DATA STRUCTURE

3RD SEMESTER

LECTURE NOTES

Prepared By:- MONALISA SWAIN

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEEERING

SWAMI VIVEKANANDA SCHOOL OF ENGINEERING & TECHNOLOGY

MADANPUR, BHUBANESWAR, PIN-752054

Data structure Chapter-1

Introduction

Data :- Data can be defind as raw facts or figures or numbers

- Information :- Information can be defind as processed data
- Data structure :- The Mathematical logical Model used for a particular organisation or data is known as data structure.

OR

Organised collection of data and the operations that can be applied on that data is known as data structure.

- Data type :- (i) Data type can be defind as type of the data or nature of the data.

(ii) The various data types that are commonly use are - (1) Integer
(2) Character
(3) Float
(4) double.

Data structure operation :- The various types of operations that can be applied on any data structure are :- (1) Searching
(2) Traversing
(3) Insertion
(4) Deletion
(5) Sorting
(6) Merg

Data type	keyword used	size in bytes	Range	use
character	char	1	-128 to 127	To store characters
Integer	int	2	-32768 to 32767	To store integer number
Floating point	float	4	3.4E-38 to 3.4E+38	To store floating point numbers
Double	double	8	1.7E-308 to 1.7E+308	To store big floating point numbers
Valueless	void	0	valueless	—

30.07.19

1) Searching :- Searching is an operation that deals with finding the location of the record with a given key value or finding the location of all the records with satisfies one or more condition.

2. Traversing :- Traversing is an operation that deals with accessing a record or items exactly once so that sharfun items in the records may be processed.

3. Insertion :- Insertion is an operation that deals with Adding a new record or item to the existing data structure.

(4) Deletion :- Deletion is an operation that deals with removing or deleting a record from an existing data structure.

(5) ~~sort~~ sorting :- Sorting is an operation that deals with arranging or sorting the elements in a particular order (either ~~asce~~ ascending or descending order).

(6) Meanging :- Meanging is an operation that deals with combining two data sets to get a new data set.

Imp * Abstract Data Type (ADT)

(i) To manage the complexity a program and problem solving process, computer science uses abstraction to allow them to focus on the big picture without getting bogged in the deatails

(ii) An 'ADT' is a logical description or how to view the data and the operations that are carried on them.

(iii) The implementation of and abstract data type ~~upon~~ of term refer to as the data structure provides us with a physical view the data.

(iv) Ex- stack, queue, graph, list, map, etc.

Time-space Tradeoff

(i) In computer science, Time-space Tradeoff is a way of solving a problem or calculation in less time by using more memory space or by solving a problem in a very little space by spending a longtime.

(ii) So if our problem is taking a longtime But not much memory, a space time ~~sparad~~ Tradeoff good let us used more memory and solve the problem more quickly or if it could be solf very quickly But requires more memory then we can try to spend more time solving the problem in the limited Memory.

(iii) Some times we may also s· a tradeoff between space & Time complexcity.

For ex: we may have to choose a data structure that requires a lot of storage in order reduce the computation time. Therefore the programmer must make an ~~con~~ important decision from an informy point of view.

(iv) The time-space Tradeoff refers to a choice between algorithmic solution of

a data processing problem that allows decrease the running of an algorithmic solution by increasing the space to store the data and by.

(v) An algorithm 'A' for a problem capital 'P' if save to have time complexcity of T(n), If the number of steps required to complete and execusion for an input size small an is always <= T(n)

(vi) An algorithm 'A' for a problem 'p' is said to have space complexcity if the number of bits required to complete is execusion for an input size small an is always <= S(n).

Assignment-1

(1) what are the various types of operations that can be performon d s ?

(2) Explain ?

(3) Explain time-space Tradeoff ?

Chapter-2 02.08.19

STRING PROCESSING

(i) Each programing language consists of a character set, that is used to conunicate with the computer. This set usevally includes the following.

A, B, C, ... up to Z.

a, b, C, ... up to z

0, 1, 2, 3, ... up to 9

+, -, *, /, ., ", ', <, >, &, %.

(ii) A finite sequence 'S' of one or more characters is called as a string.

(iii) The number of characters in a string is called as its length.

(iv) The string with zero character is called 03.08.19 as Empty string or Null string.

(v) specific strings will be denoted by enclosing there characters in single (or) quotation marks.

(vi) quotation marks also solve as string delimiters.

* Storing string

(1) In fixed length storage each line of print is viewed as a record where all the records have the same length or each record accomodates the same number of characters.

* Algorithm & its complexity

Algorithm is a weed defind list of steps for solving a particular problem. one major propose is to developed efficiant Algorithm for processing of our data. Therefore, time and space are to major of of an algorithm.

(i) The complexity of an algorithm are the functions which gives us the learning time and space in terms of input size.

(ii) Each of our algorithm will involve a particular data structure.

(iii) Accordingly, we may not be always able to use the most efficient algorithms, since the choice of data structure depends on many things including the type of data and frequency which various data ~~operating~~ operations are applied.

31.07.19

(iv) The complexity of an algorithm $f(n)$ which measures the time and space use by an algorithm in terms of input size (n).

(v) The choice of a particular algorithm depends on following performance anal. and Measurement -

1. space complexity.
2. Time "

Fixed length storage (Advantages)

(i) The ease of accessing data from any record.

(ii) The ease of updating data on any record as long as the length of the new data doesn't exceed record length.

Disadvantages

(i) Time is wasted reading and entire record if most of the storage consists of

(ii) certain records main required more space than available.

(iii) When the correction consists of more or f characters than the original texts, changing a miss ward requires the entire record will be change.

variable length storage

(i) There are advantages of knowing the actual length of each string

(ii) The storage of variable length in memory cells can be done in two ways -

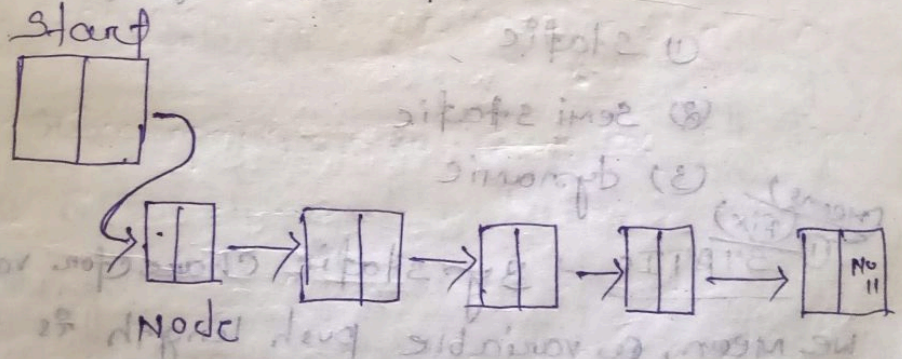
(a) one can use a marker such as $ (carpet-sign) to signal the end of a string.

(B) one can list the length of the string as an additional item in the pointer array.

* Link storage

(i) computers are being frequently use for word processing i.e. inputing, processing and outputting printed matter. Therefore the computer must be able to current and Modifie (which usually means deleding, changing and insording words, phrases, sentences and even paragraphs in the texts). How ever the fixed length memory storage doesn't easily lend themselfs for these operations.

(ii) Therefore the most extensive word processing applications, strings are stored by the means of Link list.



05.08.19

- **Character data type**

Various programming languages have there own way of handling character types data.

- **constants**

Many programming Languages denote string constants by placing the string in either single or double cotation mark.

Ex : A = "ABC"

A = 'ABC'

- **Variables**

Each programming Language has a fixed set up rules for forming character variables. However such variables can fall into one of the following three categorys. -

(1) static

(2) semi static

(3) dynamic

(means) (Fix)

(1) STATIC → By static character variable we mean, a variable push length is defind before the program is executed and cannot be change throughout the program.

2. semi static :- By a semi static character variable we mean the variable whose length may vary during the exiqution of the program.

3. Dynamic :- By a dynamic character variable we mean a variable whose length can change during the exiqution of the program.

- STRING OPERATIONS

The following operations can be performed on a strings

(1) Length :- The number of characters in a strings is called as its length. To find the length of a string we can use :-

strlen (string).

Ex :- strlen ("computer") = 8

strlen ("_ _") = 2

strlen (" ") = 0 (Null string / Empty)

2. Sub string :- A string 'y' is called as a substring of 'S' if there exist strings 'x' & 'z' such that -

$S = x \| y \| z$

Ex :- S = computer

y = am

(3) Index : If we are finding the position where the string pattern 'p' 1st appears in a given string 'T', then this operations is called as index and is denoted by -

Index (Text, pattern)

Example: His father is ☐ the professor

Index (T, 'the') = 7

Index (T, 'then') = 0

Index (T, '☐the') = 15

Index (T, 'he') = 8

(4) Concatenation : Let S1 & S2 be two strings. The string concatenation of characters of S1 followed by the characters S2 is known as string concatenation of S1 & S2 and is denoted by S1 || S2.

Ex : S1 = Hello

S2 = world

S1 || S2 = HelloWorld

S2 || S1 = WorldHello

Assignment-2

(2) What are the various types of string variables ?

(1) What is a string ?

(3) What are the various types of string | string operations that can be performed on them

06.08.19

Chap-3
Arrays

(i) Data structure are classifys as either linear or non-le

(ii) A data structure is said to be linear if its elements are stored in a sequential list. A data structure is said to be non-linear if its elements are not store in a sequential list.

(iii) There are two ways of representing linear structure in memory - (A) one way is two have linear relationship betn the elements by means of sequential memory location.

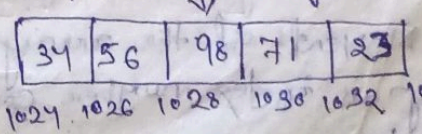
(B) The other way is two have linear relationship by means of pointers are ~~linked~~ Linked-list.

Arrays :

Array is a linear data structure, collection of similar types of data elements arranged in a linear sequence.

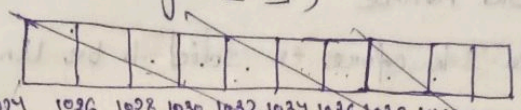
Ex:- Syntax → data type arrayname [size];

int array-1 [5];



(i) Declare a character array of size 10 and a float array size 15.

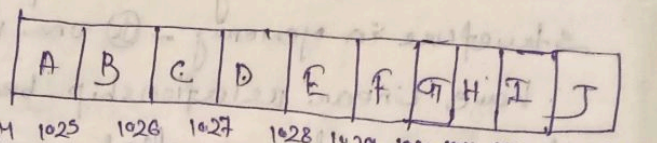
→ Char array1[10];



float array1[15];

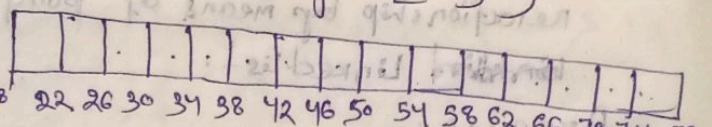
Char array1[10];

array1



float Float array1[15];

array1



(ii) Array can be devided into two categorys (1 dimensional array (2) Multidimensional array)

Linear array

A linear array is the list of finite number 'n' of homogeneous data elements such that

(a) Elements of the array are referenced by an index number consisting of consucuted numbers.

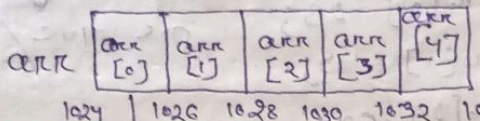
(B) The elements of the array are stored respect-ively in successive memory location.

07.08.[ ]

(C)

(D) Array elements that can be processed or accessed by using a single index along with a subscript operator is called as one dimensional array.

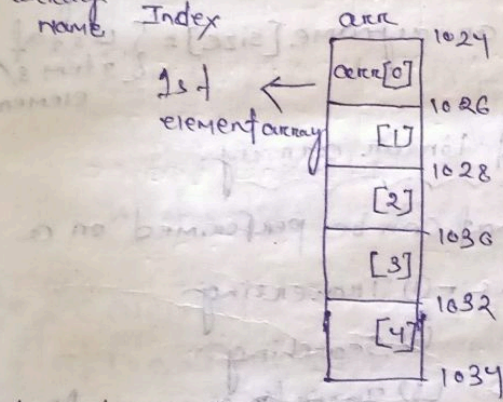
Ex: int arr[5];



1st element.

arr [0]

arr → array name
[0] → Index
[ ] → Subscript operator



1st dimensional array / Declaration

The syntax for declaration of Multi array

datatype arrayname [size];

int array1 [5];

- Initialisation of an array

Ex: int arr[5]; // array decl^n

int arr[5] = {23, 45, 69, 72, 91}; //

array initialisation

or

int arr[0] = 23; one value

int arr[1] = 45;

int arr[2] = 69;

int arr[3] = 72;

int arr[4] = 91;

(ii) The syntax per initialisation of array is datatype arrayname[size] = {list of item s/ elements}

(iii) ~~operation~~

Operation of Linear array

(i) The operations that can be performed on a Linear array are - (1) Traversing

(2) Searching

(3) ~~Inserting~~ Insertion

(4) Deletion

(5) Merging

(6) Sorting.

Memory allocation in 1(d) array

Memory allocation in one dimensional array can be done a ~~continuous~~ Contineous memory where each element will be side of the

address next two the other element.

int arr[5];

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
1016	1018	1020	1022	1024 1026

char arr[5];

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
1030	1031	1032	1033	1034 1035

08.08.19

Calculation of Memory address in 1(d) array

The formula to calculate the memory address of a particular element in 1(d) array is given by

$$\text{Address of } Arr[i] = \text{Base address} + i \times w$$

where Arr denotes the array name

'i' = denotes the index of

base address = denotes the starting of the array

'w' = denotes the word size of the data type being stored.

Ex: Calculate the memory address of location '5' where the array is an int array with base address 1015

int = 2 bytes
char = 1 "
Float = 4 "
double = 8 "
long double = 10 "

Soln

i = 5

Base address = 1015

w = 2

Arr[i] = Base address + i × w

Arr[5] = 1015 + 5 × 2

= 1025

Ex-Q2.

Calculate the memory address of position '7' in a floating point array where the base address is 2017

i = 7

Base address = 2017

w = 4

Arr[5] = 2017 + 5 × 4

= 2045

Q3

Calculate the memory add of position 11 in a long double array where base address is 1012

i = 11

Base = 1012

w = 10

Arr[i] = Base add + i × w

Arr[11] = 1012 + 11 × 10

= 1122

④ Calculate the memory address of position 25 in a Char array where base addr is 2056

Char : 25

B.a = 2056

w = 1

arr[c] = Base add + c × w

arr[25] = 2056 + 25 × 1

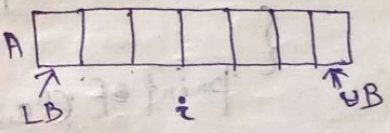
= 2081

① Traversing (operation of array)

Traversing is a process of visiting each element of the array exactly once i.e. starting from the 1st elements of to the last element.

Algorithm For traversing a Linear array

- 'A' is a Linear array on which traversing is to be done.
- 'LB' denotes the ~~integer~~ Lower bound index of the array 'A'.
- 'UB' denotes upper bound index of the array 'A'.
- 'i' denotes a counter



(1) Start
(2) i = LB
(3) • For i = LB to UB
(4) print A[i]
(5) Increment i by 1
(6) Stop

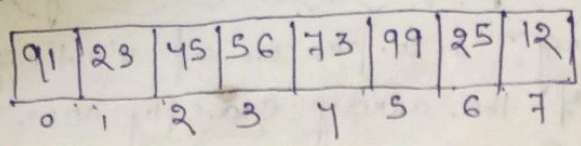
Ex:

23	45	96	71	19
0	1	2	3	4

LB = 0 UB = 4

(1) i = 0 (true) 0-4 print A[0] o/p → 23
(2) i = 1 (true) o/p 45 print A[1]
(3) i = 2 (true) o/p 96 print A[2]
(4) i = 3 (true) o/p 71 print A[3]
(5) i = 4 (true) 19 print A[4]
(6) i = 5 [false] Stop

Assignment-3

① 

09.08.19

Program

```
# include <stdio.h>
# include <conio.h>
  void main ()
{
  int i, a[5];
  clrscr();
  for (i=0; i<=4; i++)
  {
    printf ("enter element %d \n");
    scanf ("%d", &a[i]);
  }
  printf ("After traversing");
  for (i=0; i<=4; i++)
  {
    printf ("element %d is %d \n", i, a[i]);
  }
  getch ();
}
```

Control-f9 to

Enter element

1
2
3
4
5

2nd program

```
#include <stdio.h>
#include<conio.h>
  voide main()
{
 int i, n, a[15];
 clrscr();
printf("Enter size of the array");
scanf("%d",&n);
 for(i=0; i<=n-1; i++)
 {
printf("Enter element\n");
 scanf("%d",&a[i]);
 }
printf("after traversing");
for(i=0; i<=n-1; i++)
{
printf("Element %d is %d\n", i,a[i]);
}
getch();
}
```

(Enter) ctrl-f9 (8)

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

10.08.19

2. Insertion : To insert a new element in an existing array, following different positions can be used. -

① At the end of the array
② At the beigning of the array
③ At a given position.

① Insertion of at the end of the array —

'A' is an array on which insertion will be done.

MAX : Max is the total number of Location in the given array 'A'.

DATA : Data is the new item to be inserted into the array 'A'

UB : upper Bound denotes index of the array 'A'.

Algorithm for insertion at end of the array

① START
2. If UB = MAX then write array overflow and stop.
3. Read DATA
4. UB ← UB + 1
5. A[UB] ← DATA
6. STOP

Ex 1 | 23 | 45 | 63 | 95 | 21 | MAX = 10

Insert 53 at the end of the array

Soln

23	45	63	95	21
0	1	2	3	4

LB = 0 UB = 4

DATA = 53 MAX = 10

$4 \neq 10$

DATA = 53

UB = UB + 1 = 4 + 1 = 5

A[5] = 53

A

23	45	63	95	21	53
0	1	2	3	4	5

Q2

31	41	51	61	71	81

MAX = 15

3. Insert 91 at the end of the given array.

4. Insert 101 at the end of the Modified array.

5. Insert 111 at the end of the pre-Modified array.

Ans

31	41	51	61	71	81
0	1	2	3	4	5

LB = 6 UB = 5

DATA = 91 MAX = 15

$5 \neq 15$

DATA = 91

UB = UB + 1 = 5 + 1 = 6

A[6] = 91

A

31	41	51	61	71	81	91
0	1	2	3	4	5	6

(4)

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

LB = 0 UB = 6

DATA = 101 MAX = 50

6 ≠ 15

DATA = 101

UB = UB + 1 = 6 + 1 = 7

A[7] = 101

A

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

(5)

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

UB = 7

DATA = 111 MAX = 50

7 ≠ 15

DATA = 111

UB = UB + 1 = 7 + 1 = 8

A[8] = 111

A

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

Algorithm for insertion at beginning of the array

'A' is the array in which insertion is to be done.

'MAX' is the total number of locations in the given array.

'LB' is the lowyer bound index of array 'A'.

'UB' is the upper bound index of array 'A'

'DATA' is the new item to be insertion into the array 'A'.

'K' is the counter value.

Algorithm

1) START
2) IF UB = MAX then print "Array overflow" and STOP.
3) Read DATA
4) K ← UB
5) Repeat steps 6 to 7 while K ≥ LB.
6) A[K+1] ← A[K]
7) K ← K − 1
8) A[LB] ← DATA
9) STOP

Ex:- A

11	22	33	44
0	1	2	3

Insertion 55 MAX = 10

Soln

11	22	33	44
0	1	2	3

MAX = 10

LB = 0 UB = 3

DATA = 55

UB ≠ MAX

DATA = 55

K = UB = 3

3 > 0 (True)

A,

				44
0	1	2	3	4

⇒ A[K+1] = A[K]

⇒ A[3+1] = A[3]

⇒ A[4] = A[3]

⇒ A[4] = 44

⇒ K = K-1 = 3-1 = 2

2 > 0 (True)

⇒ A[K+1] = A[K]

⇒ A[2+1] = A[2]

⇒ A[3] = A[2]

A

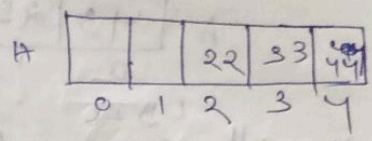
			33	44
0	1	2	3	4

K = 2-1 = 1

1 > 0 (true)

⇒ A[K+1] = A[K]

⇒ A[1+1] = A[1]



⇒ A[2] = A[1] = 2

K = K-1

K = 0

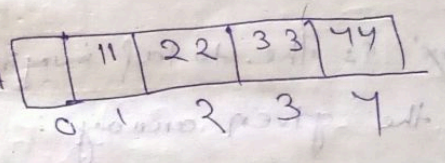
0 = 0 (true)

⇒ A[K+1] = A[K]

⇒ A[0+1] = A[0]

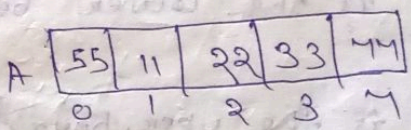
⇒ A[1] = A[0]

⇒ A[1] = [11]



A[LB] = DATA

⇒ A[0] = 55



K = 0-1 = -1

-1 ≱ 0 (false)

A[LB] = DATA

A[0] = 55

Q2 Assignment:

Insert 92 and beginings of the array?

	0	1	2	3	4	5
A	43	25	37	41	59	62

LB = 0 UB = 5

Algorithm for insertion at a given location of the array?

~~If UB = MAX then print "Array overflow" and stop~~

Set

'A' is the array in which insertion is to be done.

'MAX' is the total number of locations in the given array.

'LB' is the lower bound index of Array 'A'

'UB' is the upper bound index of Array 'A'.

'DATA' is the new item to be insertion into the array.

'K' is the counter value

LOC ← LOC is the location at which new item is to be inserted.

(1) START
2. If UB = MAX then print "Array overflow" and STOP.
3. Read DATA
(4) Read LOC
(5) K ← UB
(6) Repeat step 7 to 8 while K ≥ LOC
(7) A[K+1] ← A[K]
(8) K ← K-1
(9) A[LOC] ← DATA
(10) STOP

Ex 1. A

7	2	5	8	15	12	17
0	1	2	3	4	5	6

- insert and element '10' at position 4

MAX = 10

Soln

A

7	2	5	8	15	12	17
0	1	2	3	4	5	6

LB = 0 UB = 6 MAX = 10 DATA = 10

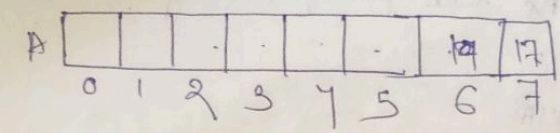
6 ≠ 10 (false) LOC = 4

DATA = 10 LOC = 4

K = 6 6 > 4 (true)

1. A[K+1] = A[K]
2. A[6+1] = A[6]

⇒ A[7] = A[6] = 17



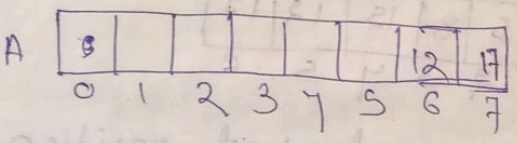
K = 6 − 1 = 5

K = 5 5 > 4 (true)

⇒ A[K+1] = A[K]

⇒ A[5+1] = A[5]

⇒ A[6] = A[5]

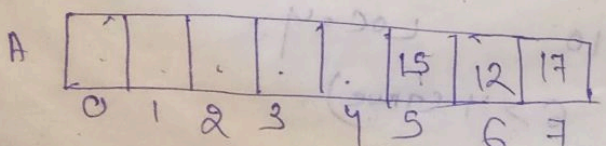


⇒ K = K − 1 = 5 − 1 = 4

⇒ A[K+1] = A[K]

⇒ A[4+1] = A[4]

⇒ A[5] = A[4]



K = K-1 = 4-1 = 3

⇒ A[K+1] = A[K]

⇒ A[3+1] = A[3]

⇒ A[4] = A[3]

A

				10	15	12	17	
0	1	2	3	4	5	6	7	

K = K-1 = 3-1 = 2

⇒ A[K+1] = A[K]

⇒ A[2+1] = A[2]

⇒ A[3] = A[2]

A

			8	10	15	12	17
0	1	2	3	4	5	6	7

K = K-1 = 2-1 = 1

⇒ A[K+1] = A[1]

⇒ A[1+1] = A[1]

⇒ A[2] = A[1]

		5	8	10	15	12	17
0	1	2	3	4	5	6	7

K = K-1 = 1-1 = 0

⇒ A[K+1] = A[1]

⇒ A[0+1] = A[1]

A[1] = A[0]

	2	5	8	10	15	12	17
0	1	2	3	4	5	6	7

k = k-1 = 0-1 = 0

A[k+1] = 1 to 0

17.08.19

3.3 Deletion

⇒ like insertion, Deletion is also posible at any position the array. like

1. From the end
2. From the begning
3. From the given elements.

Before deleting any element we have to chek the underflow condition.

① Deletion from the end of an array

'A' is the array on which deletion is to be done.

'LB' Lower bound index of Array 'A'.

'UB' is the upper bound index of Array 'A'

'N' is the number of element or the size of the array 'A'

⇒ START
2. If N = 0 then print "Array underflow" and STOP
3. A[UB] = NULL
4. UB ← UB - 1
5. STOP

$$N = UB - LB + 1$$

Ex-1

21	43	27	51	93

Delete the element from the End of the array

Soln

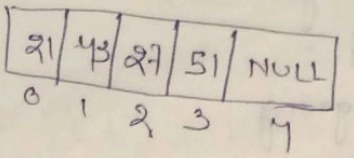
A

21	43	27	51	93
0	1	2	3	4

LB = 0 UB = 4

N = 4 - 0 + 1 = 5

A[UB] = A[4] = NULL



Deletion from the begning of array

Algorithm for deletion from the begning of an array.

'A' is the array on which deletion is to be done.

'LB' is the lower Bound index of Array 'A'

'UB' is the upper bound index of Array 'A'

'N' is the number

'K' is a counter

1) START
2. If N = 0 then print "Array underflow" and stop.
3. K ← LB
4. Repeat Step 5 to 6 while K < UB
5. A[K] ← A[K+1]
6. K ← K+1
7) UB ← NULL
8) UB ← UB - 1
9) Stop

22	11	33	55	

Delete the element from the beginning of the array?

Soln

A

22	11	33	55
0	1	2	3

LB = 0 UB = 3

N = 4

K = 0 0, 1, 2

A[K] = A[K+1]

⇒ A[0] = A[0+1] = A[1] = 11

A

11			
0	1	2	3

K = K+1 = 0+1 = 1

K = 1

A[K] = A[K+1]

⇒ A[1] = A[1+1] = A[2] = 33

A

11	33		
0	1	2	3

K = K+1 = 1+1 = 2

K = 2

A[K] = A[K+1]

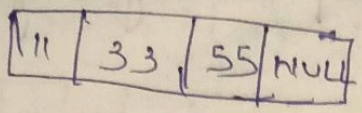
⇒ A[2] = A[2+1] = A[3] = 55

A

11	33	55	
0	1	2	3

k = k+1 = 2+1 = 3

k = 3 (false)



• Multi dimensional array

(i) An array with more than one dimension is known as multi dimensional array.

(ii) There is no restrictions upon the number of dimensions that we can have. But a multi dimensional array must be handle with or used with utmost care

Ex → int a[ ][ ]

Two dimensional array

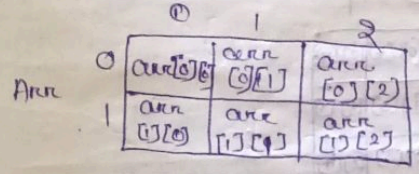
An array with two dimensions is known as two-dimensional array or 2(d) array

20.08.19

Syntax for 2-d array

Datatype arrayname [no. of rows] [no. of columns];

Ex → int arr [2] [3];



Representation of 2(d) array in memory

We can represent a 2(d) array in memory in 2 ways.

1) Row major order representation

2) Column Major order representation.

Row major order representation

(i) In this type of representation the elements of array are stored row wise.

(ii) That means zero (0) row element then through will be stored first, Then first row will be stored, Then second will be stored and show on.

Ex 1 Arr

	0	1	2
0	arr[0][0]	arr[0][1]	arr[0][2]
1	arr[1][0]	arr[1][1]	arr[1][2]

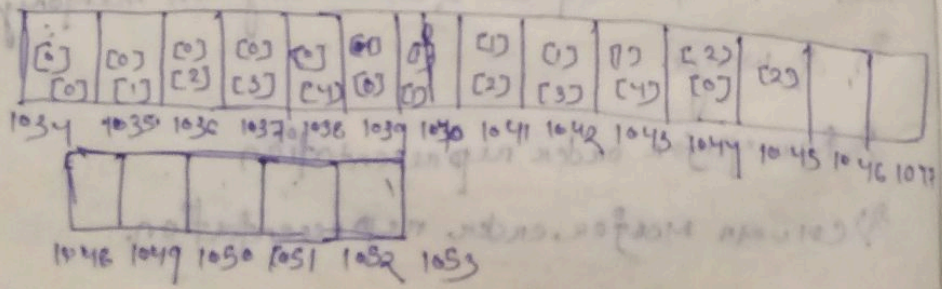
The row major order representation of the array is -

arr[0][0]	arr[0][1]	arr[0][2]	arr[1][0]	arr[1][1]	arr[1][2]
1034	1036	1038	1040	1042	1044	1046

Ex-2

- char arr[4][5]

	0	1	2	3	4
0	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
1	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
2	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
3	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

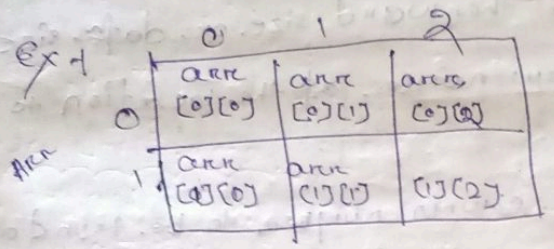


2. Column major order representation

(i) In this type of representation the elements of array will be stored column wise.

(ii) That means '0' column elements will be ~~stored~~ stored first, Then first column elements will be stored, Then second column will be ~~show~~ stored and show on.

Ex-1

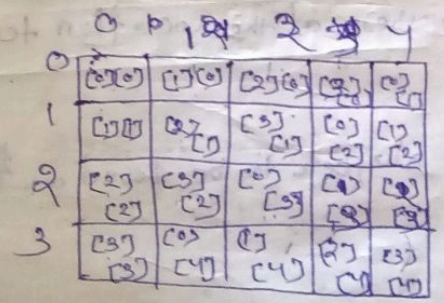


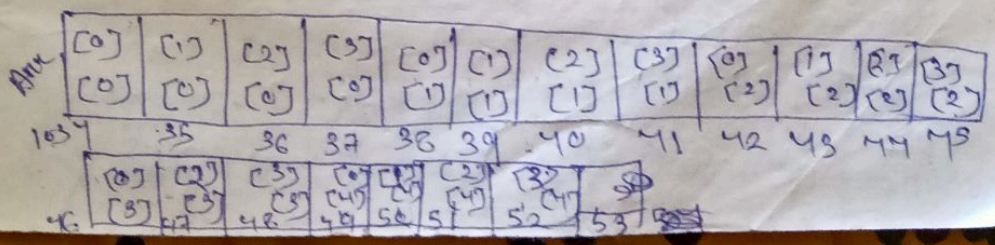
The column major order representation of the above array will be -

arr [0][0]	arr [1][0]	arr [0][1]	arr [1][1]	arr [0][2]	arr [1][2]

Ex-2

Char arr [4] [5]





Address Calculation of 2(d) array

(i) Row Major order representation

(a) LOC A[i][j] = Base Add(A) + w[n×i + j]

(b) where 'A' is an 2(d) array of size m×n

(c) 'w' denotes the word size or data size

(d) 'i' is the row number of the location to be found out.

(e) 'j' is the column number to be found out.

(ii) Column Major

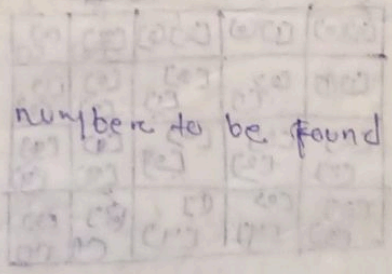
(a) LOC A[i][j] = Base Add(A) + w[m×j + i]

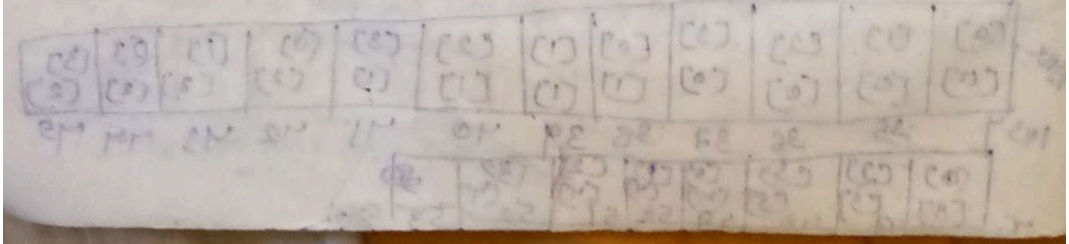
(b) where 'A' is an 2(d) array of size m×n

(c) 'w' denotes the word size or data size.

(d) 'i' is the row number of the location to be find out.

(e) 'j' is the column number to be found out.





Sparse Matrix

A Matrix have been more number of '0' elements as compair to non'0' elements is known as a sparse matrix.

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

3 - Tuple method

3 - tuple method is a convenient method for storing or representing a sparse matrix in memory.

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

Sparse matrix

3-tuple method of the sparse matrix

No of row	Row of column	value of non-zero element
0	0	5
1	1	1
2	2	4
3	3	3

pointer

pointer is a variable which stores the address of another variable.

ex → *c = &a

* pointer array

23.08.19

Let 'DATA' be an array. A variable 'AP' is called as a pointer if p[2] and

element in DATA. That is 'p' contains the address of an element in 'DATA'.

(ii) An array 'PTR' is called as a pointer array if each element of 'PTR' is a pointer.

(iii) pointer and pointer array are used to facilitate the processing of information in DATA.

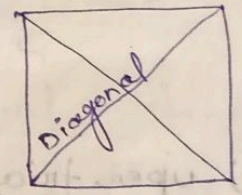
(2) special matrixes

A square matrix as the same number of rows and columns. Some forms of square matrixes are as follows

(a) Diagonal matrix

A matrix 'A' is called as a diagonal matrix. If and only if -

$$A[i][j] = 0 \text{ for } i \neq j$$



ex :

	0	1	2
0	8	0	0
1	0	2	0
2	0	0	5

(b) Tridiagonal matrix

A matrix 'A' is called as a tridiagonal matrix if and only if -

$$A[i][j] = 0 \text{ for } |i-j| > 1$$

Ex.

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

$A[0][0] = |0-0|$
$= 0 > 1$
$A[0][1] = |0-1|$
$= 1 \neq 1$
$A[0][2] = |0-2|$
$= |-2| = 2 > 1$

(C) Lower Triangular matrix

A matrix 'A' is lower triangular matrix if and only if →

$A[i][j] = 0$ for $i < j$

Ex →

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

(D) upper triangular matrix

A Matrix 'A' is upper triangular matrix if and only if-

$A[i][j] = 0$ for $i > j$

Ex.

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

$A[0][0] = 0-0$
$= 0 > 1$
$A[0][1] = 0-1$
$= 1 \neq 1$

Chapter-4

STACKS & QUEUES

STACKS → (i) stack is a list of element in which an element may be inserted or deleted at one end only.

→ It is a linear data structure in which insertion or deletion of an element will be done at one end only called as 'TOP' of the stack.

→ It is also known as push down stack or list.

→ It is one of the restricted data structure.

→ The last element which is the address in the stack is the 1st element to be deleted from the stack.

→ principle of stack or the logic that a stand follows is 'Last-In-First-out' or LIFO.

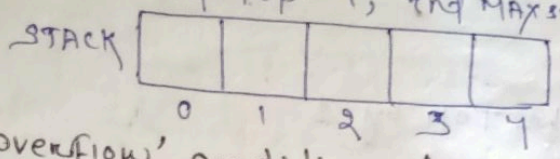
→ Example of stack is collection of fits..

→ special terminologies used for two basic operations in stack are -

(1) PUSH → push is turn use to insert an element in to the stack.

(2) pop → pop is a turn use to delete an element from the stack.

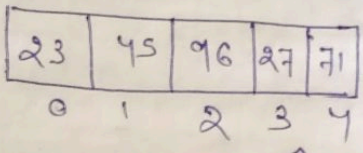
Ex: int STACK[5];
int TOP=-1, int MAXSTK=5



'overflow' condition of STACK

→ If the STACK is full then it is called as overflow condition of STACKS.

Ex



→ If the pointing

Then we can say that the stack is full.

→ The ~~area~~ to check the overflow condition

Algorithm for PUSH operation

~~PUSH~~ € 'STACK' is an array on which insertion is to be done.

'TOP' is the index of the Top element of the STACK.

'MAXSTK' is the size of the STACK.

'ITEM' is the element to be inserted to the STACK.

~~PUSH~~ PUSH (STACK, TOP, MAXSTK, ITEM)

① Check for overflow condition

```
if (TOP == MAXSTK-1)
{
printf ("STACK overflow")
```

}

exit

(2) Increment the TOP by 1

TOP = TOP + 1

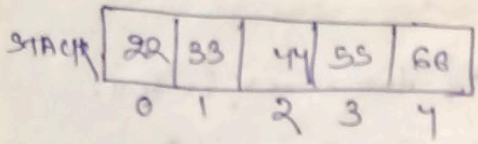
(3) Insert the new item into the STACK.

STACK[TOP] = ITEM

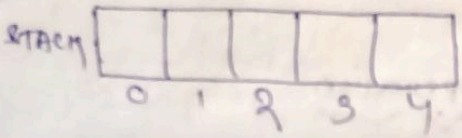
(4) STOP

Q1

Q1.



Soln



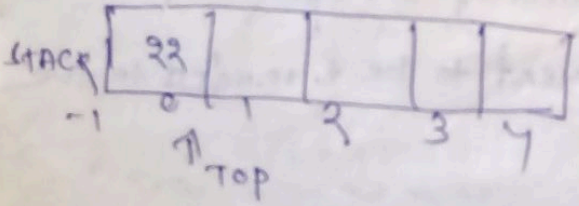
TOP = -1 MAXSTK = 5 ITEM = 22

-1 ≠ 4

TOP = TOP - 1 1

= -1 + 1 = 0

STACK[0] = 22

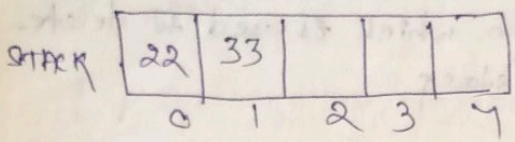


TOP = 0 MAXSTK = 5 ITEM = 33

0 ≠ 4

TOP = TOP + 1 = 0 + 1 = 1

STACK [TOP] = STACK [1] = 33

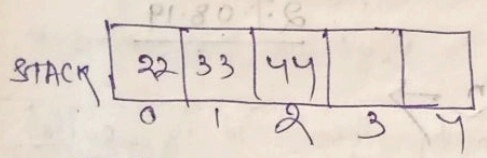


TOP = 1 MAXSTK = 5 ITEM = 44

1 ≠ 4

TOP = TOP + 1 = 1 + 1 = 2

STACK [TOP] = STACK [2] = 44

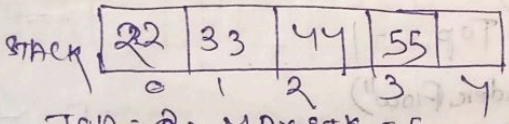


TOP = 2 MAXSTK = 5 ITEM = 55

2 ≠ 4

TOP = TOP + 1 = 2 + 1 = 3

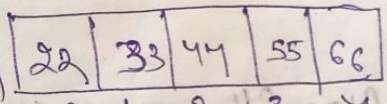
STACK [TOP] = STACK [3] = 55



TOP = 3 MAXSTK = 5 ITEM = 66

3 ≠ 4

TOP = TOP + 1 = 3 + 1 = 4



STACK [TOP] = STACK [4] = 66

TOP = 4 MAXSTK = 5 ITEM =

4 = 4

STACK OVERFLOW

26.08.19

Pop operation

→ pop is an operation which is used to delete an element from the stack

→ These operation can be perform in ~~two~~ two steps

1) ITEM = STCK [TOP]
2) TOP = TOP - 1

27.08.19

underflow condition →

→ If the stack is empty then it is called as underflow condition of stack.

→ If the top is poinfling to -1 then the stack is empty.

```
if (TOP = = - 1).
                Top = -1
printf ("stack underflow")
}
```

Algorithm for pop operation

→ 'STACK' is an array on which deletion is to be done.

→ 'TOP' is an index of the last element of the stack.

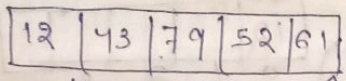
→ 'MAXSTK' is the size of the STACK.

→ 'ITEM' is the element to be deleted from the STACK

POP (STACK, TOP, MAXSTK, ITEM)

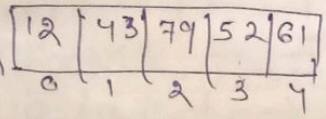
1) START
2. check for underflow condition
   IF (TOP=-1)
   print "stack underflow" and next
3. ITEM = STACK [TOP]
4. Decrement Top by 1
   Top = TOP - 1
5. STOP

Ex



Soln Delete elements till the STACK is empty.

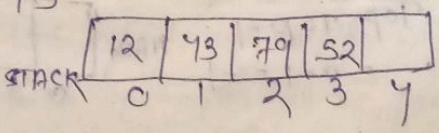
STACK



→ Top = 4 MAXSTK = 5

4 ≠ -1 (false)

ITEM = STACK [TOP]
= STACK [4]
= 61

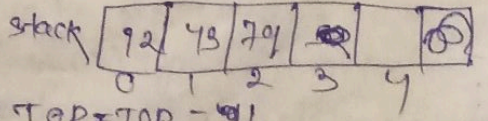


TOP = TOP - 4
= 4 - 3

→ Top = 3 MAXSTK = 5

3 ≠ -1 (false)

ITEM = STACK [TOP]
= STACK [3]
= 52



TOP = TOP - 1
= 3 - 1

→ TOP = 2 MAXSTK = 5

2 ≠ 1 (false)

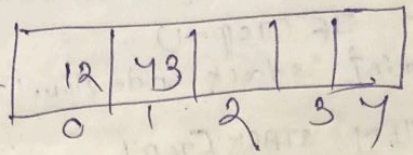
ITEM = STACK [TOP]

= STACK [2]

= 79

TOP = TOP - 1

= 2 - 1



→ TOP = 1 MAXSTK = 5

1 ≠ -1 (false)

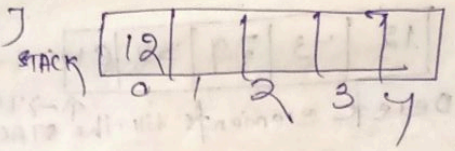
ITEM = STACK [TOP]

= STACK [1]

= 43

TOP = TOP - 1

= 1 - 0



→ TOP = 0 MAXSTK = 5

0 ≠ -1

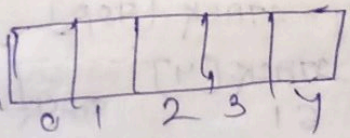
ITEM = STACK [TOP]

= STACK [0]

= 12

TOP = TOP - 1

= 0 - 1 = -1



* Arithmetic expression →

→ Five binary operations are presented in an arithmetic expression -

1) exponential operation (^)
2) Division operation (/)
3) Multiplication operation (*)
4) Addition operation (+)
5) subtraction operation (-)

* operator precedence →

Most precedence	( )	1	L to R
	^, ↑	2	L to R
	/, *	3	L to R
Least precedence	+, -	4	L to R

* Associativity →

→ If more than one 1) same precedence operation are present in a single expression, The rule which define which operation will be perform 1st and which operation will be perform Latter is known as operator associativity.

* Types of arithmetic expression →

→ Arithmetic expression can be represented in which can be using three (3) Types of notation-

① Infix notation (polish notation) ③ postfix notation
② prefix notation

① Infix notation → operands a, b; operator ⊕

→ If an operator is present in bet two operands. Then it is known as an infix notation.

→ ex :- a + b
b * c

② prefix / polish notation

→ If an operator is present before the operands then it is known as prefix notation.

→ ex :- + a b
* b c

③ postfix notation

→ If an operator is present after the operands Then it is known as postfix notation.

→ ex :- a b +
b c *

30. 8.19

conversion of an infix expression to prifix expression →

ex → a - b * c
= a - * bc
= - a * bc

ex-2 → a - b * c + d / f
= a - b * c + / df
= a - * bc + / df
= a - + * bc / df
= - a + * bc / df

(3) $a * (b-c) \wedge d / f + e$

$= a * -bc \wedge d / f + e$

$= a * \wedge - bc \; df + e$

$= a * / \wedge - bc \, df + e$

$= * a / \wedge - bc df + e$

$= + * a / \wedge - bc df e$

(4) $a + b * c - d / e$

$= a + b * c - de$ Wrong

$= + ab * c - de$

$= - + a * bc / de$

- Conversion of infix expression to postfix expression

→ (1) $a - b * c$

$= a - bc *$

$= abc * -$

(2) $a - b * c + d / f$

$= a - b * c + df /$

$= a - bc * + df /$

$= a - bc * df / +$

$= abc * df / + -$

(3) $a * b + (c-d) \wedge e$

$= a * b + cd - \wedge e$

$= a * b + cd - e \wedge$

$= ab * + cd - e \wedge$

$= ab * cd - e \wedge +$

(4) $a * (b-c) / d \wedge e + f$

$= a * bc - / d \wedge e + f$

$= a * bc - / de \wedge + f$

$= a * bc - de \wedge / + f$

$= abc - de \wedge / + f$

$= abc - de \wedge / * f +$

• Applications of STACKS →

The various application of STACK are -

① Evaluation of arithmetic expression

② postponed decision

③ Quick Sort

④ Recursion.

Quick sort → ① Quick sort is a devide and a conquer approach.

② It picks and element as the pibot element and partition any given array.

③ Quick short is an efficient sorting algorithm 31.08.19

systematic method for placing the elements of array in order

④ Recursion

→ Recursion is a process of expressing a function in terms of itself.

→ Recursion can also be defind as a process in which a function calles it self with reduced input and has a base condition to stop.

→ programming in iterative method helps the programmer to perform sorteen task repeatedly for a known number of times.

→ We can also do this by using the for loop, while loop, do-while loop... But our interest is to make a function repeatedly in terms of itself. So these mechanism is called as recursion.

* QUEUE →

→ Queue is a linear data structure in which insertion of element will be done at rear end and deletion of an element will be done at front end.

→ It is one of the n data structure which follows first in first out (FIFO) principle.

→ Ex → Queue of people in front of a ticket counter.

* Types of queue →

There are 3 types of queue -

1. Linear queue
2. Circular queue
3. priority queue

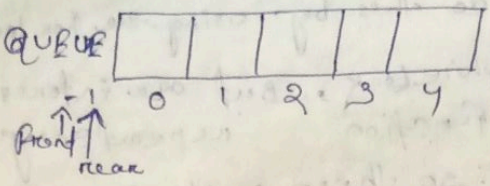
① Linear queue →

→ A linear queue can be represented in memory in 2 ways - ① using array
② using link list

② Representation of queue using array →

→ Array representation of a queue can be done by the following steps -

```
int QUEUE [5];
int front = -1;
int rear = -1;
int MAX = 5;
```



→ There are two operations that can be perform on a queue - ① Insertion
② Deletion

overflow condition

If rear = = MAX queue - 1

It means that the queue is full or overflow i.e., no more insertion is possible

insertion → rear end

04.09.19

• Insertion in

→ Insertion of an element will be done at the rear end of the queue.

Queue [rear] = item

rear = rear + 1

• Algorithm for insertion on linear queue →

→ 'QUEUE' is an array in which insertion of an element is to be done.

→ 'ITEM' is the element which is to be inserted

→ 'REAR' is the index at which insertion can be done.

→ 'FRONT' is the index at which deletion can be done.

→ 'MAXQ' is the total number of elements in the queue.

1) START

2) [check overflow condition]

if rear == Maxq-1

print "queue overflow" and stop

3) Set rear

if front == -1

rear = 0

front = 0

else

rear = rear + 1

(==) → equality

(=) → assign

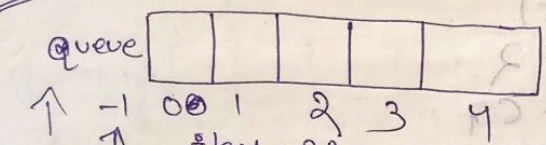
4) [store item]

queue[rear] = item

5) stop

Q. Insert 23, 55, 67, 29, 95 into an empty 'q'

Soln



item = 23

Maxq = 5

Maxq-1 = 5-1

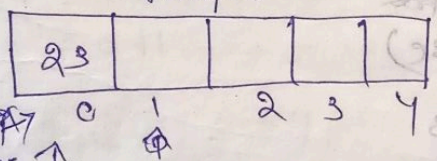
= 4

-1 == 4 (false)

front == -1 (true)

rear = 0

front = 0



queue[rear] = Item

queue[0] = 23

item = ~~23~~ 55

max q = 5

item = 55

rear = 0

0 == 4 (false)

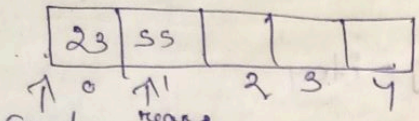
front == -1 (~~true~~ false)

rear = rear + 1

= 0 + 1

= 1

Queue [1] = 55



item = 67

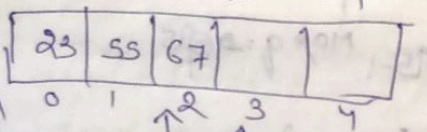
1 == 4 (false)

0 == -1 (false)

rear = rear + 1

= 1 + 1 = 2

Queue [2] = 67



item = 29

2 == 4 (false)

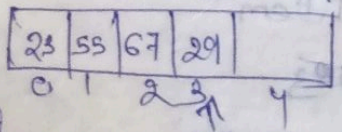
~~front == -1 (fa~~

0 == -1 (false)

rear = rear + 1

= 2 + 1 = 3

Queue [3] = 29



item = 95

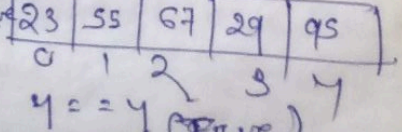
3 == 4 (false)

0 == -1 (false)

rear = rear + 1

= 3 + 1 = 4

Queue [4] = 95



4 == 4 (true)

Queue overflow

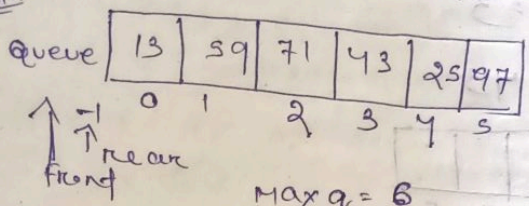
rear == 4 (True)

Queue overflow

~~Assignment~~

Q1 Insert 13, 59, 71, 43, 25 & 97 into an Empty 'q'

Soln



Max q = 6

Max q - 1 = 6 - 1

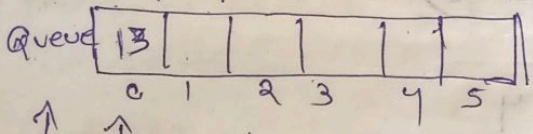
= 5

rear == 5 (False)

front == -1 (True)

rear = 0

front = 0



item = 0

~~Max q = 5~~

~~rear == 5~~

0 == 5 (False)

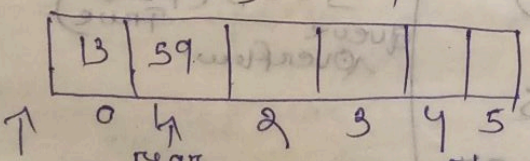
front == -1 (False)

rear = rear + 1

= 0 + 1

= 1

queue [1] = 59



item = 71

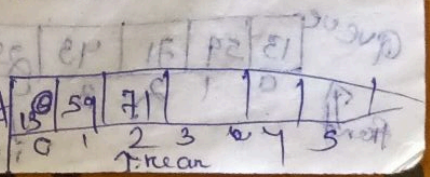
1 == 5 (False)

0 == -1 (False)

rear = rear + 1

= 1 + 1 = 2

queue [2] = 71



item = 43

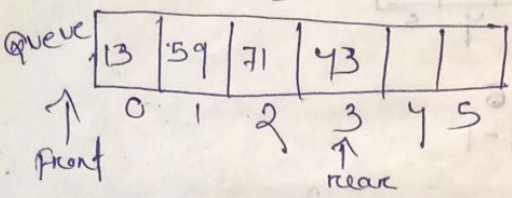
2 == 5 (false)

0 == -1 (false)

rear = rear + 1

⇒ 2 + 1 = 3

Queue[3] = 43



item = 25

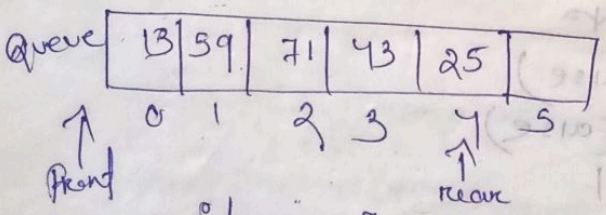
3 == 5 (false)

0 == -1 (false)

rear = rear + 1

⇒ 3 + 1 = 4

Queue[4] = 25



item = 97

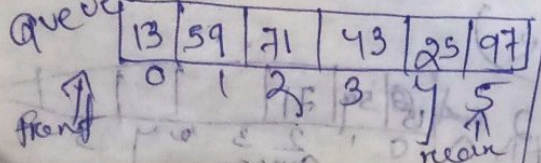
4 == 5 (false)

0 == -1 (false)

rear = rear + 1

⇒ 4 + 1 = 5

Queue[5] = 97



5 == 5 (false / True)

Queue Overflow

06.09.19

- for deletion we must 1st check the queue underflow condition i.e, if (front == -1)
  {
  printf "queue underflow"
  }

- Algorithm for deletion in linear queue

→ 'QUEUE' is an array in which deletion to be done.

→ 'ITEM' is the element which is to be deleted from the queue

→ 'REAR' is the index at which insertion can be done.

→ 'FRONT' is the index at which deletion can be done

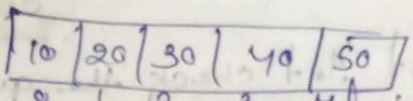
1) START

2) [check for underflow condition]
   if (front == -1)
   print "queueunderflow" and stop

3) [ITem to be deleted
   item = queue[front]

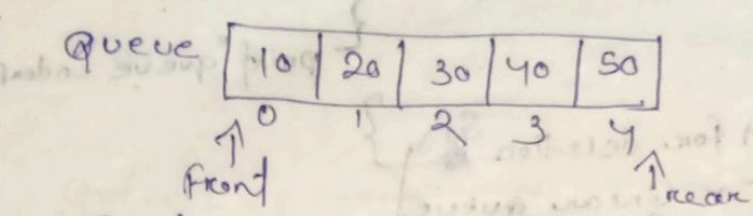
4) [set front]
   if (front == rear)
   set front = -1
   set rear = -1
   else
   set front = front + 1

5) STOP



Prob^M Delete all the element sequencially from the

Soln



Front = 0 rear = 4 item = 10

0 == -1 (false)

∴ No underflow

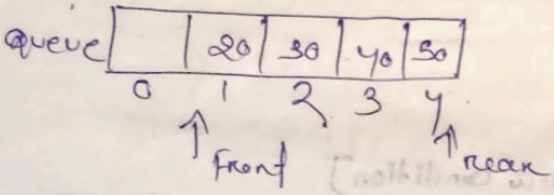
item = queue[0] = 10

0 == 4 (False)

∴ Front = front + 1

= 0 + 1

= 1



Front = 1 rear = 4

1 == -1 (false)

∴ No underflow

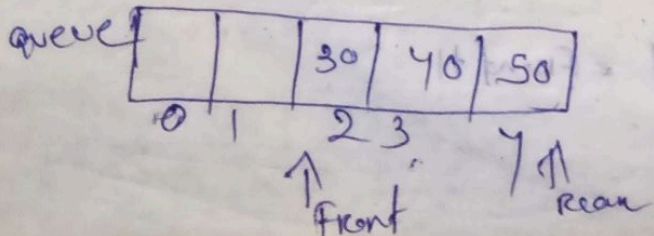
item = queue[1] = 20

1 == 4 (false)

∴ Front = Front + 1

= 1 + 1

= 2



front = 2 rear = 4

2 == -1 (false)

⇒ No underflow

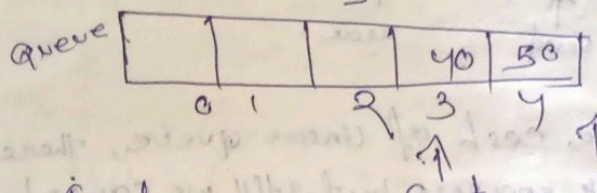
item = queue[2] = 30

2 == 4 (false)

⇒ front = front + 1

= 2 + 1

= 3



front = 3 rear = 4

3 == -1 (false)

⇒ No underflow

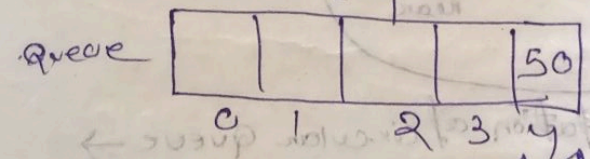
item = queue[3] = 40

3 == 4 (false)

⇒ front = front + 1

= 3 + 1

= 4



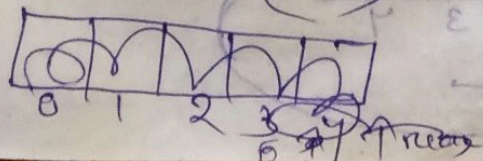
front = 4 rear = 4

4 == -1 (false)

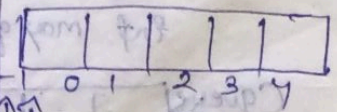
⇒ No underflow

item = queue[4] = 50

4 == 4 (True)



front = -1, rear = -1



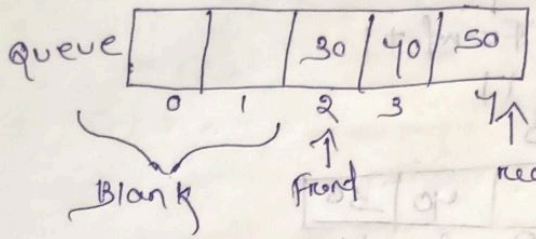
front == -1 (True)

"Queue underflow"

stop

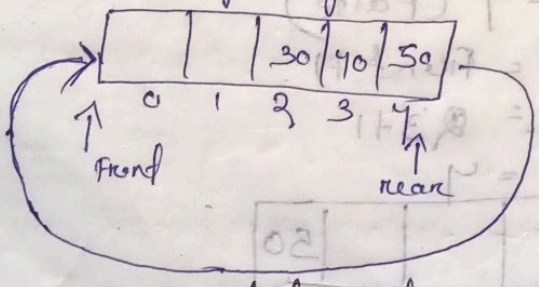
* If Front == -1, only one element is present in the queue.

- This advantage of Linear Queue ?



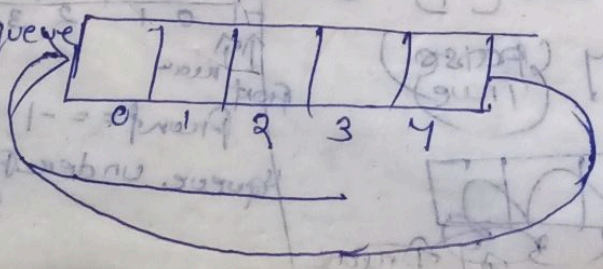
→ In the above case of linear queue, there are two blank spaces but still we cannot insert an element into the queue. Because rear is pointing to maxq - 1, which will show queue is full even if there are blank spaces.

So we can overcome these problems of Linear queue by using a circular queue



- Array representation of circular queue →

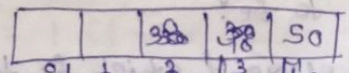
```
int cqueue [5];
int front = -1
int rear = -1
int maxq = 5
```



Algorithm for insertion in circular queue →

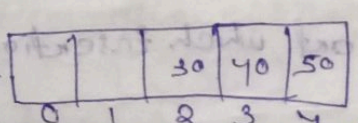
→ 'CQUEUE' is an array in which insertion of an element is to be done.
→ 'ITEM' is the element which is to be inserted.
→ "Rear" is the index at which insertion can be done.
→ 'FRONT' is the index at which deletion can be done.
→ 'MAXQ' is the total number of element in the queue.

1) START
2) [Check for overflow condition]
   if(Front == 0 && rear == MAXQ-1 or
   Front == rear+1)
   print "queue overflow" and stop";
3) Read item
4) [Set rear]
   if (Front == -1)
   set rear = 0
   set Front = 0
   else
   set rear = (rear+1) % maxq
5) Insert item
   Cqueue [rear] = item
6) stop



Prob^M Insert 10 and 20 into the blank spaces.

Sol^M

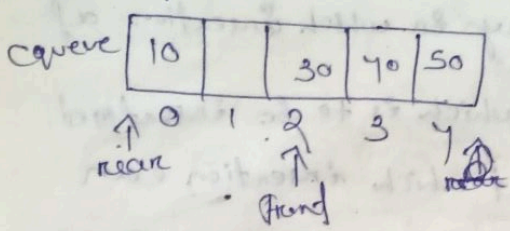


item = 10

f == -1 (false)

rear = (4+1) % 5 = 5 % 5 = 0

Cqueue[0] = 10



item = 20

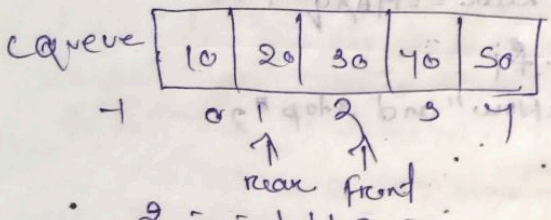
q = = -1 (False)

rear = (0+1) % 5

= 1 % 5

= 1

Cqueue[1] = 20



q = = 1+1 = 2

(True)

→ Queue overflow

07.09.19

Algorithm for Deletion in circular queue →

→ 'CQUEUE' is an array in which insertion of an element is to be.

→ 'ITEM' is the element which is to be inserted.

→ 'REAR' is the index at which insertion can be done.

→ 'front' is the index at which deletion can be done.

→ 'MAXQ' is the total number of element in the queue

1) Start

2) [check underflow condition]

3) if (front == -1)
   print "queue underflow" and exit

4) Item to be deleted
   item = queue [front]

5) Set front
   if front == rear
   set front = -1
   set rear = -1
   else
   set front = (front+1) % maxq

6) stop

Q1

10	20	30	40	50
0	1	2	3	4

rear = 1, front = 2

Soln

cqueue

10	20	30	40	50
0	1	2	3	4

rear = 1, front = 2

maxq = 5 rear = 1 front = 2

2 == -1 (false)

No underflow

item = cqueue[2] = 30

2 == 1 (false)

front = (2+1) % 5

= 3 % 5

= 3

cqueue

10	20		40	50
0	1	2	3	4

rear, front

maxq = 5 rear = 1 front = 3

3 == -1 (false)

No underflow

item = queue[3] = 40

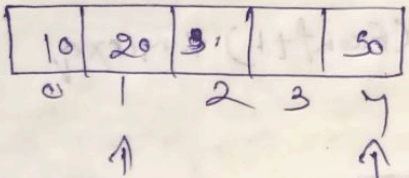
3 == 1 (false)

front = (front + 1) % maxq

= (3 + 1) % 5

= 4 % 5

= 4



maxq = 5 rear = 1 front = 4

4 == -1 (false)

No underflow

item = queue[4] = 50

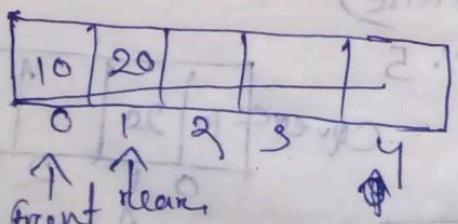
4 == 1 (false)

front = (front + 1) % maxq

= (4 + 1) % 5

= 5 % 5

= 0



maxq = 5 rear = 1 front = 0

0 == -1 (false)

NO underflow

item = queue [0] = 10

0 == 1 (false)

front = (front + 1) % 5

=

Priority queue →

→ A priority queue is a collection of elements such that each element has been assign a priority such that the order in which elements are deleted or processed comes from the following rules.

① An element of higher priority is processed before any element for lower priority.

② Two elements with the same priority are processed according to the order in which they are added to the queue.

Ex → Suppose there are five elements with the following priority.

AA → 1
BB → 3
CC → 4
DD → 7
EE → 6

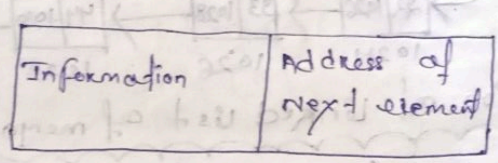
Ch-5 09.09.19

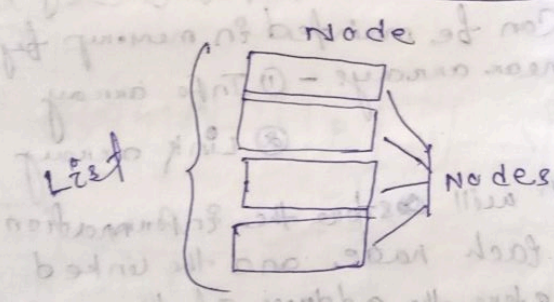
Linked list

→ LIST →

→ List refers to a collection of data items.

→ List is a linear collection of data elements known as "node", where each data element has two parts - 1st part is the information part that will hold the data & the 2nd part is called as the address or linked or next part that will hold the address of the next element.





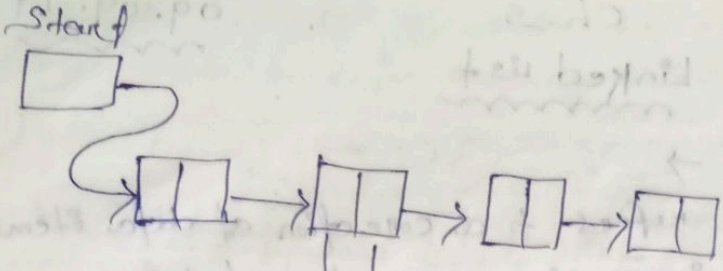
• Linked List →

→ A Linked list or one way list is a linear collection of data elements called as nodes. where a linear order is given by means of a pointer.

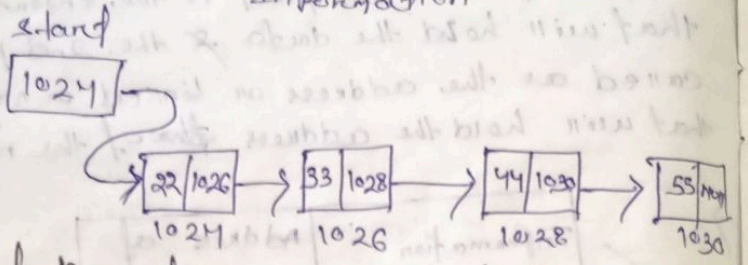
→ Each node is devided into two parts -

① contains the information of element

② contains the address of next node in the list



Ex →



Representation of Linked list of memory →

→ Link. list can be maintained in memory by using two linear arrays - ① Info array
② Link array

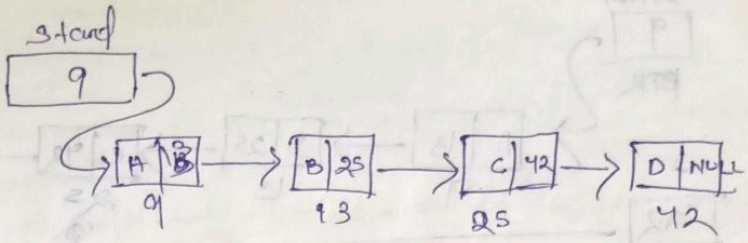
→ Info array will store the information or data of each node and the linked array will store the address of the nodes.

Traversing a Linked List →

→ 'START' contains the starting address of the elements or nodes in the linked list.

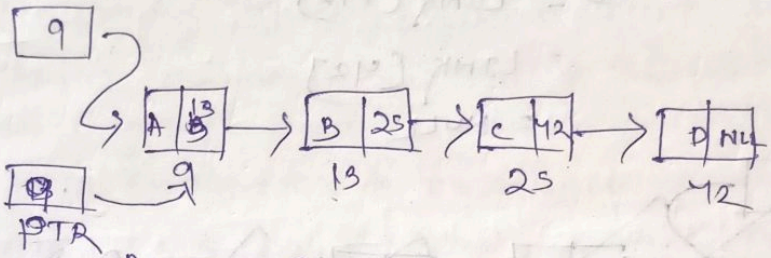
1) Start
2) set PTR = START
3) Repeat steps 4 & while PTR ≠ NULL
4) Apply PROCESS to INFO[PTR]
5) set PTR = LINK[PTR]
6) stop

ex → start



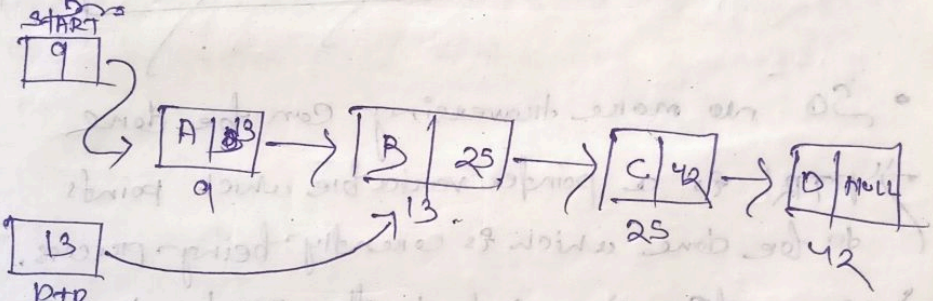
Soln

start



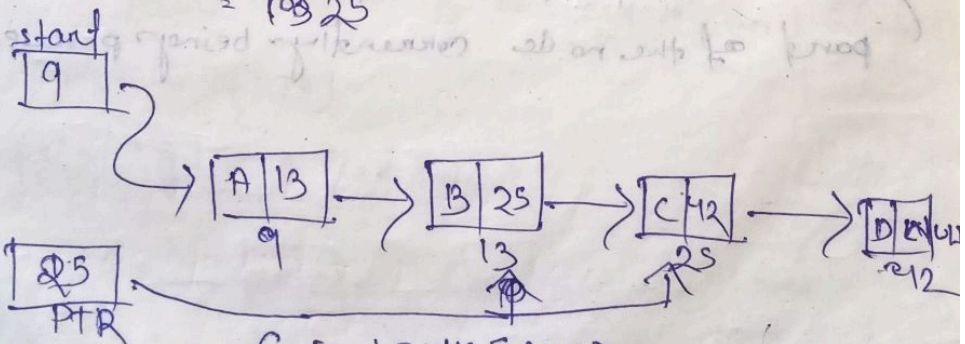
A = LINK [PTR]
= LINK [9]
= 13

start

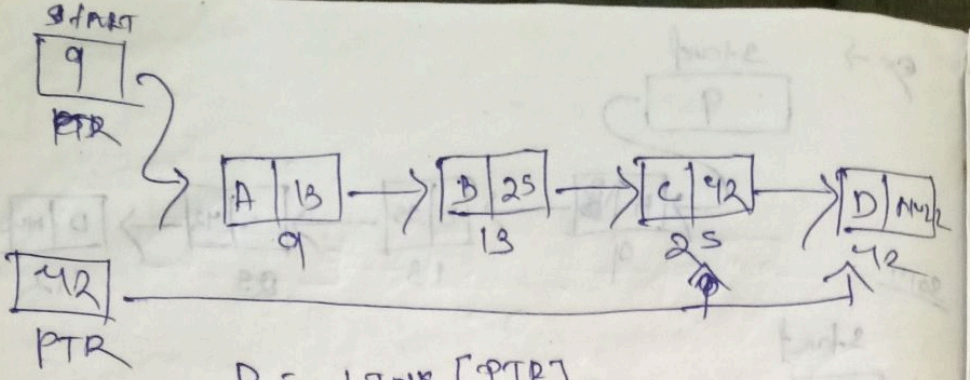


B = LINK [PTR]
= LINK [13]
= 25

start



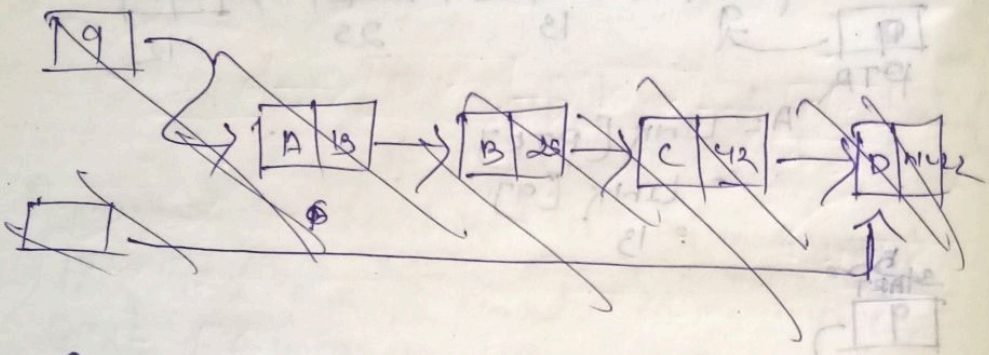
C = LINK [PTR]
= LINK [25] = 42



D = LINK [PTR]

= LINK [42]

= NULL



- So no more traversing can be done

→ 'PTR' is a pointer variable which points to the node which is currently being process.

→ Link of [PTR] points to the next node to be processed.

→ 'INFO' of 'PTR' points to the information part of the node currently being process.

10.09.19

Searching a Link List →

→ To locate or search weather any particular information is available in the linked list or not is known as searching. Basically we are performing a linear search.

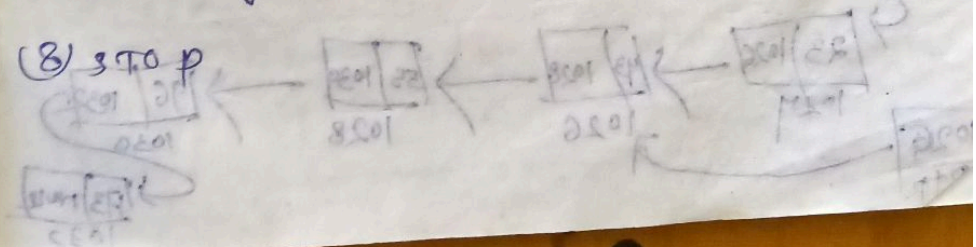
Algorithm for searching a linked list →

→ 'START' contains the starting address of the element or node in the linked list.

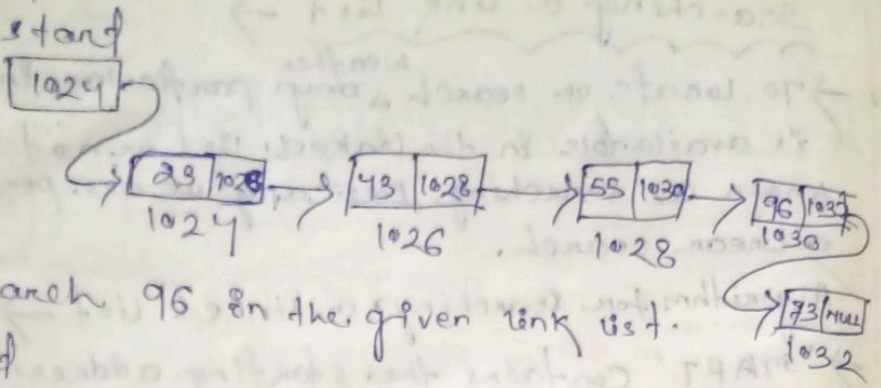
→ 'DATA' is the item to be search.

→ 'PTR' is a pointer variable which points to be node which is currently being process. Link of PTR points to the link of the current node being processed. 'INFO' of PTR points to the information part of the node currently being process.

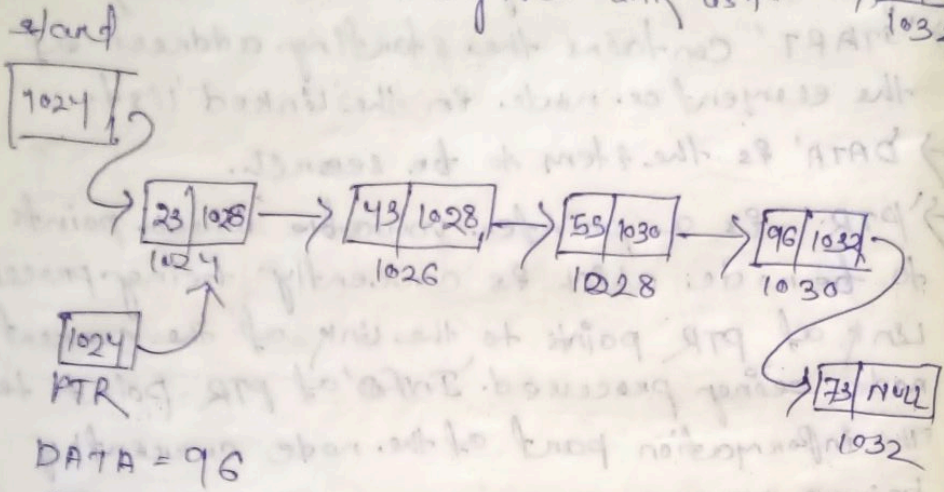
(1) START
(2) Read data
(3) PTR = START
(4) Repeat steps 5 & 6 while INFO[PTR] ≠ ~~NULL~~ DATA
(5) If INFO[PTR] == DATA
print "searching successful" and STOP
(6) PTR = LINK[PTR]
(7) print "ITEM not found"
(8) STOP



Ex →



search 96 in the given link list.



DATA = 96

PTR = START = 1024

INFO[PTR] = 23

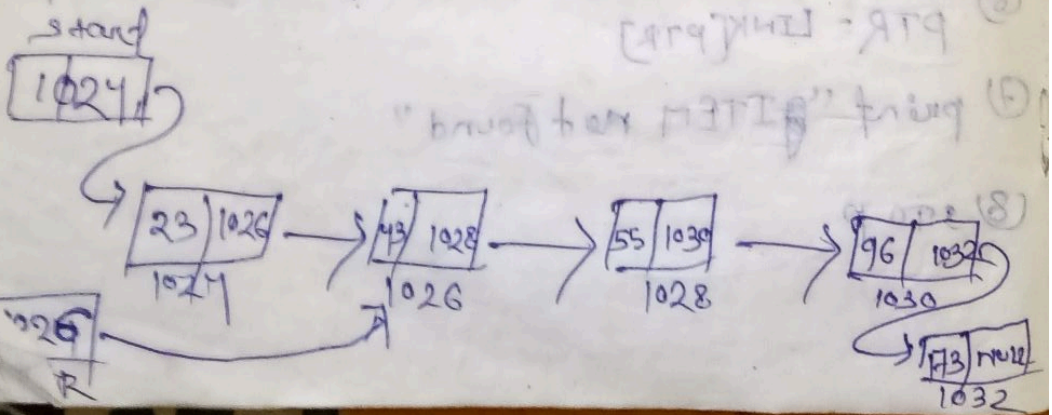
96 ≠ 23 (True)

23 == 96 (False)

PTR = LINK[PTR]

= LINK[1024]

= 1026



DATA = 96

PTR = START = 1024

INFO [PTR] = 43

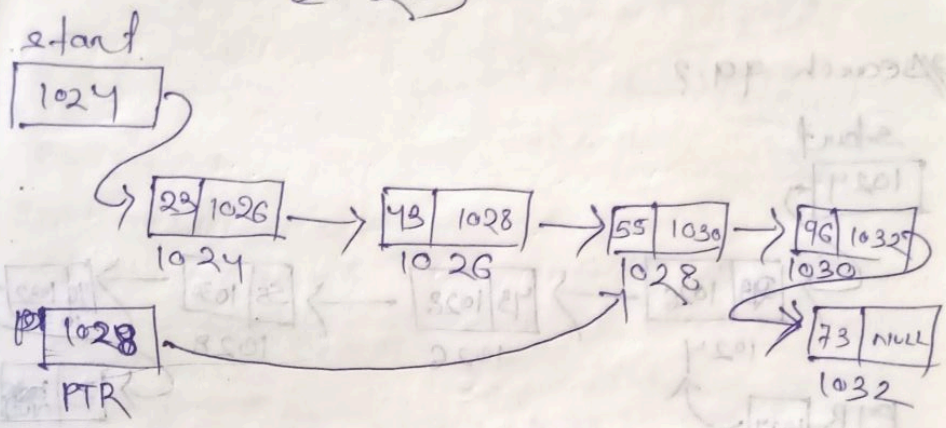
96 ≠ 43 (true)

43 = = 96 (false)

PTR = LINK [PTR]

= LINK [1026]

= 1028



DATA = 96

PTR = START = 1024

INFO [PTR] = 55

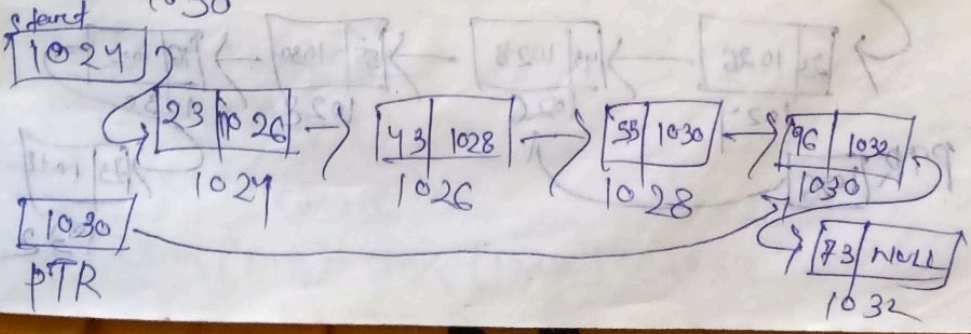
96 ≠ 55 (true)

55 = = 96 (false)

PTR = LINK [PTR]

= LINK [1028]

= 1030



~~search 99~~ ?

DATA = 96

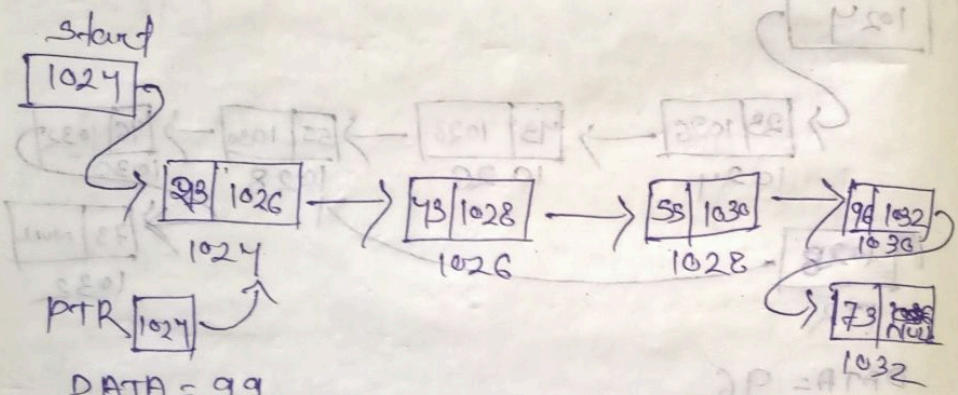
PTR = START = 1024

INFO [~~PTR~~ LINK] = 96

96 = 96

'search success'

search 99 ?



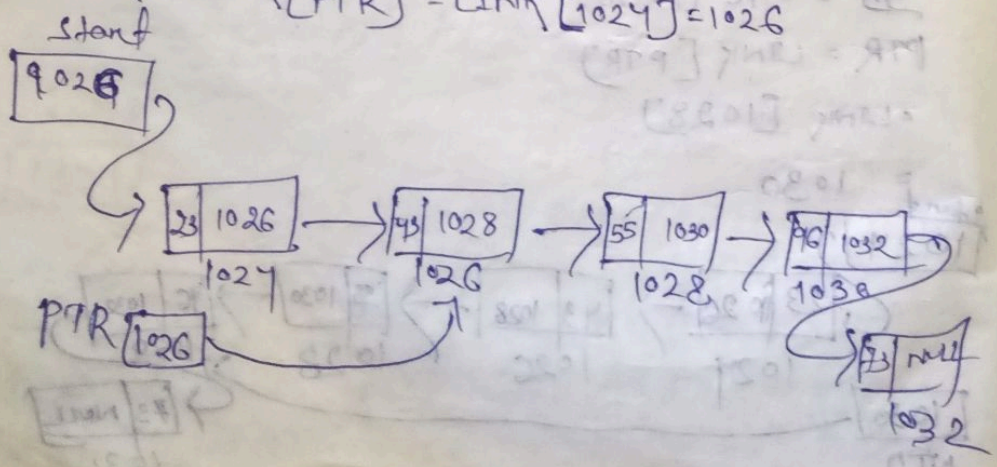
DATA = 99

INFO[PTR] = 23

23 ≠ 99 (true)

23 == 99 (false)

PTR = LINK[PTR] = LINK[1024] = 1026



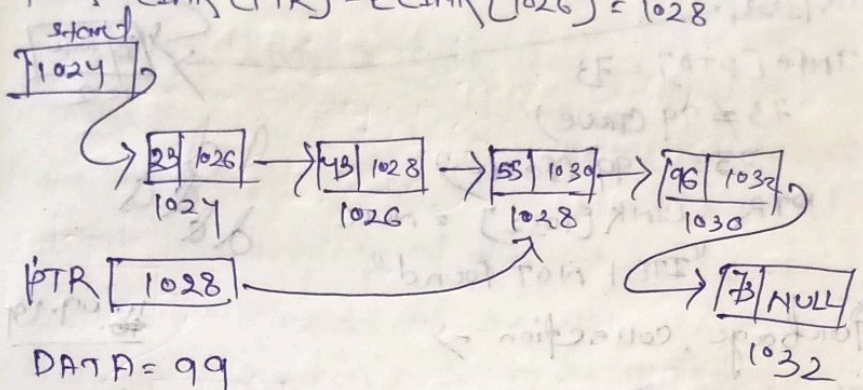
DATA = 99

INFO[PTR] = 43

43 ≠ 99 (true)

43 = = 99 (false)

PTR = LINK[PTR] = LINK[1026] = 1028



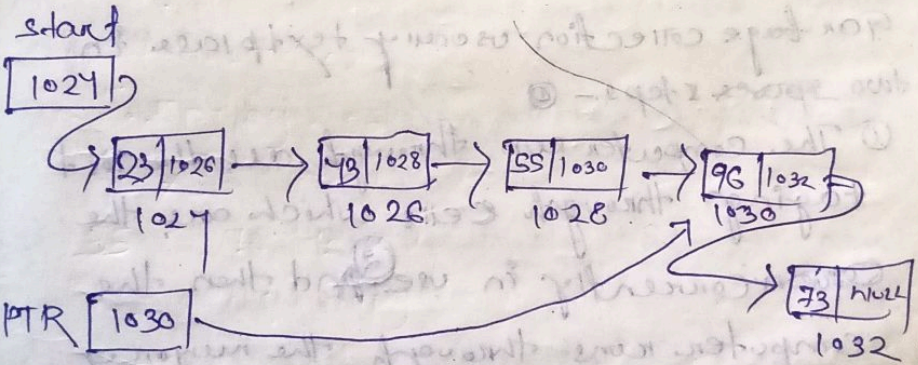
DATA = 99

INFO[PTR] = 55

55 ≠ 99 (true)

55 = = 99 (false)

PTR = LINK[PTR] = LINK[1028] = 1030



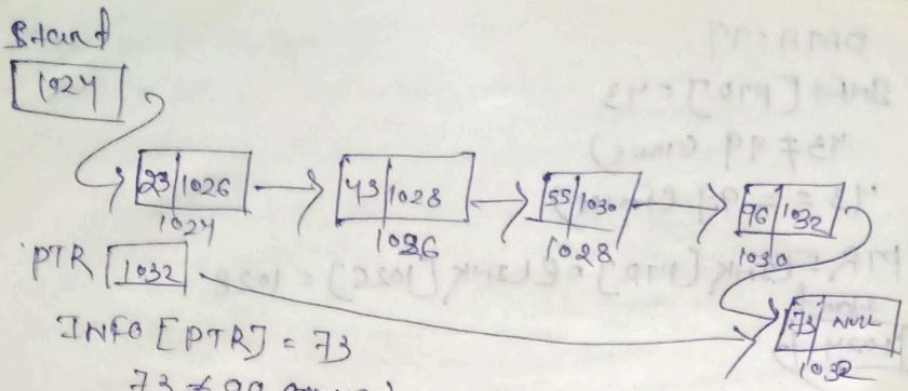
DATA = 99

INFO[PTR] = 96

96 ≠ 99 (true)

96 = = 99 (false)

PTR = LINK[PTR] = LINK[1030] = 1032



INFO[PTR] = 73

73 ≠ 99 (True)

73 = = 99 (False)

PTR = LINK[1032] = NULL

"ITEM NOT FOUND"

Inp

Garbage collection →

11.09.19

→ The operating system of a computer may periodically collect all the deleted spaces onto the free storage list. Any technique which does this collection is known as garbage collection.

→ Garbage collection usually takes place in two steps. -

① The computer run through all the list tagging through cells which are currently in use and then the computer runs through the memory collecting all the ~~on~~ tag space into the free storage in the list

③ The garbage collection make take place when there is sum amount of space on

no space actual left in the free storage list. when the CPU is 'IDLE' it has to do the collection.

(4) The garbage collection is not visible to the programmer.

Insertion into a Link list →

→ Insertion into a link list can be done in three ways -

1. At the beging of the link list.
2. At the end of the link list.
3. After a given node in the linked list.

(1) Insertion at begning of the linked list →

1) 'INFO' points to the information part of a given node.

2) 'LINK' points to the address part of a given node.

3) 'START' points to the address of the first node of the link list.

(4) 'AVAIL' is the available space for new memory allocation.

(5) 'ITEM' is the new element to be inserted into the link list.

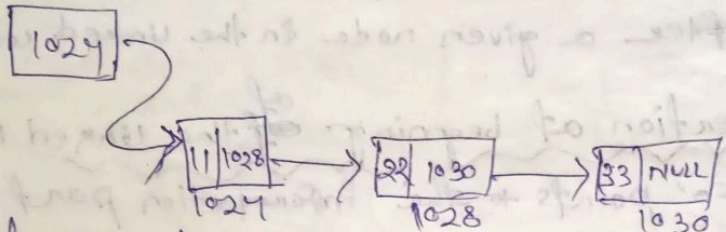
(6) 'NEW' is the new node to be created.

Algorithm

- INSERT (INFO, LINK, AVAIL, START, ITEM)

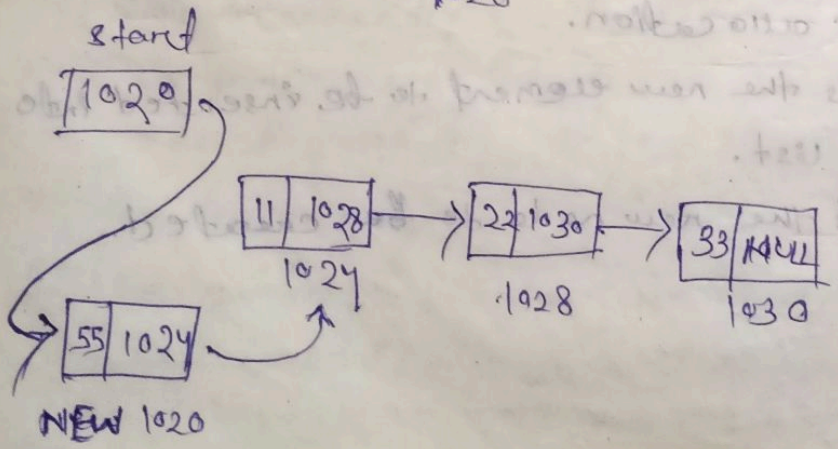
(1) Start
(2) If AVAIL = NULL then "overflow" and stop.
(3) set NEW = AVAIL and AVAIL = LINK
(4) set NEW [INFO] = ITEM
(5) set NEW [LINK] = START
(6) Set START = NEW
(7) stop

Ex → start



Insert 55 at the begning of the link list given

Start NEW [INFO] = ITEM
= 55
NEW [LINK] = start
= 1024
start = new
= 1020



14.09.19

Insertion at end of the linked list

→ 'INFO' points to the information part of a given node.

→ 'LINK' points to the address part of a given node.

→ 'START' points to the address of first node of the linked list.

→ 'PTR' points to a pointer variable.

→ 'AVAIL' is the available space for new memory allocation.

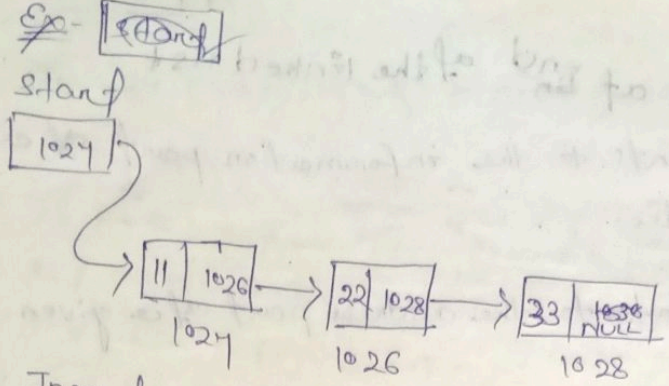
→ 'ITEM' is the new element to be inserted into the linked list.

→ 'NEW' is the new node to be created.

INSERT (INFO, LINK, AVAIL, START, ITEM)

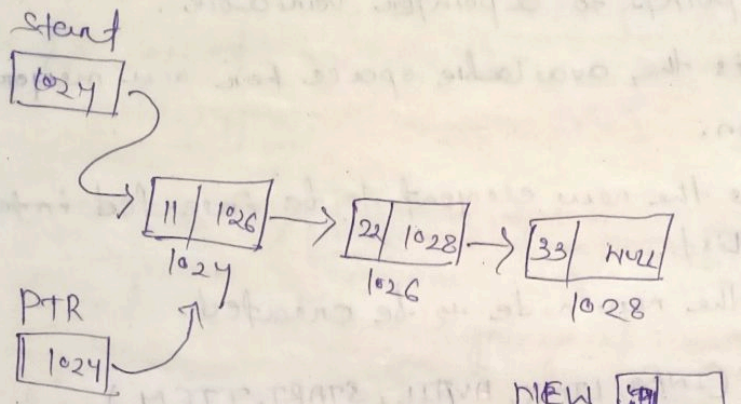
1) Start
2) IF AVAIL = NULL, then write "overflow" and stop.
3) set NEW = AVAIL and AVAIL = LINK
4) set NEW [INFO] = ITEM
5) set PTR = START
6) Repeat step 7 while PTR ≠ NULL
7) PTR = LINK [PTR]
8) set NEW [LINK] = NULL
9) set PTR [LINK] = NEW
10) stop

Ex- ~~START~~



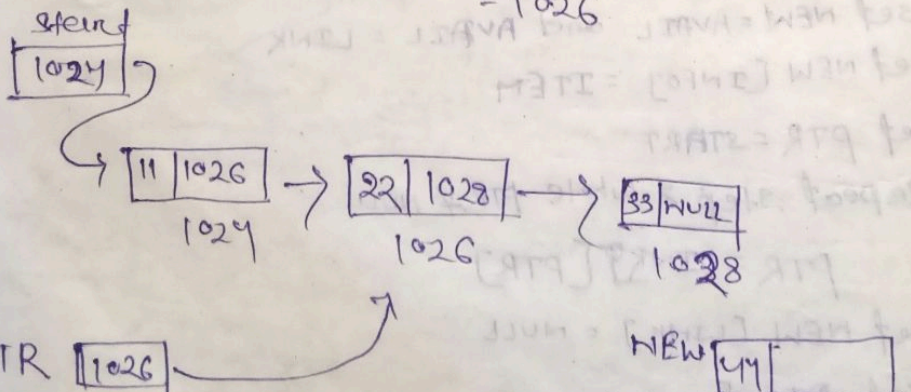
Insert 44 at the end of the given link list ?

Soln



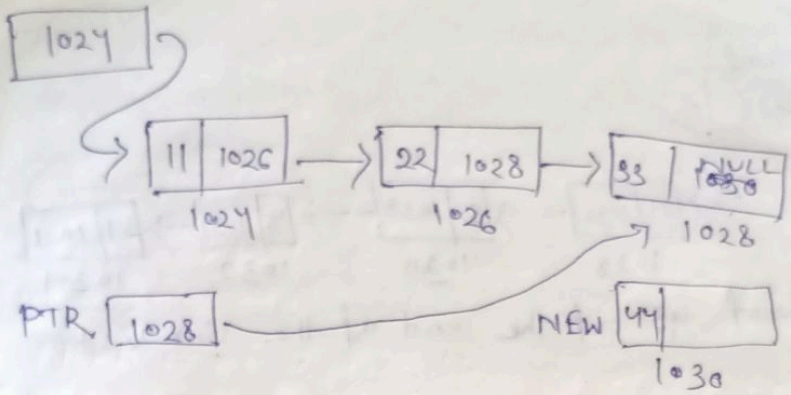
PTR = ~~LINK~~ LINK [1024]

= 1026



PTR = LINK [1026]

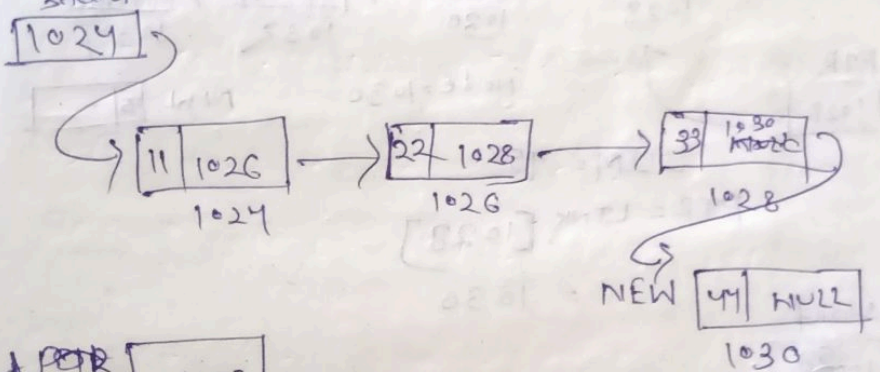
= 1028



~~PTR = ITEM [~~

PTR = LINK [1028]

= ~~1030~~ NULL

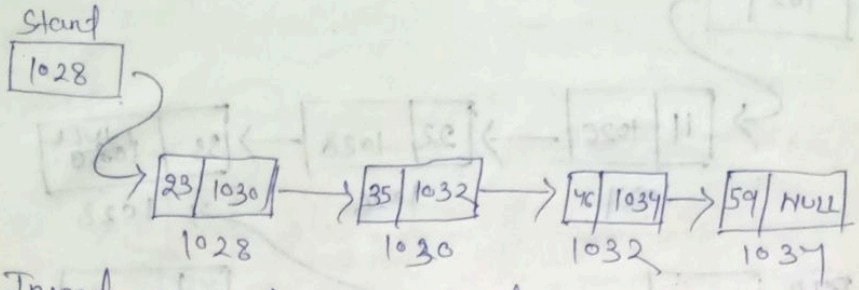


~~PTR~~ ~~1028~~

NEW [LINK] = NULL

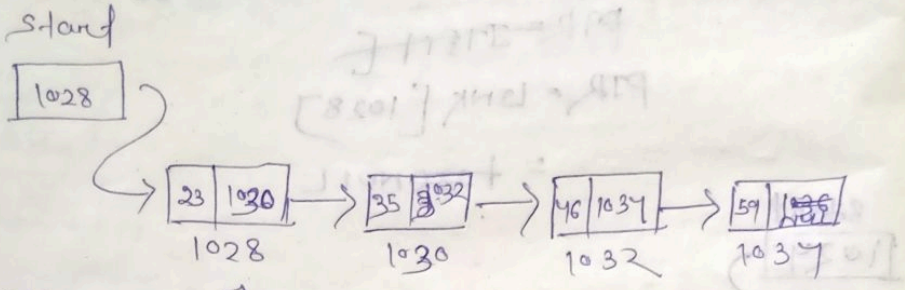
1028 [LINK] = 1030

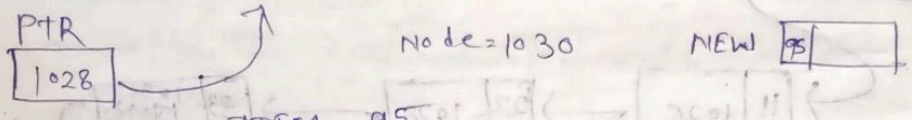
Ques 3



Insert 95 at the end of the link list.

Soln

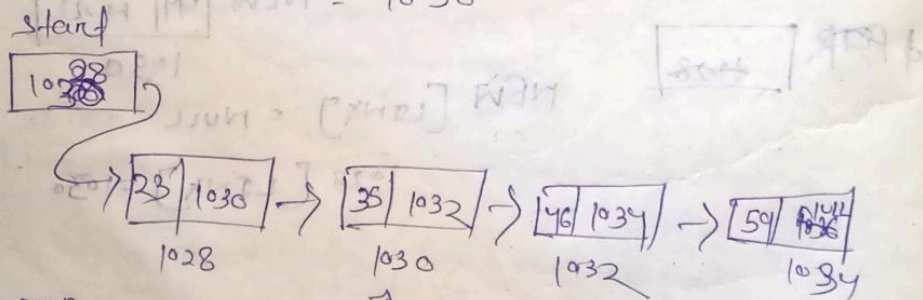


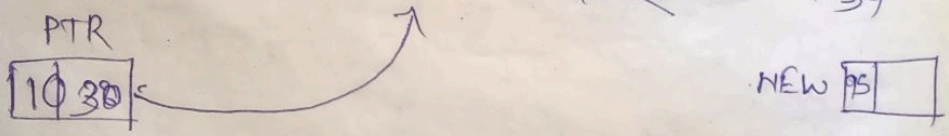


ITEM = 95

PTR = LINK [1028]

= 1030

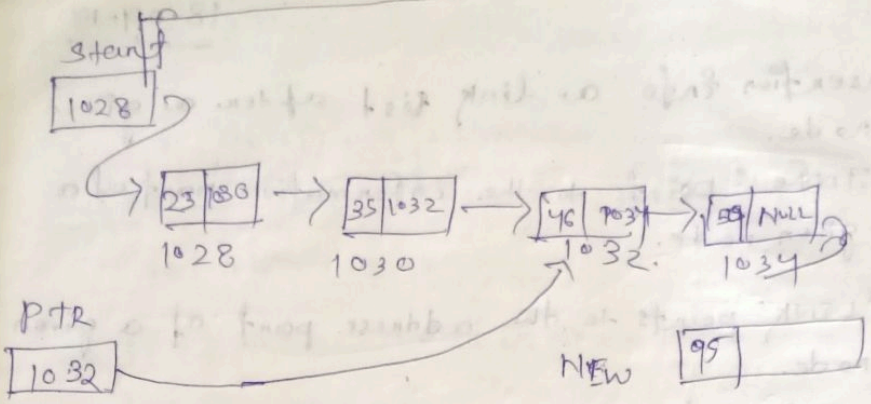




ITEM = 95

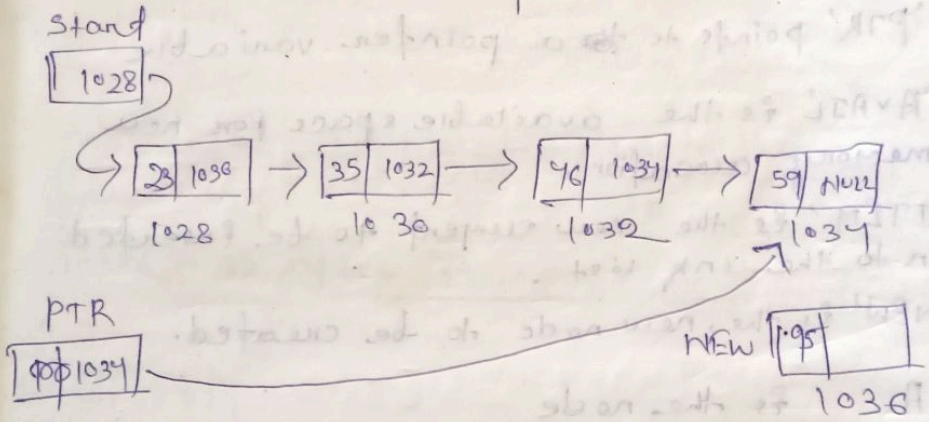
PTR = LINK [1030]

= 1032



PTR = LINK [1032]

= 1034



PTR = LINK [1034]

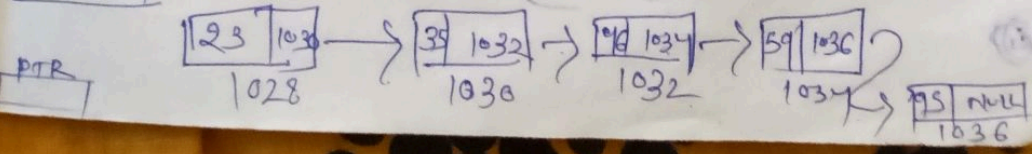
= NULL

NEW [LINK] = NULL

⇒ 1036 [LINK] = NULL

* PTR [LINK] = NEW

= 1034 [LINK] = 1036



18.09.19

Insertion into a link list after a given node.

→ "INFO" points to the information part of a given node.

→ 'LINK' points to the address part of a given node.

→ 'START' points to the address of first node of the LINK list.

→ 'PTR' points to a pointer variable

→ 'AVAIL' is the available space for new memory allocation.

→ 'ITEM' is the new element to be inserted in to the link list.

→ 'NEW' is the new node to be created.

→ A is the node

INSERT (INFO, LINK, AVAIL, START, ITEM)

1) START
2) If AVAIL = NULL then write "over flow" and exit.
3) set NEW = AVAIL and AVAIL = LINK
4) set NEW [INFO] = ITEM
5) set PTR = START
6) Reapeat step 7 while INFO [PTR] ≠ LOC
7) PTR = LINK [PTR]
8)

8) If LOC = NULL

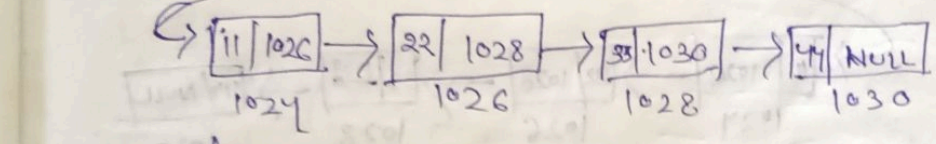
Then
set NEW[LINK] = START
and START = NEW

else
set NEW [LINK]
= LOC [LINK]

and
LOC [LINK] = NEW

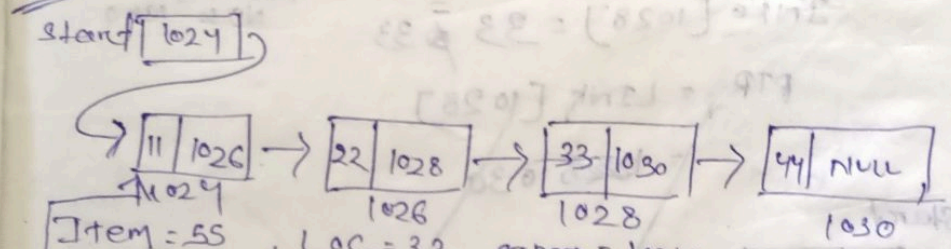
9) STOP

Ex
Start 1024



Insert 55 after 33

Soln



Item = 55, LOC = 33 START = 1024

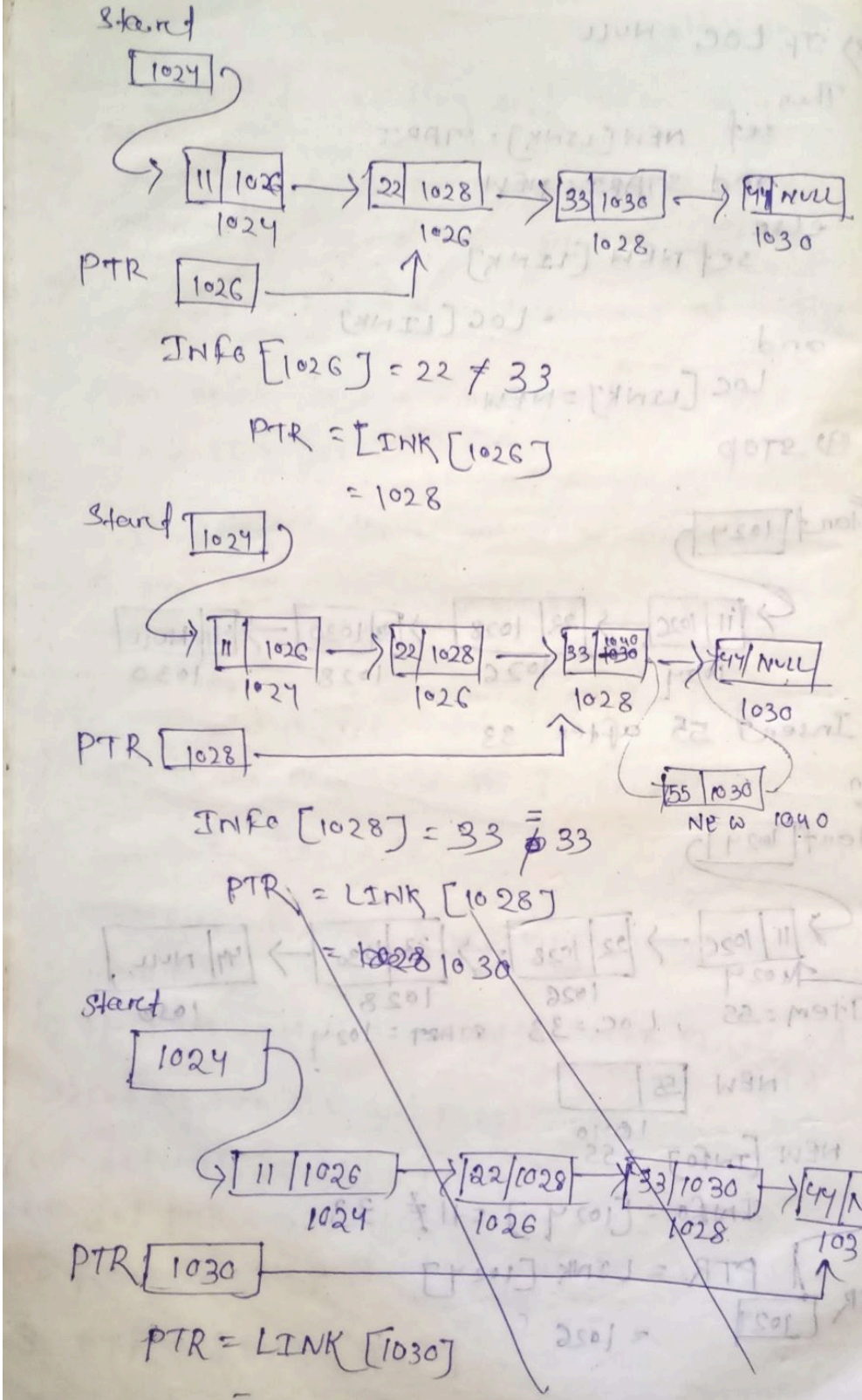
NEW 55 | 1040

NEW [INFO] = 55

INFO = [1024] = 11 ≠ 33

PTR = LINK [1024]

PTR 1024 = 1026


Start
1024
11 1026
1024
22 1028
1026
33 1030
1028
44 NULL
1030
PTR
1026
INFO [1026] = 22 ≠ 33
PTR = LINK [1026]
= 1028
Start
1024
11 1026
1024
22 1028
1026
33 1040 1030
1028
44 NULL
1030
PTR
1028
55 1030
New 1040
INFO [1028] = 33 = 33
PTR = LINK [1028]
= 1030
Start
1024
11 1026
1024
22 1028
1026
33 1030
1028
44 NULL
1030
PTR
1030
PTR = LINK [1030]
=

If LOC ≠ NULL

~~else~~

set NEW [LINK]
= LOC [LINK]

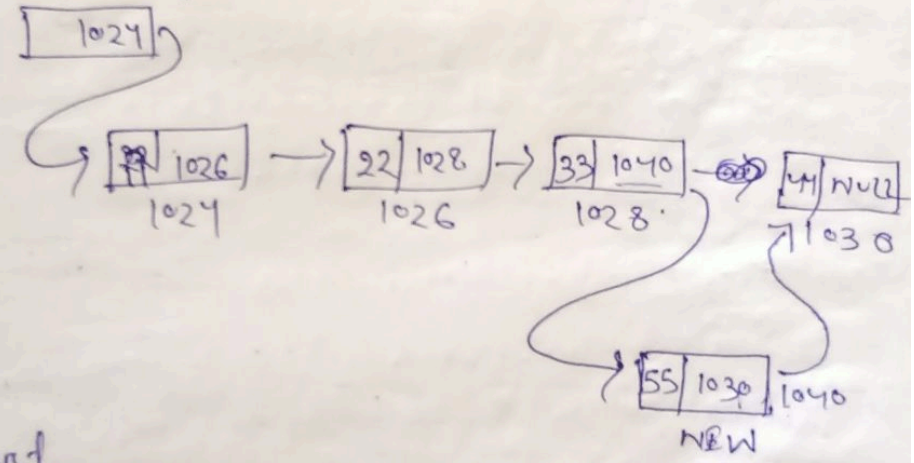
and LOC [LINK] = NEW

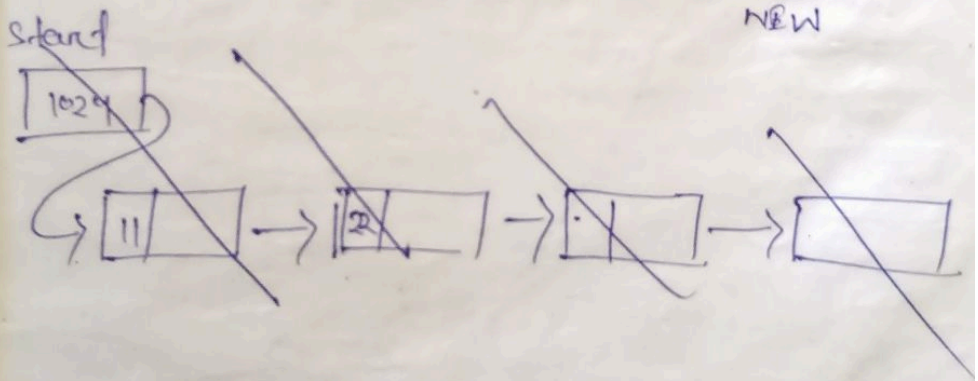
set NEW = [1040]
= 33 [1030]

and

33 [1030] = 55

~~start~~ start



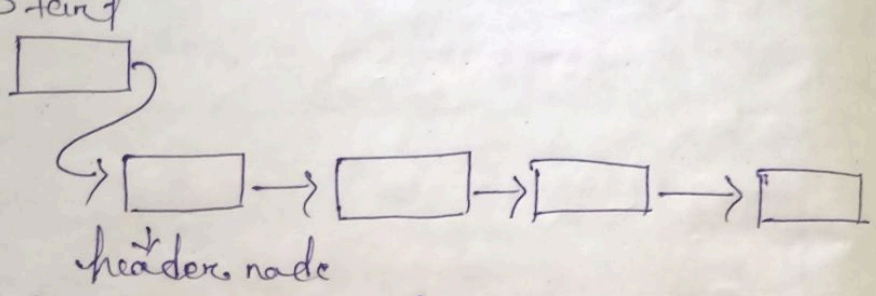


Header Linked List

20.09.19

→ A header linked list is a linked list, which always contains a special node called as the header node at the beginning of the list.

Ex →



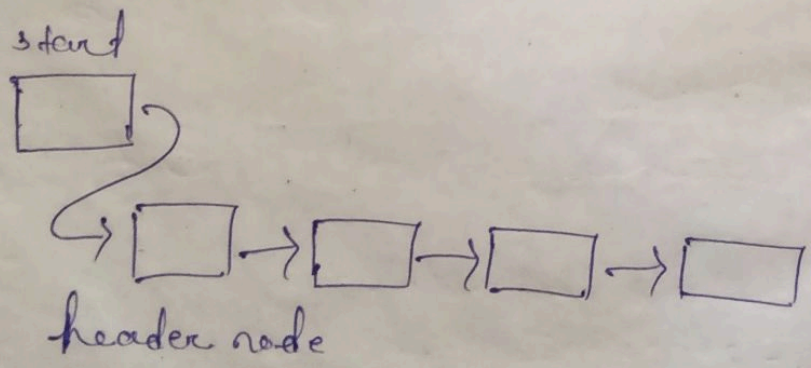
→ There are two types of header linked list →

(1) ~~Ground~~ Grounded header linked list

(2) Circular header linked list.

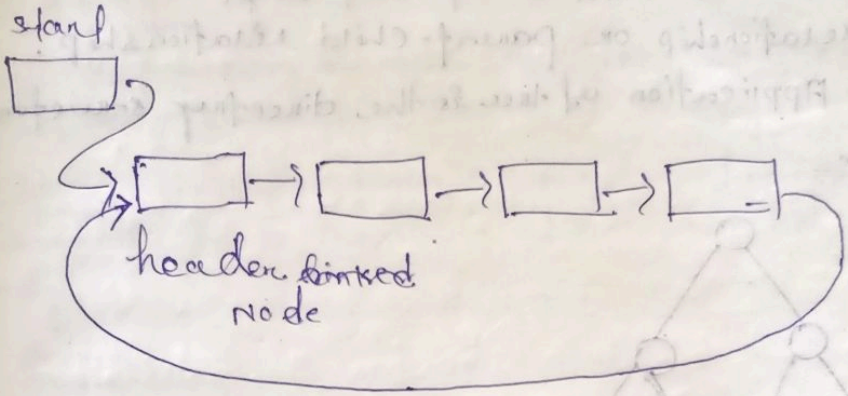
1) Grounder header linked list →

→ A grounder header linked list is a header linked list where the last node contains no address or NULL.



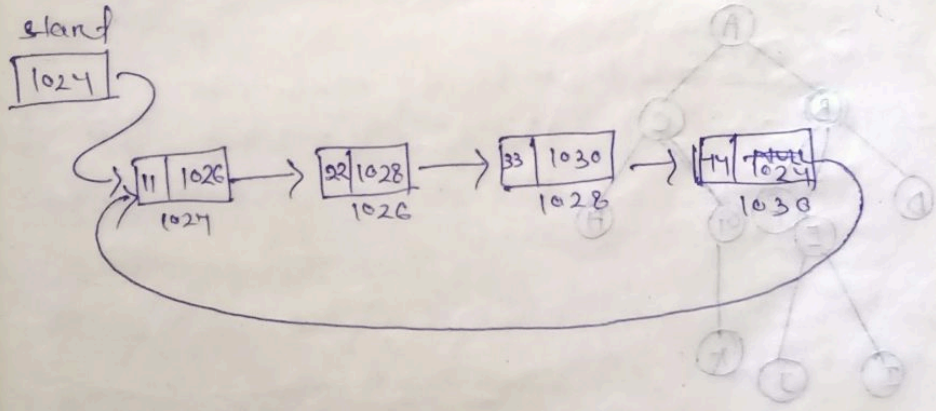
(2) Circular header linked list →

→ A circular header linked list is a header list where the last node points back to the header node.



Circular linked list →

→ If the last node of the linked list contains the address of the first node then this type of list is known as circular linked list.



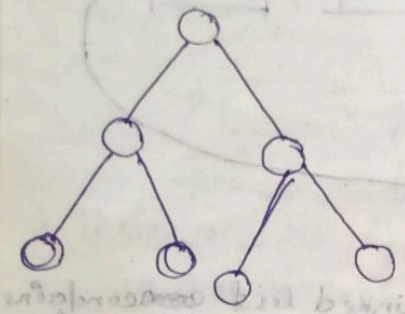
Chapter - 6

TREE

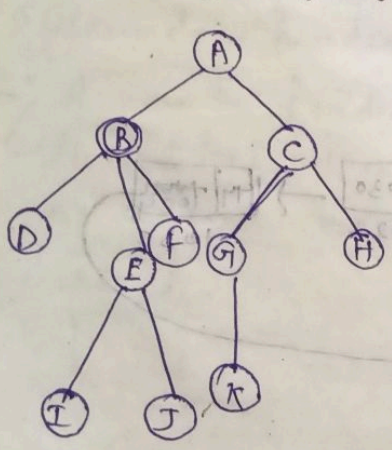
Tree → Tree is a non-linear data structure which is used to represent high relationship or parent-child relationship.

→ Application of tree is the directory structure.

Ex



→ **Basic terminology used in a tree** →



In a tree data structure we will be using the following terminologies -

Root

→ The first node in a tree data structure is called as the root node.

→ Root node is the origin of tree data structure and there must be only one root node.

→ Here 'A' is the root node.

2) Edge →

→ In a tree data structure, the connecting links bet^n any 2 nodes is called as an edge.

→ In a tree with "n" number of nodes, there will be a maximum of "n-1" number of edges.

3) Parent →

→ In a tree data structure, the node which is the predecessor of any node is called as the parent node.

→ Here 'A', B, C, D, E, F & G are the parent nodes.

4) Child →

→ In a tree data structure, the node which is the successor of any node is called as the child node.

→ In a tree data structure, all the nodes except the root node are the child nodes.

→ Here B, C, D, E, F & G are the child nodes.

5) Siblings →

→ In a Tree data structure, nodes which belong to the same parent are called as siblings.

→ Here the siblings are B-C, D-E-F, G-H, I-J.

6) LEAF →

→ In a tree data structure the node which donot have a child node is known as leaf node.

→ In a tree data structure, the leaf nodes are also known as external nodes or terminal nodes.

→ Here the leaf nodes are 'D, I, J, K, H, F'.

7) Internal nodes →

→ In a tree data structure the nodes which are atleast one child is known as internal node.

→ Nodes other than the leaf nodes will be called as internal nodes or non-terminal nodes.

→ Here the internal nodes are A, B, C, E & G.

8) Degree →

→ In a tree data structure, the total number of children of a the node is called as degree of that node.

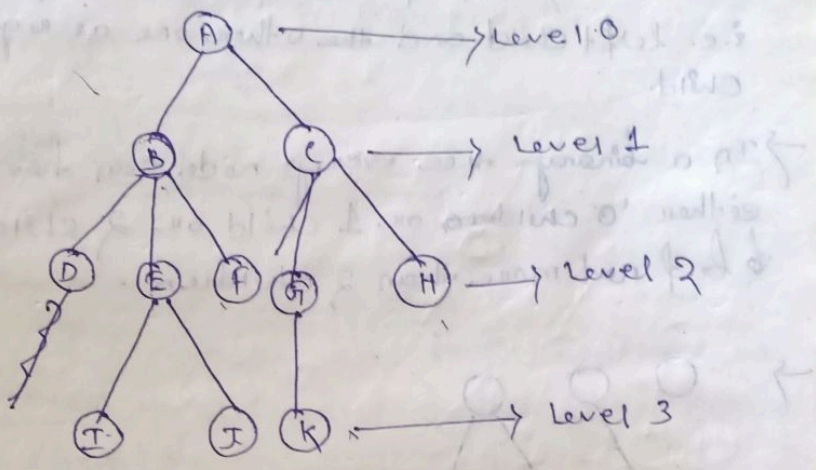
→ Here degree of A = 2
B = 3
C = 2
E = 2
G = 1

9) LEVEL →

→ In a tree data structure, the root node is said to be at level '0' and the child nodes of the root node are at level 1.

→ In a tree, each step from top to bottom is called as a level and level count starts with '0' and incremented by each level.

→



10) Hight :

→ In a tree data structure the total number of

is called as hight of

→ Here the hight of a will be 3, 'B' will be '2' hight of I, J, K will be '0'.

11) path → Each node has to be reached through a unique sequence of edges called as path.

12) Branch →

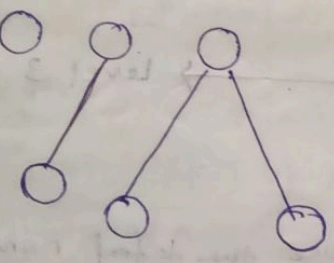
→ A path ending in a leaf node is called as a branch.

27.09.19

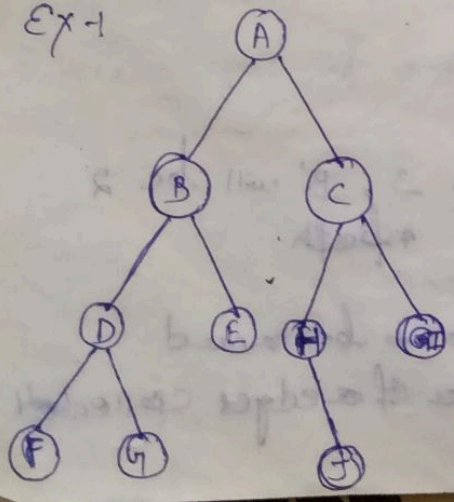
Binary tree :

→ Binary tree is a special type of tree data structure in which every node can have a maximum of two children i.e. left child and the other one as right child.

→ In a binary tree every node can have either '0' children or 1 child or 2 children, but not more than 2 children.

→ 

Ex-1



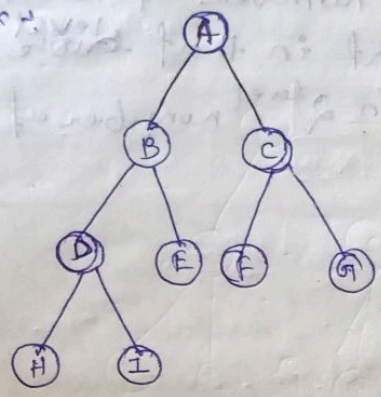
Types of Binary tree →

1) Strictly Binary tree →

→ A Binary tree in which every node has either 2 or '0' number of children is called as a strictly binary tree.

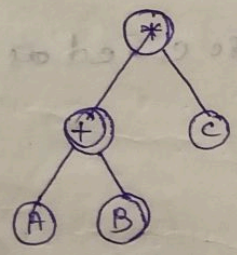
→ Strictly binary tree is also called full binary tree or proper binary tree or 2-tree.

→ Ex:-

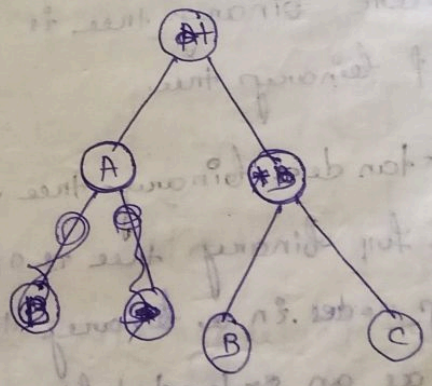


→ strictly binary tree data structure is also used to represent mathematical expressions.

→ (A+B)*C



A+B*C



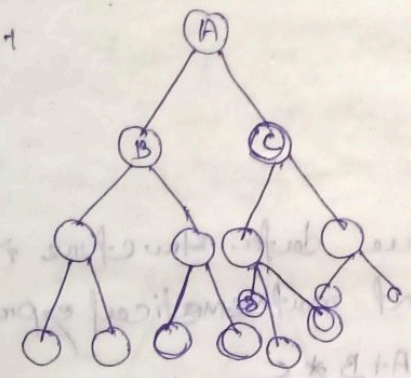
2) Complete binary tree →

→ A binary tree in which every internal node has exactly 2 children and all the leave nodes are at the same level then it is called as complete binary tree.

→ At every level of the complete binary tree there the must be full number of nodes that should be present in that level

→ Each level will contain 2^level number of nodes.

Ex →

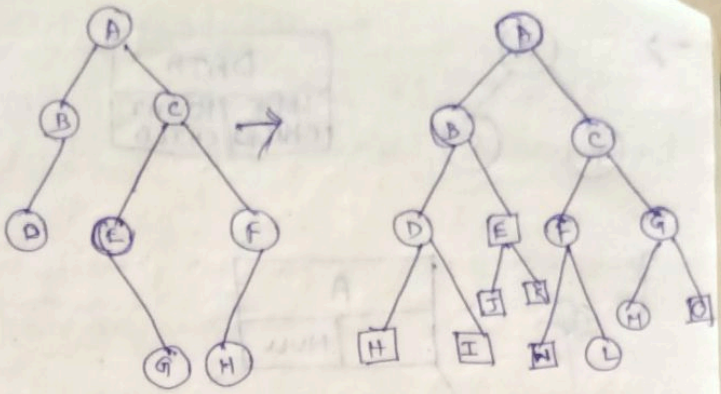


→ Complete binary tree is also called as perfect binary tree.

3) Extended binary tree →

→ If a full binary tree is extend by adding dummy nodes in a binary tree then it is called as an extended binary tree.

→ Ex 1



28.09.19

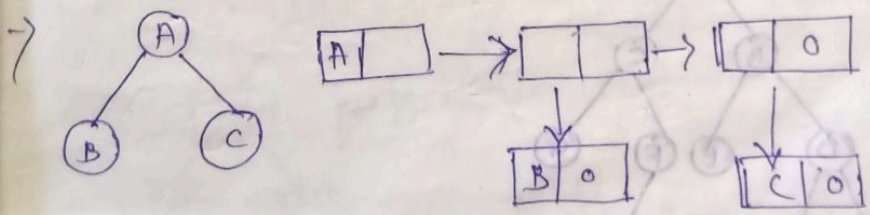
Tree representation :-

→ A Tree data structure can be represented in 2 ways describe as below.

1) List representation
2) Left child and right child representation.

1) List representation :-

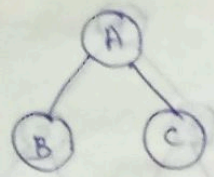
→ In this representation we used two types of nodes one for representing a node with data and another for representing references.

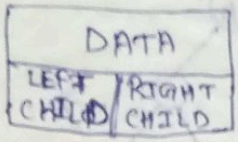


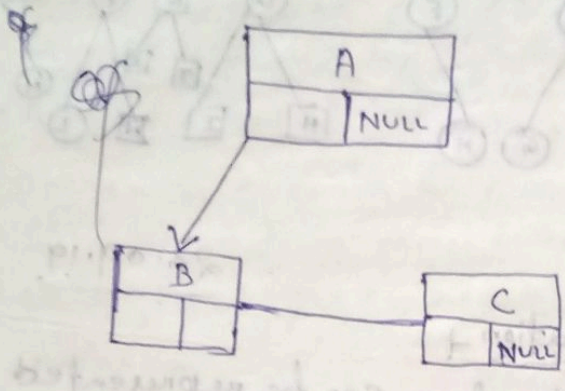
2) Left and right child representation →

→ In this type of representation we use a list with one type of node which consists of three fi namely data field, left child, and right child

→





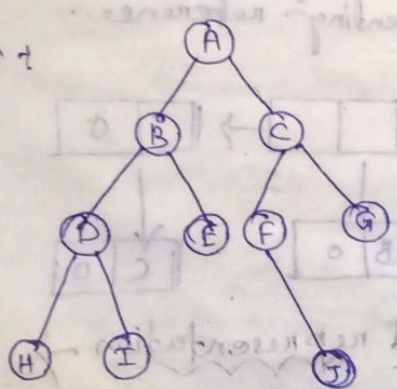


Binary tree representation →

→ A binary tree data structure is represented by using 2 methods which are described as follows -

1) Array representation
2) Link list representation.

→ Ex :-

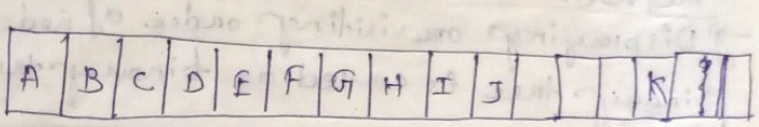


Array representation →

→ In array representation of binary tree we used a 1(d) array to represent a binary tree.

→ Considering the above example of binary tree it can be represented as -

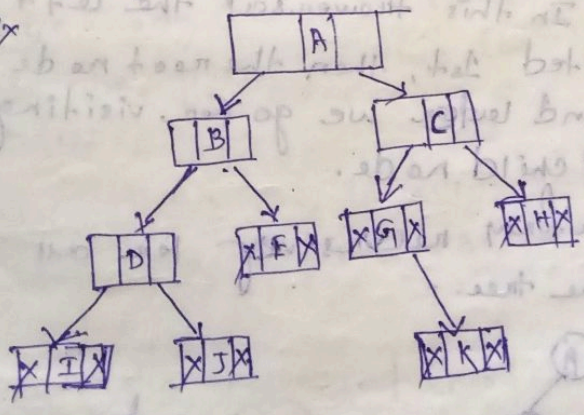
→ Ex:

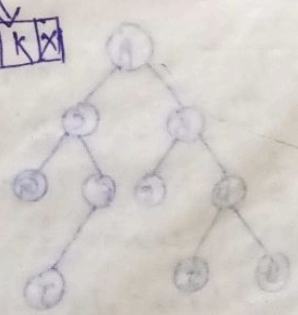


LINK LIST representation →

→ we use a double ~~link~~ list to represent a binary tree. In double link list every node consists of 3 fields i.e. 1st field for storing left child address, 2nd field for storing actual data and 3rd field for storing right child address.

→ Ex





Binary tree traversal →

→ When we want to display a binary tree we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying in the order of the nodes depends on the traversal method.

→ Displaying or visiting order of nodes in a binary tree is called as binary tree traversal.

→ There are three types of binary tree traversal -

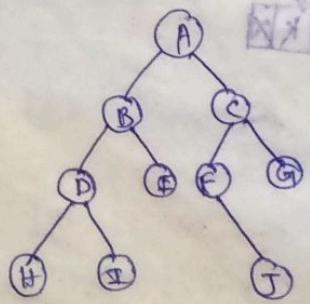
1. Inorder traversal
2. preorder traversal
3. postorder traversal

In order traversal → (Left - root - right)

→ In order traversal the root node is visited between the left child & right child. In this traversal the left node is visited 1st, then the root node is visited and later we go for visiting the right child node.

→ This is performed recursively for all nodes in the tree.

→ Ex:



→ In above example of binary tree, 1st we will try to visit the left child node of root node 'A'. But as the left child is a root node for left sub-tree, so we try to visit left child of 'b' i.e. 'D', which is again a root node for 'H & I'. So now we will visit its left child 'H', then its root node 'D' then its right node 'I'. Now we have completed the left part of node 'B' so now we can visit node 'B' and then its right child node 'E'.

→ with this we have completed the left part of root node 'A'. Now we can visit the root node A and then the right child node of 'A' i.e. 'C' But 'C' is the root node of 'F' & G. So now we need to visit the sub-tree of 'F' i.e. 'F' will be visited 1st and then its right child 'J'. Then we can visit node 'C' and lastly the right child node of 'C' i.e. node G. As 'G' is the right most node of the given tree we can stop the process now.

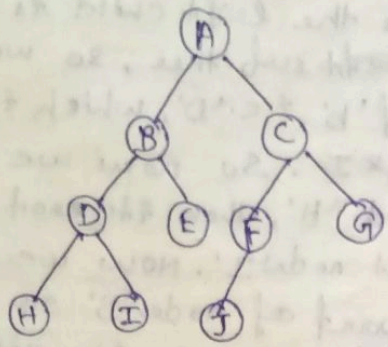
So the inorder traversal for the above example → H-D-I-B-E-A-F-J-C-G.

30.09.19

Pre-order traversal (Root-left-right)

→ In pre-order traversal, the root node is visited before the left child and right child node. It means in this type of traversal, the root node is visited first, Then its left child is

visited and later its right child is visited.



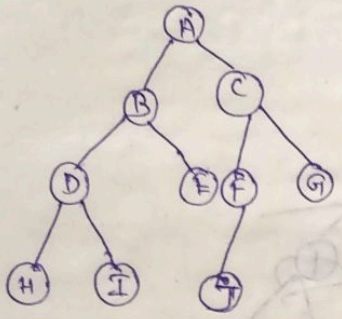
In the above example of binary tree, as per the condition, first we will visit root node 'A'. Then we are suppose the visit the left child of 'A' i,e. Node 'B' is again a root node for D & E so now we shall visit node 'B'. Then we will visit its left child 'D', which is again a root node for H & I. So first we will visit node 'D', then its left child node 'H' and then its right child node 'I'. With this we have completed visiting the left child nodes of 'B'. So Now we will visit the right child of node 'B' i,e. node 'E'. Now we have completed visiting all the nodes in left part 'A'. So now we will visit the right child of 'A' i,e node 'C'. But 'C' is again root node for nodes F & G. So now we can visit the left child node of 'C' i,e node 'F'. But 'F' is again root node for node 'J'. So now we can visit node 'J'. Now as there is no right child of 'F' we can now visit the right child node of node 'A' i,e. node 'G'

with this we have completed the traversal of the given binary tree. so the pre-order traversal of the above example is - A-B- D-H-I-E-C-F-J-G

post-order traversal : (left-right-root).

In post order traversal the root node is visited after the left child and right child. It means that first the left child node is visited, then the right child node is visited and then its root node is visited.

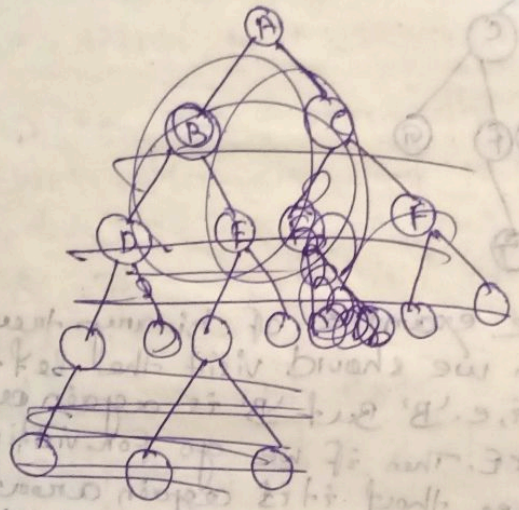
→

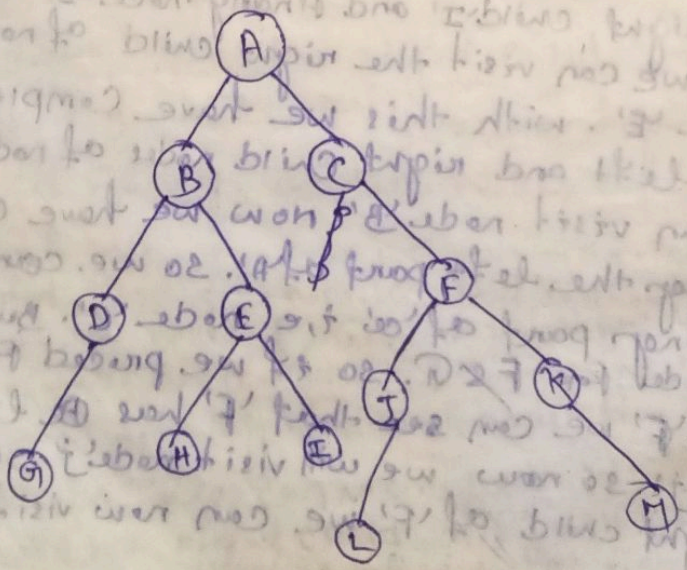


In the above example of binary tree, As per the condition we should visit the left child of node 'A' 1st i,e. 'B' But 'B' is again a root node for nodes 'D' & E. Then if we go for visiting node 'D' we can see that it is again a root node for H & I. So now we visit its left child 'H' 1st, Then its right child 'I' and finally node 'D' is visited. now we can visit the right child of node 'B' i,e. node 'E'. with this we have completed visiting the left and right child nodes of node 'B' so now we can visit node 'B'. now we have completed visiting the left part of 'A'. so we can proceed for right part of 'A' i,e. node 'C'. But 'C' is again root node for F & G. so if we proceed for its left child 'F' we can see that 'F' has a left child node 'J' so now we will visit node 'J'. As there is no right child of 'F' we can now visit node 'F'.

with this we have completed visiting all the left child node of 'C'. So now we can visit its right child 'F'. And then we can visit node 'C'. with this we completed visiting all the right child node of root node 'A'. So now we can visit root node 'A'. and the process ends there. so the post-order traversal of the above example is - H-I-D-E-B-J-K-G-L-C-A.

Q1





In-order traversal

G-D-B-E-H-E-I-A-C-L-J-F-

pre-order traversal:-

A-B-D-E-H-I-C-F-J-L-K-M

post-order traversal →

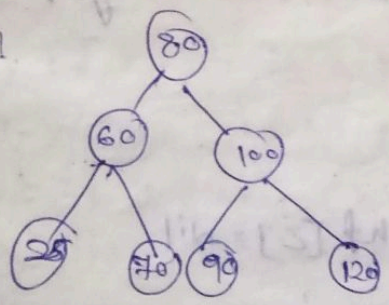
G-D-H-I-E-B-L-J-M-K-F-C-A

Binary search-tree

A binary search tree is a binary tree which satisfy the following properties:-

1) Every element in the left sub-tree are smaller than the root element.
2) Every element in the right sub-tree are largeer than the root element.
3) The left and right sub-tree are also binary search tree.

Ex-1



18.10.19

→ The sequence of steps to be followed in performing the insertion operation on a binary such tree are as follows.

→ 1) Start from the root node.

* If the data to be inserted compaire the value with the value of the root node.

(a) If their equal just skipt because this value already exits.

(b) If their different and if the data to be inserted if less than the value of the root node, choose the left subtree.

(c) If the data to be inserted is the value of root node, choose the right sub-tree.

3) Repeat step-2 until the node is encountered where the data has to be inserted.

Algorithm :-

To insert a new value 'w', into a binary search tree we use the following procedure

TREE-INSERT (x)

Let key[z] = w

left[z] = Nil and right[z] = Nil

1) y ← Nil

2) x ← root [T]
3) while x ≠ NULL
4) do y ← x
5) if key [z] < key [x]
6) then x ← left [x]
7) else x ← right [x]
8) P[z] = y
9) if y = Nil
10) then root [T] = z
11) else if key [z] < key [y]
12) then left [y] ← z
13) else right [y] ← z

searching in binary such tree

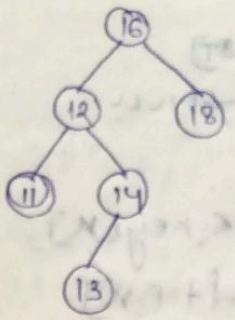
The most common operation performed on a binary search tree if searching for a key stored in the tree.

Algorithm

The TREE-SEARCH(x, k) algorithm searches the tree root at 'x' for a node push key value is equal to 'k'. It return a pointer to the node if it exists otherwise NULL.

1) if x = Nil or k = key [x]
2) then return x
3) if k < key [x]
4) then return TREE-SEARCH (left [x], k)
5) else return TREE-SEARCH (right [x], k)

ex←



x → root
k ← search number

Search 14 in the given binary in the tree.

TREE-SEARCH (16, 14)

$x \neq$ NULL

$14 < 16$ (True)

~~Then~~ TREE-SEARCH (12, 14)

$x \neq$ Nill

~~14 > 12~~ 14 > 12 (False)

TREE-SEARCH (14, 14)

$14 == 14$

Return 14

Deletion operation in binary search tree.

→ In a binary search tree, the deletion operation is performed with O(log n) time complexity.

→ Deleting a node from a binary search tree involves the following cases-

1) Deleting a leaf node.
2) Deleting a node with one child
3) Deleting a node with two children.

Case-1 → Deleting a leaf node

→ we use the following steps to delete a leaf node from a binary search tree -

step-1 : find the node to be deleted by using search operation.

step-2 : Delete the node by using free ( ) function.

Case-2 → Deleting a node with one child

→ we use the following steps to delete a node with one child from a binary search tree -

step-1 : find the node to be deleted by using search operation.

step-2 : If it has only one child then create a link b/n its parent node and the child node.

step-3 : Delete the node by using free ( ).

Case-3 → Deleting a node with two children

→ we use the following step to delete a node with two children from a value of binary search tree.

Step-1 :

Step-2 : If it has two children then find the largest node in its left sub-tree or smallest nodes in its right sub-tree.

Step-3 : Swap (exchange) the both, i.e. the deleting node and the node found in the above step.

Step-4 : Then check, weather the deleting node comes to case-1 or case-2 else go to step #2.

Step-5 : If it comes to case-1, Then delete by using case-1 logic.

Step-6 : If it comes to case-2, Then delete by using case two logic.

Step-7 : Repeat the ~~sem~~ same process until the node is deleted from the tree.

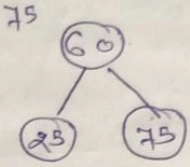
Q1 Construct a binary search tree by inserting the following elements.

60, 25, 75, 15, 50, 66, 33, 44

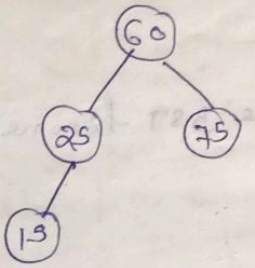
Insert 60

Insert 25

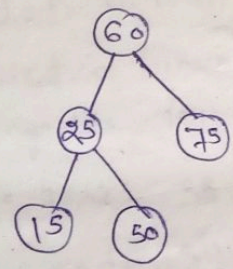
Insert 75



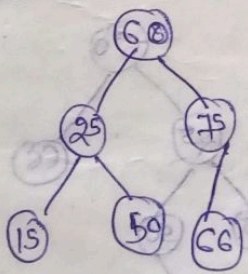
Insert 15



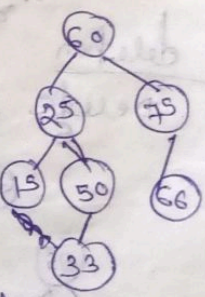
Insert 50



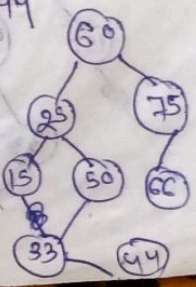
Insert 66



Insert 33

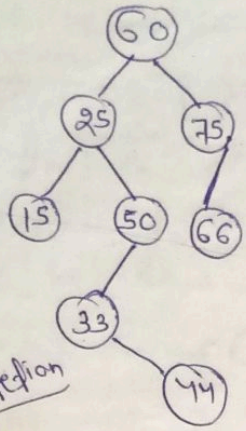


Insert 44



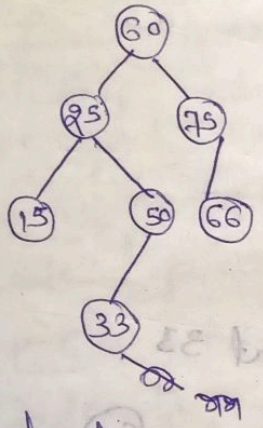
19.10.19

Delete node 44, 75, & 25



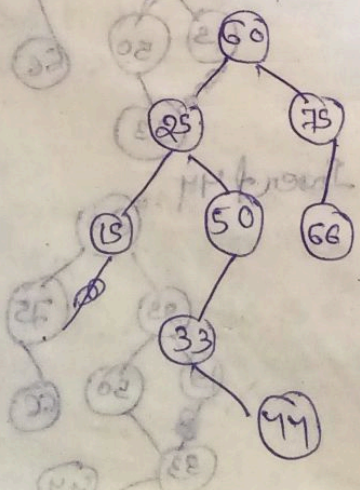
1st deletion

after deleting 44 the BST becomes

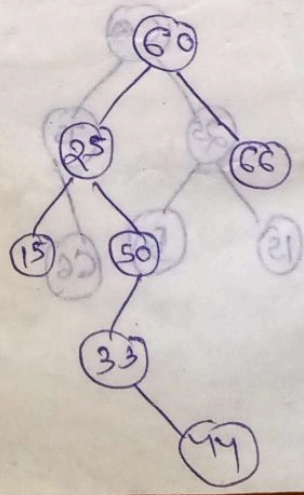


2nd deletion

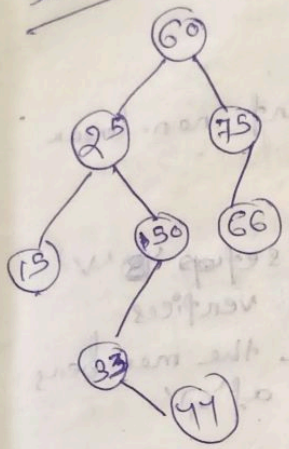
Delete 75



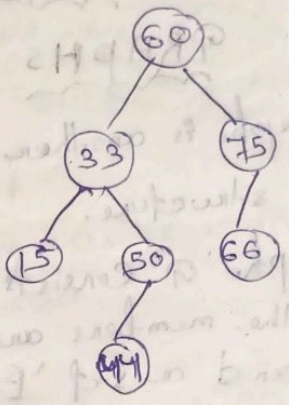
after delete 75



3rd deletion



after delete 25



Assignment-

(1) Construct a binary search tree by inserting the following elements -

J, R, D, T, G, E, A, M, H, S, Q, U, B

2 Construct a binary search tree by inserting the following elements

40, 25, 70, 22, 35, 60, 80, 90, 10, 30

→ Delete node 30, 80, & 40 :-

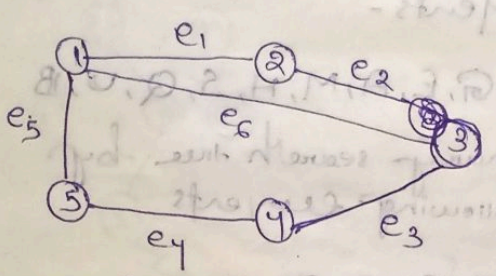
Chap-07

GRAPHS

→ A graphs is another important non-linear data structure.

→ A graph 'G' consists of a setup 'V' where the members are called vertices of 'G' and a set 'E' where the members are called as edges of 'G'.

→ The setup vertices is non empty and the setup edges contains a pair of vertices.

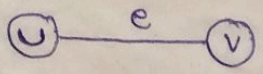


$V = \{1, 2, 3, 4, 5\}$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

→ If e is equal to (u, B) it means that e = (u, B) with vertices u and v' or e is an edge with vertices u and v' or e is said to be ~~incident~~ incident on u and v.

→



→ A graph can be of 2 types -

① undirected graph

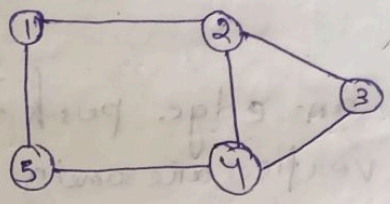
② Directed graph

1) undirected graph

→ If the pair of vertices are on order then that graph is called as undirected graph.

→ It means that if there is an edge between v_1 & v_2 then it can be represented as (v_1, v_2) or (v_2, v_1).

→ Ex :-

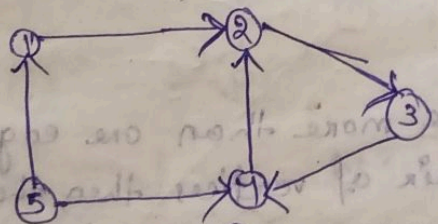


$V = \{1, 2, 3, 5\}$

$E = \{(1,2), (2,3), (3,4), (4,5), (2,4), (1,5)\}$

2. Directed graph :-

→ If the pair of vertices are order, then that graph is called as directed graph.



$N = \{1, 2, 3, 4, 5\}$

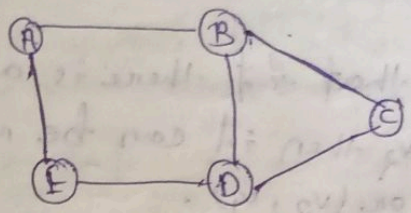
$E = \{$

Terminology used in graph

① weighted graph →

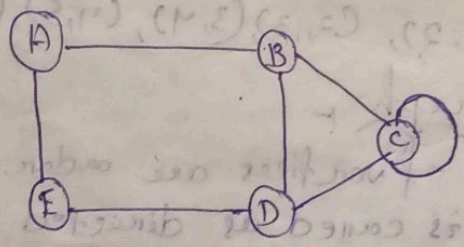
→ A graph is said to be a weighted graph if all the edges are left with some numbers

→ Ex-



→ If their is an edge push starting and ending vertices are same in graph then it is a self loop.

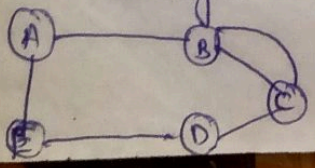
→ Ex :-



(3) Parallel

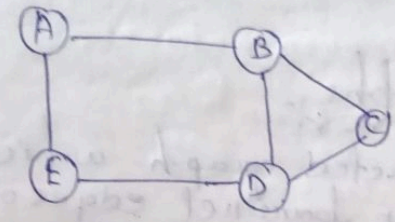
→ If their are more than one edge bet^{n} the same pair of vertices then their known as parallel edges.

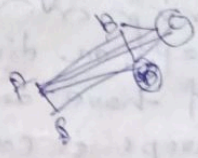
→ Ex :-



(4) Adjacent vertices

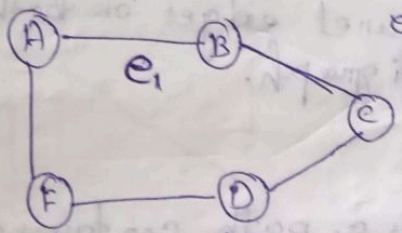
→ A vertics

→ 



(5) Incidence :

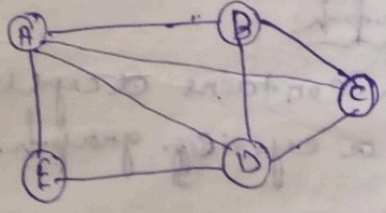
→ In an undirected graph, the edge (u,v) is incident on u & v.

→ Ex : 

edge e_1 ve

(6) Degree of a vertics

→ The degree of a vertics is the number of edges incident to that vertics.

→ Ex 

Degree of a = 4, b.

(a) In degree of a vertics

→ The in degree of a vertics if the number of edges coming into that vertics.

(6) Out degree of a vertics

→ The out degree of a vertics is the number of edges going out the number of vertics.

(7) Simple graphs :-

→ A graph or directed graph which doesn't have any parallel edges or self loops is called as a simple graphs.

(8) Multi graph :-

→ A graph which has either a self loop or parallel edges or both is called as a multi graph.

(9) cycle :-

→ If their is a path containing one or more edges which starts from a vertics and terminets at the same vertics, the path is called as a cycle.

(10) cyclic graph

→ If a graph contains a cycle then it is called as a cyclic graph.

(11) A cyclic graph

→ If a graph or a path doesn't contain any cycle then it is called as a cyclic graph.

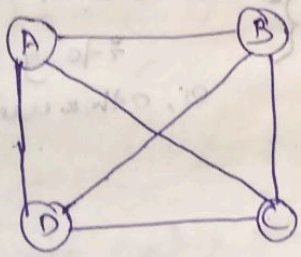
Representation of a graph

→ A graph can be represented in many ways - 1) set representation
2) sequential representation
3) Linked representation

1) set representation

→ In this representation a graph is represented or maintained by using two sets - (1) set of vertices, V
(2) set of vertices, E

→



$V(G) = \{A, B, C, D\}$

$E(G) = \{(A,B), (B,C), (C,D), (D,A), (B,D), (C,A)\}$

(2) Sequential representation 21.10.19

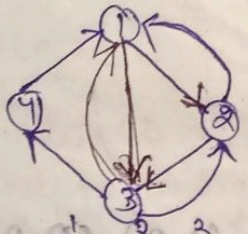
→ In sequential representation, the graphs are represented in the form of matrixes.

There are two common matrixes:

1) Adjacency matrixes
2) Incidence matrixes

1) Adjacency matrixes

→ A matrix is said to be and adjacent matrixes if a_{ij} = { 1; if there is an edge from i to j
0, otherwise

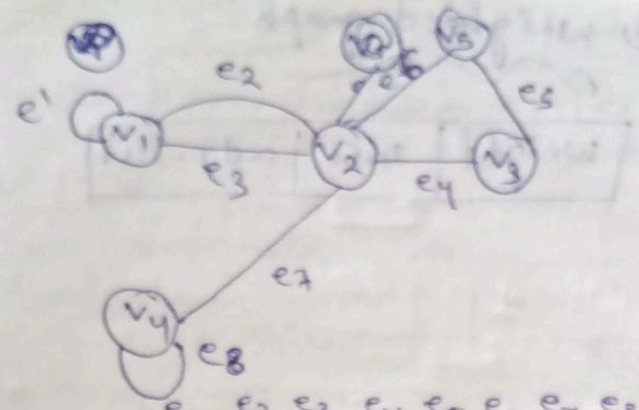
→ 

→ $A =$

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

(2) Incidence matrixes

→ A matrix is said to be an incidence matrix if a_{ij} = { 1, if j^{th} edge is incident on i^{th} vertex
0, otherwise



$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

path matrix :

→ A path matrix for a graph can be represented as.

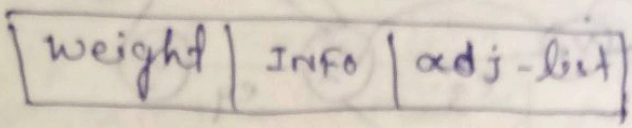
$$P_{ij} = \begin{cases} 1, & \text{if there is a path from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$$

(3) Linked representation

→ In linked representation node structure are used - for non-weighted graph

INFO	adj-List

② weighted graph



Chap-8 22.10.19

Sorting, searching & merging

Sorting : sorting is a process or mechanism of arranging a sequence of records so that the value of a sequence of records the value of there key field forms a decreasing order.

(1) Bubble sort → Bubble sort, also known as exchange sort is a simple sorting algorithm which was repeatedly through a list to be sorted.

→ The basic idea of bubble sort is comparing to items at a time and swapping them if they are in the wrong order.

→ This process is repeated through the list until the list is sorted.

Algorithm :

'A' is the array to be sorted.

'i & j' are two counter values.

① for i ← 1 to length [A]

② for j ← length [A] down i+1

③ if A[j] < A[j-1]

④ exchange (A[j], A[j-1])

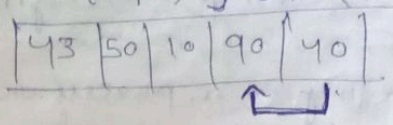
Ex 43, 50, 10, 90, 40

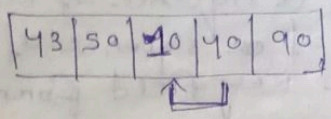
Sort the gaboves sequence by using bubble sort.

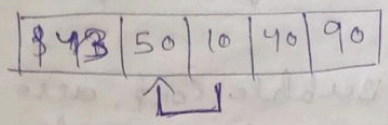
A

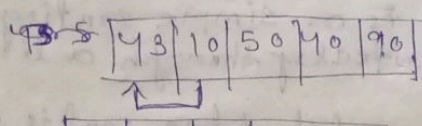
43	50	10	90	40

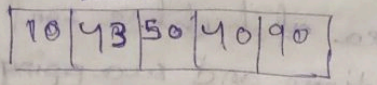
1st $i = 1$, $j = 5$ to 2



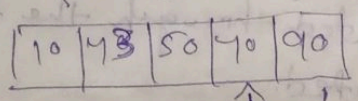


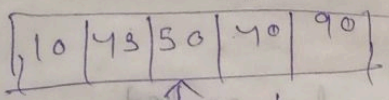


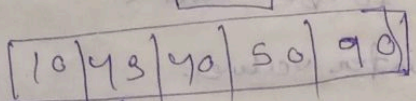


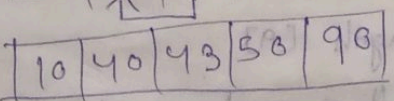


2nd $i = 2$ $j = 5$ to 3

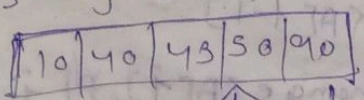


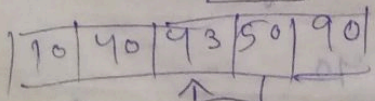






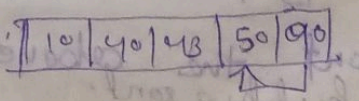
3rd $i = 3$ $j = 5$ to 4





→ So the sorted sequence is 10, 40, 43, 50, 90.

4th $i = 4$, $j = 5$ to 5



Assignment

① 5, 2, 1, 4, 3, 7, 6, sort the above sequence by using bubble sort

Quick sort

→ Quick sort is an implementation of devid an conquer.

→ In quick sort we devide the original list into two sub list. and choose a key or pivot element from which all the left side element are smaller and all the right side elements are greater than that element.

→ Quick sort works by partitioning a given array.

A[p ..... r] into two non empty sub arrays A[p ..... q] and A[q+1 .. r] such that every key in A[p to q] is less than or equal to every key in A[q+1 ..... r].

→ Then the two sub arrays are sorted by recursive calles to which solve algorithm.

Algorithm QUICK-SORT (A, P, r)

1) If p < r then
2) q ← partition (A, P, r)
3) QUICK-SORT (A, P, q-1)
4) QUICK-SORT (A, q+1, r)

PARTITION (A, P, r)

1) $x \leftarrow A[r]$
2) $i \leftarrow p-1$
3) for $j \leftarrow p$ to $r-1$
4) do if $A[j] \leq x$
5) then $i = i+1$
6) exchange $A[i] \leftrightarrow A[j]$
7) exchange $A[i+1] \leftrightarrow A[r]$
8) return $i+1$

36	15	40	1	60	20	55	25	50	20
1	2	3	4	5	6	7	8	9	10

r = 10

P = 1 ; 1 < 10 (True)

x = 20

i = 1 - 1 = 0

j = 1 to 9

36 ≤ 20 (False)

j = 2

15 ≤ 20 (True)

i = i + 1 = 0 + 1 = 1

A[i] ↔ A[j]

j = 3 i = 1

40 ≤ 20 (False)

j = 4

1 ≤ 20

A[i] ↔ A[j]

25.10.19

Searching

Searching can be of two types -

1. Linear search
2. Binary search

Linear search

→ In linear search, each element one by one sequentially and it is compared with the desired element.

→ A search will be ~~unsuccesses~~ unsuccessful if all the elements are read and the desire element is not found.

→ It is the ~~s~~ for searching and element in a list.

→ It searches the element sequentially in a list, no matter weather the list is sorted or unsorted.

Draw backs or disadvantages

→ It is a very slow ~~pro~~ process

→ It is used only for small amount of data.

→ It is a very ~~time~~ time consuming method.

Algorithm

→ 'A' is an array with some elements.

→ 'loc' is the location at which item is found.

→ 'i' is a counter value.

1) Input an array A of 'n' elements and "data" to be searched and initialise
   loc = -1
2) Initialise i=0 and repeat through step 3 if (i<n) by incrementing i by one
3) if (data == A[i]
   a) LOC = i
   b) Goto step 4
4) if LOC >= 0
   print "data is found and search is successful"
   else
   print "data is not found and search is unsuccessful".
5) exit

Ex search 33 by using linear search

11	22	33	44	55	66

Soln

11	22	33	44	55	66
0	1	2	3	4	5

data = 33 loc = -1

i = 0 A[i] = A[0] = 11

33 == 11 (false)

i = i+1 = 0+1 = 1

A[1] = 22

33 == 22 (false)

i = i+1 = 1+1 = 2

A[2] = 33

33 = = 33 (True)

LOC = i ≥ 0 (True)

(Search successful)

Binary search :-

→ Binary search is an extremely efficient algorithm as compare to linear search.

→ In binary search we first compair the key with the item in the middle position of the array. If there is a match we can return immediately.

→ If the key is ≤ than the middle key then the item lies in the lower half of the array else it lies in the upper have of the array.

→ Binary search is only applicable on sorted array.

Algorithm

→ 'A' is an array of elements.

→ 'Data' is the item to be searched in the array.

→ 'LB' is the lower bound index of the array

→ 'UB' is the upper bound index of the array

1) Input an array 'A' of 'n' element and 'data' to be searched ~~and initialise~~

~~loc = 0~~

2) LB = 0, UB = n, mid = int (LB + UB)/2

3) Repeat step 4 and 5 while (LB <= UB) and A[mid] != data

4) if (data < A[mid]
   UB = mid - 1
   else
   LB = mid + 1

5) mid = int((LB + UB)/2)

6) if (A[mid] == data)
   print "Data Found"
   else
   print "Data not Found"

7) exit

Ex search 15 by using binary search

5 8 15 25 30 40 55

Soln

5	8	15	25	30	40	55
0	1	2	3	4	5	6

data = 15

LB = 0 UB = 6 mid = 3

A[mid] = A[3] = 25

15 < 25 (true)

UB = mid - 1 = 3 - 1 = 2

mid = 1 A[mid] = A[1] = 8

15 < 8 (false)

LB = mid + 1 = 1 + 1 = 2

mid = 2 A[mid] = 15

15 = 15 Data found.

Chap - 9 26.10.19

File organisation

Introduction to file :

Now a days most organisations use data collection application which collect large amount of data in one from or other.

→ for example : when you seek addmission in a college, A lot of data like our name, address, phone number, course which be want to study etc. are collected. All these data where traditionally store on paper document but handling documents had been a difficult cost. so it became necessary to store the data in computer in the from of files.

File organisation

→ file is a collection of related records or records are organised inside the file because there is significant effect on system performance.

→ organisation of record means logical arrangement of records in the file. so choosing an appropriate file organisation is an important decision

→ It must be done keeping the priority of acchivement and good performance is likely to be achived by the use of file.

→ Therefore before selecting appropriate file organisation method. The following considerations should be keep in mind.

1) Rapid access of one or more records
2) Efficient storage of records.
3) Ease of inserting or deleting or updating one or more records.

→ using data redundancy to ensure data integrity.

Techniques used for File organisation

→ The various techniques used for file organisation are - 1) sequential file orgn
2) Relative file orgn
3) Indexed sequential

1) sequential file orgn

→ A sequentially organised file stores the record in the order in which they were entered.
i,e. The 1st record is entered as 1st record in the file, 2nd record is entered as second record and the file and so on.

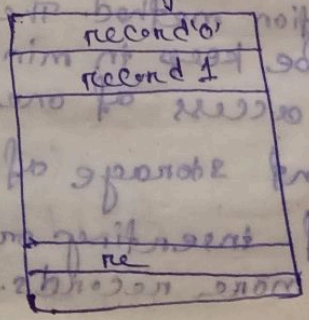
→ sequential file organisation is the most basic way to organised a large collection of record in a file.

→ sequential files can be read only sequentially.

→ once we stored the record in a file we cannot make any changes to the record.

→ In sequential organisation all the record ~~require~~ ~~need~~ have same size & same field format & every field has a fixed size.

→ sequential organisation are generally used to generate reports or perform sequential reading of large amount of data which some programms need to do such as pay roll processing of all employees of an organisation.



Advantages

→ simple to an easy-to-handle can be stored on magnetic disk as well as magnetic tapes.

Disadvantages →

→ Records can be read sequentially, a simple deletion also requires reflection the orginal file with the new file with the desire changes.

Q) Relative File organisation

→ Relative file organisation provides an effective way to access individual record directly by using a relative key.

→ It means that the record number represent the location of the record related to the begining of the file.

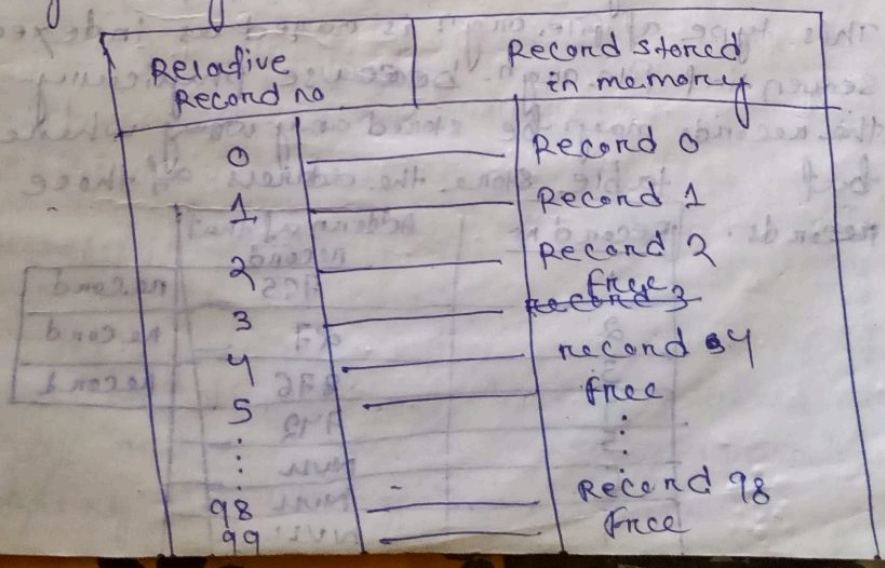
→ For example :- Let say their are records ranging from '0' to 'n-1' for 'n' number of records.

→ So the record number '0' is the 1st record in the file, record number '1' is the second record in the file and so on.

→ In relative file organisation records are organised in assending relative record number.

→ Relative file organisation can be used for both random and sequential access.

→ Relative file organisation provide support From only one key.



Advantages →
1) Ease of processing
2) Random access of record makes access faster
3) Allows deletion and addition in the same file.
4) New records can be easily added in the free location.
5) provides random as well as sequential access.

Disadvantages:
1) Records can be of fixed length only.
2) Use of relative files is restricted to disk devices.
3) For accessing a record, record number must be ~~none~~ known in advanced.

(3) Indexed ~~number~~ sequential file orgn

→ Index Sequential file organisation store data for retrival.
→ Records are of fixed length and every record is uniquely identify by the key field.
→ An indexed table is maintained with stores the record number and address of all the records.
i,e. for every file we have an indexed table.
→ This type of file orgn is called as indexed sequential file orgn. because physically the records may be stored ~~any~~ where but table store the address of those records.

Record no	Address of that record	
1	765	record
2	87	record
3	876	record
4	742	
5	NULL	
6	NULL	
7	NULL	

Advantages :- Indexes are small and can be searched quickly allowing the data base to access only the records it needs.

→ Records can be access sequentially as well as randomly

→ Allows objects updating records in the same file.

Disadvantages

→ Indexed sequential file can be store only on disk.

→ Needs extra space to store indexes.

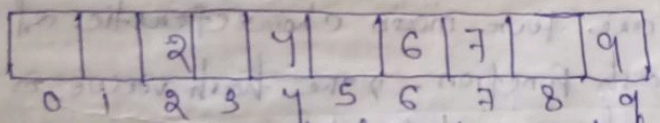
→ Supports only fixed length record.

Introduction to ~~hashing~~ hashing

→ There are different searching techniques where the search time is basically depended on the number of elements. Sequential search, binary search and all the search tree are totally depend on the number of elements and so many key comparisons are also involved.

Our basic need is to search an element in a constant time. and less key comparisons should be used. Suppose there are 'n' number of elements in an array. So all the keys should be unique and in the range of '0' to 'n-1'.

Let there be five records 9, 4, 6, 7, 2.
This will be stored as-



Now we can see that each record has a key value that can be directly access by using the array index. Now comes the idea of hashing where we will convert the key into array index and put the record in an array and in the same way for searching the record. convert the key into index array and get the record in the array.

The general array index uses hash function which converts the key into array index and the array which supports hashing for storing record or searching record

record is called as hash table.

Collision → (ఘర్షణ)

→ If each key is a match on a unique hash table address, then this situation is an ideal situation but their may be possibility that our hash function is generating same hash table address for different keys. This situation is called as collision.

Hash function

→ Hash function is a function which when applied to a key produces an integer value which can be used as an address in a hash table.

→ perfect hash function is a function which when applied to all the members of a set up items produces a unique set up integer with in a suitable range.

→ Good hash function minimize collision by spreading the elements uniformly throughout the array.

Characteristic of

→ There are four main characteristic of a good hash function - 1) The hash value is only determine by the data being a

2) The hash function uses all the input data

3) The hash function uniformly distributes the data across the entire possible set up hash values.

4) The hash function generate very different hash value for similar strings.

Types of hash function

Some common type of hash functions are -

1) Division method
2) multiplication method
3) mid square method
4) folding method

1) Division method

→ In the division method, for creating hash functions we make a key 'k' to be stored into one of the 'm' slots by taking the remainder of k/m i.e. the hash function is h(k) →

$$h(k) = k \bmod m$$

or

$$h(k) = k \bmod m + 1$$

For example → If there is a data hundred and a hash table of size 12

$h(100) = 100 \bmod 12$

$= 4$

or

$h(100) = 100 \bmod 12 + 1$

$= 5$

→ Flooring
⌊ ⌋ → symbol
⌈ ⌉ → ceiling

2) Multiplication method

→ The multiplication method can be used for creating hash function by using the formula

$h(k) = \lfloor m(kA \bmod 1) \rfloor$

while $0 < A < 1$

most probably $A = \frac{\sqrt{5}-1}{2}$

$= 0.618$

3) Mid square method

→ A mid square method, we square the key. After gating the number, we take sum digit from the middle of that number as an address.

Ex: 1337, 1273, 1391, 1026

Square values: 1787569, 1620524, 1934881, 1052676

Address value: 87, 20, 34, 52

4) Folding method

→ In folding method, the key is broken into pieces, then we add them and get the hash address.

→ Ex: 82394561, 87139465, 83567271, 85943228

- 82394561 → 823 + 94 + 561 = 1478
- 87139465 → 871 + 39 + 465 = 1375
- 83567271 → 835 + 67 + 271 = 1106
- 85943228 → 859 + 432 + 228 = 1120

Now we can store the above data in a hash table of size '0 to 999' and the hash address are -

h(82394561) = 1478

h(87139465) = 1375

h(83567271) = 1106

h(85943228) = 1120

Collision resolution techniques :-

→ suppose we want to add a record 'R' with key 'k' & two ways are there but suppose the memory location address h(k) has already occupied.

→ This situation is called as collision, i,e a collision occures when more than one keys are stored to the same hash value in the hash table.

→ The following ways are used to resolve the collision - 1) Collision resolution by open addressing

2) Collision resolution by separate chaining.

The performance of these methods depends on the load factor i,e - $\lambda = n/m$

where, n is the number of keys in K and m is the number of hash addresses

Collision resolution by open addressing (or) Hashing with open addressing

→ In open addressing all the elements are stored in the hash table itself i,e. each table entry contains an element of the dynamic set. when searching for an element we systematically examine table slots until the desired element is found or it is clear that the element is either found not found not in the table. Thus in open addressing, The load factor can never exceed one.

→ Three techniques are commonly use to compute the program sequence required for open addressing.

1) Linear probing
2) Quadratic probing
3) Double hashing

1) Linear probing

→ The method of linear probing uses the hash function.

$$h(K,i) = (h'(K) + i) \bmod m$$

2) Quadratic probing

→ The method of quadratic probing uses the hash function -

$$h(K,i) = (h'(K) + c_1 i + c_2 i^2) \bmod m.$$

3) Double hashing

→ The method of double hashing use the hash function -

$$h(K,i) = (h_1(K) + i\,h_2(K)) \bmod m$$

where $h'(K) = K \bmod m$.

(2) Hashing with separate chaining

→ This method maintens the chain of elements which have the same hash address. we can take the hash table has an array of pointers, size of the hash table can be number of records there each pointer will point to one linked list and the elements which have same hash address will be maintain in that linked list. we can maintain

The linked list in sorted order. An each element of the linked list will contain the hole record with key.

~~Quick sort~~

Questions

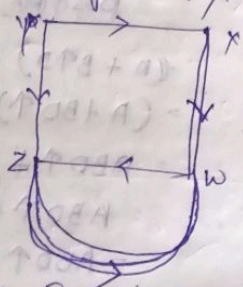
(1) Discuss about different types of data structure operations

(2) State how a two dimensional array is represented in memory.

(3) (A+B↑D) / (E-F) +G
Convert it to prefix and postfix notation.

(4) Discuss the algorithm to input and delete element from a queue.

(5) Write adjacency matrix of the following graph



(6) Convert the following into postfix
(i) A * B - (C/D) ↑5 (ii) (X + Y) - Z ↑ 3

(7) Define queue. Explain the insertion and deletion operation in a queue with suitable diagram. Take 4 elements in a queue.

(8) Define binary search tree. Construct a BST with the following nodes:

(9) List out the string operations and explain them with suitable example.

(10) Define graph. Draw a graph and write adjacency matrix?

(11) Why does collision occur? Explain the various technique to resolve a collision.

(12) Define searching. write algorithm for linear searching?

(13) Define algorithm and explain the space time trade off of algorithm complexities.

(14) write an algorithm to insert element in an array.

(15) write an algorithm to traverse a linked list?

(3) $(A+B\uparrow D)/(E-F)+G$

Convert it to prefix and postfix notation.

(A) prefix

$(A+B\uparrow D)/(E-F)+G$

$= (A+\uparrow BD)/(E-F)+G$

$= +A\uparrow BD/-EF+G$

$= /+A\uparrow BD-EF+G$

$= +/+A\uparrow BD-EFG$

postfix

$(A+B\uparrow D)/(E-F)+G$

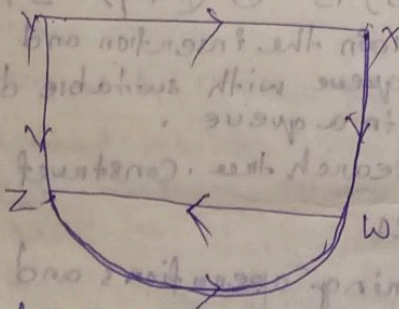
$= (A+BD\uparrow)/(E-F)+G$

$= ABD\uparrow+/EF-+G$

$= ABD\uparrow+EF-/+G$

$= ABD\uparrow+EF-/G+$

(5)



(A) A matrix is said to be an adjacent matrix where,

$$A_{ij} = \begin{cases} 1, & \text{if there is an edge from } i \text{ to } j \\ 0, & \text{otherwise.} \end{cases}$$

So, the adjancy matrix for the given graph will be

	w	x	y	z
w	0	0	0	1
x	1	0	0	0
y	0	1	0	1
z	1	1	0	0

6) postfix

(i) A * B - (C / D) ↑ 5

- A * B - (C / D) ↑ 5
- = A * B - CD / ↑ 5
- = A * B - CD / 5 ↑
- = AB * - CD / 5 ↑
- = AB * CD / 5 ↑ -

(ii) (x + y) - z ↑ 3

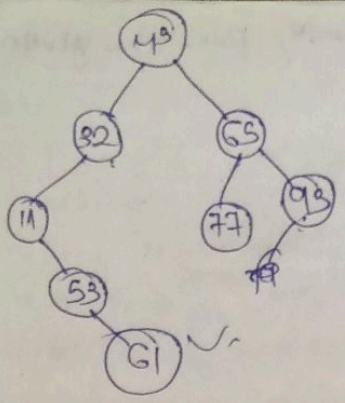
(x + y) - z ↑ 3

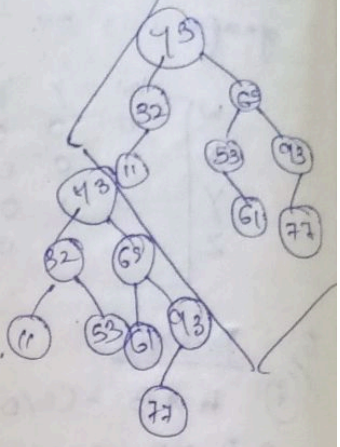
= xy + - z 3 ↑

= xy + z 3 ↑ -

(5) A matrix is said to be an adjacent matrix when.

Aij = { 1, if there is an edge from i to j . 0, otherwise }






```
int arr[5], i;
clrscr();
printf("enter 5 elm");
for(i=0; i<5; i++)
scanf("%d", &arr[i]);
```

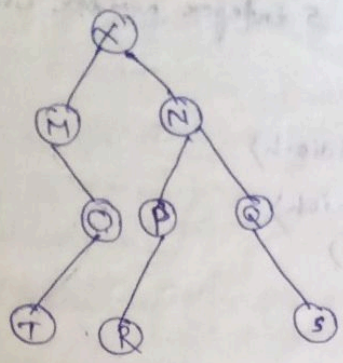
Prabhat
Britto

```
for(i=0; i<5; i++)
{
printf("\n%d", arr[i]);
}
```

Internal-3

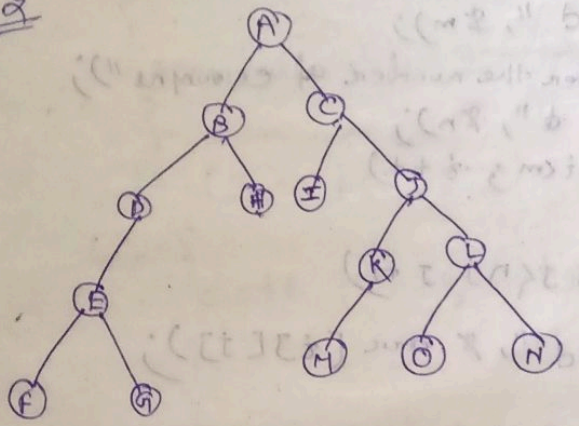
1) what do you mean by binary search tree ?
2) Define recursion ?
3) Define degree of a vertex ?
4) what do you mean by adjacent matrix ?
5) what is garbage collection ?
6) construct a binary search tree with the following nodes ?
45, 32, 65, 11, 93, 77, 53, 61
7) Define binary tree, explain pre-order tree traversal with ex !

Q1



T - O - M - X - R - P - N - Q - S

Q2



F - E - G - D - B - H - A - I - C - M - K - J - O - L - N

2) wap to input 5 integer number using 2D array:-

```
# include <stdio.h>
# include <conio.h>
  void main()
{
 int m,n,i,j;
 int arr[m][n];
  clrscr();
printf("Enter the number of rows");
 scanf("%d", &m);
printf("Enter the number of columns");
 scanf("%d", &n);
for (i=0; i<m; i++)
{
for (j=0; j<n; j++)
{
scanf("%d", &arr[i][j]);
}
}
for (i=0; i<m; i++)
{
For (j=0; j<n; j++)
{
printf("%d\t", arr[i][j]);
}
}
}
```

output :- Enter the number of rows 2
→ Enter the number of columns 3

1 2 3
4 5 6 → 1, 2, 3, 4, 5, 6

write a program to input 5 integer number using 1(d) array ?

```
#include <stdio.h>
#include <conio.h>
   void main()
{
 int array[5], i, j;
  clrscr();
printf("Enter 5 number");
for (i=0; i<5; i++)
  {
   scanf("%d", &arr[i]);
  }
  for (j=0; j<5; j++)
    {
     printf("%d", arr[j]);
    }
   }
```

output -> Enter the 5 number;
2, 3, 4, 5

-> priti.c

write a program in c to add two matrix

```
#include<stdio.h>
#include<conio.h>
int a[3][3], b[4][5], c, i, j, k, p, s;
int m, n;
printf("enter row size & column size \n");
scanf("%d %d", &m, &n");
for (i=0; i<m; i++)
{
for (j=0; j<n; j++)
{
printf("enter element");
scanf("%d", &a[i][j]);
}
}
for (i=0; i<m; i++)
for (j=0; j<n; j++)
{
printf("\n show the result %d", a[i][j]);
}
int c, d;
print("enter row size & column size \n");
scanf("%d %d", &c, &d);
for (k=0; k<c; k++)
{
p=0; p<d; p++)
{ printf("enter element");
scanf("%d", &b[k][p]);
}
}
for(k=0; k<c; k++)
for (p=0; p<d; p++)
{printf("\n show
the result %d", b
[k][p]);
}
s = a[i][j] + b
[k][p]
```

→ Stacks can be implemented in memory in two ways. —

① using array
② using linked list

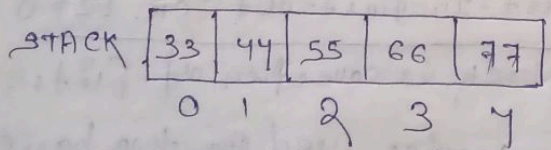
- **Representation of stack using array :**

→ Stack can be maintained for created by using 3 things which are -

① An array : STACK
② A variable : Top
③ Another variable : MAXSTK

where STACK is an array,

→ Top is the index of top element of the STACK
→ MAXSTK is the size of the stack.



Top = 4

MAXSTK → 5

- **PUSH OPERATION**

→ PUSH operation is used to insert an element into the STACK. This operation can be performed in two steps -

① Implement the TOP by one
② Insert the new item at the position to which Top is pointing