LAB MANUAL OF

DATA STRUCTURES LAB

(ETCS-255)



Maharaja Agrasen Institute of Technology, PSP area,
Sector – 22, Rohini, New Delhi – 110085

(Affiliated to Guru Gobind Singh Indraprastha University, New Delhi)

INDEX OF THE CONTENTS

1. **Introduction to the Data Structures.**
2. **Hardware available in the lab.**
3. **List of practicals (as per syllabus prescribed by G.G.S.I.P.U)**
4. **Format of the lab record to be prepared by the students.**
5. **Marking scheme for the practical exam.**
6. **Steps to be followed (for each practical).**
7. **Sample programs.**
8. **List of viva questions.**
10. **Steps to be followed for advanced practicals.**

INTRODUCTION TO DATA STRUCTURES LAB

The study of data structures is an essential part of every under graduate and graduate program in computer science. Data structure is structuring and organizing data in efficient manner so that it can be accessed and modified easily. It determines the logical linkages between data elements and affect the physical processing of data. All software programs use data structures of some kind.

Data structures knowledge is required of people who design and develop systems software e.g.operating systems, language compilers and communications processors etc. Data structure is a primary factor that determines program performance.

Algorithms presented in the manual are in a form which is machine and language independent.The prerequisite for this course is a course in programming with experience using elementary features of C.

The essential goals of doing data structures lab is to acquire skills and knowledge in imperative programming.

- Have a working knowledge of basic algorithms and data structures. This includes:
    - a working knowledge of implementation of data structures as arrays and pointers
    - the use of data structures to implement abstract data types such as stacks, queues, lists, sets, trees, and algorithms such as searching, and sorting.
- Understanding standard algorithm design strategies. These are standard concepts (like ``divide and conquer'') that help to design own algorithms.
- Understanding graph algorithms.

HARDWARE AVAILABLE IN THE LAB

CPU HCL {Intel CPU (P-IV 3.0GHz.HT)
512 MB RAM/
80 GB HDD/
Intel 865 GLC M.B.
On Board sound & 3D Graphics Card
Lan card key board
Mouse
CDRW Drive
15'' Color Monitor
UPS

Printer Dot Matrix Printer , 1 LaserJet Printer1160

Software C++

DATA STRUCTURES LAB

Paper Code: ETCS-255 **L T/P**

Paper: Data Structures Lab
0 2 1

List of Experiments

1. Perform Linear Search and Binary Search on an array.
Description of programs:
a. Read an array of type integer.
b. Input element from user for searching.
c. Search the element by passing the array to a function and then returning the position of the element from the function else return -1 if the element is not found.
d. Display the position where the element has been found.

2. Implement sparse matrix using array.
Description of program:
a. Read a 2D array from the user.
b. Store it in the sparse matrix form, use array of structures.
c. Print the final array.

3. Create a linked list with nodes having information about a student and perform
I. Insert a new node at specified position.
II. Delete of a node with the roll number of student specified.
III. Reversal of that linked list.

4. Create doubly linked list with nodes having information about an employee and perform Insertion at front of doubly linked list and perform deletion at end of that doubly linked list.

5. Create circular linked list having information about an college and perform Insertion at front perform Deletion at end.

6. Create a stack and perform Pop, Push, Traverse operations on the stack using Linear Linked list.

7. Create a Linear Queue using Linked List and implement different operations such as Insert, Delete, and Display the queue elements.

8. Create a Binary Tree (Display using Graphics) perform Tree traversals (Preorder, Postorder, Inorder) using the concept of recursion.

9. Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree with the information in the tree about the details of a automobile (type, company, year of make).

10. To implement Insertion sort, Merge sort, Quick sort, Bubble sort, Bucket sort, Radix sort, Shell sort, Selection sort, Heap sort and Exchange sort using array as a data structure.

NOTE:- At least 8 Experiments out of the list must be done in the semester.

FORMAT OF THE LAB RECORDS TO BE PREPARED BY THE STUDENTS

The students are required to maintain the lab records as per the instructions:

1. All the record files should have a cover page as per the format.
2. All the record files should have an index as per the format.
3. All the records should have the following :

I. Date
II. Aim
III. Algorithm Or The Procedure to be followed.
IV. Program
V. Output

MARKING SCHEME
FOR THE
PRACTICAL EXAMINATION

There will be two practical exams in each semester.
1. Internal Practical Exam
2. External Practical Exam

INTERNAL PRACTICAL EXAMINATION

It is taken by the concerned lecturer of the batch.

MARKING SCHEME FOR INTERNAL EXAM IS:

Total Marks: 40

Division of 40 marks is as follows

1. Regularity: 25

    2. Performing program in each turn of the lab
    3. Attendance of the lab
    4. File

5. Viva Voice: 10

6. Presentation: 5

NOTE : For the regularity, marks are awarded to the student out of 10 for each experiment performed in the lab and at the end the average marks are given out of 25.

EXTERNAL PRACTICAL EXAMINATION

It is taken by the concerned lecturer of the batch and by an external examiner. In this exam student needs to perform the experiment allotted at the time of the examination, a sheet will be given to the student in which some details asked by the examiner needs to be written and at the last viva will be taken by the external examiner.

MARKING SCHEME FOR THIS EXAM IS:

Total Marks: 60

Division of 60 marks is as follows

1. Sheet filled by the student: 15
2. Viva Voice: 20
3. Experiment performance: 15
4. File submitted: 10

NOTE:

- Internal marks + External marks = Total marks given to the students
  (40 marks) (60 marks) (100 marks)
- Experiments given to perform can be from any section of the lab.

INTRODUCTION TO PROGRAMS TO BE DONE IN THE DATA STRUCTURES LAB

The programs to be done in Data Structures are divided in five sections.

The first section has the programs related to arrays and linked list which are the simplest data structures. Stacks and queues are to be implemented using arrays as well as linked list.

The second section includes the programs related to trees. In computer science, a **tree** is a widely-used data structure that emulates a tree structure with a set of linked nodes. It is a special case of a graph.. Each node has zero or more **child nodes**, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's **parent node** (or *ancestor node*, or superior). A node has at most one parent .The topmost node in a tree is called the **root node**. Being the topmost node, the root node will not have parents. It is the node at which all operations on the tree begin. All other nodes can be reached from it by following edges or links. (In the formal definition, each such path is also unique). In diagrams, it is typically drawn at the top. In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the sub tree rooted at that node.

The third section includes the programs of searching. In computer science, a **search algorithm**, broadly speaking, is an algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions. Most of the algorithms studied by computer scientists that solve problems are kinds of search algorithms. The set of all possible solutions to a problem is called the search space.

The fourth section has programs on sorting .A **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non decreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

ARRAYS, STACK, QUEUE AND LINKED LIST

ARRAYS

In computer programming, a group of homogeneous elements of a specific data type is known as an **array**, one of the simplest data structures. Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is given by an index, which is also called a subscript. The index uses a consecutive range of integers. Some arrays are *multi-dimensional*, meaning they are indexed by a fixed number of integers, for example by a tuple of four integers. Generally, one- and two-dimensional arrays are the most common.

Advantages and disadvantages

Arrays permit efficient (constant-time, O(1)) random access but not efficient for insertion and deletion of elements (which are O(*n*), where *n* is the size of the array). Linked lists have the opposite trade-off. Consequently, arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion, and linked lists are best for a list of data which will be accessed sequentially and updated often with insertions or deletions.

Another advantage of arrays that has become very important on modern architectures is that iterating through an array has good locality of reference, and so is much faster than iterating through (say) a linked list of the same size, which tends to jump around in memory. However, an array can also be accessed in a random way, as is done with large hash tables, and in this case this is not a benefit.

Arrays also are among the most compact data structures; storing 100 integers in an array takes only 100 times the space required to store an integer, plus perhaps a few bytes of overhead for the pointer to the array (4 on a 32-bit system). Any pointer-based data structure, on the other hand, must keep its pointers somewhere, and these occupy additional space.

LINEAR ARRAY

TRAVERSAL

This algorithm traverses a linear array LA with lower bound LB and upper bound UB.It traverses LA applying an operation PROCESS to each element of LA.

Step1. Repeat for K = LB to UB
Apply PROCESS to LA[K].
{End of loop}.

Step2. Exit.

INSERTION

Here LA is a liner array with N elements and K is a positive integer such that k<=N. This algorithm inserts an element ITEM into the Kth position in LA.

Step1. [Initialize counter] Set J := N

Step2.Repeat Steps 3 and 4 while J>=K.

Step3. Set LA[J+1] := LA[J]. [Move Jth element downward.]

Step4. Set J := J-1. [Decrease counter.]
[End of step 2 loop.]

Step5. Set LA[K] := ITEM. [Insert element.]

Step6. Set N := N+1. [Reset N.]

Step7. EXIT.

DELETION FROM A LINEAR ARRAY

Here LA is a linear array with N elements and k is a positive integer such that k<=N. This algorithm deletes the Kth element from LA.

Step1. Set ITEM := LA[K].

Step2. Repeat for J = K to N-1:
Set LA[J] := LA[J+1]. [Move J+1st element upward.]
[End of Loop]

Step3. Set N := N-1; [Reset the number N of elements in LA.]

Step4. Exit.

STACKS AND QUEUES

STACK

A **stack** is a data structure based on the principle of Last In First Out *(LIFO)*. Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls.

A *stack-based* computer system is one that stores temporary information primarily in stacks, rather than hardware CPU registers (a *register-based* computer system).

QUEUE

A collection of items in which only the earliest added item may be accessed . Basic operations are add (to the tail) or enqueue and delete (from the head) or dequeue. Delete returns the item removed. Queue is also known as "first-in, first-out" or FIFO data structures.Report Format (Instruction for the students for preparation of lab record)

ALGORITHM FOR IMPLEMENTATION OF STACK

PUSH OPERATION

Step1: [Check for stack overflow]
If TOS>=Size-1
Output" Stack Overflow" and exit

Step2: [Increment the pointer value by one]
TOS = TOS+1

Step3: [Perform Insertion]
S[TOS] = Value

Step4: Exit

POP OPERATION

Step1: [Check whether the stack is empty]
If TOS= 0
Output" Stack Underflow" and exit

Step2: [Remove the TOS information]
Value = S[TOS]
TOS = TOS-1

Step3: [Return the former information of the stack]
Return (Value)

ALGORITHM FOR IMPLEMENTATION OF QUEUE

INSERTION IN A QUEUE

Step1: [Check overflow condition]
If Rear>=Size-1
Output" Overflow" and return
Step2: [Increment Rear pointer]
Rear = Rear+1
Step3: [Insert an element]
Q [Rear] = Value

Step4: [Set the Front pointer]
If Front = -1
Front = 0
Step5: Return

DELETION FROM A QUEUE

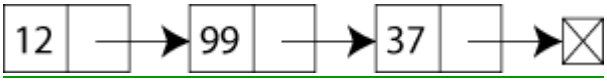
Step1: [Check underflow condition]
If Front = -1
Output "Underflow" and return
Step2: [Remove an element]
Value = Q[Front]
Step3: [Check for Empty queue]
If Front = Rear
Front = -1
Rear = -1
Else
Front = Front + 1
Step4: Return (Value)

LINKED LIST

In computer science, a linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential datatype because it contains a pointer or link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access. Several different types of linked list exist: singly-linked lists, doubly-linked lists, and circularly-linked lists.

Linearly-linked List (Singly-linked list)

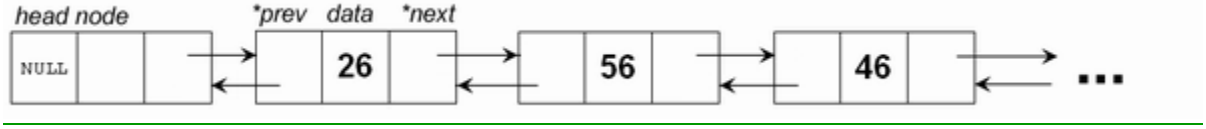
The simplest kind of linked list is a singly-linked list ,which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.



A singly linked list containing three integer values

Doubly-linked list(Two way linked list)

A more sophisticated kind of linked list is a doubly-linked list or two-way linked list. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.



An example of a doubly linked list.

ALGORITHM TO IMPLEMENT SINGLY LINKED LIST

INSERTION OF A NODE

1. AT THE BEGINNING OF A LIST

Step 1: [Check for free space]
If New1 = NULL output "OVERFLOW" and Exit
Step 2: [Allocate free space]
New 1=new link
Step 3: [Read value of information part of a new node]
Info [New1]=Value
Step 4: [Link address part of the currently created node with the address of start]
Next [New1]=Start
Step 5: [Now assign address of newly created node to the start]
Start =New1
Step 6: Exit

2. AT THE END OF A LIST

Step 1: [Check for free space]
If New 1= NULL output "OVERFLOW" and exit
Step 2: [Allocate free space]
New1 = new Link
Step 3: [Read value of information part of a new node]
Info [New1]= Value
Step 4: [Move the pointer to the end of the list]
Repeat while Node <> NULL
Node =Next [Node]
Previous= Next [Previous]
Step 5: [Link currently created node with the last node of the list]
Next [New1]=Node
Next [Previous]= New1
Step 6:Exit

INSERT A NODE IN A LINKED LIST

3.AT A DESIRED PLACE, WHEN NODE NUMBER IS KNOWN

Step 1: [Check for availability space]
If New1 = NULL output "OVERFLOW" and exit

Step 2: [Initialization]
Node _number= 0
Node= Start. Next [Points to first node of the list]
Previous = Address of Start [Assign address of start to previous]

Step 3: [Read node number that we want to insert]
Input Insert_node

Step 4: [Perform insertion operation]
Repeat through step 5 while Node <> NULL

Step 5: [Check if insert_ node is equal to the node_ number]
If Node _ number + 1= Insert _node
Next [New1] = Node
Previous->Next = New1
Info [New1]= Value
Return

Else [move the pointer forward]

Node = Next [Node]
Previous= Next [Previous]

Node_number = Node_number + 1

Step 6: Exit

INSERT A NODE IN A LINKED LIST

4.AT A DESIRED PLACE WHEN INFORMATION IS KNOWN

Step 1: [Check for availability space]
If New1 = NULL output "OVERFLOW" and exit

Step 2: [Initialization]
Node_number = 0
Node = Start. Next [points to first node of the list]
Previous = Address of Start [assign address of start to previous]

Step 3: [Read information of new node number]
Insert_node = value

Step 4: [Perform insertion operation]
Repeat through step 6 while Node <> NULL

Step 5: [Check place where new should be insert]
If Info [Node] < Insert_node
Next [new1] = node
Previous->next = new1
Info [new1] = insert_node
Return

Else [move the pointer forward]
Node = next [node]
Previous = next [previous]

Step 6: Node_number = Node_number + 1

Step 7: Exit

ALGORITHM FOR DELETION OF A NODE

1. FROM THE BEGINNING OF A LINKED LIST

Step 1: [Initialzation]
Node = start. Next [points to the first node in the list]
Previous = assign address of start

Step 2: [Perform deletion operation]
If node = NULL
Output "underflow" and exit
Else [delete first node]
Next [previous] = next [node] [move pointer to next node in the list]
Free the space associated with Node

Step 3: Exit

ALGORITHM FOR DELETION OF A NODE

2. **FROM THE END OF THE LIST**

Step 1: [Initialization]
Node = start. Next [points to the first node in the list]
Previous = assign address of start
Node_number = 0

Step 2: [Check list is empty or not]
If node = NULL
Output "UNDERFLOW" and exit

Step 3: [Scan the list to count the number of nodes in the list]
Repeat while node<> NULL
Node = next [node]
Previous = next [previous]
Node_number = node_number + 1

Step 4: [Initialzation once again]
Node = start. Next
Previous = assign address of start

Step 5: [Scan list for 1 less to size of the list]
Repeat while node_number <> 1
Node = next [node]
Previous = next [node]
Node_number = node_number – 1

Step 6: [Check if last node is arising]
If node_number = 1
Next [previous] = next [node]
Free (node)

Step 7: Exit

ALGORITHM FOR DELETION OF A NODE

3.DELETION ON THE BASIS OF NODE NUMBER

Step 1: [Initialization]
Node = start. Next [points to the first node in the linked list]
Previous = Address of start

Step 2: [Initialize node counter]
Node_number = 1

Step 3: [Read node number]
Delete_node = Value

Step 4: [Check list is empty or not]
If node = NULL
Output "underflow" and exit

Step 5: [Perform deletion operation]
Repeat through step 6 while node<> NULL
If node_number = delete_node
1.Next [previous] = next[node] [make link of previous node to the next node]
2.Delete (Node) [delete current node]
3.Exit

Step 6: Node_number = node_number + 1

Step 7: Exit

ALGORITHM TO IMPLEMENT DOUBLY LINKED LIST

INSERTING A NODE

1.TO THE LEFT OF THE LEFT MOST NODE IN A LIST

Step 1: [Initialization]
Node = start. Next [points to the first node in the list]

Step 2: [Check whether list is empty or not]
If node = NULL
Output "list is empty insert as a first in the list"

Step 3: [Create new node]
New1 = new Double [Double is structure tag name]

Step 4: [Input value for newly created node]
Info [new1] = Value

Step 5: [Make link of newly created node with first node in the list]

Next [new1]
Previous [new1] = previous [node]
Next [previous [node]] = New1
Previous [node] = new1

Step 6: Exit

INSERTING A NODE IN DOUBLY LINKED LIST

2. INSERT A NODE AT THE END OF A LIST

Step 1: [Initialization]
Node = Start

Step 2: [Create a new node]
New1 = new Double

Step 3: [Read information associated with newly created node]
Info [new1] = value

Step 4: [Exhaust the list]
Repeat while node <> NULL

Step 5: [Perform the insertion]

1. Next [new1] = node
2. Previous [new1] = previous [node]
3. Next [previous [node]] = new1
4. Next [node] = new1

STEP 6: Exit

INSERTING A NODE IN DOUBLY LINKED LIST

3. INSERT A NODE AT DESIRED PLACE

Step 1: [Initialization]
New1 = Start
Step 2: [Perform insertion]
Repeat while new1 <>NULL

1. Temp = new1
2. New1 = next [new1]
3. Found = 0
4. Node = start
   Repeat through while node <>NULL and found <>0
5. If info [node] = info [new1]

a) Next [Temp] = node
b) Previous [Temp] =previous [node]
c) Next [previous [node] = temp
d) Previous [node] = temp
e) Found = 1
Else
Node = next [node]

6. if found <> 0
7. if info [node] > info [temp]

a) Next [temp] = node
b) Previous [temp] = previous [node]
c) Next [Previous [node]] = temp

Else

a) Next [temp] = NULL
b) Previous [temp] = node
c) Next [node] = temp

Step 3: Exit

DELETION OF A NODE FROM A DOUBLY LINKED LIST

1.LEFT MOST NODE

Step 1: [Initialization]
Node = start [points to the first node in the list]

Step 2: [Check the list]
If node <> NULL
Output "UNDERFLOW" and exit

Step 3: [Perform deletion]
Next [previous [node]] = next [node]
Previous [next [node]] = previous [node]
Delete (node)

Step 4: Exit

DELETION OF A NODE FROM A DOUBLY LINKED LIST

2. DELETION OF THE RIGHT MOST NODE

Step 1: [Initialization]
Node = Start

Step 2: [Initialize counter]
Counter = 0

Step 3: [Check the list]
If node = NULL
Output "underflow" and exit

Step 4: [Count the number of nodes available in the list]
Repeat while node<>NULL
(1) Node = Next [node]
(2) Counter = counter + 1

Step 5: [Perform deletion]
Node = start
Repeat through (2) while counter <> 1
(1) Node = next [node]
(2) Counter = counter – 1
(3) Next [previous [node]] = next [node]
(4) Previous [next [node]] = previous [node]
(5) Delete (node)

Step 6: Exit

DELETION OF A NODE FROM A DOUBLY LINKED LIST

3.DELETION OF A DESIRED NODE

Step 1: [Initialization]
Node = Start

Step 2: [Check the list]
If node = NULL
Output “underflow” and exit

Step 3: [Set the counter]
Search = 0

Step 4: [Input the node number to which you want to delete]
Node_delete = Value

Step 5: [Perform deletion]
Repeat while node <>NULL
If search = delete_node
Next [previous [node]] = next [node]
Previous [next [node]] = previous [node]
Delete (node)
Else
Node = next [node]
Search = search + 1

Step 6: Exit

CIRCULARLY-LINKED LIST

In a circularly-linked list, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, we can begin at any node and follow the list in either direction until we return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end.The pointer pointing to the whole list is usually called the end pointer.

Singly-circularly-linked list

In a singly-circularly-linked list, each node has one link, similar to an ordinary singly-linked list, except that the next link of the last node points back to the first node. As in a singly-linked list, new nodes can only be efficiently inserted after a node we already have a reference to. For this reason, it's usual to retain a reference to only the last element in a singly-circularly-linked list, as this allows quick insertion at the beginning, and also allows access to the first node through the last node's next pointer.

Doubly-circularly-linked list

In a doubly-circularly-linked list, each node has two links, similar to a doubly-linked list, except that the previous link of the first node points to the last node and the next link of the last node points to the first node. As in a doubly-linked lists, insertions and removals can be done at any point with access to any nearby node.

Sentinel nodes

Linked lists sometimes have a special dummy or sentinel node at the beginning and/or at the end of the list, which is not used to store data. Its purpose is to simplify or speed up some operations, by ensuring that every data node always has a previous and/or next node, and that every list (even one that contains no data elements) always has a "first" and "last" node.

CIRCULAR LINKED LIST

INSERTING A NODE

1.INSERTION AT THE BEGINNING

1. [Check for overflow?]
   If PTR = NULL, then
     Print, overflow
     Exit
   else
        PTR = (node*)malloc(sizeof(node));
   End if

2. if Start = = NULL then
        Set ptr □num = Item
        Set ptr □ next = ptr
        Set start = PTR
        Set last = PTR
     Endif
3. Set PTR □ Num = Item
4. Set ptr □ next = start
5. Set start = ptr
6. Set Last □ next = ptr

2.INSERTION AT THE END

1. [check for overflow ?]
   If ptr = NULL
    Print “overflow”
   Exit
    Else
     PTR = (node*)malloc(sizeof(node));
     Endif

2. if START = NULL, then
       Set ptr □ num = Item
       Set ptr □next = ptr
       Set last = ptr
       Set start = ptr
    Endif
3. Set ptr □ num = Item
4. Set last = ptr
5. Set last = ptr;
6. Set last □ next = start

CIRCULAR LINKED LIST

DELETING A NODE

1.FROM THE BEGINNING

2. [Check for underflow?]
   If start = NULL then
      Print ‘circular list empty’
   Exit
   End if
3. Set ptr = start
4. Set start = start □ next
5. print, element deleted is =, ptr □ Info
6. Set last □ next = start
7. free (ptr)

DELETING A NODE

2.FROM THE END

1. [Check for underflow?]
   If start = NULL then
   Print ‘circular list empty’
   Exit
   End if
2. Set ptr = start
3. repeat step 4 and 5 until
   ptr!= NULL
4. Set ptr1 = ptr
5. Set ptr = ptr next;
6. print “element deleted is”
   ptr num
7. Set q next = p next
8. Set last = q
9. return start

LINKED LIST VS ARRAYS

Linked lists have several advantages over arrays. Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may become wastefully empty or need to be made smaller.
Further memory savings can be achieved, in certain cases, by sharing the same "tail" of elements among two or more lists — that is, the lists end in the same sequence of elements. In this way, one can add new elements to the front of the list while keeping a reference to both the new and the old versions — a simple example of a persistent data structure.

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or Boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element.

DOUBLY-LINKED VS. SINGLY-LINKED

Double-linked lists require more space per node, and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. (Compared with singly-linked lists, which require the previous node's address in order to correctly insert or delete.) Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

CIRCULARLY-LINKED VS. LINEARLY-LINKED

Circular linked lists are most useful for describing naturally circular structures, and have the advantage of regular structure and being able to traverse the list starting at any point. They also allow quick access to the first and last records through a single pointer (the address of the last element). Their main disadvantage is the complexity of iteration.

QUEUE USING LINK LIST

INSERTION IN A QUEUE

Step1. Create a node (P) by using new pointer

Step2. If Front := NULL
Front = P
Rear = P
Else
Rear[Next] = P
Rear = P

Step3. Exit

DELETION FROM A QUEUE

Step1. If Front := NULL,
then Print Underflow and exit.

Step2. Set Item := Queue[Front]

Step3. Front = Front[Next]

Step4. Return and Exit

STACK USING LINK LIST

PUSH (Insertion)

Step1. Create a node(P) by using new pointer
P[Info] := Item

Step2. If TOP := NULL
TOP = P
Else
P[Next] := TOP
TOP = P

Step3. EXIT

POP (Deletion)

Step1. If TOP := NULL, then
Underflow and exit

Step2. [Info] Item = Stack[TOP]

Step3. TOP = TOP[Next]

Step4. EXIT

BINARY SEARCH TREE

A **binary search tree** (**BST**) is a binary tree which has the following properties:

- Each node has a value.
- A total order is defined on these values.
- The left sub tree of a node contains only values less than the node's value.
- The right sub tree of a node contains only values greater than or equal to the node's value.

The major advantage of binary search trees is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

If a BST allows duplicate values, then it represents a multiset. This kind of tree uses non-strict inequalities. Everything in the left subtree of a node is strictly less than the value of the node, but everything in the right subtree is either greater than *or* equal to the value of the node.

If a BST doesn't allow duplicate values, then the tree represents a set with unique values, like the mathematical set. Trees without duplicate values use strict inequalities, meaning that the left subtree of a node only contains nodes with values that are less than the value of the node, and the right subtree only contains values that are greater.

The choice of storing equal values in the right subtree only is arbitrary; the left would work just as well. One can also permit non-strict equality in both sides. This allows a tree containing many duplicate values to be balanced better, but it makes searching more complex.

TREE TRAVERSAL

Tree traversal is the process of visiting each node in a tree data structure. Tree traversal, also called **walking the tree**, provides for sequential processing of each node in what is, by nature, a non-sequential data structure. Such traversals are classified by the order in which the nodes are visited. There are three different ways of traversal such as pre order , in rder and post order traversal.

STEPS TO IMPLEMENT PRE–ORDER TRAVERSAL

1. [do through step 3]
   If node <> NULL
   Output info [node]
2. Call pre-order(left_child[node])
3. Call pre-order(right_child[node])
4. Exit

STEPS TO IMPLEMENT IN-ORDER TRAVERSAL

In–order(node)

1. [Do through step4]
   If node <> NULL
2. Call In order (Left_child[node])
3. Output info[node]
4. Call In order (Right_child[node])
5. Exit

STEPS TO IMPLEMENT POST – ORDER TRANSVERSAL

Post – order (node)

1. [do through step 4]
   If node<> NULL
2. Call post–order (Left_child[node])
3. Call post–order) Right_child[node])
4. Output info[node]
5. Exit

SEARCHING TECHNIQUES

LINEAR SEARCH

Linear search is a search algorithm also known as **sequential search** , that is suitable for searching a set of data for a particular value.
It operates by checking every element of a list one at a time in sequence until a match is found. Linear search runs in O(N). If the data are distributed randomly, on average N/2 comparisons will be needed. The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed. The worst case is that the value is not in the list, in which case N comparisons are needed.
The following pseudocode describes the linear search technique.

For each item in the list.
  Check to see if the item being looked for matches with the item in the list.
    If it matches.
      Return where it was found (the index).
    If it does not match.
      Continue searching until the end of the list is reached.

Linear search can be used to search an unordered list. The more efficient binary search can only be used to search an ordered list.

BINARY SEARCH

The most common application of binary search is to find a specific value in a sorted list. The search begins by examining the value in the center of the list; because the values are sorted, it then knows whether the value occurs before or after the center value, and searches through the correct half in the same way. Here is simple pseudocode which determines the index of a given value in a sorted list a between indices left and right.

```
FUNCTION BINARYSEARCH(A, VALUE, LEFT, RIGHT)
  if right < left
     return not found
  mid := floor((left+right)/2)
  if value > a[mid]
     return binarySearch(a, value, mid+1, right)
  else if value < a[mid]
     return binarySearch(a, value, left, mid-1)
  else
     return mid
```

Because the calls are tail-recursive, this can be rewritten as a loop, making the algorithm as follows:

```
FUNCTION BINARYSEARCH(A, VALUE, LEFT, RIGHT)
  while left ≤ right
     mid := floor((left+right)/2)
     if value > a[mid]
        left  := mid+1
     else if value < a[mid]
        right := mid-1
     else
        return mid
  return not found
```

SORTING TECHNIQUES

INSERTION SORT

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. Shell sort is a variant of insertion sort that is more efficient for larger lists.

FUNCTION FOR INSERTION SORT

```
insert(array a, int length, value)
 {
    int i := length - 1;
    while (i ≥ 0 and a[i] > value)
{
        a[i + 1] := a[i];
        i := i - 1;
    }
    a[i + 1] := value;
 }
 insertionSort(array a, int length)
 {
    int i := 1;
    while (i < length)
 {
        insert(a, i, a[i]);
        i := i + 1;
    }}
```

EXCHANGE SORT

Bubble sort, sometimes shortened to **bubblesort**, also known as **exchange sort**, is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. Because it only uses comparisons to operate on elements, it is a comparison sort. Although bubble sort is one of the simplest sorting algorithms to understand and implement, its $\Theta(n^2)$ complexity means it is far too inefficient for use on lists having more than a few elements. Even among simple $\Theta(n^2)$ sorting algorithms, algorithms like insertion sort are usually considerably more efficient.

ALGORITHM FOR BUBBLE SORT

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

Step 1. Repeat steps2 and 3 for K= 1 to N-1.

Step 2. Set PTR := 1. [Initializes pass pointer PTR.]

Step 3. Repeat while PTR <= N-K [Executes pass]

a) If DATA[PTR] > DATA[PTR+1], then:
Interchange DATA[PTR] and DATA[PTR+1].
[End of If structure]

b)Set PTR := PTR+1;
[End of Inner loop]
[End of step1 outer loop]

Step 4.EXIT

SELECTION SORT

Selection sort algorithm iterates through a list of n unsorted items, has a worst-case, average-case, and best-case run-time of Θ(n2), assuming that comparisons can be done in constant time. Among simple worst case Θ(n2) algorithms, it is generally outperformed by insertion sort, but still tends to outperform contenders such as bubble sort.
Selection sort can be implemented as a stable sort. If, rather than swapping in step 2, the minimum value is inserted into the first position (that is, all intervening items moved down), this algorithm is stable (but slower). Selection sort is an in-place algorithm.

STEPS TO BE FOLLOWED ARE

1.find the minimum value in the list

2.swap it with the value in the first position

3.sort the remainder of the list (excluding the first value)

```
Function for selectionSort(int *array,int length)//selection sort function
{
        int i,j,min,minat;
        for(i=0;i<(length-1);i++)
        {
                minat=i;
                min=array[i];

    for(j=i+1;j<(length);j++) //select the min of the rest of array
         {
                 if(min>array[j])   //ascending order for descending reverse
                 {
                         minat=j;  //the position of the min element
                         min=array[j];
                 }
         }
         int temp=array[i] ;
         array[i]=array[minat];  //swap
         array[minat]=temp;
        }
}
```

QUICK SORT

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array,an element, called a pivot is choosen,all smaller elements are moved before the pivot, and all greater elements are moved after it. This can be done efficiently in linear time and in-place. Then recursively sorting can be done for the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, this makes quicksort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower (O(n2)) performance, but if at each step we choose the median as the pivot then it works in O(n log n).
Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

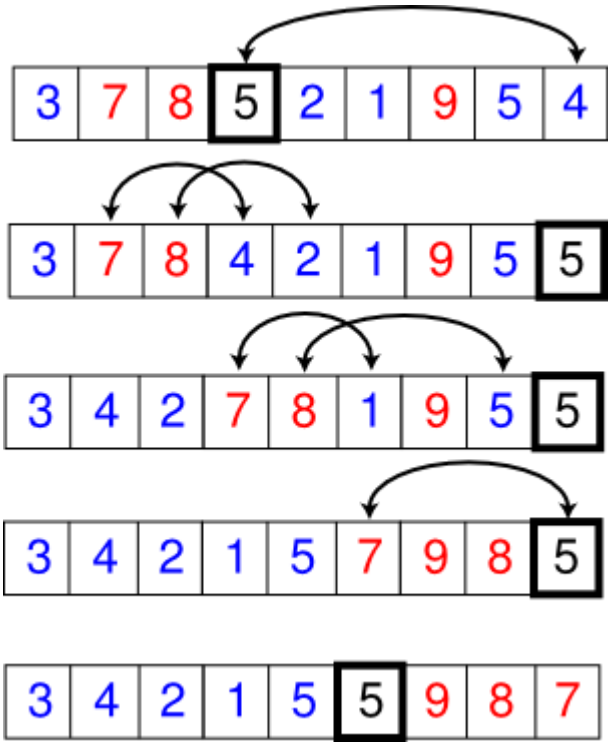
Pick an element, called a pivot, from the list.
Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Pseudocode For partition(a, left, right, pivotIndex)

```
    pivotValue := a[pivotIndex]
    swap(a[pivotIndex], a[right]) // Move pivot to end
    storeIndex := left
    for i from left to right-1
        if a[i] ≤ pivotValue
            swap(a[storeIndex], a[i])
            storeIndex := storeIndex + 1
    swap(a[right], a[storeIndex]) // Move pivot to its final place
    return storeIndex
```

Pseudocode For quicksort(a, left, right)

```
    if right > left
        select a pivot value a[pivotIndex]
        pivotNewIndex := partition(a, left, right, pivotIndex)
        quicksort(a, left, pivotNewIndex-1)
        quicksort(a, pivotNewIndex+1, right)
```



SHELL SORT

Shell sort was invented by Donald shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort. Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements (sets with less than 1000 or so elements). Another advantage of this algorithm is that it requires relatively small amounts of memory.

```
void shellsort (int[] a, int n)
{
   int i, j, k, h, v;
   int[] cols = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,
                 1968, 861, 336, 112, 48, 21, 7, 3, 1}
   for (k=0; k<16; k++)
   {
      h=cols[k];
      for (i=h; i<n; i++)
      {
         v=a[i];
         j=i;
         while (j>=h && a[j-h]>v)
         {
            a[j]=a[j-h];
            j=j-h;
         }
         a[j]=v;
      }
   }
}
```

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

MERGE SORT

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e. 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists.
Merge sort works as follows:

1. Divide the unsorted list into two sublists of about half the size
2. Sort each of the two sublists
3. Merge the two sorted sublists back into one sorted list.

pseudocode for mergesort

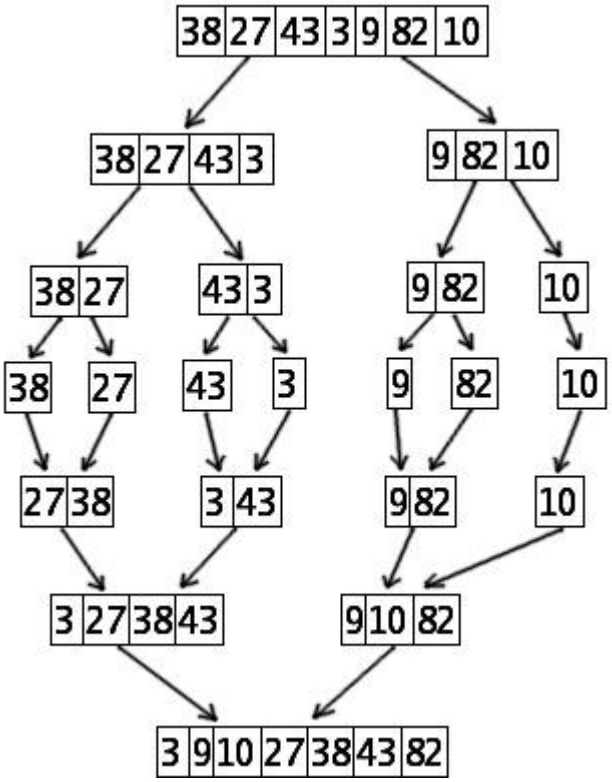
```
mergesort(m)
   var list left, right
   if length(m) ≤ 1
      return m
   else
      middle = length(m) / 2
      for each x in m up to middle
         add x to left
      for each x in m after middle
         add x to right
      left = mergesort(left)
      right = mergesort(right)
      result = merge(left, right)
      return result
```

There are several variants for the merge() function, the simplest variant could look like this:

Pseudocode for merge

```
merge(left,right)
   var list result
   while length(left) > 0 and length(right) > 0
      if first(left) ≤ first(right)
         append first(left) to result
         left = rest(left)
      else
         append first(right) to result
         right = rest(right)
   if length(left) > 0
      append left to result
   if length(right) > 0
      append right to result
   return result
```

MERGE SORT TREE



SPARSE MATRICES

In the mathematical subfield of numerical analysis a **sparse matrix** is a matrix populated primarily with zeros.

Sparsity is a concept, useful in combinatorics and application areas such as network theory, of a low density of significant data or connections. This concept is amenable to quantitative reasoning. It is also noticeable in everyday life.

Huge sparse matrices often appear in science or engineering when solving problems for linear models.

When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to used specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard matrix structures and algorithms are slow and consume large amounts of memory when applied to large sparse matrices. Sparse data is by nature easily compressed, and this compression almost always results in significantly less memory usage. Indeed, some very large sparse matrices are impossible to manipulate with the standard algorithms.

Viva Questions

Course code:ETCS-255

Course Title:Data Structures

1. What is a data structure?
2. What is linear data structure?
3. What are the ways of representing linear structure?
4. What is a non-linear data structure?
5. What are various operations performed on linear structure? What is a square matrix?
6. What is a sparse matrix?
7. What is a triangular matrix?
8. What is a tridiagonal matrix?
9. What is row major ordering?
10. What is column major ordering?
11. What is a linked list?
12. What is a null pointer?
13. What is a free pool or free storage list or list of available space?
14. What is garbage collection?
15. What is overflow?
16. What is underflow? What is a header node?
17. What is a header linked list?
18. What is a header node?
19. What is a grounded linked list?
20. What is circular header list?
21. What is a two-way list?
22. What is a stack?
23. What is a queue?
24. What is infix notation?
25. What is polish notation?
26. What is reverse polish notation?
27. What is recursive function?
28. W hat is a priority queue?
29. Define a deque.
30. Define Tree,Binary tree,Binary search tree.
31. What are various ways of tree traversal?
32. What is an AVL tree?
33. What are similar trees, and when the trees are called copies of each other?
34. What is searching?

35. What is linear search?
36. What is binary search?
37. Why binary search cannot be applied on a linked list?
38. What is a connected graph?
39. What is depth-first traversal?
40. What is breadth-first traversal?
41. Why is the algorithm for finding shortest distances called greedy?
42. What are advantages of selection sort over other algorithms?
43. What are disadvantages of insertion sort?
44. Define the term divide and conquer.
45. What is a pivot?

List of Advanced Practicals

Laboratory Name: **Data Structures** **Subject Code :** **ETCS 255**

GRAPH:

1. WAP to Implement Depth-First-Search in a graph.
2. WAP to Implement Breadth-First-Search in a graph.

TRAVERSAL IN A GRAPH

Depth-first search (**DFS**) is an algorithm for traversing or searching a graph. Intuitively, one starts at the some node as the root and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the graph that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a LIFO stack for expansion.

STEPS FOR IMPLEMENTING DEPTH FIRST SEARCH

1. Define an array B or Vert that store Boolean values, its size should be greater or equal to the number of vertices in the graph G.
2. Initialize the array B to false
3. For all vertices v in G
   if B[v] = false
   process (v)
4. Exit

Breadth first search (BFS) is an uninformed search method that aims to expand and examine all nodes of a graph systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

STEPS FOR IMPLEMENTING BREADTH FIRST SEARCH

1. Initialize all the vertices by setting Flag = 1
2. Put the starting vertex A in Q and change its status to the waiting state by setting Flag = 0
3. Repeat through step 5 while Q is not NULL
4. Remove the front vertex v of Q . process v and set the status of v to the processed status by setting Flag = -1
5. Add to the rear of Q all the neighbour of v that are in the ready state by setting Flag = 1 and change their status to the waiting state by setting flag = 0
6. Exit

Space complexity of DFS is much lower than BFS (breadth-first search). It also lends itself much better to heuritic methods of choosing a likely-looking branch. Time complexity of both algorithms are proportional to the number of vertices plus the number of edges in the graphs they traverse.

ANNEXURE I

COVER PAGE OF THE LAB RECORD TO BE PREPARED BY THE STUDENTS

DATA STRUCHERS LAB

ETCS-255

(size 20'' , italics bold , Times New Roman)

Faculty Name:

(12'' , Times New Roman)

Student Name:

Roll No.:

Semester:

Batch :

(12'', Times New Roman)



Maharaja Agrasen Institute of technology, PSP area,
Sector – 22, Rohini, New Delhi – 110085

(18'' bold Times New Roman)

ANNEXURE II

FORMAT OF THE INDEX TO BE PREPARED BY THE STUDENTS

Student's Name

Roll No.

INDEX

S.No.	Name of the Program	Date	Signature & Date	Remarks