Kiran.P
Assoc. Prof.
Dept of CS&E

90+20=110

Module 1

Data Structure & Applications

Introduction

Data are simply values or set of values.

Ex P= { 9845440059 9845552 1 } ... set of phone numbers

Data item refers to a single unit of values

ex one of the Element of P is 9845440059

& is refered as data item.

Defenition: of Data Structure

Logical or mathematical model of a particular organization of data.

Data are organized into complex typm of Structurm.

This is required because the collection of data are frequently organzed into hierarchy of fields, records & files.

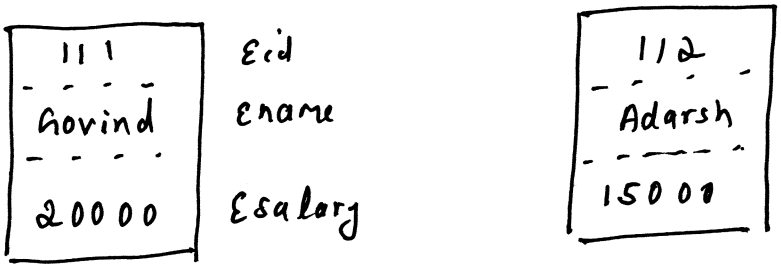
Ex EMPLOYEE Contains details like

Eid - Empid

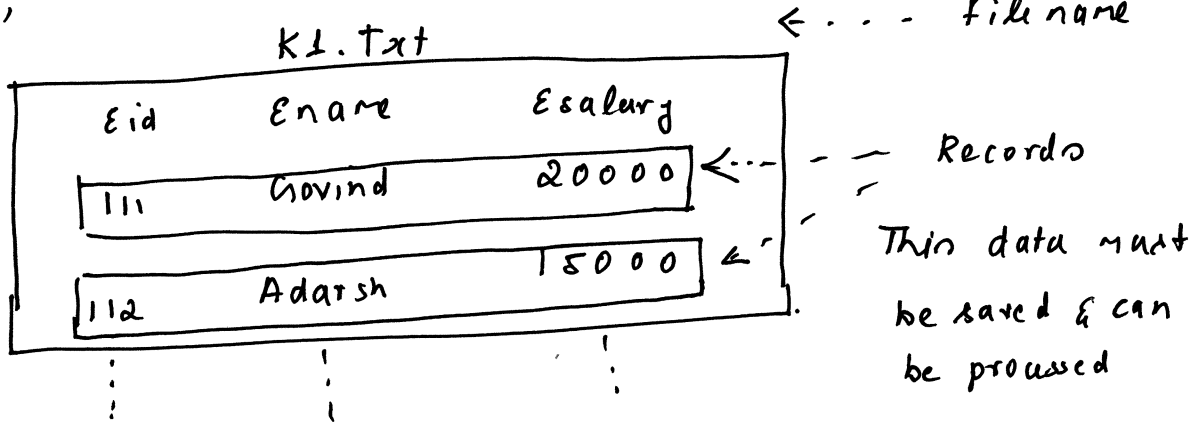
Ename - Emp name.

Esal - Emp salary

The data for the above mentioned example include



This data must be saved in a hierarchy with fields, records & files



The study of the above said D.S includes the following three steps

i) Logical / mathematical descriptn of the structure

ii) Implementation of the structure on a computer

iii) Quantitative analysis of the structure.

Different Categories of D.S

i) primitive Data structure: are predefined types of data which are supported by the programming language.

Ex: Int, float, char

ii) Non primitive data structures

are structures derived from primitive Structures.

Are further divided in to

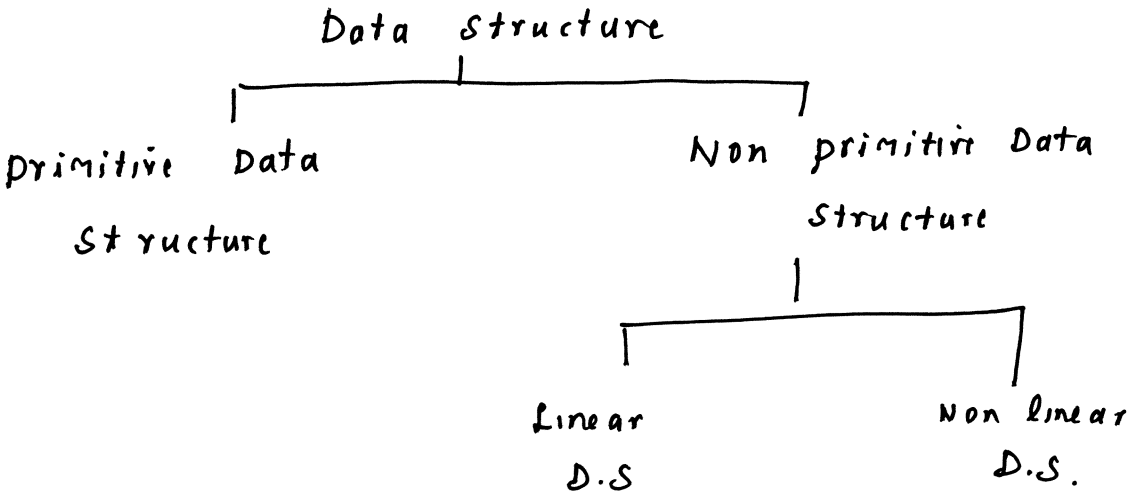
Linear D.S: are structures which traverse the data Element sequentially, Memory is allocated sequentially

Ex: Arrays, linked list, stack & Queues

Non linear D.S: Every data item is attached or retrieves several other data items.

Data items are not allocated sequentially

Ex: trees, graphs.



Arrays.

List of finite number 'n' of similar data Elements.

in C language

EMP information must be saved & processed, this requirin data structure (DS)

The following are some of the DS that can be used based on certain applications.

1) Arrays

is the simplest type of ds

- can store a fixed sze sequential collection of elements of the same type.

→ linear array (one dimensional array), we mean a list of

→ a finite number 'n' of similar data elements referenced respectively by a set of n consecutive no.

usually 1, 2, 3, ..., n.

name A then Elements are denoted by

A[0]
A[1]
A[2]
A[3]
⋮

- n elements
- consecutive (one after the other)

... referred by 0, 1, 2, ... n

The number k in A[k] is called a subscript & A[k] is called a subscripted variable.

Ex

EMP

1	Gov
2	Adar
3	Bhar

EMP[1] 'Gov'

EMP[2] 'Adar'

Linear array are called one dimensional array, because each element is referred by one subscript.

EMP[1] ← single reference.

A Two-dimensional array is a collection of similar data elements where each element is referenced by two subscripts (are called Matrices) or tables.

Linked lists is an ordered set of data elements each containing a link to its successor

EMPDET

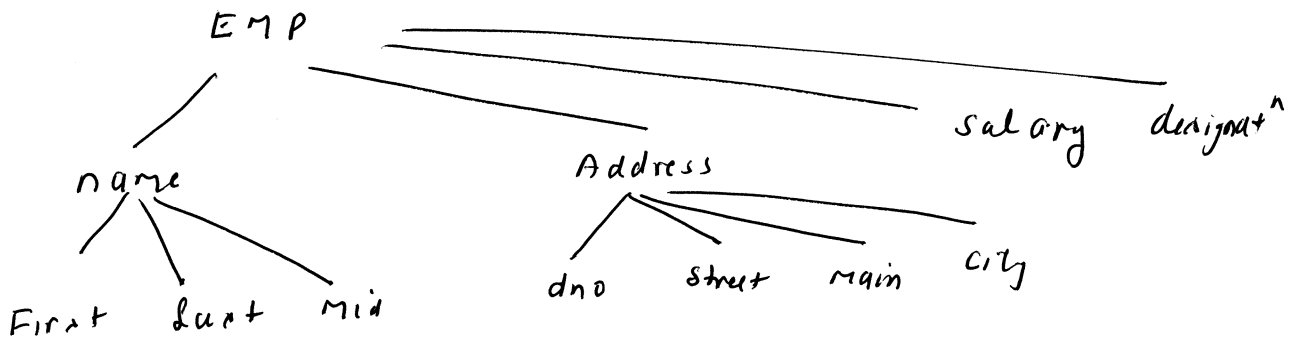
Eid	En	deptname
111	Gov	Computer scienu
112	Ad	Computer scienu
113	Bh	Computer scienu
114	Chet	Information scienu
115	sita	Information science

The other way of storing this information is

Eid	En	dnpointer
111	Gov	1
112	Ad	1
113	Bh	1
114	Ch	2
115	sita	2

deptname	
Computer Science	1
Information Scienu	2

Trees: Data frequently contain a hierarchical relationship between various elements. The DS which reflects this relationship is called tree.



There are other D.s like.

Stack: is also called a last in first-out system (LIFO)
insertion & deletion can happen at only one end. called top.

Queue: also called first in first out (FIFO) system, is a linear list in which deletion can take place at only one end. of the list, which is front. Insertion can happen at the rear of the list

Graph: consists of finite set of vertices or nodes

Data sometimes contain a relationship blw pairs of elements which is not necessarily hierarchical in nature.

*** Data structure operations

(1) Traversing Accessing each record exactly once so that certain items in the record may be processed. (visiting the record)

(2) Searching: Finding the location of the record with a given key. or finding the location of all records which satisfy one or more conditions

(3) Inserting: Adding a new record to the structure.

(4) Deleting: Removing a record from the structure.

The following two operations which can be used in special situations

(1) sorting: Arranging the records in some logical order

(2) Merging: combining the records in two different sorted files in to a single sorted file.

Algorithm : Linear Search

LINEAR (Data, N , ITEM, LOC)

Data is a linear array with N Elements

ITEM is the value to be searched.

This algorithm finds the location LOC of ITEM in DATA or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA]
   Set DATA [N+1] := ITEM

2. [Initialize Counter] Set LOC := 1:

3. [Search for ITEM]
   Repeat while DATA [LOC] ≠ ITEM :
   Set LOC := LOC + 1
   [End of Loop]

4. [Successful ?] If LOC = N+1 then
   Set LOC = 0

5. Exit

Algorithm Binary Search.

BINARY (DATA , LB, UB, ITEM , LOC)

DATA is a Sorted array with lower bound LB & upper bound UB & ITEM is the value to be searched.

The variable BEG, END & MID denote begining, end & middle location of the array. This algorithm finds the location LOC of ITEM in DATA or sets LOC := NULL.

1 [Initialize Segment Variable]

Set BEG := LB , END := UB &

MID := INT ((BEG + END) / 2)

2. Repeat Step 3 & 4 while BEG ≤ END &

DATA [MID] ≠ ITEM

3. If ITEM < DATA [MID] then

set END := MID - 1 :

Else

set BEG := MID + 1 :

[End of If statement]

4. Set MID := INT ((BEG + END) / 2)

[End of step 2 loop]

5. If DATA [MID] = ITEM then

set LOC := MID

Else

set LOC := NULL

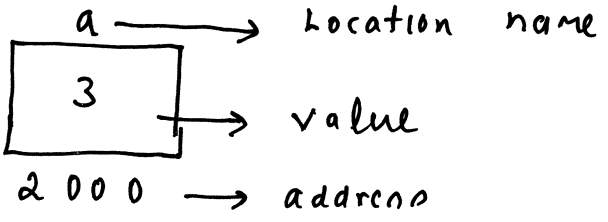
[End of If structure]

6. Exit

Pointer

Def: A pointer is a variable which holds the address of another variable.

```
int a = 3;
```




```
int * p;
```

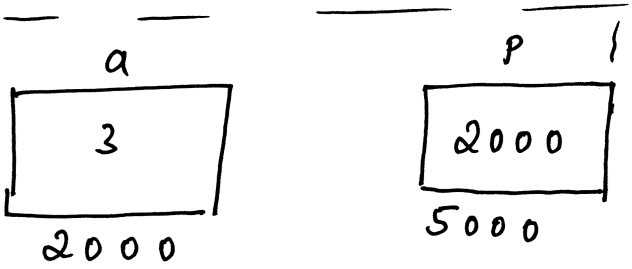
p is a pointer variable of type int & can hold the addrem of integr variable

initial representation

---

p = & a

p = 2000



'&' : is called addressing operator.

& a

is read as: retrieve the addrem of variable a.

The other operator which is generally used is the dereferencing operator.

* : this is used in retrieving the value within the address.

* p

⇓ internally value of p will be substituted which is 2000

* 2000

This retrieves value within the address 2000

⇓

3.

pf ("%d", * p) will display 3.

To insert value within a using scanf

scanf : inserts the value in to the address.

scanf ("%d", & a)

I/p is ⇒ 50 ⇒ & a (2000) ⇒ copies it into memory location 2000

The same execution can be done using

scanf ("%d", p)

I/p is ⇒ 50 ⇒ p (value of p is 2000) ⇒ copies it into memory location 2000

common mistakes

q
50
2000

X scanf ("%d", a)

I/p ⇒ 50 ⇒ a (50) ⇒ copies it into memory location 50

p
2000
5000

80
50

X scanf ("%d", *p)

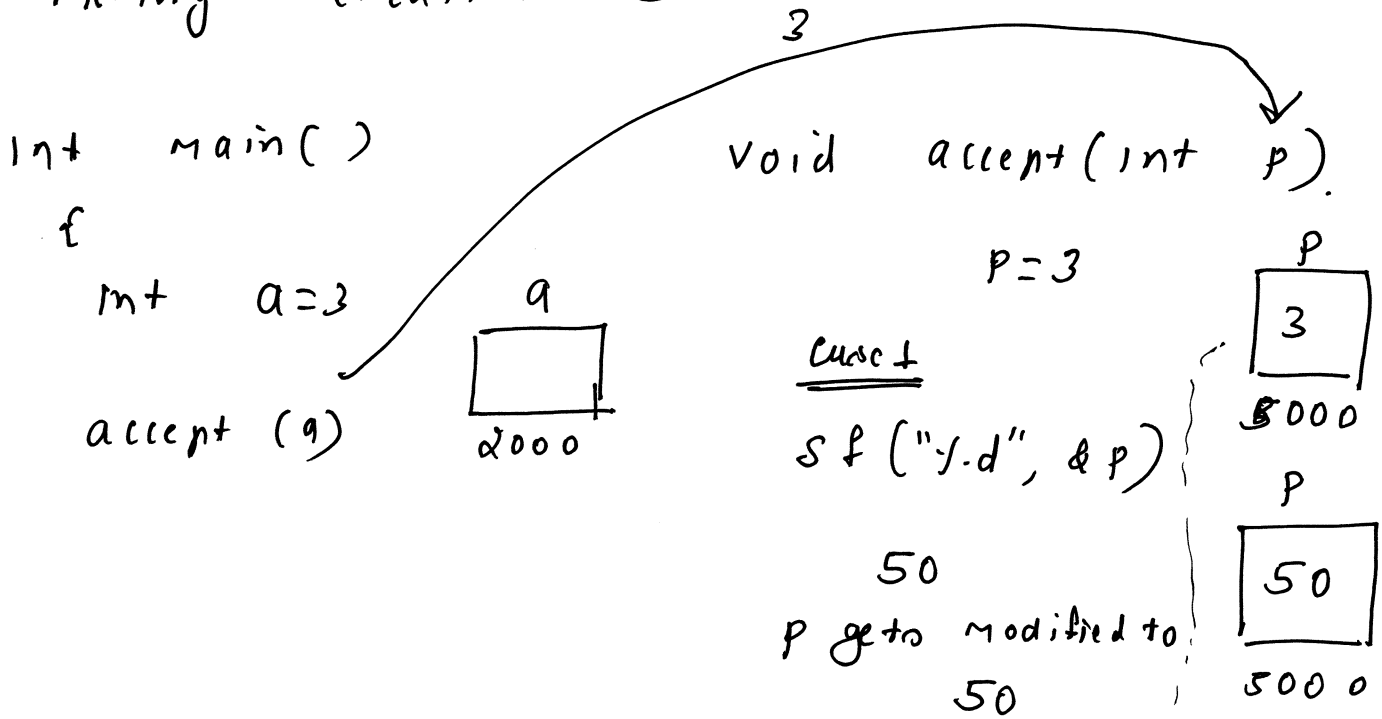
I/p ⇒ 80 ⇒ *p (50) ⇒ write the value into memory location 50.

There are two ways of passing values to functions

(i) call by value: changes made will not be reflected in the main function

(ii) call by reference: changes made to the variable will be reflected in the main function

the value of "50" will be written in memory location "3"



* This is ~~not~~ call by value.

program to swap two numbers using call by reference

```
#include <stdio.h>
void accept(int * a)
{
pf(" Enter the value \n");
sf("%d", a);
}
```

```
// This function swaps the value of *a & *b
void swap (int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
int main()
{
    int a, b;
    printf ("Enter the value of a \n");
    accept (&a);
    printf (" Enter the valw of b \n");
    accept (&b);
    swap (&a, &b);
    printf (" The swaped valwes are %d %d \n",
            a, b);
}.
```

Array initialization

initialization of array can be done using different ways.

```
int a[5] = { 2, 4, 6 }
```

: max "5" elements

a[0] = 2
a[1] = 4
a[2] = 6

the other values are initialized to zero

a[3] = 0
a[4] = 0

int a[] = { 2, 4, 6, 5 }

a[0] = 2

a[1] = 4

a[2] = 6

a[3] = 5

String initalization is done differently.

char a[] = "INDIA"

a[0] = I

a[1] = N

a[2] = D

a[3] = I

a[4] = A

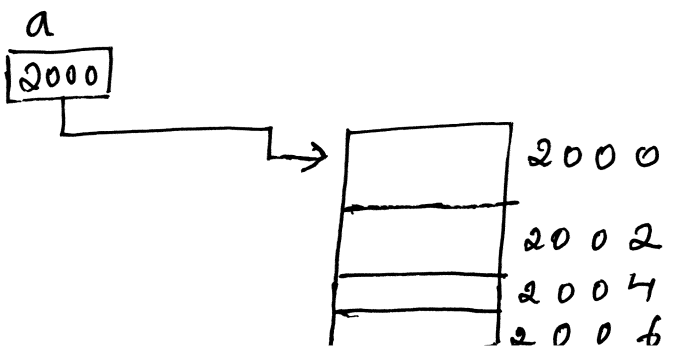
a[5] = '\0' — String is terminated by NULL character.

passing array (using call by reference)

single dimensional array

int a[4];

Note: by default array is a pointer. It holds the base address.



within main function

acceptance of value

scanf ("%d", &a[0])

⇓

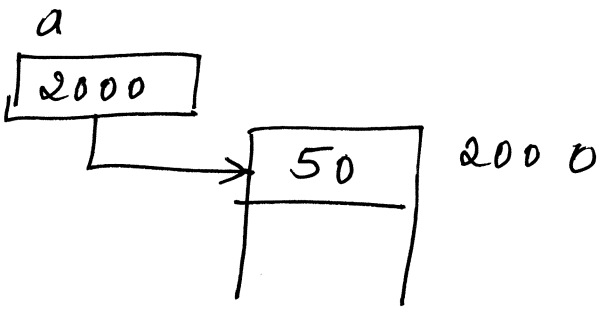
internal execution of &a[0] would be

&(*(a+0))

&(*(2000 + 0)) get the address of

2000+0 = 2000 memory location

50 ⟹



Ex

a[0] internal execution is

* (a + 0) subscript value.

↑ dereferencing base address

a[1]

*(a + 1 * sizeof(int))

*(2000 + 1 * 2) // assume int takes

*(2000 + 2) // two bytes of memory

*(2002)

In scanf we use '&' operator, to insert the value in to the address while retrieving we indicate as it is

`scanf("%d", &a[1])`

⇓

`scanf("%d", 2002)`

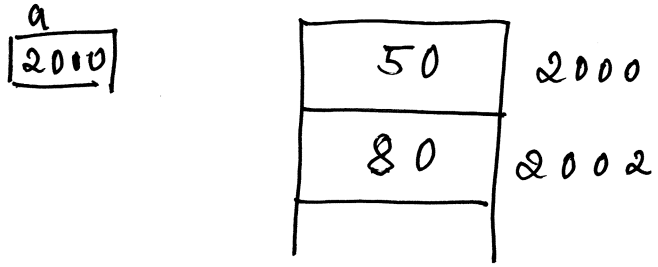
80 — insert '80' into the address 2002

`printf("%d", a[1])`

`printf("%d", *(a+1*2))`

`*(2002)`

retrieve the value from address 2002.



To pass the address of array to a subfunction

Array Name contain the base address, hence array name can be passed from main function

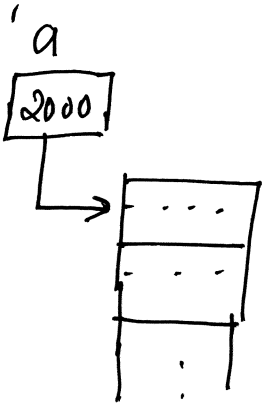
```
int main()
{
  int a[10];
  accept(a);
```

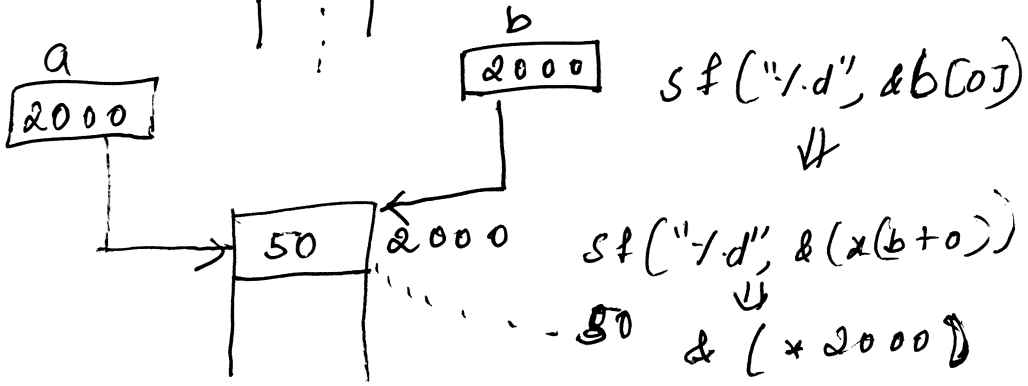
```
void accept(int b[10])
```

b=a

b is an array, it is a pointer, it can hold the address of a.

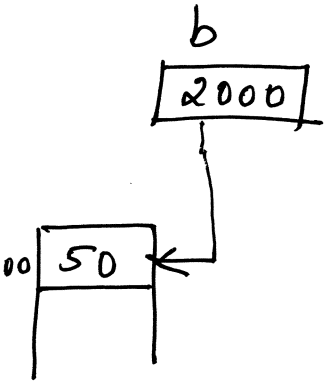
b=2000





To Manipulate values in function, the same procedure must be used as in main funct^

```
pf ("%d", b[0])
      ⇓
pf ("%d", *(b+0))
pf ("%d", *(2000))
```

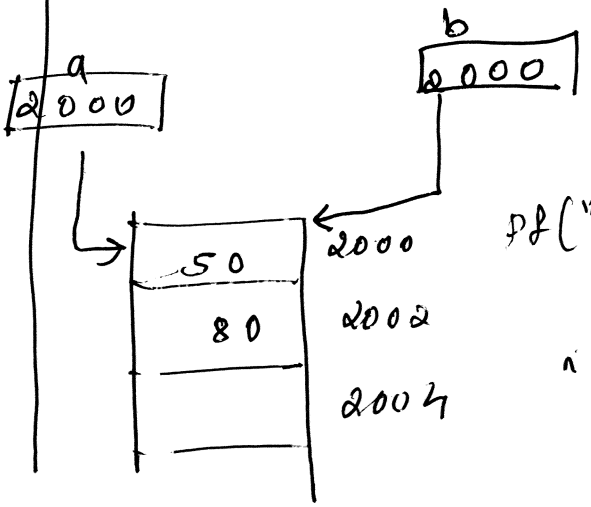


retrieve the value within the address 20000, "50" will be displayed.

Multi dimensional array.

```
int main()
{
 int a[10][10];

 accept(a)
// base address can be
   passed.

// insertn & display in main
sf("%d", &a[0][0])
                50
pf("%d", a[0][0]);
          ⇓
       *(a+0+0)
```

```
void accept(int b[10][10])
{                          b=a
// insertn & disply.
sf("%d", &b[0][1])
        "80"
```



Structure & union *

A structure is a collection of one or more variables possibly of different types, grouped under a single name for convenient access

Ex

```
struct
{ char name[10];
  int age;
  float salary;
} EMP;
```

EMP is unnamed structure, since only one structure of EMP can be used.

Multiple instances of structure can be defined if it has a name.

```
struct EMP
{ char name[10];
  int age;
  float salary;
};
```

```
typedef struct EMP E;
        S1         S2
```

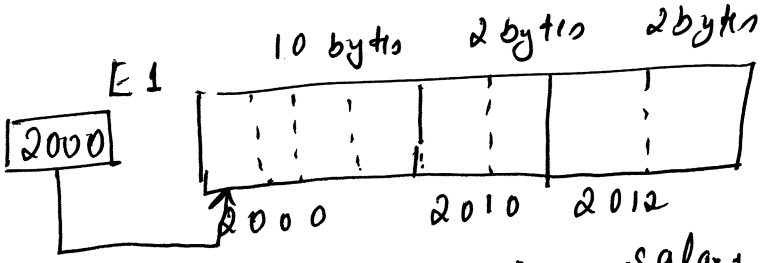
It means where ever E appears, it is replaced with struct EMP.

E E1;

variable name substituted by typedef

⇓

Struct EMP E1;

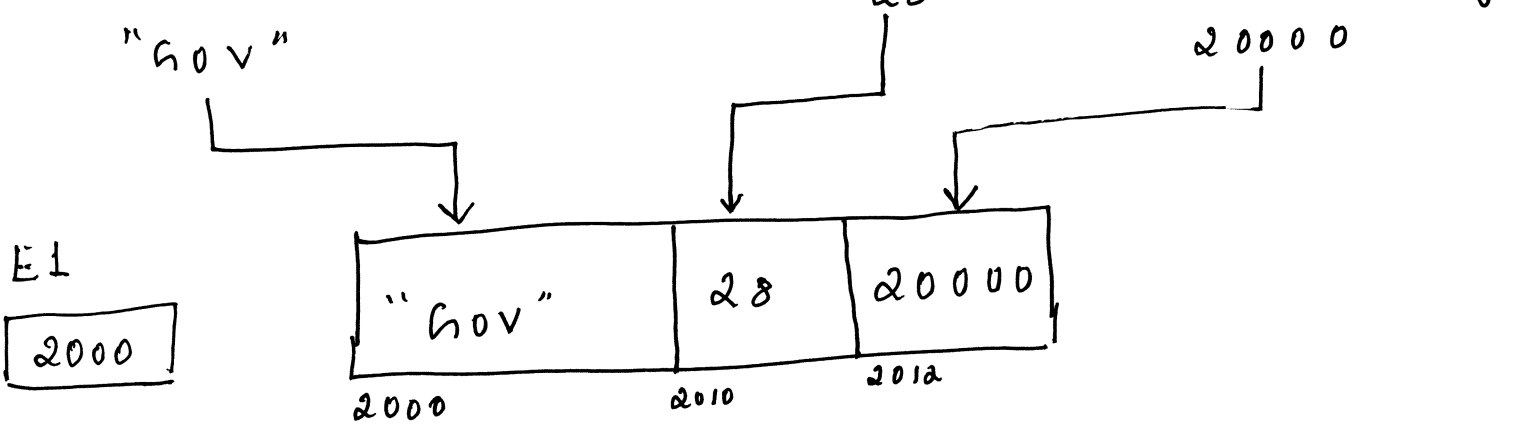


To insert values we use addressing operator

sf("%s", &E1.name) sf("%d", &E1.age)

sf("%f", &E1.salary)

member selector



Note: E1 with respect to structure, symbolizes all elements of structure.

To retrieve the values.

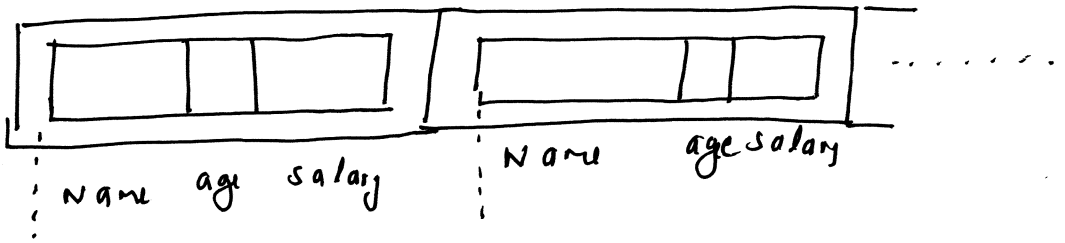
pf("%s", E1.name) pf("%d", E1.age) pf("%f", E1.salary)

"hov" "28" "2000"

Array of structure

E E1[10];

accepts 10 Employee details.



Retrieval

E1[0].Name

E1[0].age

E1[0].salary

Insertion

&(E1[1].Name)

&(E1[1].age)

&(E1[1].salary)

How to pass address of a structure

Single structure

E EM;

// similar to int a;

accept(&EM)

// accept(&a)

void accept(E *E2)
{
// similar to accept(int *P)
sf("%s", E2->Name);
sf("%d", &(E2->age));

Array

E E2[10];

// similar to int a[10]

accept(E2)

// similar to accept(a)

Void accept(E E3[10])
{
//similar to accept(int b[10])
sf("%s", &(E3[0].Name));
sf("%d", &(E3[0].age));

Embedding a structure with in a structure

Ex ~~typedef~~ struct date

```
{
    int month;
    int day;
    int year;
} ~~date~~, typedef struct date D;

~~typedef~~ struct EMP
{
    char name [10];
    int age;
    D dob;     // structure.
} typedef struct EMP E;
```

One of the component of the structure refers to another structure.

<u>Initialization</u>

month, day, year

```
E E2 = { {"Gov", 28, {10, 20, 2000} }
```

name, age, dob

```
E2. name = "Govind";
E2. age = 28;
E2. dob. month = 10;
E2. dob. day = 20;
E2. dob. year = 2000;
```

self referencing structures

is one in which one or more of its components is a pointer to itself.

Ex
```
Struct list
{
  Char data;
  struct list *link;
};
```

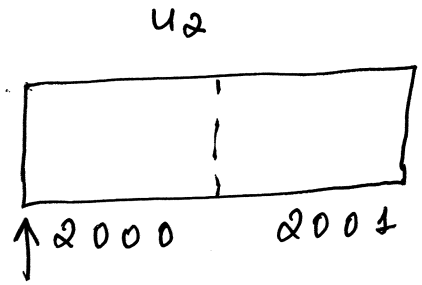
link is a pointer to a list structure.

unions

It is similar to a structure, All fields of union point to the starting address of allocated memory space.

ex
```
union u
{
  int a;          union u u1;
  char c;
}
```

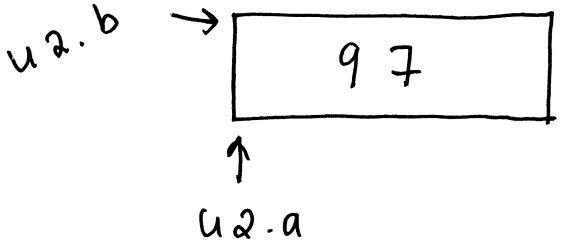
The component which has the highest memory space will be used as the memory space for the Entire union. In the above Example "a" occupies 2 bytes & hence the union 'u' is allocated 2 bytes of memory



u2.a
↑
u2.b

} both components are pointing to the same memory space.

u2.a = 97



pf ("%c", u2.b) Since u2.b will also point to the same memory space, letter 'a' will be displayed. It means in union both or all components of it points to the same memory space.

Difference between Structure & union

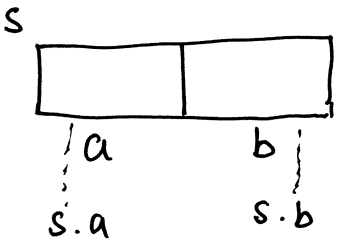
Structure	union
- Total memory size is equal to the sum of all the individual members within a structure	- Total memory space allocated is equal to the member with larger size

Structure

→ All members will point to different memory allocations

Ex

```
struct {
    int a;
    char b;
} s;
```



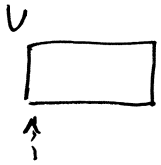
size = sizeof(a) + sizeof(b)

- Since each location address is different, & can be used for saving information for Efficient access.

union

→ All members will point to same memory allocation.

```
union {
    int a;
    char b;
} u
```



u.a or u.b

sizeof(a) > sizeof(b) & hence size = sizeof(a);

- provides an efficient way of using the same memory location for multiple-purpose

String processing

Character set.

Alphabets: A B C D E ... .

Digits : 0 1 2 3 4 5

special characters : + − / * ( )

A finite sequence of S of Zero or more characters is called a **string**.

char a[10] = {"INDIA"}

No of characters in the string is called its **length**

a[10] = "INDIA"

5 char length

char b[10];

b has not been initialized so it is referred as **Empty string**

In some languages single quote can also be used.

Note: Single quote can be used to represent a single char in 'c' char c = 'a';

Storing String

stored in three types of structures

(i) Fixed length structures

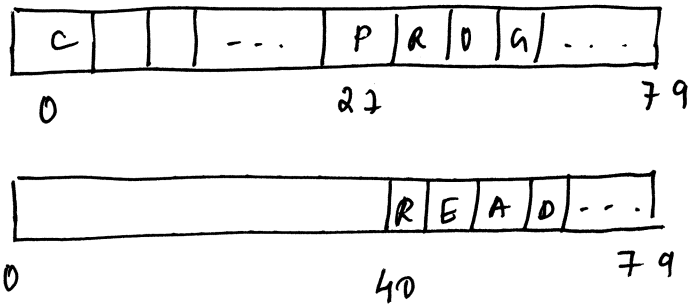
(ii) Variable length structures

(iii) Linked structures

Fixed length

- record oriented
- each line is viewed as a record where all records have the same length

80 length char (fixed for all lines)



Advantage: Easely accessable, updating is easy

Disadvantage: Time is wasted reading an entire record. If most of them contain blank spaces.

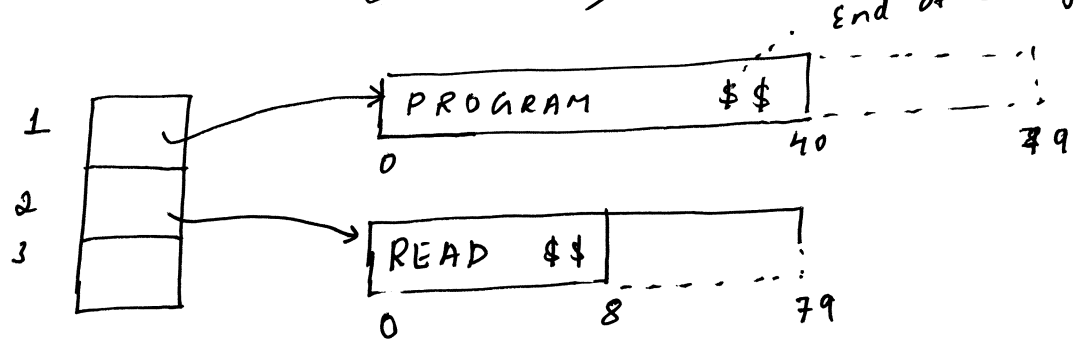
- certain records may require more space than available.
- length of the input string is too small, memory gets wasted.

→ changing a word requires the entire record to be changed.

Variable Length storage with fixed maximum

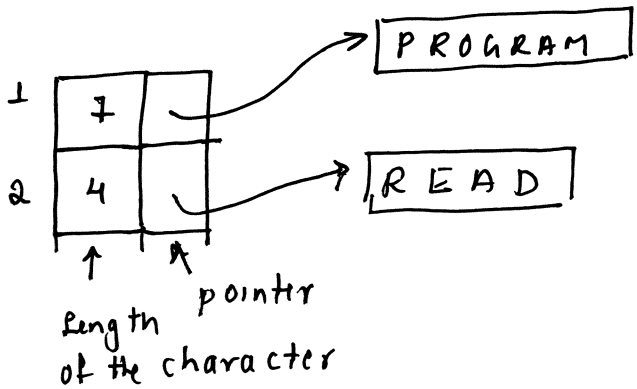
storage of variable length string in memory cells with fixed length can be done in two general ways

i) using a marker (delimiter)



A marker is mentioned at the end of string but entire memory is allocated.

ii) The length of the characters is stored within the ~~delimiter~~. table

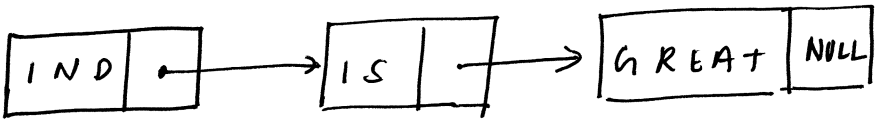


Linked storage

In word processing, string information is saved, processed & displayed. This can be done easily ~~in the~~ using linked list.

Linked list : linearly ordered sequence of memory cells called nodes.

Each node contains an item called a link which points to the next node in the list.



Each memory cell is assigned one character or a fixed no of character.

String operations

i) sub string : Access a substring from a given string

substring (string, initial, length)

length of the substring.

string itself

position of the first character of the substring.

Ex substring ("INDIA IS GREAT", 7, 8)

IS GREAT

PL/I : SUBSTR (S, 4, 7)

FORTRAN: S(4:10)

ii) Indexing : refers to finding the position where a string pattern P first appears in a given string text T.

INDEX (text, pattern)

INDEX("INDIA IS GREAT", "IS")

7

PL/1:
INDEX(text,pattern)

PASCAL:
POS (pattern, text)

III) Concatenation: is the operation of joining two strings together.

format S1 // S2

S1 = "INDIA" S2 = "GREAT"

S1 // "IS" // S2

⇓

INDIA IS GREAT

PL/1: S1 || S2

FORTRAN: S1 // S2

IV) Length: is to find no of characters in a string

format: LENGTH(string)

LENGTH("INDIA")

5

PL/1: LENGTH (string)

PASCAL: — ⊥ — ⊥ —.

String handling function in C

1) int strlen(char str[]):

Ex

strlen("INDIA")

↓

5

2) int strcpy(char dest[], char src[]):

Copies the src string to dest including '\0'

char s1[] = "INDIA"

char s2[20];

strcpy(s2, s1)

3) int strcat(char s1[], char s2[]):

Copies all characters of s2 to s1

char s1[] = "INDIA"

char s2[] = " GREAT"

strcat(s1, s2)

s1 = "INDIA GREAT"

4) strcmp(s1, s2)

returns 0 if s1 = s2

+ve if s1 > s2

-ve if s1 < s2

5) strrev(str)

strrev("INDIA")

pattern matching algorithm

finds a given pattern P in a string text T. Ps length must not exceed the length of T.

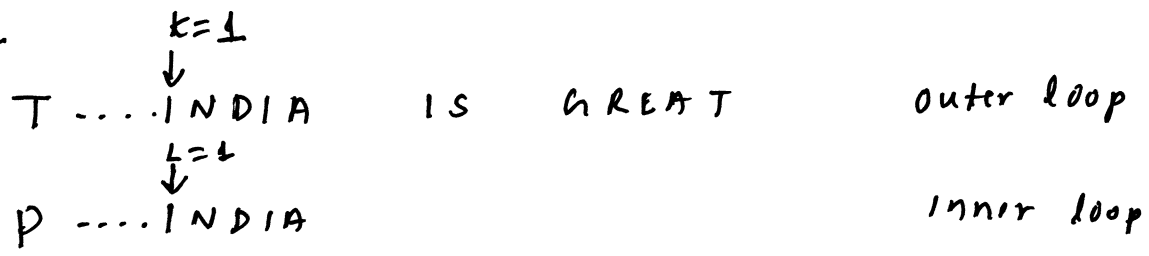
(1) window technique or brute force method

Algorithm

pattern matching P & T are string with lengths R & S, respectively & are stored as arrays with one character per Element. This algorithm finds the index (INDEX) of P in T.

1. [Initialy] set K := 1 & MAX := S - R + 1
2. Repeat steps 3 to 5 while K ≤ MAX
3. Repeat for L = 1 to R: [Test each char of P]
   If P[L] ≠ T[K + L - 1] then goto step 5
   [End of inner loop]
4. [Success] set INDEX = K & EXIT
5. set K := K + 1
   [END of step 2 Outer Loop]
6. [Failure] set INDEX = 0
7. EXIT.

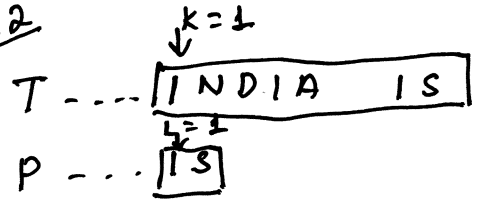
Ex 1



Compares Every Element

If it successfull INDEX = K & Exit

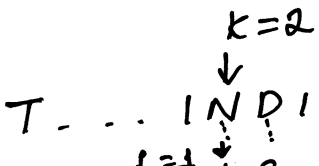
Ex 2



① ✓ N ≠ unsuccessfull
① ✓ S ≠

Inner loop

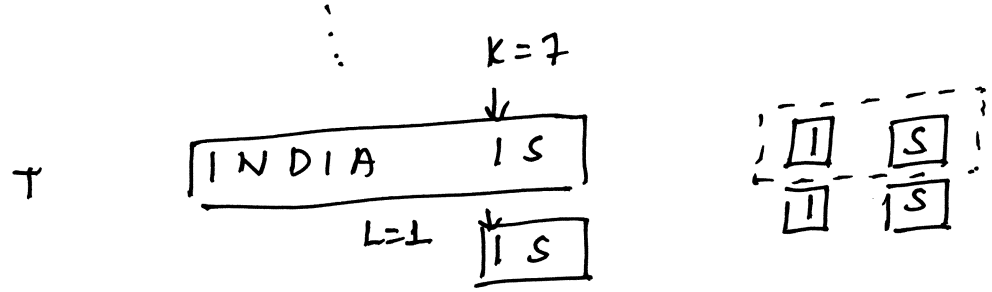
Increment k = 2.



L=1 will get reinitialized

unsuccessfull

Increment k = 3.



successfull

return k = 7

② pattern matching algorithm

This algorithm works on the basis of creating a state diagram & the table

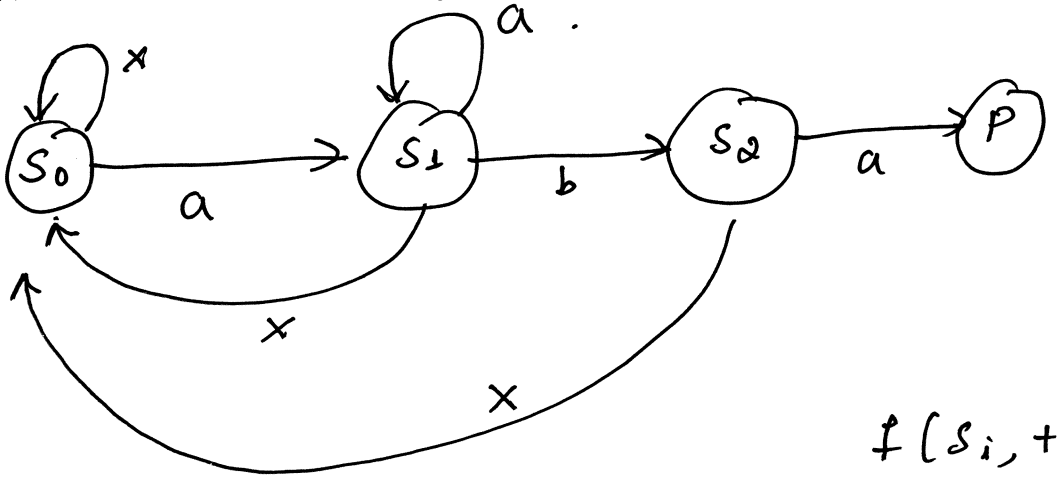
→ The table is derived from a particular pattern P but is independent of the text T.

→ consider the pattern P = "aba" & the text has one of the following form

a) T = aab... b) T = aaba c) T = aX

→ "X" is a character different from 'a' or 'b'

→ The pattern matching graph for the above pattern is as given below (P = "aba")



f(Si, t)	a	b	X
S0	S1	S0	S0
S1	S1	S2	S0
S2	P	S0	S0

f(Si, t) accepts the state & single Character of text

Algorithm (pattern Matching)

The pattern matching table $F(S_i, T)$ of a pattern P is in memory, & the input is an N - character string $T = T_1, T_2 .. T_N$. This algorithm finds the index of P in T

1. [Initialize] set $k := 1$ & $S := S_0$
2. Repeat step 3 to 5 while $S \neq P$ & $k \leq N$
3. Read T_k
4. set $S := F(S, T_k)$ [Find the next state]
5. set $k := k + 1$ [update counter]

[End of step 2 loop]

6. [Successful ?]
   If $S = P$ then INDEX = k - LENGTH(P)
   Else
   INDEX = 0
   [End of If structure]

7. EXIT.

Different memory allocation functions

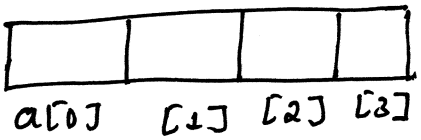
(1) Static allocation – Memory space is allocated during ~~Execution~~ copilation.

– Memory space cannot be changed.

cannot be increased nor decreased.

Ex

int a[4]

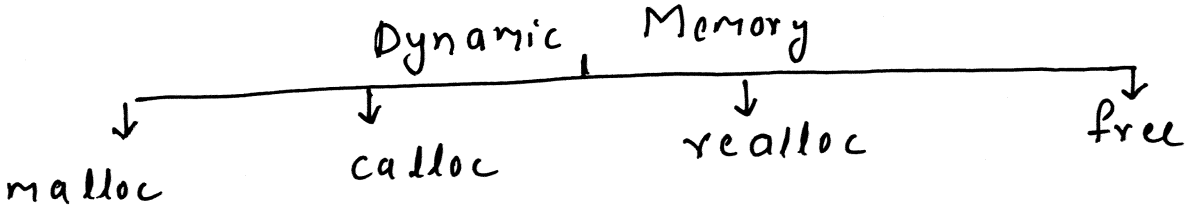


Disadvantage is that memory gets wasted when it is not utilized completly.

(2) Dynamic memory allocation

process of allocating & deallocating memory during the execution of the program is called dynamic memory allocation.

'C' provides four library functions under "stdlib.h" for dynamic memory allocation



malloc (memory allocation)

- allocates a block of memory as specified in size

prototype `#include <stdlib.h>`

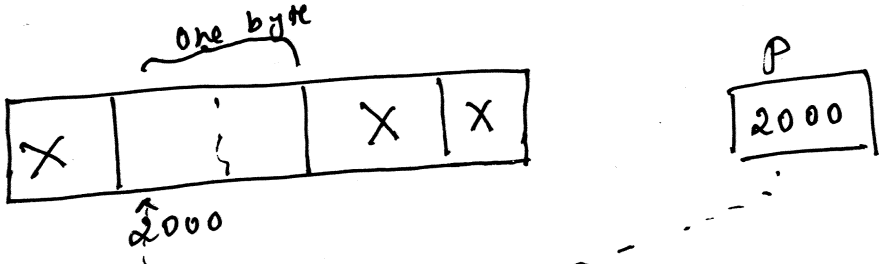
`void * malloc (size_t size);`

- returns the address of first byte of allocated memory or NULL if the memory cannot be allocated.
- memory is not initialized.

→ when memory is allocated then the return type must be typecasted to the respective pointer.

Ex `int *p;`

`p = (int *) malloc (sizeof (int))`



If memory is not allocated

p = NULL.

calloc

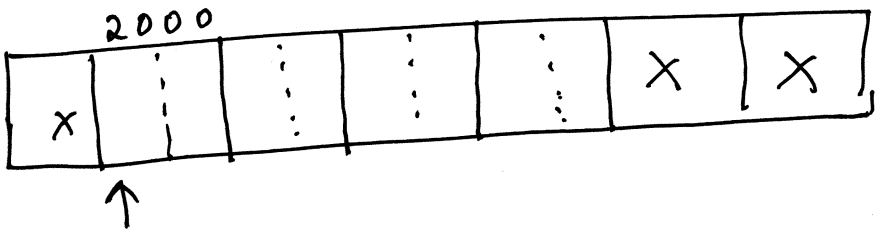
- used for allocating multiple blocks

- contiguous allocation of multiple blocks

→ used for allocating arrays

→ Memory values are initialized to zero

```
int *ptr;
ptr = (int *) calloc (4, sizeof(int));
```



ptr = 2000

prototype

```
#include <stdlib.h>
void * calloc(size_t num, size_t size)
```

num indicates the number of blocks of size which is passed as arguments

insertion to memory

```
scanf("%d", ptr+0)
```

retrieval

```
printf("%d", *(ptr+0))
```

Difference between malloc & calloc

malloc	calloc
- memory allocation may or may not be continious	contiguous allocation of multiple blocks
void * malloc(size_t size)	void * calloc(size_t num, size_t size)
- accepts a single argument size	accepts two arguments num & size
Total size allocated will be equal to the size argument	Total size size = num * size.
- memory is not initialized	memory is initialized to zero.

realloc

is used for modifying the size of the allocated block. which is allocated by either malloc() or calloc()

prototype

```
#include<stdlib.h>
void * realloc( void * ptr, size_t size)
```

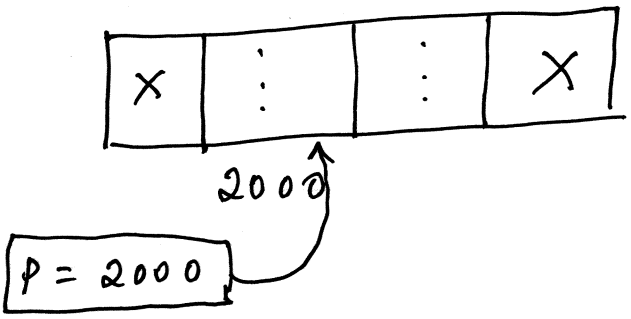
realloc is used to increase the allocated memory.

realloc changes the size of the block by extending or deleting the memory at the end of the block.

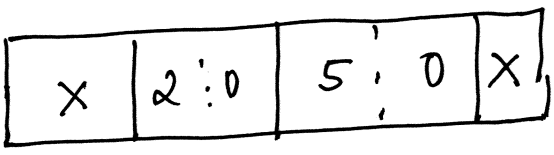
- ptr will not get changed if the memory is available within the same space.
- ptr will get changed if the space is not sufficient

Note: previous contents will be restored.

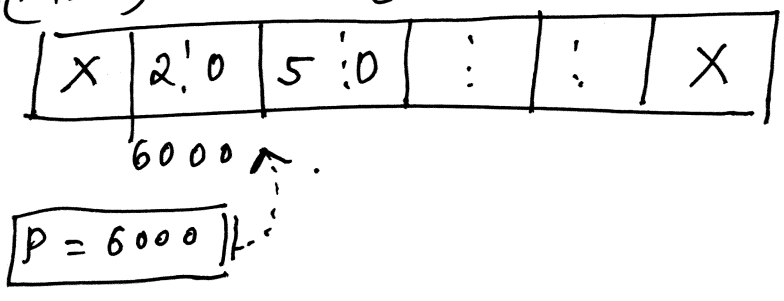
```
int *p;
p = (int *) malloc( sizeof (int) * 2)
```




```
*(p+0) = 20;
*(p+1) = 50;
```




```
p = (int *) realloc ( p, sizeof (int) * 4)
```



size is filled & gets allocated in a different memory space.

free

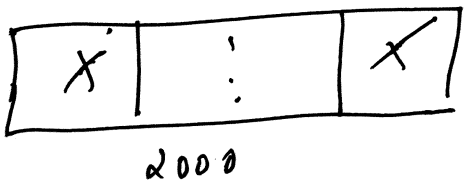
is used for deallocating memory.

```
#include <stdlib.h>
void * free ( void *ptr)
```

accepts the pointer of the previously allocated memory.

Ex

```
int *ptr:
ptr = (int *) malloc (sizeof (int)):
```



// memory gets // allocated

```
ptr = 2000
*(ptr+0) = 50
```

```
free (ptr)
```

// memory gets deallocated.

polynomials

Abstract data type

is a mathematical model for data types, where data type is defined by its behaviour

Ex Day of week (S, M, T, W, Thu, F, Sat)

operations

- reading - retriving i^{th} Element
- adding $i^{th} + 2$

polynomials is a sum of terms each term has a form

$a x^e$

x is the variable
a is the coefficient
e is the exponent.

$$A(x) = 3x^5 + 2x^4$$

The largest exponent of a polynomial is called its degree.

Addition of two polynomials

$$A(x) = \sum a_i x^i$$

$$B(x) = \sum b_i x^i \quad \text{then}$$

$$A(x) + B(x) = \sum (a_i + b_i) x^i$$

$$A(x) = 3x^5 + 2x^4$$

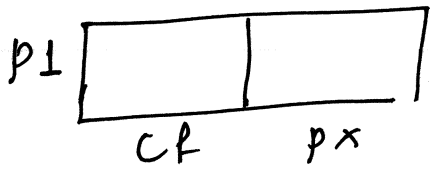
$$B(x) = 6x^4 + 3x^3$$

$$3x^5 + 8x^4 + 3x^3$$

In c language structure can be used to define the polynomial.

```
struct POLY
{
    int cf;    // coefficient
    int px;    // power
};
typedef struct POLY p;
```

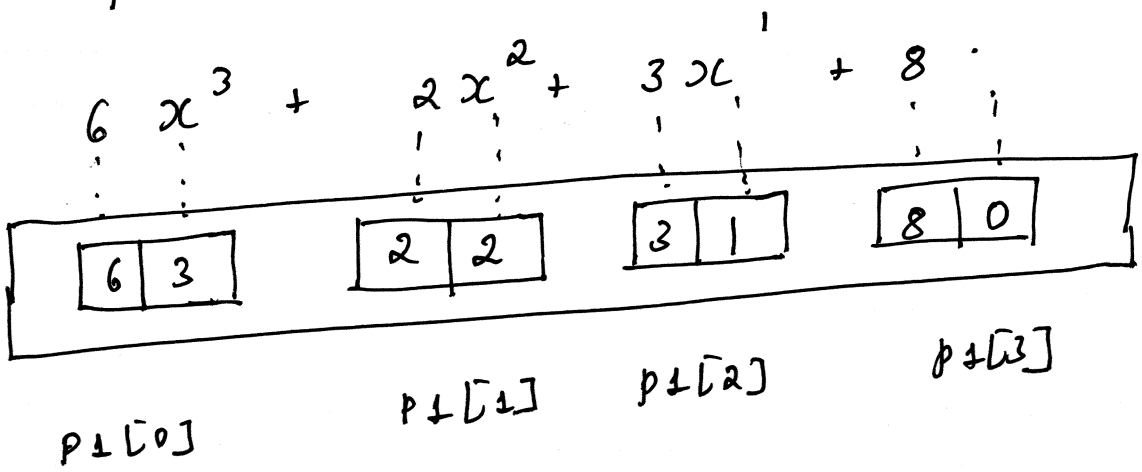
p p1



p1.cf } to retrieve
p1.px

This can be used for defining a single term, to define multiple terms an array is defined

p p1[4]



Sparse matrix

A sparse matrix is a matrix in which most of the element are zero

$$A = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 8 \end{bmatrix}$$

Nonzero elements = 3

Zero elements = 6 . & hence it is sparse matrix or else it is referred as dense

Representation of sparse

The following representatn can be used for representing a single element of a sparse matrix

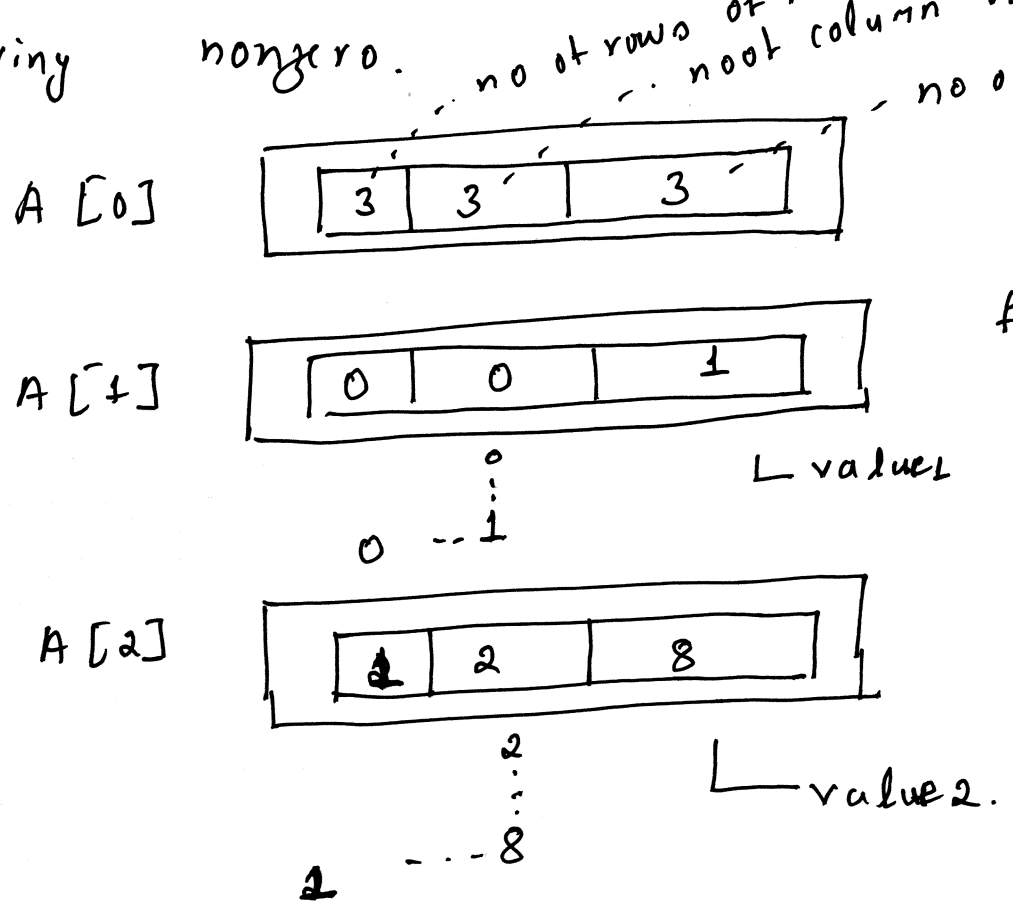
```
struct sparse
{
    int row;    // row no is saved
    int col;    // col no
    int value;  // the actural value
};
Typedef struct sparse S;
```

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 8 \\ 0 & 0 & 0 \end{bmatrix}$$

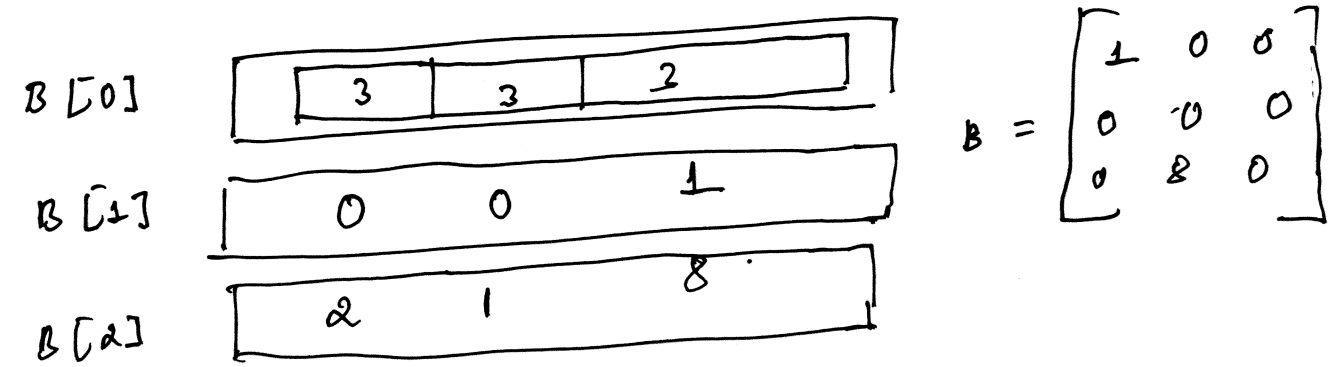
This is how, the above matrix is saved

```
S  A[10];
```

A can save 10 elements, in the above matrix we have only two elements having nonzero.



The resultant after transpose.



Module 2

Stacks & Queues.

Kiran P
3 sem CS

D.S

A stack is an ordered list in which insertion (called push) and deletion (called pop) are made at one end called top.

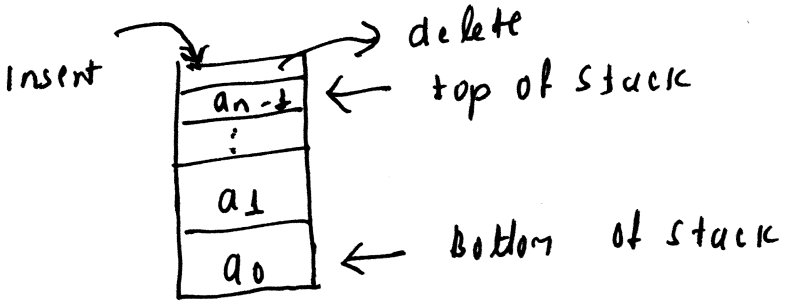
push: is the term used to insert an Element into stack.

pop: is used to delete an Element from a stack.

- It is a linear data structure.

stack is also called Last in first out D.S.

$S = \{ a_0, a_1, \ldots, a_{n-1} \}$



Stack operations

① push – Insert Element

② pop – Delete Element.

③ Display content of stack.

Algorithm

push (stack, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a stack.

1. [stack is already full ?]

   If TOP = MAXSTK

   Then print overflow & return.

2. Set TOP := TOP + 1

3. Set STACK [TOP] = ITEM

4. Return.

Algorithm

pop (STACK, TOP, ITEM)

This procedure deletes the top element of STACK & assign it to variable ITEM

1. [Stack has an item to be removed]
   If TOP := 0 then print underflow & return.

2. Set ITEM := STACK [TOP]

3. Set TOP = TOP − 1

4. Return.

Applications of Stack

- Conversion of Expressions:
  Infix expression will be converted to appropriate expression by compiler using stack.

- Evaluation of Expressions: postfix, prefix or infix Expressions can be evaluated (checked for correctness)

- Recursion:
  - checking palindrome.

Algebraic Expression

is a legal combination of operators and operands.

Operands may be a variable x, y, z or a constant 5, 4, 6 etc.

An algebraic expression can be represented using three different notation.

(i) Infix it is the form of an arithmetic expression in which operator is in between operands.

Ex (A + B) * (C - D)

(ii) prefix: operator is before the two operands

- also called "polish notation"

* + AB - CD

(iii) postfix: operator is after the two operands

AB + CD - *

Precedence & Associativity of the operators.

precedence rule: defines the order in which different operators are evaluated.

		priority	Associativity
Exponential	(↑, ^)	6	Right to left
Multiplication	(*)	4	Left to right
Division	(/)	4	
Mod	(%)	4	
Addition	(+)	2	Left to right
Subtraction	(−)	2	

Symbol	Stack precedence function F	Input precedence function G.
	2	1
+, −	4	3
*, /, %	5	6
$, ^	8	7
operand	0	9
(	−	0
)	−1	
#		

A palindrome is a word number, phrase or sequence of characters which reads the same backward as forward

Ex 1 MADAM — (a)

reverse MADAM — (b) a & b both are equal & hence palindrome

Ex 2. INDIA — (a)

reverse AIDNI — (b) a & b both are different & hence not a palindrome.

function to check for palindrome

```
int check_palindrome(char str[])
{
    int i, top
    char s[20]
    top = -1;
    // push all the characters
    for (i=0; i< strlen(str); i++)
        push(s, str[i], &top)
    // pop and check char from begining
```

```
for (i=0; i<strlen(str); i++)
    if (str[i] != pop(s, &top))
        return -1;
return 1;
}
```

procedure to convert from infix to postfix

As long as the precedence of symbol on top of the stack is greater than the precedence of input symbol, pop the element and place it in the postfix expression.

```
while F(s[top]) > G(symbol)
    postfix[j++] = s[top--]
```

If the precedence of the symbol on top of the stack is not equal to the precedence of the input symbol, push the symbol on to the stack.

Else delete the element from the stack

```
if F(s[top]) != G(symbol))
    s[++top] = symbol
else
    top--;
```

Queue

is a specialized type of Data structure where elements are inserted from one end called rear & deleted from one end called front.

First element inserted is the first element which gets deleted.

FIRST IN FIRST OUT Data structure

Types of Queues

Linear Queue

Circular Queue

Double Ended Queue

priority Queue.

Linear Queue

- It is a linear D.S
- It is also referred as Ordinary Queue

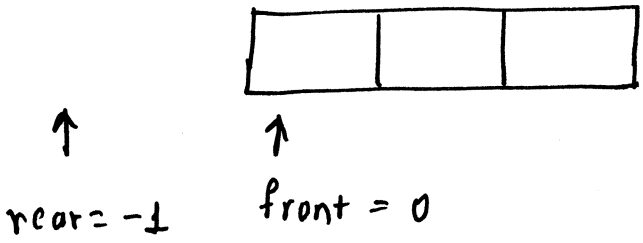
Operations of queue

(I) Insert

(II) Delete

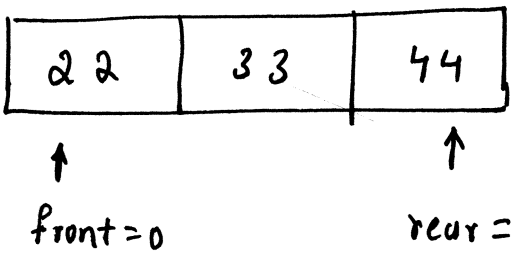
(III) Display

Initialization.



Condition to Check Empty Queue

```
If ( rear < front )
    pf ( " under flow")
```



Condition to Check Queue is full

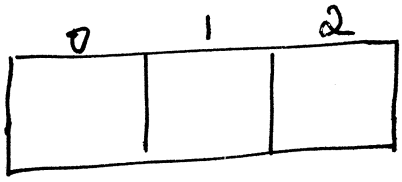
```
If ( rear == MAX_ELEM_QUEUE -1)
```

Circular Queue

To overcome the disadvantages of Linear Queue, Circular Queue can be used.

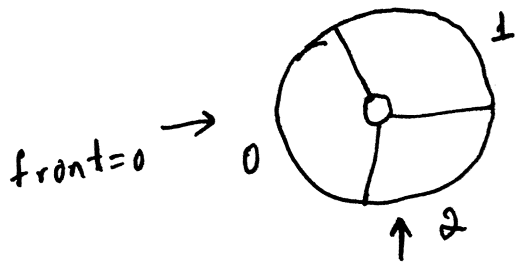
Advantage of circular Queue is that it allows the entire array to store the elements without shifting any data with in the queue.

For ordinary Queue, initialization is



rear = -1 front = 0

Circular as the name specifies, it is interrelated or "previous to front"



rear = 2

rear = ME − 1

{maximum element of Queue} − {1}

Operations on circular Queue

→ Insertion

→ Deletion

→ Display

Note
To check the no of Elements in the Queue a saperate counter variable must be used. initial value count = 0

Insertion - inserting Elements, insertion is done in the rear end.

→ check for overflow condition,

if (count == Max_ elem_ Queue)

→ Else
Insertion is possible.

rear value is incremented and modulous is applied to keep the value within the range of the maximum Queue length

initial value rear = 2

Insert = '22'

count = 0 // Its not a overflow condition

rear = rear + 1

= 2 + 1

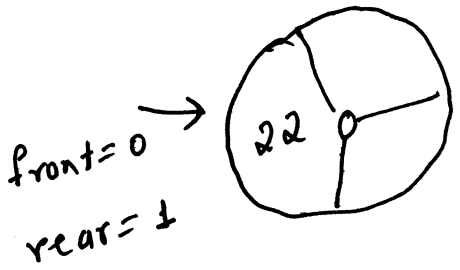
rear = 3

// take modulous of ME

rear = rear % MAX_Eleme__Queue

= 3 % 3

= 0



Q[rear] = 22

Increment count

count = 1

Insert '33'

count = 1 // not a overflow condition

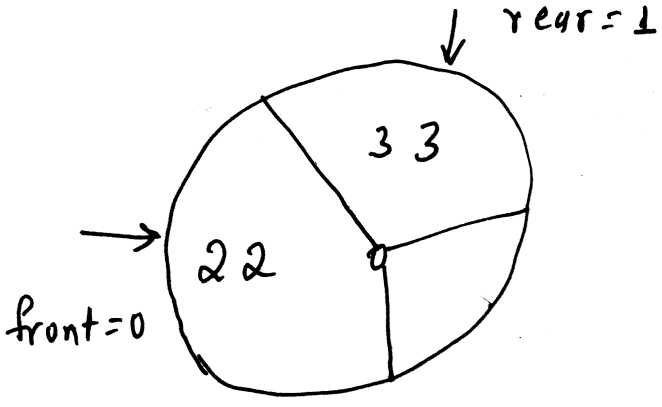
rear = rear + 1

= 0 + 1

= 1

= 1 % 3

= 1



Q[rear] = 33

Increment count count=2

Insert 44

count = 2

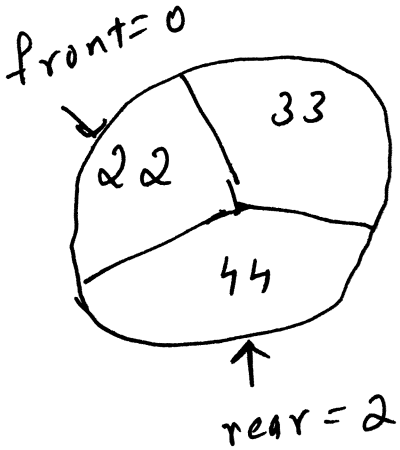
rear = rear + 1

= 2 % 3

= 2

a[rear] = 44

count = 3



Insert 55

count == MAX_ELEM_QUEUE

// insertion is not possible.

Deletion

- Removing Element from the circular Queue.
- deletion is always performed at the front end.

→ similar to insertion, to keep the front value with in the range of Queue., modulous operator is used

Check for underflow or Queue empty

If count == 0

Then Queue in empty

return.

Else

display the Element

increment the front with modulous

operation

Delete.

count == 3 // 1th not empty.

~~front = 0~~ front = 0

elem = ~~Q[front]~~ Q[front]

= Q[0] // 22

~~rear = rear~~

front = (front + 1) % MAX_ELE_QUEUE

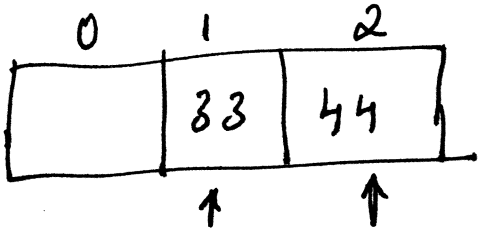
= (0 + 1) % 3

= 1 % 3

= 1.

*** Decrement count

count = 2



front = 1 rear = 2

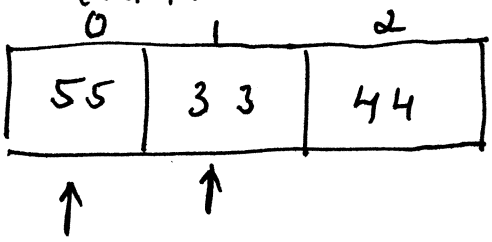
Insert 55

count = 2

rear = (2 + 1) % 3 = 0

Q[rear] = 55

count = 3



front = 1

rear = 0

Delete

count = 3

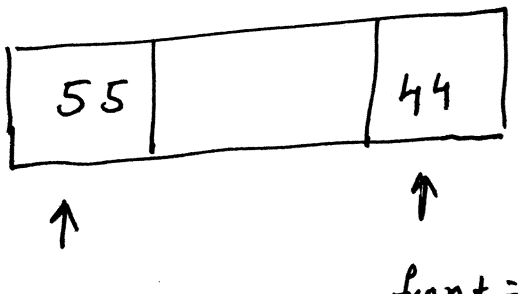
front = (front + 1) % 3

= 2 % 3 = 2

Elm = Q[front] 1133

count must be decremented

count = 2.



rear = 0

front = 2

Double Ended Queue

→ It is also referred as Dequeue or Deck.

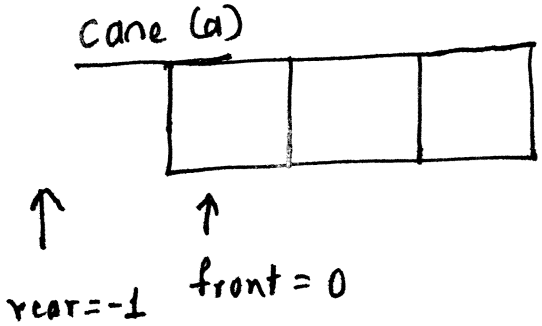
→ This is a special type of DS in which insertion and deletion are done from both ends.

Operations

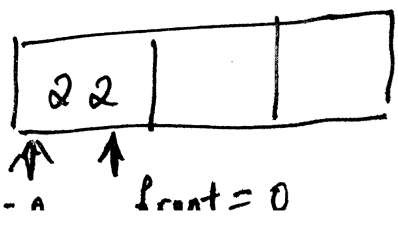
* Insertion of an item from front end
* — " — " — rear end
* Delete an item from front end
* — " — " — rear end

Insertion from front end

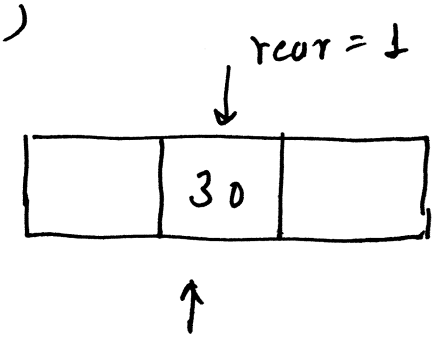
Case (a)




```
if front == 0 && rear == -1
Q[ ++ front ] = item;
```



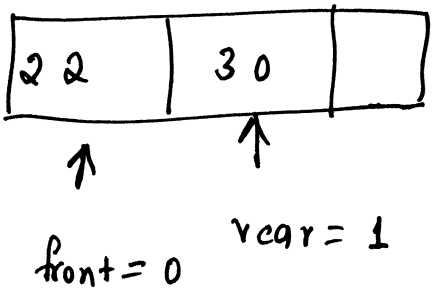
Case (b)



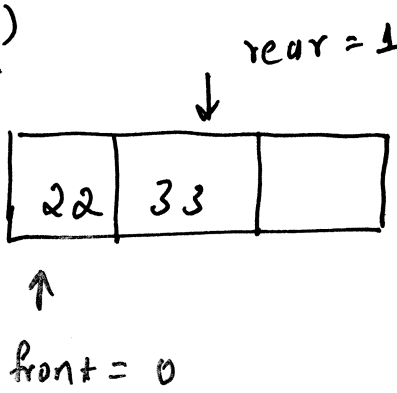
if f != 0

q[--f] = item

return.



Case (c)



Front Insertn cannot be done.

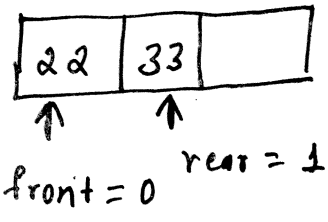
Deletion from rear

a) Check for Empty Queue

if front > rear // Empty Queue.

return -1;

b)

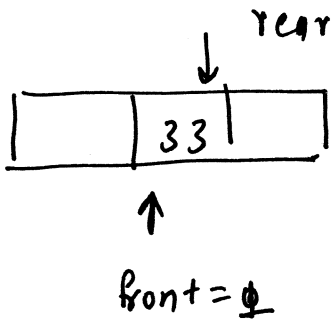


deletion can be performed

Elem = Q[rear] //33

rear = rear - 1

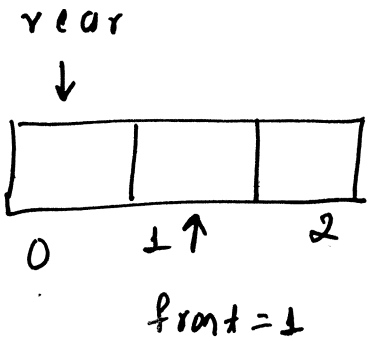
c)



only one Element

Elem = Q [rear] // 33

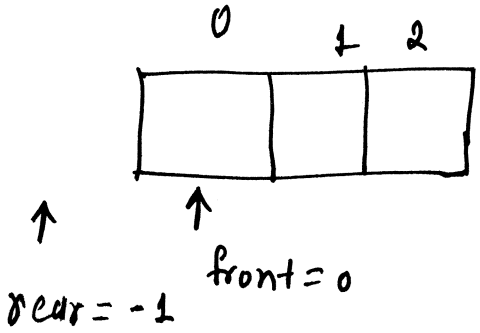
rear = 0



After deletion if

if rear ≠ front

then it is equivalent to empty Queue. & hence both the values are reinitialized to 0



rear = -1

front = 0

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and

processed comes from the following rules

i) An element of higher priority is processed before any element of lower priority

ii) Two elements with the same priority are processed according to FIFO

There are various ways of maintaining priority queue.

① One-way list representation of a priority queue.

a) Each node in the list will contain three items of information

INFO - information

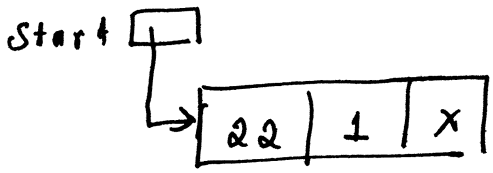
PRN - priority

LINK - link to the next node.

(b) A node 'X' precedes a node 'Y' in the list (i) when 'X' has higher priority than 'Y' or (ii) when both have the same priority but 'X' was added to the list before 'Y'.

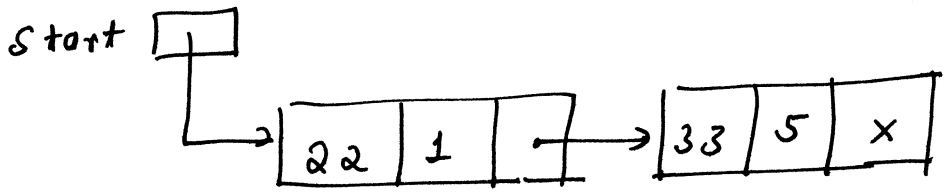
① INFO - 22

PRN - 1



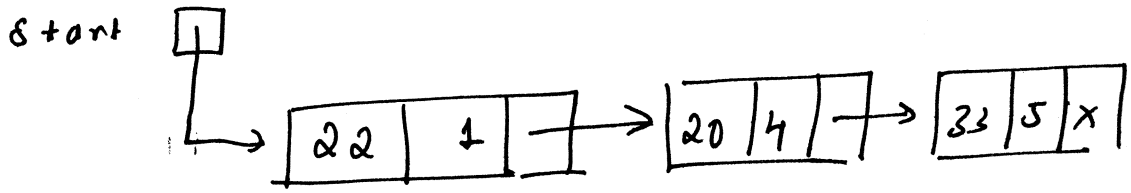
② INFO - 33

PRN - 5



③ INFO = 20

PRN - 4



*** Lower priority number has the highest priority.

② Array representation of a priority Queue.

→ use a separate Queue for each level of priority

→ Each Queue will appear in its own circular array and must have its own pair of pointers

Rowno	Front	Rear
1	2	2
2	1	3
3	1	3

1	2	3	4
	AAA		
BBB	CCC	XXX	

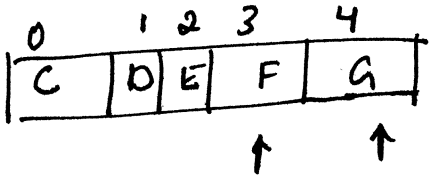
Circular Queue using dynamically allocated Arrays

Let "capacity" be the number of position in the array queue.

To add an Element to a full Queue

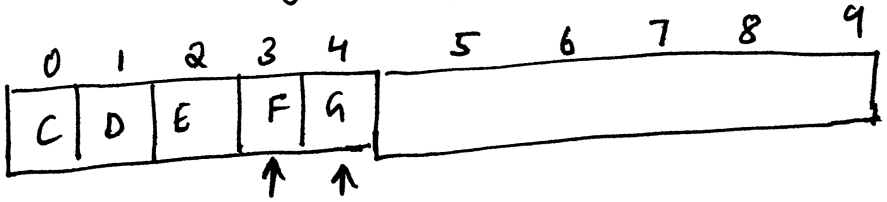
→ first increase the size, realloc

we use array doubling



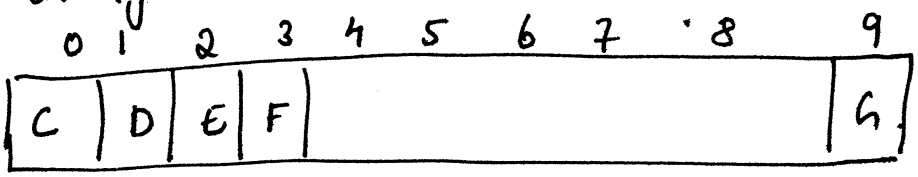
rear = 3 front = 4

→ array doubling is done

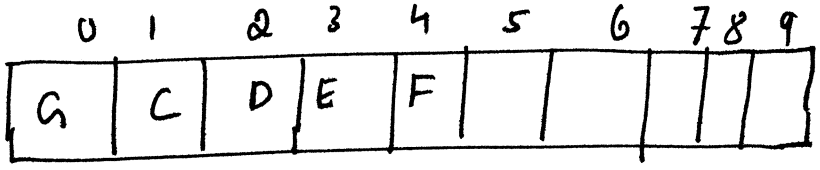


rear=3 front=4

→ Slide the Elements in the right segment to the right of the array



→ Alternative configuration must be made.



front = 0 rear = 4

Following steps are used for dynamic memory allocation.

i) Create a new array newQueue of twice the capacity.

ii) Copy the second segment queue[front] through queue[capacity-1] to position in newQueue

iii) Copy the first segment

Recursion

- can be defined as a function which calls itself directly or indirectly.
- certain problems can be solved easily using recursion.

There are two types of recursion

(a) Direct recursion

(b) Indirect recursion.

Direct recursion

A recursive function which invokes directly

Ex

```
int fact (int n)
{
    if n==0 return 1
    return n * fact(n-1);
}.
```

Indirect recursion

A function which calls some other function & intern it calls the first function.

Recursive function must have the following two properties.

(i) There must be some base criteria, for which the function does not call itself.

(ii) Each time the function does call itself, it must be closer to the base criteria.

A recursive procedure with these two properties is said to be well-defined

Factorial Function

Definition:

Factorial

(a) if $n = 0$ then $n! = 1$

(b) if $n > 0$ then $n! = n * (n-1)!$

$n!$ is recursive

(1) $3! = 3 \cdot 2!$

(2) $2! = 2 \cdot 1!$

(3) $1! = 1 \cdot 0!$

$0! = 1$

(4)

(5) $1! = 1 \cdot 1 = 1$

(6) $2! = 2 \cdot 1 = 2$

(7) $3! = 3 \cdot 2 = 6$

Factorial (FACT, N) .. Iterative approach

1 if N = 0 then set FACT = 1 and Return

2. set FACT = 1

3. Repeat for K = 1 to N

Set FACT = K * FACT

[End of loop]

4. Return.

FACTORIAL (FACT, N) ... recursive approach

1. if N = 0 then set FACT = 1 & Return.

2. call FACTORIAL (FACT, N - 1)

3. set FACT = N * FACT

4. Return

Fibonacci Sequence

is denoted as follows

0, 1, 1, 2, 3, 5, 8, ...

ie $F_0 = 0$ and $F_1 = 1$ and each succeding term is the sum of the two preceding terms.

A formal defenition is

Definition

(a) if $n = 0$ or $n = 1$ then $F_n = n$

(b) if $n > 1$ then $F_n = F_{n-2} + F_{n-1}$

base values are 0 and 1

FIBONACCI (FIB, N)

1. if $N = 0$ or $N = 1$ then set FIB = N & return.

2. call FIBONACCI (FIBA, N-2)

3 call FIBONACCI (FIBB, N-1)

4. set FIB = FIBA + FIBB

5. Return.

Divide and conquer algorithm.

is a design paradigm, it works by recursively dividing a problem in to sub problems of the same or related type.

Algorithm solves a problem using following three steps

i) Divide: break the problem into subproblem of the same type

ii) Conquer: Recursively solve these subproblems

iii) Combine: Combine the results

Akerman Function

is a function with two arguments each of which can be assigned any non-negative numbers 0, 1, 2 ...

Definition (Akerman Function)

(a) If $m = 0$ then $A(m,n) = n + 1$

(b) If $m \neq 0$ but $n = 0$, then $A(m,n) = A(m-1, 1)$

(c) If $m \neq 0$ and $n \neq 0$ then $A(m,n) =$

$A(m-1, A(m, n-1))$

Tower of Hanoi

There are three pegs say A, B and C. Discs are placed in decreasing order of there size. In A, The objective of the game is to move the discs from peg A to peg C using peg B as an auxilary (temp)

c function to implement tower of hanoi

```
void transfer (int n, char s, chat t, chor d)
{
    if (n==0) return
        tranmfer( n-1, s, d, t)
        printf("Move disc %d from %c to
                %c \n", n, s, d) :
        transfer( n-1, t, s, d) :
}
```

List refers to a linear collection of data items.

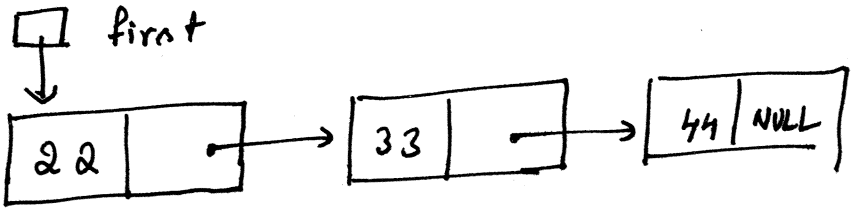
A linked list, or oneway list, is a linear collection of data Elements called nodes, where the linear order is given by means of pointers.

Each node is divided into two parts

First part contains the information of the Element

Second part called the link field, contains the next node in the list.

Ex



Types of linked lists

i) singly linked list – only one link

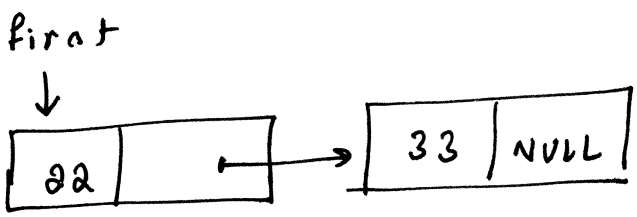
ii) Doubly linked list → two link

iii) Circular singly linked list

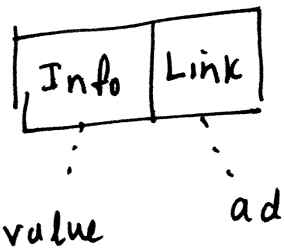
Singly linked list

is a collection of zero or more nodes where each node is connected to ~~one~~ next node using address. Address represents pointer.

Ex



How to create a node



Declaration

```
struct node
{
  int info;
  struct node * link;
};
typedef struct node * NODE;
```

Creating Empty list

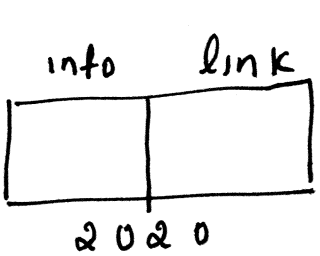
- It is done by assigning to NULL.

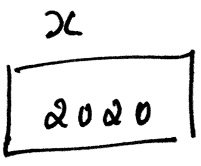
```
NODE first;
first = NULL;
```

Create a node

- is created using dynamic memory allocation.

```
NODE x;
x = (NODE) malloc( sizeof ( struct Node));
if (x == NULL)
{
    printf (" Memory is full")
    Exit(0);
}
```



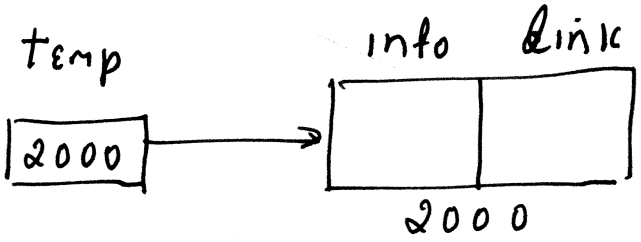


x = 2020

```
NODE    getnode()
{
        NODE  x;
        x = (NODE) malloc(sizeof(struct node));
        if (x == NULL)
        {
            printf("Memory full")
            exit(0);
        }
        return x
}
```

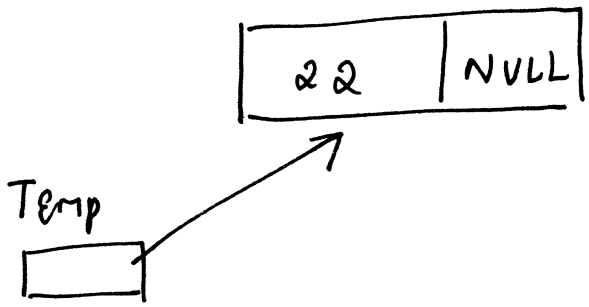
Inserting value into a node

```
NODE  temp;
temp = getnode()
```



temp → info = 22

temp → link = NULL



or

(*temp).info = 22

(*temp).link = NULL.

To delete a node

free(temp), memory of temp gets deallocated.

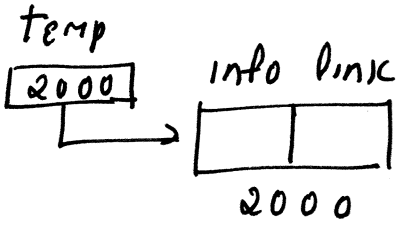
Operations on singly linked list

The basic operations that can be performed on a linked list

- Inserting a node into the list
- Deleting a node from the list
- → searching a element in the list
- → Display the contents.

Insert a node at the front End

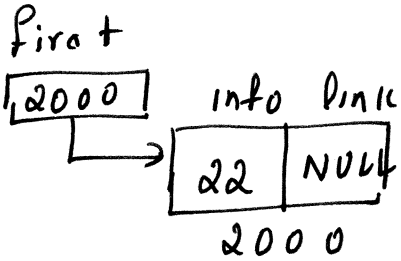
(a) There are no Elements



temp = getnode()

If (first == NULL)

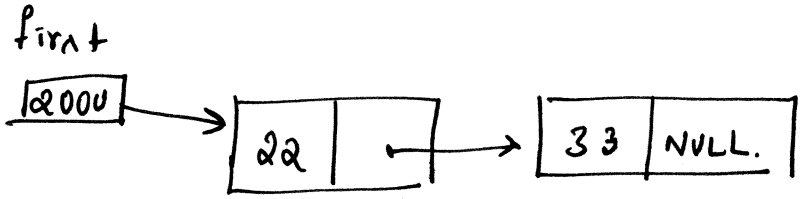
first = temp.



first → info = 22

first → link = address
= NULL

(b) There are nodes / Elements

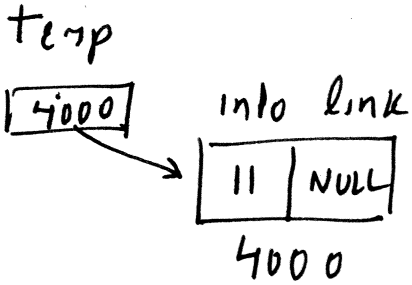


If (first != NULL)

temp = getnode()

temp → info = 11

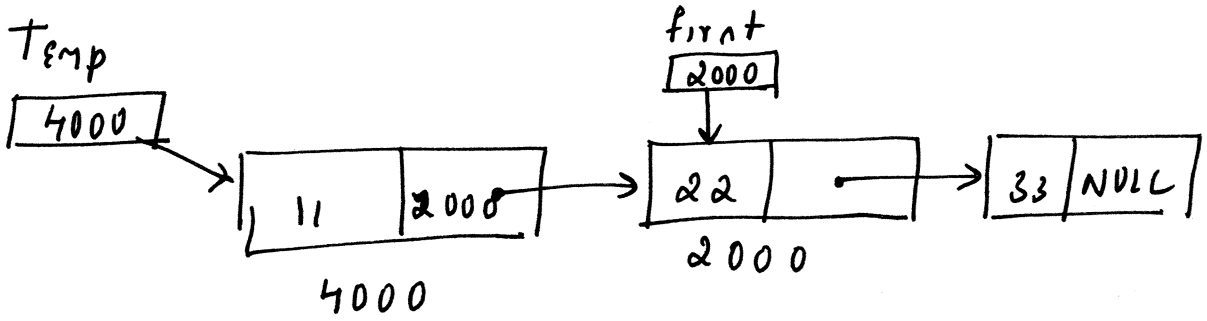
temp → link = NULL



~~NODE~~

~~first~~

temp → link = first



first = temp;

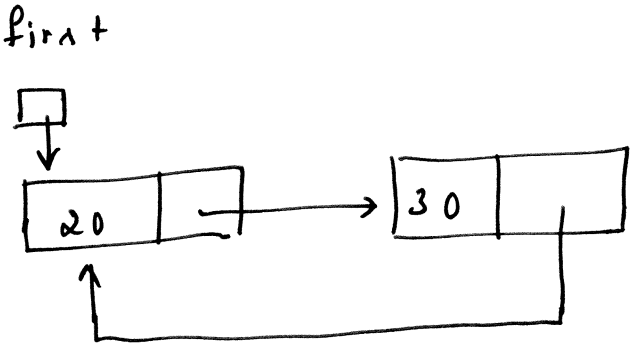
```
NODE    insert_front(NODE first)
{
    NODE temp;
    temp = getnode();
    temp -> link = NULL;
    pf("Enter the value :\n");
    sf("%d", &temp -> info);
    // check empty SLL
    if (first == NULL)
        return temp;
    Else if (first -> link == NULL) // single node
    {                               // or
                                    // multiple node
        temp -> link = first;
        return temp;
    }
}
```

Display the Elements of the SLL

```
void display ( NODE first )
{
    NODE cur ;
    // empty SLL
    if ( first == NULL )
    {
        printf ( " SLL is empty \n") ;
        return ;
    }
    cur = first ;
    while ( cur != NULL)
    {
        printf(" Info value is %d",
                                cur -> info) ;
        cur = cur -> link ;
    }
}
```

Circular list

is a singly linked list where the link field of the last node of the list contains address of the first node.



Advantage

- Every node is accessible from given node by traversing successively using the link field.

Empty list

```
if last == NULL
    // list is empty
```

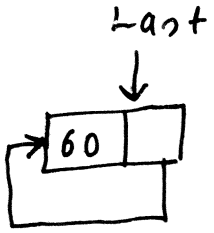
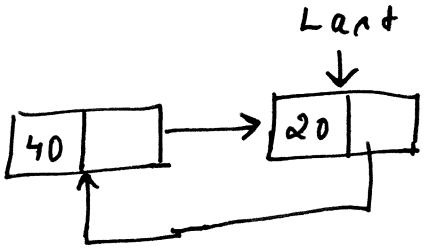


fig (a)

A list with only node can be written as fig (a)

```
If ( last -> link == last )
    // there will be only one in circular list
```

Insert a node to the front end

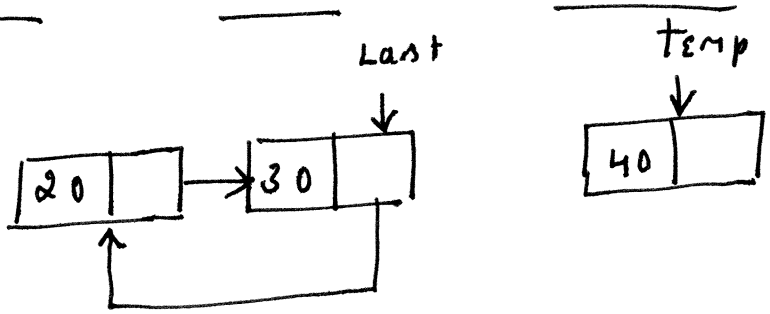



```
temp = getnode()
temp -> info = item;
temp -> link = last -> link ;
last -> link = temp;
```

```
NODE insert_front(int item, NODE last)
{
    NODE temp;
    temp = getnode();
    temp -> info = item;
    if ( last == NULL)
        last = temp;
    Else
```

```
last -> link = temp;

return last

}
```

Insert a node at the rear end




```
if (last != NULL)
        temp -> link = last -> link
```

if last is not NULL, make temp itself the last node.

Make temp as the last node.

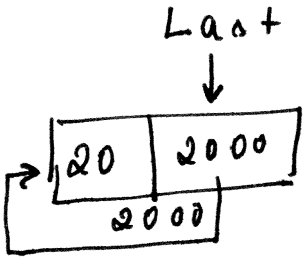
```
last -> link = temp;

return temp.

NODE insert_rear ( int item, NODE last)
{
        NODE temp;
```

```
temp = getnode();
temp -> info = item;
if (last == NULL)
        last = temp;
else
        temp -> link = last -> link;
last -> link = temp;
return temp;
```

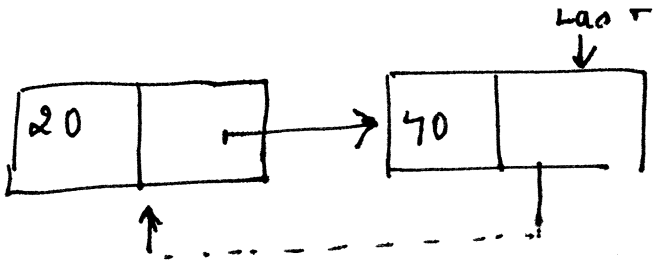
}.

Delete a node from front end

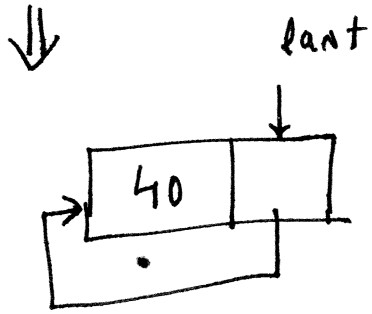
Empty



Only one node.



After deletion




```
first = last -> link;

last -> link = first -> link;

NODE    delete_front( NODE last)

{
    NODE first;
    if (last == NULL)
    {
        pf("List is empty");
        return NULL;
    }
    // only one node
    if ( last -> link == last)
    { pf("Item deleted %d", last->info);
      free(last);
      return NULL; }
```

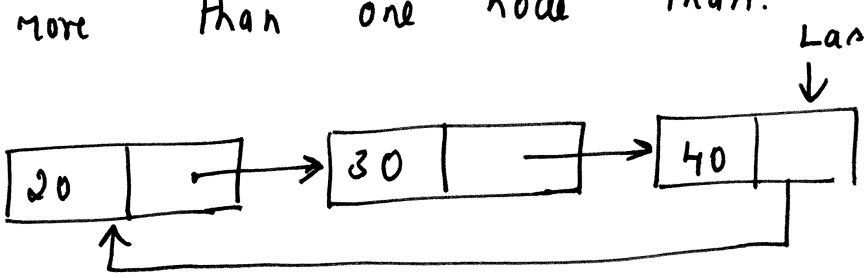
Display circular singly linked list

```
void display( NODE last)
{
    NODE temp;
    if ( last == NULL)
    {
        pf( "List is empty \n");
        return;
    }
    pf(" content of the list\n");
    temp = last -> link;
    while ( temp != last)
    {
        pf("%d", temp -> info);
        temp = temp -> link;
    }
    pf(" %d ", temp -> info);
}
```

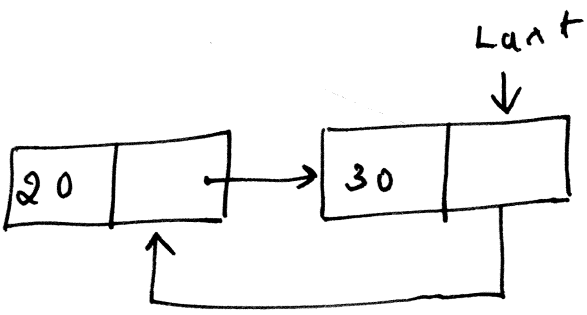
```
first = last -> link;
last -> link = first -> link;
printf("Item deleted is %d", first -> info);
free(first);
return last;
}
```

Delete from rear

First two cases are same., if there are more than one node than.



After deletion



To find the address of the last but one node.

```
while (prev -> link != last)
    prev = prev -> link;
```

```
        prev → link = last → link :

        return  prev.

NODE    delete_rear( NODE  last )
{
        NODE  prev ;
        if ( last == NULL)
        {
            pf (" List is  Empty ") :
            return  NULL :
        }
        if  (last → link == last )
        {
            pf (" ITEM deleted is %d", last→info ) :
            free ( last) :
            return  NULL :
        }
        prev = last → link
        while  ( prev → link != last )
                prev =  prev → link ;
        prev →  link = last → link :
        pf (" Item  deleted  is %d \n", last → info) :
        free ( last)
        return  prev :   }.
```

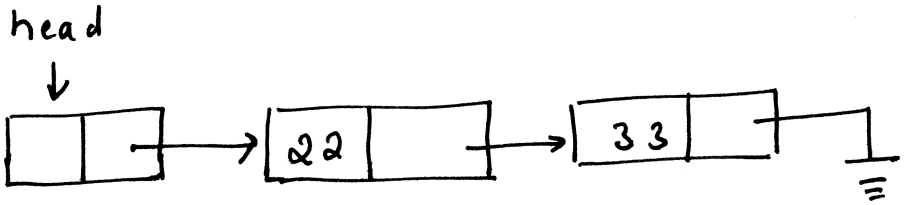
Header linked list

A header linked list is a linked list which always contain a special node called the header node, at the begining of the list The following are two kinds of widely uned header list.

There are two types of Header linked list

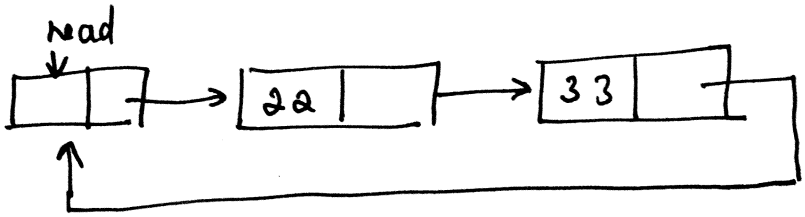
(i) grounded header linked list

The last node of the linked list will have NULL value.

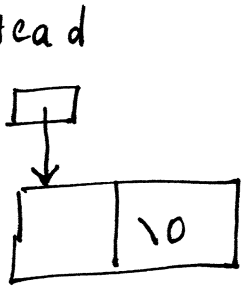


(ii) Circular header linked list

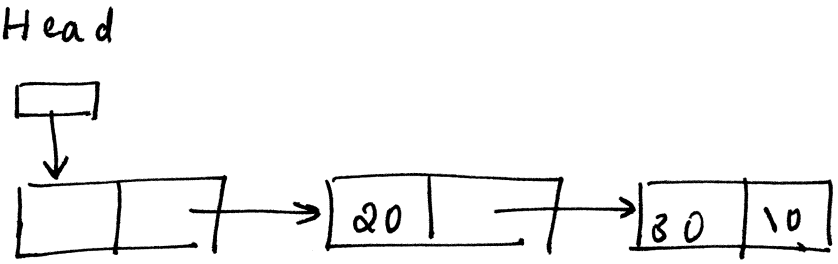
Last node of the linked list will point back to header node



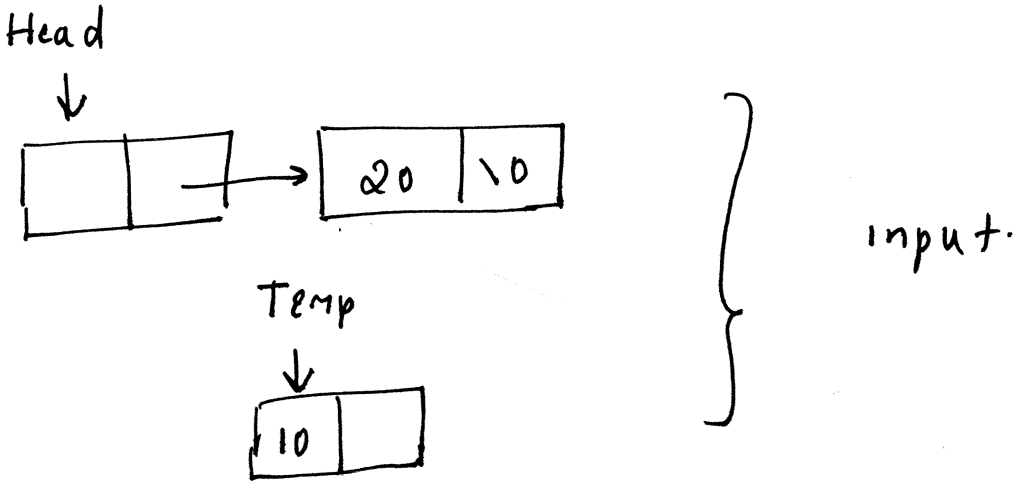
Empty header node.



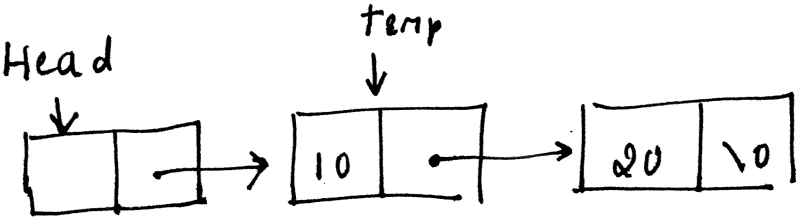
Header node with items/nodes.



Insert a node at the front End



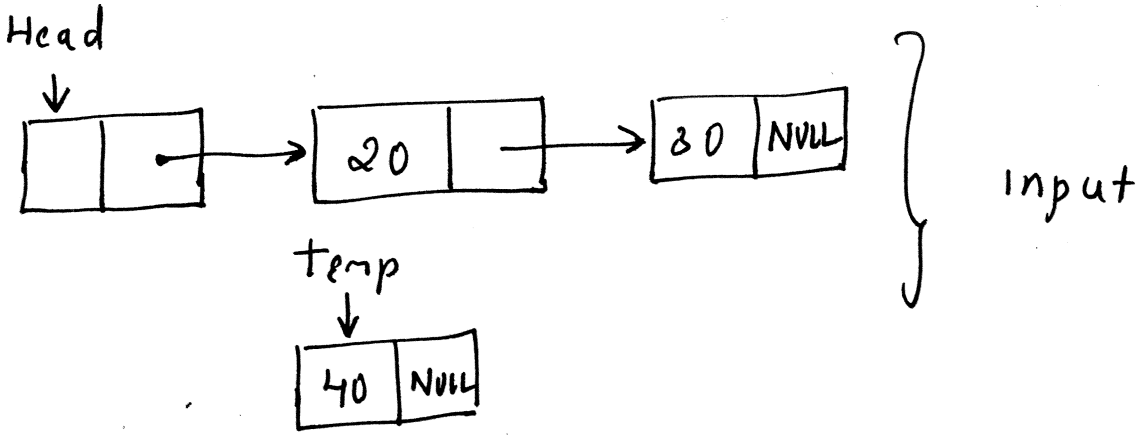
O/P

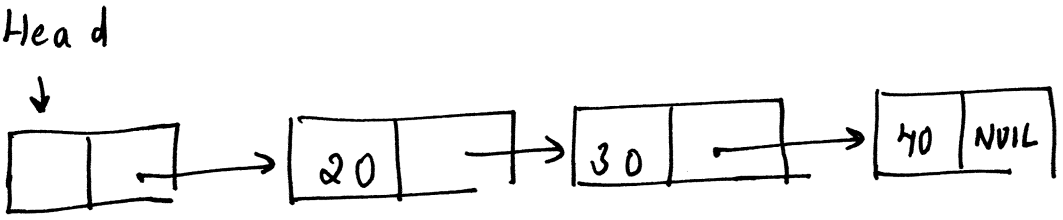


```
first = head -> link;

temp -> link = first

head -> link = first;
```

```
NODE    insert_front ( int item, NODE head)
{
    NODE  temp;
    temp = get node ();
    temp -> info = item;
    first = head -> link
    temp -> link = first;
    head -> link = first; return head;
}
```

Insert node at the rear End





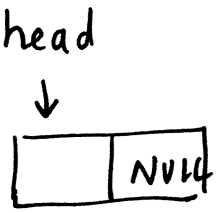
To obtain addrem of the last node.

```
cur = head -> link;
while (cur -> link != NULL)
    cur = cur -> link;

cur -> link = temp.
```

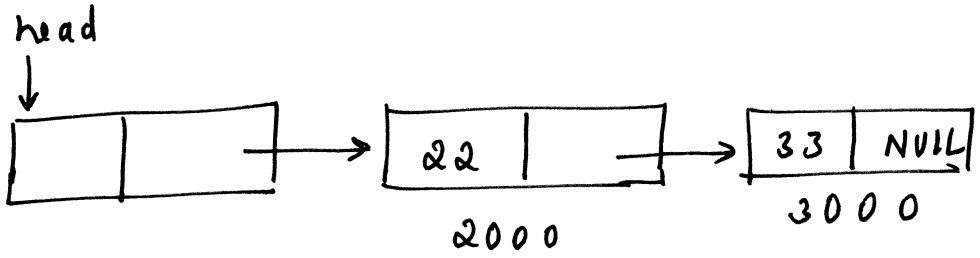
```
NODE    insert_rear( int item, NODE head)
{
    NODE temp, cur;
    temp = getnode();
    temp -> info = item;
    temp -> link = NULL;
    cur = head -> link;
    while( cur -> link != NULL)
        cur = cur -> link
    cur -> link = temp;
    return head;
}
```

Delete a node from the front end

List is empty if head → link = NULL



Multiple nodes

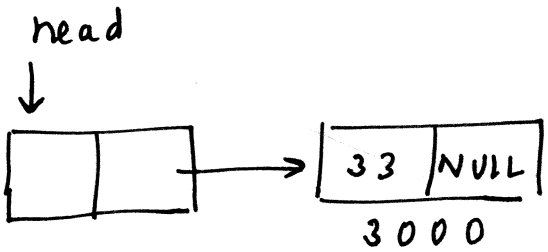


first = head → link // 2000

next = first → link // 3000

free (first)

head → link = first

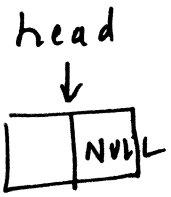



```
void delete_front(NODE head)
{
    NODE first, next;
    if (head -> link == NULL)
        printf("Header linked list is empty");
```

```
else
{
    first = head -> link;
    next = first -> link;
    pf("Deleted elem = %d", first -> info);
    free (first)

    head -> link = next

}
}.
```

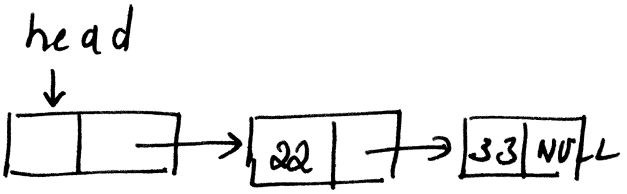
Delete rear

Empty list




```
if (head -> link == NULL)
```

Multiple Node.




```
prev = head
cur = head -> link.
while (cur -> link != NULL)
{
    prev = cur
    cur = cur -> link
}
```

```
void delete_rear( NODE head)
{
    NODE prev, cur;
```

```
if (head → link == NULL)
        pf (" Header linked list is empty\n");
else
 {
        prev = ~~NULL~~ head;
        cur = head → link;
        while ( cur → link != NULL)
         {
                prev = cur;
                cur = cur → link;
         }
        pf (" Deleted Elem = %d ", cur → info);
        free ( cur);
        prev → link = NULL
 }
}
```

Display

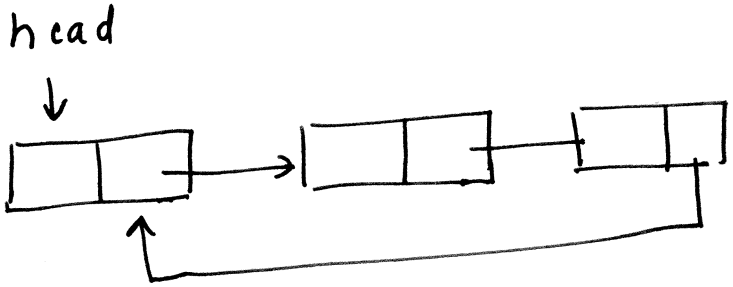
its similar to single linked list.

The difference is the first node info must not be displayed.

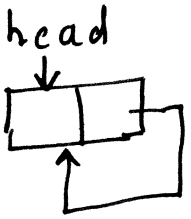
```
void display (NODE head)
{
    NODE cur;
    cur = head -> link
    if (cur == NULL)
        pf ("Empty header linked list");
    else
    {
        while (cur != NULL)
        {
            pf ("%d", cur -> info);
            cur = cur -> link;
        }
    }
}
```

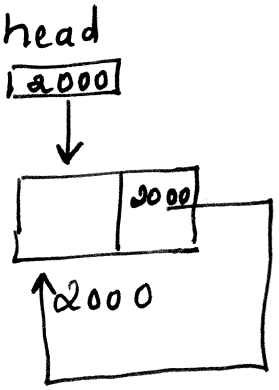
Circular list - Header

is a singly linked list where the link field of the last node contains the address of the header

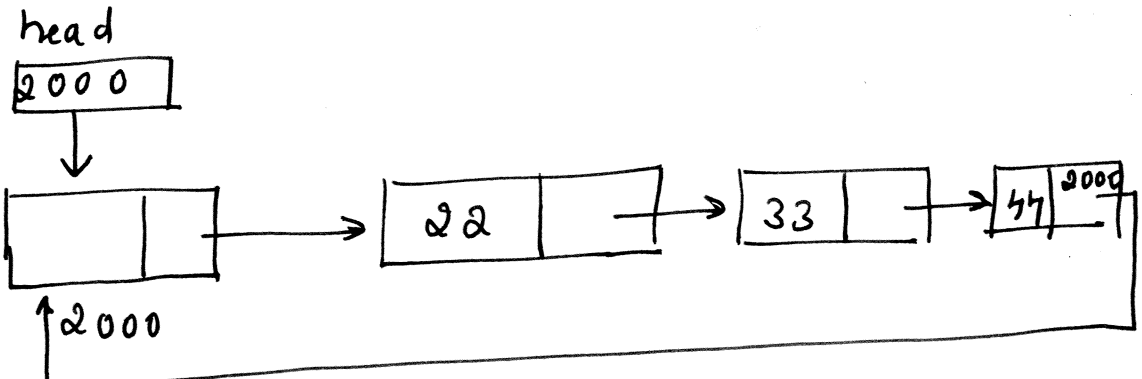


(i) Empty circular list





(ii) Non Empty Circular list



*** Initialization : head = getnode()

head → link = head

Insert node at the front end

Create a node

```
temp = getnode()
temp → info = item
```

Obtain the address of the first node.

```
first = head → link;
```

Make new node as the first node

```
head → link = temp
temp → link = first
```

```
void
~~NODE~~ insert_front(int ITEM, NODE head)
{
    NODE temp;
    temp = getnode();
    temp → info = ITEM
    first = head → link;
    head → link = temp;
    temp → link = first;
}
```

Insert node at the rear

```
Void    insert_rear (int ITEM, NODE head)
{
        NODE    temp, cur;
        temp = getnode();
        temp → info = ITEM
        cur = head → link;
        while ( cur → link != head)
                cur = cur → link;
        cur → link = temp;
        temp → link = head;
}
```

Delete a node from rear

List is empty

```
if (head → link == head)
        pf("List is empty\n")
```

```
Void    delete_rear( NODE  head)
{
        NODE    prev, cur ;
        if (head -> link == head)
        {     pf ("Empty list \n") ;
              return  head;
        }
        cur = head -> link ;
        prev = head
        while ( cur -> link != head)
        {
              prev = cur ;
              cur = cur -> link ;
        }
        prev -> link = head ;
        Pf (" ITEm deleted %d\n", cur -> info) ;
        free ( cur) ;
}
```

Delete node front

```
NODE    delete_front( NODE head)
{
        NODE    first, next;
        If (head -> link ==head)
        {       pf ("Empty list\n");
                return head;
        }
        first = head -> link;
        next = first -> link;
        head -> link = next;
        Pf (" Deleted Elem is %-d", first -> info);
        free (first);
        return head;
}
```

Display

```
        void    display ( NODE head)
        {
                NODE    cur;
```

```
if (head -> link == head)
{
        pf ("Empty list\n");
        return;
}
cur = head -> link;
while (cur != head)
{
        pf ("%d", cur -> info);
        cur = cur -> link;
}
}.
```

Module 4

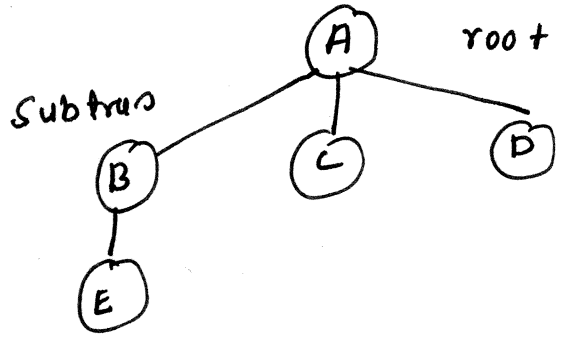
3sem CS
DS

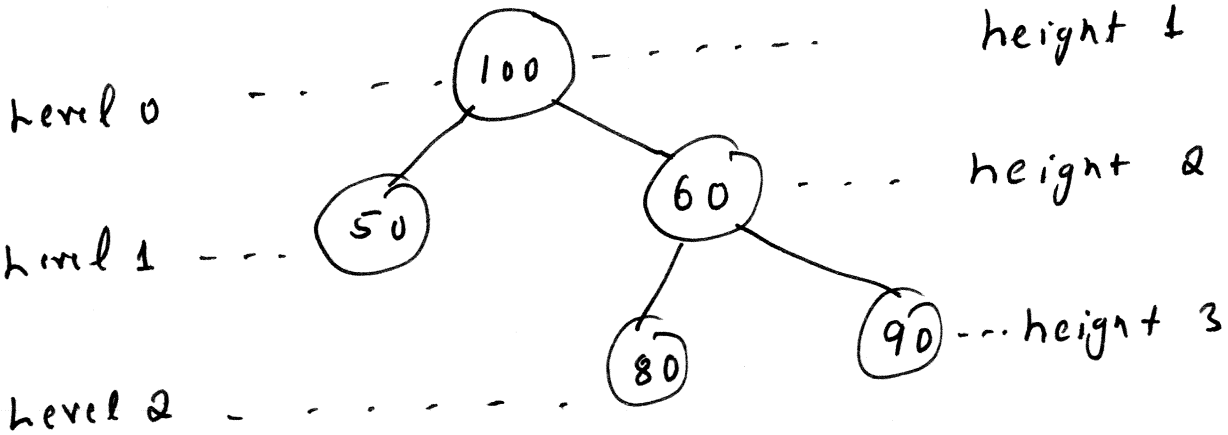
25

Trees

Tree is a set of links of one or more nodes that shows parent-child relation

Tree has
→ special node called the root node.
→ Remaining nodes are partitioned into disjoint subsets $T_1, T_2, \ldots, T_n$ $n \geq 0$





Root node: First node written at the top is root node.

Child Node obtained from parent node is called child Node.

Sibling Two or more nodes having the same parent.

Ancestor: Nodes obtained in the path from the specified node x while moving upwards towards the root.

Descendants. Nodes in the path below the parent.

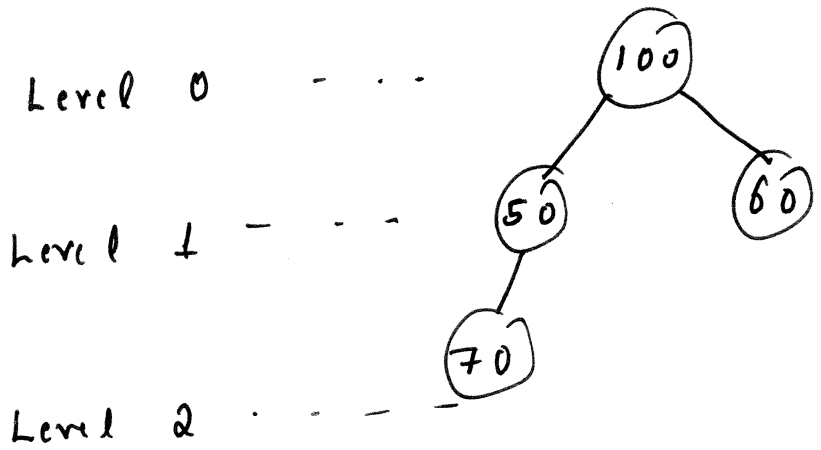
Left descendants Nodes that lie towards Left subtree

Right descendant Nodes that lie towards right subtree.

Degree. no. of subtrees of a node is called its degree.

Leaf: A node in a tree that has a degree Zero.

** Level the distance of a node from the root is called level of the node.



Height (Depth) is defined as the maximum level of any leaf in the tree.

Different ways of representing a tree

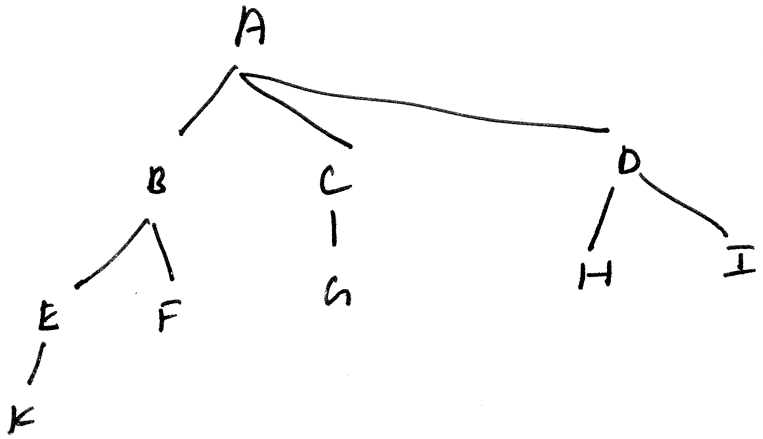
i) list representation

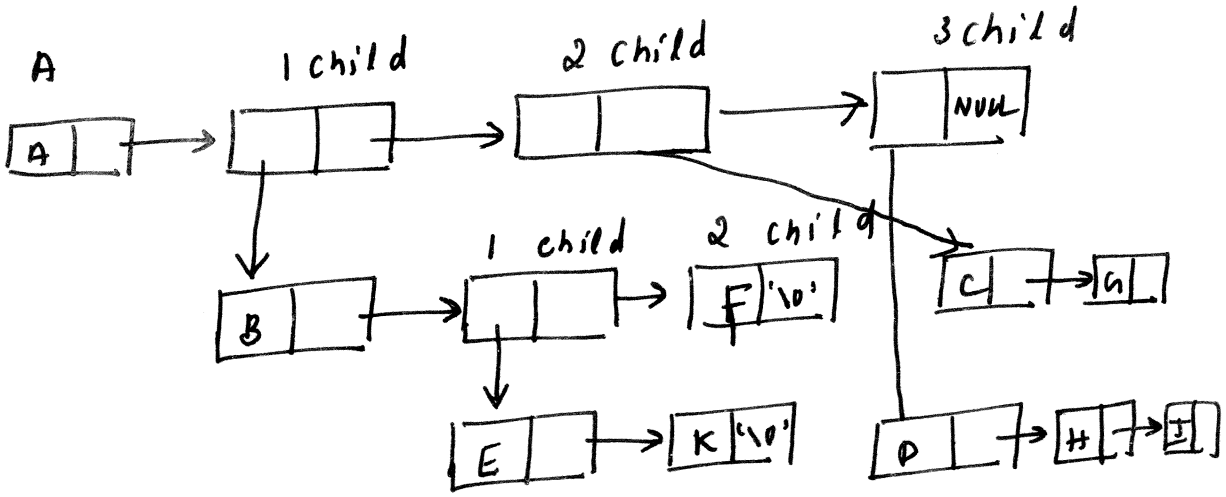
ii) Left child - right sibling representatn

iii) Binary tree representatn

List representation

- Root node comes first, it is immediately followed by a list of subtrees of that node.
→ It is recursively repeated for each subtree.





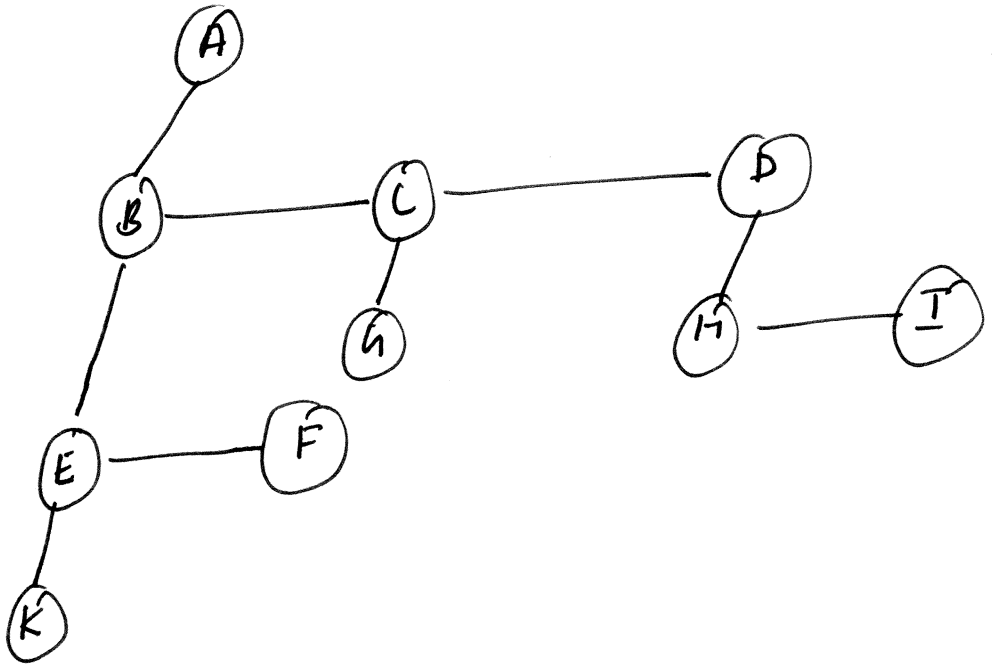
Left child Right sibling representation

A left child - right sibling representation of a given tree can be obtained as

Shown below

Left pointer of a node in the tree will be the left child in the representation

Remaining children of a node in the tree are inserted horizontally to the left child.



Binary Tree (Degree two)

A binary tree representation is called Left child - right child tree representation or degree two representation.

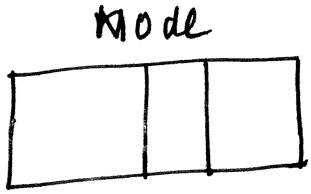
Obtain left child right sibling representation

Rotate the horizontal lines clockwise by 45 degrees.

****

Binary tree is a tree which has finite set of nodes that is either empty or consists of a root & two subtrees called left subtree & right subtree.

Data structure for binary tree



'node' in a tree has three fields

llink : Address of the left subtree

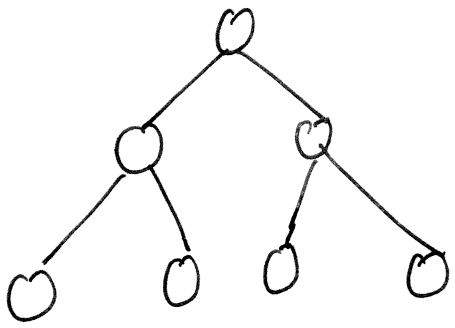
rlink : —"— right —"—

info : stores the actual data

Properties of binary tree

Lemma : a) Max no of nodes on level i of a binary tree $= 2^i$ for $i \geq 0$

b) The max no of nodes in a binary tree of depth $k = 2^k - 1$



Level

$0 = 2^0$

$1 = 2^1$

$2 = 2^2$

Types of binary trees

(a) Strict binary tree (full binary tree)

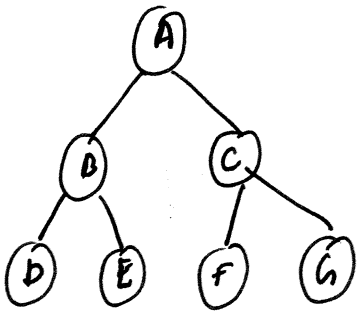
(b) skewed tree

(c) complete binary tree

Strict binary tree

A binary tree having 2^i nodes in any given level i is called strict binary tree.

Ex



Level 0 = $2^0 = 1$

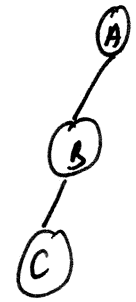
Level 1 = $2^1 = 2$

Level 2 = $2^2 = 4$

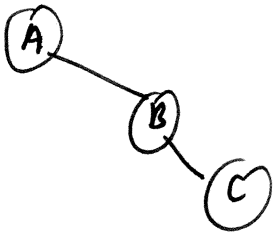
Skewed tree

is a tree consists of only left subtree or only right subtree.

A tree with only left subtree is called left skewed binary tree



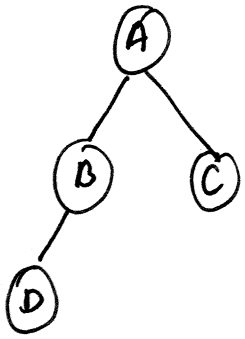
Left

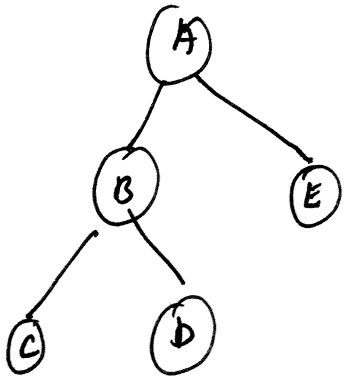


Right

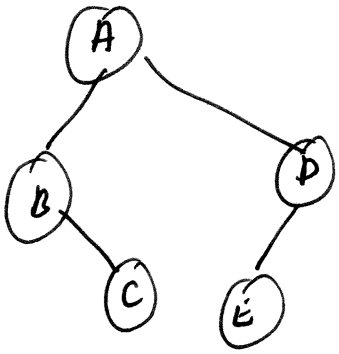
Complete binary tree

is a binary tree in which every level except possibly the last level is completely filled.





Trees below are not complete binary tree since the last level are not filled from left to right.



Binary tree traversal

is a method of visiting each node of a Tree exactly once in a systematic order

Different types of traversal

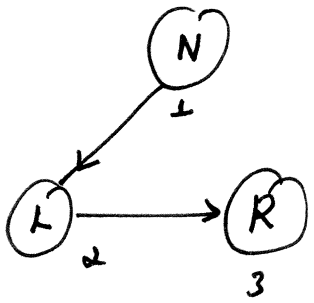
(i) preorder
(ii) postorder
(iii) inorder

Preorder

process the root node

traverse the left

traverse the right




```
void preorder ( Node root)
{
    if (root == NULL) return
    pf ("%d", root → info) ;
    preorder (root → llink) ;
    preorder ( root → rlink) ;
}
```

post order

Traverse left node

——→ right node

process the root

```
void postorder( Node root)
{
    if( root == NULL) return;
    postorder ( root -> llink);
    postorder( root -> rlink);
    printf(" %d", root -> info);
}
```

Inorder traversal

Traverse the left subtree in inorder
process the root node
Traverse the right subtree in inorder

```
void ~~preorder~~ inorder ( NODE root)
{
    if (root == NULL) return;
    inorder (root -> llink);
    pf(" %d", root -> info);
    inorder( root -> rlink);
}.
```

Binary Search Tree (BST)

It a binary tree in which for each node say x in the tree, elements in the left sub-tree are less than info(x) and elements in the right subtree are greater than info(x)

Insert Elements in BST

```
NODE insert ( int item, NODE root)
{
    NODE temp, cur, prev;
    temp = get node();
    Temp → info = item;
    Temp → llink = NULL;
    Temp → rlink = NULL;
    if (root == NULL) return temp;
    prev = NULL;
    cur = root;
```

```
while (cur != NULL)
{
        prev = cur
        if (item < cur -> info)
                cur = cur -> llink;
        else
                cur = cur -> rlink;
}
if (item < prev -> info)
        prev -> llink = temp;
else
        prev -> rlink = temp;
return root;
```

3.

Searching

```
NODE search (int item, NODE root)
{
        NODE cur
        if (root == NULL) return NULL;
        cur = root;
```

```
while (cur != NULL)
{
        if (item == cur->info) return cur;

        if( item < cur->info)
                cur = cur->llink;
        else
                cur = cur->rlink;
}
return NULL;
}.
```

Iterative Traversal of Binary search Tree

iterative preorder traversal

```
void preorder (NODE root)
{
    NODE cur, s[20];
    int top = -1;
    if (root == NULL)
    {
        pf ("Tree is empty");
        return;
    }
    cur = root;
    for (;;)
    {
    while (cur != NULL)
    {
        printf ("%d", cur -> info);
        s[++top] = cur;
        cur = cur -> llink;
    }
    if (top != -1)
    {
        cur = s[top--];
        cur = cur -> rlink;
    }
    else
        return
    }
}
```

Iterative inorder traversal

```
void inorder (NODE root)
{
    NODE cur, s[20];
    int top = -1;
    if (root == NULL)
    {
        pf ("Tree Empty");
        return;
    }
    cur = root;
    for (;;)
    {
        while (cur != NULL)
        {
            s[++top] = cur;
            cur = cur -> llink;
        }
        if (top != -1)
        {
            cur = s[top--];
            pf ("%d", cur -> info);
            cur = cur -> rlink;
        }
        else return;
    }
}
```

```
for (;;)
{
        while (cur != NULL)
        {
            top++;
            s[top].address = cur;
            s[top].flag = 1;
            cur = cur->llink;
        }
        while (s[top].flag < 0)
        {
            cur = s[top].address;
            top--;
            pf("%d", cur->info);
            if (top == -1) return;
        }
        cur = s[top].address;
        cur = cur->rlink;
        s[top].flag = -1;
    }
}.
```

Iterative postorder traversal

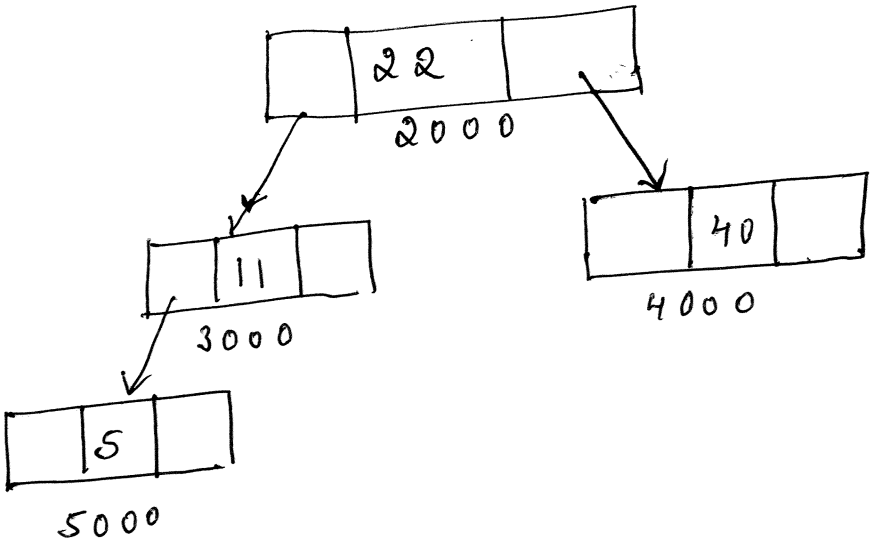
In postorder traversal, traverse the left subtree in postorder, then traverse the right subtree in postorder and finally visit the node.

stack once during traversing the left-subtree and another while traversing towards right.

To distinguish that traversing the right subtree is over, we set flag to -1.

flag = -ve right subtree is traversed and one can visit the node.

```
void postorder(NODE root)
{
    struct stack
    {
        NODE address;
        int flag;
    };
    NODE cur;
    struct stack s[20];
    int top=-1;
    if (root == NULL)
    {
        pf("Tree is empty\n");
        return;
    }
    cur = root;
```

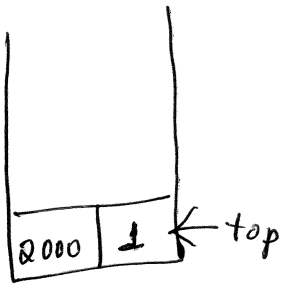


I iterat^{n}

a) cur = 2000

s[top].addr = cur

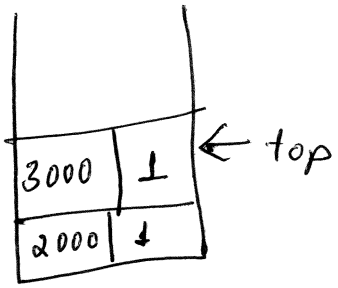
.flag = 1



cur = 3000

s[top].addr = cur

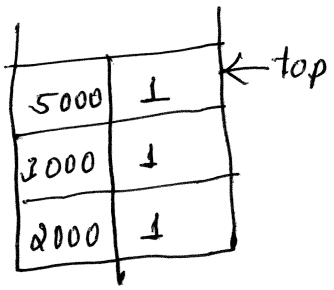
.flag = 1



cur = 5000

s[top].addr = cur

.flag = 1

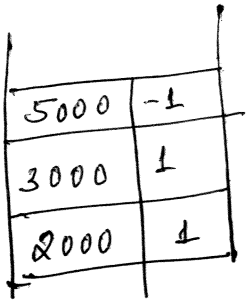


(b) while(1 < 0)

(c) cur = 5000

cur = NULL

flag = -1



II Itrat^n

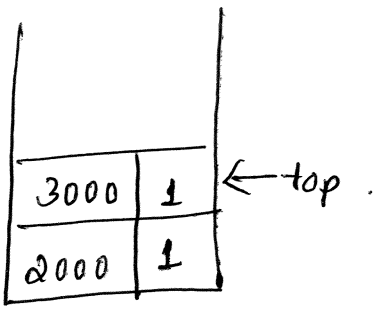
(a) cur = NULL

(b) while (s[top].flag < 0).

cur = 5000

top--

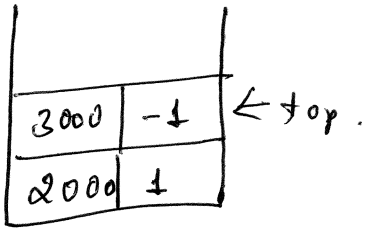
① - - - - - - pf ("5").



(c) cur = 3000

cur = NULL

flag = -1.



III iteration

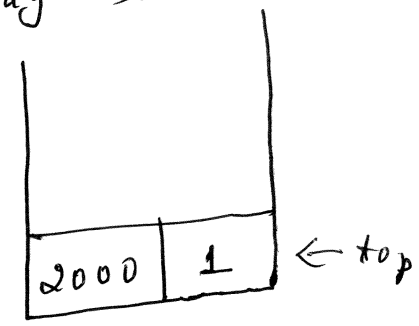
(a) cur = NULL

(b) while (s[top].flag < 0)

cur = 3000

top--

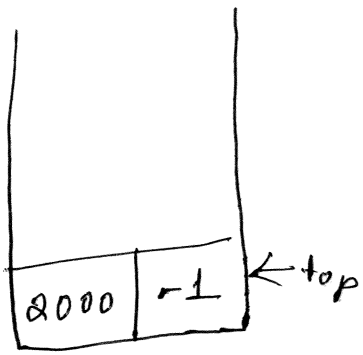
② - - - - - pf ("11");



(c) cur = 2000

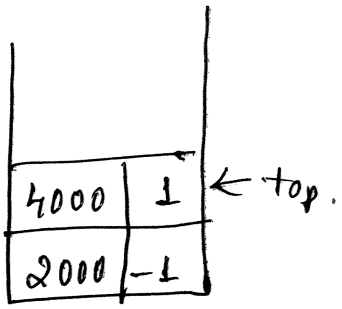
cur = 4000 (right link)

flag = -1.



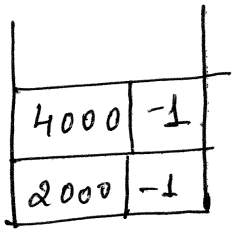
IV Iteration

(a) cur = 4000
top++
s[top].addr = cur
s . flag = 1



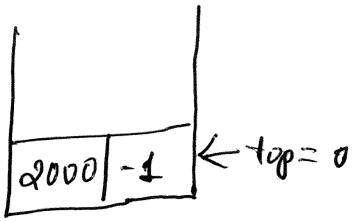
(b) while (s[top].flag < 0)

(c) cur = 4000
cur = NULL
s[top].flag = -1



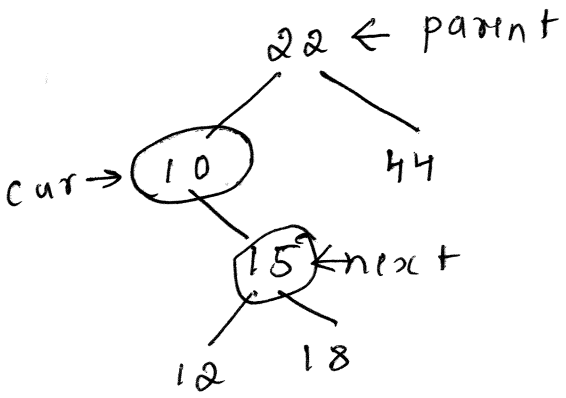
V (a) cur = NULL

(b) while (s[top].flag < 0)
cur = s[top].addr
pf("40");
top--
cur = s[top].addr
pf("20")
top == -1 return



Delete a Node from BST

Case 1: Deleted node has either left tree or right tree.



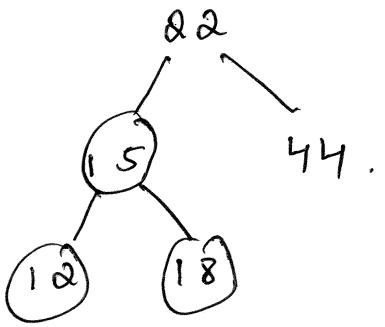
cur - node to be deleted

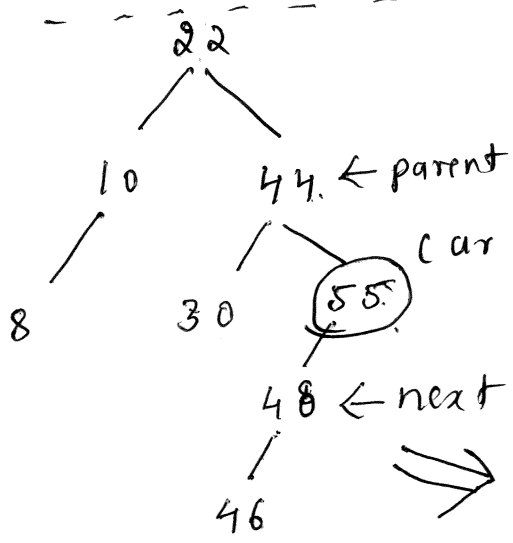
cur has only right node.

If (cur → rlink == NULL)

~~next → llink = cur~~

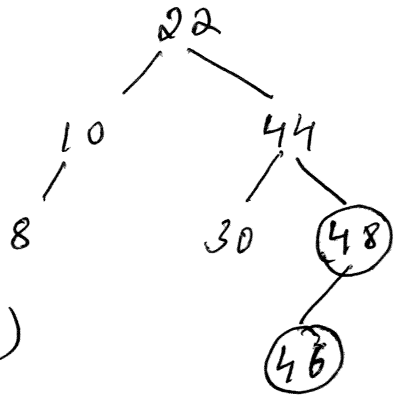
parent → llink = next





cur - node to be deleted

cur has only left node



If (cur → rlink == NULL)

~~next → llink~~

parent → rlink = next

Case2 Non Empty left subtree and non-empty right subtree

Find inorder successor of node then copy the contrat of inode successor

case:1

if cur → llink == NULL

(i) Find the inorder successor.

```
suc = cur → rlink;
while( suc → llink != NULL)
{
    suc = suc → llink;
}
```

(ii) Attach left subtree of the node to the

suc → llink = cur → link;

```
NODE    delet_item(int item, NODE root)
{
        NODE    cur, parent, suc, q;
        if (root == NULL)
        {
                pf(" Tree is empty")
                return root
        }
        parent = NULL;
        cur = root;
        while ( cur != NULL)
        {
                if ( item == cur→info) break;
                parent = cur;
                if (item < cur → info)
                        cur = cur → llink
                else
                        cur = cur → rlink
        }
        if (cur == NULL)
        {
                pf(" item not found")
                return root.
        }
```

```
/*   item   not   found */

if  (cur → llink == NULL)
      &&   cur → rlink == NULL

{
      pf ("leaf node");

      if ( parent → rlink != CUR )

            parent → rlink = NULL;

      else
            parent → llink = NULL;   free(cur);

      return;

}

// Either   left   or   right   tree is empty

if  (cur → llink == NULL)

      q = cur → rlink;

else if (cur → rlink == NULL)

      q = cur → llink;

else
{     // obtain   inorder succ

      suc =  cur → rlink;

      while  (suc → llink != NULL)

            suc = suc → llink;
```

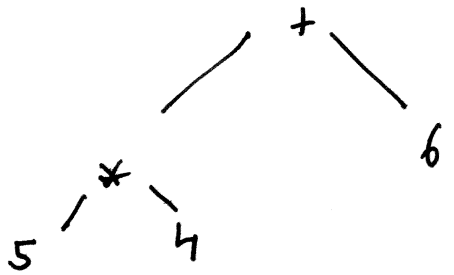
```
        suc -> llink = cur -> llink ;
        q = cur -> rlink ;
    }
    if (parent == NULL) return q ;
// connecting parint node to deleted node q
    if (cur == parent -> llink)
            parent -> llink = q ;
    else
            parent -> rlink = q ;
    free (cur) ;
    return root ;
}
```

Expression tree

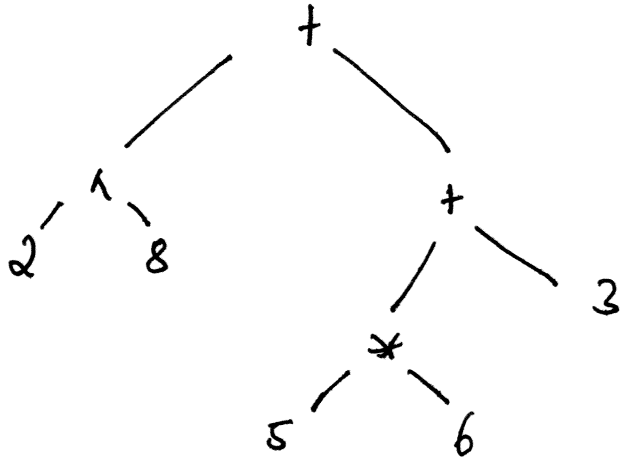
is a BST, that exhibit the following properties

- Any leaf is an operand.
- root and internal nodes are operators.
- subtree represent sub expression with root of the subtree as an operator.

5 * 4 + 6



2 ^ 8 + (5 * 6 + 3)



Given infix expression is converted to postfix form and then a binary expression tree is created

Algorithm:

1. Scan the symbol from left to right
2. Create a node for the scanned symbol
3. If the symbol is an operand push the corresponding node to stack.
4. If the symbol is an operator, pop topmost node from stack and attach it to the right of operator node.
   pop next node and attach it to left
   push the node back onto stack.
5. Repeat until all symbols are used in postfix expression
   Finally topmost node in stack, will be the root node.

Threaded Binary Tree

Disadvantage of binary tree

- There will be more links with NULL values
- Only downward movement is possible

These disadvantage can be overcome by replacing the NULL with address so as to facilitate upward movement.

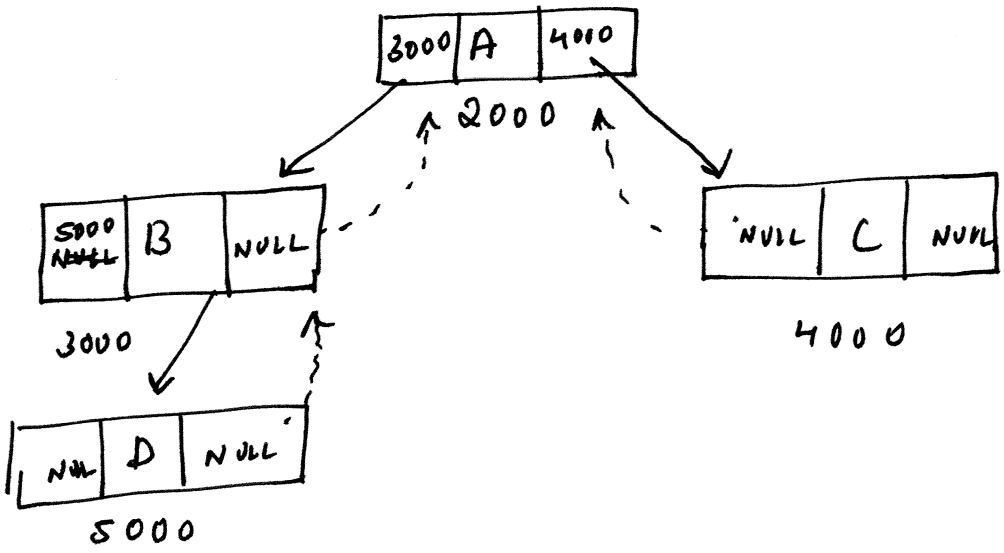
These extra links which contain address of some nodes are called threads. and binary tree with such pointers are called threaded trees.

Full threaded binary tree

lbit (Node which has NULL) is replaced by address of inorder predecessor

rbit (Node which has right link NULL) is replaced by address of inorder successor

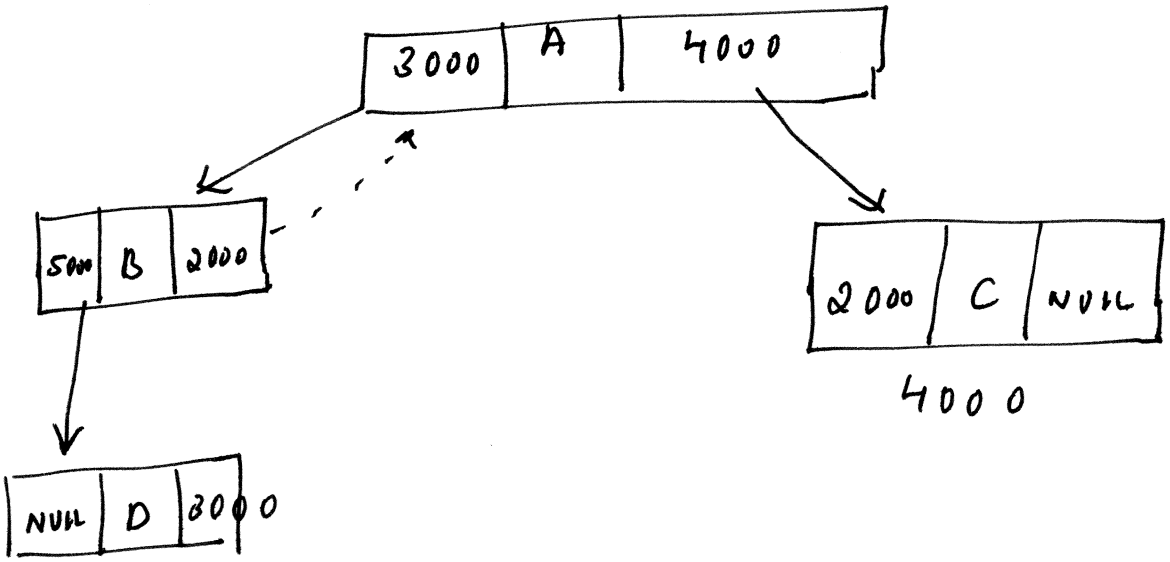
Example



Inorder

D B A C

After threading



llink in D &
rlink in C are referred as dangling pointer.

To differentiate between actual llink/rlink and a thread flag values are used.

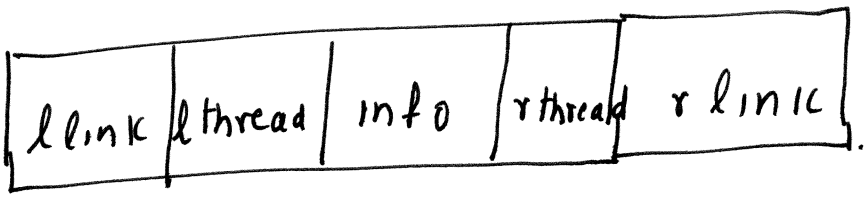
Flags

lthread

rthread

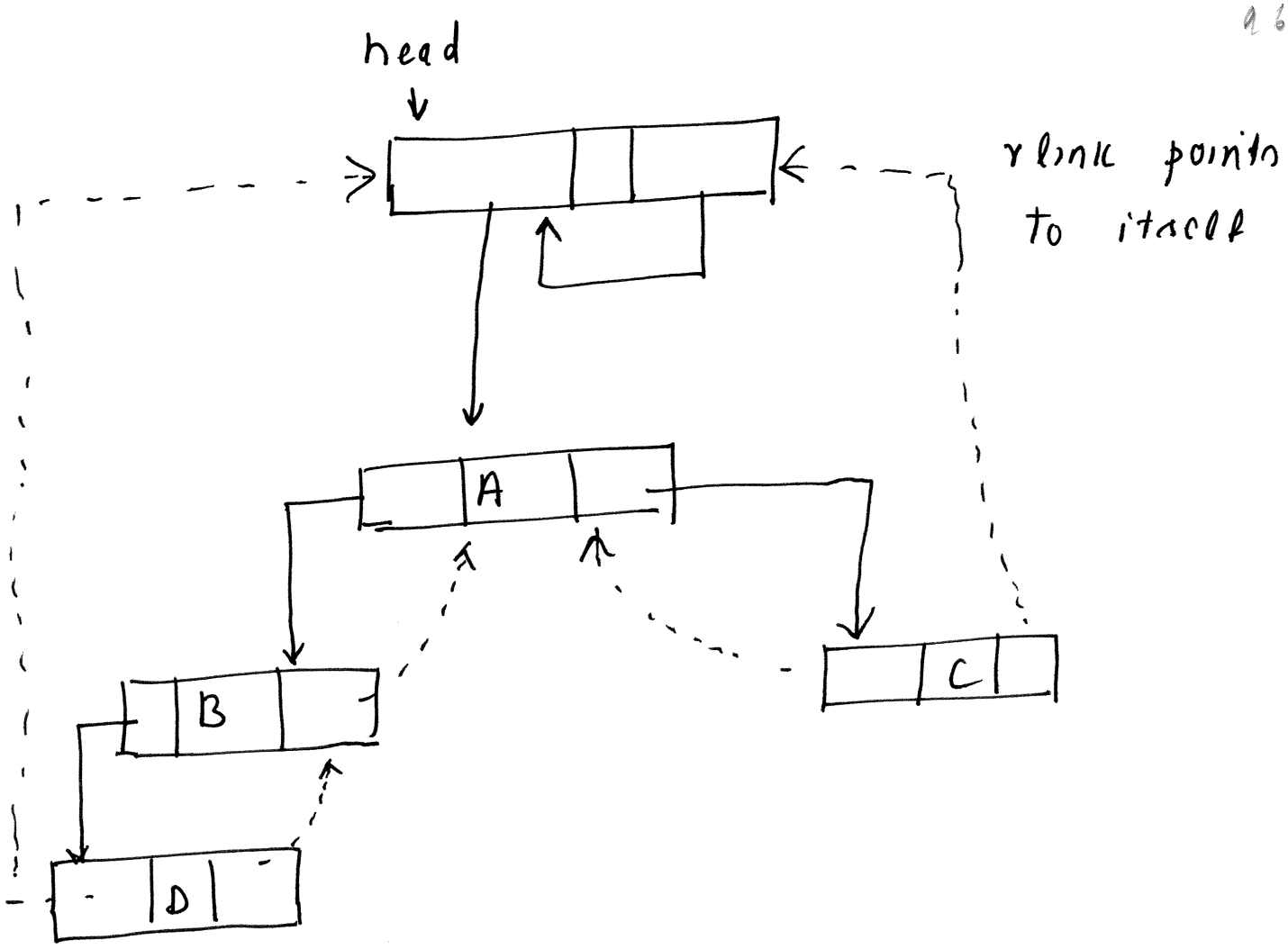
0 – not used as thread

1 – used as thread.




```
struct node
{
    struct node * llink;
    int lthread;
    int info;
    int rthread;
    struct node * rlink;
}
```

To overcome the problem of dangling pointer we use a header.


head
rlink points to itself
A
B
C
D

Module 5

Graph Theory

Vertex: It is symbolized by a node, diagramatically indicated by a circle.

Edge: line joining between two vertices u & v

There is no direction we call it as undirected edge.

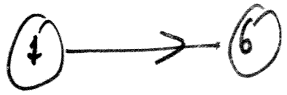
Graph — A graph consists of two sets V & E

V - is a finite, non empty set of vertices

E - is a set of pairs of vertices these pairs are called edges.

$e = (u, v)$ — u & v are called end points

(1, 6) or (6, 1)

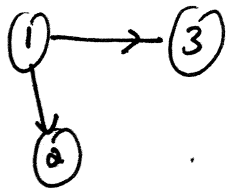


(1, 6) 1 - tail
6 - head.

The directed edge has direction

$\langle u, v \rangle$ u - tail
v - head

Ex



graph $G = (V, E)$

where

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

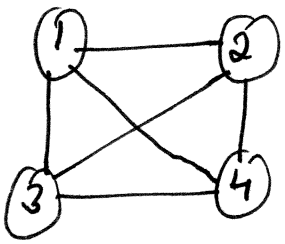
$E = \{ \langle 1,3 \rangle, \langle 1,2 \rangle \}$.

$|V|$ represent the total no of vertices $= 3$

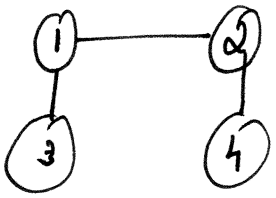
$|E|$ —— —— of Edges $= 2$.

Complete graph

A graph $G = (V, E)$ is said to be a complete graph, if there exists an edge between every point of vertices



Complete graph



Not a complete graph

A subgraph of G is a graph G' such that

$V(G') \subseteq V(G)$ & $E(G') \subseteq E(G)$

A path from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \ldots i_k, v$ such that $(u, i_1), (i_1, i_2), \ldots (i_k, v)$ are edges in $E(G)$

Simple path

A path that does not have repeated vertices is called a simple path.

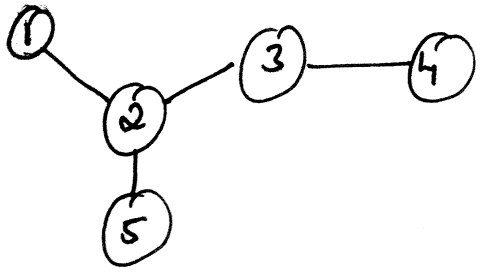
Length of a path is the number of edges on it.

path from ① → ② can be written as

$(1, 3), (3, 4), (4, 2)$ length is 3

Cycle is a simple path in which the first & last vertices are the same.

A tree is an undirected graph with no cycles and a vertex chosen to be the root of the tree.



Graph Representation

(i) Adjacency Matrix

(ii) —"— list

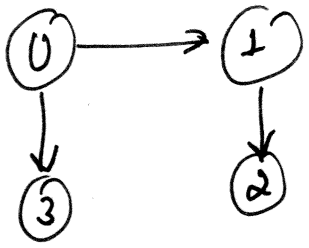
Adjacency Matrix Let $G = (V, E)$ be a graph with n vertices $n \geq 1$.

The adjacency matrix of G is a two dimentional $n \times n$ array say a, with the property that

$a[i][j] = 1$ iff the edge (i, j) is in $E(G)$

$a[i][j] = 0$ if there is no such edge in G.

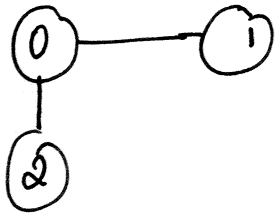
Ex



Directed Graph

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

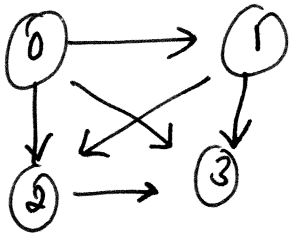
undirected graph

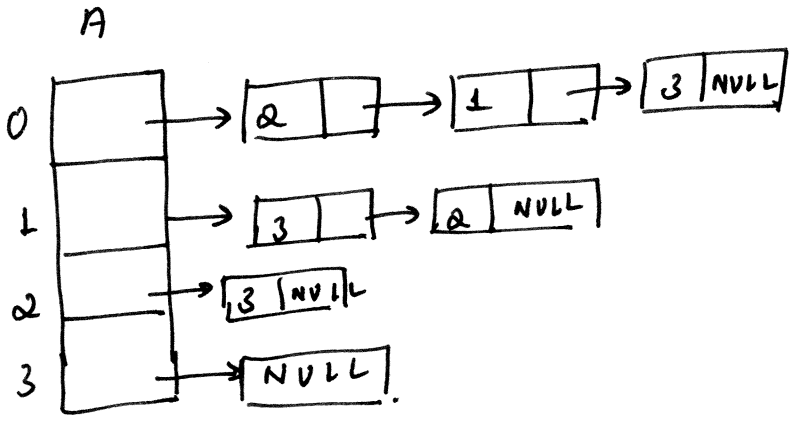


	0	1	2
0	0	1	1
1	1	0	0
2	1	0	0

Adjacency list in this representation of graph the n rows of the adjacency matrix are represented as 'n' chains there is one chain for each vertex G.

The nodes in chain i represent the vertices that are adjacent from vertex i





Graph traversals

process of visiting each node of a graph systematically in some order is called graph traversal.

Two important graph traversal techniques

Breadth First search (BFS)

Depth First search (DFS)

DFS: Algorithm is to traverse the graph in such a way that it tries to go far from root. stack is used for implementation of DFS

Algorithm:

Step 1: push the root node in the stack

Step 2: Loop until stack is empty

Step 3: Read the node from the stack

Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed & push it on stack.

Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

```
void dfs(int u)
{
    int v;
    s[u] = 1;
    pf("%d", u);
```

```
    for ( v=0; v <n; v++)
      {
          if ( a[u][v]==1  && s[v]==0)
             dfs(v);
      }
}
void    read_adj_matrix ( int  a[10][10], int n)
 {
        int  i,j;
        for( i=0; i<n; i++)
         {     for( j = 0 ; j<n ; j++)
                  sf("%d", &a[i][j])
          }
}
int     a[10][10], s[10],n;
void    main()
  {
       int i, soure;
       pf(" Entr  the  no of nodes in the
                                  graph \n");
       sf("%d", &n);
       pf(" Ender  adj matrix \n")
       read_adj_matrix(a, n);
       for ( soure = 0; soure <n; soure++)
        {
```

```
for (i=0; i<n ; i++)   s[i]=0 ;

pf( " Nodes  reachable from %d : ", source) :

dfs (source) :

}
```

3.

Using stack

```
int  n, visited [10] :
void     dfs ( int   soure )

{
        int s[10] , top = -1, ele, i :

        visited [soure] = 1 :

        pf ( " Nodes reachable from source %d",
                                    source) :

        push ( s, soure , &top) :

        while ( top != -1 )

        {
                ele = pop( s, &top) :

                for ( v =0 ; v<n ; v++)

                {
                        if ( a[ elem] [v] == 1 &&
                            visited [v] == 0)

                        { pf ("%d", v) :

                          visited[v] = 1 :

                          push (s, v, &top) :

                }       }

}
```

Inorder traversal of a threaded binary tree

It can be implemented using stack or Queue

First step is to find inorder successor of a node

```
NODE insucc( NODE tree)
{
    NODE temp;
    temp = tree->rlink;
    if ( ! temp->rthread)   // tree->rthread flag 0
        while ( ! temp->lthread) // temp->lthread flag
            temp = temp->llink;
    return temp;
}

void tinorder (NODE tree)
{
    NODE temp=tree;
    for (;;)
    {
        temp = insucc(temp);
        if (temp == tree) break;
        print("%c", temp->data);
    }
}
```