKTWIN"

iv) **Indexing:-** Indexing also called pattern matching, refers to finding the position where a string pattern P first appears in a given text T. We call this operation INDEX and write,

$$\text{INDEX}(\text{text}, \text{pattern})$$

EX→ INDEX ("calcutta", "cutta") = 4

Time Complexity:→ Amount of time taken up by an algorithm or code as function of input size.

Linear search Analysis:-

Worst Case:→ The worst case occurs when ITEM is the last element in the array DATA or is not there all. In either situation, we have

1	-8	2	3	5

$$C(n) = n$$

Accordingly, $C(n)=n$ is the worst-case-complexity of the Linear search algorithm.

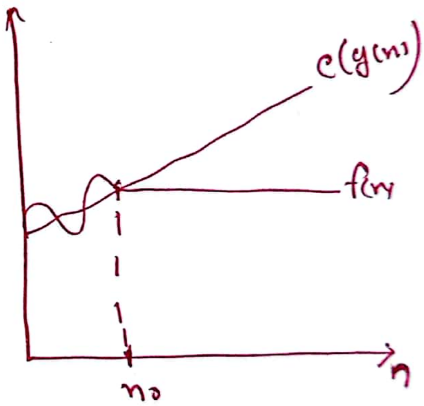
Average Case:→

Here we assume that ITEM does appear in DATA, and that is equally likely to occur at any position in the array. Accordingly, the number of comparison C can be any of the numbers $1, 2, 3, \ldots, n$ and each number occurs with possiblity $p = \frac{1}{n}$

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

This agrees with our intutive feeling that the average number of comparison need to find the location of ITEM is approximately equal to half the numbers of elements in the DATA List.

Big O Notation:-



$O(g(n)) = \{ f(n)$: There exist positive constant c and n_0 such that
$$0 \le f(n) \le c\,g(n) \text{ for all } n \ge n_0 \}$$

O-notation gives an upper bound for a function to within a constant factor. we write, $f(n) = O(g(n))$ if there are positive constant n_0 and c such that to the right of n_0 the value of $f(n)$ always lies on or below.