# GUI PACKAGE ON DATA STRUCTURES

## A THESIS

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

**MASTER OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

**TO**

**THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY**
**(DEEMED UNIVERSITY)**
**PATIALA - 147 001**


**SUPERVISOR**

**Dr. RENU VIG**
**ASSISTANT PROFESSOR**
**DEPARTMENT OF COMPUTER SCIENCE**
**TTTI,CHANDIGARH**

**SUBMITTED BY**

GUMUDAVALLY MOHAN BABU
REGD. NO. ME(R)-129/99(1)


DEPARTMENT OF COMPUTER SCIENCE
TECHNICAL TEACHERS' TRAINING INSTITUTE
SECTOR 26, CHANDIGARH-160019
JANUARY-2001

# CERTIFICATE

Certified that this thesis entitled "GUI Package On Data Structures" being submitted by Gumudavally Mohan Babu in the partial fulfilment of the requirements, for the award of degree of Master of Engineering (Computer Science and Engineering) of Thapar Institute of Engineering and Technology, Patiala is a record of candidate's own work carried under our supervision and guidance.

This thesis has not been earlier submitted to any other university or institution for the award of any degree.

SUPERVISOR

**(Dr.Renu Vig)**
Assistant Professor,
Department of Computer Science,
T.T.T.I.,Chandigarh.

**(Prof. V. P. Puri)**
Head,
Department of Computer Science,
T.T.T.I.,Chandigarh.

# ACKNOWLEDGEMENTS

I am grateful to **Dr. S. Krishnamurthy**, Principal, TTTI, Chandigarh for providing me the opportunity to carryout this thesis.

I express my sincere gratitude to **Dr. Renu Vig**, Assistant Professor, Department of Computer Science, TTTI, Chandigarh, for her valuable guidance, proper advice and constant encouragement during the course of my work on this thesis.

I am grateful to **Prof. Vijay P. Puri**, Head, Department of Computer Science, TTTI, Chandigarh, for providing day and night computer facilities in the department, which helped me to complete my thesis within the stipulated time.

I am thankful to all the faculty and staff members of the Computer Science Department for their direct and indirect help and cooperation.

Sincere thanks to the Department of Technical Education and Training, Govt. of Andhra Pradesh for sponsoring me for this course.

(MOHAN BABU)

# ABSTRACT

The study of data structures is an essential part of virtually everybody in computer science field. An apprentice carpenter may want only a saw, but a master craftsman employs many precision tools. Computer programming like wise requires sophisticated tools to cope with the complexity of real applications ,and only practice with these tools will build skill in their use. Hence this structured problem solving is the process of data abstraction and structuring,and comparative study of algorithms as fundamental tools of program design.

The main structuring techniques are linear lists, trees and binary trees. Distinction between sequential and linked allocation of storage , the number and types of pointers and the modes of data organization are all important in planning software.

The goal of programming is the construction of programs that are clear ,complete and functional. The process of data specification and abstraction ,similarly ,comes before the selection of data structures and their implementation.

# CONTENTS

# CHAPTER - 1 Data Structures

**Introduction :-**

The problem to be solved by a computer pose several interrelated problems: These problems may include development of algorithms, arrangement of data for input and output etc. Before developing a method for solution to a problem, one has to organize the input data very carefully. Apart from organizing the input data, the need of planning the output cannot be ignored. These objectives can be achieved only if a clear and thorough understanding of the relationship between those data items, that are relevant to the solution of the problem, is known. In other words, a thorough understanding about the data is must, otherwise there is every likelihood of getting erroneous results irrespective of processing of this data either manually or by a computer.

In general, data consists of elementary items such as integers, fractional numbers, bits or characters. Before attempting to process this data, it has to be structurally organized . The manner in which elementary data items are organized leads to ***"data structures"***. Data structures describes the representation of data and the operations that can be performed on the data .

A computer is a machine that manipulates information. The study of computer science includes the study of how information is organized in a computer, how it can be manipulated, and how it can be utilized. Thus it is exceedingly important for a student of computer science to understand the concepts of information organization and manipulation in order to continue study of the field .

A desirable set of goals for a course in data structures is :

* To introduce the student to those aspects of data structures which are required in subsequent computer science courses.

* To motivate the student by illustrating the key concepts with various examples in computer science.

* To increase the students intuitive understanding of concepts.

Finally, I would like to distinguish between data and information. The term data refers to the raw data or raw material whereas the information refers to the processed data .

Ex :- If we take the marks obtained in a subject by different students of a class as data then finding the students list who have got the marks greater than 75 is called as an information.

**Definition :-**

The data may be organized in many different ways; the logical / mathematical model of a particular organization of data is called as a ***Data Structure.*** The choice of a particular data model depends on two factors. It must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

The greatest room for variability in algorithm design is generally in the way in which the data of the program are stored as :

* How they are arranged in relation to each other,

* Which data are kept in memory when necessary,

* Which are calculated when needed,

* Which are kept in files, and how are the files arranged.

To present several elegant, yet fundamentally simple ideas for the organization of data, and several powerful algorithms for important tasks within data processing such as Sorting and Searching.

## 1.1 Classification :-

Data structures can be classified based on their nature and construction. The classification involves the following types : -

1. Fundamental / Elementary data structures or Linear data structures.
2. Complex data structures or Non-linear data structures.

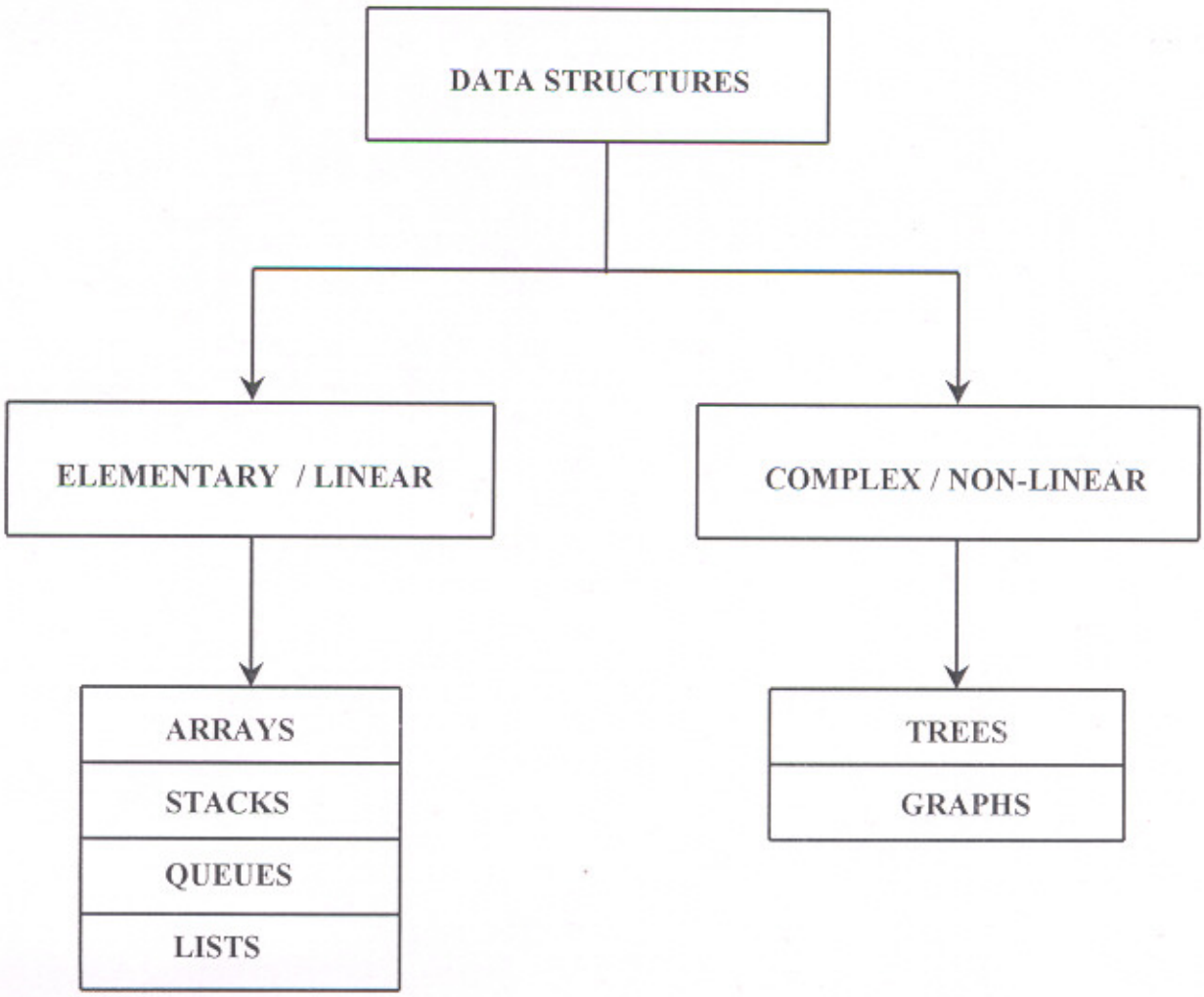



**Fig 1.1 :- Classification of data structures**

As shown in the Fig 1.1, Fundamental / elementary data structures are primitive in nature and are the building blocks used for the complex structure construction. These data structures

encounter more frequently in the usage. Arrays, stacks, queues, lists fall under this category. These are also called as linear data structures since their elements form a sequence.

Complex data structures are the data structures which are complex in nature and are constructed using the elementary data structures. For some operations on data, the elementary structures are not useful as well as efficient and in such cases, complex data structures can be preferred. Graphs, trees fall under this category. These are also called as non-linear data structures since their elements represents a hierarchical relationship between them.

## 1.2 Linear Data structures :-

### Arrays :-

An array is a list of a finite number 'n' of *homogeneous* data elements (i.e., data elements of the same type) such that :

1. The elements of the array are referenced respectively by an *index set* consisting of 'n' consecutive numbers.
2. The elements of the array are stored respectively in successive memory locations .

### Sparse Matrices :-

Matrices with a relatively high proportion of '0' entries are called sparse matrices. i.e. the number of non-zero elements are less. Two general types of n-square sparse matrices which occur in various applications are

1. Triangular matrix
2. Tridiagonal matrix.

The natural method of representing matrices in memory as 2-dimensional arrays may not be suitable for sparse matrices. i.e. one may save space by storing only those entries which may be non-zero.

Ex :-

$$A = \begin{bmatrix} a11 & 0 & 0 \\ a21 & a22 & 0 \\ a31. & a32 & a33 \end{bmatrix}$$

In the above triangular matrix it would be wasteful to store those entries above the main diagonal of A. Hence we store only the other entries of A in a linear array B as indicated by B[1] = a11, B[2] = a21, B[3] = a22 and so on. So, B will contain only elements which is about half the elements as a 2-dimentional n*n array. The final representation of a sparse matrix will be as follows :

1. Number of rows as in the original matrix,
2. Number of columns as in the original matrix,
3. Number of non-zero elements.

**Stacks :-**

A stack is data structure in which items may be added / removed only at one end of the stack called the *top* of the stack. These are also called as Last In First Out "*LIFO*" lists. The operations performed on a stack are *PUSH, POP*. PUSH - To insert an element into a stack. POP - To delete an element from the stack . Other names are "*Piles*", "*Push down lists*".

Real life examples are :-

* Stack of dishes,
* Stack of pennies,
* Stack of folded towels,
* Stack of coins,
* Stack of trays in busy cafeteria,
* Stack of books,

* Railway shunting system.

*Overflow* and *Underflow* are the terms associated with the *Push* and *Pop* respectively.

The simplest way to represent a stack is by using a 1-dimensional array say stack(1:n). Where 'n' is the number of maximum allowable entries. The first or bottom element of the stack will be stored at stack(1), the 2nd at stack(2) and the $i^{th}$ at the stack(i). Associated with the array will be available typically called *TOP* , which points to the top elements in the stack. To test if the stack is empty we ask "if top = 0'. If not, the top most element is at stack(top).

**Queues :-**

Queue is a linear list of elements in which deletions can take place only at one end, called the *FRONT* and insertions can take place only at the other end called the *REAR*. These are also called as First In First Out "*FIFO*" lists. i.e. the order in which the elements enter a queue is the order in which they leave. Queues abound in everyday life.

Ex:- 1. In a time-sharing system in which the programs with the same priority form a queue while waiting to be executed.

2. Automobiles waiting to pass through an intersection form a queue.

3. In Mathematical queueing theory.

4. Computer networks.

Real life examples :-

1. Queue lines waiting for buses.

2. Queue lines waiting to be served at post office.

3. Queue lines waiting to be served at banks.

4. Queue at cinema theatre.

5. Queue at ration shops.

**Circular Queues :-**

An efficient queue representation is obtained by taking an array declared as Q( 0 : n-1 ) and treating it as if it was a circular.

Q[1] comes after Q[N] in the array. We insert the item into the Q by assigning ITEM to Q[1]. Instead of increasing REAR to N +1 we reset REAR = 1 and then

Q[REAR] = ITEM

Similarly, if FRONT = N and an element of Q is deleted then we reset FRONT = 1 instead of increasing FRONT to N+1.

If the Q contains only one element then

FRONT = REAR <> NULL

If that element is deleted then

FRONT = NULL and REAR = NULL to indicate that the Q is empty.

**Linked Lists :-**

Linked list is a data structure which supports the logical adjacency of elements rather than the physical adjacency. This logical adjacency is maintained by using pointers. There are two types of linked lists. 1.Singly linked lists, 2. Doubly linked lists.

**Singly linked lists :-**

A Singly linked list is a linear collection of data elements called *nodes*, where the linear order is given by means of pointers. This is also called as *One way linked list.* i.e. each node is divided into 2 parts . The first part contains the information of the element, and the second part called the link field or next pointer field contains the address of the next node in the list. The pointer of the last node contains the special value called the *NULL*

pointer which is any invalid address. This will denote the end of the list. A special case is the list that has no nodes, such a list is called a *NULL* list.

**Doubly Linked lists :-**

a Doubly linked list is a linear collection of data elements called nodes where each node N is divided into 3 parts,

1. An information field "*INFO*" which contains data of N,
2. A pointer field "*FORW*" which contains the location of the next node in the list,
3. A pointer field "*BACK*" which contains the location of the preceding node in the list.

This is used for traversing a list in 2 directions. Hence it is called a 2-way list.

i.e. from the beginning of the list to end ( forward direction ) and

from the end of the list to the beginning ( backward direction )

One has immediate access to both the next node and the preceding node in the list. Using the variable FIRST and the pointer field FORW we can traverse a 2-way list in the forward direction and using the variable LAST and the list pointer field BACK we can traverse the list in backward direction. Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list

Suppose LOCA, LOCB are the locations respectively of nodes A & B in a 2-way list. Then the way that the pointers FORW & BACK are defined gives us the following.

FORW[LOCA] = LOCB if and only if BACK[LOCB] = LOCA

i.e. The statement that node B follows node A is equivalent to the statement that the node A precedes node B. To implement in memory requires 2 list pointer variables FORW, BACK. Generally speaking, storing data as a 2-way list which requires extra space for the backward pointers and extra time to change the added pointers as compared to a 1-way list.

Two way lists may be maintained in memory by means of linear arrays in the same way as one-way lists except that now we require two pointer arrays, FORW and BACK, instead of one pointer array LINK, and we require two list pointer variables, FIRST and LAST, instead of one list pointer variable START. On the other hand, the list of available space in the arrays will still be maintained as a one-way list using FORW as the pointer field.

**Operations on linked lists :-**

Different types of operations can be performed on linked lists are

1. Sorting
2. Searching
3. Reversing
4. Maximum
5. Minimum

## 1.2.1 An Application :-

## Recursion :-

It is defined as a function which *calls* itself repeatedly for solving a particular problem.

Ex 1:-The factorial function defined recursively as

$$n! = 1, \text{ if } n=0$$

$$n! = n * (n-1)!, \text{ if } n>0$$

Ex 2:-Multiplication of natural numbers is defined recursively as

$$a * b = a, \text{ if } b=1$$

$$a * b = a * (b-1) + a, \text{ if } b>1$$

One important requirement of a recursive algorithm is that it should not generate an infinite sequence of *calls* on itself i.e. it should terminate after required number of *calls* to itself.

**The Towers of Hanoi problem :-**

As shown in the Fig 3.2, there are three pegs A, B and C. Three disks of differing diameters are placed on peg A so that a larger disk is always below a smaller disk. The objective is to move the three disks to peg C using peg B as an auxiliary. Only the top disk on any peg may be moved to any other peg, and a larger disk may never rest on a smaller one.

The solution is shown in Fig 3.3. To move 'n' disks from A to C using B as an auxiliary,

1. If n = 1, then move the single disk from A to C and stop.
2. Move the top (n-1) disks from A to B using C as auxiliary.
3. Move the remaining disk from A to C.
4. Move the (n-1) disks from B to C, using A as auxiliary.

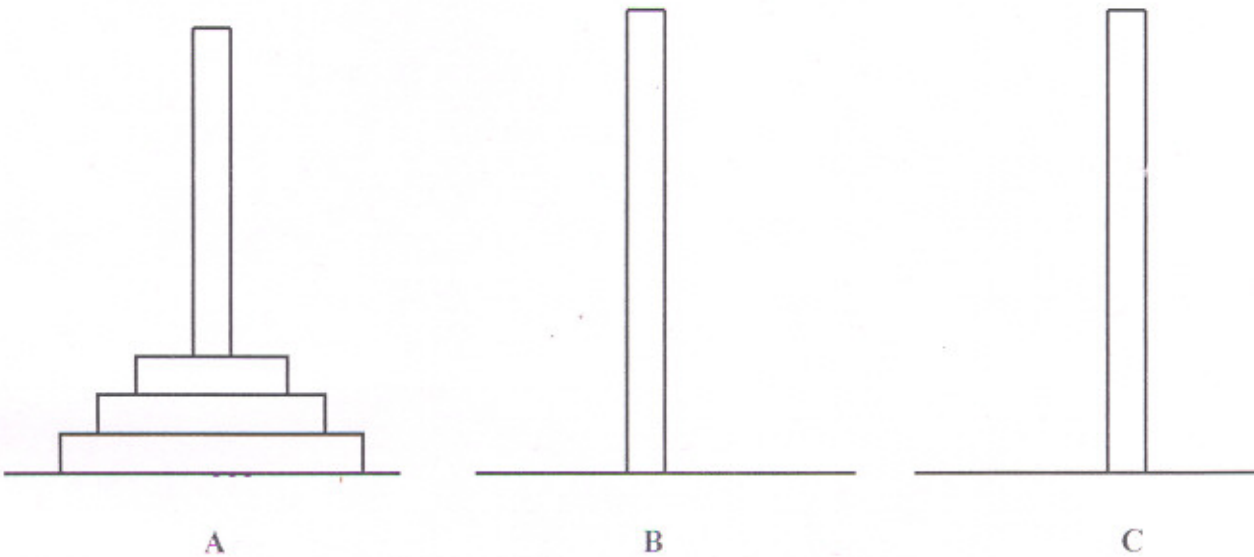



**Fig 3.2 :- Initial set up of Towers of Hanoi problem**

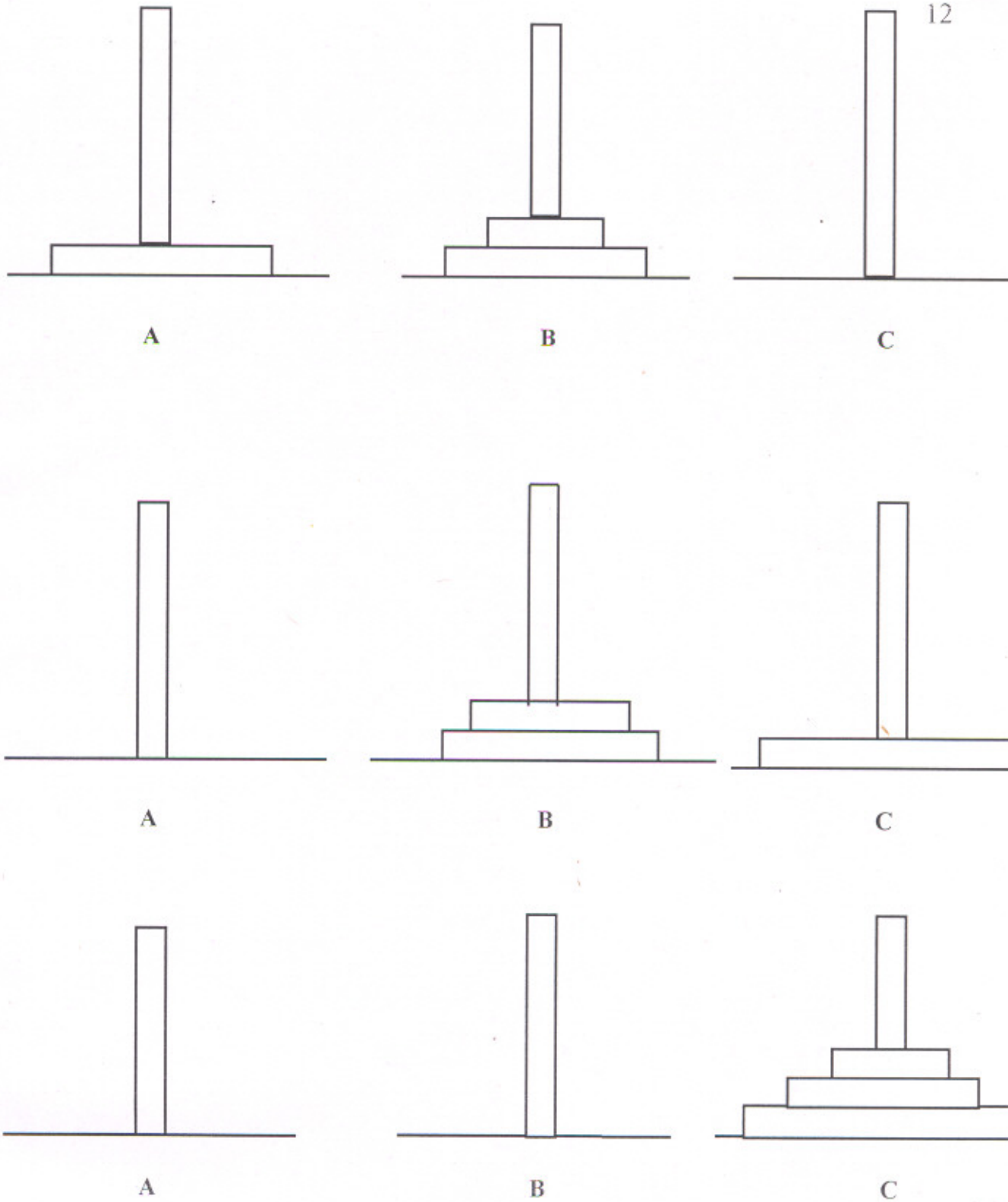



Fig 3.3:- Recursive solution to Towers of Hanoi problem

## 1.3 Non-Linear Data Structures :-

### Trees :-

A general tree is defined to be a non-empty finite set T of elements called nodes such that 1.T contains a distinguished element R called the root of T and 2. The remaining elements of T form an ordered collection of zero or more disjoint trees T1,T2,T3…$T_n$. The trees T1, T2, T3 ,…,$T_n$ are called sub trees of the root R and the roots of these sub trees are called successors of R.

The number of sub trees of a node is called its "degree". Nodes that have degree zero are called "leaf" or "terminal nodes". The other nodes are "non-terminal" nodes. Children of the same parent are called as "siblings". The "ancestors" of a node are all the nodes along the path from the root to that node The "height/depth" of a tree is defined to be the maximum level of any node in the tree. A forest is a set of n >=0 disjoint trees.

### Binary trees :-

A binary tree T is defined as a finite set of elements such that

1. T is empty or

2. T contains a distinguished node R called the root of T and the remaining nodes of T form an ordered pair of disjoint binary trees T1,T2. T1,T2 are called as left and right sub trees respectively.

The binary tree of depth k which has exactly $2^k$ -1 nodes is called a complete binary tree. The sequential representation of is good for a complete binary tree. For skewed trees there is a waste of space. The depth of complete binary tree T with n nodes is given by

$$D_n = [\log_2 n + 1]$$

If K is a node in binary tree then the left & right children of K are 2K and 2K + 1 respectively. If the two binary trees have the same structure then they are called " similar". If the two binary

trees have the same contents at the corresponding nodes then those are called as " copies ".

**Graphs :-**

A graph G consists of two things.

1. A set of V elements called vertices and
2. A set of E edges such that each edge e in E is identified with a unique pair [u,v] of nodes in V denoted by e = [u,v]. Hence G = (V, E ).

Other names of a graph are linear graph, 1-complex, 1-dimensional complex.

Other names of a vertex are node, junction, point, zero cell, zero simple.

Other names of an edge are branch, line, element, 1-cell, arc, 1-simplex.

An edge having the same vertex as both of its end vertices is called a "self loop". More than one edge associated with a given pair of vertices are called as "parallel edges". A graph that has neither self loops nor parallel edges is called a "simple graph". A graph is said to be "complete" if there is an edge between every pair of vertices. A graph with finite number of vertices as well as finite number of edges is called a " finite graph" otherwise it s an "infinite graph". The number of edges incidenting on a vertex ' v ' with self loops counted twice is called its "degree". A graph in which all vertices are of equal degree is called a "regular graph". A vertex of degree one is called a "pendant vertex". A graph without any edges is called a "null graph". A vertex with degree zero is called an "isolated vertex". The number of vertices of odd-degree in a graph is always even.

## 1.3.1 An Application :-

**Backtracking :-**

Backtracking is a problem solving method. This method is useful in situations where many possibilities may first appear, but few will withstand after subjection to certain tests.

**Definition :-**

Backtracking is a method which attempts to complete a search for a solution to a problem by means of constructing partial solutions, always ensuring that the partial solution remain consistent with the requirements of the problem.

This method attempts to extend a partial solution towards completion of the problem, but when an inconsistency with the requirements of the problem occurs, the method backs up or backtracks by removing the most recently constructed part of the solution and trying for another possibility.

Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some conditions can be solved by using backtracking method.

**Eight queens problem :-**

A classic combinatorial problem is to place 8 queens on an 8x8 chess board so that no two of them are on the same row, column or diagonal. Let us number the row and columns of the chess board 1 through 8 . The queens may also be numbered 1 through 8. Since each queen must be on a different row, we can without loss of generality assume queen 'i' is to be placed on row 'i'. All solutions to the 8-queens problem can therefore be represented as 8-tuples (x1,x2,...,x8) where $x_i$ is the column on which queen 'i' is placed. The explicit constraints using this formulation are Si = { 1, 2, ... ,8 },1<= i <= n .Therefore the solution space consists of $8^8$ 8-tuples. The implicit constraints for this problem are that no two $x_i$ can be the same and no two queens can be on the same diagonal. The first of these two constraints implies that all solutions are permutations of 8-tuple (1,2,...,8). This realization reduces the size of the solution space from $8^8$-tuples to 8!-tuples. the second constraint can be formulated as follows. If we imagine the squares of the chess board being numbered as the indices of the 2-dimensional array then we

observe that for every element on the same diagonal which runs from the upper left to the lower right, each element has the same "row - column" value. Also every element on the same diagonal which goes from the upper right to the lower left has the same "row + column" value. Suppose two queens are placed at positions (i,j) and (k,l) then by the above description they are on the same diagonal only if

$$i - j = k - l \quad \ldots\ldots I \quad \text{or} \quad i + j = k + l \quad \ldots\ldots II$$

$$\text{from I}, \quad j - l = i - k$$

$$\text{from II}, \quad j - l = k - I$$

Therefore two queens lie on the same diagonal if and only if $| j - l | = | i - k |$.

# CHAPTER-2 Sorting and Searching Algorithms

## 2.1 Sorting Algorithms:-

The operation of arranging data in some given order such as increasing or decreasing with numerical / character data. This is frequently applied to a file of records with a given primary key. There are 2 types of sorting. 1. Internal sorting 2. External sorting.

1.Internal sorting :-

If the records to be sorted are in main memory then that type of sorting is called as an internal sorting.

2.External sorting :-

If the records to be sorted are in auxiliary storage then that type of sorting is called as an external sorting.

Stable sorting :-

A sorting is said to be stable if and only if the original list itself is in a sorted order before applying any of the sorting method.

**Linear Sort :-**

Suppose an array A with 'n' elements in memory as A[1],A[2],A[3],...,A[n]. First find the smallest element in the list and put it in the first position . Then find the $2^{nd}$ smallest element in the list and put it in the $2^{nd}$ position, and so on.

Pass1 :- Find the location LOC of the smallest in the list of 'n' elements . A[1], A[2],...,A[n] and then interchange A[LOC] and A[1], then A[1] is sorted .

Pass2 :- Find the location LOC of the smallest in the subset of n-1 elements A[2], A[3],...,A[n] and then interchange A[LOC] and A[2] , then A[1], A[2] is sorted since A[1] <= A[2].

.....

.....

Pass n-1:- Find the location LOC of the smaller of the elements A[n-1], A[n] and then interchange A[LOC] and A[n-1] then A[1], A[2], A[3],...,A[n] is sorted since A[n-1] <= A[n]. Thus A is sorted after n-1 passes.

**Bubble Sort :-**

Suppose the list of numbers A[1], A[2], A[3],...,A[n] are in memory.

Pass1 :- Compare A[1], A[2] and arrange them in desired order so that A[1] < A[2]. Then compare A[2], A[3] and arrange them so that A[2] <A[3] continue until we compare A[n-1] and A[n] and arrange them so that A[n-1] < A[n]. In this step there are (n-1) comparisons. During this the largest element is "bubbled" up to the $n^{th}$ position. After this step is completed A[n] will contain the largest element.

Pass2 :- Repeat pass1 with one less comparison. Now we stop after rearranging A[n-2] and A[n-1]. This involves n-2 comparisons, after this step is completed the $2^{nd}$ largest element will occupy A[n-1].

.....

.....

Pass n-1 :- Compare A[1] with A[2] and arrange them so that A[1] < A[2] . After n-1 passes the list will be sorted in the ascending order.

**Quick Sort :-**

This is a divide and conquer method of algorithm A(1:n) becomes A[i] <= A[j] for all i between 1, m and for all j between m+1,n for some m,1 <= m <= n. Thus the elements in A[1,m], A[m+1, n] may be independently sorted. The problem of solving a set is reduced to the problem of sorting two smaller sets. The rearrangement of elements is accomplished by picking some element of A, say t = A[s] and then reordering the other elements so that all elements appearing before t in A[1:n] are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearrangement is called as Partitioning.

**Insertion Sort :-**

Suppose an array A with n elements A[1], A[2],...,A[n] is in memory. The insertion sort algorithm scans from A[1] to A[n], inserting each element A[k] into its proper position in the previously sorted sub array. A[1], A[2],...,A[k-1] i.e.

Pass1 :- A[1] by itself is trivially sorted .

Pass2 :- A[2] is inserted either before or after A[1] so that A[1], A[2] is sorted.

Pass3 :- A[3] is sorted into its proper place in A[1], A[2].That is before A[1] between A[1] and A[2] or after A[2] so that A[1], A[2], A[3] is sorted.

.....

.....

Pass n :- A[n] is inserted into its proper place in A[1], A[2],...,A[n-1] so that A[1],A[2],...,A[n] is sorted. This algorithm is used frequently when n is small. It is very popular with bridge players when they are first sorting their cards.

**Merge Sort :-**

Another example of divide-and-conquer technique is merge sorting. Here we assume that the elements are to be sorted in non-decreasing order . Given a sequence of n elements A[1],

A[2] ,...,A[n] then the general idea is to imagine them split into two sets A[1], A[2],...,A[n/2] and A[n/2+1] , ...,A[n]. Each set is individually sorted and the resulting sequences are merged to produce a single sorted sequence of n elements. Finally we combine the two sets into one using merging.

**Heap Sort :-**

Suppose an array with n elements is given, then this algorithm consists of 2 phases.

Phase A :- Build a heap out of the elements of A.

Phase B :- Repeatedly delete the root element of H. Since the root of H always contains the largest node in H, phase B deletes the elements of A in decreasing order.

## 2.2 Searching Algorithms :-

**Searching :-**

Searching refers to the operation of finding the location of a given item in a collection of items. There are two types of searching algorithms.

**Linear searching :-**

Suppose DATA is linear array with 'n' elements, then to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one, i.e. first we test whether DATA (1) = ITEM and then we test whether DATA(2) = ITEM and so on .This method which traverses DATA sequentially to locate ITEM is called linear / sequential search.

**Binary Search :-**

Suppose DATA is an array which is sorted in increasing order. During each stage of algorithm the search for ITEM is reduced to a segment of elements of DATA.

DATA[BEG], DATA[BEG+1],...,DATA[END]. We compare ITEM with the middle element DATA[MID] where MID = INT((BEG + END)/2). Initially BEG = 1, END = n

If DATA[MID] = ITEM, then the search is successful and we reset LOC = MID else

a) IF ITEM < DATA[MID] then ITEM can appear only in the left half of the segment

b) DATA[BEG],...,DATA[MID-1]. So we reset END = MID –1 and begin searching again.

IF ITEM > DATA[MID] then the ITEM can appear in the right half of the segment

DATA[MID+1],...,DATA[END]. So we reset BEG = MID +1.

If ITEM is not in DATA then eventually we obtain END < BEG. This will signal that the search is unsuccessful and in that case we assign LOC = NULL, i.e. NULL is a value that lies outside the set of indices of DATA.

## 2.3 Performance Evaluation :-

Performance evaluation can be loosely divided into two major phases :

1. A priori estimates / performance analysis
2. A posteriori testing / performance measurement

### 2.3.1 Performance Analysis :-

This can be explained by the following factors :

1. Space complexity.
2. Time complexity.

### Space Complexity :-

The *space complexity* of a program is the amount of memory it needs to run to completion. The space needed by every program is seen to be the sum of the following components :

1. A fixed part that is independent of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables and fixed size component variables, space for constants, etc.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables and the recursion stack space.

**Time Complexity -**

The *time complexity, T(P)* of a program is the amount of computer time it needs to run to completion. The sum of the compile time and the run time taken by a program P, i.e. the number of steps taken by the program to compute the function it was written for.

Step :-

A step is any computation unit that is independent of the characteristics. Thus 10 additions can be one step; 100 multiplications can also be one step; but 'n' additions cannot. Nor can m/2 additions, p+q subtractions, and so on be counted as one step. There are three kinds of step counts :

1. Best-case :- The best-case step count is the minimum number of steps that can be executed for the given parameters.

2. Worst-case :- The worst-case step count is the maximum number of steps that can be executed for the given parameters.

3. Average-case :- The average-case step count is the average number of steps executed on instances with the given parameters.

**Asymptotics :-**

The word *asymptotics* means that the study of functions of a parameter *'n'* as *n* becomes larger and larger without bound.

**The Big Oh Notation :-**

If f(n) and g(n) are functions defined for positive integers, then to write $f(n) = O(g(n))$

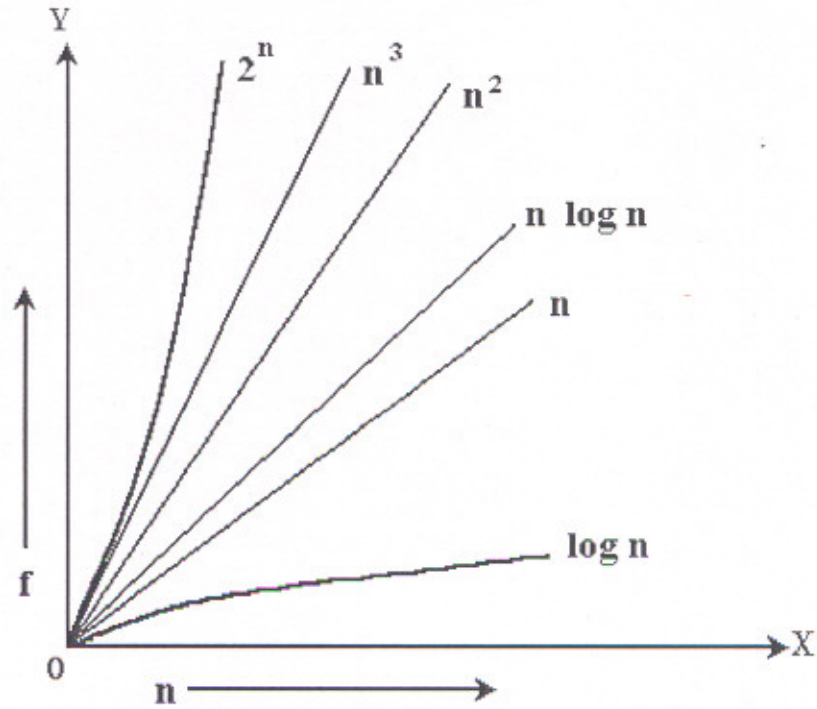



Fig 8.1 Growth rates of common functions

means that there exists a constant c such that | f(n) | <= c * | g(n) | for all n, n >= $n_0$. Under these conditions we also say that " f(n) has order at most g(n) " or " f(n) grows no more rapidly than g(n) ".

When we apply this notation, f(n) will normally be the operation count or time for some algorithm, and we wish to choose the form of g(n) to be as simple as possible. We thus write *O(1)* to mean computing time that is bounded by a constant, *O(n)* means that the time is directly proportional to n, and is called *linear time.* We call $O(n^2)$ as *quadratic*, $O(n^3)$ as *cubic*, $O(2^n)$ exponential. These five orders together with *logarithmic time O(logn)* and *O(nlogn),* are the ones most commonly used in analyzing algorithms. Fig 8.1 shows how these seven functions grow with n. Notice how much slower log n grows than n, both the function 1 and the function log n become farther and farther below all the others for large n. Also notice how much more rapidly $2^n$ grows than any other functions. An algorithm for which the time grows exponentially with n will prove usable only for very small values of n.

### 2.3.2 Performance Measurement :-

Performance measurement is concerned with obtaining the actual time and space requirements of a program. These quantities are dependent on the particular compiler and options used as well as on the specific computer on which the program is run. Certainly the space and time needed for compilation are important during program testing. The focus is on measuring the computing time of a program. To obtain the computing of a program, we need a clocking function.

## 2.4 Comparison of Methods :-

The best way to compare the different sorting methods is by the following factors :

1. Use of storage space.
2. Use of computer time.
3. Programming effort.
4. Statistical analysis.
5. Empirical testing.

# Chapter-3 Development of GUI Package

The development of software titled *" GUI Package On Data Structures "* includes three files namely, *TOOLS.H, PROJ.CPP* and *HELP.TXT.*

## 3.1 TOOLS.H FILE :-

*TOOLS.H* is the header file (user defined) which includes all the tools which are needed for this thesis. All these tools are written using C++ and Graphics. The graphics has been initialized in the tools.h file, i.e. the *initgraph()* is initialized in this base class. Since a help is included for every data structure, a help.txt file has been opened in read mode in tools.h. These initializations have been written in a zero argument constructor tools(), immediately a destructor has been written for de-allocating or releasing the memory that was allocated for the object by the constructor. A destructor has the same name as the constructor but is preceded by a tilde symbol.

The tools.h has the data members mx, my, button, scan, ascii, *fp declared in the *protected* access specifier. Protected members behave just like private members until a new class is derived from the base class that has protected members. Protected members are public to derived classes but private to the rest of the program.

The class *tools* is the base class of this thesis. The class *datastru* is derived from the base class tools. A derived class inherits all the capabilities of the base class, but can add new features of its own. By making these additions the base class remains unchanged. The *TOOLS.H* file has been included as a header file. The public access specifier means that the protected members of the base class are protected members of the derived class and the public members of the base class are public members of the derived class. Different member functions which are present in TOOLS.H of *public* access specifier are as follows :

*Int inptextxy(int, int, int, int, int);* This is member function which is used to get the input from the user. It has 5 arguments namely the coordinates of x, y, font , size and the color of the data which is entered through the keyboard. In this complete thesis to get the input for any member function we have to use this member function only.

*Void textbox(int, int, int, int, int, char *, int, int, int, int, int);* A tool in *TOOLS.H* " textbox( ) " is a member function in *TOOLS.H* to draw a textbox with attributes x1, y1, x2, y2, name of the textbox, i.e. a pointer to an array, font, size, color, fill color and border color.

*Void helpscreen(int);* This is a member function used to get the help whenever a user presses F1 key and at the same time the key Esc is used to close the help screen. It is created using the graphics command *imagesize.* This command will create a help screen of required length using *malloc()* function. The size of the image is stored in the memory.

*Int showarm(int, int, int, int);* Whenever the cursor highlights a rectangular text box, an arm shape will be shown on the text box. This is used for this purpose only. The co-ordinates of the mouse position are stored into one variable and that will be compared with all the four arguments x1, y1, x2, y2 of the showarm() function. If the coordinates are within the boundary of the mouse coordinates then the arm will be displayed.

*Int showplaces(int *, int, int, int);* It has four arguments and the first one is a pointer to an array. In this array the coordinates of the rectangular text box will be stored. The second argument is the number of text boxes in that menu. The third text box is the starting number of the text box and the last argument is the end number of the text box. Generally the fourth one is used to return to the main program, i.e. exit to the main program.

*Void winmenu( );* This function is used to display the start menu of this thesis. This has been written using different graphics commands such as line(), rectangle(), outtextxy().

The usage of mouse as well as keyboard is included by using the interrupts of 8086 given by *int86( )* function. Ex:- int22H is the number of the keyboard interrupt, int33H is the number of the mouse interrupt.

GUIs and mouse go hand in hand. Though some GUIs do exist which manage the show without a mouse, the mouse has more or less become a standard input device with any GUI worth its name. A mouse is used to point at the icons which form the menu in a GUI-much like the way a child points to something he wants.

These point-and-shoot means of GUI bring along ease and convenience along with all the added agility of the real-life look alike of the mouse. As a result more and more packages today are not only menu driven, but also mouse driven. The use of a mouse requires a program to sense its presence. Load the device driver program that understands the mouse. A device driver is a program which senses the signals coming from the port to which the mouse is attached. On sensing the signals, the driver translates these into the related action on the screen. The following member functions are all related to mouse functions. For all these functions the interrupt number 33h is issued using the int86() function. The mouse functions can be accessed by setting up the AX register with different values or service numbers. The service number '0' is used to reset and get the status of the mouse. Service number '1' is used to show the mouse pointer, here it is an arrow. Service number '2' is used to hide the mouse pointer, '3 is used to get the mouse position. Using all these service numbers the following mouse functions are performed such as *initmouse(), showmouse( ), hidemouse( ), getmousepos( ), clearmousestat( ), changecursor().*

*Void armcursor( );* This is used to display the arm shape on the text box whenever the mouse pointer points to the text box.

*Void arrowcursor( );* In this function the bitmap of the arrow shape is stored in a 32-bit bit

map. The '1's in the bitmap indicate that the pixel would be drawn else the pixel is erased.

*Void click( int, int, int, int, int);* This member function is used to select the text box. Whenever the textbox is selected, a color with dots will be displayed around the text box. The 4 arguments are the co-ordinates of the text box and the fifth one is the border color which is red color here.

*Int getkey( );* In this function a standard library function *kbhit()* is used. So long as a key is not hit, kbhit() keeps returning '0'. As soon as the key is hit from the keyboard, kbhit() returns a truth value. Interrupt number 22h is issued using int86() and the variables ascii, scan have been declared as global variables. Once set these variables are accessible to other functions as well.

Let us now see the usage of the keyboard such as getkey( ). When a key is hit from the keyboard, its ascii code and scan code get stored in the keyboard buffer. For normal keys like A, B, Z, !, # etc., these are unique ascii codes, whereas for special keys like arrow keys, Home, End, Pgup, Pgdn etc., the ascii codes are zero. Therefore to identify these special keys, their scan codes are used, which are unique for every key present on the keyboard. The ascii and scan codes of the key hit can be retrieved from the keyboard buffer by issuing the interrupt 22. This number can be issued using int86( ).

## 3.2 PROJ.CPP FILE :-

*PROJ.CPP* file is the main C++ program file. The most powerful feature of object oriented programming is the inheritance. Inheritance is the process of creating new classes, called derived classes, here *datastru* class is a derived class from existing class. These existing classes are often called as base classes which in this case is the tools.h. The derived class inherits all the capabilities of the base class but can add new features and refinements of its own. By adding these refinements the base class remains unchanged. Hence the inheritance will give the

most important advantage, i.e. the reusability of the code. Once a base class is written and debugged, it need not be touched again but at the same time it can be adapted to work in different situations. Reusing will save time, money and reliability of the program.

```
Void main( ) // This is the main menu of the thesis .
    {
        flushall( );
        datastru ds;
        ds.winmenu( );
        ds.mainselect( );
        closegraph( );
    }
```

The *mainselect()* function is used to start the process of execution of all data structures. In this function it includes all the classification of data structures like linear, nonlinear, sorting, searching. This will be displayed using the text box() member function. The coordinates of all these text boxes are stored into an array. By using the showplaces() function we select the one of the options from the list of options. If the option is end number then the control will be transferred to the main program, otherwise the corresponding member functions will be executed. Assume that if the option is '1' then the member function *lin()* is invoked and the same procedure is repeated at here also. ds is an object of the class datastru, ds.winmenu() is used to display the start menu of the thesis. A program can declare objects of both the base and derived classes. The two objects are of independent of one other. An object of a derived class cannot access private or protected members of base class. Private members of base class cannot be accessed by derived class member functions or objects of derived class. The functions lin(), nonlin(), sorting(), searching() are used to display their corresponding sub menus.

*Lin()* :- This function is used to display the submenu consisting of linked list, stack, queue,

arrays, back and exit by using the member function in tools.h such as text box() and showplaces(). If the option is 1 then the link list() is selected, if the option is 2 then the *queuemain()* is selected, if the option is 3 then the *stack()* is selected, if it is 4 then the *array()* is selected, if the option is 6 then the control is transferred to the main program.

Link list() :-In this function different linked lists and their operations are included such as *singly, doubly, operations.* From this function the control next goes to the member function *linkinsdel()*. In this function there are 4 sub options and their linking to the singly linked, doubly linked and the operations. If the item = = 1 then it is a singly menu if it is 2 then it is a doubly menu and for 3 it is about operations on lists.

In *Create list()* function a variable is assigned a value 'y', it is always compared with the letter typed from the keyboard, if it is 'n' then a message is displayed like 'successfully created' . If the option is 'y' then list[i] = item and the next[i] = NULL , next[i-1] = i. Initially we assign i as zero and next[i]= NULL. To display the messages we use the outtextxy().

*Singlyinsert()* function is used to insert an item in the list. First it will check for the presence of the item, i.e. if the item = =list[i] then found = 1, i=next[i], next[i] = cnt, next[cnt] = temp. Now we have to insert the item. List[cnt] = inptextxy(), where cnt is a variable.

*Displaylist()* function is used to display the final linked list. First we check whether the list[0] != NULL and next[i] != NULL. Once these two are satisfied then using the line command we draw the rectangle to represent he nodes and then assign i = next[i]. Now draw two other line commands to draw the NULL symbol which is like a cross.

Non() function is used to display the nonlinear menu giving options such as tree, graph, back and exit. The function sorting() is used to display the different text boxes for different sorting methods which are linear, bubble, quick, insertion, merge and heap sorts. Similarly searching() is

used to display the different searching methods, such as linear search binary search these two options have been written using the text box(), showplaces() member functions. In linear search we find the location of an item in an array. First we set the item to1 in the array, i.e. the first location of the array and now the search starts to search the entire array for the item, if the item is found then we display the corresponding location of it, otherwise a message will be displayed stating that the item is not found in the list. For binary search first we assign beg =1, end =n and we calculate the middle of the list of elements by mid = (beg + end)/2. Now we compare the item to be found with the array of middle, if the item is below the array of middle then we assign end =mid -1 and repeat the above procedure else assign mid +1 to beg variable. If the item = array of the middle then the location of the item is the middle.

In *queuemain()* there are different queues list such as general and circular queues. In general queues there are two options insert item and delete item. For inserting an item we first check whether the queue is full or not, if the queue is not full then we increment the rear by 1 finally the rear will take the item. Similarly, for deleting an item, first we check whether the queue is empty or not, if the queue is not empty then front will be incremented by 1 and the item will be assigned the value of front. The *stack()* is used to display the different operations on the stacks such as push(), pop() and recursion. For push() first we check the stack full condition, if the stack is not full then the TOP will be incremented by 1. After this the TOP will take the item. Similarly, for pop function first of all we have to check for the stack empty condition, if the stack is not empty then the item is assigned the TOP element of the stack and then TOP will be decremented by 1.

The function datastru() is a single argument constructor, in this all the initializations have been declared such as list[0] = next[0] =NULL, row = -1, front = rear = stktop =0, maxq =6,

maxstk =10. For constructors there is no return type is used. The constructor is a special member function that allows us to set up values while defining the object, without the need to make a separate call to a member function. Thus a constructor is a member function that is executed automatically whenever an object is created.

The *tree()* is used to display the different sub options in this menu such as *binary tree, general tree* and different traversal methods such as *inorder, preorder* and *postorder*. In a general tree different properties were written like depth, siblings, root.

*HELP.TXT* is a text file which includes the help of all data structures. Whenever any user presses F1 key, then the help about the corresponding textbox will be displayed completely. To close the help screen we have to press Esc key. The user who does not know about any data structure can get the help from this screen. In this way it is helpful to any end-user.

# CHAPTER-4 Implementation of Linear Data Structures

Screen-1 :- The screen-1 is the first output screen of the thesis. It will give the name of the thesis. When we press any key then the control will go to the main menu of the data structures. This has been implemented using the *winmenu()* function. Different graphics commands such as setcolor(), settextstyle(), outtextxy() etc. are used to display this menu.

Screen-2 :- This is the second output screen after pressing a key in screen-1. In this screen there are 4 options, which are *linear, nonlinear, sorting and searching.* The fifth one is the *exit* to exit into the main program. We can select any of these options using either a mouse or down arrow of the keyboard as it was displayed on the screen. In this option the exit option is highlighted. If we click on it using either Enter key or mouse click then the exit will be selected and the control will be transferred to the main program.

Screen-3 :- In this screen the option *linear* has been selected. The selection is through the down arrow . mouse followed by a Enter key. So, in linear option there are 5 sub options namely, *linked lists, queues, stacks, arrays* and *back*. Now as shown in the screen-4, the *linked list* is highlighted. If we press this button then we move on to the next screen.

Screen-4 :- There are 3 options in this screen. Which are *singly, doubly* and different *operations* on the lists and *back*. Exit is common to all the screens to exit into the main program.

Screen-5 :- In this screen the *singly* option has been selected and again there are five sub options in that singly text box which are *create list, insert item, delete item, display list* and *back* to go back to the screen-4.

Screen-6:- In the screen-5 the option *create list* has been highlighted. When the user Enters against this option then the screen-6 will be displayed. In this screen first we have to enter the item into the list. Next the user has been asked to continue or not. If the user types 'n' then the

message 'successfully created' will be displayed and by default the back option will be selected.

Screen-7 :- If we press *back* then we again go back to the menu of the *singly* list. In that option the display list is selected and the enter is pressed to get this screen. It contains a singly linked list and the last node is pointed to a null pointer, which is any invalid address.

Screen-8 :- Similarly, we have to go back to the *singly* menu and select the *insert* option, to insert an item into the list. Here the item 66 has been inserted into the list after '6' and finally the list has been displayed after insertion.

Screen-9 :- In this screen the option delete item from *singly* menu has been selected. Here the item '66' is being deleted and gone back to *linked list* menu.

Screen-10 :- The *doubly linked list* has been selected and the *create list* in doubly has selected and created a doubly linked list as is done in the case of singly linked lists. But here a double pointer is used to indicate that it is a doubly linked list.

Screen-11 :- This tells that insertion into a doubly linked list. So, the new item to be inserted is '33' which is after'3'.

Screen-12 :- Displaying of a doubly linked list after the insertion operation as in the screen-11. Now we press back to go to the previous menu.

Screen-13 :- The third option in linked lists is that the operations on lists which are *sorting, searching, reversing, maximum and minimum* of a list. When a *sorting* is selected the list becomes a sorted one. We can *search* an item, *reverse* a list also we can find the *maximum* and *minimum* of the list.

Screen-14 :- this is the second option in the *linear* menu. The sub options are *queue, circular queue* and *back*. Here the option queue has been selected.

Screen-15 :- In this screen the *general queue* operations have been implemented. The

pointers of rear / front will be incremented as per the insertion/deletion of items. The rear is now at '4' and the front is at '0'.

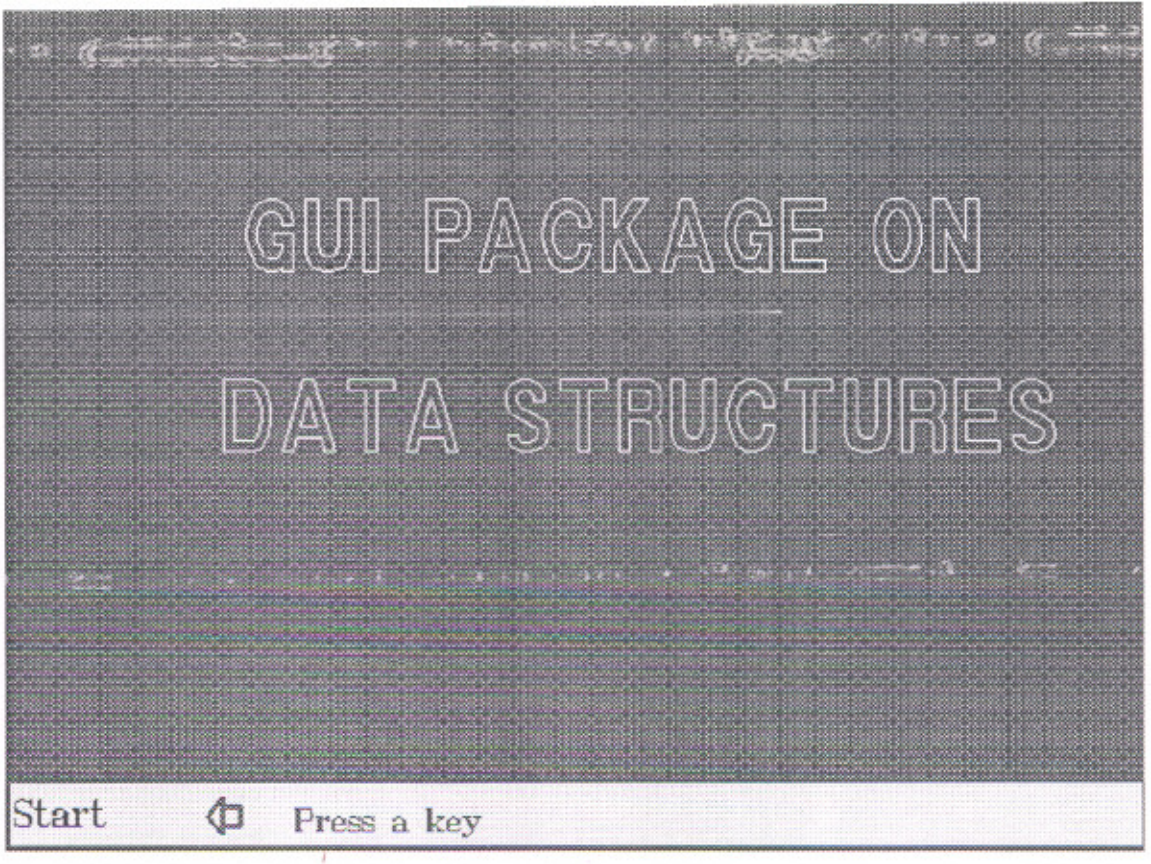
Screen-16 :- This is a *circular queue.* In this if the rear = 6 then after insertion the rear will be pointed to 1. Now one can insert at queue[1], since it is a circular queue.

Screen-17 :- This is the third option from the *linear* menu. There are 4 sub options in this menu, namely *push, pop, recursion.* The maximum stack length is 10. Now *push* has been selected. When the user Enters a key against push then the top will be incremented by 1, if top reaches to 10, again if we insert into the stack then the message overflow will be displayed.

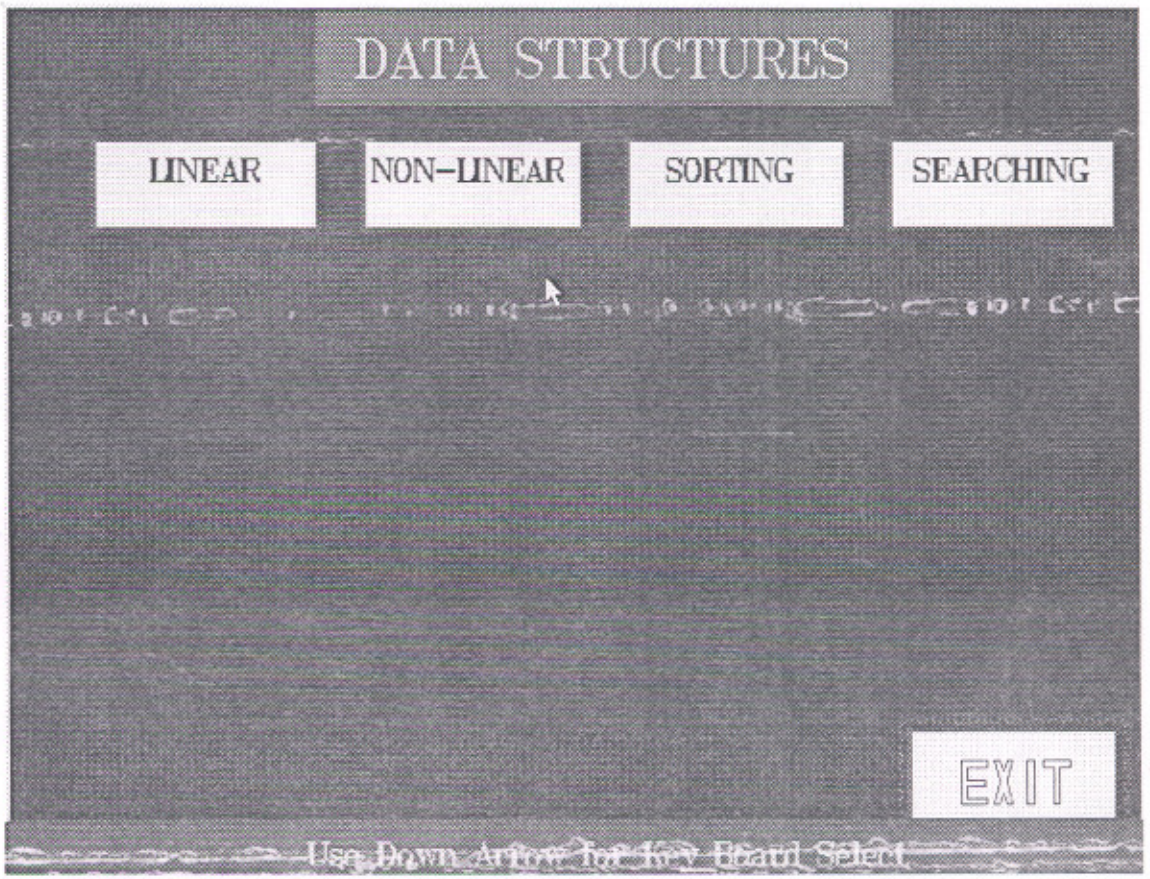
Screen-18 :- In this screen the option *pop* has been selected so, the top most element on the stack which was '77' has been popped from the stack and the top has been decremented by 1.

Screen-19 :- The third option in stacks is *recursion.* Here the user has been asked to enter the number of disks. These disks have been moved from spindle to the to spindle using auxiliary spindle. During moving disks the order should be same.

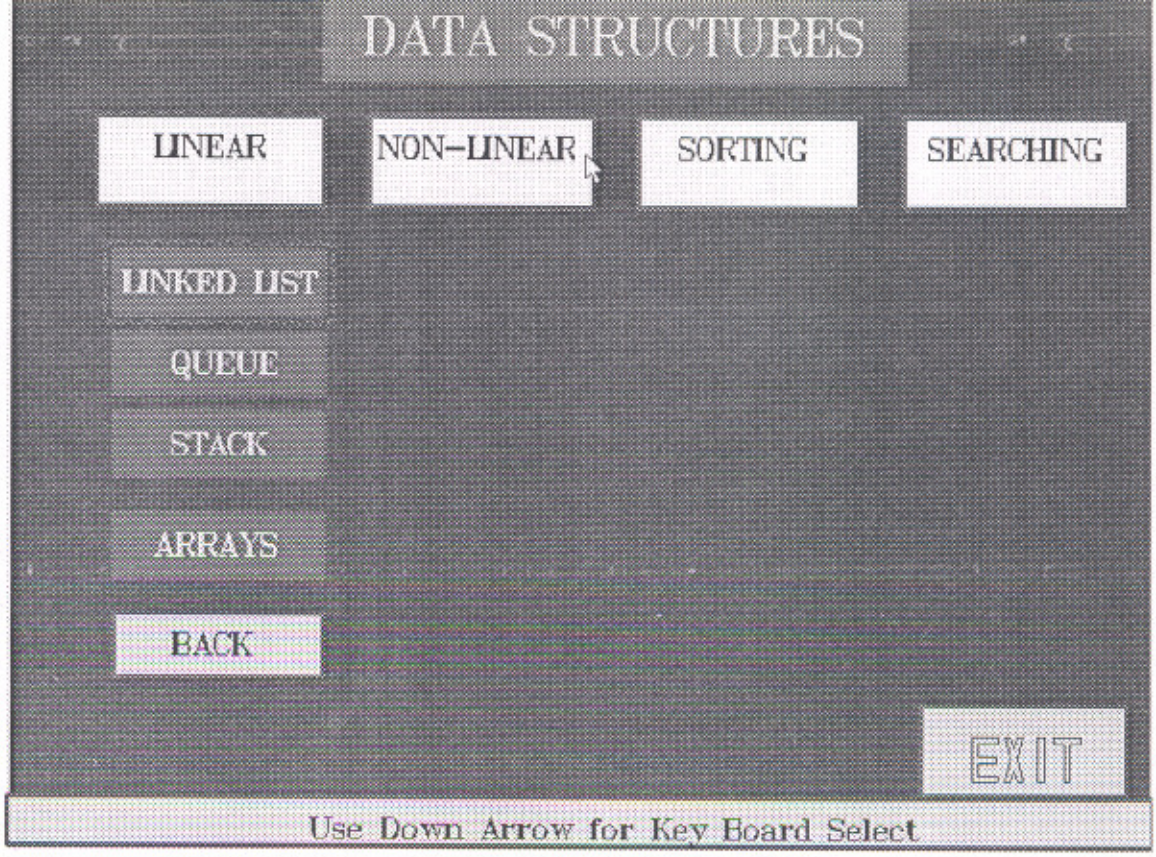
Screen-20 :- In the screen-19, when the user Enters a *back* option the control transfers to the *linear* menu. In this screen the *sparse matrices* and its *transpose* has been implemented on the right side of it.



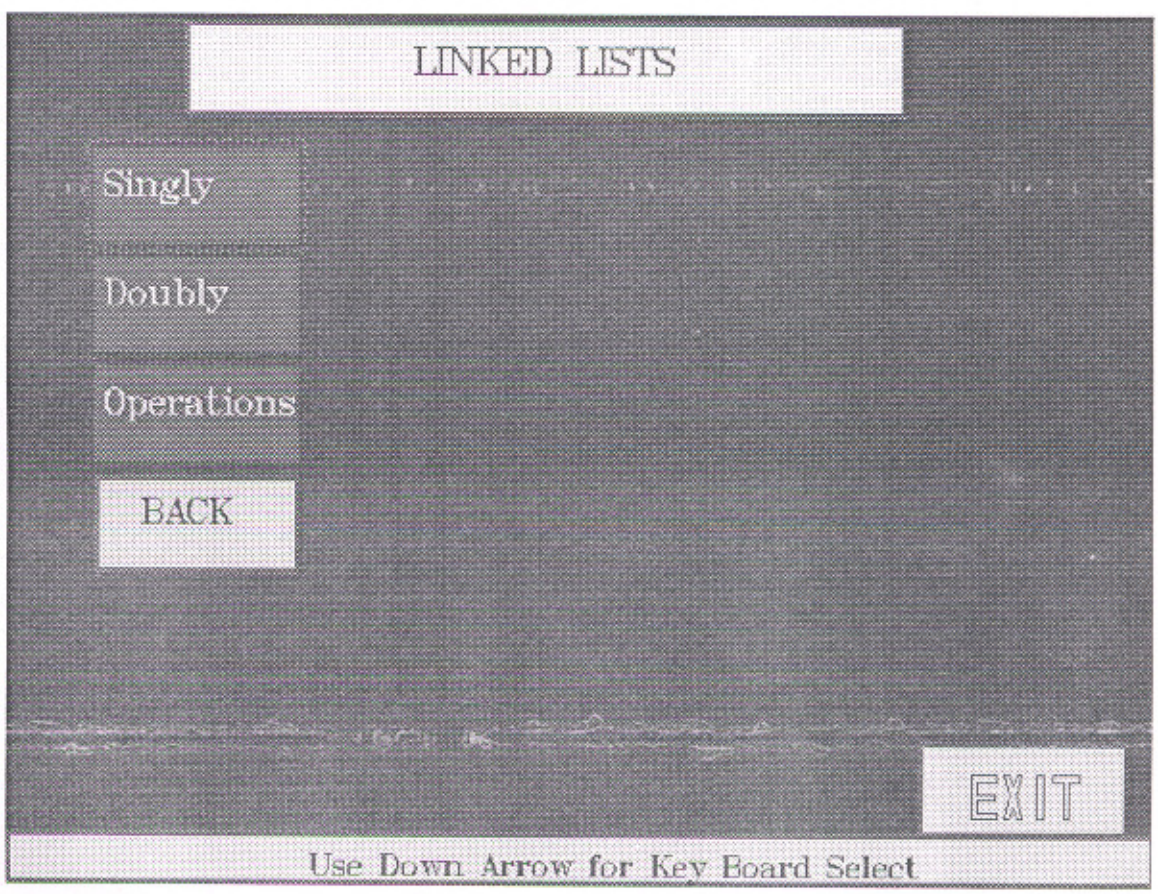

SCREEN-1



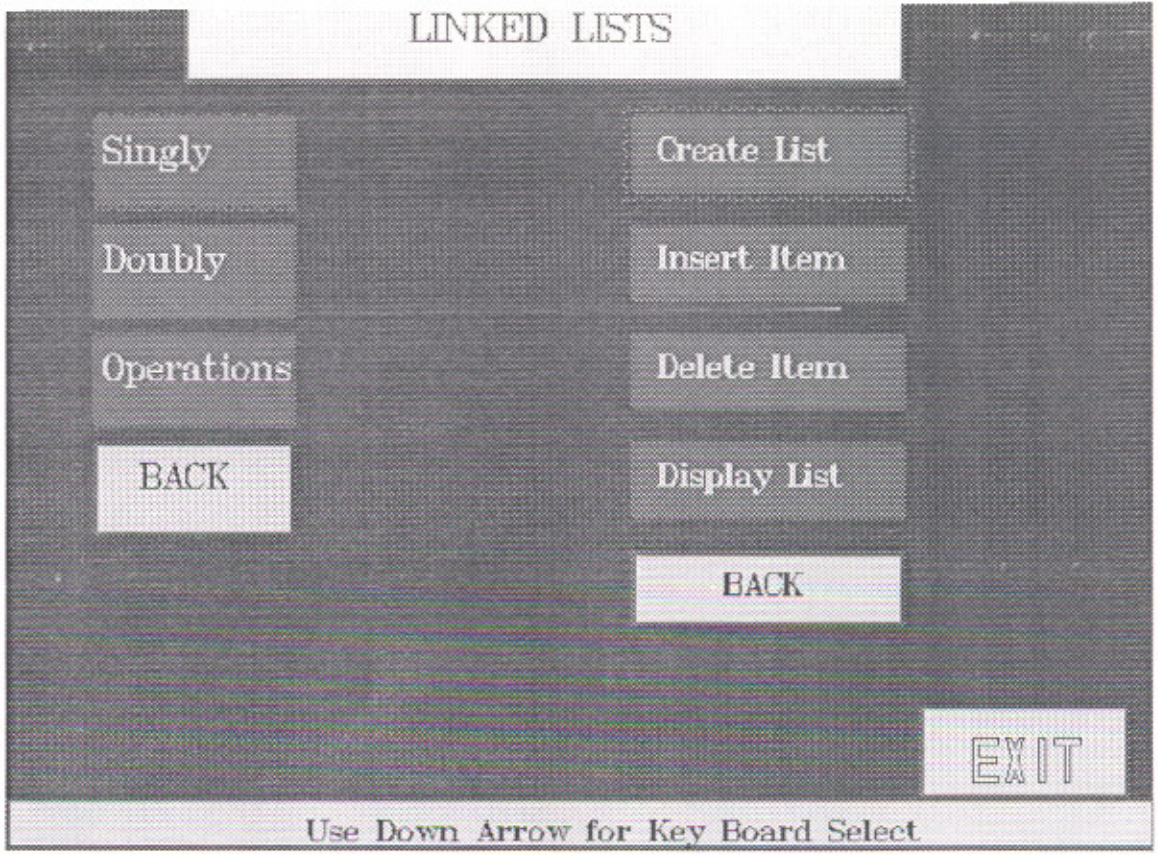

SCREEN-2



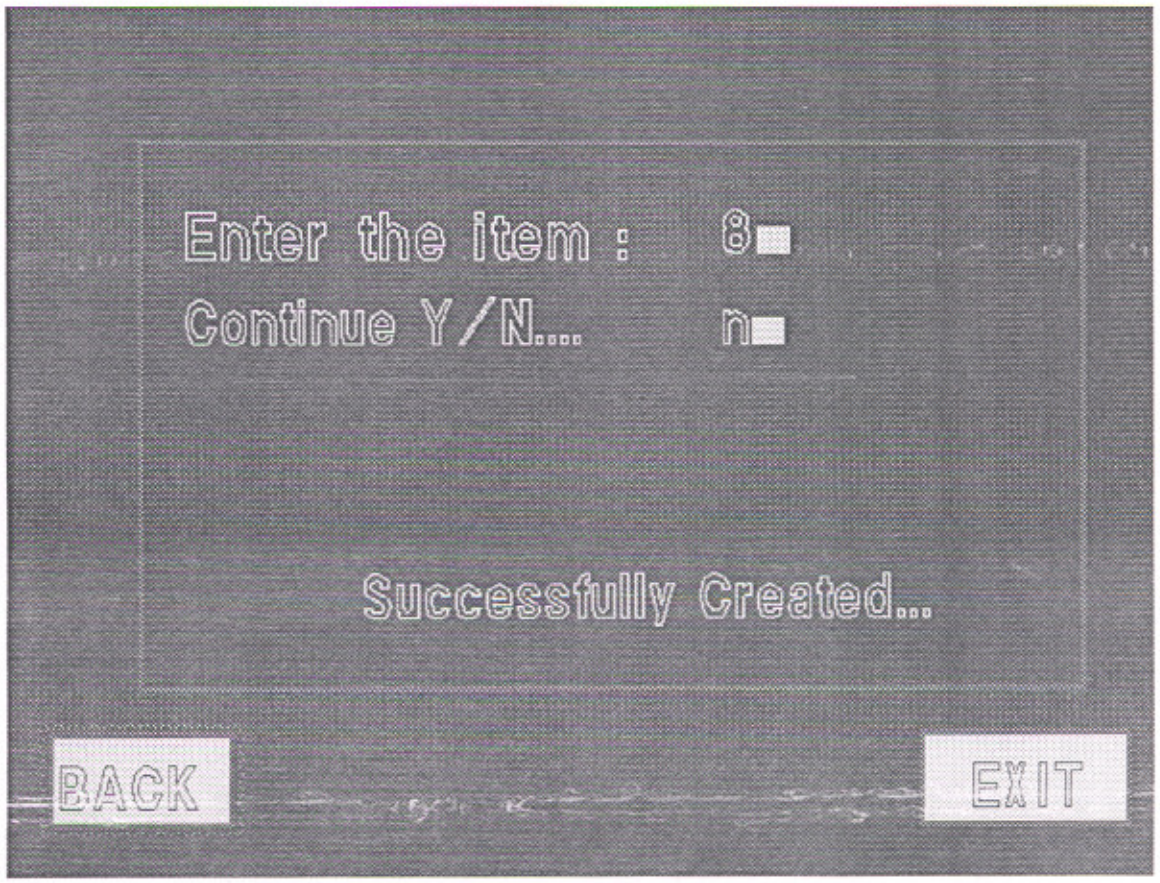

SCREEN-3



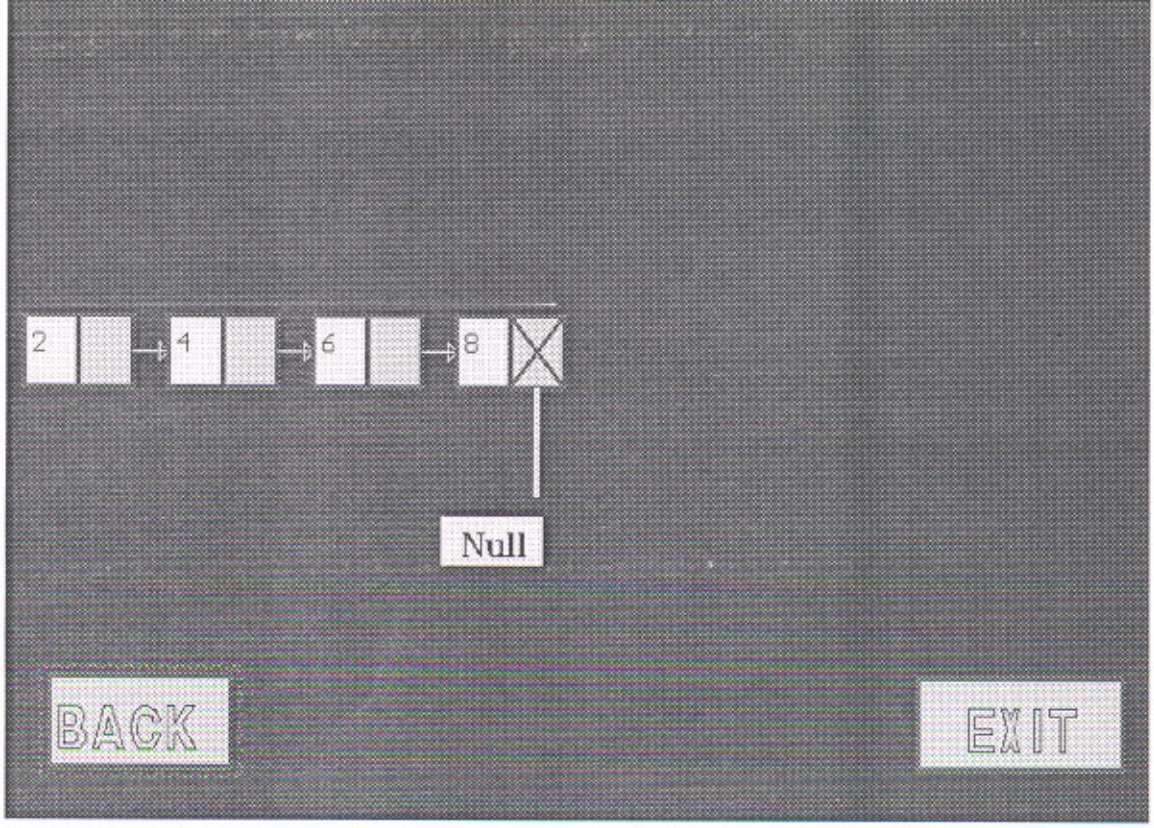

SCREEN-4



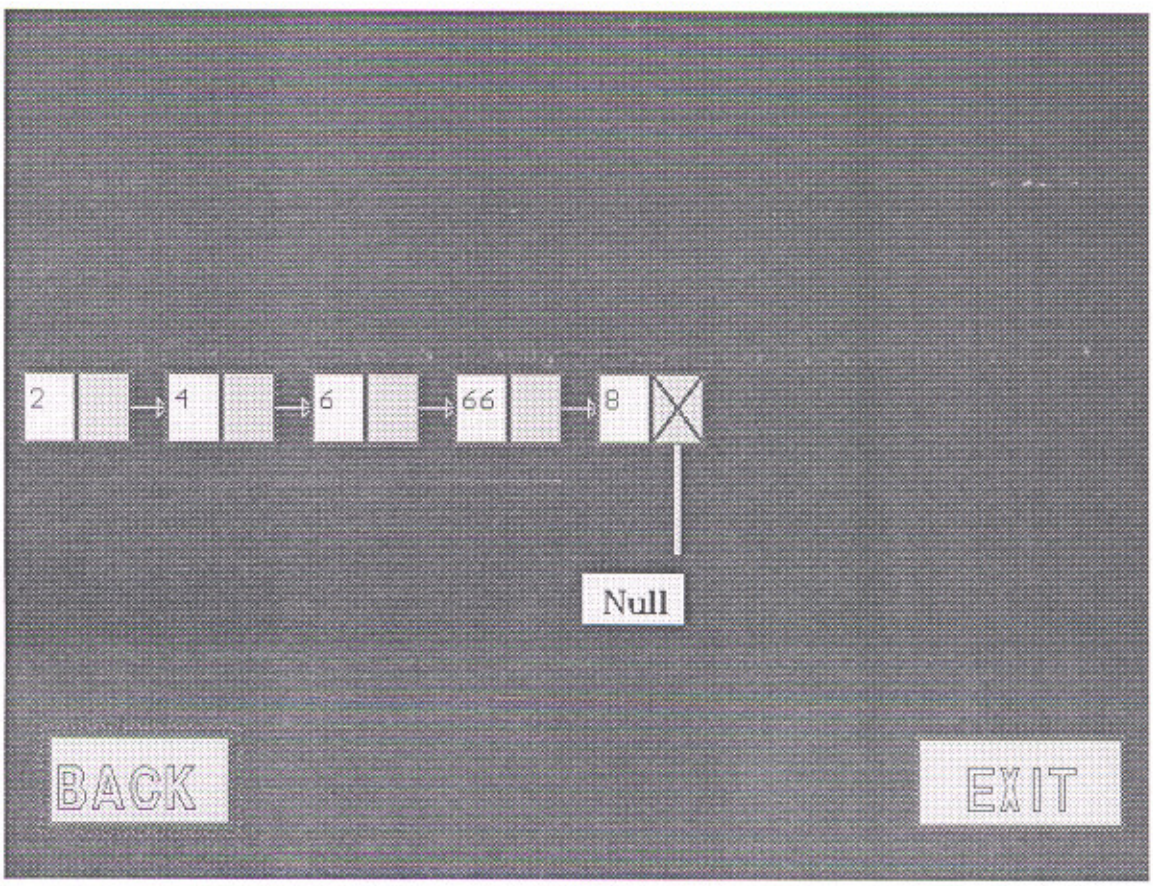

SCREEN-5



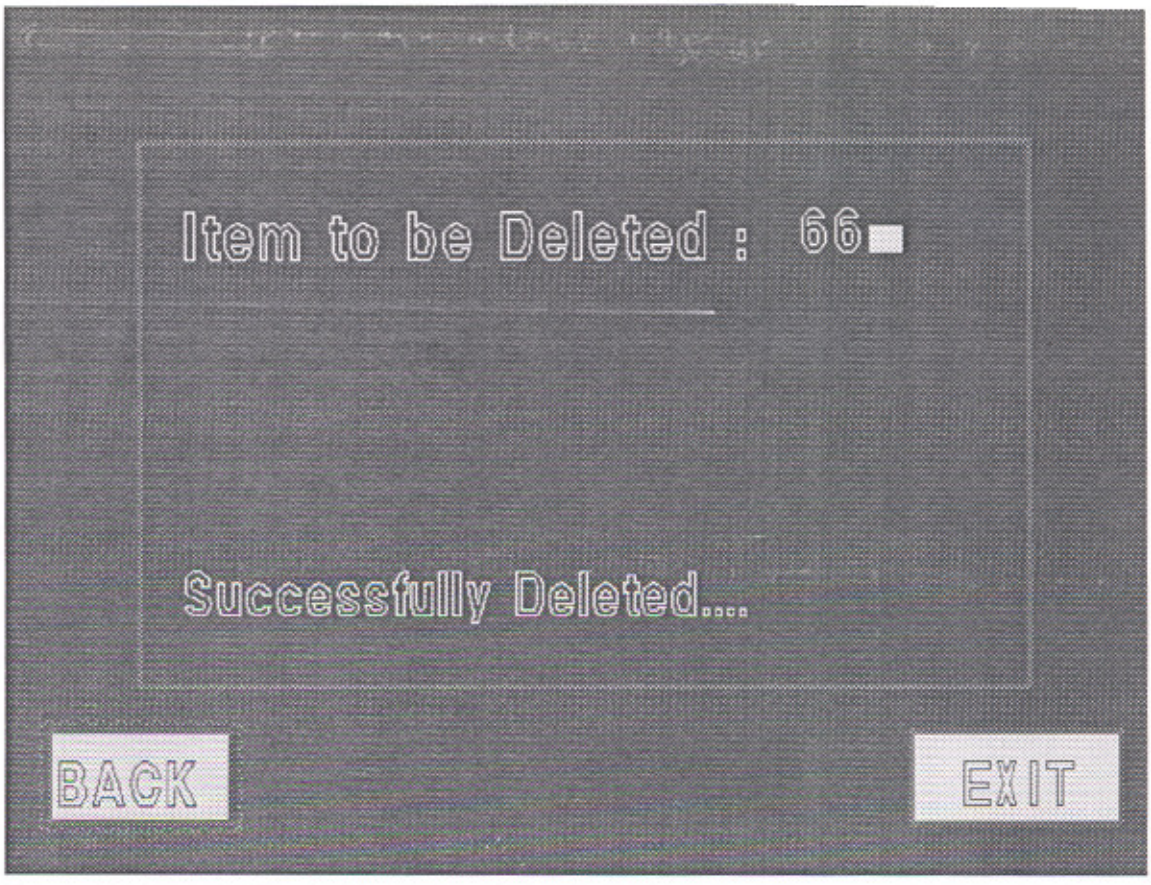

SCREEN-6



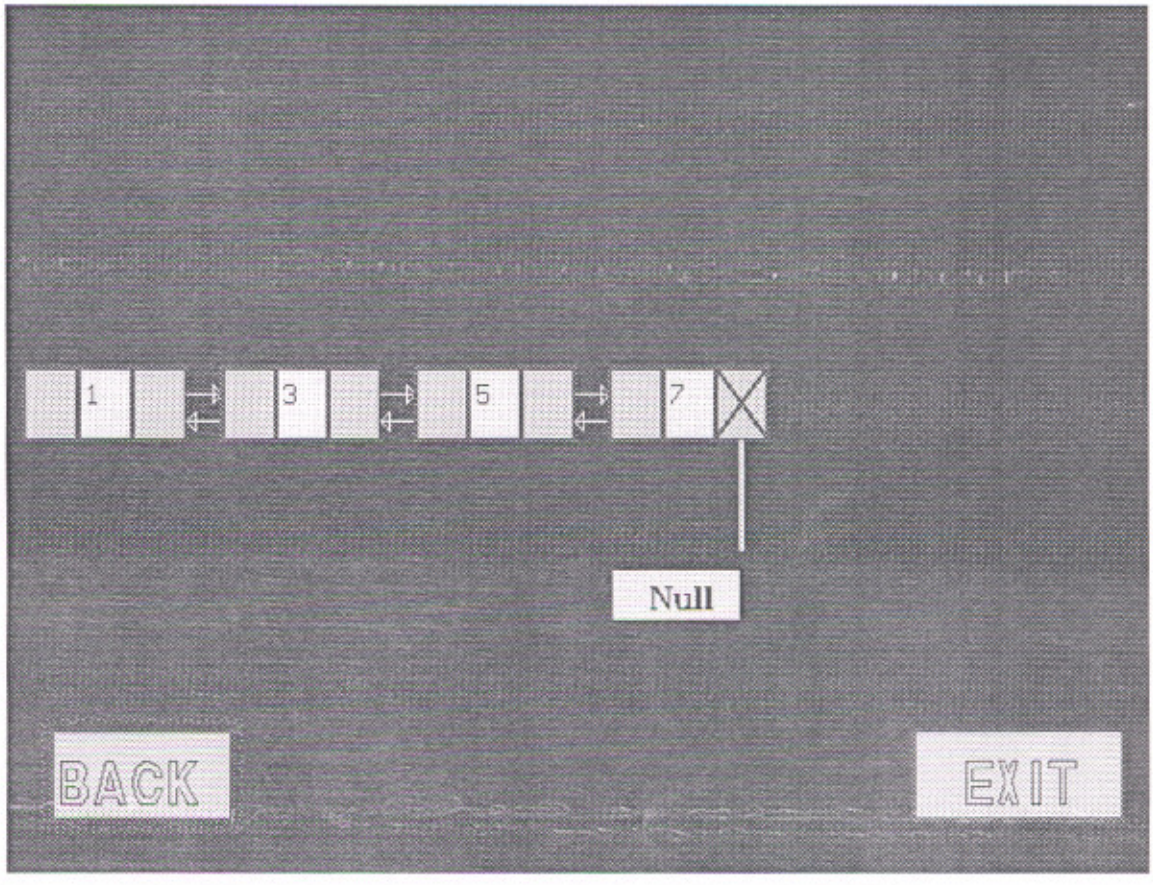

SCREEN-7



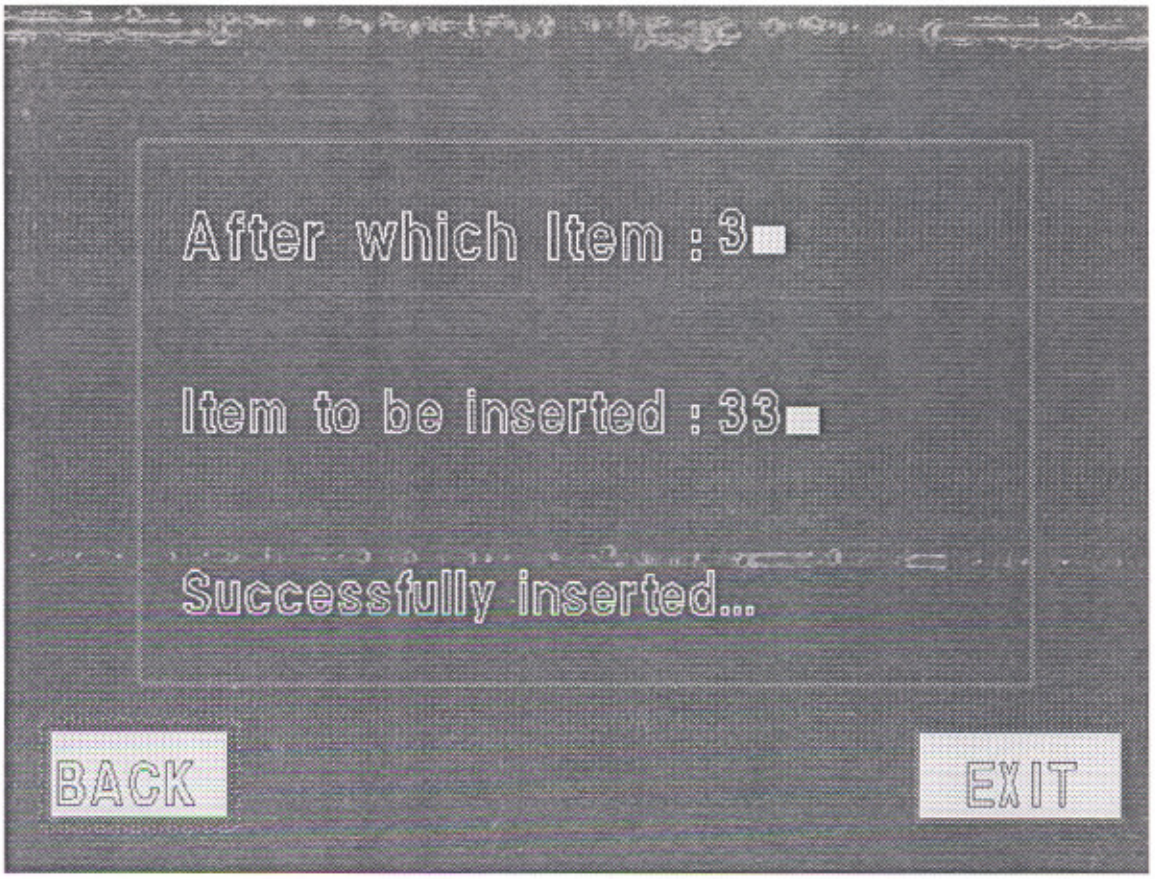

SCREEN-8



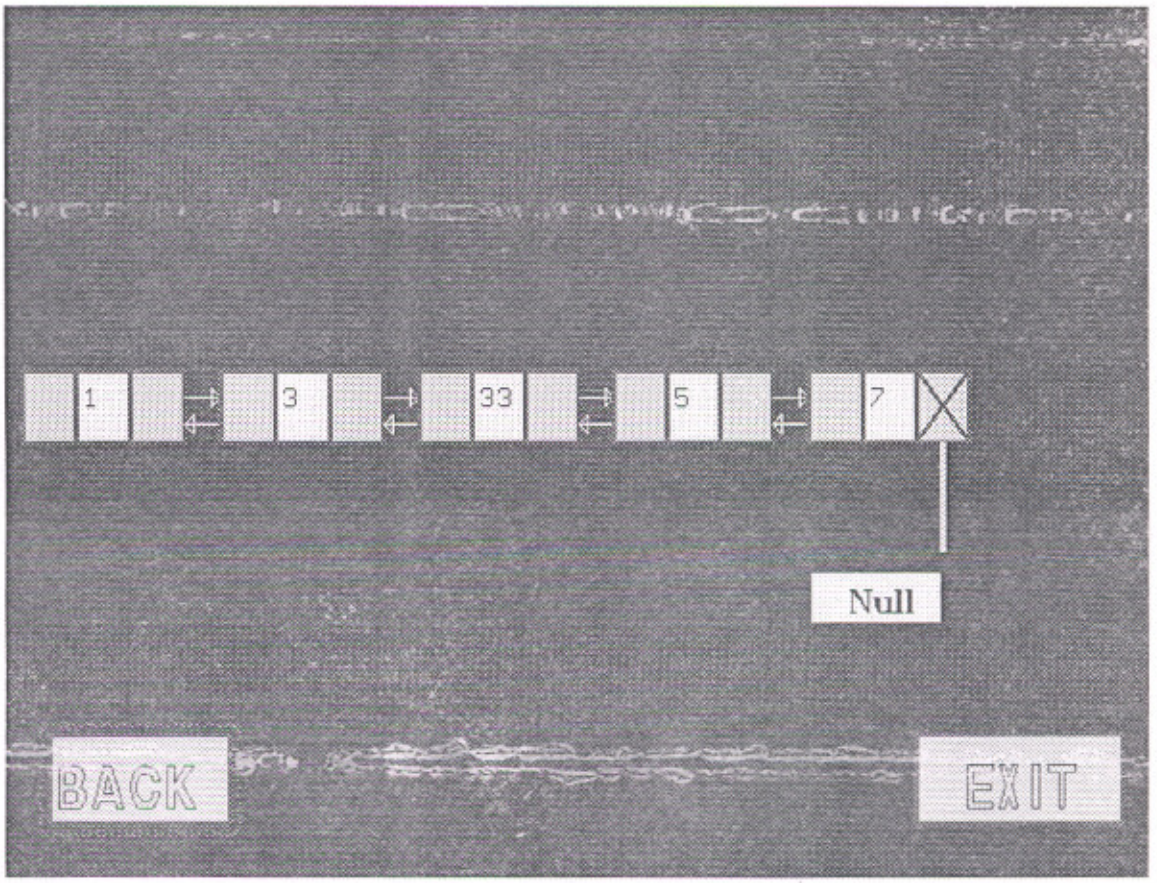

SCREEN-9



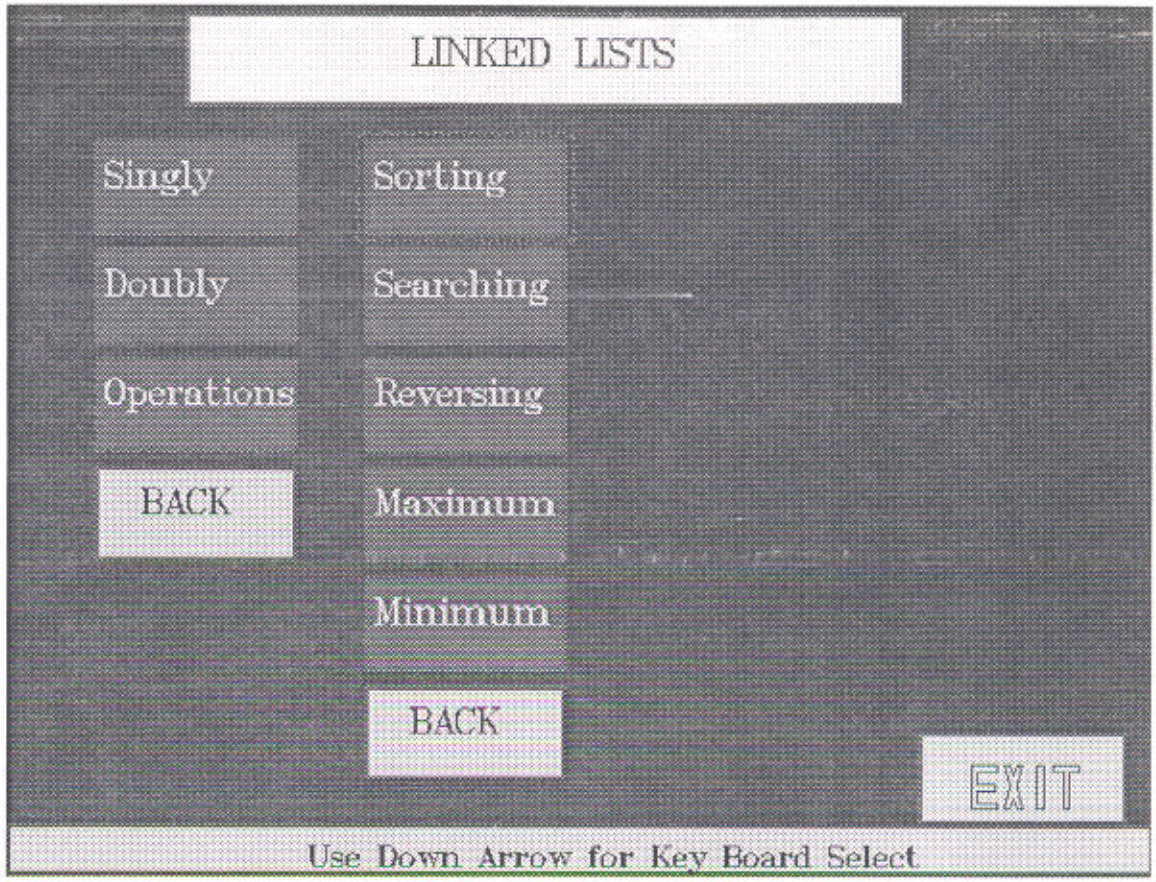

SCREEN-10



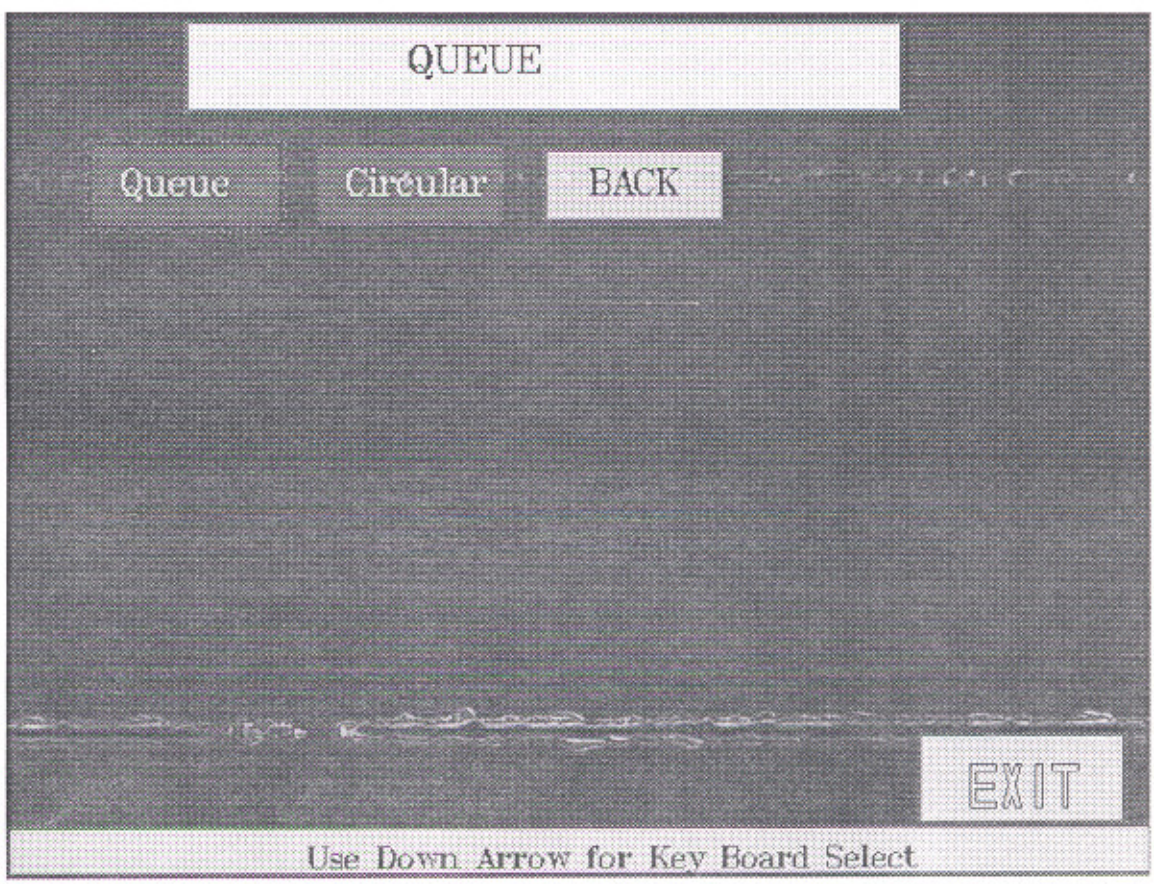

SCREEN - 11



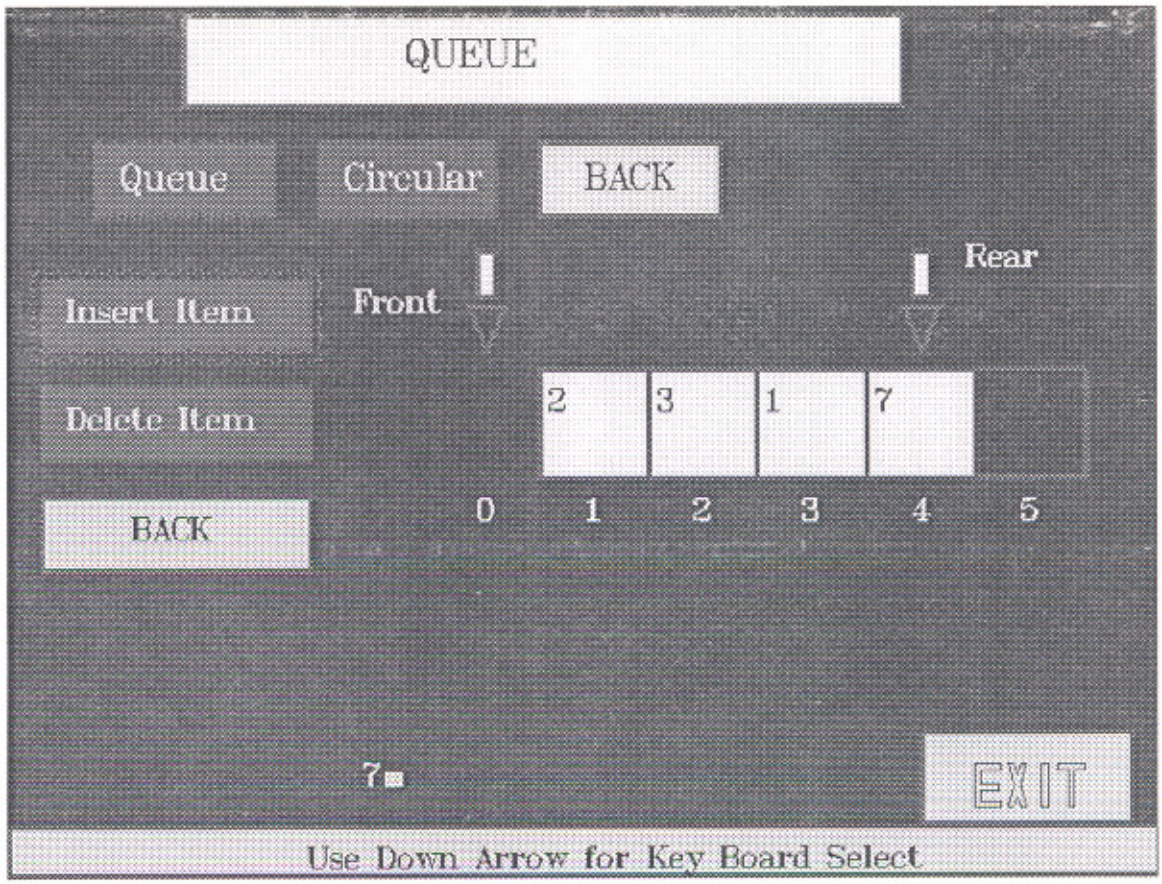

SCREEN - 12



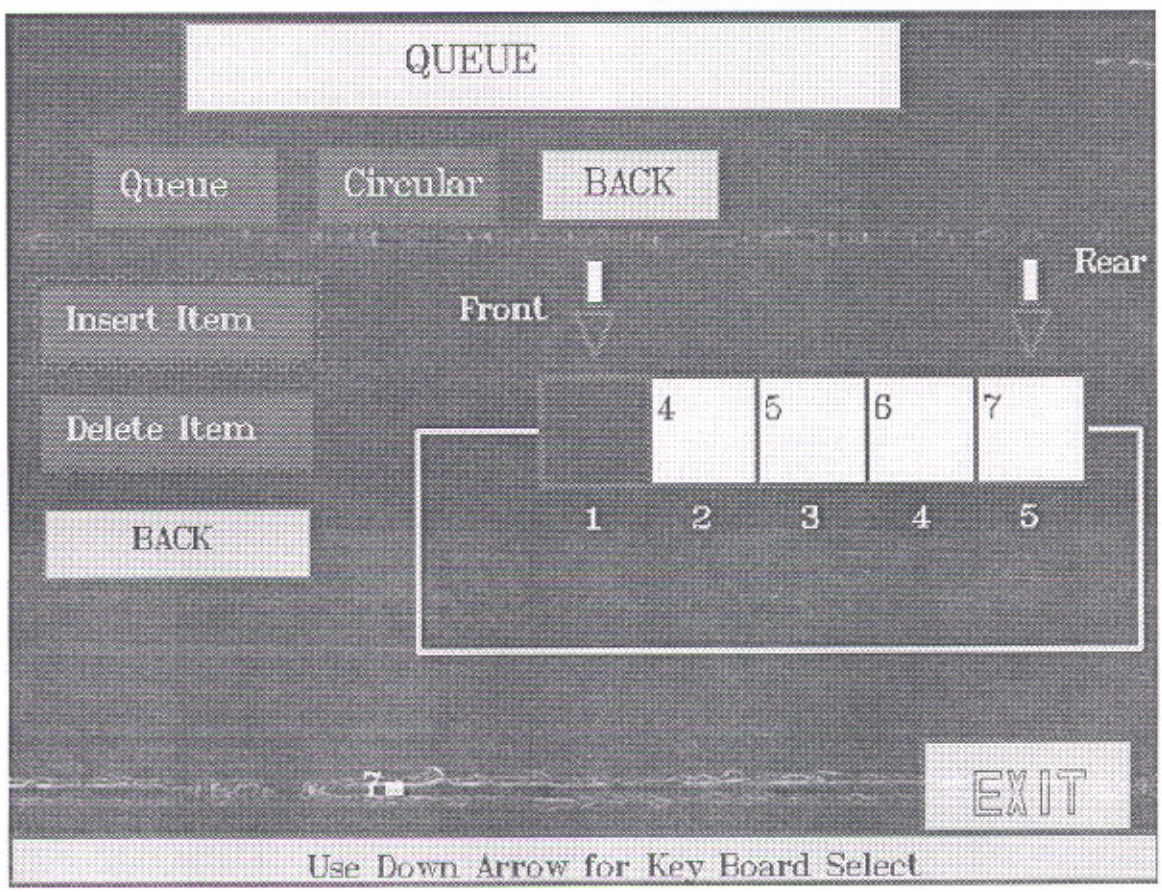

SCREEN-13



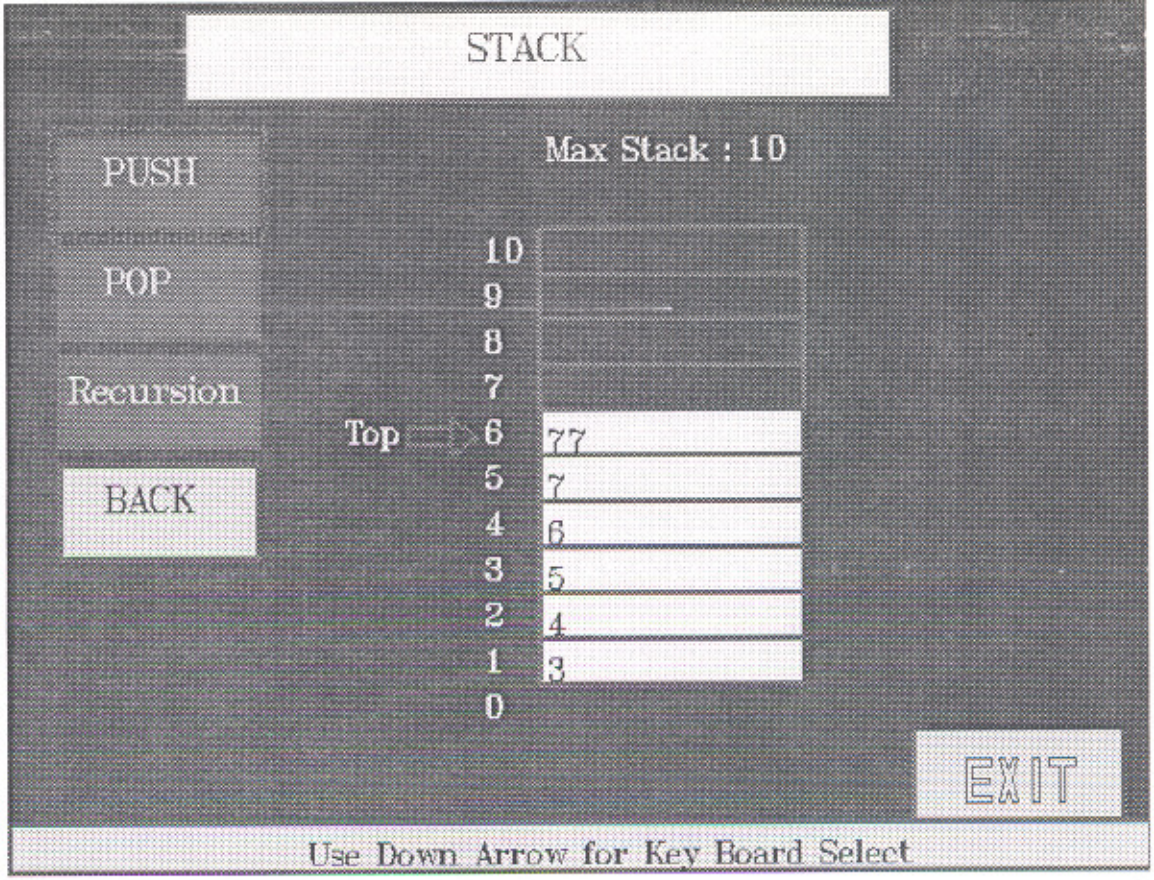

SCREEN-14



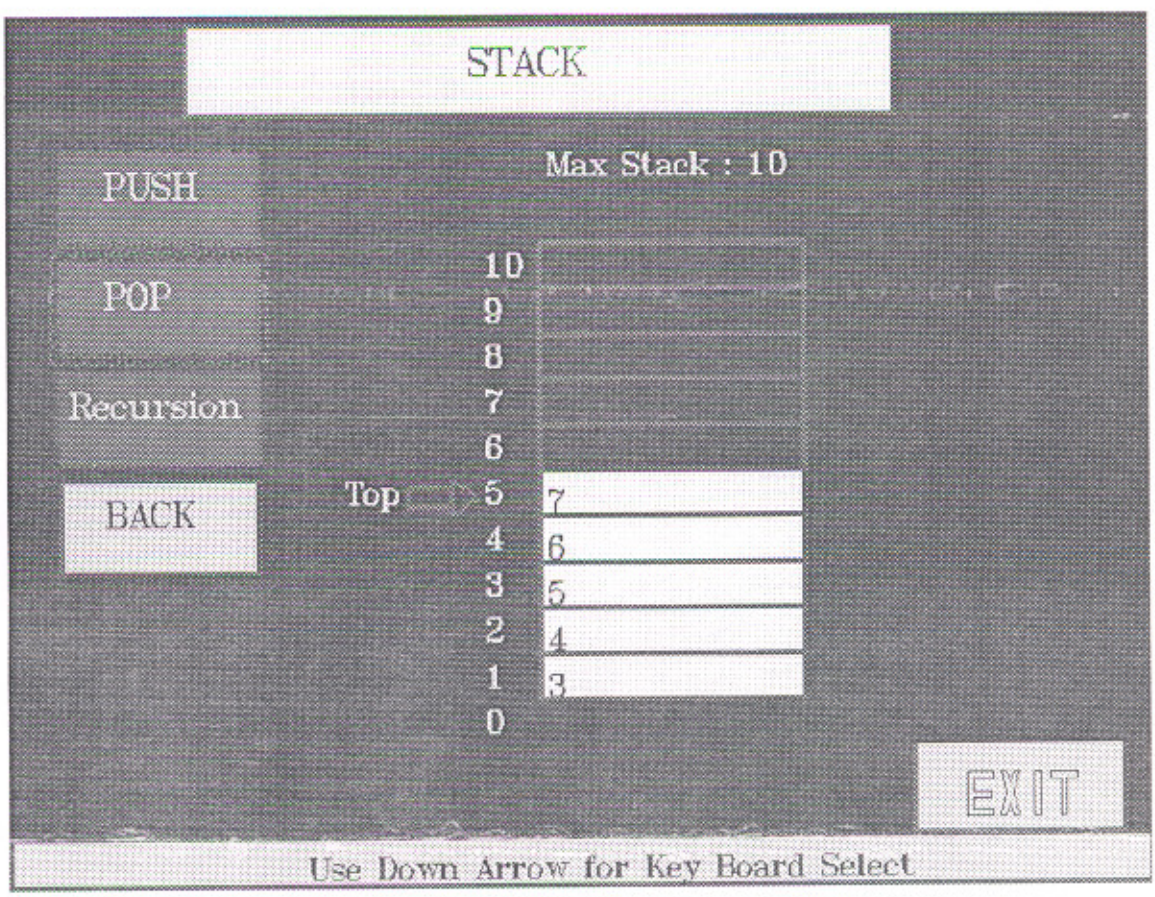

SCREEN-15



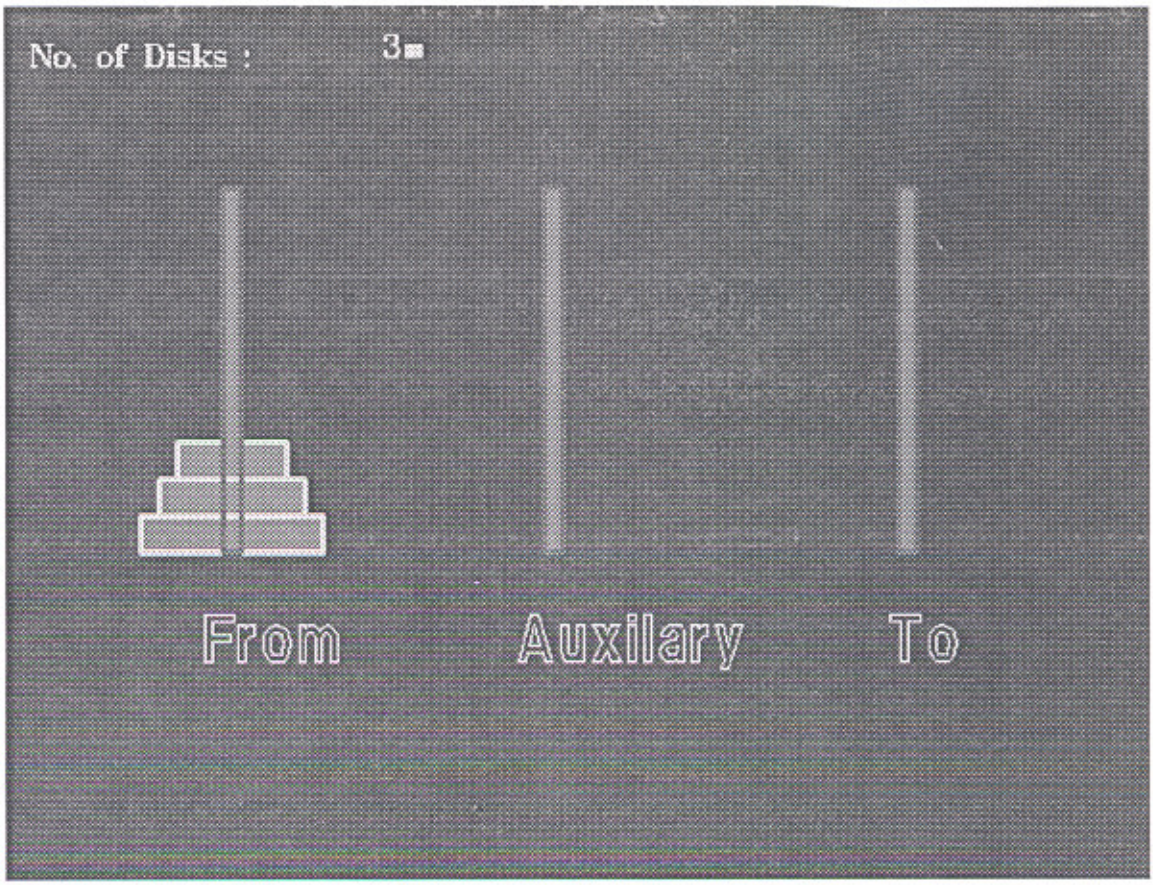

SCREEN-16



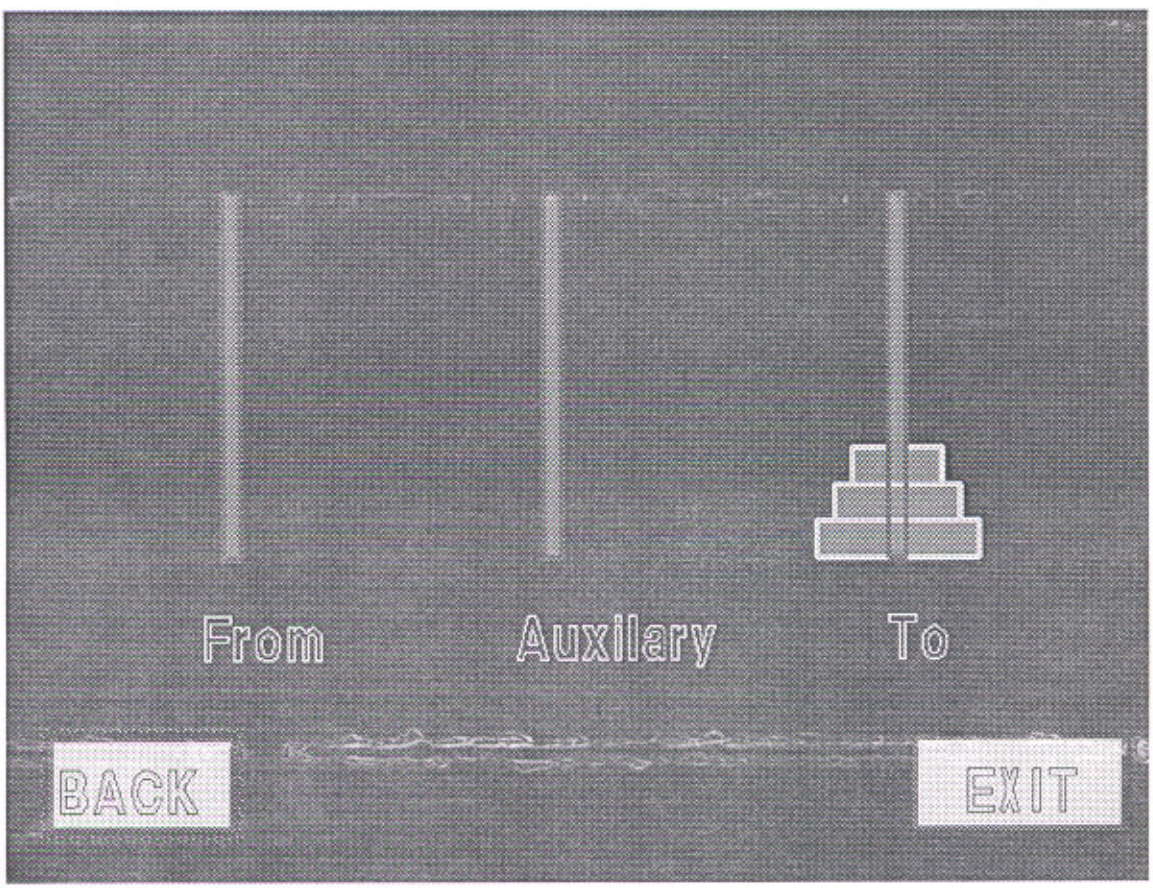

SCREEN-17



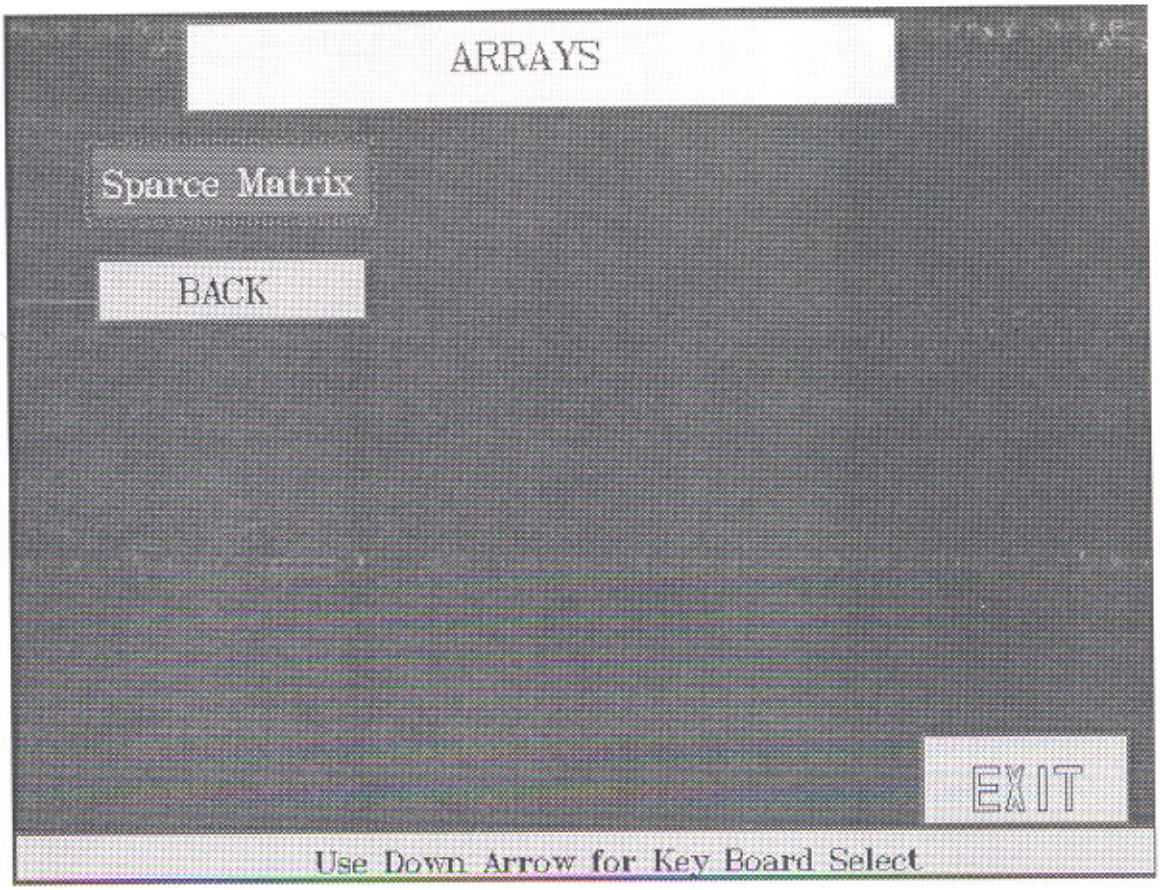

SCREEN-18



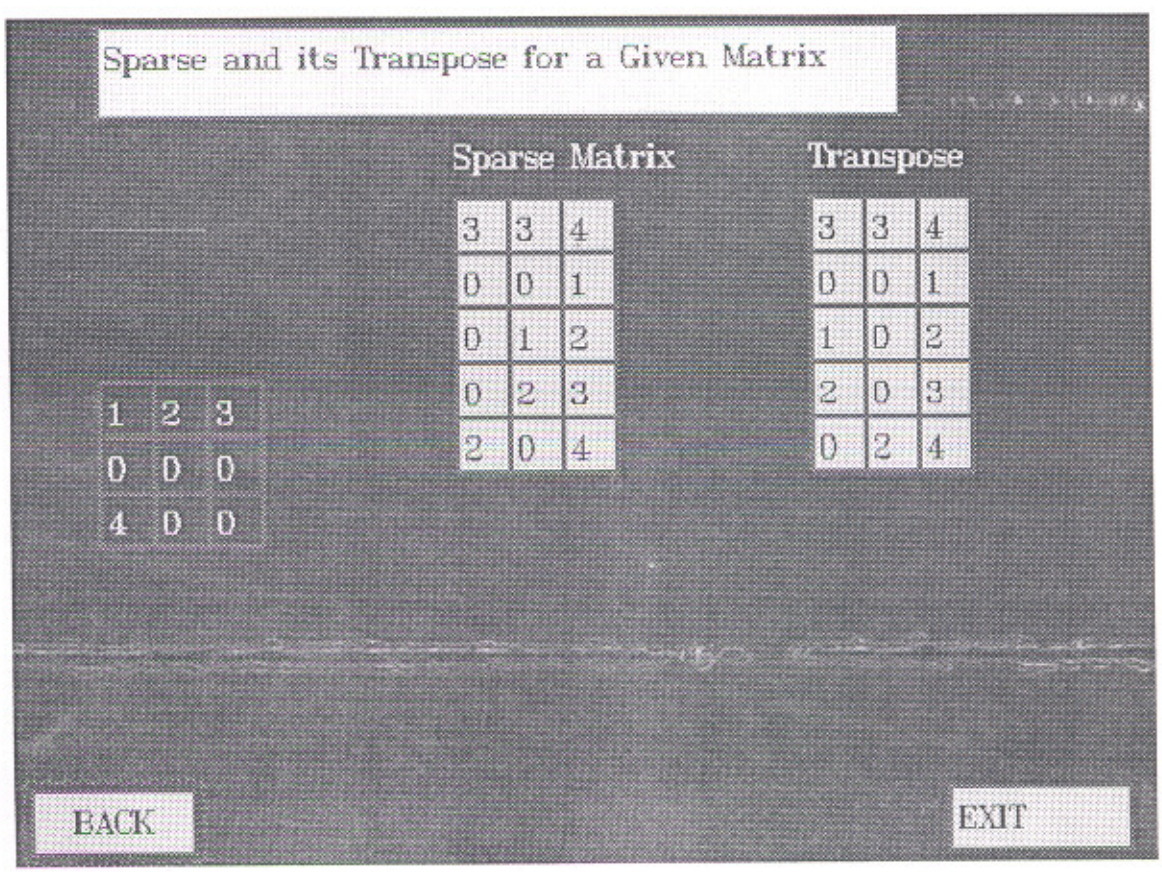

SCREEN - 19




SCREEN-20




SCREEN-21




SCREEN-22

# CHAPTER-5 Implementation of Non-linear Data Structures

Screen-1 :- By moving the down arrow of the keyboard or mouse pointer on to *non-linear* option we get those 3 options namely *tree, graph, back.* Currently the *tree* is highlighted and selected.

After pressing the Enter against the text box *'tree'*, the screen-2 will be displayed. This screen has three sub menus as *binary tree, general tree* and the *back* to go back to the *non-linear* menu.

Screen-3 :- From the previous menu we have selected *binary tree* option. The different options under this are *create tree*, and different traversal methods, like *inorder, preorder, postorder* and *back.* Now *create tree* option is selected to go on to the next screen

Screen-4 :- In this screen the *binary tree* has been drawn. The maximum depth we can give is up to 4. So, the number of nodes which can be displayed on the screen are $2^4-1 = 15$, since it is a binary tree. Enter on the button *back* to go back to the *binary tree* menu. Now we can traverse the tree using different traversal methods as follows :

**Binary tree traversals :-**

There are three ways of traversing a binary tree T with root R.

1. Preorder :- In this method of traversing

   a) Process the root R.

   b) Traverse the left sub tree of R in Preorder.

   c) Traverse the right sub tree of R in Preorder.

2. In order :- In this method of traversing

   a) Traverse the left sub tree of R in In order.

   b) Process the root R.

c) Traverse the right sub tree of R in In order .

3. Post order :-

a) Traverse the left sub tree of R in Post order.

b) Traverse the right sub tree of R in Post order

c) Process the root R.

The above algorithms are called as recursive algorithms and a stack is used to implement the traversing.

Screen-5 :- This shows a *general tree* and its *properties* are listed. When a user Enters on *depth* box then the corresponding depth of the tree is displayed. Similarly , *the levels, siblings* and the *root* will be displayed.

Screen-6 :- This screen is regarding the *graph* and its matrix representation, like *adjacency, path* and the *incidence* matrices. In this graph there are five vertices and 6 edges and the degree of this graph is 3. The fourth option is *graph*. Using this option any graph can be drawn i.e. the generalization of the graph.

Screen-7 :- Here the first statement is the number of nodes given as the input to the graph. After this the adjacency matrix of the graph is displayed. The presence of an edge between two nodes represents a '1' in the adjacency matrix else a '0' is displayed. In the next step the coordinates of the three nodes are displayed.

Screen-8 :- In this screen the final graph will be seen as per the adjacency matrix in the screen-7. Firstly, from node'0' to node'1' and then node'0' to node'2', lastly from node'1' to node'2' the graph has been drawn.

**Matrix representation of graphs :-**

1.Incidence matrix :- It is defined as

$a_{ij}$ = 1, if $j^{th}$ edge is incidenting on $i^{th}$ vertex.

= 0, otherwise.

It is represented by the letter A(G) of a graph.

2. Circuit matrix :- It is defined as

$b_{ij}$ = 1, if $i^{th}$ circuit includes $j^{th}$ edge.

= 0, otherwise.

It is represented by the letter B(G).

3. Path matrix :- It is defined as

$p_{ij}$ = 1, if $j^{th}$ edge lies in $i^{th}$ path

= 0,otherwise.

It is represented by the letter P(vi,vj). Another name for the path matrix is" Reachability matrix ".

4. Adjacency matrix :- It is defined as

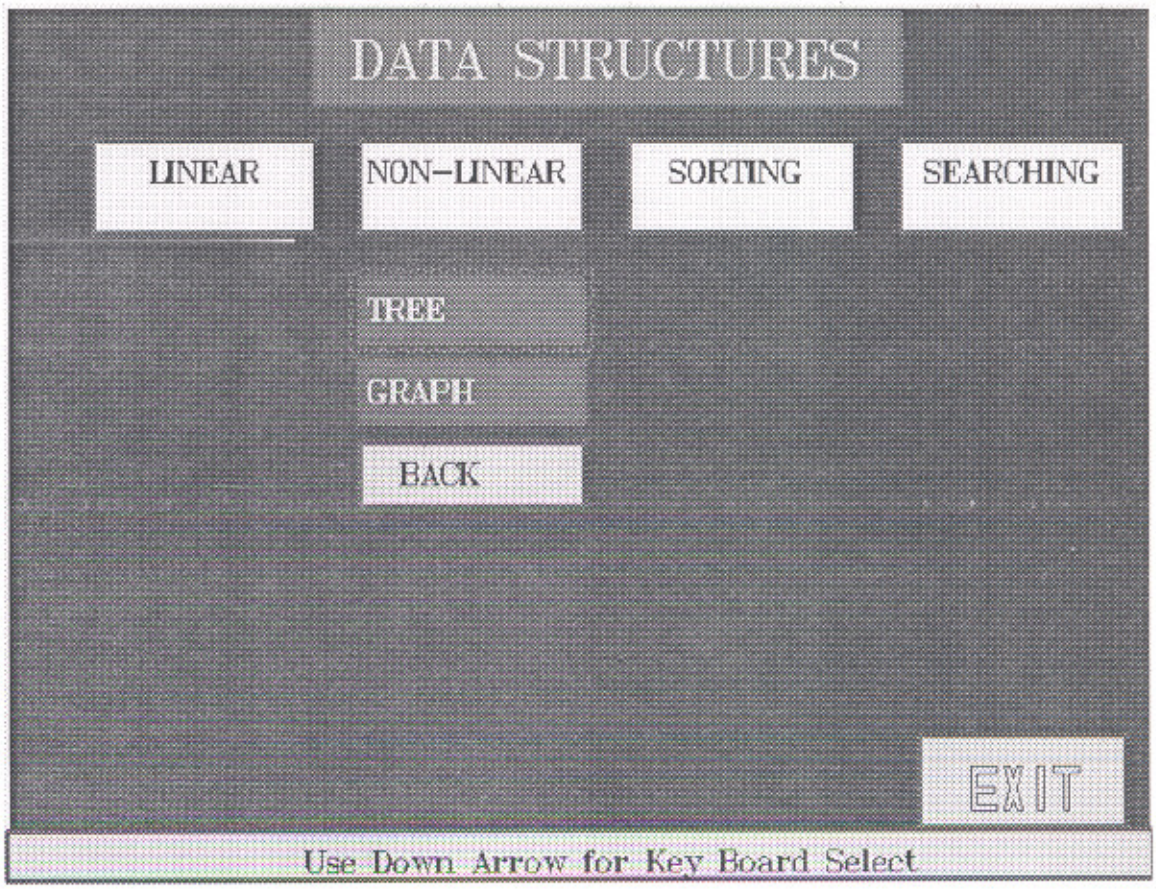
$x_{ij}$ = 1, if there is an edge between $i^{th}$ and $j^{th}$ vertices

= 0, otherwise.

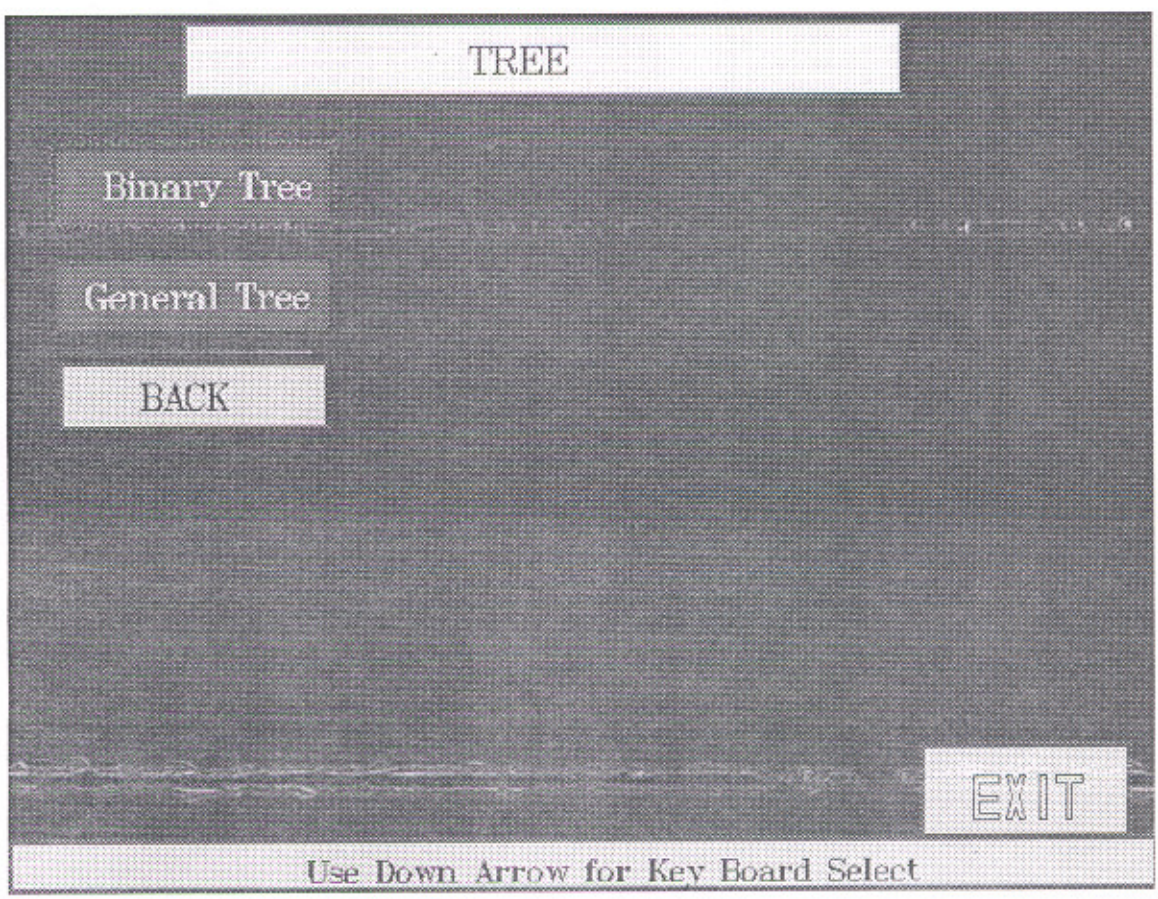
It is represented by the letter X(G). Another name for the Adjacency matrix is " Connection matrix ".

Screen-9 :- This is an *8-queens* problem solved using the *backtracking* method. There are 8-queens which are placed on an 8x8 chessboard so that no two queens are in the same row, column, and the same diagonal, i.e. these 8-queens are non attacking queens. This is one of the solution to the 8-queens problem.

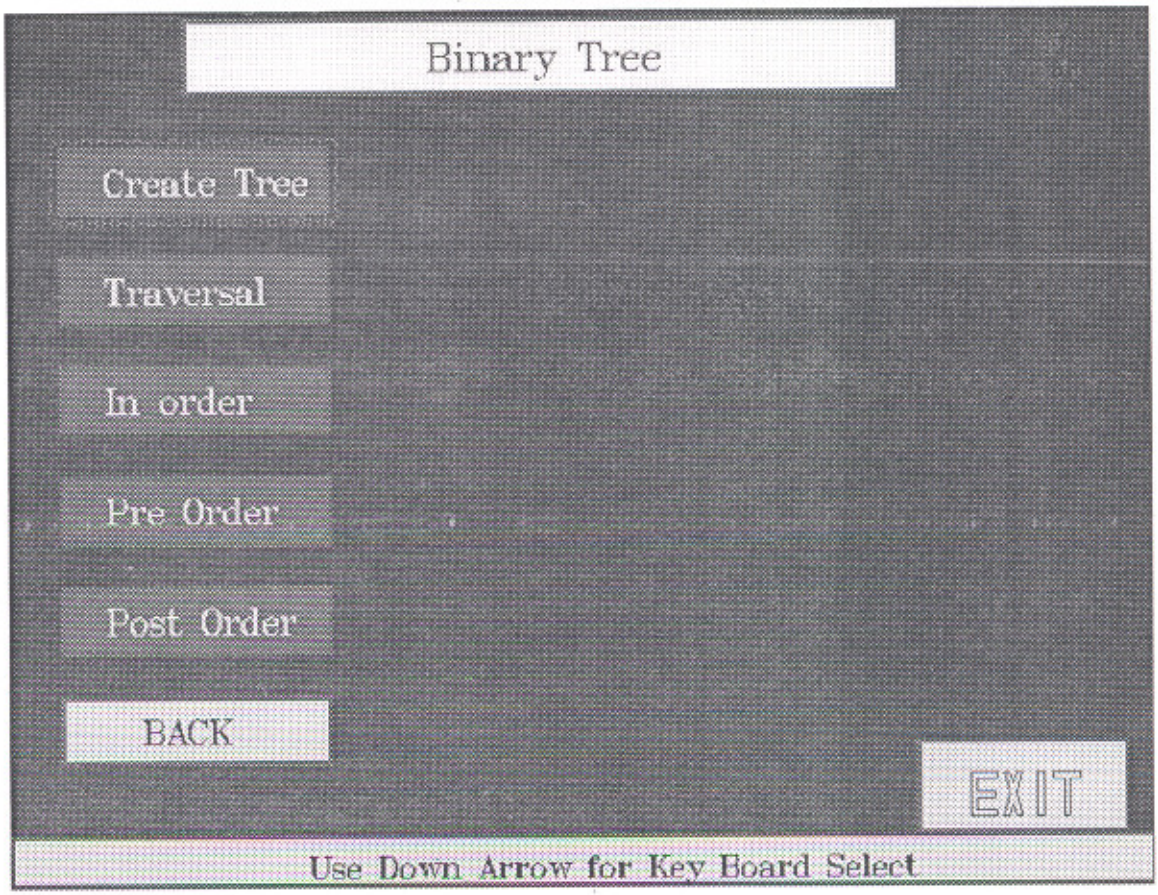
Thapar Institute of Engg. & Tech.
PATIALA-147001
CENTRAL LIBRARY
Acc. No. 91648 Dt. 19/4/2001



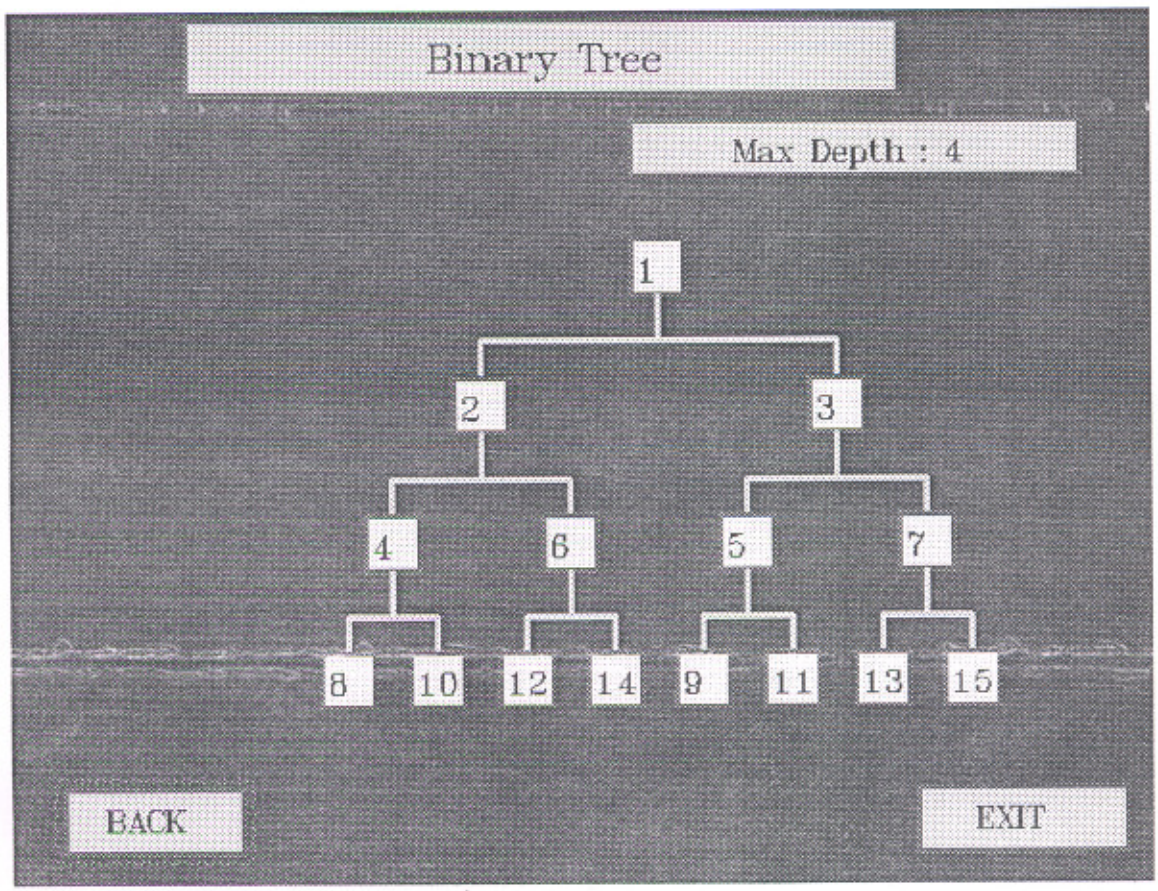

SCREEN-1

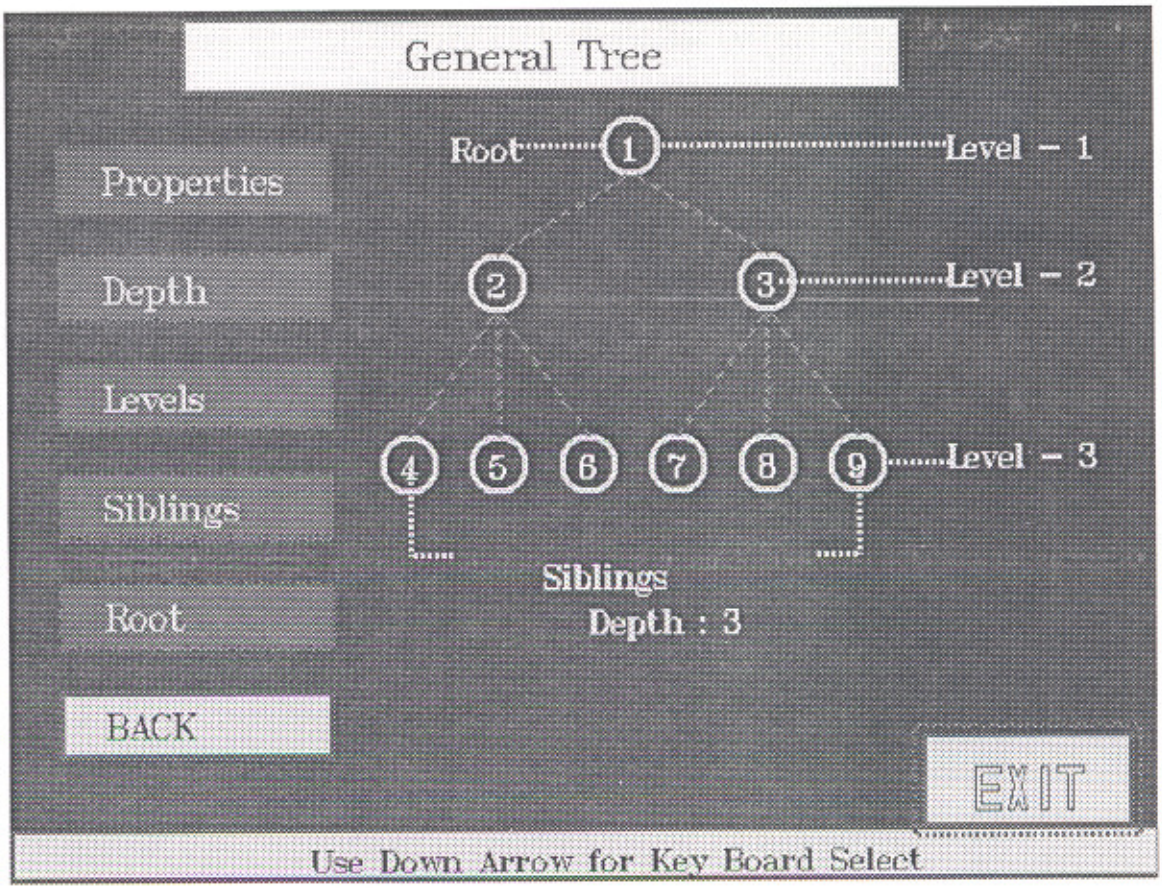



SCREEN-2

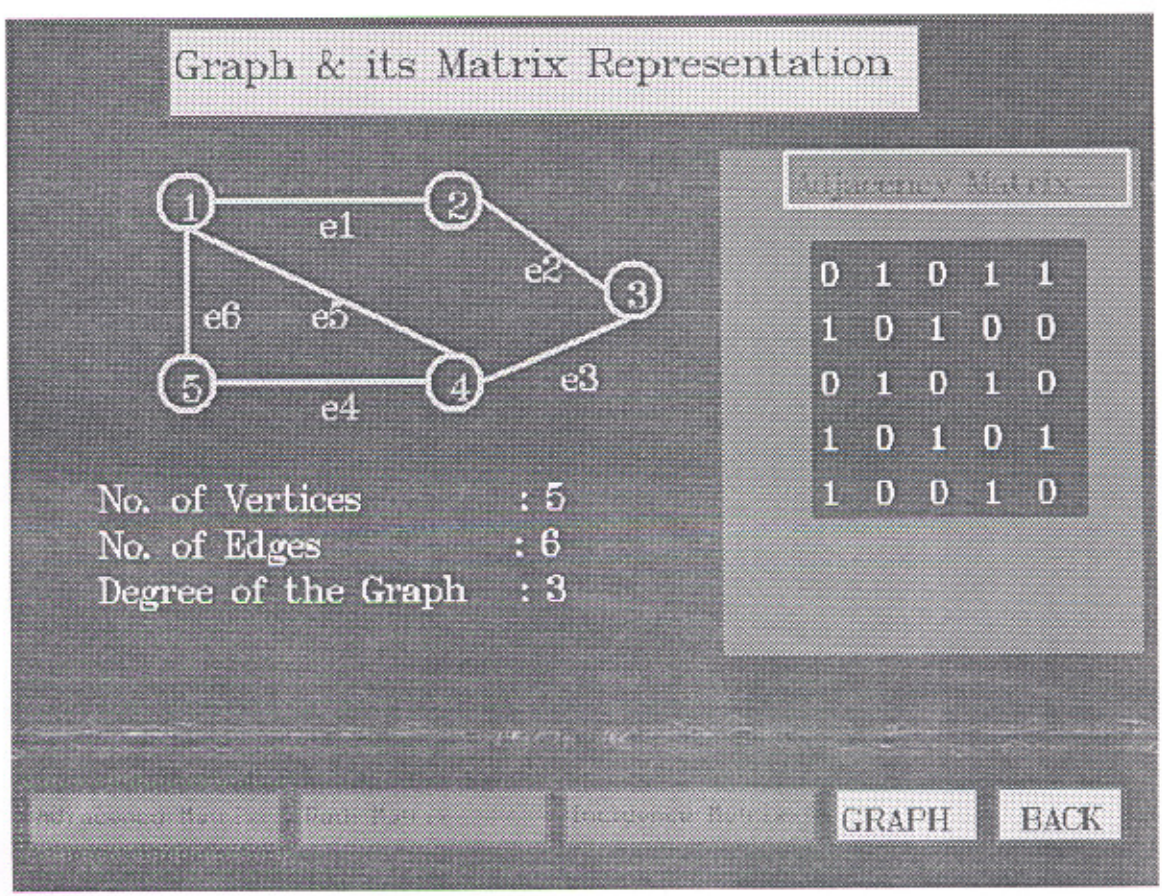

Binary Tree
Create Tree
Traversal
In order
Pre Order
Post Order
BACK
EXIT
Use Down Arrow for Key Board Select

SCREEN-3

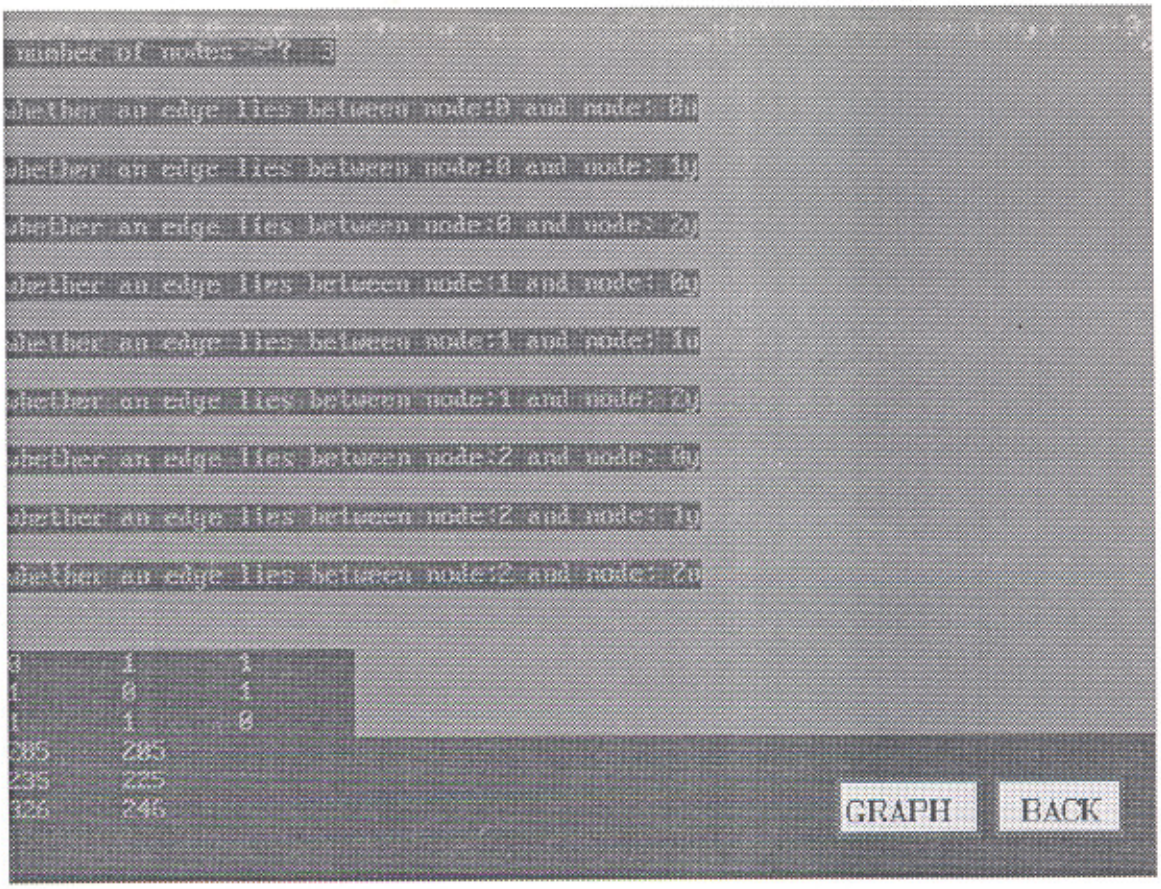

Binary Tree
Max Depth : 4
1
2
3
4
6
5
7
8
10
12
14
9
11
13
15
BACK
EXIT



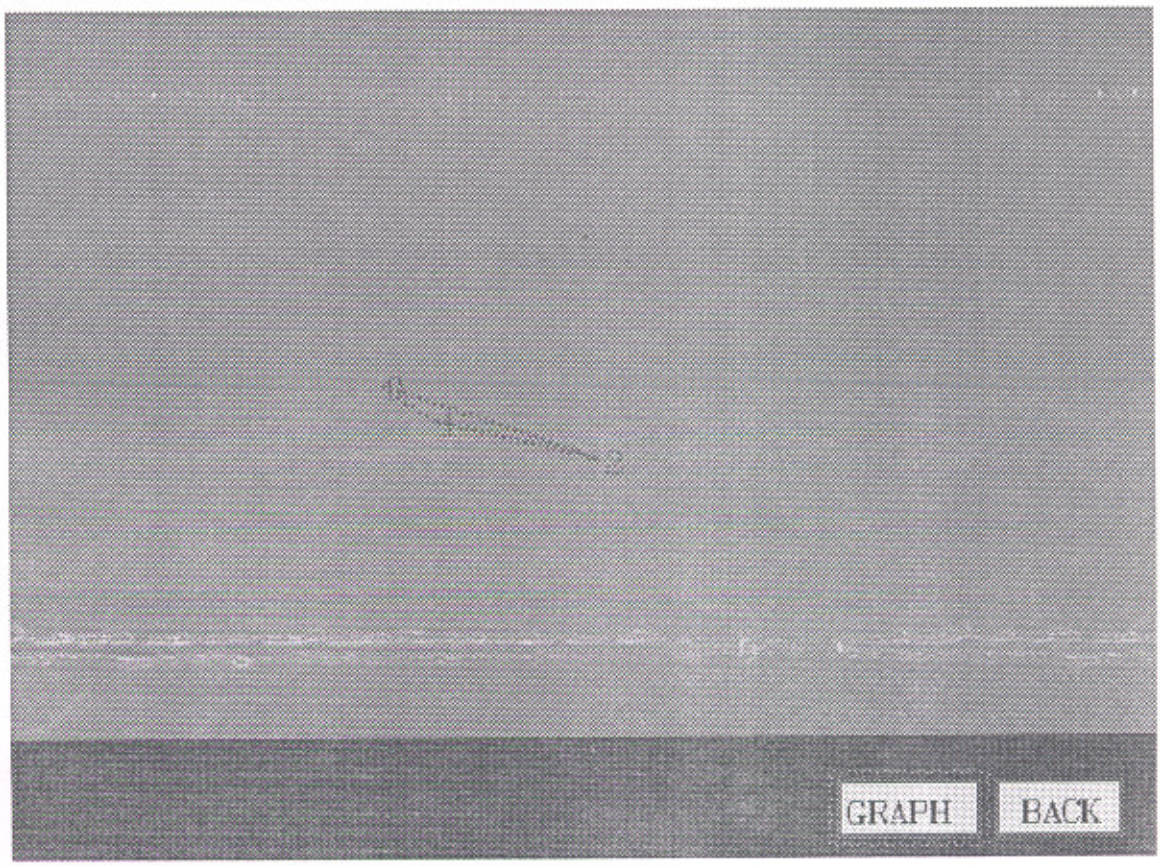

SCREEN-5



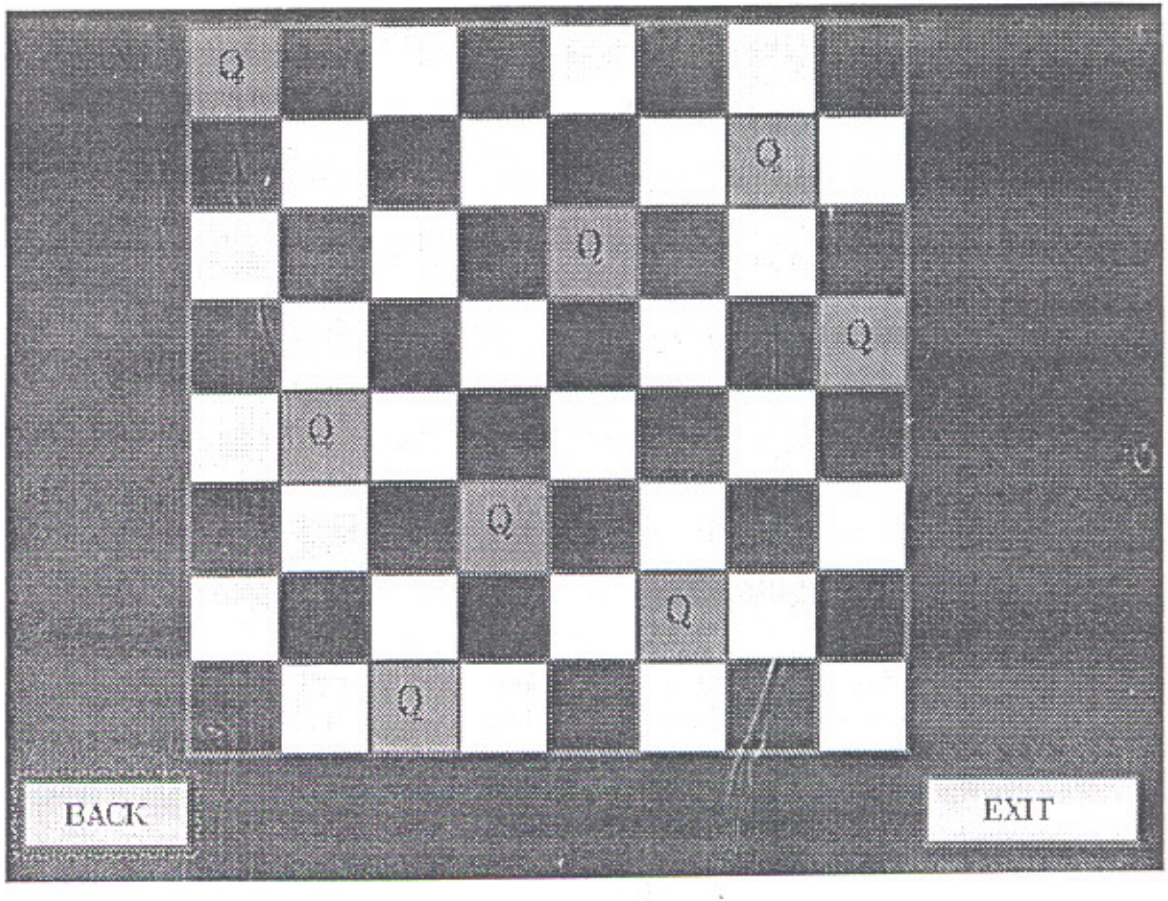

SCREEN-6




SCREEN-7




SCREEN-8




SCREEN-9

# CHAPTER-6 Sorting and Searching methods

Screen-1 :- This screen can be selected using the down arrow movement on the main menu. If a user selects a *sorting* option then the different sorting methods will be displayed on the screen, which are *linear , bubble, insertion, quick, merge* and *heap* sorting methods. Using the down arrow key we select one of the sorting methods. Currently the *linear sort* is selected.

Screen 2:- when a user Enters against *linear sort* which is the currently selected option, then this screen will be displayed. The user has been asked to enter the numbers from the keyboard into the array as shown in the screen-2. After entering the numbers into the array there will be options *back* and *sort. Back* is used to go back to the sorting methods menu. *Sort* is used to sort the elements. The location of element in *temp* is also shown for each element as per the algorithm of the linear sort.

Screen-3 :- This is the *bubble sort* menu as the heading is shown on top of the screen. Here also the user has been asked to enter the elements , if a user presses a 'n' key then the entering of numbers will be stopped and the *back, sort* options will be displayed. As in the screen-2 the location of the element in the *temp* will be shown for each element during the comparison.

Screen-4 :- In this screen of *quick sort* the user will enter the numbers as input, from that list one pivot point will be selected and the partitioning will be done. So, there is no temp in this screen. This is faster than all other sorting methods since the algorithm is a divide and conquer technique.

Screen-5 :- This sorting method is like the linear and bubble where there is a concept of temp. During each comparison the location of *temp* is shown in the temp text box. After all the numbers have been entered as input, we have to press the *sort* button to sort the elements. This is

very good sorting method if the input is less.

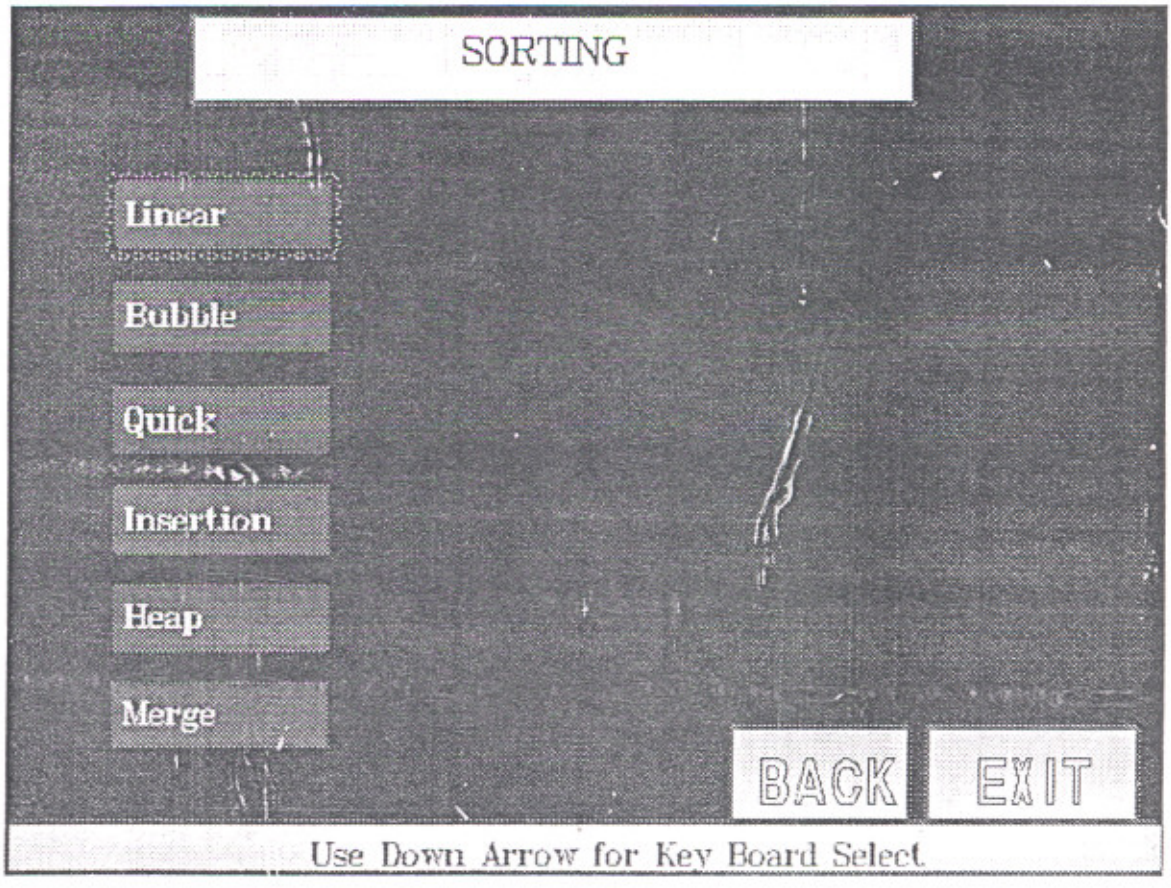
Screen-6 :- The heading of the *merge sort* is displayed on the top of the screen. This is also a divide and conquer technique. We assume that the numbers to sorted are in non-decreasing order. Given array has been split into 2 sets and each set is individually sorted to produce a single sorted sequence. Finally we combine the 2 sets into one using merging.

Screen-7 :- This is about the *heap sort*. Here the user will enter the elements in the array and after the completion of entering the elements the there will be 2 options on the screen, which are *back* and *sort*. The sort will be selected to sort the elements. The screen shows the elements after applying the sorting method.

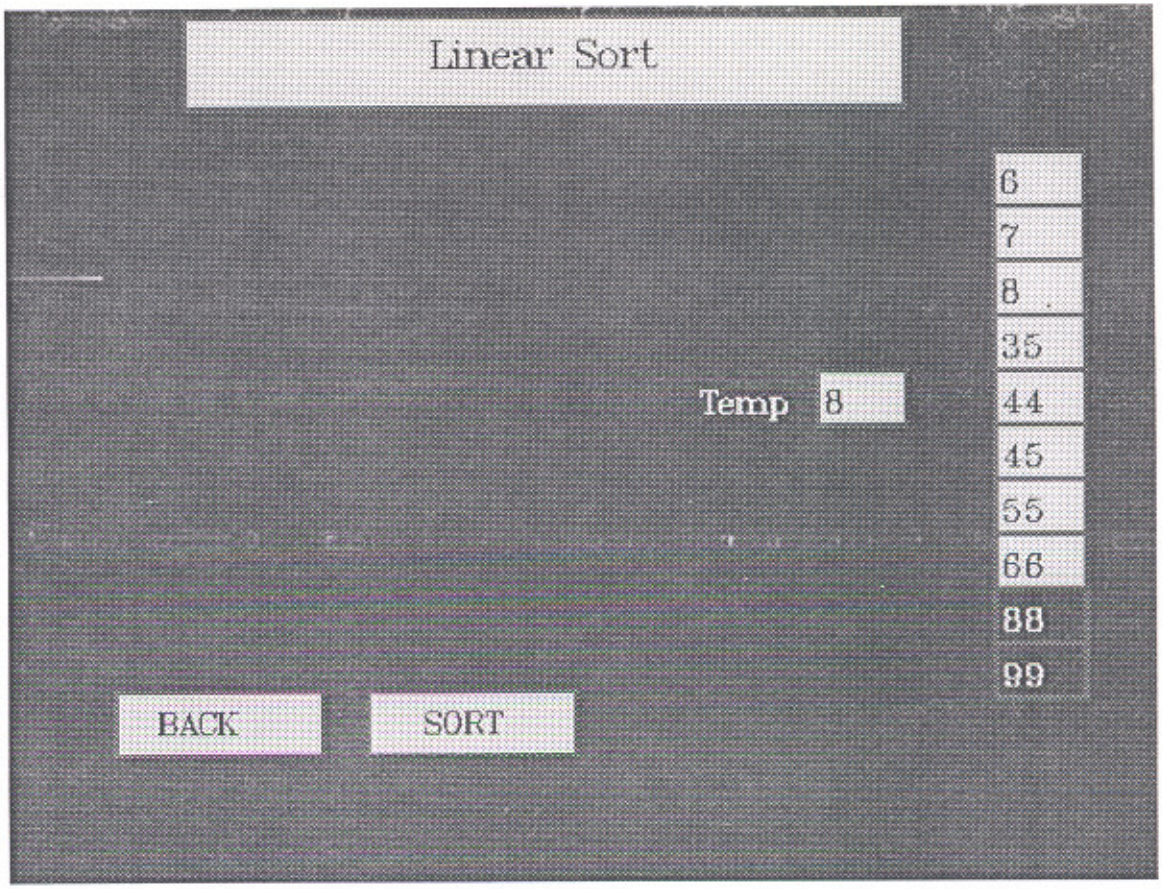
Screen-8 :- This screen can be selected from the main menu of data structures . There are *linear* and *binary search* options in the *searching* methods. *Back* option is used to go back to the main menu of the searching methods. To select one of the searching methods one can use the down arrow or the mouse pointer.

Screen-9 :- This is the *linear search* screen. The user first enter the input into the array, then the user has been asked to search a particular element in the list. The element to be searched is compared with all the items in the list. If the matching is true then the corresponding location will be displayed and giving the message like 'item is found' also seen on the screen. In this way we can search any element in the list.

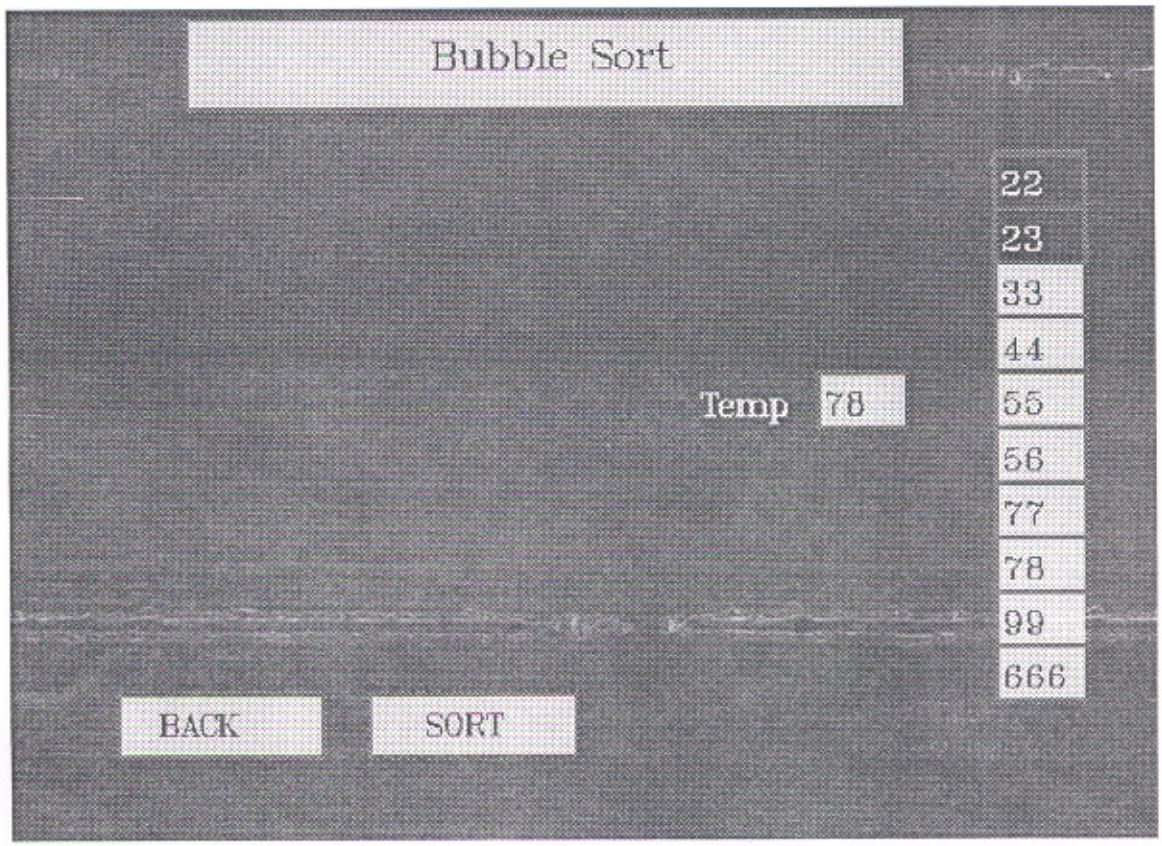
Screen-10 :- This is the final screen and it is the *binary search* algorithm. Here the numbers have to be entered only in the ascending order, since it is a limitation. As per the algorithm the begin will be assigned to '1', end will be assigned to last element in the list. Mid =(beg + end)/2. If the item is in the upper half of the list then the end will take the value of Mid-1 and the procedure will be repeated until the required item is found.


SORTING
Linear
Bubble
Quick
Insertion
Heap
Merge
BACK
EXIT
Use Down Arrow for Key Board Select

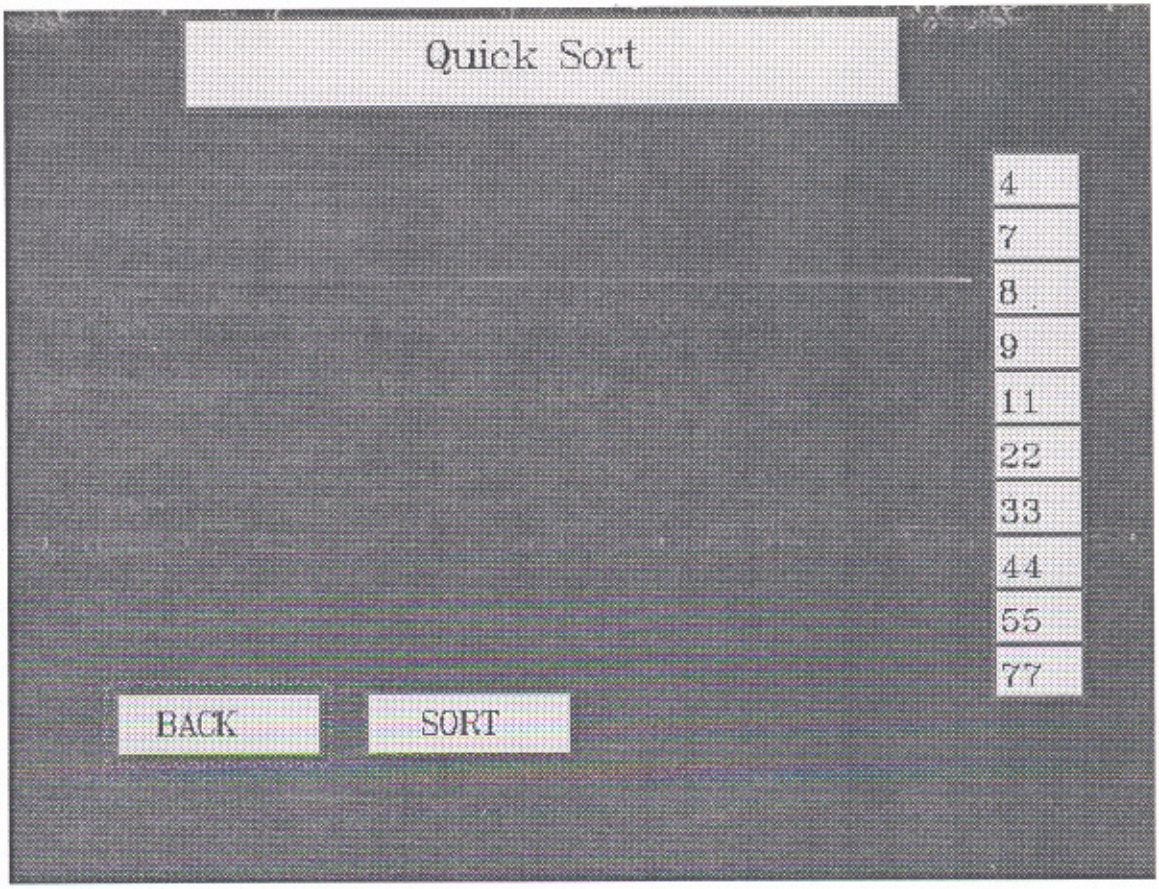
SCREEN-1



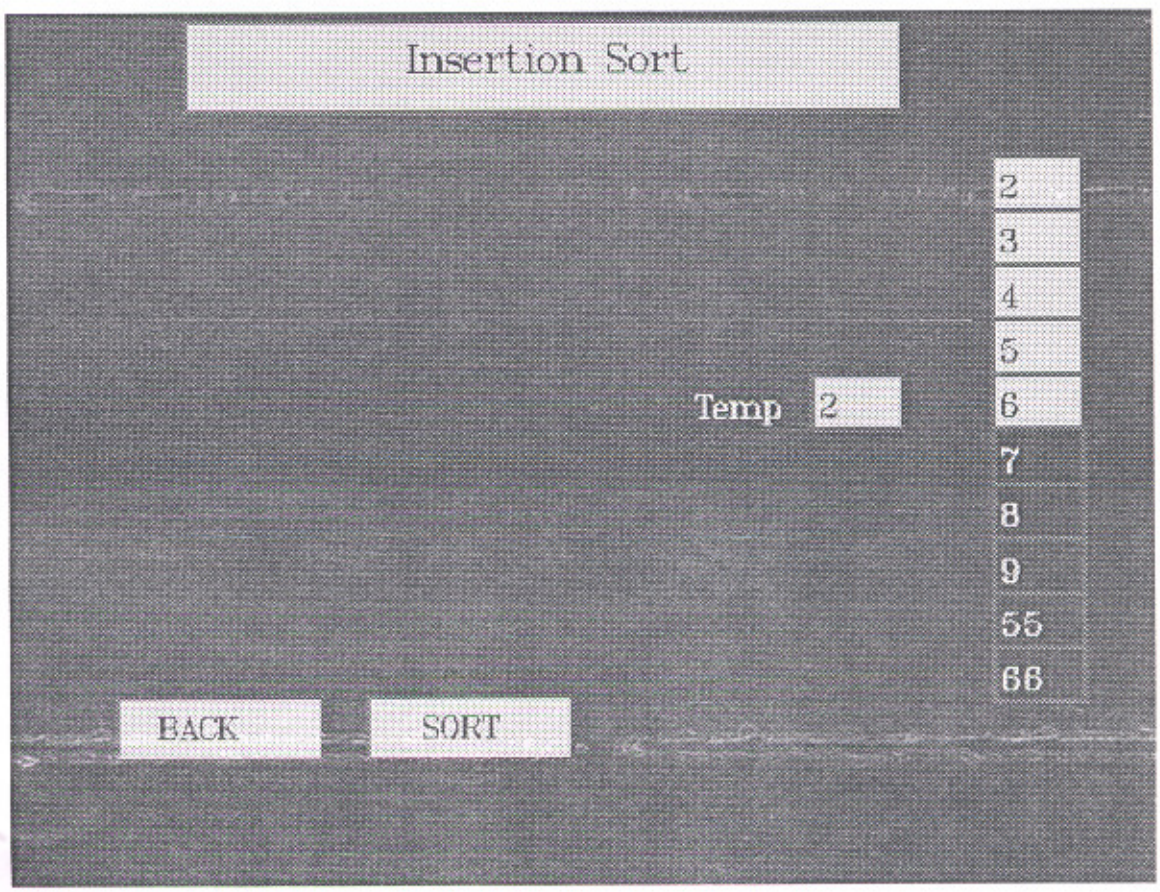

SCREEN-2



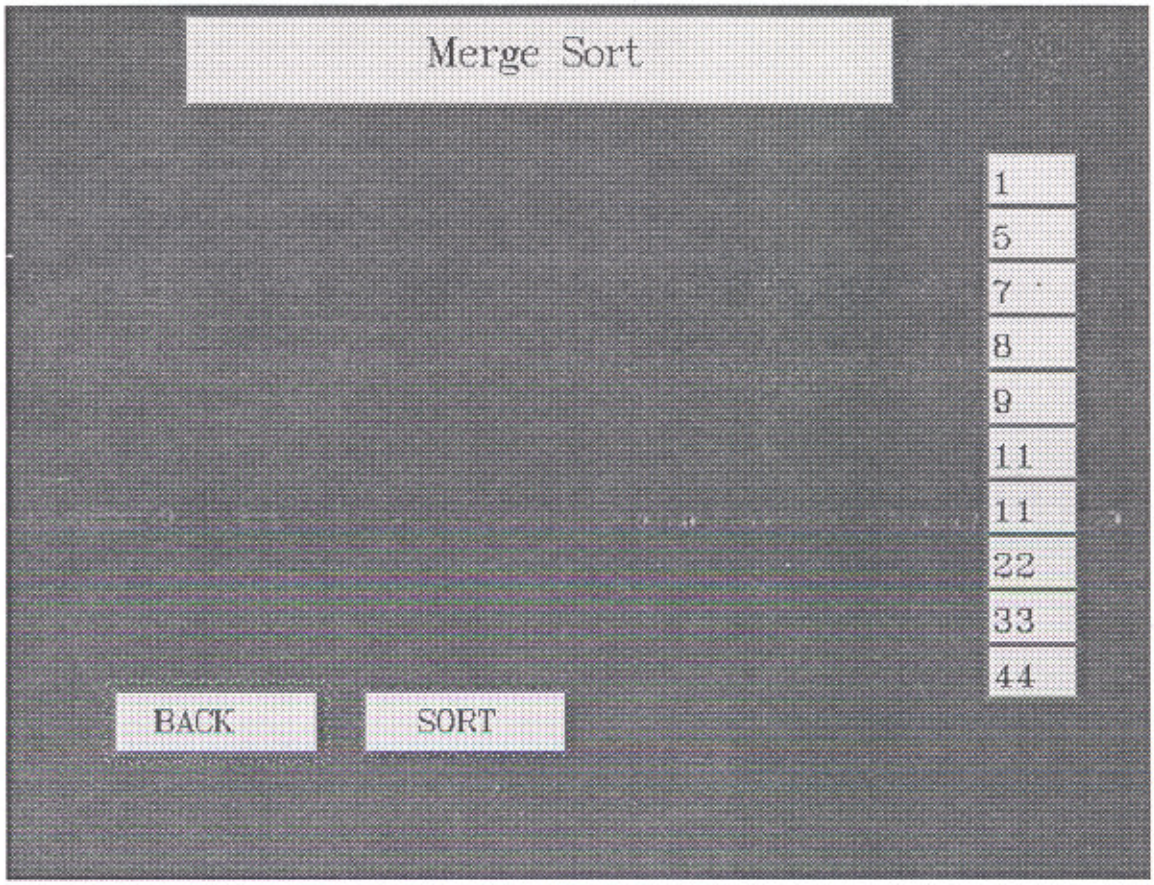

SCREEN-3



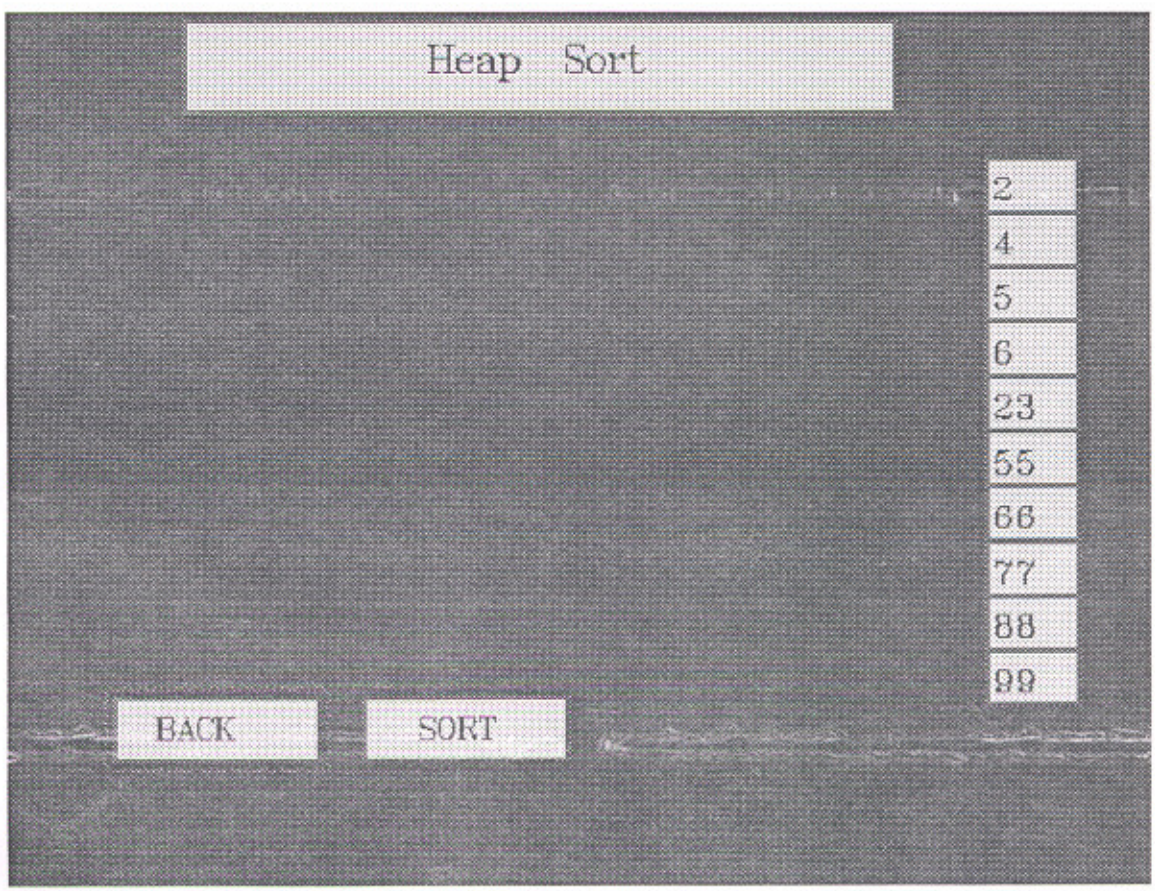

SCREEN-4



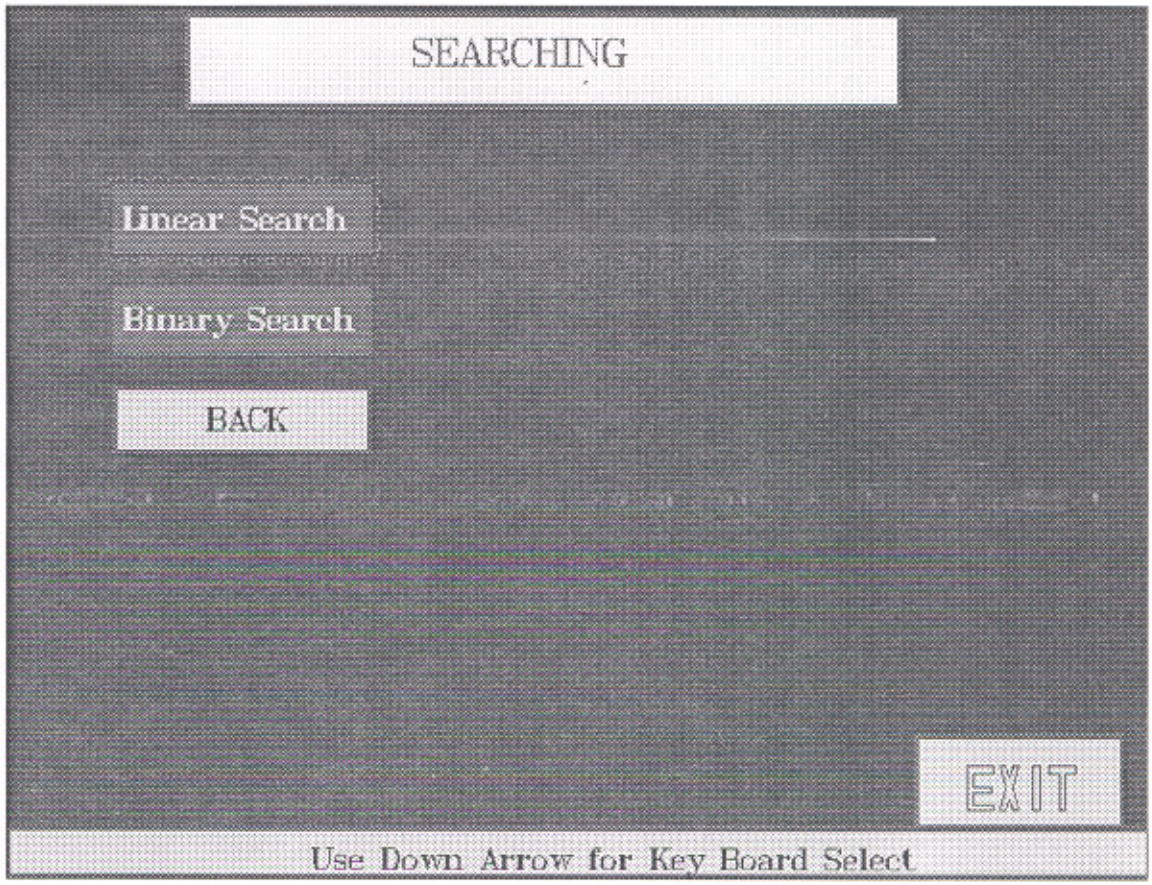

SCREEN-5



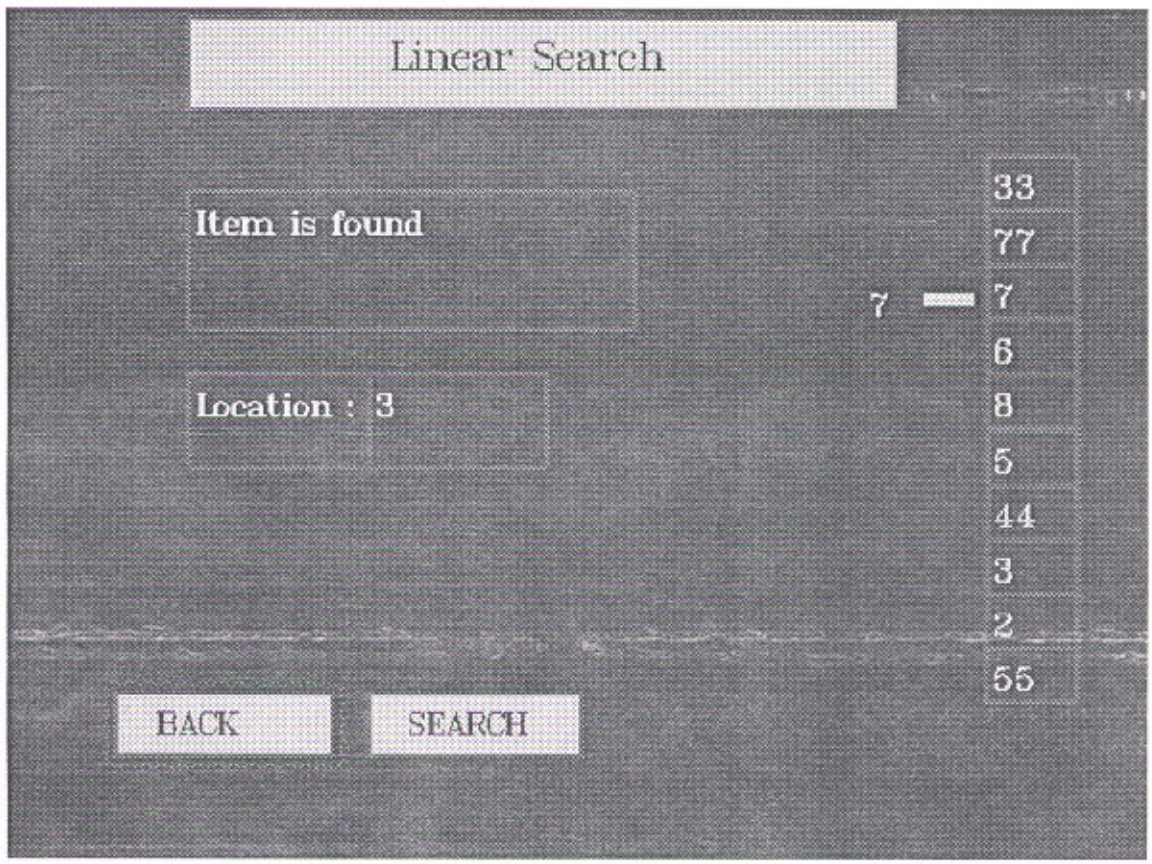

SCREEN-6

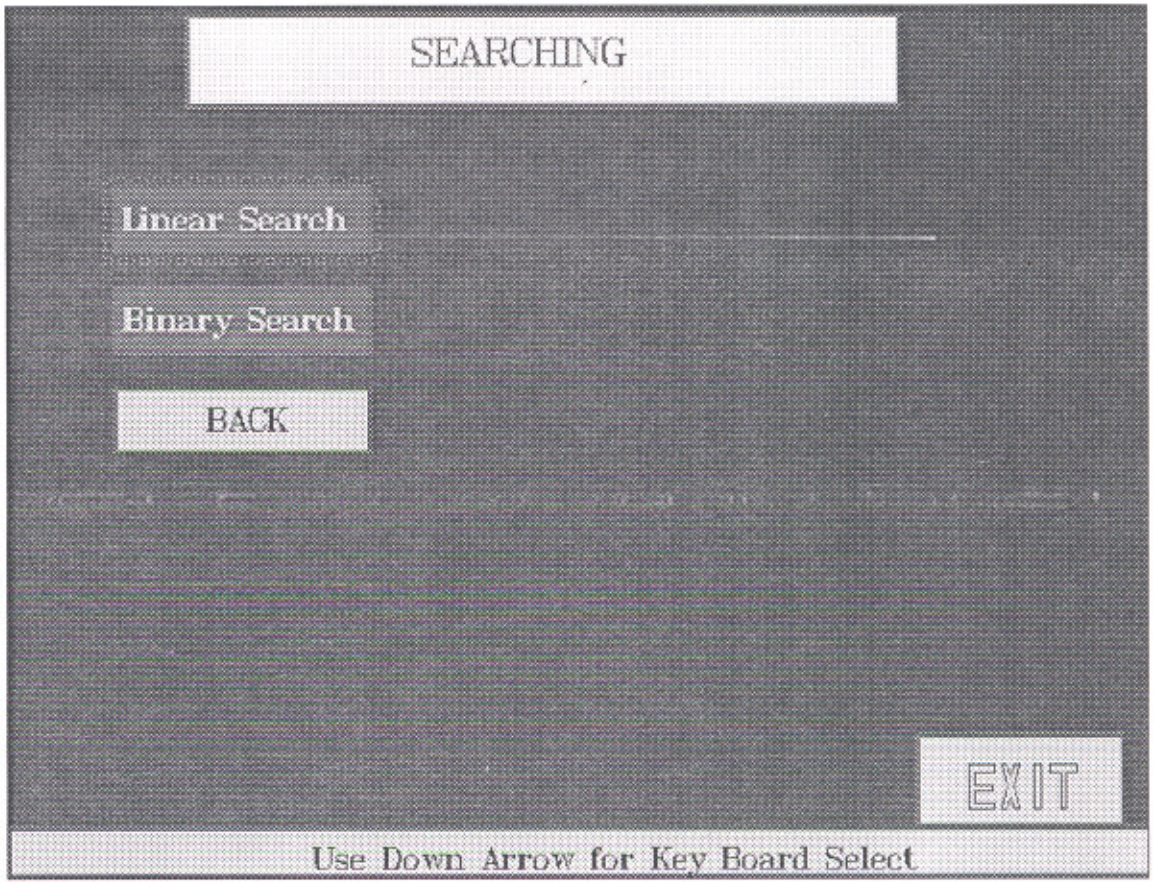



SCREEN-7

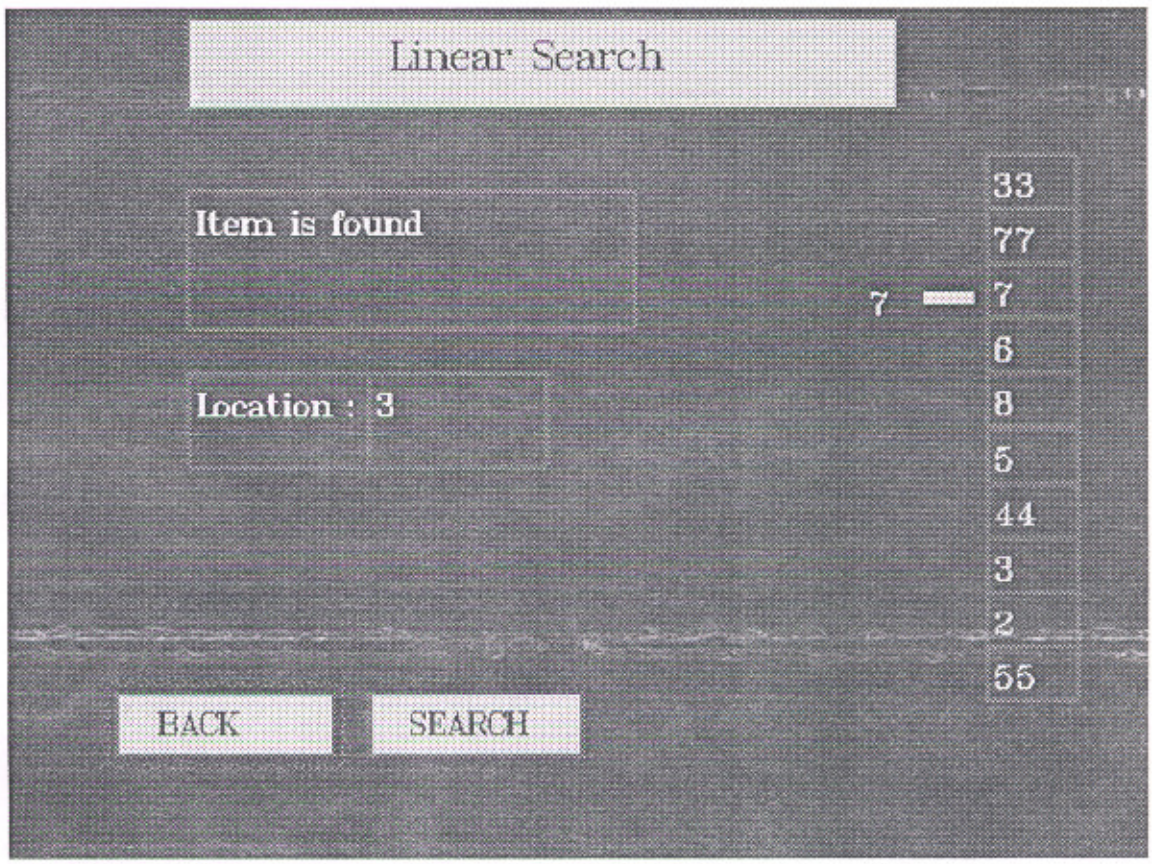

SEARCHING
Linear Search
Binary Search
BACK
EXIT
Use Down Arrow for Key Board Select

SCREEN-8

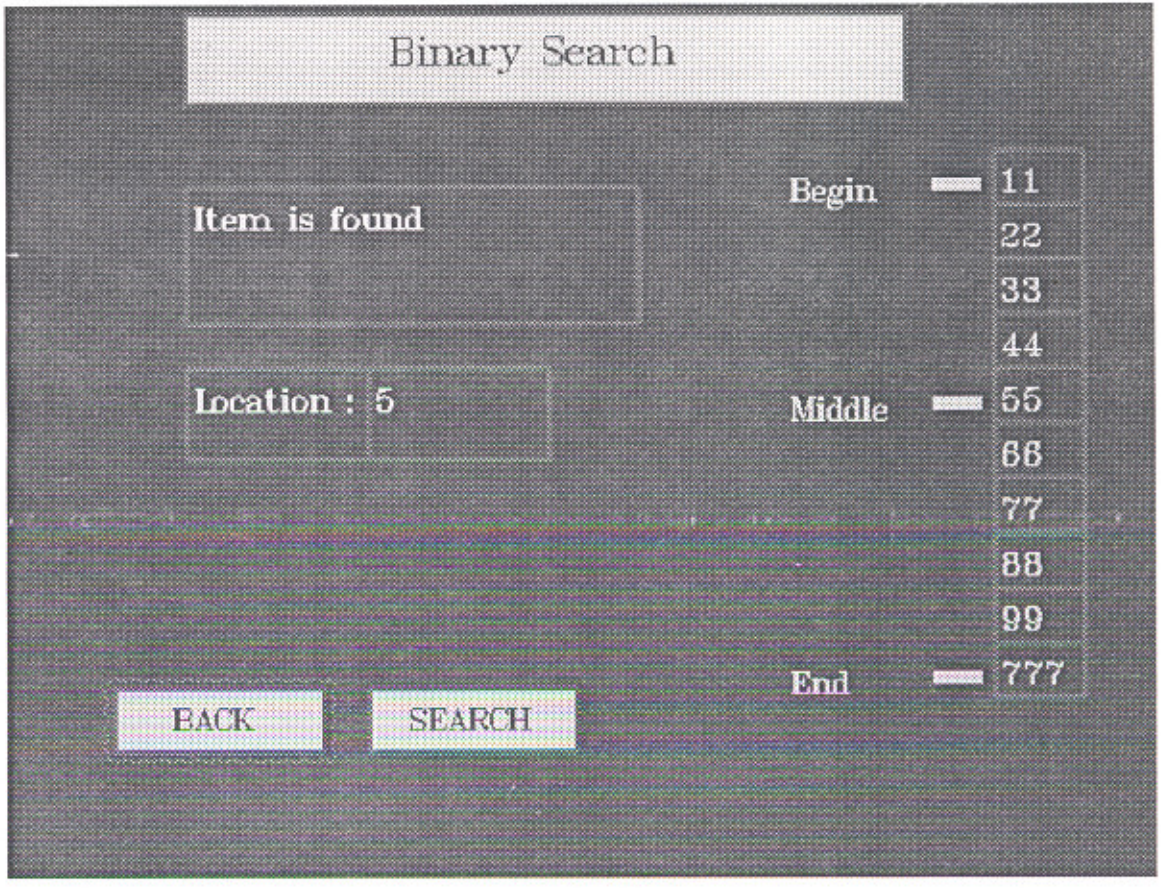

Linear Search
Item is found
Location : 3
7
33
77
7
6
8
5
44
3
2
55
BACK
SEARCH




SCREEN-8




SCREEN-9


Binary Search
Item is found
Location : 5
Begin
Middle
End
11
22
33
44
55
66
77
88
99
777
BACK
SEARCH

SCREEN-10

## References :-


1. *Seymour Lipschutz,* (1986), " Theory and Problems of Data Structures ", $M_c$ Graw-Hill Publications private limited, New Delhi.
2. *Ellis Horowitz, Sartaj Sahni and Dinesh Mehta,* (1995), " Fundamentals of Data Structures in C++ ", Galgotia publications private limited, New Delhi.
3. *Aaron M.Tenenbaum and Moshe J. Augenstein, Yedidyah Langsam,* (1999), " Data Structures using C and C++ ", Prentice-Hall of India private limited, New Delhi.
4. *Robert L.Kruse, Bruce P.Leung and Clovis L. Tondo,* (1991), " Data Structures and Program Design in C ", Prentice-Hall of India private limited, New Delhi.
5. *Ellis Horowitz and Sartaj Sahni,* (1984), " Fundamentals of Computer Algorithms ", Galgotia Publications private limited, New Delhi.
6. *Mark Allen Weiss* , (1999), " Data Structures and Algorithm Analysis in C++ ", Addison-Wesley Publishing Company.
7. *Yashawant Kanetkar,* (1999), " Let Us C++ ", BPB Publications, New Delhi.
8. *Yashawant Kanetkar,* (1998), " Graphics Under C ", BPB Publications, New Delhi.
9. *Meeta Gandhi, Tilak Shetty and Raj V Shah,* (1992), " The 'C' Odyssey C++ and Graphics ", BPB Publications, New Delhi.
10. *E. Balagurusamy,* (1999), " Object oriented programming with C++ ", Tata $M_c$ Graw-Hill Publishing company private limited, New Delhi.
11. *Angela B.Shiflet,* (1997), " Elementary Data Structures with Pascal ", West-Publishing Company, USA.
12. *Narsing Deo,* (1984), " Graph Theory with Applications to Engineering and Computer Science ", Prentice-Hall of India private limited, New Delhi.

13. *Aho, Hopcroft and Ullman,* (1999), " The Design and Analysis of Computer Algorithms", Addison-Wesley Longman, Singapore private limited.

14. *Jean Paul Tremblay and Paul G. Sorenson,* (1984), " An Introduction to Data Structures with Applications ", $M_c$ Graw-Hill Publications private limited, New Delhi.

15. *Nell Dale, Susan C. Lilly,* (1985), " Pascal plus Data Structures, Algorithms, and Advanced Programming ", Tata $M_c$ Graw-Hill Publishing company private limited, New Delhi.

16. *Niklaus Wirth,* (1988), " Algorithms + Data Structures = Programs ", Prentice-Hall of India private limited, New Delhi.

17. *Herbert Schildt,* (1998), " Teach Yourself C++ ", Tata $M_c$ Graw-Hill Publishing company private limited, New Delhi.

18. *Donald Hearn, M. Pauline Baker,* (1986), " Computer Graphics ", Prentice-Hall of India private limited, New Delhi.

19. *Rick Decker,* (1989), " Data Structures ", Prentice-Hall of India private limited, New Delhi.