Study Guide

For

Data Structures and Algorithm Analysis

Department of Computer Science

Faculty of Informatics

Prepared by

Mesfin Fantaye

January 2012

Title of the Study Guide: Data Structures and Algorithm Analysis

Summary of the study guide

Representing information is fundamental to computer science. The primary purpose of most computer programs is to store and retrieve information usually as fast as possible. For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science. And that is what this course is about - helping you to understand how to structure information to support efficient processing.

This course has three primary goals. The first is to present the commonly used data structures. The second goal is to introduce the idea of tradeoffs and reinforce the concept that there are costs and benefits associated with every data structure. This is done by describing, for each data structure, the amount of space and time required for typical operations. The third goal is to teach how to measure the effectiveness of a data structure or algorithm.

To this end you are required to make your readings and programming practices as per the objectives sated in this study guide. Please make sure that you meet all the objectives listed under the respective units of this guide.

Unit 1 Review of Arrays, structures, and pointers

Unit summary

A pointer is an object that can be used to access another object. A pointer provides *indirect* access rather than *direct* access to an object. In C++ a pointer is an object that stores an address (i.e., a location in memory) where other data are stored. An address is expected to be an integer, so a pointer object can usually be represented internally as an (unsigned) int. What makes a pointer object more than just a plain integer is that we can access the datum being pointed at. Doing so is known as *dereferencing* the pointer.

The **array** is the basic mechanism for storing a collection of identically-typed objects. A different type of aggregate type is the structure, which stores a collection of objects that need not be of the same type.

General objectives

The general objectives of the unit is to guide the student gear their studies so that they will be able to explain about why arrays, structures and pointers are important; how the vector is used to implement arrays in C++; how the string is used to implement strings in C++; how basic pointer syntax and *dynamic memory allocation* are used; and how pointers, arrays, and structures are passed as parameters to functions.

Specific objectives

Pertinent to this unit, can you able to;

- explain about arrays
- declare arrays and manipulate data in arrays
- explain about ‘‘array index out of bounds’’
- explain the restrictions on array processing
- pass an array as a parameter to a function
- demonstrate the knowledge of declaring C++ strings
- Examine the use of string functions to process strings
- Input data into—and output data from—a string
- Explicate the pointer data type and pointer variables

- Declare and manipulate pointer variables
- Use the address of operator and the dereferencing operator
- Declare, implement and use dynamic variables
- Explore how to use the new and delete operators to manipulate dynamic variables
- Perform pointer arithmetic
- Discover dynamic arrays
- Become aware of the shallow and deep copies of data
- Discover the peculiarities of classes with pointer member variables
- Distinguish the relationship between the address of operator and classes

Questions with answers

Q.1 The smallest element of an array's index is called its

(A) lower bound. **(B)** upper bound. **(C)** range. **(D)** extraction. **Ans. A**

Q.2 The extra key inserted at the end of the array is called a,

(A) End key. **(B)** Stop key. **(C)** Sentinel. **(D)** Transposition. **Ans. (C)**

Q.3 The largest element of an array index is called its

(A) lower bound. **(B)** range. **(C)** upper bound. **(D)** All of these. **Ans. (C)**

Q.4 How does an array differ from an ordinary variable?

Ans.

Array Vs. Ordinary Variable

Array is made up of similar data structure that exists in any language. Array is set of similar data types. Array is the collection of similar elements. These similar elements could be all int or all float or all char etc. Array of char is known as string. All elements of the given array must be of same type. Array is finite ordered set of homogeneous elements. The number of elements in the array is pre-specified.

An ordinary variable of a simple data type can store a single element only.

For example: *Ordinary variable*: - int a

Array: - int a[10]

Q.5 What values are automatically assigned to those array elements which are not explicitly initialized?

Ans.

Garbage values are automatically assigned to those array elements that are not explicitly initialized. If garbage class is auto then array is assumed to contain garbage values. If storage class were declared static or external then all the array elements would have a default initial value as zero.

Q.6 A two dimensional array TABLE [6] [8] is stored in row major order with base address 351. What is the address of TABLE [3] [4]?

Ans.

TABLE [6] [8]

Base address 351

Let us assume that TABLE[6] [8] is of type integer.

The formula for row major form to calculate address is

Loc(a[i] [j]) = base (a) + w [n (i –lbr) + (j-lbc)], where 'n' no. of column in array, 'w' no of bytes per storage location, 'lbr' lower bound for row index and 'lbc' lower bound for column index.

Loc(TABLE[3] [4]) = 351 + 2[8(3-0) + (4-0)]

= 351 +2[24+4]

= 351 + 56

=407

Q.7 Define the term array. How are two-dimensional arrays represented in memory? Explain how address of an element is calculated in a two dimensional array. **(8)**

Ans:

An *array* is a way to reference a series of memory locations using the same name. Each memory location is represented by an array element. An *array element* is similar to one variable except it is identified by an index value instead of a name. An *index* value is a number used to identify an array element.

Declaring a two dimensional array- A two dimensional array is declared similar to the way we declare a one-dimensional array except that we specify the number of elements in both dimensions. For example,

int grades[3][4];

The first bracket ([3]) tells the compiler that we are declaring 3 pointers, each pointing to an array. We are not talking about a pointer variable or pointer array.

Instead, we are saying that each element of the first dimension of a two dimensional array reference a corresponding second dimension. In this example, all the arrays pointed to by the first index are of the same size. The second index can be of variable size. For example, the previous statement declares a two dimensional array where there are 3 elements in the first dimension and 4 elements in the second dimension.

Two-dimensional array is represented in memory in following two ways:

1. Row major representation: To achieve this linear representation, the first row of the array is stored in the first memory locations reserved for the array, then the second row and so on.

2. Column major representation: Here all the elements of the column are stored next to one another.

In row major representation, the address is calculated in a two dimensional array as per the following formula. The address of a[i][j]=base(a)+(i*m+ j)*size where base(a) is the address of a[0][0], m is second dimension of array a and size represent size of the data type.

Q.8 What is a linear array? Explain how two dimensional arrays are represented in memory.

Ans:

An *array* is a way to reference a series of memory locations using the same name. Each memory location is represented by an array element. An *array element* is similar to one variable except it is identified by an index value instead of a name. An *index* value is a number used to identify an array element. The square bracket tells the computer that the value inside the square bracket is an index.

grades[0]

grades[1]

Representation of two-dimensional arrays in memory:-

Let grades be a 2-D array as grades[3][4]. The array will be represented in memory by a block of 3*4=12 sequential memory locations. It can be stored in two ways:- row major wise or column major wise. The first set of four array elements is placed in memory, followed by the second set of four array elements, and so on.

Questions without answers

1. The memory address of the first element of an array is called

a. floor address c. first address b. foundation address d. base address

2. The memory address of fifth element of an array can be calculated by the formula

a. LOC(Array[5]=Base(Array)+w(5-lower bound), where w is the number of words per memory cell for the array

b. LOC(Array[5])=Base(Array[5])+(5-lower bound), where w is the number of words per memory cell for the array

c. LOC(Array[5])=Base(Array[4])+(5-Upper bound), where w is the number of words per memory cell for the array d. None of above

3. Which of the following data structures are indexed structures?

a. linear arraysb. B. linked lists c. both of above d. none of above

4. Two dimensional arrays are also called

a. tables arrays b. matrix arrays c. both of above d. none of above

5. A variable P is called pointer if

a. P contains the address of an element in DATA.

b. P points to the address of first element in DATA

c. P can store only memory addresses

d. P contain the DATA and the address of DATA

6. Which of the following data structure can't store the non-homogeneous data elements?

a. Arrays b. Records c. Pointers d. None

7. Which of the following data structure store the homogeneous data elements?

a. Arrays b. Records c. Pointers d. None

8. Each data item in a record may be a group item composed of sub-items; those items which are indecomposable are called

a. elementary items c. Scalars b. Atoms d. all of above

9. The difference between linear array and a record is

a. An array is suitable for homogeneous data but hte data items in a record may have different data type

b. In a record, there may not be a natural ordering in opposed to linear array.

c. A record form a hierarchical structure but a linear array does not

d. All of above

10. Which of the following statement is false?

a. Arrays are dense lists and static data structure

b. data elements in linked list need not be stored in adjecent space in memory

c. pointers store the next data element of a list

d. linked lists are collection of the nodes that contain information part and next pointer

11. When new data are to be inserted into a data structure, but there is no available space; this situation is usually called

a. underflow c. Housefull b. Overflow d. saturated

12. What is the output of the following program segment?

```
int temp[5];
for (int i = 0; i < 5; i++)
        temp[i] = 2 * i - 3;
for (int i = 0; i < 5; i++)
        cout << temp[i] << " ";
cout << endl;
temp[0] = temp[4];
temp[4] = temp[1];
temp[2] = temp[3] + temp[0];
for (int i = 0; i < 5; i++)
        cout << temp[i] << " ";
cout << endl;
```

13. Suppose list is an array of five components of type int. What is stored in list after the following C++ code executes?

```
for (int i = 0; i < 5; i++)
{
        list[i] = 2 * i + 5;
        if (i % 2 == 0)
                list[i] = list[i] - 3;
}
```

14. What is wrong with the following code?

```
int *p;
int *q;
p = new int[5];
*p = 2;
for (int i = 1; i < 5; i++)
```

```
        p[i] = p[i - 1] + i;
q = p;
delete [] p;
for (int j = 0; j < 5; j++)
        cout << q[j] << " ";
cout << endl;
```

15. What is the output of the following code?

```
int *p;
int *q;
int i;
p = new int[5];
p[0] = 5;
for (i = 1; i < 5; i++)
        p[i] = p[i - 1] + 2 * i;
cout << "Array p: ";
for (i = 0; i < 5; i++)
        cout << p[i] << " ";
cout << endl;
q = new int[5];
for (i = 0; i < 5; i++)
        q[i] = p[4 - i];
cout << "Array q: ";
for (i = 0; i < 5; i++)
        cout << q[i] << " ";
cout << endl;
```

16. a. Write a statement that declares sales to be a pointer to a pointer of type double.

b. Write a C++ code that dynamically creates a two-dimensional array of five rows and seven columns and sales contains the base address of that array.

c. Write a C++ code that inputs data from the standard input device into the array sales.

d. Write a C++ code that deallocates the memory space occupied by the two-dimensional array to which sales points.

Unit 2 Overview of ADTs, Data Structures and Algorithms

Unit summary

Solving a problem involves processing data, and an important part of the solution is the careful organization of the data. This requires that we identify the collection of data items and basic operations that must be performed on them. Such a collection of data items together with the operations on the data is called an abstract data type (commonly abbreviated as ADT). The word abstract refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. We are thinking of what can be done with the data, not how it is done. A data structure is the implementation for an ADT.

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. We study data structures so that we can learn to write more efficient programs.

An **algorithm** is a clearly specified set of instructions a computer follows to solve a problem. Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources, such as time and space that the algorithm will require. This step is called *algorithm analysis.*

The amount of time that any algorithm takes to run almost always depends on the amount of input that it must process. We expect, for instance, that sorting 10,000 elements requires more time than sorting 10 elements. The running time of an algorithm is thus a function of the input size. The exact value of the function depends on many factors, such as the speed of the host machine, the quality of the compiler, and in some cases, the quality of the program.

General objective

The general objective of this chapter is to the set the stage for the rest of what is to follow, by presenting some higher order issues related to the selection and use of data structures. The process by which a designer selects a data structure appropriate to the task at hand will be

examined and then the role of abstraction in program design also be considered. Simple data types will be used with the ADTs they model, and how they are implemented to demonstrate ADTs.

Specific objectives

Pertinent to this unit, can you able to;

- Explain the principles of algorithm analysis,
- appreciate the significant effects of the physical medium employed
- discuss the tradeoffs between time and space requirements, or vice versa.
- Explicate the commonly used data structures, their related algorithms,
- design an algorithm that makes efficient use of the computer's resources
- discuss the approaches to solving a problem and the method of choosing between them
- describe methods for evaluating the efficiency of an algorithm or computer program
- explain the role of abstraction in program design.
- Explain the relationship between problems, algorithms, and programs
- Elucidate the Need for Data Structures
- Differentiate among the concepts type simple type, aggregate type, composite type, data item, data type, abstract data type, data structure

Questions with answers

Q.1 What is an algorithm? What are the characteristics of a good algorithm?

Ans:

An **algorithm** is "a step-by-step procedure for accomplishing some task" An algorithm can be given in many ways. For example, it can be written down in English (or French, or any other "natural" language). However, we are interested in algorithms which have been precisely specified using an appropriate mathematical formalism--such as a programming language.

Every algorithm should have the following five characteristics:

- **Input:** The algorithm should take zero or more input.
- **Output:** The algorithm should produce one or more outputs.
- **Definiteness:** Each and every step of algorithm should be defined unambiguously.

- **Effectiveness:** A human should be able to calculate the values involved in the procedure of the algorithm using paper and pencil.
- **Termination:** An algorithm must terminate after a finite number of steps.

Q.2 How do you find the complexity of an algorithm? What is the relation between the time and space complexities of an algorithm? Justify your answer with an example.

Ans:

Complexity of an algorithm is the measure of analysis of algorithm. Analyzing an algorithm means predicting the resources that the algorithm requires such as memory, communication bandwidth, logic gates and time. Most often it is computational time that is measured for finding a more suitable algorithm. This is known as time complexity of the algorithm. The running time of a program is described as a function of the size of its input. On a particular input, it is traditionally measured as the number of primitive operations or steps executed.

The analysis of algorithm focuses on time complexity and space complexity. As compared to time analysis, the analysis of space requirement for an algorithm is generally easier, but wherever necessary, both the techniques are used. The space is referred to as storage required in addition to the space required storing the input data. The amount of memory needed by program to run to completion is referred to as space complexity. For an algorithm, time complexity depends upon the size of the input, thus, it is a function of input size 'n'. So the amount of time needed by an algorithm to run to its completion is referred as time complexity.

The best algorithm to solve a given problem is one that requires less memory and takes less time to complete its execution. But in practice it is not always possible to achieve both of these objectives. There may be more than one approach to solve a same problem. One such approach may require more space but takes less time to complete its execution while the other approach requires less space but more time to complete its execution. Thus we may have to compromise one to improve the other. That is, we may be able to reduce space requirement by increasing running time or reducing running time by allocating more memory space. This situation where we compromise one to better the other is known as Time-space tradeoff.

Q.3 What do you mean by complexity of an algorithm? Explain the meaning of worst case analysis and best case analysis with an example.

Ans:

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. Therefore, the term "complexity" shall refer to the running time of the algorithm.

We find the complexity function f(n) for certain cases. The two cases one usually investigates in complexity theory are :-

i. Worst case:- the maximum value of f(n) for any possible input

ii. Best case:- the minimum possible value of f(n)

If we take an example of linear search in which an integer Item is to searched in an array Data. The complexity if the search algorithm is given by the number C of comparisons between Item and Data[k].

Worst case:-

The worst case occurs when Item is the last element in the array Data or is it not there at all. In both the cases, we get C(n)=n

The average case, we assume that the Item is present it the array and is likely to be present in any position in the array. Thus, the number of comparisons can be any of the numbers 1, 2, 3……..n and each number occurs with probability p = 1/n.

$C(n) = 1.\ 1/n + 2.1/n + \ldots + n.1/n$

$= (n+1) / 2$

thus the average number of comparisons needed to locate the Item in to array Data is approximately equal to half the number of elements in the Data list.

Q.4 Why do we use a symptotic notation in the study of algorithm? Describe commonly used asymptotic notations and give their significance.

Ans:

The running time of an algorithm depends upon various characteristics and slight variation in the characteristics varies the running time. The algorithm efficiency and performance in comparison to alternate algorithm is best described by the order of growth of the running time of an

algorithm. Suppose one algorithm for a problem has time complexity as c_3n^2 and another algorithm has $c_1n^3 + c_2n^2$ then it can be easily observed that the algorithm with complexity c_3n^2 will be faster than the one with complexity $c_1n^3 + c_2n^2$ for sufficiently larger values of n. Whatever be the value of c_1, c_2 and c_3 there will be an 'n' beyond which the algorithm with complexity c_3n^2 is faster than algorithm with complexity $c_1n^3 + c_2n^2$, we refer this n as breakeven point. It is difficult to measure the correct breakeven point analytically, so Asymptotic notation are introduced that describe the algorithm efficiency and performance in a meaningful way. These notations describe the behavior of time or space complexity for large instance characteristics. Some commonly used asymptotic notations are:

Big oh notation (O): The upper bound for the function 'f' is provided by the big oh notation (O). Considering 'g' to be a function from the non-negative integers to the positive real numbers, we define O(g) as the set of function f such that for some real constant c>0 and some non negative integers constant n_0 , f(n)>=cg(n) for all n>=n_0. Mathematically, O(g(n))={f(n): there exists positive constants such that 0<=f f(n)<=cg(n) for all n, n>=n_0} , we say "f is oh of g".

Big Omega notation (Ω): The lower bound for the function 'f' is provided by the big omega notation (Ω). Considering 'g' to be a function from the non-negative integers to the positive real numbers, we define Ω(g) as the set of function f such that for some real constant c>0 and some non negative integers constant n0 , f(n)>=cg(n) for all n>=n_0. Mathematically, Ω(g(n))={f(n): there exists positive constants such that 0<=cg(n) <=f(n) for all n, n>=n_0}.

Big Theta notation (Θ): The upper and lower bound for the function 'f' is provided by the big oh notation (Θ). Considering 'g' to be a function from the non-negative integers to the positive real numbers, we define Θ(g) as the set of function f such that for some real constant c1 and c2 >0 and some non negative integers constant n_0 , c1g(n)<=f f(n)<=c2g(n) for all n_n0. Mathematically, Θ(g(n))={f(n): there exists positive constants c1 and c2 such that c1g(n)<=f f(n)<=c2g(n) for all n, n>=n_0} , we say "f is oh of g"

Q5. Explain the concept of primitive data structures.

Ans.

Primitive Data Structure

These are the basic structure and are directly operated upon by the machine instructions. These in general have different representations on different computer. Integer floating point number, character constraints, string constants, pointer etc fall in this category.

Q6. Define ADT

Ans.

Abstract Data Type:- A useful tool for specifying the logical properties of data type is the abstract data type or ADT. A data type is a collection of values and a set of operation on those values. The term "ADT" refer to the basic mathematical concept that defined the data types. ADT is not concerned with implementation details at all. By specifying the mathematical and logical properties of data type or structures, the ADT is a useful guideline to implementation or a useful tool to programmers who wish to use the data type correctly.

An ADT consists of 2 parts: a value definition and an operation definition

The **value definition** defines the collection of values for the ADT and consists of 2 parts: a definition clause and a condition clause.

Operation definition: - Each operation is defined as an abstract form with 3 parts: a leader, the operational pre-condition and the post condition.

Q7. Explain the following:

(i) Complexity of an Algorithm. **(ii)** The space-time trade off algorithm.

Ans.

(i) Complexity of an Algorithm

An algorithm is a sequence of steps to solve a problem; there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends upon following consideration:-

1) Time Complexity 2) Space Complexity

Time Complexity:- The time complexity of an algorithm is the amount of time it needs to run to completion. Some of the reasons for studying time complexity are:-

- We may be interested to know in advance whether the program will provide a satisfactory real time response.
- There may be several possible solutions with different time requirement.

Space Complexity:- The space complexity of an algorithm is the amount of memory it needs to run to completion. Some of the reasons to study space complexity are: -

- There may be several possible solutions with in different space requirement.
- To estimate the size of the largest problem that a program can solve.

(ii) The Space – Time Trade Off

The best algorithm to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice it is not always possible to achieve both of these objectives. There may be more than one approach to solve a problem. One approach may require more space but less time to complete its execution. The 2nd approach may require less space but takes more time to complete execution. We choose 1st approach if time is a constraint and 2nd approach if space is a constraint. Thus we may have to sacrifice one at cost of the other. That is what we can say that there exists a time space trade among algorithm.

8. Define data structure.

Data Structure is a way of representing data that contains both the data item & their relationship with each other

Questions without answers

1. Define an ADT for character strings. Your ADT should consist of typical functions that can be performed on strings, with each function defined in terms of its input and output. Then define two different physical representations for strings.
2. Define an ADT for a list of integers. First, decide what functionality your ADT should provide. Then, specify your ADT in C++ in the form of an abstract class declaration, showing the functions, their parameters, and their return types.
3. Two main measures for the efficiency of an algorithm are

   a. Processor and memory c. Time and space b. Complexity and capacity d. Data and space
4. The time factor when determining the efficiency of algorithm is measured by

   a. Counting microseconds c. Counting the number of statements

   b. Counting the number of key operations d. Counting the kilobytes of algorithm
5. The space factor when determining the efficiency of algorithm is measured by

   a. Counting the maximum memory needed by the algorithm

   b. Counting the minimum memory needed by the algorithm

c. Counting the average memory needed by the algorithm

d. Counting the maximum disk space needed by the algorithm

6. Which of the following case does not exist in complexity theory

   a. Best case b. Worst case c. Average case d. Null case

7. The Worst case occur in linear search algorithm when

   a. Item is somewhere in the middle of the array

   b. Item is not in the array at all

   c. Item is the last element in the array

   d. Item is the last element in the array or is not there at all

8. The Average case occur in linear search algorithm

   a. When Item is somewhere in the middle of the array

   b. When Item is not in the array at all

   c. When Item is the last element in the array

   d. When Item is the last element in the array or is not there at all

9. The complexity of the average case of an algorithm is

   a. Much more complicated to analyze than that of worst case

   b. Much more simpler to analyze than that of worst case

   c. Sometimes more complicated and some other times simpler than that of worst case

   d. None or above

Unit 3 Algorithm Analysis

Unit summary

This unit introduces the motivation, basic notation, and fundamental techniques of algorithm analysis. We focus on a methodology known as asymptotic algorithm analysis, or simply asymptotic analysis. Asymptotic analysis attempts to estimate the resource consumption of an algorithm. It allows us to compare the relative costs of two or more algorithms for solving the same problem. Asymptotic analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program.

The unit concludes with a brief discussion of the practical difficulties encountered when empirically measuring the cost of a program, and some principles for code tuning to improve program efficiency.

General objectives

the student is is will be made to understand the concept of a growth rate, the rate at which the cost of an algorithm grows as the size of its input grows; the concept of upper and lower bounds for a growth rate, and how to estimate these bounds for a simple program, algorithm, or problem; and the difference between the cost of an algorithm (or program) and the cost of a problem.

Specific objectives

Pertinent to this unit, can you able to;

- compare two algorithms for solving some problem in terms of efficiency
- justify the needs and benefits of **Asymptotic algorithm analysis**
- demonstrate the application of **Asymptotic algorithm analysis**
- explain the reasons for ignoring constants when estimating the growth rate for the running time or other resource requirements of an algorithm.
- identify the resources required for the algorithm and the data structure
- distinguish factors that affect the running time of a program
- identify the basic operations required by an algorithm to process an input of a certain size
- distinguish size (or the number of inputs) processed
- define the **growth rate (running time)** for an algorithm

- differentiate among **linear growth rate (running time), quadratic growth rate, and exponential growth rate**
- differentiate among **best-case, average-case** and **worst-case** analysis of an algorithm
- make a distinction among upper bound (big-Oh) for the growth of the algorithm's running time, the lower bound (Ω) for an algorithm and Θ Notation
- determine the running-time equation for an algorithm
- Classify Function f(n) into either of O(g(n)), Ω(g(n)), or Θ(g(n))
- calculate the running time for a program
- determine the space requirements (space bounds) for the data structure
- explain the space/time tradeoff principle in algorithm design.
- Differentiate among the different techniques/methods of algorithm analysis, such as asymptotic method, empirical method, and simulation method

Questions with answers

Q.1 The complexity of multiplying two matrices of order m*n and n*p is

(A) mnp **(B)** mp **(C)** mn **(D)** np **Ans:A**

Q.2 Merging 4 sorted files containing 50, 10, 25 and 15 records will take____time

(A) O (100) **(B)** O (200) **(C)** O (175) **(D)** O (125) **Ans:A**

Q.3 The minimum number of multiplications and additions required to evaluate the polynomial $P = 4x^3+3x^2-15x+45$ is

(A) 6 & 3 **(B)** 4 & 2 **(C)** 3 & 3 **(D)** 8 & 3 **Ans: C**

Q.4 O(N) (linear time) is better than O(1) constant time.

(A) True **(B)** False **Ans:B**

Q.5 An algorithm is made up of two independent time complexities *f (n)* and *g (n)*. Then the complexities of the algorithm is in the order of

(A) *f(n)* x *g(n)* **(B)** *Max (f(n),g(n))* **(C)** *Min (f(n),g(n))* **(D)** *f(n) + g(n)* **Ans:B**

Q.6 Time complexities of three algorithms are given. Which should execute the slowest for large values of N?

(A) $\mathbf{O}(N^{1/2})$ **(B)** $O(N)$ **(C)** $O(\log N)$ **(D)** *None of these* **Ans:B**

Q.7. The complexity of linear search algorithm is

a. O(n) b. O(log n) c. $O(n^2)$ d. O(n log n) Ans. a

Q.8. The complexity of Binary search algorithm is

a. O(n) b. O(log) c. O(n^2) d. O(n log n) Ans. b

Q.9. The complexity of Bubble sort algorithm is

a. O(n) b. O(log n) c. O(n2) d. O(n log n) Ans. c

Q.10. The complexity of merge sort algorithm is

a. O(n) b. O(log n) c. O(n2) d. O(n log n) Ans. d

Q11. What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2n on the same machine.

Ans.

F (n) = 100 n2

G(n) = 2n

Value of n so that 100 n2 < 2n

The smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2n on the same machine is 15.

Questions without answers

1. Programs **A** and B are analyzed and are found to have worst-case running times no greater than 15ON log N and N2, respectively. Which program has the better guarantee on the running time for large values of N (N > 10,000)?

2. In terms of N, what is the running time of the following algorithm to compute ***X N :***

```
double power( double x, int n )
{
    double result = 1.0;

    for( int i = 0; i < n; i++ )
        result *= x;
    return result;
}
```

3. An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 (assuming that low-order terms are negligible) if the running time is

a. linear. b. O(Nlog *N).* c. quadratic. d. cubic.

4. An algorithm takes 0.5 ms for input size 100. How large a problem can be solved in 1 min (assuming that low-order terms are negligible) if the running time is

a. linear. b. O(N log N) c. quadratic. d. cubic.

5. Order the following functions by growth rate: N, $\sqrt{N}$, $N^{1.5}$, N^2, N log N, N log log N, N $\log^2 N$, N log (N^2),2 /N, 2^N, $2^{N/2}$, 37, N^3, and $N^2 \log$ ***N.*** Indicate which functions grow at the same rate.

6. Graph the functions 8n, 4nlog n, $2n^2$, n^3, and 2^n using a logarithmic scale for the x- and y-axes. That is, if the function is f(n) is y, plot this as a point with x-coordinate at log n and y-coordinate at log y.

Unit 4 Introduction to list

Unit summary

We define a list to be a finite, ordered sequence of data items known as elements. "Ordered" in this definition means that each element has a position in the list. A list is said to be empty when it contains no elements. The number of elements currently stored is called the length of the list. The beginning of the list is called the head, the end of the list is called the tail. There might or might not be some relationship between the value of an element and its position in the list. For example, sorted lists have their elements positioned in ascending order of value, while unsorted lists have no particular relationship between element values and positions.

In computer programs, lists are very useful abstract data types. They are members of a general category of abstract data types called containers, whose purpose is to hold other objects.

From a theoretical point of view, a list is a homogeneous collection of elements, with a linear relationship between elements. *Linear* means that, at the logical level, each element in the list except the first one has a unique predecessor, and each element except the last one has a unique successor. (At the implementation level, a relationship also exists between the elements, but the physical relationship may not be the same as the logical one.) The number of items in the list, which we call the length of the list, is a property of a list. That is, every list has a length.

Lists can be unsorted—their elements may be placed into the list in no particular order—or they can be sorted in a variety of ways. For instance, a list of numbers can be sorted by value, a list of strings can be sorted alphabetically, and a list of grades can be sorted numerically. When the elements in a sorted list are of composite types, their logical (and often physical) order is determined by one of the members of the structure, called the key member. For example, a list of students on the honor roll can be sorted alphabetically by name or numerically by student identification number. In the first case, the name is the key; in the second case, the identification number is the key. Such sorted lists are also called *key-sorted lists*.

If a list cannot contain items with duplicate keys, it is said to have *unique* keys. This chapter deals with both unsorted lists and lists of elements with unique keys, sorted from smallest to largest key value.

General objectives

In some languages, the list is a built-in structure. In C++, while lists are provided in the Standard Template Library, the techniques for building lists and other abstract data types are so important that we guide you in this unit what you should read to design and write your own. Moreover, you will also learn how to manipulate lists using their basic operations. This unit also requires you to identify different types of lists and analyze their performance of lists.

Specific objectives

Pertinent to this unit, can you able to;

- Use the list operations to implement utility routines to do the following application-level tasks:
    - Print the list of elements
    - Create a list of elements from a file
- Implement list operations for both unsorted lists and sorted lists:
    - Create and destroy a list
    - Determine whether the list is full
    - Insert an element
    - Retrieve an element
    - Delete an element
- Explain the use of Big-O notation to describe the amount of work done by an algorithm
- Compare the unsorted list operations and the sorted list operations in terms of Big-O approximations
- Define class constructors
- Overload the relational operators less than (<) and equality (==)
- Identify and apply the phases of an object-oriented methodology
- Implement a circular linked list
- Implement a linked list with a header node, a trailer node, or both
- Implement a doubly linked list
- Distinguish between shallow copying and deep copying
- Overload the assignment operator

- Implement a linked list as an array of records
- Implement dynamic binding with virtual functions

Questions with answers

Q.1 Consider a linked list of n elements. What is the time taken to insert an element after an element pointed by some pointer?

(A) O (1) **(B)** O (log2 n) **(C)** O (n) **(D)** O (n log2 n)

Ans:A

Q.2 In a circular linked list

(A) components are all linked together in some sequential manner.

(B) there is no beginning and no end.

(C) components are arranged hierarchically.

(D) forward and backward traversal within the list is permitted.

Ans:B

Q.3 In a linked list with n nodes, the time taken to insert an element after an element pointed by some pointer is

(A) 0 (1) **(B)** 0 (log n) **(C)** 0 (n) **(D)** 0 (n 1og n)

Ans:A

Q.4 A linear collection of data elements where the linear node is given by means of pointer is called

(A) linked list **(B)** node list **(C)** primitive list **(D)** None of these

Ans:A

Q.5 Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?

(A) Deleting a node whose location in given

(B) Searching of an unsorted list for a given item

(C) Inverting a node after the node with given location

(D) Traversing a list to process each node

Ans. (A)

Q.6 The time required to delete a node x from a doubly linked list having n nodes is

(A) O (n) **(B)** O (log n) **(C)** O (1) **(D)** O (n log n)

Ans. (C)

Q.7 Write an algorithm to insert a node in the beginning of the linked list.

Ans:

```
/* structure containing a data part and link part */
struct node
{
        int data ;
        struct node * link ;
} ;
/* Following adds a new node at the beginning of the linked list */
void addatbeg ( struct node **q, int num )
{
        struct node *temp ;
        /* add new node */
        temp = malloc ( sizeof ( struct node ) ) ;
        temp -> data = num ;
        temp -> link = *q ;
        *q = temp ;
}
```

Q.8 Write an algorithm to evaluate a postfix expression. Execute your algorithm using the following postfix expression as your input: a b + c d +*f^.

Ans.

```
To evaluate a postfix expression:
      clear the stack
      symb = next input character
      while(not end of input)
      {
            If(symb is an operand)
               Push onto the stack;
            else
            {
                  pop two operands from the stack;
                  result = op1 symb op2;
                  push result onto the stack;
                  }
                  symb = next input character;
            }
}
return (pop(stack));
```

*The given input as postfix expression is:- ab+cd+*f^*

Symb	Stack	Evaluation
a	a	
b	a b	
+	pop a and b	(a + b)
	Push (a + b)	
c	(a + b) c	
d	(a + b) c d	
+	pop c and d	(c + d)
	Push (c + d)	
*	pop (a + b) and (c + d)	(a + b) * (c + d)
f	(a + b) * (c + d) f	
^	pop (a + b) * (c + d) and f	(a + b) * (c + d) ^ f

The result of evaluation is ***((a + b) * (c + d)) ^ f***

Q.9. Define a linked-list? How are these stored in the memory? Suppose the linked list in the memory consisting of numerical values. Write a procedure for each of the following:

(i) To find the maximum MAX of the values in the list.

(ii) To find the average MEAN of the values in the list.

(iii) To find the product PROD of the values in the list.

Ans.

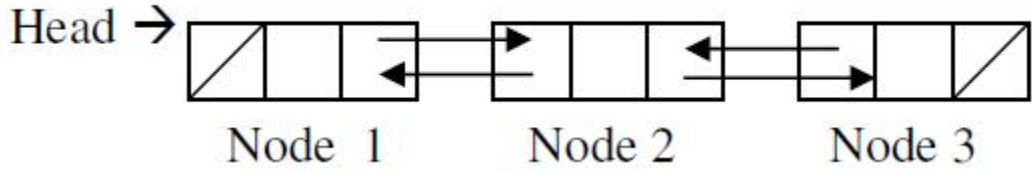
Linked List :-

A linked list is a linear collection of data elements called nodes. The linear order is given by pointer. Each node is divided into 2 or more parts. A linked list can be of the following types:-

- Linear linked list or one way list
- Doubly linked list or two way list.
- Circular linked list
- Header linked list

Representation of Linked list in Memory:-

Every node has an info part and a pointer to the next node also called as link. The number of pointers is two in case of doubly linked list. for example :



An external pointer points to the beginning of the list and last node in list has NULL. The space is allocated for nodes in the list using malloc or calloc functions. The nodes of the list are scattered in the memory with links to give linear order to the list.

```
(i)
float MAX_OF(node * start)
{
n=start;
max=n->k;
while(n!= NULL)
{
if(max<=n->k)
max=n->k;
}
return max;
(ii) float MEAN_OF(struct node * start)
{
int h=0; struct node *n;
n=start;
while (n!= NULL)
{
s+=n->k;h++;
n=n->next;
}
return s/h;
}
(iii) float PROD_OF(node *start)
{
float p; struct node *n;
n=start;
while(n!= NULL)
{
p*=n->k;
n=n->next;
}
return p;
}
```

Assume a double-linked non-circular list of integers. The order of the elements in the list is not relevant. A list element is defined as follows:

```
struct List
{
int key;
struct List* prev;
struct List* next;
}
```

A list can be accessed through a pointer to the head and end, respectively. Assume two lists. List L1 is instantiated as shown in Figure 1.

```
struct List* h1;
struct List* t1;
struct List* h2;
struct List* t2;
```

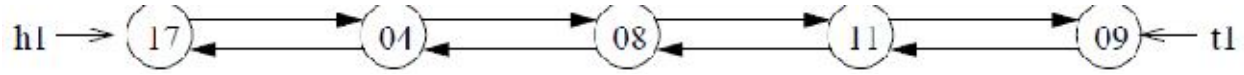


Figure 1: Original Double-linked List L1

Develop a fragment of C++ code that splits a double-linked list by removing from L1 all elements with key values less than or equal to Max. All elements less than or equal Max shall be put into a new double-linked list L2, while all other elements shall remain in the original list L1. For instance, with Max = 9 and list L1 from Figure 1, the two lists displayed in Figure 2 are produced.

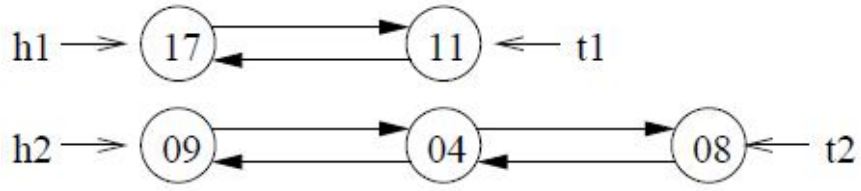


Figure 2: Resulting Double-linked Lists $L1$ and $L2$

Design your algorithm so that no memory is allocated or deallocated. Illustrate your algorithm by showing the lists at different points during the computation.

Question without answers

1. The situation when in a linked list START=NULL is
a. underflow b. Overflow c. Housefull d. Saturated
2. Which of the following is two way list?
a. grounded header list b. circular header list c. linked list with header and trailer nodes d. none
3. Two linked lists contain information of the same type in ascending order. Write a module to merge them to a single linked list that is sorted.
4. Implement a circularly and doubly linked list.
5. Suppose that a singly linked list is implemented with both a header and a tail node. Describe constant-time algorithms to
a. insert item x before position p. b. remove the item stored at position p.

Unit 5 Stack and queue

Unit summary

A **stack** is a last in, first out (LIFO) abstract data type and linear data structure. A stack can have any abstract data type as an element, but is characterized by only three fundamental operations: *push*, *pop* and *stack top*. The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed). The stack top operation gets the data from the top-most position and returns it to the user without deleting it. The same underflow state can also occur in stack top operation if stack is empty.

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

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

A simple singly linked list is sufficient to implement a stack—it only requires that the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node.

A **queue** is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data

structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.

General objective

The previous unit discussed representations for lists in general. In this unit two important list-like structures called the stack and the queue are treated. Along with presenting these fundamental data structures, the other goals of the chapter are to: (1) Give examples of separating a logical representation in the form of an ADT from a physical implementation for a data structure. (2) Illustrate the use of asymptotic analysis in the context of some simple operations that you might already be familiar with. In this way you can begin to see how asymptotic analysis works, without the complications that arise when analyzing more sophisticated algorithms and data structures. (3) Introduce the concept and use of dictionaries.

Specific objectives

Pertinent to this unit, can you able to;

- Describe a stack and its operations at a logical level
- Demonstrate the effect of stack operations using a particular implementation of a stack
- Implement the Stack ADT in an array-based implementation
- Declare variables of pointer types
- Access the variables to which pointers point
- Create and access dynamically allocated data

- Explain the difference between static and dynamic allocation of the space in which the elements of an abstract data type are stored
- Use the C++ template mechanism for defining generic data types
- Define and use an array in dynamic storage
- Describe the structure of a queue and its operations at a logical level
- Demonstrate the effect of queue operations using a particular implementation of a queue
- Implement the Queue ADT using an array-based implementation
- Use inheritance to create a Counted Queue ADT
- Implement the Stack ADT as a linked data structure
- Implement the Queue ADT as a linked data structure
- Implement the Unsorted List ADT as a linked data structure
- Implement the Sorted List ADT as a linked data structure
- Compare alternative implementations of an abstract data type with respect to performance

Questions with answers

Q.1 The postfix form of the expression (A+B)*(C*D–E)*F / G is

(A) $AB+CD*E-FG/**$ **(B)** $AB+CD*E-F**G/$
(C) $AB+CD*E-*F*G/$ **(D)** $AB+CDE*-*F*G/$

Ans: A

Q.2 A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a

(A) queue. **(B)** stack. **(C)** tree. **(D)** linked list.

Ans:A

Q.3 What is the postfix form of the following prefix expression -A/B*C$DE

(A) ABCDE$*/- **(B)** A-BCDE$*/- **(C)** ABC$ED*/- **(D)** A-BCDE$*/

Ans:A

Q.4 The data structure required to evaluate a postfix expression is

(A) queue **(B)** stack **(C)** array **(D)** linked-list

Ans:B

Q.5 The data structure required to check whether an expression contains balanced parenthesis is

(A) Stack **(B)** Queue **(C)** Tree **(D)** Array

Ans:A

Q.6 What data structure would you mostly likely see in a nonrecursive implementation of a recursive algorithm?

(A) Stack **(B)** Linked list **(C)** Queue **(D)** Trees

Ans:A

Q.7 The process of accessing data stored in a serial access memory is similar to manipulating data on a

(A) heap **(B)** queue **(C)** stack **(D)** binary tree

Ans:C

Q.8 The postfix form of A*B+C/D is

(A) *AB/CD+ **(B)** AB*CD/+ **(C)** A*BC+/D **(D)** ABCD+/*

Ans:B

Q.9 Let the following circular queue can accommodate maximum six elements with the following data

front = 2 rear = 4

queue = _____; L, M, N, ___, ___

What will happen after ADD O operation takes place?

(A) front = 2 rear = 5

queue = _____; L, M, N, O, ___

(B) front = 3 rear = 5

queue = L, M, N, O, ___

(C) front = 3 rear = 4

queue = _____; L, M, N, O, ___

(D) front = 2 rear = 4

queue = L, M, N, O, ___

Ans:A

Q.10 What is the postfix form of the following prefix *+*ab–cd*

(A) *ab+cd–** **(B)** *abc+*–* **(C)** *ab+*cd–* **(D)** *ab+*cd–*

Ans:A

Q.11 A stack is to be implemented using an array. The associated declarations are:

```
int stack[100];
int top = 0;
```

Give the statement to perform push operation.

Ans.

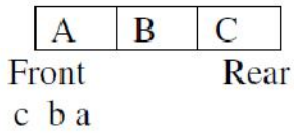
Let us assume that if stack is empty then its top has value –1.

Top ranges from 0 – 99. Let item be the data to be inserted into stack

```
int stack [100];
int top = 0;
int item ;
If (top == 99)
cout<< ("stack overflow");
Else
stack[top++] = item;
```

Q.12 Assume that a queue is available for pushing and popping elements. Given an input sequence a, b, c, (c be the first element), give the output sequence of elements if the rightmost element given above is the first to be popped from the queue.

Ans.



Q.13 A queue is a,
(A) FIFO (First In First Out) list. **(B)** LIFO (Last In First Out) list. **(C)** Ordered array. **(D)** Linear tree.
Ans. (A)
Q.14 Which data structure is needed to convert infix notation to postfix notation?
(A) Branch **(B)** Queue **(C)** Tree **(D)** Stack
Ans. (D)
Q.15 The prefix form of A-B/ (C * D ^ E) is,
(A) -/*^ACBDE **(B)** -ABCD*^DE **(C)** -A/B*C^DE **(D)** -A/BC*^DE
Ans. (C)
Q.16 What is the result of the following operation Top (Push (S, X))
(A) X **(B)** null **(C)** S **(D)** None of these.
Ans. (A)
Q.17 The prefix form of an infix expression $p + q - r * t$ is

(A) $+ pq - *rt$. **(B)** $- +pqr * t$.
(C) $- +pq * rt$. **(D)** $- + * pqrt$.

Ans. (C)
Q.18 Which data structure is used for implementing recursion?
(A) Queue. **(B)** Stack. **(C)** Arrays. **(D)** List.
Ans. (B)
Q.19 The equivalent prefix expression for the following infix expression (A+B)-(C+D*E)/F*G is
(A) -+AB*/+C*DEFG **(B)** /-+AB*+C*DEFG **(C)** -/+AB*+CDE*FG **(D)** -+AB*/+CDE*FG
Ans. (A)
Q.20 The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is
(A) 600. **(B) 350**. **(C)** 650. **(D)** 588.
Ans. (B)

Questions without answers

1 What are circular queues? Write down routines for inserting and deleting elements from a circular queue implemented using arrays.
2 Convert the following infix expressions into its equivalent postfix expressions;
(i) (A + B - D)/(E − F)+G (ii) A*(B+D)/ E − F*(G +H K)
3. Using array to implement the queue structure, write an algorithm/program to
(i) Insert an element in the queue. (ii) Delete an element from the queue.
4. Define a stack. Describe ways to implement stack.
5. Write down any four applications of queues.

6. Which of the following name does not relate to stacks?

a. FIFO lists b. LIFO list c. Piles d. Push-down lists

18. The term "push" and "pop" is related to the

a. array b. Lists c. Stacks d. all of above

19. A data structure where elements can be added or removed at either end but not in the middle

a. Linked lists b. Stacks c. Queues d. Deque

Unit 6 Sorting

Unit summary

Putting an unsorted list of data elements into order—*sorting*—is a very common and useful operation. The goal is to come up with better, more efficient sorts. Because sorting a large number of elements can be extremely time-consuming, a good sorting algorithm is very desirable. This is one area in which programmers are sometimes encouraged to sacrifice clarity in favor of speed of execution.

How do we describe efficiency? We pick an operation central to most sorting algorithms: the operation that compares two values to see which is smaller. In our study of sorting algorithms, we relate the number of comparisons to the number of elements in the list (*N*) as a rough measure of the efficiency of each algorithm. The number of swaps made is another measure of sorting efficiency. In the exercises, we ask you to analyze the sorting algorithms developed in this chapter in terms of data movements.

Another efficiency consideration is the amount of memory space required. In general, memory space is not a very important factor in choosing a sorting algorithm.

The usual time versus space tradeoff applies to sorts—more space often means less time, and vice versa. Because processing time is the factor that applies most often to sorting algorithms, we consider it in detail here. Of course, as in any application, the programmer must determine goals and requirements before selecting an algorithm and starting to write code.

General objectives

The present chapter covers several standard algorithms appropriate for sorting a collection of records that fit in the computer's main memory. We review the straight selection sort and the bubble sort, then we review a more complex sorting algorithm the quick sort algorithm and introduce two additional complex sorts: merge sort and heap sort.

Specific objectives

Pertinent to this unit, can you able to;

- Design and implement the following sorting algorithms:

  - Straight selection sort
  - Merge sort
  - Bubble sort
  - Heap sort
  - Insertion sort
  - Radix sort
- Compare the efficiency of the sorting algorithms, in terms of Big-O complexity and space requirements
- Discuss other efficiency considerations: sorting small numbers of elements, programmer time, and sorting arrays of large data elements
- Sort on several keys
- Discuss the performances of the following search algorithms:
  - Sequential search of an unsorted list
  - Sequential search of a sorted list
  - Binary search
  - Searching a high-probability sorted list
- Define the following terms:
  - Hashing
  - Linear probing
  - Rehashing
  - Clustering
  - Collisions
- Design and implement an appropriate hashing function for an application
- Design and implement a collision-resolution algorithm for a hash table
- Discuss the efficiency considerations for the searching and hashing algorithms, in terms of Big-O notation

Questions with answers

Q1. You have to sort a list L consisting of a sorted list followed by a few "random" elements.
Which of the following sorting methods would be especially suitable for such a task?
(A) Bubble sort **(B)** Selection sort **(C)** Quick sort **(D)** Insertion sort
Ans:D

Q.2 The number of interchanges required to sort 5, 1, 6, 2 4 in ascending order using Bubble Sort is
(A) 6 **(B)** 5 **(C)** 7 **(D)** 8
Ans:B

Q.3 In worst case Quick Sort has order
(A) O (n log n) **(B)** O (n^2/2) **(C)** O (log n) **(D)** O (n^2/4)
Ans:B

Q.4 A sort which relatively passes through a list to exchange the first element with any element less than it and then repeats with a new first element is called
(A) insertion sort. **(B)** selection sort. **(C)** heap sort. **(D)** quick sort.
Ans:D

Q.5 Which of the following sorting algorithms does not have a worst case running time of 2 O (n^2) ?
(A) Insertion sort **(B)** Merge sort **(C)** Quick sort **(D)** Bubble sort
Ans:B

Q.6 The quick sort algorithm exploits _____ design technique
(A) Greedy **(B)** Dynamic programming **(C)** Divide and Conquer **(D)** Backtracking
Ans: C

Q.7 The total number of companions required to merge 4 sorted files containing 15, 3, 9 and 8 records into a single sorted file is
(A) 66 **(B)** 39 **(C)** 15 **(D)** 3s
Ans: D

Q.8 Which of the following sorting methods would be most suitable for sorting a list which is almost sorted
(A) Bubble Sort **(B)** Insertion Sort **(C)** Selection Sort **(D)** Quick Sort
Ans:A

Q.9 Quick sort is also known as
(A) merge sort **(B)** heap sort **(C)** bubble sort **(D)** none of these
Ans:D

Q.10 The best average behavior is shown by
(A) Quick Sort **(B)** Merge Sort **(C)** Insertion Sort **(D)** Heap Sort
Ans:A

Q.10 Consider that n elements are to be sorted. What is the worst case time complexity of Bubble sort?
(A) O(1) **(B)** O(log2n) **(C)** O(n) **(D)** O(n2)
Ans. (D)

Q.11 Which of the following sorting algorithm is stable
(A) insertion sort. **(B)** bubble sort. **(C)** quick sort. **(D)** heap sort.
Ans. (D)

Q.12 The worst case of quick sort has order
(A) O(n2) **(B)** O(n) **(C)** O (n log2 n) **(D) O** (log2 n)
Ans. (A)

Questions without answers

Q.1 Describe insertion sort with a proper algorithm. What is the complexity of insertion sort in the worst case?

Q.2 Which sorting algorithm is best if the list is already sorted? Why?

Q.3 What is the time complexity of Merge sort and Heap sort algorithms?

Q.4 Sort the sequence 8, 1,4, I , 5,9, 2, 6, 5 by using

a. insertion sort.
b. Shellsort for the increments { 1, 3, 5 1.
c. mergesort.
d. quicksort. with the middle element as pivot and no cutoff (show all steps).
e. quicksort, with median-of-three pivot selection and a cutoff of 3.

Q. 5. When all keys are equal, what is the running time of

a. insertion sort.
b. Shellsort.
c. mergesort.
d. quicksort.

Q.6. When the input has been sorted, what is the running time of

a. insertion sort.
b. Shellsort.
c. mergesort.
d. quicksort.

Q.7. When the input has been sorted in reverse order, what is the running time of

a, insertion sort.
b. Shellsort.
c. mergesort.
d. quicksort.

Unit 7 Searching

Unit summary

Organizing and retrieving information is at the heart of most computer applications, and searching is surely the most frequently performed of all computing tasks.

Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set. The more common view of searching is an attempt to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.
We can categorize search algorithms into three general approaches:

1. Sequential and list methods.
2. Direct access by key value (hashing).
3. Tree indexing methods.

General objective

In this unit we treat the first two approaches. Accordingly, we consider methods for searching data stored in lists. List in this context means any list implementation including a linked list or an array. Most of these methods are appropriate for sequences (i.e., duplicate key values are allowed), although special techniques applicable to sets are discussed then we will proceed to the discussion of hashing, a technique for organizing data in an array such that the location of each record within the array is a function of its key value. Hashing is appropriate when records are stored either in RAM or on disk.

Specific objectives

Pertinent to this unit, can you able to;

- Distinguish among: jump search algorithm, divide and conquer algorithm, dictionary or interpolation search algorithm, Quadratic Binary Search algorithm,
- Explain hashing algorithm
- Differentiate between hash table, hash function and slot (a position in the hash table)
- Explain collision and collision resolution policy
- Explicate open hashing (separate chaining) and closed hashing (open addressing) techniques of collision resolution
- State when it will be appropriate applying the respective collision resolution techniques

- Implement both open and closed hashing
- Determine when it is appropriate to apply hashing
- Compute the cost associated each algorithm
- Implement all the search algorithms

Questions with answers

Q.1 If h is any hashing function and is used to hash n keys in to a table of size m, where n<=m, the expected number of collisions involving a particular key x is :

(A) less than 1. **(B)** less than n. **(C)** less than m. **(D)** less than n/2.

Ans:A

Q.2 A technique for direct search is

(A) Binary Search **(B)** Linear Search **(C)** Tree Search **(D)** Hashing

Ans:D

Q.3 The searching technique that takes O (1) time to find a data is

(A) Linear Search **(B)** Binary Search **(C)** Hashing **(D)** Tree Search

Ans:C

Q.4 The goal of hashing is to produce a search that takes

(A) *O(1)* time **(B)** $O(n^2)$ time **(C)** *O(log n)* time **(D)** *O(n log n)* time

Ans:A

Q.5 What do you mean by hashing? Explain any five popular hash functions.

Ans:

Hashing provides the direct access of record from the file no matter where the record is in the file. This is possible with the help of a hashing function H which map the key with the corresponding key address or location. It provides the keyto-address transformation.

Five popular hashing functions are as follows:

Division Method: An integer key x is divided by the table size m and the remainder is taken as the hash value. It can be defined as H(x)=x%m+1

For example, x=42 and m=13, H(42)=45%13+1=3+1=4

Midsquare Method: A key is multiplied by itself and the hash value is obtained by selecting an appropriate number of digits from the middle of the square. The same positions in the square must be used for all keys. For example if the key is 12345, square of this key is value 152399025. If 2 digit addresses is required then position 4th and 5th can be chosen, giving address 39.

Folding Method: A key is broken into several parts. Each part has the same length as that of the required address except the last part. The parts are added together, ignoring the last carry, we obtain the hash address for key K.

Multiplicative method: In this method a real number c such that 0<c<1 is selected. For a nonnegative integral key x, the hash function is defined as H(x)=[m(cx%1)]+1

Here,cx%1 is the fractional part of cx and [] denotes the greatest integer less than or equal to its contents.

Digit Analysis: This method forms addresses by selecting and shifting digits of the original key. For a given key set, the same positions in the key and same rearrangement pattern must be used. For example, a key 7654321 is transformed to the address 1247 by selecting digits in position 1,2,4 and 7 then by reversing their order.

Q6. Explain Hash Tables, Hash function and Hashing Techniques?

Ans.

Hash Table:

A hash table is a data structure in which the location of a data item is determined directly as a function of data item itself rather than by a sequence of comparison. Under ideal condition, the time required to locate a data item in a hash table is 0(1) ie. It is constant and DOES not depend on the number of data items stored.

When the set of K of keys stored is much smaller then the universe U of all possible keys, a hash table require much less storage spare than a direct address table.

Hash Function

A hash function h is simply a mathematical formula that manipulates the key in some form to compute the index for this key in the hash table. Eg a hash function can divide the key by some number, usually the size of the hash table and return remainder as the index for the key. In general, we say that a hash function h maps the universe U of keys into the slots of a hash table T[0….n-1]. This process of mapping keys to appropriate slots in a hash table is known as **hashing.**

The main reconsideration for choosing hash function is :-

1) It should be possible to compute it efficiently.
2) It should distribute the keys uniformly across the hash table i.e. it should keep the number. of collisions as minimum as possible.

Hashing Techniques:

There is variety of hashing techniques. Some of them are:-

a) Division Method:- In this method, key K to be mapped into one of the m states in the hash table is divided by m and the remainder of this division taken as index into the hash table. That is the hash function is h(k) = k mod m where mod is the modules operation.

b) **Multiplication Method**: The multiplication method operates in 2 steps. In the 1st step the key value K is multiplied by a constant A in the range O<A<1 and the fractional part of the value obtained above is multiplied by m and the floor of the result is taken as the hash values. That is the hash function is h(k) = [m (K A mod 1)] where "K A mod 1" means the fractional part KA of KA-[KA].A

A= (5-1/2)=0.6180334887

(c) **Midsquare Method**:- this operates in 2 steps. In the first step the square of the key value K is taken. In the 2nd step, the hash value is obtained by deleting digits from ends of the squared values ie. K2. The hash function is h(k) = s where s is obtained by deleting digits from both sides of K2.

Questions without answers

1. What are the array indices for a hash table of size 1 l?
2. Given the input (4371, 1323, 6173, 4199, 4344, 9679, 19891, a fixed table size of 10, and a hash function H(X) = X mod 10, show the resulting
   a. linear probing hash table.
   b. quadratic probing hash table.
   c. separate chaining hash table.

Unit 8 Trees

Unit summary

The list representations of data have a fundamental limitation: Either search or insert can be made efficient, but not both at the same time. Tree structures permit both efficient access and update to large collections of data. Binary trees in particular are widely used and relatively easy to implement. But binary trees are useful for many things besides searching. Just a few examples of applications that trees can speed up include prioritizing jobs, describing mathematical expressions and the syntactic elements of computer programs, or organizing the information needed to drive data compression algorithms. Almost all operating systems store files in trees or treelike structures. Trees are also used in compiler design, text processing, and searching algorithms.

Unit general objectives

In this unit we define a general tree and discuss how it is used in a file system, examine the binary tree, implement tree operations using recursion, and nonrecursive traversal of a tree. The basic binary search tree is also discussed and a method for adding order statistics (i.e., the find^{Kth} operation) will also be seen, the three different ways to eliminate the O(N) worst case (namely, the AVL tree, red-black tree, and AA-tree) are examined, the binary search implementation of the STL set and map, and use of the B-tree to search a large database quickly is also studied.

Specific objectives

Pertinent to this unit, can you able to;

- Define binary trees
- Clearly distinguish among the concepts: node, subtree, root node, edge, parent node, children node, leaf node, internal node, path, length of path, ancestor, descendant, depth of a node, height of a tree, level, full binary tree, and complete binary tree, tree traversal, enumeration
- Differentiate among preorder, postorder and inorder traversal
- Demonstrate the ability to traverse a binary tree
- Implement binary tree nodes by pointer-based binary tree node implementations and/or array-based implementation for complete binary trees.

- determining the space requirements for a given implementation (pointer-base or array-based)
- state the properties of binary search trees
- differentiate between Heaps and Priority Queues

Questions with answers

Q.1 If a node having two children is deleted from a binary tree, it is replaced by its
(A) Inorder predecessor **(B)** Inorder successor **(C)** Preorder predecessor **(D)** None of the above
Ans:B

Q.2 A full binary tree with 2n+1 nodes contain
(A) n leaf nodes **(B)** n non-leaf nodes **(C)** n-1 leaf nodes **(D)** n-1 non-leaf nodes
Ans:B

Q.3 If a node in a BST has two children, then its inorder predecessor has
(A) no left child **(B)** no right child **(C)** two children **(D)** no child
Ans:B

Q.4 A binary tree in which if all its levels except possibly the last, have the maximum number of nodes and all the nodes at the last level appear as far left as possible, is known as
(A) full binary tree. **(B)** AVL tree. **(C)** threaded tree. **(D)** complete binary tree.
Ans:A

Q.5 The number of different directed trees with 3 nodes are
(A) 2 **(B)** 3 **(C)** 4 **(D)** 5
Ans: B

Q.6 One can convert a binary tree into its mirror image by traversing it in
(A) inorder **(B)** preorder **(C)** postorder **(D)** any order
Ans:C

Q.7 The complexity of searching an element from a set of n elements using Binary search algorithm is
(A) O(n) **(B)** O(log n) **(C)** O(n2) **(D)** O(n log n)
Ans:B

Q.8 The number of leaf nodes in a complete binary tree of depth d is
(A) 2d **(B)** 2d–1+1 **(C)** 2d+1+1 **(D)** 2d+1
Ans:A

Q.9 A B-tree of minimum degree t can maximum _____ pointers in a node.
(A) t–1 **(B)** 2t–1 **(C)** 2t **(D)** t
Ans:D

Q.10 A BST is traversed in the following order recursively: Right, root, left. The output sequence will be in
(A) Ascending order **(B)** Descending order **(C)** Bitomic sequence **(D)** No specific order
Ans:B

Q.11 The pre-order and post order traversal of a Binary Tree generates the same output. The tree can have maximum
(A) Three nodes **(B)** Two nodes **(C)** One node **(D)** Any number of nodes

Ans:C

Q.12 A binary tree of depth "d" is an almost complete binary tree if

(A) Each leaf in the tree is either at level "d" or at level "d–1"

(B) For any node "n" in the tree with a right descendent at level "d" all the left descendents of "n" that are leaves, are also at level "d"

(C) Both **(A)** & **(B)**

(D) None of the above

Ans:C

Q.13 One of the major drawback of B-Tree is the difficulty of traversing the keys sequentially.

(A) True **(B)** False

Ans:A

Q.14 A characteristic of the data that binary search uses but the linear search ignores is the_____.

(A) Order of the elements of the list. **(B)** Length of the list.

(C) Maximum value in list. **(D)** Type of elements of the list.

Ans. (A)

Q.15 How many nodes in a tree have no ancestors.

(A) 0 **(B)** 1 **(C)** 2 **(D)** n

Ans. (B)

Q.16 In order to get the contents of a Binary search tree in ascending order, one has to traverse it in

(A) pre-order. **(B)** in-order. **(C)** post order. **(D)** not possible.

Ans. (B)

Q.17 In binary search, average number of comparison required for searching an element in a list if n numbers is

(A) log2 n . **(B)** n / 2 . **(C)** n. **(D)** n – 1.

Ans. (A)

Q.18 In order to get the information stored in a Binary Search Tree in the descending order, one should traverse it in which of the following order?

(A) left, root, right **(B)** root, left, right **(C)** right, root, left **(D)** right, left, root

Ans. (C)

Questions without answers

1 What is a Binary Search Tree (BST)? Make a BST for the following sequence of numbers.
45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48 Traverse the tree in Preorder, Inorder and postorder.

2 What are expression trees? Represent the following expression using a tree. Comment on the result that you get when this tree is traversed in Preorder, Inorder and postorder. (a-b) / ((c*d)+e)

3 How do you rotate a Binary Tree? Explain right and left rotations with the help of an example.

4. Which of the following is not the required condition for binary search algorithm?
    a. The list must be sorted
    b. there should be the direct access to the middle element in any sublist
    c. There must be mechanism to delete and/or insert elements in list
    d. none of above
5. Which of the following is not a limitation of binary search algorithm?

a. must use a sorted array
b. requirement of sorted array is expensive when a lot of insertion and deletions are needed
c. there must be a mechanism to access middle element directly
d. binary search algorithm is not efficient when the data elements are more than 1000.

Unit 9 Graphs

Unit summary

Graphs provide the ultimate in data structure flexibility. Graphs can model both real-world systems and abstract problems, so they are used in hundreds of applications.

Here is a small sampling of the range of problems that graphs are routinely applied to.

- Modeling connectivity in computer and communications networks.
- Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
- Modeling flow capacities in transportation networks.
- Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
- Modeling computer algorithms, showing transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
- Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

General objectives

To make the student familiar with the basic graph terminology and to define the two fundamental representations for graphs, the adjacency matrix and adjacency list. Presentation will be made on graph ADT and simple implementations based on the adjacency matrix and adjacency list. The most commonly used graph traversal algorithms, called depth-first and breadth-first search will be covered. The algorithms for solving some problems related to finding shortest routes in a graph and algorithms for finding the minimum-cost spanning tree, useful for determining lowest-cost connectivity in a network will also be dealt with.

Specific objectives;

Pertinent to this unit can you able to;

- Define the following terms related to graphs:
  - Directed graph
  - Complete graph
  - Undirected graph
  - Weighted graph
  - Vertex Adjacency matrix
  - Edge Adjacency list
  - Path
- Implement a graph using an adjacency matrix to represent the edges
- Explain the difference between a depth-first and a breadth-first search,
- Implement the searching strategies using stacks and queues for auxiliary storage
- Implement a shortest-path operation, using a priority queue to access the edge
- with the minimum weight
- Identify and use the algorithms for searching and traversing digraphs.

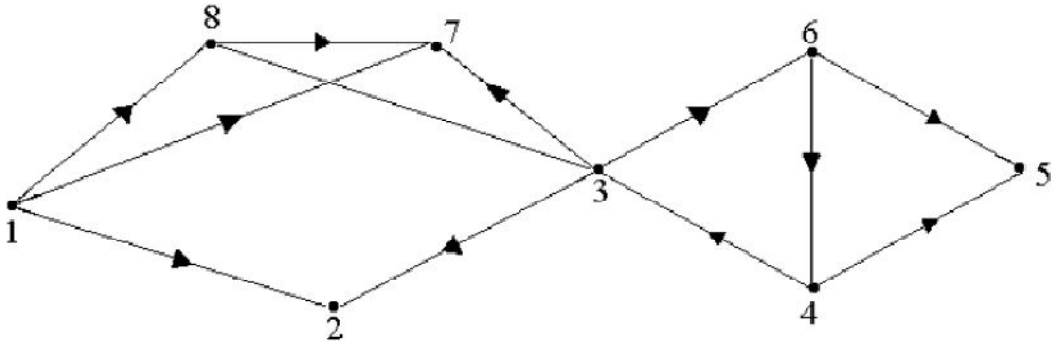
Questions with answers

Q.1 The maximum degree of any vertex in a simple graph with *n* vertices is
(A) *n–1* **(B)** *n+1* **(C)** *2n–1* **(D)** *n*
Ans: A
Q.2 The data structure required for Breadth First Traversal on a graph is
(A) queue **(B)** stack **(C)** array **(D)** tree
Ans: A
Q.3 A graph with n vertices will definitely have a parallel edge or self loop if the total number of edges are
(A) greater than n–1 **(B)** less than n(n–1) **(C)** greater than n(n–1)/2 **(D)** less than $n^2/2$
Ans:A
Q.4 In Breadth First Search of Graph, which of the following data structure is used?
(A) Stack. **(B)** Queue. **(C)** Linked List. **(D)** None of the above.
Ans. (B)
Q.5 For an undirected graph G with n vertices and e edges, the sum of the degrees of each vertex is
(A) ne **(B)** 2n **(C)** 2e **(D)** e^n
Ans. (C)
Q.6 The maximum degree of any vertex in a simple graph with *n* vertices is
(A) *n–1* **(B)** *n+1* **(C)** *2n–1* **(D)** *n*

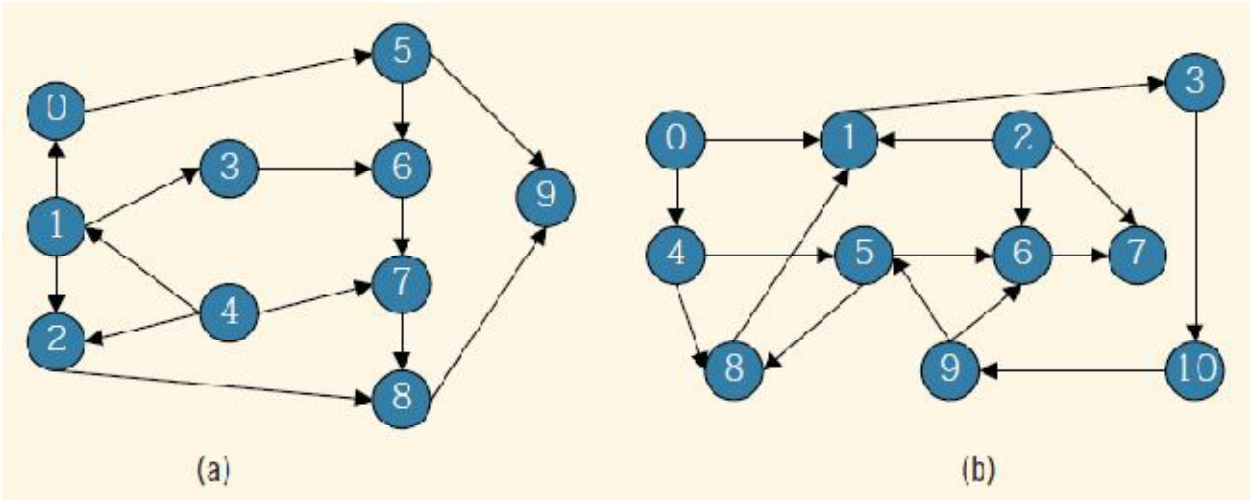
Ans: A

Questions without answers

1. Show the result of running BFS and DFS on the directed graph given below using vertex 3 as source. Show the status of the data structure used at each stage.



2. Which are the two standard ways of traversing a graph? Explain them with an example of each.
Use the graph in below for Exercises 1 through 6.



1. In Figure (a), find a path from vertex 0 to vertex 9.
2. In Figure (a), find a path from vertex 0 to vertex 9 via vertex 6.
3. In Figure (a), determine if the graph is simple. Also determine if there is a cycle in this graph.
4. In Figure (a), determine if the vertices 1 and 9 are connected. If these vertices are connected, find a path from vertex 1 to vertex 9.
5. In Figure (b), determine if the vertices 2 and 4 are connected. If these vertices are connected, find a path from vertex 2 to vertex 4.
6. In Figure (b), determine if the graph is simple. Also determine if there is a cycle in this graph.