Data Structures and Applications

Code - 15CS33

DATA STRUCTURES AND APPLICATIONS

[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2015 -2016)
SEMESTER - III

Subject Code	15CS33	IA Marks	20
Number of Lecture Hours/Week	04	Exam Marks	80
Total Number of Lecture Hours	50	Exam Hours	03

CREDITS - 04

Course objectives: This course will enable students to

- Understand, Practice and Assimilate fundamentals of data structures and their applications essential for programming/problem solving
- Describe, Analyze, Design and Evaluate the Linear Data Structures: Stack, Queues, Lists
- Describe, Analyze, Design and Evaluate the Non-Linear Data Structures: Trees, Graphs
- Describe, Analyze, Design and Evaluate the sorting & searching algorithms
- Assess appropriate data structure during program development/Problem Solving

Module -1	Teaching Hours	RBT Levels
Introduction: Data Structures, Data structure Operations, Review of Arrays, Structures, Self-Referential Structures,and Unions. Pointers and Dynamic Memory Allocation Functions. Representation of Linear Arrays in Memory, Dynamically allocated arrays, **Array Operations**: Traversing, inserting, deleting, searching, and sorting. Multidimensional Arrays, Polynomials and Sparse Matrices. **Strings:**Basic Terminology,Storing, Operations and Pattern Matching algorithms. Programming Examples. **Text 1: Ch 1: 1.2, Ch2: 2.2 -2.7** **Text 2: Ch 1: 1.1 -1.4, Ch 3: 3.1-3.3,3.5,3.7, Ch 4: 4.1-4.9,4.14**	**10Hours**	**L1, L2**
Module -2		
Stacks and Queues ***Stacks:***Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to postfix conversion, evaluation of postfix expression, **Recursion** - Factorial, GCD, Fibonacci Sequence, Tower of Hanoi, Ackerman's function. ***Queues:*** Definition, Array Representation, Queue Operations, Circular Queues, Circular queues using Dynamic arrays, Deques, Priority Queues, A Mazing Problem. Multiple Stacks and Queues. Programming Examples. **Text 1: Ch3: 3.1 -3.7** **Text 2: Ch6: 6.1 -6.3, 6.5, 6.7-6.10, 6.12, 6.13**	**10 Hours**	**L1, L2, L3, L4, L6**

Module - 3		
Linked Lists: Definition, Representation of linked lists in Memory, Memory allocation; Garbage Collection. Linked list operations: Traversing, Searching, Insertion, and Deletion. Doubly Linked lists, Circular linked lists, and header linked lists. Linked Stacks and Queues. Applications of Linked lists – Polynomials, Sparse matrix representation. Programming Examples **Text 1: Ch4: 4.1 -4.8 except 4.6** **Text 2: Ch5: 5.1 – 5.10**	**10 Hours**	**L2, L3, L4, L6**
Module-4		
Trees: Terminology, Binary Trees, Properties of Binary trees, Array and linked Representation of Binary Trees, Binary Tree Traversals - Inorder, postorder, preorder; Additional Binary tree operations. Binary Search Trees – Definition, Insertion, Deletion, Traversal, Searching; Expression tree, Threaded binary trees, and Forests. Programming Examples **Text 1: Ch5: 5.1 –5.5, 5.6,5.9** **Text 2: Ch7: 7.1 – 7.9**	**10 Hours**	**L2, L3, L4, L6**
Module-5		
Graphs: Definitions, Terminologies, Matrix and Adjacency List Representation Of Graphs, Elementary Graph operations, Traversal methods: Breadth First Search and Depth First Search. ***Sorting and Searching***: Insertion Sort, Radix sort, Address Calculation Sort. ***Hashing***: Hash Table organizations, Hashing Functions, Static and Dynamic Hashing.***Filesand Their Organization:***Data Hierarchy, File Attributes, Text Files and Binary Files, Basic File Operations, File Organizations and Indexing **Text 1: Ch6: 6.1 –6.2, Ch 7:7.2, Ch 8:8.1-8.3** **Text 2: Ch8: 8.1 – 8.7, Ch 9:9.1-9.3,9.7,9.9** **Reference 2: Ch 16: 16.1 - 16.7**	**10 Hours**	**L2, L3, L4, L6**

Course outcomes:

After studying this course, students will be able to:

- Acquire knowledge of
  - Various types of data structures, operations and algorithms
  - Sorting and searching operations
  - File structures
- Analyse the performance of
  - Stack, Queue, Lists, Trees, Graphs, Searching and Sorting techniques
- Implement all the applications of Data structures in a high-level language
- Design and apply appropriate data structures for solving computing problems.

Graduate Attributes (as per NBA)

1. Engineering Knowledge
2. Design/Development of Solutions
3. Conduct Investigations of Complex Problems
4. Problem Analysis

Question paper pattern:

The question paper will have ten questions.
There will be 2 questions from each module.
Each question will have questions covering all the topics under a module.
The students will have to answer 5 full questions, selecting one full question from each module.

Text Books:

1. Fundamentals of Data Structures in C - Ellis Horowitz and SartajSahni, 2nd edition, Universities Press,2014
2. Data Structures - Seymour Lipschutz, Schaum's Outlines, Revised 1st edition, McGraw Hill, 2014

Reference Books:

1. Data Structures: A Pseudo-code approach with C –Gilberg&Forouzan, 2nd edition, Cengage Learning,2014
2. Data Structures using C, , ReemaThareja, 3rd edition Oxford press, 2012
3. An Introduction to Data Structures with Applications- Jean-Paul Tremblay & Paul G. Sorenson, 2nd Edition, McGraw Hill, 2013
4. Data Structures using C - A M Tenenbaum, PHI, 1989
5. Data Structures and Program Design in C - Robert Kruse, 2nd edition, PHI, 1996

Data structures & Applications

Module – 1 (Introduction)

→ Data structure, Operations.
→ Review of Arrays, structures, self-Referential structures & Unions.
→ Pointers & Dynamic Memory allocation func.
→ Representation of Linear Arrays in Memory, Dynamically allocated arrays.

(Array Operations)

→ Traversing, inserting, deleting, Searching sorting.
→ Multidimensional Arrays, Polynomials & Sparse Matrices.

(Strings)

→ Basic Terminology, storing, Operations & Pattern Matching algo, Programming Example

Text 1: 1.2, 2.2 – 2.7
Text 2: 1, 3.1 – 3.3, 3.5, 3.7, 4.1 – 4.9, 4.14

Applications of Data structures (DS)

1. Operating System.
   - (i) process scheduling (priority queues)
   - (ii) File structures. (Trees)
   - (iii) Paging
   - (iv) spooling Jobs / printer.
2. Editors
   - (i) storage of words (linked list)
   - (ii) word search / prediction. (iii) undo or (stack)
3. Compilers (Trees)
   - (i) symbol table / hash table
   - (ii) parsers (Trees)
   - (iii) Memory allocation in LISP (Heap)
4. Networking
   - (i) Ip routing Table (Radix Tree)
   - (ii) Connections / Relations in social n/w sites (Graphs.)
   - (iii) N/ws of Communication (graphs)

Data Structures

Def: The logical or mathematical model of a particular organization of data is called a data structure.

Arrays in C

→ An array is a collection of data storage locations each having the same data type & the same name.

→ Each storage location in an array is called an array element.

→ Array declaration

```
int marks[30];
```

> An array is a collection of similar Elements

→ Array is also known as subscripted variable.

→ However big an array its elements are always stored in contiguous memory locations.

→ Array Initialisation

```
int num[6] = {2, 4, 12, 5, 56, 45}
int n[] = {2, 4, 12, 5, 56, 45}
float press[] = {12.3, 34.2, -23.4, -11.3};

int std[4][2] = {
                  {12, 56},
                  {21, 33},
                  {14, 80},
                  {13, 78}
                };
int std[4][2] = {12, 56, 21, 33, 14, 80, 13, 78};
int std[][3] = {12, 56, 21, 33, 14, 80, 13, 78};
```

→ Data are simply values or sets of values.
→ A Data item refers to a single unit of values.
→ Data items that are divided into subitems are called group items, those that are not are called Elementary Items.
→ Collections of data are frequently organized into a hierarchy of fields, records & files.
→ Data are organized into complex types of structures. The study of such data structures includes the following three steps:
① Logical / Mathematical description of the structure.
② Implementation of the structure on a Computer.
③ Quantitative analysis of the structures, which includes determining the amount of memory needed to store the structure & the time required to process the structure.
→ The logical or mathematical model of a particular organization of data is called a Data structure.
→ Data structure is categorized into two:
① Linear Data structure.
② Non-Linear Data structure.

→ Linear DS: A linear data structure traverses the data elements sequentially, in which only one data element can directly be reached
Ex: Arrays, Linked lists, stacks & Queues.

→ Non-Linear DS: Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in sequential structure.
Ex: Trees, Graphs.

→ Arrays: Array is a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually
1, 2, 3, ... , n
if we choose the name A for the array, then the elements of A are denoted by subscript notation

$$a_1, a_2, a_3, \ldots a_n$$

or by the parenthesis notation

$$A(1), A(2), A(3), \ldots A(N)$$

or by the bracket notation

$$A[1], A[2], A[3], \ldots A[N]$$

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

Example1: A linear array STUDENT consisting of the names of six students

Example2: A chain of 28 stores, each store having 4 departments, listing its weekly sales.

Linked lists: It is an ordered set of data elements each containing a link to its successor (& sometimes its predecessor).

Ex: Suppose a brokerage firm maintains a file where each record contains a customer's name & his/her salesperson, & suppose the file contains the data as appearing below, clearly the file could be stored in computer by such a table. ie by two columns of nine names.

	Customer	Salesperson
1	Adams	Smith
2	Brown	Ray
3	Clark	Jones
4	Drew	Ray
5	Evans	Smith
6	Farmer	Jones
7	Geller	Ray
8	Hill	Smith
9	Infield	Ray

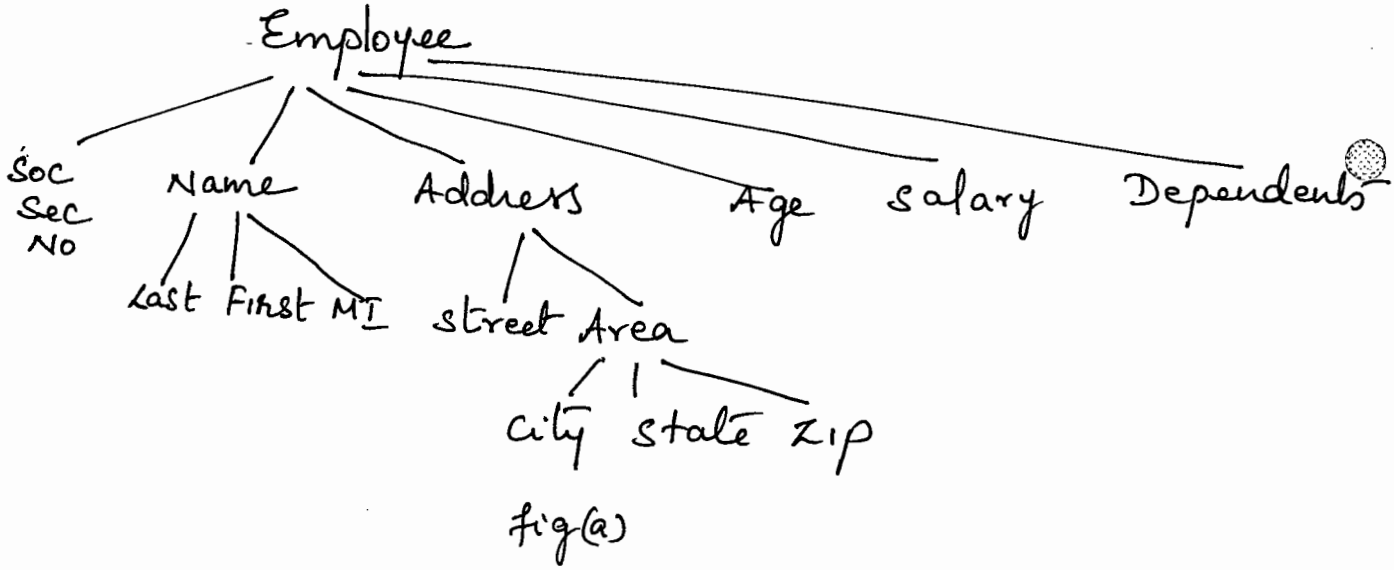
	Customer	Pointer
1	Adams	3
2	Brown	2
3	Clark	1
4	Drew	2
5	Evans	3
6	Farmers	1
7	Geller	2
8	Hill	3
9	Infield	2

Salesperson	
Jones	1
Ray	2
Smith	3

Another way of storing the data is shown above

Trees: It is a abstract data type (ADT) that simulates a hierarchical relationship between various elements. It contains a root value & subtrees of children with a parent node, represented as a set of linked nodes.

Ex: Employee Record.



fig(a)

```
01 Employee
    02 Social Security Number
    02 Name
        03 Last
        03 First
        03 Middle Initial
    02 Address
        03 street
        03 Area
            04 city
            04 state
            04 zip
    02 Age
    02 Salary
    02 Dependents
```

fig(b)

Stack: A stack, also called a last-in first-out (LIFO) system, is a linear list in which insertions & deletions can take place only at one end, called the top. This structure is similar in its operation to a stack of dishes on a spring system.

Queue: A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the front of the list, & insertions can take place only at the other end of the list, the rear of the list.

Ex: i) people waiting at a bus stop.
ii) Job assigned to the printer.

Graph: Graph DS consists of a finite set of vertices or nodes, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph.

The relationship between pairs of elements represented by vertices/nodes are not necessarily hierarchical in nature.

Data Structure Operation.

There are several operation associated with DS, four major once are listed below

i) Traversing: Accessing each record exactly once so that certain items in the record may be processed.

2) Searching : Finding the location of the record with a given key value, or finding the locations 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 are used in special situations.

1) Sorting : Arranging the records in some logical order.

2) Merging: Combining the records in two different sorted files into a single sorted file.

Algorithm : LinearSearch.

LINEAR (DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements & ITEM is a given item of information. 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.

```
// Program to implement linear search
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int array[100], n, key, i;
    clrscr();
    Printf("Enter number of elements");
    scanf("%d",n);
    Printf("\nEnter the elements in the array");
    for(i=0; i<n; i++)
        scanf("%d", &array[i];
    Printf("\n Enter the element to be
            searched");
    scanf("%d", &key);
    a[n]=key;
    for(i=0; i<=n; i++)
        if(a[i]==key)
            break;
    if(i==n)
        printf("\nElement is not found
            in the array");
    Else
        printf("\n Element found at
            location %d", i);
    getch();
}
```

Algorithm Binary Search.

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

Here DATA is a sorted array with lower Bound LB & upper Bound UB & ITEM is a given item of information. The variables BEG, END & MID denote respectively, the beginning, end & middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables]
   Set BEG := LB, END := UB & MID := INT((BEG + END)/2).
2. Repeat steps 3 & 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3. If ITEM < DATA[MID], then
   set END := MID − 1
   Else
   set BEG := MID + 1
   [End of If structure.]
4. Set MID := INT((BEG + END)/2).
   [End of Step2 loop.]
5. If DATA[MID] = ITEM, then:
   Set LOC := MID.
   Else:
   Set LOC := NULL.
   [End of If Structure.]
6. Exit.

---

(1) (11), 22, 30, 33, 40, 44, [55], 60, 66, 77, 80, 88, (89)
(2) 11, 22, 30, 33, 40, 44, 55, (60), 66, [77], 80, 88, (89)
(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80), [88], (89)
(4) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80), 88, 89 unsuccessful.

Binary Search for ITEM = 85

Structure & Unions.

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

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

members of Structures — 10 + 4 + 8 = 22 (size)

→ The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members.

```
Ex  strcpy(person.name, "james");
    person.age = 10;
    person.salary = 35000;
```

→ The structure data type is created by using typedef. statement.

```
typedef struct
{
    char name[10];
    int age;
    float salary;
} humanBeing;
humanBeing person1, person2.
```

→ humanBeing is the struct data type.

def: A structure is defined as a collection of data of same/different data types.

→ Embedding a structure with in a structure is allowed in C

Ex
```
typedef struct
{
    int month;
    int day;
    int year;
} date;
typedef struct
{
    char name[10];
    int age;
    float salary;
    date dob;
} humanBeing;
```

→ A person born on February 11, 1944 would have the values for the date struct set as

```
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

Self-Referential structures

→ A self-referential structure is one in which one or more of its components is a pointer to itself.

→ Self-referential structures usually require dynamic storage management routines (malloc & free) to explicitly obtain & release memory.

Example:

```
typedef struct
{
    char data;
    struct list *link;
} list;
```

→ data is a single char, while link is a pointer to a list structure.

→ The below C-statement creates three structures and assign values to their respective fields:

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```

→ structures item1, item2 & item3 each contain the data item a, b & c respectively & the null pointer.

Unions

→ A union declaration is similar to a structure, but the fields of a union must share their memory space.

→ Only one field of the union is active at any given time.

Example:
```
union u_tag
{
    int ival;
    float fval;
    char *sval;
} u;
```

→ The variable u will be large enough to hold the largest of the three types, the specific size is implementation-dependent.

→ Syntactically, members of a union are accessed as

union-name.member

or

union-pointer→member

Example:
```
struct
{
    char *name;
    int flags;
    int utype;
    union
    {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

→ The member ival is referred to as

symtab[i].u.ival

→ The first character of the string sval by either of

*symtab[i].u.sval

symtab[i].u.sval[0].

Internal Implementation of Structures.

→ Compilers do padding to structures to permit two consecutive components to be properly aligned within memory.

→ The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including padding that is required.

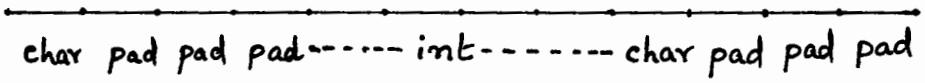
→ Structures must begin and end on the same type of memory boundary, for example, an even byte boundary or an address that is a multiple of 4, 8 or 16.

Example:

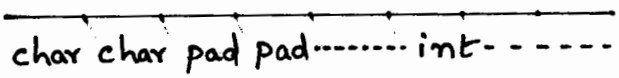
```
struct                          struct
{                               {
    char a;                         char a;
    int b;          VS              char b;
    char c;                         int c;
} x;                            } y;
```

A common way that the structs will be constructed in 32 bits system. where int is 32 bits.

struct x



struct y



Sparse Matrices

→ Matrices with a relatively high proportion of zero entries are called Sparse Matrices.

→ Two types of n-square sparse matrices are

1) Triangular Matrix
2) Tridiagonal Matrix.

Triangular Matrix

$$\begin{matrix} 4 & & \\ 6 & 7 & \\ 1 & -2 & 3 \end{matrix}$$

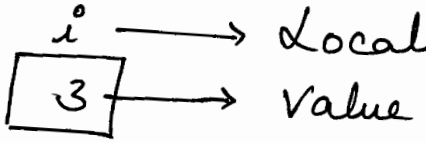
Tridiagonal Matrix.

Pointers.

def: A pointer is a variable whose value is the address of another variable. ie direct address of the memory location.

& → the address operator
* → the dereferencing (or indirection) operator.

```
int i=3;
```



Program-1

```
main()
{
    int i=3;
    Printf("\n Address of i = %u", &i);
    Printf("\n Value of i = %u", i);
}
```

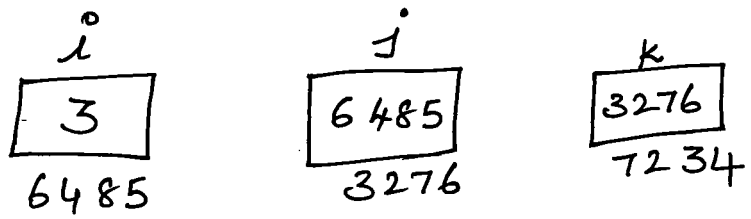
o/p

Address of i = 6485
Value of i = 3

Program-2.

```
main()
{
    int i=3;
    int *j;
    j = &i;
    Printf("\n Address of i = %u", &i);
    Printf("\n Address of i = %u", j);
    Printf("\n Address of j = %u", &j);
```

```
printf("\n value of j = %d", j);
printf("\n value of i = %d", i);
printf("\n value of i = %d", *(&i));
printf("\n value of i = %d", *j);
}
```



Address of i = 6485
Address of i = 6485
Address of j = 3276
Value of j = 6485
Value of i = 3
Value of i = 3
Value of i = 3

Program-3

```
main()
{
    int i=3;
    int *j;
    int **k;

    j=&i;
    k=&j;
    printf("\n Address of i = %u", &i);
    printf("\n Address of i = %u", j);
    printf("\n Address of i = %u", *k);
    printf("\n Address of j = %u", &j);
    printf("\n Address of j = %u", k);
    printf("\n Address of k = %u", &k);

    printf("\n Values of j = %u", j);
    printf("\n Values of k = %u", k);
    printf("\n value of i = %d", i);
    printf("\n value of i = %d", *(&i));
    printf("\n value of i = %d", *j);
    printf("\n value of i = %d", **k);
}
```

O/P

Address of i = 6485
Address of i = 6485
Address of i = 6485
Address of j = 3276
Address of j = 3276
Address of k = 7234

Value of j = 6485
value of k = 3276
Value of i = 3
value of i = 3
Value of i = 3
Value of i = 3

Dynamic memory allocation function in C.

→ The exact size of array is unknown until the compile time.
The size of array declared initially can sometimes be insufficient & sometimes more than required.

→ Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

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

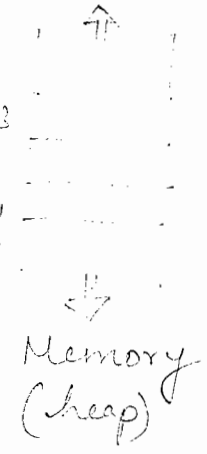
(unsigned int)

malloc – void * malloc (size_t size)
calloc – void * calloc (size_t num, size_t size)
realloc – void * realloc (void * ptr, size_t size)
free – void * free (void * ptr)

Note: Ex1: void *p = malloc (3 * sizeof (int))
(no of elem) (size of one unit)
print p
*p = 2 ✗ can not dereference a void pointer

→ void pointer is used as generic type which is normally typecasted into a pointer of a particular type

Ex2: int *p = (int *) malloc (3 * sizeof (int))
print p // 201
P[0] = 2
P[1] = 4



Ex 3

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Enter size of Array ");
    scanf("%d",&n);
    int *A = (int*) malloc(n*sizeof(int));   // dynamically allocated array
    for(int i=0; i<n; i++)
        A[i] = i+1;
    for(int i=0; i<n; i++)                   <- (b)
        printf("%d ", A[i]);
}
```

(a) →

Note.1 (a) → could be replaced by

```
int *A = (int*) calloc(n, sizeof(int));
```

Ex4 Insert this code at location ← (b) delete the rest of the code.

```
int *B = (int*) realloc(A, 2*n*sizeof(int));
printf("prev block address = %d, new addr = %d\n",A,B);
for(int i=0; i<n; i++)
    printf("%d", B[i]);
```

Note.2 substituting

```
int *B = (int*) realloc(A,0);
```

implies

```
free(A)
```

Note 3: we can also use

```
int *A = (int*) realloc(A,0);
```

Note 4: `int *B = (int*) realloc(NULL, n*sizeof(int));` is Equivalent to malloc of B.

Strings.

Basic terminologies, storing, Operations & Pattern Matching algorithms.

Def: As an ADT, string have the form $S = s_0, \ldots, s_{n-1}$ where s_i are characters taken from the character set of the programming language.

If $n=0$, then S is an empty or null string.

→ In C, strings are character arrays terminated with the null character '\0'

Example

```
# define MAX-SIZE 100
char S[MAX-SIZE] = {"dog"};
char t[MAX-SIZE] = {"house"};
```

or

```
char S[] = {"dog"}
char t[] = {"house"}
```

String Operations

(I) Substring: Access a substring from a given string. format: SUBSTRING(string, initial, length)

string itself

the position of the first character of the substring

The length of the substring

Example @: SUBSTRING('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'
SUBSTRING('THE END', 4, 4) = ' END'

(b) PL/1 : SUBSTR (S, 4, 7)

FORTRAN: S (4:10)
PASCAL : COPY (S, 4, 7)
BASIC : MID$ (S, 4, 7)

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

format : INDEX (text, pattern)

Example @ : Suppose T contains the text
'HIS FATHER IS THE PROFESSOR'
THEN
INDEX (T, 'THE'), INDEX (T, 'THEN') & INDEX(T, ' THE')
have the values 7, 0 & 14 respectively.

(b) : PL/1 : INDEX (text, pattern)
PASCAL: POS (pattern, text)

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

format $S_1 // S_2$

Example @ Suppose S_1 = 'MARK' and S_2 = 'TWAIN'
Then $S_1 // S_2$ = 'MARKTWAIN'
$S_1 // ' ' // S_2$ = 'MARK TWAIN'

(b) Concatenation is denoted as follows

PL/1 : $S_1 || S_2$
FORTRAN: $S_1 // S_2$
BASIC : $S_1 + S_2$
SNOBOL: $S_1\ S_2$

(IV): Length: is to find number of characters in a string.

format: LENGTH (string)

Example @: LENGTH('COMPUTER') = 8
LENGTH('0') = 0
LENGTH ('') = 0

(b) Some programming languages denotes this function as follows

PL/1:	LENGTH (string)
BASIC:	LEN (string)
PASCAL:	LENGTH(string)
SNOBOL:	SIZE (string)

String storage.

Strings are stored in three types of structures.

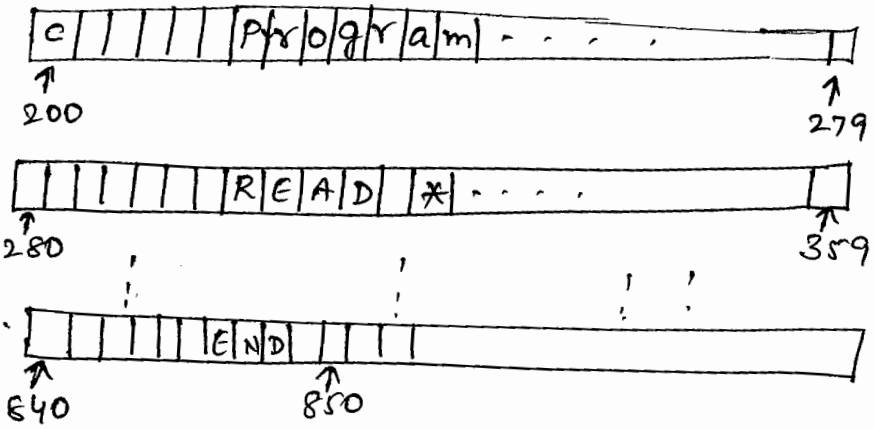
1) fixed-length structures.
2) Variable-length structures with fixed maximums
3) Linked structures.

Fixed-length (Record-Oriented) storage views records where all records have the same length.

Example: Suppose the input consist of the FORTRAN Program, using fixed length storage the input data will appear in memory as given beloro

C Program printing two integers in INC order

```
READ *, J, K
IF (J.LE.K) THEN
    PRINT *, K, J
ELSE
    PRINT *, K, J
ENDIF
STOP
END
```



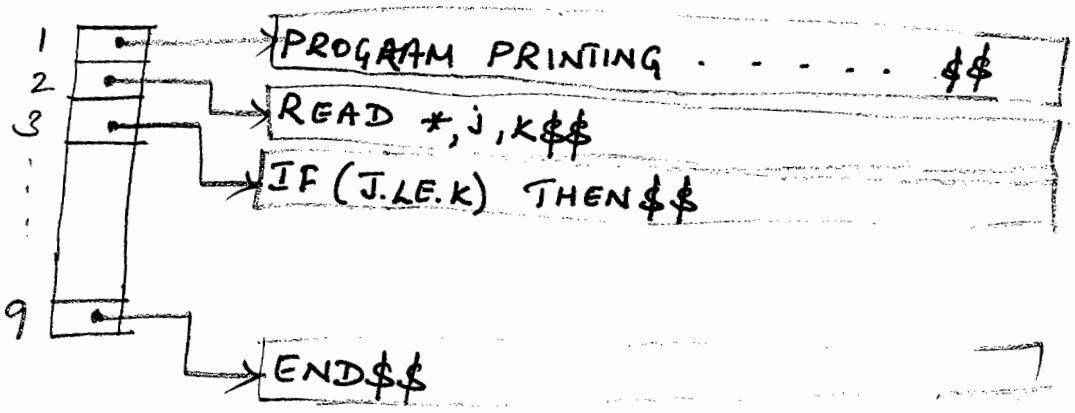
The main disadvantage of fixed-length storage are

1) Time is wasted reading an entire record if most of the storage consist of inessential blank space.
2) Certain records may require more space than available.
3) When a word correction is required the entire record has to be changed.

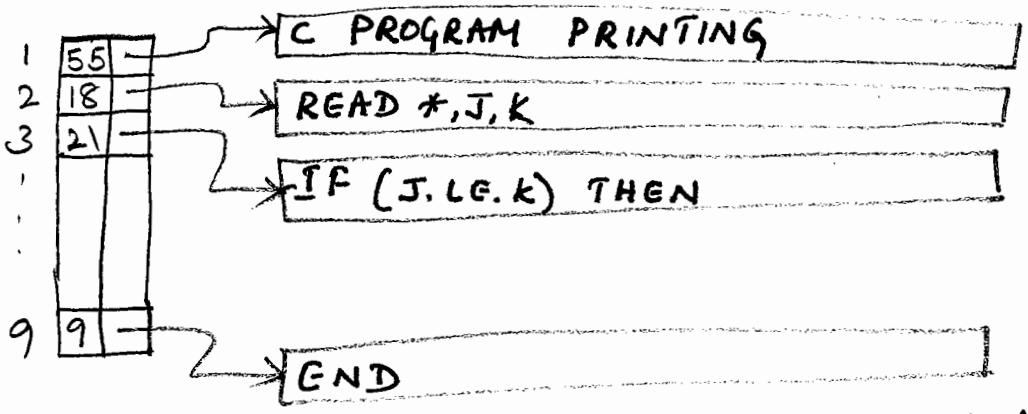
Variable length storage with fixed maximum:

The storage of variable-length strings in memory cells with fixed lengths can be done in two general ways:

1) marker such as '$$' can be used to signal the end of string.
2) Length of the string can be used as additional item in the pointer Array.



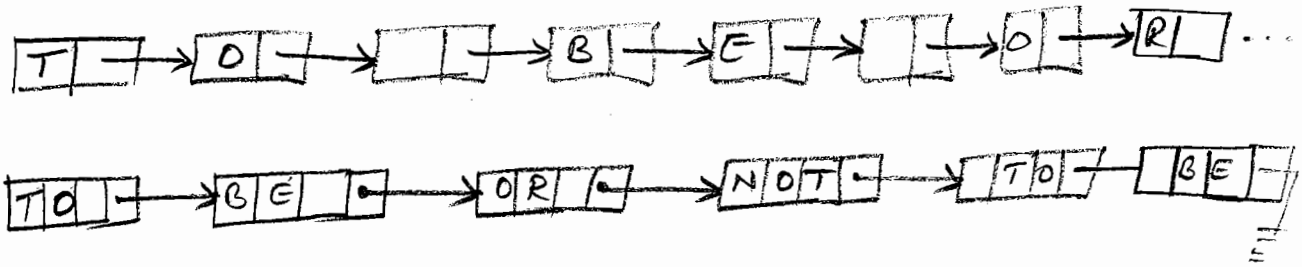
Record with sentinels ($$)



Record whose lengths are listed.

Linked storage

→ Strings may be stored in linked list.
→ A linked list is a linear ordered sequence of memory celled nodes, where each node contains an item called a link, which points to the next node in the list.



Pattern Matching Algorithm.

→ Pattern matching algorithm finds a given pattern P in a string text T. P's length is assumed not to exceed the length of T.

Algorithm

(Pattern Matching) P and T are string with lengths R and S, respectively and are stored as arrays with one character per element. This algorithm finds the index (INDEX) of P in T.

1. [Initialize.] Set K := 1 and MAX := S − R + 1.
2. Repeat Steps 3 to 5 while K ≤ MAX
3. Repeat for L = 1 to R : [Tests each character of P]
   If P[L] ≠ T[K + L − 1], then: Go to step 5
   [End of inner loop.]
4. [Success.] Set INDEX = K and EXIT.
5. Set K := K + 1.
   [END of step 2 outer loop.]
6. [Failure.] Set INDEX = 0.
7. EXIT.

T : TO BE OR NOT TO BE

P : NOT TO

R : 6 (Size of the pattern)

S : 18 (Size of the Text)

o/p : The algorithm returns 10, the first character make of the pattern in the text.

Second pattern Matching algorithm

→ This algorithm works on the bases 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 = aaba & the text has one of the following form

ⓐ T = aab... ⓑ T = aaa... ⓒ T = aax

where x is any character different from a/b.

→ The pattern matching graph for the above pattern is as given below

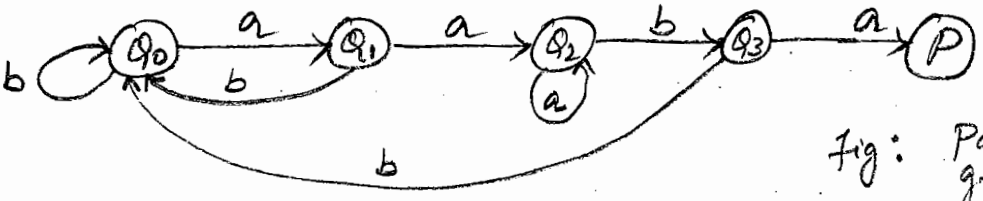


fig: Pattern matching graph.

f(Qi, t)	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

fig: pattern Matching table

→ The figure contains table used to match the pattern aaba.

→ The table is obtained as follows, Let Q denote the initial substring of P of length i, hence

$Q_0 = \Lambda$, $Q_1 = a$, $Q_2 = a^2$, $Q_3 = a^2b$, $Q_4 = a^2ba = P$

(Here $Q_0 = \Lambda$ is the empty string)

Algorithm. (Pattern Matching)

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

1. [Initialize]. Set $K := 1$ and $S_1 = Q_0$
2. Repeat steps 3 to 5 while $S_K \neq P$ and $K \leq N$.
3. Read T_K.
4. Set $S_{K+1} := F(S_K, T_K)$. [Finds next state.]
5. Set $K := K + 1$. [Updates counter.]

[End of step 2 loop.]

6. [Successful?]

If $S_K = P$, then:

INDEX = K − LENGTH(P).

Else

INDEX = 0.

[End of If structure]

7. EXIT.

```
// Program to Illustrate the usage of UNIONS in C
#include <stdio.h>
#include <string.h>
 union Data
{
     int i;
    float f;
    char str[20];
};
 int main( )
{
  union Data data;
  data.i = 10;
  data.f = 220.5;
  strcpy( data.str, "C Programming");
  printf( "data.i : %d\n", data.i);
  printf( "data.f : %f\n", data.f);
  printf( "data.str : %s\n", data.str);
  return 0;
}
```

Output:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

```
//Modified version of the previous program and see the change in the output
#include <stdio.h>
#include <string.h>
 union Data
{
    int i;
    float f;
    char str[20];
};
 int main( )
{
    union Data data;
    data.i = 10;
    printf( "data.i : %d\n", data.i);
            data.f = 220.5;
    printf( "data.f : %f\n", data.f);
    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

Write down the output of this program:

```
//Program to illustrate the usage of dynamic memory allocation for arrays
//This Program adds n numbers stored in the array
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
    if(ptr==NULL)
    {
            printf("Error! memory not allocated.");
            exit(0);
    }
    printf("Enter elements of array: ");
     for(i=0;i<n;++i)
    {
            scanf("%d",ptr+i);
            sum+=*(ptr+i);
    }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

Snapshot to illustrate the creation of two dimension array using dynamic memory management

Equivalent memory allocation for char a[10][20] would be as follows.

```
char **a;

a=(char **) malloc(10*sizeof(char *));

for(i=0;i<10;i++)
    a[i]=(char *) malloc(20*sizeof(char));
```

```
int main(int argc, char *argv[])
{
     int i;

     double* p;     // We uses this reference variable to access
                    // dynamically created array elements

    p = calloc(10, sizeof(double) );  // Make double array of 10 elements

       for ( i = 0; i < 10; i++ )
          *(p + i) = i;             // put value i in array element i

       for ( i = 0; i < 10; i++ )
          printf("*(p + %d) = %lf\n", i, *(p+i) );


       free(p);       // Un-reserve the first array

       putchar('\n');

       p = calloc(4, sizeof(double) );  // Make a NEW double array of 4
elements

       // ***** Notice that the array size has CHANGED !!! ****

       for ( i = 0; i < 4; i++ )
          *(p + i) = i*i;             // put value i*i in array element i

       for ( i = 0; i < 4; i++ )
          printf("*(p + %d) = %lf\n", i, *(p+i) );

       free(p);       // Un-reserve the second array
    }
```

Output

```
*(p + 0) = 0.000000
*(p + 1) = 1.000000
*(p + 2) = 2.000000
*(p + 3) = 3.000000
*(p + 4) = 4.000000
*(p + 5) = 5.000000
*(p + 6) = 6.000000
*(p + 7) = 7.000000
*(p + 8) = 8.000000
*(p + 9) = 9.000000

*(p + 0) = 0.000000
*(p + 1) = 1.000000
*(p + 2) = 4.000000
*(p + 3) = 9.000000
```

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
      int r = 3, c = 4;
      int *arr = (int *)malloc(r * c * sizeof(int));

      int i, j, count = 0;
      for (i = 0; i <  r; i++)
        for (j = 0; j < c; j++)
           *(arr + i*c + j) = ++count;

      for (i = 0; i <  r; i++)
        for (j = 0; j < c; j++)
           printf("%d ", *(arr + i*c + j));

   /* Code for further processing and free the
       dynamically allocated memory */

   return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10 11 12
```

```
#include<stdio.h>

#define MAX_TERMS 101
typedef struct
{
      int col;
      int row;
      int value;
} term;
term a[MAX_TERMS]

void printsparse(int[][3]);
void readsparse(int[][3]);
void transpose(int[][3],int[][3]);

int main()
{
      int b1[MAX][3],b2[MAX][3],m,n;
      printf("Enter the size of matrix (rows,columns):");

      scanf("%d%d",&m,&n);
      b1[0][0]=m;
      b1[0][1]=n;

      readsparse(b1);
      transpose(b1,b2);
      printsparse(b2);
}

void readsparse(int b[MAX][3])
{
      int i,t;
      printf("\nEnter no. of non-zero elements:");
      scanf("%d",&t);
      b[0][2]=t;

      for(i=1;i<=t;i++)
      {
          printf("\nEnter the next triple(row,column,value):");
          scanf("%d%d%d",&b[i][0],&b[i][1],&b[i][2]);
      }
}

void printsparse(int b[MAX][3])
{
      int i,n;
      n=b[0][2];    //no of 3-triples

      printf("\nAfter Transpose:\n");

      printf("\nrow\t\tcolumn\t\tvalue\n");
      for(i=0;i<=n;i++)
          printf("%d\t\t%d\t\t%d\n",b[i][0],b[i][1],b[i][2]);
```

```
}

void transpose(int b1[][3],int b2[][3])
{
      int i,j,k,n;
      b2[0][0]=b1[0][1];
      b2[0][1]=b1[0][0];
      b2[0][2]=b1[0][2];

      k=1;
      n=b1[0][2];

      for(i=0;i<b1[0][1];i++)
          for(j=1;j<=n;j++)
              //if a column number of current triple==i then insert
the current triple in b2
              if(i==b1[j][1])
              {
                  b2[k][0]=i;
                  b2[k][1]=b1[j][0];
                  b2[k][2]=b1[j][2];
                  k++;
              }
}
```

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- *Stack.*
- *Queue.*

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

4.1. STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push*: is the term used to insert an element into a stack.
- *Pop*: is the term used to delete an element from a stack.

"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

4.1.1. Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by push(). Figure 4.1 shows the creation of a stack and addition of elements using push().

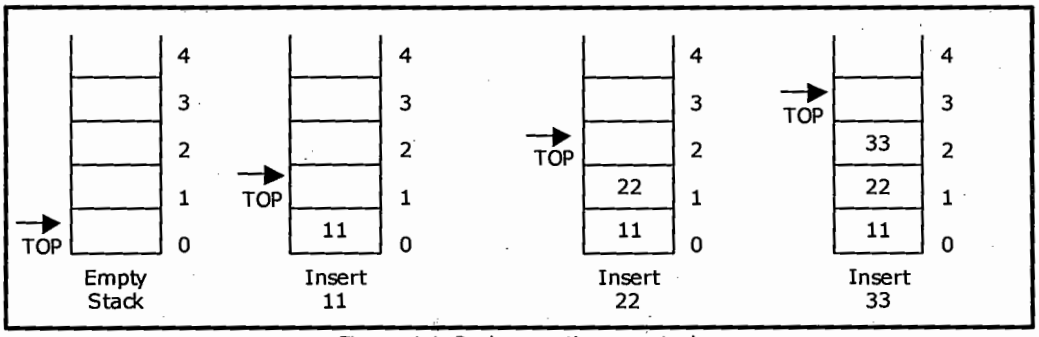


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

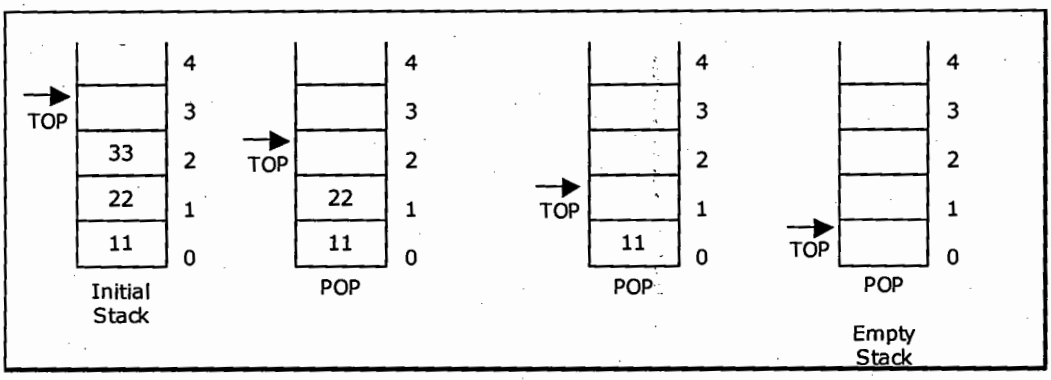


Figure 4.2. Pop operations on stack

4.1.2. Source code for stack operations, using array:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# define MAX 6
int stack[MAX];
int top = 0;
int menu()
{
        int ch;
        clrscr();
        printf("\n ... Stack operations using ARRAY... ");
        printf("\n ----------**********----------\n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void display()
{
        int i;
        if(top == 0)
        {
                printf("\n\nStack empty..");
```

```
                return;
        }
        else
        {
                printf("\n\nElements in stack:");
                for(i = 0; i < top; i++)
                        printf("\t%d", stack[i]);
        }
}

void pop()
{
        if(top == 0)
        {
                printf("\n\nStack Underflow..");
                return;
        }
        else
                printf("\n\npopped element is: %d ", stack[--top]);
}

void push()
{
        int data;
        if(top == MAX)
        {
                printf("\n\nStack Overflow..");
                return;
        }
        else
        {
                printf("\n\nEnter data: ");
                scanf("%d", &data);
                stack[top] = data;
                top = top + 1;
                printf("\n\nData Pushed into the stack");
        }
}

void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                push();
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                display();
                                break;

                        case 4:
                                exit(0);
                }
                getch();
        } while(1);
}
```

4.1.3. Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

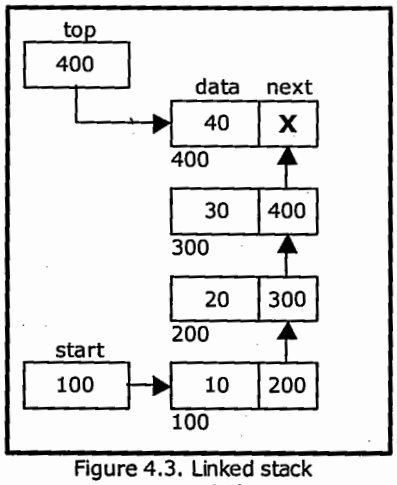


Figure 4.3. Linked stack representation

4.1.4. Source code for stack operations, using linked list:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct stack
{
        int data;
        struct stack *next;
};

void push();
void pop();
void display();
typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
        node *temp;
        temp=(node *) malloc( sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void push(node *newnode)
{
        node *temp;
        if( newnode == NULL )
        {
                printf("\n Stack Overflow..");
                return;
        }
```

```
        if(start == NULL)
        {
                start = newnode;
                top = newnode;
        }
        else
        {
                temp = start;
                while( temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
                top = newnode;
        }
        printf("\n\n\t Data pushed into stack");
}
void pop()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t Stack underflow");
                return;
        }
        temp = start;
        if( start -> next == NULL)
        {
                printf("\n\n\t Popped element is %d ", top -> data);
                start = NULL;
                free(top);
                top = NULL;
        }
        else
        {
                while(temp -> next != top)
                {
                        temp = temp -> next;
                }
                temp -> next = NULL;
                printf("\n\n\t Popped element is %d ", top -> data);
                free(top);
                top = temp;
        }
}
void display()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t\t Stack is empty ");
        }
        else
        {
                temp = start;
                printf("\n\n\n\t\t Elements in the stack: \n");
                printf("%5d  ", temp -> data);
                while(temp != top)
                {
                        temp = temp -> next;
                        printf("%5d   ", temp -> data);
                }
        }
}
```

```
char menu()
{
        char ch;
        clrscr();
        printf("\n \tStack operations using pointers.. ");
        printf("\n ----------**********-----------\n");
        printf("\n 1. Push ");
        printf("\n 2. Pop ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}

void main()
{
        char ch;
        node *newnode;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                newnode = getnode();
                                push(newnode);
                                break;
                        case '2' :
                                pop();
                                break;
                        case '3' :
                                display();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while( ch != '4' );
}
```

4.2. Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as

polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression un-ambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

4.3. Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

4.3.1. Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	

*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(	a b c * +	+ (	
d	a b c * + d	+ (	
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	

(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.2. Program to convert an infix to postfix expression:

```
# include <string.h>

char postfix[50];
char infix[50];
char opstack[50];                    /*  operator stack  */
int i, j, top = 0;

int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                     /*  priority value of  op  */
        int pv2;                     /*  priority value of op_at_stack  */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};
        if( op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}
```

```
void push(char op)        /* op - operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != '(' )
                {
                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        postfix[j] = opstack[--top];
                                        j++;
                        }
                }
                opstack[top] = op;      /* pushing onto stack  */
                top++;
        }
}

pop()
{
        while(opstack[--top] != '(' )             /*  pop until '(' comes   */
        {
                postfix[j] = opstack[top];
                j++;
        }
}

void main()
{
        char ch;
        clrscr();
        printf("\n Enter Infix Expression  : ");
        gets(infix);
        while( (ch=infix[i++]) != '\0')
        {
                switch(ch)
                {
                        case ' ' : break;
                        case '(' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push  */
                                break;
                        case ')' :
                                pop();
                                break;
                        default :
                                postfix[j] = ch;
                                j++;
                }
        }
        while(top >= 0)
        {
                postfix[j] =  opstack[--top];
                j++;
```

/* before pushing the operator 'op' into the stack check priority of op with top of opstack if less then pop the operator from stack then push into postfix string else push op onto stack itself */

```
        }
        postfix[j] = '\0';
        printf("\n Infix  Expression    :  %s ", infix);
        printf("\n Postfix Expression  :  %s ", postfix);
        getch();
}
```

4.3.3. Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B - C into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the infix expression (A + B) * (C - D) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
D	D	)	
-	D	) -	
C	C D	) -	
(	- C D		
*	- C D	*	
)	- C D	*)	
B	B - C D	*)	
+	B - C D	*) +	
A	A B - C D	*) +	
(	+ A B – C D	*	
End of string	* + A B – C D	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.4. Program to convert an infix to prefix expression:

```
# include <conio.h>
# include <string.h>

char prefix[50];
char infix[50];
char opstack[50];                    /* operator stack  */
int j, top = 0;

void insert_beg(char ch)
{
        int k;
        if(j == 0)
                prefix[0] =  ch;
        else
        {
                for(k = j + 1; k > 0; k--)
                        prefix[k] = prefix[k - 1];
                prefix[0] = ch;
        }
        j++;
}
```

```
int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                    /* priority value of  op  */
        int pv2;                    /* priority value of op_at_stack  */
        char operators[] = {'+', '-', '*', '/', '%', '^', ')'};
        int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
        if(op_at_stack == ')' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if( op_at_stack == operators[k] )
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}

void push(char op)              /* op - operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != ')')
                {
                        /* before pushing the operator 'op' into the stack check priority of op
                with top of operator stack if less pop the operator from stack then push into
                postfix string else push op onto stack itself  */

                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        insert_beg(opstack[--top]);
                        }
                }
                opstack[top] = op;                      /* pushing onto stack   */
                top++;
        }
}

void pop()
{
        while(opstack[--top] != ')')                    /*  pop until ')' comes;  */
                insert_beg(opstack[top]);
}


void main()
{
        char ch;
        int l, i = 0;
        clrscr();
        printf("\n Enter Infix Expression  : ");
```

```
        gets(infix);
        l = strlen(infix);
        while(l > 0)
        {
                ch = infix[--l];
                switch(ch)
                {
                        case ' ' : break;
                        case ')' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push  */
                                break;
                        case '(' :
                                pop();
                                break;
                        default :
                                insert_beg(ch);
                }
        }
        while( top > 0 )
        {
                insert_beg( opstack[--top] );
                j++;
        }
        prefix[j] = '\0';
        printf("\n Infix Expression    :  %s ", infix);
        printf("\n Prefix Expression  :  %s ", prefix);
        getch();
}
```

4.4. Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

4.4.1. Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20

int isoperator(char ch)
{
        if(ch == '+' || ch == '-' ||  ch == '*' || ch == '/' ||  ch == '^')
                return 1;
        else
                return 0;
}
```

```
void main(void)
{
        char postfix[MAX];
        int val;
        char ch;
        int i = 0, top = 0;
        float val_stack[MAX], val1, val2, res;
        clrscr();
        printf("\n Enter a postfix expression: ");
        scanf("%s", postfix);
        while((ch = postfix[i]) != '\0')
        {
                if(isoperator(ch) == 1)
                {
                        val2 = val_stack[--top];
                        val1 = val_stack[--top];
                        switch(ch)
                        {
                                case '+':
                                        res = val1 + val2;
                                        break;
                                case '-':
                                        res = val1 - val2;
                                        break;
                                case '*':
                                        res = val1 * val2;
                                        break;
                                case '/':
                                        res = val1 / val2;
                                        break;
                                case '^':
                                        res = pow(val1, val2);
                                        break;
                        }
                        val_stack[top] = res;
                }
                else
                        val_stack[top] = ch-48;  /*convert character digit to integer digit */
                top++;
                i++;
        }
        printf("\n Values of %s is : %f ",postfix, val_stack[0] );
        getch();
}
```

4.5. Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.

2. Stack is used to evaluate a postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor's stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

4.6. Queue:

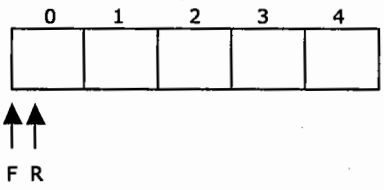
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

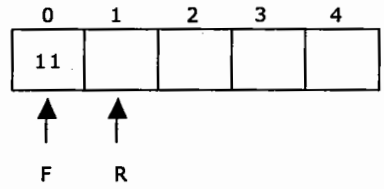
4.6.1. Representation of Queue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



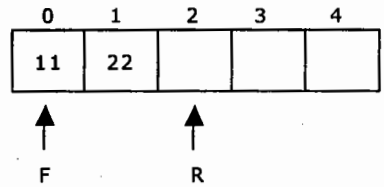
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



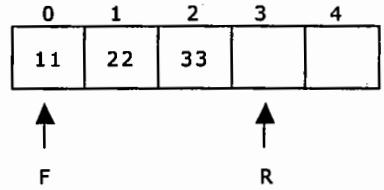
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



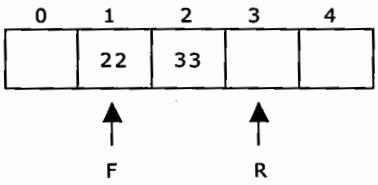
REAR = REAR + 1 = 2
FRONT = 0

Again insert another element 33 to the queue. The status of the queue is:



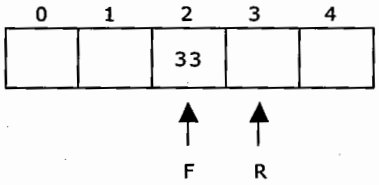
REAR = REAR + 1 = 3
FRONT = 0

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



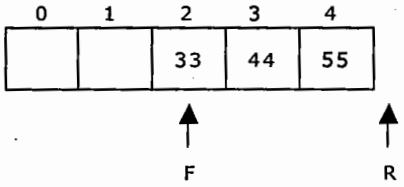
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



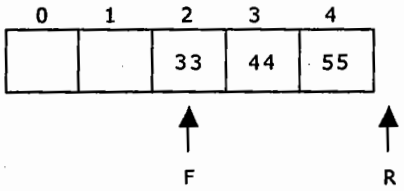
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



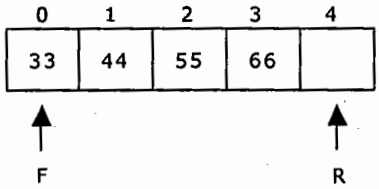
REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



REAR = 4
FRONT = 0

This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

4.6.2. Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

```
# include <conio.h>
# define MAX 6
int Q[MAX];
int front, rear;

void insertQ()
{
        int data;
        if(rear == MAX)
        {
                printf("\n Linear Queue is full");
                return;
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                Q[rear] = data;
                rear++;
                printf("\n Data Inserted in the Queue ");
        }
}
void deleteQ()
{
        if(rear == front)
        {
                printf("\n\n Queue is Empty..");
                return;
        }
        else
        {
                printf("\n Deleted element from Queue is %d", Q[front]);
                front++;
        }
}
void displayQ()
{
        int i;
        if(front == rear)
        {
                printf("\n\n\t Queue is Empty");
                return;
        }
        else
        {
                printf("\n Elements in Queue are: ");
                for(i = front; i < rear; i++)
```

```
                {
                        printf("%d\t", Q[i]);
                }
        }
}
int menu()
{
        int ch;
        clrscr();
        printf("\n \tQueue operations using ARRAY..");
        printf("\n ----------**********----------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertQ();
                                break;
                        case 2:
                                deleteQ();
                                break;
                        case 3:
                                displayQ();
                                break;
                        case 4:
                                return;
                }
                getch();
        } while(1);
}
```

4.6.3. Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

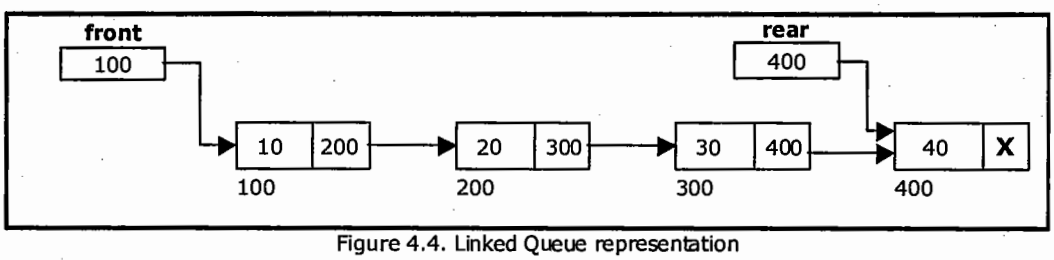


Figure 4.4. Linked Queue representation

4.6.4. Source code for queue operations using linked list:

```
# include <stdlib.h>
# include <conio.h>

struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear =  NULL;

node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
                rear = newnode;
        }
        else
        {
                rear -> next = newnode;
                rear = newnode;
        }
        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t Empty Queue..");
                return;
        }
        temp = front;
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp -> data);
        free(temp);
}
```

```
void displayQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t\t Empty Queue ");
        }
        else
        {
                temp = front;
                printf("\n\n\n\t\t Elements in the Queue are: ");
                while(temp != NULL )
                {
                        printf("%5d  ", temp -> data);
                        temp = temp -> next;
                }
        }
}

char menu()
{
        char ch;
        clrscr();
        printf("\n \t..Queue operations using pointers.. ");
        printf("\n\t ----------**********------------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}

void main()
{
        char ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                insertQ();
                                break;
                        case '2' :
                                deleteQ();
                                break;
                        case '3' :
                                displayQ();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while(ch != '4');
}
```

4.7. Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

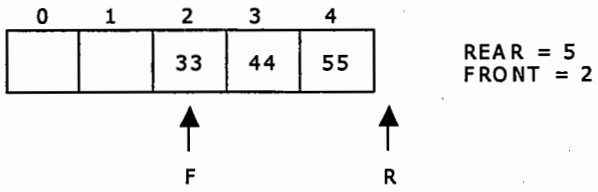
4.8. Circular Queue:

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

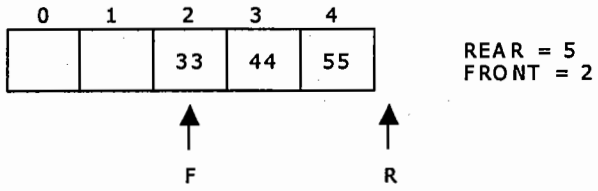
There are two problems associated with linear queue. They are:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

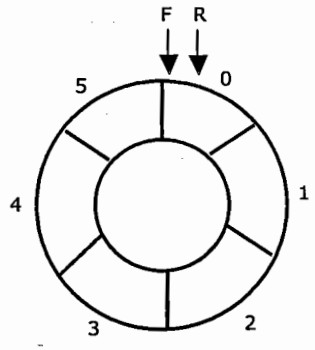


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

4.8.1. Representation of Circular Queue:

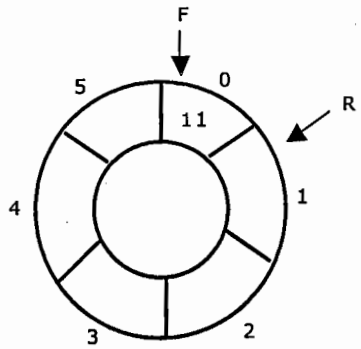
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

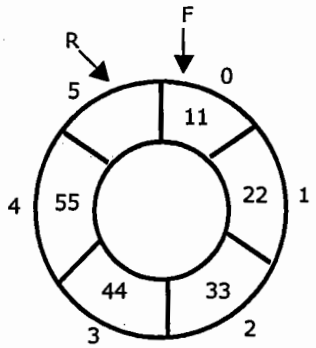
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

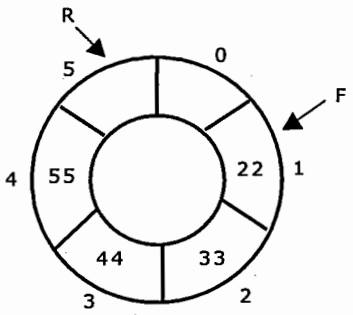
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

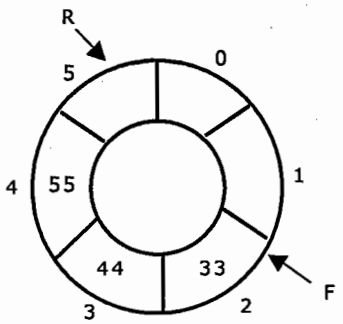
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

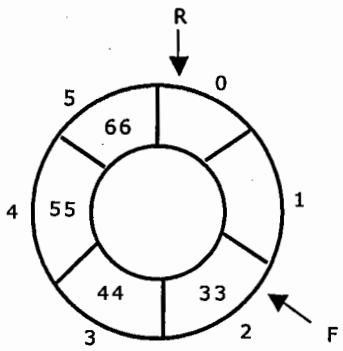
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

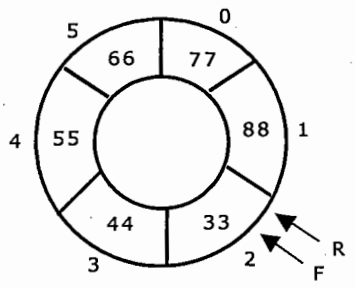
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

4.8.2. Source code for Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6

int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;

void insertCQ()
{
        int data;
        if(count == MAX)
        {
                printf("\n Circular Queue is Full");
        }
        else
        {
                printf("\n Enter data: ");
                scanf("%d", &data);
                CQ[rear] = data;
                rear = (rear + 1) % MAX;
                count ++;
                printf("\n Data Inserted in the Circular Queue ");
        }
}

void deleteCQ()
{
        if(count == 0)
        {
                printf("\n\nCircular Queue is Empty..");
        }
        else
        {
                printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
                front = (front + 1) % MAX;
                count --;
        }
}
```

```
void displayCQ()
{
	int i, j;
	if(count == 0)
	{
		printf("\n\n\t Circular Queue is Empty ");
	}
	else
	{
		printf("\n Elements in Circular Queue are: ");
		j = count;
		for(i = front; j != 0; j--)
		{
			printf("%d\t", CQ[i]);
			i = (i + 1) % MAX;
		}
	}
}

int menu()
{
	int ch;
	clrscr();
	printf("\n \t Circular Queue Operations using ARRAY..");
	printf("\n ----------**********----------\n");
	printf("\n 1. Insert ");
	printf("\n 2. Delete ");
	printf("\n 3. Display");
	printf("\n 4. Quit ");
	printf("\n Enter Your Choice: ");
	scanf("%d", &ch);
	return ch;
}

void main()
{
	int ch;
	do
	{
		ch = menu();
		switch(ch)
		{
			case 1:
				insertCQ();
				break;
			case 2:
				deleteCQ();
				break;
			case 3:
				displayCQ();
				break;
			case 4:
				return;
			default:
				printf("\n Invalid Choice ");
		}
		getch();
	} while(1);
}
```

4.9. Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

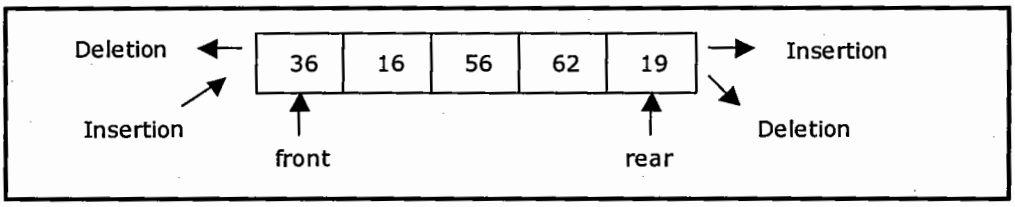


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

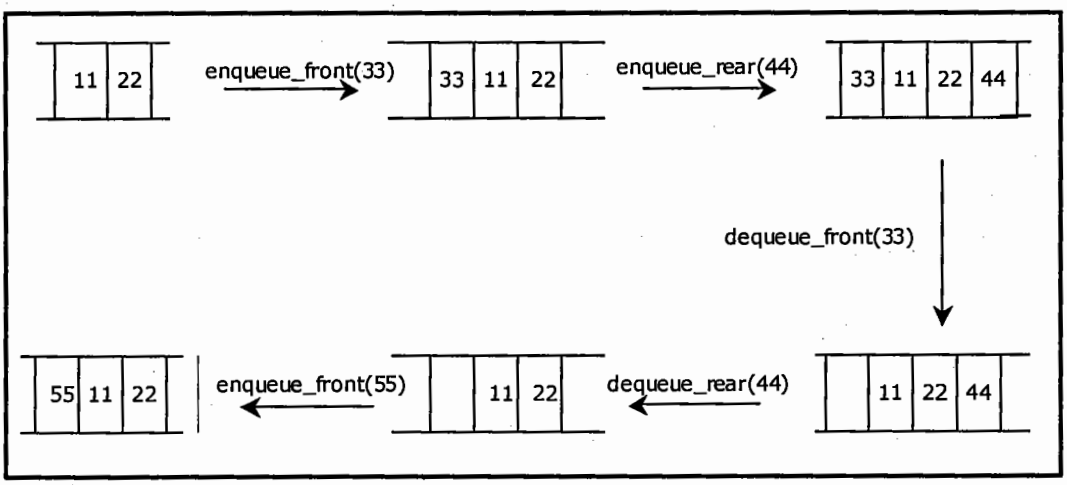


Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

4.10. 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:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Infix to postfix Conversion

```
typedef enum {lparen, reparen , plus, minus,
              times, divide, mod, eos,
              Operand} precedence;
precedence getToken(char *symbol, int *n)
{
    *symbol = expr[(*n)++];
    switch (*symbol)
    {
        case '(' : return lparen;
        case ')' : return reparen;
        case '+' : return plus;
        case '-' : return minus;
        case '/' : return divide;
        case '*' : return times;
        case '%' : return mod;
        case '\0' : return eos;
        default : return Operand;
    }
}
```

```
/* isp & icp arrays - index is value of
 precedence lparen, rparen, . . . . eos */
int isp[] = {0, 19, 12, 12, 13, 13, 13, 0};
int icp[] = {20, 19, 12, 12, 13, 13, 13, 0};
```

```
void postfix (void)
{
    char symbol;
    precedence token;
    int n=0;
    int top=0;
    stack[0] = eos;
    for(token = getToken(&symbol, &n);
        token != eos;
        token = getToken(&symbol, &n))
    {
        if(token == operand)
            printf("%c", symbol);
        else if (token == rparen)
        {
            while (stack[top] != lparen)
                printToken(pop());
            pop();
        }
        else
        {
            while (isp[stack[top]] >=
                   icp[token])
                printToken(pop());
            push(token);
        }
    }
    while ((token = pop()) != eos)
        printToken(token);
    printf("\n");
}
```

// Multiple stack Implementation.

```
#include <stdio.h>
#include <stdlib.h>
#define max 20
int push(int [max], int, int [], int [], int *);
int pop(int [max], int, int [], int [], int *);
Void main()
{
    int stack[max], data, n, size, sno;
    int top[10], bott[10], limit[10];
    int i, option, reply;
    Printf("In How many stacks? ");
    scanf("%d", &n);
    size = max / n;
    // Initializing bottom of each stack.
    bott[0] = -1;
    for (i=1; i<n; i++)
        bott[i] = bott[i-1] + size;
    // Initialize limit of each stack
    for (i=0; i<n; i++)
        limit[i] = bott[i] + size;
    // Initialize top of each stack.
    for (i=0; i<n; i++)
        top[i] = bott[i]
```

```
// process the stack
do
{
    printf ("\n Multiple stack Imple \n");
    Printf (" \n1. Push");
                2. Pop
                3. Exit
                select the option
    scanf ("%d", &option);
    Switch (option)
    {
        Case 1: printf ("\n Enter a logical stack no
                                    : %d", n-1);
                scanf ("%d", &sno);
                Printf ("\n Enter a value : ");
                scanf ("%d", &data);
                reply = push (stack, sno, top, limit,
                                    &data);
                if (reply == -1)
                    Printf ("\n stack %d is Full", sno);
                else
                    printf ("\n%d is Pushed in
                                stack No %d \n", data, sno);
                break;
        case 2 : printf ("\n Enter a logical stack
                        number (0 to %d) : ", n-1);
                scanf ("%d", &sno);
                reply = pop (stack, sno, top, bott, &data);
                if (reply == -1)
                    printf ("\n stack %d is Empty \n",
                                sno);
                else
                    printf ("\n%d is popped from
                        stack no %d \n", data, sno);
                break;
        Case 3 : exit(0);
    }
} while (1);
}
```

```
int push (int stack[max], int sno, int top[]
                                    int limit[], int *data)
{
    if (top[sno] == limit[sno])
        return(-1);
    else
    {
        top[sno]++;
        stack[top[sno]] = *data;
        return(1);
    }
}
int pop (int stack[max], int sno, int top[]
                                    int limit, int *data)
                                    int bott[]
{
    if (top[sno] == bott[sno])
        return(-1);
    else
    {
        *data = stack[~~sno~~top[sno]];
        top[sno]--;  // top[sno] = top[sno] - 1
        return(1);
    }
}
```

Recursion

Factorial, GCD, Fibonacci Series, Tower of Hanoi, Ackerman's function.

Recursion: A recursive procedure contains either a call statement to itself (Direct recursion) or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P (Indirect Recursion).

An recursive procedure should adhere to two properties

1) There must be a base criteria, for which the procedure does not call itself.
2) Each time the procedure does call itself, it must be closer to the base criteria.

Factorial Function.

→ The product of the +ve integers from 1 to n, inclusive, is called "n factorial" & is denoted as $n!$. ie $n = 1.2.3.\ldots.(n-2)(n-1)n$

note: $0! = 1$

→ The factorial function is given by

If $n=0$, then $n! = 1$

If $n>0$, then $n! = n.(n-1)!$

Recurrence Relation

$$n! = \begin{cases} 1 & \text{if } n=1 \\ n.(n-1)! & \text{if } n>1 \end{cases}$$

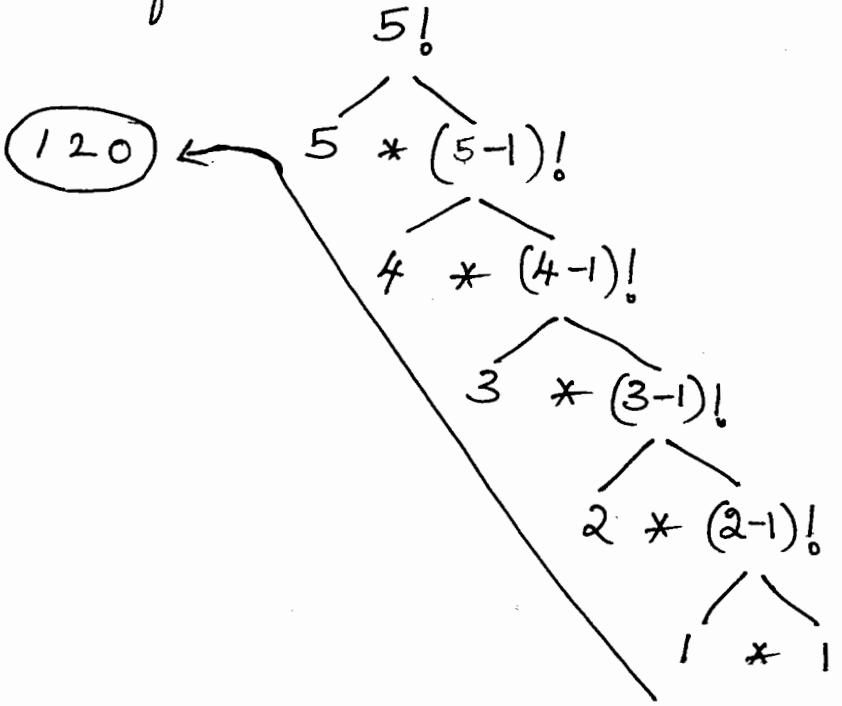
Algorithm

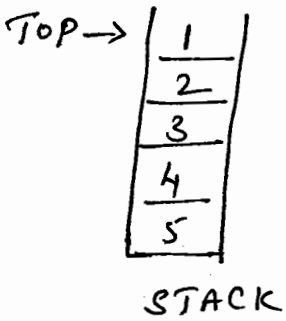
FACTORIAL (FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N=0, then: Set FACT := 1 and Return.
2. Call FACTORIAL(FACT, N-1).
3. Set FACT := N * FACT
4. Return.

```
// C function.
int factorial (int n)
{
    if (n>0)
        return (n * factorial (n-1));
    else
        return (1);
}
```

The recursive tree below shows the working of this function.





Fibonacci Sequence.

→ Is as follows 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 . . . . . .

→ The fibonacci sequence function is given by

a) If $n=0$ or $n=1$, then $F_n = n$.

b) If $n>1$, then $F_n = F_{n-2} + F_{n-1}$

Recurrence Relation (linear Equation)

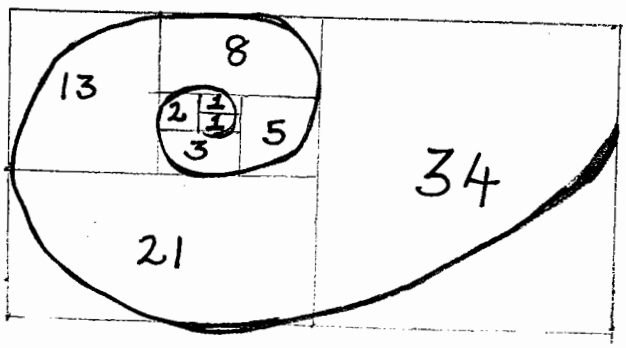
$$F_n = F_{n-1} + F_{n-2}$$

with $F_1 = F_2 = 1$, $F_0 = 0$

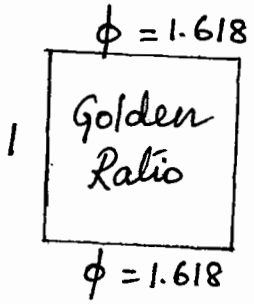
FIBONACCI (FIB, N)

This procedure calculates F_N & returns the value in the first parameter FIB.

1. If N=0 or N=1, then: set FIB:=N and Return.
2. Call FIBONACCI (FIBA, N-2)
3. Call FIBONACCI (FIBB, N-1)
4. Set FIB:= FIBA + FIBB.
5. Return.



when we make squares with those widths, we get a nice spiral.



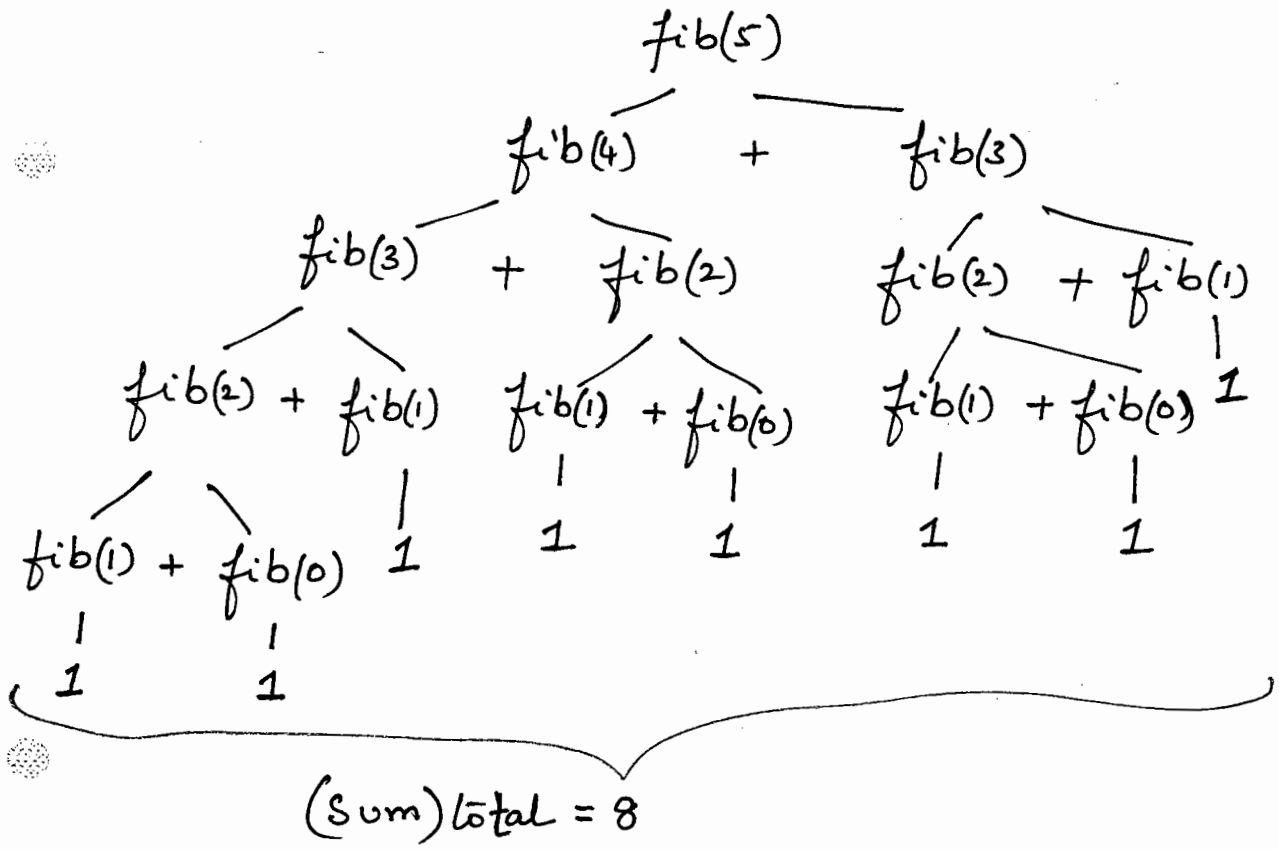
☺ Golden Ratio: when we take any two successive Fibonacci numbers, their ratio is approximately 1.618034...

```
int Fibonacci (int n)
{
    if (n==0)
        return 0;
    Else if (n==1)
        return 1;
    Else
        return (Fibonacci(n+1) + Fibonacci(n-2));
}
```

0 1 2 3 4 5

```
/* Fibonacci Series - 1, 1, 2, 3, 5, 8, 13 ... */
int fib (int n)
{
    if (n==0 || n==1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

The recursive tree for fib(5) is shown below



(Sum) total = 8

GCD (Greatest common divisor)

→ A well-known algorithm for finding the greatest common divisor of two integers is Euclid's Algorithm.

→ The GCD function is defined by the following recurrence relation

$$GCD(m,n) = \begin{cases} GCD(n,m), & \text{if } n>m \\ m, & \text{if } n=0 \\ GCD(n, MOD(m,n)), & \text{otherwise} \end{cases}$$

// c program for GCD of two numbers

```
int gcd (int m, int n)
{
    if (n>m)
        gcd (n,m);
    Else
        if (n==0)
            return (m);
        Else
            gcd (n, mod (m,n));
}
```

Recursive tree for gcd (84, 246)

gcd (84, 246)
|
gcd (246, 84) gcd(n, m)
|
gcd (84 ; 246 mod 84) gcd (n, mod(m,n))
(78)
|
gcd (84, 78) gcd(n, m)
|
gcd (78, 6) gcd (n, mod(m,n))
|
return 6. return m

Ackermann Function.

→ The Ackermann function is a function with two arguments each of which can be assigned an non-negative integer 0, 1, 2 ....

→ It is defined for nonnegative integers m & n as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1, A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

C - Implementation of Ackermann function.

```
int Ack_Fun(int m, int n)
{
    if (m==0) return n+1;
    if (m>0 && n==0) return A(m-1,1);
    if (m>0 && n>0) return A(m-1, A(m,n-1));
}
```

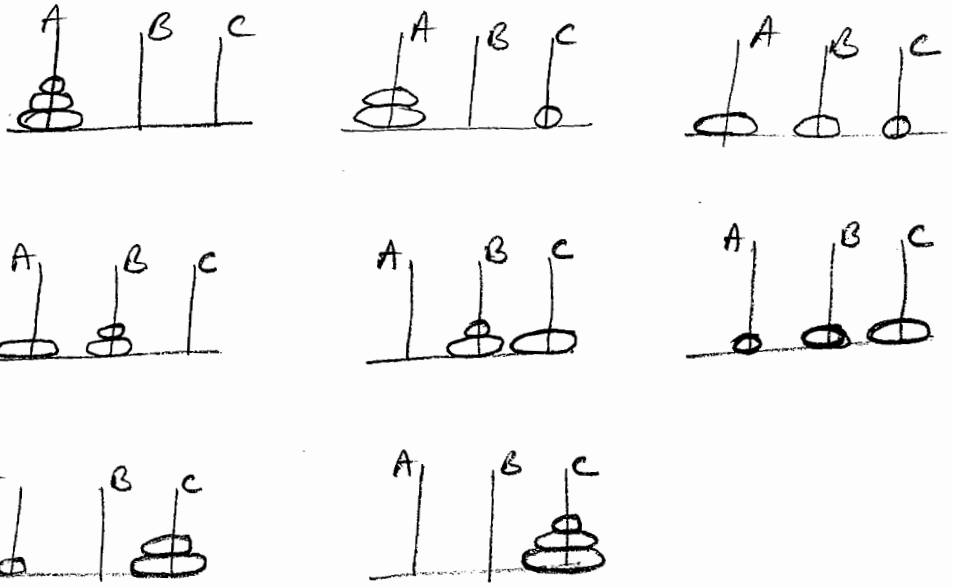
Tower of Hanoi

The following rules are adopted when playing the game of tower of Hanoi.

1) Only one disk may be moved at a time. only the top disk on any peg may be moved to any other peg.
2) At no time can a larger disk be placed on a smaller disk.

→ Suppose three pegs labeled A, B & C are given The objective of the game is to move the disks from peg A to peg C using peg B as an Auxillary.

→ The diagram below shows the moves.



Algorithm for Tower of Hanoi

Tower (N, BEG, AUX, END)
This procedure gives a recursive solution to Tower of Hanoi problem for N disks.

1. If N = 1, then
   a) Write: BEG → END
   b) Return.
   [End of If Structure]

2. [Move N-1 disks from peg BEG to peg AUX]
   Call TOWER(N-1, BEG, END, AUX)
3. Write BEG → END.
4. [Move N-1 disks from peg AUX to peg END]
   Call Tower(N-1, AUX, BEG, END)
5. Return.

C program to Implement Tower of Hanoi

```
Void TOH (int, char, char, char);
int main()
{
    int num;
    printf("\n Enter number of plates:");
    scanf("%d", &num);
    TOH (num-1, 'A', 'B', 'C');
    return(0);
}
Void TOH (int num, char x, char y, char z)
{
    if (num>0)
    {
        TOH (num-1, x, z, y);
        printf("\n%c → %c", x, z);
        TOH (num-1, y, x, z);
    }
}
```

printf("Move a Disc from %d to %d", x, z);

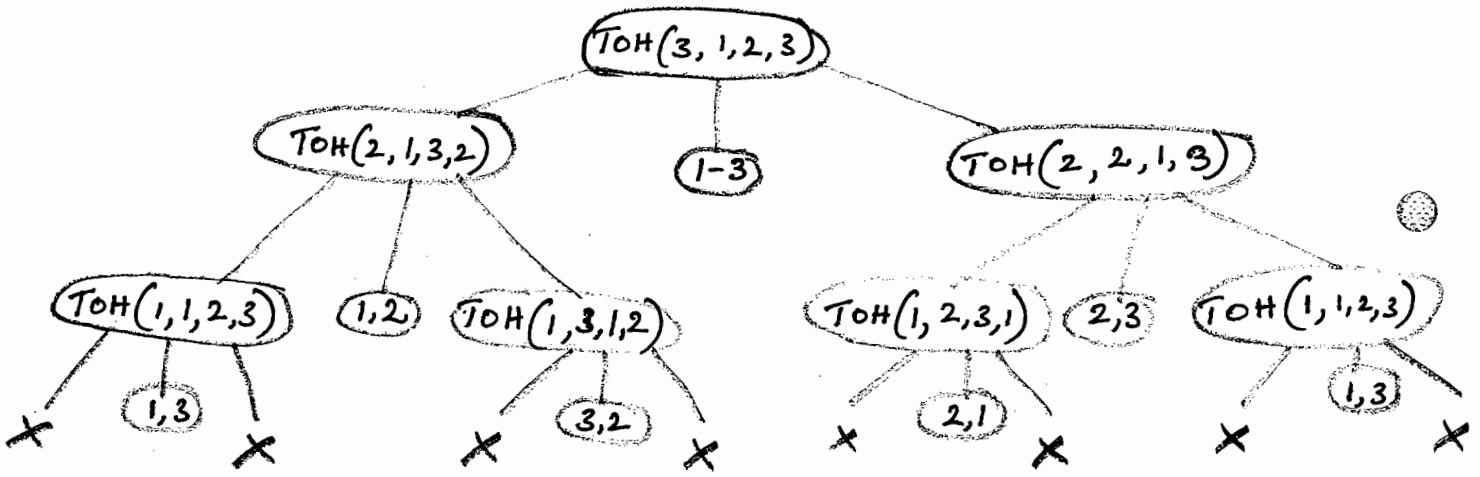
Note:

1. you can only move 1 disk at a time.
2. Larger disk can never be above smaller disk.

To move a disk from peg A to C

1. Move (n-1) disk from peg A to B using C as intermediate peg.
2. Move nth disk from peg A to C
3. Move (n-1) disk from B to C using A as intermediary.

```
Void TOH (int n, int A, int B, int C)
{
    if (n>0)
    {
        TOH (n-1, A, C, B);
        Printf("Move a Disk from %d to %d", A, C);
        TOH(n-1, B, A, C);
    }
}
```



1, 3
1, 2
3, 2
1, 3
2, 1
2, 3
1, 3

} 7 steps

Recursion.

① find the largest element of an array by recursion.

```
int size; // Global
int MaxElement(int arr[])
{
    static int i=0, Max = -9999;
    if (i< size)
    {
        if(max < arr[i])
            max = arr[i];
        i++;
    }   MaxElement(arr);
    return Max;
}
```

C program to implement an Ordinary queue.

```
#include <stdio.h>
#include <stdlib.h>
#define QSIZE 5
int qfull(int rear)
{
    return (r == QSIZE-1) ? 1 : 0;
}
int qempty(int front, int rear)
{
    return (front > rear) ? 1 : 0;
}
void q_insert(int item, int queue[], int *rear)
{
    if (qfull(*rear))
    {
        printf("Queue full");
    }   return;
    queue[++(*rear)] = item;
}
void q_delete(int queue[], int *front, int *rear)
{
    if (qempty(*front, *rear))
    {
        printf("Queue Empty");
        return;
    }
    printf("The element deleted is %d \n",
                        queue[(*f)++]);
    if (*front > *rear)
    {
        *front = 0; *rear = -1;
    }
}
```

```
void display (int queue[], int front, int rear)
{
    int i;
    if (qempty(front, rear))
    {
        printf("Queue is empty \n"); return;
    }
    printf("Contents of queue is \n");
    for (i=front; i<=rear; i++)
        Printf("%d \n", queue[i]);
}
void main()
{
    int choice, item, front, rear, queue[10];
    front=0; rear=-1;
    for(;;)
    {
        printf("1. Insert \n 2. Delete \n");
        printf("3. Display \n 4. Exit \n");
        printf("Enter the choice \n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("Enter the item to be
                        inserted \n");
                    scanf("%d", &item);
                    qinsert(item, queue, &rear);
                    break;
            case 2: q-delete(queue, &front, &rear);
                    break;
            case 3: display(queue, front, rear);
                    break;
            case 4: exit(0);
        }
    }
}
```

Double ended Queue (Deque)

Definition: A Deque is a special type of data-structure in which insertions & deletions will be done either at the font end or at the rear end of the queue.

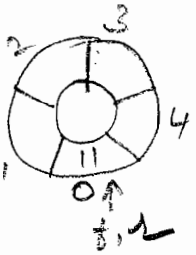
```
// insertion from front for the queue.
// Special case - Dequeue.
void q_insert_front (int item, int queue[],
                     int *front, int *rear)
{
    if (*front == 0 && *rear == -1)
    {
        queue[++(*rear)] = item;
        return;
    }
    if (*front != 0)
    {
        queue[--(*front)] = item;
        return;
    }
    printf ("Front insertion not possible \n");
}
// Delete from rear, Special case - Dequeue.
void qdelete_rear (int queue[], int *front,
                   int *rear)
{
    if (qempty (*front, *rear)
    {
        printf ("Queue underflow");
        return;
    }
    printf ("The element deleted is %d \n",
                queue[(*rear)--]);
    if (*front > *rear)
    {
        *front = 0; *rear = -1;
    }
}
```

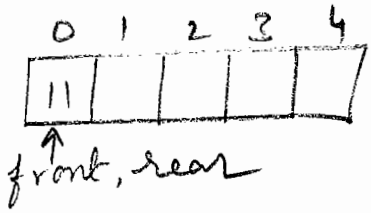
Circular Queue.

→ In an ordinary queue, as an item is inserted the rear end reaches the q_size-1, now the queue is full.

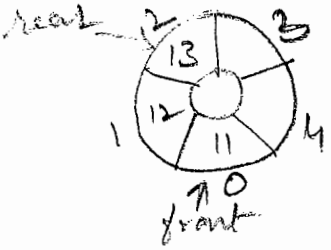
→ If some of the items are deleted & an attempt is made to insert items in the queue is not possible because rear has reached q_size-1.

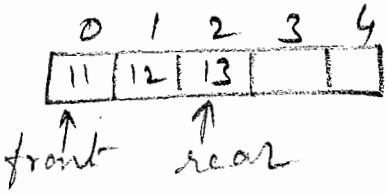
→ This disadvantage is overcome using circular queue.



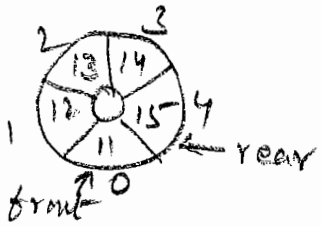


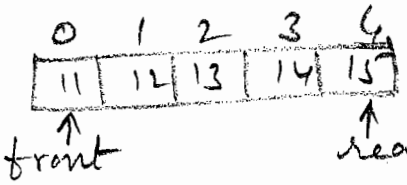
add 11



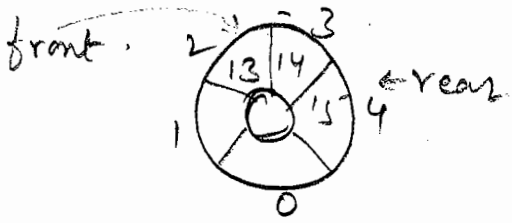


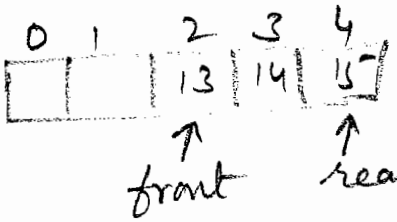
add 12, 13



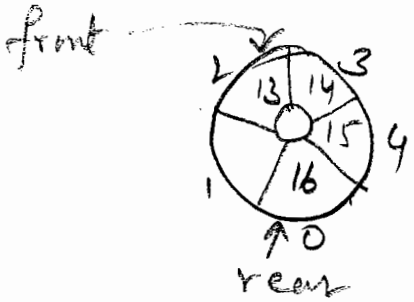


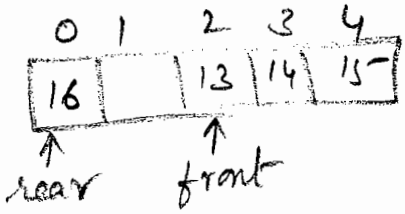
add 14, 15





delete 11, 12





add 16

C program to implement circular queue.

```
#include <stdio.h>
#include <stdlib.h>
#define qsize 5
int qfull(int count)
{
    return (count == qsize) ? 1 : 0;
}
int qempty(int count)
{
    return (count==0) ? 1 : 0;
}
void cq_insert(int item, int queue[],
               int *rear, int *count)
{
    if (qfull(*count))
    {
        printf("Overflow of queue \n");
        return;
    }
    *rear = (*rear+1) % qsize;
    queue[*rear] = item;
    *count += 1;
}
void cq_delete(int queue[], int *front, int *count)
{
    if (qempty(*count))
    {
        printf("Queue empty \n");
        return;
    }
    printf("The deleted element is %d \n",
                                queue[*front]);
    *front = (*front +1) % qsize;
    *count -= 1;
}
```

```
void display(int queue[], int front, int count)
{
    int i, j;
    if (qempty(count))
    {
        printf("Queue Empty");
        return;
    }
    printf("Contents of queue is \n");
    i=front;
    for (j=1; j<=count; j++)
    {
        printf("%d", queue[i]);
        i = (i+1) % qsize;
    }
    printf("\n");
}
void main()
{
    int choice, item, front, rear, count, queue[20];
    front=0; rear=-1; count=0;
    for(;;)
    {
        printf(" 1: Insert  2: Delete \n");
        printf("3: Display  4: Exit \n");
        printf("Enter the choice \n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("Enter the item to be
                    inserted \n");
                    scanf("%d", &item);
                    cq-insert(item, queue, &rear, &
                    count);
                    break;
            case 2: cq-delete(queue, &front, &count);
                    break;
            case 3: display(queue, front, count);
                    break;
            default: exit(0);
        }
    }
}
```

Priority Queues

Def: The priority queue is a special type of data structure in which items can be inserted or deleted based on the priority.

If the elements in the queue are of same priority, then the element, which is inserted first into the queue, is processed.

Case study: Priority queues are used in job scheduling algorithms in the design of operating system where jobs with highest priority has to be processed first.

Priority queues may be classified into two:

1. Ascending priority queue: In an ascending priority queue elements can be inserted in any order, but only the smallest element is removed first while deleting.
2. Descending priority queue: In descending priority queue the elements are inserted in any order, but only the largest element is removed first while deleting.

C program to implement priority queues

```
#include <stdio.h>
#include <stdlib.h>
#define Queue_SIZE 5
int q_full (int rear)
{
    return (rear == Queue_SIZE-1) ? 1 : 0;
}
int q_empty (int front, int rear)
{
    return (front > rear) ? 1 : 0;
}
void delete_front (int queue[], int *front,
                   int *rear)
{
    if (q_empty (*front, *rear))
    {
        printf ("Queue underflow");
        return;
    }
    printf ("The element deleted is %d \n",
            queue[(*front)++]);
    if (*front > *rear)
    {
        *front=0; *rear = -1;
    }
}
void display (int queue [], int front, int rear)
{
    int i;
    if (q_empty (front, rear))
    {
        printf ("Queue is empty");
        return;
    }
    printf ("Contents of queues is \n");
    for (i = front ; i <= rear ; i++)
        printf ("%d \n", queue [i]);
}
```

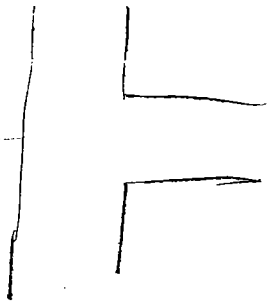
```
void insert_item(int item, int queue[], int *rear
{
    int j;
    if (q-full(*rear))
    {
        printf("Q is full \n");
        return;
    }
    j = *rear;
    // finding the right position.
    while (j>=0 && item < queue[j])
    {
        queue[j+1] = queue[j];
        j--;
    }
    queue[j+1] = item;
    *rear = *rear+1;
}
void main()
{
    int choice, item, front, rear, queue[10];
    front=0; rear=-1;
    for(;;)
    {
        printf("1:Insert 2:Delete \n");
        printf("3: Display 4: Exit \n");
        printf("Enter the choice \n");
        scanf("%d", &choice);
        switch(choice)
        {
            case1 : printf("Enter item to be inser-
                    ted");
                    scanf("%d", &item);
                    insert_item(item, queue,
                                &rear);
                    break;
```

```
        case 2: delete_front (queue,
                &front, &rear);
                break;
        case 3: display (queue, front,
                rear);
                break;
        default: exit(0);
    }
  }
}
```

```
// Program to implement Queue using array
#include<stdio.h>
#include<stdlib.h>
#define max 5

struct queue
{
      int queue[max];
      int front;
      int rear;
}Q;

void q_insert(int);
void q_delete(void);
int q_full(void);
int q_empty(void);
void q_display(void);
int menu(void);

void main()
{
      int ch,n;
      Q.rear=-1;
      Q.front=-1;
      system("clear");
      do{
          ch=menu();
          switch(ch)
          {
              case 1: if(q_full())
                                  printf("\nCan not insert an Item
- Queue full");
                                else
                                {
                                  printf("\nEnter the item to be
inserted into the Queue ");
                                  scanf("%d",&n);
                                  q_insert(n);
                                }
                            break;
              case 2: if(q_empty())
                                  printf("\nCan not delete an Item
- Queue overflow");
                                else
                                  q_delete();
                                break;
              case 3: q_display();
                      break;
                 case 4: exit(0);
              default: printf("\nEnter a valid choice!!");
          }
      }while(1);
```

```
}

int menu(void)
{
      int ch;
      printf("\nStack");
      printf("\n1.Insert Item\n2.Delete Item\n3.Display
Queue\n4.Exit");
      printf("\nEnter your Choice:");
      scanf("%d",&ch);
      return ch;
}

int q_full()
{
      if(Q.rear >= max-1)
            return 1;
      else
            return 0;
}

int q_empty()
{
      if((Q.front == -1) ||  (Q.front > Q.rear))
            return 1;
      else
            return 0;
}


void q_insert(int item)
{
      if(Q.front == -1)
            Q.front++;
      Q.queue[++Q.rear] = item;
}

void q_delete(void)
{
      int item;
      item = Q.queue[Q.front++];
      printf("\nThe item deleted is %d", item);
}


void q_display(void)
{
      int i;
      printf("\nThe item in the Queue are \n");
      for(i=Q.front;i<=Q.rear;i++)
            printf("%d\n",Q.queue[i]);
}
```

```
// Program to implement Queue using circular array
#include<stdio.h>
#include<stdlib.h>
#define max 5

struct queue
{
      int queue[max];
      int front;
      int rear;
}Q;

void q_insert(int);
void q_delete(void);
int q_full(void);
int q_empty(void);
void q_display(void);
int menu(void);

void main()
{
      int ch,n;
      Q.rear=0;
      Q.front=-1;
      system("clear");
      do{
          ch=menu();
          switch(ch)
          {
              case 1: if(q_full())
                            printf("\nCan not insert an
Item - Queue full");
                      else
                      {
                            printf("\nEnter the item to be
inserted into the Queue ");
                            scanf("%d",&n);
                            q_insert(n);
                      }
                    break;
              case 2: if(q_empty())
                            printf("\nCan not delete an
Item - Queue overflow");
                      else
                            q_delete();
                      break;
              case 3: q_display();
                      break;
                 case 4: exit(0);
              default: printf("\nEnter a valid choice!!");
          }
      }while(1);
}

int menu(void)
{
```

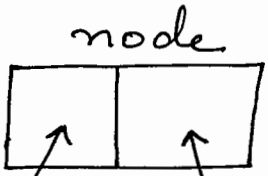


Linked lists. Module-3

Definition, Representation of linked lists in Memory
Memory allocation, Garbage collection.
Linked list operations: Traversing, searching, Insertion & Deletion.
Doubly linked list, Circular linked lists and header linked list.
Linked stacks & Queues.
Application of linked lists: polynomials, sparse matrix representation, programming Example.

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

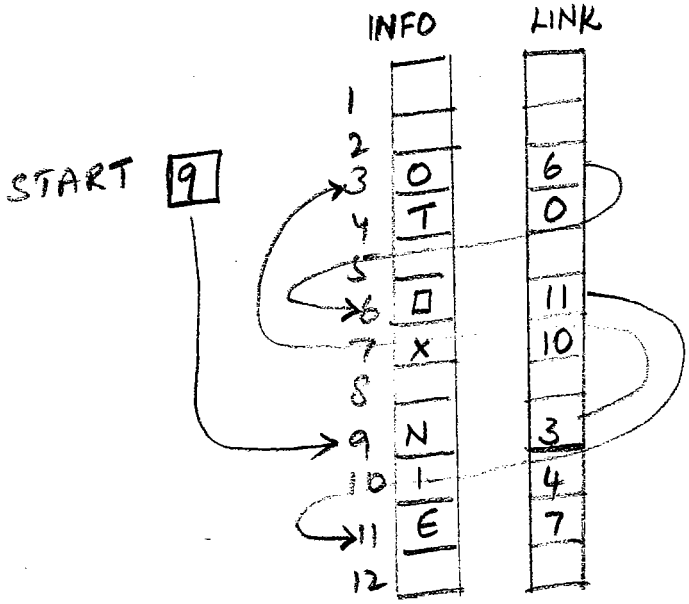
Each node is divide into two parts, the first part contains the information of the element & the second part, called the link field or next pointer field, contains the address of the next node in the list.



information link/next pointer

Representation of LL in memory.

The figure in the next page pictures a linked list in memory where each node of the list contains a single character.

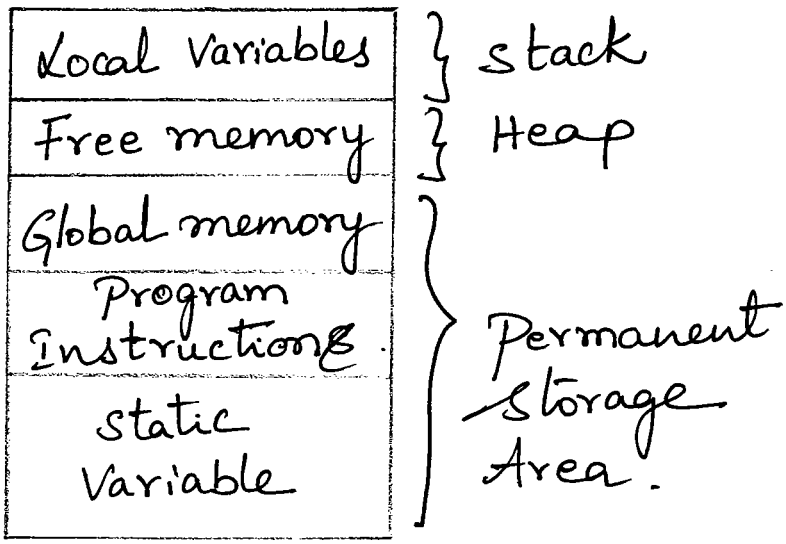


START = 9, so INFO[9] = N is the first character.
LINK[9] = 3 so INFO[3] = O is the second character.
So on, it continues until link [LINK] = 0
which stored the character string NO EXIT.

Memory Allocation.

→ Global variables, static variables & program instructions get their memory in permanent storage area.

→ Local variables are stored in stack.



→ The memory between these two regions are known as Heap area. The Dynamic memory allocation is done here.

→ The process of allocating memory at runtime is known as dynamic memory allocation.
→ Library routines known as "Memory management functions" are used for allocating & freeing memory during execution of a program.
→ Dynamic memory management functions are malloc(), calloc(), free(), realloc().
→ These functions are defined in stdlib.h

Garbage Collection

→ The Operating System of a Computer may Periodically collect all the deleted space on to the free storage list.
→ Any technique which does this collection is called garbage collection. (GC)
→ GC takes place in two steps
① The system runs through all lists, tagging those cells which are currently in use, & then the Computer runs through the memory collecting all untagged space onto the free-storage list.
② The GC may take place when there is only some minimum amount of space or no space at all left in the free-storage list or when the CPU is idle & has time to do the collection.

Representing chains in C

The program to represent linked list has to have the following capabilities

(1) A mechanism for defining a node's structure, ie the fields it contains.
self-referential structures may be used.

(2) A way to create new nodes when needed.
malloc() to be used.

(3) A way to remove nodes that we no longer need. free() is used.

Necessary declaration

```
struct node
{
    int info;          // Item in the node
    struct node *link; // pointer to the next node
};
typedef struct node *NODE;

NODE first = NULL;
```

} defining a node

→ Linked list with no nodes.

Implementation of single linked list.

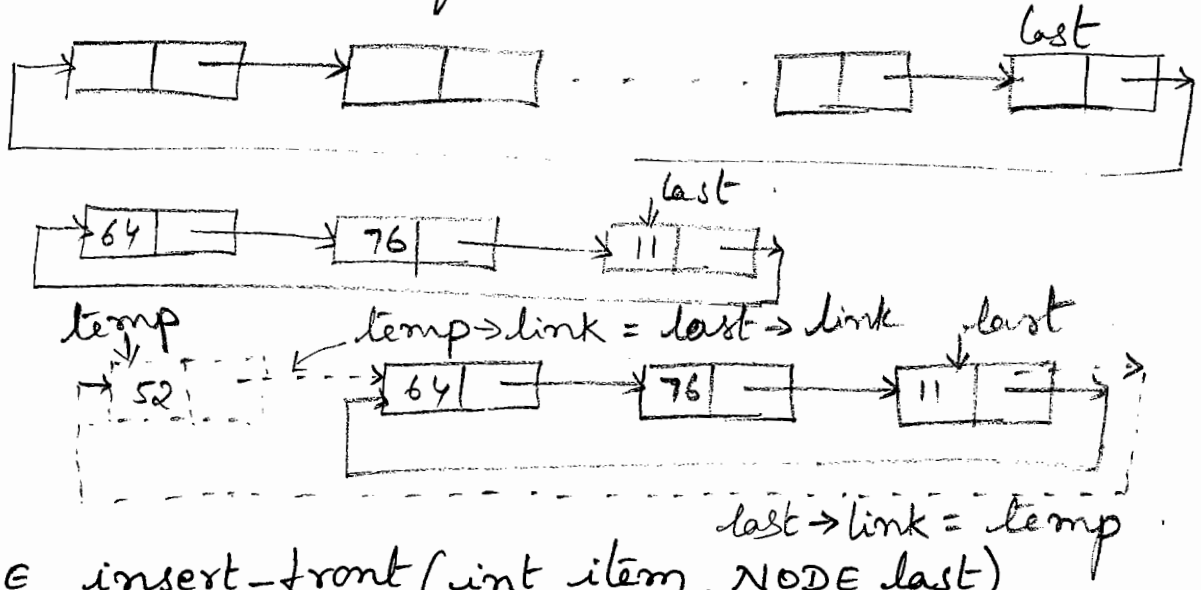
```
#include <stdio.h>
#include <alloc.h>

struct node
{
    int info;
    struct node *link;
};
typedef struct node *NODE;
// function to create a node
NODE getnode()
{
    NODE temp;
    temp = (NODE) malloc (sizeof (struct node));
    if (temp == NULL)
    {
        printf ("out of Memory");
        exit(0);
    }
    return temp;
}
// function to insert an item at the front of
   the list.
NODE insert_front (int item, NODE first)
{
    NODE temp;
    temp = getnode();
    temp->info = item;
    // Attach new node to the first node
    temp->link = first;
    return temp;
}
```

```
// function to display the nodes in the list
void display (NODE first)
{
    NODE temp;
    if ( first == NULL)
    {
        printf("List of is empty \n");
        return;
    }
    print("The contents of singly linked list \n");
    temp = first;
    while (temp != NULL)
    {
        printf (" %d", temp->info);
        temp = temp->link;
    }
    printf ("\n");
}
void main()
{
    NODE first = NULL; //start with a Empty list
    int choice, item;
    for(;;)
    {
        printf ("1: Insert front   2: Display \n");
        printf ("3: Quit \n");
        printf ("Enter the choice \n");
        scanf ("%d", &choice);
        switch(choice)
        {
            case 1 : printf("Enter the item to
                             be inserted \n");
                     scanf ("%d", &item);
                     first = insert_front(item,first);
                     break;
```

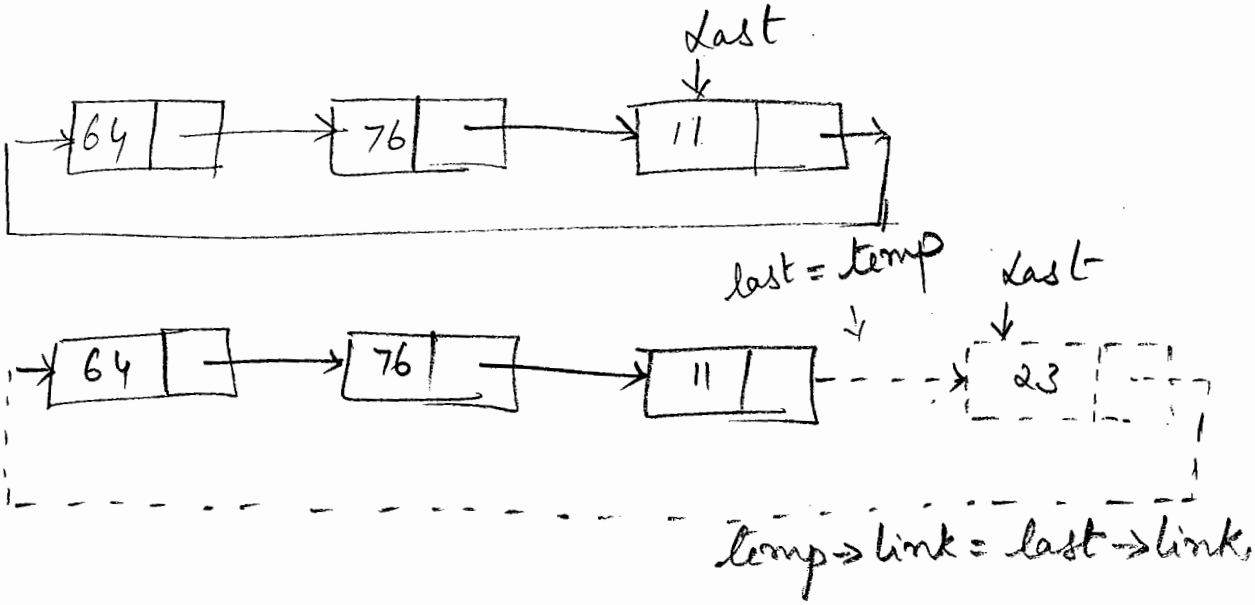
```
            case 2: display(first);
                    break;
            default: exit(0);
        }
    }
}
```

Circular singly linked list.

Insert a node at the front end.



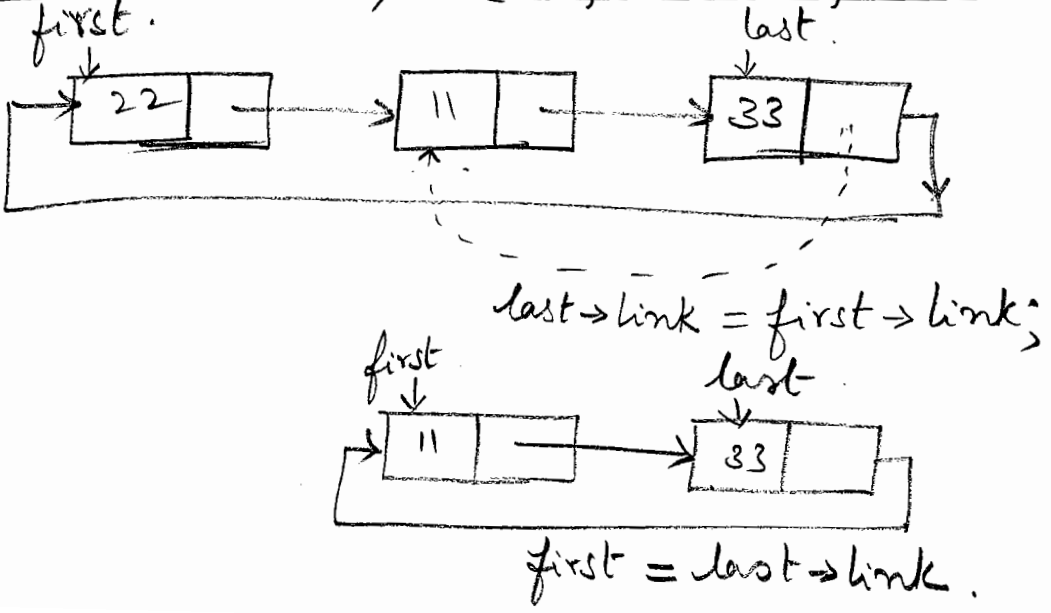

```
NODE insert_front(int item, NODE last)
{
    NODE temp;
    temp = getnode();
    temp->info = item;
    if(last==NULL)
        last = temp;
    else
        temp->link = last->link;
    last->link = temp;
    return last;
}
```



function to insert an item at the rear end of the circular SLL.

```
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;
}
```

Function to delete an item from the front end.



```
NODE delete_front (NODE last)
{
    NODE temp, first;
    if (last == NULL)
    {
        printf("List is empty \n");
        return NULL;
    }
    if (last->link == last)
    {
        print ("The item deleted is %d \n", last->info);
        freenode (last);
        return NULL;
    }
    first = last->link;
    last->link = first->link;
    printf("The item deleted is %d \n", first->info);
    freenode (first);
    return(last);
}
```

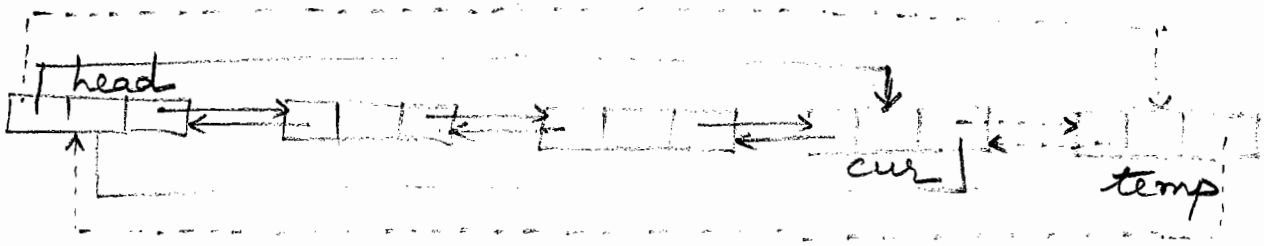
change this code for rear delete

```
// delete rear function.
// NODE prev;
    .
    .
    .
prev = last->link;
while (prev->link != last)
        prev = prev->link;
prev->link = last->link;
printf("The item deleted is %d \n", last->info);
freenode (last);
return prev;
```

Note: Replace last with first in this program

Linked list with head node.

```
NODE insert_front (int item, NODE head)
{
    NODE temp, cur;
    temp = getnode();
    temp->info = item;
    cur = head->rlink;
    head->rlink = temp;
    temp->llink = head;
    temp->rlink = cur;
    cur->llink = temp;
    return head;
}
```



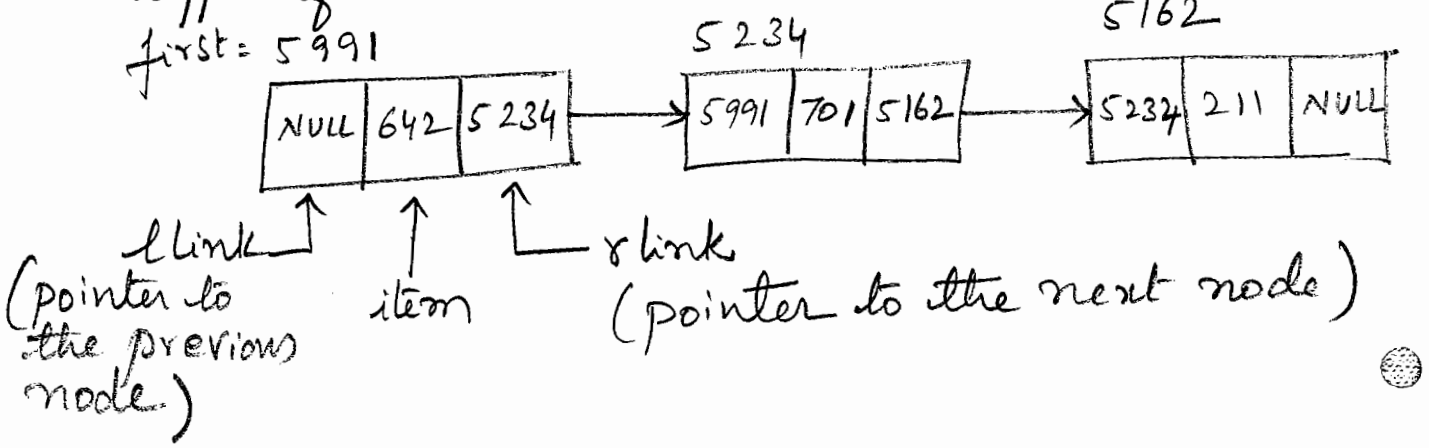

```
NODE insert_rear (int item, NODE head)
{
    NODE temp, cur;
    temp = getnode()
    temp->info = item;
    cur = head->llink;
    head->llink = temp;
    temp->rlink = head;
    temp->llink = cur;
    cur->rlink = temp;
    return head;
}
```

Doubly linked lists.

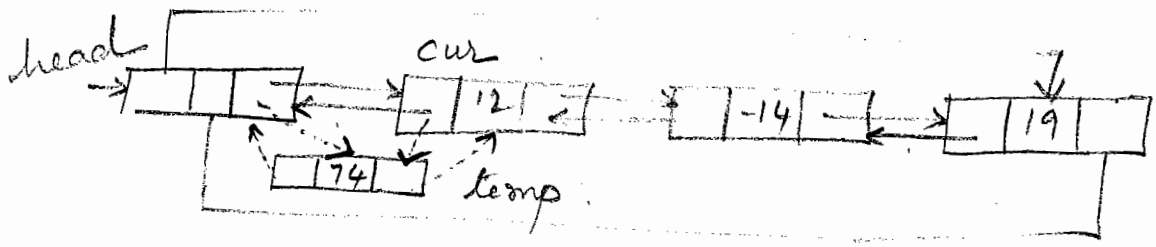
Doubly linked list is a variation of linked list in which navigation is possible in both ways either forward & backward easily as compared to single linked list.

Node declaration for doubly linked list.

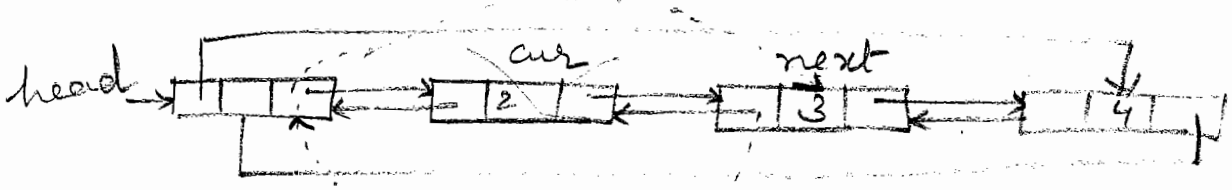
```
struct node
{
    int item; // Information to be stored in the node.
    struct node *llink; // left link pointer
    struct node *rlink; // right link pointer
};
typedef struct node *NODE;
```



Insert a node at the front end.

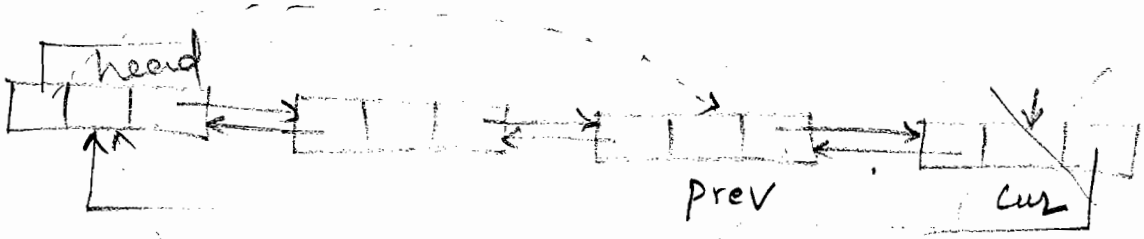


Delete a node from the front end.




```
NODE delete_front (NODE head)
{
    NODE cur, next;
    if (head->rlink == head)
    {
        printf ("Empty LL");
    }   return head;
    cur = head->rlink;
    next = cur->rlink;

    head->rlink = next;
    next->llink = head;
    printf ("The node to be deleted is %d\n", cur->info);
    freenode (cur);
    return head;
}
```

Delete a node from the rear end.



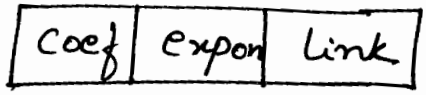
Polynomials

Representation

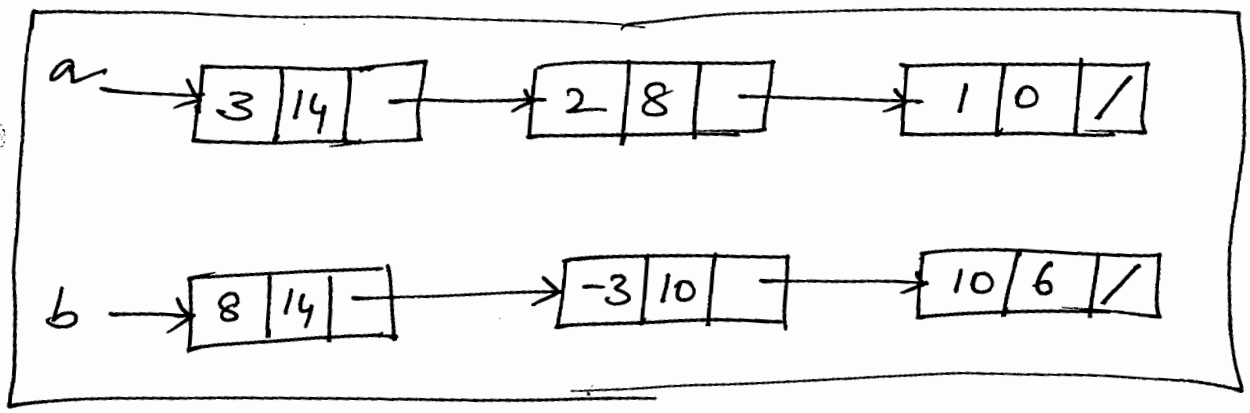
$$a = 3x^{14} + 2x^8 + 1$$
$$b = 8x^{14} - 3x^{10} + 10x^6$$

```
typedef struct polyNode * polyPointer;
typedef struct
{
    int coef;
    int expon;
    polyPointer link;
} polyNode;
polyPointer a, b;
```

→ PolyNode's



Representation using LL




```
void attach (float coefficient, int exponent,
             polyPointer *ptr)
{
    polyPointer temp;
    MALLOC (temp, sizeof (*temp));
    temp->coef = coefficient; temp->expon = exponent;
    (*ptr)->link = temp; *ptr = temp;
}
```

```
polyPointer padd (polyPointer a, polyPointer b)
{
    polyPointer c, rear, temp;
    int sum;
    MALLOC (rear, sizeof (*rear));
    c = rear;
    while (a && b)
      switch(COMPARE (a -> expon, b -> expon))
      {
        case -1: attach (b -> coeff, b -> expon, &rear);
                 b = b -> link;
                 break;
        case 0 : sum = a -> coeff + b -> coef;
                 if (sum)
                     attach (sum, a -> expon, &rear);
                 a = a -> link ; b = b -> link;
                 break;
        case 1 : attach(a -> coef , a -> expon, &rear);
                 a = a -> link;
      }
    for (; a; a = a -> link)
            attach (a -> coef, a -> expon, &rear);
    for (; b; b = b -> link)
            attach (b -> coef , b -> expon, &rear);
    rear -> link = NULL;
    temp = c; c = c -> link; free (temp);
    return c;
}.
```

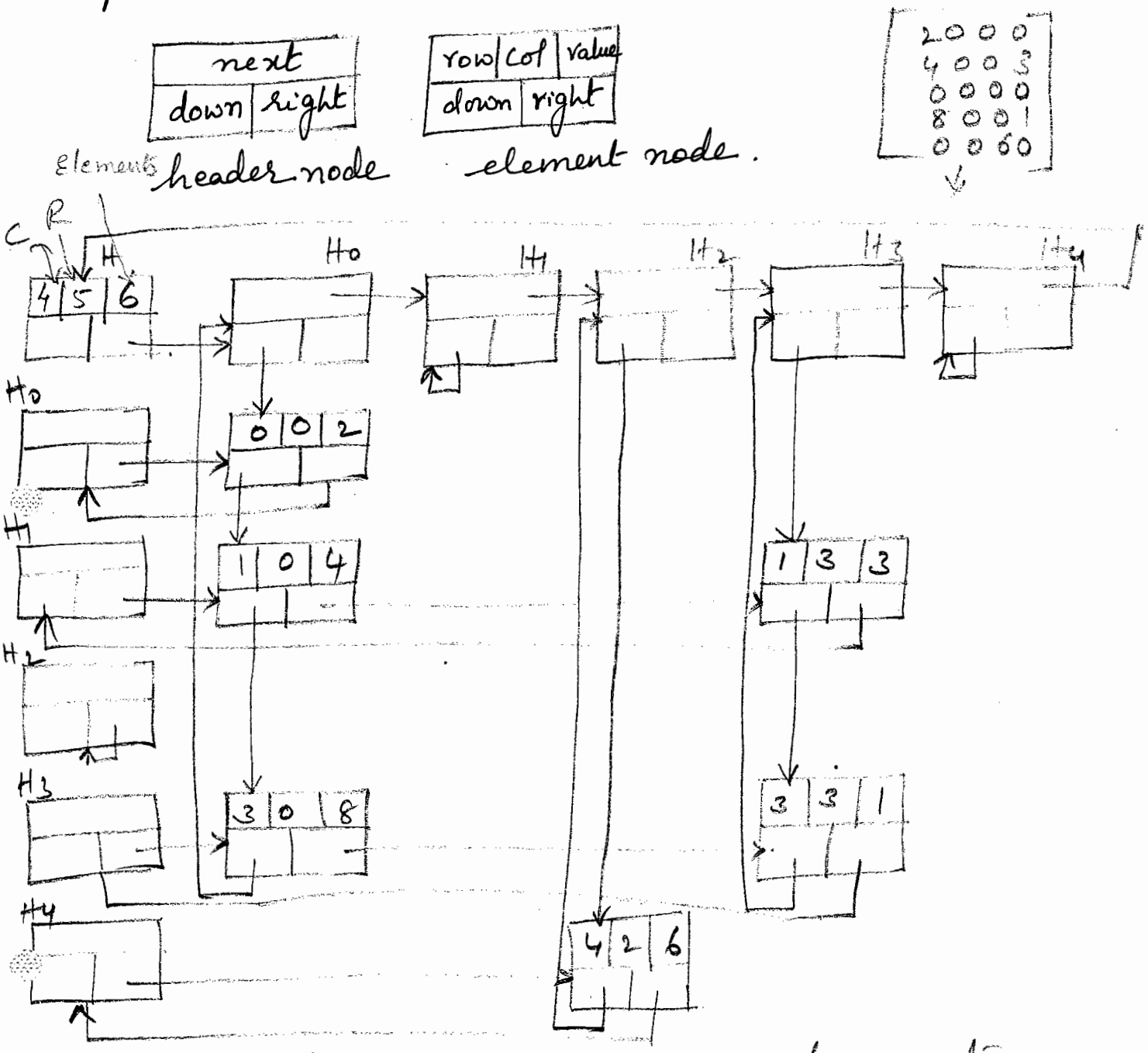
Function to add two polynomials

```
struct node
{
    int coef;
    int expon;
    struct node *link;
};
typedef struct node *poly; Poly a,b;
Poly polyadd (poly a, poly b)
{
    poly rear;
    int sum;
    while (a && b)
        switch (COMPARE (a->expon, b->expon))
        {
            case -1 : rear = attach (b->coef,
                          b->expon, rear);
                      b=b->link; break;
            case 0 : sum = a->coef + b->coef;
                     if (sum)
                       rear = attach (sum,
                           a->expon, rear);
                     a=a->link; b=b->link; break;
            case 1 : rear= attach (a->coef,
                          a->expon, rear);
                     a=a->link; break;
        }
    for (; a; a=a->link) rear= attach (a->coef, a->expon, rear);
    for (; b; b=b->link) rear= attach (b->coef, b->expon, rear);
    return (rear);
}
```

~~Void attach~~

```
poly attach(float coefficient, int exponent,
            poly ptr)
{
    poly temp;
    temp = getnode();
    temp->coef = coefficient;
    temp->expon = exponent;
    ptr->link = temp;
    ptr = temp;
    return(ptr);
}
```

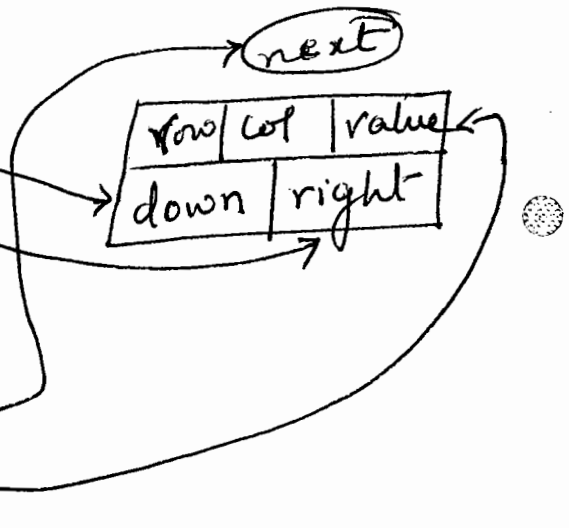
Sparse Matrix.



Linked Representation of the Sparse Matrix

```
#define MAX_SIZE 50
typedef enum {head, entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct
{
    int row;
    int col;
    int value;
} entryNode;
typedef struct
{
    matrixPointer down;
    matrixPointer right;
    tagfield tag;
    union
    {
        matrixPointer next;
        entryNode entry;
    } u;
} matrixNode;
matrixPointer hdnode[MAX_SIZE];
```

union: next / row, col, value.



function to read in a Sparse Matrix.

```
matrixPointer mread (void)
{
    int row, col, value, currentRow;
    int numRows, numCols, numTerms, numHeads, i;
    matrixPointer temp, last, node;
    printf("Enter the number of rows, columns
                & number of nonzero terms: ");
    scanf ("%d %d %d", &numRows, &numCols, &numTerms);
    numHeads = (numCols > numRows) ? numCols: numRows;
// set up header node for the list of header nodes */
    node = newNode(); node -> tag = entry;
    node -> u.entry.row = numRows;
    node -> u.entry.col = numCols;
    node -> u.value = numTerms
    if (!numHeads) node->right = node;
    else
    {
        for (i=0; i < numHeads; i++)
        {
            temp = newNode();
            hdnode[i]= temp; hdnode[i] -> tag = head;
            hdnode[i]->right = temp;
            hdnode[i] -> u.next = temp;
        }
        currentRow = 0;
        last = hdnode[0] // last node in current row.
        for (i=0; i < numTerms; i++)
        {
            printf ("Enter row, col & value ");
            scanf ("%d %d %d", &row, &col, &value);
            if (row > currentRow) // close current row
            {
                last -> right = hdnode [currentRow];
                currentRow = row, last = hdnode [row];
            }
```

```
        MALLOC(temp, sizeof(*temp));
        temp->tag = entry; temp->u.entry.row = row;
        temp->u.entry.col = col;
        temp->u.entry.value = value;
        last->right = temp;
        last = temp;
        // link into column list
        hdnode[col]->u.next->down = temp;
        hdnode[col]->u.next = temp;
    }
    // close last row
    last->right = hdnode[currentRow];
    // close all column lists
    for (i=0; i<numcols; i++)
        hdnode[i]->u.next->down = hdnode[i];
    // link all header nodes together
    for(i=0; i< numHeads-1 ; i++)
        hdnode[i]->u.next = hdnode[i+1];
    hdnode[numHeads-1]->u.next = node;
    node->right = hdnode[0];
}
return node;
}
```

```
// write out a sparse Matrix
void mwrite (matrixPointer node)
{ * print out the matrix in row major form */
  int i;
  matrixPointer temp, head = node -> right;
  /* Matrix dimensions */
  Printf (" \n numRows = %d, numCols = %d \n",
          node -> u. entry. row, node -> u. entry. col);
  Printf ("The matrix by row, col & value \n");
  for (i=0; i< node -> u. entry. row; i++)
{
    /* print out the entries in each row */
    for (temp = head -> right ; temp != head;
         temp = temp -> right)
      printf (" %5d %5d %5d \n", temp -> u.
      entry. row, temp -> u. entry. col,
                temp -> u. entry. value);
  }
}
```

```
// Single LL
#include<stdio.h>
#include<stdlib.h>
struct node
{
      int info;
      struct node *link;
};

typedef struct node *NODE;

NODE getnode()
{
      NODE temp;
      temp = (NODE) malloc (sizeof (struct node));
      if (temp == NULL)
      {
            printf("Out of Memory ");
            exit(0);
      }
      return (temp);
}


NODE insert_front(int item, NODE first)
{
      NODE temp;
      temp = getnode();
      temp->info = item;
      temp->link = first;
      return temp;
}

NODE insert_rear(int item, NODE first)
{
      NODE temp,next,prev;
      temp = getnode();
      temp->info = item;
      temp->link = NULL;
      if(first==NULL)
            return temp;
      else
      {
            next = first;
            while(next->link != NULL)
            {
                  next = next->link;
            }
            next->link = temp;
      }
      return first;
}
```

```
NODE delete_rear(NODE first)
{
      NODE temp,prev,cur;

      if(first==NULL)
      {
            printf("\n There are no elements in the linked list \n");
            return NULL;
      }
      else
            if(first->link == NULL)
            {
                  printf("\n The element deleted is %d\n",first->info);
                  free(first);
                  return NULL;
            }
      cur = first;
      while(cur->link != NULL)
      {
            prev = cur;
            cur = cur->link;
      }
      printf("\n The element deleted is %d ",cur->info);
      prev->link = NULL;
      free(cur);
      return first;
}


NODE delete_front(NODE first)
{
      NODE temp;
      if(first==NULL)
      {
            printf("\n There are no elements in the linked list \n");
            return first;
      }
      temp = first;
      printf("\n The element deleted is %d\n",temp->info);
      first = first->link;
      free(temp);
      return first;
}

void display(NODE first)
{
      NODE temp=first;
      printf("\n The  elements in the linked list are \n");
      while(temp != NULL)
      {
            printf("%d\n",temp->info);
```

```
            temp = temp->link;
        }
        return;
}

void main()
{
        NODE first = NULL;
        int choice, item;
        for(;;)
        {
            printf("\nEnter 1. for Insert front \n");
            printf("Enter 2. for Insert rear \n");
            printf("Enter 3. for deleting from front  \n");
            printf("Enter 4. for deleting from rear  \n");
            printf("Enter 5. for Display \n");
            printf("Enter 6. to Quit ");
            scanf("%d",&choice);
            switch(choice)
            {
                case 1: printf("Enter the item to be inserted to the LL ");
                        scanf("%d",&item);
                        first = insert_front(item,first);
                        break;
                case 2: printf("Enter the item to be inserted to the LL ");
                        scanf("%d",&item);
                        first = insert_rear(item,first);
                        break;
                case 3: first = delete_front(first);
                        break;
                case 4: first = delete_rear(first);
                        break;
                case 5: display(first);
                        break;
                case 6: exit(0);
            }
        }
}
```

```
// Program to implement Ordered Linked List
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
        NODE temp;
        temp = (NODE) malloc (sizeof (struct node));
        if (temp == NULL)
        {
                printf("Out of Memory ");
                exit(0);
        }
        return (temp);
}

NODE insert_order(int item, NODE first)
{
        NODE temp,prev,cur;
        temp = getnode();
        temp->info = item;
        temp->link = NULL;

        if (first == NULL)
                return temp;
        if (item < first->info)
        {
                temp->link = first;
                return temp;
        }
        cur=first;prev=NULL;
        while(cur != NULL && item > cur->info)
        {
                prev = cur;
                cur = cur->link;
        }
        prev->link=temp;
        temp->link=cur;
        return first;

}
```

```
NODE delete(int item, NODE first)
{
        NODE temp,prev,cur;

        if(first==NULL)
        {
                printf("\n There are no elements in the linked list \n");
                return NULL;
        }
        else
                if(first->info == item )
                {
                        printf("\n The element deleted is %d\n",first->info);
                        free(first);
                        return NULL;
                }
        cur = first;
        while(cur != NULL && item != cur->info )
        {
                prev = cur;
                cur = cur->link;
        }
        if (cur == NULL)
                printf("\n The element is not found in the list ");
        else
        if (item == cur->info)
        {
                temp = cur;
                prev->link = cur->link;
                free(temp);
        }

        return first;
}

void display(NODE first)
{
        NODE temp=first;
        printf("\n The  elements in the linked list are \n");
        while(temp != NULL)
        {
                printf("%d\n",temp->info);
                temp = temp->link;
        }
        return;
}
```

```
void main()
{
        NODE first = NULL;
        int choice, item;
        for(;;)
        {
                printf("\nEnter 1. for Insertion in order \n");
                printf("Enter 2. for deleting in order  \n");
                printf("Enter 3. for Display \n");
                printf("Enter 4. to Quit ");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: printf("Enter the item to be inserted to the LL ");
                                scanf("%d",&item);
                                first = insert_order(item,first);
                                break;
                        case 2: printf("Enter the item to be deleted in the LL ");
                                scanf("%d",&item);
                                first = delete(item,first);
                                break;
                        case 3: display(first);
                                break;
                        case 4: exit(0);
                }
        }
}
```

Linked List: A linked list is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. It is a data structure consisting of a group of nodes which together represent a sequence.
Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list.

Advantages of Linked List:

1. Linked List is Dynamic data Structure .
2. Linked List can grow and shrink during run time.
3. Insertion and Deletion Operations are Easier
4. Efficient Memory Utilization ,i.e no need to pre-allocate memory
5. Faster Access time, can be expanded in constant time without memory overhead
6. Linear Data Structures such as Stack,Queue can be easily implemented using Linked list

Singly Linked Lists are a type of data structure. It is a type of list. In a singly linked list each node in the list stores the contents of the node and a pointer or reference to the next node in the list. It does not store any pointer or reference to the previous node.

Applications of Singly linked list can be

A list of images that need to be burned to a CD in a medical imaging application

A list of users of a website that need to be emailed some notification

A list of objects in a 3D game that need to be rendered to the screen

A **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

Following are **advantages/disadvantages** of doubly linked list over singly linked list. 1) A DLL can be traversed in both forward and backward direction. 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given. ... 1) Every node of DLL Require extra space for an previous pointer.

Applications of Doubly linked list can be
- A great way to represent a deck of cards in a game.
- The browser cache which allows you to hit the BACK button (a linked list of URLs)
- Applications that have a Most Recently Used (MRU) list (a linked list of file names)
- A stack, hash table, and binary tree can be implemented using a doubly linked list.
- Undo functionality in Photoshop or Word (a linked list of state).

Queue: Queue is a linear data structure, in which the first element is inserted from one end called REAR(also called tail), and the deletion of existing element takes place from the other end called as FRONT(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.

Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Double ended queue: double-ended queue (dequeue, often abbreviated to deque, pronounced deck) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).

Trees Module-4.

Terminology, Binary Trees, properties of Binary trees Array & linked list representation of Binary trees, Binary Tree Traversals - Inorder, postorder, pre-order Additional Binary tree operations, Threaded binary Trees, Binary Search trees - Definition Insertion, Delelion, Traversal, searching, App.. of Trees - Evaluation of Expression, programming Examples.

1) program tree Traversal.
2) Threaded binary Trees.
3) program BST.
4) Generating tree using inorder / postorder

Module - IV

Tree: A tree is a finite set of one or more nodes such that

(1) There is a specially designated node called the root.

(2) The remaining nodes are partitioned into n≥0 disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are called the subtrees of the root.

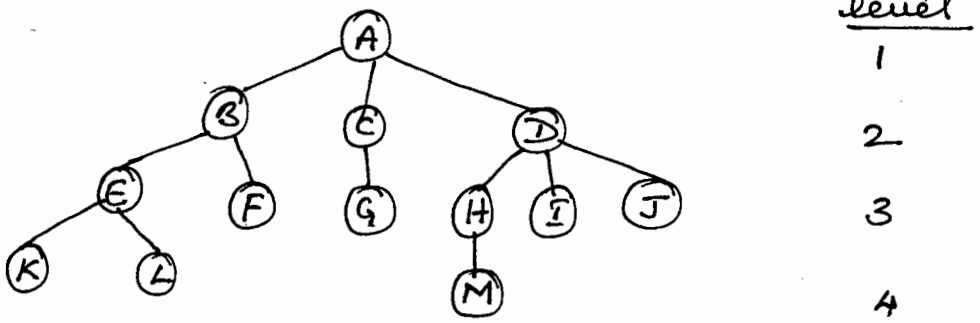
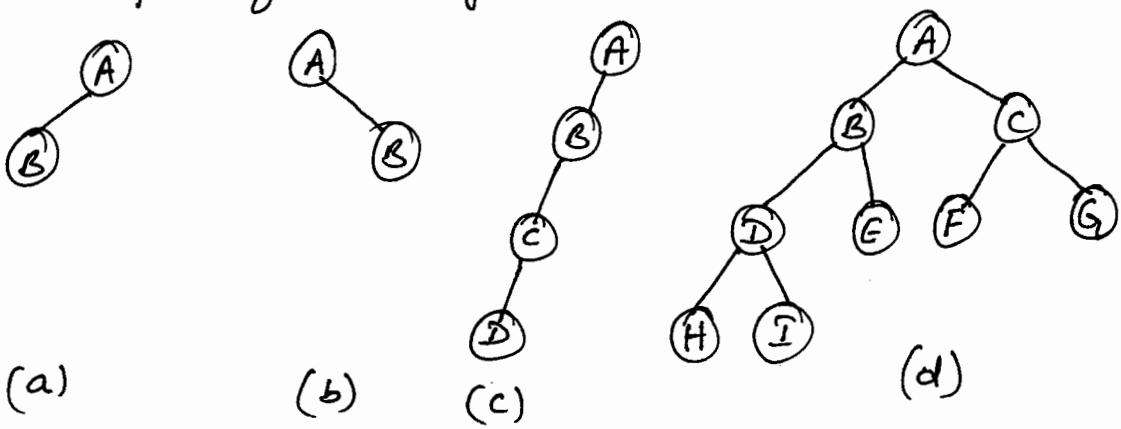


fig: A sample tree

Binary Tree: A binary tree is a finite set of nodes that is either empty or consists of a root & two disjoint binary trees called the left subtree & the right subtree.

Examples of binary trees



(a) (b) (c) (d)

Note:

* The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
* The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Full binary Tree: A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

complete binary Tree: A binary tree with n nodes & depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

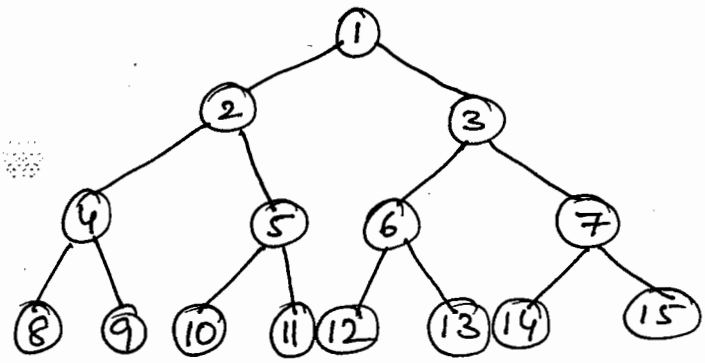


fig: full binary tree of depth 4 with sequential nodes numbers

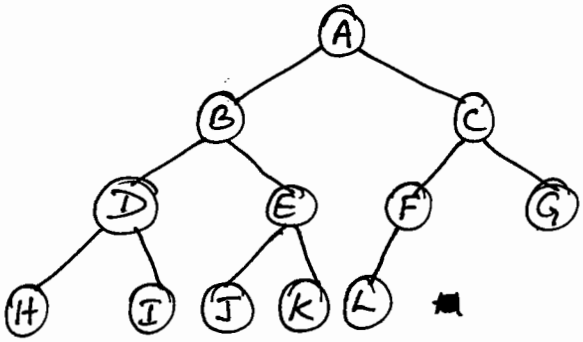
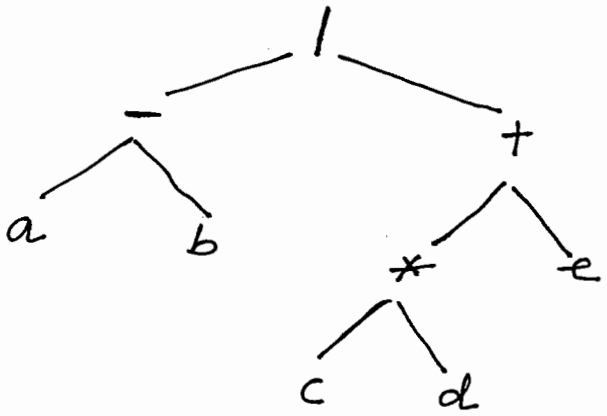
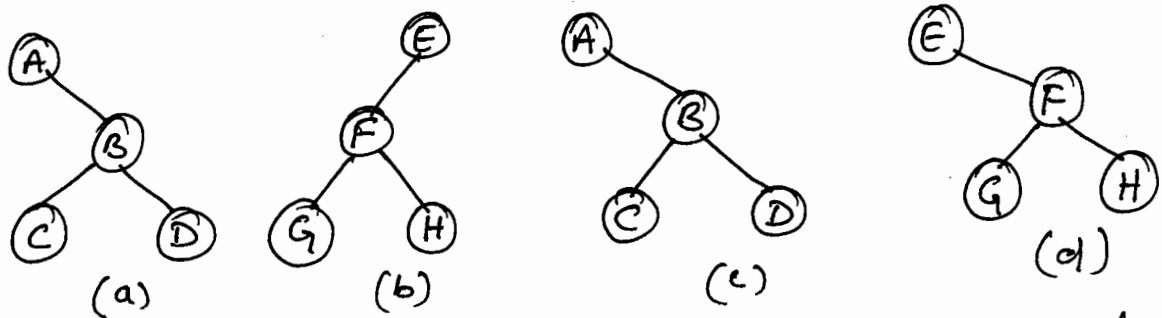


fig: A Complete binary tree in which every level, except possibly the last is completely filled & all nodes are as far left as possible.

→ Binary tree representing the Algebraic Expression $E = (a-b) / ((c*d) + e)$



→ Consider the four binary trees shown in the figures (a), (b), (c) & (d) below



→

→ The figures (a) & (c) are copies since they also have the same data at corresponding nodes

→ The tree (b) is either similar nor a copy of the tree (d) because in a binary tree we distinguish b/w a left successor & a right successor even when there is only one successor.

Binary Tree Representation.

Array — linked.

Array Representation.

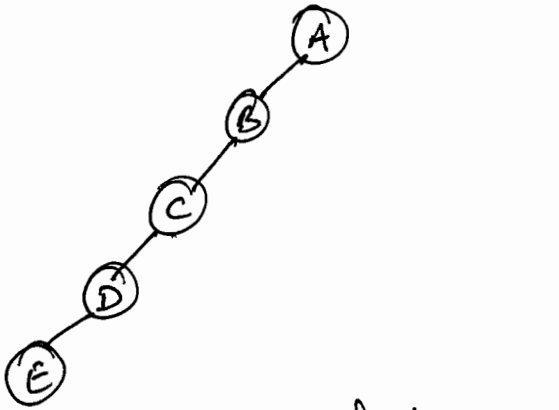


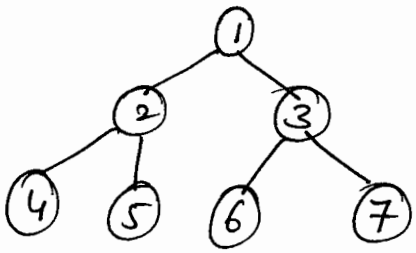
fig: Skewed binary tree.

tree

0	-
1	A
2	B
3	-
4	C
5	-
6	-
7	-
8	D
9	-
	:
16	E

→ In the array representation of tree, since the nodes are numbered from 1 to n, we can use one dimensional array to store the nodes.

→ Position 0 of this array is left empty & the node numbered i in the below figure is mapped to position i of the array.

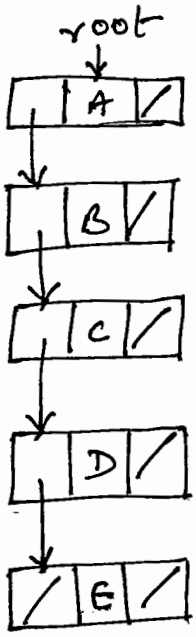


→ Although the array representation is good for Complete binary tree, it is wasteful for many other binary trees.

→ Insertion & deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in level number of these nodes.

→ These problems can be overcome easily through the use of a linked representation.

linked representation.



Note: / represents null in each node.

Example 2.

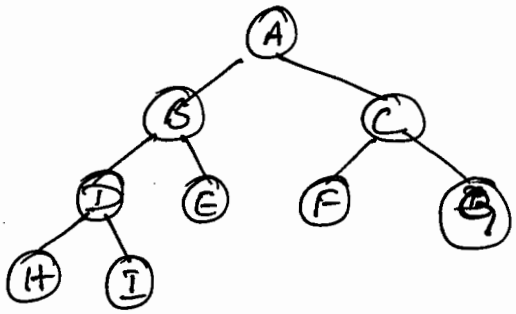
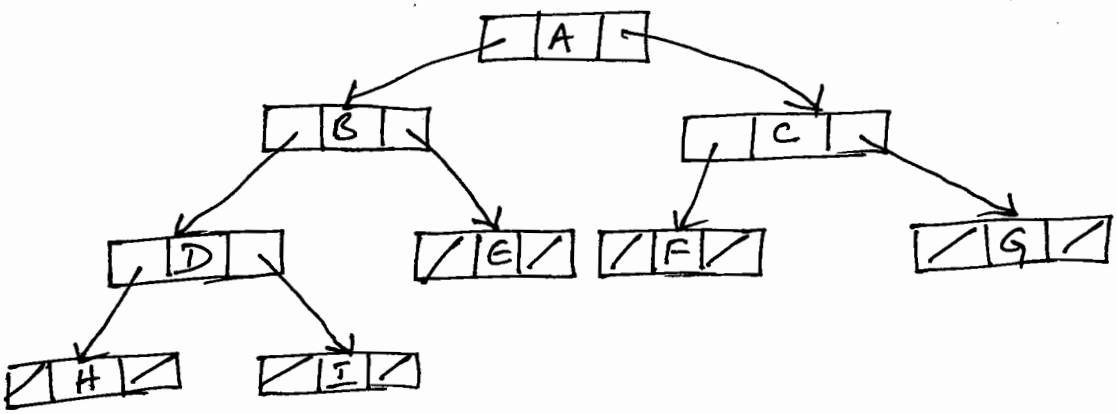


fig: Complete binary tree

array representation.

Index of the array	
0	-
1	A
2	B
3	C
4	D
5	G
6	F
7	G
8	H
9	I

linked representation



```
TNODE insert (int ele, TNODE root)
{
    TNODE new = getnode();
    TNODE prev, present;
    new->data = ele;
    new->rlink = new->llink = NULL;
    if (root == NULL)
        return new;
    if (ele < root->data)
        root->llink = insert(ele, root->llink);
    else
    if (ele > root->data)
        root->rlink = insert(ele, root->rlink);
    return root;
}
```

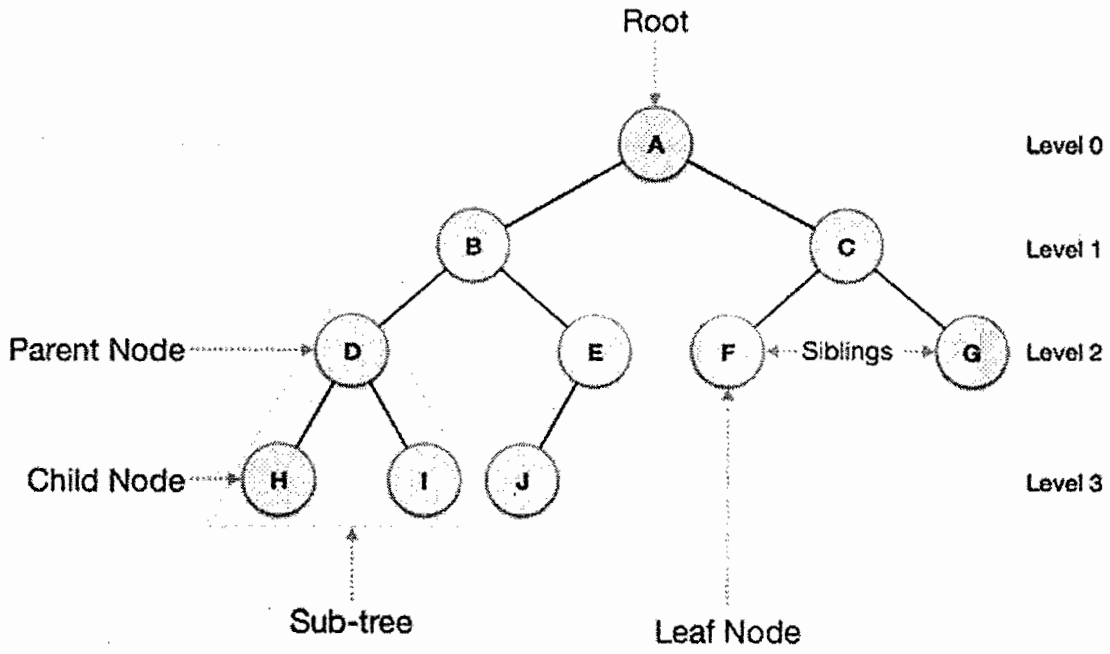
depth: The depth of a node N is given as the length of the path from Root R to the node N. The depth of the root node is zero.

V. Sampath Kumar 1967 © Govt. Copy

Important Terms

Following are the important terms with respect to tree.

- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.
- **Degree** - The number of subtrees of a node is called the degree of the node. In a binary tree, all nodes have degree 0, 1, or 2. A node of degree zero is called a terminal node or leaf node. A non-leaf node is often called a branch node. The degree of a tree is the maximum degree of a node in the tree.



Six Different Characterizations of a Tree

Trees have many possible characterizations, and each contributes to the structural understanding of graphs in a different way. Let T be a graph with n vertices. Then the following statements are equivalent.

(1) T is a tree.
(2) T contains no cycles and has $n - 1$ edges.
(3) T is connected and has $n - 1$ edges.
(4) T is connected, and every edge is a cut-edge.
(5) Any two vertices of T are connected by exactly one path.
(6) T contains no cycles, and for any new edge e, the graph $T + e$ has exactly one cycle.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

1) Trees reflect structural relationships in the data.
2) Trees are used to represent hierarchies.
3) Trees provide an efficient insertion and searching.
4) Trees are very flexible data, allowing to move subtrees around with minumum effort.

Binary Trees

A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

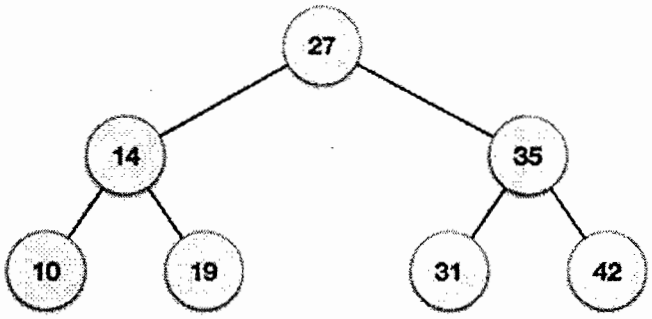
Properties of Binary Tree

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.
2) Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.
3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $\lceil \text{Log}_2(N+1) \rceil$.
4) A Binary Tree with L leaves has at least $\lceil \text{Log}_2 L \rceil + 1$ levels.
5) In Binary tree, number of leaf nodes is always one more than nodes with two children.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.



BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder Traversal** – Traverses a tree in a post-order manner.

Tree traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

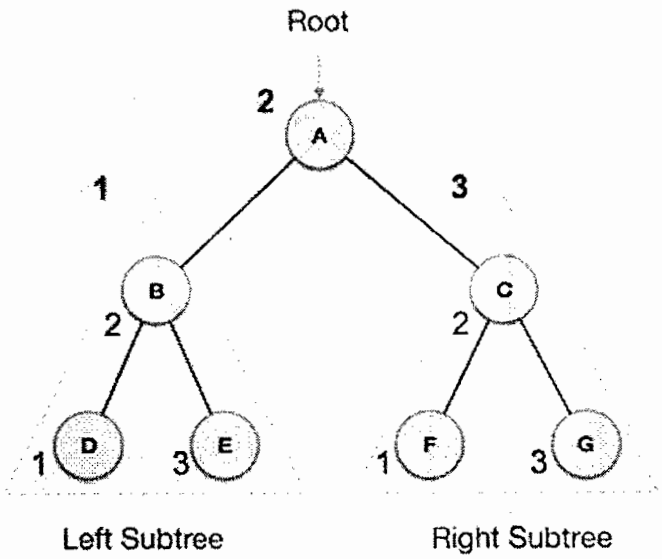
Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

```
// C-code: Inoder traversal
Void inorder (treePointer ptr)
{
      If (ptr) {
            inorder(ptr->leftChild);
            Printf("%d",ptr->data);
            inorder(ptr->rightChild);
      }
}
```

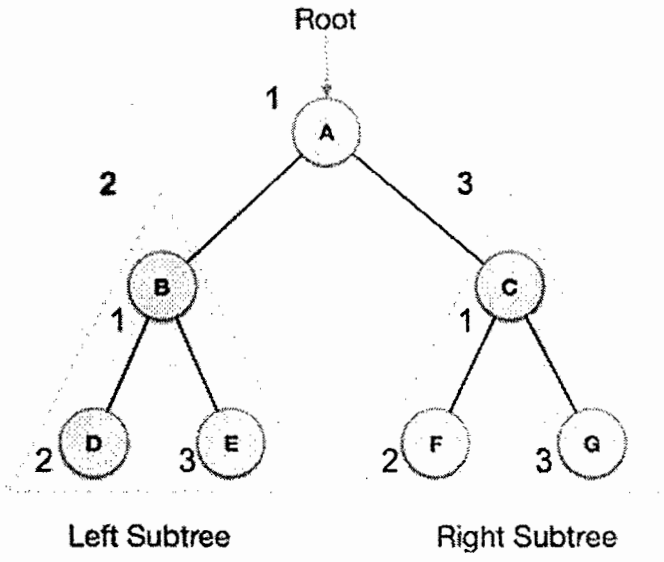


We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

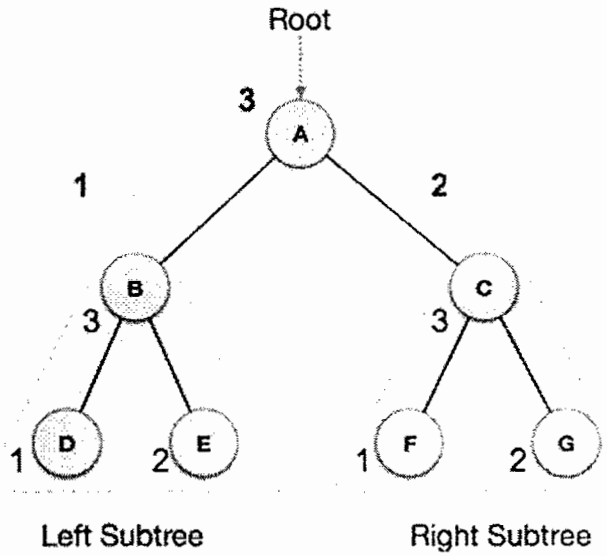


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

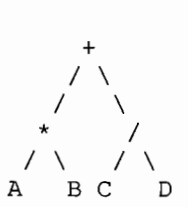


We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

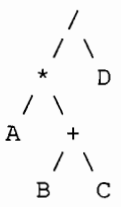
$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Evaluation of Expression using Trees

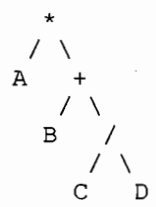
To generate an expression tree, given an expression, we first convert the expression to its postfix form. From the postfix expression, we read one symbol at a time from left to right. If the current symbol is an operand, then we push a tree of one node consisting of the operator onto a stack. If the symbol is an operator, then we pop two trees from the stack and create a tree with the root containing the operator and the result of the pop operations as the right and the left subtrees in that order. The resulting tree is again pushed onto the stack. When we have read all the symbols, the equivalent expression tree is the only element in the stack and a pop operation would give us the tree.



((A*B)+(C/D))



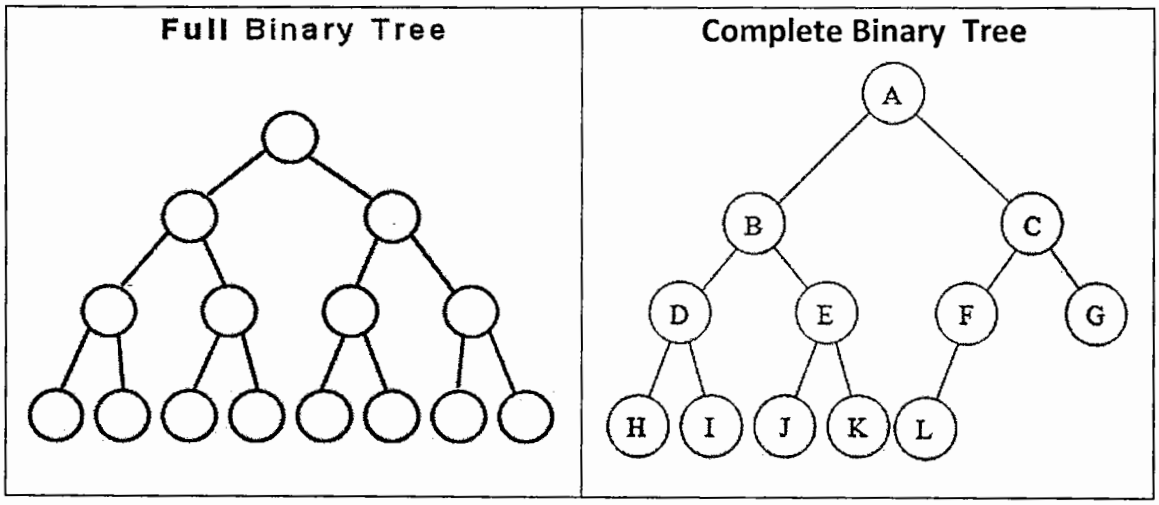
((A*(B+C))/D)



(A*(B+(C/D)))

Full v.s. Complete Binary Trees

A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children. A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Deleting in the Binary search tree

In a binary tree (BST) the node to be deleted will have two cases.

1) An empty left subtree & non-empty right subtree or an empty right subtree & non-empty left subtree. (A node having empty left child & empty right child is also deleted using this case).

2) Non empty left subtree & non empty right subtree.

Case 1:

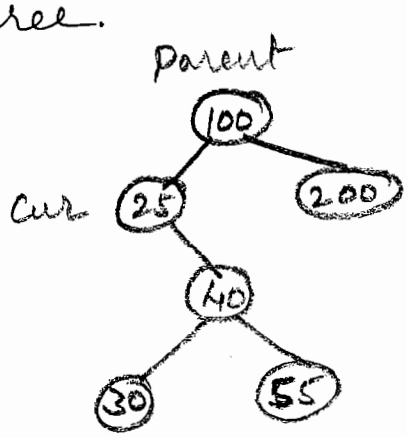


fig: a

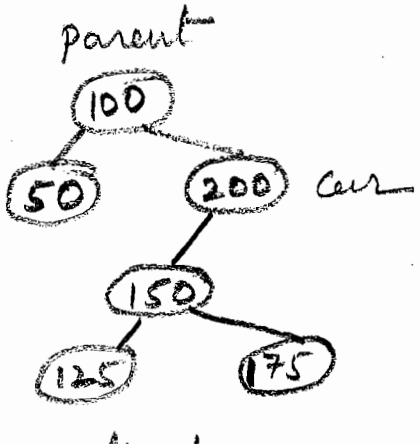


fig b

→ Consider the figure ⓐ & ⓑ where cur denotes the node to be deleted & in both cases one of the subtrees is empty & the other is non-empty

→ The node identified by parent is the parent of the node cur. The non empty subtree can be obtained & is saved in a variable q.

→ The corresponding code is:

```
if (cur -> llink == NULL)
        q = cur -> rlink;
else
 if (cur -> rlink == NULL)
        q = cur -> llink;
```

→ The non empty subtree identified by q should be attached to the parent of the node to be deleted & then delete the cur node.

Case:2.

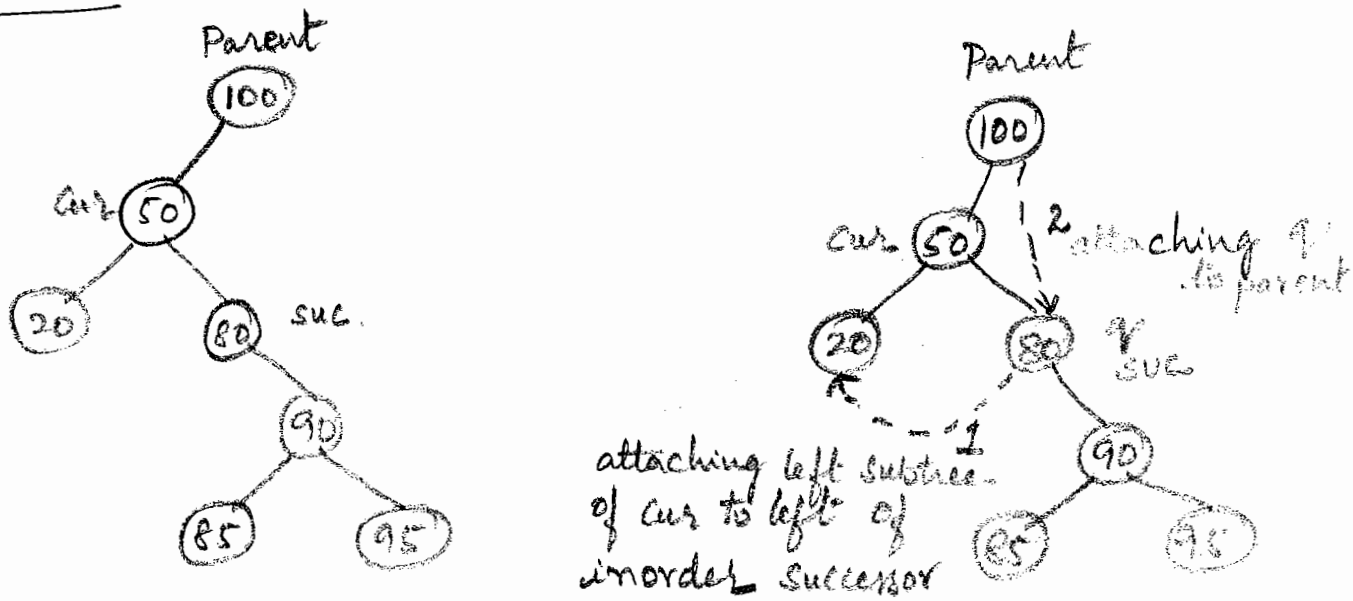


fig c: To delete a node from the tree

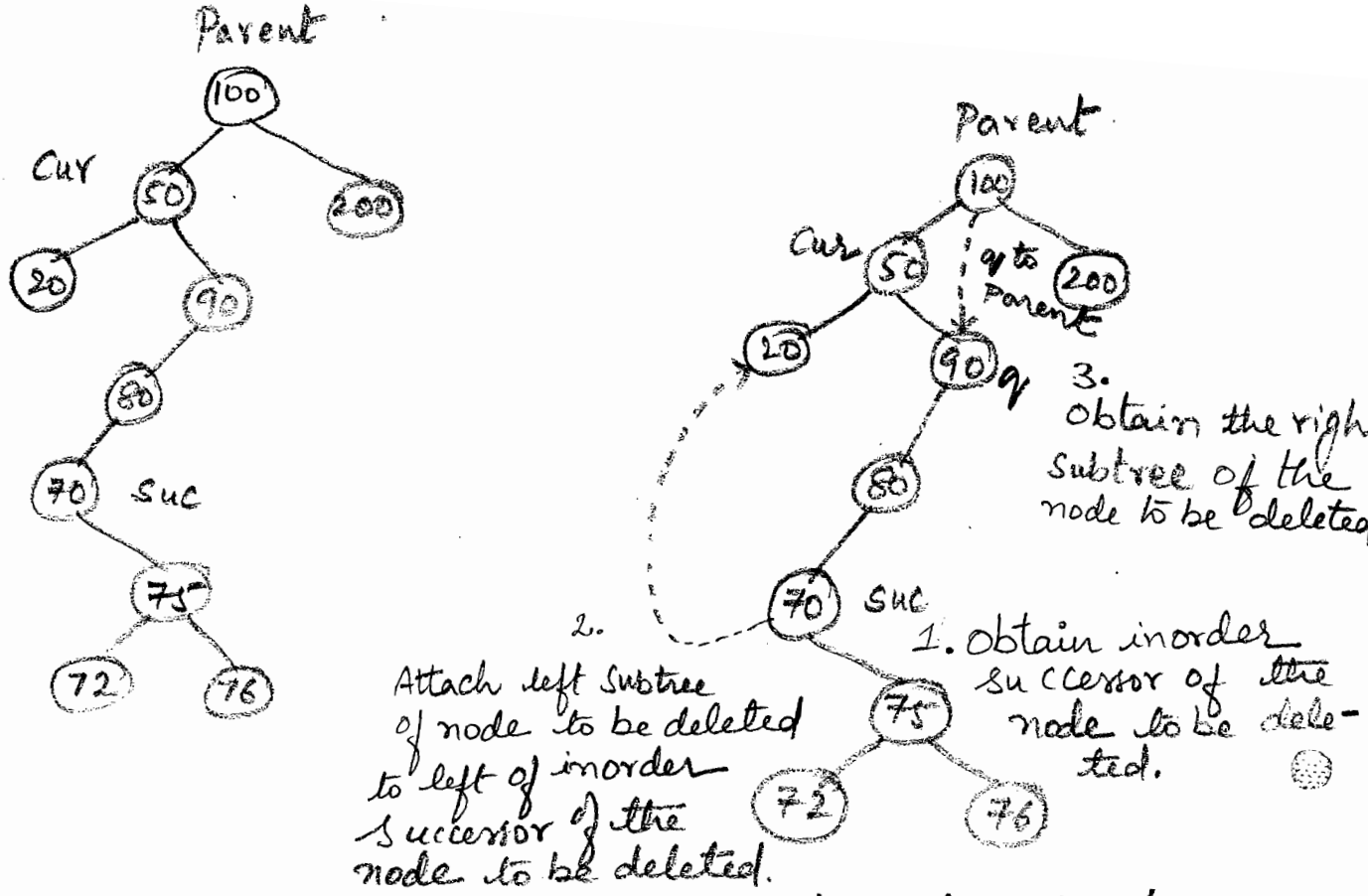


Fig d : To delete an item from the tree.

→ consider fig.c & fig.d. where cur denotes the node to be deleted.

→ In case(both) both the subtrees are non-empty. The node identified by parent is the parent of the node cur.

→ The node can be easily deleted using the following procedure in sequence:

① Find the inorder successor of the node to be deleted. The corresponding code is

```
suc = cur->rlink;
while (suc->llink != NULL)
    suc = suc->llink.
```

2. Attach left subtree of the node to be deleted to the left of successor of the node to be deleted. The corresponding code is:

```
suc->llink = cur->llink;
```

3. Obtain the right subtree of the node to be deleted. The corresponding code is:

```
q -> cur->rlink;
```

4. Attach the right subtree of the node to be deleted to the parent of the node to be deleted.

C-function to delete an item from the tree.

```
NODE delete_item (int item, NODE root)
{
    NODE cur, parent, suc, q;
    if (root == NULL)
    {
        printf("Tree is empty ! Item not found\n");
        return root;
    }
    /* obtain the position of the node to be
       deleted & its parent */
    Parent = NULL, cur = root;
    while (cur != NULL && item != cur->info)
    {
        parent = cur;
        cur = (item < cur->info) ? cur->llink :
                                   cur->rlink;
    }
    if (cur == NULL)
    {
        printf("Item not found");
        return root;
    }
```

```
/* Item found & delete it */
/* Case : 1 */
if (cur -> llink == NULL) // if left subtree is empty
        q = cur -> rlink;  /* obtain the addr of
                              nonempty right subtree */
else if (cur -> rlink == NULL)  // if right subtree is Empty
        q = cur -> llink;  /* obtain the address
                              of non empty left subtree*/
Else
{  // Case:2
   // obtain the inorder Successor
   suc = cur -> rlink; /* inorder successor lies
                          towards right */
   while (suc -> llink != NULL) /* & immediately
                                   keep traversing left*/
          suc = suc -> llink;
   suc -> llink = cur -> llink; /* Attach left of
                                   node to be deleted
                                   to left of successor
                                   of the node to be
                                   deleted */
   q = cur -> rlink; /* right subtree is
                        obtained*/
}
if (parent == NULL) return q; /* if parent
                                 does not exist */
/* Connecting parent of the node
   to be deleted to q */
if (cur == parent -> llink)
        parent -> llink = q;
else
        parent -> rlink = q;
free node (cur);
return root;
} // end of the function.
```

```
/*Design, Develop and Implement a menu driven Program in C for
the following operations
     on Binary Search Tree (BST) of Integers
   a. Create a BST of N Integers: 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5,
2
   b. Traverse the BST in Inorder, Preorder and Post Order
   c. Search the BST for a given element (KEY) and report the
ppropriate message
   e. Exit*/
     //RECURSIVE VERSION

     #include<stdio.h>
     #include<stdlib.h>
     struct tree
     {
          int data;
          struct tree *llink;
          struct tree *rlink;
     };

     typedef struct tree *TNode;

     // acquiring the memory for the node
     TNode getnode()
     {
          TNode temp=(TNode)malloc(sizeof(struct tree));
          if(temp==NULL)
          {
               printf("out of memory\n");
               return NULL;
          }
          return temp;
     }

      // recursive insert function
     TNode insert(TNode new, TNode root)
     {
          if(root==NULL)
            return new;
          if (new->data<root->data)
                  root->llink=insert(new,root->llink);
               // return(insert(ele,root->llink));
          else if (new->data>root->data)
                 root->rlink=insert(new,root->rlink);
           // else return(insert(ele,root->rlink));
          return root;
     }

     //recursive postorder function
     void inorder(TNode root)
     {
          if(root!=NULL)
          {
               inorder(root->llink);
               printf("%d\n",root->data);
               inorder(root->rlink);
          }
     }
```

```
//recursive preorder function
void preorder(TNode root)
{
      if(root!=NULL)
      {
         printf("%d\n",root->data);
         preorder(root->llink);
         preorder(root->rlink);
      }
}
// recursive inorder function
void postorder(TNode root)
{
      if(root!=NULL)
      {
            postorder(root->llink);
            postorder(root->rlink);
            printf("%d\n",root->data);
      }
}

 // recursive search function
int search(TNode root, int key)
{
      if(root!=NULL)
      {
            if  (root->data==key)
                  return key;
            if (key<root->data)
                  return search(root->llink, key);
            return search(root->rlink,key);
      }
      return -1;
}

void main()
{
      TNode new,root=NULL;
      int ch,ele,key,flag;
    for(;;)
      {
            printf(" 1 to Insert\n"
                        " 2 to Inorder\n"
                        " 3 to Preorder\n"
                        " 4 to Post order\n"
                        " 5 to Search\n"
                        " 6 to exit\n");
            scanf("%d",&ch);
            switch(ch)
            {
                  case 1 :printf("Enter the element to be
inserted\n");
                              scanf("%d",&ele);
                              new=getnode();
                              new->data=ele;
                              new->rlink=new->llink=NULL;
                              root=insert(new,root);
                              break;
```

```
                        case 2 :if(root==NULL)
                                        printf("TREE EMPTY\n");
                                else
                                {
                                        printf("The contents are
\n");
                                        inorder(root);
                                }
                                break;
                        case 3 :if(root==NULL)
                                        printf("TREE EMPTY\n");
                                else
                                {
                                        printf("The contents are
\n");
                                        preorder(root);
                                }
                                break;
                        case 4 :if(root==NULL)
                                        printf("TREE EMPTY\n");
                                else
                                {
                                        printf("The contents are
\n");
                                        postorder(root);
                                }
                                break;
                        case 5 :printf("Enter the node to be
searched\n");
                                scanf("%d",&key);
                                flag=search(root,key);
                                if(flag==-1)
                                        printf("UNSUCCESSFUL
SEARCH\n");
                                else
                                        printf("SUCCESSFUL SEARCH\n");
                                break;
                        case 6 : exit(0);
                }
        }
    }
```

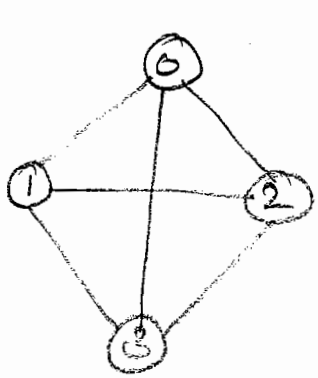
MODULE - V

Graphs: Definitions, Terminologies, Matrix & Adjacency list Representation of Graphs
Elementary Graph Operations,
Traversal methods: BFS & DFS.

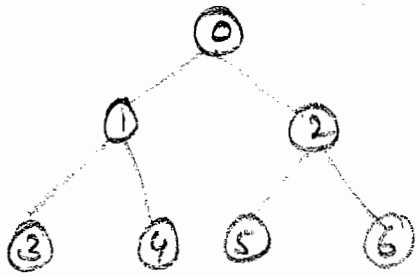
Def: A graph, consists of two sets, V & E.
V is a finite, non-empty set of vertices.
E is a set of pairs of vertices, these pairs are called Edges.

→ In an undirected graph the pair of vertices representing any edge is unordered. Thus the pair (u, v) & (v, u) represent the same edge.

→ In a directed graph each edge is represented by a directed pair $\langle u, v \rangle$, u is the tail & v the head of the edge.
Therefore, $\langle v, u \rangle$ & $\langle u, v \rangle$ represent two different edges.



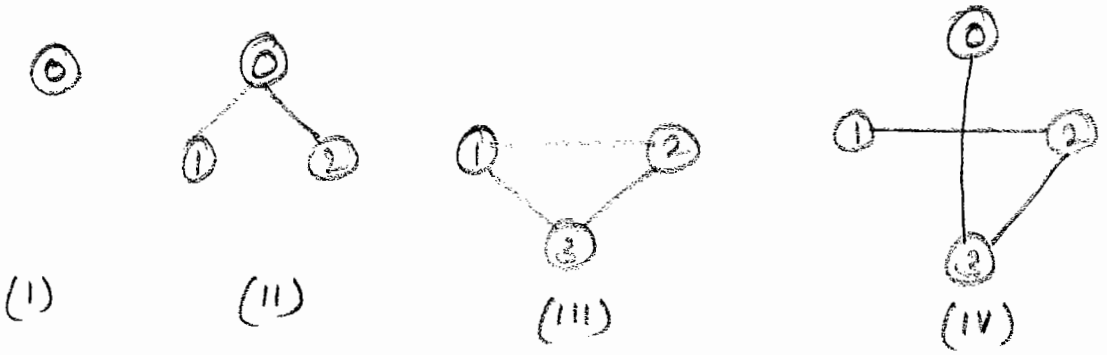
G_1



G_2

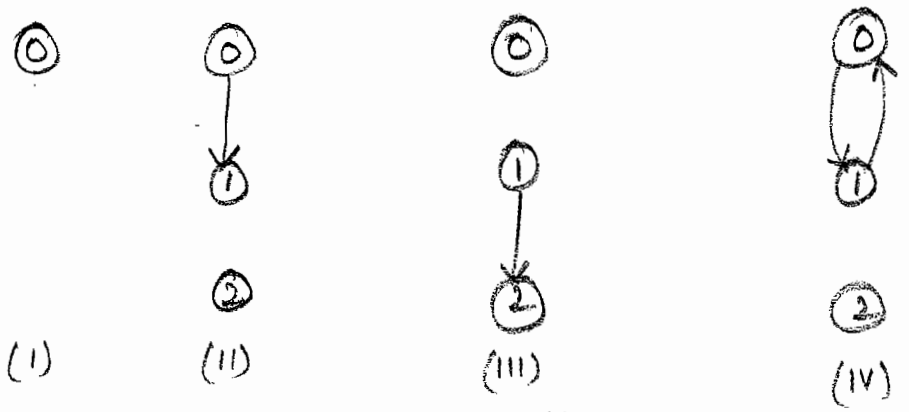
G_3

G_1, G_2 : Undirected graphs
G_3 : Directed graph



(I) (II) (III) (IV)

Some of the subgraphs of G_1



(I) (II) (III) (IV)

Some of the subgraph of G_3

A <u>subgraph</u> of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

A <u>path</u> 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 $E(G)$.

The <u>length</u> of a path is the number of edges on it.

A <u>cycle</u> is a simple path in which the first & last vertices are the same.

A <u>tree</u> is a connected acyclic (ie has no cycles) graph.

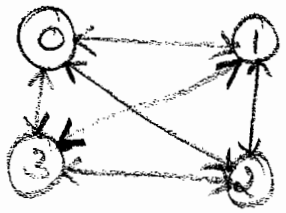
The <u>degree</u> of a vertex is the number of edges incident to that vertex.

Graph Representation

1) Adjacency Matrix.
2) Adjacency List.

Adjacency Matrix: Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two dimensional $n \times n$ array, say a, with the property that $a[i][j] = 1$ iff the edge (i,j) ($\langle i,j \rangle$ for a directed graph) is in $E(G)$.

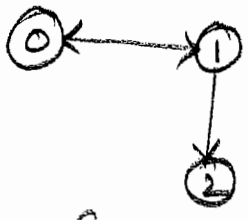
$a[i][j] = 0$ if there is no such edge in G.



G_1

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

Adjacency matrix.



G_2

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

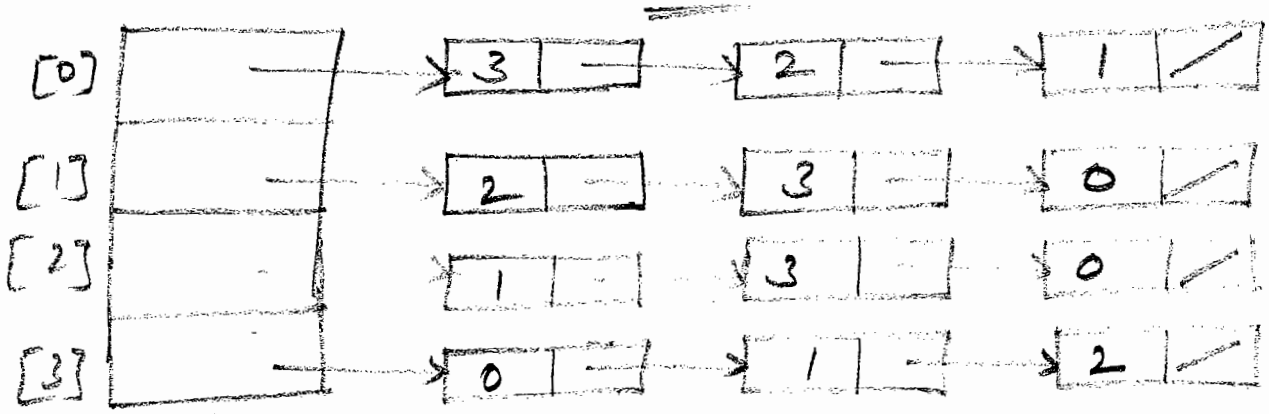
Adjacency matrix.

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 in G.

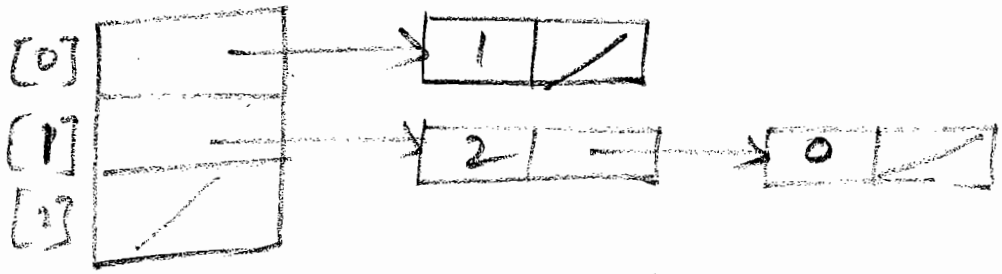
The nodes in chain i represent the vertices that are adjacent from vertex i. The data field of a chain node stores the index of an adjacent vertex.

The adjacency lists for graph G_1 & G_2 is shown below.

G_1

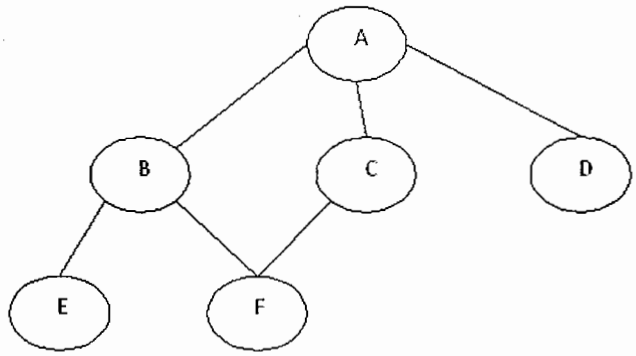


G_2



Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:



As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

Step 1: Push the root node in the Stack.

Step 2: Loop until stack is empty.

Step 3: Peek the node of the stack.

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

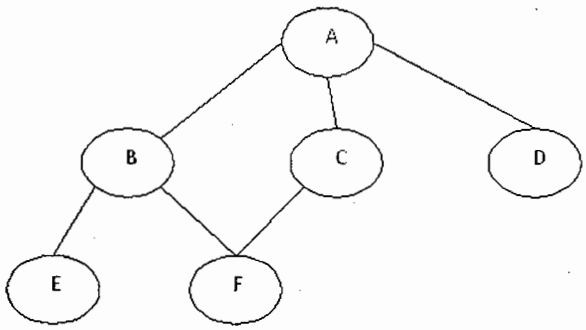
Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

```
/*  Design, Develop and Implement a Program in C for the
following operations on Graph(G) of Cities
          a. Create a Graph of N cities using Adjacency Matrix.
          b. Print all the nodes reachable from a given starting node
in a digraph using DFS/BFS method */

        #include<stdio.h>

    int a[20][20],reach[20],n;
    void dfs(int v)
          {
                int i;
                reach[v]=1;
                for (i=1;i<=n;i++)
                      if(a[v][i] && !reach[i])
                      {
                            printf("\n %d->%d",v,i);
                            dfs(i);
                }
    }
    void main()
    {
                int i,j,count=0;
                //system("clear");
                printf("\n Enter number of vertices:");
                scanf("%d",&n);
                for (i=1;i<=n;i++)
                {
                      reach[i]=0;
                      for (j=1;j<=n;j++)
                         a[i][j]=0;
                }
                printf("\n Enter the adjacency matrix:\n");
                for (i=1;i<=n;i++)
                  for (j=1;j<=n;j++)
                   scanf("%d",&a[i][j]);
                printf("\n The reachable nodes from the source are:
\n");
                dfs(1);
                printf("\n");
                for (i=1;i<=n;i++)
                {
                      if(reach[i])
                         count++;
                }
          /*    if(count==n)
                  printf("\n Graph is connected"); else
            printf("\n Graph is not connected"); */
    }
```

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:

If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.



Algorithmic Steps

Step 1: Push the root node in the Queue.

Step 2: Loop until the queue is empty.

Step 3: Remove the node from the Queue.

Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

```
// BFS
#include<stdio.h>
#include<conio.h>
int a[23][12],q[23],visited[23],n,front=0,rear=-1;
void BFS_search(int s)  /* s is starting vertex - Source */
{
      int d;          /*d means destination */
      for(d=1;d<=n;d++)/*checking is there a route from source ? */
      if(a[s][d] && visited[d]!=1)
      {
            rear=rear+1;
            q[rear]=d;
      }
      if(front<=rear)/*is there  possible paths entries in queue*/
      {                       /* = is must to say, at least one */
            visited[q[front]]=1; /* visted[node number]*/
            BFS_search(q[front++]);
      }
}
void main()
{
      int s,col,row;
      clrscr();
      printf("Enter the number of vertices \n");
      scanf("%d",&n);
      for(col=1;col<=n;col++)
      {
          q[col]=0;
          visited[col]=0;
      }
      printf("Enter the graph data in the matrix form\n\n");
      for(row=1;row<=n;row++)
            for(col=1;col<=n;col++)
                  scanf("%d",&a[row][col]);
      printf("Enter the starting vertex\n\n");
      scanf("%d",&s);
      BFS_search(s);
      printf("Node reachable are\n\n");
      for(col=1;col<=n;col++)
            if(visited[col])
                  printf("%d\t",col);
}
```

Sorting & Searching.

- → Insertion Sort
- → Radix Sort
- → Address Calculation sort.

Insert Sort.

Ex: Assume the n=5 & the input key sequence is 5, 4, 3, 2, 1. After each iteration we have.

	[1]	[2]	[3]	[4]	[5]
-	5	4	3	2	1
2	4	5	3	2	1
3	3	4	5	2	1
4	2	3	4	5	1
5	1	2	3	4	5

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for(i=1; i<n; i++)
    {
        key = arr[i];
        j = i-1;
        while(j>=0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

```
// C program for insertion sort
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++)
   {
         key = arr[i];
         j = i-1;

         /* Move elements of arr[0..i-1], that are
            greater than key, to one position ahead
            of their current position */
         while (j >= 0 && arr[j] > key)
         {
             arr[j+1] = arr[j];
             j = j-1;
         }
         arr[j+1] = key;
   }
}

// A utility function ot print an array of size n
void printArray(int arr[], int n)
{
   int i;
   for (i=0; i < n; i++)
         printf("%d ", arr[i]);
   printf("\n");
}


/* Driver program to test insertion sort */
int main()
{
      int arr[] = {5, 4, 3, 2, 1};
      int n = sizeof(arr)/sizeof(arr[0]);

      insertionSort(arr, n);
      printArray(arr, n);

      return 0;
}
```

```
/* Bubble sort code */

#include <stdio.h>

int main()
{
      int array[100], n, c, d, swap;

      printf("Enter number of elements\n");
      scanf("%d", &n);

      printf("Enter %d integers\n", n);

      for (c = 0; c < n; c++)
      scanf("%d", &array[c]);

      for (c = 0 ; c < ( n - 1 ); c++)
      {
      for (d = 0 ; d < n - c - 1; d++)
      {
         if (array[d] > array[d+1])                    {
                  swap       = array[d];
                  array[d]   = array[d+1];
                  array[d+1] = swap;
                  }
      }
      }

      printf("Sorted list in ascending order:\n");

      for ( c = 0 ; c < n ; c++ )
      printf("%d\n", array[c]);

      return 0;
}
```

Radix sort

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers. Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.[1]

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are least significant digit (LSD) radix sorts and most significant digit (MSD) radix sorts. LSD radix sorts process the integer representations starting from the least digit and move towards the most significant digit. MSD radix sorts work the other way around.

Following example shows how Radix sort operates on seven 3-digits number.

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

```
// C program for Radix sort
#include<stdio.h>

// Function to find largest element
int largest(int a[], int n)
{
      int large = a[0], i;
      for(i = 1; i < n; i++)
      {
          if(large < a[i])
              large = a[i];
      }
      return large;
}

// Function to perform sorting
void RadixSort(int a[], int n)
{
      int bucket[10][10], bucket_count[10];
      int i, j, k, remainder, NOP=0, divisor=1, large,
pass;

      large = largest(a, n);
      printf("The large element %d\n",large);
      while(large > 0)
      {
          NOP++;
          large/=10;
      }

      for(pass = 0; pass < NOP; pass++)
      {
          for(i = 0; i < 10; i++)
          {
              bucket_count[i] = 0;
          }
          for(i = 0; i < n; i++)
          {
              remainder = (a[i] / divisor) % 10;
              bucket[remainder][bucket_count[remainder]] =
a[i];
              bucket_count[remainder] += 1;
          }

          i = 0;
          for(k = 0; k < 10; k++)
          {
              for(j = 0; j < bucket_count[k]; j++)
              {
                  a[i] = bucket[k][j];
                  i++;
              }
          }
          divisor *= 10;

          for(i = 0; i < n; i++)
```

```
            printf("%d  ",a[i]);
        printf("\n");
    }
}

//program starts here
int main()
{
    int i, n, a[10];
    printf("Enter the number of elements :: ");
    scanf("%d",&n);
    printf("Enter the elements :: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d",&a[i]);
    }
    RadixSort(a,n);
    printf("The sorted elements are ::  ");
    for(i = 0; i < n; i++)
        printf("%d  ",a[i]);
    printf("\n");
    return 0;
}
```

Address Calculation Sort (Hashing)

- In this method a function f is applied to each key.
- The result of this function determines into which of the several subfiles the record is to be placed.
- The function should have the property that: if x <= y , f (x) <= f (y), Such a function is called order preserving.
- An item is placed into a subfile in correct sequence by placing sorting method – simple insertion is often used.

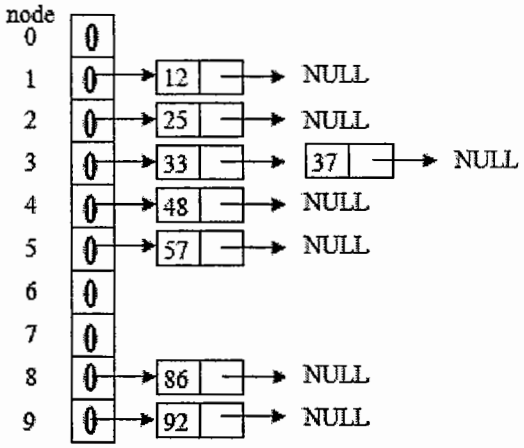
Example:

25 57 48 37 12 92 86 33

Let us create 10 subfiles. Initially each of these subfiles is empty. An array of pointer f(10) is declared, where f(i) refers to the first element in the file, whose first digit is i. The number is passed to hash function, which returns its last digit (ten's place digit), which is placed at that position only, in the array of pointers.

num= 25 – f(25) gives 2
57 – f(57) gives 5
48 – f(48) gives 4
37 – f(37) gives 3
12 – f(12) gives 1
92 – f(92) gives 9
86 – f(86) gives 8
33 – f(33) gives 3 which is repeated.

Thus it is inserted in 3rd subfile (4th) only, but must be checked with the existing elements for its proper position in this subfile.



```
// Address Calculation sort
#include<stdio.h>
#include<stdlib.h>
struct node
{
      int info;
      struct node *link;
};

typedef struct node *NODE;
NODE head[10];
NODE getnode()
{
      NODE temp;
      temp = (NODE) malloc (sizeof (struct node));
      if (temp == NULL)
      {
            printf("Out of Memory ");
            exit(0);
      }
      return (temp);
}

NODE insert_order(NODE first,int item)
{
      NODE temp,prev,cur;
      temp = getnode();
      temp->info = item;
      temp->link = NULL;

      if (first == NULL)
      {
            first = temp;
            return first;
      }
      if (item < first->info)
      {
            temp->link = first;
            return temp;
      }
      cur=first;prev=NULL;
      while(cur != NULL && item > cur->info)
      {
            prev = cur;
            cur = cur->link;
      }
      prev->link=temp;
      temp->link=cur;
      return first;

}
```

```
int fun(int n)
{
      return(n/10);
}

void AddrCalculation(int arr[],int size)
{
      int i,j=0,pos;
      for(i=0;i<size;i++)
      {
          pos = fun(arr[i]);
          head[pos] = insert_order(head[pos],arr[i]);
      }
      for(i=0;i<10;i++)
          while(head[i] != NULL)
          {
                arr[j++] = head[i]->info;
                head[i] = head[i]->link;
          }
      printf("\nThe sorted elements are:\n");
      for(i=0;i<size;i++)
          printf("%d\n",arr[i]);
}


void main()
{
      int a[10],i,n;
      printf("\n Enter the total number of numbers to sort ?");
      scanf("%d",&n);
      for(i=0;i<n;i++)
          scanf("%d",&a[i]);
      AddrCalculation(a,n);
}
```

```
/* Design, Develop and Implement a Program in C for the
following operationson
     Strings a. Read a main String (STR),
     a. Pattern String (PAT) and a Replace String(REP)
     b. Perform Pattern Matching Operation: Find and Replace all
occurrences of PAT in STR with REP if PAT exists in STR. Report
suitable messages in case PAT does not exist in STR Support the
program with functions for each of the above operations. Don't use
Built-in functions.     */

     #include<stdio.h>
     #include<stdlib.h>
     #include<string.h>
     char str[30];
     int strln(char *str)
     {
          int len=0;
          while(*str++) len++;
          return(len);
     }

     int replace(char pat[], char rep[],int k,int r,int p)
     {
          char temp[30];
          int i,j;
          if(r==p)
          {
               for(i=k-1,j=0;j<p;i++,j++)
                    str[i]=rep[j];
          }
          else
          {
               for(j=k+r-1,i=0;str[j]!='\0';i++,j++)
                    temp[i]=str[j];
               temp[i]='\0';
               for(i=k-1,j=0;j<p;i++,j++)
                    str[i]=rep[j];
               for(j=0;temp[j]!='\0';j++,i++)
                    str[i]=temp[j];
               str[i]='\0';
          }
          return strlen(str);
     }
     int main()
     {
          int l,k=1,max,index,s,r,i,j,p,flag=0;
          char pat[10],rep[10],ele;
          printf("enter a base string: ");
          scanf("%[^\n]s",str);
          scanf("%c",&ele);
          printf("\nenter pattern string: ");
          scanf("%[^\n]s",pat);
          scanf("%c",&ele);
          printf("\nenter replacement string: ");
          scanf("%[^\n]s",rep);
          s=strln(str);
          r=strln(pat);
          p=strln(rep);
```

```
    while(k<=(s-r+1))
    {
        for(l=0;l<r;l++)
        {
            if(pat[l] != str[k+l-1])
            {
                //index=0;
                break;
            }
            //index=k;
        }
        if(l==r)
        {
            flag=1;
            s=replace(pat,rep,k,r,p);
            k=k+p;
        }
        else
            k=k+1;
    }
    if(flag==0)
        printf("\nPattern not found");
    else
    {
        printf("\nBase string after replacement is: %s",str);
    }
    return 0;
}
```

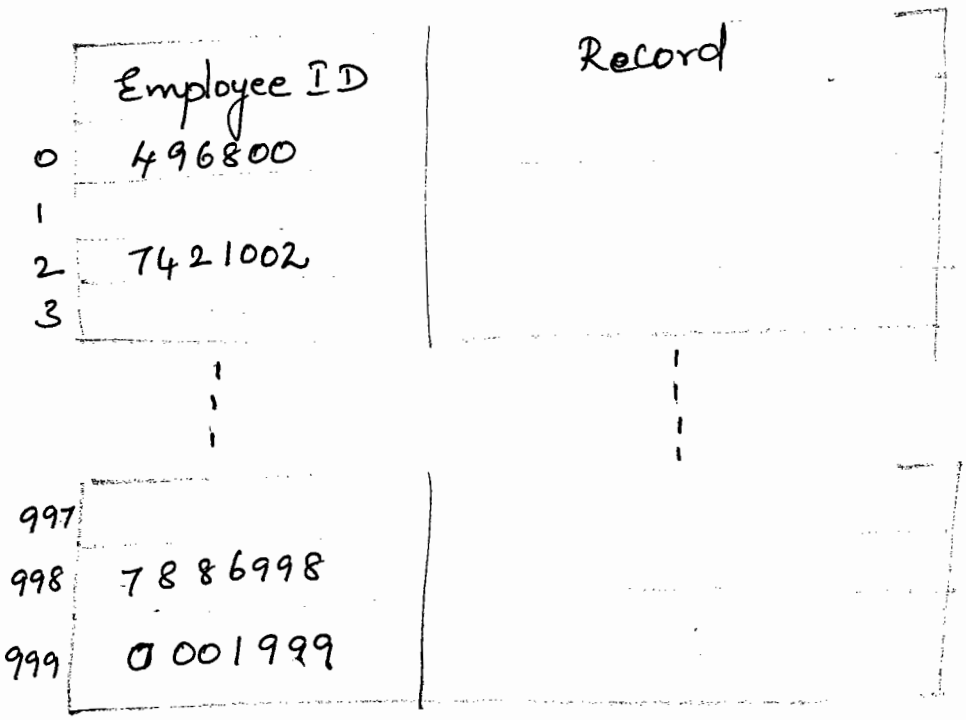
Hashing: Hash table Organizations. Hash functions Static & Dynamic Hashing.

Hashing is a technique that enables us to perform the dictionary operations like search insert & delete in O(1) expected time.
Hashing is the process of mapping large amount of data items to a smaller table with the help of a hashing function.

[In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can ~~be~~ map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found].
- wikipedia.org

Hash Table:
Hash table is a DS used for storing & retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.

For example for storing an employee record in the hash table the employee-ID will work as a key.



$H(key) = key \% 1000$

Searching

linklist – $O(n)$

Binary search — $O(\log_2 n)$

Hashing — $O(1)$

Case study:

$H(x) = x \bmod 10$

x : value

21, 56, 72, 39, 48

986, 13, 75

$H(21) = 21 \bmod 10 = 1$

$H(56) = 56 \bmod 10 = 6$

0	
1	21
2	72
3	13
4	
5	75
6	56, 986 [collision]
7	
8	48
9	39

Hash Functions:

Hash function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table.

The integer returned by the hash function is called hash key.

There are various type of hash functions that are used to place the record in the hash table.

1. Division method: The hash function depends upon the remainder of division. Typically the divisor is table length.

   Ex: If the record 54, 72, 89, 37 is to be placed in the hash table & if the table size is 10 then

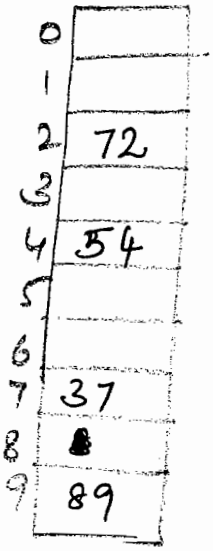
   h(key) = record % table size

   4 = 54 % 10

   2 = 72 % 10

   9 = 89 % 10

   7 = 37 % 10



2. Mid Square: In this method, the key is squared & the middle or mid part of the result is used as the index. If the key is a string it has to be preprocessed to produce a number.

   Ex: Consider that if we want to place a record 3111 then $3111^2 = 9678321$ for the hash table of size 1000

   H(3111) = 783 (the middle 3 digits).

3. Folding method: The key is divided into seperate parts & using some simple operations these parts are combined to produce the hash key.
for example, Consider a record 12365412 then it is divided into seperate parts as 123 654 12 & these are added together.

H(key) = 123 + 654 + 12 = 0789

The record will be placed at location 789 in the hash table.

4. Digit Analysis: The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix r. Then we examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

function-1 to convert a string into a non-negative integer

```
unsigned int stringToInt (char *key)
{ /* simple additive approach to create a
   natural number that is within the integer
   range */
   int number=0;
   while (*key)
       number += *key++;
   return number;
}
```

function-2 to convert a string into a non-negative integer

```
unsigned int stringToInt (char *key)
{ /* alternative additive approach to create
   a natural number that is within the
   integer range */
   int number=0;
   while(*key)
   {
       number += *key++;
       if (*key) number += ((int) *key++)<<8;
   }
   return number;
}
```

Static Hashing.

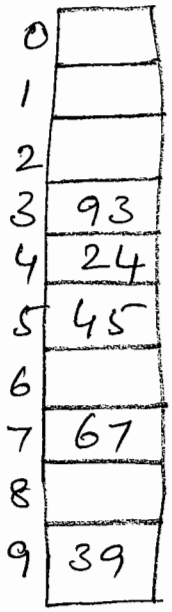
Static hashing is a technique in which the resultant bucket size remains the same. There will not be any change in the bucket address. Various techniques of static hashing are linear probing, chaining.

As the number of buckets are fixed, this type of hashing can not handle over-flow of elements.

Example: Consider the hash function h(10). Consider the elements that can be inserted in the hash table are -

24, 93, 45, 39, 67

24 % 10 = 4
93 % 10 = 3
45 % 10 = 5
39 % 10 = 9
67 % 10 = 7



HASHING

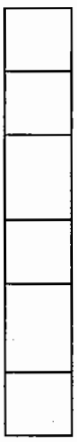
Introduction:

Hashing involves less key comparison and searching can be performed in constant time.

Suppose we have keys which are in the range 0 to n-1 and all of them are unique.

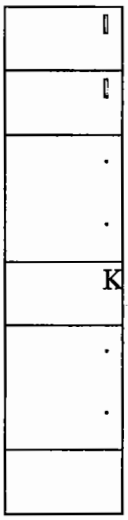
We can take an array of size n and store the records in that array based on the condition that key and array index are same.

The searching time required is directly proportional to the number of records in the file. We assume a function *f* and if this function is applied on key K it returns *i*, an index so that *i=f(K)*.then entry in the access table gives the location of record with key value *K*.



Access Table

INFORMATIONRETRIVAL



File storage of n records

USINGACCESS TABLE.

HASH FUNCTIONS

The main idea behind any hash function is to find a one to one correspondence between a key value and index in the hash table where the key value can be placed. There are two principal criteria deciding a hash function H:K->I are as follows:

i. The function H should be very easy and quick to compute.

ii. The function H should achieve an even distribution of keys that actually occur across the range of indices.

Some of the commonly used hash functions applied in various applications are:

DIVISION:

It is obtained by using the modulo operator. First convert the key to an integer then divide it by the size of the index range and take the remainder as the result.

H(k)=k%i if indices start from 0

H(k)=(k%i)+1 if indices start from 1

MID –SQUARE:

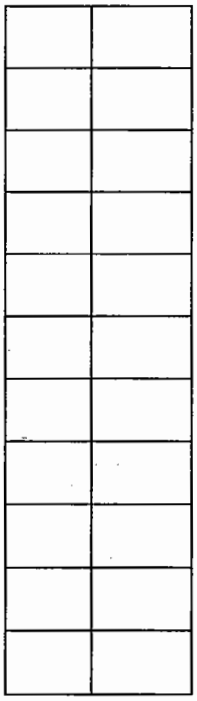
The hash function H is defined by H(k)=x where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value *k*.it is criticized as it is time consuming.

FOLDING:

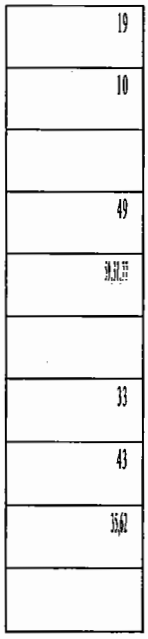
The key is partitioned into a number of parts and then the parts are added together. There are many variation in this method one is *fold shifting method* where the even number parts are each reversed before the addition. Another is the *fold boundary method* here the two boundary parts are reversed and then are added with all other parts.

3. COLLISION RESOLUTION TECHNIQUES

If for a given set of key values the hash functions does not distribute then uniformly over the hash table, some entries are there which are empty and in some entries more than one key value are to be stored. Allotment of more than one key values in one location in the hash table is called *collision.*



HASH TABLE



H:K->I

Collision in hashing cannot be ignored whatever the size of the hash tables. There are several techniques to resolve collisions. Two important methods are:

i. Closed hashing(linear probing)
ii. Open hashing(chaining)

CLOSED HASHING:

The simplest method to resolve a collision is closed hashing. Here the hash table is considered as circular so that when the last location is reached the search proceeds to the first location of the table. That is why this is called closed hashing.

The search will continue until any one case occurs:

1. The key value is found.
2. Unoccupied location is encountered.
3. It reaches to the location where the search was started.

DRAWBACK OF CLOSED HASHING:

As the half of the hash table is filled there is a tendency towards clustering. The key values are clustered in large groups and as a result sequential search becomes slower and slower.

Some solutions to avoid this are:

a. Random probing
b. Double hashing or rehashing
c. Quadratic probing

RANDOM PROBING:

This method uses a pseudo random number generator to generate a random sequence of locations, rather than an ordered sequence as was the case in linear probing method. The random sequence generated by the pseudo random number generator contains all positions between 1 and h, the highest location of the hash table.

I=(i+m)%h+1

i is the number in the sequence

m and *h* are integers that are relatively prime to each other.

DOUBLE HASHING:

When two hash functions are used to avoid secondary clustering then it is called double hashing. The second function should be selected in such a way that hash address generated by two hash functions are distinct and the second function generates a value *m* for the key *k* so that *m* and *h* are relatively prime.

(k)=(k%h)+1

(k)=(k%(h-4))+1

QUADRATIC PROBING:

It is a collision resolution method that eliminates the primary clustering problem of linear probing. For quadratic probing the next location after *i* will be *i+,i++..... etc.*

H(k)+%h for *i*=1,2,3.....

OPEN HASHING

In closed hashing two situations occurs 1. If there is a table overflow situation 2. If the key values are haphazardly intermixed. To solve this problem another hashing is used *open chaining*.

4.ADVANTAGES AND DISADVANTAGES OF CHAINING

1. Overflow situation never arises. Hash table maintains lists which can contain any number of key values.
2. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys so that keys can be searched quickly.
3. Insertion and deletion become quick and easy task in open hashing. Deletion proceeds in exactly the same way as deletion of a node in single linked list.

4. Open hashing is best suitable in applications where number of key values varies drastically as it uses dynamic storage management policy.
5. The chaining has one disadvantage of maintaining linked lists and extra storage space for link fields.

Overflow Handling.

↳ Overcome Collision.

↳ There are 3 common Collision resolution strategies.

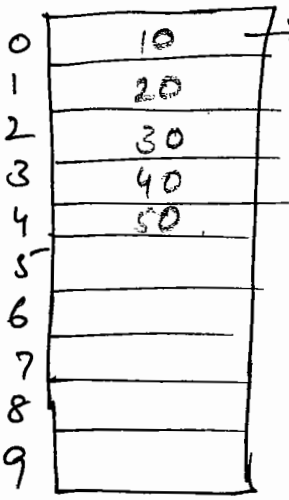
open addressing & Chaining.

1. Linear Probing [linear open addressing]
2. Quadratic probing ~~Double hashing~~.
3. Double hashing
4. Rehashing.
5. Random probing.

Linear Probing.

$H(x) = x \bmod 10$

I/P: 10, 20, 30, 40, 50,



→ 50, 40, 30, 20, 10.

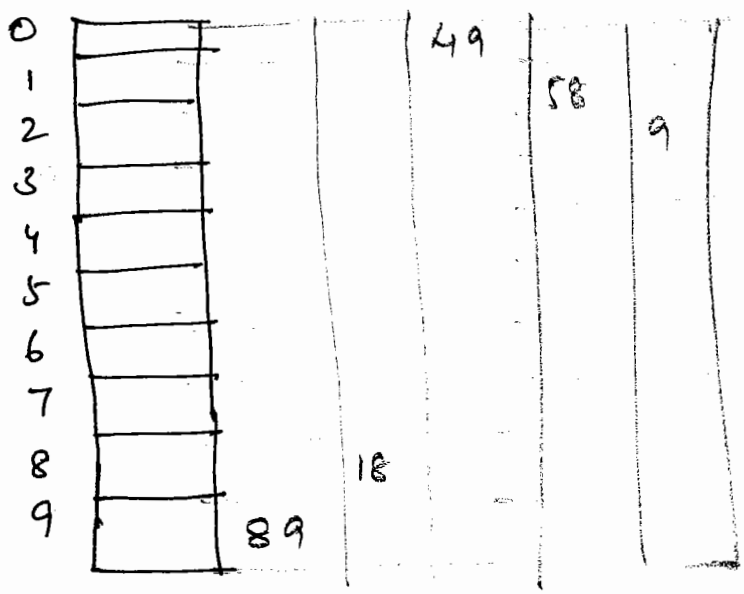
$H_i(x) = (Hash(x) + f(i)) \bmod 10$

$f(i) = i$

$H_0(x) = (Hash(x) + f(0)) \bmod 10.$

- More formally.
  - Cells $h_0(x), h_1(x), h_2(x) \ldots$ are tried in succession where $h_i(x) = (hash(x) + f(i)) \bmod TableSize$, with $f(0) = 0$.
  - In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
    * ie f is a linear function of i, typically $f(i) = i$.

Ex1: hash (89,10) = 9
(18,10) = 8
(49,10) = 9
(58,10) = 8
(9,10) = 9



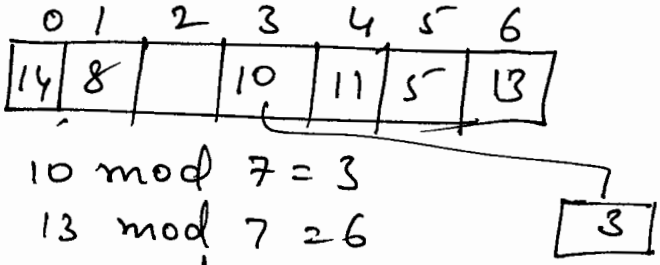
→ 58 would require 4 probes

→ primary clustering problem

```
element * search (int k)
{
  /* search the linear probing hash table ht
     if a pair with key k is found, return a
     ptr to this pair, otherwise, return NULL */
  int homeBucket, currentBucket;
  homeBucket = h(k);
  for (currentBucket = homeBucket;
       ht[currentBucket] && ht[currentBucket]->key != k;)
  {
    currentBucket = (currentBucket + 1) % b;
                    /* circular table */
    if (currentBucket == homeBuket)
      return NULL;
  }
  if (ht[currentBucket]->key == k)
    return ht[currentBucket];
  return NULL;
}
```

Seperate chaining.

Insert 10, 13, 11, 3, 8, 5, 14 into hash table. of size 7 using $f(x) = x \bmod 7$.

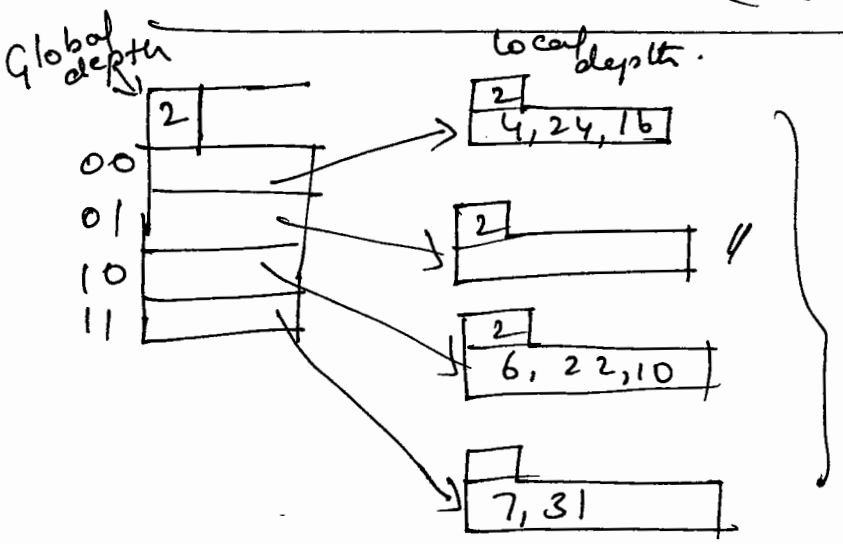


$f(x) = 10 \bmod 7 = 3$
$13 \bmod 7 = 6$
$11 \bmod 7 = 4$
$3 \bmod 7 = 3$
$8 \bmod 7 = 1$
$5 \bmod 7 = 5$
$14 \bmod 7 = 0$

```
element * search (int k)
{ /* search the chained hash table ht for k.
   if a pair with this key is found,
   return a pointer to this pair;
   otherwise, return NULL.
  node Pointer current;
  int homeBucket = h(k);
  /* search the chain ht[homeBucket] */
  for (current = ht [homeBucket];
       current;
       current = current -> link)
    if (current -> data.key == k)
        return & current -> data;
  return NULL;
}
```

Ex
[0] → acos atoi atol
[1] → NULL
[2] → char ceil cos clime
[3] → define
[4] → exp

Extended Hashing (Dynamic)

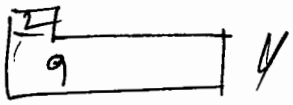


To added 9, 20, 26.

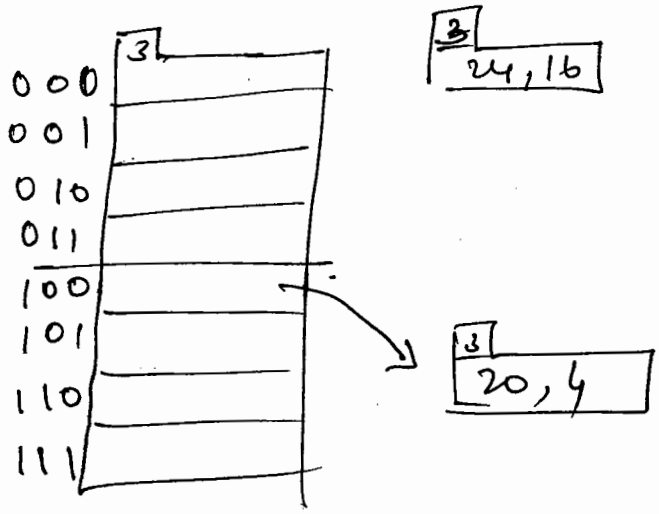
Bucket capacity is 3

16 8 4 2 1
1 0 0 1

step1 add 9 (1001)



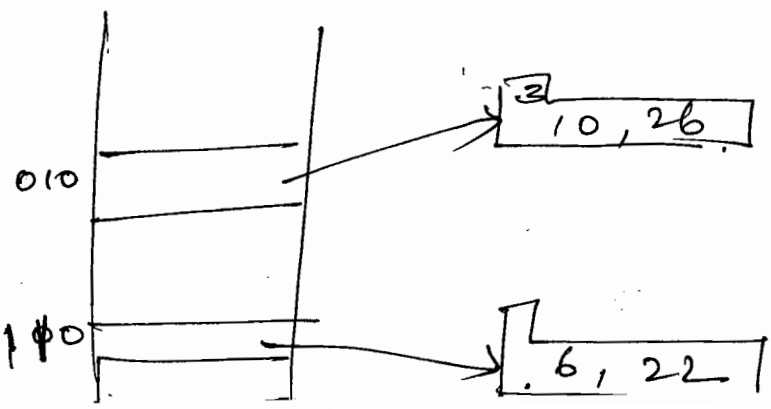
step2 add 20 (10100) should add 2 4, 24, 16 20



apply mod 8 for 4, 24, 16

Step3 add 26 no Expansion but split.

1 1 0



Extended Hashing (Dynamic Hashing)

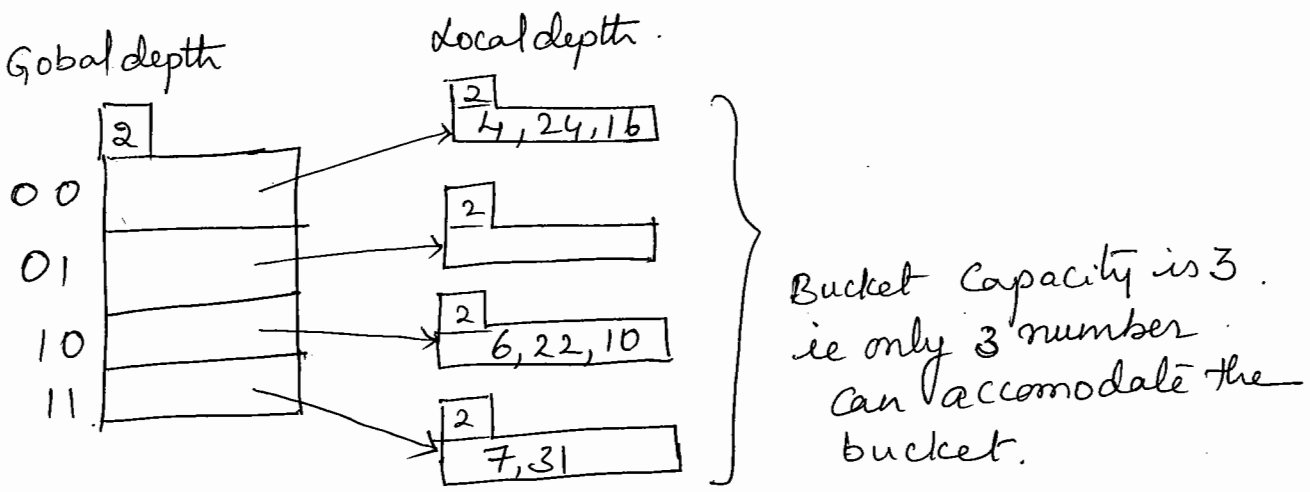
As the database grows over time, we have three options:

- Choose hash function based on current file size. Get performance degradation as file grows.
- Choose hash function based on anticipated file size. Space is wasted initially.
- Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.

Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.

- **Extendable hashing** is one form of dynamic hashing.
- Extendable hashing splits and coalesces buckets as database size changes.
- This imposes some performance overhead, but space efficiency is maintained.
- As reorganization is on one bucket at a time, overhead is acceptably low.

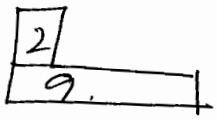
Step1



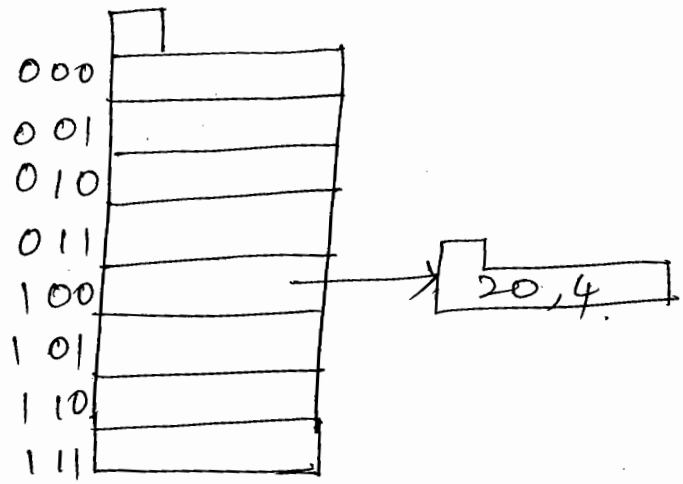
Bucket capacity is 3. ie only 3 number can accomodate the bucket.

Step2 Now add 9, 20, 26.

→ binary value for 9 is 1001
∴ 9 should occupy the 2nd bucket, but the 2nd bucket is Empty.
→ ∴ insert 9 in 2nd bucket.



Step 3: Insert 20.
→ binary value for 20 is 10100
∴ 20 should be inserted in 1st bucket but the 1st bucket is full with 3 values
→ Double the Hash table (ie increase the hash table by 100 %)
→



→ Now Consider 3 binary digits from 10100
→ Insert 20 in the 5th bucket.
→ Also recalculate the binary values for all the numbers from the 2nd bucket & distribute the number respectively.
→ ∴ 4 moves to 100 bucket from 000.

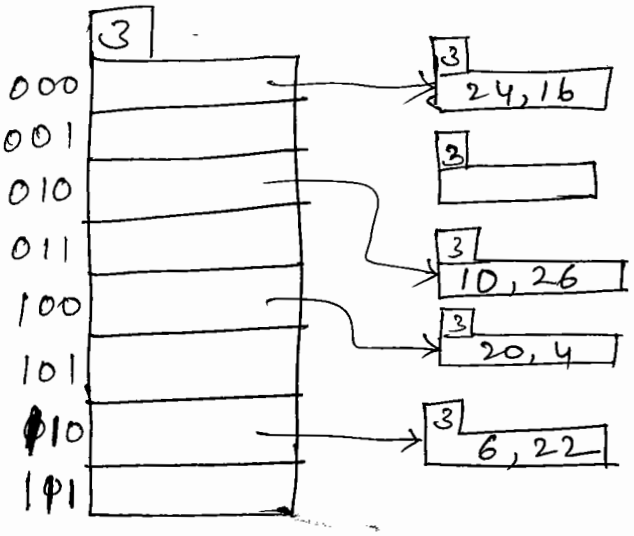
Step4: Insert 26, binary value for 26 => 11010

→ 26 % 8 = 2.

→ The 3rd bucket contains 3 elements (ie it is full.

→ Recalculate the mod values for the existing no: from the 3rd Bucket.

→ Redistribute the values.



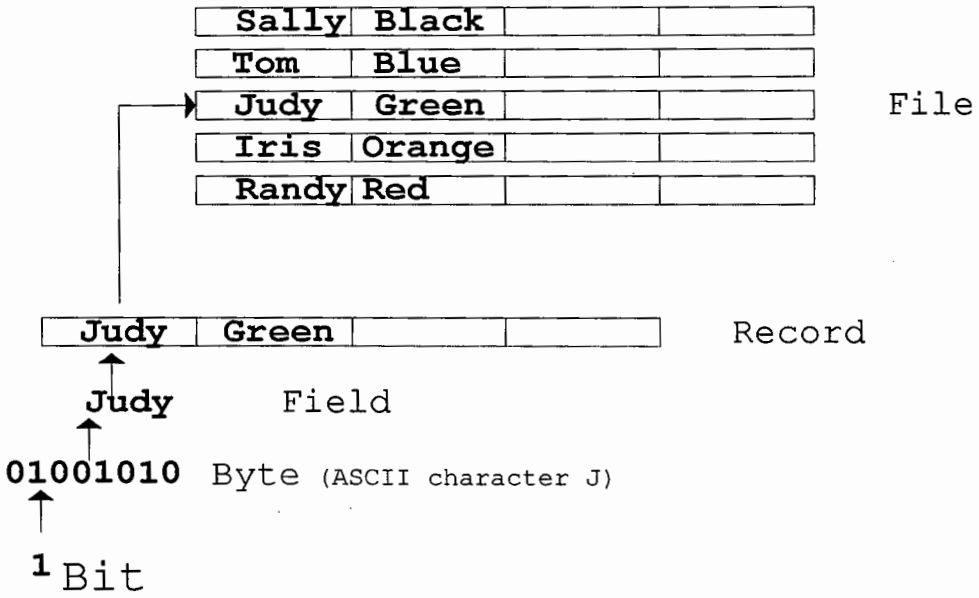
Files (Module-5)

Data Hierarchy

- Bit – smallest data item
  - Value of 0 or 1
- Byte – 8 bits
  - Used to store a character
    - Decimal digits, letters, and special symbols
- Field – group of characters conveying meaning
  - Example: your name
  - Record – group of related fields
  - Represented by a struct or a class
  - Example: In a payroll system, a record for a particular employee that contained his/her identification number, name, address, etc.

- File – group of related records
  - Example: payroll file

Database – group of related files



- A **file** is a collection of related data that a computer treats as a single unit.
- C uses a structure called FILE (defined in stdio.h) to store the attributes of a file.

Flat-File (one table)

Patient Id	Name	D.o.B	Gender	Phone	Doctor Id	Doctor	Room
134	Jeff	4-Jul-1993	Male	7876453	01	Dr Hyde	03
178	David	8-Feb-1987	Male	8635467	02	Dr Jekyll	06
198	Lisa	18-Dec-1979	Female	7498735	01	Dr Hyde	03
210	Frank	29-Apr-1983	Male	7943521	01	Dr Hyde	03
258	Rachel	8-Feb-1987	Female	8367242	02	Dr Jekyll	06

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:
Text files
Binary files

1. Text files
   - Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.
   - When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
   - They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files
   - Binary files are mostly the .bin files in your computer.
   - Instead of storing data in plain text, they store it in the binary form (0's and 1's).
   - They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations

In C, you can perform four major operations on the file, either text or binary:

- Creating a new file
- Opening an existing file
- Closing a file
- Reading from and writing information to a file

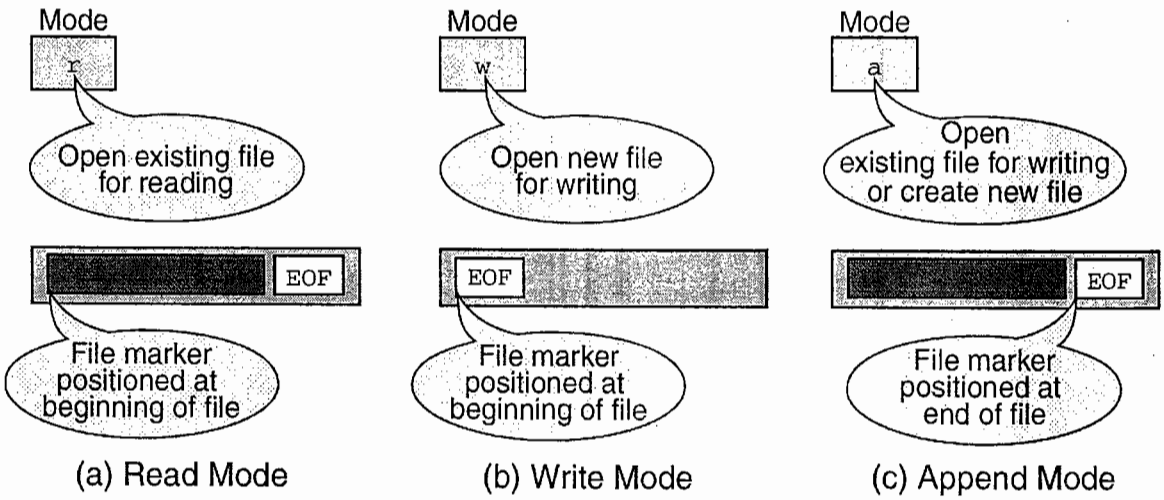
The Basic file operation

- fopen - open a file- specify how its opened (read/write) and type (binary/text)
- fclose - close an opened file
- fread - read from a file
- fwrite - write to a file
- fseek/fsetpos - move a file pointer to somewhere in a file.
- ftell/fgetpos - tell you where the file pointer is located.

File Open Modes

Opening Modes in Standard I/O		
File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.

Opening Modes in Standard I/O		
File Mode	Meaning of Mode	During Inexistence of file
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.



(a) Read Mode (b) Write Mode (c) Append Mode

Text Files and Binary

Text File	Binary File
Text files contain plain text data.	Binary file contain the data in binary form.
Does not contain graphical data.	Contains data such as text, graphics, image and sound.
Directly read and interpreted.	Cannot be read directly. With help of some tool the binary file can be read. HexDump is used to read the binary file.
They are not executable files.	Binary files can be executable files.

File Organizations

The various file organization methods are:

- Sequential access.
- Direct or random access.
- Index sequential access.

The selection of a particular method depends on:

- Type of application.
- Method of processing.
- Size of the file.
- File inquiry capabilities.
- File volatility.
- The response time.

Sequential access method: Here the records are arranged in the ascending or descending order or chronological order of a key field which may be numeric or both. Since the records are ordered by a key field, there is no storage location identification. It is used in applications like payroll management where the file is to be processed in entirety, i.e. each record is processed. Here, to have an access to a particular record, each record must be examined until we get the desired record. Sequential files are normally created and stored on magnetic tape using batch processing method.

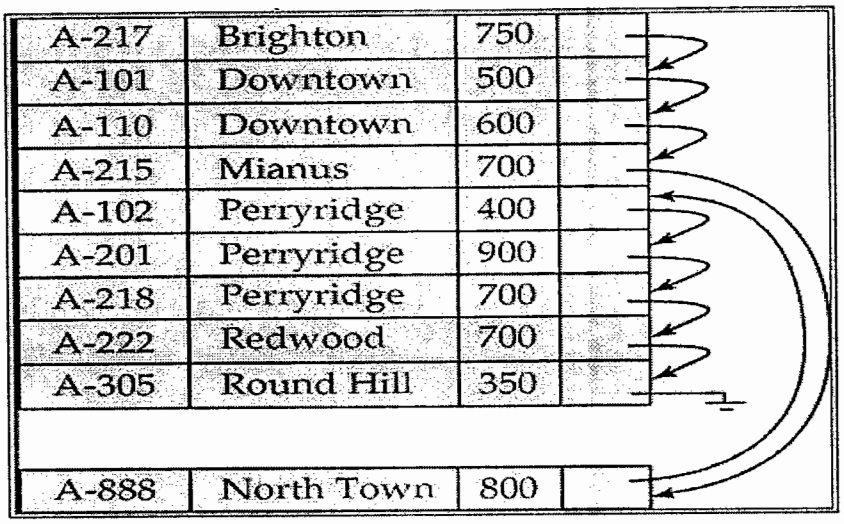


Fig: Sequential access

Advantages:

- Simple to understand.
- Easy to maintain and organize
- Loading a record requires only the record key.
- Relatively inexpensive I/O media and devices can be used.
- Easy to reconstruct the files.
- The proportion of file records to be processed is high.

Disadvantages:

- Entire file must be processed, to get specific information.
- Very low activity rate stored.
- Transactions must be stored and placed in sequence prior to processing.
- Data redundancy is high, as same data can be stored at different places with different keys.
- Impossible to handle random enquiries.

Direct access files organization: (Random or relative organization). Files in his type are stored in direct access storage devices such as magnetic disk, using an identifying key. The identifying key relates to its actual storage position in the file. The computer can directly locate the key to find the desired record without having to search through any other record first. Here the records are stored randomly, hence the name random file. It uses online system where the response and updation are fast.

Advantages:

- Records can be immediately accessed for updation.
- Several files can be simultaneously updated during transaction processing.
- Transaction need not be sorted.
- Existing records can be amended or modified.
- Very easy to handle random enquiries.
- Most suitable for interactive online applications.

Disadvantages:

- Data may be accidentally erased or over written unless special precautions are taken.
- Risk of loss of accuracy and breach of security. Special backup and reconstruction procedures must be established.
- Less efficient use of storage space.
- Expensive hardware and software are required.
- High complexity in programming.
- File updation is more difficult when compared to that of sequential method.

Indexed sequential access organization: Here the records are stored sequentially on a direct access device i.e. magnetic disk and the data is accessible randomly and sequentially. It covers the positive aspects of both sequential and direct access files. The type of file organization is suitable for both batch processing and online processing. Here, the records are organized in sequence for efficient processing of large batch jobs but an index is also used to speed up access to the records. Indexing permit access to selected records without searching the entire file.

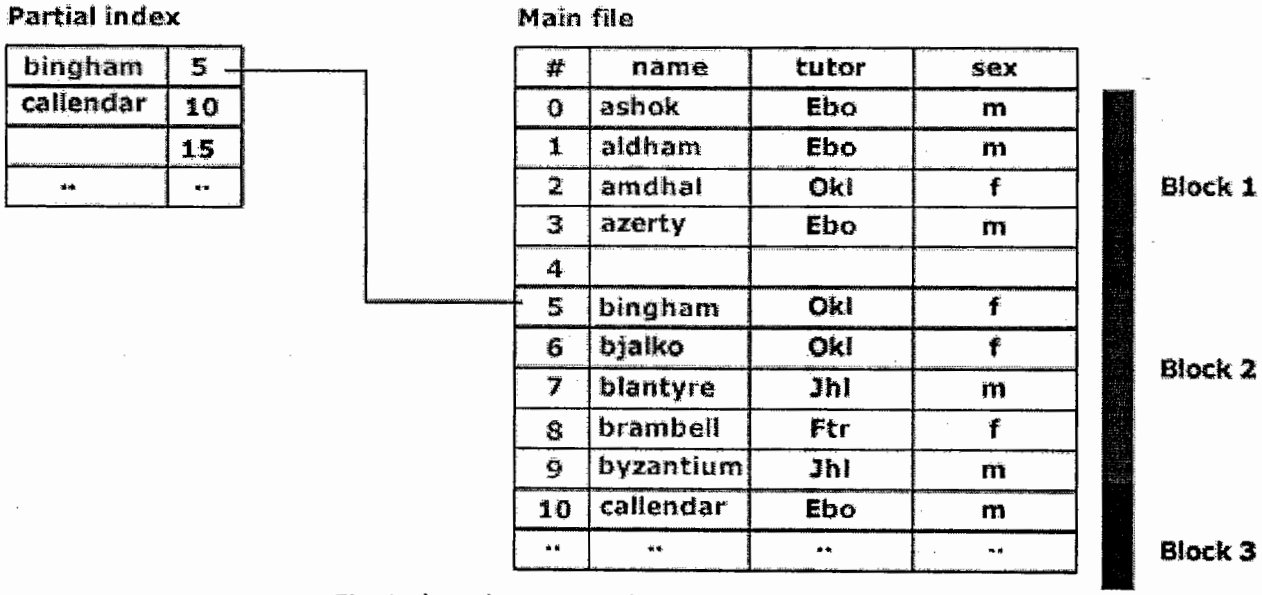


Fig: Indexed sequential access organization

Advantages:

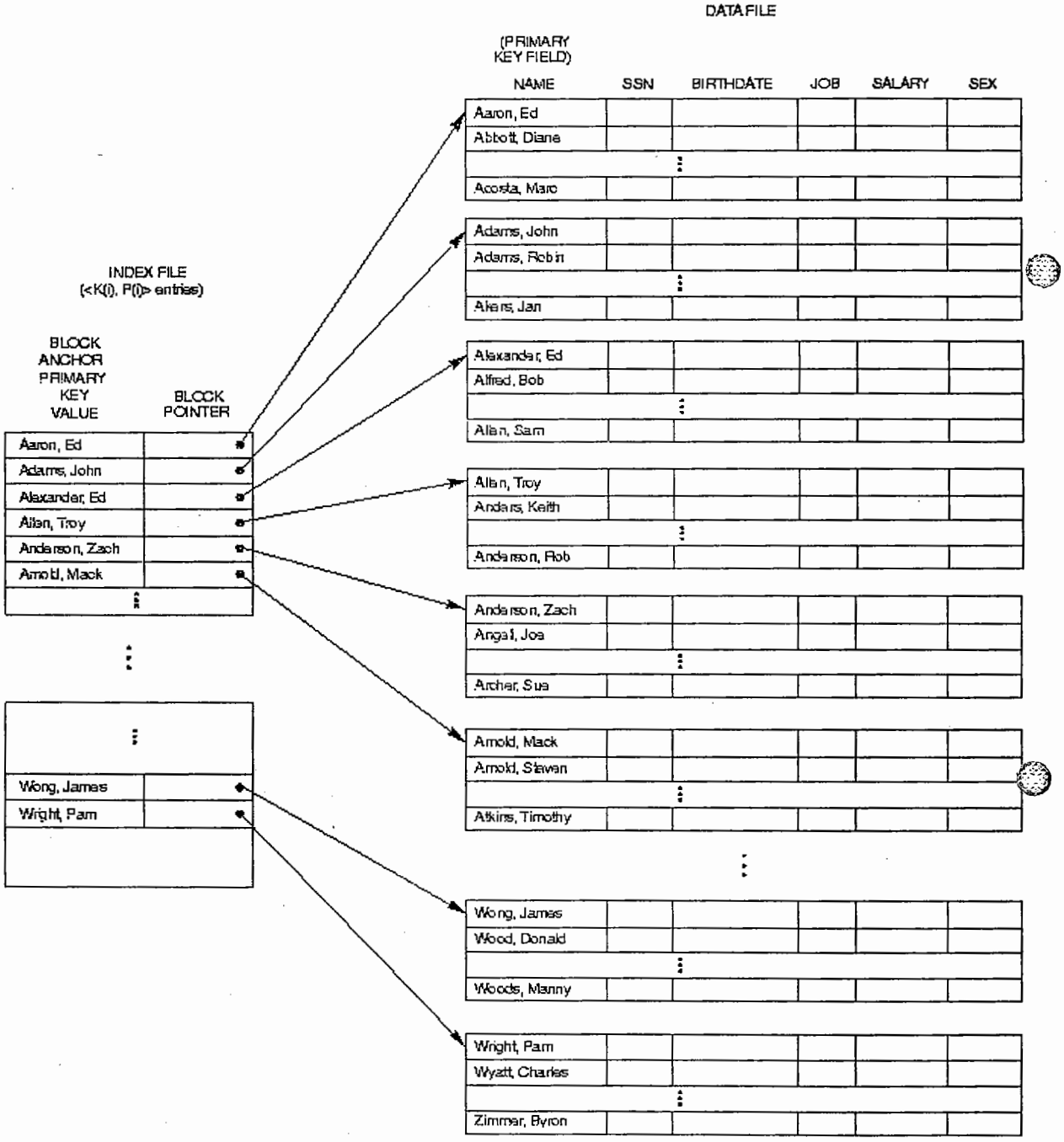
- Permits efficient and economic use of sequential processing technique when the activity rate is high.
- Permits quick access to records, in a relatively efficient way when this activity is a fraction of the work load.

Disadvantages:

- Slow retrieval, when compared to other methods.
- Does not use the storage space efficiently.
- Hardware and software used are relatively expensive.

Indexing

Indexes are additional auxiliary access structures with typically provide either faster access to data or secondary access paths without effecting the physical storage of the data. They are based on indexing field(s) that are used to construct the index.



/* Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F. Assume that file F is maintained in memory by a Hash Table(HT) of m memory locations with L as the set of memory addresses (2- digit) of locations in HT. Let the keys in K and addresses in L are Integers. Design and develop a Program in C that uses Hash function H: K → L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L. Resolve the collision (if any) using linear probing.*/

```
#include<stdio.h>
#include<stdlib.h>
typedef struct emp
 {
   int empno; // 3 digit
   char name[20];
   int sal; //3 digit
 } EMPLOYEE;

 void main()
  {
    EMPLOYEE E,f ;
    FILE  *fp;
    char filename[30];
    int i,n,s = 28,index,id,indexcopy,choice,flag=0,j,minusone = -1;

      //printf("%ld %ld
%ld",sizeof(EMPLOYEE),sizeof(int),sizeof(char));
    printf("Enter number of records\n");
    scanf("%d",&n);
    //printf("\nEnter name of the file : ");
    //scanf("%s",filename);
    fp=fopen("emp1.txt","w+");
    for(i=0;i<n;i++)
      {
            fwrite(&minusone,sizeof(int),1,fp);
            //for(j=0;j<(24);j++)
                  //fprintf(fp," ");
            fseek(fp,s - sizeof(int),SEEK_CUR);
      }
      while(1)
      {
            printf("\n1. Add record\n2. Retrieve record\n3. Display
all\n4. Exit\n");
            scanf("%d",&choice);
            flag = 0;
            switch(choice)
            {
            case 1:
                  printf("\nEnter eno,name,sal: ");
                  scanf("%d%s%d",&E.empno,E.name,&E.sal);
                  index = indexcopy = E.empno % n; //hashing function
             // printf("%d\t %d\n",index, s);
                  fseek(fp,index*s,SEEK_SET);
                  fread(&id,sizeof(int),1,fp);
                  //printf("\n%d",id);
                  //Linear probing
                  while(id != -1)
```

```
                {
                        //printf("\n%d",id);
                        index++;
                        //printf("%d \t ",index);
                        fseek(fp,index*s,SEEK_SET);
                        flag = 1;
                        if(index == n)
                                index = 0;
                        if(index == indexcopy)
                        {
                                printf("\nFile full! Cannot insert!");
                        break;
                       }
                      fread(&id,sizeof(int),1,fp);
                }
                if(!(index == indexcopy && flag))
                {
                        fseek(fp,index*s,SEEK_SET);
                        fwrite(&E,sizeof(EMPLOYEE),1,fp);
                }
                break;
        case 2:
                printf("\nEnter employee id :");
                scanf("%d",&E.empno);
                index = indexcopy = E.empno % n;
                fseek(fp,index * s,SEEK_SET);
                fscanf(fp,"%d",&id);
                while(id != E.empno && id != -1)
                {
                        index++;
                        fseek(fp,index * s,SEEK_SET);
                        if(index == n)
                                index = 0;
                        if(index == indexcopy)
                        {
                                flag = 1;
                                break;
                        }
                        fscanf(fp,"%d",&id);
                }
                if(id == -1 || flag)
                        printf("\nRecord not found!");
                else{
                        fseek(fp,index*s,SEEK_SET);
                        fscanf(fp,"%d%s%d",&E.empno,E.name,&E.sal);
                        printf("\nRecord details : %d %s
%d",E.empno,E.name,E.sal);
                }
                break;
        case 3:
                printf("\nRecords in file are: \n");
                for(index = 0;index<n;index++)
                {
                        fseek(fp,index*s,SEEK_SET);
                        fread(&E.empno,sizeof(int),1,fp);
                        printf("\n%d ",E.empno);
                        if(E.empno != -1)
                        {
```

```
                                fread(E.name,sizeof(EMPLOYEE) -
sizeof(int),1,fp);
                                printf("%s %d",E.name,E.sal);
                            }

                        }
                        break;
                case 4:
                fclose(fp);
                exit(0);
        }
       }
  }
```

```
/* Write a C program to write all the members of an array of
strcures to a file using fwrite(). Read the array from the file and
display on the screen.*/
#include <stdio.h>
#include <stdlib.h>
struct student
{
   char name[50];
   int height;
};
int main()
{
      struct student stud1[5], stud2[5];
      FILE *fptr;
      int i;

      fptr = fopen("file.txt","wb");
      for(i = 0; i < 3; ++i)
      {
          fflush(stdin);
          printf("Enter name: ");
          gets(stud1[i].name);
            getchar();

          printf("Enter height: ");
          scanf("%d", &stud1[i].height);
      }

      fwrite(stud1, sizeof(stud1), 1, fptr);
      fclose(fptr);

      fptr = fopen("file.txt", "rb");
      fread(stud2, sizeof(stud2), 1, fptr);
      for(i = 0; i < 3; ++i)
      {
          printf("Name: %s\nHeight: %d", stud2[i].name,
stud2[i].height);
      }
      fclose(fptr);
}
```

Continuation of Hashing..

Concept of collision.

Def: The situation in which the hash function returns the same hash key for more than one record is called collision & two same hash keys returned for different records is called Synonym.

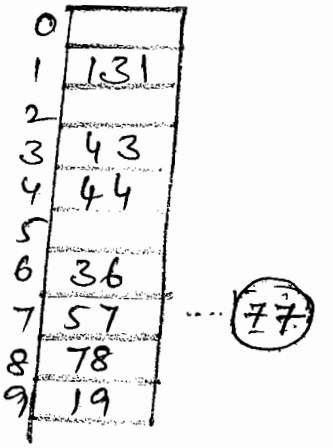
for example: Consider a hash function H(key) = recordkey % 10 having the hash table of size 10. The record keys to be placed are 131, 44, 43, 78, 19, 36, 57 & 77

→ we try to place all the values in the hash table.

→ 77 with trying to place in the hashtable yeilds a index value of 7.

→ The location is already occupied by 57.

→ This situation is called collision



Threaded Binary Tree.

→ It may be observed that in the linked representation of binary tree, there are more null links than actual Pointers.

→ There are n+1 null links out of 2n total links.

→ These null links are replaced by pointers called threads to other nodes in the tree.

→ To Contruct the threads we use the following rules

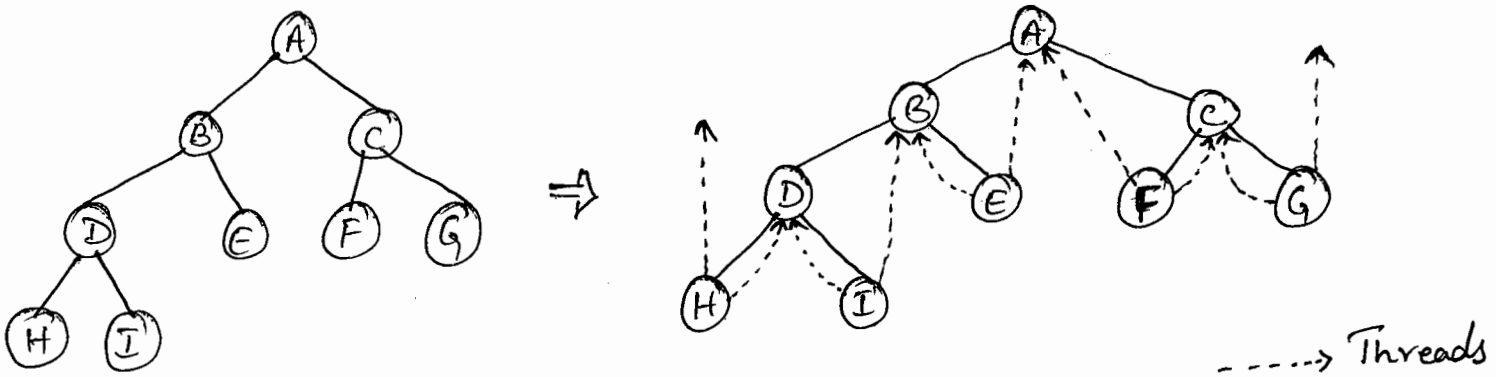
① If ptr→leftchild is null, replace ptr→leftchild with a pointer to the node that would be visited before ptr in an inorder traversal.

② If ptr→rightchild is null, replace ptr→rightchild with a pointer to the node that would be visited after ptr in an inorder traversal.

→ Data structure used for Threaded binary tree

```
typedef struct threadedTree *threadedPtr;
typedef struct
{
    short int leftThread, rightThread;
    threadedPtr leftChild, rightChild;
    char data;
} threadTree;
```

Complete binary Tree. Threaded tree

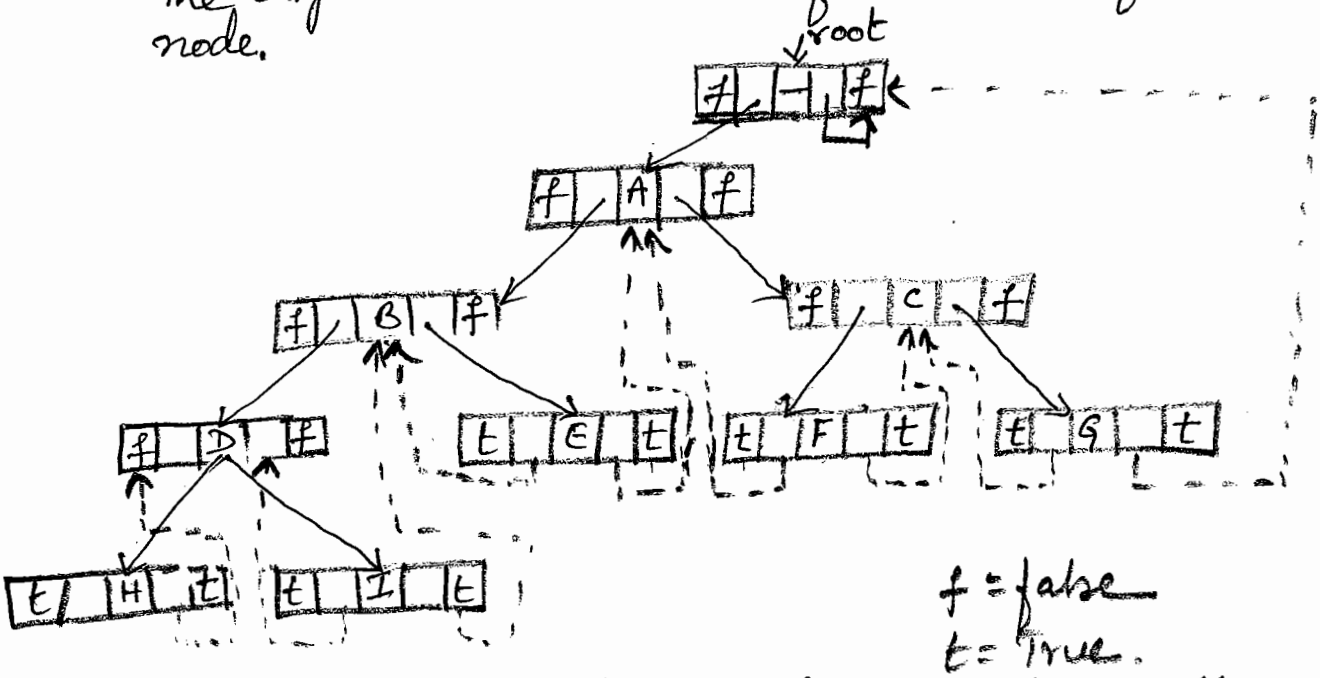


-----> Threads

→ In the above threaded tree we find two threads been left dangling. ie one in the left child of H the other in the right child of G.

→ In order, not to leave these threads loose, we assume a new header node for all threaded binary trees.

→ The original tree is the left subtree of the header node.



f = false
t = true.

Memory representation of threaded tree

Inorder traversal of a Threaded Binary Tree

```
threadedPtr insucc (threadedPtr tree)
{ /* find the inorder sucessor of tree in a
     threaded binary tree */
     threadedPtr temp;
     temp = tree->rightchild;
     if(!tree->rightThread)
          while(!temp->leftThread)
               temp-> temp->leftchild;
     return temp;
}
```

→ By using threads, we perform inorder traversal without making use of a stack.

Inserting a Node into a threaded Binary Tree

```
void tinorder (threadptr tree)
{ /* traverse the threaded binary tree inorder */
     threadedptr temp = tree;
     for (;;)
     {
          temp = insucc (temp);
          if (temp == tree) break;
          printf ("%3c", temp-> data);
     }
}
```

University

Question Papers

Data Structures and Applications

15CS33

Global Academy of Technology

Department of Computer Science and Engg.

Data Structures and Applications (15CS33)

Question Bank

(Repository of Frequently Asked Questions)

FAQ's Module 1

December 2011

- Develop a structure to represent planets in the solar system. Each planet has fields for the planet's name, its distance from the sun in miles and the number of moons it has. Write a program to read data for each planet and store. Also print the planet that has highest number of moons.
- For the given sparse matrix and its transpose, give the triplet representation using one dimensional array, a is the sparse matrix, b will be the transpose.

$$A = \begin{matrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{matrix}$$

- Consider 2 polynomials $A(x)= 2x^{1000}+1$ and $B(x)=x^4+10x^3+3x^2+1$, show diagrammatically how these two polynomials can be stored in a single one-dimensional array. Also give its C representation.

June 2012

- Define a pointer. Write a C function to swap two numbers using pointers.
- Explain the functions supported by C to carry out dynamic memory allocation.
- Define structure and union with suitable examples.

- Write a C program with an appropriate structure definition and variable declaration to store information about an employee using nested structures. Consider the following fields Ename, Empid, DOB and salary(Basic, DRA, HRA).

Jan 2013

- What are pointer variables? How to declare a pointer variable?
- What are the various memory allocation techniques? Explain how memory can be dynamically allocated using malloc() ?
- What is the difference between int *a and int a[5] and int *[5]?
- What is a structure? How to declare and initialize a structure?
- Write a program in C to read a sparse matrix of integer values and search this matrix for an element specified by the user.

June/July 2013

- Define pointer. With examples, explain pointer declaration, pointer initialization and use of the pointer in allocating a block of memory dynamically.
- What is a structure? Give three different ways of defining structure and declaring Variables and method of accessing members of structures using a student structure withroll number, name and marks in 3 subjects as members of that structure as example
- Define sparse matrix and show with a suitable example sparse matrix representation storing as triplets. Give simple transpose function to transpose sparse matrix.
- How would you represent two sparse polynomials using array of structure and also write afunction to add that polynomial and store the result in the same array.

Dec 2013/Jan 2014

- Develop a structure to represent planet in the solar system. Each planet has fields for theplanet's name, its distance from the sun in miles and the

number of moons it has. Write a program to read the data for each planet and store. Also print the name of the planet that has less distance from the sun.

- What is polynomial? What is the degree of the polynomial? Write a function to add two polynomials?
- For the given sparse matrix A and its transpose, give the triplet representation. A is the given sparse matrix, and B will be its transpose.

15	0	0	22	0	-15
0	11	3	0	0	0
0	0	0	-6	0	0
0	0	0	0	0	0
91	0	0	0	0	0
0	0	-28	0	0	0

Dec 2014/Jan 2015

- Define structure and union with suitable example.
- Write a C program with an appropriate structure definition and variable declaration to store information about an employee, using nested structures. Consider the following fields like : ENAME, EMPID, DOJ (Date, Month, Year) and Salary (basic, DA, HRA).

June 2016

- What is a structure? Give three different ways of defining structure with example to each.

FAQ's Module 2

December 2011

- Write a recursive function to implement binary search.
- Define stack. Give C implementation of push and pop functions. Include check for empty and full conditions of stack.
- Write a C program to evaluate postfix expression.
- For a given circular queue as shown, write the values of front and rear in the table after each specified operation is performed. Queue full or empty conditions must be considered. 0-7 indicates array indices.

June 2012

- Write a C program to implement the two primate operations on stack using dynamic memory allocation.
- Write an algorithm to convert infix to postfix expression and apply the same to convert the following expressions from infix to postfix:
  i. (a*b)+c/d
  ii. (((a/b)-c)+(d*e))-a*c)

Jan 2013

- Obtain the postfix and prefix expression for

  (((A+(B-C)*D)^E)+F).

- Define stack. List the operations on stack.
- What is recursion? What are the various types of recursion?

June/July 2013

- Define recursion. Give two conditions to be followed for successive working of recursive program. Given recursive implementation of binary's search with proper method.
- Give definition of a stack with necessary function. Explain implementing stacks to hold records with different type of fields in stack.
- Give the disadvantages of ordinary queue and how it is solved in circular queue. Explain the same. Explain with suitable example how you would implement circular queue using dynamically allocated arrays.
- Convert the infix expression **a/b-c+d*c-a*c** into postfix expression. Write a function to evaluate that postfix expression and trace that for given data a=6, b=3, c=1, d=2, e=4.

Dec 2013/Jan 2014

- Define stack. Implement push and pop functions for stack using arrays.
- Implement addq and deleteq functions for the circular queue.
- Write the postfix form of the following expressions
  - (a+b)*d+e/(f+a*d)+c
  - ((a/(b-c+d))*(e-a)*c)

- a/b-c+d*e-a*c.

Dec 2014/Jan 2015

- Define stack. Give the C implementation of push and pop functions. Include check for empty and full conditions of stack.
- Write an algorithm to convert infix to postfix expression and apply the same to convert the following expression from infix to postfix :
  i) (a*b)+c/d
  ii) (((a/b(-e)+(d*e))-(a*c)).

June 2015

- Write a recursive function to sum a list of numbers and also show the total stop counts for the function.
- Convert the following infix expressions to postfix using stack:
  i. A*(b+c)*d
  ii. ((a+b)*d+e)/((f+a*d)+c))
- Write a C function to evaluate a postfix expression and apply the same to evaluate: AB+CDE-*/ ; A=5; B=6; C=4; D=3; E=7

FAQ's Module 3

December 2011

- Explain how a chain can be used to implement a queue. Write the function to insert and delete elements from such a queue.
- Describe doubly linked lists with advantages and disadvantages. Write a C function to delete a node from a doubly linked list, ptr is the pointer which points to the node to be deleted. Assume that there are nodes on either side of the node to be deleted.
- For the given sparse matrix, give the diagrammatic linked representation:

	0	1	2
A=	3	0	0
	0	0	0

June 2012

- Define linked list. Write a C program to implement the insert and delete operation on a queue using linked list.
- Write a C function to add 2 polynomials using linked list representation. Explain with suitable examples.

Jan 2013

- What is a linked list? Explain the different types of linked lists with diagram.
- Write a function to insert a node at front and rear end in a circular linked list. Write down sequence of steps to be followed.

June /July 2013

- Give the node structure to create a linked list of integers and write C functions to perform the following:
  i) Create a three-node list with data 10,20 and 30
  ii) Insert a node with data value 15 in between the nodes having data values 10 and 20
  iii) Delete the node which is followed by a node whose data value is 20
  iv) Display the resulting singly linked list.
- With node structure show how would you store the polynomials in linked lists? Write C function for adding two polynomials represented as circular lists.
- Write a note on: i) Linked representation of sparse matrix ii) Doubly linked list

Dec 2014/Jan 2015

- Define linked list. Write a C program to implement the insert and delete operation on queue using linked list.
- Explain the different types of linked lists with diagrams.

June 2015

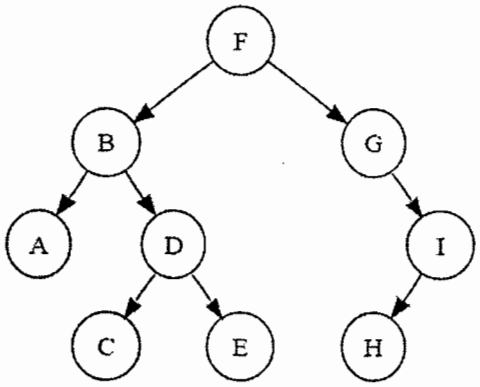
- Write a C function to insert a node at front and rear end in a circular linked list.
- Write a C function to reverse the given linked list.
- Write a C function to concatenate two singly linked list.

- Represent the following polynomial using a circular singly linked list. Give the C representation as well.
  - $4x^5+8x^3+8^x+9$
  - $12x^4y^2z+7x^8y^2z^5-4x^6yz^3+16xyz$

FAQ's Module 4

December 2011

- With reference to the diagram answer the following:
  i. Is it a binary tree?
  ii. Is it a complete tree?
  iii. Give the preorder traversal
  iv. Give the postorder traversal.
  v. Give the inorder traversal.



- Construct B-tree from given traversals:
  i. Preorder: ABDCEF
  ii. Inorder: BDAEFC
  iii. Postorder: DBFECA

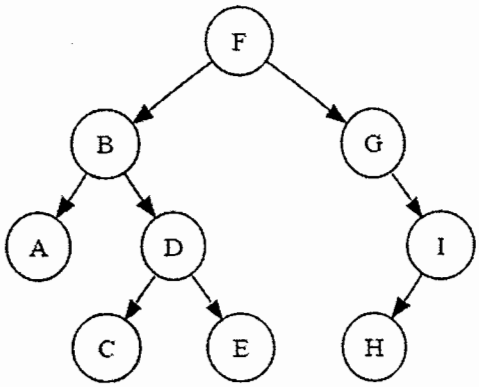
June 2012

- Define binary tree. For the given binary tree find the following:
  i. Siblings
  ii. Leaf nodes
  iii. Non-leaf nodes

iv. Ancestors

v. Level of trees

- Write the C routines to traverse the given tree using
  i) Inorder ii) Postorder iii)Preorder



- Define a binary search tree. Write the iterative search function and recursive search function of BST.

January 2013

- What is a tree? Explain : i) root node , ii) child, iii) siblings, iv) ancestors using structure representation.
- What is a binary tree? How it is represented using array and linked list.
- What is a binary search tree? Draw the binary search tree for the following input: 14,5,6,2,18,20,16,18,-1,21.

June/ July 2013

- Define a binary tree and with example show array representation and linked presentationof binary tree.
- Write an expression tree for an expression A/B+C*D+E. Give the algorithm for inorder, postorder and preorder traversals and apply that traversal method to the expression treeand give the result of traversals.
- Define a binary search tree and construct a binary search tree. With elements {22,28,20,25,22,15,18,10,14}. Give recursive search algorithm to search an element in that tree.
- Construct a binary tree having the following sequences.
  i) Preorder sequence : ABCDEFGHI
  ii) Inorder sequence : BCAEDGHFI

Show the steps in constructing binary tree in the above example.

Dec 2014/Jan 2015

- Define the following
  i) Binary tree
  ii) Complete binary tree
  iii) Almost complete binary tree
  iv) Binary search tree
  v) Depth of a tree
- In brief describe any five applications of trees.
- What is a threaded binary tree? Explain right and left in threaded binary tree.
- Write a C function for the following tree traversal:
  i) Inorder
  ii) Preorder
  iii) Postorder

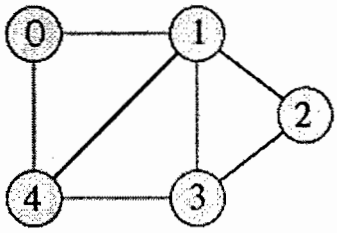
June 2015

- What is a binary tree? Show the array representation and linked representation for the following binary tree:
- Write an expression tree for the expression A/B*C*D+E. Give C functions for inorder, preorder and postorder traversals and apply the traversal methods to the expression tree and give the result of traversals.
- Draw binary search tree for the following input: 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9. Give recursive search function to search an element in that tree.
- Construct binary tree for the following traversals:
  Preorder: ABDGCEHIF
  Inorder: DGBAHEICF

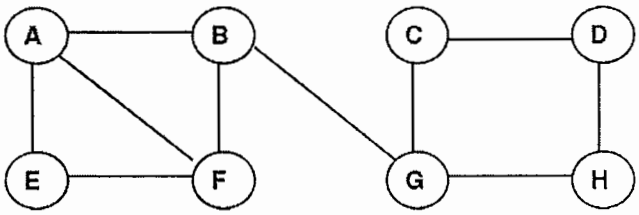
FAQ's Module 5

- Define a graph with an example,
- Explain (i) path (ii) cycle (iii) adjacency (iv)digraph with an example

- Give the adjacency matrix and adjacency lists representation for the graph shown below



- List and explain the graph traversal methods.
- Write the functions to obtain BFS and DFS traversals for a given graph
- Traverse the following graph using both DFS and BFS traversals



- List and explain various hash functions.
- What is meant by collision in hashing?
- With an example explain how linear probing can be employed in resolving a collision
- With an example show the working of

  (i) Insertion sort

  (ii) Radix sort

  (iii) Address calculation sort

- Define a file
- Explain data hierarchy
- Explain various file attributes

- List and explain various operations performed on a file
- Write a note on different file organization

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

USN | | | | | | | | | | 10CS3

Third Semester B.E. Degree Examination, June/July 2015
Data Structures with C

Time: 3 hrs. Max. Marks:100

Note: *Answer FIVE full questions, selecting at least TWO questions from each part.*

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

PART – A

1 a. What is an algorithm? Briefly explain the criteria that an algorithm must satisfy. (06 Marks)

b. Write an recursive function to sum a list of numbers and also show the total step counts for the function. (07 Marks)

c. Define three asymptotic notations and give the asymptotic representation of function 10n+5 in all the three notations. (07 Marks)

2 a. What is a structure? Give three different ways of defining a structure with example to each. (07 Marks)

b. What is the degree of the polynomial? Consider the two polynomials $A(x) = x^{1000} + 1$ and $B(x) = 10x^3 + 3x^2 + 1$. Show diagrammatically how these two polynomials can be represented in a array. (05 Marks)

c. For the given sparse matrix and its transpose, give the triple using one dimensional array. A is the given sparse matrix, B will be its transpose. (08 Marks)

$$A = \begin{bmatrix} 15 & 0 & 2 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Fig. Q2 (c)

3 a. Convert the following infix expression into postfix expression using stack:

i) a*(b+c)*d

ii) $\dfrac{(a+b)*d+e}{(f+a*d)*}$ (08 Marks)

b. Write a C function to evaluate a postfix expression and apply the same to evaluate AB + CD[illegible] */, A = [illegible] = 6, C = 4, D = 3, E = 7. (12 Marks)

4 a. Write a C function to insert a node at front and rear end in a circular linked list. (10 Marks)

b. i) Write a C function to reverse the given singly linked list.

ii) Write a C function to concatenate two singly linked list. (10 Marks)

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

PART – B

5 a. What is a binary tree? Show the array representation and linked representation for the following binary tree. (05 Marks)

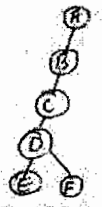


Fig. Q5 (a)

b. Write an expression tree for the expression A/B*C*D+E. Give the C function for inorder, preorder, postorder traversals and apply the traversal methods to the expression tree and give the result of traversals. (10 Marks)

c. What is a max heap? Construct the max heap for 7, 8, 3, 6, 9, 4, 10, 5 (05 Marks)

6 a. What is a binary search tree? Draw the binary search tree for the input:
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9
Give recursive search function to search an element in that tree. (10 Marks)

b. Construct the binary tree from the given traversals:
Preorder : A B D G C E H I F
Inorder : D G B A H E I C F (05 Marks)

c. What is a winner tree? Explain with suitable example winner tree for k= 8. (05 Marks)

7 a. What is Fibonacci Heap? Give example. Give the steps for election of node and decrease key of specified node in F-heap. (10 Marks)

b. Write short note on: i) Binomial heaps ii) Leftist trees. (10 Marks)

8 a. Starting with an empty AVL tree perform following sequence of intersion, MARCH, MAX, NOVEMBER, AUGUST, APRIL, JANUARY, DECEMBER, JULY. Draw the AVL tree following each insertion and state rotation type if any for insertion operation. (10 Marks)

b. Explain the red-black tree with example. (06 Marks)

c. Let h be the height of a red-black tree, let n be the number of internal nodes in the tree and r be the rank of the root then prove that
i) $h \leq 2r$ ii) $n \geq 2^r - 1$ (04 Marks)

* * * * *

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

USN | | | | | | | | | | 10[illegible]55

Third Semester B.E. Degree Examination, Dec.2014/Jan.2015
Data Structures with C

Time: 3 hrs. Max. [illegible]:100

Note: ***Answer any FIVE full questions, selecting atleast TWO questions from each part.***

PART – A

1 a. What are pointer variables? How to declare a pointer variable? (05 Marks)
b. What are the various memory allocation techniques? Explain how dynamic allocation is done using malloc( )? (10 Marks)
c. What is recursion? What are the various types of recursion? (05 Marks)

2 a. Define structure and union with suitable example. (08 Marks)
b. Write a C program with an appropriate structure definition and variable declaration to store information about an employee, using nested structures. Consider the following fields like: ENAME, EMPID, DOJ (Date, Month, Year) and Salary (Basic, DA, HRA). (12 Marks)

3 a. Define stack. Give the C implementation of push and pop functions. Include check for empty and full conditions of stack. (08 Marks)
b. Write an algorithm to convert infix to postfix expression and apply the same to convert the following expression from in fix to post fix:
i) (a * b) + c/d ii) (((a/[illegible] + (c [illegible] e)) – (a * c)). (12 Marks)

4 a. Define linked list. Write a C program to implement the insert and delete operation on queue using linked list. (10 Marks)
b. Explain the different types of linked list with diagram (10 Marks)

PART – B

5 a. Define the following:
i) Binary tree
ii) Complete binary tree
iii) Almost complete binary tree
iv) Binary search tree
v) Depth of a tree (10 Marks)
b. In brief describe any five application of trees. (05 Marks)
c. What is threaded binary tree? Explain right and left in threaded binary tree. (05 Marks)

6 a. Write C function for the following tree traversals:
i) inorder ii) preorder iii) postorder. (10 Marks)
Explain min and max heap with example. (10 Marks)

Implement Fibonacci heap. (10 Marks)
b. What is binomial heap? Explain the steps involved in the deletion of min element from a binomial heap. (10 Marks)

8 a. Explain AVL tree. (10 Marks)
b. Explain the red-black tree. Also, state its properties. (10 Marks)

* * * * *

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

USN | | | | | | | | | | |

10CS35

Third Semester B.E. Degree Examination, Dec. 2013/Jan. 2014
Data Structures with C

Time: 3 hrs. Max. Marks:100

Note: *1. Answer FIVE full questions, selecting atleast TWO questions from each part.*
2. Missing data, if any, may be suitably assumed.

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

PART – A

1 a. What is an ADT? Briefly explain the categories that classify the functions of a data type. Write an ADT for natural number. **(10 Marks)**

b. What is time complexity? Determine the time complexity of an iterative and recursive functions that adds n elements of the array using tabular method. **(10 Marks)**

2 a. Develop a structure to represent planets in the solar system. Each planet has fields for the planet's name, its distance from the sun in miles and the number of moons it has. Write a program to read the data for each planet and store. Also print the name of the planet that has less distance from the sun. **(08 Marks)**

b. What is a polynomial? What is the degree of the polynomial? Write a function to add two polynomials? **(08 Marks)**

c. For the given sparse matrix A and its transpose, give the triplet representation. A is the given sparse matrix, and B will be its transpose.

$$A = \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -28 & 0 & 0 & 0 \end{bmatrix}$$

Fig. Q2(c) Sparse Matrix **(04 Marks)**

3 a. Define stack. Implement push and pop functions for stack using arrays. **(08 Marks)**

b. Implement addq and deleteq functions for the circular queue. **(06 Marks)**

c. Write the postfix form of the following expressions
i) (a + b) * d + c/(f + a * d) + c
ii) ((a/(b – c + d)) * (e – a) * c)
iii) a/b – c + d * e – a * c. **(06 Marks)**

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

4 a. Write the different polynomial representation, with an example. (06 Marks)

b. For the given sparse matrix write the diagrammatic linked list representation? (06 Marks)

$$A = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 2 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$

Fig. Q4(b) : 5 × 4 Sparse Matrix

c. Define equivalence class. Write the linked list representation for the twelve polygons numbered 0 through 11 using the following pairs overlap?
0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0 (08 Marks)

PART – B

5 a. What is a tree? Explain
i) Root node
ii) Degree
iii) Siblings
iv) Depth of a tree and give examples. (08 Marks)

b. What is a binary tree? State its properties? How can it be represented using array and linked list, give example? (08 Marks)

c. Define a max heap? Write a C function to insert an item into max heap? (04 Marks)

6 a. Explain the following, with an example
i) Selection trees
ii) Forests and its traversals. (08 Marks)

b. Describe the binary search tree with an example. Write a recursive function to search for a key value in a binary search tree. (08 Marks)

c. Write the adjacency matrix and adjacency list for the following graph. (04 Marks)

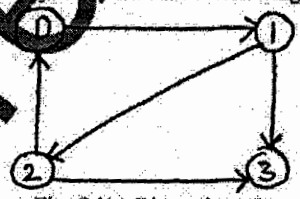


Fig. Q6(c) Directed graph

7 a. Briefly explain the following, with an example :
i) HBLT ii) WBLT. (08 Marks)

b. Write short notes on :
i) Priority queues
ii) Binomial heaps
iii) Fibonacci heaps. (12 Marks)

8 a. What is an AVL tree? Write the algorithm to insert an item into AVL tree. (08 Marks)

b. Explain the Red-Black-Tree with an example. State its properties. (08 Marks)

c. What is a splay tree? Briefly explain the different types of splay trees. (04 Marks)

* * * * *

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

USN | | | | | | | | | | 10CS35

Third Semester B.E. Degree Examination, June/July 2013
Data Structure with C

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting atleast TWO questions from each part.***

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

PART – A

1 a. Define pointer. With examples, explain pointer declaration, pointer initialization and use of the pointer in allocating a block of memory dynamically. (06 Marks)

b. Define recursion. Give two conditions to be followed for successful working of recursive program. Given recursive implementation of binary's search with proper comments. (06 Marks)

c. Define three asymptotic notations and give the asymptotic representation of function $3n + 2$ in all the three notations and prove the same from first principle method. (08 Marks)

2 a. What is a structure? Give three different ways of defining a structure and declaring variables and method of accessing members of structures using a student structure with roll number, name and marks in 3 subjects as members of that structure as example. (06 Marks)

b. Give ADT sparse matrix and show with a suitable example sparse matrix representation storing as triples. Give simple transpose function to transpose sparse matrix and give its complexity. (08 Marks)

c. How would you represent two sparse polynomials using array of structure and also write a function to add that polynomials and store the result in the same array. (06 Marks)

3 a. Give ADT stack and with necessary function, explain implementing stacks to hold records with different type of fields in each. (06 Marks)

b. Give the disadvantage of ordinary queue and how it is solved in circular queue. Explain the same. Explain with suitable example how would you implement circular queue using dynamically allocated arrays. (08 Marks)

c. Convert the infix expression a/b – c + d * e – a * c into postfix expression. Write a function to evaluate that postfix expression and trace that for given data a=6, b=3, c=1, d=2, e=4. (06 Marks)

4 a. Give the node structure to create a linked list of integers and write C functions to perform the following :
i) Create a three –node list with data 10, 20 and 30
ii) Insert a node with data value 15 in between the nodes having data values 10 and 20
iii) Delete the node which is followed by a node whose data value is 20
iv) Display the resulting singly linked list. (08 Marks)

b. With node structure show how would you store the polynomials in linked lists? Write C function for adding two polynomials represented as circular lists. (06 Marks)

c. Write a note on :
i) Linked representation of sparse matrix
ii) Doubly linked list. (06 Marks)

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

PART – B

5 a. Define a binary tree and with example show array representation and linked presentation of binary tree. (06 Marks)

b. Write an expression tree for an expression A/B + C * D + E. Give the algorithm for inorder, postorder and preorder traversals and apply that traversal method to the expression tree and give the result of transversals. (08 Marks)

c. Define a Max Heap. Explain clearly inserting an element that has value 21 in the heap shown in Fig. Q5(c), given below and show the resulting heap. (06 Marks)

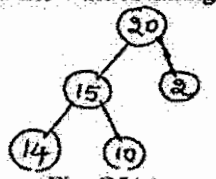


Fig. Q5(c)

6 a. Define a binary search tree and construct a binary search tree. With elements {22, 28, 20, 25, 22, 15, 18 10, 14}. Give recursive search algorithm to search an element in that tree. (06 Marks)

b. With is a winner tree? Explain with suitable example a winner tree for k = 8. (06 Marks)

c. Construct a binary tree having the following sequences.
i) Preorder sequence : ABCDEFGHI
ii) Inorder sequence : BCAEDGHFI
Show the steps if constructing binary tree in the above example. (03 Marks)

d. Give the adjaunccy matrix and adjacinty lists representation for the graph shown in Fig. Q6(d). (05 Marks)

Fig. Q6(d)

7 a. Define the following :
i) Single ended priority queues
ii) Double ended priority queues
iii) Height –based leftist trees
iv) Weight – based leftist trees
v) A binomial tree
vi) Extended binary tree. (06 Marks)

b. With suitable example, explain leftist trees and give structure of nodes. (06 Marks)

c. What is Fibonacci heap? Give suitable example and give the steps for deletion of node and decrease key of specified node in F – heap. (08 Marks)

8 a. What is an AVL tree? Stating with an empty AVL tree perform the following sequence of insertions, MARCH, MAY NOVEMBER, AUGUST, APRIAL, JANUARY, DECEMBER, JULY, FEBRYARY, DRAW the AVL tree following each insertion and state rotation type if any for any insert operation. (10 Marks)

b. Define RED – BLACK trees and give its additional properties starting with an empty red-black tree insert the following keys in the given order {50, 10, 80, 90, 70, 60, 65, 62}, giving color changing and rotation instances. (10 Marks)

* * * * *

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green

DOWNLOAD THIS FREE AT www.vturesource.com

USN | | | | | | | | | | |

10CS/IS35

Third Semester B.E. Degree Examination, January 2013
Data Structures with C

Time: 3 hrs. Max. Marks:100

Note: ***Answer FIVE full questions, selecting at least TWO questions from each part.***

PART – A

1 a. What are pointer variables? How to declare a pointer variable? **(05 Marks)**

b. What are the various memory allocation techniques? Explain how memory can be dynamically allocated using malloc ( )? **(10 Marks)**

c. What is recursion? What are the various types of recursion? **(05 Marks)**

2 a. What is the difference between int *a and int a[5] and int *[5]? **(06 Marks)**

b. What is a structure? How to declare and initialize a structure? **(06 Marks)**

c. Write a program in C o read a sparse matrix of integer values and search this matrix for an element specified by the user. **(08 Marks)**

3 a. Define stack. List the operations on stack. **(08 Marks)**

b. Obtain the postfix and prefix expression for $(((A+(B-C)*D)\hat{}E)+F)$. **(06 Marks)**

c. What is system stack? How the control is transferred to or from the function with the help of activation record? **(06 Marks)**

4 a. What is a linked list? Explain the different types of linked list with diagram. **(10 Marks)**

b. Write a function to insert a node at front and rear end in a circular linked list. Write down sequence of steps to be followed. **(10 Marks)**

PART – B

5 a. What is a tree? Explain: i) root node, ii) child, iii) siblings, iv) ancestors using structure representation. **(06 Marks)**

b. What is a binary tree? How it is represented using array and linklist? **(10 Marks)**

c. What is a heap? Explain the different types of heap? **(04 Marks)**

6 a. What is a binary search tree? Draw the binary search tree for the following input: 14, 5, 6, 2, 18, 20, 16, 18, -1, 21. **(10 Marks)**

b. What is a forest? Explain the different method of traversing a tree with following tree:

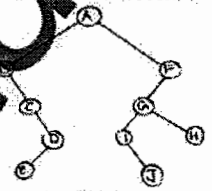


Fig.Q6(b) **(10 Marks)**

7 a. What is priority queue? Explain the various types of priority queues. **(08 Marks)**

b. Write short notes on: i) Binomial heaps, ii) Fibonacci heap. **(06 Marks)**

c. What is leftist tree? Explain different types of leftist trees. **(06 Marks)**

8 a. What is an AVL tree? Write the algorithm to insert an item in to AVL tree. **(08 Marks)**

b. Write short notes on: i) Red-Black tree, ii) Splay trees. **(06 Marks)**

Explain the different types of rotations of an AVL tree. **(06 Marks)**

* * * * *

www.vturesource.com

ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS

A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

USN | | | | | | | | | | |

10CS35

Third Semester B.E. Degree Examination, June 2012
Data Structures with C

Time: 3 hrs. Max. Marks:100

Note: *Answer FIVE full questions, selecting at least TWO questions from each part.*

Important Note : 1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg, 42+8 = 50, will be treated as malpractice.

PART – A

1 a. Define a pointer. Write a C function to swap two numbers using pointers. (05 Marks)
b. Explain the functions supported by C to carryout dynamic memory allocation. (05 Marks)
c. Explain performance analysis and performance measurement. (10 Marks)

2 a. Define structure and union with suitable example. (08 Marks)
b. Write a C program with an appropriate structure definition and variable declaration to store information about an employee using nested structures. Consider the following fields like Ename, Empid, DOJ (Date, Month, Year) and salary (Basic, DA, HRA). (12 Marks)

3 a. Write a C-program to implement the two primitive operations on stack using dynamic memory allocation. (08 Marks)
b. Write an algorithm to convert infix to postfix expression and apply the same to convert the following expression from infix to postfix :
i) (a * b) +c/d
ii) (((a/b)-c) + (d * e)) – (a * c) (12 Marks)

4 a. Define linked list. Write a C program to implement the insert and delete operation on a queue using linked list. (10 Marks)
b. Write a C-function to add two polynomials using linked list representation. Explain with suitable example. (10 Marks)

PART – B

5 a. Define binary tree. For the given tree find the following :
i) Siblings
ii) Leaf nodes
iii) Non-leaf nodes
iv) Ancestors
v) Level of tree (08 Marks)

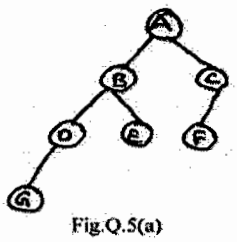


Fig.Q.5(a)

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Save trees use e-Question Paper Go green
DOWNLOAD THIS FREE AT www.vturesource.com

b. Write the C-routines to traverse the given tree using i) inorder ; ii) pre order ; iii) post order. (12 Marks)

6 a. Define ADT of binary search tree. Write the iterative search function and recursive search function of BST. (08 Marks)

b. Construct the binary tree for the given expressions :
   i) Pre order : / + * 1 $ 2 3 4 5
      A B D G C E H I F
   ii) In order : 1 + 2 * 3 $ 4 – 5
      D G B A H E I C F. (08 Marks)

c. Define furest with example. (04 Marks)

7 a. Define leftlist trees. Explain varieties of leftlist trees. (08 Marks)

b. Write short notes on :
   i) Priority queues
   ii) Binomial heaps
   iii) Priority heaps
   iv) Fibonacci heaps. (12 Marks)

8 a. Define AVL trees. Write a C-routine for
   i) Inserting into an AVL tree
   ii) LL and LR rotation. (10 Marks)

b. Explain the following with example :
   i) Red-black trees
   ii) Splay trees. (10 Marks)

* * * *

www.vturesource.com
ALL BRANCHES | ALL SEMESTERS | NOTES | QUESTON PAPERS | LAB MANUALS
A Vturesource Go Green initiative

Question Bank

Data Structures and Applications

15CS33

QUESTION BANK

MODULE 1

Introduction to Data Structures

1. What is a pointer variable? How pointers are declared & initialized in C? Can we have multiple pointer to a variable? Explain Lvalue and Rvalue expression.(june/jul 2014) (Dec 2014/Jan 2015)

2. Give atleast 2 differences between : i. Static memory allocation and dynamic memory allocation.

ii. Malloc() and calloc(). (Dec 2014/Jan 2015)(Dec/Jan-16)

3. What is dangling pointer reference & how to avoid it? (june/jul 2014)

4. With suitable example, explain dynamic memory allocation for 2-D arrays. (june/jul 2014)(Dec/Jan16)(jun/Jul16)

5. Define a structure for the employee with the following fields: Emp_Id(integer), Emp_Name(string), Emp_Dept(string) & Emp_age(integer) Empid, DOJ (date,month,year) and salary (Basic, DA,HRA). Write the following functions to process the employee data: (i) Function to read an employee record. (ii) Function to print an employee record. (june/jul 2014) (dec 2014/jan 2015)

6. What is a structure ? Give three different ways of defining structure & declaring variables & method of accessing members of structures using student structure with roll number, names & marks in 3 subjects as members of that structure as example. (dec 2014/jan 2015) (june/jul 2014)

7. What is the purpose of free()? With an example explain the problem when it is not used. (Dec-Jan-16)(Jun-Jul 16)

8. Explain how memory can be allocated using realloc()?(Dec/Jan -16)(Jun/Jul-16)

9. Describe unions in c? How is it different from structures? (Jun/Jul 16)

MODULE 2

Linear Data Structures and their Sequential Storage Representation

1. Define Recursion. What are the various types of recursion? Give two conditions to be followed for successive working of recursive program. Give recursive implementation of binary's search with proper comments. (june/july 2015) (Dec 2014/Jan 2015)(Dec/Jan-16)(Jun/Jul 16)

2. Estimate the space complexity of a recursive function for summing a list of numbers. (june/jul 2014)

3. Write an algorithm to convert a valid infix expression to a postfix expression. Also evaluate the following suffix expression for the values: A=1 B=2 C=3.

AB+C-BA+C$- and convert i) a*(b+c)*d ii) (a+b)*d+e/(f+a*d)+c

iii) ((a/(b-c+d))*(e-a)*c) iv) a/b-c+d*e-a*c iv) (a*b) +c/d

v) ((a/b)c)+(d*e)) (a*c) to postfix.

4. What is the advantage of circular queue over ordinary queue? Mention any 2 applications of queues. Write an algorithm CQINSERT for static implementation of circular queue. (june/jul 2014) (Dec/Jan 16)

5. Define stack. Implement push & pop functions for stack using arrays. (Dec 2014/Jan 2015) (June/Jul 2014)(Dec/Jan 16)

MODULE 3

Linear Data Structures and their Linked Storage Representation

1. List out any two applications of linked list and any two advantages of doubly linked list over singly linked list. (june/jul 2014) (Jun-Jul 16)(Dec/Jan 16)

2. Write short note on circular lists. Write a function to insert a node at front and rear end in a circular linked list. Write down sequence of steps to be followed. (june/jul 2015)

3. Write the following functions for singly linked list: i) Reverse the list ii)Concatenate two lists. (june/jul 2014) (june/jul 2015)(Jun/Jul 16)

4. Write the node structure for linked representation of polynomial. Explain the algorithm to add two polynomial represented using linked lists. (june/jul 2014) (june/july 2013) (Dec/Jan 16)

5. What is a linked list? Explain the different types of linked list with diagram. Write C program to implement the insert and delete operation on a queue using linked list. (dec 2014/jan 2015)(Dec/Jan 16)

MODULE 4

Nonlinear Data Structures

1. Define the tree & the following

i) Binary tree

ii) Complete binary tree

iii) Almost complete binary tree

iv) Binary search tree

v) Depth of a tree

vi) Degree of a binary tree

vii) Level of a binary tree

viii) Sibling

ix) Root node

x) Child

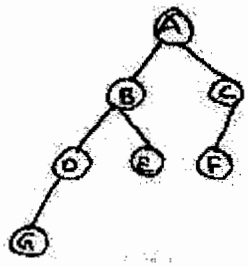
xi) Ancestors (Dec 2014/Jan 2015) (June/Jul 2014) (Jun/Jul 16)

2. What is threaded binary tree? Explain right in and left in threaded binary trees. Advantages of TBT over binary tree. (june/jul 2014) (Dec 2014/Jan 2015)

3. What is a heap? Explain the different types of heap? (june/july 2015)

4. Define binary trees. For the given tree find the following : (June 2013) (june/july 2015)

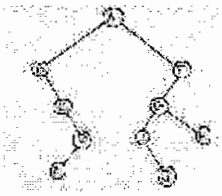
i. siblings
ii. leaf nodes
iii. non – leaf nodes
iv. ancestors
v. level of trees



MODULE 5

Graph

1. Explain the following with an example: i) forest & its traversals. Explain the different method of traversing a tree with following tree



ii) graph iii) winner tree .iv) Selelction trees (june/jul 2015).

2. Describe the binary search tree with an example. Write a iterative & recursive function to search for a key value in a binary search tree. Define ADT of binary search tree. Write BST for the elements { 22,28,20,25,22,15,18,10,14} {14, 5, 6, 2, 18, 20, 16, 18, -1, 21} (june/jul 2014).

3. Explain selection trees, with suitable example. (june/jul 2014)

4. What is a forest. With suitable example illustrate how you transform a forest into a binary tree. (june/jul 2014)

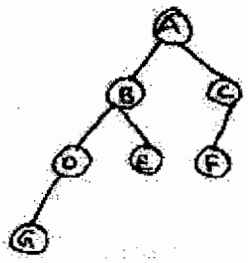
5. What is a winner tree ? Explain with suitable example a winner tree for k=8. (june/jul 2015).

6. Construct a binary tree having the following sequences.(dec 2014/jan 2015) (june/july 2014)

i) Preorder sequence:

ii) Inorder sequence:

1) pre order:



Internal Questions

&

Solution Scheme

Data Structures and Applications

15CS33

USN | | | | | | | | | | |

RNS Institute of Technology
Department of CSE
III Semester - I Test – September-2016
Data Structures and Applications (15CS33)

Duration: 90 mins. **Max Marks: 40** **Time: 9-10.30 am** **Date: 09/09/16**

NOTE: Answer *FIVE* full questions.
Don't write anything on question paper other than USN.

Q.No.			Marks	Bloom's Learning Levels L1-L6
1.	a)	Define Data Structure. Give its classification with example for each.	2+4	L1
	b)	Mention the operations that can be performed on Data Structures.	2	L1
		OR		
	a)	Define Pointers in C. Elaborate the usage of pointers to represent 2D array.	2+2	L1, L5
	b)	With a C program demonstrate the usage of pointers for swapping integer values.	4	L3
2	a)	Design, Develop and Implement a menu driven Program in C for the following Array operations a. Creating an Array of N Integer Elements b. Display of Array Elements with Suitable Headings c. Inserting an Element (ELEM) at a given valid Position (POS) d. Deleting an Element at a given valid Position(POS) Support the program with functions for each of the above operations.	2 x 4	L3
		OR		
	a)	Give the syntax and example code snapshot for the following functions: (i) malloc() (ii) calloc() (iii) realloc() (iv) free(). Explain their purposes.	2 x 4	L1
3.	a)	Define a Structure, give its syntax and explain it with an example.	2+2	L1
	b)	Define a Union, give its syntax. Differentiate between unions and structures.	2+2	L2
		OR		
	a)	How are polynomials represented using arrays in C. Discuss in brief.	3	L1
	b)	Write a function to add two polynomials.	5	L3
4	a)	Define Stack in Data Structures.	2	L1
	b)	Write a C program to implement the operations of stack on integer values.	6	L3
		OR		
	a)	Recommend any two applications to which stacks may be used.	2	L6
	b)	Write a C program to implement dynamic stack operations using realloc().	6	L3

5.	a)	Convert the following Infix expression to postfix expression. i) A+(B*(C-D)/E) ii) A*B+C*D	2+2	L3
	b)	Write a C function to convert Infix expression to Postfix expression.	4	L3
		OR		
	a)	List and explain any four operations on strings.	2	L1
	b)	Design a pattern matching algorithm to match a pattern in the text given. Considering the Text as : TO BE OR NOT TO BE and pattern as: OR NOT, show the pattern matching steps by tracing the algorithm.	6	L3

A * B + C * D

AB * CD * +

A + (B * (C - D) / E)

ABCD - * E / +