KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2017onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT:DATA STRUCTURES **SEMESTER**: III
CODE: 17CAU301 **CLASS**: II B.C.A

SCOPE

Data structures and algorithms are the building blocks in computer programming. This course will give students a comprehensive introduction of common data structures, and algorithm design and analysis. This course also intends to teach data structures and algorithms for solving real problems that arise frequently in computer applications, and to teach principles and techniques of computational complexity.

OBJECTIVES

- possess intermediate level problem solving and algorithm development skills on the computer
- be able to analyze algorithms using big-Oh notation
- understand the fundamental data structures such as lists, trees, and graphs
- understand the fundamental algorithms such as searching, and sorting

UNIT-I

Arrays-Single and Multi-dimensional Arrays, Sparse Matrices (Array and Linked Representation).Stacks Implementing single / multiple stack/s in an Array; Prefix, Infix and Postfix expressions, Utility and conversion of these expressions from one to another; Applications of stack; Limitations of Array representation of stack

UNIT-II

Linked Lists Singly, Doubly and Circular Lists (Array and Linked representation); Normal and Circular, representation of Stack in Lists; Self Organizing Lists; Skip Lists Queues, Array and Linked representation of Queue, De-queue, Priority Queues

UNIT-III

Trees - Introduction to Tree as a data structure; Binary Trees (Insertion, Deletion , Recursive and Iterative Traversals on Binary Search Trees); Threaded Binary Trees (Insertion, Deletion, Traversals); Height-Balanced Trees (Various operations on AVL Trees).

UNIT-IV

Searching and Sorting: Linear Search, Binary Search, Comparison of Linear and Binary Search, Selection Sort, Insertion Sort, Insertion Sort, Shell Sort, Comparison of Sorting Techniques

UNIT-V

Hashing - Introduction to Hashing, Deleting from Hash Table, Efficiency of Rehash Methods, Hash Table Reordering, Resolving collusion by Open Addressing, Coalesced Hashing, Separate Chaining, Dynamic and Extendible Hashing, Choosing a Hash Function, Perfect Hashing, Function

Suggested Readings

1. Adam Drozdek. (2012). Data Structures and algorithm in C++(3rded.). New Delhi: Cengage Learning.
2. SartajSahni.(2011). Data Structures, Algorithms and applications in C++(2nded.). New Delhi: Universities Press.
3. Aaron, M. Tenenbaum., Moshe, J. Augenstein., &YedidyahLangsam.(2009). Data Structures Using C and C++(2nd ed.). New Delhi: PHI.
4. Robert, L. Kruse.(1999). Data Structures and Program Design in C++. New Delhi: Pearson.
5. Malik, D.S.(2010). Data Structure using C++(2nd ed.). New Delhi: Cengage Learning,.
6. Mark Allen Weiss.(2011). Data Structures and Algorithms Analysis in Java (3rd ed.). New Delhi:Pearson Education.
7. Aaron, M. Tenenbaum., Moshe, J. Augenstein.,& YedidyahLangsam.(2003). Data Structures Using Java.New Delhi: PHI.
8. Robert Lafore.(2003). Data Structures and Algorithms in Java(2^{nd} ed.). New Delhi: Pearson/ Macmillan Computer Pub.
9. John Hubbard.(2009). Data Structures with JAVA(2nd ed.). New Delhi: McGraw Hill Education (India) Private Limited.
10. Goodrich, M., & Tamassia, R.(2013). Data Structures and Algorithms Analysis in Java(4th ed.). New Delhi: Wiley.
11.Herbert Schildt.(2014).Java The Complete Reference (English)(9th ed.). New Delhi: Tata McGraw Hill.
12. Malik, D. S., &Nair, P.S. (2003).Data Structures Using Java. New Delhi: Course Technology.

WEB SITES

http://en.wikipedia.org/wiki/Data_structure
http://www.cs.sunysb.edu/~skiena/214/lectures/
www.amazon.com/Teach-Yourself-Structures-Algorithms

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University Established under Section 3 of UGC Act 1956)
Pollachi Main Road, Eacharani Post, Coimbatore-641 021

DEPARTMENT OF COMPUTER APPLICATIONS

SUBJECT NAME: DATA STRUCTURES

SUBJECT CODE: 17CAU301 **SEMESTER: III**

LECTURE PLAN

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT – I			
1.	1	Introduction to Arrays, Single & Multi Dimensional Array	T1:1 to 7
2.	1	Sparse Matrix	T1: 51 to 61,W1
3.	1	Array & Linked Representation	W1
4.	1	Stack Implementation Single / Multiple stack in an array	T1: 77 to 86
5.	1	Prefix , Postfix , Infix Expression	W1
6.	1	Utility & conversion of these Expression From one to another	W1
7.	1	Application of stack	R1: 97 to 100
8.	1	Limitation of Array Representation of stack	W1
9.	1	Revision	
	Total no. of Hours planned for Unit – I		9 hrs

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT – II			
1.	1	Introduction to Linked List, Singly	T1:106 to 110
2.	1	Doubly, Circular List	T1:140 to 154
3.	1	Array & Linked Representation.	T1: 140 to 154
4.	1	Normal & circular List	W1
5.	1	Representation of Stack in List	R1:128 to 134
6.	1	Self Organizing List, Skip List Queue	W1
7.	1	Array, Linked Representation of Queue	W1
8.	1	De-Queue, Priority Queue	T1:112 to 115
9.	1	Revision	
	Total no. of Hours planned for Unit – II		9 hrs

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT – III			
1.		Trees	T1:218 to 223
2.	1	Introduction to Trees-Introduction to Trees as a Data Structures	T1:218 to 223
3.	1	Binary Trees	T1:223 to 230
4.		Insertion , Deletion, Recursive	T1:223 to 230
5.	1	Iterative Traversal on Binary Search Trees	T1: 230 to 238
6.	1	Threaded Binary Trees	T1:239 to 242
7.	1	Insertion, Deletion, Traversal	W1
8.	1	Height-Balanced Trees Various Operations on AVL Trees	W1
9.	1	Revision	W1
	Total no. of Hours planned for Unit – III		9 hrs

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT – IV			
1.	1	Introduction to Searching, Sorting	T1:335 to 340
2.	1	Linear Search, Binary Search	T1:340 to 341
3.	1	Comparison of Linear & Binary Search	T1:341 to 345
4.	1	Selection sort, , Quick sort	T1: 345 to 347
5.	1	Insertion sort	T1: 347 to 350
6.	1	Shell Sort	T1: 352 to 360
7.	1	Comparison of Sorting Technique	T1:382 to 4000
8.	1	Comparison of Sorting Technique Cont…	T1: 382 to 400
9.	1	Revision	
	Total no. of Hours planned for Unit – IV		9 hrs

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
UNIT – V			
1.	1	Hashing-Introduction to Hashing	T1: 423 to 425
2.	1	Deleting from Hash Table	T1: 423 to 425
3.	1	Efficiency of Re-Hashing Methods, Hash Table Re-Ordering	T1:456 to 457
4.	1	Re-solving, Collusion by Open Addressing	W1
5.	1	Coalesced Hashing	W1
6.	1	Separate changing, Dynamic & Extendible Hashing	W1
7.	1	Choosing a Hash Function	W1
8.	1	Perfect Hashing Function	W1
9.	1	Recapitalization of Previous Year End Semester question Paper	-
10.	1	Discussion of ESE question Paper	-
11.	1	Discussion of ESE question Paper	-
12.	1	Discussion of ESE question Paper	-
	Total no. of Hours planned for Unit – V		12hrs

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

UNIT-I

Arrays–Single and Multi–dimensional Arrays, Sparse Matrices (Array and Linked Representation).Stacks Implementing single / multiple stack/s in an Array; Prefix, Infix and Postfix expressions, Utility and conversion of these expressions from one to another; Applications of stack; Limitations of Array representation of stack

Overview of Data Structures:

1.Introduction:

* To represent and store data in main memory or secondary memory we need a model. The different models used to organize data in the main memory are collectively referred as **data structures.**

***** The different models used to organize data in the secondary memory are collectively referred as file **structures.**

2. Basic terminologies of Data Organization:

Data:

The term 'DATA' simply refers to a value or a set of values. These values may represent anything about something, like it may be Roll No of a student, marks of a student, name of an employee, address of a person etc.

Data item:

A data item refers to a single unit of value. For example, roll number, name, date of birth, age, address and marks in each subject are data items. Data items that can be divided into sub items are called group items whereas those who cannot be divided into sub items are called elementary items. For example, an 'address' is a 'group item' as it is usually divided into sub items such as house-number, street number, locality, city, pin code etc. Likewise, a 'date' can be divided into day, month and year, a name can be divided into first name and surname. On the other hand, roll number, marks, city, pin code, etc. are normally treated as 'elementary items'.

Entity:

An entity is something that has a distinct, separate existence, though it need not be a material existence. An entity has certain 'attributes' or 'properties', which may be assigned values. The values

assigned may be either numeric or non-numeric. For example, a student is an entity. The possible attributes for a student can be roll number, name, date of birth, sex and class. The possible values for these attributes can be 32, kanu, 12/03/84, F, 11.

Entity Set:

An entity set is a collection of similar entities. For example, students of a class, employees of an organization etc. forms an entity set.

Record:

A record is a collection of related data items. For example, roll number, name, date of birth, sex, and class of a particular student such as 32, kanu, 12/03/84, F, 11. In fact, a record represents an entity.

File:

A file is a collection of related records. For example, a file containing records of all students in class, a file containing records of all employees of an organization. In fact, a file represents an entity set.

Key:

A key is a data item in a record that takes unique values. only one data item as a key called **primary key.** The other key are known as **alternate key**. Combination of some fields is known as **composite key.**

Information:

The terms data and information are same. Data is collection of values(raw data).Information is a processed data.

3. Concept of a Data Type:

A Data-Type in programming language is an attribute of a data, which tells the computer (and the programmer) important things about the concerned data. This involves what values it can take and what operations may be performed upon it. i.e. it declare:

Ø Set of values

Ø Set of operations

Most programming languages require the programmer to declare the data type of every data object, and most database systems require the user to specify the type of each data field. The available data types vary from one programming language to another, and from one database application to another, but the following usually exist in one form or another:

Integer:

Whole number; a number that has no fractional part. It takes digits as its set of values. The operations on integers include the arithmetic operations i.e. addition (+), subtraction (-), multiplication (*), and division (/).

Floating-point: A number with a decimal point. For example, 3 is an integer, but .5 is a floating-point number.

Character (text): Readable text.

3.1 Primitive Data-Type:

A primitive data type is also called as basic data-type or built-in data type or simple data-type. The primitive data-type is a data type for which the programming language provides built-in support; i.e. you can directly declare and use variables of these kinds. You need not to define these data-types before use. So we can also say that primitive data-type is data type that is predefined. These primitive data types may be different for different programming languages. For example, C programming language provides built-in support for integers (int, long), reals (float, double) and characters (char).

3.2 Abstract Data-Type:

In computing, an abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data; and this is organized in such a way that the specification of values and operations on those values are separated from the representation of the values and the implementation of the operations. For example, consider 'list' abstract data type. The primitive operations on a list may include adding new elements, deleting elements, determining number of elements in the list etc. Here, we are not concerned with how a list is represented and how the above-mentioned operations are implemented. We only need to know that it is a list whose elements are of given type, and what can we do with the list.

3.3 Polymorphic Data-types:

A heterogeneous list is one that contains data element of variety of data types. It is desirable to create a data type that is independent of the values stored in the list. This kind of data type is known as polymorphic data types.

4. Data Structure Defined:

In computer science, a **data structure** is a particular way of organizing data in a computer so that it can be used efficiently.

The study of data structures includes:

* Logical description of data structures.

* Implementation of data structures.

* Quantitative analysis of the data structures.

5.Description of various Data Structures:

The various data structures are divided into following categories:

Linear Data-Structures:

A data structure whose elements form a sequence, and every element in the structure has a unique predecessor and unique successor. Examples of linear data structures are arrays, link-lists, stacks and queues.

Non-linear Data-Structures:

A data structure whose elements do not form a sequence, there is no unique predecessor or unique successor. Examples of non-linear data structures are trees and graphs.

5.1. Arrays:

An array is a collection of variables of the same type that are referred to by a common name. Arrays offer a convenient means of grouping together several related variables, in one dimension or more dimensions:

• product part numbers:

```
int part_numbers[] = {123, 326, 178, 1209};
```

One-Dimensional Arrays:

A one-dimensional array is a list of related variables. The general form of a one-dimensional array declaration is:

```
type variable_name[size]
```

• **type**: base type of the array,determines the data type of each element in the array

• **size**: how many elements the array will hold

• **variable_name**: the name of the array

Examples:

```
int sample[10];

float float_numbers[100];
```

char last_name[40];

Two-Dimensional Arrays:

A two-dimensional array is a list of one-dimensional arrays.To declare a two-dimensional integer array two_dim of size 10,20 we would write:
int matrix[3][4];

Multidimensional Arrays:

C++ allows arrays with more than two dimensions.

The general form of an N-dimensional array declaration is:

type array_name [size_1] [size_2] ... [size_N];

For example, the following declaration creates a 4 x 10 x 20 character array, or a matrix of strings:

char string_matrix[4][10][20];

This requires 4 * 10 * 20 = 800 bytes.

If we scale the matrix by 10, i.e. to a 40 x 100 x 20 array, then 80,000 bytes are needed.

5.2 Linked List:

A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence**.**

A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Singly linked list

Singly linked lists contain nodes which have a data field as well as a ***next*** field, which points to the next node in line of nodes.

A singly linked list whose nodes contain two fields: an integer value and a link to the next node

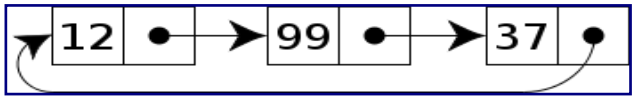
Doubly linked list

In a **doubly linked list**, each node contains, besides the next-node link, a second link field pointing to the *previous* node in the sequence. The two links may be called **forward**(**s**) and **backwards**, or **next** and **prev**(**previous**).

A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

Circular list

In the last node of a list, the link field often contains a null reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be 'circular' or 'circularly linked'; otherwise it is said to be 'open' or 'linear'.



A circular linked list

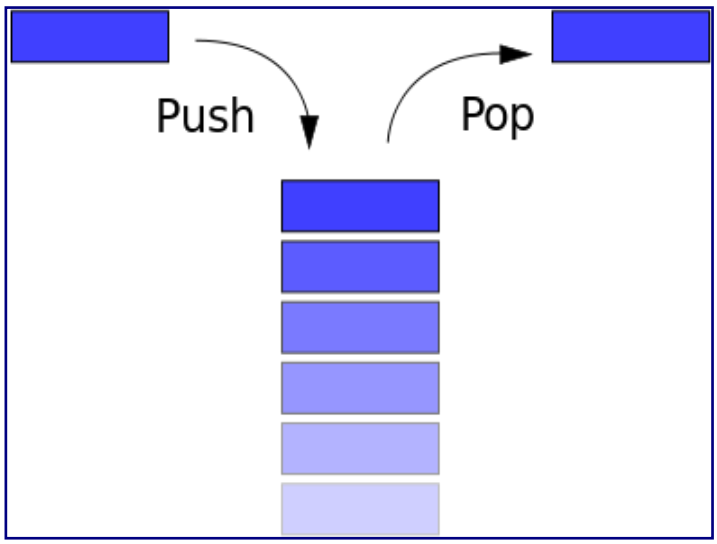
In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

5.3 Stack:

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

The basic implementation of a stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data, since as we will see there are various variations of stack implementations.

There are basically three operations that can be performed on stacks . They are 1) inserting an item into a stack (push). 2) deleting an item from the stack (pop). 3) displaying the contents of the stack(pip).



5.4 Queues:

A queue is a basic data structure that is used throughout programming. You can think of it as a

line in a grocery store. The first one in the line is the first one to be served. Just like a queue. A queue is also called a FIFO (First In First Out) to demonstrate the way it accesses data.

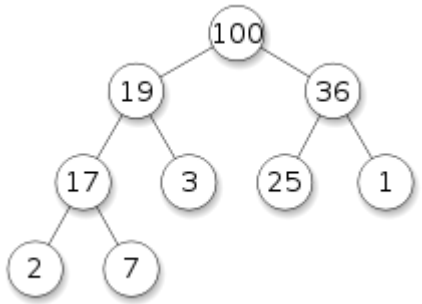
5.5 Trees:

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

A binary tree is a tree data structure in which each node has at most two children (referred to as the *left* child and the *right* child). In a binary tree, the *degree* of each node can be at most two. Binary trees are used to implement binary search trees and binary heaps, and are used for efficient searching and sorting.

5.6 Heaps:

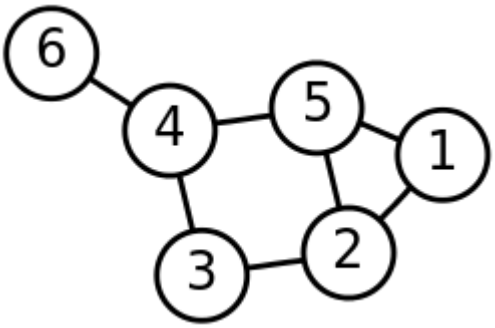
A heap is a specialized tree-based data structure that satisfies the *heap property:* If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. Either the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node (this kind of heap is called *max heap*) or the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node (*min heap*).



max heap

5.7 Graphs:

A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices. As in mathematics, an edge (*x*,*y*) is said to point or go from *x* to *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references**.**

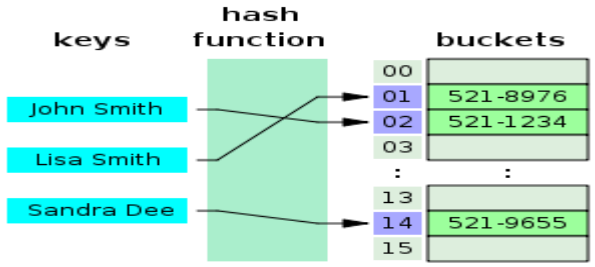


A labeled graph of 6 vertices and 7 edges.

5.8 Hash Table:

A hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the correct value can be found.

Ideally, the hash function will assign each key to a unique bucket, but this situation is rarely achievable in practice (usually some keys will hash to the same bucket). Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.



A small phone book as a hash table.

6.Common Operation on data structures:

The following the main operations that can be performed on the data structures :

1. **Traversing :** It means reading and processing the each and every element of a data structure at least once.
2. **Inserting :** It means inserting a value at a specified position in a data structure, this is also know as insertion.
3. **Deletion :** It means deleting a particular value from a specified position in a data structure.
4. **Searching :** It means searching a particular data in created data structure.
5. **Sorting :** It means arranging the elements of a data structure in a sequential manner i.e. either in ascending order or in descending order.
6. **Merging**: Combining the elements of two similar sorted structures into a single structure.

7.Program Development Life Cycle:

1. Analyze the problem Precisely define the problem to be solved, and write program specifications – descriptions of the program's inputs, processing, outputs, and user interface.

2. Design the program Develop a detailed logic plan using a tool such as pseudocode, flowcharts, object structure diagrams, or event diagrams to group the program's activities into modules; devise a method of solution or algorithm for each module; and test the solution algorithms.

3. Code the program Translate the design into an application using a programming lanaguage or application development tool by creating the user interface and writing code; include internal documentation – comments and remarks within the code that explain the purpose of code statements.

4 .Test and debug the program Test the program, finding and correcting errors (debugging) until it is error free and contains enough safeguards to ensure the desired results.

5. Formalize the solution Review and, if necessary, revise internal documentation; formalise and complete end-user (external) documentation

6. Maintain the program Provide education and support to end users; correct any unanticipated errors that emerge and identify user-requested modifications (enhancements). Once errors or enhancements are identified, the program development life cycle begins again at Step one.

8.Introduction to Algorithms:

An algorithm is the step-by-step solution to a certain problem. An algorithm can be expressed in English like language, called 'pseudocode', in a programming language, or in the form of a 'flowchart'. Every algorithm must satisfy the following criteria:

Input:There are zero or more values, which are externally supplied.

Output: At least one value is produced.

Definiteness: Each step must be clear and unambiguous.

Finiteness: If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.

Effectiveness: Each step must be sufficiently basic that it can in principle be carried out by a person using only paper and pencil. We use algorithms every day. For example, a recipe for baking a cake is an algorithm. Most programs, with the exception of some artificial intelligence applications, consist of algorithms.

8.1. A Typical Example:

Simple Example of algorithm:

One of the simplest algorithms is to find the largest number out of given three number. Then we can write an algorithm like this:

Algorithm MaxFind:

Let a, b and c be three integer numbers. This algorithm will find the maximum numbers out of these three.

Step 1: Begin

Step 2:read a, b, c

Step 3:If (a > b) then

Step 3.1 :if (a > c) then

Step 3.1.1:write: a +" is largest"

Step 3.1.2:else

Step 3.1.3:write: c + "is largest"

Step 3.1.4:end if

Step 3.2 :else

Step 3.2.1:if (b > c) then

Step 3.2.2:write: b + "is largest"

Step 3.2.3:else

Step 3.2.4:write: c + "is largest"

Step 3.2.5:end if

Step 3.3 :end if

Step 4: END.

8.2. Algorithm Description:

As we know, algorithm is just step by step solution of a given problem written in simple English. But the standard written algorithms follow some convention. It must be noted that an efficient algorithm is one, which is capable of giving the solution of the problem using minimum resources of the system such as memory used and processor's time. The format for presentation of the algorithms is language free, well structured and detailed. It will enable the readers to translate it into a computer program using any high-level language such as FORTRAN, Pascal or C/C++.

The format for the formal presentation of the algorithm consists of two parts:

• The first part describes the input data, the purpose of the algorithm and identifies the variables used in the algorithms.

• The second part is composed of sequence of instruction that lead to the solution of the problem.

The following description summarizes certain conventions used in presenting the above algorithm.

Comments:

Each instruction may be followed by a comment. The comments begin with a double slash, and explain the purpose of the instruction, such as:

// this is sample comment

Appropriate use of comments enhances the readability of the algorithm, which in turn helps in maintaining the algorithm.

Variable Names:

For variable names, we can use any descriptive names like max, loc, etc. For variable names, we will use lowercase letters such as 'max', 'loc' etc. whereas for defined constants, if any, we will use uppercase letters.

Assignment Statement:

The assignment statement will use the notation as

Set max := a[i]

to assign the value of a[i] to max. The right hand side of the assignment statement can have a value, a variable or an expression.

Input/Output:

Data may be inputted and assigned to variables by means of a 'read' statement with the following format:

read: variable-list

Where the variable-list consists one or more variables separated by comma.Similarly, the data held by the variables and the messages, if any, enclosed in double quotes can be output by means of a write statement with the following

format:

write: message and/or variable-list (or)

print: message and/or variable list

where the message and the variables in the variable-list are separated by comma.

Execution of Instructions:

The instructions in the algorithm are usually executed one after the other as they appear in the algorithm. However, there may be instances when some instructions are skipped or some instructions may be repeated as a result of certain conditions.

Completion of the Algorithm:

The algorithm is completed with the execution of the last instruction. However, the algorithm can be terminated at any intermediate state using the exit instruction.With the help of these conventions, one can write the algorithm easily in the standard fashion.

9. **Structured Programming Constructs:**

In structured programming, the program is divided into small parts (functions) and each part performs a specific job. The main importance is on functions rather than data. The same data can be used and manipulated by several functions and such type of data is made global.

Structured Programming Constructs:

1. Sequence Control Structure (Sequence Logic or Sequential Flow).

2. Selection Control Structure (Selection Logic or Conditional Flow).
3. Repetition Control Structure (Iteration Logic or Repetitive Flow).

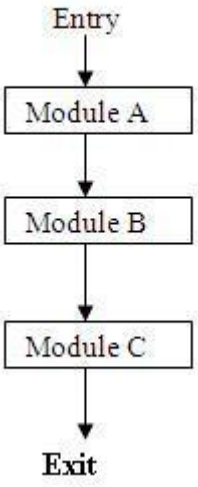
1. **Sequence Control Structure (Sequence Logic or Sequential Flow)**

If nothing is specified then the modules the executed in the sequence in which they are written. Thus in it the modules are executed one after the other.
The sequence control structure is shown below:

Algorithm

.
.
.
Module A
Module B
Module C
.
.

Flowchart



.

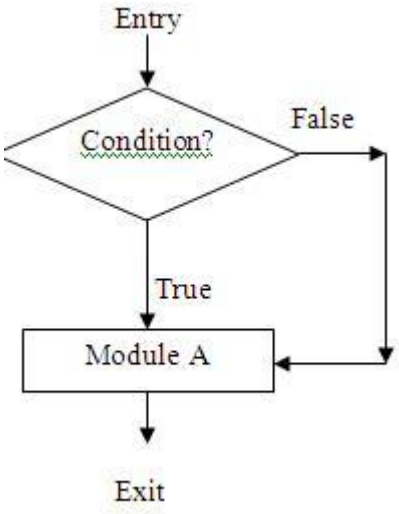
2. Selection Control Structure (Selection Logic or Conditional Flow)

In it there are various conditions on the basis of which one module is selected out of several given modules. The structures which implement this logic are called conditional structures. The conditional structures can be divided into following categories:

1. **Single Alternative:** This structure has the following form:

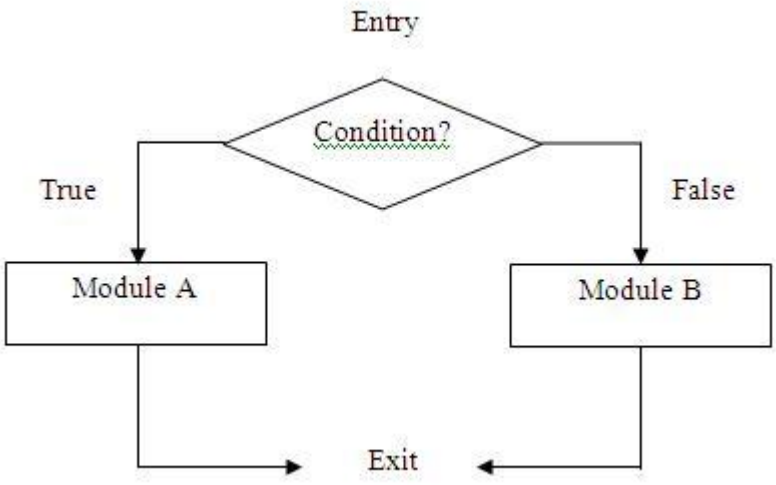
If condition, then:
[Module A]
[End of If structure]

According to this if the condition is true then module A is executed. Otherwise, module A is skipped. Its flowchart is:



2. Double Alternative: This structure has the following form:

If condition, then:
    [Module A]
Else:
    [Module B]
[End of If structure]

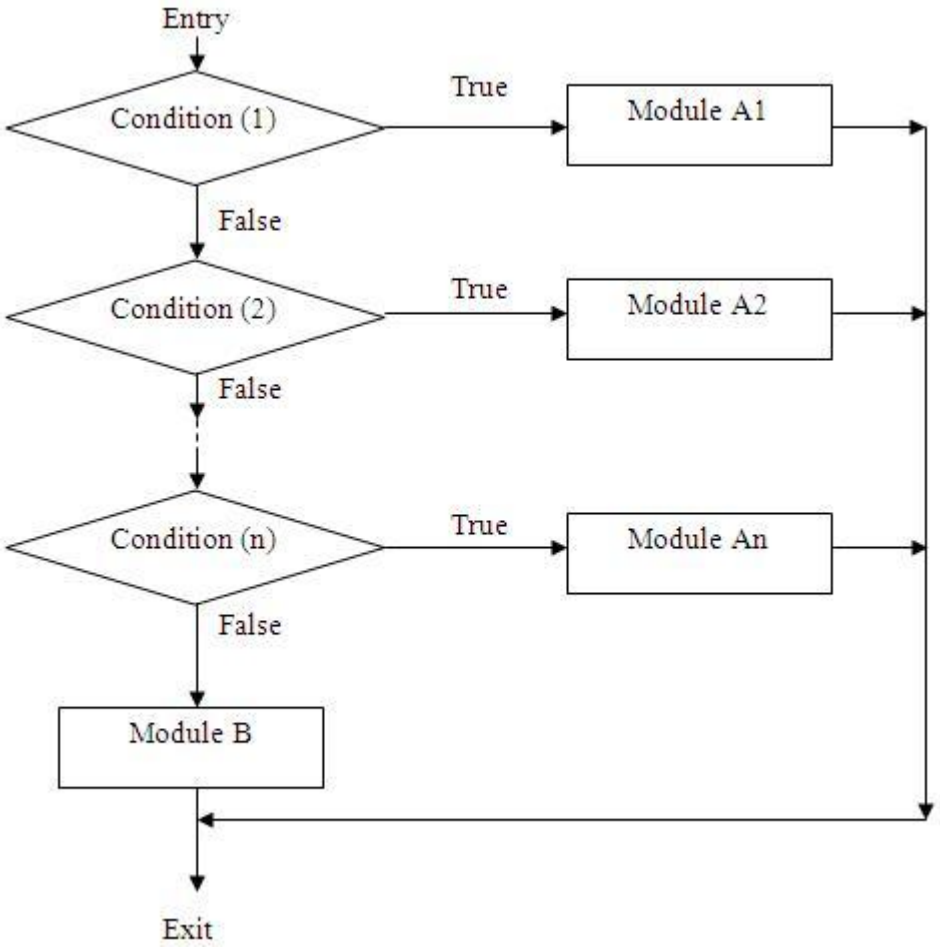


3. Multiple Alternative: This structure has the following form:

If condition (1), then:
    [Module A1]

Else If condition (2), then:
    [Module A2]
    .
    .
    .
Else If condition (n), then:
    [Module An]
Else:
    [Module B]
[End of If structure]

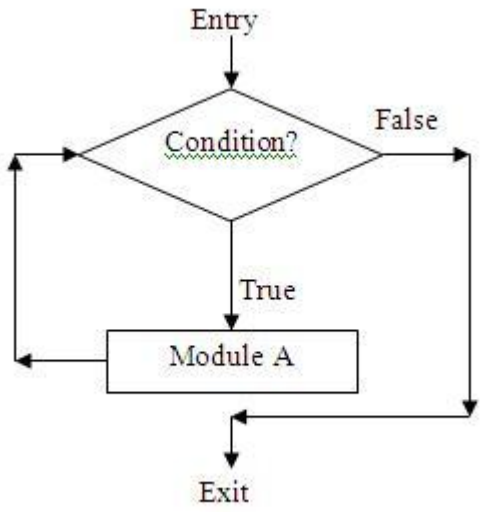
According to this only one of the modules will be executed. If condition 1 is true then module A1 will be executed. Otherwise condition 2 will be checked. If condition 2 is true then module A2 will be executed. And so on. If none of the conditions is true then module B will be executed. Its flowchart is:



4. Repetition Control Structure (Iteration Logic or Repetitive Flow): In it loops are implemented. Loop is used to implement those statements which are to be repeated again and again until some condition is satisfied. It implements while and for loops. Each of these begin with Repeat statement. For example is case of Repeat-while structure:

Repeat steps 1 to n while condition:
[Steps]
[End of Repeat-while structure]

Its flowchart is:



10.Algorithm Complexity:

As an algorithm is a sequence of steps to solve a problem, there may be more than one algorithm to solve a problem. So we have to choose one algorithm as a solution. Also we would like to choose best algorithm in terms of resources used. This needs to analyze each algorithm deeply. Analyzing an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it.

Algorithm analysis helps us to choose the best algorithm for a given application. Two main factors on which the performance of a program depends are the amount of computer memory consumed and the time required to execute it successfully. There are two ways in which we can analyze the performance of an algorithm. One of them is to in carry out experiments with the algorithm by actually executing it and recording the space and time required. This method can be used only after successful implementation.

Another method, which can be used, is the analytical method. We can approximately find out the space and time required before implementation. Implementing each algorithm and then recording their complexity is not a practical way, as there may be thousands of solution (so, algorithms) of a given problem. So, algorithm should be compared at the pseudocode stage, i.e. by analytical method.

The choice of particular algorithm depends on following considerations:

- Ø Performance requirements, i.e., time complexity
- Ø Memory requirements, i.e., space complexity
- Ø Programming requirements

Since programming requirements are difficult to analyze precisely, complexity theory concentrate on performance and memory requirements. Performance requirements are usually more

critical than memory requirements; hence, in general, it is not necessary to worry about memory unless they grow faster than performance requirements. Therefore, in general, the algorithms are analyzed only on the basis of performance requirements, i.e., running-time efficiency.

Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run to completion. Some of the reasons for studying space complexity are:

- If the program is to run on multi-user system, it may be required to specify the amount of memory to be allocated to the program.
- We may be interested to know in advance that whether sufficient memory is available to run the program.
- There may be several possible solutions with different space requirements.
- Can be used to estimate the size of the largest problem that a program can solve.

In general, the total space needed by a program can be divided in two parts:

1. **A static part**, which is independent of the instance characteristics. This includes the space required to store the code. Again, the space required to store the code is compiler and machine dependent. Other components are constants, variables, complex data types etc.
2. **A dynamic part**, which consists of components whose memory requirements,depends on the instance of the problem being solved. Dynamic memory allocations and recursion are few components of this type. The factors, which can help us in determining the size of the problem instance, are number of inputs, outputs etc.

Time Complexity:

Time complexity of a program is the amount of time required to execute successfully. Some of the reasons for studying time complexity are:

- We may be interested to know in advance that whether the program will provide a satisfactory real-time response. For example, an interactive program, such as an editor, must provide such a response. If it takes even a few seconds to move the cursor one page up or down, it will not be acceptable to the user.
- There may be several possible solutions with different time requirements.

To measure the time complexity accurately, we can count the all sort of operations performed in an algorithm. If we know the time for each one of the primitive operations performed on a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from system to system.

A more acceptable approach will be to estimate the execution time of an algorithm irrespective of the computer on which it will be used. Hence, the more reasonable approach is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm The time complexity can now be expressed as a function of a number of key operations performed.

Time-space Trade-off:

The best algorithm, hence best program, 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. Also, there may be more than one approach to solve a same problem. One such approach may require more space but takes less lime to complete its execution, while the other approach requires less space but takes more time to complete its execution. Thus,we may have to sacrifice one at the cost of the other. That is what we can say that there exists a time-space trade among algorithms.

Therefore, if space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand, if time is our constraints such as in real-time systems, we have to choose a program that takes less time to complete its execution at the cost of more space.

In the analysis of algorithms, we are interested in the 'average case', the amount of time a program might be expected to take on typical input data, and in the 'worst case', the amount of time a program would take on the worst possible input configuration.

Expressing Space and Time Complexity:

The space and/ or time complexity is usually expressed in the asymptotic notation. The asymptotic notation is nothing but to assume the value of a function. In this notation the complexity is usually expressed in the form of a function f(n), where 'n' is the input size for a given instance of the problem being solved. Expressing space and/or time complexity as a function f(n) is important because of following reasons:

• We may be interested to predict the rate of growth of complexity as the size of the problem increases.

• To compare the complexities of two or more algorithms solving the same problem in order to find which is more efficient. Time and space complexity is measured in terms of asymptotic notations. For example, let us consider a program which stores 'n' elements:

```
void store()
{
int i, n;
printf("Enter the number of elements ");
scanf("%d", &n);
for( i=0, i<n, i++)
{
//store the elements
}
}
```

In the above program space is required to store the executable code and to store the 'n' number of elements. The memory that is required to store the executable code is static. The memory required to store the 'n' elements depends on the value of 'n'. The time required to execute the code also depends on the value of 'n'. In the above program we have two executable statements printf and scanf. Let us assume there are 'x' statements in the for loop. Then the time required to execute the program will be equal to x*n+2. 'n' is the instance characteristics. At last this calculated time can be expressed in one of the asymptotic notation in a function form f(n).

The most commonly used asymptotic notations are: Big Oh (O), Big Omega (Ω), and

Big Theta (Θ).In these notations we use some terms like upper bound, lower bound etc. Upper bound will give the maximum time or space required for a program. Lower bound will give the minimum time/ space required. The most important notation used to express this function 'f(n)' is Big Oh notation, which provides the upper bound for the complexity, and is described in next section.Since in modern computers, the memory is not a severe constraint; therefore, our analysis of algorithms will be on the basis of time complexity.

11.Big Oh Notation:

The algorithm complexity can be determined ignoring the implementation dependent factors. This is done by eliminating constant factors in the analysis of the algorithms. Basically, these are the constant factors that differ from computer to computer. Clearly, the complexity function f(n) of an algorithm increases as 'n'increases. It is the rate of 'f(n)' that we want to examine.

Big-O is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

Based on Big Oh notation, the algorithms can be categorized as follows:

• Constant time (O(1)) algorithms

• Logarithmic time (O(log2n)) algorithms

• Linear time (O(n)) algorithms

• Polynomial time (O(nk), for k>1) algorithms

• Exponential time (O(kn), for k>1) algorithms

Many algorithms are of O(n log2n)

n \ f(n)	$\log_2 n$	n	n^2	n^3	2^n	$n\log_2 n$
5	3	5	25	125	32	15
10	4	10	100	10^3	10^3	40
100	7	100	10^4	10^6	10^{30}	700
1000	10	10^3	10^6	10^9	10^{300}	10^4

Rate of growth of some standard functions

Observe that the logarithmic function log2n grows most slowly, whereas the exponential function 2n grows most rapidly, and the polynomial function nk growsaccording to the exponent 'k'.

Limitations of Big-Oh Notation:

Big Oh notation has two basic limitations:

- It contains no consideration of programming efforts
- It masks (hides) potentially important constants.

As an example of later limitation, imagine two algorithms, one using 500000n2 time, and the other n3 time. The first algorithm is O(n2), which implies that it will take less time than the other algorithm which is O(n3). However, the second algorithm will be faster for n<500000, and this would be faster for many applications.

12.Arrays & Matrices:

Introduction:

An array is a data structure. It is a collection of similar type of (homogeneous) data elements and is represented by a single name.

It has the following features:

1. The elements are stored in continuous memory locations.
2. The n elements are numbered by consecutive numbers i.e. 1, 2, 3, ... , n.

E.g.

An array STUDENT containing 8 records is shown below:

STUDENT

Ritika	1
Gurpreet	2
Anupama	3
Hanish	4
Harsh	5
Navdeep	6
Shalini	7
Kapil	

13.Linear Arrays:

The simplest type of data structure is a linear array. This is also called one-dimensional array. In computer science, an **array data structure** or simply an **array** is a data structure consisting of a collection of *elements* (values or variables), each identified by at least one *array index* or *key*. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula.

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

Because the mathematical concept of a matrix can be represented as a two-dimensional grid, two-dimensional arrays are also sometimes called matrices. In some cases the term "vector" is used in computing to refer to an array, although tuples rather than vectors are more correctly the mathematical equivalent. Arrays are often used to implement tables, especially lookup tables; the word *table* is sometimes used as a synonym of *array*.

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

Arrays are useful mostly because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation. The set of valid index tuples and the addresses of the elements (and hence the element addressing formula) are usually,[3][5] but not always,[2] fixed while the array is in use.

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

The term is also used, especially in the description of algorithms, to mean associative array or "abstract array", a theoretical computer science model (an abstract data type or ADT) intended to capture the essential properties of array.

14.Two dimensional Arrays:

Implementing a database of information as a **collection** of arrays can be inconvenient when we have to pass many arrays to utility functions to process the database. It would be nice to have a single data structure which can hold all the information, and pass it all at once.

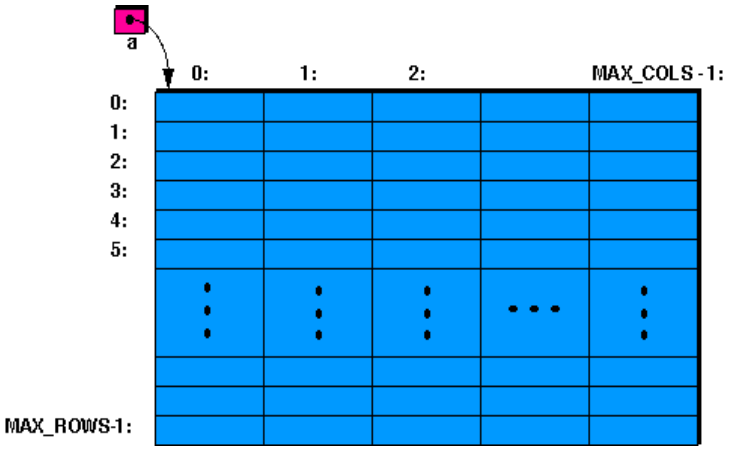
2-dimensional arrays provide most of this capability. Like a 1D array, a 2D array is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.

How do we declare a 2D array?

Similar to the 1D array, we must specify the data type, the name, and the size of the array. But the size of the array is described as the number of rows and number of columns. For example:

```
int a[MAX_ROWS][MAX_COLS];
```

This declares a data structure that looks like:



How do we access data in a 2D array?

Like 1D arrays, we can access individual cells in a 2D array by using subscripting expressions giving the indexes, only now we have two indexes for a cell: its row index and its column index. The expressions look like:

```
a[i][j] = 0;    or    x = a[row][col];
```

We can initialize all elements of an array to 0 like:

```
for(i = 0; i < MAX_ROWS; i++)
    for(j = 0; j < MAX_COLS; j++)
        a[i][j] = 0;
```

15.Matrices:

A *matrix* is a rectangular array of numbers or other mathematical objects, for which operations such as addition and multiplication are defined Most commonly, a matrix over a field *F* is a rectangular array of scalars from *F*.Most of this article focuses on *real* and *complex matrices*, i.e., matrices whose elements are real numbers or complex numbers, respectively. More general types of entries are discussed below. For instance, this is a real matrix:

$$\mathbf{A} = \begin{bmatrix} -1.3 & 0.6 \\ 20.4 & 5.5 \\ 9.7 & -6.2 \end{bmatrix}.$$

The numbers, symbols or expressions in the matrix are called its *entries* or its *elements*. The horizontal and vertical lines of entries in a matrix are called *rows* and *columns*, respectively.

16.Special Matrices:

Triangular Matrices

Definition 1 Given an n × n matrix A

- A is called upper triangular if all entries below the main diagonal are 0.
- A is called lower triangular if all entries above the main diagonal are 0.
- A is called diagonal if only the diagonal entries are non-zero.

If D is a diagonal matrix with diagonal entries d1 , d2 , . . . dn , we may write it as diag(d1 , d2 , . . . , dn)

1. Upper Triangular
2. Lower Triangular
3. Diagonal

1. A matrix in REF is upper triangular.
2. The transpose of an upper triangular matrix is lower triangular and visa versa.
3. The product of two Upper triangular matricies is upper triangular.
4. The product of two Lower triangular matricies is Lower triangular.
5. The product of two Diagonal matricies is Diagonal.
6. The transpose of a Diagonal matrix is Diagonal.

A diagonal, upper or lower triangular matrix is invertable if and only if its diagonal entries are all non-zero.

17.Sparse Matrices:

In numerical analysis, a **sparse matrix** is a matrix populated primarily with zeros as elements of the table. By contrast, if a larger number of elements differ from zero, then it is common to refer to the matrix as a **dense matrix**. The fraction of zero elements (non-zero elements) in a matrix is called the **sparsity** (**density**).

Conceptually, sparsity corresponds to systems which are loosely coupled. Consider a line of balls connected by springs from one to the next; this is a sparse system. By contrast, if the same line of balls had springs connecting each ball to all other balls, the system would be represented by a **dense matrix**. The concept of sparsity is useful in combinatorics and application areas such as network theory, which have a low density of significant data or connections.

Huge sparse matrices often appear in science or engineering when solving partial differential equations. When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard dense-matrix structures and algorithms are relatively slow and consume large amounts of memory when applied to large sparse matrices. Sparse data is by nature easily compressed, and this compression almost always results in significantly less computer data storage usage. Indeed, some very large sparse matrices are infeasible to manipulate using standard dense algorithms.

Example of sparse matrix

[11 22 0 0 0 0 0]
[0 33 44 0 0 0 0]
[0 0 55 66 77 0 0]
[0 0 0 0 0 88 0]
[0 0 0 0 0 0 99]

The above sparse matrix contains only 9 nonzero elements of the 35, with 26 of those elements as zero

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

POSSIBLE QUESTIONS

UNIT-I

2 MARKS:

1. Write about the Basic Terminology of Data Structures?
2. Define Array with example.
3. Define Data Structure.
4. Define Stack..
5. What is a Queue.

6 MARKS:

1. Define Data Structure. Explain in detail about various data structures.
2. Explain about Single and Multidimensional array with example.
3. Define Sparse Matrix and how it is represented in array and Linked List.
4. Elaborate about Prefix, Infix and Postfix Expressions with example.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject : Data Structures
Class: II BCA **Subject code: 17CAU301**
Objective Type Questions

UNIT I

S.No	QUESTION	OPT 1	OPT 2	OPT 3	OPT 4	ANSWER
1	The logical or mathematical model of a particular data organization is called as_____	Data Structure	Software Engineering	Data Mining	Data Ware Housing	Data Structure
2	An algorithms _____ is measured in terms of computing time ad space consumed by it.	performance	effectiveness	finiteness	definiteness	performance
3	Which of the following is not structured data type?	Arrays	Union.	Queue	Linked list.	Union.
4	Which of the following is a valid non - linear data structure.	Stacks	Trees	Queues	Linked list.	Trees
5	Combining elements of two _____ data structure into one is called Merging	Similar	Dissimilar	Even	Un Even	Similar
6	Data structures are classified as _____ data type.	User Defined	Abstract	Primitive & Non Primitive	predefined only	Primitive & Non Primitive
7	_____ are the commonl used ordered list.	Graphs	Trees	Stack and Queues	List	Stack and Queues
8	Data structure can be classified as _____ data type based on relationship with complex data element.	Linear & Non Linear	Linear	Non Linear	None of the above	Linear & Non Linear
9	A data structure whose elements forms a sequence of ordered list is called as _____ data structure.	Non Linear	Linear.	Primitive	Non Primitive	Linear.

10	A data structure which represents hierarchical relationship between the elements are called as _____ data structure.	Linear	Primitive.	Non Linear	Non Primitive	Non Linear
11	A data structure, which is not composed of other data structure, is called as _____ data structure.	Linear	Non Primitive	Non Linear	Primitive	Primitive
12	Data structures, which are constructed from one or more primitive data structure, are called as _____ data structure.	Non Primitive	Primitive.	Non Linear	Linear	Non Primitive
13	_____ is the term that refers to the kinds of data that variables may hold in a programming language.	data type	data structure	data Object	data	data type
14	The _____ model of a particular data organization is called as Data Structure.	software Engineering	logical or mathematical	Data Mining	Data Ware Housing	logical or mathematical
15	The design approach where the main tast is decomposed into subtasks and each subtask is further decomposed into simpler solutions is called _____	top down approach	bottom up approach	hierarchical approach	merging approach	top down approach
16	_____ is a sequence of instructions to accomplish a particular task	Data Strucuture	Algorithm	Ordered List	Queue	Algorithm
17	_____ criteria of an algorithm ensures that the algorithm terminate after a particular number of steps.	effectiveness	finiteness	definiteness	particular	finiteness
18	An algorithm must produce _____ output(s)	many	only one	atleast one	zero or more	atleast one
19	_____ criteria of an algorithm ensures that the algorithm must be feasible.	effectiveness	finiteness	definiteness	infinite	effectiveness
20	_____ criteria of an algorithm ensures that each step of the algorithm must be clear and unambiguous.	effectiveness	finiteness	definiteness	infinite	definiteness

21	The time factor whrn determining the efficiency of the algorithm is measured by	counting micro seconds	counting the number of key operatiuons	counting the number of statements	counting the kilobyte of the algorithm	counting the number of key operatiuons
22	Which of the following data strucutre is linear data structure	Trees	Graphs	Arrays	Union	Arrays
23	The operation of processing each element in a list is called	sorting	merging	Inserting	Traversal	Traversal
24	Finding the location of the element with a given value is	Traversal	Search	Sort	Merging	Search
25	Which of the following data structure are indexed structures?	Linear array	Linked list	Stack and Queues	queue	Linear array
26	Size of the int data type is	2 byte	4 byte	compiler dependent	varies all the time	compiler dependent
27	Each array declaration need not give, implicitely or explicitely the information about	name of array	data type of array	first data from the set to be stored	index set of array	first data from the set to be stored
28	Which of the following data strucutre cannot store the non-homogeneous data elements?	Arrays	Records	Pointers	Union	Arrays
29	Which of the following data strucutre can only store the homogeneous data elements?	Union	Arrays	Pointers	Strucutres	Arrays
30	Which of the following data strucutre is linear type?	Strings	Trees	Graph	B Tree	Strings
31	An algorithm that directly calls itself is called	Sub algorithm	Recursion	polish notation	traversal algorithm	Recursion
32	What term is used to describe O(N) algorithm?	constant	linear	logarithmic	quadratic	linear
33	Which of these is the correct Big O expression for 1 + 2 + 3 + …. + n ?	O(log n)	O(n log n)	O(n)	O(n2)	O(n2)
34	Which of these is the correct Big O expression for 35n + 6?	O(log n)	O(n log n)	O(n)	O(n2)	O(n)
35	Find out the complexity of x = 3*y + 2; z=z+1;	O(log n)	O(n log n)	O(n)	O(1)	O(1)

36	Representation of data structure in memory is known as:	recursive	abstract data type	storage structure	file structure	abstract data type
37	If the address of A[1][1] and A[2][1] are 1000 and 1010 respectively and each element occupies 2 bytes then the array has been stored in _____ order.	row major	column major	matrix major	tuple major	row major
38	When the maximum entries of (m*n) matrix are zeros then it is called as _____.	Transpose matrix	Sparse Matrix	Inverse Matrix	tridiagonal matrix	Sparse Matrix
39	A matrix of the form (row, col, n) is otherwise known as _____.	Transpose matrix	Inverse Matrix	Sparse Matrix	Diagonal matrix	Sparse Matrix
40	A list of finite number of homogeneous data elements are called as _____	Stacks	Records	Arrays	Linked list.	Arrays
41	No of elements in an array is called the _____ of an array.	Structure	Height	Width	Length.	Length.
42	The size or length of an array = _____.	UB – LB + 1	LB + 1	UB - LB	UB – 1	UB – LB + 1
43	Searching is the Process of finding the _____ of the element with the given value or a record with the given key.	Place	Location	Value	Operand	Location
44	Length of an array is defined as _____ of elements in it.	Structure	Height	Size	Number	Number
45	_____ is a set of pairs, index and value.	stack	queue	Arrays	Set	Arrays
46						
47	Sum of terms of the form ax^e is called _____.	Array	Matrix	Expression	Polynomial	Polynomial
48	_____is a collection of data and links.	Links	Node	List	Item	Node
49	Each item in a node is called a_____.	Field	Data item	Pointer	Data	Field
50	The elements in the list are stored in a one dimensional array called a _____	Value	List	Data	Link	Data

51	Data movement and displacing the pointers of the Queue are tedious proplems in _____ representation of a Queue.	Array	Linked	Circular	lenear list	Array
52	Solving different parts of a program directly and combining these pieces into a complete program is called _____	top down approach	bottom up approach	hierarchical approach	merging approach	bottom up approach
53	The number of times a statement in a program is executed is called its _____	Piori estimate	Posteriori estimate	Frequency count	Program count	Frequency count
54	_____ means the computing time of the algorithm is constant	O(log n)	O(n)	O(n log n)	O(1)	O(1)
55	_____ means the computing time of the algorithm is linear	O(log n)	O(n)	O(n log n)	O(1)	O(n)
56	Algorithms with time complexity $O(n^2)$ are called _____	linear	exponential	quadratic	cubic	quadratic
57	For large data set algorithms with complexity greater than _____ are impractical	O(log n)	O(n)	O(n log n)	O(1)	O(n log n)
58	Which of the following case does not exist in complexity theory?	Best case	Worst case	Average case	Null case	Null case
59	Two main measures for the efficiency of the algorithm are	Processor and memory	Complexity and capacity	Time and space	Data and space	Time and space

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

UNIT-II

Linked Lists Singly, Doubly and Circular Lists (Array and Linked representation); Normal and Circular, representation of Stack in Lists; Self Organizing Lists; Skip Lists Queues, Array and Linked representation of Queue, De-queue, Priority Queues

Linked list:

1.Introduction:

A linked list is a data structure which can change during execution.

– Successive elements are connected by pointers.

– Last element points to NULL head

– It can grow or shrink in size during execution of a program.

– It can be made just as long as required.

– It does not waste memory space.

Keeping track of a linked list:

– Must know the pointer to the first element of the list (called start, head, etc.).

• Linked lists provide flexibility in allowing the items to be rearranged efficiently.

– Insert an element.

– Delete an element.

For insertion:

– A record is created holding the new item.

– The next pointer of the new record is set to link it to the item which is to follow it in the list.

– The next pointer of the item which is to precede it must be modified to point to the new item.

For deletion:

– The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

2.Implementing Lists:

2.1.Implementing Lists using Arrays:

Arrays are suitable for:

– Inserting/deleting an element at the end.

– Randomly accessing any element.

– Searching the list for a particular value.

2.2.Implementing Lists using Linked List:

Linked lists are suitable for:

Inserting an element.

Deleting an element.

Applications where sequential access is required.In situations where the number of elements cannot be predicted beforehand.

3.Linked List Defined:

Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

– Linear singly-linked list (or simply linear list).

Circular linked list:
• The pointer from the last element in the list points back to the first element.

Doubly linked list:

• Pointers exist between adjacent nodes in both directions.

• The list can be traversed either forward or backward.

• Usually two pointers are maintained to keep track of the list, head and tail.

Basic Operations on a List:

Creating a list

Traversing the list

Inserting an item in the list

Deleting an item from the list

Concatenating two lists into one

Example: Working with linked list

• Consider the structure of a node as follows:

```
struct stud {
                int
                roll;
                 char name[25];
int
age;
struct stud *next;
};
/* A user-defined data type called “node” */
typedef struct stud node;
node *head;
```

Creating a List

```
node *create_list()
{
int k, n;
node *p, *head;
printf ("\n How many elements to enter?");
scanf ("%d", &n);
for
```

```
{
(k=0; k<n; k++)
if (k == 0) {
head = (node *) malloc(sizeof(node));
p = head;
}
else {
p->next = (node *) malloc(sizeof(node));
p = p->next;
}
scanf ("%d %s %d", &p->roll, p->name, &p->age);
}
p->next = NULL;
return (head);
}
```

To be called from main() function as:

```
node *head;
.....
head = create_list();
```

Traversing the List:

Once the linked list has been constructed and head points to the first node of the list,

– Follow the pointers.

– Display the contents of the nodes as they are traversed.

– Stop when the next pointer points to NULL.

```
void display (node *head)
{
int count = 1;
node *p;
p = head;
while (p != NULL)
{
printf ("\nNode %d: %d %s %d", count,
p->roll, p->name, p->age);
count++;
p = p->next;
}
printf ("\n");
}
```

To be called from main() function as:

```
node *head;
.....
display (head);
```

Inserting a Node in a List:

The problem is to insert a node before a specified node.

– Specified means some value is given for the node (called key).

– In this example, we consider it to be roll.

• Convention followed:

– If the value of roll is given as negative, the node will be inserted at the end of the list.

When a node is added at the beginning, Only one next pointer needs to be modified.

• head is made to point to the new node.

• New node points to the previously first element.

• When a node is added at the end,

– Two next pointers need to be modified.

• Last node now points to the new node.

• New node points to NULL.

• When a node is added in the middle,

– Two next pointers need to be modified.

• Previous node now points to the new node.

• New node points to the next node.

```
void insert (node **head)
{
int k = 0, rno;
node *p, *q, *new;
new = (node *) malloc(sizeof(node));
printf ("\nData to be inserted: ");
scanf ("%d %s %d", &new->roll, new->name, &new->age);
printf ("\nInsert before roll (-ve for end):");
scanf ("%d", &rno);
p = *head;
if (p->roll == rno)
{
new->next = p;
*head = new;
}
else
```

```
{
while ((p != NULL) && (p->roll != rno))
{
q = p;
p = p->next;
}
if
{
(p == NULL)
/* At the end */
q->next = new;
new->next = NULL;
}
else if
(p->roll
```

The pointers q and p always point to consecutive nodes.

```
== rno)
/* In the middle */
{
q->next = new;
new->next = p;
}
}
}
```

To be called from main() function as:

```
node *head;
.....
insert (&head);
```

Deleting a node from the list:

Here also we are required to delete a

specified node.

– Say, the node whose roll field is given.

• Here also three conditions arise:

– Deleting the first node.

– Deleting the last node.

– Deleting an intermediate node.

```
void delete (node **head)
{
int rno;
node *p, *q;
printf ("\nDelete for roll :");
scanf ("%d", &rno);
p = *head;
if (p->roll == rno)
/* Delete the first element */
{
*head = p->next;
free (p);
}
```

```
else
{
while ((p != NULL) && (p->roll != rno))
{
q = p;
p = p->next;
}
if
(p == NULL)
/* Element not found */
printf ("\nNo match :: deletion failed");
else if (p->roll == rno)
/* Delete any other element */
{
q->next = p->next;
free (p);
}
}
}
```

4.More Types of lists:

4.1 Circular Linked List

1. Circular Linked List is Divided into 2 Categories .
    - Singly Circular Linked List
    - Doubly Circular Linked List
2. In Circular Linked List Address field of Last node contain address of "First Node".
3. In short First Node and Last Nodes are adjacent .
4. Linked List is made circular by linking first and last node , so it looks like circular chain [shown in Following diagram].
5. Two way access is possible only if we are using "Doubly Circular Linked List"

6. Sequential movement is possible
7. No direct access is allowed.

4.2. Header Linked List:

The following steps are used to create linked list with header:

1. Three pointers, i.e. header, first and rear are declared. The header pointer is initialized with NULL. For example, header=NULL where, header is a pointer to structure. If it remains NULL, it implies that the list has no element. Such list is known as NULL list or empty list, which is shown in Fig. 6.12(a).

Figure 6.12(a). Empty list

2. In the second step, memory is allocated for the first node of the linked list. For example, the address of first node is 1888. An integer, say 3, is stored in the variable num and value of header is assigned to pointer next.

Figure 6.12(b). Link list

The address of first node is initialized by both the header and rear. The statement would be,

header=first;
rear=first;

3. The address of pointer first is assigned to pointers header and rear. The rear pointer is used to identify the end of the list and to detect the NULL pointer.
4. Again, create a memory location suppose 1890 for the successive node.

Figure 6.12(c).

5. Link the element of 1890 by assigning the value of node rear->next. Move the rear pointer to the last node.

5.Application of Linked Lists:

5.1 Polynomial manipulation:

Representing a polynomial using a linked list

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.

However, for any polynomial operation , such as addition or multiplication of polynomials , you will find that the linked list representation is more easier to deal with.

First of all note that in a polynomial all the terms may not be present, especially if it is going to be a very high order polynomial. Consider

5 x12 + 2 x9 + 4x7 + 6x5 + x2 + 12 x

Now this 12th order polynomial does not have all the 13 terms (including the constant term).

It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial.

Each node will need to store the variable x, the exponent and the coefficient for each term.It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial.

Thus we need to define a node structure to hold two integers , viz. exp and coff

Compare this representation with storing the same polynomial using an array structure.

In the array we have to have keep a slot for each exponent of x, thus if we have a polynomial of order 50 but containing just 6 terms, then a large number of entries will be zero in the array.

It would be also easy to manipulate a pair of polynomials if they are represented using linked lists.

Addition of two polynomials

Consider addition of the following polynomials

5 x12 + 2 x9 + 4x7 + 6x6 + x3

7 x8 + 2 x7 + 8x6 + 6x4 + 2x2 + 3 x + 40

The resulting polynomial is going to be

5 x12 + 2 x9 + 7 x8 + 6 x7 + 14x6 + 6x4 +x32x2 + 3 x + 40

Now notice how the addition was carried out. Let us say the result of addition is going to be stored in a third list. We started with the highest power in any polynomial. If there was no item having same exponent , we simply appended the term to the new list, and continued with the process.

Wherever we found that the exponents were matching, we simply added the coefficients and then stored the term in the new list. If one list gets exhausted earlier and the other list still contains some lower order terms, then simply append the remaining terms to the new list.

Now we are in a position to write our algorithm for adding two polynomials.

Let phead1 , phead2 and phead3 represent the pointers of the three lists under consideration.

Let each node contain two integers exp and coff .

Let us assume that the two linked lists already contain relevant data about the two polynomials.

Also assume that we have got a function append to insert a new node at the end of the given list.p1 = phead1;

p2 = phead2;

Let us call malloc to create a new node p3 to build the third list

p3 = phead3;

/* now traverse the lists till one list gets exhausted */

while ((p1 != NULL) || (p2 != NULL))

{

/ * if the exponent of p1 is higher than that of p2 then

the next term in final list is going to be the node of p1* /

while (p1 ->exp

> p2 -> exp)

{

p3 -> exp = p1 -> exp;

p3 -> coff = p1 -> coff ;

append (p3, phead3);

/* now move to the next term in list 1*/

p1 = p1 -> next;

}

/ * if p2 exponent turns out to be higher then make p3

same as p2 and append to final list * /while (p1 ->exp

< p2 -> exp)

```
{
p3 -> exp = p2 -> exp;
p3 -> coff = p2 -> coff ;
append (p3, phead3);
p2 = p2 -> next;
}
/* now consider the possibility that both exponents are
same , then we must add the coefficients to get the term for
the final list */
while (p1 ->exp
= p2 -> exp )
{
p3-> exp = p1-> exp;
p3->coff = p1->coff + p2-> coff ;
append (p3, phead3) ;
p1 = p1->next ;
p2 = p2->next ;
}
}
```

/* now consider the possibility that list2 gets exhausted , and there are terms remaining only in list1. So all those terms have to be appended to end of list3. However, you do not have to do it term by term, as p1 is already pointing to remaining terms, so simply append the pointer p1 to phead3

```
*/
if ( p1 != NULL)append (p1, phead3) ;
else
append (p2, phead3);
```

Now, you can implement the algorithm in C, and maybe make it more efficient.

5.2. Sparse Martrix:

Linked List Representation Of Sparse Matrix .

If most of the elements in a matrix have the value 0, then the matrix is called spare matrix.

Example For 3 X 3 Sparse Matrix:

| 1 0 0 |
| 0 0 0 |
| 0 4 0 |

3-Tuple Representation Of Sparse Matrix Using Arrays:

| 3 3 2 |
| 0 0 1 |
| 2 1 4 |

Elements in the first row represents the number of rows, columns and non-zero values in sparse matrix.
First Row - | 3 3 2 |
3 - rows
3 - columns
2 - non- zero values

Elements in the other rows gives information about the location and value of non-zero elements.

| 0 0 1 | (Second Row) - represents value 1 at 0th Row, 0th column
| 2 1 4 | (Third Row) - represents value 4 at 2nd Row, 1st column

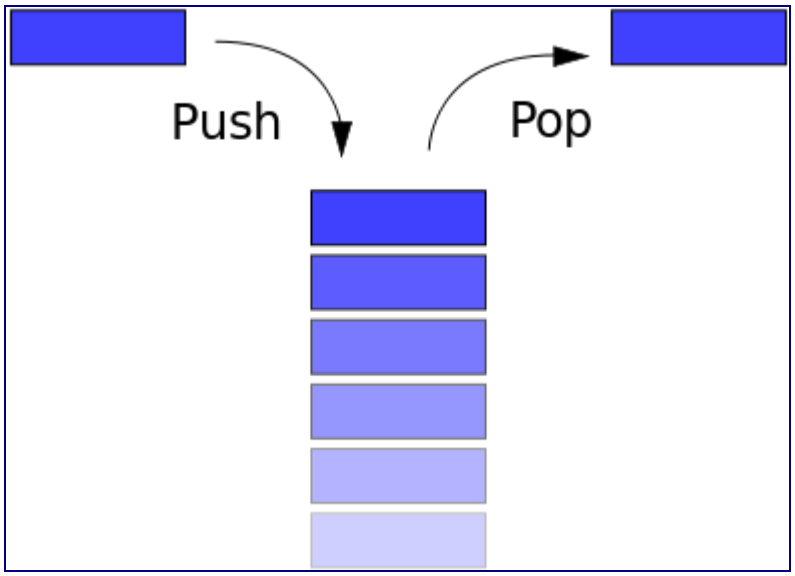
6.Stack:

6.1 Introduction:

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

The basic implementation of a stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data, since as we will see there are various variations of stack implementations.

There are basically three operations that can be performed on stacks . They are 1) inserting an item into a stack (push). 2) deleting an item from the stack (pop). 3) displaying the contents of the stack(pip).



6.2 Operation on Stacks:

Stack<item-type> Operations

push(*new-item*:item-type)	
	Adds an item onto the stack.
top():item-type	
	Returns the last item pushed onto the stack.
pop()	
	Removes the most-recently-pushed item from the stack.
is-empty():Boolean	
	True if no more items can be popped and there is no top item.
is-full():Boolean	
	True if no more items can be pushed.
get-size():Integer	
	Returns the number of elements on the stack.

All operations except get-size() can be performed in $O(1)$ time. get-size() runs in at worst $O(N)$.

6.3 Representing of a stack in Memory using Arrays:

Stack can be represented using a one-dimensional array. Allocate a block of memory required to accommodate the full capacity of the stack, and the items of a stack are stored in a sequential manner from the first location of the memory block.

Figure 6.2 Stack and its array representation

In Figure 6.2(a), stack is a one-dimensional array. *Top* is a pointer that points to the top element in the stack.

Stack operations.

Insert a new item onto stack.

• push()

Remove and return the item most recently added.

• pop()

• isEmpty() Is the stack empty?

push

pop

```
public static void main(String[] args)

{

StackOfStrings stack = new StackOfStrings();

while(!StdIn.isEmpty())

{

String s = StdIn.readString();

stack.push(s);

}

while(!stack.isEmpty())

{

String s = stack.pop();

StdOut.println(s);
```

```
}

}
```

a sample stack client

6.3 Representing of a stack in Memory using Linked List:

A Linked List is an abstract data type for representing lists as collections of linked items

Instead of having an overall representation of the list, the ordering of the list is represented locally

– That is, the information about what element comes next in a list is stored as a pointer within the element object.

– No list object (element) knows about any other elements in the list, just the ones to which it is adjacent

.

The basic linked list implementation is one of the easiest linked list implementations you can do. Structurally it is a linked list.

```
 type Stack<item_type>
data list:Singly Linked List<item_type>

constructor()
  list := new Singly-Linked-List()
end constructor
```

Most operations are implemented by passing them through to the underlying linked list. When you want to **push** something onto the list, you simply add it to the front of the linked list. The previous top is then "next" from the item being added and the list's front pointer points to the new item.

```
 method push(new_item:item_type)
  list.prepend(new_item)
end method
```

To look at the **top** item, you just examine the first item in the linked list.

```
method top():item_type
  return list.get-begin().get-value()
end method
```

When you want to **pop** something off the list, simply remove the first item from the linked list.

```
method pop()
  list.remove-first()
end method
```

A check for emptiness is easy. Just check if the list is empty.

```
method is-empty():Boolean
  return list.is-empty()
```

```
  end method
```

A check for full is simple. Linked lists are considered to be limitless in size.

```
  method is-full():Boolean
    return False
  end method
```

A check for the size is again passed through to the list.

```
  method get-size():Integer
    return list.get-size()
  end method
end type
```

A real Stack implementation in a published library would probably re-implement the linked list in order to squeeze the last bit of performance out of the implementation by leaving out unneeded functionality. The above implementation gives you the ideas involved, and any optimization you need can be accomplished by inlining the linked list code.

6.4 Multiple Stacks:

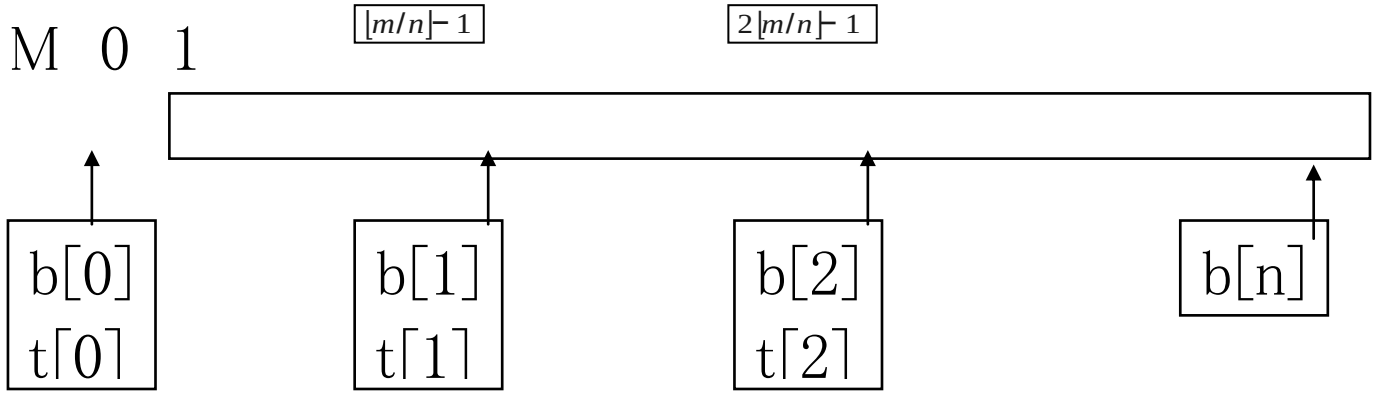
1. None fixed size of the stacks:
    - Stack 1 expands from the 0^{th} element to the right
    - Stack 2 expands from the 12^{th} element to the left
    - As long as the value of Top1 and Top2 are not next to each other, it has free elements for input the data in the array
    - When both Stacks are full, Top1 and Top 2 will be next to each other
    - There is no fixed boundary between Stack 1 and Stack 2
    - Elements –1 and –2 are using to store the information needed to manipulate the stack (subscript for Top 1 and Top 2)

2. Fixed size of the stacks:
    - Stack 1 expands from the 0^{th} element to the right
    - Stack 2 expands from the 6^{th} element to the left
    - As long as the value of Top 1 is less than 6 and greater than 0, Stack 1 has free elements to input the data in the array
    - As long as the value of Top 2 is less than 11 and greater than 5, Stack 2 has free elements to input the data in the array
    - When the value of Top 1 is 5, Stack 1 is full
    - When the value of Top 2 is 10, stack 2 is full
    - Elements –1 and –2 are using to store the size of Stack 1 and the subscript of the array for Top 1 needed to manipulate Stack 1
    - Elements –3 and –4 are using to store the size of Stack 2 and the subscript of the array for Top 2 needed to manipulate Stack 2

Sequential mapping of stacks into an array

- M[0..m-1]
- Example, two stacks, use M[0], M[m-1]

- Example, more than two stacks, n, use b[i]=t[i]=(m/n)*i-1



6.5 Application of Stack:

Stacks have a wide range of applications. The following are some of the applications of stack:

- Expression evaluation
- Recursion
- Balancing of the matching parenthesis

Expression Evaluation

An arithmetic expression can be represented in three ways:

1. *Infix*: An operator between two operands is an infix expression.

   <operand><operator><operand>

   Ex: a+b

2. *Postfix*: An operator that follows two operands is a postfix expression.

   <operand><operand><operator>

   Ex: a b+

3. *Prefix*: An operator that is followed by two operands is a prefix expression.

   <operator><operand><operand>

   Ex: +ab

Stacks can be used to evaluate expressions and also to convert expressions from one form to another.

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

Possible Questions

PART – A (20 Marks)
(Q.No 1 to 20 Online Examinations)

PART – B (2 Marks)

1. Define Linked List.
2. What is a Circular List.
3. What is a Doubly linked list.
4. What is Self Organizing List?
5. Define De-Queue

PART – C (6 Marks)

1. Discuss about Singly Linked List.
2. Discuss about Doubly Linked List.
3. Discuss about Circular List in detail.
4. Discuss about Representation of Stack in List.
5. Explain Normal and Circular List.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject : Data Structures
Class: II BCA **Subject code: 17CAU301**
Objective Type Questions

UNIT II

1	The data field of the _____ node usually donot conatain any information.	first	head	tail	last	head
2	A_____is a linked list in which last node of the list points to the first node in the list.	Linked list	Singly linked circular list	Circular list	Insertion node	Singly linked circular list
3	A_____in which each node has two pointers, a forward link and a Backward link.	Doubly linked circular list	Circular list	Singly linked circular list	Linked list	Doubly linked circular list
4	In sparse matrices each nonzero term was represented by a node with _____ fields.	Five	Six	Three	Four	Three
5	We want to represent n stacks with $1 \le i \le n$ then T(i)_____	Top of the i^{th} stack	Top of the $(i + 1)^{th}$ stack	Top of the $(i – 1)^{th}$ stack	Top of the $(i - 2)^{th}$ stack	Top of the i^{th} stack
6	We want to represent m queues with $1 \le i \le m$ then F(i)_____	Front of the $(i + 1)^{th}$ Queue	Front of the i^{th} Queue	Front of the $(i – 1)^{th}$ Queue	Front of the $(i -2)^{th}$ Queue	Front of the i^{th} Queue
7	We want to represent m queues with $1 \le i \le m$ then R(i)_____	Rear of the $(i + 1)^{th}$ Queue	Rear of the i^{th} Queue	Rear of the $(i – 1)^{th}$ Queue	Rear of the $(i - 2)^{th}$ Queue	Rear of the i^{th} Queue
8	_____ list allows traversing in only one direction.	Singly linked list	Doubly linked list	Circular Doubly Linked List	Ordered List	Singly linked list
9	_____ allows traversing in both direction.	Singly linked list	Doubly linked list	Circular Singly Linked List	Circular Queue	Doubly linked list

10	The best application of Doubly Linked list in computers is _____	Job scheduling in Time sharing environment	Processing Procedure calls	Dynamic Storage Management	Evaluating postfix expressions	Dynamic Storage Management
11	The computing time for manipulating the list is _____for sequential Representation	Less then	Greater than	Less then equal	Greater than equal	Less then
12	In singly linked list ,each node has _____field.	One	Two	Three	Five	Two
13	In linked list ,each node has fields namely_____	Link, Value	Link, Link	Data, Link	Data, Data	Data, Link
14	In Doubly linked list ,each node has at least _____field.	One	Two	Three	Five	Three
15	In Doubly linked list ,each node has fields namely_____	Link, Data1, Data2	Data and Link	Only Llink and Rlink	Llink, Data, Rlink	Llink, Data, Rlink
16	The doubly linked list is said to be empty if it conatins _____	no nodes at all.	nodes with data fields empty.	only a head node.	a node with its link fields points to null	only a head node.
17	In Linked representation of Sparse Matrix, DOWN field used to link to the next nonzero element in the same _____	Row	List	Column	Diagonal	Column
18	In Linked representation of Sparse Matrix, RIGHT field used to link to the next nonzero element in the same _____	Row	Matrix	Column	Diagonal	Row
19	The time complexity of the MREAD algorithm that reads a sparse matrix of n rows, n columns and r nonzero terms is ____	O(max {n, m, r})	O(m * n * r)	O(m + n + r)	O(max {n, m})	O(m + n + r)
20	A _____ is a set of characters is called a string.	Array	String	Heap	List	String
21	Adding a new element into a data structure called _____	Merging	Insertion	Searching	Sorting	Insertion

22	The Process of finding the location of the element with the given value or a record with the given key is _____.	Merging	Insertion	Searching	Sorting	Searching
23	Arranging the elements of a data structure in some type of order is called _____.	Merging	Insertion	Searching	Sorting	Sorting
24	What is the index number of the last element of an array with 29 elements?	29	28	0	25	28
25	The memory address of the first element of an array is called	Floor address	Foundation address	First address	Base address	Base address
26	Two dimensional array are also called as	Table arrays	Matrix arrays	both a and b	Special array	both a and b
27	Arrays are best data structure	for relatively permanent collection of data	for data are constantly changing	both a and b	none of the above	for relatively permanent collection of data
28	In a singly linkedlist how many fields are there?	1	2	3	4	2
29	Name the fields in circular linked list	Data and link	2Data and link	Data and 2link	2Data and 2link	2Data and link
30	To implement Sparse matrix dynamically, the following data structure is used	Stacks	linked list	Trees	Graphs	linked list
31	How many fields are there in doubly linked list?	1	2	3	4	3
32	In the last node of the circular linked list the link field contains	null	pointer data item	pointer to next node	pointer to first node	pointer to first noc
33	Name the fields in doubly linked list	Data and link	2Data and 2link	Data and 2link	2Data and 2link	Data and 2link
34	How many fields are there in circular doubly linked list?	1	2	3	4	3
35	In the last node of the circular doubly linked list the link field contains	null	pointer data item	pointer to next node	pointer to first node	pointer to first node
36	What member function places a new node at the end of the linked list?	addNode()	appendNode()	lastNode()	newNode()	appendNode()

37	The largest element of an array index is called its	lower bound	range	upper bound	subscript	upper bound
38	If the elements “A”, “B”, “C” and “D” are placed	ABCD	DCBA	DCAB	ABDC	DCBA
39	Consider the usual implementation of parentheses	1	2	3	4	3
40	Assume that the operators +,-, X are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, X, +, -. The postfix expression corresponding to the infix expression a + b X c – d ^ e ^ f is	abc X+ def ^ ^ -	abc X+ de ^ f ^ -	ab+c Xd – e ^ ^ f	-+aXbc ^ ^ def	abc X+ def ^ ^ -
41	The memory address of the first element of an array is called	floor address	base address	first address	foundation address	base address
42	What is the minimum number of stacks of size n required to implement a queue of size n?	1	3	2	4	2
43	The situation when in a linked list START=NULL is	underflow	overflow	housefull	saturated	underflow
44	A linear collection of data elements where the linear node is given by means of pointer is called	linked list	node list	primitive list	queue	linked list
45	Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?	Deleting a node whose location in given	Searching of an unsorted list for a given item	Inverting a node after the node with given location	Traversing a list to process each node	Deleting a node whose location in given
46	How many nodes in a tree have no ancestors.	0	1	2	n	1

le

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

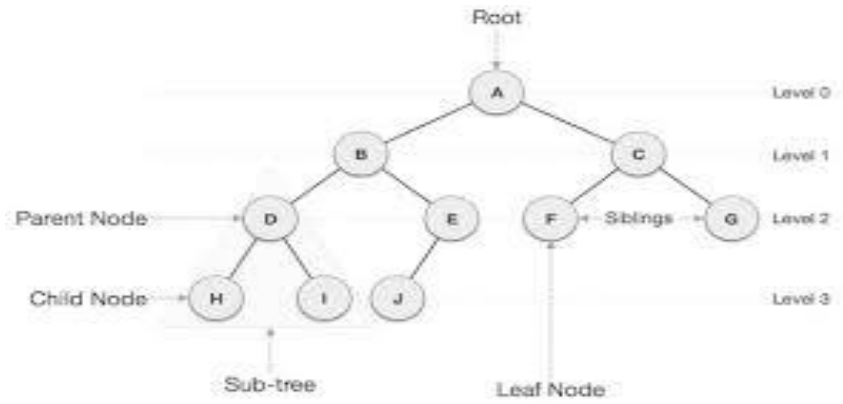
UNIT-III:

Trees - Introduction to Tree as a data structure; Binary Trees (Insertion, Deletion , Recursive and Iterative Traversals on Binary Search Trees); Threaded Binary Trees (Insertion, Deletion, Traversals); Height-Balanced Trees (Various operations on AVL Trees).

Trees:

Introduction to Tree as a data structure:

A tree is a data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the null or empty tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.



Tree

Terminology used in trees:

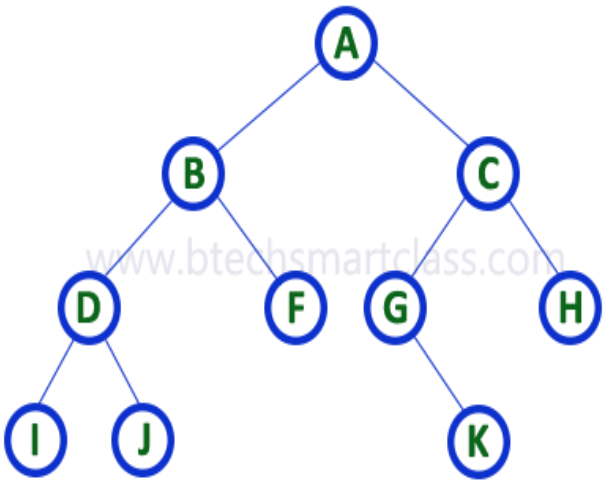
Root
The top node in a tree.
Child
A node directly connected to another node when moving away from the Root.
Parent
The converse notion of a child.
Siblings
A group of nodes with the same parent.
Descendant
A node reachable by repeated proceeding from parent to child.
Ancestor
A node reachable by repeated proceeding from child to parent.

Leaf
(less commonly called External node)
A node with no children.
Branch
Internal node
A node with at least one child.
Degree
The number of sub trees of a node.
Edge
The connection between one node and another.

Path
A sequence of nodes and edges connecting a node with a descendant.
Level
The level of a node is defined by 1 + (the number of connections between the node and the root).
Height of node
The height of a node is the number of edges on the longest path between that node and a leaf.
Height of tree
The height of a tree is the height of its root node.
Depth
The depth of a node is the number of edges from the tree's root node to the node.
Forest
A forest is a set of $n \geq 0$ disjoint trees.

Binary Trees:

In a normal tree, every node can have any number of children**. Binary tree** is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.
A tree in which every node can have a maximum of two children is called as **Binary Tree**.
In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.
Example:



Binary Search Trees:

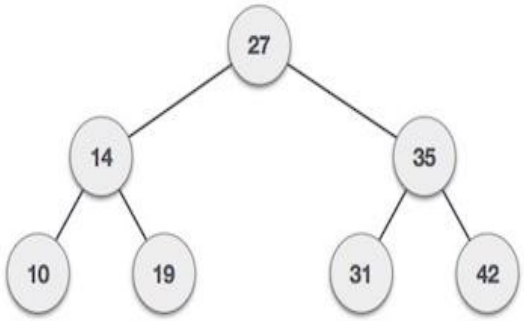
A **Binary Search Tree (BST**) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)
Representation:
BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



Binary Search Tree

We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations:

Following are the basic operations of a tree −

Search − Searches an element in a tree.

Insert − Inserts an element in a tree.

Pre-order Traversal − Traverses a tree in a pre-order manner.

In-order Traversal − Traverses a tree in an in-order manner.

Post-order Traversal − Traverses a tree in a post-order manner.

Node:

Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Search Operation:

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm:

```
struct node* search(int data){
struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){
```

```
    if(current != NULL) {
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data){
        current = current->leftChild;
      }//else go to right tree
      else {
        current = current->rightChild;
      }

      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }
  return current;
}
```

Insert Operation:
Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

```
  void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;

    while(1) {
      parent = current;
```

```
      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }//go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
          parent->rightChild = tempNode;
          return;
        }
      }
    }
  }
}
```

TRAVERSAL:

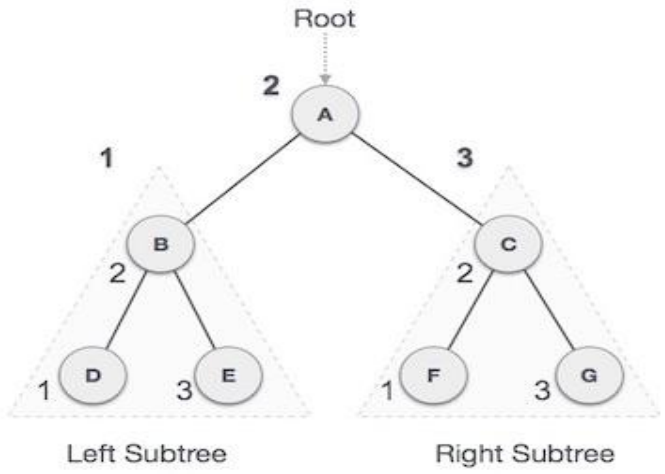
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

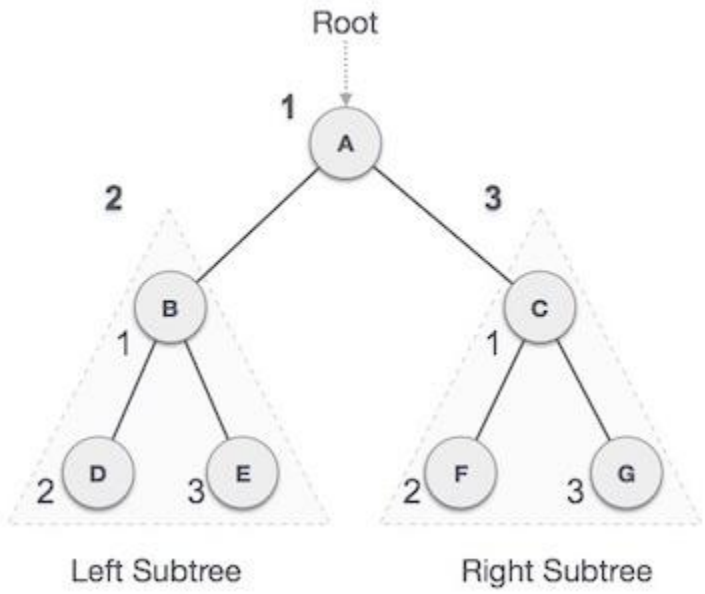
In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself. If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −
D → B → E → A → F → C → G

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal:
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

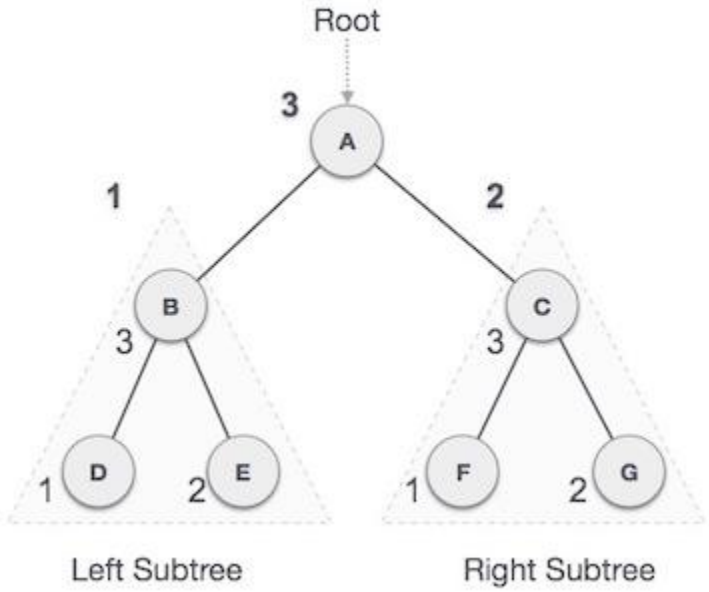


We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal:
In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following pre-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

THREADED BINARY TREES:
Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all

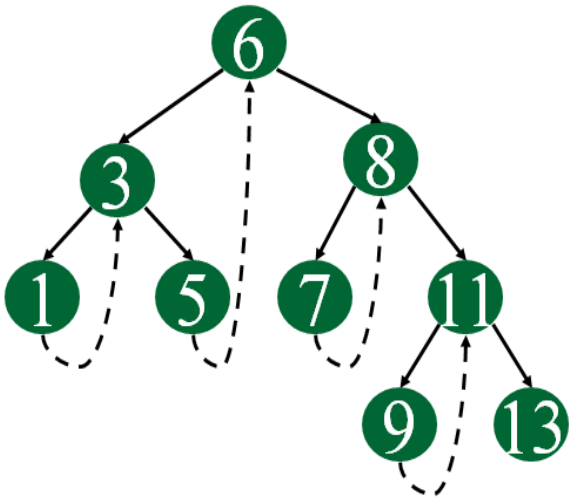
right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Representation of a Threaded Node:

```
struct Node
{
   int data;
   Node *left, *right;
   bool right Thread;
}
```

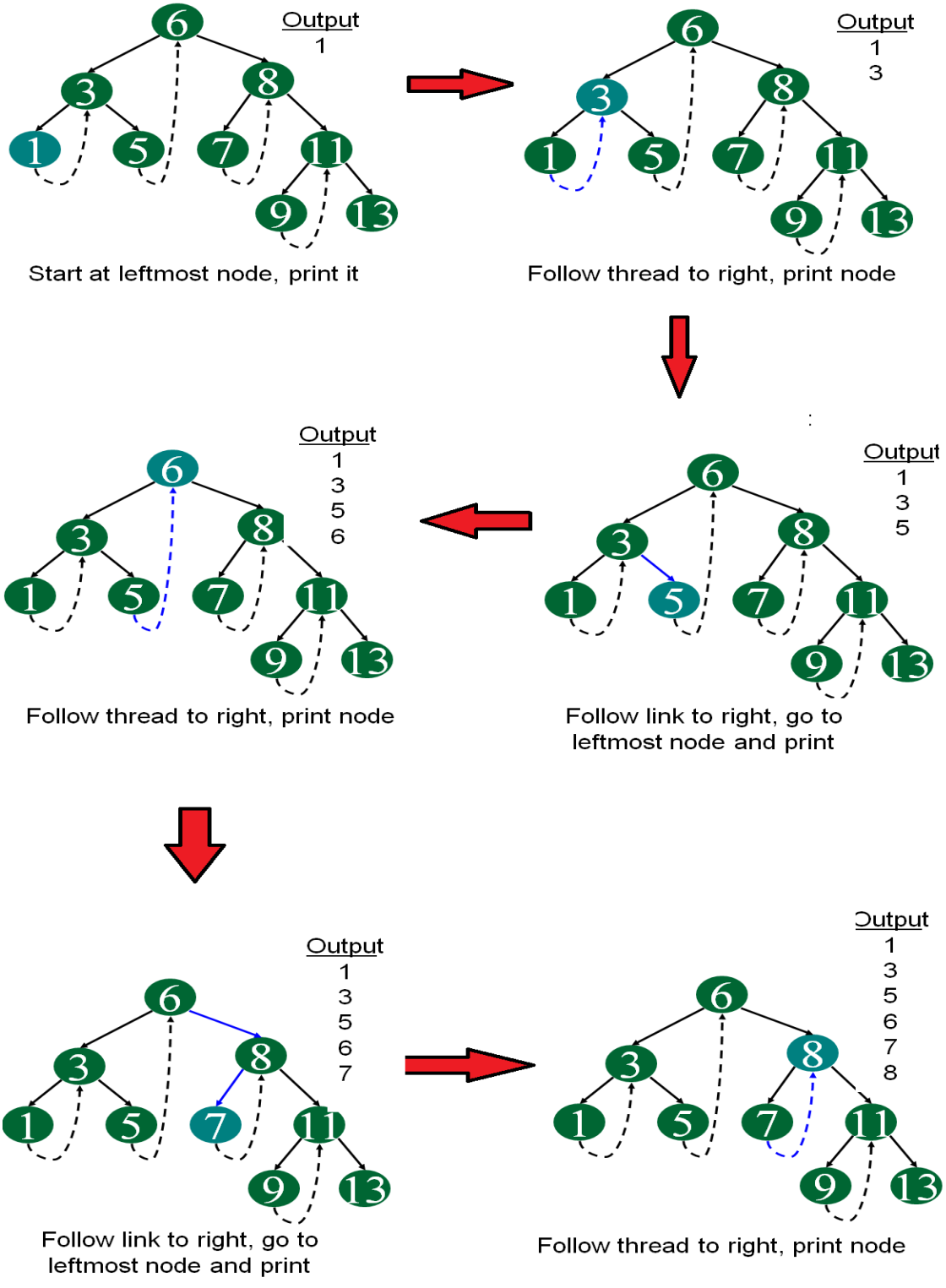
Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads
Following code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
   if (n == NULL)
     return NULL;

   while (n->left != NULL)
      n = n->left;

   return n;
}
```

// code to do inorder traversal in a threaded binary tree

```
void inOrder(struct Node *root)
{
   struct Node *cur = leftmost(root);
   while (cur != NULL)
   {
      printf("%d ", cur->data);

      // If this node is a thread node, then go to
      // inorder successor
      if (cur->rightThread)
         cur = cur->rightThread;
      else // Else go to the leftmost child in right subtree
         cur = leftmost(cur->right);
   }
}
```



continue same way for remaining node.....

INSERTION:

Insertion in Binary threaded tree is similar to insertion in binary tree but we will have to adjust the threads after insertion of each element.

representation of Binary Threaded Node:

```
struct Node
{
  struct Node *left, *right;
  int info;

  // True if left pointer points to predecessor
  // in Inorder Traversal
  boolean lthread;

  // True if right pointer points to successor
  // in Inorder Traversal
  boolean rthread;
};
```

In the following explanation, we have considered Binary Search Tree (BST) for insertion as insertion is defined by some rules in BSTs.

Let tmp be the newly inserted node. **There can be three cases during insertion**:

Case 1: Insertion in empty tree
Both left and right pointers of tmp will be set to NULL and new node becomes the root.

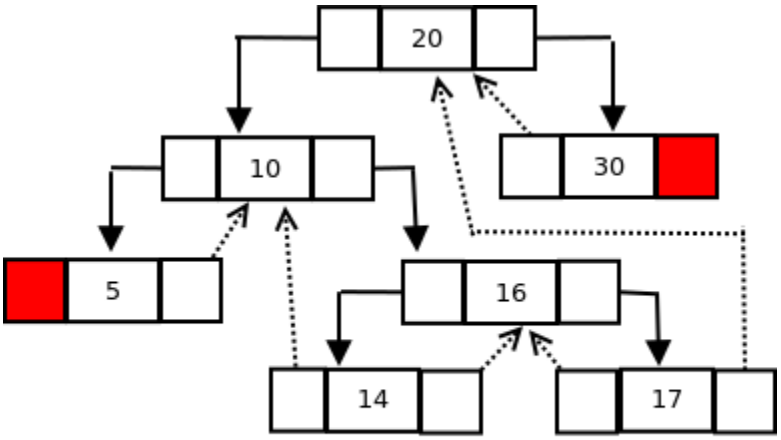
```
root = tmp;
tmp -> left = NULL;
tmp -> right = NULL;
```

Case 2: When new node inserted as the left child
After inserting the node at its proper place we have to make its left and right threads points to inorder predecessor and successor respectively. The node which was inorder successor. So the left and right threads of the new node will be-

```
tmp -> left = par ->left;
tmp -> right = par;
```

Before insertion, the left pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

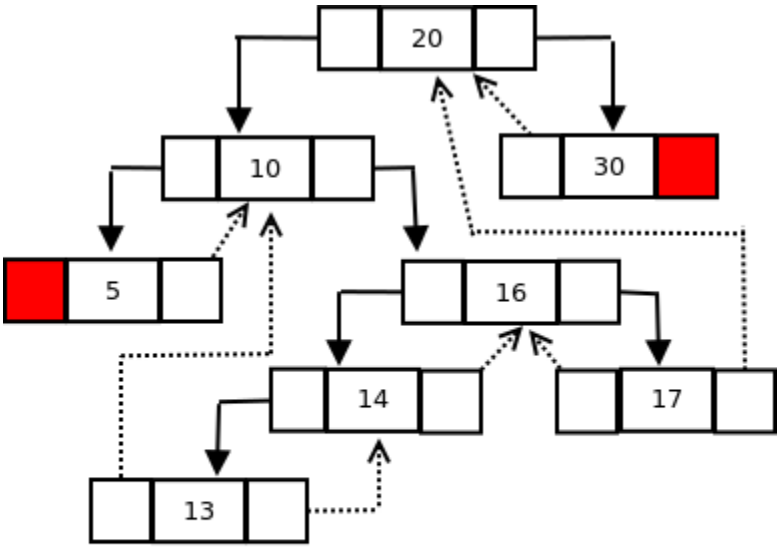
```
par -> lthread = par ->left;
par -> left = temp;
```



Insert 13

Inorder : 5 10 14 16 17 20 30

After insertion of 13,



13 inserted as left child of 14

Inorder : 5 10 13 14 16 17 20 30

Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10.
Successor of 13 is 14, so right thread of 13 points to left child which is 13.
Left pointer of 14 is not a thread now, it points to left child which is 13.

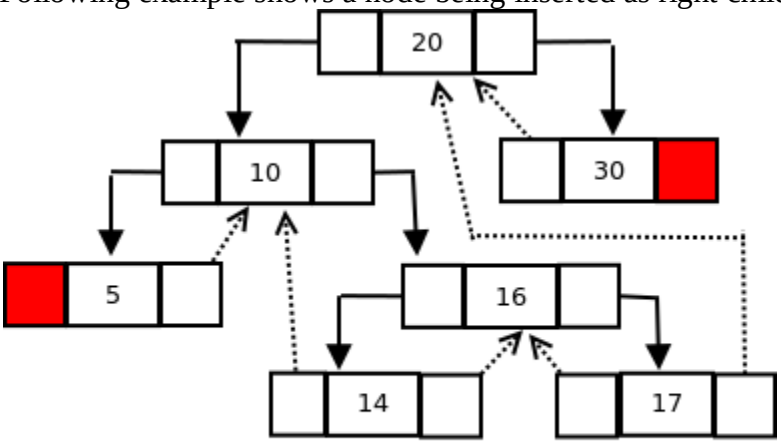
Case 3: When new node is inserted as the right child

The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

tmp -> left = par;

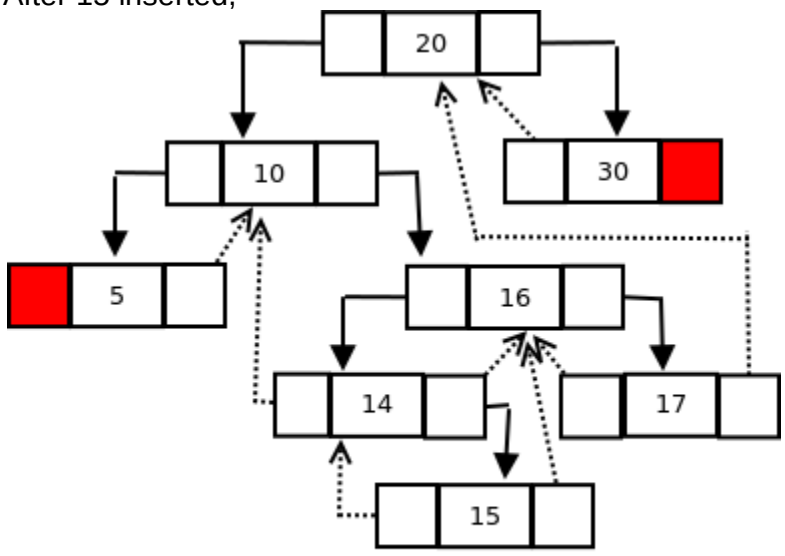
tmp -> right = par -> right;
Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

par -> rthread = false;
par -> right = tmp;
Following example shows a node being inserted as right child of its parent.



Insert 15
Inorder : 5 10 14 16 17 20 30

After 15 inserted,

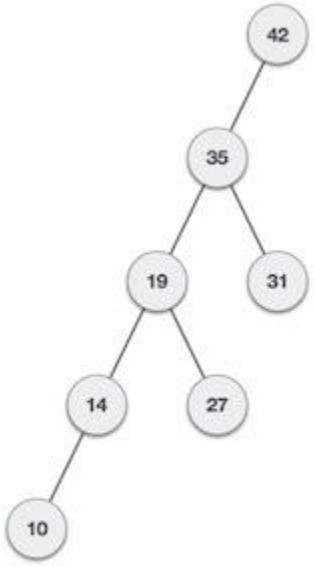


15 inserted as right child of 14
Inorder : 5 10 14 15 16 17 20 30

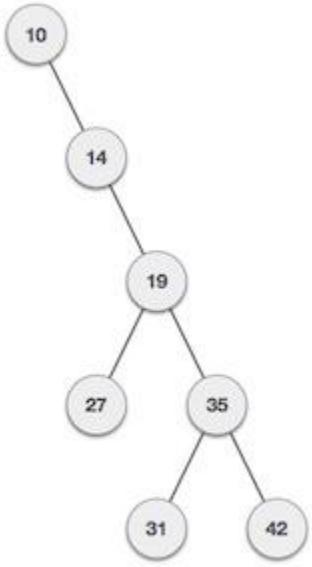
Successor of 14 becomes the successor of 15, so right thread of 15 points to 16
Predecessor of 15 is 14, so left thread of 15 points to 14.
Right pointer of 14 is not a thread now, it points to right child which is 15.

Height-Balanced Trees:

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



If input 'appears' non-increasing manner

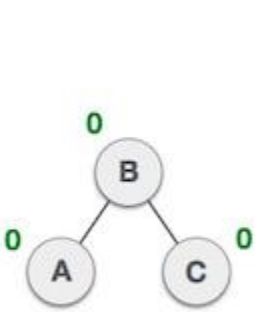


If input 'appears' in non-decreasing manner

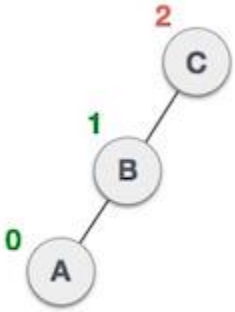
It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski & Landis**, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

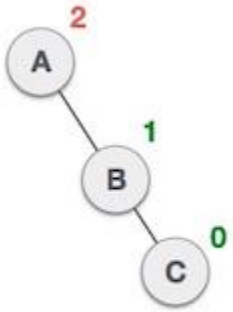
Here we see that the first tree is balanced and the next two trees are not balanced −



Balanced



Not balanced



Not balanced

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

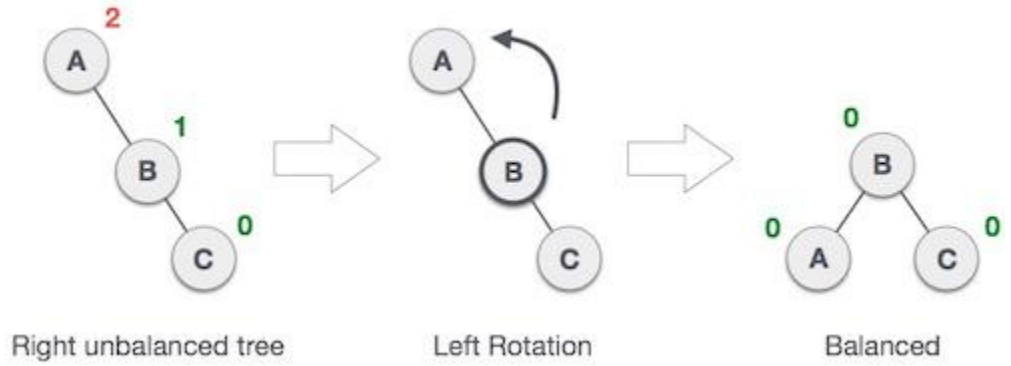
BalanceFactor = height(left-sutree) − height(right-sutree)
If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations:
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

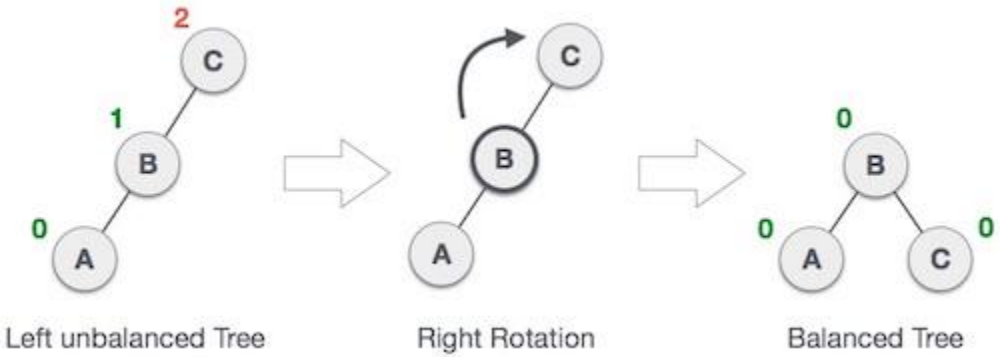
The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



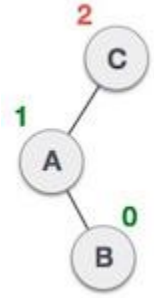
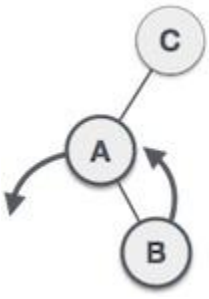
In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

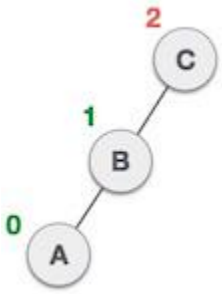
Right Rotation:
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



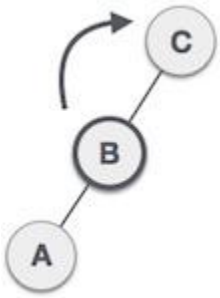
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation:Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

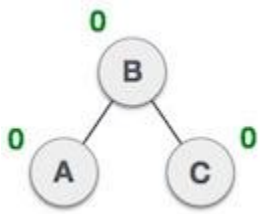
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.



Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.



We shall now right-rotate the tree, making **B**the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

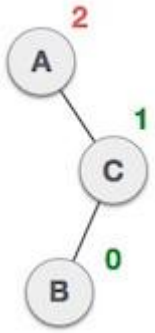


The tree is now balanced.

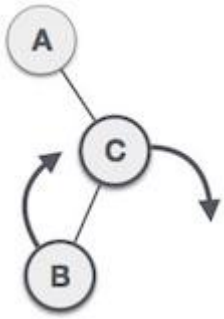
Right-Left Rotation:

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

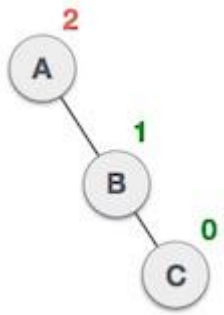
State	Action



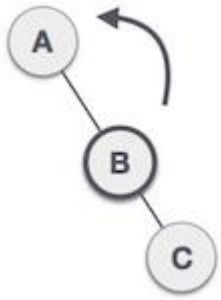
A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.



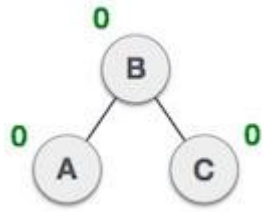
First, we perform the right rotation along **C**node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.



The tree is now balanced.

Operations on an AVL Tree:

The following operations are performed on an AVL tree

- Search
- Insertion
- Deletion

Search Operation in AVL Tree:

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree: In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

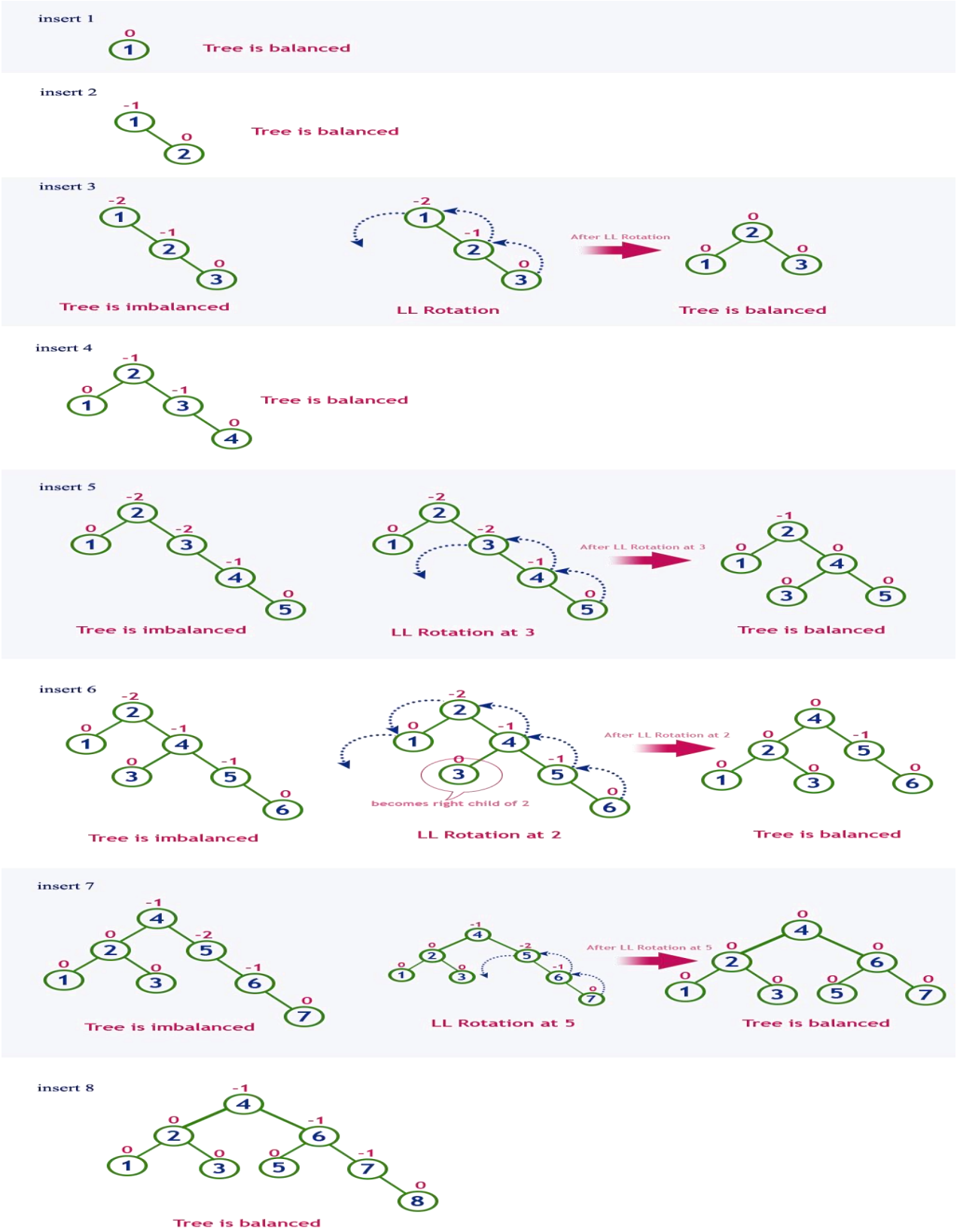
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

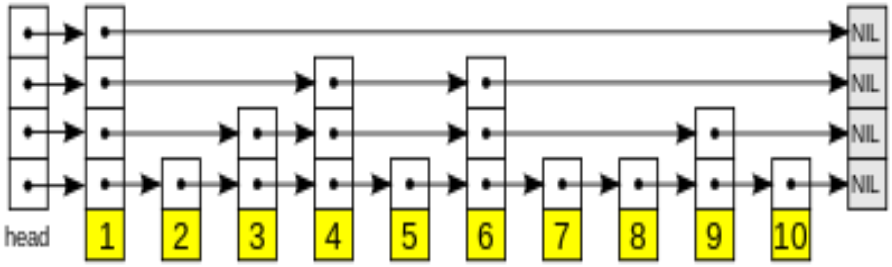

insert 1
Tree is balanced
insert 2
Tree is balanced
insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced
insert 4
Tree is balanced
insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced
insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced
insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced
insert 8
Tree is balanced

Deletion Operation in AVL Tree:

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

Skip List (Introduction):

Can we search in a sorted linked list in better than O(n) time?
The worst case search time for a sorted linked list is O(n) as we can only linearly traverse the list and cannot skip nodes while searching. For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

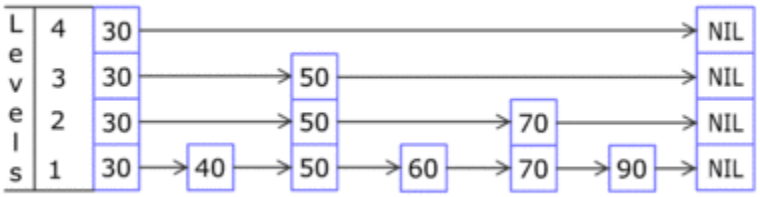


A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes (in yellow) at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer i+1 with some fixed probability p (two commonly used values for p are 1/2 or 1/4).

Implementation details:

The elements used for a skip list can contain more than one pointer since they can participate in more than one list.
Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list.



Inserting element to skip list

Can we augment sorted linked lists to make the search faster?
The answer is Skip List. The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an “express lane” which connects only main outer stations, and the lower layer works as a “normal lane” which connects every station. Suppose we want to search

for 50, we start from first node of "express lane" and keep moving on "express lane" till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on "express lane", we move to "normal lane" using pointer from this node, and linearly search for 50 on "normal lane". In following example, we start from 30 on "normal lane" and with linear search, we find 50.

What is the time complexity with two layers?

The worst case time complexity is number of nodes on "express lane" plus number of nodes in a segment (A segment is number of "normal lane" nodes between two "express lane" nodes) of "normal lane". So if we have n nodes on "normal lane", √n (square root of n) nodes on "express lane" and we equally divide the "normal lane", then there will be √n nodes in every segment of "normal lane" . √n is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be O(√n). Therefore, with O(√n) extra space, we are able to reduce the time complexity to O(√n).

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT : DATA STRUCTURES **SEMESTER**: III

CODE: 17CAU301 **CLASS**: II B.C.A

POSSIBLE QUESTIONS

UNIT-III

2 MARKS:

1. What is a Tree?
2. Define Binary Tree.
3. Write about Threaded Binary Tree.
4. Define Height-Balanced Tree.
5. Explain about AVL Trees.

6 MARKS:

1. Explain Insertion, Deletion and Recursive Operations in Binary Search Tree.
2. What is Threaded Binary Tree explain in detail.
3. Write in detail about the Operations of Binary Search Tree.
4. Write about Iterative, Traversal Operations on Binary Search Trees.
5. Write about (i) Tree (ii)Binary Tree (iii)Height Balanced Trees.

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

UNIT-IV

Searching and Sorting: Linear Search, Binary Search, Comparison of Linear and Binary Search, Selection Sort, Insertion Sort, Insertion Sort, Shell Sort, Comparison of Sorting Techniques

Heap: Introduction

Heaps are based on the notion of a **complete tree**,

Formally:

A binary tree is **completely full** if it is of height, *h*, and has 2^{h+1}-1 nodes.

A binary tree of height, *h*, is **complete** *iff*

a. it is empty *or*
b. its left subtree is complete of height *h*-1 and its right subtree is completely full of height *h*-2 *or*
c. its left subtree is completely full of height *h*-1 and its right subtree is complete of height *h*-1.

A complete tree is filled from the left:

- all the leaves are on
  - the same level *or*
  - two adjacent ones *and*
- all nodes at the lowest level are as far to the left as possible.

A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, we are left with two sub-trees and we must *efficiently* re-create a single tree with the heap property.

The value of the heap structure is that we can both extract the highest priority item and insert a new one in **O (log*n*)** time.

A heap is a binary tree that satisfies the following properties:

- Shape property
- Order property

By the shape property, we mean that heap must be a complete binary tree, whereas by order property we mean that for every node in the heap, the value stored in that node is greater than or equal to the value of its children.

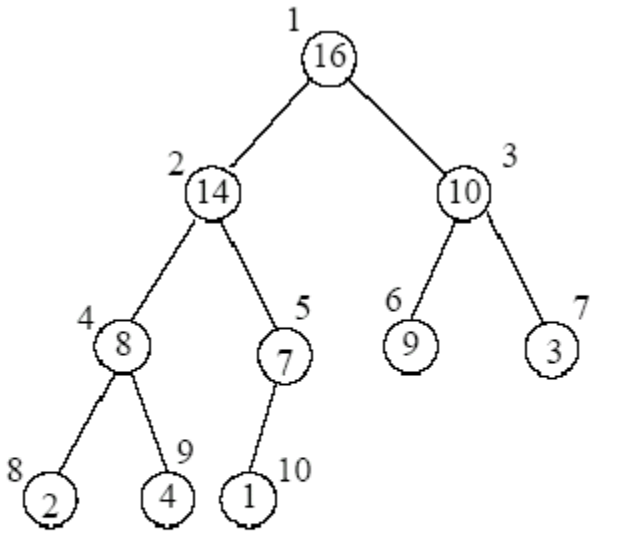
A heap satisfies these properties are known as max heap.

Representing a heap in memory

Heap is *represented in memory using linear arrays. i.e. by sequential representation.*

- length[A]: the size of the array
- heap-size[A]: the number of items stored into the array A
- **Note:** heap-size[A] <= length[A]
- The root of the tree is at A[1], i.e., the indexing typically begins at index 1 (not 0). A[0] can be reserved for the variable heap-size[A].

Since a heap is a complete or nearly complete binary tree, therefore a heap of size n is represented in memory using linear array of size n.



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Figure 1: Binary tree and array representation for the MaxHeap containing elements (that has the priorities) [16,14,10,8,7,9,3,2,4,1].

Routines to access the array

Lets consider the *i*:th **node** in a Heap that has the value A[i]

PARENT(i) = i/2	Return the index of the father node
LEFT(i) = 2i	Return the index of the left child
RIGHT(i) = 2i+1	Return the index of the right child

Operations on Heaps

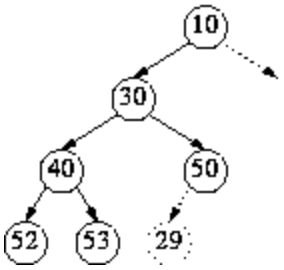
Insertion

Because our heaps are complete trees, we know where the new node *must* go. We have no choice, it must go in the bottom level, as far left as possible. The new value is placed in this node. We then check if the resulting tree is a heap: the place chosen for the new node guarantees

that the structural property will be satisfied, but the ordering property might be be violated. The ordering property is re-established by the `SIFT UP' operation.

The `SIFT UP' operation starts with a value in a leaf node. It moves the value up the path towards the root by successively exchanging the value with the value in the node above. The operation continues until the value reaches a position where it is less than its parent, or, failing that, until it reaches the root node.

Example: insert 29 into the above heap.

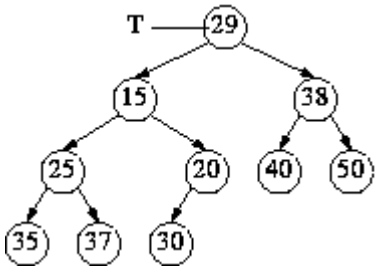


`29' must start where indicated; then it is *sifted up*. Complexity = O(height) = O(logN)

The DELETE operation is based on `SIFT DOWN', which, as the name suggests, is the exact opposite of SIFT UP.

SIFT DOWN starts with a value in any node. It moves the value *down* the tree by successively exchanging the value with the *smaller* of its two children. The operation continues until the value reaches a position where it is less than both its children, or, failing that, until it reaches a leaf.

Example: sift down 29 in this tree:



Steps:

- 29 is compared with 15 and 38. Exchange 29 and 15 and repeat.
- 29 is compared with 25 and 20. Exchange 29 and 20 and repeat.
- 29 is compared with 30. 29 is smaller, so stop.

Note: The tree's shape does not change. if it was a complete tree to start with, it will still be complete when the operation is done.

Complexity = O(height) = O(logN)

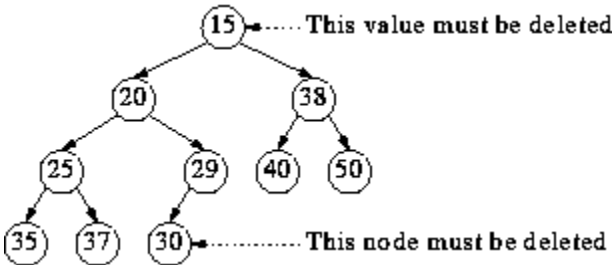
Deleting a Value From a Heap

Delete has two postconditions that seem contradictory:

1. V must not be in the resulting heap
2. the resulting heap must be a *complete* tree.

Condition (2) tells us which node must disappear: we must take away the *rightmost* node in the bottom level. This node must be `deleted' even if it is *not* the node containing V!

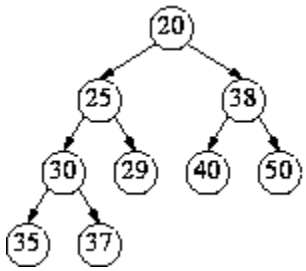
Example: delete 15 (the root) from this tree:



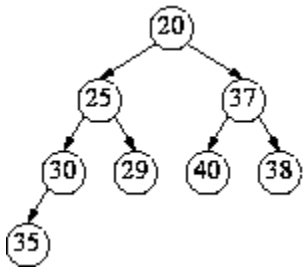
So we end up with a value (30) that has no node, and a node (`15') that has no value. The algorithm, then, is obvious.

1. save the value, X, in the rightmost node in the bottom level then delete this node.
2. Put X into the node containing V and then SIFT X UP, if it is smaller than its parent, or SIFT it DOWN if it is larger than either of its children.

Example: Delete 15 - delete the bottom right node, put its value (30) in the node where 15 was. 30 is smaller than 20 so we sift 30 down, with the final result:



Example: Delete 50 from this tree - delete the bottom right node, put its value (37) in the node where 50 was. 37 is smaller than 38 so we sift 37 UP, with the final result:



A special case of the DELETE operation is GET_SMALLEST, which returns the smallest value in a given heap and deletes the value from the heap.

Applications of heaps

The main applications of heaps are:

- Implementing a priority queue
- Sorting an array using efficient technique known as heap sort.

Priority queue:

A priority queue is a structure with an interesting accessing function: only the highest priority element can be accessed.

The operations defined for the priority queue are very much similar to the operation specified for FIFO queue.

For example, a priority queue *pq* with storage capacity *MAX* is defined, the various operations that can be implemented are described below

CreateEmptyPQ(pq,n)

```
Begin
        Set n=0
End.
IsEmptyPQ(pq,n,status)
Begin
        If(n-0)then
                Set status=true
        Else
                Set status=false
        Endif
End
IsFullPQ(pq,n,status)
Begin
        If(n=MAX) then
                Set status=true
        Else
                Set status=false
        Endif
End
EnqueuePQ(pq,n,element)
Begin
        Call IsFullPQ(pa,n,status)
        If(status=true) then
                Print "Overflow"
        Else
                Call InsertElement(pq,n,element)
        Endif
End.
DequeuePQ(pa,n,element)
Begin
        Call IsEmptyPQ(pq,n,element)
        If(status=true)then
                Print "Overflow"
        Else
                Call DeleteElement(pq,n,element)
        Endif
End.
```

Building a Heap

Before applying heap sort technique for sorting an array, the first task is to build a heap(to convert unsorted array into a heap).

Downheap operation is the interchange the value of the root node with the value of the child which is the largest among the children.

Apply the downheap operation from the leftmost non leaf node to all the subtrees in this level. So that the

```
Heapify(a,n)
Begin
        Set index=Parent of node with index n
        For i=index to 1 by -1 do
                Call Downheap9a,I,n)
        End for
End.
```

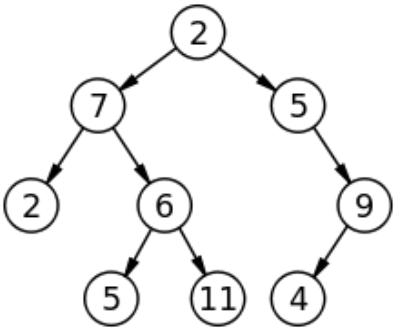
Graph: Introduction, representation of graphs

Graph is an important non-linear data structure. This data structure is used to represent relationship between pairs of elements which are not necessarily hierarchical in nature.

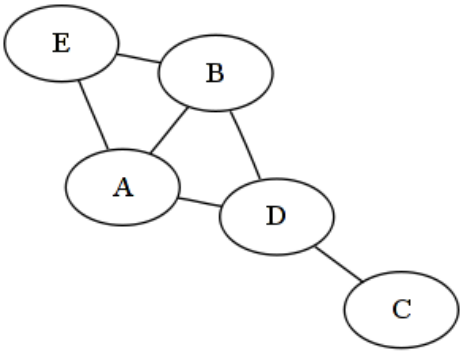
A graph is defined as "*Graph G is an ordered set(V,E) where V(G) represent the set of elements, called vartices, and E(G) represents the edges between these vertices".*

A graph can be either directed or undirected. In an undirected graph there is no specific direction associated with the edges.

In an *undirected graph*, the edges are represented by an unordered pair whereas in a *directed graph*, the edges are represented by an ordered pair.



Directed Graph



Undirected graph

Graph terminologies

Adjacent vertices – as an edge e is represented by pair of vertices denoted by [u,v]. The vertices u and v are called endpoints of e. these vertices are also called adjacent vertices are neighbours.

Degree of a vertex – the degree of vertex u, written as deg(u), is the number of edges containing u. if deg(u)=0, this means that vertex u does not belong to any edge, then vertex u is called an isolated vertex.

Path – a path p of length n from a vertex u to vertex v is defined as a sequence of (n+1) vertices, i.e.

P=(v1,v2,v3,…vn+1)

Such that v=v1, v=vn+1, and vi-1 is adjacent to vi for i=2,3,..,(n+1)

The path is said to be closed if the end points of the path are same i.e. v1=vn+1.

The path is said to be simple if all the vertices in the sequence are distinct, with the exception that v1=vn+1. In that case it is known as closed simple path.

Cycle – a cycle is a closed simple path with length 2 or more. Sometimes, a cycle of length k(i.e. k distinct vertices in the path) is known as k-cycle.

Connected graph- A graph G is said to be connected id there is path between any two of its vertices. i.e. there is no isolated vertices.

A connected graph without any cycles is called a tree. Thus it is also called as special graph.

Complete graph – A graph G is said to complete or fully connected if there is path from every vertex to every other vertex. A complete graph with n vertices will have n(n-1)/2 edges.

Weighted graph – A graph is said to weighted graph id every edge in the graph is assigned some data. The weight of the edge, denoted by w(e), is a non-negative value that may be representing the cost of moving along that edge or distance between the vertices.

Multiple edges – Distinct edges e and e' are called multiple edges if they connect the same endpoints. i.e. if e=[u,v] and e'=[u,v].

Multigraph – A graph containing multiple edges.

Loop – An edge is a loop if it has identical end points. i.e. if e=[u,u].

Representation of graphs

Following two are the most commonly used representations of graph.
1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

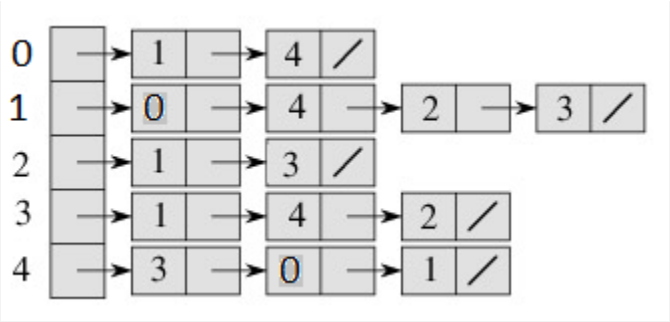
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph
Let max number of vertices as 50.

```
#define MAX 50
Typedef struct node_type
{
        Int vertex;
        Struct node_type *link;
}node;
```

Operations on graph

Creating an empty graph

To create an empty graph, the entire adjacency list is set to NULL.

```
Void CreateGraph(node *adj[], int num)
{
        int i;
        for (i=1;i<=num;i++)
                adj[i]=(node *) NULL;
}
```

Entering Graph Information

```
void InputGraph (node *adj[], int num)
{
        node *ptr, *last;
        int i,j,m,val,wt;
        for(i=1;i<=num;i++)
        {
                last=(node*)NULL;
                printf("Number of nodes in the adjacency list of node %d:",i);
                scanf("%d",&m);
                for(j=1;j<=m;j++)
                {
                        printf("Enter node #%d:",j);
                        scanf("%d",&val);
                        Printf("Enter weight for edge(%d,%d):",i,val);
                        Scanf("%d",&wt);
                        Ptr=(node *)malloc(sizeof(node));
                        Ptr->vertex=val;
                        Ptr->weight=wt;
                        Ptr->link=(node *)NULL;
                        If(adj[i]==(node *)NULL)
                                Adj[i]=last=ptr;
                        Else
                        {
                                Last->link=ptr;
                                Last=ptr;
                        }
                }
        }
```

```
}
```

Outputting a Graph

```
Void PrintGraph(node *adj[],int num)
{
        Node *ptr;
        Int I;
        For(i=1;i<=num;i++)
        {
                Ptr=adj[i];
                Printf("%d",i);
                While(ptr!=NULL)
                {
                        Printf("->(%d,%d)",ptr->vertex,ptr->weight);
                        Ptr=ptr->link;
                }
                Printf("\n");
        }
}
```

Deleting a graph

```
Void DeleteGraph(node *adj[],int n)
{
        Int I;
        Node *temp,*ptr;
        For(i=1;i<=n;i++)
        {
                Ptr=adj[i];
                While(ptr!=(node*)NULL)
                {
                        Temp=ptr;
                        Ptr=ptr->link;
                        Free(temp);
                }
                Adj[i]=(node*)NULL;
        }
}
```

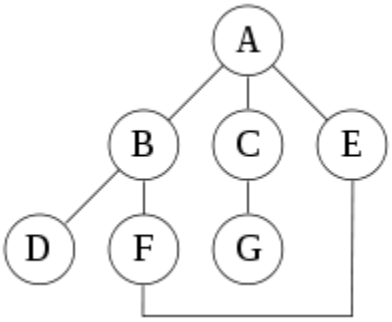
Traversal:Breadth first search

breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the

neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.
The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:
Enqueue the root node
Dequeue a node and examine it
If the element sought is found in this node, quit the search and return a result.
Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
If the queue is not empty, repeat from Step 2.
Pseudocode
Input: A graph G and a root v of G

```
1  procedure BFS(G,v) is
2      create a queue Q
3      create a set V
4      add v to V
5      enqueue v onto Q
6      while Q is not empty loop
7        t ← Q.dequeue()
8        if t is what we are looking for then
9          return t
10       end if
11       for all edges e in G.adjacentEdges(t) loop
12         u ← G.adjacentVertex(t,e)
13         if u is not in V then
14             add u to V
15             enqueue u onto Q
16         end if
17       end loop
18     end loop
19     return none
20 end BFS
```

Traversal:Depth first search
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root(selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch beforebacktracking.
Example
For the following graph:



a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

Applications of Graphs:

Spanning Tree:

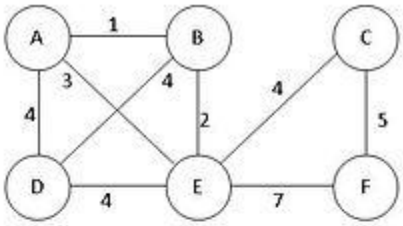
A spanning tree for a graph, G=(V,E), is a subgraph of G that is a tree and contains all the vertices of G. In a weighted graph, the weight of a graph is the sum of the weights of the edges of the graph.

A minimum spanning tree(MST) for a weighted graph is a spanning tree with minimum weight. If graph G with n vertices, then the MST will have (n-1) edges, assuming that the graph is connected. In general, a weighted graph may have more than one MST.

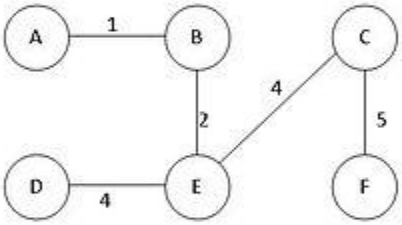
If G is not connected, then it cannot have any spanning tree.

One example would be a telecommunications company laying cable to a new neighborhood.

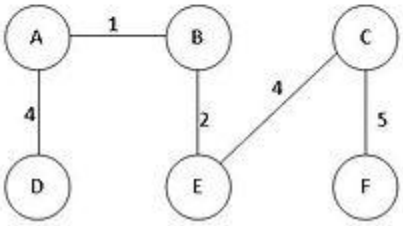
EXAMPLE:



Weighted graph



MST 1



MST2

The Dijikstra/Prim algorithm begins by selecting an arbitrary starting vertex, and then branches out from the part of the tree constructed so far by choosing a new vertex and edge at each iteration, during the course of the algorithm, the vertices may be thought of as divided into three disjoint categories as follows:
Tree vertices – Those in the tree constructed so far
Fringe vertices – Those vertices which are not in tree, but are adjacent to some vertex in the tree.
Unseen vertices – Remaining vertices of the graph.

The key stem in the algorithm is the selection of a vertex from the fringe vertices and an incident edge.
Since the weights are on the edges, the focus of the choice is on the edge, not the vertex. The Prim algorithm always chooses an edge from a tree vertex to a fringe vertex of minimum weight,

The general structure of the algorithm can be described as follows:

Begin
Select an arbitrary vertex to start the tree
while there are fringe vertices do
select an edge of minimum weight between a tree and a fringe vertex

add the seleted edge and the fringe vertex to the tree
end while
End

After the each iterations of the algorithm's loop, there may be new fringe vertices, and the set of edges from which the next selection is made. For each fringe vertex, only one edge to it from the tree is being tracked. That edge is the one with lowest weight. Such edges are called as candidate edges.

Algorithm:

```
MinimumSpanningTree(adj,n)
Begin
    for i=2 to n by 1 do
        Set status[i] =UNSEEN
    endfor
    set x=1
    set status[x] = INTREE
    set edge-count=0
    set stuck = false
    Create fringe-list
    while((edge-count<n-1)and(not stuck))do
        Set ptr=adj[x]
        while(ptr ≠NULL)do
            Set y=ptr→vertex
            If((status[y]=fringe)and(ptr→weight<fringe-wt[y])) then
              Set parent[y]=x
              Set fringe-wt[y]=ptr→weight
            else if (status[y]=UNSEEN)then
              Set status[y]=FRINGE
              Insert vertex y into fringe-list
              Set parent[y]=x
              Set fringe-wt[y]=ptr→weight
            endif
            Set ptr=ptr→link
        endwhile
        if(fringe-list is empty)then
            Set stuck =true
else
    Traverse the fringe list to find a vertex with minimum weight
    Set x=the fringe vertex incident with the edge
    Remove x from the fringe-list
    Set status[x]=INTREE
```

```
        Set edge-count=edge-count+1
Endif
        endwhile
for x=2 to n by 1 do
                Print x, parent[x]
endfor
End.
```

Topological sort

A topological sort of a directed graph without cycles, also known as directed acyclic graph or simple DAG,G=(V,E) is a linear ordering of all its vertices such that it G contains an edge(u,v), then u appears before v in the ordering.
If the graph contains cycle(s), i.e. graph is not DAG, then no linear ordering is possible.
A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

Directed acyclic graphs are used in many applications to indicate precedence among events.

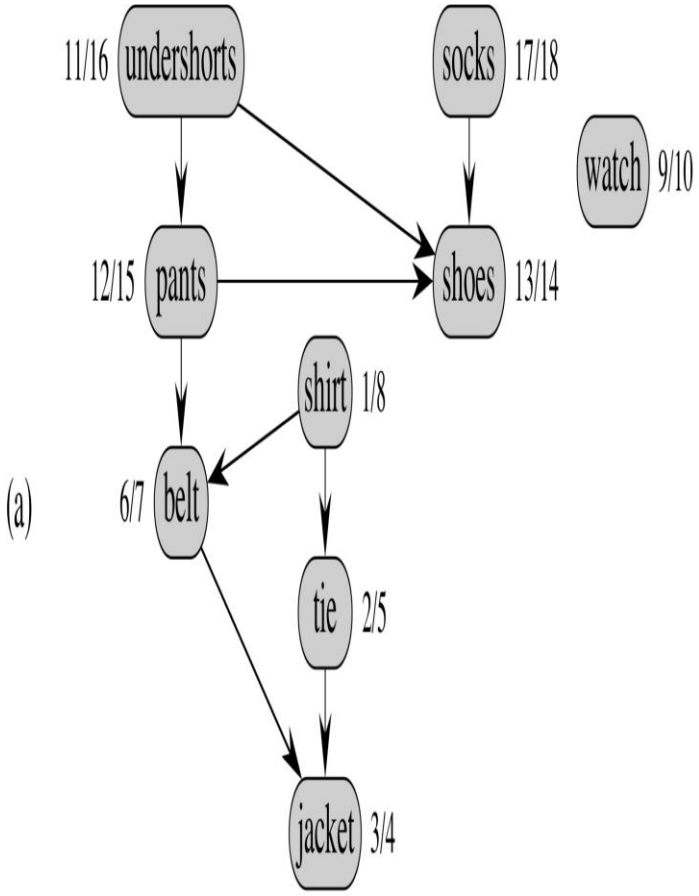
Algorithm:

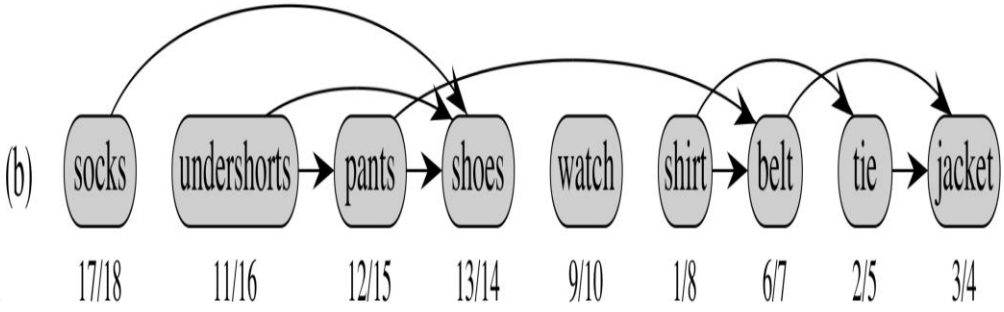
```
TopologicalSort(adj,n)
Begin
        Create linear linked LIST
        for i=1 to n by 1 do
                Set color[i] = WHITE
        endfor
        for i=1 to n by 1 do
                if(color[i] =WHITE)then
                        Call DfsVisitModified(adj,n,i)
                endif
        endfor
        Set ptr=LIST
        while (ptr≠NULL)do
                Print ptr→info
                Set ptr=ptr→link
        endwhile
End.
```

Algorithm for DfsVisitModified

```
DfsVisitModified(adj,n,u)
Begin
        Set color[u]=GRAY
```

```
Set ptr=adj[u]
while(ptr≠NULL)do
        Set v=ptr→info
        if(color[v]=WHITE)then
                Call DfsVisitModified(adj,n,v)
        endif
        Set ptr=ptr→link
endwhile
Insert vertex u in beginning of linear linked list LIST
Set color[u]=BLACK
```


11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
(a)
6/7
belt
tie
2/5
jacket
3/4


(b)
socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

End.

Shortest path algorithm:

A path from source vertex v to x is shortest path from v to w if there is no path from v to w with lower weights. The shortest paths are not necessarily unique.

The distance from a vertex x to a vertex y, denoted by d(x,y), is the weight of a shortest path from vertex x to vertex y. The dijikstra's shortest-path algorithm finds the shortest paths in order of increasing distance from v. Like MST, it branches out by selecting certain edges that lead to new edges.

Like MST, the candidate edge (the best so far) is being tracked for each fringe vertex. For each fringe vertex z, there is atleast one path $v(=v_0),v_1,v_2,\ldots v_k,z$ such that all the vertices except $_z$ are already in the tree. The candidate edge for x is the edge v_kz from a shortest path of this form. This information is stored in linear array DIST.

Algorithm:

```
ShortestPath(adj,n,s,d)
Begin
        for i=1 to n by 1 do
                Set status [i] =UNSEEN,parent[i]=0
        endfor
        Setstatus[s]=INTREE
        Setdist[s]=0
        Create fringe=list
        Set x=s
        Set stuck=false
        while ( ( x≠d)and(not stuck))do
                Set ptr=adj[x]
                while (ptr≠NULL)do
                  Set y=ptr→node
                  if((status[y]=fringe)and(dist[x]+ptr→weight<dist[y]))then
                        Set parent[y]=x
                        Set dist[y]=dist[x]+ptr→weight
                  else if (status[y]=UNSEEN)then
                        Set status[y]=FRINGE
                        Insert vertex y into fringe-list
                        Set parent[y]=x
                        Set dist[y]=dist[x]+ptr→weight
```

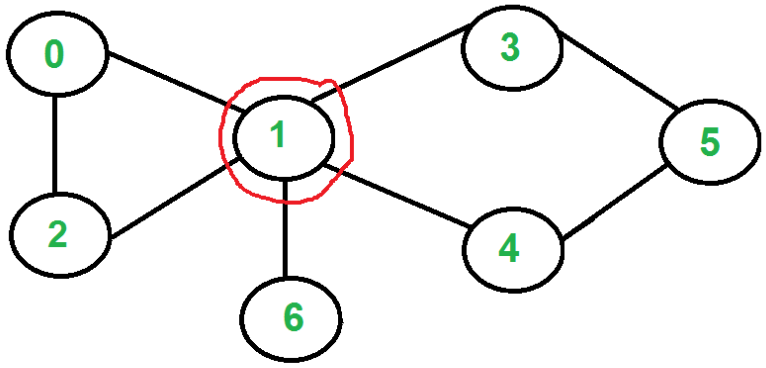
```
                endif
                Set ptr=ptr→link
                endwhile
        if(fringe-list is empty) then
                Set stuck=true
        else
                Traverse the fringe-list to find a vertex with minimum distance
                Set x=vertex with minimum distance from s
                Remove x from the fringe-list
                Set status[x]=INTREE
        endif
  endwhile
  if(parent[d]=0)then
        Print"No path from ",s,"to",d,"exists"
  else
        Call print-path(s,d,parent)
  endif
End.
```

Algorithm for PrintPath

```
PrintPath(s,d,parent)

Begin
        if(s=d)then
                Print s
        else
                Call PrintPath(s,parent[d],parent)
        endif
End.
```

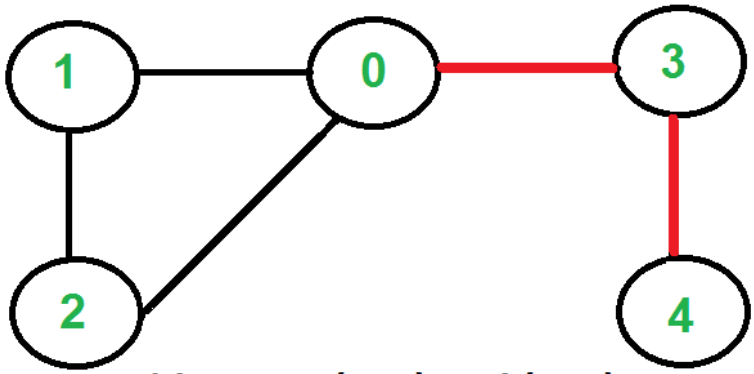
Articulation points,Bridges and Biconnected Components

Let G=(V,E) be a connected, undirected graph. An Articulation point, also called a cut point, of G is a vertex whose removal disconnects G.



Articulation Point is 1

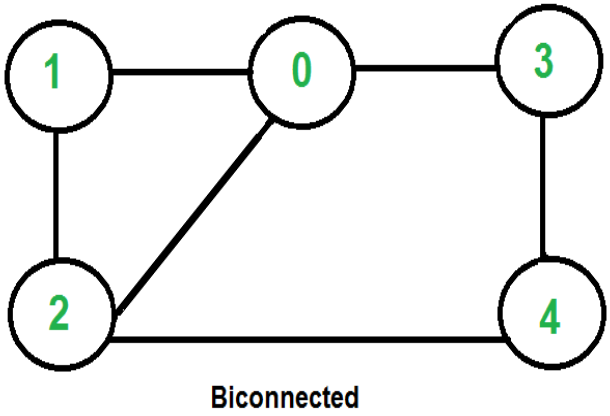
A bridge of G is an edge whose removal disconnects G.



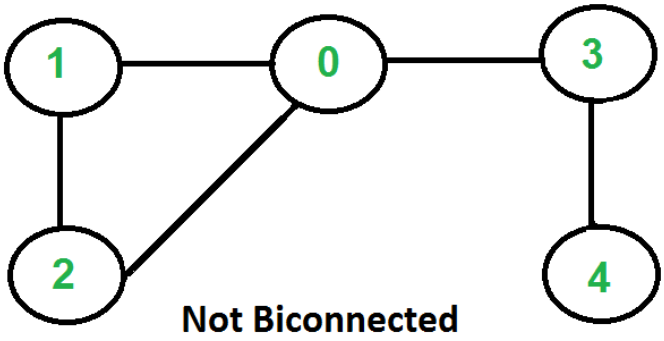
Bridges are (0, 3) and (3, 4)

A biconnected component of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. A graph with no articulation point is called biconnected. In other words, a graph is biconnected iff any vertex is deleted the graph remains connected. A biconnected graph is a maximal biconnected subgraph.

Biconnected and Non biconnected graphs:



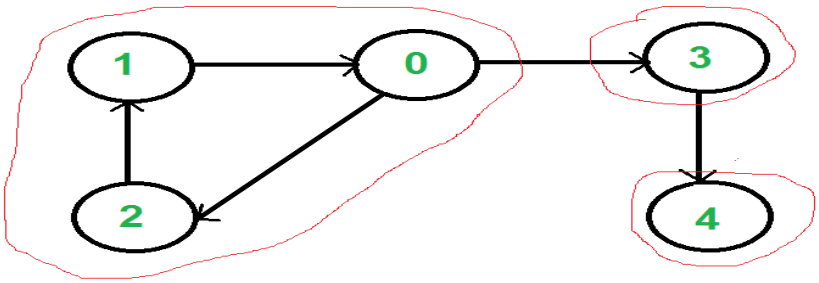
Biconnected

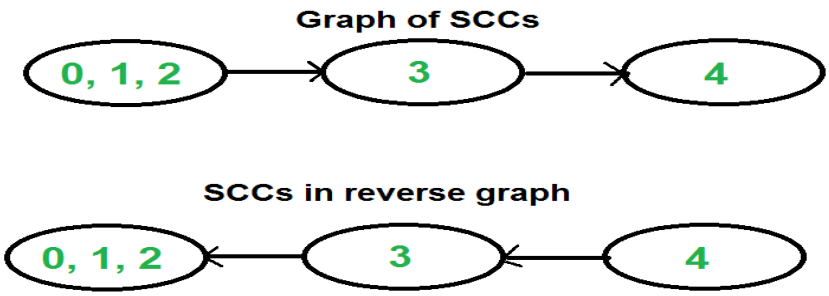


Not Biconnected

Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.





EULERIAN TOUR

An Eulerian tour through an undirected graph is a path whose edge list contains each edge of the graph exactly once. An Eulerian graph is a graph that possesses an Eulerian tour. It has been proved that an undirected graph is Eulerian graph if and only if it is connected and has either zero or two vertices with an odd degree.

HAMILTONIAN TOUR

A Hamiltonian tour through an undirected graph is a path whose vertex list contains each vertex of the graph exactly once. A Hamiltonian graph is a graph that possesses a Hamiltonian tour.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT : DATA STRUCTURES **SEMESTER**: III
CODE: 17CAU301 **CLASS**: II B.C.A

POSSIBLE QUESTIONS

PART – A (1X20=20)

(Multiple Choice Questions)

PART-B (2 MARKS)

1. Define Searching.
2. What is Sorting.
3. What is Linear Search.
4. What is Binary Search.
5. Define Shell Sort.

PART C – (6 MARKS)

1. Define Searching. Write an Algorithm for Linear Search.
2. Write an Algorithm for Binary Search.
3. Compare Linear and Binary Search .
4. Write an Algorithm for Binary Search.
5. Write an Algorithm for Linear Search.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject : Data Structures
Class: II BCA **Subject code: 17CAU301**
Objective Type Questions

UNIT IV

S.No	QUESTION	OPT 1	OPT 2	OPT 3	OPT 4	ANSWER
1	Which of the following operations is performed more efficiently by doubly linked list than by singly linked list	Deleting a node whose location is given.	Searching of an unsorted list for a given item.	Inserting a new node after node whose location is given.	Traversing the list to process each node.	Deleting a node whose location is given.
2	Nodes that have degree zero are called _____.	end node	leaf nodes	subtree	root node	leaf nodes
3	A binary tree with all its left branches supressed is called a _____	balanced tree	left sub tree	full binary tree	right skewed tree	right skewed tree
4	All node except the leaf nodes are called_____.	terminal node	percent node	non terminal	children node	non terminal
5	The roots of the subtrees of a node X, are the _____ of X.	Parent	Children	Sibling	sub tree	Children
6	X is a root then X is the _____ of its children.	sub tree	Parent	Sibilings	subordinate	Parent
7	The children of the same parent are called _____.	sibiling	leaf	child	subtree	sibiling
8	_____of a node are all the nodes along the path form the root to that node.	Degree	sub tree	Ancestors	parent	Ancestors
9	The _____of a tree is defined to be a maximum level of any node in the tree.	weight	length	breath	height	height
10	A_____ is a set of n ≥ 0 disjoint trees	Group	forest	Branch	sub tree	forest

11	A tree with any node having at most two branches is called a _____.	branched tree	sub tree	binary tree	forest	binary tree
12	A _____of depth k is a binary tree of depth k having 2^K-1 nodes.	full binary tree	half binary tree	sub tree	n branch tree	full binary tree
13	Data structure represents the hierarchical relationships between individual data item is known as _____.	Root	Node	Tree	Address	Tree
14	Node at the highest level of the tree is known as _____.	Child	Root	Sibiling	Parent	Root
15	The root of the tree is the _____of all nodes in the tree.	Child	Parent	Ancestor	Head	Ancestor
16	_____is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree	Subtree
17	A node with no children is called _____.	Root Node	Branch	Leaf Node	Null tree	Leaf Node
18	In a tree structure a link between parent and child is called _____	Branch	Root	Leaf	Subtree	Branch
19	Height – balanced trees are also referred as as _____trees.	AVL trees	Binary Trees	Subtree	Branch Tree	AVL trees
20	Visiting each node in a tree exactly once is called _____	searching	travering	walk through	path	travering
21	In_____traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order	PreOrder
22	In_____traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order	Inorder
23	In_____traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order	PostOrder
24	Inorder traversal is also sometimes called_____	Symmetric Order	End Order	PreOrder	PostOrder	Symmetric Order
25	Postorder traversal is also sometimes called_____	Symmetric Order	End Order	PreOrder	PostOrder	End Order
26	One can determine whether a Binary tree is a Binary Search Tree by traversing it in	Preorder	Inorder	Postorder	Any order	Inorder

27	Nodes of any level are numbered from _____	Left to right	Right to Left	Top to Bottom	Bottom to Top	Left to right
28	In Threaded Binary Tree ,LCHILD(P) is a normal pointer When LBIT(P) = ____	1	2	3	0	1
29	In Threaded Binary Tree ,LCHILD(P) is a Thread When LBIT(P) = ____	1	2	3	0	0
30	In Threaded Binary Tree ,RCHILD(P) is a normal pointer When RBIT(P) = ____	2	1	3	0	1
31	In Threaded Binary Tree ,RCHILD(P) is a Thread When LBIT(P) = ____	1	2	0	4	0
32	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	m-way search	Binary search
33	_____ algorithm can be used only with sorted lists.	Linear search	Binary search	insertion sort	merge sort	Binary search
34	_____ search involves comparision of the element to be found with every elements in a list.	Linear search	Binary search	fibonacci search	m-way search	Linear search
35	Binary search algorithm in a list of n elements takes only _____ time.	$O(\log_2 n)$	O(n)	$O(n^3)$	$O(n^2)$	$O(\log_2 n)$
36	_____ is used for decision making in eight coin problem.	trees	graphs	linked lists	array	trees
37	The Linear search algorithm in a list of n element takes _____ time to compare in worst case.	constant	linear	quadratic	exponential	constant
38	In _____ search method the search begins by examining the record in the middle of the file.	sequential	fibonacci	binary	non-sequential	binary
39	A binary tree with external nodes added is an ---------- binary tree	extended	expanded	internal	external	extended
40	The search technique for searching a sorted file that requires increased amount of space is	indexed sequential search	interpolation search	sequential search	tree search	indexed sequential search

41	If hl and hr are the heights of the left and right subtrees of a tree respectively and if |hl-hr|<=1 then this tree is called _____	extended binary tree	binary search tree	skewed tree	height balanced tree	height balanced tree
42	If hl and hr are the heights of the left and right subtrees of a tree respectively then |hl-hr| is called its _____	Average height	minimal depth	Maximum levels	Balance factor	Balance factor
43	For an AVL Tree the balance factor is =____	0	2	3	4	0
44	In a binary search tree all values of the left subtree is ____ than the root's value	smaller	larger	equal	multiples	smaller
45	In a binary search tree all values of the right subtree is ____ than the root's value	smaller	larger	equal	multiples	larger
46	In BST, we can search for a value in	O(log n)	O(n)	O(n2)	O(1)	O(log n)
47	In tree construction, which is the suitable efficient data structure?	array	linked list	stack	queue	linked list
48	In a balance binary tree, the height of two subtrees of every node cannot differ by more than?	4	8	5	3	8
49	The number of possible binary trees with 3 nodes is	15	10	8	5	5
50	the number of possible binary trees with 4 nodes is	14	10	15	12	14
51	the order of binary search algorithm is	n	log n	n log n	1	log n
52	A connected graph T without any cycle is called	Tree	stack	queue	graph	tree
53	A binary tree with 10 nodes has _____ null branches	12	11	21	22	21
54	binary search algorithm can be applied to	sorted binary trees	sorted graph	pointer array	sorted linked list	sorted linked list
55	If aaa, bbb and ccc are the elements of a lexically ordered binary tree, then in pre order traversal which node will be traversed first?	aaa	bbb	ccc	cannot be determined	aaa

56	In an array representation of binary tree the right child of the root will be at location of	1	2	3	0	3
57	The maximum number of nodes in a binary tree of depth 5 is	31	16	32	15	31
58	A complete binary tree with n leaf nodes has	n+1 nodes	2n-1 nodes	2n+1 nodes	n(n-1)/2 nodes	2n-1 nodes
59	In a binary tree, certain null entries are replaced by special pointers which point to nodes higher in tree for efficiency. These special pointers are called	leaf	path	branch	thread	thread
60	A binary tree whose every node has either zero or two children is called	complete binary tree	extended binary tree	binary search tree	all of above	extended binary tree
61	when cinverting binary tree into extended binary tree, all the original nodes in binary tree are	internal nodes on extended binary tree	external nodes on extended binary tree	vanished on extended binary tree	vanished on b tree	internal nodes on extended binary tree
62	The inorder traversal will yield a sorting list of elements of tree in	binary tree	binary search tree	heaps	AVL tree	binary search tree
63	Which of the following traversal technique lists the nodes of a binary search tree in ascending order?	postorder	inorder	preorder	insertion	inorder
64	Which of the following need not be a binary tree?	B-Tree	AVL tree	heaps	Search tree	B-Tree
65	A binary tree can be converted in to its mirror image by traversing it in	postorder	inorder	preorder	insertion	preorder
66	The prefix form of A-B/ (C * D ^ E) is,	-/*^ACBDE	-ABCD*^DE	-A/B*C^DE	-A/BC*^DE	-A/B*C^DE
67	Consider that n elements are to be sorted. What is the worst case time complexity of Bubble sort?	O(1)	O(log2n)	O(n)	O(n2)	O(n2)

68	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	Factorial	Binary search
69	_____ are genealogical charts which are used to present the data	Graphs	Pedigree and lineal chart	Line , bar chart	pie chart	Pedigree and lineal chart
70	A __ is a finite set of one or more nodes, with one root node and remaining form the disjoint sets forming the subtrees.	tree	graph	list	set	tree
71	A _____ is a graph without any cycle.	tree	path	set	list	tree
72	In binary trees there is no node with a degree greater than _____	zero	one	two	three	two
73	Which of this is true for a binary tree.	It may be empty	The degree of all nodes must be <=1	It contains a leaf node	The degree of all nodes must be <2	It may be empty
74	Overflow condition in linked list may occur when attempting to_____	Create a node when free space pool is empty.	Traverse the nodes when free space pool is empty.	Create a node when linked list is empty.	Search for node	Create a node when free space pool is empty.
75	The Number of subtrees of a node is called its _____.	leaf	terminal	children	degree	degree

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Under section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

UNIT-V

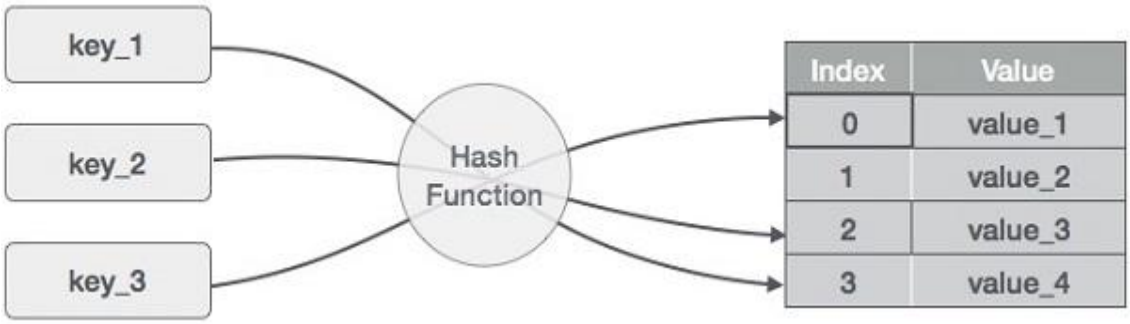
Hashing - Introduction to Hashing, Deleting from Hash Table, Efficiency of Rehash Methods, Hash Table Reordering, Resolving collusion by Open Addressing, Coalesced Hashing, Separate Chaining, Dynamic and Extendible Hashing, Choosing a Hash Function, Perfect Hashing, Function

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



Hash Function

(1,20)
(2,70)
(42,80)

(4,25)
(12,44)
(14,32)
(17,11)
(13,78)
(37,98)

Sr. No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr. No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

delete − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
```

```
  int data;
  int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
  return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
  int key = item->key;

  //get the hash
  int hashIndex = hashCode(key);

  //move in array until an empty
  while(hashArray[hashIndex] !=NULL) {

    if(hashArray[hashIndex]->key == key) {
      struct DataItem* temp = hashArray[hashIndex];

      //assign a dummy item at deleted position
      hashArray[hashIndex] = dummyItem;
      return temp;
    }

    //go to next cell
    ++hashIndex;
```

```
    //wrap around the table
    hashIndex %= SIZE;
  }

  return NULL;
}
```

EFFICIENCY OF REHASH METHODS:

RE-HASHING:

Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found.

Rehashing code:

```
// Grows hash array to twice its original size.
 private void rehash() {
 List<Integer>[] oldElements = elements;
 elements = (List<Integer>[])
 new List[2 * elements.length];
 for (List<Integer> list : oldElements) {
 if (list != null) {
 for (int element : list) {
 add(element);
 }
 }
 }
}
```

Efficiency of rehash methods:

Hash table	
Type	Unordered associative array
Invented	1953

Time complexity in big O notation

Algorithm	Average	Worst Case
Space	O(*n*)	O(*n*)

Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

Hash Table Reordering:

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries n and of the number m of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If m elements are inserted into that table, the total number of extra re-insertions that occur in all dynamic resizings of the table is at most m − 1. In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

Alternatives to all-at-once rehashing:
Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

Disk-based hash tables almost always use some alternative to all-at-once rehashing, since the cost of rebuilding the entire table on disk would be too high.

Incremental resizing:
One alternative to enlarging the table all at once is to perform the rehashing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move r elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least (r + 1)/r during resizing.

RESOLVING COLLUSION :

When two different keys produce the same address, there is a **collision**. The keys involved are called **synonyms**. Coming up with a hashing function that avoids collision is extremely difficult. It is best to simply find ways to deal with them. **The possible solution, can be:**

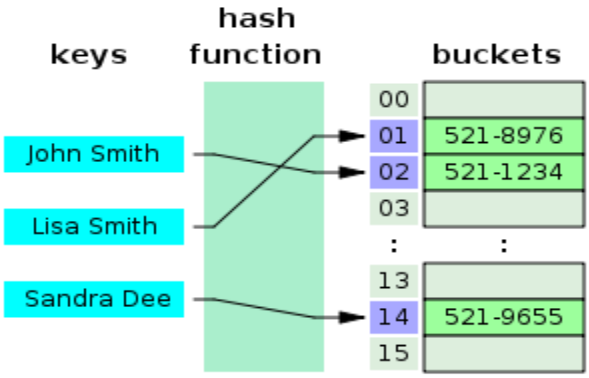
Spread out the records

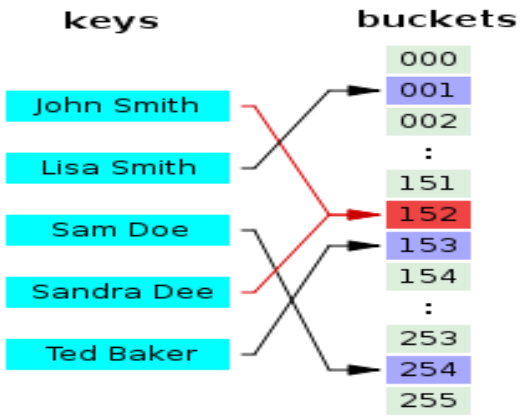
Use extra memory
Put more than one record at a single address.
An example of Collision

Hash table size: 11
Hash function: key mod hash size
So, the new positions in the hash table are:

Key	23	18	29	28	39	13	16	42	17
Position	1	7	7	6	6	2	5	9	6

Some collisions occur with this hash function as shown in the above figure.
Another example (in a phonebook record):





Here, the buckets for keys 'John Smith' and 'Sandra Dee' are the same. So, its a collision case.

Collision Resolution: Collision occurs when h(k1) = h(k2), i.e. the hash function gives the same result for more than one key. The strategies used for collision resolution are:

- **Chaining**
  - Store colliding keys in a linked list at the same hash table index
- **Open Addressing**
  - Store colliding keys elsewhere in the table

Chaining:

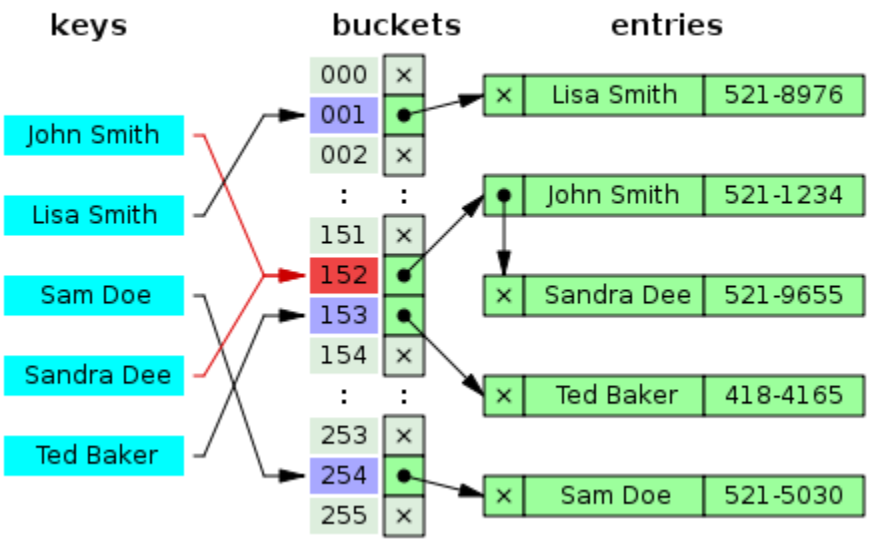


Fig: Separate Chaining

Strategy:

Maintains a linked list at every hash index for collided elements.

Lets take the example of an insertion sequence: {0 1 4 9 16 25 36 49 64 81}.

Here, h(k) = k mod tablesize = k mod 10 (tablesize = 10)

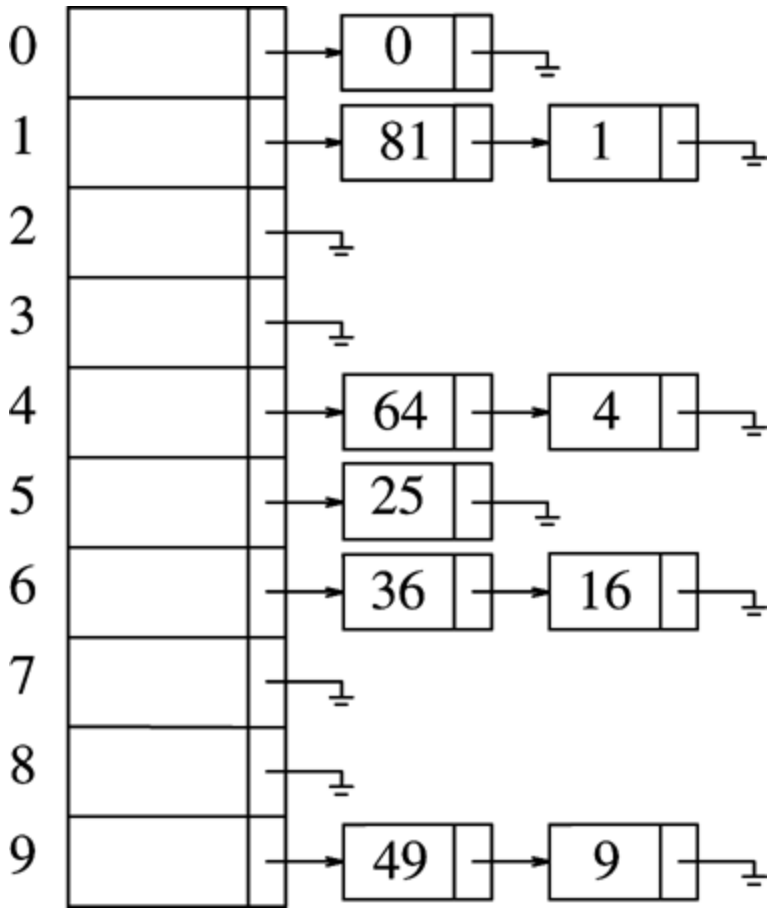
Hash table T is a vector of linked lists

Insert element at the head (as shown here) or at the tail

Key k is stored in list at T[h(k)]

So, the problem is like: "Insert the first 10 preface squares in a hash table of size 10"

The hash table looks like:



Collision Resolution by Chaining: Analysis

- **Load factor** λ of a hash table T is defined as follows:
  N = number of elements in T ("current size")
  M = size of T ("table size")
  λ = N/M (" load factor")
  i.e., λ is the average length of a chain

- Unsuccessful search time: O(λ)
  Same for insert time

- Successful search time: O(λ/2)

- Ideally, want λ ≤ 1 (not a function of N)

Potential diadvantages of Chaining

- Linked lists could get long
  Especially when N approaches M
  Longer linked lists could negatively impact performance
- More memory because of pointers

- Absolute worst-case (even if N << M):
  All N elements in one linked list!
  Typically the result of a bad hash function

Open Addressing:

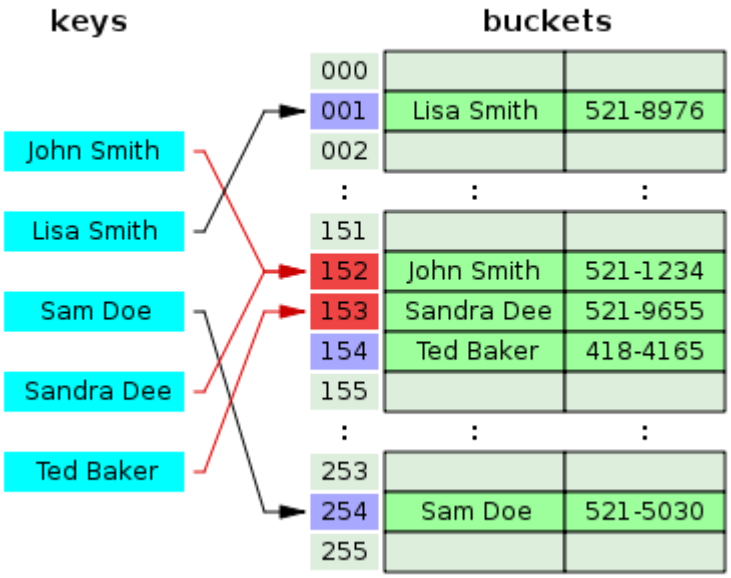


Fig: Open Addressing

As shown in the above figure, in open addressing, when collision is encountered, the next key is inserted in the empty slot of the table. So, it is an 'inplace' approach.

Advantages over chaining

- No need for list structures
- No need to allocate/deallocate memory during insertion/deletion (slow)

Diadvantages

- Slower insertion – May need several attempts to find an empty slot
- Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance
  Load factor $\lambda \approx 0.5$

Probing

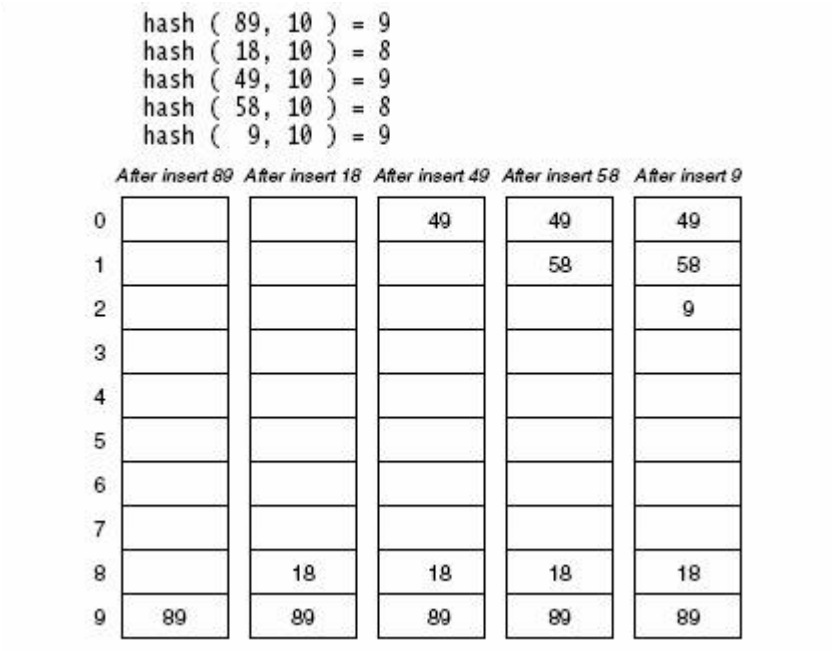
The next slot for the collided key is found in this method by using a technique called **"Probing".** It generates a probe sequence of slots in the hash table and we need to chose the proper slot for the key 'x'.

- h0(x), h1(x), h2(x), …
- Needs to visit each slot exactly once
- Needs to be repeatable (so we can find/delete what we've inserted)
- Hash function
    - hi(x) = (h(x) + f(i)) mod TableSize
    - f(0) = 0 ==> position for the 0th probe
    - f(i) is "the distance to be traveled relative to the 0th probe position, during the ith probe".

Some of the common methods of probing are:

1. Linear Probing:

Suppose that a key hashes into a position that has been already occupied. The simplest strategy is to look for the next available position to place the item. Suppose we have a set of hash codes consisting of {89, 18, 49, 58, 9} and we need to place them into a table of size 10. The following table demonstrates this process.



The first collision occurs when 49 hashes to the same location with index 9. Since 89 occupies the A[9], we need to place 49 to the next available position. Considering the array as circular, the next available position is 0. That is (9+1) mod 10. So we place 49 in A[0].

Several more collisions occur in this simple example and in each case we keep looking to find the next available location in the array to place the element. Now if we need to find the element, say for example, 49, we first compute the hash code (9), and look in A[9]. Since we do not find it there, we look in A[(9+1) % 10] = A[0], we find it there and we are done.

So what if we are looking for 79? First we compute hashcode of 79 = 9. We probe in A[9], A[(9+1)]=A[0], A[(9+2)]=A[1], A[(9+3)]=A[2], A[(9+4)]=A[3] etc. Since A[3] = null, we do know that 79 could not exists in the set.

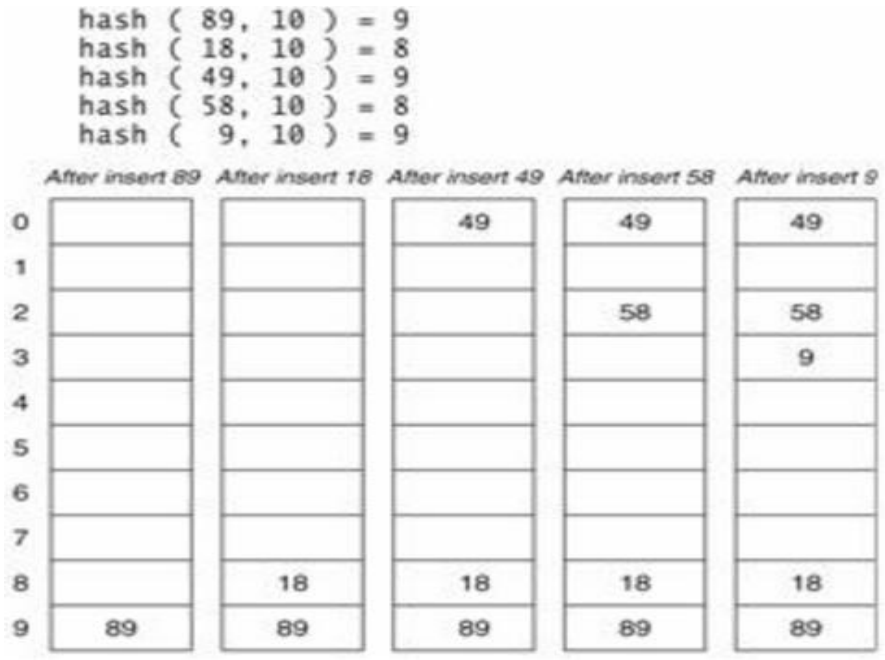
Issues with Linear Probing:

- Probe sequences can get longer with time
- Primary clustering
    - Keys tend to cluster in one part of table
    - Keys that hash into cluster will be added to the end of the cluster (making it even bigger)
    - Side effect: Other keys could also get affected if mapping to a crowded neighborhood

2. Quadratic Probing:

Although linear probing is a simple process where it is easy to compute the next available location, linear probing also leads to some clustering when keys are computed to closer values. Therefore we define a new process of Quadratic probing that provides a better distribution of keys when collisions occur. In quadratic probing, if the hash value is K , then the next location is computed using the sequence K + 1, K + 4, K + 9 etc..

The following table shows the collision resolution using quadratic probing.



- Avoids primary clustering
- f(i) is quadratic in i: eg: $f(i) = i^2$
- $h_i(x) = (h(x) + i^2)$ mod tablesize

Quadratic Probing: Analysis

- Difficult to analyze
- Theorem
  New element can always be inserted into a table that is at least half empty and TableSize is prime
- Otherwise, may never find an empty slot, even is one exists

- Ensure table never gets half full
  If close, then expand it
- May cause "secondary clustering"
- Deletion
  Emptying slots can break probe sequence and could cause find stop prematurely
- Lazy deletion:Differentiate between empty and deleted slot
  When finding skip and continue beyond deleted slots
  If you hit a non-deleted empty slot, then stop find procedure returning "not found"
- May need compaction at some time

3. Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

4. Hashing with Rehashing:

Once the hash table gets too full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name rehashing)
- This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is ok though since it doesn't happen that often.

Coalesced Hashing:

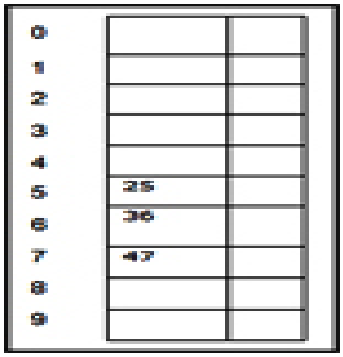
The chaining method discussed above requires additional space for maintaining pointers. The table stores only pointers but each node of the linked list requires storage space for data as well as one pointer field. Thus, for n keys, n + MAX_SIZE pointers are needed, where MAX_SIZE is the maximum size of the table in which

values are to be inserted. If the value of n is large, the space required to store this table is quite large.

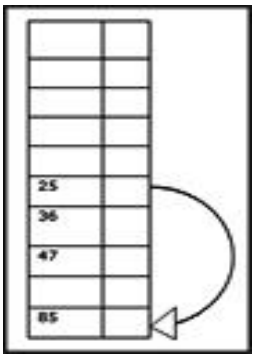
The solution to this problem is called coalesced hashing or coalesced chaining. This method is the hybrid of chaining and open addressing. Each index position in the table stores key value and a pointer to the next index position. The pointer generally points to the index position where the colliding key value will be stored.

In this method, the next available position is searched for a colliding key and is placed in that position. After each such insertion, pointer re – adjustment is required. After inserting the key values at the right place, the next pointer of the previous position is made to point to the position where the colliding key is inserted. In this method, instead of allocating new nodes for the linked list of keys with collision, empty position from the table itself is allocated.

For Example, the values 25, 36, and 47 will be inserted thus in the table –

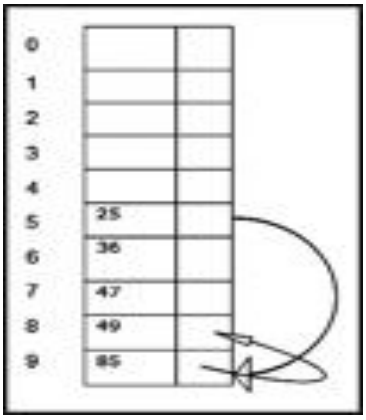


Now, we insert key value 85 into this table. This method starts inserting the collided key values from the bottom of the table. Key value 85 will go in at index position 9 in the table and the pointer will be re – adjusted. That is, the next pointer of position 5 will point to index position 9.



Index position 9 is full and any key value hashing into this position will have to be inserted into the next available empty location, starting from the bottom of the table.

So, if we insert key value 49 into the table, it will go into index position 8 with pointer re – adjustment. The table will look like –



This process will continue for all the colliding key values.

DYNAMIC AND EXTENDIBLE HASHING:

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

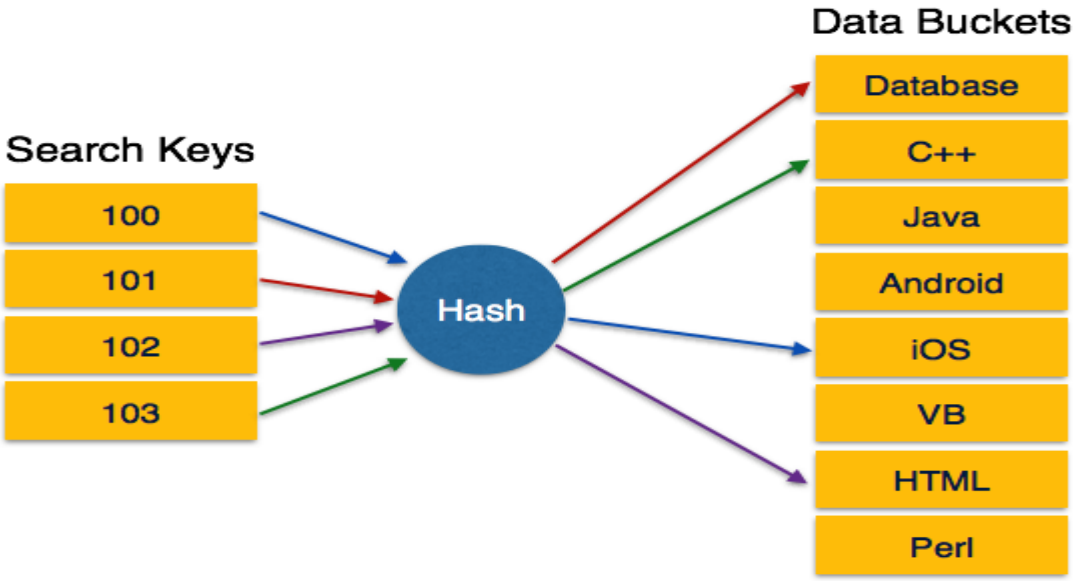
Hash Organization:

Bucket − A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function − A hash function, h, is a mapping function that maps all the set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



Operation

- **Insertion** − When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

  Bucket address = h(K)

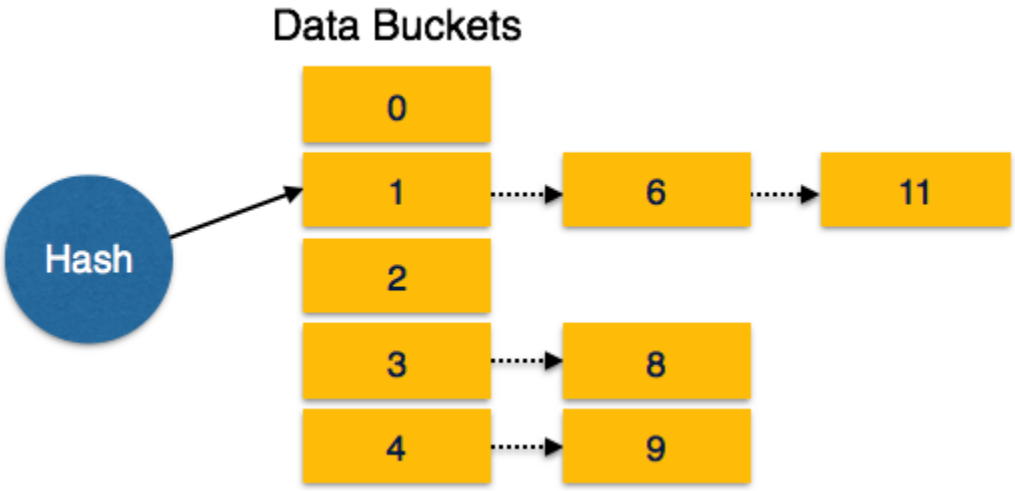
- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

- **Delete** − This is simply a search followed by a deletion operation.

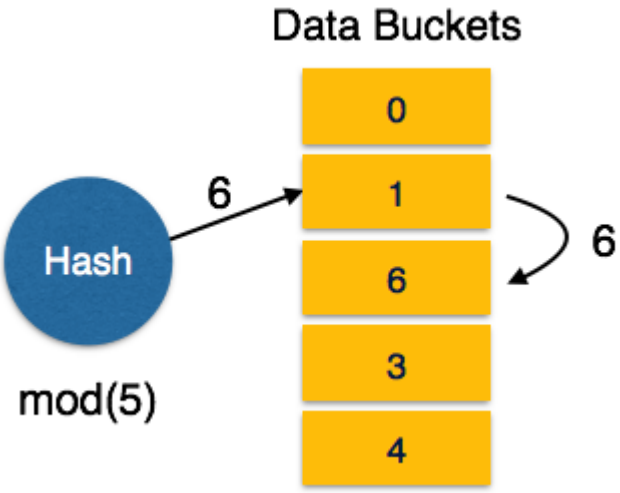
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** − When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



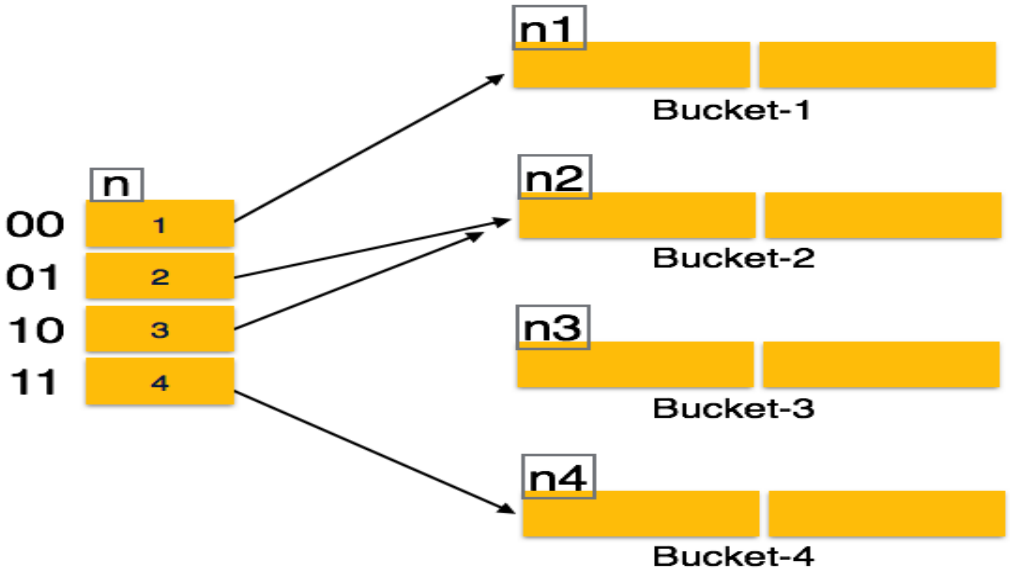
- **Linear Probing** − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing:

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.

Operation

Querying − Look at the depth value of the hash index and use those bits to compute the bucket address.

Update − Perform a query as above and update the data.

Deletion − Perform a query to locate the desired data and delete the same.

Insertion − Compute the address of the bucket

- If the bucket is already full.
    1. Add more buckets.

2. Add additional bits to the hash value.
3. Re-compute the hash function.

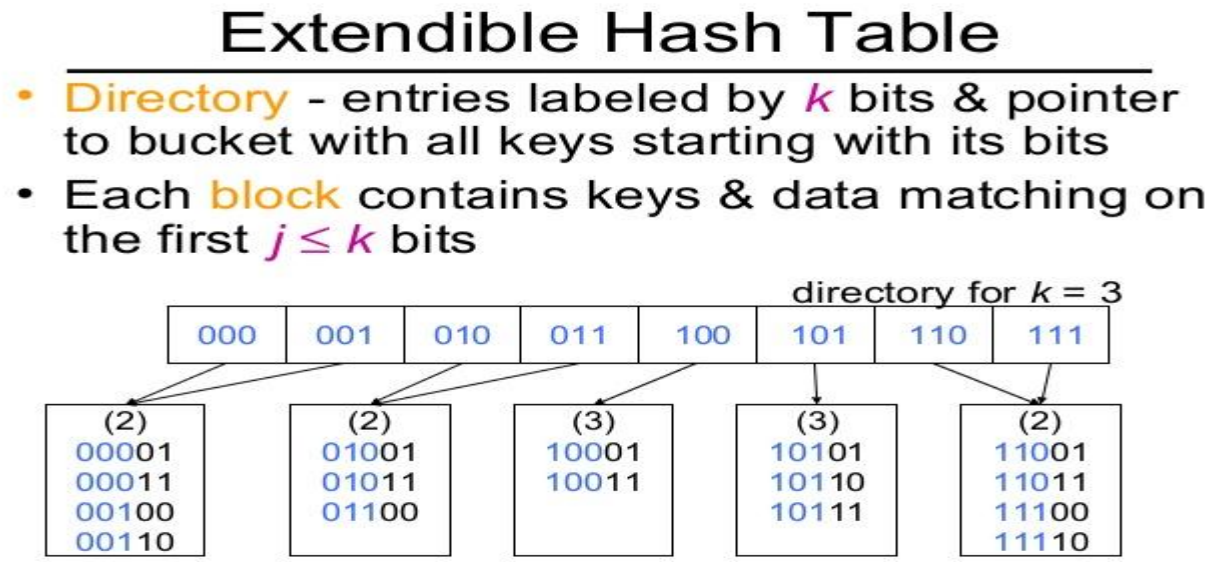
- Else
  1. Add data to the bucket,
- If all the buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

Extendible hashing:

Extendible hashing is a type of hash system which treats a hash as a bit string, and uses a trie for bucket lookup. Because of the hierarchical nature of the system, re-hashing is an incremental operation (done one bucket at a time, as needed). This means that time-sensitive applications are less affected by table growth than by standard full-table rehashes.



Choosing a Hash Function:

Choosing a good hash function is of the utmost importance. An **uniform** hash function is one that equally distributes data items over the whole hash table data structure. If the hash function is poorly chosen data items may tend to **clump** in one area of the hash table and many collisions will ensue. A non-uniform dispersal pattern and a high collision rate cause an overall data structure performance degradation. There are several strategies for maximizing the uniformity of the hash function and thereby maximizing the efficiency of the hash table.

One method, called the **division method** , operates by dividing a data item's key value by the total size of the hash table and using the remainder of the division as the hash function return value. This method has the advantage of being very simple to compute and very easy to understand.

Selecting an appropriate hash table size is an important factor in determining the efficiency of the division method. If you choose to use this method, avoid hash table sizes that simply return a subset of the data item's key as the hash value. For instance, a table one-hundred items large will result put key value 12345 at location forty-five, which is undesirable. Further, an even data item key should not always map to an even hash value (and, likewise, odd key values should not always produce odd hash values). A good rule of thumb in selecting your hash table size for use with a division method hash function is to pick a prime number that is not close to any power of two (2, 4, 8, 16, 32...).

```
int hash_function(data_item item)

{

  return item.key % hash_table_size;

}
```

Sometimes it is inconvenient to have the hash table size be prime. In certain cases only a hash table size which is a power of two will work. A simple way of dealing with table sizes which are powers of two is to use the following formula to computer a key: k = (x mod p) mod m. In the above expression x is the data item key, p is a prime number, and m is the hash table size. Choosing p to be much larger than m improves the uniformity of this key selection process.

Yet another hash function computation method, called the **multiplication method**, can be used with hash tables with a size that is a power of two. The data item's key is multiplied by a constant, k and then bit-shifted to compute the hash function return value.

A good choice for the constant, k is N * (sqrt(5) - 1) / 2 where N is the size of the hash table.

The product key * k is then bitwise shifted right to determine the final hash value. The number of right shifts should be equal to the log2 N subtracted from the number of bits in a data item key. For instance, for a 1024 position table (or 210) and a 16-bit data item key, you should shift the product key * k right six (or 16 - 10) places.

```
int hash_function(data_item item)

{

  extern int constant;

  extern int shifts;

 return (int)((constant * item.key) >> shifts);

}
```

Note that the above method is only effective when all data item keys are of the same, fixed size (in bits). To hash non-fixed length data item keys another method is **variable string addition** so named because it is often used to hash variable length strings. A table size of 256 is used. The hash function works by first summing the ASCII value of each character in the variable length strings. Next, to determine the hash value of a given string, this sum is divided by 256. The remainder of this division will be in the range of 0 to 255 and becomes the item's hash value.

```
int hash_function (char *str)

{

  int total = 0;

 while (*str) {

    total += *str++;

  }

  return (total % 256);

}
```

Yet another method for hashing non fixed-length data is called **compression function** and discussed in the one-way hashing section.

Perfect hash function:

In computer science, a **perfect hash function** for a set *S* is a hash function that maps distinct elements in *S* to a set of integers, with no collisions. In mathematical terms, it is an injective function.

- In most general applications, we cannot know exactly what set of key values will need to be hashed until the hash function and table have been designed and put to use.
- At that point, changing the hash function or changing the size of the table will be extremely expensive since either would require re-hashing every key.

- A **perfect hash function** is one that maps the set of actual key values to the table without any collisions.
- A **minimal perfect hash function** does so using a table that has only as many slots as there are key values to be hashed.
- If the set of keys IS known in advance, it is possible to construct a specialized hash function that is perfect, perhaps even minimal perfect.
- Algorithms for constructing perfect hash functions tend to be tedious, but a number are known.

Dynamic perfect hashing:

Using a perfect hash function is best in situations where there is a frequently queried large set, S, which is seldom updated. This is because any modification of the set S may cause the hash function to no longer be perfect for the modified set. Solutions which update the hash function any time the set is modified are known as dynamic perfect hashing, but these methods are relatively complicated to implement.

Minimal perfect hash function

A **minimal perfect hash function** is a perfect hash function that maps n keys to n consecutive integers – usually the numbers from 0 to n − 1 or from 1 to n. A more formal way of expressing this is: Let j and k be elements of some finite setS. F is a minimal perfect hash function if and only if F(j) = F(k) implies j = k (injectivity) and there exists an integer a such that the range of F is a..a + |S| − 1.

Order preservation

A minimal perfect hash function F is order preserving if keys are given in some order a_1, a_2, ..., a_n and for any keys a_j and a_k, j < k implies $F(a_j) < F(a_k)$. In this case, the function value is just the position of each key in the sorted ordering of all of the keys. A simple implementation of order-preserving minimal perfect hash functions with constant access time is to use an (ordinary) perfect hash function or cuckoo hashing to store a lookup table of the positions of each key. If the keys to be hashed are themselves stored in a sorted array, it is possible to store a small number of additional bits per key in a data structure that can be used to compute hash values quickly. Order-preserving minimal perfect hash functions require necessarily Ω(n log n) bits to be represented.

KARPAGAM ACADEMY OF HIGHER EDUCATION
Coimbatore - 641021.
(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT : DATA STRUCTURES **SEMESTER**: III
CODE: 17CAU301 **CLASS**: II B.C.A

POSSIBLE QUESTIONS

UNIT-V

2 MARKS:

1. What is Hashing?
2. Explain about Hash Table.
3. Define Hash Function.
4. Write about Resolving Collisions.
5. Write about Separate Chaining.

6 MARKS:

1. Write about Deleting from Hash Table.
2. Discuss about Efficiency of Rehash Methods.
3. Discuss about Resolving Collusion by Open Addressing.
4. What is Coalesced Hashing
5. What is Resolving Collusion by Open Addressing.

Karpagam Academy of Higher Education
Department of Computer Applications
Subject : Data Structures

Class: II BCA **Subject code: 17CAU301**

Objective Type Questions

UNIT V

S.No	QUESTION	OPT 1	OPT 2	OPT 3	OPT 4	ANSWER
1	The postfix form of A*B+C/D is	*AB/CD+	AB*CD/+	A*BC+/D	ABCD+/*	AB*CD/+
2	A characteristic of the data that binary search uses but the linear search ignores is the_____.	Order of the elements of the list.	Length of the list.	Maximum value in list.	Type of elements of the list.	
3	In order to get the contents of a Binary search tree in ascending order, one has to traverse it in	pre-order	in-order	post order	not possible.	in-order
4	Which of the following sorting algorithm is stable	insertion sort.	bubble sort.	quick sort	heap sort.	heap sort.
5	The time required to delete a node x from a doubly linked list having n nodes is	O (n)	O (log n)	O (1)	O (n log n)	O (1)
6	Which of these sorting algorithm uses the Divide and Conquere technique for sorting	selection sort	insertion sort	merge sort	heap sort	merge sort
7	A_____ is defined to be a complete binary tree with the property that the value of root node is at least as large as the value of its children node.	quick	radix	merge	heap	heap
8	Binary trees are used in _____ sorting.	quick sort	merge sort	heap sort	lrsort	heap sort
9	The ____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch	Root
10	_____ is a internal sorting method.	sorting with disks	quick sort	balanced merge sort	sorting with tapes	quick sort

11	Quick sort reads _____ space to implement the recursion.	stack	queue	circular stacks	circular queue	stack
12	The most popular method for sorting on external storage devices is _____.	quick sort	radix sort	merge sort	heap sort	merge sort
13	The 2-way merge algorithm is almost identical to the _____procedure.	quick	merge	heap	radix	merge
14	In threaded binary tree all right child pointer points to	postorder successor of the node	Inorder successor of the node	Inorder predecessor of the node	Preorder successor of the node	
15	What is the A of AVL tree?	part of approach	name of one of the inventors	algorithmic abbreviation	property of tree	name of one of the inventors
16	In AVL Tree the difference between height of left and right subtree is never more than	0	2	3	1	1
17	In an AVL tree, if the balance factor becomes 2 or -2 then the tree rooted at this node is	balanced	unbalanced	pruned	rotated	unbalanced
18	Which of the following is not the required condition for binary search algorithm?	The list must be sorted	there should be the direct access to the middle element in any sublist	There must be mechanism to delete and/or insert elements in list	none of above	There must be mechanism to delete and/or insert elements in list
19	Which of the following is not a limitation of binary search algorithm?	must use a sorted array	requirement of sorted array is expensive when a lot of insertion and deletions are needed	there must be a mechanism to access middle element directly	binary search algorithm is not efficient when the data elements are more than 1000.	binary search algorithm is not efficient when the data elements are more than 1000.

20	If a node having two children is deleted from a binary tree, it is replaced by its	Inorder predecessor	Inorder successor	Preorder predecessor	Preorder successor of the node	Inorder successor
21	The postfix form of the expression (A + B) (C D − E) F / G is	AB + CD E − FG /	AB + CD E − F G/	AB + CD E − F G/	AB + CDE − F G/	AB + CD E − FG /
22	In worst case Quick Sort has order	O (n log n)	O (n2 /2)	O (log n)	O (n2 /4)	O (n2 /2)
23	The complexity of searching an element from a set of n elements using Binary search algorithm is	O (log n)	O (n2 /2)	O (log n)	O (n2 /4)	O(log n)
24	Which of the following sorting methods would be most suitable for sorting a list which is almost sorted	bubble sort	quick sort	merge sort	radix sort	bubble sort
25	The pre-order and post order traversal of a Binary Tree generates the same output. The tree can have maximum _____ nodes	1	2	3	4	1
26	The disadvantage of _____ sort is that is need a temporary array to sort.	Quick	Merge	Heap	Insertion	Merge
27	_____ techiniques are used for sorting large files	Topological sort	External sorting	Linear Sorting	Heap sort	External sorting
28	Sorting is not possible by using which of the following methods?	Insertion	Selection	radix	Deletion	Deletion
29	which of the following sorting procedure is slowest?	quick sort	heap sort	shell sort	bubble sort	bubble sort
30	The space factor when determining the efficiency of the algorithm is measured by	counting the maximum memory needed by the algorithm	counting the minimum memory needed by the algorithm	counting the average memory needed by the algorithm	counting the maximum disk space needed by the algorithm	counting the maximum memory needed by the algorithm

31	The worst case occur in linear search algorithm when	item is in middle of array	item is not in array	item is in last element	item is in first of the array	item is in last element
32	which of the following sorting algorithm is divide and conquer type?	bubble sort	insertion sort	quick sort	selection sort	quick sort
33	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	insertion sort.	selection sort	heap sort.	quick sort.	quick sort.
34	An undirected graph G with n vertices and e edges is represented by adjacency list. What is the time required to generate all the connected components?	O (n)	O €	O (e+n)	O (n +2)	O (e+n)
35	A graph with n vertices will definitely have a parallel edge or self loop of the total number of edges are	more than n	more than n+1	more than (n+1)/2	more than n(n-1)/2	more than n(n-1)/2
36	The maximum degree of any vertex in a simple graph with n vertices is	n-1	n+1	2n-1	2n+1	n-1
37	The data structure required for Breadth First Traversal on a graph is	queue	stack	Arrays	tree	queue
38	The quick sort algorithm exploit _____ design technique	) Greedy	Dynamic programming	Divide and Conquer	Backtracking	Divide and Conquer
39	A graph with n vertices will definitely have a parallel edge or self loop if the total number of edges are	greater than n–1	less than n(n–1)	greater than n(n–1)/2	less than n2 /2	greater than n–1
40	The complexity of Bubble sort algorithm is	O(n)	O(log n)	O(n2)	O(n log n)	O(log n)
41	The complexity of merge sort algorithm is	O(n)	O(log n)	O(n2)	O(n log n)	O(n log n)
42	The complexity of linear search algorithm is	O(n)	O(log n)	O(n2)	O(n log n)	O(n)

43	Graphs are represented using	12	Adjacency linked list	Adjacency graph	Adjacency queue	Adjacency linked list
44	The average case complexity of Insertion Sort is	O(2n)	O(n3)	O(n2)	O(2n)	O(n2)
45	The sorting technique where array to be sorted is partitioned again and again in such a way that all elements less than or equal to partitioning element appear before it and those which are greater appear after it, is called	merge sort	quick sort	selection sort	bubble sort	quick sort

Register Number_____

[17CAU0301]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

Coimbatore - 641021.

COMPUTER APPLICATIONS

Third Semester

First Internal

Data Structures

Date & Session: **Class : II BCA**

Duration: 2 Hours **Maximum : 50 Marks**

Answer Key

PART-A (20 X 1 = 20 Marks)
Answer ALL the Questions

1. In Stack we can add elements at _____.

   a)Bottom **b)Top** b)Front d)Rear

2. In Stack we can delete elements at _____

   a)Front b)Rear **c)Top** d)Bottom

3. When Top = Bottom in stack, the total no of element in the stack is

   a)1 b)2 c)3 d)0

4. When FRONT = REAR in queue, the total no of element in the queue is

   a)0 **b)1** c)2 d)3

5. In Stack the TOP is decremented by one after every ___ operation.

   a)AddQ **b)Pop** c)Push d)DelQ

6. In Stack the TOP is incremeted by one before every ___ operation.

   a)AddQ b)Pop **c)Push** d)DelQ

7. Data structure used for recursion is

   a)stack b)queue **c)array** d)linked list

8. The data structure required to check whether an expression contains balanced parenthesis is?

   a)stack b)queue c)array d)linked list

9. Removing the element from the top of the stack is called the

   a)push **b)pop** c)delete d)remove

10. What data structure would you mostly likely see in a non recursive implementation of a recursive algorithm?

    a)stack b)queue c)array d)linked list

11. Which data structure is used to implement the queue most efficiently

    a)array **b)linked list** c)structure d)union

12. The process of accessing data stored in a serial access memory is similar to manipulating data on a --?

    a)stack b)queue c)array **d)linked list**

13. Which data structure is used to implement the double ended queue?

a)doubly linked list b)structure c)Trees d)graphs

14. What is the data structure used to implement circular queue?

    a) circular linked list b)singly linked list c)double linked list d)multi list

15. Which data structure allows deleting data element from front and inserting at rear?

    a)stack **b)queue** c)dequeue d)binary search tree

16. A _____ is a procedure or function which calls itself.

    a)Stack **b)Recursion** c)Queue d)Recursion

17. An example for application of stack is _____.

    a) Train **b)piles of plates** c) tree d)ticket counter

18. What is the strategy of Stack?

    a)LILO **b)FIFO** c)FILO d)LIFO

19. Inserting element in a stack is known as

    a) Insertion **b) Push** c) pop d) Addition

20. deleting element from a stack is known as

    a) addition **b) pop** c) push d)deletion

PART-B (3 X 2 = 6 Marks)
(Answer ALL the Questions)

21. **What is a data structure? Give examples.**

- A **data structure** is a specialized format for organizing and storing **data**.

22. **Define : Sparse Matrix**

- A **matrix** is a two-dimensional **data** object made of m rows and n columns, therefore having total m x n values. If most of the elements of the **matrix** have 0 value, then it is called a **sparse matrix**.

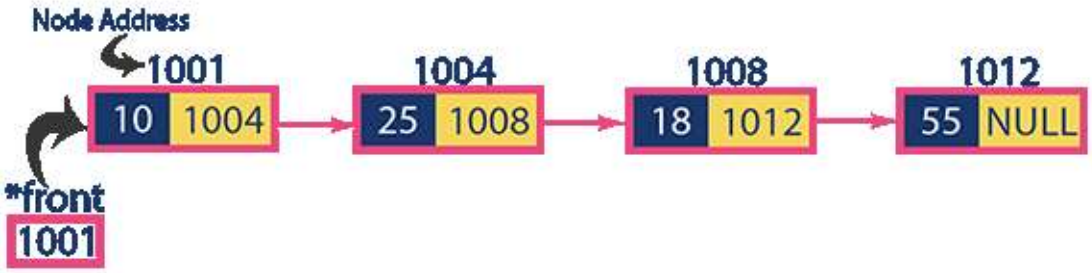
23. **What is prefix and postfix expression?**

- Operators are written after their operands.
- Operators are written before their operands.

PART-C (3 X 8 = 24 Marks)
(Answer ALL the Questions)

24. **a) Explain the concept of linked list with example. (OR)**

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

- In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.
- Example
- 

- Operations
- In a single linked list we perform the following operations...
    - Insertion
    - Deletion
    - Display

b) List the basic operations in stack with example.

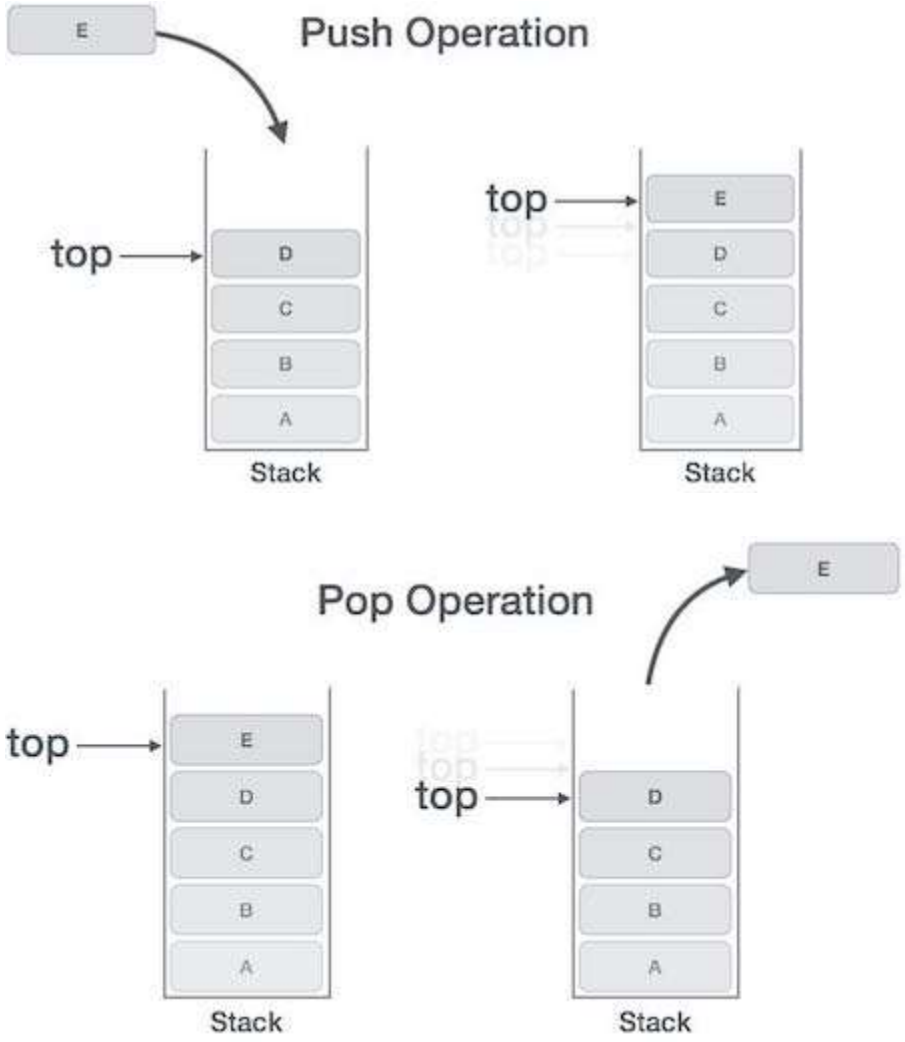
Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.



25. **a). Write a C program for implementing stack using array. (OR)**

```
#include<stdio.h>
 int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t----------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
```

```
            }
            case 2:
            {
               pop();
               break;
            }
            case 3:
            {
               display();
               break;
            }
            case 4:
            {
               printf("\n\t EXIT POINT ");
               break;
            }
            default:
            {
               printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
            }

         }
      }
      while(choice!=4);
      return 0;
}
void push()
{
      if(top>=n-1)
      {
         printf("\n\tSTACK is over flow");

      }
      else
      {
         printf(" Enter a value to be pushed:");
         scanf("%d",&x);
         top++;
         stack[top]=x;
      }
}
void pop()
{
      if(top<=-1)
      {
         printf("\n\t Stack is under flow");
      }
      else
      {
         printf("\n\t The popped elements is %d",stack[top]);
         top--;
      }
}
void display()
{
      if(top>=0)
      {
         printf("\n The elements in STACK \n");
         for(i=top; i>=0; i--)
```

```
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }

}
```

b).Write a C program for implementing stack using linked list

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int info;
    struct node *ptr;
}*top,*top1,*temp;

int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();

int count = 0;

void main()
{
    int no, ch, e;

    printf("\n 1 - Push");
    printf("\n 2 - Pop");
    printf("\n 3 - Top");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Dipslay");
    printf("\n 7 - Stack Count");
    printf("\n 8 - Destroy stack");

    create();

    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            push(no);
            break;
        case 2:
```

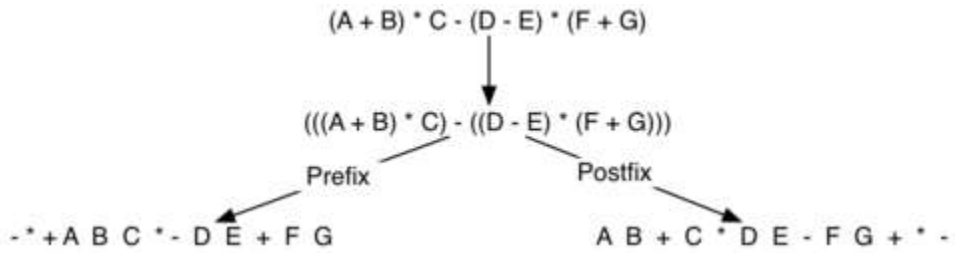
```
            pop();
            break;
        case 3:
            if (top == NULL)
                printf("No elements in stack");
            else
            {
                e = topelement();
                printf("\n Top element : %d", e);
            }
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
        case 6:
            display();
            break;
        case 7:
            stack_count();
            break;
        case 8:
            destroy();
            break;
        default :
            printf(" Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* Create empty stack */
void create()
{
    top = NULL;
}

/* Count stack elements */
void stack_count()
{
    printf("\n No. of elements in stack : %d", count);
}

/* Push data into stack */
void push(int data)
{
    if (top == NULL)
    {
        top =(struct node *)malloc(1*sizeof(struct node));
        top->ptr = NULL;
        top->info = data;
    }
    else
    {
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
```

```
    }
    count++;
}

/* Display stack elements */
void display()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("Stack is empty");
        return;
    }

    while (top1 != NULL)
    {
        printf("%d ", top1->info);
        top1 = top1->ptr;
    }
 }

/* Pop Operation on stack */
void pop()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("\n Error : Trying to pop from empty stack");
        return;
    }
    else
        top1 = top1->ptr;
    printf("\n Popped value : %d", top->info);
    free(top);
    top = top1;
    count--;
}

/* Return top element */
int topelement()
{
    return(top->info);
}

/* Check if stack is empty or not */
void empty()
{
    if (top == NULL)
        printf("\n Stack is empty");
    else
        printf("\n Stack is not empty with %d elements", count);
}

/* Destroy entire stack */
void destroy()
{
    top1 = top;
```

```
    while (top1 != NULL)
    {
       top1 = top->ptr;
       free(top);
       top = top1;
       top1 = top1->ptr;
    }
    free(top1);
    top = NULL;

    printf("\n All stack elements destroyed");
    count = 0;
}
```

26. **a). Infix notation: (A + B) * C - (D - E) * (F + G)**

Convert this into prefix and postfix notation (**OR)**



b) What is circular list? Explain with example.

Circular linked list *is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*

Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue.

3) Circular lists are useful in applications to repeatedly go around the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

****

Reg No. ……..

[17CAU301]

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 64 021
BCA Degree Examination
(For the candidates admitted from 2017 onwards)
Third Semester
Second Internal Exam August 2018
Data Structures

Duration: 2 Hrs **Maximum Marks: 50 Marks**
Date & Session: **Class: II BCA**

Answer Key

Part - A (20 X 1 = 20 Marks)
(Answer all the Questions)

1. Nodes that have degree zero are called _____.
   a) end node **b)leaf nodes** c)subtree d)root node
2. All node except the leaf nodes are called_____.
   a)terminal node b)percent node **c)non terminal** d)children node
3. X is a root then X is the _____ of its children.
   a) sub tree **b)Parent** c)Sibilings d)subordinate
4. A tree with any node having at most two branches is called a _____.
   a)branched tree b)sub tree **c)binary tree** d)forest
5. In _____ graph the pair of vertices joined by any edge is unordered.
   a)directed **b)undirected** c)sub d)multi
6. A _____ is a graph without any cycle.
   a)tree **b)**path c)set d)list
7. In binary trees there is no node with a degree greater than _____
   a)zero b)one c)two d)three
8. The Number of subtrees of a node is called its _____.
   a)leaf b)terminal c)children d)**degree**
9. In a graph G(V,E), V is a finite non-empty set of _____ and E is a set of edges.
   a)Nodes b)Items **c)Vertices** d)Circles
10. In a undirected graph G two vertices v1 and v2 are said to be _____if there is a path in G from v1 to v2.
    a)connected b)adjacent c)neighbours d)incident
11. A _____of depth k is a binary tree of depth k having 2^K-1 nodes.
    a)full binary tree b)half binary tree c)sub tree d) n branch tree
12. Data structure represents the hierarchical relationships between individual data item is known as _____.
    a)Root b)Node **c)Tree** d)Address
13. The children of the same parent are called _____.
    a)sibiling b)leaf c)child d) subtree
14. Node at the highest level of the tree is known as _____.
    a)Child **b)Root** c)Sibiling d)Parent
15. A _____ is a connected acyclic graph.
    a)graph b)component **c)tree** d)list

16. In a graph ____ of a vertex is the number of edges incident to it.
a)path **b)degree** c)depth d)height
17. In a Graph G if there are n vertices the adjacency Matrix of the graph consists of _____ rows and colums.
a)n/2 b)2n c)n-1 **d)n**
18. Directed graph is also called _____.
a)Line Graph b)sub graph c)connected graph **d)di graph**
19. A graph with weighted edge is called a _____.
a)strong graph **b)network** c)component d)sub graph
20. Visiting each node in a tree exactly once is called _____
a)searching **b)travering** c)walk through d)path

Part - B (3 X 2=6 Marks)
(Answer all the Questions)

21. What is binary tree? Give an example.

- A **binary tree** is a **tree** data structure in which each node has at most two children, which are referred to as the left child and the right child.

22. Differentiate height and depth of the tree.

- The **depth** of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0.
- The **height** of a node is the number of edges on the longest path from the node to a leaf. A leaf node will have a height of 0.

23. Define: De-queue

- A **deque**, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection.

Part – C (3 X 8=24 Marks)
(Answer all the Questions)

24. a).Write a C program to calculate the GCD of 2 numbers with recursion and without recursion.

```
#include<stdio.h>

// declaring the recursive function
int find_gcd(int , int );

int main()
{
   printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
   int a, b, gcd;
   printf("\n\nEnter two numbers to find GCD of \n");
   scanf("%d%d", &a, &b);
   gcd = find_gcd(a, b);
   printf("\n\nGCD of %d and %d is: %d\n\n", a, b, gcd);
   printf("\n\n\t\t\tCoding is Fun !\n\n\n");
   return 0;
```

```
}

// defining the function
int find_gcd(int x, int y)
{
   if(x > y)
      find_gcd(x-y, y);

   else if(y > x)
      find_gcd(x, y-x);
   else
   return x;
}
```

(OR)

b).Write a C program to implement binary search tree.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct BST
{
   int data;
   struct BST *left;
   struct BST *right;
}node;

node *create();
void insert(node *,node *);
void preorder(node *);

int main()
{
   char ch;
   node *root=NULL,*temp;

   do
   {
      temp=create();
      if(root==NULL)
         root=temp;
      else
         insert(root,temp);

      printf("nDo you want to enter more(y/n)?");
      getchar();
      scanf("%c",&ch);
   }while(ch=='y'|ch=='Y');

   printf("nPreorder Traversal: ");
   preorder(root);
```

```
    return 0;
}

node *create()
{
    node *temp;
    printf("nEnter data:");
    temp=(node*)malloc(sizeof(node));
    scanf("%d",&temp->data);
    temp->left=temp->right=NULL;
    return temp;
}

void insert(node *root,node *temp)
{
    if(temp->data<root->data)
    {
        if(root->left!=NULL)
            insert(root->left,temp);
        else
            root->left=temp;
    }

    if(temp->data>root->data)
    {
        if(root->right!=NULL)
            insert(root->right,temp);
        else
            root->right=temp;
    }
}

void preorder(node *root)
{
    if(root!=NULL)
    {
        printf("%d ",root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

25. a).What is AVL Tree? What is the need for balance? Discuss various rotations associated with it.

- AVL tree is a self-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.
- A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.
- There are four rotations and they are classified into two types.
- Single Left Rotation (LL Rotation)

- In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree...

- Single Right Rotation (RR Rotation)
  - In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree...
- Left Right Rotation (LR Rotation)
  - The LR Rotation is combination of single left rotation followed by single right rotation. In LR Roration, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...
- Right Left Rotation (RL Rotation)
  - The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree...

(OR)

b).What is tree traversal? List and discuss the various tree traversal methods with suitable example.

- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

- Inorder Traversal:

  Algorithm Inorder(tree)
  1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  2. Visit the root.
  3. Traverse the right subtree, i.e., call Inorder(right-subtree)

  Uses of Inorder
  In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

- Preorder Traversal:

  Algorithm Preorder(tree)
  1. Visit the root.
  2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  3. Traverse the right subtree, i.e., call Preorder(right-subtree)

  Uses of Preorder
  Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.

- Postorder Traversal:

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

26. a).What is threaded binary tree? Discuss the various operations of it.

- Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
- There are two types of threaded binary trees.
  Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
- ***Double Threaded:*** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

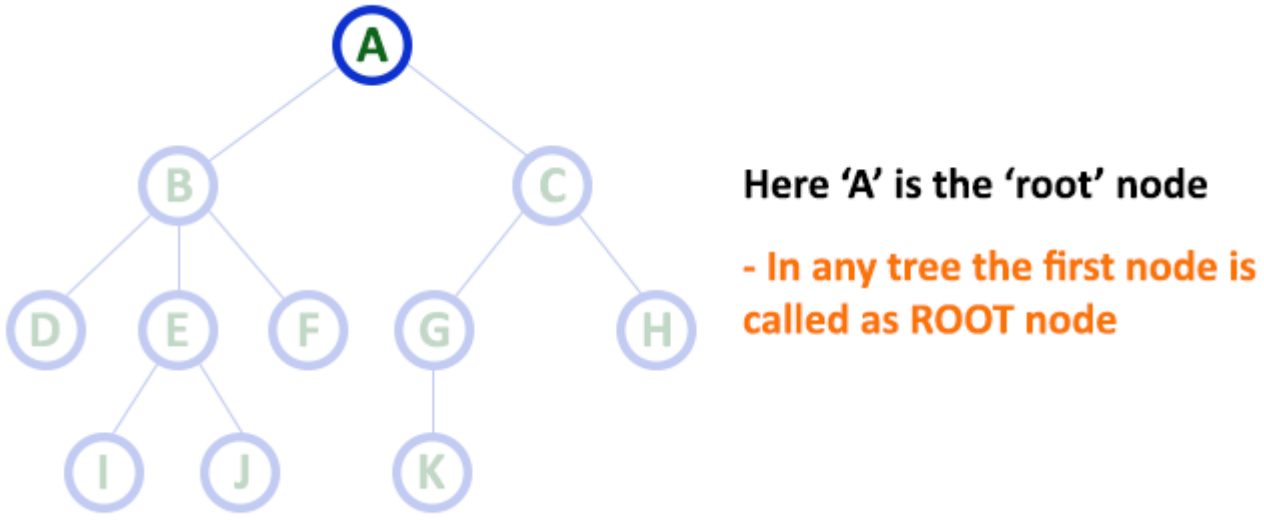
(OR)

b). List the various terminologies that associated with tree data structure and explain.

In a tree data structure, we use the following terminology...

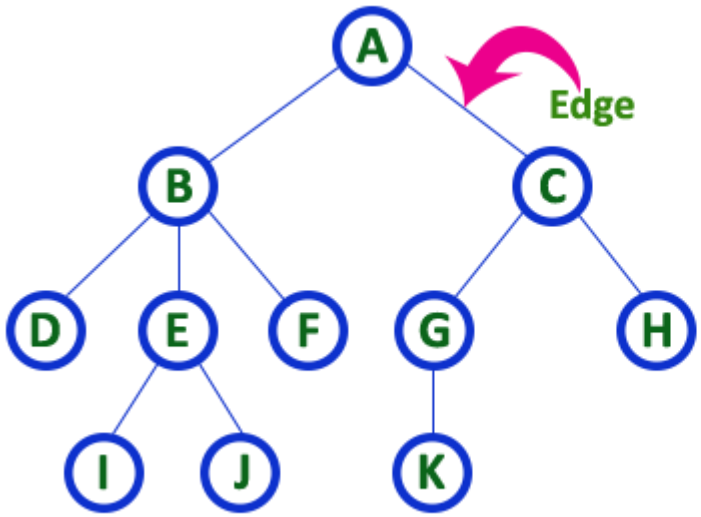
1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



2. Edge

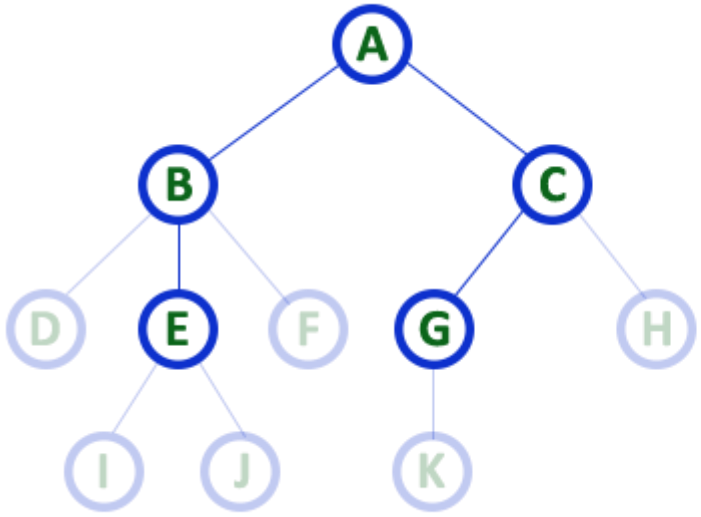
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".

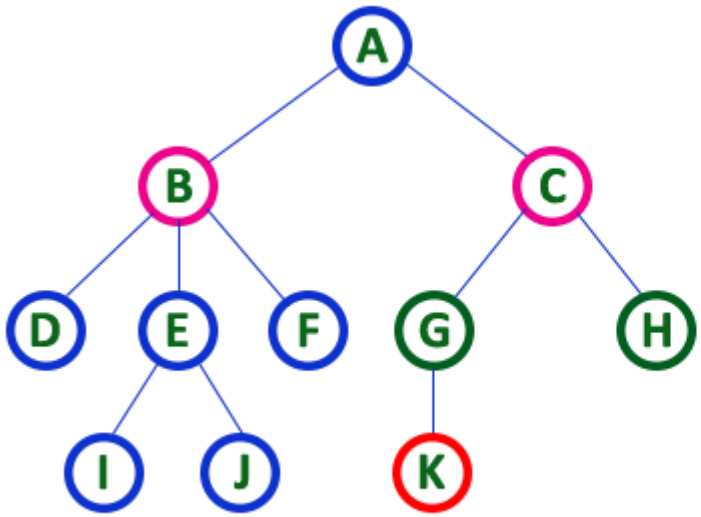


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

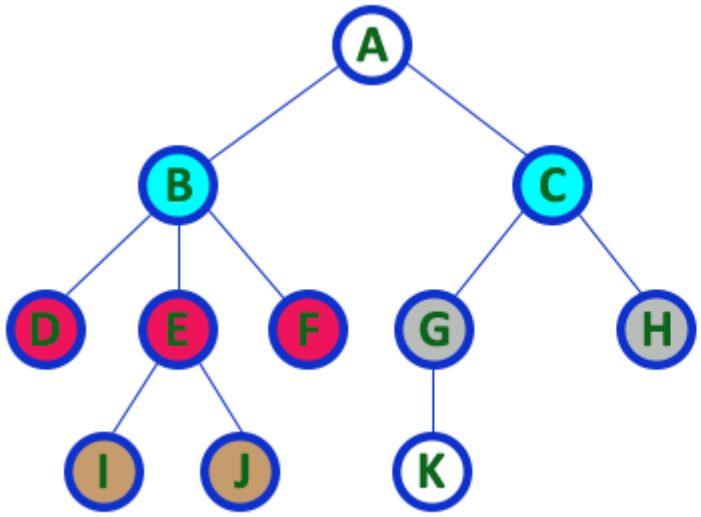
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



Here B & C are Siblings

Here D E & F are Siblings

Here G & H are Siblings

Here I & J are Siblings

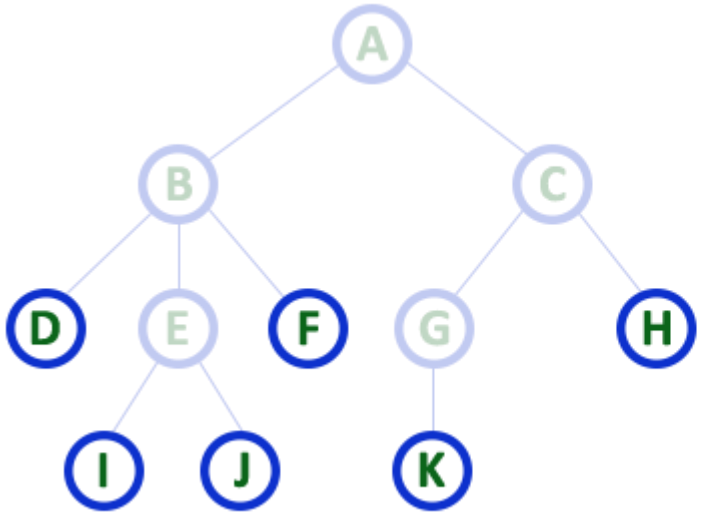
- In any tree the nodes which has same Parent are called 'Siblings'

- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



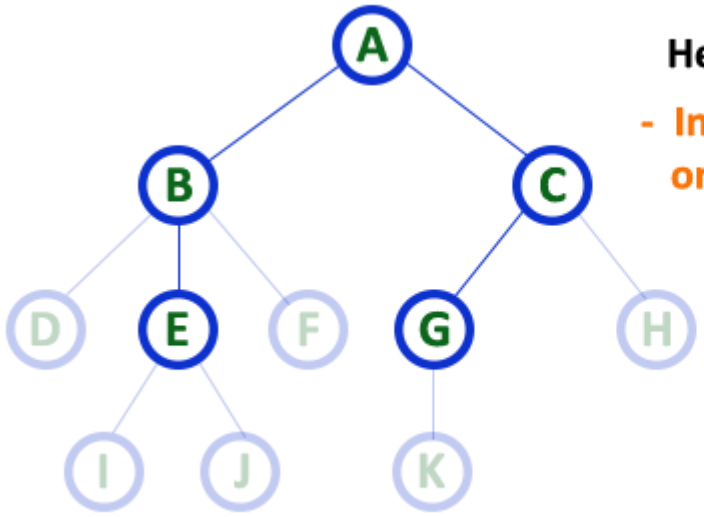
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.

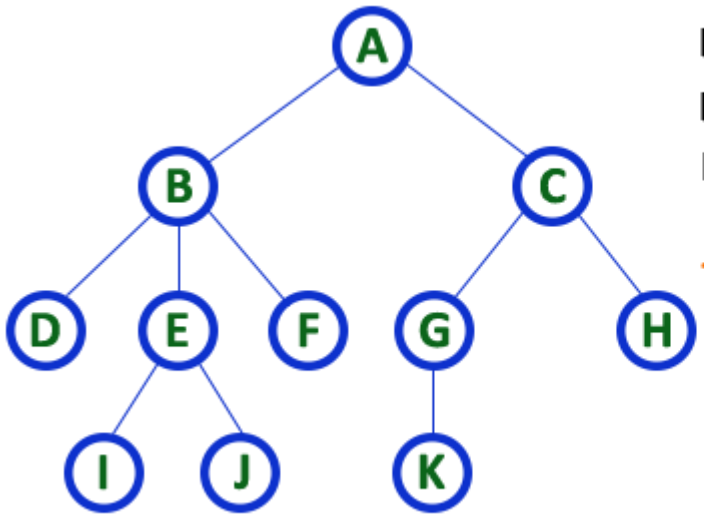


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



Here Degree of B is 3

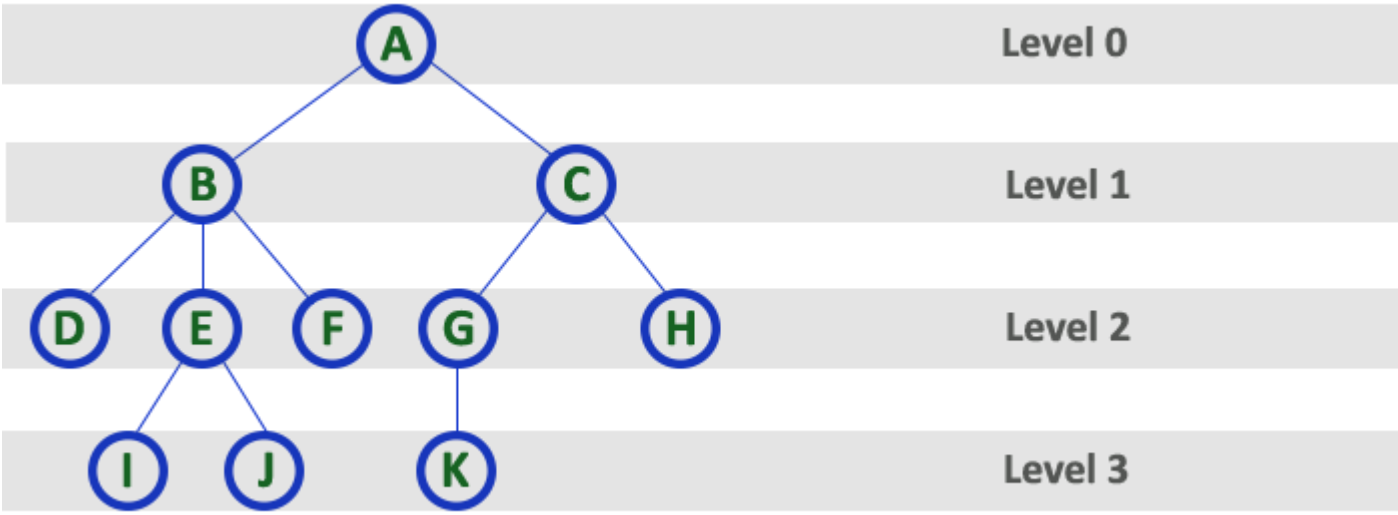
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

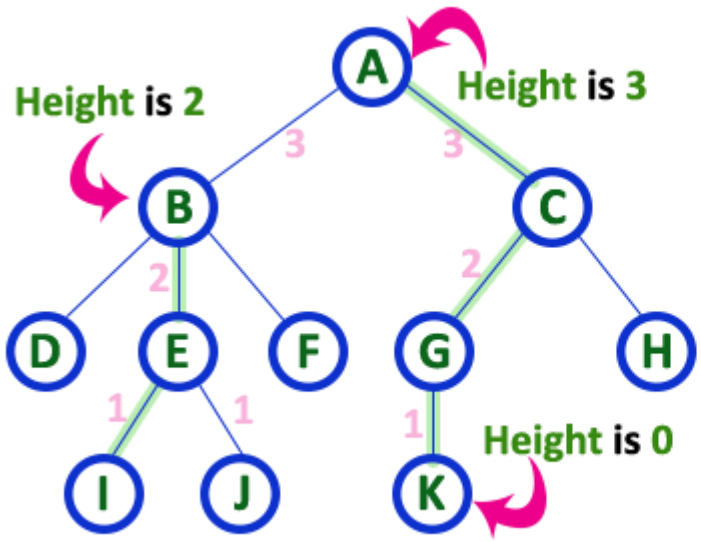
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**

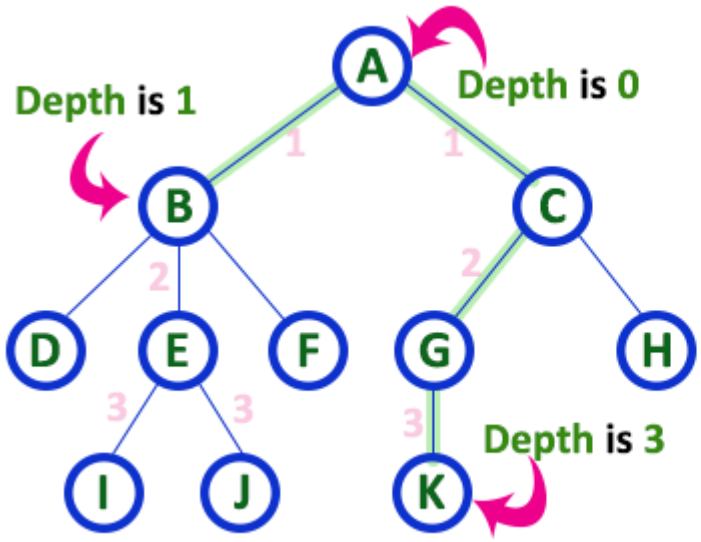


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

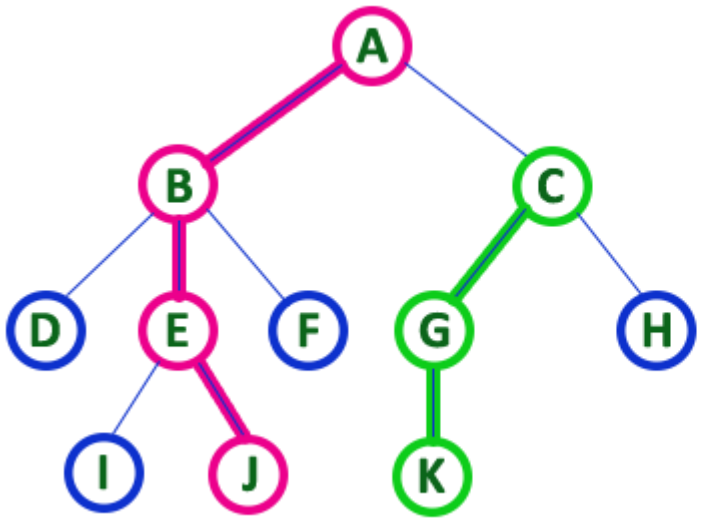


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



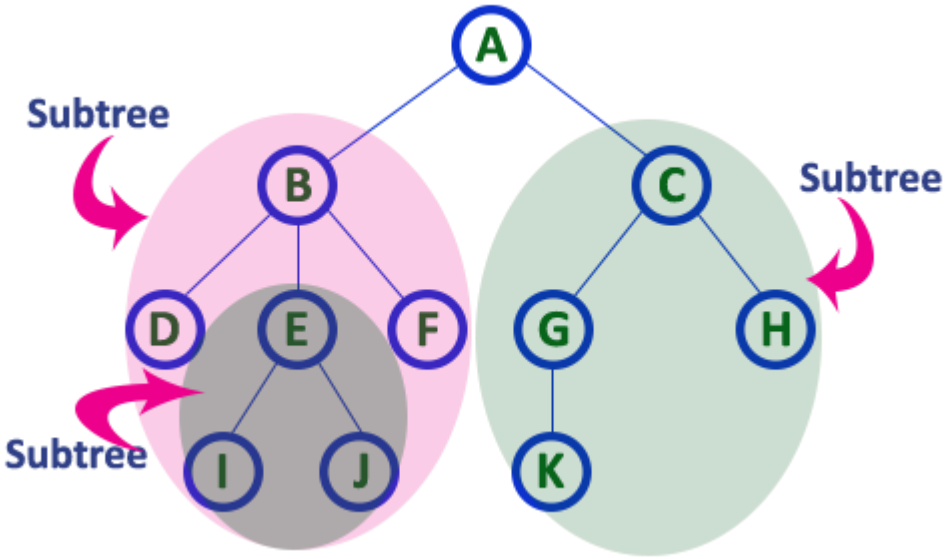
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Reg. No.....

[11CAU302]

KARPAGAM UNIVERSITY

(Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2011 onwards)

BCA DEGREE EXAMINATION, APRIL 2015

Third Semester

COMPUTER APPLICATIONS

DATA STRUCTURES AND ALGORITHMS

Time: 3 hours Maximum : 100 marks

PART – A (15 x 2 = 30 Marks)
Answer ALL the Questions

1. List and explain the two major phases of performance evaluation of an algorithm.
2. How a postfix expression can be evaluated?
3. What is shell sort?
4. How will you represent a polynomial in a linked list?
5. Why linked stacks and queues are needed?
6. What is a head node?
7. What is a Tree?
8. What are the three types of binary tree traversals?
9. What is tree indexing?
10. What is adjacency list?
11. What is a connected and strongly connected component?
12. What is a minimum cost spanning tree?
13. Differentiate static tree table and dynamic tree table.
14. What is a hashing function?
15. What is an AVL tree?

PART B (5 X 14= 70 Marks)
Answer ALL the Questions

16. a. Explain the operations associated with stack, its structure and its implementation.

Or

b. With an example explain the following,
i. Selection Sort ii. 2-Way Merge Sort

17. a. How will you perform insertion and deletion in linked list?

Or

b. With neat diagrams explain the dynamic storage management.

18. a. Explain in detail the three types of binary tree traversal.

Or

b. Discuss about B Trees.

19. a. Explain in detail the three most commonly used representations of graph.

Or

b. Discuss about spanning trees and the stages in Kruskal's Algorithm which leads to minimum cost spanning tree.

20. a. Write in detail about balanced merge sort.

Or

b. Explain the height balanced binary trees in detail.

Reg. No.....

[13CAU302]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2013 onwards)

BCA DEGREE EXAMINATION, NOVEMBER 2015

Third Semester

COMPUTER APPLICATIONS

DATA STRUCTURES AND ALGORITHMS

Time: 3 hours Maximum : 60 marks

PART – A (10 x 2 = 20 Marks)
Answer any TEN Questions

1. List the criteria upon which an algorithm must be analyzed.
2. What is an array? How it can be represented?
3. What is sorting and what are its two broad categories?
4. Compare singly linked list and doubly linked list.
5. Give the procedures used to allocate and release a node.
6. What is the node structure for a sparse matrix representation?
7. How will you find the degree of a node in a tree?
8. Define a skewed tree.
9. Give any two properties of binary search.
10. How is a graph represented?
11. What is a network in a graph?
12. What is a spanning tree?
13. What is a dynamic tree table?
14. Define concept of loading factor.
15. What is a bucket?

PART B (5 X 8= 40 Marks)
Answer ALL the Questions

16. a. Explain the operations associated with queue, its structure and its implementation.

Or

b. With an example explain the following,
i. Selection Sort ii. 2-Way Merge Sort

17. a. Discuss in detail the technique involved in sorting with disks.

Or

b. Explain polyphase merge on tapes with example.

18. a. What is a threaded binary tree? Explain it.

Or

b. Discuss about B Trees.

19. a. Given an undirected graph G = (V, E) and a vertex v in V (G), explain the two ways of visiting all vertices in G that are reachable from V.

Or

b. Explain the following shortest path algorithms
i. Dijkstra's shortest path algorithm ii. All Pairs shortest path algorithm

20. **Compulsory : -**

If your application involves more insertions and deletions of records which data structure you would prefer, array or linked list? Why? Write algorithm to perform insertion and deletion operation.

Reg. No.....

[15CAU301/15CSU301]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2015 onwards)

BCA., B.Sc., DEGREE EXAMINATION, NOVEMBER 2016

Third Semester

COMPUTER APPLICATIONS/COMPUTER SCIENCE

DATA STRUCTURES AND ALGORITHMS

Time: 3 hours

Maximum : 60 marks

PART – A (20 x 1 = 20 Marks) (30 Minutes)
(Question Nos. 1 to 20 Online Examinations)

PART B (5 x 8 = 40 Marks) (2 ½ Hours)
Answer ALL the Questions

21. a. How are the arrays represented? Explain.

Or

b. Discuss QUEUE structure with its operations.

22. a. Write and illustrate the procedure to create, insert and delete nodes in a singly linked list.

Or

b. Explain ALLOCATE procedure in dynamic storage management.

23. a. Elucidate the binary tree traversal procedures.

Or

b. Explain the following : (i) Heap Sort (ii) Binary search

24. a. What are all the ways to represent the graphs?

Or

b. Illustrate the Kruskal algorithm with an example.

25. a. Elucidate polyphase merge sort with an example.

Or

b. Discuss Division and Folding hash functions.

Reg. No.....

[14CAU302]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2014onwards)

BCA DEGREE EXAMINATION, JANUARY 2017

Third Semester

COMPUTER APPLICATIONS

DATA STRUCTURES AND ALGORITHMS

Time: 3 hours Maximum : 60 marks

PART – A (20 x 1 = 20 Marks)
Answer ALL the Questions

1._____ is a sequence of instructions to accomplish a particular task
a. Data Strucuture b. Algorithm c. Ordered List d. Queue

2. _____ criteria of an algorithm ensures that the algorithm terminate after a particular number of steps.
a. effectiveness b. finiteness c. definiteness d. recursive

3. An algorithm must produce _____ output(s)
a. many b. only one c. atleast one d. zero or more

4. _____ criteria of an algorithm ensure that the algorithm must be feasible.
a. effectiveness b. output c. finiteness d. input

5. In Linked List, a _____is a collection of data and link fields.
a. Link b. Node c. Stack d. Data

6. Each item in a node is called a_____.
a. Field b. Data item c. Pointer d. Data

7. The Link field contains the _____ of the next node it points
a. name b. type c. size d. address

8. Data movement and displacing the pointers of the Queue are tedious problems in _____ representation of a Queue.
a. Array b. Linked c. Circular d. Matrix

9. A data structure whose elements forms a sequence of ordered list is called as _____ data structure.
a. Non Linear b. Linear. c. Primitive d. Non Primitive

10. A data structure which represents (non-sequenctial) hierarchical relationship between the elements are called as _____ data structure.
a. Linear b. Primitive. c. Non Linear d. Non Primitive

11. Which of the following is a valid non - linear data structure?
a. Stacks b. Trees c. Queues d. Linked list.

12. _____ are genealogical charts.
a. Stack and Queue b. Pedigree and lineal chart c. Line and bar chart
d. Flow charts

13. In a undirected graph G two vertices v1 and v2 are said to be _____if there is a path in G from v1 to v2.
a. connected b. adjacent c. neighbours d. incident

14. The maximum number of edges in an undirected graph with n veritices is _____
a. n(n-1)/2 b. n/2 c. n-1/2 d. n(n-1)

15. The maximum number of edges in an directed graph with n veritices is _____
a. n(n-1)/3 b. n/3 c. n-1/3 d. n(n-1)

16. In a Graph G if there are n vertices the adjacencies list then consists of _____ nodes.
a. n/2 b. 2n c. n-1 d. n

17. Every binary search tree wth n nodes has _____ sqare node (external nodes).
a. n/2 b. n+1 c. n-1 d. 2^n

18. In a Hash table the address of the identifier x is obtained by applying
a. sequence of comparisions b. binary searching c. arithmetic function
d. collision

19. The partitions of the hash table are called _____
a. Nodes b. Buckets c. Roots d. Fields