KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore - 641021.

(For the candidates admitted from 2015 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT:DATA STRUCTURES **SEMESTER**: III

CODE: 16CSU301 **CLASS**: II B.Sc.CS

Course Objective:

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.

Course Outcome:

A student who successfully completes this course should, at a minimum, be able to:

- 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

Text Book:

1. SartajSahni.(2011). Data Structures, Algorithms and applications in C++(2nded.). New Delhi: Universities Press.
2. Aaron, M. Tenenbaum., Moshe, J. Augenstein., &YedidyahLangsam.(2009). Data Structures Using C and C++(2nd ed.). New Delhi: PHI.

Reference Book:

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

WEB SITES:

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

END SEMESTER MARK ALLOCATION

1.	**PART – A** **20*1=20** **ONLINE EXAMINATION**	**20**
2	**PART – B** **5*2=10**	**10**
3	**PART – C** **5*6=30** **EITHER OR TYPE**	**30**
4	**TOTAL**	**60**

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

KARPAGAM ACADEMY OF HIGHER EDUCATION

LECTURE PLAN

DEPARTMENT OF COMPUTER SCIENCE

Subject : DATA STRUCTURES **Subcode : 16CSU301**

CLASS : II-B.Sc (CS) **Semester: III**

Staff Name: S.Joyce

UNIT-I

S.No	Lecture Duration (Hours)	Topics to be Covered	Support Materials
1	1	Introduction, Arrays	T1.Pg:223, 1.Pg:224
2	1	Single dimensional Arrays	T1.Pg:225,226
3	1	Multi-dimensional Arrays	T1.Pg:227,228
4	1	Sparse Matrices (Array Representation)	T1.Pg:252
5	1	Sparse Matrices (Linked Representation)	T1.Pg:254
6	1	Stacks	T1.Pg:258
7	1	Implementing single multiple stacks in an Array	T1.Pg:259
8	1	Prefix, Infix and Postfix expressions	T2.Pg:95-99
9	1	Utility and conversion of these expressions from one to another	T2.Pg:99-106
10	1	Applications of stack	T1.Pg:284-300
11	1	Limitations of Array representation of stack	W1
12	1	Recapitulation and Discussion of Important Questions	
		Total No.of Hours Planned for Unit I	12

Text Book:

1. Sartaj Sahni. (2011). Data Structures, Algorithms and applications in C++(2nd ed.). New Delhi: Universities Press.

2. Aaron, M. Tenenbaum., Moshe, J. Augenstein., & Yedidyah Langsam. (2009). Data Structures Using C and C++(2nd ed.). New Delhi: PHI.

WEB SITES:

1. http://en.wikipedia.org/wiki/Data_structure

UNIT-II

1	1	Linked Lists Singly	T1.Pg:172-174
2	1	Doubly Lists	T1.Pg:192
3	1	Doubly and Circular Lists (Array and Linked representation)	T1.Pg:194
4	1	Normal representation of Stack in Lists	W1
5	1	Circular representation of Stack in Lists	W1
6	1	Self Organizing Lists	T1.Pg:323
7	1	Skip Lists Queues	T1.Pg:324
8	1	Array representation of Queue	T1.Pg:325
9	1	Linked representation of Queue	T1.Pg:326-330
10	1	De-queue	T1.Pg:464
11	1	Priority Queues	T1.Pg:466
12	1	Recapitulation and Discussion of Important Questions	
		Total No. of Hours Planned for -Unit II	12

Text Book:

1. Sartaj Sahni. (2011). Data Structures, Algorithms and applications in C++(2nd ed.). New Delhi: Universities Press.

WEB SITES:

1. http://en.wikipedia.org/wiki/Data_structure

UNIT-III

1	1	Trees	T1.Pg:420
2	1	Introduction to Tree as a data structure	T1.Pg:421
3	1	Binary Trees (Insertion)	T1.Pg:425
4	1	Binary Trees (Deletion)	T1.Pg:426
5	1	Binary Trees (Recursive and Iterative)	T1.Pg:427
6	1	Traversals on Binary Search Trees	T1.Pg:432
7	1	Threaded Binary Trees	W1
8	1	Threaded Binary Trees (Insertion, Deletion, Traversals	W1
9	1	Height-Balanced Trees	T1.Pg:563-567

10	1	AVL Trees	T1.Pg:568-572
11	1	Operations on AVL Trees	T1.Pg:573
12	1	Recapitulation and Discussion of Important Questions	
		Total No. of Hours Planned for -Unit III	12

Text Book:

1. Sartaj Sahni. (2011). Data Structures, Algorithms and applications in C++(2nd ed.). New Delhi: Universities Press.

WEB SITES:

1.http://www.cs.sunysb.edu/~skiena/214/lectures/

UNIT-IV

1	1	Searching	T2.Pg:351
2	1	Sorting	T2.Pg:352
3	1	Comparison of Linear Search	W2
4	1	Binary Search	W2
5	1	Comparison of Linear Search	W2
6	1	Comparison of Binary Search	W1
7	1	Selection Sort	T2.Pg:351
8	1	Insertion Sort	T2.Pg:353
9	1	Shell Sort	T2.Pg:366
10	1	Comparison of Sorting Techniques	W2
11	1	Recapitulation and Discussion of Important Questions	
		Total No. of Hours Planned for -Unit IV	11

Text Book:

1. Sartaj Sahni. (2011). Data Structures, Algorithms and applications in C++(2nd ed.). New Delhi: Universities Press.

2.Aaron, M. Tenenbaum., Moshe, J. Augenstein., & Yedidyah Langsam. (2009). Data Structures Using C and C++(2nd ed.). New Delhi: PHI.

WEB SITES:

1.http://en.wikipedia.org/wiki/Data_structure

2.http://www.cs.sunysb.edu/~skiena/214/lectures/

UNIT-V

1	1	Hashing	T2.Pg:468
2	1	Introduction to Hashing	T2.Pg:469
3	1	Deleting from Hash Table	T2.Pg:473
4	1	Efficiency of Rehash Methods	T2.Pg:474
5	1	Hash Table Reordering	T2.Pg:476
6	1	Resolving collusion by Open Addressing,	W2
7	1	Coalesced Hashing, Separate Chaining	T2.Pg:485-488
8		Dynamic and Extendible Hashing	T2.Pg:494
9	1	Choosing a Hash Function, Perfect Hashing, Function	T2.Pg:505-508
10	1	Recapitulation and Discussion of Important Questions	
11	1	Discussion of Previous ESE Question Papers	
12	1	Discussion of Previous ESE Question Papers	
13	1	Discussion of Previous ESE Question Papers	
		Total No. of Hours Planned for Unit V	13

Text Book:

1. Sartaj Sahni. (2011). Data Structures, Algorithms and applications in C++(2nd ed.). New Delhi: Universities Press.

2. Aaron, M. Tenenbaum., Moshe, J. Augenstein., & Yedidyah Langsam. (2009). Data Structures Using C and C++(2nd ed.). New Delhi: PHI.

WEB SITES:

1.http://en.wikipedia.org/wiki/Data_structure

2.http://www.cs.sunysb.edu/~skiena/214/lectures/

SYLLABUS

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:
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.**

Introduction to Algorithms:

***Why write algorithms*:**

(1) To get it out of the head, human memory is unreliable!

(2) To communicate with the programmer and other algorithm developers.

(3) To prove its correctness, to analyze, to improve its efficiency, …

ALGORITHM:

What is a computer program?

(1) Steps for solving some problem, and

(2) the steps written (implemented) in a language to be understood by a compiler program (and eventually by a CPU).

Problem	Algorithm	Program

Calculate $\sum_{i=i}^{n} i^2$	for i = 1 through n do
	accumulate i^2 in x;
	return x.
		public static int sum(int n) {
		int partialSum = 0;
		for (int i=1; i<=n; i++)
		partialSum += i*i;
		return partialSum; }

The first step in solving a computational problem is developing an algorithm.

It could be written in a loose fashion or in a rigorous way. Its primary purpose is to communicate the problem solving steps *unambiguously*.

It must be provably *correct* and should *terminate*.

So, specification = problem definition;

algorithm = solution of the problem;

program = translation of the algorithm in a computer language.

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.

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.

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).

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.

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.

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.

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.

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:

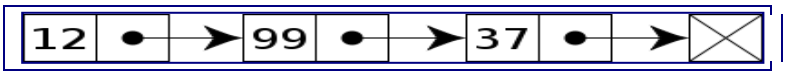
```
char string_matrix[4][10][20];
```

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.

Linked List:

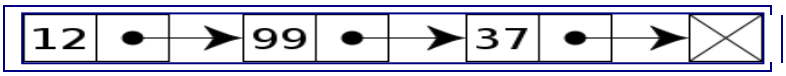
A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure allows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

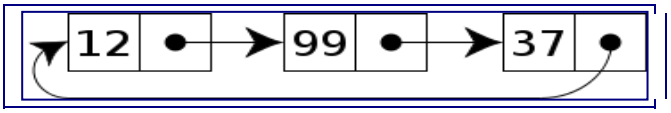
Doubly linked list

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

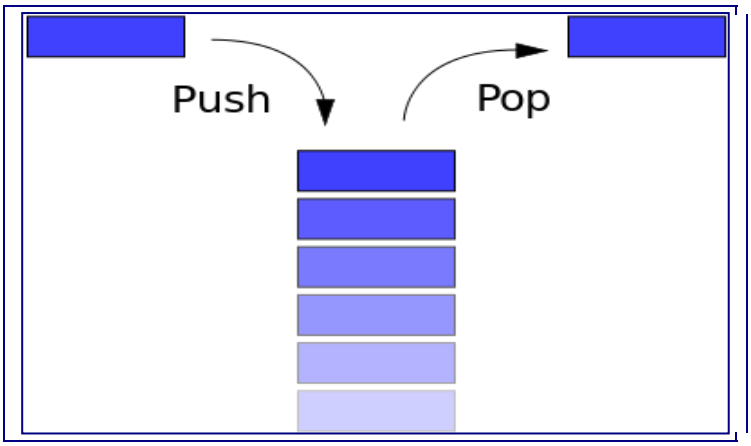
In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

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).



Stack

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.

Trees:

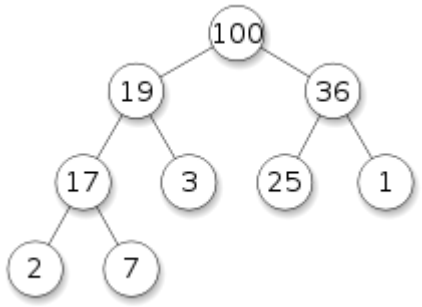
A tree is a non-linear data structure that consists of a root node and potentially many levels of additional nodes that form a hierarchy. A tree can be empty with no nodes called the null or empty tree or a tree is a structure consisting of one node called the root and one or more subtrees.

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.

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*

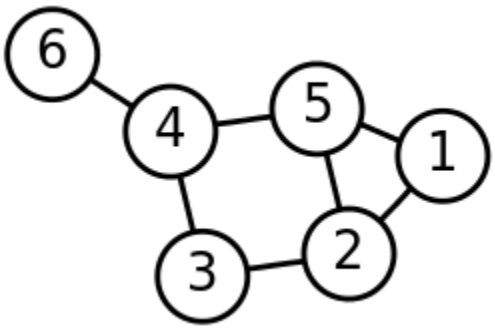
heap).



max heap

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**.**

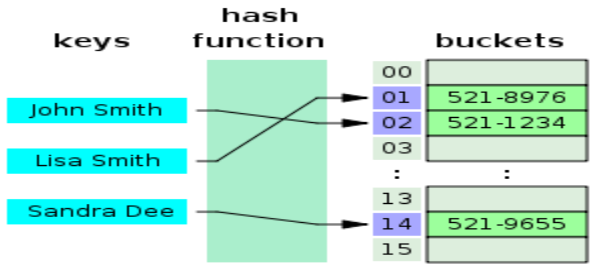


A labeled graph of 6 vertices and 7 edges.

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.

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.

- 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.

Arrays :

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
Gurpreet
Anupama
Hanish
Harsh
Navdeep
Shalini
Kapil

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

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.

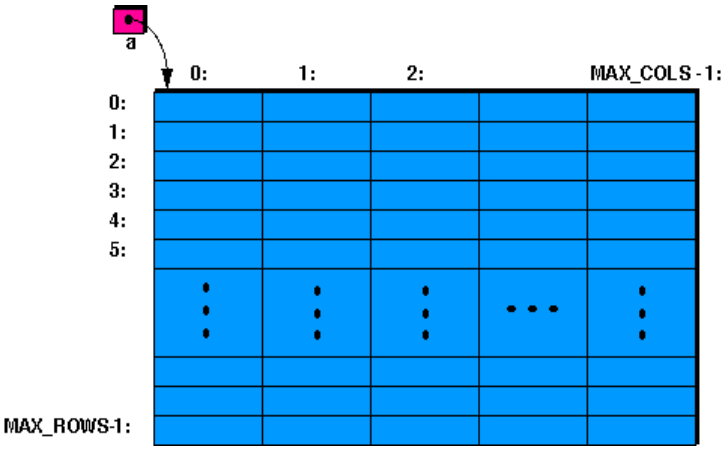
2-dimensional arrays provide most of this capability. Like a 1D array, a 2D array is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.

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;
```

Multiple Stacks:

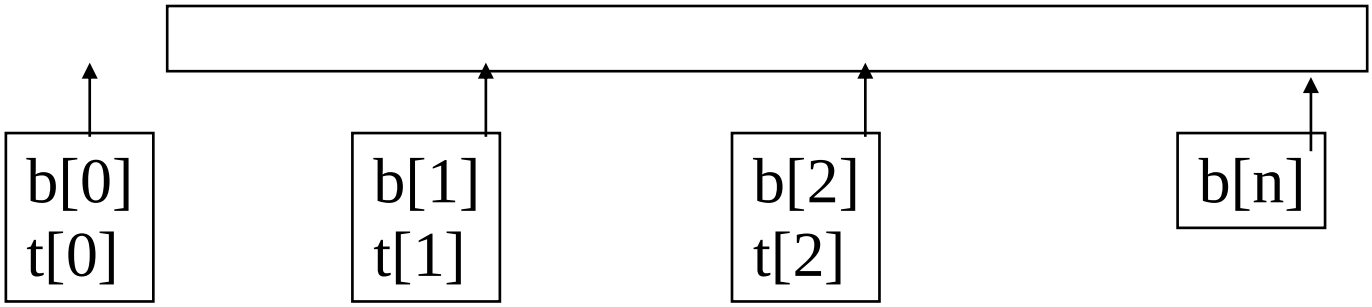
1. None fixed size of the stacks:
    - Stack 1 expands from the 0^{th} element to the right
    - Stack 2 expands from the 12^{th} element to the left
    - As long as the value of Top1 and Top2 are not next to each other, it has free elements for input the data in the array
    - When both Stacks are 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**

M[0..m-1]



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.

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

Sparse matrices using array and linked representation:

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.

Why to use Sparse Matrix instead of simple matrix ?

Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

Sparse Matrix Representations can be done in many ways following are two common representations:

Array representation

Linked list representation

Method 1: Using Arrays

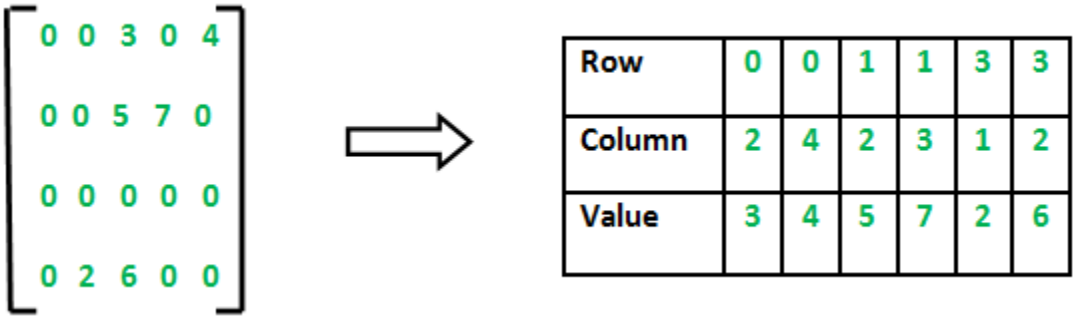
2D array is used to represent a sparse matrix in which there are three rows named as

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row,column)

Sparse Matrix Array Representation

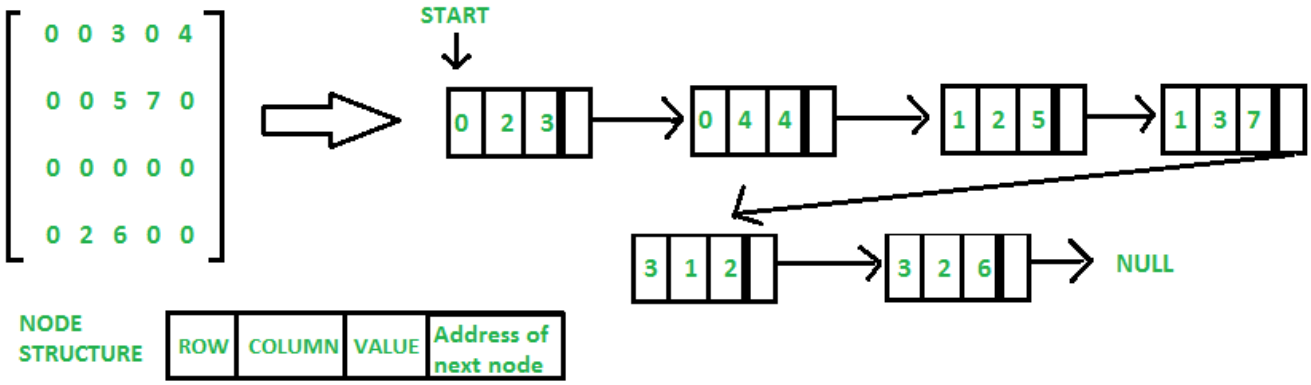


Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node

Using Arrays



Stack Using Array:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
Step 2: Declare all the functions used in stack implementation.
Step 3: Create a one dimensional array with fixed size (int stack[SIZE])
Step 4: Define a integer variable 'top' and initialize with '-1'. (int top = -1)
Step 5: In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

Step 1: Check whether stack is FULL. (top == SIZE-1)
Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step 1: Check whether stack is EMPTY. (top == -1)
Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

Step 1: Check whether stack is EMPTY. (top == -1)

Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).
Step 3: Repeat above step until i value becomes '0'.

Expressions
What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.
In above definition, operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

Infix Expression
Postfix Expression
Prefix Expression
Infix Expression:

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example

Infix Expression
Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the

Operands".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator
Example

Postfix Expression

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2
Example

Prefix Expression

Any expression can be represented using the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

Utility and conversion of these expressions from one to another:
Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following

procedure...

Find all the operators in the given Infix Expression.
Find the order of operators evaluated according to their Operator precedence.
Convert each operator into required type of expression (Postfix or Prefix) in the same order.
Example

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

Step 1: The Operators in the given Infix Expression : = , + , *
Step 2: The Order of Operators according to their preference : * , + , =
Step 3: Now, convert the first operator * ----- D = A + B C *
Step 4: Convert the next operator + ----- D = A BC* +
Step 5: Convert the next operator = ----- D ABC*+ =
Finally, given **Infix Expression is converted into Postfix Expression** as follows...

D A B C * + =

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

Read all the symbols one by one from left to right in the given Infix Expression.
If the reading symbol is operand, then directly print it to the result (Output).
If the reading symbol is left parenthesis '(', then Push it on to the Stack.
If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.
Example

Consider the following Infix Expression...

(A + B) * (C - D)

The final Postfix Expression is as follows...

A B + C D - *

APPLICATION OF STACK:

Expression evaluation and syntax parsing:

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations and conversion from one form to another may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most programming languages are context-free languages, allowing them to be parsed with stack based machines.

Backtracking:

Another important application of stacks is backtracking. Consider a simple example of finding the correct path in a maze. There are a series of points, from the starting point to the destination. We start from one point. To reach the final destination, there are several paths. Suppose we choose a random path. After following a certain path, we realize that the path we have chosen is wrong. So we need to find a way by which we can return to the beginning of that path. This can be done with the use of stacks. With the help of stacks, we remember the point where we have reached. This is done by pushing that point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking.

The prototypical example of a backtracking algorithm is depth-first search, which finds all vertices of a graph that can be reached from a specified starting vertex. Other applications of backtracking involve searching through spaces that represent potential solutions to an optimization problem. Branch and bound is a technique for performing such backtracking searches without exhaustively searching all of the potential solutions in such a space.

Runtime memory management:

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.

Almost all calling conventions—the ways in which subroutines receive their parameters and return results—use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. The functions follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C programming language is typically implemented in this way. Using the same stack for both data and procedure calls has important security implications (see below) of which a programmer must be aware in order to avoid introducing serious security bugs into a program.

Efficient algorithms:

Several algorithms use a stack (separate from the usual function call stack of most programming languages) as the principledata structure with which they organize their information. These include:

- Graham scan, an algorithm for the convex hull of a two-dimensional system of points. A convex hull of a subset of the input is maintained in a stack, which is used to find and remove concavities in the boundary when a new point is added to the hull.
- Part of the SMAWK algorithm for finding the row minima of a monotone matrix uses stacks in a similar way to Graham scan.
- All nearest smaller values, the problem of finding, for each number in an array, the closest preceding number that is smaller than it. One algorithm for this problem uses a stack to maintain a collection of candidates for the nearest smaller value. For each position in the array, the stack is popped until a smaller value is found on its top, and then the value in the new position is pushed onto the stack.
- The nearest-neighbor chain algorithm, a method for agglomerative hierarchical clustering based on maintaining a stack of clusters, each of which is the nearest neighbor of its predecessor on the stack. When this method finds a pair of clusters that are mutual nearest neighbors, they are popped and merged.

LIMITATIONS OF ARRAY REPRESENTATION OF STACK:

Under the array implementation, a fixed set of nodes represented by an array is established at the start of execution. A pointer to a node is represented by the relative position of the node within the array. The disadvantage of that approach is twofold. First, the number of nodes that are needed often cannot be predicted when a program is written. Usually, the data with which the program is executed determines the number of nodes necessary. Thus no matter how many elements the array of nodes contains, it is always possible that the program will be executed with input that requires a larger number.

The second disadvantage of the array approach is that whatever number of nodes are declared must remain allocated to the program throughout its execution. For example, if 500 nodes of a given type are declared, the amount of storage required for those 500 nodes is reserved for that purpose. If the program actually uses only 100 or even 10 nodes in its execution the additional nodes are still reserved and their storage cannot be used for any other purpose.

The solution to this problem is to allow nodes that are dynamic, rather than static. That is, when a node

is needed, storage is reserved for it, and when it is no longer needed, the storage is released. Thus the storage for nodes that are no longer in use is available for another purpose. Also, no predefined limit on the number of nodes is established. As long as sufficient storage is available to the job as a whole, part of that storage can be reserved for use as a node.

We have seen that we can use arrays whenever we have to store and manipulate collections of elements.

- the dimension of an array is determined the moment the array is created, and cannot be changed later on.
- the array occupies an amount of memory that is proportional to its size, independently of the number of elements that are actually of interest.
- if we want to keep the elements of the collection ordered, and insert a new value in its correct position, or remove it, then, for each such operation we may need to move many elements (on the average, half of the elements of the array);this is very inefficient.

POSSIBLE QUESTIONS

UNIT-I

PART-A **(20 MARKS)**

(Q.NO 1 TO 20 Online Examination)

PART-B **(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.

PART-C **(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 H
Coimbatore - 641
(For the candidates admitted f

DEPARTMENT OF COMPUTE

UNIT I :(Objective Type/Multiple choice Quest

PART-A (Online Exam

S.NO	QUESTIONS	OPTION 1
1	_____ is a sequence of instructions to accomplish a particular task	Data Strucuture
2	_____ criteria of an algorithm ensures that the algorithm terminate after a particular number of steps.	effectiveness
3	An algorithm must produce _____ output(s)	many
4	_____ criteria of an algorithm ensures that the algorithm must be feasible.	effectiveness
5	_____ criteria of an algorithm ensures that each step of the algorithm must be clear and unambiguous.	effectiveness
6	The logical or mathematical model of a particular data organization is called as_____	Data Structure
7	An algorithms _____ is measured in terms of computing time ad space consumed by it.	performance
8	Which of the following is not structured data type?	Arrays
9	What is the strategy of Stack?	LILO
10	What is the strategy of Queue?	LILO
11	Data structures are classified as _____ data type.	User Defined
12	_____ are the commonl used ordered list.	Graphs
13	Data structure can be classified as _____ data type based on relationship with complex data element.	Linear & Non Linear
14	A data structure whose elements forms a sequence of ordered list is called as _____ data structure.	Non Linear
15	A data structure which represents hierarchical relationship between the elements are called as _____ data structure.	Linear

16	A data structure, which is not composed of other data structure, is called as _____ data structure.	Linear
17	Data structures, which are constructed from one or more primitive data structure, are called as _____ data structure.	Non Primitive
18	_____ is the term that refers ti the kinds of data that variables may hold in a programming language.	data type
19	_____ refers to the set of elements that belong to a particular type.	data type
20	The triplet *(D,F,A) r* efers to a _____ where *D* is a set of Domains, *F* is a set of Functions and *A* is a set of Axioms	data type
21	_____estimation is the method of analysing an algorithm before it is executed.	Preprocess
22	In queue we can add elements at _____.	Top
23	In queue we can delete elements at _____.	Front
24	In Stack we can add elements at _____.	Bottom
25	In Stack we can delete elements at _____	Front
26	When Top = Bottom in stack, the total no of element in the stack is	1
27	When FRONT = REAR in queue, the total no of element in the queue is	0
28	In Stack the TOP is decremeted by one after every ___ operation.	AddQ
29	In Stack the TOP is incremeted by one before every ___ operation.	AddQ
30	To add an item into the queue,	FRONT is incremented by one
31	In Queue FRONTis incremented then, the operation performed on it is _____.	DelQ
32	When the maximum entries of (m*n) matrix are zeros then it is called as _____.	Transpose matrix
33	A matrix of the form (row, col, n) is otherwise known as _____.	Transpose matrix

34	Which of the following is a valid linear data structure.	Stacks
35	Which of the following is a valid non - linear data structure.	Stacks
36	A list of finite number of homogeneous data elements are called as _____	Stacks
37	No of elements in an array is called the _____ of an array.	Structure
38	_____ is the art of creating sample data upon which to run the program	Testing
39	If a program fail to respond corectly then _____ is needed to determine what is wrong and how to correct it.	Testing
40	A _____ is a linear list in which elements can be inserted and deleted at both ends but not at the Middle	Queue
41	A _____ is a collection of elements such that each element has been assigned a priority.	Priority Queue
42	A _____ is made up of Operators and Operands.	Stack
43	A _____ is a procedure or function which calls itself.	Stack
44	An example for application of stack is _____.	Time sharing computer system
45	An example for application of queue is _____.	Stack of coins
46	Combining elements of two similar data structure into one is called _____	Merging
47	Adding a new element into a data structure called _____	Merging
48	The Process of finding the location of the element with the given value or a record with the given key is _____.	Merging
49	Arranging the elements of a data structure in some type of order is called _____.	Merging
50	The size or length of an array = _____.	UB – LB + 1
51	The _____ model of a particular data organization is called as Data Structure.	software Engineering
52	Combining elements of two _____ data structure into one is called Merging	Similar
53	Searching is the Process of finding the _____ of the element with the given value or a record with the given key.	Place
54	Length of an array is defined as _____ of elements in it.	Structure

55	In _____ search method the search begins by examining the record in the middle of the file.	sequential
56	_____ is a internal sorting method.	sorting with disks
57	Quick sort reads _____ space to implement the recursion.	stack
58	The most popular method for sorting on external storage devices is _____.	quick sort
59	The 2-way merge algorithm is almost identical to the _____procedure.	quick
60	A _____ merge on m runs requires at most [$\log_k m$] passes over the data.	n-way

HIGHER EDUCATION
1021.
rom 2016 onwards)

R SCIENCE,CA & IT

:ions each Question carries one Mark)

ıination)

OPTION 2	OPTION 3	OPTION 4	KEY
Algorithm	Ordered List	Queue	Algorithm
finiteness	definiteness	All the above	finiteness
only one	atleast one	zero or more	atleast one
finiteness	definiteness	All the above	effectiveness
finiteness	effectiveness	All the above	effectiveness
Software Engineering	Data Mining	Data Ware Housing	Data Structure
effectiveness	finiteness	definiteness	performance
Union.	Queue	Linked list.	Union.
FIFO	FILO	LIFO	LIFO
FIFO	FILO	LIFO	FIFO
Abstract	Primitive & Non Primitive	None of the above	Primitive & Non Primitive
Trees	Stack and Queues	All the above	Stack and Queues
Linear	Non Linear	None of the above	Linear & Non Linear
Linear.	Primitive	Non Primitive	Linear.
Primitive.	Non Linear	Non Primitive	Non Linear

Non Primitive	Non Linear	Primitive	Primitive
Primitive.	Non Linear	Linear	Non Primitive
data structure	data Object	data	data type
data structure	data Object	data	data Object
data structure	data Object	data	data structure
Verification	Priori	Posteriori	Priori
Bottom	Front	Rear	Rear
Bottom	Top	Rear	Front
Top	Front	Rear	Top
Rear	Top	Bottom	Top
2	3	0	0
1	2	3	0
Pop	Push	DelQ	Pop
Pop	Push	DelQ	Push
FRONT is decremented by one	REAR is decremented by one	REAR is incremeted by one	REAR is incremeted by one
Pop	Push	AddQ	DelQ
Sparse Matrix	Inverse Matrix	None of the above.	Sparse Matrix
Inverse Matrix	Sparse Matrix	None of the above.	Sparse Matrix

Records	Trees	Graphs	Stacks
Trees	Queues	Linked list.	Trees
Records	Arrays	Linked list.	Arrays
Height	Width	Length.	Length.
Designing	Analysis	Debugging	Testing
Designing	Analysis	Debugging	Debugging
DeQueue	Enqueue	Priority Queue	DeQueue
De Queue	Circular Queue	En Queue	Priority Queue
Expression	Linked list	Queue	Expression
Recursion	Queue	Tree	Recursion
Waiting Audience	Processing of subroutines	None of the above	Processing of subroutines
Stack of bills	Processing of subroutines	Job Scheduling in TimeSharing	Job Scheduling in TimeSharing
Insertion	Searching	Sorting	Merging
Insertion	Searching	Sorting	Insertion
Insertion	Searching	Sorting	Searching
Insertion	Searching	Sorting	Sorting
LB + 1	UB - LB	UB – 1	UB – LB + 1
logical or mathematical	Data Mining	Data Ware Housing	logical or mathematical
Dissimilar	Even	Un Even	Similar
Location	Value	Operand	Location
Height	Size	Number	Number

fibonacci	binary	non-sequential	binary
quick sort	balanced merge	sorting with	quick sort
queue	circular stacks	circular queue	stack
radix sort	merge sort	heap sort	merge sort
merge	heap	radix	merge
m-way	k-way	q-way	k-way

SYLLABUS

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.

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);
```

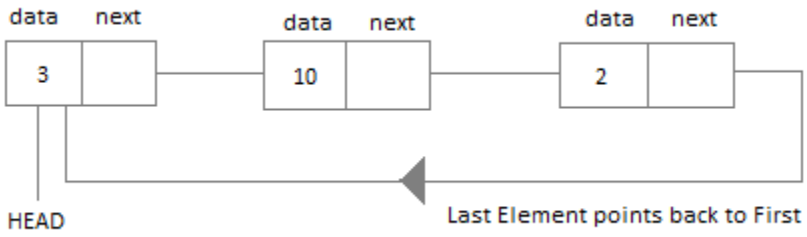
```
}
else
{
while ((p != NULL) && (p->roll != rno))
{
q = p;
p = p->next;
}
if
(p == NULL)
/* Element not found */
printf ("\nNo match :: deletion failed");
else if (p->roll == rno)
/* Delete any other element */
{
q->next = p->next;
free (p);
}
}
}
```

Circular lists:

Circular linked representation:

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked

list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Circular Linked List

Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

```
class Node {
  public:
  int data;
  //pointer to the next node
  node* next;

  node() {
```

```
   data = 0;
   next = NULL;
  }

  node(int x) {
   data = x;
   next = NULL;
  }
}
```

Circular Linked List:

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods**.**

```
class CircularLinkedList {
  public:
  node *head;
  //declaring the functions

  //function to add Node at front
  int addAtFront(node *n);
  //function to check whether Linked list is empty
  int isEmpty();
  //function to add Node at the End of list
  int addAtEnd(node *n);
  //function to search a value
  node* search(int k);
  //function to delete any Node
  node* deleteNode(int x);

  CircularLinkedList() {
   head = NULL;
  }
}
```

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.

5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int CircularLinkedList :: addAtFront(node *n) {
  int i = 0;
  /* If the list is empty */
  if(head == NULL) {
    n->next = head;
    //making the new Node as Head
    head = n;
    i++;
  }
  else {
    n->next = head;
    //get the Last Node and make its next point to new Node
    Node* last = getLastNode();
    last->next = n;
    //also make the head point to the new first Node
    head = n;
    i++;
  }
  //returning the position where Node is added
  return i;
}
```

Insertion at the End:

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```
int CircularLinkedList :: addAtEnd(node *n) {
  //If list is empty
  if(head == NULL) {
    //making the new Node as Head
    head = n;
    //making the next pointer of the new Node as Null
    n->next = NULL;
  }
```

```
  else {
    //getting the last node
    node *last = getLastNode();
    last->next = n;
    //making the next pointer of new node point to head
    n->next = head;
  }
}
```

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* CircularLinkedList :: search(int x) {
  node *ptr = head;
  while(ptr != NULL && ptr->data != x) {
    //until we reach the end or we find a Node with data x, we keep moving
    ptr = ptr->next;
  }
  return ptr;
}
```

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

```
node* CircularLinkedList :: deleteNode(int x) {
  //searching the Node with data x
  node *n = search(x);
  node *ptr = head;
```

```
  if(ptr == NULL) {
    cout << "List is empty";
    return NULL;
  }
  else if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```

Linked lists using arrays of nodes

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an array of records, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If records are also not supported, parallel arrays can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
 record Entry {
    integer next; // index of next entry in array
    integer prev; // previous entry (if double-linked)
    string name;
    real balance;
 }
```

A linked list can be built by creating an array of these structures, and an integer variable to store the index of the first element.

```
integer listHead
Entry Records[1000]
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

Index	Next	Prev	Name	Balance
0	1	4	Jones, John	123.45
1	−1	0	Smith, Joseph	234.56
2 (listHead)	4	−1	Adams, Adam	0.00
3			Ignore, Ignatius	999.99
4	0	2	Another, Anita	876.54

In the above example, ListHead would be set to 2, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a ListFree integer variable, a free list could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries could be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead
while i ≥ 0 // loop through the list
    print i, Records[i].name, Records[i].balance // print entry
    i := Records[i].next
```

Representing of a stack in 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.

Self-organizing list:

A **self-organizing list** is a list that reorders its elements based on some self-organizing heuristic to improve average access time. The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list. A self-organizing list achieves near constant time for element access in the best case. A self-organizing list uses a reorganizing algorithm to adapt to various query distributions at runtime.

Efficiency of self-organizing lists

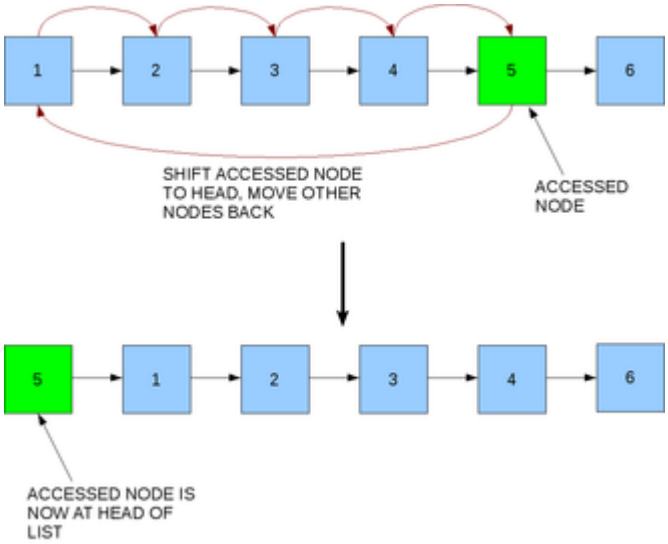
A self organizing list rearranges the nodes keeping the most frequently accessed ones at the head of the list. Generally, in a particular query, the chances of accessing a node which has been accessed many times before are higher than the chances of accessing a node which historically has not been so frequently accessed. As a result, keeping the commonly accessed nodes at the head of the list results in reducing the number of comparisons required in an average case to reach the desired node. This leads to better efficiency and generally reduced query times.

Techniques for Rearranging Nodes:

While ordering the elements in the list, the access probabilities of the elements are not generally known in advance. This has led to the development of various heuristics to approximate optimal behavior. The basic heuristics used to reorder the elements in the list are:

Move to Front Method (MTF)

This technique moves the element which is accessed to the head of the list. This has the advantage of being easily implemented and requiring no extra memory. This heuristic also adapts quickly to rapid changes in the query distribution. On the other hand, this method may prioritize infrequently accessed nodes-for example, if an uncommon node is accessed even once, it is moved to the head of the list and given maximum priority even if it is not going to be accessed frequently in the future. These 'over rewarded' nodes destroy the optimal ordering of the list and lead to slower access times for commonly accessed elements. Another disadvantage is that this method may become too flexible leading to access patterns that change too rapidly. This means that due to the very short memories of access patterns even an optimal arrangement of the list can be disturbed immediately by accessing an infrequent node in the list.

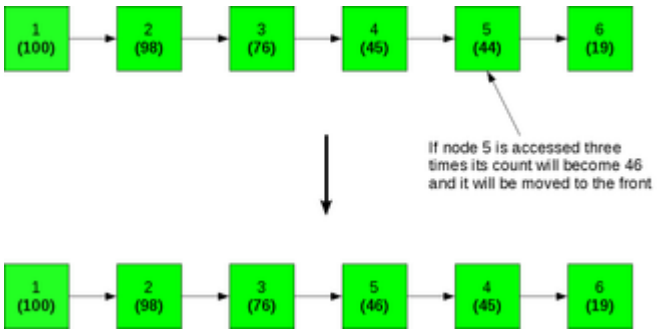


If the 5th node is selected, it is moved to the front

```
At the t-th item selection:
    if item i is selected:
        move item i to head of the list
```

Count Method:

In this technique, the number of times each node was searched for is counted i.e. every node keeps a separate counter variable which is incremented every time it is called. The nodes are then rearranged according to decreasing count. Thus, the nodes of highest count i.e. most frequently accessed are kept at the head of the list. The primary advantage of this technique is that it generally is more realistic in representing the actual access pattern. However, there is an added memory requirement, that of maintaining a counter variable for each node in the list. Also, this technique does not adapt quickly to rapid changes in the access patterns. For example: if the count of the head element say A is 100 and for any node after it say B is 40, then even if B becomes the new most commonly accessed element, it must still be accessed at least (100 - 40 = 60) times before it can become the head element and thus make the list ordering optimal.

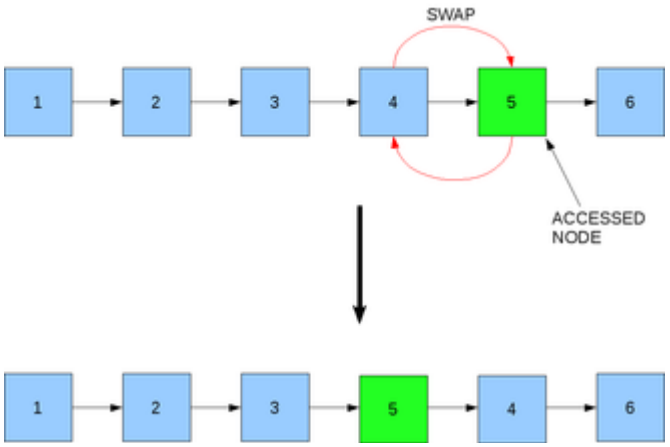


If the 5th node in the list is searched for twice, it will be swapped with the 4th

```
init: count(i) = 0 for each item i
At t-th item selection:
  if item i is searched:
     count(i) = count(i) + 1
     rearrange items based on count
```

Transpose Method:

This technique involves swapping an accessed node with its predecessor. Therefore, if any node is accessed, it is swapped with the node in front unless it is the head node, thereby increasing its priority. This algorithm is again easy to implement and space efficient and is more likely to keep frequently accessed nodes at the front of the list. However, the transpose method is more cautious. i.e. it will take many accesses to move the element to the head of the list. This method also does not allow for rapid response to changes in the query distributions on the nodes in the list.



If the 5th node in the list is selected, it will be swapped with the 4th

```
At the t-th item selection:
   if item i is selected:
      if i is not the head of list:
            swap item i with item (i - 1)
```

The worst case search time for a sorted linked list is O(n). With a Balanced Binary Search Tree, we can 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.
One idea to make search faster for Linked Lists is Skip List. Another idea (which is discussed in this post) is to place more frequently accessed items closer to head.. There

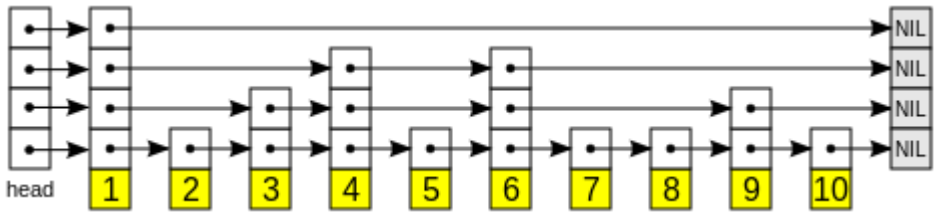
can be two possibilities. offline (we know the complete search sequence in advance) and online (we don't know the search sequence).
In case of offline, we can put the nodes according to decreasing frequencies of search (The element having maximum search count is put first). For many practical applications, it may be difficult to obtain search sequence in advance.

Competitive Analysis:

The worst case time complexity of all methods is O(n). In worst case, the searched element is always the last element in list. For average case analysis, we need probability distribution of search sequences which is not available many times.
For online strategies and algorithms like above, we have a totally different way of analyzing them called *competitive analysis* where performance of an online algorithm is compared to the performance of an optimal offline algorithm (that can view the sequence of requests in advance). Competitive analysis is used in many practical algorithms like caching, disk paging, high performance computers.

Skip List Data Structure:

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsequences of the items. A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.



Skip List

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.

Structure of Skip List

A skip list is built up of layers. The lowest layer (i.e. bottom layer) is an ordinary ordered linked list. The higher layers are like 'express lane' where the nodes are skipped (observe the figure).

Searching Process

When an element is tried to search, the search begins at the head element of the top list. It proceeds horizontally until the current element is greater than or equal to the target. If current element and target are matched, it means they are equal and search gets finished.

If the current element is greater than target, the search goes on and reaches to the end of the linked list, the procedure is repeated after returning to the previous element and the search reaches to the next lower list (vertically).

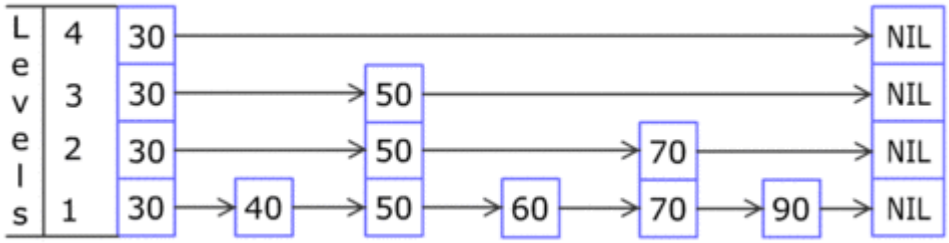
Implementation Details

1. The elements used for a skip list can contain more than one pointers since they are allowed to participated in more than one list.

2. Insertion and deletion operations are very similar to corresponding linked list operations.

we could make the level structure quasi-random in the following way:

```
make all nodes level 1
j ← 1
while the number of nodes at level j > 1 do
  for each i'th node at level j do
    if i is odd
      if i is not the last node at level j
        randomly choose whether to promote it to level j+1
      else
        do not promote
      end if
    else if i is even and node i-1 was not promoted
      promote it to level j+1
    end if
  repeat
  j ← j + 1
repeat
```



Insertion in Skip List

Applications of Skip List:
1. Skip list are used in distributed applications. In distributed systems, the nodes of skip list represents the computer systems and pointers represent network connection.

2. Skip list are used for implementing highly scalable concurrent priority queues with less lock contention (struggle for having a lock on a data item).

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.

Array and Linked Representation of Queue:

Queue Using Array

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to **create an empty queue.**

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
Step 2: Declare all the user defined functions which are used in queue implementation.
Step 3: Create a one dimensional array with above defined SIZE (int queue[SIZE])
Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)
Step 5: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

Step 1: Check whether queue is FULL. (rear == SIZE-1)
Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3: If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

deQueue() - Deleting a value from the Queue

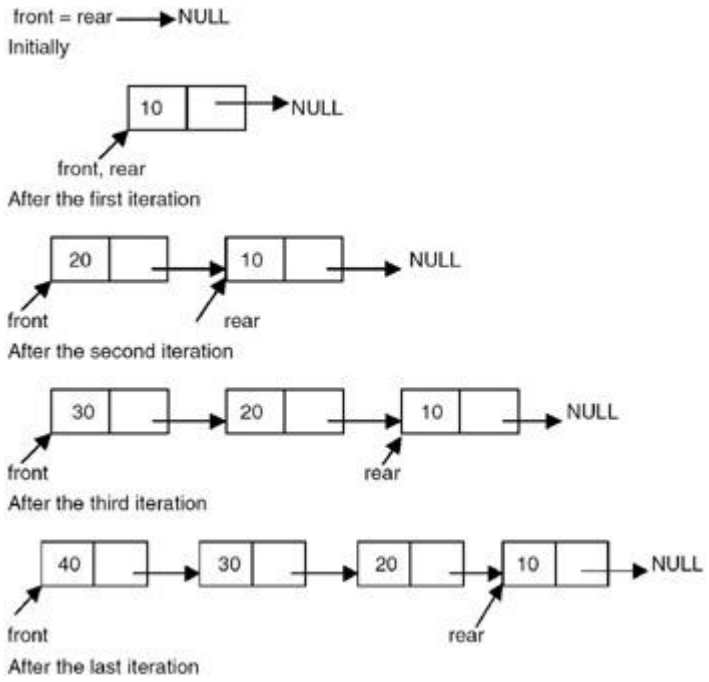
In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

Step 1: Check whether queue is EMPTY. (front == rear)
Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

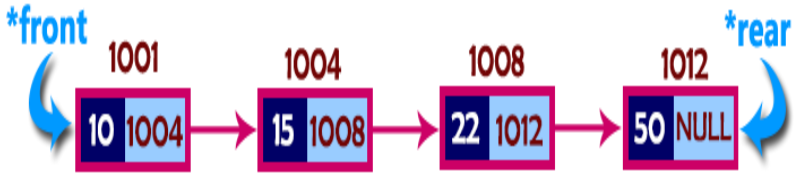
Step 1: Check whether queue is EMPTY. (front == rear)
Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
Step 3: Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value is equal to rear (i <= rear)



Queue using Linked List:

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.
Example



In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.
Step 2: Define a 'Node' structure with two members data and next.
Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.
Step 4: Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.
enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

Step 1: Create a newNode with given value and set 'newNode → next' to NULL.
Step 2: Check whether queue is Empty (rear == NULL)
Step 3: If it is Empty then, set front = newNode and rear = newNode.
Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.
deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1: Check whether queue is Empty (front == NULL).
Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
Step 3: If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).
display() - Displaying the elements of Queue

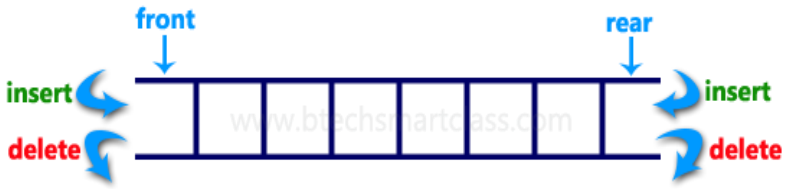
We can use the following steps to display the elements (nodes) of a queue...

Step 1: Check whether queue is Empty (front == NULL).
Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
Step 4: Finally! Display 'temp → data ---> NULL'.

DEQUE:

Double Ended Queue (Dequeue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



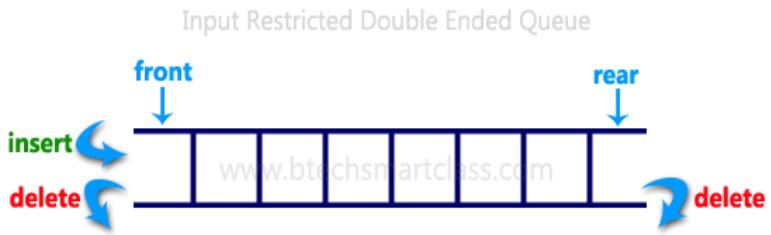
Double Ended Queue

Double Ended Queue can be represented in TWO ways, those are as follows...

Input Restricted Double Ended Queue
Output Restricted Double Ended Queue

Input Restricted Double Ended Queue:

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue:

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.

Dequeue Operation:

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue**operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.

Algorithm for dequeue operation

```
procedure dequeue
  if queue is empty
    return underflow
  end if
```

```
   data = queue[front]
   front ← front + 1
   return true
end procedure
```

Operations on Deque:
Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.
insertLast(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported
getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.

Priority Queue:

Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

There are two types of priority queues they are as follows...
1. Max Priority Queue
2. Min Priority Queue

1. Max Priority Queue:

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

The following are the operations performed in a Max priority queue...

1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMax()** - Find maximum value in the queue.
4. **remove()** - Delete maximum value from the queue.

Max Priority Queue Representations:

There are 6 representations of max priority queue.

1. **Using an Unordered Array (Dynamic Array)**
2. **Using an Unordered Array (Dynamic Array) with the index of the maximum value**
3. **Using an Array (Dynamic Array) in Decreasing Order**
4. **Using an Array (Dynamic Array) in Increasing Order**
5. **Using Linked List in Increasing Order**
6. **Using Unordered Linked List with reference to node with the maximum value**

1. Using an Unordered Array (Dynamic Array):

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at the end of the queue. This operation requires O(1) time complexity that means constant time.

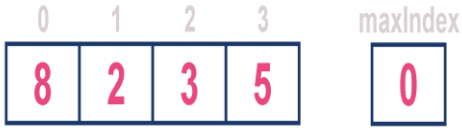
findMax() - To find maximum element in the queue, we need to compare with all the elements in the queue. This operation requires O(n) time complexity.

remove() - To remove an element from the queue first we need to perform findMax() which requires O(n) and removal of particular element requires constant time O(1). This operation requires O(n) time complexity.

2. Using an Unordered Array (Dynamic Array) with the index of the maximum value:

In this representation elements are inserted according to their arrival order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at the end of the queue with O(1) and for each insertion we need to update maxIndex with O(1). This operation requires O(1) time complexity that means constant time.

findMax() - To find maximum element in the queue is very simple as maxIndex has maximum element index. This operation requires O(1) time complexity.

remove() - To remove an element from the queue first we need to perform findMax() which requires O(1) , removal of particular element requires constant time O(1) and update maxIndex value which requires O(n). This operation requires O(n) time complexity.

3. Using an Array (Dynamic Array) in Decreasing Order:

In this representation elements are inserted according to their value in decreasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at a particular position in the decreasing order into the queue with O(n), because we need to shift existing elements inorder to insert new element in decreasing order. This operation requires O(n) time complexity.

findMax() - To find maximum element in the queue is very simple as maximum element is at the beginning of the queue. This operation requires O(1) time complexity.

remove() - To remove an element from the queue first we need to perform findMax() which requires O(1), removal of particular element requires constant time O(1) and rearrange remaining elements which requires O(n). This operation requires O(n) time complexity.

4. Using an Array (Dynamic Array) in Increasing Order:

In this representation elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If 'front == -1' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at a particular position in the increasing order into the queue with O(n), because we need to shift existing elements inorder to insert new element in increasing order. This operation requires O(n) time complexity.

findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requires O(1) time complexity.

remove() - To remove an element from the queue first we need to perform findMax() which requires O(1), removal of particular element requires constant time O(1) and rearrange remaining elements which requires O(n). This operation requires O(n) time complexity.

5. Using Linked List in Increasing Order

In this representation, we use a single linked list to represent max priority queue. In this representation elements are inserted according to their value in increasing order and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.

Now, let us analyse each operation according to this representation...

isEmpty() - If 'head == NULL' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at a particular position in the increasing order into the queue with O(n), because we need to the position where new element has to be inserted. This operation requires O(n) time complexity.

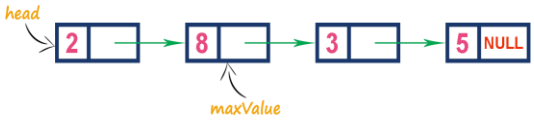
findMax() - To find maximum element in the queue is very simple as maximum element is at the end of the queue. This operation requires O(1) time complexity.

remove() - To remove an element from the queue is simply removing the last node in the queue which requires O(1). This operation requires O(1) time complexity.

6. Using Unordered Linked List with reference to node with the maximum value

In this representation, we use a single linked list to represent max priority queue. Always we maitain a reference (maxValue) to the node with maximum value. In this representation elements are inserted according to their arrival and node with maximum value is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyse each operation according to this representation...

isEmpty() - If 'head == NULL' queue is Empty. This operation requires O(1) time complexity that means constant time.

insert() - New element is added at end the queue with O(1) and update maxValue reference with O(1). This operation requires O(1) time complexity.

findMax() - To find maximum element in the queue is very simple as maxValue is referenced to the node with maximum value in the queue. This operation requires O(1) time complexity.

remove() - To remove an element from the queue is deleting the node which referenced by maxValue which requires O(1) and update maxValue reference to new node with maximum value in the queue which requires O(n) time complexity. This operation requires O(n) time complexity.

2. Min Priority Queue Representations:

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

The following operations are performed in Min Priority Queue...

1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMin()** - Find minimum value in the queue.
4. **remove()** - Delete minimum value from the queue.

Min priority queue is also has same representations as Max priority queue with minimum value removal.

POSSIBLE QUESTIONS

UNIT-I

PART-A **(20 MARKS)**

(Q.NO 1 TO 20 Online Examination)

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 EDUC
Coimbatore - 641021.
(For the candidates admitted from 2016 onwa

DEPARTMENT OF COMPUTER SCIENCE,C

UNIT II :(Objective Type/Multiple choice Questions each Quest

PART-A (Online Examination)

S.NO	QUESTIONS	OPTION 1
1	_____is a collection of data and links.	Links
2	Each item in a node is called a_____.	Field
3	The elements in the list are stored in a one dimensional array called a _____	Value
4	Data movement and displacing the pointers of the Queue are tedious proplems in _____ representation of a Queue.	Array
5	_____ list allows traversing in only one direction.	Singly linked list
6	In storage management every block is said to have three fields namely _____	Llink, Data, Rlink
7	_____ allows traversing in both direction.	Singly linked list
8	_____ is the process of allocating and deallocating memory to various programs in a multiprocessing enviroment.	Job scheduling in Time sharing environment
9	The best application of Doubly Linked list in computers is _____	Job scheduling in Time sharing environment
10	List containing link to all of the available nodes is called _____ list.	Free
11	A _____is a list that reorders its element.	self-organizing list
12	The computing time for manipulating the list is _____for sequential Representation	Less then
13	_____contains all nodes that are not currently being used	Dynamic Memory

14	In singly linked list ,each node has _____field.	One
15	In linked list ,each node has fields namely_____	Link, Value
16	In Doubly linked list ,each node has at least _____field.	One
17	In Doubly linked list ,each node has fields namely_____	Link, Data1, Data2
18	The doubly linked list is said to be empty if it conatins _____	no nodes at all.
19	The data field of the _____ node usually donot conatain any information.	first
20	To search down the list of free blocks to find the first block greater than or equal to the size of the program is called_____ allocation strategy.	Best Fit
21	Finding a free block whose size is as close as possible to the size of the program (N), but not less than N is called _____ allocation strategy.	Near fit
22	_____ strategy distributes the small nodes evenly and searching for a new node starts from the node where the previous allocation was made.	Best Fit
23	Problem in _____ allocation stratery is all small nodes collect in the front of the av-list.	Best Fit
24	_____ is the storage allocation method that fits the program into the largest block available.	Best Fit
25	The back pointer for each node will be referred as _____.	Blink
26	Forward pointer for each node will be referred as _____.	Forward
27	A_____is a linked list in which last node of the list points to the first node in the list.	Linked list
28	A_____in which each node has two pointers, a forward link and a Backward link.	Doubly linked circular list
29	In sparse matrices each nonzero term was represented by a node with _____ fields.	Five
30	We want to represent n stacks with $1 \leq i \leq n$ then T(i)_____	Top of the i^{th} stack

31	We want to represent m queues with $1 \leq i \leq m$ then F(i)_____	Front of the $(i + 1)^{th}$ Queue
32	We want to represent m queues with $1 \leq i \leq m$ then R(i)_____	Rear of the $(i + 1)^{th}$ Queue
33	In Linked representation of Sparse Matrix, DOWN field used to link to the next nonzero element in the same _____	Row
34	In Linked representation of Sparse Matrix, RIGHT field used to link to the next nonzero element in the same _____	Row
35	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})
36	In Available Space list combining the adjacent free blocks is called _____	Defragmenting
37	In Available Space list, the first and last word of each block are reserved for _____	Data
38	In Storage management, in the Available Space List, the first word of each free block has _____fields.	4
39	In Available Space list, the last word of each free block has _____fields.	4
40	The first and last nodes of each block have tag fields, this system of allocation and freeing is called the _____.	Tag Method
41	In Available Space list ,Tag field has the value one when the block is _____	Allocated
42	Available Space list ,Tag field has the value Zero when the block is _____	Allocated
43	The _____field of each storage block indicates if the block is free are in-use.	rlink
44	In storage management the _____ field of the free block points to the start of the block	rlink
45	_____ is the process of collecting all unused nodes and returning them to the available space.	Compaction
46	Moving all free nodes aside to form a single contiguous block of memeory is called _____	Compaction
47	_____ of disk space reduces the average retrieval time of allocation.	Compaction
48	_____ is done in two phases 1) marking used nodes and 2) returning all unmarked nodes to available space list.	Compaction
49	Which of these sorting algorithm uses the Divide and Conquere technique for sorting	selection sort

50	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search
51	The disadvantage of _____ sort is that is need a temporary array to sort.	Quick
52	A _____ is a set of characters is called a string.	Array
53	The straight forward find operation for pattern matching,pat of size m in string of size n needs _____ time.	O(mn)
54	Knuth,Morris and Pratt's method of pattern matching in strings takes _____ time, if pat is of sixe m and string is size n.	O(mn)
55	_____ representation always need extensive data movement.	Linked
56	Which of these representations are used for strings.	sequential representation
57	In _____ method the node is moved to the front.	Front Method
58	In _____ method the node stores count of the no of times	Front Method
59	In _____ method any node searched is swapped with the preceding node.	Count Method
60	A _____ is a data structure that is used for storing a sorted list of items.	Skip list

:ATION

rds)

A & IT

:ion carries one Mark)

OPTION 2	OPTION 3	OPTION 4	KEY
Node	List	Item	Node
Data item	Pointer	Data	Field
List	Data	Link	Data
Linked	Circular	All the above	Array
Doubly linked list	Circular Doubly Linked List	Ordered List	Singly linked list
Link, Data, Size	Size, Link and Unusable	Size, Llink and Uplink	Size, Link and Unusable
Doubly linked list	Circular Singly Linked List	Circular Queue	Doubly linked list
Processor Management	Dynamic Storage Management	Garbage Collection	Dynamic Storage Management
Processing Procedure calls	Dynamic Storage Management	Evaluating postfix expressions	Dynamic Storage Management
Empty	AV	Ordered	AV
organizing list	Self list	self reorganizing list	self-organizing list
Greater than	Less then equal	Greater than equal	Less then
Storage pool	Garbage	Waste memory	Storage pool

Two	Three	Five	Two
Link, Link	Data, Link	Data, Data	Data, Link
Two	Three	Five	Three
Data and Link	Only Llink and Rlink	Llink, Data, Rlink	Llink, Data, Rlink
nodes with data fields empty.	only a head node.	a node with its link fields points to null	only a head node.
head	tail	last	head
First Fit	Worst Fit	Next Fit	First Fit
First fit	Best fit	Next Fit	Best fit
First Fit	Worst Fit	Next Fit	Next Fit
First Fit	Worst Fit	Next Fit	First Fit
First Fit	Worst Fit	Next Fit	Worst Fit
Break	Back	Clear	Blink
Flink	Front	Data	Flink
Singly linked circular list	Circular list	Insertion node	Singly linked circular list
Circular list	Singly linked circular list	Linked list	Doubly linked circular list
Six	Three	Four	Three
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

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
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
List	Column	Diagonal	Column
Matrix	Column	Diagonal	Row
O(m * n * r)	O(m + n + r)	O(max {n, m})	O(m + n + r)
Coalescing	Joining	Merging	Coalescing
Allocation Information	Link	Value	Allocation Information
3	2	1	4
3	2	1	2
Boundary Method	Free Method	Boundary Tag Method	Boundary Tag Method
Coalesced	Free	Merge	Allocated
Coalesced	Free	Merged	Free
tag	size	uplink	tag
llink	uplink	top	uplink
Coalescing	Garbage collection	Deallocation	Garbage collection
Coalescing	Garbage collection	Deallocation	Compaction
Coalescing	Garbage collection	Deallocation	Compaction
Coalescing	Garbage collection	Deallocation	Garbage collection
insertion sort	merge sort	heap sort	merge sort

Binary search	fibonacci search	None of the above	Binary search
Merge	Heap	Insertion	Merge
String	Heap	List	String
$O(n^2)$	$O(m^2)$	O(m+n)	O(mn)
$O(n^2)$	$O(m^2)$	O(m+n)	O(m+n)
sequential	tree	graph	sequential
Linked representation with fixed sized blocks	Linked representation with variable sized blocks	All the above	All the above
Move Method	Move-to-Front Method	Count Method	Move-to-Front Method
Move-to-Front Method	Count Method	Transpose Method	Count Method
Transpose Method	Move-to-Front Method	Front Method	Transpose Method
self-organizing list	Self list	Node list	Skip list

SYLLABUS

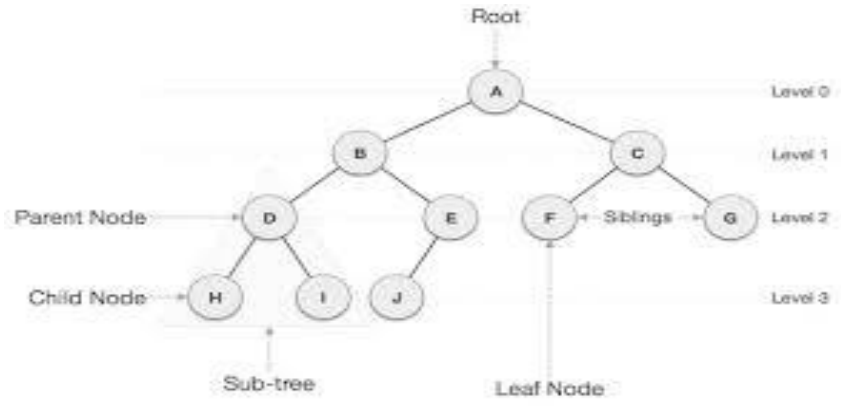
UNIT-III

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.

Binary Search Trees:

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

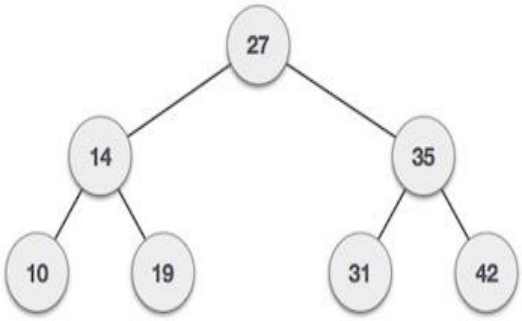
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

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 −

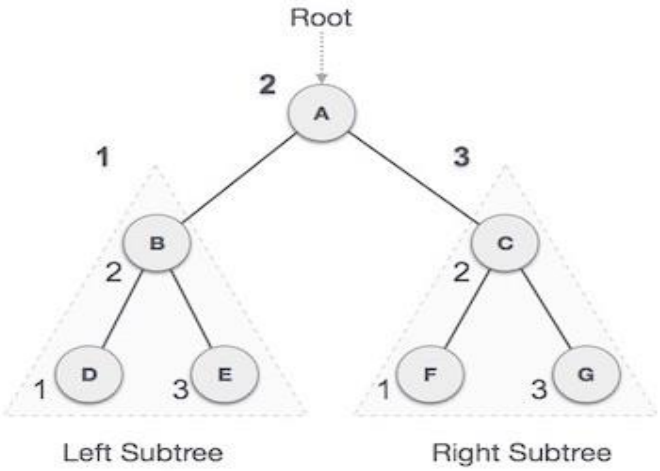
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

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.

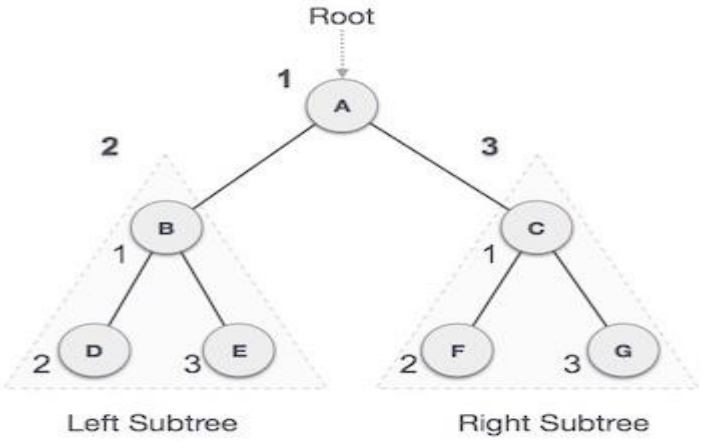
If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



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.

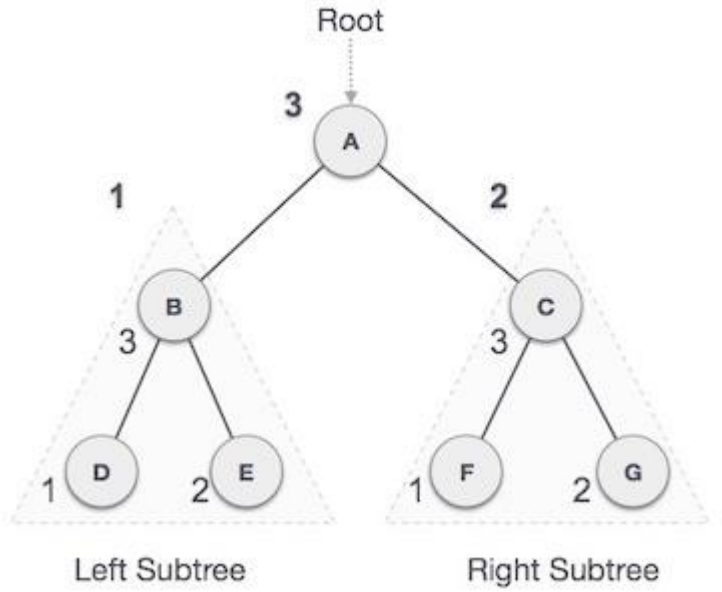


We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

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

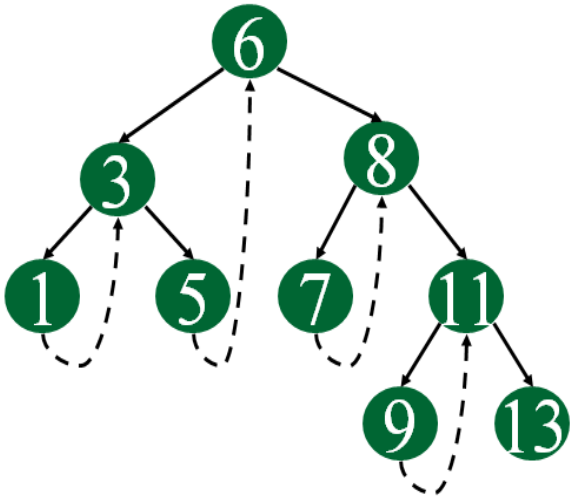
right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

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;
}
```

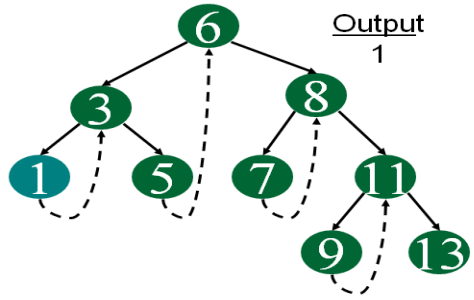
Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

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

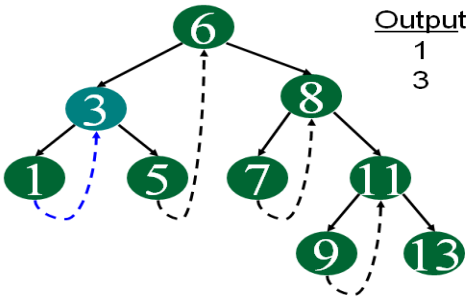
```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
   if (n == NULL)
     return NULL;

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

   return n;
}
```

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

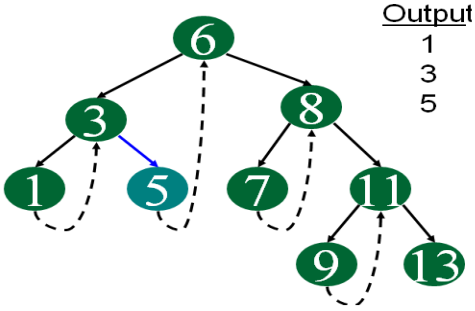
```
void inOrder(struct Node *root)
{
   struct Node *cur = leftmost(root);
   while (cur != NULL)
   {
      printf("%d ", cur->data);

      // 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);
   }
}
```



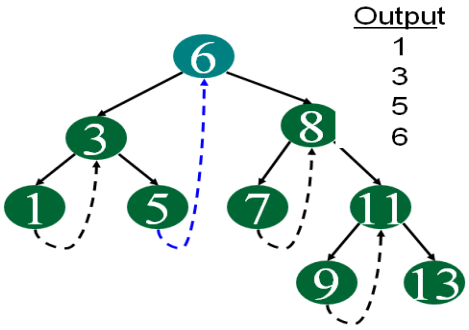
Start at leftmost node, print it



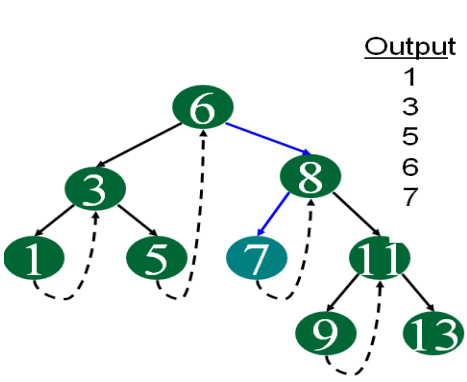
Follow thread to right, print node



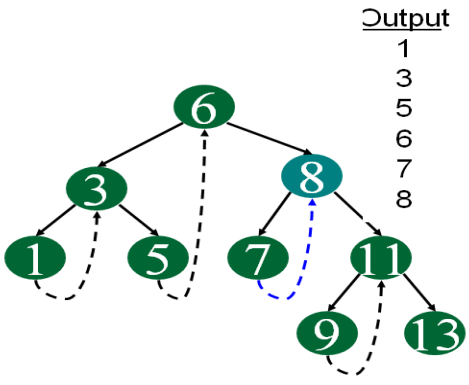
Follow link to right, go to leftmost node and print



Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow thread to right, print node

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.

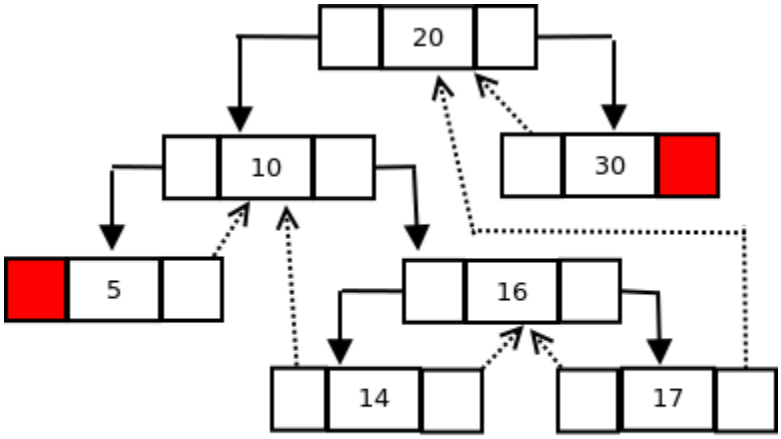
```
root = tmp;
tmp -> left = NULL;
tmp -> right = NULL;
```

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.

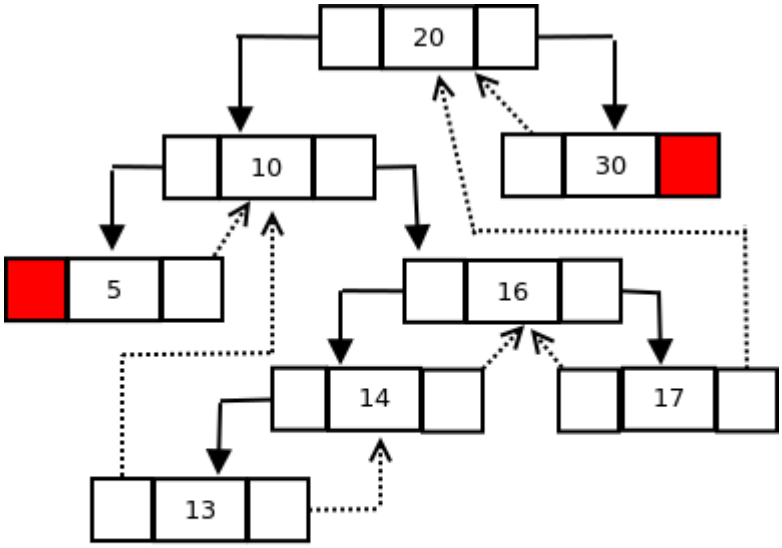
```
par -> lthread = par ->left;
par -> left = temp;
```



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

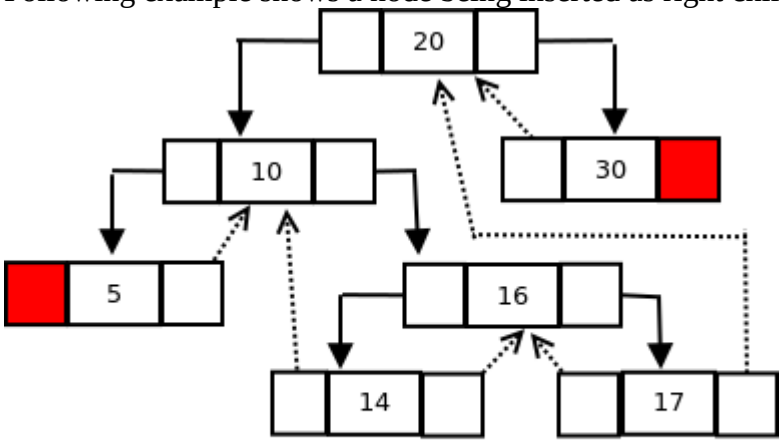
Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10.
Successor of 13 is 14, so right thread of 13 points to left child which is 13.
Left pointer of 14 is not a thread now, it points to left child which is 13.

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-

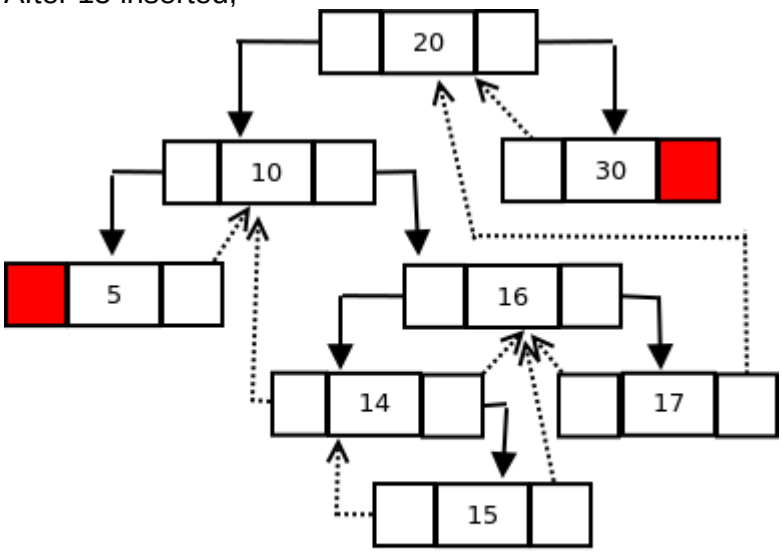
tmp -> left = par;
tmp -> right = par -> right;
Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

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,

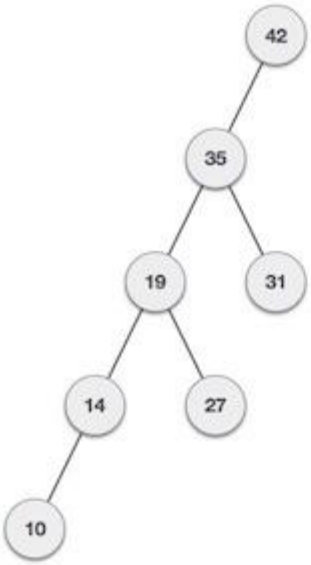


15 inserted as right child of 14
Inorder : 5 10 14 15 16 17 20 30

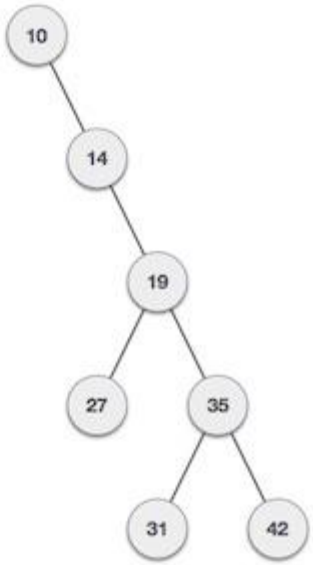
Successor of 14 becomes the successor of 15, so right thread of 15 points to 16
Predecessor of 15 is 14, so left thread of 15 points to 14.
Right pointer of 14 is not a thread now, it points to right child which is 15.

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

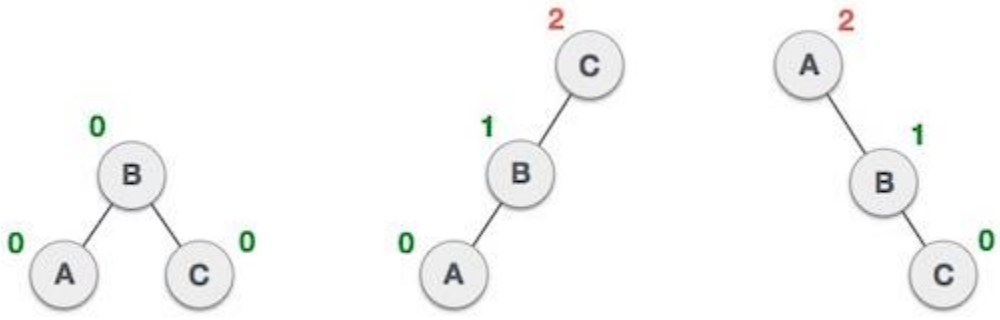


If input 'appears' in non-decreasing manner

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 −



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.

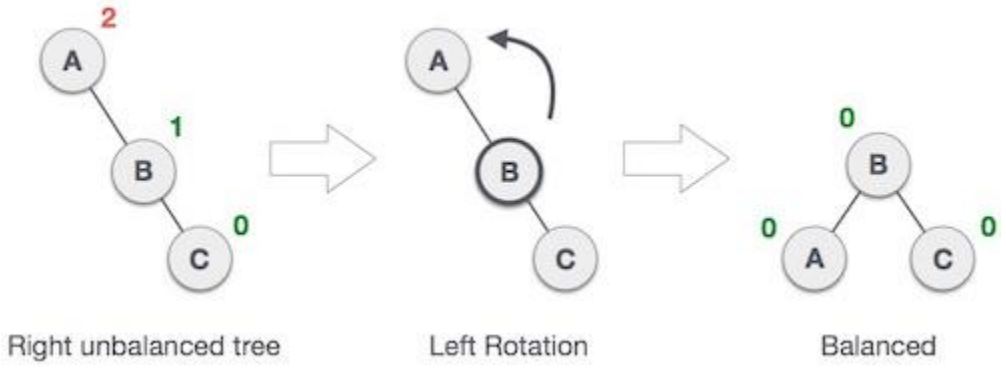
BalanceFactor = height(left-sutree) − height(right-sutree)
If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

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

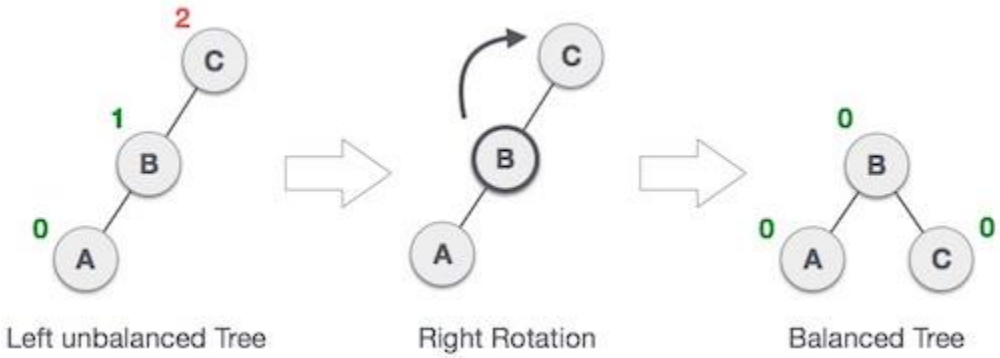
The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

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 −



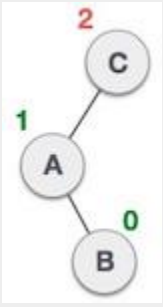
In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

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.

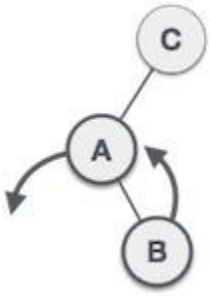


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

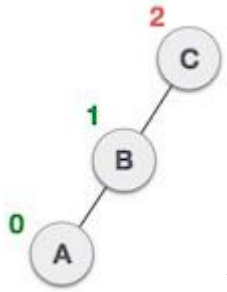
Left-Right Rotation:Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation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

State	
	

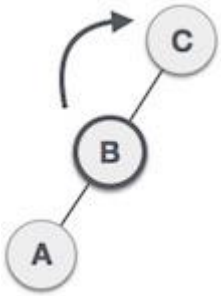
A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.



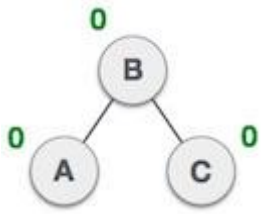
We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.



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



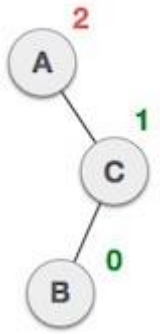
We shall now right-rotate the tree, making **B**the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.



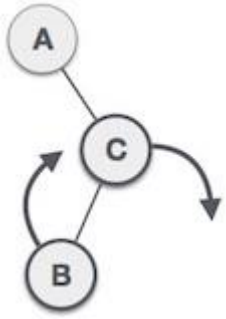
The tree is now balanced.

Right-Left Rotation:

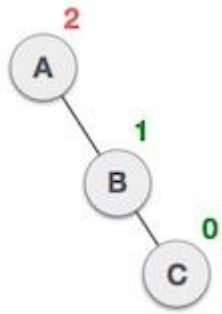
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation



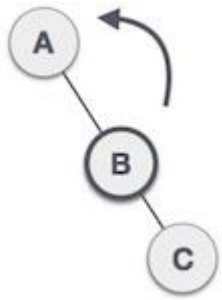
A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.



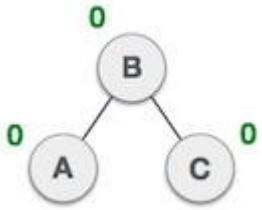
First, we perform the right rotation along **C**node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



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...

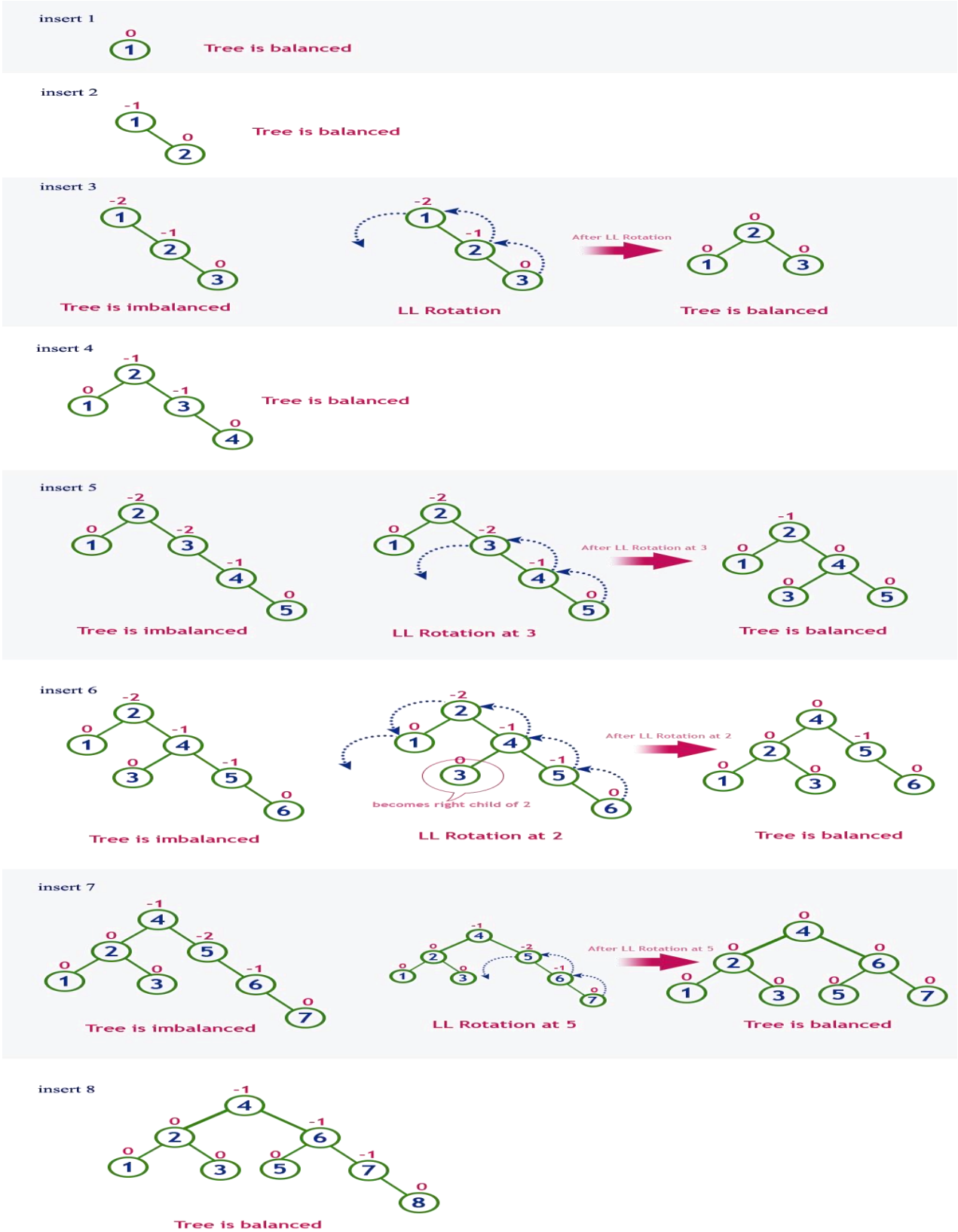
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

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

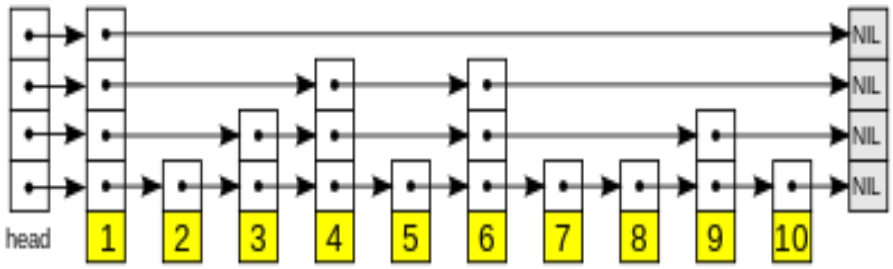
Deletion Operation in AVL Tree:

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.



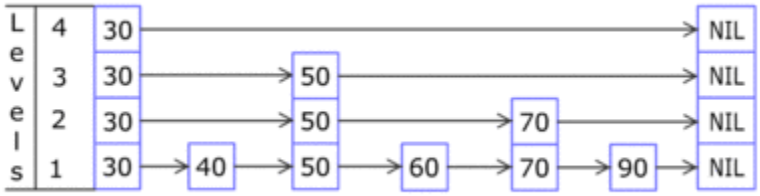
A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes (in yellow) at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the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

POSSIBLE QUESTIONS

UNIT-III

PART-A (20 MARKS)

(Q.NO 1 TO 20 Online Examination)

PART-B (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.

PART-C (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 O
Coimbatore -
(For the candidates admitte

DEPARTMENT OF COMPU

UNIT III :(Objective Type/Multiple choice Qu
PART-A (Online Ex

1	_____ are genealogical charts which are used to present the data	Graphs
2	A __ is a finite set of one or more nodes, with one root node and remaining form the disjoint sets forming the subtrees.	tree
3	A _____ is a graph without any cycle.	tree
4	In binary trees there is no node with a degree greater than _____	zero
5	Which of this is true for a binary tree.	It may be empty
6	The Number of subtrees of a node is called its _____.	leaf
7	Nodes that have degree zero are called _____.	end node
8	A binary tree with all its left branches supressed is called a _____	balanced tree
9	All node except the leaf nodes are called_____.	terminal node
10	The roots of the subtrees of a node X, are the _____ of X.	Parent
11	X is a root then X is the _____ of its children.	sub tree
12	The children of the same parent are called _____.	sibiling
13	_____of a node are all the nodes along the path form the root to that node.	Degree
14	The _____of a tree is defined to be a maximum level of any node in the tree.	weight
15	A_____ is a set of $n \geq 0$ disjoint trees	Group
16	A tree with any node having at most two branches is called a _____.	branched tree
17	A _____of depth k is a binary tree of depth k having 2^K-1 nodes.	full binary tree
18	Data structure represents the hierarchical relationships between individual data item is known as _____.	Root
19	Node at the highest level of the tree is known as _____.	Child
20	The root of the tree is the _____of all nodes in the tree.	Child
21	_____is a subset of a tree that is itself a tree.	Branch
22	A node with no children is called _____.	Root Node
23	In a tree structure a link between parent and child is called _____	Branch
24	Height – balanced trees are also referred as as _____trees.	AVL trees
25	Visiting each node in a tree exactly once is called _____	searching

26	In_____traversal ,the current node is visited before the subtrees.	PreOrder
27	In_____traversal ,the node is visited between the subtrees.	PreOrder
28	In_____traversal ,the node is visited after the subtrees.	PreOrder
29	Inorder traversal is also sometimes called_____	Symmetric Order
30	Postorder traversal is also sometimes called_____	Symmetric Order
31	Nodes of any level are numbered from _____	Left to right
32	_____ search involves only addition and subtraction.	binary
33	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
34	Binary trees are used in _____ sorting.	quick sort
35	The ____ of the heap has the largest key in the tree.	Node
36	In Threaded Binary Tree ,LCHILD(P) is a normal pointer When LBIT(P) = ____	1
37	In Threaded Binary Tree ,LCHILD(P) is a Thread When LBIT(P) = ____	1
38	In Threaded Binary Tree ,RCHILD(P) is a normal pointer When RBIT(P) = ____	2
39	In Threaded Binary Tree ,RCHILD(P) is a Thread When LBIT(P) = ____	1
40	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search
41	_____ algorithm can be used only with sorted lists.	Linear search
42	_____ search involves comparision of the element to be found with every elements in a list.	Linear search
43	Binary search algorithm in a list of n elements takes only _____ time.	$O(\log_2 n)$
44	_____ is used for decision making in eight coin problem.	trees
45	The Linear search algorithm in a list of n element takes _____ time to compare in worst case.	constant
46	Which of these is an application of trees.	Finding minimum cost spanning tree
47	_____ is an operation performed on sets	union
48	In sets _____ is used to find the set containing the element i	subset(i)
49	Sets are represented as _____	arrays
50	_____ is an example of application of trees in decision making.	Binay search
51	In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called _____	threads
52	The _____ of each element in a binary tree are ordered.	Trees
53	Tree is _____the no of nodes is equal to 0	subtree
54	The _____ operation on binary search tree is similar to applying binary search technique to an sorted linear array	search
55	The deletion of a _____ node that has only a single child is also easily.	leaf

56	A _____ is a complete binary tree that is also a max tree.	max heap
57	In any binary tree linked list representation, if there are 2N number of reference fields, then _____ number of reference fields are filled with NULL	N+1
58	If there is no in-order predecessor or in-order successor, then it point to _____node.	root
59	_____ pointer does not play any role except indicating there is no link	root
60	_____ which make use of NULL pointer to improve its traversal processes	Binary Tree

F HIGHER EDUCATION
641021.
d from 2016 onwards)

TER SCIENCE,CA & IT

uestions each Question carries one Mark)
:amination)

Pedigree and lineal chart	Line , bar chart	pie chart	Pedigree and lineal chart
graph	list	set	tree
path	set	list	tree
one	two	three	two
The degree of all nodes must be <=2	It contains a root node	All the above	All the above
terminal	children	degree	degree
leaf nodes	subtree	root node	leaf nodes
left sub tree	full binary tree	right skewed tree	right skewed tree
percent node	non terminal	children node	non terminal
Children	Sibling	sub tree	Children
Parent	Sibilings	subordinate	Parent
leaf	child	subtree	sibiling
sub tree	Ancestors	parent	Ancestors
length	breath	height	height
forest	Branch	sub tree	forest
sub tree	binary tree	forest	binary tree
half binary tree	sub tree	n branch tree	full binary tree
Node	Tree	Address	Tree
Root	Sibiling	Parent	Root
Parent	Ancestor	Head	Ancestor
Root	Leaf	Subtree	Subtree
Branch	Leaf Node	Null tree	Leaf Node
Root	Leaf	Subtree	Branch
Binary Trees	Subtree	Branch Tree	AVL trees
travering	walk through	path	travering

PostOrder	Inorder	End Order	PreOrder
PostOrder	Inorder	End Order	Inorder
PostOrder	Inorder	End Order	PostOrder
End Order	PreOrder	PostOrder	Symmetric Order
End Order	PreOrder	PostOrder	End Order
Right to Left	Top to Bottom	Bottom to Top	Left to right
fibonacci	sequential	non sequential	fibonacci
radix	merge	heap	heap
merge sort	heap sort	lrsort	heap sort
Root	Leaf	Branch	Root
2	3	0	1
2	3	0	0
1	3	0	1
2	0	4	0
Binary search	fibonacci search	None of the above	Binary search
Binary search	insertion sort	merge sort	Binary search
Binary search	fibonacci search	None of the above	Linear search
O(n)	$O(n^3)$	$O(n^2)$	$O(\log_2 n)$
graphs	linked lists	array	trees
linear	quadratic	exponential	constant
Decision tree	Storage management	Job sequencing	Decision tree
sort	rename	traverse	union
Disjoin(i)	Union(i)	Find(i)	subset(i)
linked lists	graphs	trees	trees
Optimal merge pattern	Eight Coins problem	Huffman's Message coding	Eight Coins problem
nodes	pointers	tree	threads
subtrees	binary tree	thread	subtrees
one	empty	none	empty
insert	delete	display	search
tree	nonleaf	nonleaf tree	nonleaf

min heap	heap	max-min	max heap
2n+1	2n	n+1	N+1
leaf nodes	null	empty	root
leaf nodes	null	empty	null
Threaded Tree	non Threaded Binary Tree	Threaded Binary Tree	Threaded Binary Tree

SYLLABUS

UNIT-IV

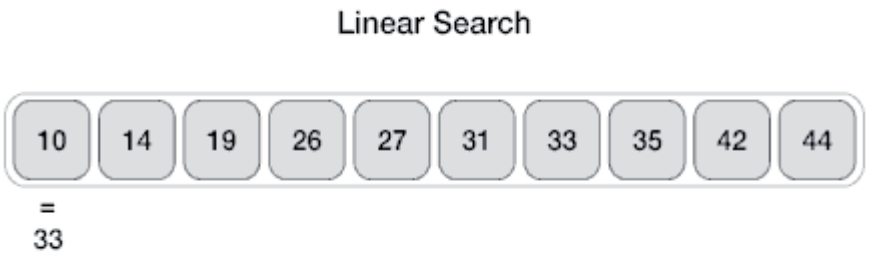
Searching and Sorting: Linear Search, Binary Search, Comparison of Linear and Binary Search, Selection Sort, Insertion Sort, Insertion Sort, Shell Sort, Comparison of Sorting Techniques

SEARCHING:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

LINEAR SEARCH:

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm:

Linear Search (Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Pseudocode

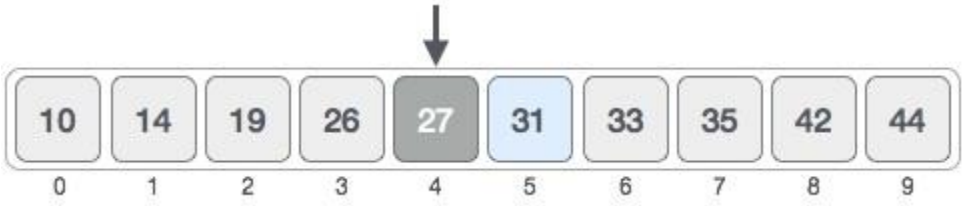
```
procedure linear_search (list, value)
   for each item in the list
  if match item == value
```

```
    return the item's location
    end if
  end for
end procedure
```

BINARY SEARCH:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.
How Binary Search Works:
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −
mid = low + (high - low) / 2
Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

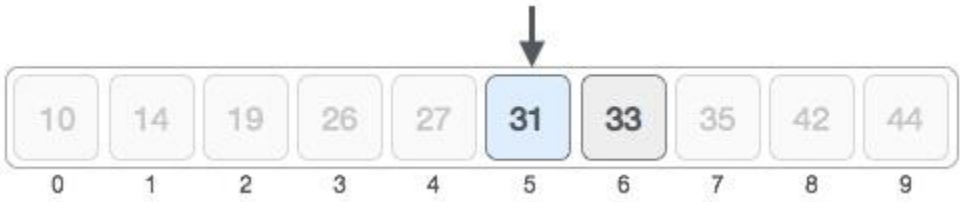


Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = low + (high - low) / 2
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this –

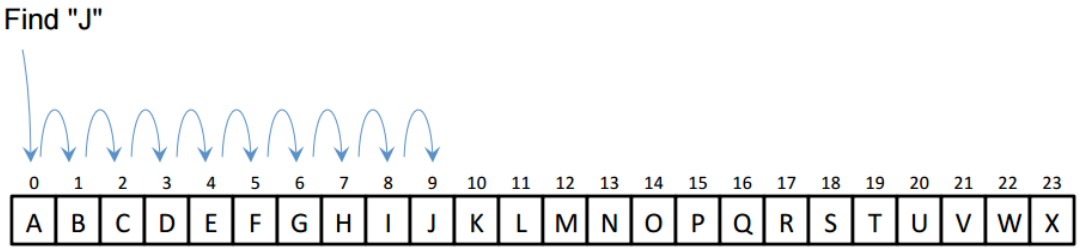
Procedure binary_search

```
A ← sorted array
n ← size of array
x ← value to be searched

Set lowerBound = 1
Set upperBound = n

while x not found
  if upperBound < lowerBound
    EXIT: x does not exists.

  set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

  if A[midPoint] < x
    set lowerBound = midPoint + 1

  if A[midPoint] > x
    set upperBound = midPoint - 1

  if A[midPoint] = x
    EXIT: x found at location midPoint
 end while
end procedure
```

Comparison of Linear Search vs Binary Search:

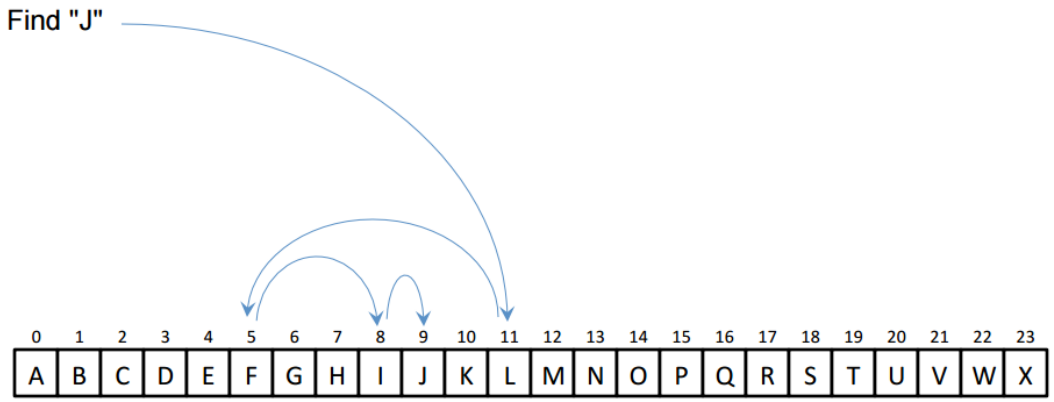
Linear Search
Binary Search
A **linear search** scans one item at a time, without jumping to any item .
The worst case complexity is O(n), sometimes known an O(n) search
Time taken to search elements keep increasing as the number of elements are increased.

A **binary search** however, cut down your search to half as soon as you find middle of a sorted list.
The middle element is looked to check if it is greater than or less than the value to be searched.
Accordingly, search is done to either half of the given list
Important Differences
Input data needs to be sorted in Binary Search and not in Linear Search
Linear search does the sequential access whereas Binary search access data randomly.
Time complexity of linear search -O(n) , Binary search has time complexity O(log n).
Linear search performs equality comparisons and Binary search performs ordering comparisons
Let us look at an example to compare the two:
Linear Search to find the element "J" in a given sorted list from A-X



linear-search

Binary Search to find the element "J" in a given sorted list from A-X



binary-search

SORTING:

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term **Sorting** comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Selection Sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

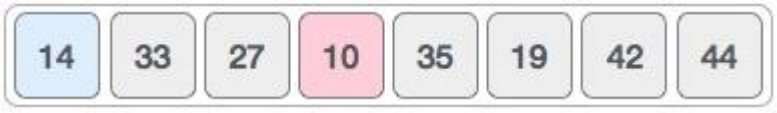
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works:

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process −

Algorithm:

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

Pseudocode:

```
procedure selection sort
   list  : array of items
    n     : size of list
for i = 1 to n - 1
   /* set current element as minimum*/
      min = i
    /* check the element to be minimum */
    for j = i+1 to n
        if list[j] < list[min] then
           min = j;
        end if
      end for
   /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
```

end for

end procedure

Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

Insertion Sort Works:

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the

value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Pseudocode

```
procedure insertionSort ( A : array of items )
  int holePosition
  int valueToInsert
  for i = 1 to length(A) inclusive do:
    /* select value to be inserted */
    valueToInsert = A[i]
```

```
    holePosition = i

   /*locate hole position for the element to be inserted */

  while holePosition > 0 and A[holePosition-1] > valueToInsert do:

   A[holePosition] = A[holePosition-1]

      holePosition = holePosition -1

    end while

  /* insert the number at hole position */

    A[holePosition] = valueToInsert

  end for

end procedure
```

Shell Sort:

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

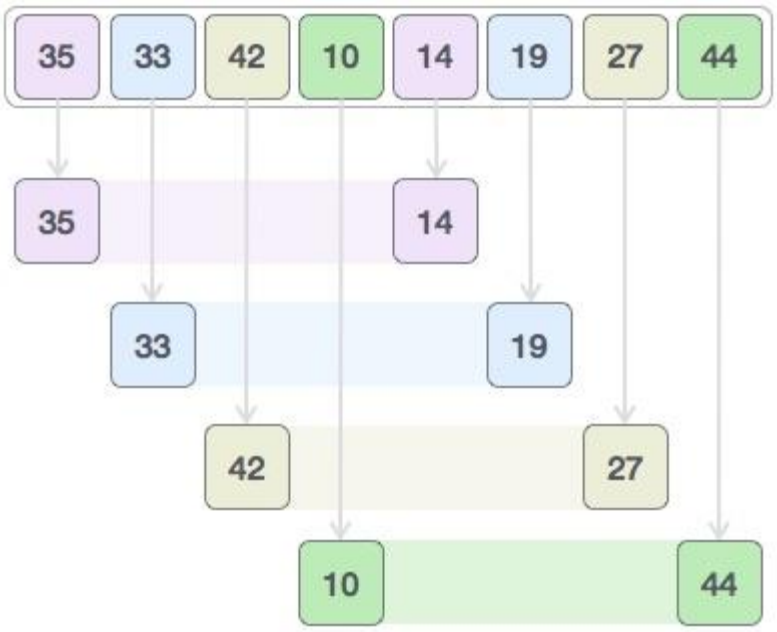
This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as −

Knuth's Formula

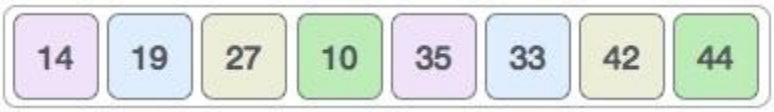
h = h * 3 + 1 where − h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of O(n), where n is the number of items.

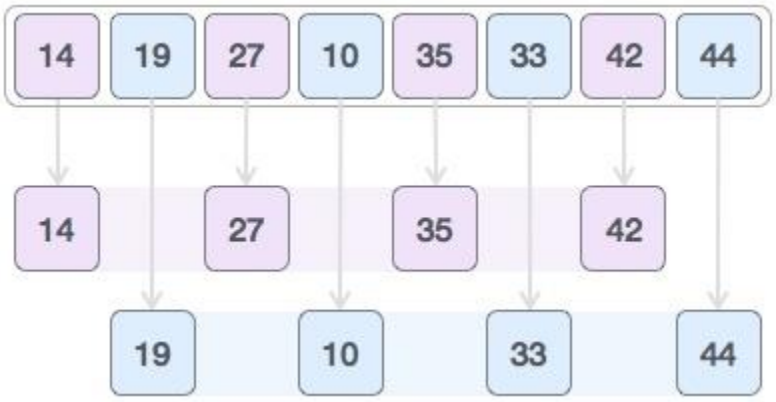
Shell Sort Works: Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



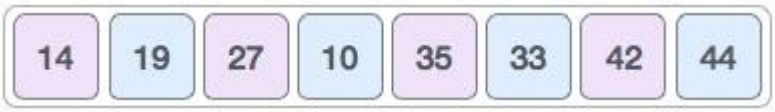
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −

We see that it required only four swaps to sort the rest of the array

Algorithm:

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using **insertion sort**
Step 3 − Repeat until complete list is sorted

Pseudocode:

Following is the pseudocode for shell sort.

```
procedure shellSort()
   A : array of items
 /* calculate interval*/
  while interval < A.length /3 do:
     interval = interval * 3 + 1
  end while
     while interval > 0 do:
   for outer = interval; outer < A.length; outer ++ do:
   /* select value to be inserted */
     valueToInsert = A[outer]
     inner = outer;
 /*shift element towards right*/
        while inner > interval -1 && A[inner - interval] >= valueToInsert do:
           A[inner] = A[inner - interval]
           inner = inner - interval
        end while
 /* insert the number at hole position */
     A[inner] = valueToInsert
 end for
 /* calculate interval*/
```

```
interval = (interval -1) /3;
end while
end procedure
```

Comparison of Sorting Techniques:

Sorting: The process of ordering of elements is known as sorting. It is very important in day to day life. Nor we neither computer can understand the data stored in an irregular way. Sorting of comparisons can be done on the basis of complexity.

Complexity: Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).There are two types of complexity: 1.Space complexity 2. time complexity

Space complexity measures the space used by algorithm at running time. **Time complexity** for an algorithm is different for different devices as different devices have different speeds so, we measure time complexity as the no. of statements executed indifferent cases of inputs.

SORTING TECHNIQUES

1**.Selection Sorting**:-In selection sort we find the smallest number and place it at first position, then at second and so on.

Complexity: - An array in sorted or unsorted form doesn't make any difference. It is same in both best & worst cases. The first pass makes (n-1) comparisons to find smallest number, second pass makes (n-2) and so on, then Time Complexity T(n) will be :

2**.Insertion Sort**: -It takes list in two parts, sorted list and unsorted list. In this sorting technique, first element of unsorted list gets placed in previous sorted list and runs till all elements are in sorted list.

Complexity:-

Best Case: -All elements are sorted or almost sorted. Therefore, comparison occurs atleast one time in inner loop, then time Complexity T(n) will be

Average Case: - We consider that there will be approximately (n-1)/2 comparisons in inner loop.

Worst Case: - In this case comparison in inner loop is done almost one in first time, 2 times in second turn, and (n-1) times in (n-1) turns.

3.**Shell Sort**: - This technique is mainly based on insertion sort. In a pass it sorts the numbers when are separated at equal distance. In each consecutive pass distance will be gradually decreases till the distance becomes 1. It uses insertion sort to sort elements with a little change in it.

Complexity: - Shell sort analysis is very difficult some time complexities for certain sequences of increments are known.

Base Case: - O (n)
Average Case: - nlog 2n or n 3/2
Worse Case: - It depends on gap sequence. The best known is nlog 2n.

POSSIBLE QUESTIONS

UNIT-IV

PART-A (20 MARKS)

(Q.NO 1 TO 20 Online Examination)

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.

S.NO	QUESTIONS
1	In a graph G(V,E), V is a finite non-empty set of _____ and E is a set of edges.
2	Iff the degree of each vertex is even, a walk starting from one vertex and going through all the other vertices exactly once and returning to the starting vertex is called _____
3	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.
4	In a Graph G if there are n vertices the adjacency list then consists of _____ nodes.
5	In a Graph G if there are n vertices the adjacency Matrix of the graph consists of _____ rows and colums.
6	In _____ graph the pair of vertices joined by any edge is unordered.
7	In_____ graph each edge is represented by the directed pair<v1,v2>.
8	The_____ of a path is the number of edges on it.
9	An n vertex undirected graph with exactly n(n-1)/2 distinct edges is said to be _____
10	A_____ is a simple path in which the first & last vertices are the same.
11	A connected component of an undirected graph is a _____ connected subgraph.
12	A _____ is a connected acyclic graph.
13	In a graph with n vertices the number of distinct unordered pairs(vi,vj) with vi not equal to vj is _____
14	In a graph ____ of a vertex is the number of edges incident to it.
15	The _____ of a vertex is defined as the number of edges for which v is the head.
16	The _____ is defined to be the number of edges for which v is the tail
17	Directed graph is also called _____.
18	In a directed graph G(V,E) a vertex vj is _____ vi iff there is an edge <vi,vj> in E
19	A vertex vi is adjacent to vj iff _____
20	An edge connected by any 2 vertices i & j can be determined by_____
21	An edge <vi,vj> is said to _____ on two vertices vi and vj.
22	A _____ from vertex vi to vj is a seqence of vertices from vi to vj with an edge connecting them in pairs
23	The _____ of a path is the number if edges on it.

24	The adjacency matrix of an undirect graph is always _____.
25	A graph with weighted edge is called a _____.
26	Any tree consisting solely of edges in G and including all vertices in G is called _____.
27	The spanning tree resulting from a call to DFS is known as a _____ spanning tree.
28	When BFS is used the resulting spanning tree is called a _____ spanning tree.
29	_____ and _____ are the searching techniques used in graphs.
30	Starting from a vertex v and visiting all vertices adjacent to it before moving to the next vertex is called _____ search
31	The all pairs shortest path problem calls for finding the _____ paths between all pairs of vertices.
32	A _____ graph is a graph in which each vertex of G is adjacent to every other vertex in G.
33	_____ is a collection of records, each record having one or more fields.
34	A directed graph with no directed cycle is called _____
35	If i is the predecessor of j in a network then i preceed j in the linear ordering. Such an ordering in graphs is called _____
36	In a AOV network if i is a predecessor of j then j is the _____ of i
37	A precedence relation which is both transitive and irreflexive is a _____
38	For a network with n vertices and e edges the asymptotic computing time of the topological ordering algorithm is _____
39	For a network with n vertices and e edges the asymptotic computing time of the topological ordering algorithm is _____
40	A directed graph in which the vertecies represemt tasks or activities and edges represent precedence relation between tasks is _____
41	All connected graphs with n-1 edges are called __
42	_____ of a undirected graph each edge <vi,vj> is represented by two entries, one on the list for vi and the other on the list for vj.
43	Introducing any one edge into a spanning tree will result in a ____
44	The cost of spanning tree is the _____ cost of the edges in that tree
45	_____ assigned to the edges of a graph are called its cost or length of the link.
46	All algorithms using adjacency matrix will require atleast _____ time.
47	The fields used to distinguish among the records are known as _____.
48	In _____ search method the search begins by examining the record in the middle of the file.
49	_____ search involves only addition and subtraction.
50	Kruskal formulated a method to determine _____ in graphs.

51	_____ sort is done in graphs
52	An _____ requires that the collection of data fit entirely in the computer's main memory.
53	An _____ when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
54	_____ are a specific type of Algorithms that specialize in taking in multiple sorted lists and merging them into a single sorted list.
55	A _____ data structure, the size of the structure is fixed.
56	A _____sort is one in which successive elements are selected.
57	The straight selection sort is also known as _____
58	An _____is one that sorts a set of records by inserting records into an existing sorted file.
59	The _____sorts separate subfilesof the original file.
60	A table of records in which a key is used for retrieval is called a_____ .

HIGHER EDUCATION
1021.
from 2016 onwards)

R SCIENCE,CA & IT

stions each Question carries one Mark)
nination)

OPTION 1	OPTION 2	OPTION 3	OPTION 4	KEY
Nodes	Items	Vertices	Circles	Vertices
Kruskals path	Hamiltonian cycle	Eulerian walk	Koenisberg bridge	Eulerian walk
connected	adjacent	neighbours	incident	connected
n/2	2n	n-1	n	n
n/2	2n	n-1	n	n
directed	undirected	sub	multi	undirected
undirected	multi	directed	sub	directed
tree	maximal	length	cycle	length
connected	complete	directed	cyclic	complete
Cycle	graph	component	matrix	Cycle
tree	strongly	weekly	maximal	maximal
graph	component	tree	list	tree
n(n-1)/2	n-1	n(n-1)	n/2	n(n-1)/2
path	degree	depth	height	degree
out-degree	pre-degree	in-degree	post-degree	in-degree
in-degree	out-degree	pre-degree	post-degree	out-degree
Line Graph	sub graph	connected graph	di graph	di graph
adjacent from	adjacency matrix	adjacent from	adjacency list	adjacent from
vi=vj	<vi,vj> is an edge in E	if it belongs to a graph	vi is starting vertex	<vi,vj> is an edge in E
sparse matrix	adjacency matrix	linked lists	tree graph	adjacency matrix
parallel	incident	degree	loop	incident
path	length	cycle	tree	path
degree	length	degree	height	length

Unit Matrix	Diagonal Matix	Identity Matrix	SymetricMatrix	SymetricMatrix
strong graph	network	component	sub graph	network
graph	spanning tree	tri graph	bread the first spanning tree	spanning tree
Independent	breadth first	depth first	dependent	depth first
breadth first	depth first	independent	dependent	breadth first
Inorder, preorder	firstfit, bestfit	depth first, breadth first	prefix, postfix	depth first, breadth first
breadth first	depth first	pre order	first fit	breadth first
longest	shortest	minimal	maximal	shortest
Complete	Incomplete	connected	un connected	Complete
data base	file	directory	address	file
topological graph	subgraph	connected graph	acyclic graph	acyclic graph
Heap sort	Merge sort	Topological sort	linear search	Topological sort
root	successor	path	ancester	successor
spanning tree	depth first searching	partial ordering	topological sorting	partial ordering
O(e+n)	O(e)	O(n)	O(en)	O(e+n)
linear	constant	quadratic	exponential	linear
activity on vertex network	topological network	complete network	precedence relation network	activity on vertex network
digraphs	cyclic graphs	trees	subgraphs	trees
Adjacency list	Inverse adjacency list	Adjacency multilist	adjacency matrix	Adjacency list
cycle	component	digraph	tree	cycle
maximum	average	sum	minimum	sum
weights	size	depth	degree	weights
$O(n^2)$	O(n)	$O(n^3)$	O(2n)	$O(n^2)$
records	address	pointers	keys	keys
sequential	fibonacci	binary	non-sequential	binary
Binary	fibonacci	sequential	non sequential	fibonacci
Breadth first search	connected component	adjacency matrix	minimum cost spanning tree	minimum cost spanning tree

Merge sort	Heap	Topological sort	Linear sort	Topological sort
Internal sort	external sort	sorting	searching	Internal sort
Internal sort	external sort	sorting	searching	external sort
Multiway	Merges	Multiway Merges	Dynamic Merges	Multiway Merges
static	dynamic	non terminal	non sequential	static
insertion	deletion	Selection	shell	Selection
push-down sort	selection	shell sort	shell	push-down sort
down sort	insertion sort	selection	shell	insertion sort
down sort	insertion sort	shell sort	shell	shell sort
insert table	delete table	records	search table	search table

SYLLABUS

UNIT-V

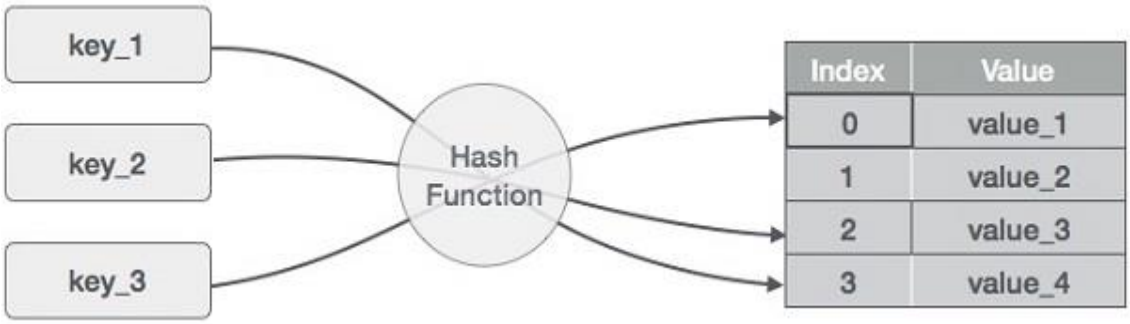
Hashing - Introduction to Hashing, Deleting from Hash Table, Efficiency of Rehash Methods, Hash Table Reordering, Resolving collusion by Open Addressing, Coalesced Hashing, Separate Chaining, Dynamic and Extendible Hashing, Choosing a Hash Function, Perfect Hashing, Function

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 :

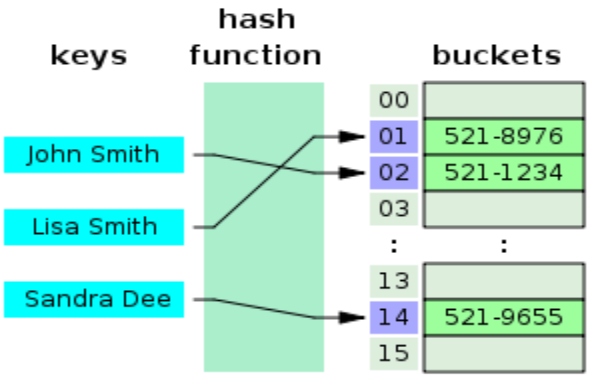
When two different keys produce the same address, there is a **collision**. The keys involved are called **synonyms**. Coming up with a hashing function that avoids collision is extremely difficult. It is best to simply find ways to deal with them. **The possible solution, can be:**

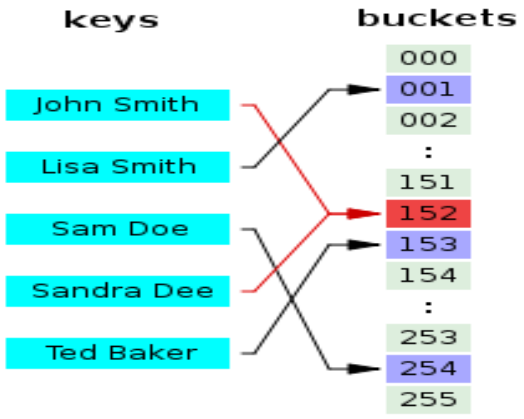
Spread out the records
Use extra memory
Put more than one record at a single address.
An example of Collision

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):



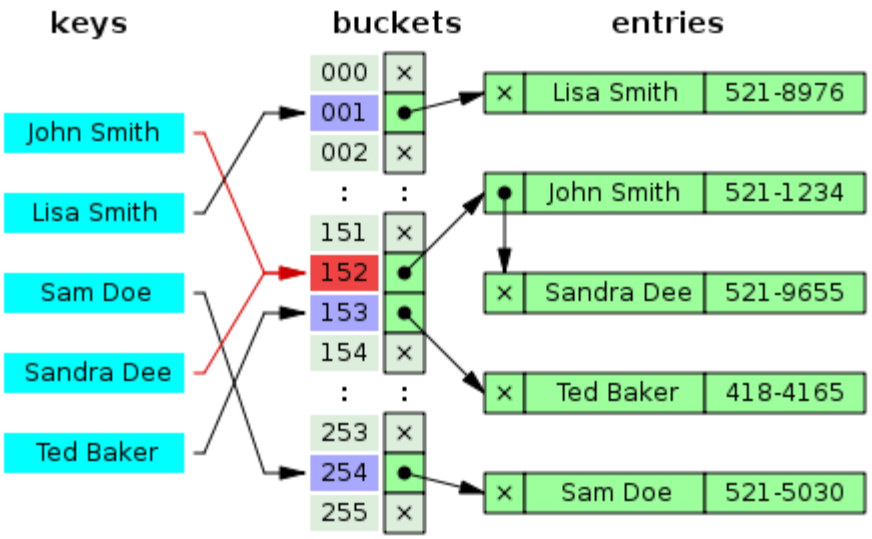


Here, the buckets for keys 'John Smith' and 'Sandra Dee' are the same. So, its a collision case.

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:



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)

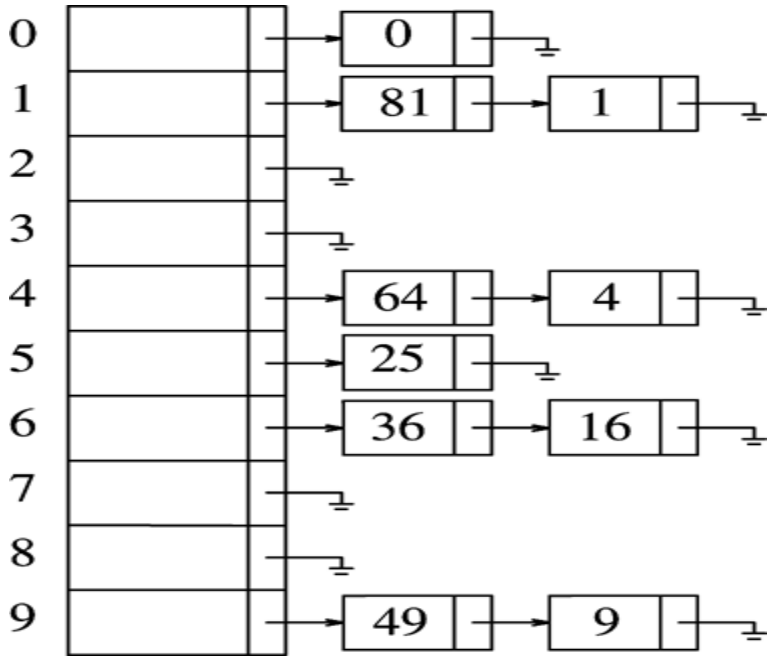
Hash table T is a vector of linked lists

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)

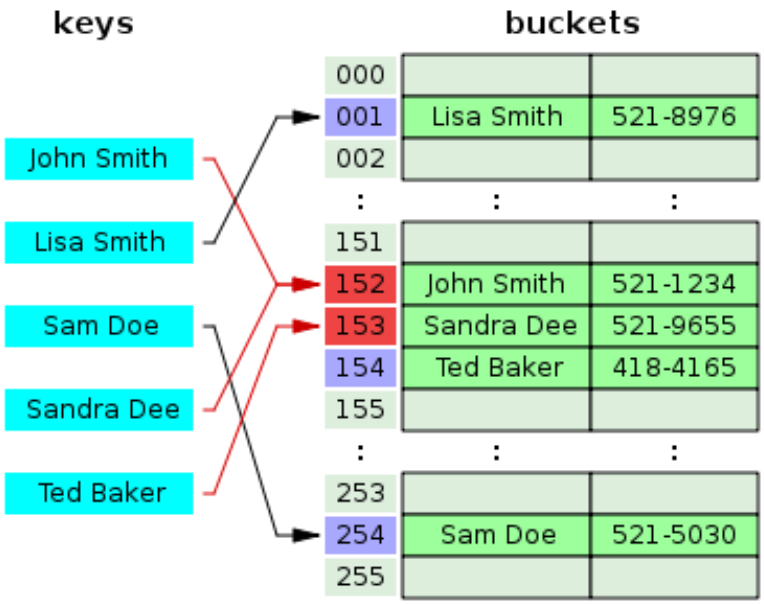
- Ideally, want λ ≤ 1 (not a function of N)

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:



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

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

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:

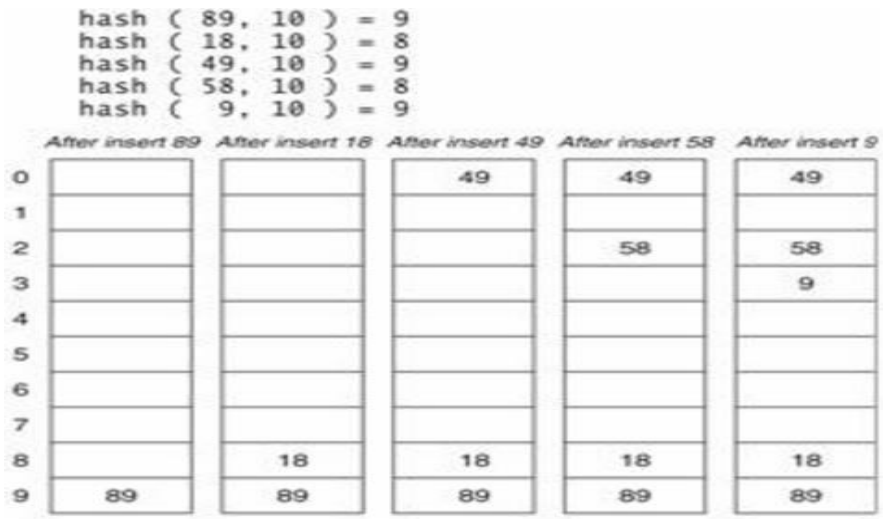
- Probe sequences can get longer with time

- 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:

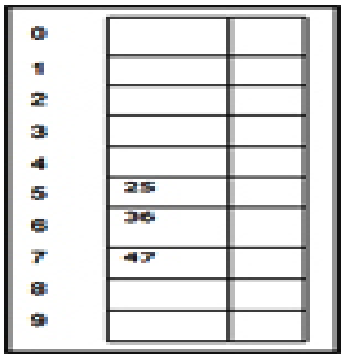
The chaining method discussed above requires additional space for maintaining pointers. The table stores only pointers but each node of the linked list requires

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.

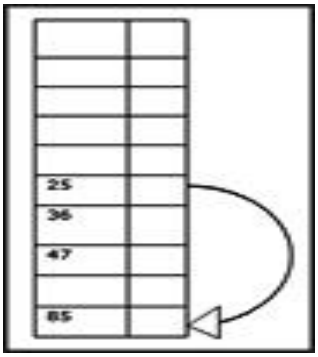
The solution to this problem is called coalesced hashing or coalesced chaining. This method is the hybrid of chaining and open addressing. Each index position in the table stores key value and a pointer to the next index position. The pointer generally points to the index position where the colliding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

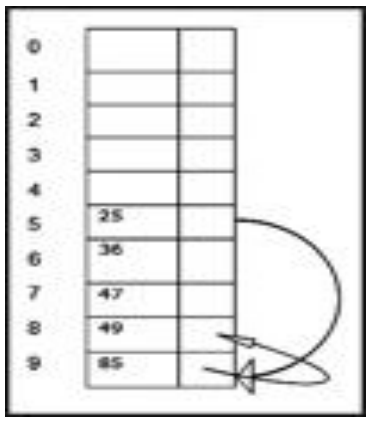
For Example, the values 25, 36, and 47 will be inserted thus in the table –



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.

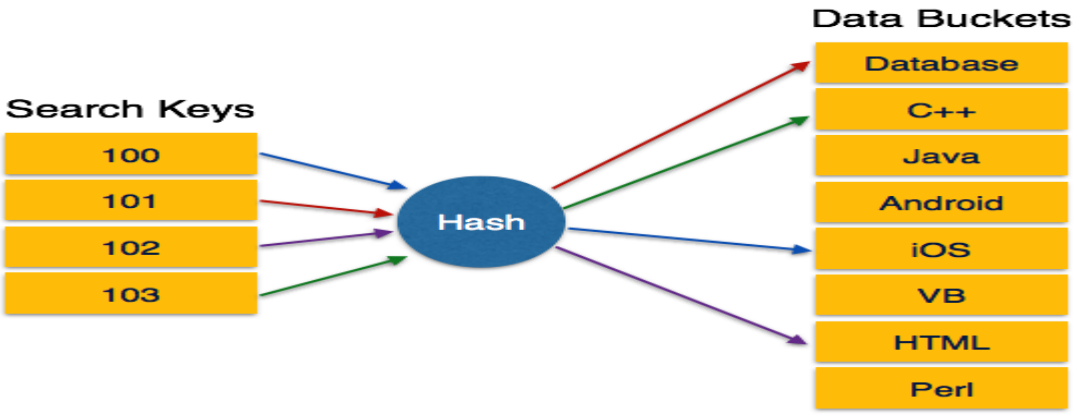
Hash Organization:

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)

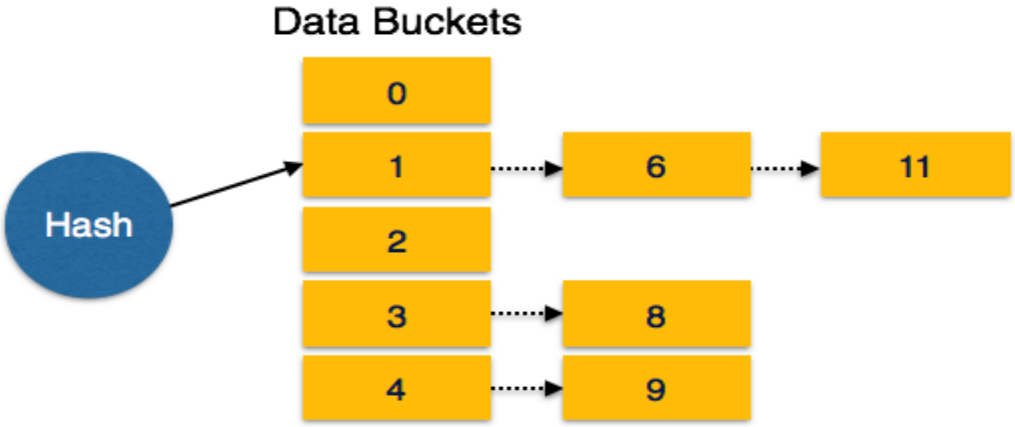
- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

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

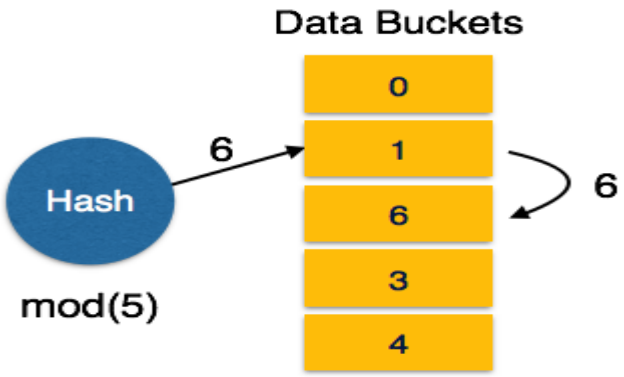
Bucket Overflow

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**.



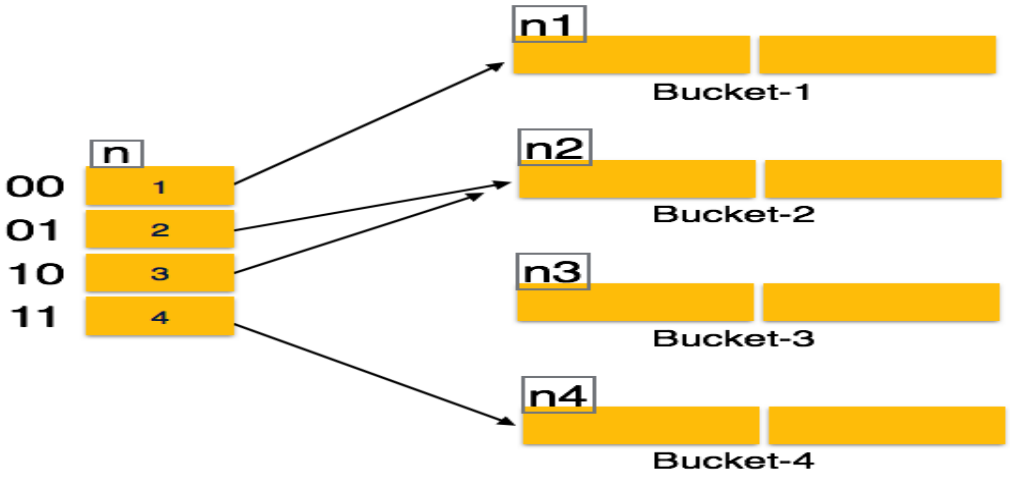
- **Linear Probing** − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



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.

POSSIBLE QUESTIONS

UNIT-V

PART-A (20 MARKS)

(Q.NO 1 TO 20 Online Examination)

PART-B (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.

PART-C (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.

S.NO
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

KARPAGAM ACADEMY OF HIGHER EDUCATION
Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

KARPAGAM ACA
Coi
(For the candidate

DEPARTMENT OF

UNIT V :(Objective Type/Multiple
PART-A

QUESTIONS
The sum over all internal nodes of the length of the paths from the root to those node is called
_____ is the sum over all external nodes of the lengths of paths from the root to those nodes.
The _____node are not a part of original tree and are represented as square nodes.
The external nodes are in a binary search treeare also known as _____ nodes
A binary tree with external nodes added is an ---------- binary tree
_____ is a set of name-value pairs.
Each name in the symbol tales is associated with an___
This is not an operation perform on the symbol table.
If the identifiers are known in advance and no deletion/insertions are allowed then this symbol
The cost of decoding a code word is ---------- to the number of bits in the code
The solution of finding a binary tree with minimum weighted external path length has been given
_____ symbol table allows insertion and deletion of names.
_____ is an application of Binary trees with minimal weighted external path lengths.
If hl and hr are the heights of the left and right subtrees of a tree respectively and if |hl-hr|<=1
If hl and hr are the heights of the left and right subtrees of a tree respectively then |hl-hr| is called
For an AVL Tree the balance factor is =____
If the names are _____ in the symbol table, searching is easy.
_____ allocation is not desirable for dynamic tables, where insertions and deletions are
A search in a hash table with n identifiers may take ------time
_____ data structure is used to implement symbol tables
Every binary search tree wth n nodes has _____ sqare node (external nodes).
In a Hash table the address of the identifier x is obtained by applying
The partitions of the hash table are called _____
The arithmetic functions used for Hashing is called _____
Each bucket of Hash table is said to have several _____

A_____ occurs when two non_identical identifiers are hashed in the same bucket.
A hashing function f transforms an identifier x into a _____ in the hash table
When a new identifier I is mapped or hashed by the function f into a full bucket then
If f(I) and F(J) are equal then Identifiers I and J are called_____
A ---tree is a binary tree in which external nodes represent messages
The identifier x is divided by some number m and the remainder is used as the hash address for x
The identifier is folded at the part boundaries and digits falling into the same position are added
In hash table, if the identifier x has an equal chance of hashing into any of the buckets, this
Each head node is smaller than the other nodes because it has to retain
Each chain in the hash tables will have a
Folding of identifiers fron end to end to obtain a hashing function is called ____
Average number of probes needed for searching can be obtained by ---------- probing
Rehashing is _____
_____ is a method of overflows handling.
The number of _____ over the data can be reduced by using a higer order merge (k-way
A _____ is a binary tree where each node represents the smaller of its two children
In External sorting data are stored in _____
_____ techiniques are used for sorting large files
In _____ a k-way merging uses only k+1 tapes
Before merging the next phase is is necessary to _____ the output tapes
To reduce the rewind time it is overlapped with read/write on other tapes. This modification need
Two records cannot occupy the same position such a situation is called _____
A general method for resolving hash clashes called _____
Two keys that hash into different values compete with each other in successive rehashes is
_____ involves two hash functions.
A hash table organized in this way is called an _____
The simplest of the chaining methods is called _____
Another method of resolving hash clashes is called _____
The most common hash function uses the _____ method
_____ function depends on every single bit of the key.
_____ method the key is multiplied by itself.
_____ method breaks up a key into several segments
No clashes occur under a _____ hash function.
A perfect hash function can be developed using a techinque called _____
In _____ hashing each bucket contains an indication of the no of bits.

DEMY OF HIGHER EDUCATION
nbatore - 641021.
s admitted from 2016 onwards)

COMPUTER SCIENCE,CA & IT

choice Questions each Question carries one Mark)
(Online Examination)

OPTION 1	OPTION 2	OPTION 3	OPTION 4
internal path length	external path length	depth of the tree	level of the tree
internal path length	external path length	depth of the tree	level of the tree
internal node	external node	intermediate node	terminal node
internal	search	failure	round
extended	expanded	internal	external
Symbol table	Graph	Node	Record
name value pairs	element	attribute	entries
insert a new name and	retrieve the attribute	search if a name is	Add or subtract two
static	empty	dynamic	automatic
equal	not equal	proportional	inversely
Huffman	Kruskal	Euler	Hamilton
Hashed	Sorted	Static	Dynamic
Finding optimal merge patterns	Storage compaction	Recursive Procedure calls	Job Scheduling
extended binary tree	binary search tree	skewed tree	height balanced tree
Average height	minimal depth	Maximum levels	Balance factor
0	-1	1	Any of the above
sorted	short	bold	upper case
Linear	Sequential	Dynamic	None
O(n)	O(1)	O(2)	O(2n)
directed graphs	binary search trees	circular queue	None
n/2	n+1	n-1	2^n
sequence of	binary searching	arithmetic function	collision
Nodes	Buckets	Roots	Fields
Logical operations	Rehashing	Mapping function	Hashing function
slots	nodes	fields	links

collision	contraction	expansion	Extraction
symbol name	bucket address	link field	slot number
underflow	overflow	collision	rehashing
synonyms	antonyms	hash functions	buckets
decode	uncode	extended	none
m mod x	x mod m	m mod f	none of these
folding at the	shift method	folding method	Tag method
Equal hash function	uniform hash	Linear hashing	unequal Hashing
only a link	only a link and a	only two link	only the record
tail node	link node	head node	null node
Shift folding	boundary folding	expanded folding	end to end folding
quadratic	linear	rehashing	Sequential
series of hash function	linear probing	quadratic functions	Rebuild function
linear open addressing	Adjacency lsit	sequential	Indexed address
records	passes	tapes	merges
search tree	decision tree	extended tree	selection tree
RAM memory	Cache memory	secondary storage	Buffers
Topological sort	External sorting	Linear Sorting	Heap sort
Internal soring	Polyphase merging	Linking	Hashing
replace	rewind	remove	None
double the number of	only two tapes	one additional tape	k+1 tapes
hash collision	hash	collision	clashes
hash collision	rehashing	hashing	collision
primary clustering	primary	clustering	collision
primary clustering	double hashing	double	hashing
ordered hash table	double hashing	double	hashing
standard hashing	standard coalesced	hashing	coalesced hashing
standard hashing	standard coalesced	separate chaining	coalesced hashing
division	hash	chaining	collision
hash	division	chaining	collision
folding	midsquare	hash functions	chaining
folding	midsquare	hash functions	chaining
folding	perfect	midsquare	collision
folding	segment	segmentation	perfect
folding	extendible	segmentation	perfect

KEY
internal path length
external path length
external node
failure
extended
Symbol table
attribute
Add or subtract two
static
proportional
Huffman
Dynamic
Finding optimal merge patterns
height balanced tree
Balance factor
Any of the above
sorted
Sequential
O(n)
binary search trees
n+1
arithmetic function
Buckets
Hashing function
slots

collision
bucket address
overflow
synonyms
decode
x mod m
folding at the
uniform hash
only a link
head node
boundary folding
quadratic
series of hash
linear open
passes
selection tree
secondary storage
External sorting
Polyphase merging
rewind
one additional tape
hash collision
rehashing
primary clustering
double hashing
ordered hash table
standard coalesced
separate chaining
division
hash
midsquare
folding
perfect
segmentation
extendible

Register Number_____

[16CSU301]

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore-641021.

B.Sc COMPUTER SCIENCE

FIRST INTERNAL EXAMINATION - JULY 2017

Third Semester

DATA STRUCTURES – ANSWER KEY

Class : II B.Sc(CS) **Duration : 2 Hours**

Date & Session : 17 .7.2017 **Maximum : 50 Marks**

SECTION A – (20 X 1 = 20 Marks)

ANSWER ALL THE QUESTIONS

1. A ------ is a specialized format for organizing and storing data.
   a) Data b) **Data structure** c) Data item d) entity
2. The term ---- simply refers to a value or set of values.
   a) Entity b) **Data** c) Data item d) Record
3. A --- is a collection of related data items.
   a) Data b) Entity c) **Record** d) None
4. An ---- is a list of finite number of elements of same data type.
   a**) Array** b) Linear array c) Non-linear d) List
5. A linked list is a linear collection of data elements called -----
   a) Pointer b) List c) Data d**) Nodes**
6. In ---- linked list each node is divided into three parts.
   a) Singly b) **Doubly** c) Circular d) None
7. A stack also called a ----- system.
   a) LIFO b**) FIFO** c) LILO d) FILO
8. A ---- is a tree that can have almost two children.
   a) **Binary tree** b) Tree c) Heap d) Graph
9. A ---- is an ordered set (V, E) of elements called nodes.
   a) Heap b) Tree c) **Graph** d) Sort
10. ----- is accessing each element exactly once.
    a) Searching b) **Traversal** c) Insertion d) Deletion
11. A matrix is set be ------ if many of its element are zero.
    a) Dense b) **Sparse** c) Diagonal d) Square
12. ------ returns the number of elements in the stack.
    a**) Size ( )** b) Empty ( ) c) pop ( ) d) push(x)
13. In ---- notation the operator symbol is placed before its two operands.
    a) Infix b) Postfix c) **Prefix** d) Suffix

14. The elements are stored column by column in ------ major order.
    a) **Column** b) Row c) Array d) mxn
15. A matrix is ------ iff B(i,j)=0 for i≠j
    a) **Diagonal** b) Tridiagonal c) Lower d) Upper
16. A ----- list is also called a two- way list
    a) Singly b) **Doubly** c) Circular d) All the above
17. A ----- list is a linear linked list except that the last element points to the first element.
    a) Singly b) Doubly c) **Circular** d) All the above
18. A stack represented using a linked list is also known as ------
    a**) Linked stack** b) Stack c) Array d) List
19. A Queue also called a ----- system.
    a) LIFO b) **FIFO** c) LILO d) FILO
20. A ---- is an effective data structure for implementing Dictionaries.
    a) **Hash table** b) Graph c) Tree d) Array

PART-B (3 X2 = 6 Marks)
(Answer ALL the Questions)

21. Discuss about the Basic terminology of Data structures.

The Basic terminologies of Data Organization are,

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.

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.

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.

22. Define Array. Explain about Single Dimensional Array .

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};

Single -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];

23. Write about the Limitations of Array Representation of Stack.

Under the array implementation, a fixed set of nodes represented by an array is established at the start of execution. A pointer to a node is represented by the relative position of the node within the array. The disadvantage of that approach is twofold. First, the number of nodes that are needed often cannot be predicted when a program is written. Usually, the data with which the program is executed determines the number of nodes necessary. Thus no matter how many elements the array of nodes contains, it is always possible that the program will be executed with input that requires a larger number.
The second disadvantage of the array approach is that whatever number of nodes are declared must remain allocated to the program throughout its execution.
The solution to this problem is to allow nodes that are dynamic, rather than static. That is, when a node is needed, storage is reserved for it, and when it is no longer needed, the storage is released. Thus the storage for nodes that are no longer in use is available for another purpose. Also, no predefined limit on the number of nodes is established. As long as sufficient storage is available to the job as a whole, part of that storage can be reserved for use as a node.

We have seen that we can use arrays whenever we have to store and manipulate collections of elements.

- the dimension of an array is determined the moment the array is created, and cannot be changed later on.
- the array occupies an amount of memory that is proportional to its size, independently of the number of elements that are actually of interest.
- if we want to keep the elements of the collection ordered, and insert a new value in its correct position, or remove it, then, for each such operation we may need to move many elements (on the average, half of the elements of the array);this is very inefficient.

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

24. a. Define Data structure. What are various Data structures?

Data Structures:

- 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.

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.

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:

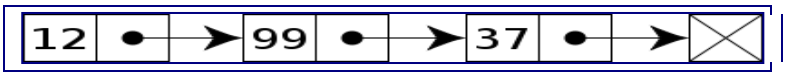
C++ allows arrays with more than two dimensions. The general form of an N-dimensional array declaration is: type array_name [size_1] [size_2] ... [size_N];

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.

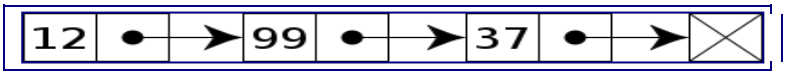
Linked List: A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure 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

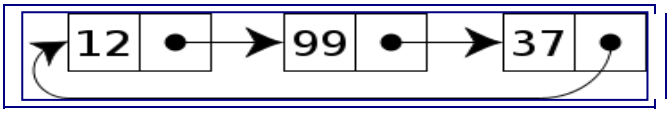
Doubly linked list

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

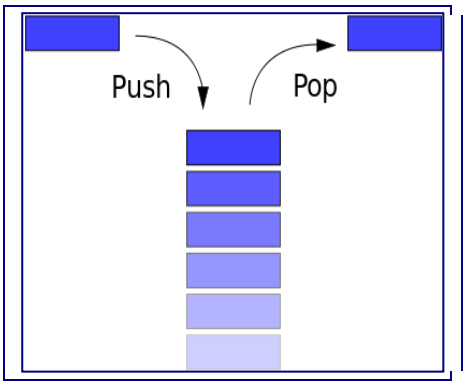
In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

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).



Stack

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.

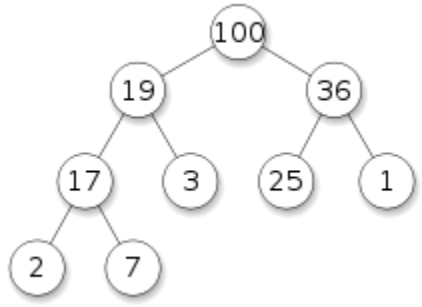
Trees:

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.

Heaps:

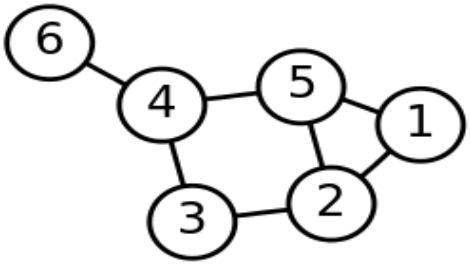
A heap is a specialized tree-based data structure that satisfies the *heap property:* If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. Either the keys of parent nodes are always greater than or equal to those 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

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**.**

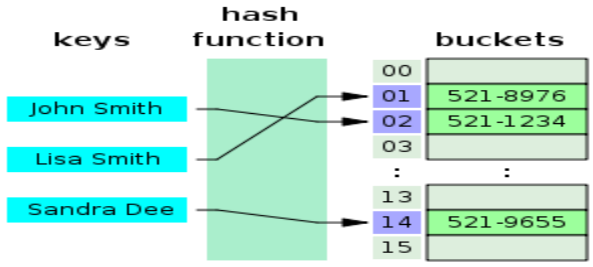


A labeled graph of 6 vertices and 7 edges.

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.

[OR]

b. Define Sparse matrix and how it is Represented in Array and Linked list?

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

Sparse matrices using array and linked representation:

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.

Why to use Sparse Matrix instead of simple matrix ?

Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

Sparse Matrix Representations can be done in many ways following are two common representations:

Array representation

Linked list representation

Method 1: Using Arrays

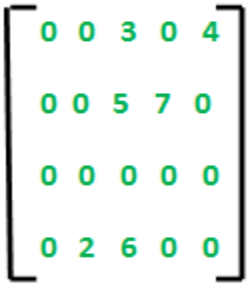
2D array is used to represent a sparse matrix in which there are three rows named as

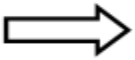
Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row,column)

Sparse Matrix Array Representation





Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

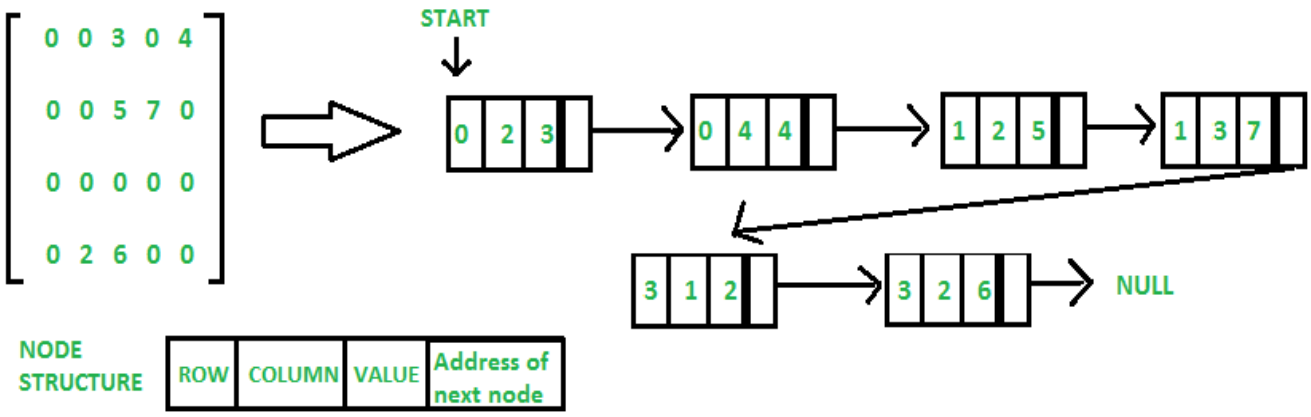
Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row,column)

Next node: Address of the next node

Using Arrays



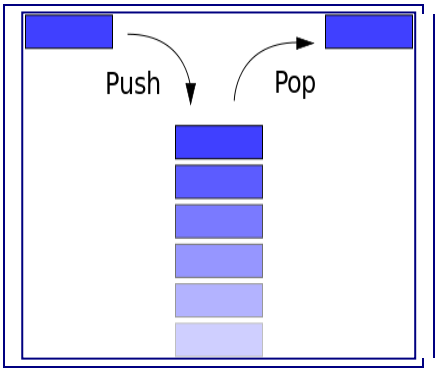
25. a. Explain in detail about Stack.

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).



Stack

Stack Using Array:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all the functions used in stack implementation.

Step 3: Create a one dimensional array with fixed size (int stack[SIZE])

Step 4: Define a integer variable 'top' and initialize with '-1'. (int top = -1)

Step 5: In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

Step 1: Check whether stack is FULL. (top == SIZE-1)

Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step 1: Check whether stack is EMPTY. (top == -1)

Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

Step 1: Check whether stack is EMPTY. (top == -1)

Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).

Step 3: Repeat above step until i value becomes '0'.

[Or]

b. Discuss about the Applications of stack.

APPLICATION OF STACK:

Expression evaluation and syntax parsing:

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations and conversion from one form to another may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most programming languages are context-free languages, allowing them to be parsed with stack based machines.

Backtracking:

Another important application of stacks is backtracking. Consider a simple example of finding the correct path in a maze. There are a series of points, from the starting point to the destination. We start from one point. To reach the final destination, there are several paths. Suppose we choose a random path. After following a certain path, we realize that the path we have chosen is wrong. So we need to find a way by which we can return to the beginning of that path. This can be done with the use of stacks. With the help of stacks, we remember the point where we have reached. This is done by pushing that

point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking.

The prototypical example of a backtracking algorithm is depth-first search, which finds all vertices of a graph that can be reached from a specified starting vertex. Other applications of backtracking involve searching through spaces that represent potential solutions to an optimization problem. Branch and bound is a technique for performing such backtracking searches without exhaustively searching all of the potential solutions in such a space.

Runtime memory management:

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine.

Almost all calling conventions—the ways in which subroutines receive their parameters and return results—use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. The functions follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C programming language is typically implemented in this way. Using the same stack for both data and procedure calls has important security implications (see below) of which a programmer must be aware in order to avoid introducing serious security bugs into a program.

Efficient algorithms:

Several algorithms use a stack (separate from the usual function call stack of most programming languages) as the principledata structure with which they organize their information. These include:

- Graham scan, an algorithm for the convex hull of a two-dimensional system of points. A convex hull of a subset of the input is maintained in a stack, which is used to find and remove concavities in the boundary when a new point is added to the hull.

- Part of the SMAWK algorithm for finding the row minima of a monotone matrix uses stacks in a similar way to Graham scan.
- All nearest smaller values, the problem of finding, for each number in an array, the closest preceding number that is smaller than it. One algorithm for this problem uses a stack to maintain a collection of candidates for the nearest smaller value. For each position in the array, the stack is popped until a smaller value is found on its top, and then the value in the new position is pushed onto the stack.
- The nearest-neighbor chain algorithm, a method for agglomerative hierarchical clustering based on maintaining a stack of clusters, each of which is the nearest neighbor of its predecessor on the stack. When this method finds a pair of clusters that are mutual nearest neighbors, they are popped and merged.

26. a. Elaborate about Prefix, Infix, and Postfix Expression with example.

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.
In above definition, operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

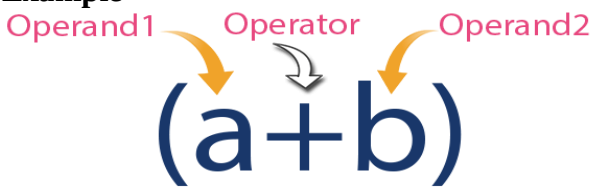
Infix Expression
Postfix Expression
Prefix Expression
Infix Expression:

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



Infix Expression

Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example

Postfix Expression

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Example

Prefix Expression

Any expression can be represented using the above three different types of expressions. And we can convert an expression from one form to another form like Infix to Postfix, Infix to Prefix, Prefix to Postfix and vice versa.

Utility and conversion of these expressions from one to another:
Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

Find all the operators in the given Infix Expression.
Find the order of operators evaluated according to their Operator precedence.
Convert each operator into required type of expression (Postfix or Prefix) in the same order.
Example

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

Step 1: The Operators in the given Infix Expression : = , + , *
Step 2: The Order of Operators according to their preference : * , + , =
Step 3: Now, convert the first operator * ----- D = A + B C *
Step 4: Convert the next operator + ----- D = A BC* +
Step 5: Convert the next operator = ----- D ABC*+ =
Finally, given **Infix Expression is converted into Postfix Expression** as follows...

D A B C * + =

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

Read all the symbols one by one from left to right in the given Infix Expression.
If the reading symbol is operand, then directly print it to the result (Output).
If the reading symbol is left parenthesis '(', then Push it on to the Stack.
If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.
Example

Consider the following Infix Expression...

(A + B) * (C - D)

The final Postfix Expression is as follows...

A B + C D - *

[OR]

b. Discuss about Representation of Stack in List.

Stack using Linked List

What is linked list ?

"A linked list is a recursive data structure that is either empty(null) or a reference to a node having a generic item and reference to a linked list."

Node class can be written as

```
    private class Node
1   {
2           Item item;
3           Node next;
4   }
5
```

We can represent a Linked List with a variable of type Node simply by ensuring that its value is either null or a reference to a Node whose next field is a reference to a linked list.

Node first = new Node();

Node second = new Node();

Node third = new Node();

now we set the item field in each of the nodes to the desired value.

first.item = "I";

second.item = "will";

thrid.item = "succeed";

and set the next fields to build the linked list.

first.next = second;

second.next = third;

Now third.next will be null which is the value it was initialized when created.

As a result third is a linked list since it is reference to a node that had a reference to null, which is the null reference to an empty linked list and second is a linked list since it is a reference to a node that has a reference to third which is a linked list, and first is a linked list which is reference to a node that has a reference to second which is a linked list.

Operations on Linked List

Insert at the beginning:

To insert the string "You" at the beginning of a linked list whose node is first, we should save first in oldfirst, assign to first a new Node and assign its item field to "You" and its next field to oldfirst.

Since the code for inserting a node at the beginning of a linked list involves few assignment statements, the amount of time that its takes is independent of the length of the list.

```
Node oldfirst = first;
first = new Node();

first.item = "You";
first.next = oldfirst;
```

Remove from the beginning this much more simpler: assign to first the value of first.next;

first = first.next

Insert at the End:

To do this operation we need a link to the last node in the list, because the node's link has to be changed to reference a new node containing the item to be inserted.

```
1  Node oldlast = last;
2  last = new Node();
3  last.item = "You will, Don't Worry"
4  oldlast.next = last;
```

To insert/remove at other positions doubly linked list is more useful that single linked list. In double linked list each node has two links, one in each direction.

For Traversal

To traverse through the list we use code like in following loop

```
1  for(Node x = first; x!= null; x = x.next)
2  {
3   //Process x.item
4  }
```

Register no_____

[16CSU302]

KARPAGAM ACADEMY OF HIGHER EDUCATION
KARPAGAM UNIVERSITY
Coimbatore – 641 021
B.Sc Computer Science
FIRST INTERNAL EXAMINATION – JULY 2017
Third Semester
OPERATING SYSTEMS

Date & Session : .07.2017 &
Maximum: 50 marks **Duration: 2 hours**

SECTION - A (20 X 1= 20 Marks)
ANSWER ALL THE QUESTIONS

1. The term " Operating System " means _____.
   a) A set of programs which controls computer working
   b) The way a computer operator works
   c) Conversion of high-level language in to machine level language
   d) The way a floppy disk drive operates

2. …… is a example of an operating system that support single user process and single thread.
   a) UNIX **b) MS-DOS** c) OS/2 d) Windows 2000

3. File management function of the operating system includes
   a) File creation and deletion b) Disk scheduling c) Process scheduling d) Multiprogramming

4. The operating system of a computer serves as a software interface between the user and the _____.
   a) Hardware b) Peripheral c) Memory d) Screen

5. What is a shell ?
   a) It is a hardware component
   b) It is a command interpreter
   c) It is a part in compiler
   d) It is a tool in CPU scheduling

6. A ..... architecture assigns only a few essential functions to the kernel, including address spaces, Inter process communication(IPC) and basic scheduling.
   a) Monolithic kernel **b) Micro kernel** c) Macro kernel d) Mini kernel

7.……is a lightweight process where the context switching is low
   a) Process **b)Thread** c)Kernel d) Minikernel

8. Process is .....
   a) A program in execution b) kernel c) thread d) deadlock

9. Which of the following are the states of a process model?

a) Delete b) Run c) New **d) Both ii and iii**

10. ..... refers to the ability of an operating system to support multiple threads of execution with a single process.

a) Multithreading b) Multiprocessing c) Multiexecuting d) Bi-threading

11. …… are very effective because a mode switch is not required to switch from one thread to another.

a) Kernel-level threads **b) User-level threads** c) Alterable threads d) Application level threads

12. ..... is a large kernel, including scheduling file system, networking, device drivers, memory management and more.

a) Monolithic kernel b) Micro kernel c) Macro kernel d) Mini kernel

13. To access the services of operating system, the interface is provided by the:

a) system calls b) API c) library d) assembly instructions

14. The main function of the command interpreter is:

a) to get and execute the next user-specified command

b) to provide the interface between the API and application program

c) to handle the files in operating system

d) none of the mentioned

15. The systems which allows only one process execution at a time, are called:

a) uniprogramming systems b) uniprocessing systems c). unitasking systems d) none of the mentioned

16. In Unix, Which system call creates the new process?

a) fork b) create c) new d) none of the mentioned

17. What is the ready state of a process?

a) when process is scheduled to run after some execution

b) when process is unable to run until some task has been completed

c) when process is using the CPU

d) none of the mentioned

18. The primary job of an OS is to

a) command resource **b) manage resource** c) provide utilities d) Be user friendly

19. As OS that has strict time constraints

a) Sensor Node OS **b) Real Time** OS c) Mainframe OS d) Timesharing OS

20. The OS that groups similar jobs is called as

a) Network OS b) Distributed OS c) Mainframe OS **d) Batch OS**

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

21. What is Operating System?

An operating system is a program that manages the computer hardware. An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

22. Briefly describe Real Time System?

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period – the correctness depends not only on the logical result but also the time it was delivered – failure to respond is as bad as the wrong response!

23. What is a Process?

A process generally consists of:

• The program's instructions (aka. the "program text")

• CPU state for the process (program counter, registers, flags, …)

• Memory state for the process

• Other resources being used by the process

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

24. a) Write about Basic OS Functions.

Basic OS Functions

Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management

- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Following are some of the important activities that an Operating System performs −

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

[OR]

24. b) Explain Batch Systems in detail.

Batch Processing Systems

To avoid the problems of early systems the batch processing systems were introduced. The problem of early systems was more setup time. So the problem of more set up time was reduced by processing the jobs in batches, known as *batch processing system.*In this approach similar jobs were submitted to the CPU for processing and were run together.

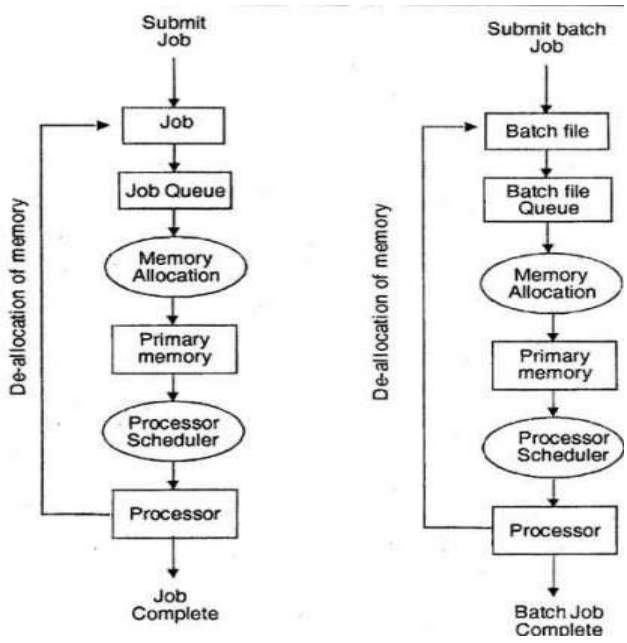
The main function of a batch processing system is to automatically keep executing the jobs in a batch. This is the important task of a batch processing system i.e. performed by the 'Batch Monitor' resided in the low end of main memory.

This technique was possible due to the invention of hard-disk drives and card readers. Now the jobs could be stored on the disk to create the pool of jobs for its execution as a batch. First the pooled jobs are read and executed by the batch monitor, and then these jobs are grouped; placing the identical jobs (jobs with the similar needs) in the same batch, So, in the batch processing system, the batched jobs were executed automatically one after another saving its time by performing the activities (like loading of compiler) only for once. It resulted in improved system utilization due to reduced turn around time.

In the early job processing systems, the jobs were placed in a job queue and the memory allocate or managed the primary memory space, when space was available in the main memory, a job was selected from the job queue and was loaded into memory.

Once the job loaded into primary memory, it competes for the processor. When the processor became available, the processor scheduler selects job that was loaded in the memory and execute it.

In batch strategy is implemented to provide a batch file processing. So in this approach files of the similar batch are processed to speed up the task.



Traditional Job Processing **Batch File Processing**

In batch processing the user were supposed to prepare a program as a deck of punched cards. The header cards in the deck were the "job control" cards which would indicate that which compiler was to be used (like FORTRAN, COBOL compilers etc). The deck of cards would be handed in to an operator who would collect such jobs from various users. Then the submitted jobs were 'grouped as FORTRAN jobs, COBOL jobs etc.

In addition, these jobs were classified as 'long jobs' that required more processing time or short jobs which required a short processing time. Each set of jobs was considered as a batch and the processing would be done for a batch. For instance, there maybe a batch of short FORTRAN jobs. The output for each job would be separated and turned over to users in a collection area. So in this approach, files of the similar batch were processed to speed up the task.

In this environment there was no interactivity and the users had no direct control. In this system, only one job could engage the processor at a time and if there was any input/ output

operation the processor had to sit idle till the completion of I/O job. So it resulted to the underutilization of CPU time.

In batch processing system, earlier; the jobs were scheduled in the order of their arrival i.e. First Come First Served (FCFS).Even though this scheduling method was easy and simple to implement but unfair for the situations where long jobs are queued ahead of the short jobs. To overcome this problem, another scheduling method named as 'Shortest Job First' was used. As memory management is concerned, the main memory was partitioned into two fixed partitions. The lower end of this partition was assigned to the resident portion of the OS i.e. named as Batch Monitor. Whereas, the other partition (higher end) was assigned to the user programs.

Though, it was an improved technique in reducing the system setup time but still there were some limitations with this technique like as under-utilization of CPU time, non-interactivity of user with the running jobs etc. In batch processing system, the jobs of a batch were executed one after another. But while these jobs were performing I/O operations; meantime the CPU was sitting idle resulting to low degree of resource utilization.

25. a) Explain the Types of Operating Systems.

Types of Operating Systems

Mainframe Operating Systems

At the high end are the operating systems for mainframes, those room-sized computers still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and millions of gigabytes of data is not unusual; a personal computer with these specifications would be the envy of its friends.

Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions. The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O.

They typically offer three kinds of services: batch, transaction processing, and timesharing.

Batch systems

A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode.

Transaction-processing systems

Transaction-processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second.

Timesharing systems

Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related; mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360. However, mainframe operating systems are gradually being replaced by UNIX variants such as Linux.

Server Operating Systems

One level down are the server operating systems. They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Websites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are Solaris, FreeBSD, Linux and Windows Server 201x.

Multiprocessor Operating Systems

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication, connectivity, and consistency.

With the recent advent of multicore chips for personal computers, even conventional desktop and notebook operating systems are starting to deal with at least small-scale multiprocessors and the number of cores is likely to grow over time. Luckily, quite a bit is known about multiprocessor operating systems from years of previous research, so using this knowledge in multicore systems should not be hard. The hard part will be having applications make use of all this computing power. Many popular operating systems, including Windows and Linux, run on multiprocessors.

Personal Computer Operating Systems

The next category is the personal computer operating system. Modern ones all support multiprogramming, often with dozens of programs started up at boot time. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets, games, and Internet access. Common examples are Linux, FreeBSD, Windows 7, Windows 8, and

Apple's OS X. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

Handheld Computer Operating Systems

Continuing on down to smaller and smaller systems, we come to tablets, smartphones and other handheld computers. A handheld computer, originally known as a **PDA** (**Personal Digital Assistant**), is a small computer that can be held in your hand during operation. Smartphones and tablets are the best-known examples. As we have already seen, this market is currently dominated by Google's Android and Apple's iOS, but they hav e many competitors. Most of these devices boast multicore CPUs, GPS, cameras and other sensors, copious amounts of memory, and sophisticated operating systems. Moreover, all of them have more third-party applications (**''apps''**) than you can shake a (USB) stick at.

Embedded Operating Systems

Embedded systems run on the computers that control devices that are not generally thought of as computers and which do not accept user-installed software. Typical examples are microwave ovens, TV sets, cars, DVD recorders, traditional phones, and MP3 players. The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it. You cannot download new applications to your microwave oven—all the software is in ROM. This means that there is no need for protection between applications, leading to design simplification. Systems such as Embedded Linux, QNX and VxWorks are popular in this domain.

Sensor-Node Operating Systems

Networks of tiny sensor nodes are being deployed for numerous purposes. These nodes are tiny computers that communicate with each other and with a base station using wireless communication. Sensor networks are used to protect the perimeters of buildings, guard national borders, detect fires in forests, measure temperature and precipitation for weather forecasting, glean information about enemy movements on battlefields, and much more.

The sensors are small battery-powered computers with built-in radios. They have limited power and must work for long periods of time unattended outdoors, frequently in environmentally harsh conditions. The network must be robust enough to tolerate failures of individual nodes, which happen with ever-increasing frequency as the batteries begin to run down.

Each sensor node is a real computer, with a CPU, RAM, ROM, and one or more environmental sensors. It runs a small, but real operating system, usually one that is event driven, responding to external events or making measurements periodically based on an internal clock. The operating system has to be small and simple because the nodes have little RAM and battery lifetime is a major issue. Also, as with embedded systems, all the programs are loaded in advance; users do not suddenly start programs they downloaded from the Internet, which makes the design much simpler. TinyOS is a well-known operating system for a sensor node.

Real-Time Operating Systems

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process- control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time. If, for example, a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system**. Many of these are found in industrial process control, avionics, military, and similar application areas. These systems must provide absolute guarantees that a certain action will occur by a certain time.

A **soft real-time system**, is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Digital audio or multimedia systems fall in this category. Smartphones are also soft realtime systems.

Since meeting deadlines is crucial in (hard) real-time systems, sometimes the operating system is simply a library linked in with the application programs, with ev erything tightly coupled and no protection between parts of the system.

An example of this type of real-time system is eCos. The categories of handhelds, embedded systems, and real-time systems overlap considerably. Nearly all of them have at least some soft real-time aspects. The embedded and real-time systems run only software put in by the system designers; users cannot add their own software, which makes protection easier. The handhelds and embedded systems are intended for consumers, whereas real-time systems are more for industrial usage. Nevertheless, they hav e a certain amount in common.

Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU chip. They hav e very severe processing power and memory constraints. Some are powered by contacts in the reader into which they are inserted, but contactless smart cards are inductively powered, which greatly limits what they can do. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions. Often these are proprietary systems.

Some smart cards are Java oriented. This means that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

[OR]

25. b) What are Multiprogramming Systems. Explain in detail.

Multiprogramming Systems

To overcome the problem of underutilization of CPU and main memory, the multiprogramming was introduced. The multiprogramming is interleaved execution of multiple jobs by the same computer.

In multiprogramming system, when one program is waiting for I/O transfer; there is another program ready to utilize the CPU. So it is possible for several jobs to share the time of the CPU. But it is important to note that multiprogramming is not defined to be the execution of jobs at the same instance of time. Rather it does mean that there are a number of jobs available to the CPU (placed in main memory) and a portion of one is executed then a segment of another and so on.

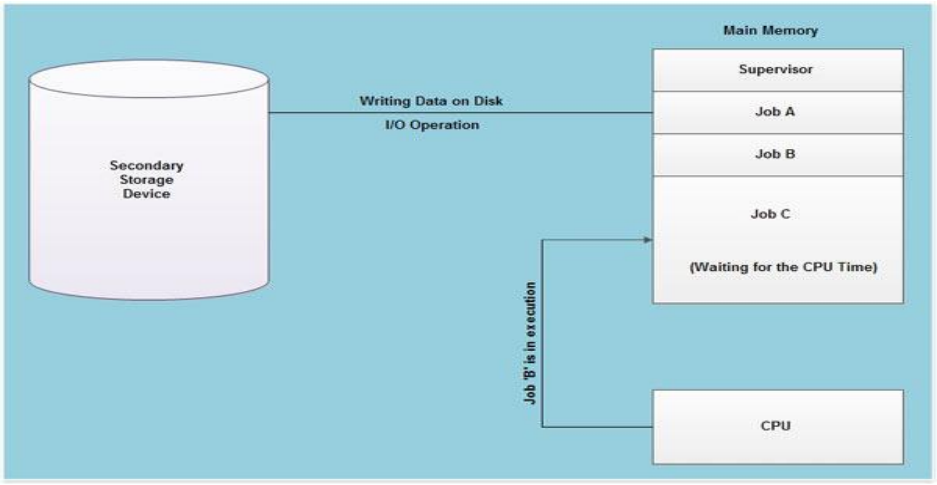


Figure 1.3. A simple process of multiprogramming.

as shown in fig, at the particular situation, job' A' is not utilizing the CPU time because it is busy in I/ 0 operations. Hence the CPU becomes busy to execute the job 'B'. Another job C is waiting for the CPU for getting its execution time. So in this state the CPU will never be idle and utilizes maximum of its time.

A program in execution is called a "Process", "Job" or a "Task". The concurrent execution of programs improves the utilization of system resources and enhances the system throughput as compared to batch and serial processing. In this system, when a process requests some I/O to allocate; meanwhile the CPU time is assigned to another ready process. So, here when a process is switched to an I/O operation, the CPU is not set idle.

Multiprogramming is a common approach to resource management. The essential components of a single-user operating system include a command processor, an input/ output control system, a

file system, and a transient area. A multiprogramming operating system builds on this base, subdividing the transient area to hold several independent programs and adding resource management routines to the operating system's basic functions.

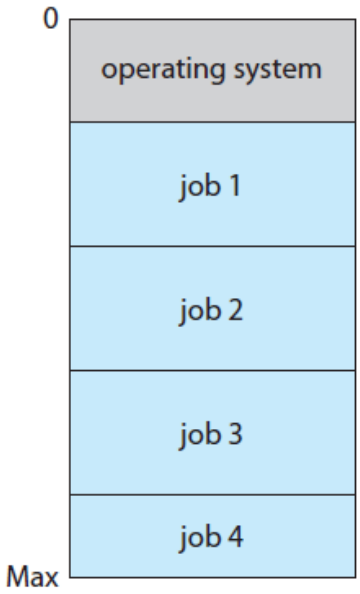


Figure 1.4 Memory layout for a multiprogramming system.

Multiprogramming increases CPUutilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job.

When ***that*** job needs to wait, the CPU switches to ***another*** job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle. This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

Time sharing (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device.

Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user. Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**.

Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text. In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk. A more common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory.

This arrangement frees programmers from concern over memory-storage limitations.

A time-sharing system must also provide a file system. The file system resides on a collection of disks; hence, disk management must be provided. In addition, a time-sharing system provides a mechanism for protecting resources from inappropriate use.

To ensure orderly execution, the system must provide mechanisms for job synchronization and communication, and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another.

26. a) Explain System Calls and System Programs **[OR]**

System Calls

The system call provides an interface to the operating system services.

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

- Portability: as long a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier then using the actual system call.

System Call Parameters

Three general methods exist for passing parameters to the OS:

1. Parameters can be passed in registers.
2. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register.
3. Parameters can also be pushed on or popped off the stack by the operating system.

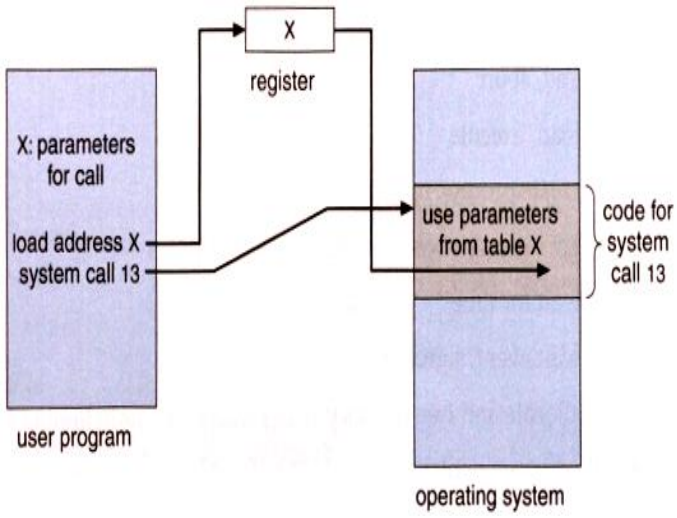


Fig 2.2 System call parameters

Types of System Calls

There are 5 different categories of system calls:

process control, file manipulation, device manipulation, information maintenance and communication.

Process Control

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

File Management

Some common system calls are *create*, *delete*, *read*, *write*, *reposition*, or *close*. Also, there is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls.

Device Management

Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

User programs *request* the device, and when finished they *release* the device. Similar to files, we can *read*, *write*, and *reposition* the device.

Information Management

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*.

The OS also keeps information about all its processes and provides system calls to report this information.

Communication

There are two models of interprocess communication, the message-passing model and the shared memory model.

- Message-passing uses a common mailbox to pass messages between processes.
- Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.

System Call

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process. It is because of the critical nature of operations that the operating system itself does them every time they are needed. For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Programs

System programs, also known as **system utilities**, provide a convenient environment for program development and execution.

Some of them are simply user interfaces to system calls.

These programs are not usually part of the OS kernel, but are part of the overall operating system.

They can be divided into these categories:

• **File management**. These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

• **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are

more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.

• **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

• **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.

• **Program loading and execution**. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

• **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

• **Background services**. All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or daemons.

One example is the network daemon, a system needed a service to listen for network connections in order to connect those requests to the correct processes.

Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons.

In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities. Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations.

Such **application programs** include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the Mac OS X operating system, the user might see the GUI, featuring a

mouse-and-windows interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways.

26. b) Explain Process in detail and Process hierarchies.

The Process

A process generally consists of:

• The program's instructions (aka. the "program text")

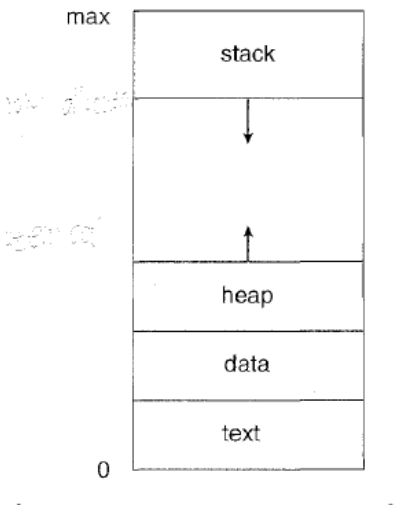
• CPU state for the process (program counter, registers, flags, …)

• Memory state for the process

• Other resources being used by the process

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 2.1.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)



Process in memory

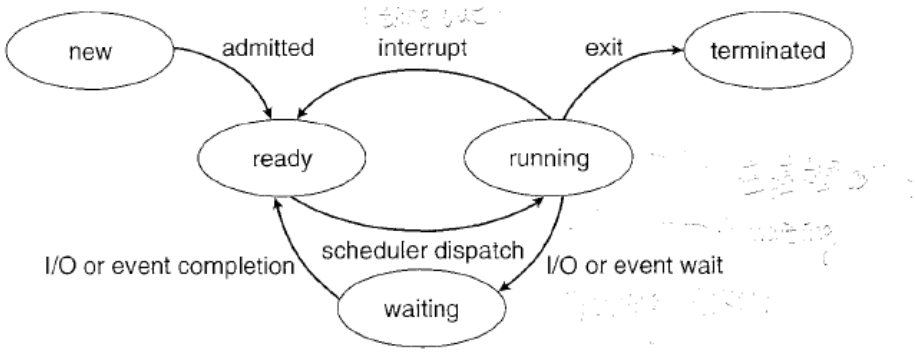


Fig 2.3 Diagram of process state

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

Process Hierarchies

Modern general purpose operating systems permit a user to create and destroy processes.

- In unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.

- After a fork both parent and child keep running (indeed they have the *same* program text) and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a wait() system call it would block until G finished.

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is *automatically* blocked and waits until the second is finished.

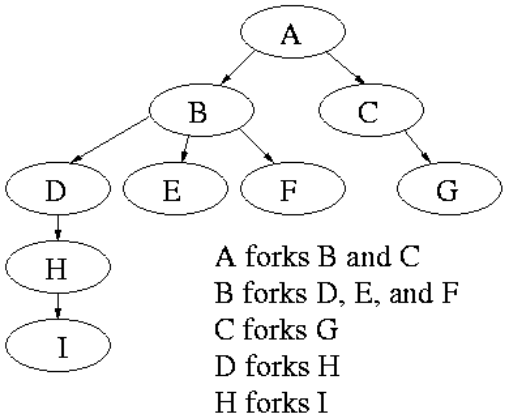


Fig 2.5 Process Hierarchy

Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy.

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually ail active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a role, let us look at how UNIX initializes itself when it is started. A special process, called *init*, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off one new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root.

In contrast, Windows has no concept of a process hierarchy. All processes are equal. The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a **handle)** that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.