**Data structures**

* 1. **Introduction**

A Computer is a machine that manipulates information. The meaningful and processed data is called information. The way in which data is processed depends on the structure i.e. representation of data. The study of different structures for representation of data and the algorithms that operate on these structure constitute what we call study of data structures.

Knowledge of data structures is required for people who design and develop computer programs of any kind: Systems software or application software. As you know already, the ***data*** means a collection of facts, concepts or instructions in a formalized manner suitable for communication or processing. Processed data is called as ***Information***. A ***data structure*** is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. ***Algorithm****s*, on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting. The subject of data structures and algorithms is concerned with the Coding phase. The use of data structures and algorithms is to store and manipulate data by the programmers.

* 1. **Importance of Data Structures**

With ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Worse yet, as tasks become more complex, they become less like our everyday experience. Today’s computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term “data structure” to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring. Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. Thus,

it is possible to perform all necessary operations on any data structure. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days.

A solution is said to be efficient if it solves the problem within the required resource constraints. Examples of resource constraints include the total space available to store the data—possibly divided into separate main memory and disk space constraints — and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The cost of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints. It should go without saying that people write programs to solve problems. However, it is crucial to keep this truism in mind when selecting a data structure to solve a particular problem. Only by first analyzing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job. Poor program designers ignore this analysis step and apply a data structure that they are familiar with but which is inappropriate to the problem. The result is typically a slow program. Conversely, there is no sense in adopting a complex representation to “improve” a program that can meet its performance goals when implemented using a simpler design.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.

2. Quantify the resource constraints for each operation.

3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a datacentered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation. Resource constraints on certain key operations, such as search, inserting data

records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure:

• Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically require only simpler data structures to get an efficient implementation than do dynamic applications.

• Can data items be deleted? If so, this will probably make the implementation more complicated.

• Are all data items processed in some well-defined order, or is search for specific data items allowed? “Random access” search generally requires more complex data structures.

* 1. **Types of Data Structure**

The data structures are classified as either linear or nonlinear. A data structure is said to be linear if its elements form a sequence, or, in other words, a *linear list otherwise it is called non-linear*. Examples to non-linear structures are trees and graphs.

* + 1. **Linear Data Structure**

There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called ***arrays***. The other ways is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called ***linked lists***. In other words data is arranged in linear order. The data values are stored in sequence. Array, linked list, stack and queue are the example of linear data structure.

**Array:** The simplest one dimensional representation of data is called linear array. Linear array means a sequence of similar finite numbers referenced by any one of the following notations

Subscript notation: a1, a2, a3 ………..an

Parenthesis notation: A (1), A (2), A(3),……………A(N)

Bracket notation: A[1], A[2], A[3],……………A[N]

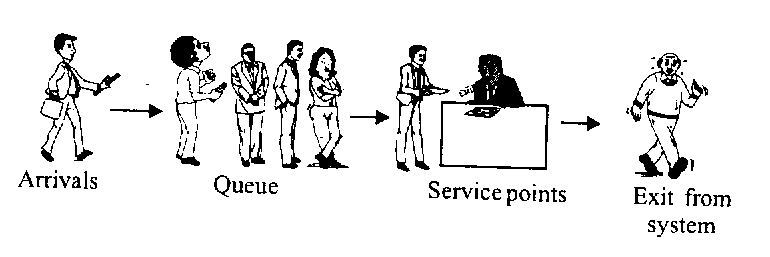
Generally bracket notations are in use. Linear array is also called one dimensional structure as only one subscript is required to store the data values in the system. When more than one subscript is used, the structure is called multi dimensional array.

**Linked List** data is stored in sequence in the system. List contains two fields for each data elements. One field is for value or information and other is for pointer that points to the next value in the list.

Linked list can be linear list or it may be circular list. Linear list contains linear order of elements where order is given by means of pointers. Circular list is differing in the way that the last node is connected back to the first node forming a circle.   
 **Stack:** stack is a linear ordered list of elements where values are inserted and deleted from one end called *top* of the list. Due to this reason stack is also called a Last-In-First-Out (LIFO) structure system. For example stack of dishes on a spring system. New dishes can be inserted from the top of the stack also dishes can be ejected from top of the stack. Clear concept is last element inserted will be the first to be deleted.

****

**Queue:** Queue is a linear ordered list of elements where values are inserted from one end called *rear* of the list and all the deletion will takes from the other end called the *front* of the list. Due to this reason Queue is also called a First-In-First-Out (FIFO) structure system. For example, a waiting line of people waiting for service. Clear concern is that first element inserted will be the first element to be deleted.

****

We may perform the following operations on any structure, whether it is an array or a linked list.

(a) Traversal: Processing each element in the list.

(b) Search: Finding the location of the element with a given value or the record with a given key.

(c) Insertion: Adding a new element to the list.

(d) Deletion: Removing an element from the list.

(e) Sorting: Arranging the elements in some type of order.

(f) Merging: Combining two lists into a single list.

* + 1. **Non Linear Data Structure**

The data elements are not arranged in sequential order. The relationship between data values are not in sequence, instead they are grouped in some non linear fashion. Tree and graph are the examples of non linear data structures.

**Tree**: Tree is a non- linear arrangement of data elements where data elements are related or arranged in a hierarchical relationship. This structure is represented as a rooted tree graph, or simply, a tree. Figure shows the relationship.

**Graph:** sometimes the data elements have the relationship which is not necessarily hierarchical in nature. For example, airline flies only between the cities connected by lines. The data structure that shows this type of relationship is called *Graph.* Figure clearly depicts the concept.

* 1. **Complexity- Time Space Trade Off**

An algorithm is a well-defined list of steps for solving a particular problem. One major purpose of this text is to develop efficient algorithms for the processing of our data. The time and space it uses are two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size. Each of our algorithms will involve a particular data structure. Accordingly, we may not always be able to use the most efficient algorithm, since the choice of data structure depends on many things, including the type of data and the frequency with which various data operations are applied. Sometimes the choice of data structure involves a time-space trade-off, by increasing the amount of space for storing the data. One may be able to reduce the time needed for processing the data or, vice versa. We illustrate these ideas with two examples.

**Summary**

Data storing and handling is the main objective of a computer system. Data is stored in the computer memory by means of bits. Data can be of different types. According to their nature various data types are derived and to manipulate the data efficiently different data structures are derived. Data manipulation or operation on data is depended on the specific data structure. the operation technique can be described by a set of instructions which is known as algorithm. Basically algorithm and data structure are the tools by which operation is done.

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

* The major operations associated with any data structure are traversing, searching, inserting and deleting.
* Data abstraction when supported as a data type in a language is called abstract data type.
* Algorithms must satisfy the criteria: input, output, definiteness, finiteness and effectiveness.

**What is data Structure?** A ***data structure*** is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

**Why data structure is needed?** Using the proper data structure can make the difference between a program running in a few seconds and one requiring many days. By storing the data in a proper structure the more efficient solutions can be found for solving the problems in time and within the resource constraints.

**Types of Data structure available**

* Arrays
* Stack
* Linked Lists
* Queue
* Tree
* Graph

**What is Algorithm?**

An algorithm is a well-defined list of steps for solving a particular problem.

**What is complexity of algorithm?**

The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size.

**Important Terms**

* **Data: A** piece of meaningful information.
* **Data type:** Category having unique set of logical properties imposed on the data.
* **Data structure:** Collection of data components that are constructed in a regular and characteristic way.
* **Array:** It is an ordered set which contains a fixed number of objects.
* **Linked list:** It is an ordered set consisting of variable number of elements to which insertions, deletions and other operations can be performed.
* **Data Abstraction:** It is a tool that allows each data structure to be developed in relative isolation from the rest of the solution.
* **Complexity:** interconnections of measuring units.
* **Algorithm: A** well-defined list of steps for solving a particular problem.
* **Flow chart:** Diagrammatic representation of an algorithm.
* **Data item:** A single unit of values.
* **Group items:** Data items that are divided into sub items.
* **Pointer:** A reference to a data structure.
* **Information:** if we organize data or process data so that it reflects some meaning then this meaningful or processed data is called information.
  1. **Review Questions**

**Q1.** What is data? Is raw data useful? What can we do to make it useful?

**Q2.** State the difference between data and information.

**Q3.** What is data structure, and why do we need it?

**Q4.** Explain different data types.

**Q5.** Explain abstract data type and its significance.

**Q6.** What are the different factors that should be attached to any algorithm?

**Q7.** How can we compare two algorithms?

**Q8.** Define and explain complexity of any algorithm (program) in terms of space required and time needed.

**Unit 1**

**This Chapter covers the following topics of the syllabus**

* **Problem solving concept top down and bottom up design, structured programming**
* **Concept of data types, variables and constants**
* **Concept of pointer variables and constants**

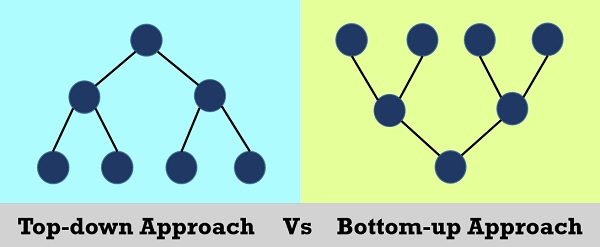
1. **Problem solving concept top down and bottom up design**

A computer is a very powerful machine capable of performing a variety of different tasks, yet it has no intelligence or thinking power. The intelligence Quotient (I.Q) of a computer is zero. A computer performs many tasks exactly in the same manner as it is told to do. This places responsibility on the user to instruct the computer in a correct manner, so that the machine is able to perform the required job in a proper way. A wrong instruction may sometimes prove harmful. In order to instruct a computer correctly, the user must have clear understanding of the problem to be solved. A part from this he should be able to develop a method, in the form of series of sequential steps, to solve it. Once the problem is well-defined and a method of solving it is developed, then instructing the computer to solve the problem becomes relatively easier task.

**1.1 There are two basic approaches for solving any problems. these are:**

1. **Top-down approach**
2. **Bottom-up approach**

The ideas behind top-down approach is, a bigger problem is divided into some smaller sub-problems called modules, these modules are then solved individually and then integrated together to get the complete solution to the problem. In bottom-up approach on the other hand, the process starts with elementary modules and then combining together to get the desired result.



**Top-Down Approach**

The basic idea in top-down approach is to break a complex algorithm or a problem into smaller segments called modules, this process is also called as *modularization.*The modules are further decomposed until there is no space left for breaking the modules. The top-down way of solving a program is step-by-step process of breaking down the problem into chunks for organizing and solving the whole problem. The C- programming language uses the top-down approach of solving a problem in which the flow of control is in the downward direction.

**Bottom-Up Approach**

As the name suggests, this method of solving a problem works exactly opposite of how the top-down approach works. In this approach we start working from the most basic level of problem solving and moving up in conjugation of several parts of the solution to achieve required results. The most fundamental units, modules and sub-modules are designed and solved individually, these units are then integrated together to get a more concrete base to problem solving. This bottom-up approach works in different phases or layers. Each module designed is tested at fundamental level that means unit testing is done before the integration of the individual modules to get solution.

**1.2 Difference between Top-Down and Bottom –Up Approach**

|  |  |
| --- | --- |
| **Top-Down Approach** | **Bottom-Up Approach** |
| Divides a problem into smaller units and then solve it. | Starts from solving small modules and adding them up together. |
| This approach contains redundant nformation. | Redundancy can easily be eliminated. |
| A well-established communication is not required. | Communication among steps is mandatory. |
| The individual modules are thoroughly analyzed. | Works on the concept of data-hiding and encapsulation. |
| Structured programming languages such as C uses top-down approach. | OOP languages like C++ and Java, etc. uses bottom-up mechanism. |
| Relation among modules is not always required. | The modules must be related for better communication and work flow. |
| Primarily used in code implementation, test case generation, debugging and module documentation. | Finds use primarily in testing. |

**2. Structured Programming**

**Structured programming defined as**

A technique for organizing and [coding](https://www.its.bldrdoc.gov/fs-1037/dir-007/_1049.htm) [computer](https://www.its.bldrdoc.gov/fs-1037/dir-008/_1184.htm) programs in which a hierarchy of modules is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used: sequential, test, and iteration. The definition highlights following major characteristics of structured programming:

* Hierarchical structure of models that is either top-down or bottom-up
* Program have single entry and single exit
* control is passed in only downward direction i.e the logical structure of program is similar to physical structure of programs.
* Avoid goto statements which causes unconditional branches to any position (Up or down) in the program
* use only sequential, iteration and loop control.

**Structured programming** is a [programming approach](https://en.wikipedia.org/wiki/Programming_paradigm) aimed at improving the clarity, quality, and development time of a [computer program](https://en.wikipedia.org/wiki/Computer_program) by making maximum use of the structured control flow constructs of selection ([if/then/else](https://en.wikipedia.org/wiki/Conditional_(computer_programming))) and repetition (while and [for](https://en.wikipedia.org/wiki/For_loop)), [block structures](https://en.wikipedia.org/wiki/Block_(programming)), and [subroutines](https://en.wikipedia.org/wiki/Subroutines) and no use of goto statements.

**1.3 Concept of data types, variables and constants**

A variable, according to its name, is something that allows its value to vary. In programming, we use variables to store data while we run our programs. and a constant is one whose value is not allowed to change. In programming, we use constants to store information that will never be going to change. For example The number of seconds in one hour is a constant

**Constant**

A constant consists of two things: a name, and a value. The name should clearly show what the constant is all about, and be self-explanatory. The value is the actual value of the constant. constants are commonly named with uppercase letters. This allows us to clearly identify them. Constants are very useful when we want to set a value once, and then repeat it in many locations inside our program. By declaring a constant once and then re-using it, we can easily change the value of the constant at one place and it is done at every place where it is used.

**Variables**

Variables are similar to constants, but the main difference is that while a constant can not change once you have assigned it a value, a variable can, and oftentimes does. When we declare a variable, we simply inform the computer of our wish to use the variable in our program, so that the computer can allocate memory for it. Declaration can be thought of as the same as initialization, but without giving the variable a value. Sometimes we may prefer to not give our variables a value immediately, and in some languages, we are actually forced to declare all our variables in the beginning of the code, regardless of where in the code we want to use them. Let us assume that you want to write a program that copy the act of rolling of a dice 10 times, and at the end sums up all the rolls and presents that number on the screen. In this case, we would need one variable to keep track of the sum, and one to keep track of how many times we have rolled the dice.

When we make use of programs and sub-programs we have two types of variables:

* **Local Variables**
* **Global Variables**

Local variables are the variables that are declared inside a sub-program or module and they are used inside that module only. Their scope is limited and they can not be used in outer module or main program. On the other hand global variables have a larger scope and they can be accessed and used in all the modules and main program.

**Data types**

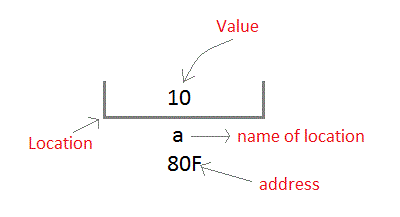
A data type, in programming, is a something that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. A string, for example, is a data type that is used to classify text and an integer is a data type used to classify whole numbers. In computer science and computer programming, a data type or simply type is a characteristic of data which tells the compiler or interpreter how the programmer wants to use the data. Most programming languages support common data types of real, integer and boolean. Some programming languages require the programmer to define the data type of a variable before assigning it a value. Other languages can automatically assign a variable's data type when the initial data is entered into the variable.

**Concept of pointer variables and constants**

A Pointer is a variable which holds the address of another variable of same data type. Pointers are used to access memory and manipulate the address. Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the pointer. A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

Let us assume that system has allocated memory location 80F for a variable a.

int a = 10;



We can access the value 10 either by using the variable name a or by using its address 80F.

The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**. A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



**Benefits of using pointers**

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

**Unit 2 (Arrays)**

The objective of this chapter is to present the most commonly used data structure Array. The various ways of representing the arrays in memory are discussed and the different operations that can be performed on arrays are described in the form of algorithms. The Sequential and binary search method are explained and compared at the end of the chapter.

**Introduction**

Data structures are of two types Linear and non- linear. In Linear data structure, data elements are arranged in a linear order i.e. in sequence. Array, linked list, stack and queue are the examples of linear data structure whereas tree and graph are non- linear data structures. In this chapter our concern is for array where elements are linearly related to each other having sequential memory locations. Operations performed on the any of the linear structure are as following:

* + Traversal: visiting each element in the data structure to process it.
  + Search: traversal through the data structure to find the location of a given data element.
  + Insertion: addition of new element in the data structure.
  + Deletion: Removing an element from the data structure.
  + Sorting: Arranging elements of the data structure in some type of order.
  + Merging: combining two similar data structures into one.

**Linear Array**

A data structure is said to be linear if its elements form a sequence or a linear list. For example months of a year and values in a card deck. Linear array is a collection of elements having sequential memory locations. A ***linear array*** is a list of a finite number *n* of inhomogeneous data elements (i.e., data elements of the same type) such that:

1. The elements of the array are referenced respectively by an index set consisting of *n*

consecutive numbers.

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

The number *n* of elements is called the *length* or *size* of the array. If not explicitly stated, we may assume the index set consists of the integers 1, 2, 3, ….., *n.* An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. The number n of elements is called the length or size of the array. If not explicitly stated, the length of the array can be calculated from the index set by the formula

Length= UB-LB+1

Where UB is the Upper Bound also called largest index And LB is the Lower Bound also called smallest index.The simplest form of an array is one dimensional array or vector. We already know, the various elements of an array are distinguished by giving each piece of data separate index or subscript. The subscript of an element designates its position in array’s ordering. The elements of an array A may be denoted by any one of the following notation.

Subscript notation: A1, A2, A3 ……………….An

Parenthesis notation: A(1), A(2), A(3)………..,A(N)

Bracket notation: A[1], A[2], A[3],………..A[N]

Let us consider the following *example*

1. Array DATA be a 6-element linear array of integers such that DATA[1]=247, DATA[2]=56, DATA[3]=429, DATA[4]=135, DATA[5]=87, DATA[6]=156.

|  |
| --- |
| 247 |
| 56 |
| 429 |
| 135 |
| 87 |
| 156 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 247 | 56 | *429* | *135* | *87* | *156* |

1. **(b)**

Let we consider the other *example*

b) An automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 through 1984. Rather than beginning the index set with 1, it is more useful to begin the index set with 1932, so we may learn that,

AUTO[K] = number of automobiles sold in the year K

Then, LB = 1932 is the lower bound and UB=1984 is the upper bound of AUTO. When we use the equation (1.4a), we can find out the length of this array,

Length = UB – LB + 1 = 1984 + 1 = 55.

From this equation, we may conclude that AUTO contains 55 elements and its index set consists of all integers from 1932 through 1984.

**Representation of Linear Arrays in memory**

Consider LA be a linear array in the memory of the computer. As we know that the memory of the computer is simply a sequence of addressed location as pictured in figureas given below.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

1001

1002

1003

1004

1005

.

.

.

Figure: Computer memory

Let we use the following notation when calculate the address of any element in linear arrays in memory, LOC(LA[K]) = address of the element LA[K] of the array LA. As previously noted, the elements LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of ever element of LA, but needs to keep track only of the address of the first element of LA, which is denoted by *Base*(LA) and called the *base address* of LA. Using this address *Base*(LA), the computer calculates the address of any element of LA by the following formula:

LOC(LA[K]) = *Base*(LA) + *w* (K- lower bound) (3.1)

where *w* is the number of words per memory cell for the array LA. Let we observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

Consider the previous *example* array AUTO, which records the number of automobiles sold each year from 1932 through 1984. Array AUTO appears in memory is pictured in Figure Assume, *Base*(AUTO) = 200 and *w* = 4 words per memory cell for AUTO. Then the base addresses of following arrays are,

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204, LOC(AUTO[1934]) = 208, …….

Let we find the address of the array element for the year K = 1965. It can be obtained by using Equation (3.1):

LOC(AUTO[1965]) = *Base*(AUTO) + w(1965 – lower bound) = 200 + 4(1965-1932) = 332

Again, we emphasize that the contents of this element can be obtained without scanning any other element in array AUTO.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| **.**  **.**  **.**  **.**  **.** |

200

201

202 AUTO[1932]

203

204

205

206 AUTO[1933]

207

208

.

.

.

Figure: Array represented in memory

**Operations on Arrays**

Following operations can be performed on the elements of array.

**Traversing**

Traversing is the act of visiting each element of array to process it. In other terms, traversing is accessing each record exactly once so that certain items in the record may be processed. Consider let A be a collection of data elements stored in the memory of the computer. Suppose we want to either print the contents of each element of A or to count the number of elements of A with a given property. This can be accomplished by *traversing* A, that

is, by accessing and processing (frequently called *visiting*) each element of A exactly once. The following ***algorithm is used to traversing a linear array LA***.

**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]

1. Exit.

Detail of the algorithm is as follows: start the counter named K with lower bound value. Visit every data element by incrementing the counter until the counter K counts up to upper bound value. We also state an alternative form of the algorithm which uses a repeat-for loop instead of repeat-while loop.

**Algorithm .** (Traversing a Linear Array) This algorithm traverses linear array LA with lower bound LB and upper bound UB.

1. Repeat for K= LB to UB.

Apply PROCESS to LA [K].

[End of loop].

1. Exit.

Again consider the previous one [*example 1.3(b)*], array AUTO, which records the number of automobiles sold each year from 1932 through 1984. Each of the following algorithms carry out the given operation involves traversing AUTO.

1. Find the number NUM of years during which more than 300 automobiles were sold.

1. [Initialization Step] Set NUM := 0.

2. Repeat for K = 1932 to 1984:

If AUTO[K] > 300, then: Set NUM := NUM + 1

[End of loop]

3. Return

(b) Print each year and the number of automobiles sold in that year.

1. Repeat for K = 1932 to 1984:

Write: K, AUTO[K].

[End loop]

2. Return.

**Insertion**

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 of the elements from A. Let we discuss the inserting and

deleting an element when A is a linear array. Inserting an element at the “end” of a 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 locations to accommodate the new element and keep the order of the other elements. Similarly, deleing an element at the “end” of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to “fill up” the array. Since leaner arrays are usually pictured extending downward, as given below in figure below, the term “*downward*” refers to locations with larger subscripts, and the term “*upward*” refers to location with smaller subscripts.

|  |
| --- |
| Red |
| Green |
| Blue |
| Black |
| White |
|  |

1

2

3

4

5

Consider TEST has been declared as a 5-element array but data have been recorded only for TEST [1], TEST [2], and TEST [3]. If X is the value to the next element, then we may simply assign, TEST [4] := X to add X to the Linear Array. Similarly, if Y is the value of the subsequent element, then we may assign, TEST [5] := Y to add Y to the Linear Array. Now, we may conclude that we can not add any new element to this Linear Array due to the reach of upper bound.

**Example:** Suppose NAME is an 8-element linear array, and suppose five names are in the array, as in shown in following *Figure (a)*. Observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. If, *Ford* is adding to the array, then, *Johnson, Smith* and *Wagoner* must each be move downward one location, as given in *Figure (b)*. Next if we add *Taylor* to this array; then *Wagner* must be move, as in *Figure (c)*. Last, when we remove *Davis* from the array, then, the five names *Ford, Johnson, Smith*, *Taylor* and *Wagner* must each be move upward one location, as in *Figure (d)*. We may observe, clearly that such movement of data would be very expensive if thousands of names are in the array.

**\Algorithm .**  (Inserting into a 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. This algorithm inserts an element ITEM into 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.].

1. [Insert element.] Set LA[K] :=ITEM.
2. [Reset N] Set N := N+1.
3. Exit

Details of the algorithm are as follows: Initialize the counter J with N. Repeat for until the last element. Firstly make the location for the element by setting the Jth position to (J+1)th position. Then inserts the item and increase the no. of elements from N to N+1.

**Deletion**

The following ***algorithm deletes the Kth element*** from a linear array LA and assigns it to a variable ITEM i.e. DELETE{LA.N,K,ITEM).

**Algorithm 3.4:** (Deleting into a Linear Array) DELETE (LA, N, K, ITEM)

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

1. Set ITEM: = LA [K].
2. Repeat for J =K to N-1...
3. [Move J+1st element upward.] Set LA [J]:=LA [J+1].

[End of loop.].

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

Detail of the algorithm is as follows: select the item to be deleted. Delete the item and make free the space of that particular item by shifting the position of J+1th element to Jth position. And decrease the number of elements as N-1 from N.

**Searching**

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. Searching refers to the operation of finding the location LOC of ITEM in DATA or printing some message that ITEM does not appear there. The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise. Frequently one may want to add the element ITEM to DATA after an unsuccessful search for ITEM in DATA. One then uses a search and insertion algorithm, rather than simply a search algorithm. There are many different searching algorithms. The algorithm that one chooses generally depends on the way the information in DATA is organized. The complexity of searching algorithms is measured in terms of the number of comparisons required to find ITEM in DATA where DATA contains n elements. We shall show that linear search is a linear time algorithm, but that binary search is a much more efficient algorithm, proponional in time to log2 n. On the other hand, we also discuss the drawback of relying only on the binary search algorithm.

**Sequential Search or Linear Search**

Suppose DATA is a linear array with n elements. Given 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. That is, first we test whether DATA[1] = ITEM and then we test whether DATA[2] = ITEM, and so on. This method which traverses DATA sequentially to locate ITEM, is called linear search or sequential search, To simplify the matter we first assign ITEM to DATA[N + 1], the position following the last element of DATA. Then the outcome LOC = N + l where LOC denotes the location where ITEM first occurs in DATA signifies the search is unsuccessful. The purpose of this initial assignment is to avoid repeatedly testing whether or not we have reached the end of the array DATA. This way the search must eventually "succeed." A formal presentation of linear search is shown in Algorithm 3.5. Observe that Step 1 guarantees that the loop in Step 3 must terminate. Without Step 1 , the Repeat statement in Step 3 must be replaced by the following statement. Which involves two comparisons not one:

Repeat while LOC ≠N and DATA[LOC] ≤ ITEM:

On the other hand, in order to use Step 1, one must guarantee that there is an unused memory location at the end of the array DATA.

**Algorithm** : (Linear Search) LINEAR (DATA, N, ITEM, LOC)

Here DATA is a linear array with n elements and ITEM is a given item of information. The algorithm finds the location of ITEM in DATA or set LOC=0 if the search is unsuccessful.

1. [Insert ITEM at the end of the array] Set DATA[N+1]:= ITEM
2. [Initialize counter] Set LOC:= 1
3. [Search ITEM] Repeat While DATA[LOC] ≠ ITEM

Set LOC: = LOC+1

[End of Loop]

1. [Successful?] If LOC= N+1, then Set LOC:= 0.
2. Exit.

**Binary Search**

This is also called mid point searching technique. Suppose a list of elements with n elements. The list should be a sorted list. Get the item to be searched. Start searching in the list by getting the mid point of the list. Mid point implies the middle element of the list. Compare the element to be searched with the middle element. If the element is less than the middle element then start searching in the first half of the list otherwise if the element is greater than the middle element then start searching in second half of the list. Again subdivide the half list and get the middle element of the half list. Again compare the element if it is less than or greater than the middle element. Continue with this method until the element is not met that is to be searched. If the element is not met that means the element is not in the list the results in the unsuccessful operation.

**Algorithm**  (Binary Search) BINARY (DATA, LB, UB, ITEM. LOC)

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

1. [Initialize segment variables] Set BEG := LB, END := UB and

MID = INT (BEG + END)/2).

1. Repeat Steps 3 and 4 while BEG ≤ END and DATA [MID] ≠ ITEM.

3. If ITEM < DATA [MID], then:

Set END : = MID - l.

Else:

Set BEG : = MID + 1.

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

5. If DATA[MID] = ITEM. then:

Set LOC := MID.

Else:

Set LOC := NULL.

6. Exit.

**Multidimensional Arrays**

The linear arrays discussed so far are also called one dimensional arrays, since each element in the array is referenced by a single subscript. Most programming languages allow two- dimensional and three- dimensional arrays, ie., arrays where elements are referenced, respectively, by two and three subscripts. In fact, some programming languages allow the number of dimensions for an array to be high as 7. this section discusses these multidimensional arrays.

**Two-Dimentional arrays**

A two-dimensional *m* x *n* array A is a collection of m **.** n data elements such that each element is

specified by a pair of integers (such as J, K), called *subscripts*, with the following property that,

I < = J < = *m* and I <= K <= *n*

The element of A with first subscripts *j* and second subscript *k* will be denoted by

A J, K or A[J, K]

The following figure shows a 2-D array of order 3×4

columns 1 2 3 4

1 A[1,1] A[1,2] A[1,3] A[1,4]

Rows

2 A[2,1] A[2,2] A[2,3] A[2,4]

3 A[3,1] A[3,2] A[3,3] A[3,4]

**3.4 Representation of Multidimensional Arrays in Memory**

Let A be a two-dimensional m x n array. Although A is pictured as a rectangular array of elements with *m* rows and *n* columns, the array will be represented in memory by a block of *m.n* sequential memory locations. If they are being stored in sequence, then how are they sequenced? Is it that the elements are stored row wise or column wise? Again, it depends on the operating system. Specifically, the programming languages will store the array A in either,

* Column by column, called *column-major order*, or
* Row by row, called *row-major order*.

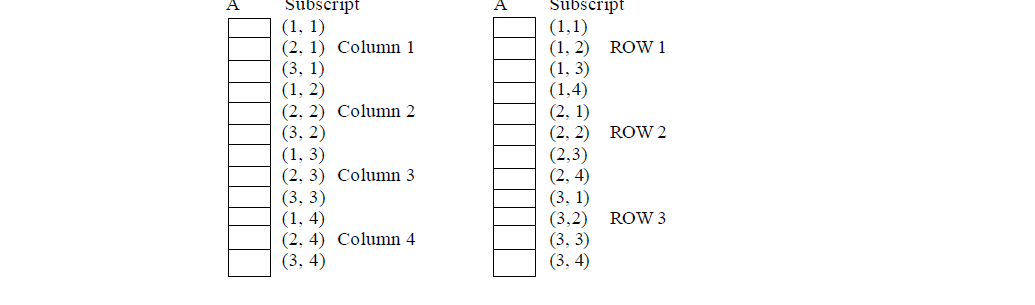
As with one dimentional arrays, the computer keeps track of *Base*(A) – the address of the first element A[1,1] of A and computes the address LOC(A[J, K]) of A[J, K]. The formula for *column and row major order* is,

LOC(A[J, K]) = Base(A) + w[M(K-1) + (J-1)]

The formula for *row major order* is,

LOC(A[J, K]) = Base(A) + w[N(J-1) + (K-1)]

Again, *w* denotes the number of words per memory location for array A. Note that the formulas are linear in J and K, and that we may find the address LOC(A[J, K]) in time independent of J and K. The following *figures (a) & (b)* shows these two ways when A is a two-dimensional 3 x 4 array.

****

**(a) Column-major order (b) Row-major order**

**Example :** Consider the example of 25 x 4 matrix array SCORE. Suppose Base(SCORE) = 200 and there are w = 4 words per memory cell. Further more let the programming language stores two-dimensional arrays using row-major order. Then the address of SCORE[12,3], the third test of the twelfth student, follows:

LOC(SCORE[12,3]) = 200 + 4[4(12 -1) + (3 -1)] = 200 + 4[46] = 284

**General Multidimensional Arrays**

General multidimensional arrays are defined analogously. More specifically, an n dimensional **m1 x m2 x ……x mn,** array B is a collection of **m1, m2…….. mn** data elements in which each element is specified by a list of n integers such as **K1, K2,…. Kn** called *subscripts*, with the property that

**1 <= K1 <= m1, 1 <= K2 <= m2, ……….. 1 <= Kn <= mn,**

The element of B with subscripts **K1, K2,…. Kn** will be denoted by **B K1, K2,…. Kn .** The array will be stored in memory in a sequence of memory locations. Specifically, the programming language will store the array B either in *row-major order* or *column-major order.*

**Summary**

Data structure which displays the relationship of Adjacency between elements is said to be “linear”. Length finding, traversing from left to right, retrieval of any element, storing any element, deleting any element are the main operations which can be performed on any linear data structure. Arrays are one of the Linear Data structures. Single dimension as well as multi-dimension arrays is represented in memory as one dimension arrays. Elements of any multidimensional array can be stored in two forms- row major and column major.

**Important Terms**

* **Data structure:**  A data Structure, which displays the relationship of adjacency between elements.
* **One-Dimensional Array:**  A list of finite number n of homogenous data elements i.e. data elements of the same type.
* **Two-Dimensional Array:** A collection of m.n data elements such that each element is specified by a pair of integers such as J, K called subscript, with property that, 1<=j<=m and 1<=k<=n.
* **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
* **Searching:** Finding the location of the record with a given key value.
* **Inserting:** Adding a new record to the structure.
* **Deleting:** Removing a record from the structure.
* **Sorting:** Arranging the records in some logical way.
* **Merging:** Combining the records in two different sorted files into a single sorted file.
  1. **Review Questions**

**Q1.** What are linear data structures and what are the limitations associated with linear data structures.

**Q2.** What is an ordered list?

**Q3.** What is the main drawback of linear array?

**Q4.** Explain how a two-dimensional array is represented in memory, using row major form.

**Q5.** Give the generalized formula to find the address of any element in an n- dimensional array.

**Q6.** Let A and B be two lower triangular matrices, each with n rows. The total number of elements in the lower triangles is n (n+1). Devise a scheme to represent both the triangles an array C[1….n][1….n+1]

**Q7.** Write an algorithm to perform matrix multiplication.

**Unit 3** (**Linked List)**

**Introduction**

A linked list is made up of a series of objects, called the nodes of the list. Besides being a nearly ubiquitous structure used in everything from operating systems to video games, it is also a building block with which many other data structures can be created. In a very general sense, the purpose of a linked list is to provide a consistent mechanism to store and access an arbitrary amount of data. As its name implies, it does this by linking the data together into a list. Array data is stored as a single contiguously allocated chunk of memory that is logically segmented. The data stored in the array is placed in one of these segments and referenced via its location, or index, in the array. This is a good way to store data. Most programming languages make it very easy to allocate arrays and operate on their contents. Contiguous data storage provides performance benefits (namely data locality), iterating over the data is simple, and the data can be accessed directly by index (random access) in constant time. There are times, however, when an array is not the ideal solution. What we need is a collection that allows us to add an arbitrary number of integer values and then enumerate over those integers in the order that they were added. The collection should not have a fixed maximum size and random access indexing is not necessary.

**Linked List/Singly Linked List**

Schematic diagram of a linked list with 6 nodes is shown in the figure below, Each node is pictured with two parts. The left part represents the information part of the node, which may contain an entire record of data items (e.g., NAME, ADDRESS,...). The right part represents the Next pointer field of the node, and there is an arrow drawn from it to the next node in the list. This follows the usual practice of drawing an arrow from a field to a node when the address of the node appears in the given field. The pointer of the last node contains special value, called the null pointer, which is any invalid address.

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including stacks, queues, associative arrays, and symbolic expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation. The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require scanning most or all of the list elements.

**Representation of Linked List in memory**

Let LIST be a linked list. Then LIST will be maintained in memory as follows. First of all, LIST requires two linear arrays-we will call them here INFO and LINK - such that INFO[K] and LINK[K] contain the information part and the next pointer field of a node of LIST respectively. START contains the location of the beginning of the list, and a next pointer sentinel-denoted by NULL-which indicates the end of the list. The following *examples* of linked lists indicate that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.

START = 9, so INFO[9] = N is the first character.

LINK[9] = 3, so INFO[3] = O is the second character.

LINK[3] = 6, so 1NFO[6] = (blank) is the third character.

LINK[6] = 11, so INFO[11] = E is the fourth character.

LINK[11] = 7, so INFO[7] = X is the fifth character.

LINK[7] = 10, so INFO[10] = I is the sixth character.

LINK[10] = 4, so INFO[4] = T is the seventh character.

LINK[4] = 0, so the NULL value, so the list has ended.

**Operations on Linked List**

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

**Traversing a Linked List**

Let LIST be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST. Suppose we want to traverse LIST in order to process each node exactly once. Our traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed. Accordingly, LINK[PTR] points to the next node to be processed. The assignment PTR := LINK[PTR] moves the pointer to the next node in the list.

The details of the algorithm are as follows. Initialize PTR. Then process INFO[PTR], the information at the first node. Update PTR by the assignment PTR:=LINK[PTR], and then process INFO[PTR], the information at the second node and so on until PTR=NULL, which signals the end of the list.

***Algorithm*** : (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR := START. [Initializes pointer PTR].

2. Repeat Steps 3 and 4 while PTR # NULL.

3. Apply PROCESS to INFO[PTR].

4. Set PTR:=LINK[PTR]. [PTR now points to the next node.]

[End of Step 2 loop.]

5. Exit.

**Examples of Traversing: Printing the elements of the Linked list and counting the number of elements in the linked list.**

**COUNT (INFO, LINK, START, NUM)**

1. Set NUM := 0. [Initializes counter.]

2. Set PTR := START. [Initializes pointer.]

3. Repeat Steps 4 and 5 while PTR # NULL.

4. Set NUM := NUM+1. [Increases NUM by 1]

5. Set PTR:LINK[PTR]. [Updates pointer.]

[End of Step 3 loop.]

6. Return.

**PRINT(INFO, LINK, START)**

1. Set PTR := START.

2. Repeat Steps 3 and 4 while PTR # NULL:

3. Write: INFO[PTR].

4. Set PTR := LINK[PTR]. [Updates pointer.]

[End of Step 2 loop.]

5. Return.

**Searching a Linked List**

Let LIST be a linked list in memory. We are given an ITEM of information. In this section we are going to discuss the two searching algorithms for finding the location LOC of the node where ITEM first appears in LIST. The first algorithm does not assume that the data in LIST are sorted. The second algorithm does assume that LIST is sorted. If ITEM is actually a key value and we are searching through a file for the record containing ITEM, then ITEM can appear only once in LIST.

**LIST is Unsorted**

The data in LIST are not necessarily sorted. Then one searches for ITEM in LIST by traversing through the list using a pointer variable PTR and comparing ITEM with the contents INFOR[PTR] of each node, one by one, of LIST. Before we update the pointer PTR by

PTR := LINK[PTR]

***Algorithm 4.2:*** 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.

1. Set PTR := START.

2. Repeat Step 3 while PTR # NULL:

3. If ITEM = INFO[PTR] then:

Set LOC := PTR and Exit.

Else:

Set PTR := LINK[PTR]. [PTR now points to the next node.]

[End of If structure.]

[End of Step 2 loop.]

4. Set LOC:=NULL. [Search is unsuccessful.]

5. Exit.

we require two tests. First we have to check whether we reached the end of the list; i.e. PTR = NULL , If not, then we check to see whether INFO[PTR] = ITEM

**LIST is Sorted**

The data in the LIST are sorted. Again we search for ITEM in LIST by traversing the list using a point variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Here we can stop once ITEM exceeds INFO[PTR].

***Algorithm 4.3:*** SRCHSL(INFO, LINK, START, ITEM, LOC)

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

1. Set PTR := START.

2. Repeat Step 3 while PTR#NULL:

3. If ITEM < INFO[PTR], then:

Set PTR := LINK[PTR]. [PTR now points to next node]

Else if ITEM = INFO[PTR], then:

Set LOC := PTR, and Exit. [Search is successful.]

Else:

Set LOC := NULL, and Exit. [ITEM now exceeds INFO[PTR]].

[End of If structure.]

[End of Step 2 loop.]

4. Set LOC := NULL.

5. Exit.

**\**

**Insertion into a Linked List**

Let LIST be a linked list with successive nodes A and B, as pictured in figure (a).Suppose a node N is to be inserted into the list between nodes A and B. The schematic diagram of such an insertion appears in figure (b). That is, node A now points to the new node N, and node N points to node B, to which A previously pointed.

Figure (a) Before Insertion



Figure (b) After Insertion

Let we discuss three insertion algorithms.

(a) The first one inserts a node at the beginning of the list.

(b) The second one inserts a node into after the node with a given location.

(c) The third one inserts a node into a sorted list.

In all the following algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START; AVAIL) and that the variable ITEM contains the new information to be added to the list. Since our insertion algorithms will use a node in the AVAIL list, all of the algorithms will include the following steps:

(a) Checking to see if space is available in the AVAIL list, If not, that is, if AVAIL=NULL, then

the algorithm will print the message OVERFLOW.

(b) Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this step can be implemented by the pair of assignments

NEW := AVAIL, AVAIL := LINK[AVAIL]

(c) Copying new information into the new node. in other words, INFO[NEW] := ITEM

**Insertion at the beginning of List**

***Algorithm 4.4 :*** INSFIRST(INFO, LINK, START, AVAIL,. ITEM)

This algorithm inserts ITEM as the first node in the list.

1. [OVERFLOW?]

If AVAIL = NULL, then: Write: OVERFLOW and Exit.

2. [Remove first node from AVAIL list].

Set NEW := AVAIL and AVAIL := LINK [AVAIL].

3. [Copy new data into new node.]

Set INFO[NEW] := ITEM.

4. [Make new node now to point the original first node.]

Set LINK[NEW] := START.

5. [Change START so it points to the new node.] Set START := NEW

6. Exit.

**Inserting after a Given Node**

Suppose we are given the value of LOC where either LOC is the location of a node A in a linked LIST or LOC=NULL. The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node:

Let N denote the new node (whose location is NEW). If LOC = NULL, then N is inserted as the first node in LIST .We let node N point to node B (which originally followed node A) by the assignment LINK[NEW] := LINK[LOC] and we let node A point to the new node N by the assignment LINK[LOC] := NEW

The algorithm is as follows.

***Algorithm 4.5 :*** INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)

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

1. [OVERFLOW?] If AVAIL=NULL, then: Write: OVERFLOW, and Exit.

2. [Remove first node from AVAIL list}

Set NEW := AVAIL and AVAIL := LINK[AVAIL].

3. [Copy new data into new node.] Set INFO[NEW] := ITEM.

4. If LOC = NULL, then: [Insert as first node.]

Set LINK[NEW] := START and START = NEW.

Else: [Insert after node with location LOC.]

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

[End of If structure.]

5. Exit.

**Inserting into a Sorted Linked List**

Suppose ITEM is to be inserted into a sorted linked LIST. The linked list is not sorted. The algorithm inserts the node into a Sorted Linked List. The ITEM must be inserted between nodes A and B so that INFO(A) < ITEM < INFO(B)

The following procedure finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM. Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable PREV. Thus SAVE and PTR are updated by the assignments SAVE := PTR and PTR := LINK[PTR]. The traversing stops as soon as ITEM < INFO[PTR]. Then PTR points to node B. so SAVE will contain the location of the node A.

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

This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM, or sets LOC=NULL.

1. [List empty?] If START = NULL, then: Set LOC := NULL, and Return.

2. [Special case?] If ITEM <INFO[START], then: Set LOC := NULL, and Return.

3. [Initializes pointers.] Set PREV := START and PTR := LINK[START].

4. Repeat Steps 5 and 6 while PTR # NULL.

5. If ITEM < INFO[PTR],then:

Set LOC := PREV and Return.

6. [Updates pointers] Set PREV := PTR and PTR := LINIC[PTR].

7. Set LOC := SAVE

8. Return.

***Algorithm 4.6 :*** INSSRT(INFOR, LINK, START, AVAIL, ITEM)

This algorithm inserts ITEM into a sorted linked list.

1. [find the location of the node preceding ITEM]

Call FINDA(INFO, LINK, START, ITEM, LOC).

2. [Use *Algorithm 4.5* to insert ITEM after the node with location LOC]

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

3. Exit.

**Deletion from a Linked List**

Let LIST be a linked list with a node N between nodes A and B, Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion appears in Figure below. The deletion occurs as soon as the next pointer field of node A points to node B. (Accordingly, when performing deletions, one must keep track of the address of the node which immediately precedes the node that is to be deleted.)



**Deletion Algorithms**

We discuss two deletion algorithms.

(a) The first one deletes a node following a given node.

(b) The second one deletes the node with a given ITEM of information.

All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START; AVAIL) and that the variable ITEM contains the new information to be added to the list. All of our deletion algorithms will return the memory space of the deleted node N to the beginning of the AVAIL list. Accordingly, all of our algorithms will include the following pair of assignments, where LOC is the deleted node N:

LINK[LOC] := AVAIL and then AVAIL := LOC

If START=NULL, then the algorithm will print the message UNDERFLOW.

**Deleting the Node Following a Given Node**

***Algorithm 4.7:*** DEL(INFO, LINK, START, AVAIL, LOC, LOCP)

This algorithm deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node, LOCP=NULL.

1. [Deletes first node.] If LOCP = NULL, then:

Set START:=LINK[START].

Else: [Deletes node N.]

Set LINK[LOCP] := LINK[LOC].

[End of If structure.]

2. [Add removed node to the AVAIL list.] Set LINK [LOC] := AVAIL and AVAIL := LOC.

3. Exit.

**Deleting the Node with a Given ITEM of Information**

Let LIST be a linked list in memory. Suppose we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM. (If ITEM is a key value, then only one node can contain ITEM.). First we give a *procedure* which finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N.

***Procedure:*** FINDB(INFO, LINK, START, ITEM, LOC, LOCP)

This procedure finds the location LOC of the first node N which contains ITEM and the location LOCP of the node preceding N. If ITEM does not appear in the list, then the procedure sets LOC=NULL; and if ITEM appears in the first node, then it sets LOCP=NULL.

1. [List empty?] If START = NULL, then:

Set LOC:=NULL and LOCP:=NULL, and Return.

[End of If structure.]

2. [ITEM in first node?] If INFO[START] = ITEM, then:

Set LOC:=START and LOCP=NULL, and Return.

[End of If structure.]

3. [Initializes pointers.] Set SAVE := START and PTR := LINK[START].

4. Repeat Steps 5 and 6 while PTR # NULL.

5. If INFO[PTR] = ITEM, then:

Set LOC := PTR and LOCP := SAVE, and Return.

[End of If structure.]

6. [Updates pointers.] Set SAVE := PTR and PTR := LINK[PTR].

[End of Step 4 loop.]

7. [Search unsuccessful.] Set LOC := NULL.

8. Return.

***Algorithm 4.8:*** DFLETE (INFO, LINK, START, AVAIL, ITEM)

This algorithm deletes from a linked list the first node N which contains the given ITEM of information.

1. [Use previously given *Procedure* to find the location of N and its preceding node.]

Call FINDB(INFO, LINK, START, ITEM, LOC, LOCP)

2. [Item not found?] If LOC = NULL, then:

Write: ITEM not in list, and Exit.

3. [Delete node.] If LOCP = NULL, then:

Set START := LINK[START]. [Deletes first node.]

Else:

Set LINK[LOCP] := LINK[LOC].

[End of If structure.]

4. [Return deleted node to the AVAIL list.] Set LINK[LOC] := AVAIL and AVAIL = LOC.

5. Exit.

**Doubly Linked List**

In computer science, a **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called *links*, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders. The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Let we discuss a *two-way list*, which can be traversed in two directions, either 1. in the usual forward direction from the beginning of the list to the end, or 2. in the backward direction from the end of the list to the beginning. Furthermore, given the location LOC of a node N in the list, one now has immediate access to both the next node and the preceding node in the list. This means, in particular, we may able to delete N from the list without traversing any part of the list.

A two-way list is a linear collection of data elements, called nodes, where each node N is divided into three parts:

1. An information field INFO which contains the data of N

2. A pointer field FORW which contains the location of the next node in the list

3. A pointer field BACK which contains the location of the preceding node in the list

The list also requires two list pointer variables: FIRST, which points to the first node in the list, and LAST, which points to the last node in the list. *Following figure* contains a schematic diagram of such a list. Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list.



Observe that, using the variable FIRST and the pointer field FORW, we can traverse a two-way list in the forward direction. On the other hand, using the variable LAST and the pointer field BACK, we can also traverse the list in the backward direction. Suppose LOCA and LOCB are the locations, of nodes A and B in a two-way list respectively. Then the way that the pointers FORW and BACK are defined gives us the following: Pointer property: FORW[LOCA] = LOCB if and only if BACK[LOCB] = LOCA In other words, the statement that node B follows node A is equivalent to the statement that node A precedes rode B.

**Applications of Linked Lists**

Some applications of linked list are as follows:

* For representing Polynomials: It means in addition/subtraction /multiplication of two polynomials. Example: p1=2x2+3x+7 and p2=3x3+5x+2 then p1+p2=3x3+2x2+8x+9
* In Dynamic Memory Management
* In allocation and releasing memory at runtime.
* In Symbol Tables in Balancing parenthesis
* Representing Sparse Matrix
* Linked Lists are very useful in dynamic memory allocation. These lists are used in operating systems. Insertion and deletion in linked lists are very useful.
* Complex data structures like tree and graphs are implemented using linked lists.
* Hash tables that use chaining to resolve hash collisions typically have one linked list per bucket for the elements in that bucket.
* Simple memory allocators use a free list of unused memory regions, basically a linked list with the list pointer inside the free memory itself
* In a FAT file system, the metadata of a large file is organized as a linked list of FAT entries.

**Summary**

There are various types of data structures. Some of them are static in nature, where the size of the data structure is fixed and some of them are dynamic, where it can grow or shrink according to the requirement. Array is static type of data structure and linked list is the dynamic type of data structure. contiguous memory allocation is required for an array, whereas non- contiguous memory allocation is possible for linked list. A node of a single linked keeps the location address of the next node. In the doubly linked list node points to the previous node and the next node both. A link list can be implemented in circular fashion, where the last node or tail points to the first node or head of the list. Node can be inserted and can be deleted at any position of the list. Traversing and searching for a specific value is also common operation in the linked list. A linked list uses the memory in better way than an array. The size of an array is fixed, so if there is a chance to overflow and underflow. In case of linked list if one node has to be added then memory space has been allocated according to the requirement and if one node has been deleted then the memory space gets freed. The free memory spaces are collected and make available for further use by the operating system. This procedure is known as garbage collection. If a procedure requires memory space but it is not available at that point of time then the operating system tries to arrange and allocate the memory, this technique is known as dynamic memory allocation. List structures allow us to store individual data elements. In Linked Lists these elements are interconnected by pointers. Storage is available for allocation on peripheral devices and in the main memory. It is managed either by means of a bit-table, table of contents file or by Linked Lists. Space is allocated either using the best fit or first fit algorithm. Free space management is done by garbage collection which relocates fragmented free space and compact it to get a contiguous chunk of free space.

**Important terms**

* **Pointer:** One kind of variable that contains the address of memory location.
* **Linked list:**  a linear data structure that have a relation between the data elements with the help of pointer or link
* **Insert :** Add a new node at a given point
* **Delete:**  Remove an existing node
* **Search:** Find a node with a given key.
* **Node:**  each of the individual structure in the list is known as node
* **Head:** the first node in the list is known as the head.
* **Tail:** the last node in the list is known as tail.
* **Single Linked List:**  one kind of linked list where each node points to its next node only.
* **Doubly Linked list:** one kind of linked list where each node points to its next node as well as to its previous node.
* **Circular Linked List:** one kind of linked list where the last node points to the first node of the same list.

**Review Questions**

**Q1.** How are the drawbacks of array representation overcome by the use of linked- list?

**Q2.** Write a C program to implement a linear linked list, write functions for inserting a new node at the beginning, at the end and at the middle of any linked list. Also write a function for deletion of any node from the linked list.

**Q3.** What are the demerits of a singly linked list?

**Q4.** What is circular linked list and how does it provide solution for the drawback of singly linked list?

**Q5.** Explain doubly linked list.

**Q6.** What is the difference between circular linked list and linear linked list?

**Q7.** How Linked list uses the memory more efficiently than array?

**Q8.** What are the uses of START and NULL in singly linked list?

**Q9.** Write a procedure to count the number of nodes in a given linked list.

**Q10.** Write a procedure to search a given number in a given linked list.

**Unit 4& 5 (Trees & Graphs)**

**Introduction**

A tree is a structure whose graphical representation looks like a family tree: It starts with a root at the top, and branches downward. Typical uses of trees are the representation of the class hierarchy, storing data for fast access, and translation of program code. A tree is a two-dimensional collection of objects called nodes. Three examples are a class hierarchy tree, a family tree, and a tree of student records. The nodes of a tree can be classified as follows:

* The node at the top of the tree is called the *root*. This is the node through which the tree can be accessed. A tree has exactly one root.
* A node that is not a root and has at least one child is an *internal* node. Every internal node has exactly one parent node but it may have any number of children, unless the definition of the tree specifies otherwise.
* A node that does not have any children is called a *leaf*. Like an internal node, a leaf has exactly one parent node.

An interesting and useful property of trees is that any node in the tree can be treated as the root of a new tree consisting of all the underlying nodes. Such a tree is called a *subtree* of the original tree. Trees are very important in computer applications and a variety of trees have been devised to provide the best possible performance for different uses. The differences between different kinds of trees are in the number of children that a node may have4, and the way in which the tree is managed (how nodes are added and deleted). The subject of trees is not trivial and we will restrict our presentation to the example of a simple binary tree. In a binary tree, each node may have at most two children.

**Binary Tress**

There are two types of basic trees : unordered and ordered tree. In an unordered tree, there is no specific order for the children nodes i.e. children nodes can vary in numbers. But in ordered tree, an order is imposed for children nodes i.e. the children nodes are in specific order. The data structure built on ordered tree is named as ordered tree data structure which is most common form of tree data structure. The Binary tree is a fundamental data structure used in computer science. This is useful in rapidly storing data structures and rapidly retrieving stored data.

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. Root is the parent node and the leaves are the child nodes divided into two. One child node is placed at the left and the other child node is placed to the right of the parent node.

In other words, A Binary Tree T is defined as a finite set of elements called nodes such that

1. T, is empty (called the null true or empty tree), or
2. Contains a distinguished node R. called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2.

Two restricted forms of binary tree are sufficiently important to warrant special names. Each node in a full binary tree is either (1) an internal node with exactly two non-empty children or (2) a leaf. A complete binary tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height d, all levels except possibly level d-1 are completely full. The bottom level has its nodes filled in from the left side.

**Representation of Binary Tree**

Let T be a binary tree. This section discusses two ways of representing T in memory. The first and usual way is called the link representation of T and is analogous to the way linked lists are represented in memory. The second way, which uses a single array, called the sequential representation of T. The main requirement of any representation of T is that one should have direct access to the root R of T and, given any node N of T, one should haw direct access to the children of N.

**Linked Representation of Binary Trees**

Consider a binary tree T, Unless otherwise stated or implied. T will be maintained in memory by means of a linked representation which uses three parallel arrays. INFO. LEFT and RIGHT and a

pointer variable ROOT as follows. First of all. each node N of T will correspond to a location K such that;

(1) lNFO[K] contains the data at the node N.

(2) LEFT[K] contains the location of the left child of node N.

(3) RIGHT[K] contains the location of the right child of node N.

Furthermore, ROOT will contain the location of the root R of T. lf any subtree is empty, then the corresponding pointer will contain the null value; if the tree T itself is empty, then ROOT will contain the null value.

1. Binary Tree T (b) Linked representation of T

**Sequential Representation of Binary Tree**

This representation uses only a single array TREE as follows:

1. The root of T is stored in TREE [1].
2. If a node N occupies TREE[K], then its left child is stored in TREE[2\*K] and the right child is stored in TREE[2K+1]

Figure: Sequential representation of T

Again. NULL is used to indicate an empty subtree. In particular. TREE[l] = NULL indicates that the tree is empty. The sequential representation of the binary tree T appears above. Observe that we require 15 locations in the array TREE even though T has only 10 nodes, In fact, if we included null entries for the successors of the terminal nodes, then we would actually require TREE[31] for the right successor of TREE[l5]. Generally speaking, the sequential representation of a tree with depth d will require an array with approximately 2d+1 elements, Accordingly, this sequential representation is usually inefficient unless, the binary tree T is complete or nearly complete.

**Traversing of Binary Trees**

Often we wish to process a binary tree by “visiting” each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a traversal. Any traversal that lists every node in the tree exactly once is called an enumeration of the tree’s nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship. The reason we traverse a binary tree is to examine each of its node. Many different binary tree algorithms involve traversals. For example if we wish to count the number of nodes in a tree we must visit each node. If we wish to find the largest value in each node, we must examine the value contained in each node.

There are two fundamentally different kinds of binary tree traversals:

a). Depth- first: access is up to the depth of a tree occurred and continue.

b). Breadth-first: access nodes that are on same level and continue.

Depth-first traversal is more efficient and commonly used traversing technique compared to Breadth- first traversal. There are three different types of depth-first traversals:

***Preorder traversal:***  The root is visited first(pre) and then the left and right subtrees are traversed.

***Inorder traversal:*** The left subtree is traversed and then the root is visited and finally the right subtree is traversed.

***Postorder traversal:*** The left and right subtrees are traversed and then the root is visited afterwards (post).

Let us take the example of the binary tree shown in figure next.

The order of traversal under these schemes, of the above tree is as follows:

**Preorder**

**A B D E F C G H J K**

**Inorder**

**D B F E A G C J H K**

**Postorder**

**D F E B G J K H C A**

* 1. **Binary Search Tree**

A binary search tree, also known as Binary Sorted Tree, is a type of binary tree, which satisfies the following conditions:

* The data value in each node is a key (unique) value, that is, no two nodes can have identical values.
* The data values in the nodes of the left subtree, if exists, is smaller than the value in the root node.
* The data values in the nodes of the right subtree, if exists, is greater than the value in the root node.
* The left and the right subtrees, if exists, are also binary search trees.

Binary search tree store data in nodes such that the data in the root is greater than the left subtree parent node and lesser than the right subtree parent node. These trees can be used to search a particular data item. This is done by incrementing the pointer according to the search data being smaller or greater than the node until the search data is matched or until all the nodes are traversed. The following figure shows a binary search tree.

**Figure : Binary Search Tree**

The definition of a binary search tree given in this section assumes that all the node values are distinct. There is an analogous definition of a binary search tree which admits duplicates, that is, in which each node N has the following property; The value at N is greater than equal to value in the left subtree of N and is less than or equal to the value in right subtree of N. When this definition is,used, the operations in the next section must be modified accordingly.

**Operations on a Binary Search Tree:**

The various operations that can be done on a binary search tree are:

* Searching a node in BST
* Insertion of item in BST
* Deletion
* Case 1: Deleting a node with no child.
* Case 2: Deleting a node having a single child.
* Case 3: Deleting a node having two children.

**Search a node:**

Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order. The element to be searched is compared with the value in the root node. If the element is smaller than the value in the root node, then the searching will proceed to the left subtree and if the element is greater than the value in the root node, then the searching will proceed to the right subtree. This process is repeated again and again until the element to be searched is found or NULL value is encountered.

Suppose T is a binary search tree. we discuss the basic operations of searching and inserting with respect to T, In fact, the searching and inserting will be given by a single Search and insertion algorithm. Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree.

(i) If ITEM < N. proceed to the left child of N

(ii) If ITEM > N. proceed to the right child of N.

(b) Repeat Step (a) until one of the following occurs:

i) We meet a node N such that ITEM = N. In this case the search is successful.

(ii) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

**Deletion in Binary Search Tree**

Suppose T is a binary search tree and suppose an ITEM of information is given. This section gives an algorithm which deletes ITEM from the tree T. The deletion algorithm first uses Procedure **FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)** to find the location of the node N which contains ITEM and also the location of the parent node P(N). The way N is deleted from the tree depends primarily on the number of children of node N, There are three cases:

**Case 1.** N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N> by the null pointer. For example if node with value 18 is to be delete, then

Before deleting 18 After deleting 18

**Case 2.** N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N. suppose the node with value 23 is to be deleted then it will be replaced by its only child 18.

Before deleting 23 After deleting 23

Case 3. N has two children. Let S(N) denote the inorder successor of N. Then N is deleted from T by first deleting S(N) from T (by using Case l or Case 2) and then replacing node N in T by the node S(N). suppose the node with value 14 is to be deleted, then it will be replaced by node 18.

Before deleting 14 After deleting 14

Observe that the third case is much more complicated than the first two cases. In all three cases the memory space of the deleted node N is returned to the AVAIL list.

**Important definition**

* **Tree:** A finite set of one or more nodes used to represent data containing a hierarchical relationship between elements.
* **Node:** The item of information.
* **Degree:** The number of subtrees of a node.
* **Leaf:** Nodes that have degree zero.
* **Siblings:** children of same parent.
* **Children:** Nodes with the same root node.
* **Forest:** A group of nodes without root node.
* **Binary tree:** A finite se of elements that is either empty or is partitioned into two disjoint subsets.
* **Complete binary tree:** A complete binary tree of depth ‘d’ whose all leaves are at level d.
* **Prefix notation:** A way of writing a mathematical expression with each operator appearing before its operands.
* **Preorder:** A way to traverse a tree, visiting each node before its children, root then left and then right node.
* **Postorder:** A way to traverse a tree, visiting the children of each node(left to right) before the node itself.
* **Inorder:** A way to traverse a tree, visiting the left subtree, then the root, and then right subtree.

**Questions**

**Q1.** What is non-linear data structure and in which cases it is preferable to represent data through non-linear data structure.

**Q2.** What is a Tree?

**Q3.** What is a Binary Tree? State different characteristics associated with the Binary Trees.

**Q4.** Write short notes on the following:

1. Strictly Binary Tree
2. Complete Binary Tree
3. Height/Depth of a Tree

**Q5.** Write a program to create a Binary Tree. Also write function to insert a node into a Binary Tree.

**Unit 6(Sorting and Searching)**

**Introduction to sorting and searching**

**Search algorithm (Linear and Binary)**

**Sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Selection Sort, Merge Sort,**

**Heap Sort**

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort the records by zip code so that we can print and mail them more cheaply. We might use sorting as an

intrinsic part of an algorithm to solve some other problem, such as when computing the minimum-cost spanning tree. Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised. Some of these algorithms are straightforward adaptations of schemes we use in everyday life. Others are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer. After years of study, there are still unsolved problems related to sorting. New algorithms are still being developed and refined for special purpose applications.

**Searching Algorithms**

Consider a membership file in which each record contains among other data the name and telephone number of its member. Suppose we are given the name of a member and We want to find his or her telephone number. One way to do this is to linearly search through the file. i.e. to apply the following algorithm:

**Linear Search**

Search each record of the file, one at a time. until finding thc given Name and hence the corresponding telephone number. First of all, it is clear that the time required to execute the algorithm is proportional to the number of comparisons. Also, assuming that each name in the file is equally likely to be picked, it is intuitively clear that the average number of comparisons for a file with n records is equal to n/2; that is, the complexity of the linear search algorithm is given by C(n) = n/2.

The above algorithm would be impossible in practice if we were searching through a list consisting of thousands of names, as in a telephone book, However. if the names are sorted alphabetically, as in telephone books, then we can use an efficient algorithm called binary search.

**Binary Search**

Compare the given Name with the name in the middle of the list; this tells which hall of the list contains Name. Then compare Name with the name in the middle of the correct half to determine which quarter of the list contains Name. Continue the process until finding Name in the list. One can show that the complexity of the binary search algorithm is given by C(n)=log2n. Thus, for example, one will not require more than l5 comparisons to find a given Name in a list containing 25 000 names.

Although the binary search algorithm is a very efficient algorithm, it has some major drawbacks. Specifically, the algorithm assumes that one has direct access to the middle name in the list or a sublist. This means that the list must be stored in some type of array. Unfortunately, inserting an element in an array requires elements to be moved down the list and deleting an element from an array requires element to be moved up the list.

The telephone company solves the above problem by printing a new directory every year while keeping a separate temporary file for new telephone customers. That is, the telephone company updates its files every year. On the other hand, a bank may want to insert a new customer in its file almost instantaneously. Accordingly, a linearly sorted list may not be the best data structure for a bank.

**Selection Sort**

Consider the problem of sorting a pile of phone bills for the past year. Another intuitive approach might be to look through the pile until you find the bill for January, and pull that out. Then look through the remaining pile until you find the bill for February, and add that behind January. Proceed through the ever-shrinking pile of bills to select the next one in order until you are done. This is the inspiration for our last O(n2) sort, called Selection Sort. The ith pass of Selection Sort “selects” the ith smallest key in the array, placing that record into position i. In other words, Selection Sort first finds the smallest key in an unsorted list, then the second smallest, and so on. Its unique feature is that there are few record swaps. To find the next smallest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record in place. Thus, the total number of swaps required will be n-1 (we get the last record in place “for free”). The procedure is as follows:

Suppose an array A with n elements A[l], A[2]……..A[N] is in memorv, The selection sort

algorithm for sorting A works as follows: first find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position, And so on. More precisely:

*Pass l.* Find the location LOC of the smallest in the list of N elements A[l], A[2]. …… A[N]. and then interchange A[LOC] and A[1], Then A[1] is sorted.

*Pass 2.* Find the location LOC of the smallest in the sublist of N - l elements A[2], A[3]. .... A[N] and then interchange A[LOC] and A[2]. Then: A[l], A[2] is sorted since A[1] ≤ A[2].

*Pass 3,* Find the location LOC of the smallest in the sublist of N - 2 elements A[3], A[4] .. A[N] and then interchange A[LOC] and A[3] Then: A[1], A[2], ..., A[3] is sorted. Since A[2] ≤ A[3].

…………………………………………………………………………………………

*Pass N - 1*. Find the location LOC of the smaller of the elements A[N-1], A[N], and then interchange A[LOC] and A[N – l]. Then: A[l], A[2], ......A[N] is sorted. since A[N - 1] ≤ A[N].

Thus A is sorted after N - l passes.

**Insertion Sort**

Imagine that you have a stack of phone bills from the past two years and that you wish to organize them by date. A fairly natural way to do this might be to look at the first two bills and put them in order. Then take the third bill and put it into the right order with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This naturally intuitive process is the inspiration for our first sorting algorithm, called Insertion Sort. Insertion Sort iterates through a list of values. Each value is inserted in turn at the correct position within a sorted list composed of the data already processed. Thus, the cost for Insertion Sort in the best case is O(n). The following example explains the process of insertion sort.

Suppose an array A with n elements A[l]. A[2], ... A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1]. A[2] …..A[K - l]. That is:

*Pass 1.* A[l] by itself is trivially sorted,

*Pass 2. A[2] is inserted either before or after A[l] so that: A[1]. A[2] is sorted.*

*Pass* 3. A[3] is inserted into its proper place in A[l], A[2], that is. before A[1], between A[l] and A[2] or after A[2], so that; A[l], A[2],A[3] is sorted.

*Pass 4.* A[4] is inserted into its proper place in A[1], A[2], A[3] so that: A[l], A[2], A[3], A[4] is sorted.

……………………………………………………………………………………….

*Pass N.* A[N] is inserted into its proper place in A[1], A[2] .... A[N – l] so that: A[l], A[2], .... A[N] is sorted,

This sorting algorithm is frequently used when n is small. For example, this algorithm is very popular with bridge players when they are first sorting their Cards. While the best case is significantly faster than the worst case, the worst case is usually a more reliable indication of the “typical” running time. However, there are situations where we can expect the input to be in sorted or nearly sorted order. One example is when an already sorted list is slightly disordered by a small number of additions to the list; restoring sorted order using Insertion Sort might be a good idea if we know that the disordering is slight.

**Bubble Sort**

Our next sorting algorithm is called Bubble Sort. Bubble Sort is often taught to novice programmers in introductory computer science courses. This is unfortunate, because Bubble Sort has no redeeming features whatsoever. It is a relatively slow sort, it is no easier to understand than Insertion Sort, it does not correspond to any intuitive counterpart in “everyday” use, and it has a poor best-case running time. Bubble Sort consists of a simple double **for** loop. The first iteration of the inner **for** loop moves through the record array from bottom to top, comparing adjacent keys. If the lower-indexed key’s value is greater than its higher-indexed neighbor, then the two values are swapped. Once the smallest value is encountered, this process will cause it to “bubble” up to the top of the array. The second pass through the array repeats this process. However, because we know that the smallest value reached the top of the array on the first pass, there is no need to compare the top two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less value than the preceding pass. Determining Bubble Sort’s number of comparisons is easy. Bubble Sort’s running time is roughly the same in the best, average, and worst cases. The number of swaps required depends on how often a value is less than the one immediately preceding it in the array. We can expect this to occur for about half the comparisons in the average case, leading to O(n2) for the expected number of swaps. The actual number of swaps performed by Bubble Sort will be identical to that performed by Insertion Sort. The process is explained below:

Suppose an array A with n elements A[l]. A[2], ... A[N] is in memory. The bubble sort algorithm works as follows:

*Step l*. Compare A[l] and A[2] and arrange them in the desired order, so that A[1] <A[2]. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Then compare A[3] and A[4] and arrange them so that A[3] < A[4]. Continue until we compare A[N - 1] with A[N] and arrange them so that A[N - l] < A[N].

Observe that Step 1 involves n-l comparisons. (During Step"l, the largest element is "bubbled up” to the nth position or “sinks” to the nth position.) When Step l is completed. A[N] will contain the largest element.

*Step 2.* Repeat Step l with one less comparison; that is, now We stop after We Compare and possibly rearrange A[N - 2] and A[N-1].

*Step 3.* Repeat Step l with two fewer comparisons; that is, we stop after we compare and possibly rearrange A[N - 3] and A[N - 2].

*Step N - l*. Compare A[1] with A[2] and arrange them so that A[1] <A[2]

After n - 1 steps, the list will be sorted in increasing order.

**Merge Sort**

A natural approach to problem solving is divide and conquer. In terms of sorting, we might consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow. A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together. This is the idea behind Mergesort. Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time. Surprisingly, even though it is based on a simple concept, it is relatively difficult to implement in practice. Before discussing how to implement Mergesort, we will first examine the merge function. Merging two sorted sublists is quite simple. Function **merge** examines the first element of each sublist and picks the smaller value as the smallest element overall. This smaller value is removed from its sublist and placed into the output list. Merging continues in this way, comparing the front elements of the sublists and continually appending the smaller to the output list until no more input elements remain.

Implementing Mergesort presents a number of technical difficulties. The first decision is how to represent the lists. Mergesort lends itself well to sorting a singly linked list because merging does not require random access to the list elements. Thus, Mergesort is the method of choice when the input is in the form of a linked list. Implementing **merge** for linked lists is straightforward, because we need only remove items from the front of the input lists and append items to the output list. Breaking the input list into two equal halves presents some difficulty. Ideally we would just break the lists into front and back halves. However, even if we know the length of the list in advance, it would still be necessary to traverse halfway down the linked list to reach the beginning of the second half. A simpler method, which does not rely on knowing the length of the list in advance, assigns elements of the input list alternating between the two sublists. The first element is assigned to the first sublist, the second element to the second sublist, the third to first sublist, the fourth to the second sublist, and so on. This requires one complete pass through the input list to build the sublists. When the input to Mergesort is an array, splitting input into two subarrays is easy if we know the array bounds. Merging is also easy if we merge the subarrays into a second array. Note that this approach requires twice the amount of space as any of the sorting methods presented so far, which is a serious disadvantage for Mergesort. It is possible to merge the subarrays without using a second array, but this is extremely difficult to do efficiently and is not really practical. Merging the two subarrays into a second array, while simple to implement, presents another difficulty. The merge process ends with the sorted list in the auxiliary array. Consider how the recursive nature of Mergesort breaks the original array into subarrays, as shown in Figure 7.8. Mergesort is recursively called until subarrays of size 1 have

been created, requiring log n levels of recursion. These subarrays are merged into subarrays of size 2, which are in turn merged into subarrays of size 4, and so on. We need to avoid having each merge operation require a new array. With some difficulty, an algorithm can be devised that alternates between two arrays. A much simpler approach is to copy the sorted sublists to the auxiliary array first, and then merge them back to the original array.

**Quick Sort**

While Mergesort uses the most obvious form of divide and conquer (split the list in half then sort the halves), it is not the only way that we can break down the sorting problem. And we saw that doing the merge step for Mergesort when using an array implementation is not so easy. So perhaps a different divide and conquer strategy might turn out to be more efficient? Quicksort is aptly named because, when properly implemented, it is the fastest known general-purpose in-memory sorting algorithm in the average case. It does not require the extra array needed by Mergesort, so it is space efficient as well. Quicksort is widely used, and is typically the algorithm implemented in a library sort routine such as the UNIX **qsort** function. Interestingly, Quicksort is hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications. Before we get to Quicksort, consider for a moment the practicality of using a Binary Search Tree for sorting. You could insert all of the values to be sorted into the BST one by one, and then traverse the completed tree using an inorder traversal. The output would form a sorted list. This approach has a number of drawbacks, including the extra space required by BST pointers and the amount of time required to insert nodes into the tree. However, this method introduces some interesting ideas. First, the root of the BST (i.e., the first node inserted) splits the list into two sublists: The left subtree contains those values in the list less than the root value while the right subtree contains those values in the list greater than or equal to the root value. Thus, the BST implicitly implements a “divide and conquer” approach to sorting the left and right subtrees. Quicksort implements this concept in a much more efficient way.

Quicksort first selects a value called the pivot. Assume that the input array contains k values less than the pivot. The records are then rearranged in such a way that the k values less than the pivot are placed in the first, or leftmost, k positions in the array, and the values greater than or equal to the pivot are placed in the last, or rightmost, n-k positions. This is called a partition of the array. The values placed in a given partition need not be sorted with respect to each other. All that is required is that all values end up in the correct partition. The pivot value itself is placed in position k. Quicksort then proceeds to sort the resulting subarrays now on either side of the pivot, one of size k and the other of size n - k - 1. How are these values sorted? Because Quicksort is such a good algorithm, using Quicksort on the subarrays would be appropriate. Unlike some of the sorts that we have seen earlier in this chapter, Quicksort might not seem very “natural” in that it is not an approach that a person is likely to use to sort real objects. But it should not be too surprising that a really efficient sort for huge numbers of abstract objects on a computer would be rather different from our experiences with sorting a relatively few physical objects. In the worst case, Quicksort is O(n2). This is terrible, no better than Bubble Sort. When will this worst case occur? Only when each pivot yields a bad partitioning of the array. One such condition occurs when the array is already sorted. If the pivot values are selected at random, then this is extremely unlikely to happen.

**Heap Sort**

Our discussion of Quicksort began by considering the practicality of using a binary search tree for sorting. The BST requires more space than the other sorting methods and will be slower than Quicksort or Mergesort due to the relative expense of inserting values into the tree. There is also the possibility that the BST might be unbalanced, leading to a O(n2) worst-case running time. Subtree balance in the BST is closely related to Quicksort’s partition step. Quicksort’s pivot serves roughly the same purpose as the BST root value in that the left partition (subtree) stores values less than the pivot (root) value, while the right partition (subtree) stores values greater than or equal to the pivot (root). A good sorting algorithm can be devised based on a tree structure more suited to the purpose. In particular, we would like the tree to be balanced, space efficient, and fast. The algorithm should take advantage of the fact that sorting is a special purpose application in that all of the values to be stored are available at the start. This means that we do not necessarily need to insert one value at a time into the tree structure.

Heapsort has all of the advantages just listed. The complete binary tree is balanced, its array representation is space efficient, and we can load all values into the tree at once, taking advantage of the efficient function. The asymptotic performance of Heapsort is O(n log n) in the best, average, and worst cases. It is not as fast as Quicksort in the average case (by a constant factor), but Heapsort has special properties that will make it particularly useful when sorting data sets too large to fit in main memory. A sorting algorithm based on max-heaps is quite straightforward. First we use the heap building algorithm INSHEAP(TREE, N, ITEM) to convert the array into max-heap order. Then we repeatedly remove the maximum value from the heap using the algorithm DELHEAP(TREE, N, ITEM) restoring the heap property each time that we do so, until the heap is empty. Note that each time we remove the maximum element from the heap, it is placed at the end of the array. Assume the n elements are stored in array positions 0 through n - 1. After removing the maximum value from the heap and readjusting, the maximum value will now be placed in position n- 1 of the array. The heap is now considered to be of size n - 1. Removing the new maximum (root) value places the second largest

value in position n-2 of the array. After removing each of the remaining values in turn, the array will be properly sorted from least to greatest. This is why Heapsort uses a max-heap rather than a min-heap as might have been expected.

**Questions**

**Q1**. Why is there a need for sorting?

**Q2.** Why is bubble sort called by this name?

**Q3.** Write a ‘C’ program for bubble sort. Discuss its time complexity.

**Q4.** Explain Quick sort.

**Q5**. Write a ‘C’ program for quick sort. Explain its time complexity.

**Q6**. When is bubble sort better than the quick sort?

**Q7.** What does Insertion Sort means? Write down the methodology with example.

**Q8**. What is merging? Write a ‘C’ program for Merge-Sort.

**Q9**. Discuss heap sort.