LEARNING MATERIAL ON

DATA STRUCTURE

(For 3rd Semester CSE)

Submitted By:
Dr. Santosh Kumar Sharma

Data Structure (Syllabus)

Semester & Branch: 3rd sem CSE/IT
Theory: 4 Periods per Week
Total Periods: 60 Periods per Semester
Examination: 3 Hours

Teachers Assessment : 10 Marks
Class Test : 20 Marks End Semester Exam : 70 Marks
TOTAL MARKS : 100 Marks

Objective :

The effectiveness of implementation of any application in computer mainly depends on the that how effectively its information can be stored in the computer. For this purpose various -structures are used. This paper will expose the students to various fundamentals structures arrays, stacks, queues, trees etc. It will also expose the students to some fundamental, I/0 manipulation techniques like sorting, searching etc

INTRODUCTION: **04**

Explain Data, Information, data types
Define data structure & Explain different operations
Explain Abstract data types
Discuss Algorithm & its complexity
Explain Time, space tradeoff

STRING PROCESSING **03**

Explain Basic Terminology, Storing Strings
State Character Data Type,
Discuss String Operations

3.0 ARRAYS **07**

Give Introduction about array,
Discuss Linear arrays, representation of linear array In memory
Explain traversing linear arrays, inserting & deleting elements
Discuss multidimensional arrays, representation of two dimensional arrays in memory (row major order & column major order), and pointers
Explain sparse matrices.

STACKS & QUEUES **08**

Give fundamental idea about Stacks and queues
Explain array representation of Stack
Explain arithmetic expression ,polish notation & Conversion
Discuss application of stack, recursion
Discuss queues, circular queue, priority queues.

LINKED LIST **08**

Give Introduction about linked list
Explain representation of linked list in memory

Discuss traversing a linked list, searching,
Discuss garbage collection.
Explain Insertion into a linked list, Deletion from a linked list, header linked list

TREE **08**

Explain Basic terminology of Tree
Discuss Binary tree, its representation and traversal, binary search tree, searching,
Explain insertion & deletion in a binary search trees

GRAPHS **06**

Explain graph terminology & its representation,
Explain Adjacency Matrix, Path Matrix

SORTING SEARCHING & MERGING **08**

Discuss Algorithms for Bubble sort, Quick sort,
Merging
Linear searching, Binary searching.

FILE ORGANIZATION **08**

Discuss Different types of files organization and their access method,
Introduction to Hashing, Hash function, collision resolution, open addressing..

Books

1.Data Structure by S. Lipschutz - (Schaum Series)
2.Introduction to Data Structure in C by :A.N.Kamthane; Pearson Education
3.Data Strcture using C by Reema Thereja, Oxford University Press

INTRODUCTION

Data

Data is a set of values of qualitative or quantitative variables. Data in computing (or data processing) is represented in a structure that is often tabular (represented by rows and columns), a tree (a set of nodes with parent-children relationship), or a graph (a set of connected nodes). Data is typically the result of measurements and can be visualized using graphs or images.

Data as an abstract concept can be viewed as the lowest level of abstraction, from which information and then knowledge are derived.

Unprocessed data which is also known as *raw data* refers to a collection of numbers, characters and is a relative term; data processing commonly occurs by stages, and the "processed data" from one stage may be considered the "raw data" of the next. Field data refers to raw data that is collected in an uncontrolled environment. Experimental data refers to data that is generated within the context of a scientific investigation by observation and recording.

Information

Information is that which informs us with some valid meaning, i.e. that from which data can be derived. Information is conveyed either as the content of a message or through direct or indirect observation of some thing. Information can be encoded into various forms for transmission and interpretation. For example, information may be encoded into signs, and transmitted via signals.

Information resolves uncertainty. The uncertainty of an event is measured by its probability of occurrence and is inversely proportional to that. The more uncertain an event, the more information is required to resolve uncertainty of that event. In other words, *information is the message* having different meanings in different

contexts. Thus the concept of information becomes closely related to notions of constraint, communication, control, data, instruction, knowledge, meaning,

understanding, perception & representation.

Data Type

Data types are used within type systems, which offer various ways of defining, implementing and using the data. Different type systems ensure varying degrees of type safety.

Almost all programming languages explicitly include the notion of data type. Though different languages may use different terminology. Common data types may include:

- Integers,
- Booleans,
- Characters,
- Floating-point numbers,
- Alphanumeric strings.

For example, in the Java programming language, the "int" type represents the set of 32-bit integers ranging in value from -2,147,483,648 to 2,147,483,647, as well as the operations that can be performed on integers, such as addition, subtraction, and multiplication. Colors, on the other hand, are represented by three bytes denoting the amounts each of red, green, and blue, and one string representing that color's name; allowable operations include addition and subtraction, but not multiplication.

Most programming languages also allow the programmer to define additional data types, usually by combining multiple elements of other types and defining the valid operations of the new data type. For example, a programmer might create a new data type named "complex number" that would include real and imaginary parts. A data type also represents a constraint placed upon the interpretation of data in a type system, describing representation, interpretation and structure of values or objects stored in computer memory. The type system uses data type information to check correctness of computer programs that

access or manipulate the data.

Classes of data types

There are different classes of data types as given below.

 Primitive data type

 Composite data type

 En-numerated data type

 Abstract data type

 Utility data type

 Other data type

Primitive data types

All data in computers based on digital electronics is represented as bits (alternatives 0 and 1) on the lowest level. The smallest addressable unit of data is usually a group of bits called a byte (usually an octet, which is 8 bits). The unit processed by machine code instructions is called a word (as of 2011, typically 32 or 64 bits). Most instructions interpret the word as a binary number, such that a 32-

bit word can represent unsigned integer values from 0 to $2^{32}-1$ or signed integer values from -2^{31} to $2^{31}-1$. Because of two's complement, the machine language and machine doesn't need to distinguish between these unsigned and signed data types for the most part.

There is a specific set of arithmetic instructions that use a different interpretation of the bits in word as a floating-point number. Machine data types need to be *exposed* or made available in systems or low-level programming languages, allowing fine-grained control over hardware. The C programming language, for instance, supplies integer types of various widths, such as short and long. If a corresponding native type does not exist on the target platform, the compiler will break them down into code using types that do exist. For instance, if a 32-bit

integer

is requested on a 16 bit platform, the compiler will tacitly treat it as an array of two 16 bit integers. Several languages allow binary and hexadecimal literals, for convenient manipulation of machine data.

In higher level programming, machine data types are often hidden or abstracted as an implementation detail that would render code less portable if exposed. For instance, a generic numeric type might be supplied instead of integers of some specific bit-width.

Boolean type

The Boolean type represents the values true and false. Although only two values are possible, they are rarely implemented as a single binary digit for efficiency reasons. Many programming languages do not have an explicit boolean type, instead interpreting (for instance) 0 as false and other values as true.

Numeric types

Such as:

- The integer data types, or "whole numbers". May be subtyped according to their ability to contain negative values (e.g. unsigned in C and C++). May also have a small number of predefined subtypes (such as short and long in C/C++); or allow users to freely define subranges such as 1..12 (e.g. Pascal/Ada).
- Floating point data types, sometimes misleadingly called reals, contain fractional values. They usually have predefined limits on both their maximum values and their precision. These are often represented as decimal numbers.
- Fixed point data types are convenient for representing monetary values. They are often implemented internally as integers, leading to predefined limits.
- Bignum or arbitrary precision numeric types lack predefined limits. They are not primitive types, and are used sparingly for efficiency reasons.

Composite / Derived data types

Composite types are derived from more than one primitive type. This can be done in a number of ways. The ways they are combined are called data

structures. Composing a primitive type into a compound type generally results in a new type, e.g. *array-of-integer* is a different type to *integer*.

- An array stores a number of elements of the same type in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Arrays may be fixed-length or expandable.
- Record (also called tuple or struct) Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*.
- Union. A union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float *and* an integer; whereas, in a union, there is only one value at a time.
- A tagged union (also called a variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
- A set is an abstract data structure that can store certain values, without any particular order, and no repeated values. Values themselves are not retrieved from sets, rather one tests a value for membership to obtain a boolean "in" or "not in".
- An object contains a number of data fields, like a record, and also a number of program code fragments for accessing or modifying them. Data structures not containing code, like those above, are called plain old data structure.

Many others are possible, but they tend to be further variations and compounds of the above.

Enumerated Type

This has values which are different from each other, and which can be compared and assigned, but which do not necessarily have any particular concrete representation in the computer's memory; compilers and interpreters can represent them arbitrarily. For example, the four suits in a deck of playing cards may be four enumerators named *CLUB*, *DIAMOND*, *HEART*, *SPADE*, belonging to an enumerated type named *suit*. If a variable *V* is declared having *suit* as its data type, one can assign any of those four values to it. Some implementations allow programmers to assign integer values to the enumeration values, or even treat them as type-equivalent to integers.

String and text types

Such as:

 Alphanumeric character. A letter of the alphabet, digit, blank space, punctuation mark, etc.

 Alphanumeric strings, a sequence of characters. They are typically used to represent words and text.

Character and string types can store sequences of characters from a character set such as ASCII. Since most character sets include the digits, it is possible to have a numeric string, such as "1234". However, many languages would still treat these as belonging to a different type to the numeric value 1234.

Character and string types can have different subtypes according to the required character "width". The original 7-bit wide ASCII was found to be limited and superseded by 8 and 16-bit sets.

Abstract data types

Any type that does not specify an implementation is an abstract data type. For instance, a stack (which is an abstract type) can be implemented as an array (a contiguous block of memory containing multiple values), or as a linked list (a set

of non-contiguous memory blocks linked by pointers).

Abstract types can be handled by code that does not know or "care" what underlying types are contained in them. Arrays and records can also contain underlying types, but are considered concrete because they specify how their contents or elements are laid out in memory.

Examples include:

- A queue is a first-in first-out list. Variations are Deque and Priority queue.
- A set can store certain values, without any particular order, and with no repeated values.
- A stack is a last-in, first out.
- A tree is a hierarchical structure.
- A graph.
- A hash or dictionary or map or Map/Associative array/Dictionary is a more flexible variation on a record, in which name-value pairs can be added and deleted freely.
- A smart pointer is the abstract counterpart to a pointer. Both are kinds of reference

Utility data types

For convenience, high-level languages may supply ready-made "real world" data types, for instance *times*, *dates* and *monetary values* and *memory*, even where the language allows them to be built from primitive types.

Other data types

Types can be based on, or derived from, the basic types explained above. In some languages, such as C, functions have a type derived from the type of their return value. The main non-composite, derived type is the pointer, a data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address. It is a primitive kind of reference. (In everyday terms, a page number in a book could be considered a piece of data that refers to another one). Pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" a pointer whose value

was never a valid memory address would cause a program to crash. To ameliorate this potential problem, pointers are considered a separate type to the type of data they point to, even if the underlying representation is the same.

Data structure

In Computer Science, a data structure is a way of organizing information, so that it is easier to use. Data structures determine the way in which information can be stored in computer and used. If the focus of use is on the things that can be done, people often talk about an abstract data type (ADT). Data structures are often optimized for certain operations. Finding the best data structure when solving a problem is an important part of programming. Programs that use the right data structure are easier to write, and work better.

Basic data structures

Following are some common types of data structures frequently used in computer programming.

Array

The simplest type of data structure is a linear array. This is also called one-dimensional array. An array holds several values of the same kind. Accessing the elements is very fast. It may not be possible to add more values than defined at the start, without copying all values into a new array. In computer science, an array data structure or simply an array is a data structure consisting of a collection of *element*s (values or variables), each identified by at least one *array index* or *key*. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula.[1][2]

For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, 2036, so that the element with index *i* has the address 2000 + 4 × *i.*

Because the mathematical concept of a matrix can be represented as a two-dimensional grid, two-dimensional arrays are also sometimes called matrices. In some cases the term "vector" is used in computing to refer to an array,

although tuples rather than vectors are more correctly the mathematical equivalent. Arrays are often used to implement tables, especially look up tables; the word *table* is sometimes used as a synonym of *array*.

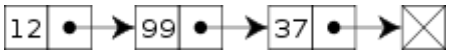
Arrays are among the oldest and most important data structures, and are used by almost every program. They are also used to implement many other data structures, such as lists and strings. They effectively exploit the addressing logic of computers. In most modern computers and many external storage devices, the memory is a one-dimensional array of words, whose indices are their addresses. Processors, especially vector processors, are often optimized for array operations.

Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually, but not always, fixed while the array is in use.[3][4]

The term *array* is often used to mean array data type, a kind of data type provided by most high-level programming languages that consists of a collection of values or variables that can be selected by one or more indices computed at run-time. Array types are often implemented by array structures; however, in some languages they may be implemented by hash tables, linked lists, search trees, or other data structures.

Linked List

A linked list data structure is a set of records linked together by references. The records are often called *nodes*. The references are often called *links* or *pointers*. From here on, the words *node* and *pointer* will be used for these concepts.



Each node points to another node.

In linked data structures, pointers are only dereference or compared for equality. Thus, linked data structures are different than arrays, which require adding and subtracting pointers.

Linked lists, search trees, and expression trees are all linked data structures.

They are also important in algorithms such as topological sort and set union-find.

Stack

A stack is a basic data structure that can be logically thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

The basic implementation of a stack is also called a —Last In First Out|| structure; however there are different variations of stack implementations.

There are basically three operations that can be performed on stacks. They are:

- inserting (—pushing||) an item into a stack
- deleting (—popping||) an item from the stack
- displaying the contents of the top item of the stack (—peeking||)

Queue

A queue is an abstract data type or a linear data structure, in which the first element is inserted from one end (the —tail||), and the deletion of existing element takes place from the other end (the —head||). A queue is a —First In First Out|| structure. The process of adding an element to a queue is called —enqueuing|| and the process of removing an element from a queue is called —dequeuing||.

Graph

A graph is an abstract data type that is meant to implement the graph and hypergraph concepts from mathematics. A graph data structure consists of a finte (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices. As in

mathematics, an edge (*x*,*y*) is said to point or go from *x* to *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references. A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute.

Tree

The tree is one of the most powerful of the advanced data structures and it often appears in advanced subjects such as AI and design. Whenever a tree is used there is a high chance that an index is involved somewhere. The simplest type of index is a sorted listing of the key field. This provides a fast lookup because you can use a binary search to locate any item without having to look at each one in turn.

The trouble with a simple ordered list only becomes apparent once you start adding new items and have to keep the list sorted - it can be done reasonably efficiently but it takes some juggling. Additionally, a linear index isn't easy to share because the whole index needs to be —locked|| when one user edits it, whereas one —branch|| of a tree can be locked, leaving the other branches editable by other users (as they cannot be affected).

Abstract data type

In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

For example, an abstract stack could be defined by three operations:

1. push, that inserts some data item onto the structure,
2. pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and
3. peek, that allows data on top of the structure to be examined without

removal.

When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs.

The term abstract data type can also be regarded as a generalised approach of a number of algebraic structures, such as lattices, groups, and rings.[2] This can be treated as part of the subject area of artificial intelligence. The notion of abstract data types is related to the concept of data abstraction, important in object-oriented programming and design by contract methodologies for software development.

Definition of abstract data type (*ADT*)

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Abstract Data Type in Computer Programming

In the course an abstract data type refers to a generalized data structure that accepts data objects stored as a list with specific behaviors defined by the

methods associated with the underlying nature of the list.

This is a very important BIG idea in computer science. It is based on the recognitation that often a group of data is simply a list in random or some specific order. A list of numbers can be either integer or real but as in arithmetic the numbers can represent any numeric quantity e.g. weight of a list of students in a classroom, in which case we would have a list of real numbers. Mathematical operations e.g. average, min, max are performed exactly the same way on any list of real numbers irrespective of the specific concrete nature of the list.

For example: a group of people queuing at the canteen can be represented as a list with certain characteristics or behaviors. People arrive and attach to the end of the queue, people get served and leave the queue from the head or start of the queue. Any simple queue can be described in exactly the same way. The same basic operations: add, remove, insert are always the same, it does not matter that the data object represent a person or something else. Adding the object to end of the queue is exactly the same operation for any set of data described as a queue.

An ADT has a generalized name e.g. Stack, Queue, Binary Tree etc. Each ADT accepts data objects that are represented as members of the underlying list e.g. an integer, a Person Object. Each ADT has a set of pre-defined methods (behaviors in OOPs terminology) that can be used to manipulate the members in the list - irrespective of what they actually in reality represent.

Some common ADTs, which have proved useful in a great variety of programming applications, are –

- Container
- Deque
- List
- Map
- Multimap
- Multiset

- Priority queue
- Queue
- Set
- Stack
- Tree
- Graph

Each of these ADTs may be defined in many ways and variants, not necessarily equivalent. For example, a stack ADT may or may not have a count operation that tells how many items have been pushed and not yet popped. This choice makes a difference not only for its clients but also for the implementation.

Implementation

Implementing an ADT means providing one procedure or function for each abstract operation. The ADT instances are represented by some concrete data structure that is manipulated by those procedures, according to the ADT's specifications.

Usually there are many ways to implement the same ADT, using several different concrete data structures. Thus, for example, an abstract stack can be implemented by a linked list or by an array.

An ADT implementation is often packaged as one or more modules, whose interface contains only the signature (number and types of the parameters and results) of the operations. The implementation of the module — namely, the bodies of the procedures and the concrete data structure used — can then be hidden from most clients of the module. This makes it possible to change the implementation without affecting the clients.

Algorithm

In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

An algorithm is an effective method expressed as a finite list of well-defined

instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Complexity of Algorithm and Time, space tradeoff

Algorithms In computer science, the are evaluated by the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, Big-omega notation and Big-theta notation are used to this end. For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a *hidden constant*.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing

machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted list to which we apply binary search has *n* elements, and we can guarantee that each lookup of an element in the list can be done in unit time, then at most log2 *n* + 1 time units are needed to return an answer.

Best, worst and average case complexity

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size *n* may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size *n*.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size *n*.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size *n*.

Time complexity

In computer science, the **time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity. For example, if the time required by an algorithm
on all inputs of size *n* is at most 5*n*3 + 3*n*, the asymptotic time complexity is O(*n*3). Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a

fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm, denoted as ***T(n)***, which is defined as the maximum amount of time taken on any input of size *n*. Time complexities are classified by the nature of the function *T*(*n*). For instance, an algorithm with *T*(*n*) = O(*n*) is called a linear time algorithm, and an algorithm with *T*(*n*) = O(2*n*) is said to be an exponential time algorithm.

space complexity

The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

STRING PROCESSING

String

A finite sequence ‗S' of zero or more characters is called a String. The string with zero character is called the empty string or null string.

- The number of characters in a string is called its length.
- Specific string will be denoted by in closing their character in single quotation mark.

For e.g. ; —THE END‖ quotation mark.

‗123'

‗THE' ‖' ‗‖ ‗END' THE END ⟶

Let $S1;_2S$ be the string consist of the character of S_1 followed by the

characters of *S*2 is called the concatenation of *S*1 & *S*2 .

It will be denoted by *S*1 , *S*2 .

For e.g., *S*1 = _XY1‘

*S*2 = _PQR‘

*S*1 || *S*2 = XY1 PQR

*S*1 = _XY1‘, *S*2 = _ _(space), *S*3 = _PQR‘

*S*1 || *S*2 = XY1 (space) PQR

The length of *S*1 || *S*2 || *S*3 is equal to the sum of length string *S*1

string Y is called a substring of a string _S‘ & if their exits string _S‘ & if their exits string X & Z. Such that S=X || Y || Z. If X is an empty string, then y is called & initial substring of _S‘ & Z is an empty string then _Y‘ is called a terminal substring of _S‘.

CHARACTER DATA TYPE:-

- The character data type is of two data type. (1) Constant (2) Variable

Constant String:

-> The constant string is fixed & is written in either _ ‘ single quote & — || double quotation.

Ex:- _SONA‘

—Sona||

Variable String:

String variable falls into 3 categories.

1.Static

2.Semi-Static

3.Dynamic

Static character variable:

Whose variable is defined before the program can be executed & cannot change

throughout the program.

Semi-static variable:

Whose length variable may as long as the length does not exist, a maximum value.

A maximum value determine by the program before the program is executed.

Dynamic variable:

A variable whose length can change during the execution of the program.

String Operation:

There are four different operations.

1.Sub string

2.Indexing

3.Concatenation

4.Length

Sub string:-

Group of conjunctive elements in a string (such as wards, purchases or sentences) called substring.

Accessing substring of a given string required 3 pieces of information.

a. The name of the string or the string itself.

b. The position of the first character of the substring in the given string.

c. The length of the substring of the last character of the substring. We called this operation SUBSTRING.

SUBSTRING (String, initial, length)

To denote the substring of string _S' beginning in the position _K' having a length _L'.

SUBSTRING (S, K, L) **T** **K L**

For e.g.; SUBSTRING (_TO BE OR NOT TO BE', 4, 7)

SUBSTRING=BE OR N

SUBSTRING (THE END, 4, 4)

SUBSTRING= END.

INDEXING:-

Indexing also called pattern matching which refers to finding the position where a string pattern _P‘. First appears in a given string text _T‘, we called this operation index and write as INDEX (text, pattern)

If the pattern _P‘ does not appear in text _T‘ then index is assign the value 0; the argument & text and pattern can either string constant or string variable.

For e.g.; T contains the text.

_HIS FATHER IS THE PROFESSOR‘

Then INDEX (T, _THE‘)

7

INDEX (T, _THEN‘)

0

INDEX (T, _ THE‘)

10

Concatenation:-

Let S1 & 2 in be the string then concatenation of S1 & 2 is denoted by S 1 2 , 1 || 2 , each the string consist of the character of 1 followed by the characters of 2 .

Ex:- S1 = _Sonalisa‘ S2 = _ ‘ S3 = _Behera‘

1 || 2 || 3 = Sonalisa Behera

Length operation:-

The number of character in a string is called its length. We will write LENGTH (string).For the length of a given string LENGTH (—Computer||). The length is 8.

Basic language LEN (STRING)

Strlen (string)

Strupper(string)

String upper

Strupr(_computer‘)

COMPUTER

String lower

Strlwr (_COMPUTER‘)

COMPUTER

String concatenatingStrcnt

String Reverse Strrev

ARRAY

Linear Array:

Linear Array

A is a list of finite number of n homogeneous data elements i.e. the elements of same data types Such that:

a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.

b) The elements of the array are stored respectively in the successive memory locations.

The number n of elements is called length or size of array. If not explicitly stated, we will assume the index set consists of integers 1, 2, 3 ...n. In general the length or the number of data elements of the array can be obtained from the index set by the formula

Length= UB – LB + 1

Where UB is the largest index, called the upper bound, and LB is the smallest index, called the lower bound. Note that length = UB when LB = 1.

The elements of an array A are denoted by subscript notation à a1, a2, a3...an

Or by the parenthesis notation -> A (1), A (2),...., A(n)

Or by the bracket notation -> A[1], A[2],....,A[n].

We will usually use the subscript notation or the bracket notation.

Representation of Linear Arrays in memory:

Let LA is a linear array in the memory of the computer. Recall that the memory of computer is simply a sequence of addressed locations.

LOC (LA[k]) = address of element LA[k] of the array LA.

As previously noted, the elements of LA are stored in the successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

Base (LA)

And called the base address of LA. Using base address the computer calculates the address of any element of LA by the following formula:

LOC (LA[k]) = Base (LA) + w (k-lower bound)

Where w is the number of words per memory cell for the array LA.

OPERATIONS ON ARRAYS

Various operations that can be performed on an array

- Traversing
- Insertion
- Deletion
- Sorting
- Searching
- Merging

Traversing Linear Array:

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A, this can be accomplished by traversing A, that is, by accessing and processing each element of a exactly ones.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked list, can also be easily traversed. On the other hand, traversal of nonlinear structures, such as trees and graph, is considerably more complicated.

Algorithm: (Traversing a Linear Array)

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

1. [Initialize counter] Set k: =LB.
2. Repeat steps 3 and 4 while k <=UB.
3. [Visit Element] Apply PROCESS to LA [k].
4. [Increase Counter] Set k: =k + 1.

[End of step 2 loop]

5.Exit.

OR

We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop.

Algorithm: (Traversing a Linear Array)

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

1. Repeat for k = LB to UB:

Apply PROCESS to LA [k].

[End of loop]

2. Exit.

Caution: The operation PROCESS in the traversal algorithm may use certain variables which must be initialized before PROCESS is applied to any of the elements in the array. Accordingly, the algorithm may need to be proceeded by such an initialization step.

Insertion and Deletion in Linear Array:

Let A be a collection of data elements in the memory of the computer.

—Inserting‖

refers to the operation of adding another element to the collection A, and

—deleting‖ refers to the operation of removing one element from A.

Inserting an element at the end of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in

the middle of the array. Then, on the average, half of the elements must be moved downward to new location to accommodate the new elements and keep the order of the other elements.

Similarly, deleting an element at the end of the array presents no difficulties, but deleting the element somewhere in the middle of the array requires that each subsequent element be moved one location upward in order to fill up the array.

The following algorithm inserts a data element ITEM in to the Kth position in the linear array LA with N elements.

Algorithm for Insertion: (Inserting into Linear Array)

INSERT (LA, N, K, ITEM)

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

1. [Initialize counter] Set J: = N.
2. Repeat Steps 3 and 4 while j >= k;
3. [Move jth element downward.] Set LA [J + 1]: =LA [J].
4. [Decrease counter] Set J: = J-1

[End of step 2 loop]

5. [Insert element] Set LA [K]:=ITEM.
6. [Reset N] Set N:=N+1
7. EXIT.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm for Deletion: (Deletion from a Linear Array)

DELETE (LA, N, K, ITEM)

Here LA is a Linear Array with N elements and K is the positive integer such that K<=N. This algorithm deletes the Kth element from LA.

1. Set ITEM: = LA [k].

2. Repeat for J = K to N – 1.

[Move J + 1st element upward] Set LA [J]: = LA [J +1].

[End of loop]

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

4. EXIT

Multidimensional Array

1.Array having more than one subscript variable is called **Multi-Dimensional array**.

2.Multi Dimensional Array is also called as **Matrix**.

Consider the Two dimensional array -

Two Dimensional Array requires **1. Two Subscript Variables**

2.Two Dimensional Array stores the values in the form of matrix.

Row 3.One Subscript Variable denotes the —|| of a matrix.

Column 4.Another Subscript Variable denotes the —|| of a matrix.

Declaration and Use of Two Dimensional Array :

int a[3][4];

Use :

```
for(i=0;i<row;i++)
  for (j=0;j<col;j++)
{
  printf("%d",a[i][j]);
}
```

Meaning of Two Dimensional Array :

1.Matrix is having 3 rows (i takes value from 0 to 2)

2.Matrix is having 4 Columns (j takes value from 0 to 3)

a 3.Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.

4.Name of 2-D array is __ and each block is identified by the row & column

number.

5. Row number and Column Number Starts from 0.

Cell Location	Meaning
a[0][0]	0th Row and 0th Column
a[0][1]	0th Row and 1st Column
a[0][2]	0th Row and 2nd Column
a[0][3]	0th Row and 3rd Column
a[1][0]	1st Row and 0th Column
a[1][1]	1st Row and 1st Column
a[1][2]	1st Row and 2nd Column
a[1][3]	1st Row and 3rd Column
a[2][0]	2nd Row and 0th Column
a[2][1]	2nd Row and 1st Column
a[2][2]	2nd Row and 2nd Column
a[2][3]	2nd Row and 3rd Column

Two-Dimensional Array : Summary with Sample Example

Summary Point Explanation	
No of Subscript Variables Required 2	
Declaration a[3][4]	
No of Rows 3	
No of Columns 4	
No of Cells 12	
No of for loops required to iterate 2	

Memory Representation

1. 2-D arrays are Stored in contiguous memory location **row wise**.

2. 3 X 3 Array is shown below in the first Diagram.

3. Consider **3×3 Array is stored in Contiguous memory** location which starts from 4000 .

4. Array element **a[0][0]** will be stored at address **4000** again **a[0][1]** will be stored to next memory location i.e Elements stored row-wise

5. After **Elements of First Row are stored** in appropriate memory location , elements of next row get their corresponding mem. locations.

6. This is integer array so each element requires 2 bytes of memory.

Array Representation:

- Column-major
- Row-major

Arrays may be represented in Row-major form or Column-major form. In Row-major form, all the elements of the first row are printed, then the elements of the second row and so on up to the last row. In Column-major form, all the elements of the first column are printed, then the elements of the second column and so on up to the last column. The ‗C' program to input an array of order m x n and print the array contents in row major and column major is given below. The following array elements may be entered during run time to test this program:

Output:

Row Major:

1 2 3

4 5 6

7 8 9

Column Major:

1 4 7

2 5 8

3 6 9

Basic Memory Address Calculation :

a[0][1] = a[0][0] + Size of Data Type

Element	Memory Location
a[0][0]	4000
a[0][1]	4002
a[0][2]	4004
a[1][0]	4006
a[1][1]	4008
a[1][2]	4010
a[2][0]	4012
a[2][1]	4014
a[2][2]	4016

Array and Row Major, Column Major order arrangement of 2 d array

An array is a list of a finite number of homogeneous data elements. The number of elements in an array is called the array length. Array length can be obtained from the index set by the formula

Length=UB-LB+1

Where UB is the largest index , called the upper bound and LB is the smallest index , called the lower bound. Suppose int Arr[10] is an integer array. Upper bound of this array is 9 and lower bound of this array is 0 ,so the length is 9- 0+1=10.

In an array ,the elements are stored successive memory cells. Computer does not need to keep track of the address of every elements in memory. It will keep the address of the first location only and that is known as base **address of an array**. Using the base address , address of any other location of an array can be calculated by the computer. Suppose Arr is an array whose base address is Base(Arr) and w is the number of memory cells required by each elements of the array Arr.The address of Arr[k] – k being the index value can be obtained by

using the formula :

Address(Arr[k])=Base(Arr)+w(k-LowerBound)

2 d Array :- Suppose Arr is a 2 d array. The first dimension of Arr contains the index set 0,1,2, ... row-1 (the lower bound is 0 and the upper bound is row-1) and the second dimension contains the index set 0,1,2,... col-1(with lower bound 0 and upper bound col-1.)

The length of each dimension is to be calculated .The multiplied result of both the lengths will give you the number of elements in the array.

Let‘s assume Arr is an two dimensional 2 X 2 array .The array may be stored in memory one of the following way :-

1. Column by column i,e column major 2. Row by row , i,e in row major order order. The following figure shows both representation of the above array.

By row-major order, we mean that the **elements in the array** are so arranged that the subscript at the extreme right varies fast than the subscript at it‘s left., while in column-major order , the subscript at the extreme left changes rapidly ,
then the subscript at it‘s right and so on. 1,1
2,1
1,2
2,2

ColumnMajorOrder
1,1
1,2

2,1
2,2

Row major order

Now we know that computer keeps track of only the base address. So the address of any **specified location of an array** , for example Arr[j,k] of a 2 d array Arr[m,n] can be calculated by using the following formula :- (Column major order
) Address(Arr[j,k])= base(Arr)+w[m(k-1)+(j-1)] (Row major order)
Address(Arr[j,k])=base(Arr)+w[n(j-1)+(k-1)] For example Arr(25,4) is an array with base value 200.w=4 for this array. The address of Arr(12,3) can be calculated using row-major order as

Address(Arr(12,3))=200+4[4(12-1)+(3-
1)] =200+4[4*11+2]
=200+4[44+2]
=200+4[46]
=200+184
=384

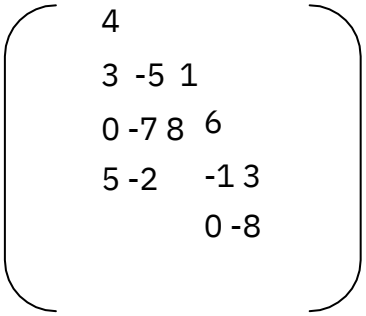
Again using column-major order
Address(Arr(12,3))=200+4[25(3-1)+(12-1)]
=200+4[25*2+11]
=200+4[50+11]

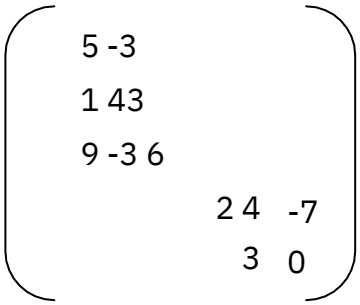
=200+4[61]
=200+244
=444

Sparse matrix

Matrix with relatively a high proportion of zero entries are called sparse matrix. Two general types of n-square sparse matrices are there which occur in various

applications are mention in figure below(It is sometimes customary to omit blocks of zeros in a matrix as shown in figure below)





Triangular matrix

Tridiagonal matrix

Triangular matrix

This is the matrix where all the entries above the main diagonal are zero or equivalently where non-zero entries can only occur on or below the main diagonal is called a (lower)Triangular matrix.

Tridiagonal matrix

This is the matrix where non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a Tridiagonal matrix. The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices i.e. one may save space by storing only those entries which may be non-zero.

STACKS & QUEUES

Fundamental idea about Stacks

In computer science, a **stack** is a particular kind of abstract data type or collection in which the principal (or only) operations on the collection are the addition of an entity to the collection, known as *push* and removal of an entity, known as *pop*. The relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) data structure. In a LIFO data structure, the last element added to the structure must be the first one to be removed. This is equivalent to the requirement that, considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the *top* of the stack. Often a *peek* or *top* operation is also implemented, returning the value of the top element without removing it.

A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition. Therefore, the lower elements are those that have been on the stack the longest.

Array representation of Stack

In most high level languages, a stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations. The following will demonstrate both implementations, using C.

Array

The **array implementation** is trivial: an array where the first element (usually at the zero-offset) is the bottom. That is, array[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack. The stack itself can therefore be effectively implemented as a two-element structure in C:

```
typedef struct {
size_t size;
    int items[STACKSIZE];
} STACK;
```

The push() operation is used both to initialize the stack, and to store values to it. It is responsible for inserting (copying) the value into the ps->items[] array and for incrementing the element counter (ps->size). In a responsible C implementation, it is also necessary to check whether the array is already full to prevent an overrun.

Algorithm 1: PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an item on to a stack.

1. [Stack already filled]?

If TOP = MAXSTK, then: Print: OVERFLOW, and Return.

2. Set TOP: = TOP + 1. [Increase TOP by 1].

3. Set STACK [TOP]: = ITEM. [Inserts ITEM in new TOP position]. 4.

Return.

Algorithm 2: POP (STACK, TOP, ITEM)

This procedure deletes the TOP element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed]

If TOP = 0, then: Print: UNDERFLOW and Return.
2. Set ITEM: =STACK [TOP]. [Assign TOP element to ITEM]. 3.
Set TOP: = TOP – 1 [Decrease TOP by 1].
4. Return.

If we use a dynamic array, then we can implement a stack that can grow or shrink as much as needed. The size of the stack is simply the size of the dynamic array. A dynamic array is a very efficient implementation of a stack, since adding items to or removing items from the end of a dynamic array is dynamically with respect to time.

Arithmetic expression, polish notation & Conversion

Arithmetic

The expression for adding the numbers 1 and 2 is, in prefix notation, written "+ 1 2" rather than "1 + 2". In more complex expressions, the operators still precede their operands, but the operands may themselves be nontrivial expressions including operators of their own. For instance, the expression that would be written in conventional infix notation as
(5 – 6) × 7
can be written in prefix as
× (– 5 6) 7
Since the simple arithmetic operators are all binary (at least, in arithmetic contexts),
any prefix representation thereof is unambiguous, and bracketing the prefix expression is unnecessary. As such, the previous expression can be further simplified to
× – 5 6 7
The processing of the product is deferred until its two operands are available (i.e., 5 minus 6, and 7). As with *any* notation, the innermost expressions are

evaluated first, but in prefix notation this "innermost-ness" can be conveyed by order rather than bracketing.
In the classical notation, the parentheses in the infix version were required, since moving them
5 – (6 × 7)
or simply removing them
5 – 6 × 7
would change the meaning and result of the overall expression, due to the precedence rule.

Similarly
5 – (6 × 7)
can be written in Polish notation as
– 5 × 6 7

Polish notation

Polish notation, also known as Polish prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. The Polish logician Jan Łukasiewicz invented this notation in 1924 in order to simplify sentential logic.

The term *Polish notation* is sometimes taken (as the opposite of *infix notation*) to also include Polish *postfix* notation, or Reverse Polish notation, in which the operator is placed *after* the operands.

When Polish notation is used as a syntax for mathematical expressions by interpreters of programming languages, it is readily parsed into abstract syntax trees and can, in fact, define a one-to-one representation for the same.

Conversion between Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?). There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Example of expression in three notations.

Infix	**Prefix**	**Postfix**
a + b + a b a b +		
a + b * c + a * b c a b c * +		
(a + b) * (c - d) * + a b - c d	a b + c d - * b * b - 4 * a * c	
40 - 3 * 5 + 1		
Postfix expressions are easily evaluated with the aid of	a stack.	

Postfix Evaluation Algorithm

Assume we have a string of operands and operators, an informal, by hand process is

1. Scan the expression left to right
2. Skip values or variables (operands)
3. When an operator is found, apply the operation to the preceding two operands
4. Replace the two operands and operator with the calculated value (three

symbols are replaced with one operand)

5. Continue scanning until only a value remains--the result of the expression

Algorithm: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add the right parentheses||)|| at the end of P.
2. Scan P from left to right and repeat step 3 and 4 for each element of P until the sentinel —)|| is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator is encountered, then:

a. Remove the two top elements from the STACK

b. Evaluates these two operators using that operator.

c. Place the result of (b) back on STACK.

[End of If structure]

[End of step 2 loop]

5. Set VALUE equal to the top element of STACK.
6. EXIT.

Transforming Infix to Postfix Expression: Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right parentheses. We assume that the operators in Q consists only the exponentials, multiplication, Division, addition and subtractions, and that they have the usual three level of precedence as given above. We also assume that the operators on the same level, including exponentiations, are performed from left to right unless otherwise indicated by the parentheses.

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right

parenthesis at the end of Q. The algorithm is completed when stack is empty.

Algorithm: Polish (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push —(—onto STACK, and add —)|| to the end of Q.
2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator is encountered, then:

a. Repeatedly POP from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than that operator.

b. Add that operator to STACK.

[End of if structure]

6. If a right parenthesis is encountered, then:

a. Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered.

b. Remove the left parenthesis.[Do not add the left parenthesis to P].

[End of if structure]

[End of step 2 loop].

7. EXIT.

Application of stack, recursion

Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other, the nested images that occur are a form of infinite recursion. The term has a variety of meanings specific to a variety of disciplines ranging from linguistics to logic. The most common application of recursion is in mathematics and computer science,

in which it refers to a method of defining functions in which the function being defined is applied within its own definition. Specifically, this defines an infinite number of instances (function values), using a finite expression that for some instances may refer to other instances, but in such a way that no loop or infinite chain of references can occur. The term is also used more generally to describe a process of repeating objects in a self-similar way.

A classic example of recursion is the definition of the factorial function, given here in C code:

```
unsigned int factorial(unsigned int n) {
if (n == 0) {
return 1;
} else {
return n * factorial(n - 1);
    }
}
```

Queues:

Queue is a linear list of elements in which deletions can take place only at one end, called the front and insertions can take place only at the other end, called the rear. The terms**"front"** and **"rear"** are used in describing a linear list only when it is implemented as a queue.

Queue are also called **first-in first-out** (FIFO) lists, since the first elements enter a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are **last-in first-out** (LIFO) lists.

Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue. In which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the

first person to be waited on; and so on. An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed.

Representation of queues:

Queues may be represented in the computer in various ways, usually by means at one-way lists or linear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty. Following figure shows the way the array in Figure will be stared in memory using an array QUEUE with N elements. Figure also indicates the way elements will be deleted from the queue and the way new elements will be added to the queue. Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

FRONT: = Rear + 1

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment

REAR: = Rear +1

This means that after N insertion, the rear element of the queue will occupy QUEUE [N] or, in other words; eventually the queue will occupy the last part of the array. This occurs even though the queue itself may not contain many elements. Suppose we want to insert an element ITEM into a queue will occupy the last part of the array, i.e., when REAR=N. One way to do this is to simply move the entire queue to thee beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive. The procedure we adopt is to assume that the array QUEUE is circular, that is, that QUEUE [1] comes after QUEUE [N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE [1]. Specifically, instead of

increasing REAR to N+1, we reset REAR=1 and then assign
QUEUE [REAR]: = ITEM
Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N +1. (Some readers may recognize this as modular arithmetic, discussed in Sec. 2.2)
Suppose that our queue contains only one element, i.e., suppose that
FRONT = REAR # NULL
And suppose that the element is deleted. Then we assign
FRONT: = NULL and REAR: = NULL

Priority Queues:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the which following rule:
(1) An element of higher priority is processes before any element of lower priority.
(2) Two elements with the same priority are processes according to the order in which they were added to the queue.
A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.
There are various ways of maintaining a priority queue in memory. We discuss two of them here: one uses a one – way list, and the other uses multiple queues. The ease or difficulty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-Way List of Representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one – way list, as follows:
(a) Each node in the list will contain three items of information — an information field INFO, a priority number PRN and a link number LINK
(b) A node X precedes a node Y in the list (1) when X has higher priority that Y or (2) when both have the same priority but X was added to the list before Y. This

means that the order in the One-way list corresponds to the order of the priority queue.
Priority numbers will operate in the usual way: the priority number, the higher the priority.

Polish Notations: From most common arithmetic operations, the operator symbol is placed between its two operands. For example,
A + B C – D E * F G/H
This is called ***Infix Notation***. With this notation we must distinguish between
(A + B) *C and A + (B * C)
Polish notation, named after the polish mathematician refers to the notation in which the operator symbol is placed before its two operands. For example:
+AB -CD *EF /GH
We translate step by step the following infix expressions into polish notations using bracket [ ] to indicate the partial translation:
(A+B) * C= [+AB]*C = *+ ABC
The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and the operands in the expression. Accordingly, one never needs parentheses when writing expressions in polish notations.
Reverse Polish Notation refers to the analogous notation in which the operator symbol is placed after its two operands:
AB+ CD- EF* GH/
Again, one never needs parentheses to determine the order of operations in any arithmetic expressions written in reverse polish notation. This notation is frequently
called ***POSTFIX*** (or***SUFIX***) notation, whereas ***prefix notation*** is term used for polish notations, discussed in preceding paragraph.
The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to postfix notation, and then it evaluates the post fix expression. In each step, the stack is the main tool that is

used to accomplish the given task.
Evaluation of a Postfix Expression: Suppose P is the arithmetic expression written in postfix notation. The following algorithm, which uses a stack to hold operands, evaluates P.

What is Pointer

A pointer variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:
type *var-name;
Here, **type** is the pointer's base type; it must be a valid C data type and **var- name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

int *ip; /* pointer to an integer */
double *dp;/* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.
How to use Pointers?
There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value

of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
    int var = 20; /* actual variable declaration */
          int *ip; /* pointer variable declaration */

   ip = &var; /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

int main ()
{
   int *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr );

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

The value of ptr is 0

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

C Pointers in Detail:

Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to a C programmer:

LINKED LIST

It is the linear collection of data elements called nodes that are stored in different memory location connected by pointers.

Advantages of linked list:-

Dynamic data structure that can grow an string. Efficient memory utilization (exact amount of data storage). Insertion deletion & pupation are easy & efficient. Data stored in RAM but not sequential .

Disadvantages of linked list:-

More memory space is needed if no. of filed are more. Logical & physical ordering of node are different. Searching is solve . Difficult to program because pointer manipulation is required.

Types of linked list:-

Linear linked list or one way linked list or single list. Double linked list or two way linked list are two way linked list. Circular linked list is two types i.e. (1) Single circular list (2) Double circular list.

- Linear linked list:- It is a one way collection of nodes where the linear order is maintained by pointers. Nodes are not in sequence, each node implemented in comp. by a self referential structure . Each node is divided in two parts. First part contain the information of the element (INFO). Second part is linked field contains the add. Of next node in the list (LINK) field or next pointer filed .In c linked list is created using structured pointer and Mallow (Allocation of memory). The structure of a node is strict node.

```
{
Into info;
Strict node*link;
};
```

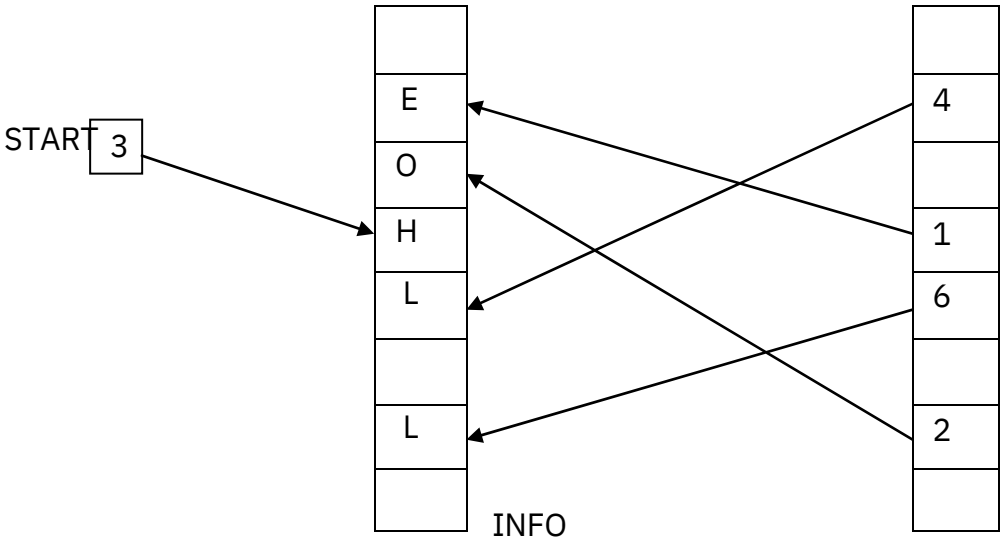
The new node is created and addresses of the new node is assigned to stack has start.

{START =(Strict node*) mallow(size of (strict node))}

Representation of the linked list in memory:-

Let list be a linked list. Then list required to linear arrays called INFO and LINK.

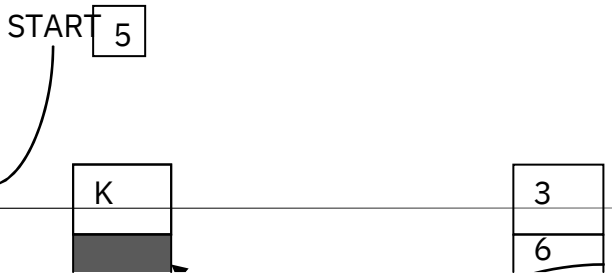
Such that INFO[k] and LINK[k] contain the information part & next pointer denoted by a NULL which indicates the end of the LIST.



Memory allocation of the linear linked list:-

The computer maintains a special list which consist of a list of all free memory calls & also has its own pointer is called the list of available space or the free storage list or the free pool .

Suppose insertion are to be performed on linked list then unused memory calls I the array will also be linked to gather to form a linked list using AVAIL. As its list pointer variable such a data structure will be denoted by writing LIST(INFO,LINK,START,AVAIL)



INFO LINK

AVAIL

Avail Space

Garbage collection definition:-

The operating system of a computer may periodically collect all the deleted space on to the free storage list. Any technique which does these collections is called garbage collection.

When we delete a particular note from an existing linked list or delete the linked list the space occupied by it must be given back to the free pool. So that the memory can be the used by some other program that needs memory space.

To the free pool is done.

The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space. The OS scans through the entire memory cell & marks those cells. That are being by some program then it collects the entire cell which are not being used & add to the free pool. So that this cells can be used by other programs. This process is called garbage collection. The garbage collection is invisible to the programmer.

Traversing of linked list:-

Algorithm:-

Let list be a linked list in memory, this algorithm traverse LIST applying & operation PROCESS to each element of LIST. The variable PTR to point to the nodes currently being processed.

Step 1:-set PTR=START [initialize pointer PTR]

Step 2:-repeat step 3 & step 4 while PTR! = NULL

Step 3:-apply PROCESS to INFO[PTR]

STEP 4:-SET PTR=LINK [PTR]

[PTR now points to the next node]

End of step 2 loop

Step 5:-exit

Algorithm for searching linked list:-

SEARCH (INFO,LINK,START,ITEM,LOC)

LIST is a linked list in memory , this algorithm finds the location LOC of the node where ITEM first appears in LIST or sets , LOC=NULL.

Step 1:-set PTR=START[initialize pointer PTR]

Step 2:-repeat step 3 while PTR ! = NULL

Step 3:-if ITEM = INFO[PTR]

Then set LOC = PTR & exit

Else

Set PTR = LINK[PTR]

[PTR now points to next node]

[End of if structure]

End of step 2 loop

Step 4:-[Search is unsuccessful]

Set LOC = NULL

Step 5:-Exit

Inserting the node at the beginning of the list:-

INSERT (INFO, LINK, START, AVAIL, ITEM)

Step 1:-[over flow?]

If AVAIL = NULL, then write over flow & exit

Step 2:-[REMOVE first node from AVAIL LIST]

Set NEW = AVAIL & AVAIL = LINK [AVAIL]

Step 3:-set INFO [NEW] = ITEM

[Copy is new data into new node]

Step 4:-set LINK [NEW] = START

[New node now points to original first node]

Step 5:-set START = NEW

[changes start so its point to new node]

Step 6:-exit

Inserting after a given node:-

INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM. so that ITEM follows the node with location [LOC] or insert ITEM as the first node when LOC = NULL

Step 1:-[over flow] if AVAIL = NULL, then write overflow & exit.

Step 2:-[Remove first node from AVAIL list]

Set NEW = AVAIL & AVAIL = LINK [AVAIL]

Step 3:-set INFO [NEW] = ITEM

[Copy is new data into new node]

Step 4:-if LOC = NULL, then [insert as first node]

Set LINK [NEW] = START & START = NEW

Else

[INSERT after node with location LOC]

Set LINK [NEW] = LINK [LOC] and LINK [LOC] = NEW

[End of if]

Step 5:-Exit

Deletion from a linked list:-

DEL (INFO, LINK, START, AVAIL, LOC, LOCP)

This algorithm deletes the node N with location LOC, LOCP is the location of the node LOCP = NULL.

Step 1:-if LOCP = NULL, then set = START=LINK [START]

[Delete first node]

Else

Set = LINK [LOCP] = LINK [LOC]

[Deletes node N]

[End of if]

Step 2:-[Return deleted node to the AVAIL LIST]

Set = LINK [LOC] = AVAIL & AVAIL = LOC

Step 3:-Exit

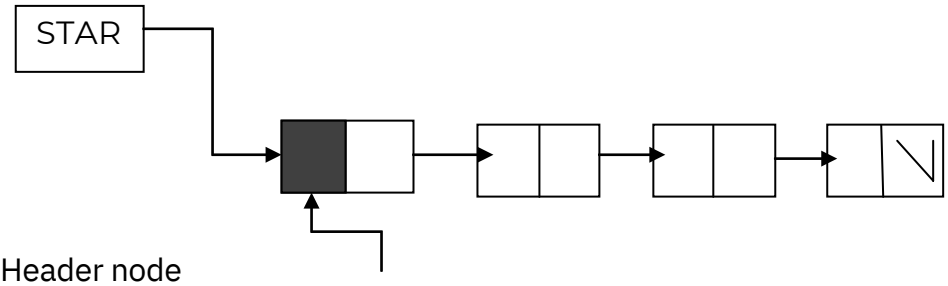
Header linked list:-

A header linked list is a special type of linked list. Which always contains a special node called header node at the beginning of the list so in a header linked list will not point to first node of the list. But start will contain the address of the header node.

There are two types of header linked list i.e. Grounded header linked list & Circular header linked list.

Grounded header linked list:-

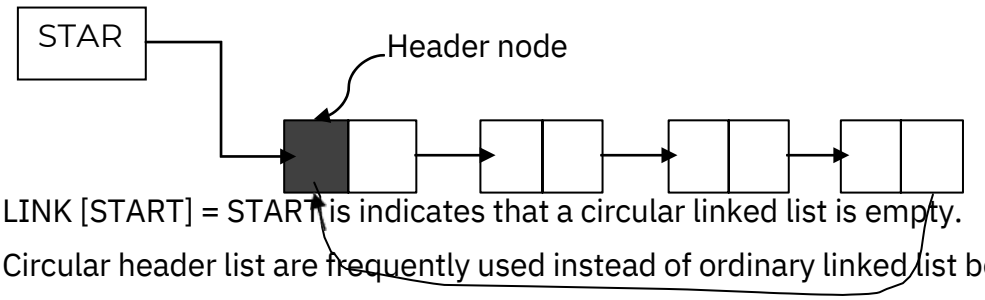
It is a header linked list where last node contains the NULL pointer. LINK [start] = NULL indicates that a grounded header linked list is empty.



Header node

Circular header linked list:-

It is a header linked list where the last node points back to the header node.



LINK [START] = START is indicates that a circular linked list is empty.

Circular header list are frequently used instead of ordinary linked list because

many operation are much easier to implement header list.

This comes from the following two properties, all circular header lists.

- The NULL pointer is not used & hence contains valid address.
- Every ordinary node has a predecessor. So the first node may not required a special case.

TREE

A tree is a non-linear data structure that consists of a root node and potentially many levels of additional nodes that form a hierarchy. A tree can be empty with no nodes called the **null or** empty tree or a tree is a structure consisting of one node called the **root** and one or more subtrees.

Thus tree is a finite set of one or more nodes such that : i>

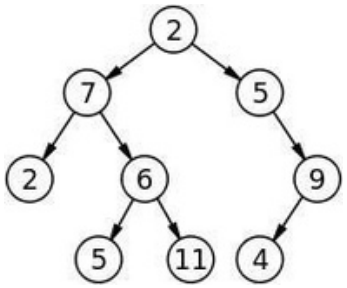
There is a specially designated node called the root

ii> The remaining nodes are partitioned into n >= 0 disjoint sets T1,T2,.....,Tn where each of these sets is a tree . T1,T2,.....,Tn are called the subtrees of the root .

Terminologies used in Trees

- **Root** - the top node in a tree.
- **Node** - the item of information .
- **Parent** - the converse notion of *child*.
- **Siblings** - nodes with the same parent.
- **Children nodes** – roots of the subtrees of a node , X , are the children of X .
  - **Descendant** - a node reachable by repeated proceeding from parent to child.
  - **Ancestor** - a node reachable by repeated proceeding from child to parent.
  - **Leaf or Terminal node** - node with no children (degree zero) .
  - **Nonterminal nodes** – nodes other than terminal nodes .
  - **Internal node** - node with at least one child.
  - **External node** - node with no children.
  - **Degree** - number of sub trees of a node.

- **Edge** - connection between one node to another.
- **Path** - a sequence of nodes and edges connecting a node with a descendant.
- **Level** - The level of a node is defined by 1 + the number of connections between the node and the root.
- **Height** - The height of a node is the length of the longest downward path between the node and a leaf.
- **Forest** - A forest is a set of n ≥ 0 disjoint trees. If we remove the root of a tree we get a forest.



A simple unordered tree

The node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.

Binary Tree

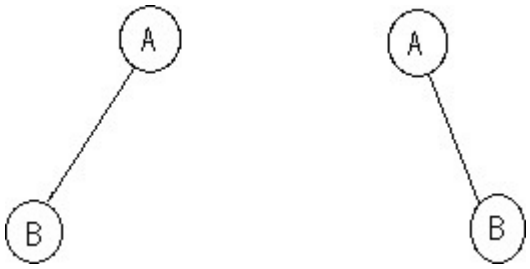
Definition: A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree. We can define the data structure binary tree as follows:

structure BTREE
declare CREATE( ) --> btree
ISMTBT(btree,item,btree) --> boolean
MAKEBT(btree,item,btree) --> btree
LCHILD(btree) --> btree

DATA(btree) --> item
RCHILD(btree) --> btree
for all p,r in btree, d in item **let**
ISMTBT(CREATE)::=**true**
ISMTBT(MAKEBT(p,d,r))::=**false**
LCHILD(MAKEBT(p,d,r))::=p; LCHILD(CREATE)::=error
DATA(MAKEBT(p,d,r))::d; DATA(CREATE)::=error
RCHILD(MAKEBT(p,d,r))::=r; RCHILD(CREATE)::=error

end
end BTREE

This set of axioms defines only a minimal set of operations on binary trees. Other operations can usually be built in terms of these. The distinctions between a binary tree and a tree should be analyzed. First of all there is no tree having zero nodes, but there is an empty binary tree. The two binary trees below are different. The first one has an empty right subtree while the second has an empty left subtree. If these are regarded as trees, then they are the same despite the fact that they are drawn slightly differently.



Binary Tree Representations

A full binary tree of depth k is a binary tree of depth k having pow(2,k)-1 nodes. This is the maximum number of the nodes such a binary tree can have. A very elegant sequential representation for such binary trees results from sequentially numbering the nodes, starting with nodes on level 1, then those on level 2 and so on. Nodes on any level are numbered from left to right. This numbering scheme

gives us the definition of a complete binary tree. A binary tree with n nodes and a depth k is complete iff its nodes correspond to the nodes which are numbered one to n in the full binary tree of depth k. The nodes may now be stored in a one dimensional array tree, with the node numbered i being stored in tree(i).

If a complete binary tree with n nodes (i.e., depth=[LOG2N]+1) is represented sequentially as above then for any node with index i, 1<=i<=n we have

(i) parent(i) is at [i/2] if i is not equal to 1. When i=1, i is the root and has no parent.

(ii) lchild(i) is at 2i if 2i <= n . If 2i > n, then i has no left child.

(iii) rchild(i) is at 2i+1 if 2i+1 <= n .If 2i+1 > n, then i has no right child.

Proof: We prove (ii). (iii) is an immediate consequence of (ii) and the numbering of nodes on the same level from left to right. (i) follows from (ii) and (iii). We prove

(ii) by induction on i. For i=1, clearly the left child is at 2 unless 2>n in which case 1 has no left child. Now assume that for all j, 1<=j<=i lchild(j) is at 2j .Then the two nodes immediately preceeding lchild(i+1) in the representation are the right child

of I and the left child of I .The left child of i is at 2i .Hence,the left child of i+1 is at

2i+2=2(i+1) unless 2(i+1)>n inwhich case i+1 has no left child. This representation

can clearly be used for all binary trees though in most cases there will be a lot of unutilized space. For complete binary trees the representation is ideal as no space is wasted.

Binary Tree Traversal

A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree we want treat each node and its subtrees in the same fashion. If we let L, D, R stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: LDR, LRD, DLR, DRL, RDL, and RLD. If we adopt the convention that we traverse left before right then only three traversals remain: LDR, LRD, and DLR. To these we assign the names inorder, postorder and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression.

Inorder Traversal: informally this calls for moving down the tree towards the left untilyou can go no farther. Then you "visit" the node, move one node to the right and continue again. If you cannot move to the right, go back one more node. A precise way of describing this traversal is to write it as a recursive procedure.

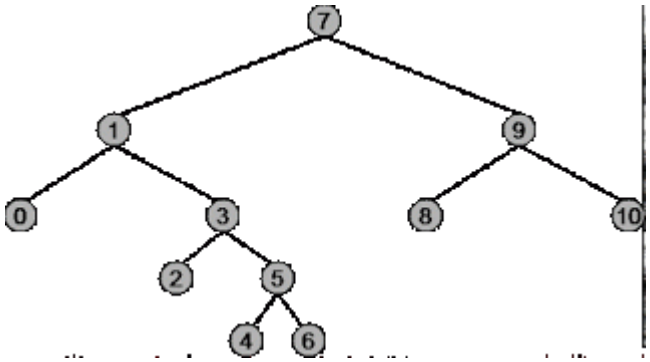
```
Procedure INORDER(T)
// T is a binary tree where each node has three fields LCHILD,DATA,RCHILD //
If T<> 0 then [call INORDER(LCHILD(T))
Print (DATA(T))
Call (INORDER(RCHILD(T))]
end INORDER
```

A second form of traversal is preorder:

```
procedure PREORDER (T)
If T<> 0 then [print (DATA(T))
Call PREORDER(LCHILD(T))
Call PREORDER(RCHILD(T))]
end PREORDER
```

The ext traversal method is called postorder .

```
procedure POSTORDER (T)
If T<> 0 then [ call POSTORDER(LCHILD(T)) Call
POSTORDER(RCHILD(T))]
[print (DATA(T))]
end POSTORDER
```



Therefore, the Preorder traversal of the above tree will be :

7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

Therefore, the Postorder traversal of the above tree will be :

0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Therefore, the Inorder traversal of the above tree will be :

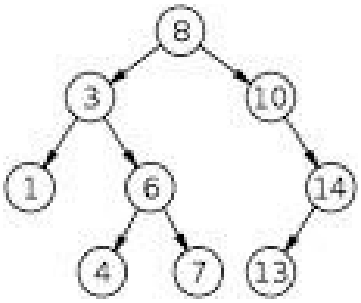
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Binary Search Tree

A binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a node-based binary tree data structure where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right sub-tree. Each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node. The common properties of a binary search tree are :

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- Each node can have up to two successor nodes.

- There must be no duplicate nodes.
- A unique path exists from the root to every other node.



Searching

1.Start at the root node as current node

2.If the search key‘s value matches the current node‘s key then found a match

3.If search key‘s value is greater than current node‘s

a.If the current node has a right child, search right

b.Else, no matching node in the tree

Insertion

4.If search key is less than the current node‘s

a.If the current node has a left child, search left

b.Else, no matching node in the tree

---

1. Always insert new node as leaf node

2. Start at root node as current node

3. If new node‘s key < current‘s key

a. If current node has a left child, search left

b. Else add new node as current‘s left child

4. If new node‘s key > current‘s key

a.If current node has a right child, search right

b.Else add new node as current‘s right child

Deletion

Basically, in can be divided into two stages:

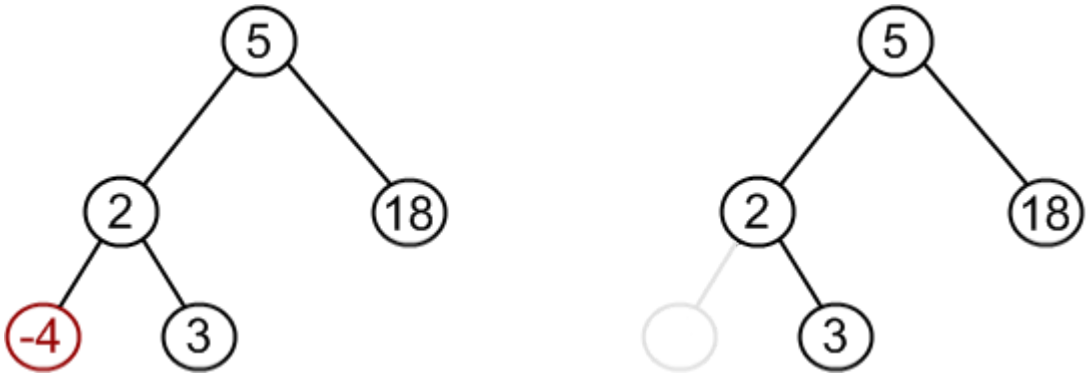
- search for a node to remove;

 if the node is found, run remove algorithm.

1. Node to be removed has no children.

Algorithm sets corresponding link of the parent to NULL and disposes the node.

Example. Remove -4 from a BST.

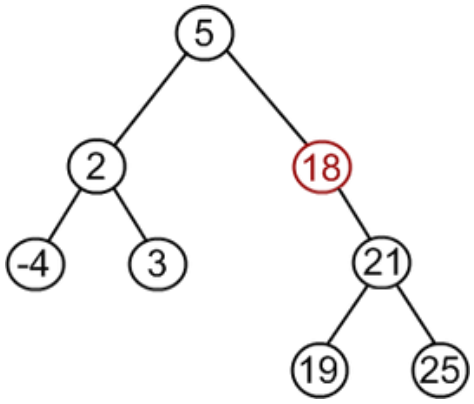


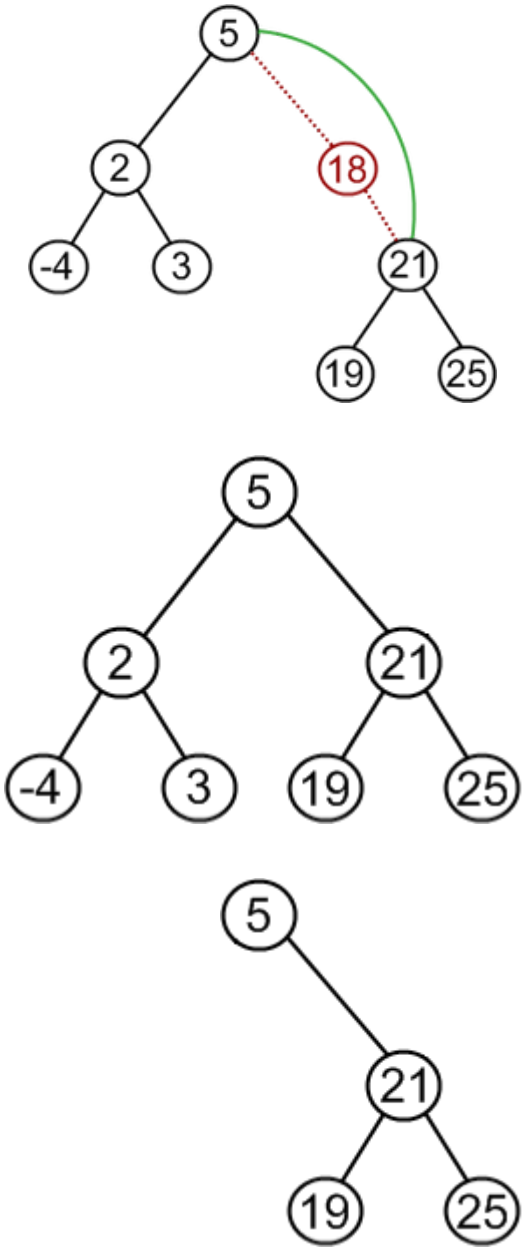
2. Node to be removed has one child.

It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

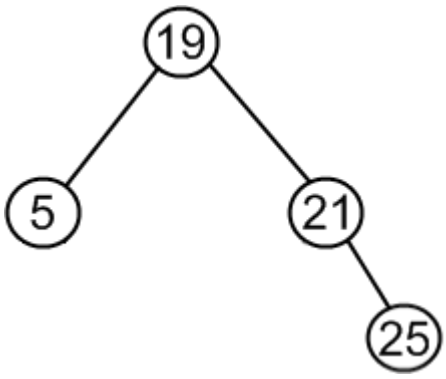
3 . Node to be removed has two children

Example. Remove 18 from a BST.





1 . Transform the above tree into the following tree



Method :

o choose minimum element from the right subtree (19 in the example);

o replace 5 by 19;

o hang 5 as a left child.

2. remove a node, which has two children:

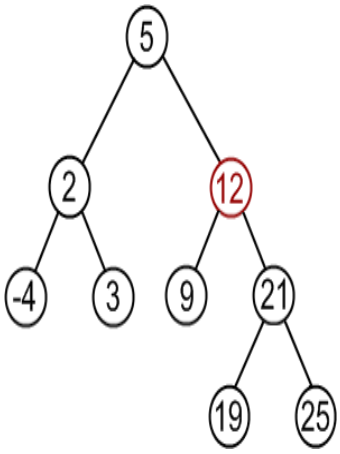
o find a minimum value in the right subtree;

o replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!

o apply remove to the right subtree to remove a duplicate.

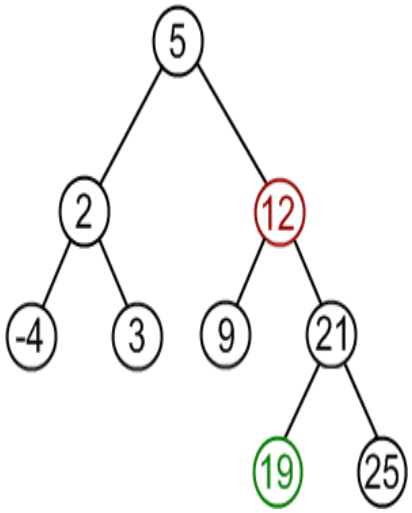
Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

Example. Remove 12 from a BST.

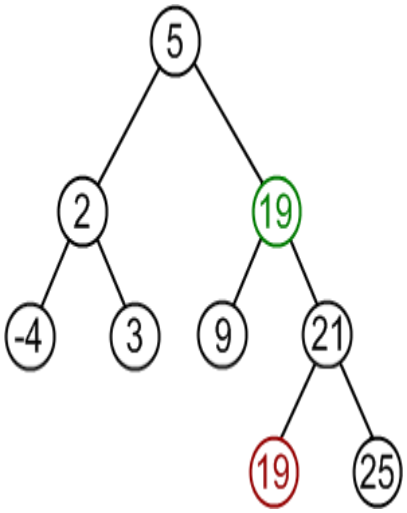


.

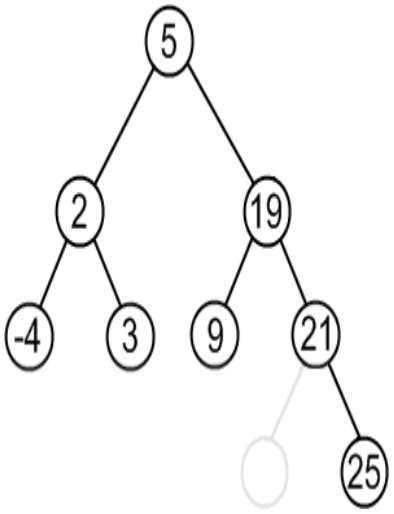
2. Find minimum element in the right subtree of the node to be removed. In current example it is 19.



3. Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



4. Remove 19 from the left subtree



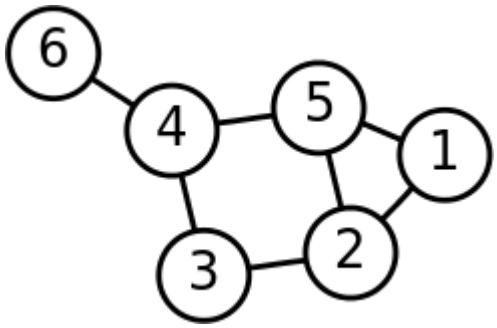
GRAPH

Definition

A graph, ,Gconsists of two sets *V* and *E*. *V* is a finite non-empty set of *vertices. E* is a set of pairs of vertices, these pairs are called *edges*. *V*(*G*) and *E*(*G*) will represent the sets of vertices and edges of graph *G*. We will also write *G* = (*V,E*) to represent a graph.

In an *undirected graph* the pair of vertices representing any edge is unordered . Thus, the pairs (*v*1, *v*2) and (*v*2, *v*1) represent the same edge.

In a *directed graph* each edge is represented by a directed pair (*v*1, *v*2). *v*1 is the *tail* and *v*2 the *head* of the edge. Therefore <*v*2, *v*1> and <*v*1, *v*2> represent two different edges.



Terminologies used in graph

 A graph *G = (V, E)* where

1. V = a set of **vertices**

2. E = a set of **edges**

 Edges:

- Each edge is defined by a pair of vertices
- An edge **connects** the vertices that define it
- In some cases, the vertices can be the same

Vertices:

- Vertices also called **nodes**
- Denote vertices with labels

Representation:

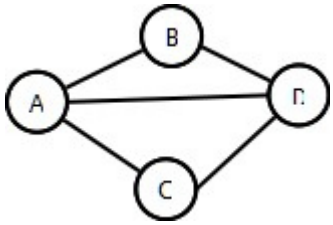
o Represent vertices with circles, perhaps containing a label

o Represent edges with lines between circles

Example:

o V = {A,B,C,D}

o E = {(A,B),(A,C),(A,D),(B,D),(C,D)}



 Path: sequence of vertices in which each pair of successive vertices is

connected by an edge

- **Cycle**: a path that starts and ends on the same vertex
- **Simple path**: a path that does not cross itself
    - That is, no vertex is repeated (except first and last)
    - Simple paths cannot contain cycles
- **Length** of a path: Number of edges in the path
    - Sometimes the sum of the weights of the edges

Representation

Adjacency list

Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency matrix

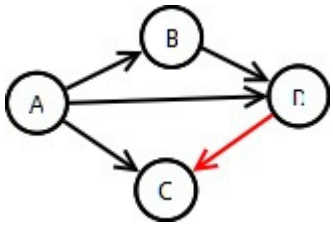
A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

Incidence matrix

A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

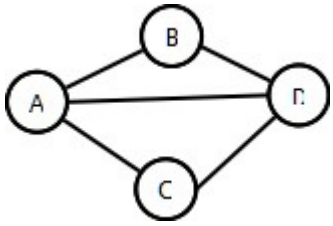
Directed graph

- **Digraphgraph** : A whose edges are directed (i.e have a direction)
    - Edge drawn as arrow
    - Edge can only be traversed in direction of arrow
    - Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}



Undirected Graph

A graph where there is no implied direction on edge between nodes



- In diagrams, edges have no direction (i.e they are not arrows)
- Can traverse edges in either directions

Adjacency matrix representation

Graphs can be classified by whether or not their edges have **weights**

 Weighted graph: edges have a weight

o Weight typically shows cost of traversing

 Unweighted graph: edges have no weight

o Edges simply show connections

 Adjacency Matrix: 2D array containing weights on edges

o Row for each vertex

o Column for each vertex

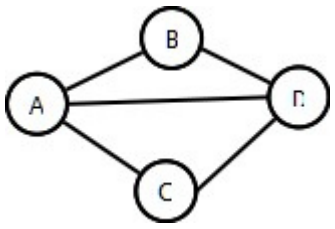
o Entries contain weight of edge from row vertex to column vertex

o Entries contain ∞ (ie Integer'last) if no edge from row vertex to column vertex

o Entries contain 0 on diagonal (if self edges not allowed)

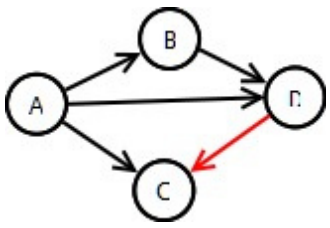
 undirected graph

	A	B	C	D
A	0	1	1	1
B	1	0	∞	1
C	1	∞	0	1
D	1	1	1	0



- directed graph

	A	B	C	D
A	∞	1	1	1
B	∞	∞	∞	1
C	∞	∞	∞	∞
D	∞	∞	1	∞



SORTING SEARCHING & MERGING

SORTING:

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing with numerical data or alphabetically with character

BUBBLE SORT:

The bubble sort has no reading characteristics. It is very slow, no matter what data it is sorting. This algorithm is included for the shake of completeness not because of any merit. As the largest element is bubble of sinks up to its final position. It is known as bubble sort.

Algorithm:

BUBBLE (DATA,N)

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

Step 1: [Loop]

Repeat step 2 and step 3 for K=1 to N-1

Step 2: [Initialize pass pointer PTR]

Set[PTR]=1

Step 3: [Execute pass]

Repeat while PTR <=N-K

a. If DATA [PTR] > DATA [PTR+1]

Then interchange DATA [PTR] & DATA [PTR+1]

[End of if structure]

b. Set PTR =PTR+1

[End of Step 1 Loop]

Step 4: Exit

Complexity of the Bubble Sort Algorithm

The time for a sorting algorithm is measured in terms of the number of

comparisons. The number f(n) of comparisons in the bubble sort is easily computed. There are n-1 comparisons during the first pass, which places the largest element in the last position; there are n-2 comparisons in the second step, which places the second largest element in the next to last position and so on.
F(n)=(n-1)+(n-2)+....+2+1=n(n-1)/2=n^2/2+0(n)=0(n^2)
The time required to execute the bubble sort algorithm is proportional to n^2, Where n is the number of input items.

QUICK SORT:

It is an algorithm of the divide and conquer type. That is the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Algorithm:

Step 1: [Initialize]
TOP=NULL
Step 2: [Push boundary values of A onto stacks when A has 2 or more elements.]
If N>1, then: TOP=TOP+1, LOWER [1] =1, UPPER[1] = N
Step 3: Repeat step 4 to 7 while TOP != NULL
Step 4: [Pop sublist from stacks]
Set BEG=LOWER[TOP],END=UPPER[TOP]
TOP=TOP-1
Step 5: Call QUICK(A,N,BEG,ENG,LOC)
Step 6: [Push left sublist onto stacks when it has 2 or more elements]
If BEG<LOC-1, then
TOP=TOP+1, LOWER[TOP]=BEG,
UPPER[TOP]=LOC-1
Step 7: [Push right sublist onto stacks when it has 2 or more elements]
If LOC+1<END, then
TOP=TOP+1, LOWER[TOP]=LOC+1,
UPPER[TOP]=END
[End of if structure]
Step 8: Exit

Complexity of the Quick Sort Algorithm

The running time of a sorting algorithm is usually measured by the number f(n) of comparisons required to sort n elements. The algorithm has a worst case running time of order n^2/2, but an average case running time of order n log n.

The worst case occurs when the list is already sorted. Then the first element will require n comparison to recognize that it remains in the first position. The first sublist will be empty, but the second sublist will have n-1 elements. Accordingly, the second element will require n-1 comparison to recognize that it remains in the second position, and so on.

F(n)=n+(n-1)+....+n(n+1)/2=n^2/2+O(n)=O(n^2)

MERGING:

The operation of sorting is closely related to the process of merging. The merging of two order table which can be combined to produce a single sorted table.

This process can be accomplishes easily by successively selecting the record with the smallest key occurring by either of the table and placing this record in a new table.

SIMPLE MERGE

SIMPLE MERGE [FIRST,SECOND,THIRD,K]

Given two orders in table sorted in a vector K with FIRST, SECOND, THIRD

The variable I & J denotes the curser associated with the FIRST & SECOND table respectively. L is the index variable associated with the vector TEMP.

Algorithm

Step 1: [Initialize]

Set I = FIRST

Set J = SECOND

Set L = 0

Step 2: [Compare corresponding elements and output the smallest]

Repeat while I < SECOND & J < THIRD

If K[I] <= K[J],then L=L+1

TEMP [L]=K[I]

I=I+1

Else L=L+1

TEMP[L]=K[J]

J=J+1

Step 3: [Copy remaining unprocessed element in output area]

If I>=SECOND

Then repeat while J<=THIRD

L=L+1

TEMP[L]=K[J]

J=J+1

Else

Repeat while I< SECOND

L=L+1

TEMP[L]=K[I]

I=I+1

Step 4: [Copy the element into temporary vector into original area]

Repeat for I=1,2,.....L

K[FIRST-I+1]=TEMP[I]

Step 5: Exit

Complexity of the Merging Algorithm

The input consists of the total number n=r+s of elements in A and B. Each comparison assigns an elements to the array C, which eventually has n elements. Accordingly, the number f(n) of comparisons cannot exceed n:

F(n) <= n = 0(n)

In other words, the merging algorithm can be run in linear time.

WORST CASE : n log= 0[n log n]

AVERAGE CASE : n log n=0[n log n]

SEARCHING:

Searching refers to finding the location i.e LOC of ITEM in an array. The search is said to be successful if ITEM appears the array & unsuccessful otherwise we have two types of searching techniques.

1.Linear Search

2.Binary Search

LINEAR SEARCH:

Suppose DATA is a linear array with n elements. No other information about DATA, the most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. First we have to test whether DATA [1]=ITEM, nad then we test whether DATA[2] =ITEM , and so on. This method which traverses DATA sequentially to locate ITEM, is called *linear search* or *sequential search*.

Algorithm

LINEAR (DATA, N, ITEM, LOC)

Step 1: [Insert ITEM at the end of data]

Set DATA [N+1] = ITEM

Step 2: [Initialize counter]

Set LOC=1

Step 3: [Search for ITEM]

Repeat while DATA [LOC]!= ITEM

Step 4: [Successful]

If LOC=N+1

Then Set LOC = 0

Step 5: Exit

Complexity of the Linear Search Algorithm

The complexity of search algorithm is measured by the number f(n) of comparisons required to find ITEM in DATA where DATA contains n elements. Two important cases to consider are the average case and the worst case.

The worst case occurs when one must search through the entire array DATA. In this case, the algorithm requires

F(n)= n+1

Thus, in the worst case, running time is proportional to n.

The running time of the average case uses the probabilistic notion of expectation. The probability that ITEM appears in DATA[K], and q is the probability that ITEM

does not appear in DATA . Since the algorithm uses k comparison when ITEM appears in DATA[K], the average number of comparison is given by

F(n) = 1 . p1 + 2 . p2 +.....+ n . pn + (n+1).q

BINARY SEARCH

Suppose DATA is an array which is sorted in increasing numerical order or equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called *binary search.*

Algorithm

Binary search (DATA, LB, UB, ITEM, LOC)

Step 1: [Initialize the segment variables]

Set BEG := LB, END := UB and MID := INT ((BEG + END)/2)

Step 2: [Loop]

Repeat Step 3 and Step 4 while BEG <= END and DATA [MID] != ITEM

Step 3: [Compare]

If ITEM < DATA [MID]

then set END := MID - 1

Else

Set BEG = MID + 1

Step 4: [Calculate MID]

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

Step 5: [Successful search]

If DATA [MID] = ITEM

then set LOC := MID

Else set LOC := NULL

Step 6: Exit

Complexity of the Binary Search Algorithm

The complexity is measured by the number f(n) of comparison to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most f(n) comparison to locate ITEM where

2f(n) > n

Or equivalently

F(n) = [log2 n] + 1

The running time for the worst case is approximately equal to log2 n and the average case is approximately equal to the running time for the worst case.

FILES AND THEIR ORGANIZATION

INTRODUCTION

Nowadays, most organizations use data collection applications which collect large amounts of data in one form or other. For example, when we seek admission in a college, a lot of data such as our name, address, phone number, the course in which we want to seek admission, aggregate of marks obtained in the last examination and so on, are collected. Similarly, to open a bank account, we need to provide a lot of input. All these data were traditionally stored on paper documents, but handling these documents had always been a chaotic and difficult task. It has become a necessary to store the data in computers in the form of files.

FILE ORGANIZATION

We know that a file is a collection of related records. The main issue in file management is the way in which the records are organized inside the file because it has a significant effect on the system performance. Organization of records means the logical arrangement of records in the file and not the physical layout of the file as stored on a storage media.

Since choosing an appropriate file organization is a design decision, it must be done keeping the priority of achieving good performance with respect to the most likely usage of the file. Therefore, the following considerations should be kept in mind before selecting an appropriate file organization method:

- Rapid access to one or more records.
- Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing records(s).
- Efficient storage of records.

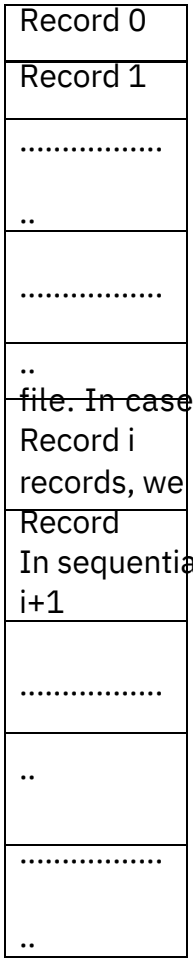
 Using redundancy to ensure data integrity.

Although one may find that these requirements are in contradiction with each other, it is the designer‘s job to find a good compromise among them to get an adequate solution for the problem at hand. For example, the ease of addition of records can be compromised to get fast access to data.

TECHNIQUES COMMONLY USED FOR FILE ORGANIZATION

1. Sequential Organization

A sequential organized file stores the record in the order in which they were entered. That is, the first record that was entered is written as the first record in the file, the second record entered is written as the second record in the file, and so on. As a result new records are added only at the end of the file.



Sequential files can be read only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of records in a file. The figure below shows n records numbered from 0 to n-1 stored in a sequential file. Once we store the records in a file, we cannot make any changes to the records. We cannot even delete the records from a sequential file. In case we need to delete or update one or more records, we have to replace the records by creating a new file.

In sequential file organization, all the records have the same size and the same field format, and every field has a fixed size. The records are sorted based on the value of one field or a combination of two or more fields. This field is known as the key. Each key uniquely identifies a record in a file. Thus, every record has a different value for the key field. Records can be sorted in either ascending or descending order.

Sequential files are generally used to generate reports or to perform

and tapes.

The table below summarizes the features, advantages, and disadvantages of sequential file organization**.**

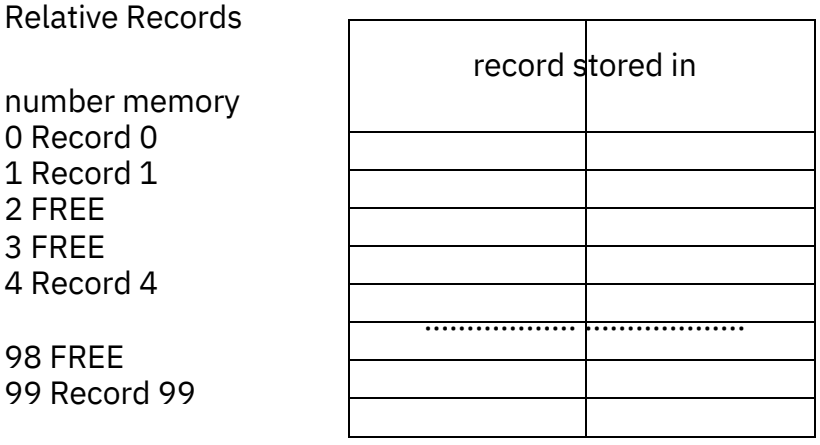
Features	Advantages	Disadvantages
□ Records are written in the order in which they are entered. □ Records are read and written sequentially. □ Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes. □ Records have the same size and the same field format. □ Records are stored on a key value. □ Generally used for report generation or sequential reading.	□ Simple and easy to handle. □ No extra overheads involved. □ Sequential files can be stored on magnetic disks as well as magnetic tapes. □ Well suited for batch oriented applications.	□ Records can be read sequentially. If 1th record has to be read, then all the i-1 records must be read. □ A new file has to be created and the original file has to be replaced with the new file that contains the desired changes. □ Cannot be used for interactive application

2. Relative File Organization

Relative File Organization provides an effective way to access individual records directly in a relative file organization, records are ordered by their relative key. It

means the record number represents the location of the record relative to the beginning of the file. The record numbers range from 0 to n-1, where n is the number of records in the file. For example, the record with number 0 is the first record in the file. The records in a relative file are of fixed length.
Therefore, in relative files, records are organized in ascending relative record number. A relative file can be thought of as a single dimension table stored on a disk, in which the relative record number is the index into the table. Relative files can be used for both random as well as sequential access. For sequential access, records are simply read one after another.

Relative files provide support for only one key, that is, the relative record number. This key must be numeric and must take a value between 0 and the current highest relative record number -1. This means that enough space must be allocated for the file to contain the records with relative record numbers between 0 and the highest record number -1. For example, if the highest relative record number is 1,000 then space must be allocated to store 1,000 records in the file. The figure below shows a schematic representation of a relative file which has been allocated enough space to store 100 records. Although it has space to accommodate 100 records, not all the locations are occupied. The locations marked as FREE are yet to store records in them. Therefore, every location in the table either stores a record or is marked as FREE.



Relative file organization provides random access by directly jumping to the

record which has to be accessed. If the records are of fixed length and we know the base address of the file and the length of the record, then any record i can be accessed using the following formula:

Address of ith record=base_address+(i-1)* record_length

Note that the base address of the file refers to the starting address of the file. We took i-1 in the formula because record numbers start from 0 rather than 1.

Consider the base address of a file is 1000 and each record occupies 20 bytes, then the address of the 5th record can be given as: 1000+ (5-1)*20

=1000+80

=1080

The Table below summarizes the features, advantages, and disadvantages of

Relative file organization.

Features	Advantages	Disadvantages
Provides an effective way to access individual records. The record number represents the location of the record relative to the beginning of the file. Records in a relative file are of fixed length. Relative files can be used for both random as well as sequential access.	Ease of processing. If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously. Random access of records makes access to relative files fast. Allow deletions and updations in the same file. Provides random as well as sequential access of	Use of relative files is restricted to disk devices. Records can be of fixed length only. For random access of records, the relative record number must be known in advance.

□ Every location in the table either stores a record or is marked as FREE.	records with low overhead. □ New records can be easily added in the free locations based on the relative record number of the record to be inserted. □ Well suited for interactive applications.	

3. Indexed Sequential File Organization

Indexed sequential file organization stores data for fast retrieval. The records in an indexed sequential file are of fixed length and every record is uniquely identified by a key field. We maintain a table known as the index table which

Record number	Address of the Record
1	765
2	27
3	876
4	742
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL

stores the record number and the address of all the records. That is for every file, we have an index table. This type of file organization is called as indexed sequential file organization because physically the records may be stored anywhere, but the index table stores the address of those records.

The ith entry in the index table points to the ith record of the file. Initially, when the file is created, each entry in the index table contains NULL. When the ith record of the file is written, free space is obtained from the free space manager and its address is stored in

the ith location of the index table.

Now, if one has to read the 4th record, then there is no need to access the first three records. Address of the 4th record can be obtained from the index table and the record can be straightaway read from the specified address (742, in our example). Conceptually, the index sequential file organization can be visualized as shown in figure.

An indexed sequential file uses the concept of both sequential file uses the concept of both sequential as well as relative files. While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly.

Indexed sequential files perform well in situations where sequential access as well as random access is made to the data. Indexed sequential files can be stored only on devices that support random access, for example, magnetic disks. For example, take an example of a college where the details of students are stored in an indexed sequential file. This file can be accessed in two ways:

- *Sequentially*-to print the aggregate marks obtained by each student in a particular course or
- *Randomly*-to modify the name of a particular student.

The Table below summarizes the features, advantages, and disadvantages of indexed sequential file organization.

Features	Advantages	Disadvantages
Provides fast data retrieval. Records are of fixed length. Index table stores the address of the records in the file. The ith entry in the index table points to the ith record of the file. While the index table is read	The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs. Supports applications that require both	Indexed sequential files can be stored only on disks. Needs extra space and overhead to store indices. Handling these files is more complicated than handling

sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly. □ Indexed sequential files perform well in situations where sequential access as well as random access is made to the data.	batch and interactive processing. □ Records can be accessed sequentially as well as randomly. □ Updates the records in the same file.	sequential files. □ Supports only fixed length records.

HASHING

The search time of each algorithm discussed so far depends on the number n of elements in the collection S of data. This section discusses a searching technique, called hashing or hash addressing, which is essentially independent of the number n.

The terminology which we use in our presentation of hashing will be oriented toward file management. First of all, we assume that there is a file F of n records with a set K of keys which uniquely determine the records in F. Secondly, we assume that F is maintained in memory by a table T of m memory locations and that L is set of memory addresses of the locations in T. For notational convenience, we assume that the keys in K and the addresses in L are (decimal) integers. (Analogous methods will work with binary integers or with keys which are character strings, such as names, since there are standard ways of representing strings by integers.)

The subject of hashing will be introduced by the following example.

Example:

Suppose a company with 68 employees assigns a 4-digit employee number to each employee which is used as the primary key in the company's employee file. We can, in fact, use the employee number as the address of the record in memory. The search will require no comparisons at all. Unfortunately, this technique will require space for 10,000 memory locations, whereas space for fewer than 30 such locations would actually be used. Clearly, this tradeoff of space for time is not worth the expense.

The general idea of using the key to determine the address of a record is an excellent idea, but it must be modified so that a great deal of space is not wasted. This modification takes the form of a function ***H*** from the set ***K*** of keys into the set ***L*** of memory addresses. Such a function,

$$H: K \rightarrow L$$

is called a ***hash function*** or ***hashing function.*** Unfortunately, such a function ***H*** may not yield distinct values: it is possible that two different keys *k1* and *k2* will yield the same hash address. This situation is called ***collision***, and some method must be used to resolve it. Accordingly, the topic of hashing is divided into two parts: (1) hash functions and (2) collision resolutions. The two parts are discussed separately.

1. Hash Functions

The two principal criteria used in selecting a hash function ***H: K L*** are as follows. First of all, the function ***H*** should be very easy and quick to compute. Secondly the function ***H*** should, as far as possible, uniformly distribute the hash addresses throughout the set ***L*** so that there are a minimum number of collisions. Naturally, there is no guarantee that the second condition can be completely fulfilled without actually knowing beforehand the keys and addresses. However, certain general techniques do help. One technique is to —chop‖ a key ***k*** into pieces and combine the pieces in some way to form the hash address **H(k)**. (The

term —hashing|| comes from this technique of —chopping|| a key into pieces.)

We next illustrate some popular hash functions. We emphasize that each of these hash functions can be easily and quickly evaluated by the computer.

a) Division method

Choose a number larger than the number n of keys in K. (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions.) The hash functions H is defined by $H(k)=k \pmod m$ or $H(k)=k \pmod m+1$

Here $k \pmod m$ denotes the remainder when is divided by The second formula is used when we want the hash addresses to range from 1to rather than from 0 to m-1.

b) Midsquare method

The key k is squared. Then the hash function H is defined by

$$H(k)=l$$

Where l is obtained by deleting digits from both ends of $k2$. We emphasize that the same positions of 2must be used for all of the keys.

c) Folding method

The key k is partitioned into a number of parts, $k1 \ldots . , kr,$ where each part, except possibly the last, has the same number of digits as the required address. Then the parts are added together, ignoring the last carry. That is,

$$H(k)=k+k+ \ldots\ldots +kr$$

Where the leading-digit carries, if any, are ignored. Sometimes, for extra —milling||, the even-numbered parts, $k2,k4,\ldots\ldots ,$ are each reversed before the addition.

Example

Consider the company in the above Example, each of whose 68 employees is assigned a unique 4-digit employee number. Suppose L consists of 100 two-digit addresses: 00, 01, 02, ..., 99. We apply the above hash functions to each of the following employee

numbers:

3205, 7148, 2345

1. Division method

Choose a prime number *m* close to 99, such as *m*=97. Then *H*(3205)=4, *H*(7148)=67, *H*(2345)=17

That is, dividing 3205 by 97 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, and dividing 2345 by 97 gives a remainder of 17. In the case that the memory addresses begin with 01 rather than 00, we choose that the function *H*(*k*)=*k*(mod *m*)+1 to obtain:

H(3205)=4+1=5, *H*(7148)=67+1=68, *H*(2345)=17+1=18

2. Midsquare method

The following calculations are performed:

k: 3205 7148

2345

k2: 10 272 025 51 093 904 5

499 025

H(*k*): 72 93

99

Observe that the fourth and fifth digits, counting from the right, are chosen for the hash address.

3. Folding method

Chopping the key k into two parts and adding yields the following hash addresses:

H(3205)=32+05=37, *H*(7148)=71+48=19, *H*(2345)=23+45=68

Observe that the leading digit 1 in *H*(7148) is ignored. Alternatively, one may want to reverse the second part before adding, thus producing the following hash addresses:

H(3205)=32+50=82, *H*(7148)=71+84+55, *H*(2345)=23+54=77

2. Collision Resolution

Suppose we want to add a new record *R* with key k to our file *F*, but suppose the memory location address *H*(*k*) is already occupied. This situation is

called ***collision***. This subsection discusses two general ways of resolving collisions. The particular procedure that one chooses depends on many factors. One important factor is the ratio of the number *n* of keys in *K* (which is the number of records in *F*) to the number m of hash addresses in L. this ratio,

$\lambda = n/m$, is called the *load factor*.

First we show that collisions are almost impossible to avoid. Specifically, suppose a student class has 24 students and suppose the table has space for 365 records. One random hash function is to choose the student's birthday as the hash address. Although the load factor $\lambda = 24/365 = 7\%$ is very small, it can be shown that there is a better than fifty-fifty chance that two of the students have the same birthday.

The efficiency of a hash function with a collision resolution procedure is measured by the average number of *probes* (key comparisions) needed to find the location of the record with a given key *k*. The efficiency depends mainly on the load factor λ. Specifically, we are interested in the following two quantities:

S(λ)= average number of probes for a successful search.

U(λ)= average number of probes for a unsuccessful search.

These quantities will be discussed for our collision procedures.

Open Addressing: Linear Probing and Modifications

R k Suppose that a new record with a key is to be added to the memory table *T*, but that the memory location with hash address $H(k)=h$ is already filled. One natural way to resolve the collision is to assign R to the first available location following *T*[*h*]. (We assume that the table t with m locations is circular, so that *T[*1] comes after T[*m*].) Accordingly, with such a collision procedure, we will search for the record R in the table T by linearly searching the locations *T*[*h*], *T*[*h*+1], *T*[*h*+2],.....until finding R or meeting an empty location, which indicates an unsuccessful search.

The above collision resolution is called linear probing. The average

numbers of probes for a successful search and for an unsuccessful search are known to be the following respective quantities:

$$s(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$ and

$$U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

(Here λ n/m is the load factor.)

Example

Suppose the table *T* has 11 memory locations, *T*[1], *T*[2], , *T*[11], and suppose the file *F* consists of 8 records, A, B, C, D, E, X, Y, and Z, with the following hash addresses:

Record: A, B, C, D, E, X, Y, Z

H(*k*): 4, 8, 2, 11, 4, 11, 5, 1

Suppose the 8 records are entered into the table *T* in the above order. Then the file *F* will appear in memory as follows:

Table *T*: X, C, Z, A, E, Y, _, B, _, _, D

Address: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

Although Y is the only record with hash address *H*(*k*)=5, the record is not assigned

to *T*[5], since *T*[5] has already been filled by *E* because of a previous collision at *T*[4]. Similarly, *Z* does not appear in *T*[1].

The average number S of probes for a successful search follows:

$$S = \frac{1+1+1+1+2+2+2+3}{8} = \frac{13}{8} = 1.6$$

The average number U of probes for a unsuccessful search follows:

$$U = \frac{7+6+5+4+3+2+1+2+1+1+8}{11} = \frac{40}{11} = 3.6$$

The first sum adds the number of probes to find each of the 8 records, and the second sum adds the number of probes to find an empty location for each of the 11 locations.

One main disadvantage of linear probing is that records tend to cluster, that is,

appear next to one another, when the load factor is greater than 50 percent. Such a clustering substantially increases the average search time for a record. Two techniques to minimize clustering are as follows:

Quadratic probing

Suppose a record R with key k has the hash address $H(k)=h$. Then, instead of searching the locations with addresses h, $h+1$, $h+2$,....., we linearly search the locations with addresses

h, $h+1$, $h+4$, $h+9$, h+16,.....h+i^2,.....

If the number m of locations in the table T is a prime number, then the above sequence will access half of the locations in T.

Double hashing

Here a second hash function H' is used for resolving a collision, as follows. Suppose a record R with key k has the hash addresses $H(k)=h$ and H'$(k)=h' \neq m$. Then we linearly search the locations with addresses

h, $h+h'$, $h+2h'$, $h+3h'$,....

If m is a prime number, then the above sequence will access all the locations in the table T.

Remark: One major disadvantage in any type of open addressing procedure is in the implementation of deletion. Specifically, suppose a record R is deleted from the location $T[r]$. Afterwards, suppose we meet $T[r]$ while searching for another record R'. This does not necessarily mean that the search is unsuccessful. Thus, when deleting the record R, we must label the location $T[r]$ to indicate that it previously did contain a record. Accordingly, open addressing may seldom be used

when a file F is constantly changing.

REVIEW QUESTIONS

1. What is the principle of bubble sort?
2. Differentiate between linear search and binary search?
3. What is searching and what are the methods of searching?
4. Why do we need files?
5. What do you understand by the term file organization?

6. Briefly summarize the different file organizations that are widely used today?

7. Consider the following 4-digit employee numbers

8. 9614, 5882, 6713, 4409, 1825

9. Find the 2-digit hash address of each number using

(a) the division method, with m=97;

(b) the mid square method;

(c) the folding method without reversing; and

(d)the folding method with reversing.

1. Consider the data. Suppose the 8 records are entered into the table T in the reverse order Z, Y, X, E, D, C, B, A.

(a) Show how the file F appears in memory.

(b) Find the average number S of probes for a successful search and the average number U of probes for an unsuccessful search.

2.What is collision resolution?

3.Define Open addressing?

4.Name two techniques to minimize clustering?

5.What do you mean by hashing?

5 Marks

6.Name the methods to evaluate the hash functions?

1.Explain quick sort and write the algorithm.

2.Write an algorithm for binary search.

3.Write an algorithm to implement bubble sort and mention its time complexity.

7 MARKS

1.Write an algorithm for merging and mention its complexity.

2.What is linear search? Write an algorithm and mention its complexity.