DATA STRUCTURE

Data Structure:

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other. Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associatively. Data structure affects the design of both the structural and functional aspects of a program.

Algorithm + Data structure = program

The representation of a particular data structure in the memory in the memory of a computer is called a storage structure. That is, a data structure should be represented in such a way and auxiliary memory of the computer. A storage structure representation in auxiliary memory is often called a file structure.

It is clear from the above discussion that the data structure and the operations on organized data items can integrally solve the problem using a computer.

Data structure = organized data + operations

NEEDS OF DATA STRUCTURE

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still seem to double every couple of years. Won't any efficiency problem we might have today be solved by tomorrow's hardware? As we develop more powerful computers, our history so far has always been to use additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater.

ARRAY AND ARRAY TYPE

An array is a collection of homogeneous data elements described by a single name. An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

One Dimensional Array

One – dimensional array (or linear array) is a set if 'n' finite numbers of homogenous data elements such as:

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. 'n' number of elements is called the length or size of an array. The elements of an array 'A' may be denoted in C as A [o], A [1], A[2]……A[n-1].

The number 'n' in A[n] is called a subscript or an index and A[n] is called a subscripted variable.

What are the characteristics of data structure?

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known.	Slow search, slow deletion, and fixed size
Ordered array	Quicker search than unsorted array	Slow insertion and deletion, fixed size
Stack	Provides last in, first – out access	Slow access to other items
Queue	Provides first – in, first – out access	Slow access to other items
Linked list	Quick insertion, quick deletion	Slow search
Binary tree	Quick search, insertion, deletion. (if tree remains balanced)	Deletion algorithm is complex
Red – black tree	Quick search, insertion, deletion. Tree always balanced	Complex
2 – 3 – 4 tree	Quick search, insertion, deletion. Tree always balanced. Similar trees good for disk storage	Complex
Hash table	Very fast access if key known. Fast insertion	Slow deletion, access slow if key not known, inefficient memory usage
Heap	Fast insertion, deletion, access to largest item.	Slow access to other items

Multi Dimensional Array

The elements of an array can be of any data type, including arrays! An array of arrays is called a multidimensional array. In this case, since we have 2 subscripts, this is two – dimensional array. In a two dimensional array, it is convenient to think of the first subscript as being the row, and the 2nd subscript as being the column. Two loops are required. Similarly the array of 'n' dimensions would require 'n' loops.

Sparse Array

Sparse array is an important application of arrays. A sparse array is an array is an array where nearly all of the elements have the same value (usually zero) and this value is a consonant. One dimensional sparse array is called sparse vectors and two dimensional sparse arrays are called sparse matrix.

DEFINE VECTORS AND LISTS

A vector is a one – dimensional ordered collection of numbers. Normally, a number of contiguous memory locations are sequentially allocated to the vector. A vector size is fixes and, therefore, required a fixed number of memory locations. A vector can be a column vector which represents a 'n' by 1 ordered collections, or a row vector which represents a 1 by 'n' ordered collections.

The column vectors appears symbolically as follows:

$$A = \begin{pmatrix} A1 \\ A2 \\ A3 \\ . \\ . \\ . \\ An \end{pmatrix}$$

A row vector appears symbolically as follows:

A = [A1, A2, A3,….. An]

Vectors can contain either real or complex numbers. When they contain real numbers, they are sometime called real vectors. When they contain complex numbers, they are called complex vectors.

Lists

A list is an ordered set consisting of a varying number of elements to which insertion and deletion can be made. A list represented by displaying the relationship between the adjacent elements is said to be a linear list. Any other list is said to be non linear. List can be implemented by using pointers. Each element is referred to as nodes; therefore a list can be defined as a collection of nodes.

STRING AND STRING REPRESENTATION

String

A finite set of sequence (alphabets, digits or special characters) of zero or more characters is called a string. The number of characters in a string is called the length of the string. If the length of the string is zero then it is called the empty string or null string. A string is defined as a sequence of characters. Examples of strings are "This is a string" or "Name?", where the double quotes (" ") are not part of the string. There is an empty string, denoted.

String Representation

String are stored or represented in memory by using following three types structures.

- Fixed length structures
- Variable length structures with fixed maximum
- Linear Structures.

Fixed length Representation

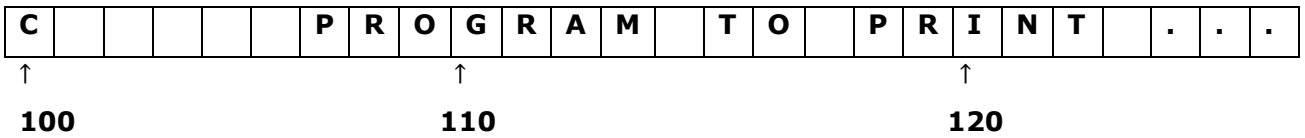
In fixed length storage each line is viewed as a record, where all records have the same length. That is each record accommodates maximum of same number of characters.

The main advantage of representing the string in the above way is:

1. To access data from any given record easily
2. It is easy to update the data in any given record.

The main disadvantages are:

1. Entire record will be read even if most of the storage consists of inessential blank space. Time is wasted in reading these blank spaces.
2. The length of certain records will be more than the fixed length. That is certain records may require more memory space than available.



Fixed Length Representation

Variable Length Representation

In variable length representation, strings are stored in a fixed length store medium. This is done in two ways.

1. One can use a marker, (any special characters) such as two – dollar sign ($$), to signal the end of the string.

2. Listing the length of the string at the first place is another way of representing strings in this method.

Linked List Representations

In linked list representations each characters in a string sequentially arranged in memory cells, called nodes, where each node contain an item and link, which points to the next node in the list (i.e., link contain the address of the next node). SUB STRING Group of consecutive elements or characters in a string (or sentence) is called sub string. This group of consecutive elements may not have any special meaning. To access a sub string from the given string we need following information:

(a) Name of the string

(b) Position of the first character of the sub string in the given string

(c) The length of the sub string

Finding whether the sub string is available in a string by matching its characters is called pattern matching.

RECURSION AND ITERATION

Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. Recursion is a programming technique in which a function (or member function) calls itself. This might sound like a strange thing to do, or even a catastrophic mistake. Recursion is, however, one of the most interesting, and surprisingly effective, techniques in programming like pulling yourself up by your bootstraps, recursion seems incredible when you first encounter it. However, it not only works, it also provides a unique conceptual framework for solving many problems.

S. No	Iteration	Recursion
1	It is a process of executing a statement or a set of statements repeatedly. Untill some specified condition is specified.	Recursion is the technique of defining anything in terms of itself
2	Iterations involves four clear – cut steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively	Not all problems have recursive solution
4	Iterative counterpart of a problem is more efficient in terms of memory 1, 1 utilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems or not recursive in nature.

EXPRESSION

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators)

1. Infix notation 2. Prefix notation 3. Postfix notation

The infix notation is what we come across in our general mathematics, where the operator is written in between the operands. For example: The expression to add two numbers A and B is written in infix notation as:

A + B

Note that the operator '+' is written in between the operands A and B.

The prefix notation is a notation, in which the operator is written before the operands. As the operator '+' is written before the operands A and B, this notation is called prefix

+ A B

The postfix notation the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as suffix notation or reverse polish notation. The above expression if written in postfix expression looks like.

A B +

CONVERTING INFIX TO POSTFIX EXPRESSION

The method of converting infix expression A + B * C to postfix form is:

A +B * C infix form

A + (B * C) Parenthesized expression

A + (B C *) convert the multiplication

A (BC *) + convert the addition

A B C *+ postfix form

POINTER

A pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address which can be used to access its location. A pointer variable points to a memory location. By making use for pointer, We can access and change the contents of the memory location. A pointer variable contains the memory location of another variable. You begin the declaration of a pointer by specifying the type of data stored in the location identified by the pointer. The asterisk tells the complier that you are creating a pointer variable. Finally you give the name of the pointer variable. The pointer declaration syntax is as shown below.

Type	*	variable	name
Example:	int		*ptr
	Float*string		

ADDRESS OPERATOR

Once we declare a pointer variable, we point the variable to another variable. We can do this by assigning the address of the variable to the pointer as in the following Example: ptr = &num

DATE STRUCTURE STACKS, STACK IMPLEMENTATION AND STACK OPERATION

STACK

A stack is one of the most important and useful non – primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added / inserted and from which items may be deleted at only one end, called top of the stack. As all the addition in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called last – in – first – out (LIFO). Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of stack.

STACK OPERATION

The primitive operations performed on the stack are follows:

Push: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add

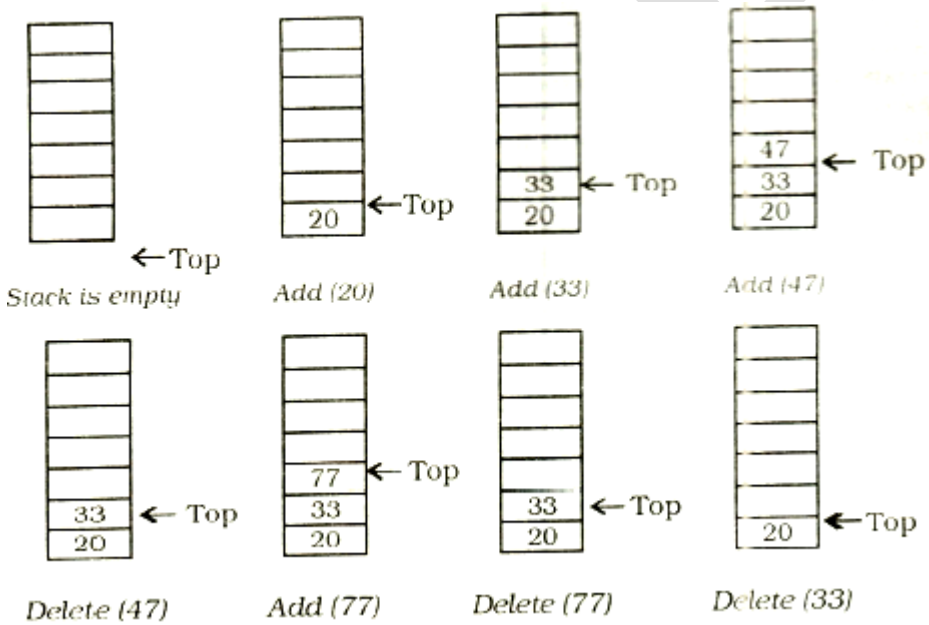


Figure: Stack

the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated. Then the stack overflow condition occurs.

POP : The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

Peek Method

Push and pop are the two primary stack operations. However, it's sometimes useful to be able to read the value from the top of the stack without removingit. The peek operation does this. By pushing the peek button a few times, you'll see the value of the item at Top copied to the Number text field, but the item is not removed from the stack, which remains unchanged. That you can only peek at the top item. By design, all the other items are invisible to the stack user

STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

Stack Implementation Using Array

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack. One choice is to make the top be at position 0 in the array. In terms of list functions, all insert and remove operations would then be on the element in position 0. This implementation is inefficient, because now every push or pop operation will require that all elements currently in the stack be shifted one position in the array, for a cost of _ (n) if there are n elements. The other choice is have the top element be at position nAUU1 when there are n elements in the stack. In other words, as elements are pushed onto the stack, they are appended to the tail of the list. Method pop removes the tail element. In this case, the cost for each push or pop operation is only _ (1).

Stack implementation using pointers or linked-list

The linked stack implementation is a simplified version of the linked list implementation. Elements are inserted and remove only from the head of the list. The header node is not used because no special case code is required for lists of zero or one elements. The only data member is top, a pointer to the first (top) link node of the stack. Method push first modifies the next field of the newly created link node to point to the top of the stack and than sets top to point to the new link node. Method pop is also quite simple. The variable temp stores the values of the top node, while temp keeps a link to the top node a it is removed from the stack. The stack is updated by setting top to point to the next element in the stack. The old top node is then returned to free store (or the free list), and the element value is returned as the value of the pop method.

QUEUE AND QUEUE IMPLEMENTATION

QUEUE

The queue is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an enqueue operation) and removed fro the front (called a desuetude operation). Queues operate like standing in line at a movie theatre ticket counter. 1 if nobody cheats then newcomers goes to the back of the line. The person at the front of the line is the next to be served. Thus queues release their elements in order of arrival. Accountants have used queues since long before the existence of computers. They call a queue a" FIFO" list. Which stands for "First- In, first-out. " this section presents two implementations for queues : the array based queue and the linked queue.

It is a homogeneous collection of elements in which new elements are added one end called rear, and the existing elements are deleted from other end called front.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (POP)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. POP operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front

– rear + 1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

Applications of Queue:

Applications of queues are, if anything, even more common than applications are applications of stacks, since in performing tasks by computer, as in all parts of life, it is often necessary to wait one's turn before having access to something. Within a computer system there may be queues of tasks waiting for the printer, for access to disk storage, or even, with multitasking, for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue.

QUEUE IMPLEMENTATION

Queue can be implemented in two ways

1. Using arrays (Static)
2. Using pointers (dynamic)

Queue Implementation using arrays.

The array – based queue is somewhat tricky to implement effectively. A simple conversion of the array based list implementation is not efficient. Assume that there are n elements in the queue. By analogy to the array – based list implementation. We could require that all elements of the queue be stored in the first n positions of the array. If we choose the rear element of the queue to be in position 0, then de - queue operations require only e (1) time because the front element of the array. However, enqueue operations will require e (n) time, because the n elements currently in the queue must each be shifted one position in the array. If instead we choose the rear element of the queue to be in position n – 1, then an enqueue operation is equivalent to an append operation on a list. This requires only e (1) time. But now, a de - queue operation requires e (n) time, because all of the elements must be shifted down by one position to retain the property that the remaining n – 1 queue elements reside in the first n – 1 positions of the array.

Linked Queues

The linked queue implementation is a straightforward adaption of the linked list. Methods front and rear are pointers to the front and rear queue elements, respectively. We will use a header link node, enqueue operation by avoiding any special cases when the queue is empty. On initialization, the front and rear pointers will point to the header node while rear points to the true last link node in the queue. Method enqueue places the new element in a link node at the end of the linked list (i.e., the node that rear point to) and then advances rear to point to the new link node. Method dequeue grabs the first element of the list removes it.

TYPES OF QUEUE

There are three major variations in a simple queue.

They are:

1. Circular queue
2. Double ended queue (de - queue)
3. Priority queue

Circular Queue: Circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue after adding the element at the last index i.e (n - 1) th as queue is starting with 0 index, the next element will be inserted at the very first location of the queue. That's why linear queue leads to wastage of the memory, and this flaw of linear queue is

overcome by circular queue. So, in all we can say that the circular queue is a queue in which first element come right after the last element that means a circular queue has a starting point but no end point.

Double Ended Queue (de – queue): A deque is a homogeneous list in which elements can be added or inserted (Called push operation) and deleted or removed from both the ends (which is called pop operation). i.e; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

Priority Queues: Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

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

For example consider a manger who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

DISTINGUISH BETWEEN QUEUE AND STACK

Stack: Represents the collection of elements in Last In First Out order. Operations includes testing null stack, finding the top element in the stack, removal of top most element and adding elements on the top of the stack.

Queue: Represents the collection of elements in First In First Out order. Operations include testing null queue, finding the next element, removal of elements and inserting the elements from the queue. Insertion of elements is at the end of the queue. Deletion of elements is from the beginning of the queue

LINKED LISTS

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next field.

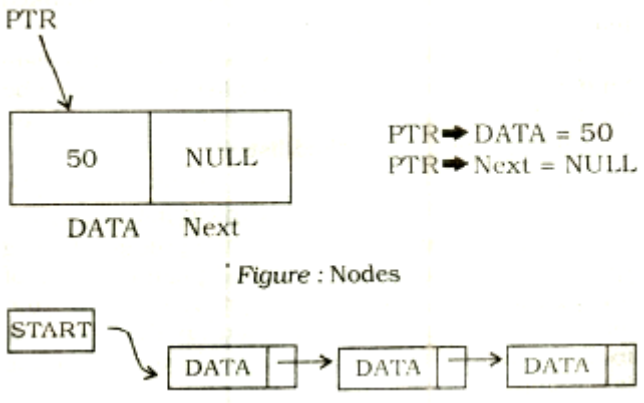


Figure : Nodes

Linked List

ADVANTAGES AND DISADVANTAGES OF LINKED LIST

Advantages

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program

2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.

3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

4. Many complex applications can be easily carried out with linked list.

Disadvantages

Linked list have many disadvantages and some of them are

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.

2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

OPERATION ON LINKED LISTS

The primitive operations performed on the linked list are as follows:

(1**) Creation:** Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

(2) Insertion: Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(a) At the beginning of the linked list

(b) At the end of the linked list

(c) At any specified position in between in a linked list

(3) Deletion: Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

(a) Beginning of a linked list

(b) End of a linked list

(c) Specified location of the linked list

(4) Traversing: Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, and nodes only. But in doubly linked list forward and backward traversing is possible.

(5) Concatenation: Concatenation is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1^{st} node of B in the (n+1) th node in A. After concatenation A will contain (n+m) nodes.

TYPES OF LINKED LIST

Basically we can divide the linked list into the following three types in the order in which they (or node) are arranged

Singly Linked List

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure.

Doubly Linked List

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every node in the doubly linked list has three fields. Left pointer, right pointer and data. L point will point to the node in the side (or previous node) that is L point will hold the address of the previous node. R Point will point to the node in the right side (or next node) that is R Point will hold the address of the next node. DATA will store the information of the node.

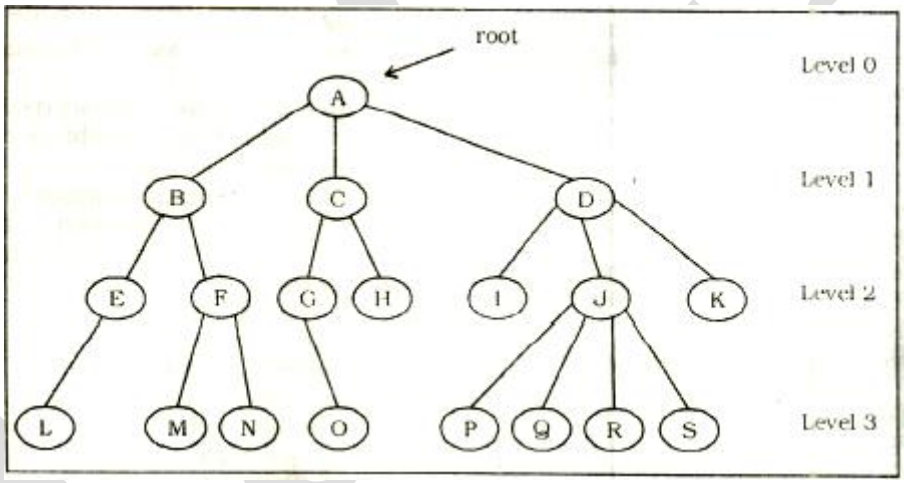
Circular Linked List

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply strong the address of the very first node in the linked field of the last node.

What is tree?

Tree

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:



TREE

1. There is a special node called the root of the tree

2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.

Basic Terminologies

Root is a specially designed node (or data items) in a tree. It is the first node in the hierarchical arrangement of the data items. 'A' is a root node in the Fig. Each data item in a tree is called a node. It specifies the data information and links (branches) to other data items.

Degree of a node is the number of sub trees of a node in a give tree. In Fig.

The degree of node A is 3

The degree of node B is 2

The degree of node C is 2

The degree of node D is 3

Degree of a Tree

The degree of a tree is the maximum degree of node in given tree. In the above tree, degree of a node J is 4. All the other nodes have less or equal degree. So the degree of the above tree is 4. A node with degree zero is called a terminal node or a lea – for example in the above tree Fig M, N, I, O etc. are leaf node. Any node whose degree is not zero is called non – terminal node. They are intermediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level n + 1.

Depth

Depth of a tree is the maximum level of any node in a given tree. That is a number of a level one can descend the tree from its root node to the terminal nodes (leaves). The term height is also used to denote the depth.

Leaf nodes

Nodes at the bottommost level of the tree are called leaf nodes. Since they are the bottommost level, they do not have any children.

Internal nodes

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

Sub trees

A sub tree is a portion of a tree data structure that can be viewed as a complete tree in it. Any node in a tree T, together with all the nodes below it, comprises a sub tree of T. The sub tree corresponding to any other node is called a proper sub tree **(**in analogy to the term proper subset).

BINARY TREE AND ITS REPRESENTATION

A binary tree is made up of a finite set of elements called nodes. This set either is empty of consists of a node called the root together with two binary trees, called the left and right sub trees, which are disjoint from each other and from the root. The roots of these sub trees are children of the root. There is an edge from a node to each of its children, and a node is said to be the parent of its children.

Binary Tree Representation

This section discuses two ways of representing binary tree T in memory.

1. Sequential representation using arrays
2. Linked list representation

Array Representation:

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially.

A[]

[0]	[1]	[2]	[3]	[4]	[5]	[6]
A	B	C	D	E		F

The array representation of the binary tree is shown in above figure. To perform any operation often we have to identify the father, the left child and right child of an arbitrary node.

Linked List Representation:

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as:

(a) Left child (Child)

(b) Information of the node (Info)

(c) Right child (Child)

The L child links to the left child node of the parent node, Info holds the information of every node and R Child holds the address of right child node of the parent node.

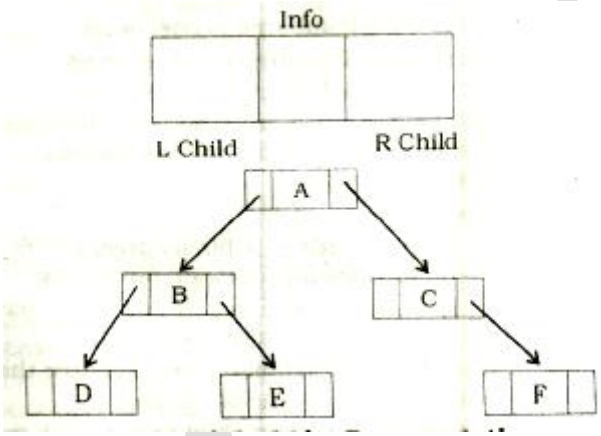


Fig: Linked List Representation

Operations on Binary Tree

The basic operations that are commonly performed a binary tree is listed below:

1. Create an empty Binary Tree
2. Traversing a Binary Tree
3. Inserting a New Node
4. Deleting a Node
5. Searching for a Node

Traversing Binary Tree Recursively

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree. They are:

1. Pre order Traversal (Node – left – right)
2. In order Traversal (Left – node – right)
3. Post Order Traversal (Left – right – node)

Pre Orders Traversal Recursively

To traverse a non – empty binary tree in pre order following steps one to be processed

1. Visit the root node
2. Traverse the left sub tree in preorder
3. Traverse the right sub tree in preorder

That is, in preorder traversal, the root node is visited (or processed) first, before travelling through left and right sub tree recursively.

In order Traversal Recursively

The in order traversal of a non – empty binary tree is defined as follows.

1. Traverse the left sub tree in order

2**.** visit the root node

3. Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root.

After visiting the root the right sub tree is traversed recursively, before visiting the root.

After visiting the root the right sub tree is traversed recursively, in order fashion.

Post order Traversal Recursively

The post order traversal of a non – empty binary tree can be defined as:

1. Traverse the left sub tree in post order
2. Traverse the right sub tree in post order
3. Visit the root node.

In post order traversal, the left and right sub tree(s) are recursively processed before visiting the root.

TYPES OF BINARY TREE

Binary search Trees

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties.

1. Every node has a value and no two nodes have the same value (i.e., all the values are unique).
2. If there exists a left child or left sub tree then its value is less than the value of the root.
3. The value(s) in the right child or right sub tree is larger than the value of the root node.

The Operations performed on binary tree can also ba applied to Binary tree can also be applied to Binary Search Tree BST). But in this section we discuss few other primitive operators performed on BST;

1. Inserting a node
2. Searching a node
3. Deleting a node

Another most commonly performed operation on BST is, traversal. The tree traversal algorithms (pre order, post – order and in order) are the standard way of traversing a binary search tree.

Thread Binary Tree

Traversing a binary tree is a common operation and it would be helpful to find more efficient method, for implementing the traversal. Moreover, half of the entries in the Lchild and Rchild field will contain NULL pointer. These fields may be used more efficiently by replacing the NULL entries by special pointers which points to nodes higher in the tree. Such types of special pointers are called threads and binary tree with such pointers are called threaded binary tree.

Balanced Binary Tree

A balanced binary tree is one in which the largest path through the left sub tree is the same length as the largest path of the right sub tree, i.e., balanced trees compared to unbalanced binary tree. i.e., balanced items are used to maximize the efficiency of the operations on the tree. There are two types of balanced trees.

B + tree: In computer science, a B + tree is a type of tree which represents sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a key. It is

a records, each of which is identified by a key. It is a dynamic, multilevel index, with maximum and minimum bounds on the number of keys in each index segment (usually called a "block" or "node"). In a B+ tree, in contrast to a B – tree, all records are stored at the leaf level of the tree; only keys are stored in interior nodes.

The primary value of a B+ tree is in storing data for efficient retrieval in a block – oriented storage context – in particular, file systems. This is primarily because unlike binary search trees. B + trees have very high fan-out (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

Parse tree: A concrete syntax tree or parse tree or parsing tree [1] is an ordered, rooted tree that represents the syntactic structure of a string according to some formal grammar. Parse trees are usually constructed according to one of two competing relations either in terms of the constituency relation of constituency grammars (=phrase structure grammars) or in terms of the dependency relation of dependency grammars. Parse trees are distinct from abstract syntax trees (also known simply as syntax trees), in that their structure and elements more concretely reflect the syntax of the input language. Parse trees may be generated for sentences in natural languages (see natural language processing), as well as during processing of computer languages, such as programming languages.

AVL Tree: In computer science, an AVL tree is a self – balancing binary search tree, and it was the first such data structure to be invented. [1] In an AVL tree, the heights of the two child sub trees of any node differ by at most one. Lookup, insertion, and deletion all take O (log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations. In other words AVL tree is a binary search tree where the height of the left sub tree differs from the height of the right sub tree by at most 1 level, if it exceeds 1 level then rebalancing occurs.

Spanning tree: A spanning tree T of a connected, undirected graph G is a tree composed of all the vertices and some (or perhaps all) of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed.

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

Digital Binary Tree

A digital search tree is a binary tree in which each node contains one element. The element – to – node assignment is determined by the binary representation of the element keys. Suppose that we number the bits in the binary representation of a key from left to right beginning at one. Then bit one of 1000 is 1, and bits two, three, and four are 0. All keys in the left sub trees of a node at level I have bit I equals to zero whereas those in the right sub trees of nodes at this level have bit I = 1.

GRAPH

A graph consists of a set of vertices and a set of edges. Graphs representations have found application in almost all subjects like geography, engineering and solving games and puzzles. A vertex can also have a weight, sometimes also called a cost.

A graph G consist of

1. Set of vertices V (called nodes), (V = {V1, V2, V3, V4……}) and
2. Set of edges E(i.e., E{e1, e2, e3….cm}

A graph can be represents as G = (V, E), where V is a finite and non empty set at vertices and E is a set of pairs of vertices called edges. Each edge 'e' in E is identified with a unique pair (a, b) of nodes in V, denoted by e = [a, b].

A graph whose pairs are ordered is called a directed graph, or just a digraph.

If a graph is not ordered, it is called an unordered graph, or just a graph.

A path is a sequence of vertices in graph such that all vertices are connected by edges. The length of a path is the number of edges from the first vertex in the path to the last vertex. A path can also consist of a vertex to itself, which is called a loop. Loops have a length of 0.

A cycle is a path of at least 1 in a directed graph so that the beginning vertex is also the ending vertex. In a directed graph, the edges can be the same edge, but in an undirected graph, the edges must be distinct. S

An undirected graph is considered connected if there is a path from every vertex to every other vertex. In a directed graph, this condition is called strongly connected. A directed graph that is not strongly connected, but is considered connected is called weakly connected. If a graph has an edge between every set of vertices, it is said to be a complete graph.

Two graphs are said to be isomorphic if there is a one to one correspondence between their vertices and between their edges such that incidence are prevented.

Weighted graph can be represented using linked list by sorting the corresponding weight along with the terminal vertex of the edge.

REPRESENTATION OF GRAPH

Graph is a mathematical structure and finds its application is many areas, where the problem is to be solved by computers. The problems related to graph G must be represented in computer memory using any suitable data structure to solve the same. The two most commonly used data structures for representing a graph (directed or undirected) are adjacency lists and adjacency matrix.

Adjacency Lists (Linked List Representation):

The adjacency lists representation of a graph G consists of an array Adj of n linked lists, one for each vertex in G, such that Adj[o] for vertex o consists of all vertices adjacent to o. This list is often implemented as a linked list. (sometimes it is also represented as a table, in which case it is called the stat representation)

Adjacency Matrix: The adjacency matrix of a graph G = (V, E) is an n × n matrix

A = [aij] in which aij = 1 if there is an edge from vertex I to vertex j in G; otherwise

Aij = 0. Note that in an adjacency matrix a self loop can be represented by making the corresponding diagonal entry to be greater than 1, but doing so is uncommon, since it is usually convenient to represent each element in the matrix by a single bit.

Trivial Graph: Trivial Graph Format (TGF) is a simple text – based file format for describing graphs. It consists of a list of node definitions, which map node IDs to labels, followed by a list of

edges, which specify node pairs and an optional edge label. Node IDs can be arbitrary identifiers, whereas labels for both nodes and edges are plain strings.

The graph may be interpreted as a directed or undirected graph. For directed graphs, to specify the concept of bidirectional in an edge, one may either specify two edges (forward and back), or differentiate the edge by means of a label.

Bipartite Graph: A bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets and such that every edge connects a vertex in to one in; that is, and are each independent sets. Equivalently, a bipartite graph is a graph that does not contain any odd – length cycles.

One often writes to denote a biparitie graph whose partition has the parts and, if a bipartite graph is not connected, it may have more than one bipartition; [5] in this case, the notation is helpful in specifying one particular bipartition that may be importance in an application. If, that is, if the two subsets have equal cardinality, then is called a balance bipartite graph. [3] If vertices on the same side of the bipartition have the same degree, then is called biregualr.

OPERATION ON GRAPH

Creating a Graph

To create a graph, first adjacency list array is created to store the vertices name, dynamically at the run time. Then the node is created and linked to the list array if an edge is there to the vertex.

Searching and Deleting from a Graph

Suppose an edge (1, 2) is to be deleted from the graph G. First we will search through the list array whether the initial vertex of the edge is in llist array or not by incrementing the list array index. Once the initial vertex is found in the list array, the corresponding link list will be search for the terminal vertex.

METHODS OF GRAPH TRAVERSING

Many application of graph requires a structured system to examine the vertices and edges of a graph G. That is a graph traversal, which means visiting all the nodes of the graph. There are two graph traversal methods.

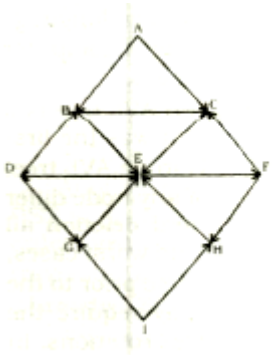
(a) Breadth First Search (BFS)

(b) Depth First Search (DFS)

Breadth First Search

An input graph G = (V, E) and a source vertex S, from where the searching starts. The breadth first search systematically traverses the edges of G to explore every vertex that is reachable from S. Then we examine all the vertices neighbor to source vertex S. Then we traverse all the neighbours of the neighbours of source vertex S and so on. A queue is used to keep track of the progress of traversing the neighbor nodes.

Vertex	Adjacency list
A	B, C
B	C, D, E



C	**E, F**
D	**G**
E	**D, F**
F	**H**
G	**E**
H	**E, G**
I	**G, H**

Depth First Search

The depth first search (DFS), as its name suggest is to search deeper in the graph, whenever possible. An input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node. Then we travel through each node along a path, which begins at S. That is we list a neighbor vertex of S and again an neighbour of a neighbor of S, and as on. The implementation of BFS is almost same except a stack is used instead of the queue.

MINIMUM SPANNING TREE

A minimum spanning tree (MST) for a graph G = (V, E) is a sub graph G1 = (V1, E1) of G contains all the vertices of G.

1. The vertex set V1 is same as that at graph G.
2. The edge set E1 is a subset of G.
3. And there is no cycle.

If a graph G is not connected graph, then it cannot have any spanning tree. In this case, it will have a spanning forest. Suppose a graph G with n vertex then the MST will have (n – 1) edges, assuming to the graph is connected.

A minimum spanning tree (MST) for a weight graph is a spanning tree with minimum weight.

That is all the vertices in the weighted graph all be connected with minimum edge with minimum weights.

SHORTEST PATH

A path from a source vertex a to b is said to be shortest path if there is no other path from a to be with lower weights. There are many instances, to ding in shortest path for travelling from one place to another. That is to find which route can reach as quick as possible or a route for which the travelling cost in minimum.

SEARCHING TECHNIC

Searching is a process of checking and finding a element from a list of elements. Information retried is one of the most important applications of computers. We are given a name and are asked for an associated telephone listing. We are given an employee name or number and are asked for the personnel records of the employee.

There are the important searching techniques:

1. Linear or sequential searching
2. Binary searching

3. Fibonacci search

Linear or sequential searching

In linear search, each element of an array is read one by one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is not found.

Time Complexity

Time complexity of the linear search is found by number of comparisons made in searching a record.In the best case, the desired element is present in the first position of the array, i.e., only one comparison is mode. So f (n) = o (1) .

In the Average case, the desired element is found inn the first position of the array, then f (n) = o [(n + 1)/2].

But in the worst case the desired element is present in the nth (or last) position of the array. so n comparisons are made. so f (n) = o (n + 1).

Binary search

Binary search is an extremely efficient algorithm when it is compared to liner search. Binary search technique searches "data" in minimum possible comparisons. Suppose the given array is a sorted the array elements. Then apply the following conditions to search a "data".

1. Find the middle element of the array (i.e., n/2 is the middle element if the array or the sub array contains n elements).

2. Compare the middle element with the data to be searched, then there are following three cases.

(a) If it is a desired element, then search is successful.

(b) If it is less than desired data, then search only the first half of the array, i.e. the elements which come to the right side of the middle element.

Repeat the same steps until an element are found or exhaust the search area.

Time Complexity

Time complexity is measured by the number f (n) of comparisons to locate "data" in A, which contain n elements. Observe that in each comparison the size of the search area is reduced by half. Hence in the worst case, at most $\log_2 n$ comparisons required. so f (n) = 0 ($[\log_2 n]+ 1$).

Time complexity in the average case is almost approximately equal to the running time of the worst case.

Fibonacci search

A possible improvement in binary search is not to use the middle element at each step, but to guess more precisely where the key being sought falls within the current interval of interest. This improved version is called Fibonacci search. Instead of splitting the array in the middle. this implementation splits the array corresponding to the Fibonacci numbers, which are defined in the following manner:

f0 = 0, f1 = 1

fn= fn-1 + fn- for n > = 2.

HASHING AND HASH FUNCTIONB

Hashing is a very common technique for storing data in such a way the data can be inserted are retrieved very quickly. Hashing uses a data structure called a has table. The searching time of the each

searching technique, depends on the comparison. i.e., n comparisons required for an array A with n elements. To increase the efficiency i.e., to reduce the searching time, we need to avoid unnecessary comparisons.

HASH FUNCTION

the basic idea of hash function is the transformation of the key into the corresponding location in the hash table. a Hash function H can be defined as a function that takes key as input and transforms it into a hash table index. Hash function are of two types:

1. Distribution – independent function
2. Distribution – Dependent function

We are dealing with Distribution – independent function. Following are the most popular Distribution-independent hash functions:

1. Division method
2. mid square method
3. Folding method.

Division Method : TABLE is an array of database file where the employee details are stored. Choose a number m, Which is larger than the number of keys k. i.e., m is greater than the total number of records in the TABLE. The number m is usually chosen to be prime number to minimize the collection. The hash function H is defined by

$H(k) = k \pmod m$

where H (k) is the hash address (or index of the array) and here k (mod m) means the remainder when k is divided by m.

Mid Square Method : The key k is squared . Then the hash function H is defined by

$H(k) = k^2 = 1$

Where 1 is obtained by digits from both the end of k^2 starting from left. Same numbers of digits must be used for all of the keys.

Folding Method: The key $k, k_1, k_2 \ldots\ldots k_r$ is partitioned into number of parts. The parts have same number of digits as the required hash address, except possibly for the last part. Then the parts are added together, ignoring the last carry. That is

$H(k) = k_1 + k_2 + \ldots\ldots\ldots\ldots k_r$

HASH COLLISION

It is possible that two non- identical keys k1, k2 are hashed into same hash address. This situation is called Hash collision.

Let us consider a hash table having 10 locations with Division method is used to hash the key.

$H(k) = k \pmod m$

Here m is chosen as 10. The Hash function produces any integer between 0 and 9 inclusions, depending on the value of the key. If we want to insert a new record with key 500 then

$H(500) = 500 \pmod{10} = 0$.

The location 0 in the table is already filled (i.e., not empty) .thus collision occurred.

Collisions are almost impossible to avoid but it can be minimized considerably by introducing any one of the following three techniques.

1. Open addressing
2. Chaining
3. Bucket addressing

Open Addressing

In open addressing method, when a key is colliding with another key, the collision is resolved by finding a nearest empty space by probing the cells.

Suppose a record R with key K has address H(k) = h. then we will linearly search h + I (where I = 0, 1,2,…….m) locations for free space (i.e., h+1, h+2, h+3…….hash address).

Location	(keys)	Records
0	210	
1	111	
2	883	
3	334	
4		
5	488	
6		
7		
8		
9		

Chaining

in chaining technique the entries in the hash table are dynamically allocated and entered into a linked list associated with each hash key.

Bucket Addressing

Another solution to the hash collision problem is to store colliding elements in the same position in table by introducing a bucket with each hash address. A bucket is a block of memory space, which is large enough to store multiple items.

Hash Deletion

A data can be deleted from a hash table. In chaining method, deleting an element leads to the deletion of a node a linked list. But in linear probing when a data is deleted with its key the position of the array index is made free. The situation is same for other open addressing methods.

SORTING

sorting is used to arrange names and numbers in meaningful ways. For examples; it is easy to look in the dictionary for a word if it is arranged (or sorted) in alphabetic order.

Let A be a list of n elements A1, A2……..An in memory. Sorting of list A refers to the operation of rearranging the contents of a so that they are in increasing (or decreasing) order (numerically or lexicographically); a1 < A2 < A# < ……< an.

Sorting can be performed in many ways. Over a time several methods are being developed to sort data (s). Bubble sort, selection sort, Quick sort, Merge sort, Heap sort, Binary sort, Shell sort and radix sort

BUBBLE SORT

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. then next element is compared with its adjacent element and the same process is repeated for all the elements in the arrays until we get a sorted array.

Time Complexity

The time complexity for bubble sort is calculated in terms of the numbers of comparisons f (n) (or of number of loops); here two loops (outer loop and inner loop) iterates (or repeated) the comparisons. The number of times the outer loop iterate3s is determined by the number of elements in the list which is asked to sort (say it is n). The inner loop is iterated one less than the number of elements in the list (i.e., n-1 times) and is reiterated upon every iteration of the outer loop

f (n) = (n-1) + (n-2) +……..+ 2 + 1

= n (n-1) = 0 (n^2).

SELECTION SORT

Selection sort algorithm finds the smallest element of the array and interchanges it with the elements in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

Time Complexity

Time complexity of a selection sort is calculated in terms of the number of comparisons f (n). In the first pass it makes n-1 comparisons; the second pass makes n-2 comparisons and so on. The outer for loop iterates for (n-1) times. But the inner loop iterates for n * (n-1) times to complete the sorting

f(n) = (n-1)+ (n-2) +…….+2+1

=(n (n-1))/2

+ o (n2)

INSERTION SORT

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file. Compare the second element with first. if the first element is greater than second; place it before the first one. Otherwise place is just after the first one. Compare the third value with second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the second. Otherwise place the second value to the third place. And compare third value with the first value. If the third value to second place, Otherwise place the first value to second place. And place the third value to first place and so on.

time Complexity

shell sort is introduced to improve the efficiency of simple insertion sort. Shell Sort is also called diminishing increment sort. In this method, sub-array, contain kth element of the original array, are sorted The complexity of shell sort is f(n) = 0 (n (log n)2)

QUICK SORT

Quick sort is one of the widely used sorting techniques and it is also called the partition exchange sort. Quick sort is a very efficient sorting algorithm invented by C.A.R Hoare. It has two phases :

- The partition phase and
- The sort phase

As we will see, most of the work is done in the partition phase- it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase. These makes Quick sort a good example of the divide and conquer strategy for solving problems. In quick sort, we divide the array of items to be sorted into two partitions and then call the quick sort procedure recursively to sort the two partitions, i.e. we divide the problem into two smaller ones and conquer by solving the smaller ones.

MERGE SORT

Merge sort works using the principle that if you have two sorted lists, you can merge them together to form another sorted list. Consequently, sorting a large list can be thought of as a problem of starting two smaller lists and then merging those two lists together. Merge sort is also our first divide-and-conquer sort. The term divide sub- problems, each of which is then solved by applying the same approach, until the sub- problems are small the same approach, until the sub-problems are small enough to be solved immediately. Merge sort guarantees O (nlog (n)) complexity because it always splits the work in half.

HEAP SORT

A heap is a partially sorted sorted binary tree. Although a heap is not completely in order, it conforms to a sorting principle; every node has a value less than either of its children. Additionally, a help is a "complete tree" ---- a complete tree is one in which there are no gaps between leaves. For instance a tree with a root node that has only one child must have its child as the left node. More precisely. a complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right , with no breaks.

Pigeonhole principle: In mathematics, the pigeonhole principle states that if n items are put into m pigeonhole must contain more than one item. This theorem is exemplified in real –life by truisms like "there must be at least two left gloves or two right gloves in a group of three gloves". It is an example of a counting argument, and despite seeming intuitive it can be used to demonstrate possibly unexpected results; for example, that two people in London have the same number of hairs on their heads

Examples:

```
#include <stdio.h>
Int main (void)
{
charch;
while ()ch = getchar())!='#')
putchar (ch)
Return 0;
}
```

ANSIC associates the stdio.h header file with using getchar() and putchar(), which is why we have included that file in the program. (Typically, getchar() and putchar() are not true functions, but are defined using preprocessor macros.

******* IACE *******