Semester – III

18CTU301 **DATA STRUCTURES** **4H – 4C**

Instruction Hours / week: L: 4 T: 0 P: 0 **Marks:** Int : **40** Ext : **60** Total: **100**

End Semester Exam : 3 Hours

Course Objectives

- To teach efficient storage mechanisms of data for an easy access.
- To design and implementation of various basic and advanced data structures.
- To introduce various techniques for representation of the data in the real world.
- To develop application using data structures.
- To teach the concept of protection and management of data.
- To improve the logical ability

Course Outcomes (COs)

Upon completion of this course,

1. Student will be able to choose appropriate data structure as applied to specified problem definition.
2. Student will be able to handle operations like searching, insertion, deletion, traversing mechanism etc. on various data structures.
3. Students will be able to apply concepts learned in various domains like DBMS, compiler construction etc.
4. Students will be able to use linear and non-linear data structures like stacks, queues , linked list etc.

Unit I

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

Unit II

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

Unit III

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

Unit IV

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

Unit V

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

Suggested Readings

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

ESE Patterns	
Part – A(Online)	20x1=20
Part – B	5x2=10
Part – C(Either or)	5x6=30
Total	60 Marks

Faculty **HOD**

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

DEPARTMENT OF COMPUTER SCIENCE, COMPUTER APPLICATION & INFORMATION TECHNOLOGY

SUBJECT : DATA STRUCTURES
SEMESTER : III
SUBJECT CODE: 18CTU301 CLASS : II B.Sc. CT

LECTURE PLAN

S.No.	Lecture Duration (Period)	Topics to be Covered	Support Materials
		Unit – I	
1	1	Introduction to Data Structures Introduction to arrays & it types	W1, S13: 22-28 S13: 39-51
2	1	Arrays • Single Dimensional • Multi Dimensional	S1: 24- 34, 37-40 S13: 105-109 S13: 116-126
3	1	Sparse Matrices • Array Representation Sparse Matrices - Linked List Representation	S14: 103-118 S14: 103-118
4	1	Stacks Implementing Operations Stacks Implementing- Single stack Stacks Implementing- Multiple stack in Array	S1:77-84 ,W5 S13: 136-142 ,W5
5	1	Stacks Implementing- Prefix expressions Stack Implementing- Infix expressions	S1: 95-110 S13: 136-142
6		Stack Implementing- Postfix expressions	S13: 136-142
7	1	Utility and conversion of expressions representation of stack	S14: 356-367
8	1	Applications of stack, Limitations of Array representation of stack	S14: 356-367
9	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-I	**09**

Unit – II			
1	1	Introduction to Linked Lists • Singly Linked List • Structure and its operation with example	S13: 143-146 S14: 52-65
2	1	Doubly Linked List • Structure with algorithm • Example	S1: 237-244 S14: 67-91
3	1	Circular Lists • Structure with algorithm • Example	S13: 167-197 S14: 71-80
4	1	Representation of Stack in Lists • Normal Representation • Circular Representation	S1: 203-207 S1: 229-230
5	1	Self Organizing Lists • Move to Front Method • Count Method • Transpose Method	S13: 227-229, 239 S14: 77 W4
6	1	Skip Lists Queues • Implementation • Index able skip list	W4, W5
7	1	Queue Implementing • Array representation • Linked representation	S1: 174-184, 186-190 S13: 371-376, 381-386
8	1	Queue Implementing • De-queue with example • Priority Queues with example	S13: 386-395
9	1	Recapitulation and Discussion of important questions	
Total No. of Hours Planned for Unit-II			**09**
Unit - III			
1	1	Trees • Introduction to Tree as a data structure Introduction to Binary Tree	S1: 249-254 S13: 411-414
2	1	Binary Trees • Insertion • Deletion	S14: 229-238 S1: 296-299
3	1	Binary Trees - Searching	S13: 426-442
4	1	• Recursive Traversals • Iterative Traversals	S14: 238-249 W2
5.	1	Threaded Binary Trees • Insertion • Deletion	S1: 272-277

6.	1	Threaded Binary Trees - Traversals	S13: 544-550
7.	1	Height-Balanced or AVL Trees • Structure • Operations on AVL Trees Insertion into AVL Trees	S1: 413 – 421 S13: 490 - 496
8.	1	operations on AVL Trees- Deletion from AVL Trees	S13: 496-508
9.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-III	**09**
		Unit - IV	
1	1	Searching and Sorting • Introduction	S1: 384-387
2	1	• Types of searching and sorting	S14: 340-342
3	1	Linear Search • Algorithm Linear Search Implementation in C	S1: 337-394 S14: 344
4	1	Binary Search • Algorithm	S1: 394-398
5	1	Selection Sort • Algorithm	S1: 351-353
6	1	Insertion Sort • Algorithm	S1: 365-366
7	1	Shell Sort • Algorithm	S1: 366-372
8	1	Comparison of Sorting Techniques	S13: 278-279
9	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-IV	**09**
		Unit - V	
1	1	Hashing	S1: 468-474
2	1	Introduction to operation on hashing Inserting and deleting from hash table	S13: 560 S1: 476-485
3	1	Rehasing: Efficiency of Rehash Methods	
4	1	Hash Table Reordering Resolving collusion by Open Addressing	S1: 468-474 S1: 470-473
5	1	Linear Probing Clustering	S1: 470-473 S1: 470-473

6	1	Coalesced Hashing Separate Chaining	S1: 485-487 S1: 493-494
7	1	Dynamic and Extendible Hashing	S1: 494-499
8	1	Choosing a Hash Function Perfect Hashing Function	S1: 505-508 S1: 508-512
9	1	Recapitulation and Discussion of important questions	
10	1	Pervious ESE Question Paper Discussion	
11	1	Pervious ESE Question Paper Discussion	
12	1	Pervious ESE Question Paper Discussion	
		Total No. of Hours Planned for Unit-V	**12**

Total No. of periods: 48

Suggested Readings

S1: Aaron M. Tenenbaum, Moshe J. Augenstein, YedidyahLangsam, "Data Structures Using C and C++, Second edition, PHI, 2009.

S13: Dharmender Singh Kushwaha, Arun Kumar Misra, “Data Structures, A Programming Approach with C”, PHI

S14: ISRD Group, Data Structure using C, Tata Mc Hill

Website

W1: https://en.wikipedia.org/wiki/Data_structure

W2: http://basicdatastructures.blogspot.in

W3: https://en.wikipedia.org/wiki/Self-organizing_list

W4: www.geeksforgeeks.org/implement-two-stack

W5: https://en.m.wikipedia.org/wiki/Skip-list

Faculty **HOD**

UNIT-I

SYLLABUS

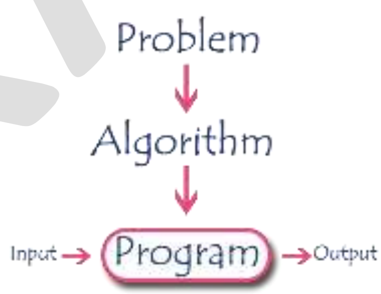
Arrays-Single and Multi-dimensional Arrays, Sparse Matrices (Array and Linked Representation).Stacks Implementing single / multiple stack/s in an Array; Prefix, Infix and Postfix expressions, Utility and conversion of these expressions from one to another; Applications of stack; Limitations of Array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

1. OVERVIEW OF DATA STRUCTURES:

An algorithm is a step by step procedure to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by computer to produce solution. Here, program takes required data as input, processes data according to the program instructions and finally produces result as shown in the following picture.



Performance of an algorithm is a process of making evaluative judgement about algorithms.

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm.

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

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.**

- Every data structure is used to organize the large amount of data
- Every data structure follows a particular principle
- The operations in a data structure should not violate the basic principle of that data structure.

Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy.

The study of data structures includes:

* Logical description of data structures.
* Implementation of data structures.
* Quantitative analysis of the data structures.

Based on the organizing method of a data structure, data structures are divided into two types.

- Linear Data Structures
- Non - Linear Data Structures

Linear Data Structures

If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure.

Example

- Arrays
- List (Linked List)
- Stack
- Queue

Non - Linear Data Structures

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure.

Example

- Tree
- Graph
- Dictionaries
- Heaps

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.

Concept of a Data Type:

Data Object

Data Object represents an object having a data.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

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

Example :

Integer, Floating-point, **Character (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.

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.

2. 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:

Example:

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

Whenever we want to work with large number of data values, we need to use that much number of different variables. As the number of variables is increasing, complexity of the program also increases and programmers get confused with the variable names. There may be situations in which we need to work with large number of similar data values. To make this work more easy, C programming language provides a concept called "Array".

An array is a variable which can store multiple values of same data type at a time.

An array can also be defined as follows...

"Collection of similar data items stored in continuous memory locations with single name".

To understand the concept of arrays, consider the following example declaration.

int a, b, c;

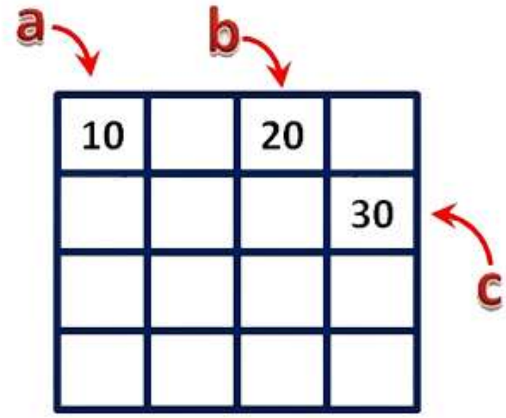
Here, the compiler allocates 2 bytes of memory with name 'a', another 2 bytes of memory with name 'b' and more 2 bytes with name 'c'. These three memory locations are may be in sequence or may not be in sequence. Here these individual variables can store only one value at a time.

In computer memory is organized as shown in figure. Here assume that each box is of 2 bytes of memory.

2 byte for 'a', another 2 bytes for 'b' and 2 more bytes for 'c'.

If we assign following values they will inserted into that memory locations.

a = 10;
b = 20;
c = 30;



One-Dimensional Arrays:

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

Syntax : 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];

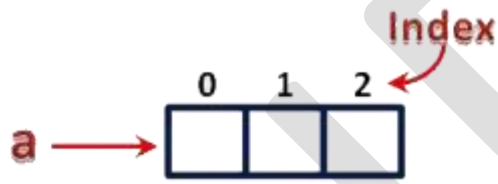
char last_name[40];

Now consider the following declaration...

int a[3];

Here, the compiler allocates total 6 bytes of continuous memory locations with single name 'a'. But allows to store three different integer values (each in 2 bytes of memory) at a time. And memory is organized as follows...

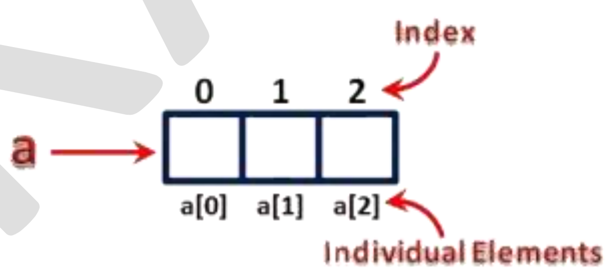
That means all these three memory locations are named as 'a'. But "how can we refer individual elements?" is the big question. Answer for this question is, compiler not only allocates memory, but also assigns a numerical value to each individual element of an array. This numerical value is called as "Index". Index values for the above example are as follows...



The individual elements of an array are identified using the combination of 'name' and 'index' as follows...

arrayName[indexValue]

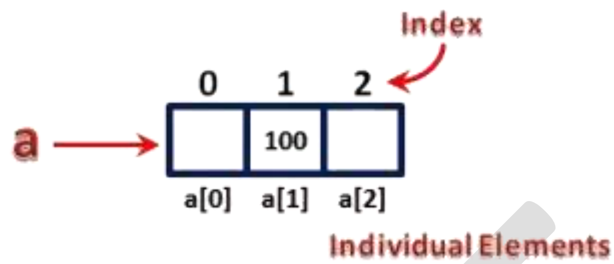
For the above example the individual elements can be referred to as follows...



If I want to assign a value to any of these memory locations (array elements), we can assign as follows...

a[1] = 100;

The result will be as follows...



Insertion Operation

Algorithm

Let **Array** be a linear unordered array of **MAX** elements.

Example

Result

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the K^{th} position of LA −

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the K^{th} position of LA.

```
1. Start
2. Set J = K
```

```
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

3. MULTIDIMENSIONAL ARRAYS:

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

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

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.

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];**

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:

Example : int matrix[3][3];

A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

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;
```

Three-Dimensional Array:

A three-dimensional array is that array whose elements are two-dimensional arrays. In practice, it may be considered to be an **array of matrices.** A three-dimensional array with *int* elements may be declared as below.

int A [m] [n] [p];

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

```
int x[2][3][2] =
  {
     { {0,1}, {2,3}, {4,5} },
     { {6,7}, {8,9}, {10,11} }
```

This requires 4 * 10 * 20 = 800 elements.

4. SPARSE MATRICES (ARRAY AND LINKED REPRESENTATION)

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

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**).

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Array Representation
2. Linked Representation

Array 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

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores total rows, total columns and total non-zero values in the matrix.

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

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

Linked Representation

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

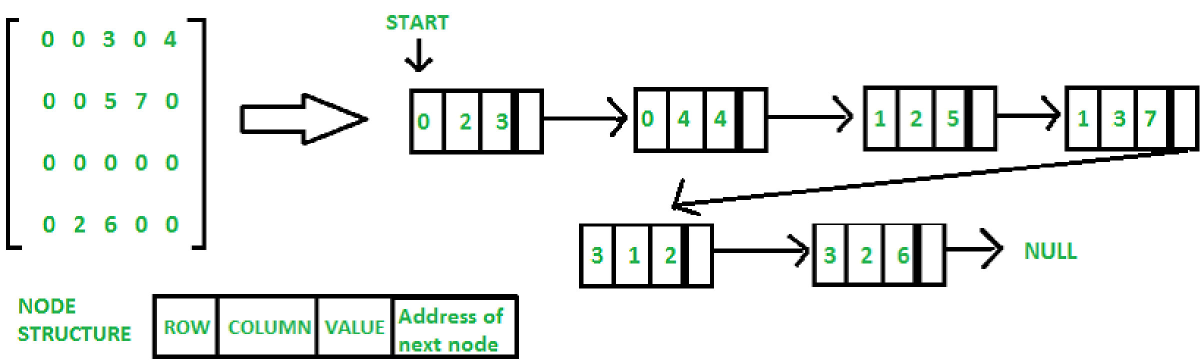
 Row: Index of row, where non-zero element is located

 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

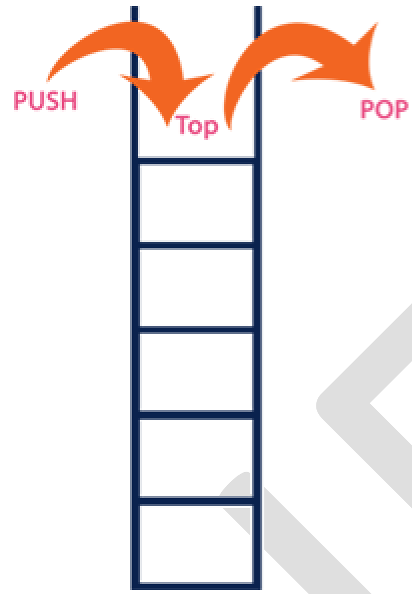


<u>5. STACKS IMPLEMENTING SINGLE / MULTIPLE STACK/S IN AN ARRAY</u>

<u>STACK</u>:

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". That means, new element is added at top of the stack and an element is removed from the top of the stack.

In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows...

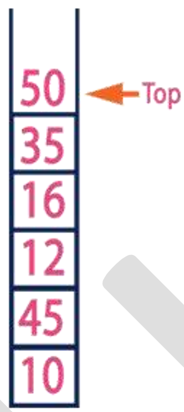
Stack is a linear data structure in which the operations are performed based on LIFO principle.

Stack can also be defined as

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below...



The following operations are performed on the stack...

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

Stack data structure can be implement in two ways. They are as follows...

1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Implementing single stack in an Array

Array representation of Stack

Stack can be represented by means of a one way list or a linear array. A pointer variable top contains the locations of the top element of the stack and a variable max stk gives the maximum number of elements of the Stack that can be held by Stack.

the condition top=0 will indicate that the stack is empty.

Push Operation on Stack

This procedure pushes an item onto the Stack via Top.

```
PUSH(Stack, Top, MaxStk, Item)

1. If Top == MaxStk  //check Stack already fill or not

   then print "Overflow" and return

2. Set Top = Top + 1  //increase top by 1

3. Stack[Top] = Item  //insert item in new top position

4. Return
```

Pop operation on Stack

This procedures deletes the top element of Stack and assigns it to the variable item.

```
POP(Stack, top, Item)

1. If Top == Null  //check Stack top element to be deleted is empty
```

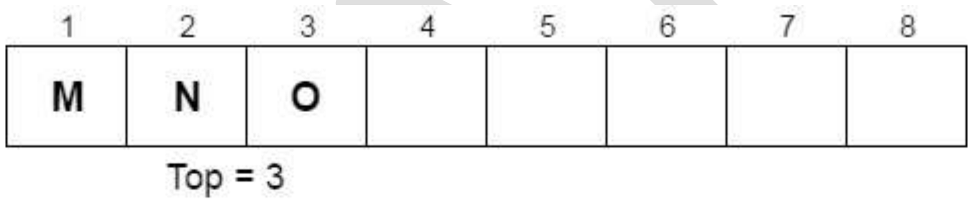
then print "Underflow" and return

2. Item = Stack[Top] //assign top element to item

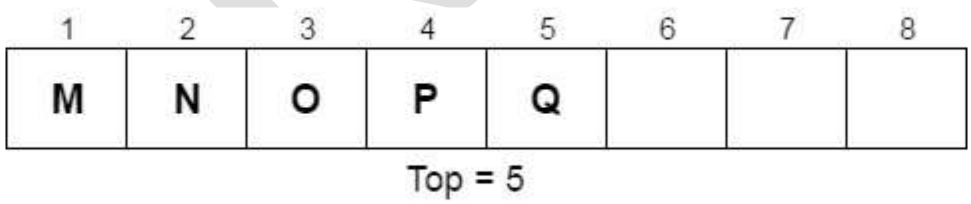
3. Set Top = Top - 1 //decrease top by 1

4. Return

Let MaxStk=8, the array Stack contains M, N, O in it. Perform operations on it



i. after insertion of P, Q



ii. pop 3 elements

1	2	3	4	5	6	7	8
M	N						

Top = 2

iii. push A, B, C

1	2	3	4	5	6	7	8
M	N	A	B	C			

Top = 5

iv. push D, E, F, G

1	2	3	4	5	6	7	8
M	N	A	B	C	D	E	F

Overflow Condition as Top=MaxStk Top = 8

v. push X

1	2	3	4	5	6	7	8
M	N	A	B	C	D	E	F

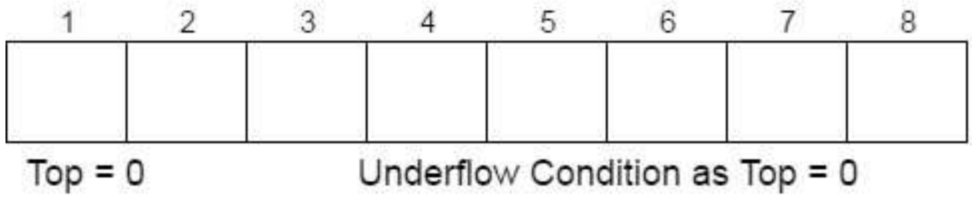
Overflow Condition as Top=MaxStk Top = 8

vi. pop 5 elements

1	2	3	4	5	6	7	8
M	N	A					

Top = 3

vii. pop 4 elements



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.

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'.

IMPLEMENT TWO STACKS IN AN ARRAY

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

push1(int x) –> pushes x to first stack

push2(int x) –> pushes x to second stack

pop1() –> pops an element from first stack and return the popped element

pop2() –> pops an element from second stack and return the popped element

Method 1 (Divide the space in two halves)

A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e., use arr[0] to arr[n/2] for stack1, and arr[n/2+1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to second stack2. When we push 4th element to stack1, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0.

The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

Create a data structure kStacks that represents k stacks. Implementation of kStacks should use only one array, i.e., k stacks should use the same array for storing elements. Following functions must be supported by kStacks.

push(int x, int sn) –> pushes x to stack number 'sn' where sn is from 0 to k-1

pop(int sn) –> pops an element from stack number 'sn' where sn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)

A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)

The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer* arrays as extra space.

Following are the two extra arrays are used:

1) top[]: This is of size k and stores indexes of top elements in all stacks.
2) next[]: This is of size n and stores indexes of next item for the items in array arr[].

Here arr[] is actual array that stores k stacks.

Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

6. PREFIX, INFIX AND POSTFIX EXPRESSIONS

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.

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

1. Infix Expression
2. Postfix Expression
3. 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

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

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

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.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

7. 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.

Prefix to Infix Conversion

Infix : An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

Example : (A+B) * (C-D)

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : *+AB-CD (Infix : (A+B) * (C-D))

Given a Prefix expression, convert it into a Infix expression. Computers usually does the computation in either prefix or postfix (usually postfix). But for humans, its easier to understand an Infix expression rather than a prefix. Hence conversion is need for human understanding.

Examples:

```
Input :  Prefix :  *+AB-CD
Output : Infix : ((A+B)*(C-D))

Input :  Prefix :  *-A/BC-/AKL
Output : Infix : ((A-(B/C))*((A/K)-L))
```

Algorithm for Prefix to Infix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack Create a string by concatenating the two operands and the operator between them.

string = (operand1 + operator + operand2)
And push the resultant string back to Stack

- Repeat the above steps until end of Prefix expression.

Prefix to Postfix Conversion

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : *+AB-CD (Infix : (A+B) * (C-D))

Postfix: An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator).

Example : AB+CD-* (Infix : (A+B * (C-D))

Given a Prefix expression, convert it into a Postfix expression. Conversion of Prefix expression directly to Postfix without going through the process of converting them first to Infix and then to Postfix is much better in terms of computation and better understanding the expression (Computers evaluate using Postfix expression).

Examples:

Input : Prefix : *+AB-CD

Output : Postfix : AB+CD-*

Explanation : Prefix to Infix : (A+B) * (C-D)

Infix to Postfix : AB+CD-*

Input : Prefix : *-A/BC-/AKL

Output : Postfix : ABC/-AK/L-*

Explanation : Prefix to Infix : A-(B/C)*(A/K)-L

Infix to Postfix : ABC/-AK/L-*

Algorithm for Prefix to Postfix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator after them.
  string = operand1 + operand2 + operator
  And push the resultant string back to Stack

- Repeat the above steps until end of Prefix expression.

Postfix to Prefix Conversion

Postfix: An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator).

Example : AB+CD-* (Infix : (A+B * (C-D))

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : *+AB-CD (Infix : (A+B) * (C-D))

Given a Postfix expression, convert it into a Prefix expression. Conversion of Postfix expression directly to Prefix without going through the process of converting them first to Infix and then to Prefix is much better in terms of computation and better understanding the expression (Computers evaluate using Postfix expression).

Examples:

Input : Postfix : AB+CD-*

Output : Prefix : *+AB-CD

Explanation : Postfix to Infix : (A+B) * (C-D)

Infix to Prefix : *+AB-CD

Input : Postfix : ABC/-AK/L-*

Output : Prefix : *-A/BC-/AKL

Explanation : Postfix to Infix : A-(B/C)*(A/K)-L

Infix to Prefix : *-A/BC-/AKL

Algorithm for Prefix to Postfix:

- Read the Postfix expression from left to right
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack Create a string by concatenating the two operands and the operator before them. **string = operator + operand2 + operand1** And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

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

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. 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 * + =

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

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. 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.

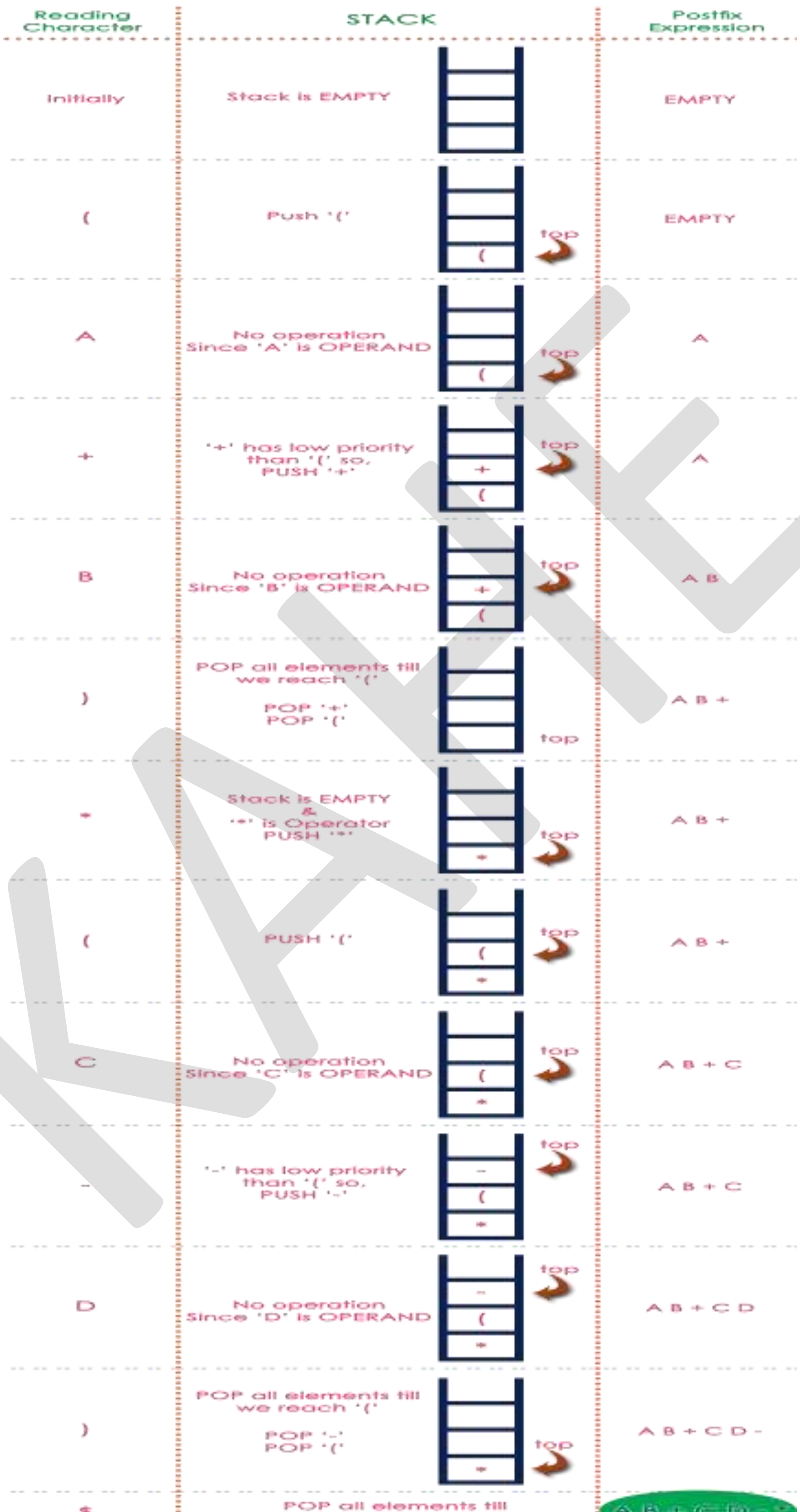
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression...

(A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as follows...


Reading Character
STACK
Postfix Expression
Initially
Stack is EMPTY
EMPTY
(
Push '('
top
(
EMPTY
A
No operation Since 'A' is OPERAND
A
+
'+' has low priority than '(' so, PUSH '+'
+
A
B
No operation Since 'B' is OPERAND
A B
)
POP all elements till we reach '('
POP '+'
POP '('
A B +
*
Stack is EMPTY & '*' is Operator PUSH '*'
*
A B +
(
PUSH '('
A B +
C
No operation Since 'C' is OPERAND
A B + C
-
'-' has low priority than '(' so, PUSH '-'
-
A B + C
D
No operation Since 'D' is OPERAND
A B + C D
)
POP all elements till we reach '('
POP '-'
POP '('
A B + C D -
$
POP all elements till
A B + C D - *

8. APPLICATIONS OF STACK

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

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.

Backtracking (game playing, finding paths, exhaustive searching)

Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal.

- Find your way through a maze.
- Find a path from one point in a graph (roadmap) to another point.
- Play a game in which there are moves to be made (checkers, chess).

In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative

Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.

Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

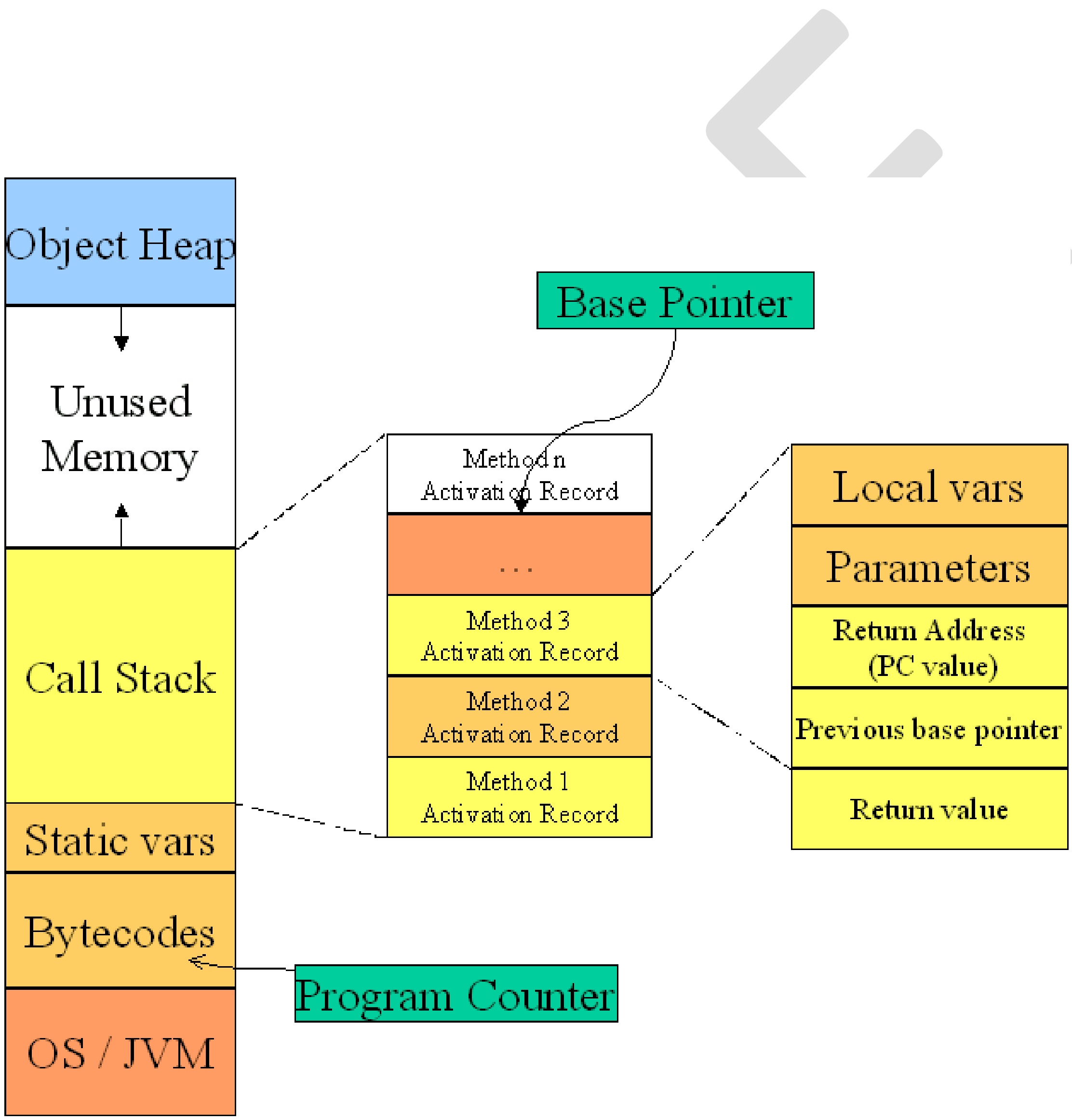
Memory Management

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.

Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.

The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix runtime environments.

Each program that is running in a computer system has its own memory allocation containing the typical layout as shown below.



Limitations of Array representation of stack

1. We must know in advance that how many elements are to be stored in array.
2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.
3. Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.
4. The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

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.

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.

Dynamic nodes use notion of pointers intensively. Pointers allow us to build and manipulate linked lists of various types. The concept of a pointer introduces the possibility of assembling a collection of building blocks, called nodes, into flexible structures. By altering the values of pointers, nodes can be attached, detached, and reassembled in patterns that grow and shrink as execution of a program progresses.

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. Explain about Stacks Implementation using single / multiple stack/s in an Array
5. Elaborate about Prefix, Infix and Postfix Expressions with example.
6. Explain about Conversion of Expressions.
7. Write about Limitations of Array representation of stack.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore - 641021.

UNIT I :(Objective Type/Multiple choice Questions each Question carries one Mark)

Subject :Data Structures Sub Code:18CTU301

PART-A (Online Examination)

Sno	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4			**ANSWER**
1	________ is a sequence of instructions to accomplish a particular task	Data Strucuture	Algorithm	Ordered List	Queue			**Algorithm**
2	_______ criteria of an algorithm ensures that the algorithm terminate after a particular number of steps.	effectiveness	finiteness	definiteness	particular			**finiteness**
3	An algorithm must produce __________ output(s)	many	only one	atleast one	zero or more			**atleast one**
4	_______ criteria of an algorithm ensures that the algorithm must be feasible.	effectiveness	finiteness	definiteness	infinite			**effectiveness**
5	_______ criteria of an algorithm ensures that each step of the algorithm must be clear and unambiguous.	effectiveness	finiteness	definiteness	infinite			**definiteness**
6	The logical or mathematical model of a particular data organization is called as_____________	Data Structure	Software Engineering	Data Mining	Data Ware Housing			**Data Structure**
7	An algorithms _____________ is measured in terms of computing time ad space consumed by it.	performance	effectivenes s	finiteness	definiteness			**performance**
8	Which of the following is not structured data type?	Arrays	Union.	Queue	Linked list.			**Union**.
9	What is the strategy of Stack?	LILO	FIFO	FILO	LIFO			**LIFO**

10	What is the strategy of Queue?	LILO	FIFO	FILO	LIFO			**FIFO**
11	Data structures are classified as _____________ data type.	User Defined	Abstract	Primitive & Non	predefined only			**Primitive & Non**
12	_______ are the commonl used ordered list.	Graphs	Trees	Stack and Queues	List			**Stack and Queues**
13	Data structure can be classified as _________ data type based on relationship with complex data element.	Linear & Non Linear	Linear	Non Linear	None of the above			**Linear & Non Linear**
14	A data structure whose elements forms a sequence of ordered list is called as ___________ data structure.	Non Linear	Linear.	Primitive	Non Primitive			**Linear**.
15	A data structure which represents hierarchical relationship between the elements are called as ____________ data	Linear	Primitive.	Non Linear	Non Primitive			**Non Linear**
16	A data structure, which is not composed of other data structure, is called as ____________ data structure.	Linear	Non Primitive	Non Linear	Primitive			**Primitive**
17	Data structures, which are constructed from one or more primitive data structure, are called as ____________ data	Non Primitive	Primitive.	Non Linear	Linear			**Non Primitive**
18	___________ is the term that refers ti the kinds of data that variables may hold in a programming language.	data type	data structure	data Object	data			**data type**
19	_________ refers to the set of elements that belong to a particular type.	data type	data structure	data Object	data			**data Object**
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	data type	data structure	data Object	data			**data structure**
21	__________estimation is the method of analysing an algorithm before it is executed.	Preprocess	Verification	Priori	Posteriori			**Priori**
22	In queue we can add elements at _________.	Top	Bottom	Front	Rear			**Rear**
23	In queue we can delete elements at _________.	Front	Bottom	Top	Rear			**Front**

24	In Stack we can add elements at _______.	Bottom	Top	Front	Rear			**Top**
25	In Stack we can delete elements at ______	Front	Rear	Top	Bottom			**Top**
26	When Top = Bottom in stack, the total no of element in the stack is	1	2	3	0			**0**
27	When FRONT = REAR in queue, the total no of element in the queue is	0	1	2	3			**0**
28	In Stack the TOP is decremeted by one after every ___ operation.	AddQ	Pop	Push	DelQ			**Pop**
29	In Stack the TOP is incremeted by one before every ___ operation.	AddQ	Pop	Push	DelQ			**Push**
30	To add an item into the queue,	FRONT is incremented	FRONT is decremented	REAR is decremented	REAR is incremeted by			**REAR is incremeted**
31	In Queue FRONTis incremented then, the operation performed on it is ________.	DelQ	Pop	Push	AddQ			**DelQ**
32	When the maximum entries of (m*n) matrix are zeros then it is called as _______.	Transpose matrix	Sparse Matrix	Inverse Matrix	tridiagonal matrix			**Sparse Matrix**
33	A matrix of the form (row, col, n) is otherwise known as _______.	Transpose matrix	Inverse Matrix	Sparse Matrix	Diagonal matrix			**Sparse Matrix**
34	Which of the following is a valid linear data structure.	Stacks	Records	Trees	Graphs			**Stacks**
35	Which of the following is a valid non - linear data structure.	Stacks	Trees	Queues	Linked list.			**Trees**
36	A list of finite number of homogeneous data elements are called as _________	Stacks	Records	Arrays	Linked list.			**Arrays**
37	No of elements in an array is called the _________ of an array.	Structure	Height	Width	Length.			**Length**.

38	_________ is the art of creating sample data upon which to run the program	Testing	Designing	Analysis	Debugging			**Testing**
39	If a program fail to respond corectly then _________ is needed to determine what is wrong and how to correct it.	Testing	Designing	Analysis	Debugging			**Testing**
40	A _________ is a linear list in which elements can be inserted and deleted at both ends but not at the Middle	Queue	DeQueue	Enqueue	Priority Queue			**DeQueue**
41	A _________ is a collection of elements such that each element has been assigned a priority.	Priority Queue	De Queue	Circular Queue	En Queue			**Priority Queue**
42	A _________ is made up of Operators and Operands.	Stack	Expression	Linked list	Queue			**Expression**
43	A _____________ is a procedure or function which calls itself.	Stack	Recursion	Queue	Tree			**Recursion**
44	An example for application of stack is __________.	Time sharing computer	Waiting Audience	Processing of	space sharing system			**Processing of subroutines**
45	An example for application of queue is __________.	Stack of coins	Stack of bills	Processing of	Job Scheduling in			**Job Scheduling in**
46	The size or length of an array = __________.	UB – LB + 1	LB + 1	UB - LB	UB – 1			**UB – LB + 1**
47	The _______________ model of a particular data organization is called as Data Structure.	software Engineering	logical or mathematica	Data Mining	Data Ware Housing			**logical or mathematical**
48	Combining elements of two _______ data structure into one is called Merging	Similar	Dissimilar	Even	Un Even			**Similar**
49	Searching is the Process of finding the ________ of the element with the given value or a record with the given	Place	Location	Value	Operand			**Location**
50	Length of an array is defined as ___________ of elements in it.	Structure	Height	Size	Number			**Number**
51	Associating an element of an ordered list A_i with an index i is called __________ mapping.	linear	sequential	ordered	indexed			**sequential**

52	________ is a set of pairs, index and value.	stack	queue	Arrays	Set			**Arrays**
53	The design approach where the main tast is decomposed into subtasks and each subtask is further decomposed into	top down approach	bottom up approach	hierarchical approach	merging approach			**top down approach**
54	Solving different parts of a program directly and combining these pieces into a complete program is called	top down approach	bottom up approach	hierarchical approach	merging approach			**bottom up approach**
55	_________ is divided into 2 major phases i.e., the priori and the posteriori estimates.	Performance Evaluation	Program Validation	Algorithm Analysis	Testing and Debugging			**Performance Evaluation**
56	___________ is the product of the time taken for single execution and the number of times a statement is executed	Piori estimate	Posteriori estimate	Frequency count	None of the above			**Piori estimate**
57	The number of times a statement in a program is executed is called its _________	Piori estimate	Posteriori estimate	Frequency count	None of the above			**Frequency count**
58	____________ means the computing time of the algorithm is constant	O(log n)	O(n)	O(n log n)	O(1)			**O(1)**
59	____________ means the computing time of the algorithm is linear	O(log n)	O(n)	O(n log n)	O(1)			**O(n)**
60	Algorithms with time complexity $O(n^2)$ are called ________	linear	exponential	quadratic	cubic			**quadratic**
61	For large data set algorithms with complexity greater than ________ are impractical	O(log n)	O(n)	O(n log n)	O(1)			**O(n log n)**
62	Sum of terms of the form ax^e is called _______.	Array	Matrix	Expression	Polynomial			**Polynomial**
63	_____________ is efficient solution to avoid data movement in array representation of Queue.	Circular Queue	Dequeue	Enqueue	queue			Circular Queue

UNIT-II

SYLLABUS

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:

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

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

– Successive elements are connected by pointers.

– Last element points to NULL head

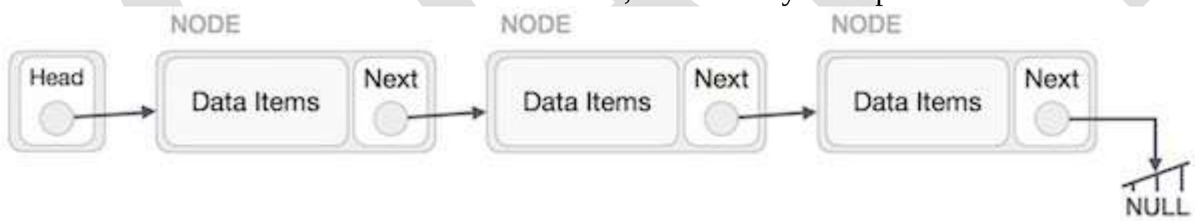
– It can grow or shrink in size during execution of a program.

– It can be made just as long as required.

– It does not waste memory space.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.

- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

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.

TYPES OF LINKED LIST

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

LINEAR SINGLY-LINKED LIST

The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

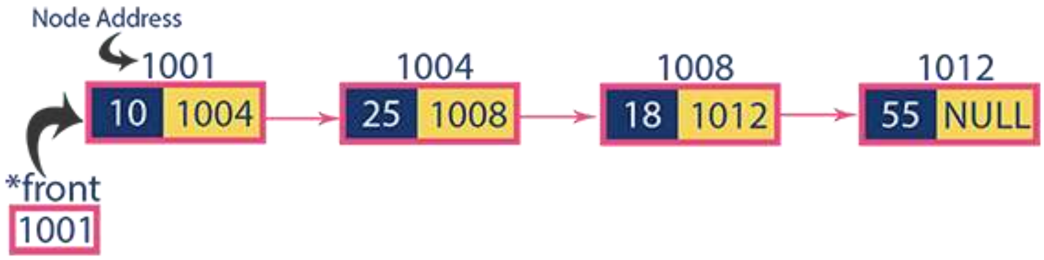
In any single linked list, the individual element is called as **"Node"**. Every **"Node"** contains two fields, **data** and **next**. The **data** field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows..

☀ In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").

☀ Always next part (reference part) of the last node must be NULL.

Example



Operations

In a single linked list we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

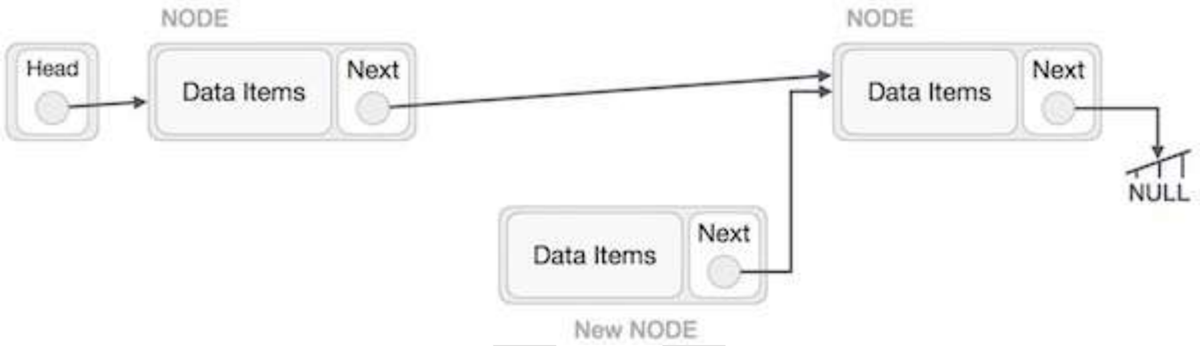
Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

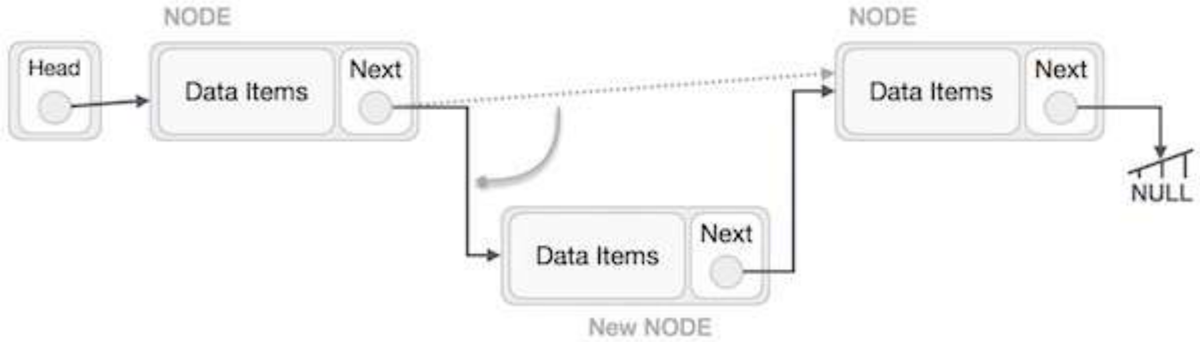
NewNode.next −> RightNode;

It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

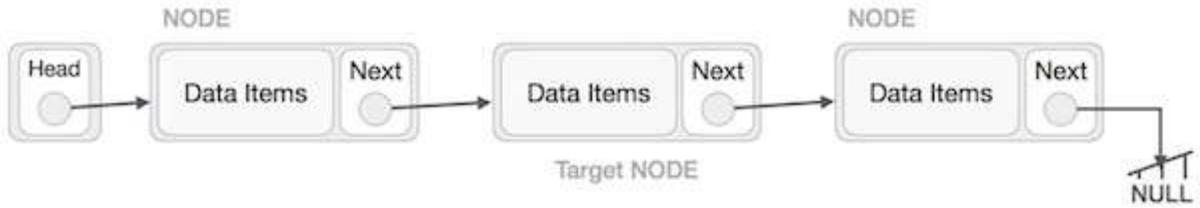


This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



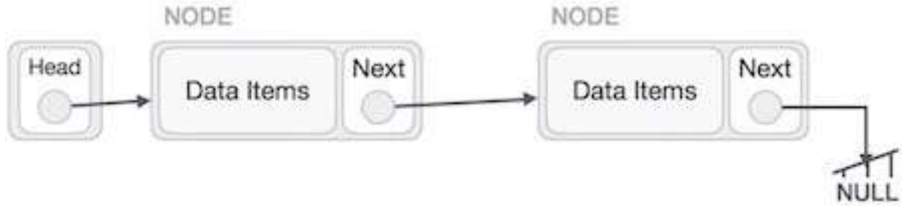
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next –> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

CIRCULAR LINKED LIST:

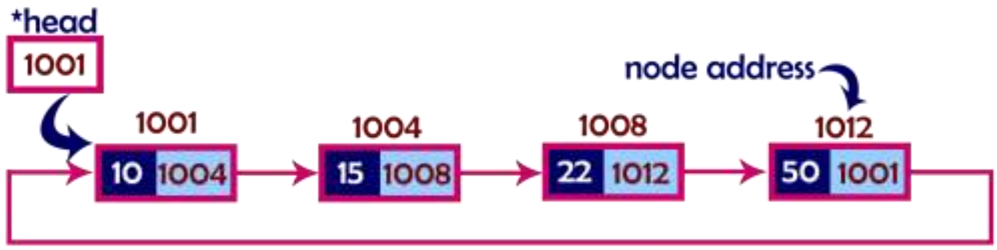
• The pointer from the last element in the list points back to the first element.

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

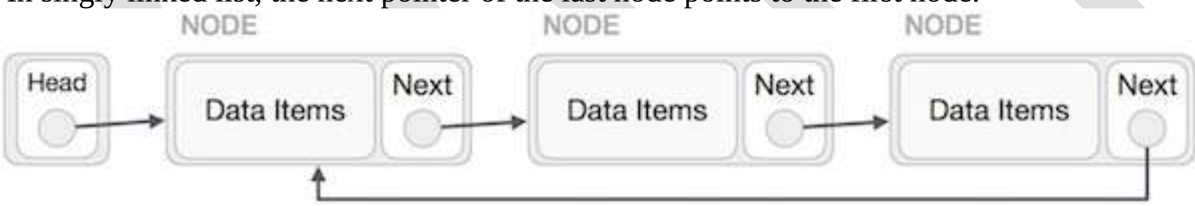
Example



Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

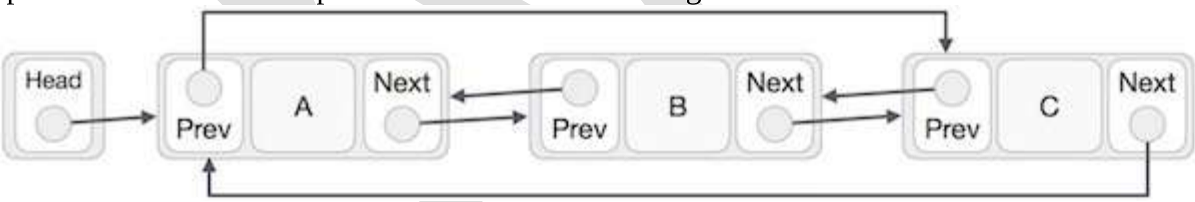
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** − Inserts an element at the start of the list.
- **delete** − Deletes an element from the start of the list.
- **display** − Displays the list.

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data= data;

   if (isEmpty()) {
      head = link;
      head->next = head;
   } else {
      //point it to old first node
      link->next = head;

      //point first to new first node
      head = link;
   }
}
```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
   //save reference to first link
   struct node *tempLink = head;

   if(head->next == head) {
      head = NULL;
      return tempLink;
   }

   //mark next to first link as first
   head = head->next;

   //return the deleted link
   return tempLink;
```

```
}
```

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

```
//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   if(head != NULL) {
      while(ptr->next != ptr) {
         printf("(%d,%d) ",ptr->key,ptr->data);
         ptr = ptr->next;
      }
   }

   printf(" ]");
}
```

Application of Circular Linked List

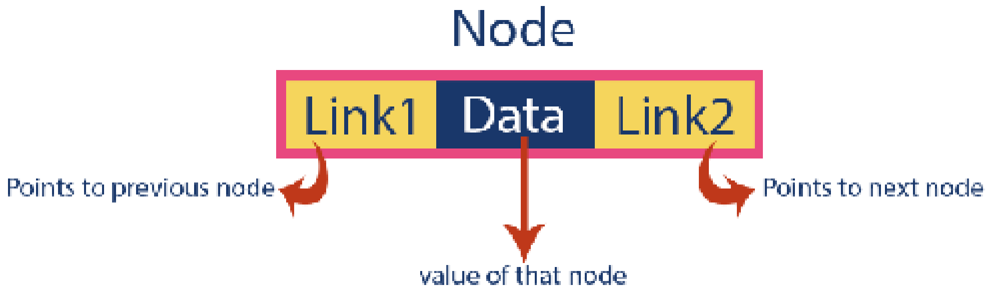
- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the 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.

DOUBLY LINKED LIST:

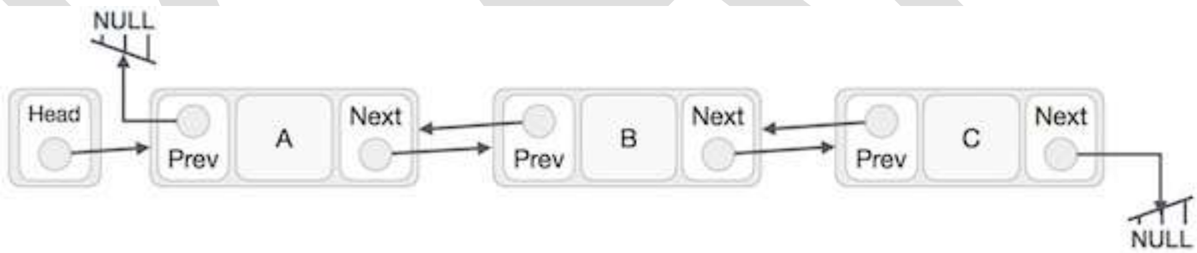
In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



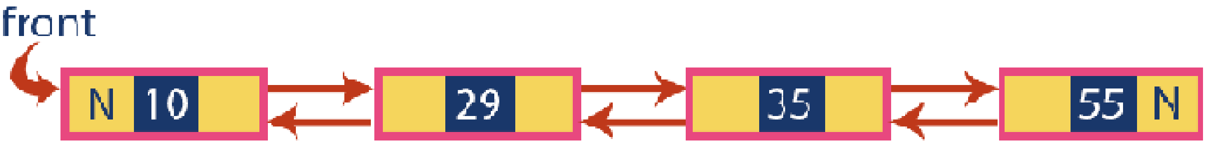
Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Example



- In double linked list, the first node must be always pointed by **head**.
- Always the previous field of the first node must be **NULL**. Always the next field of the last node must be **NULL**.
- 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.

<u>IMPLEMENTING LISTS USING ARRAYS</u>:

Arrays are suitable for:

– Inserting/deleting an element at the end.

– Randomly accessing any element.

– Searching the list for a particular value.

Array representation of linked list

An array of linked list is an unique structure which combines a static structure (an array) and a dynamic structure (linked lists) to form a useful data structure. This type of data structure is useful for applications. Like, When you know the categories under a menu but have no idea about the sub-categories under categories. For instance, we can use an array of linked lists, where each list contains words starting with a specific letter in the alphabet. When you declare all variable, then

```
node* A[n] ;    // defines an array of n node pointers
for (i=0; i<n; i++)  A[i] = NULL;  // initializes the array to NULL
```

Creating an Array of Linked Lists

Assume that a linked list needs to be created starting at A[i]. The first node would be created as follows.

```
A[i] = (node*)malloc(sizeof(node)); // allocate memory for  node
A[i] → size = 10;
A[i] → name = (char*) malloc(strlen("neha"+1));
strcpy(A[i] → name, "neha\0");
A[i] → next = NULL;
```

Now, to insert more nodes into the list.

```
int inserrnodes(node*** arrayhead, int index, node* ptr);
```

A call to the function can be as follows.

```
node* ptr = (node*)malloc(sizeof(node));
ptr → size = 10;
ptr → name = (char*) malloc(strlen("neha"+1));
strcpy(ptr → name, "neha\0");
ptr → next = NULL;
insertnodes(&A, ptr, 3);   // insert node ptr to array location 3.
```

Implementing Lists using Linked List:

Linked lists are suitable for:

Inserting an element.

Deleting an element.

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

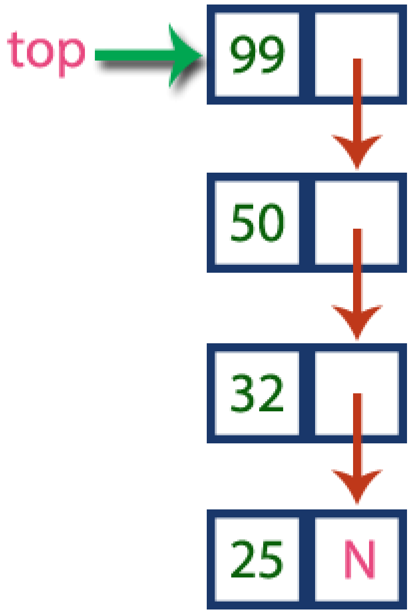
REPRESENTATION OF STACK IN LISTS

STACK USING LINKED LIST

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next** field of the first element must be always **NULL**.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Operations

To implement stack 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 a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

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

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3:** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4:** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5:** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

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

- **Step 1:** Check whether **stack** is **Empty** (**top == NULL**).
- **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 define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4:** Then set '**top = top → next**'.
- **Step 7:** Finally, delete '**temp**' (**free(temp)**).

display() - Displaying stack of elements

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

- **Step 1:** Check whether stack is **Empty** (**top** == **NULL**).
- **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 Node pointer **'temp'** and initialize with **top**.
- **Step 4:** Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp → next** != **NULL**).
- **Step 4:** Finally! Display '**temp → data** ---> **NULL**'.

Example Program

```
#include<iostream>
#include<process.h>

using namespace std;

struct Node
{
   int data;
   Node *next;
}*top=NULL,*p;

Node* newnode(int x)
```

```
{
   p=new Node;
   p->data=x;
   p->next=NULL;
   return(p);
}

void push(Node *q)
{
   if(top==NULL)
      top=q;
   else
   {
      q->next=top;
      top=q;
   }
}

void pop(){
   if(top==NULL){
      cout<<"Stack is empty!!";
   }
   else{
      cout<<"Deleted element is "<<top->data;
      p=top;
      top=top->next;
      delete(p);
   }
}

void showstack()
{
   Node *q;
   q=top;

   if(top==NULL){
      cout<<"Stack is empty!!";
   }
   else{
      while(q!=NULL)
      {
         cout<<q->data<<" ";
         q=q->next;
      }
   }
}

int main()
{
```

```
    int ch,x;
    Node *nptr;

    while(1)
    {
        cout<<"\n\n1.Push\n2.Pop\n3.Display\n4.Exit";
        cout<<"\nEnter your choice(1-4):";
        cin>>ch;

        switch(ch){
            case 1: cout<<"\nEnter data:";
                    cin>>x;
                    nptr=newnode(x);
                    push(nptr);
                    break;

            case 2: pop();
                    break;

            case 3: showstack();
                    break;

            case 4: exit(0);

            default: cout<<"\nWrong choice!!";
        }
    }

    return 0;
}
```

SELF ORGANIZING LIST

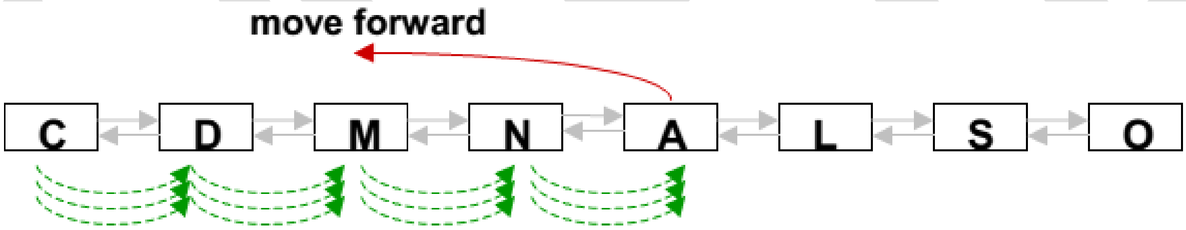
The worst case search time for a sorted linked list is O(n). With a Balanced Binary Search Tree, we can skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

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. A Self Organizing list reorders its nodes based on searches which are done. The idea is to use locality of reference (In a typical database, 80% of the access are to 20% of the items). Following are different strategies used by Self Organizing Lists.

1) ***Move-to-Front Method***: Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.
2) ***Count Method***: Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.
3) ***Transpose Method***: Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.

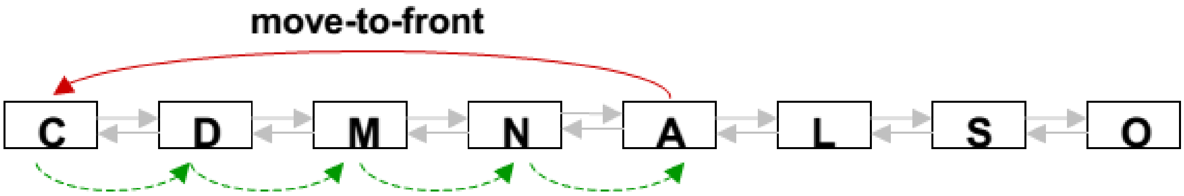
Self-Organizing Organizing – lists in which the order of elements changes based
Lists on searches which are done
• speed up the search by placing the frequently accessed
elements at or close to the head



Examples – important tel. numbers placed near the front of tel. directory

Basic Strategies in Self-Organizing Lists
(1) Move-to-Front Method
(2) Count Method
(3) Exchange Method
(1) Move-to-Front Method: any node (position) searched / requested is moved to the front
Pros:
• easily implemented & memoryless – requires no extra storage
• adapts quickly to changing access patterns
Cons:
• may over-reward infrequently accessed nodes

• relatively short memory of access pattern



(2) Count Method: each node (position) counts the number of times it was searched for – nodes are ordered by decreasing count

Pros:

• reflects the actual access pattern

Cons:

• must store and maintain a counter for each node

• does not adapt quickly to changing access pattern

(3) Transpose Method: any node searched is swapped with the preceding node

Pros:

• easily implemented & memoryless

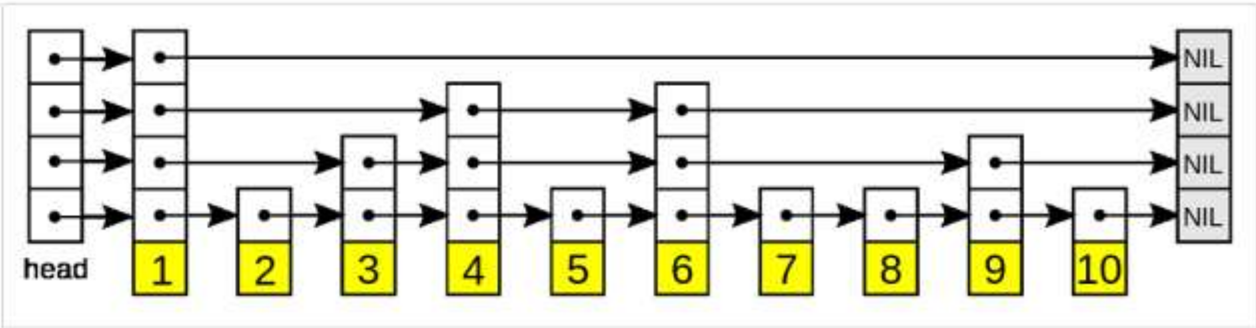
• likely to keep frequently accessed nodes near the front

Cons:

• more cautious than "Move-to-Front" (it will take many consecutive accesses to move one node to the front)

SKIP LISTS

A skip list is a data structure for storing a sorted list of items using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with efficiency comparable to balanced binary search trees (that is, with number of probes proportional to log n instead of n).



Each link of the sparser lists skips over many items of the full list in one step, hence the structure's name. These forward links may be added in a randomized way with a geometric / negative binomial distribution. Insert, search and delete operations are performed in logarithmic expected time. The links may also be added in a non-probabilistic way so as to guarantee amortized (rather than merely expected) logarithmic cost.

Complexity

	Average Case	**Worst Case**
Space	O(n)	O(nlogn)
Search	O(logn)	O(n)
Insert	O(logn)	O(n)
Delete	O(logn)	O(n)

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

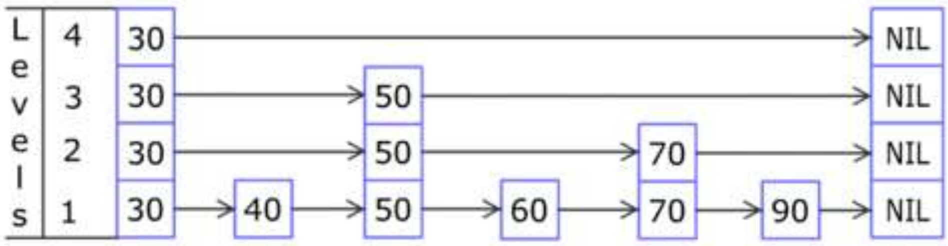
When an element is tried to search, the search begins at the head element of the top list. It proceeds horizontally until the current element is greater than or equal to the target. If current element and target are matched, it means they are equal and search gets finished.

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.



Insertion in Skip List

Applications of Skip List

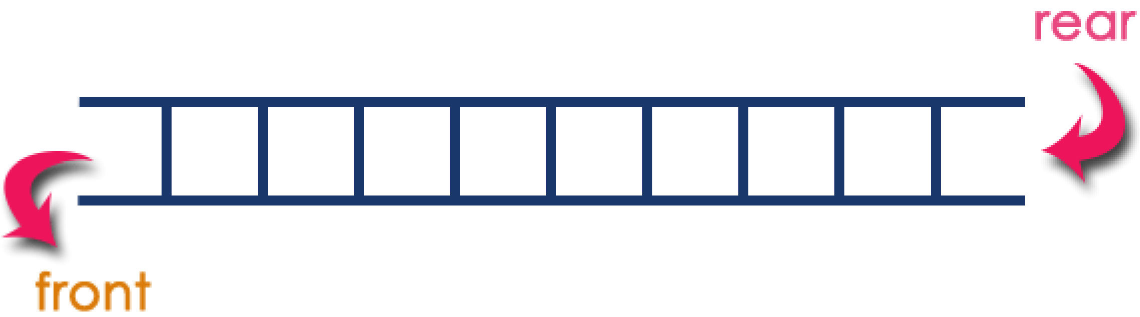
1. Skip list are used in distributed applications. In distributed systems, the nodes of skip list represents the computer systems and pointers represent network connection.

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).

<u>QUEUES</u>

What is a Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".

Queue data structure can be defined as follows...

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

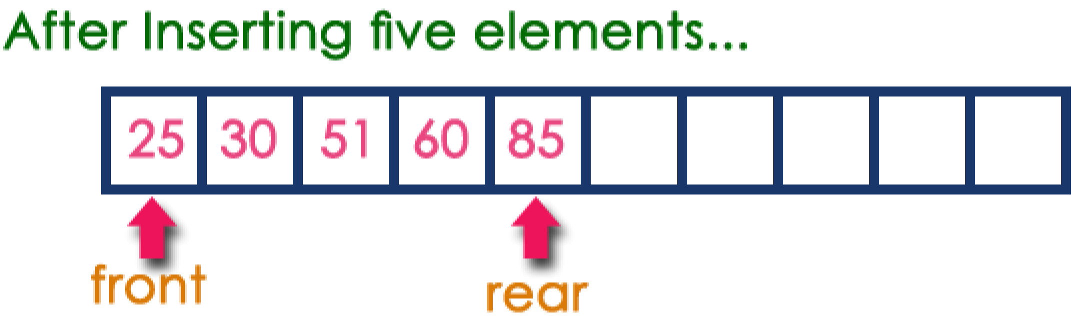
A queue can also be defined as

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Example

Queue after inserting 25, 30, 51, 60 and 85.

Operations on a



Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**

2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

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...

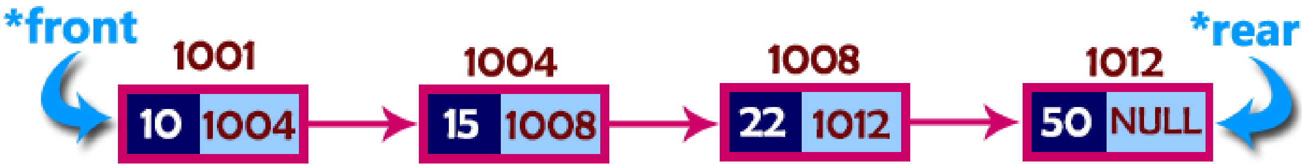
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 3:** Display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

<u>QUEUE USING LINKED LIST</u>

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...

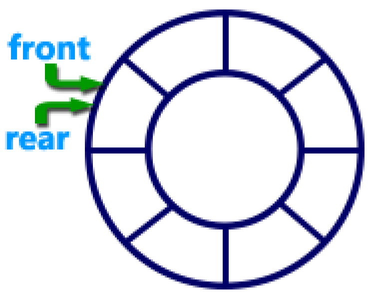
- **Step 1:** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2:** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4:** Display '**temp** → **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**'.

What is Circular Queue?

A Circular Queue can be defined as follows...

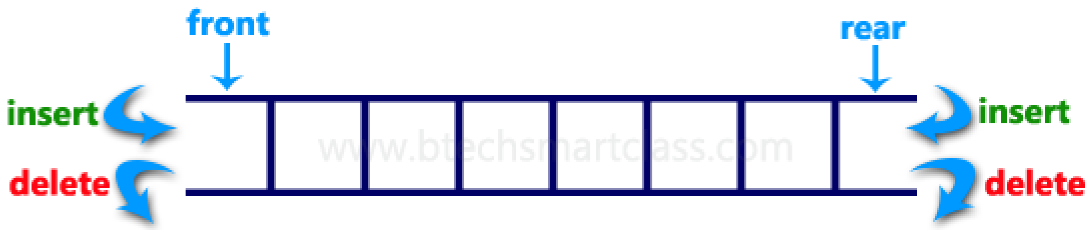
Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



<u>DOUBLE ENDED QUEUE (DEQUEUE)</u>

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.

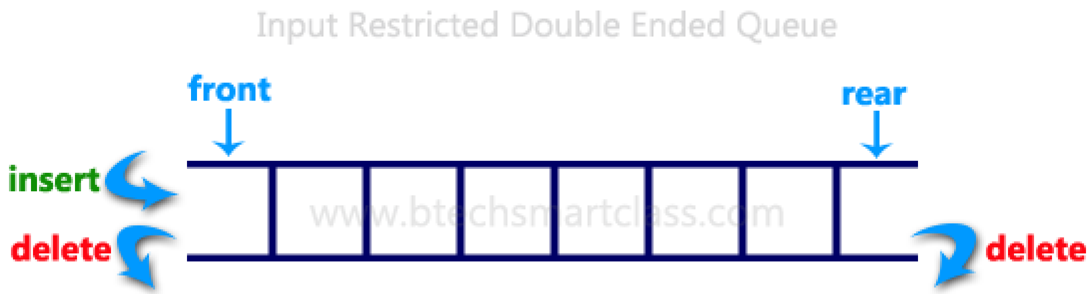


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

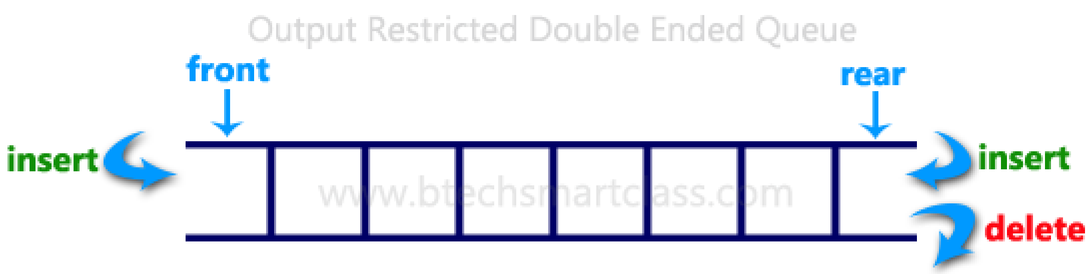
Input Restricted Double Ended Queue

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

In normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all situations.

Consider a networking application where server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows...

Now, check waiting time for each request to be complete.

1. **R1 : 20 units of time**
2. **R2 : 22 units of time (R2 must wait till R1 complete - 20 units and R2 itself requeres 2 units. Total 22 units)**
3. **R3 : 32 units of time (R3 must wait till R2 complete - 22 units and R3 itself requeres 10 units. Total 32 units)**
4. **R4 : 37 units of time (R4 must wait till R3 complete - 35 units and R4 itself requeres 5 units. Total 37 units)**

Here, average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.

Now, consider another way of serving these requests. If we serve according to their required amount of time. That means, first we serve R2 which has minimum time required (2) then serve R4 which has second minimum time required (5) then serve R3 which has third minimum time required (10) and finnaly R1 which has maximum time required (20).

Now, check waiting time for each request to be complete.

1. **R2 : 2 units of time**
2. **R4 : 7 units of time (R4 must wait till R2 complete 2 units and R4 itself requeres 5 units. Total 7 units)**
3. **R3 : 17 units of time (R3 must wait till R4 complete 7 units and R3 itself requeres 10 units. Total 17 units)**
4. **R1 : 37 units of time (R1 must wait till R3 complete 17 units and R1 itself requeres 20 units. Total 37 units)**

Here, average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.

From above two situations, it is very clear that, by using second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the 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**
3. **Max Priority Queue**

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.

Min Priority Queue

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

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
6. What is 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. Write about Queues, Array and Linked representation of Queue.
6. Explain about Normal and Circular Lis.
7. Explain De-queue operations.
8. Explain about Priority Queues
9. Write about the methods of Self Organizing Lists.
10. Explain about Skip Lists

KARPAGAM ACADEMY OF HIGHER EDUCATION
Coimbatore - 641021.
(For the candidates admitted from 2017 onwards)
DEPARTMENT OF COMPUTER SCIENCE,CA & IT

UNIT III :(Objective Type/Multiple choice Questions each Question carries one Mark)

Subject :Data Structures Sub Code:18CTU301

PART-A (Online Examination)

Sno	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4			ANSWER
1	_______________ are genealogical charts which are used to present the data	Graphs	Pedigree and lineal chart	Line , bar chart	pie chart			**Pedigree and lineal chart**
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	graph	list	set			**tree**
3	A _________ is a graph without any cycle.	tree	path	set	list			**tree**

4	In binary trees there is no node with a degree greater than _____	zero	one	two	three			**two**
5	Which of this is true for a binary tree.	It may be empty	The degree of all nodes must be <=2	It contains a root node	All the options			**All the options**
6	The Number of subtrees of a node is called its _______.	leaf	terminal	children	degree			**degree**
7	Nodes that have degree zero are called ________.	end node	leaf nodes	subtree	root node			**leaf nodes**
8	A binary tree with all its left branches supressed is called a _________	balanced tree	left sub tree	full binary tree	right skewed tree			**right skewed tree**
9	All node except the leaf nodes are called________.	terminal node	percent node	non terminal	children node			**non terminal**
10	The roots of the subtrees of a node X, are the _______ of X.	Parent	Children	Sibling	sub tree			**Children**
11	X is a root then X is the ______ of its children.	sub tree	Parent	Sibilings	subordinate			**Parent**
12	The children of the same parent are called __________________.	sibiling	leaf	child	subtree			**sibiling**

13	___________of a node are all the nodes along the path form the root to that node.	Degree	sub tree	Ancestors	parent			**Ancestors**
14	The ______________of a tree is defined to be a maximum level of any node in the tree.	weight	length	breath	height			**height**
15	A___________ is a set of n ≥ 0 disjoint trees	Group	forest	Branch	sub tree			**forest**
16	A tree with any node having at most two branches is called a _____________.	branched tree	sub tree	binary tree	forest			**binary tree**
17	A ___________of depth k is a binary tree of depth k having 2^K-1 nodes.	full binary tree	half binary tree	sub tree	n branch tree			**full binary tree**
18	Data structure represents the hierarchical relationships between individual data item is known as __________.	Root	Node	Tree	Address			**Tree**
19	Node at the highest level of the tree is known as _______.	Child	Root	Sibiling	Parent			**Root**
20	The root of the tree is the _______of all nodes in the tree.	Child	Parent	Ancestor	Head			**Ancestor**
21	_____is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree			**Subtree**

22	A node with no children is called __________.	Root Node	Branch	Leaf Node	Null tree			**Leaf Node**
23	In a tree structure a link between parent and child is called ________	Branch	Root	Leaf	Subtree			**Branch**
24	Height – balanced trees are also referred as as ____________trees.	AVL trees	Binary Trees	Subtree	Branch Tree			**AVL trees**
25	Visiting each node in a tree exactly once is called __________	searching	travering	walk through	path			**travering**
26	In_________traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order			**PreOrder**
27	In_________traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order			**Inorder**
28	In_________traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order			**PostOrder**
29	Inorder traversal is also sometimes called_______	Symmetric Order	End Order	PreOrder	PostOrder			**Symmetric Order**
30	Postorder traversal is also sometimes called_______	Symmetric Order	End Order	PreOrder	PostOrder			**End Order**

31	Nodes of any level are numbered from _________	Left to right	Right to Left	Top to Bottom	Bottom to Top			Left to right
32	________ search involves only addition and subtraction.	binary	fibonacci	sequential	non sequential			**fibonacci**
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	radix	merge	heap			**heap**
34	Binary trees are used in ______ sorting.	quick sort	merge sort	heap sort	lrsort			**heap sort**
35	The ____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch			**Root**
36	In Threaded Binary Tree ,LCHILD(P) is a normal pointer When LBIT(P) = ____	1	2	3	0			**1**
37	In Threaded Binary Tree ,LCHILD(P) is a Thread When LBIT(P) = ____	1	2	3	0			**0**
38	In Threaded Binary Tree ,RCHILD(P) is a normal pointer When RBIT(P) = ____	2	1	3	0			**1**
39	In Threaded Binary Tree ,RCHILD(P) is a Thread When LBIT(P) = ____	1	2	0	4			**0**

40	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	m-way search			**Binary search**
41	______ algorithm can be used only with sorted lists.	Linear search	Binary search	insertion sort	merge sort			**Binary search**
42	________ search involves comparision of the element to be found with every elements in a list.	Linear search	Binary search	fibonacci search	m-way search			**Linear search**
43	Binary search algorithm in a list of n elements takes only _______ time.	$O(\log_2 n)$	O(n)	$O(n^3)$	$O(n^2)$			**$O(\log_2 n)$**
44	_____ is used for decision making in eight coin problem.	trees	graphs	linked lists	array			**trees**
45	The Linear search algorithm in a list of n element takes ________ time to compare in worst case.	constant	linear	quadratic	exponential			**constant**
46	Which of these is an application of trees.	Finding minimum cost spanning tree	Decision tree	Storage management	Job sequencing			**Decision tree**
47	________ is an operation performed on sets	union	sort	rename	traverse			**union**
48	In sets _______ is used to find the set containing the element i	subset(i)	Disjoin(i)	Union(i)	Find(i)			**subset(i)**

49	Sets are represented as _____	arrays	linked lists	graphs	trees			**trees**
50	_________ is an example of application of trees in decision making.	Binay search	Optimal merge pattern	Eight Coins problem	Huffman's Message coding			**Eight Coins problem**
51	_________ is a internal sorting method.	sorting with disks	quick sort	balanced merge sort	sorting with tapes			**quick sort**
52	Quick sort reads ________ space to implement the recursion.	stack	queue	circular stacks	circular queue			**stack**
53	The most popular method for sorting on external storage devices is _____.	quick sort	radix sort	merge sort	heap sort			**merge sort**
54	The 2-way merge algorithm is almost identical to the ____________procedure.	quick	merge	heap	radix			**merge**
55	A _________ merge on m runs requires at most [log $_k$m] passes over the data.	n-way	m-way	k-way	q-way			**k-way**
56	Combining elements of two similar data structure into one is called ___________	Merging	Insertion	Searching	Sorting			**Merging**
57	Adding a new element into a data structure called ___________	Merging	Insertion	Searching	Sorting			**Insertion**

58	The Process of finding the location of the element with the given value or a record with the given key is ___________.	Merging	Insertion	Searching	Sorting			**Searching**
59	Arranging the elements of a data structure in some type of order is called ___________.	Merging	Insertion	Searching	Sorting			**Sorting**
60	In ________ search method the search begins by examining the record in the middle of the file.	sequential	fibonacci	binary	non-sequential			**binary**
61	__________ sort is done in graphs	Merge sort	Heap	Topological sort	Linear sort			**Topological sort**
62	The disadvantage of _____ sort is that is need a temporary array to sort.	Quick	Merge	Heap	Insertion			**Merge**
63	The cost of decoding a code word is -------------------- to the number of bits in the code	equal	not equal	proportional	inversely proportional			**proportional**
64	The solution of finding a binary tree with minimum weighted external path length has been given by	Huffman	Kruskal	Euler	Hamilton			**Huffman**
65	A ---tree is a binary tree in which external nodes represent messages	decode	uncode	extended	encode			**decode**
66	The identifier x is divided by some number m and the remainder is used as the hash address for x .Then f(x) is	m mod x	x mod m	m mod f	d mod x			**x mod** m

67	The identifier is folded at the part boundaries and digits falling into the same position are added together to obtain f(x).this method adding is called	folding at the boundaries	shift method	folding method	Tag method			**folding at the boundaries**

SYLLABUS

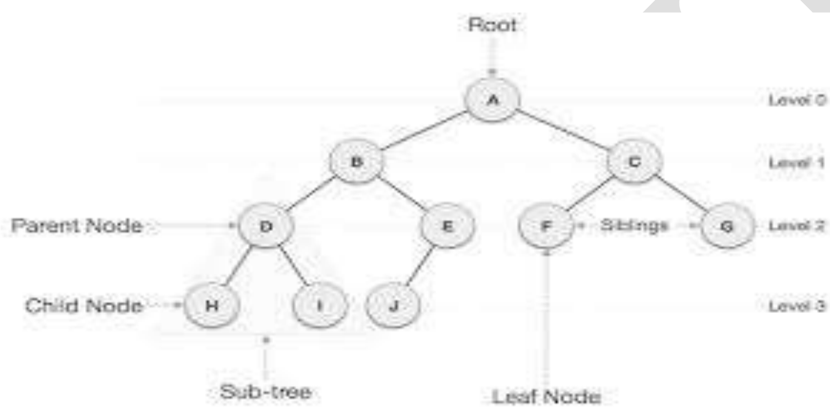
UNIT-III

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.

<u>Binary Trees</u>:

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.

<u>Binary Search Trees</u>:

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

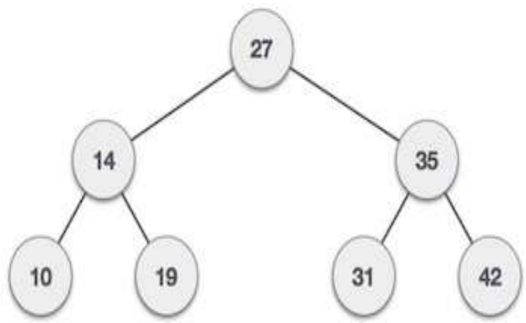
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

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

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

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

Following is a pictorial representation of BST −



Binary Search Tree

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

Basic Operations:

Following are the basic operations of a tree −

Search − Searches an element in a tree.

Insert − Inserts an element in a tree.

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

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

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

Node:

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

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

Search Operation:

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

Algorithm:

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

  while(current->data != data){

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

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

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

Insert Operation:

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

Algorithm:

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

```
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

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

      while(1) {
         parent = current;

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

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

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

TRAVERSAL:

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

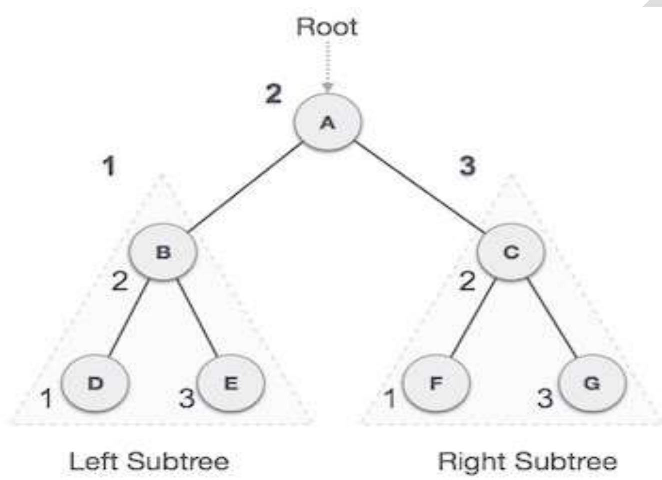
- In-order Traversal
- Pre-order Traversal

- Post-order Traversal

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

In-order Traversal

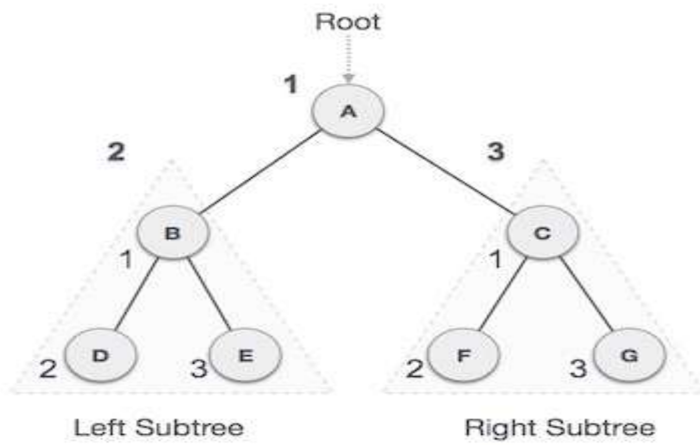
In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



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

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

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



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

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −

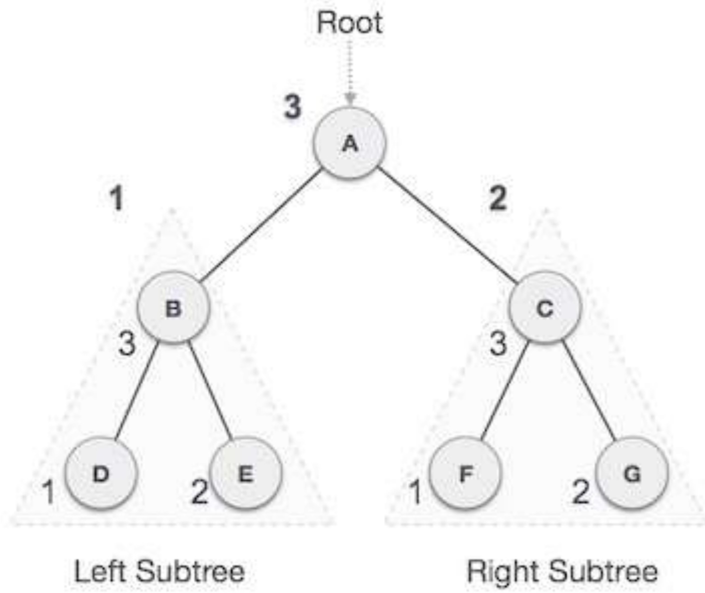
Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order Traversal:

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



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

D → E → B → F → G → C → A

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

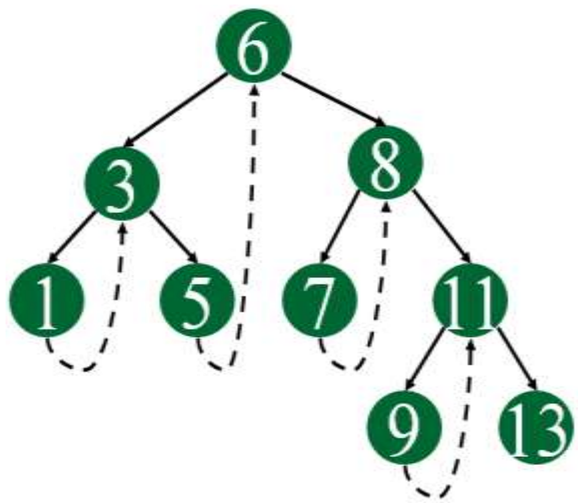
THREADED BINARY TREES:
Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

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

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

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

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



Representation of a Threaded Node:

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

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

Inorder Taversal using Threads

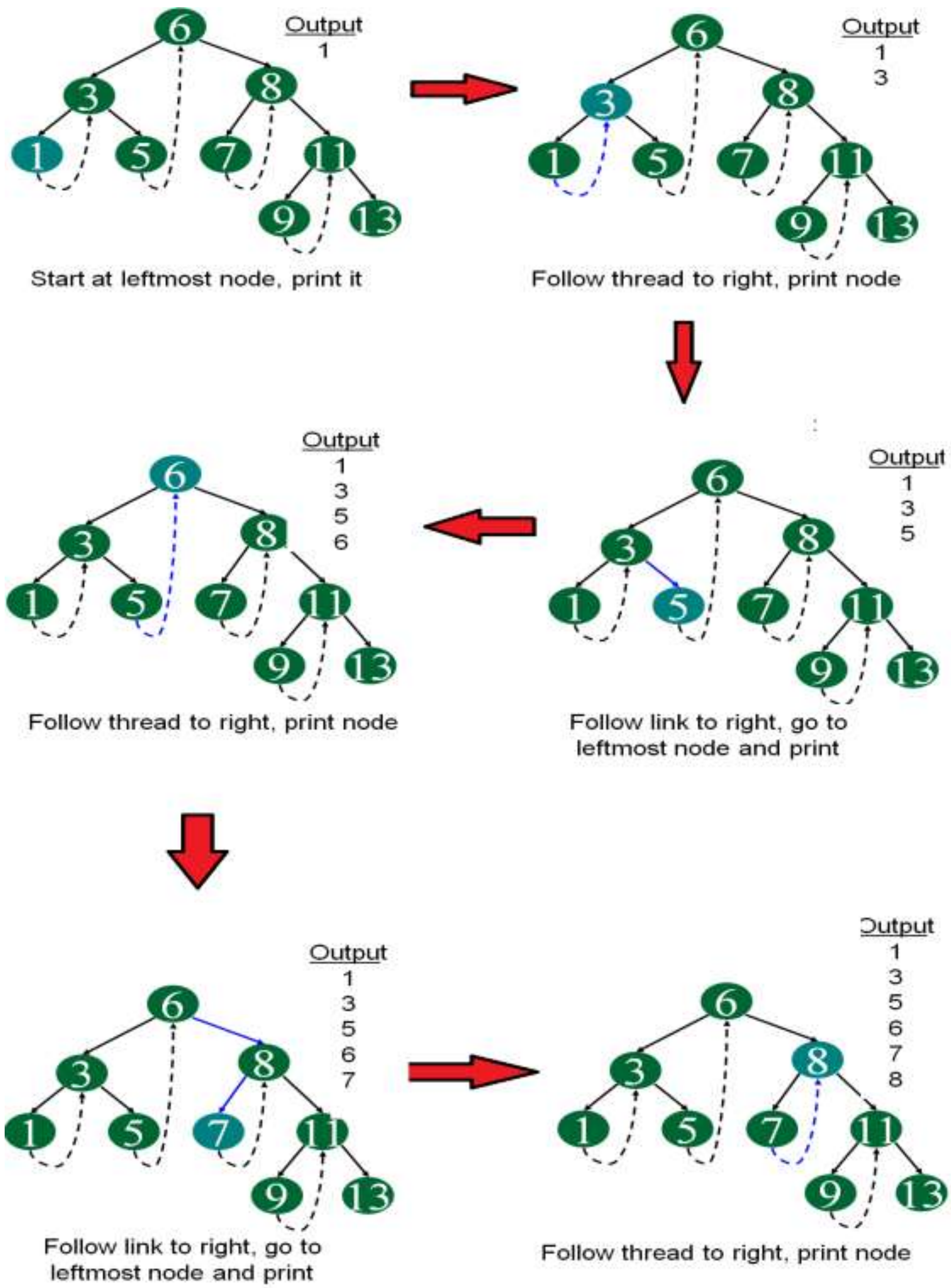
Following code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n
```

```
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
       return NULL;

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

    return n;
}
```

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

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

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



continue same way for remaining node.....

INSERTION:

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

representation of Binary Threaded Node:

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

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

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

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

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

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

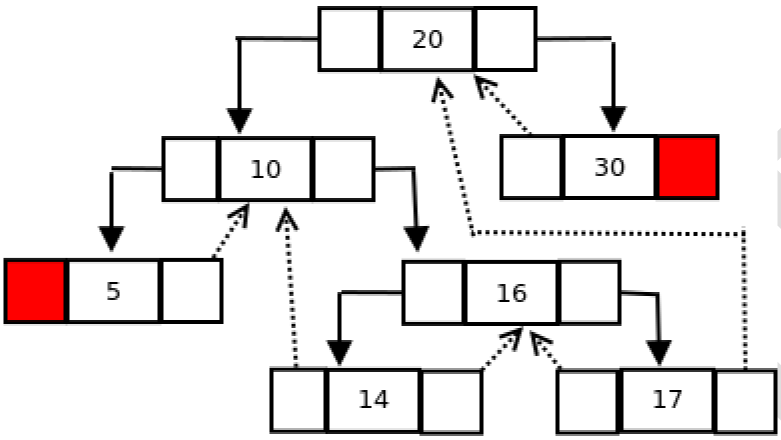
```
root = tmp;
tmp -> left = NULL;
tmp -> right = NULL;
```

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

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

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

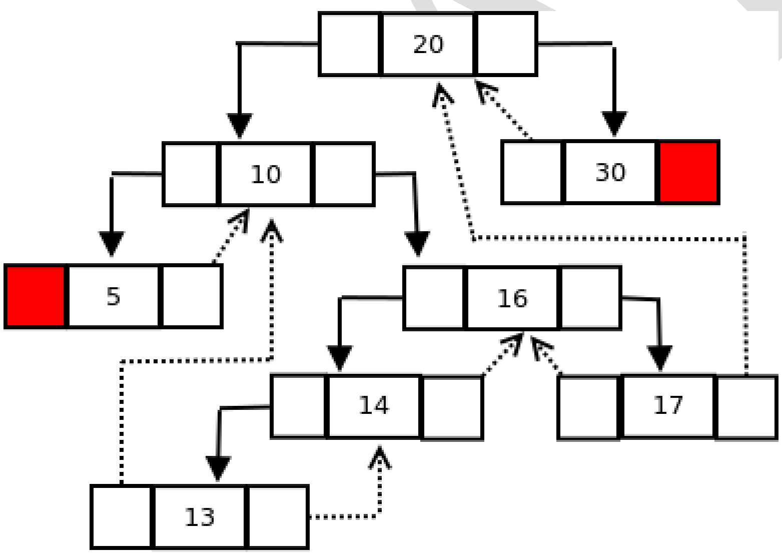
par -> lthread = par ->left;
par -> left = temp;



Insert 13

Inorder : 5 10 14 16 17 20 30

After insertion of 13,



13 inserted as left child of 14

Inorder : 5 10 13 14 16 17 20 30

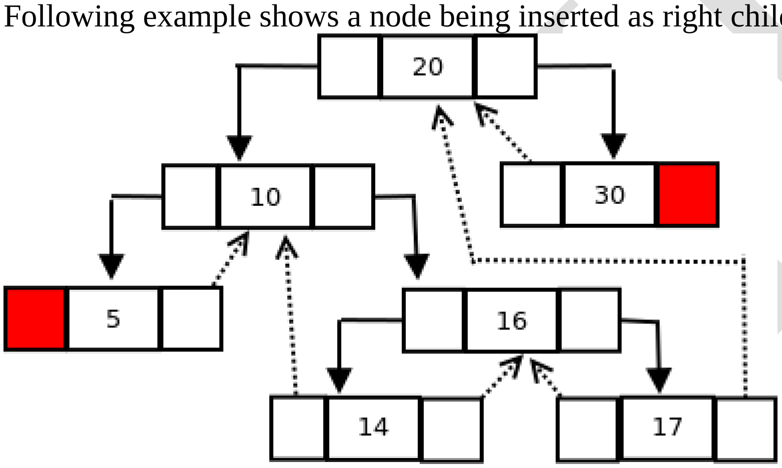
Predecessor of 14 becomes the predecessor of 13, so left thread of 13 points to 10.

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

Case 3: When new node is inserted as the right child
The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

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

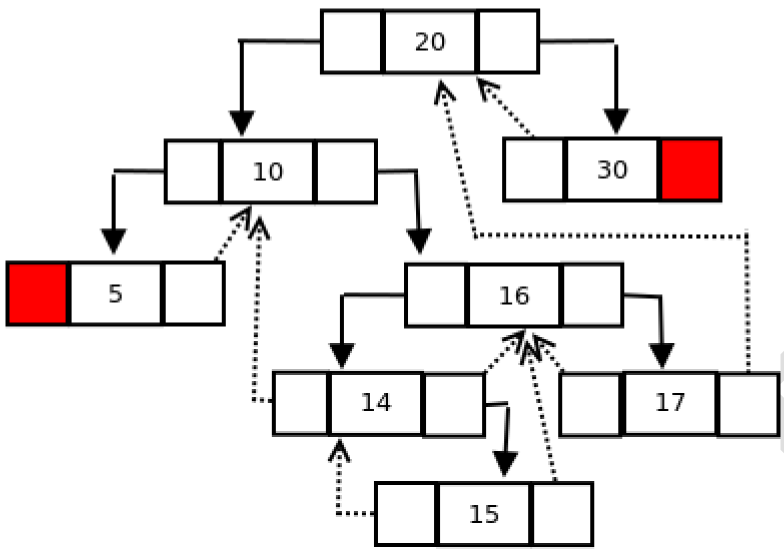
par -> rthread = false;
par -> right = tmp;
Following example shows a node being inserted as right child of its parent.



Insert 15

Inorder : 5 10 14 16 17 20 30

After 15 inserted,

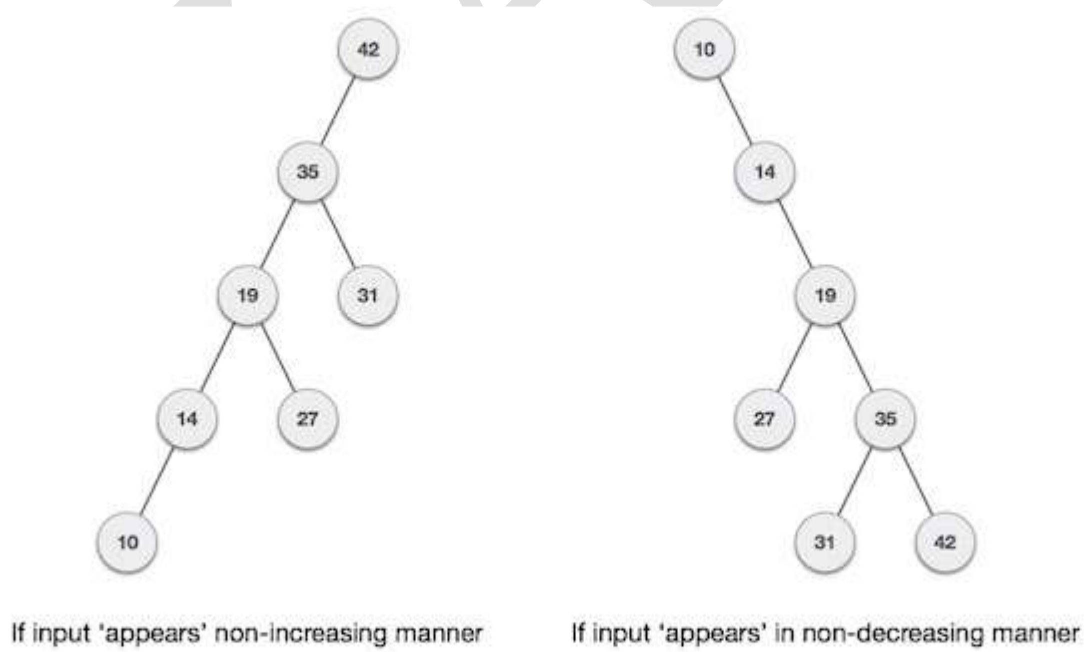


15 inserted as right child of 14
Inorder : 5 10 14 15 16 17 20 30

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

Height-Balanced Trees:

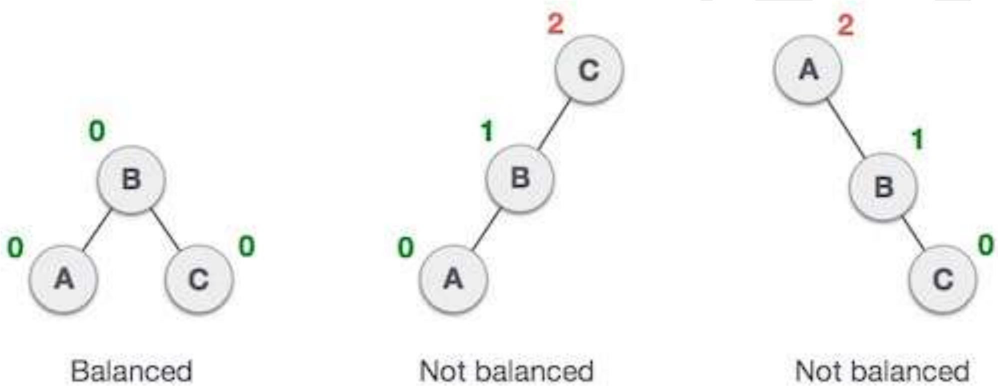
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



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 −

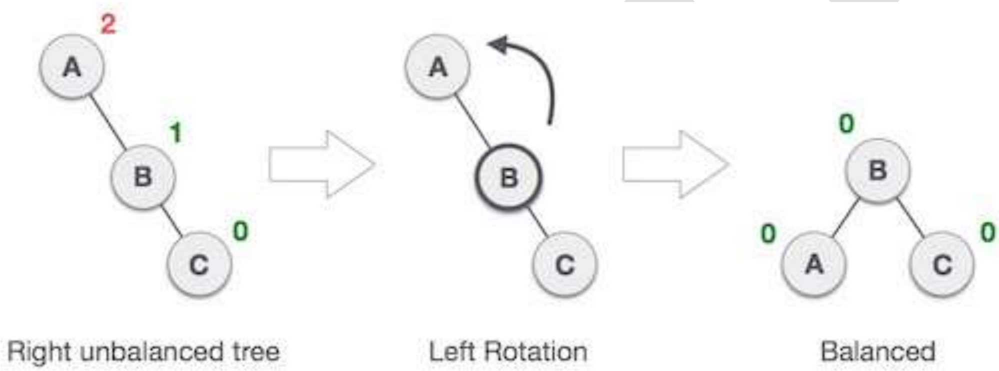
- Left rotation
- Right rotation

- 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

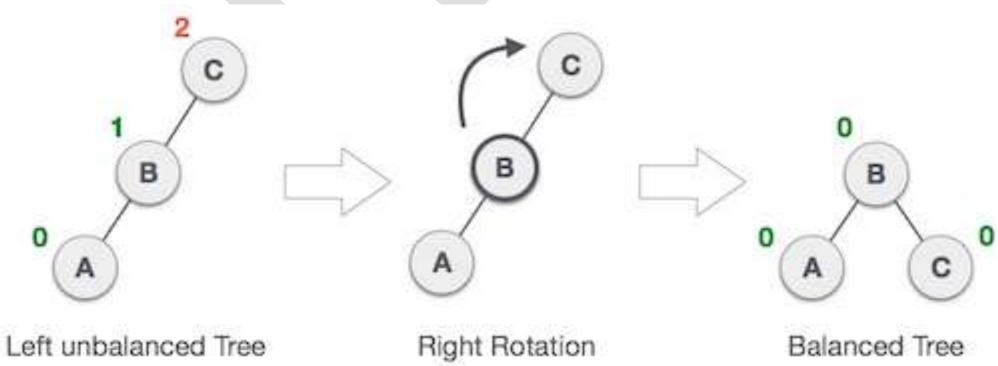
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



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.

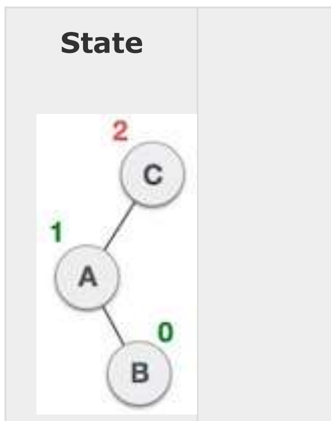
Right Rotation:

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.

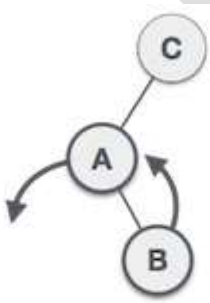


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

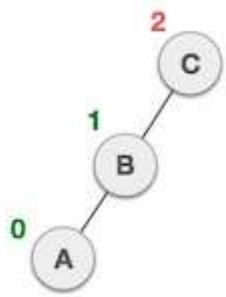
Left-Right Rotation:Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation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



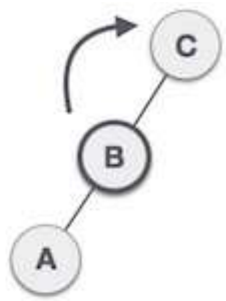
A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.



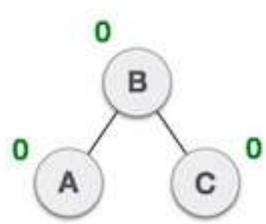
We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.



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



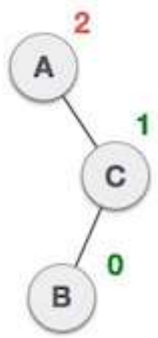
We shall now right-rotate the tree, making **B**the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.



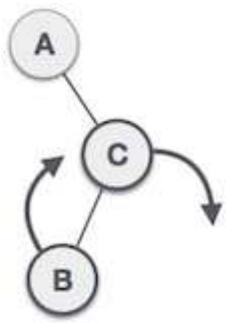
The tree is now balanced.

Right-Left Rotation:

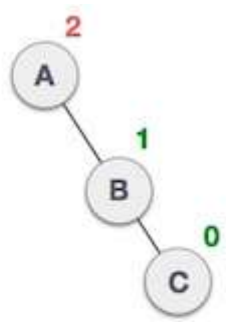
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation



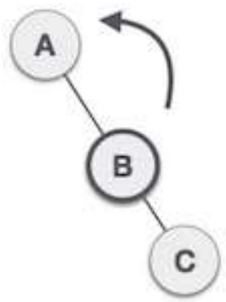
A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.



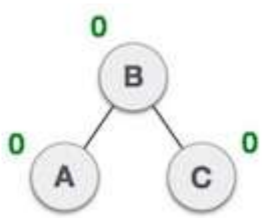
First, we perform the right rotation along **C**node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.



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...

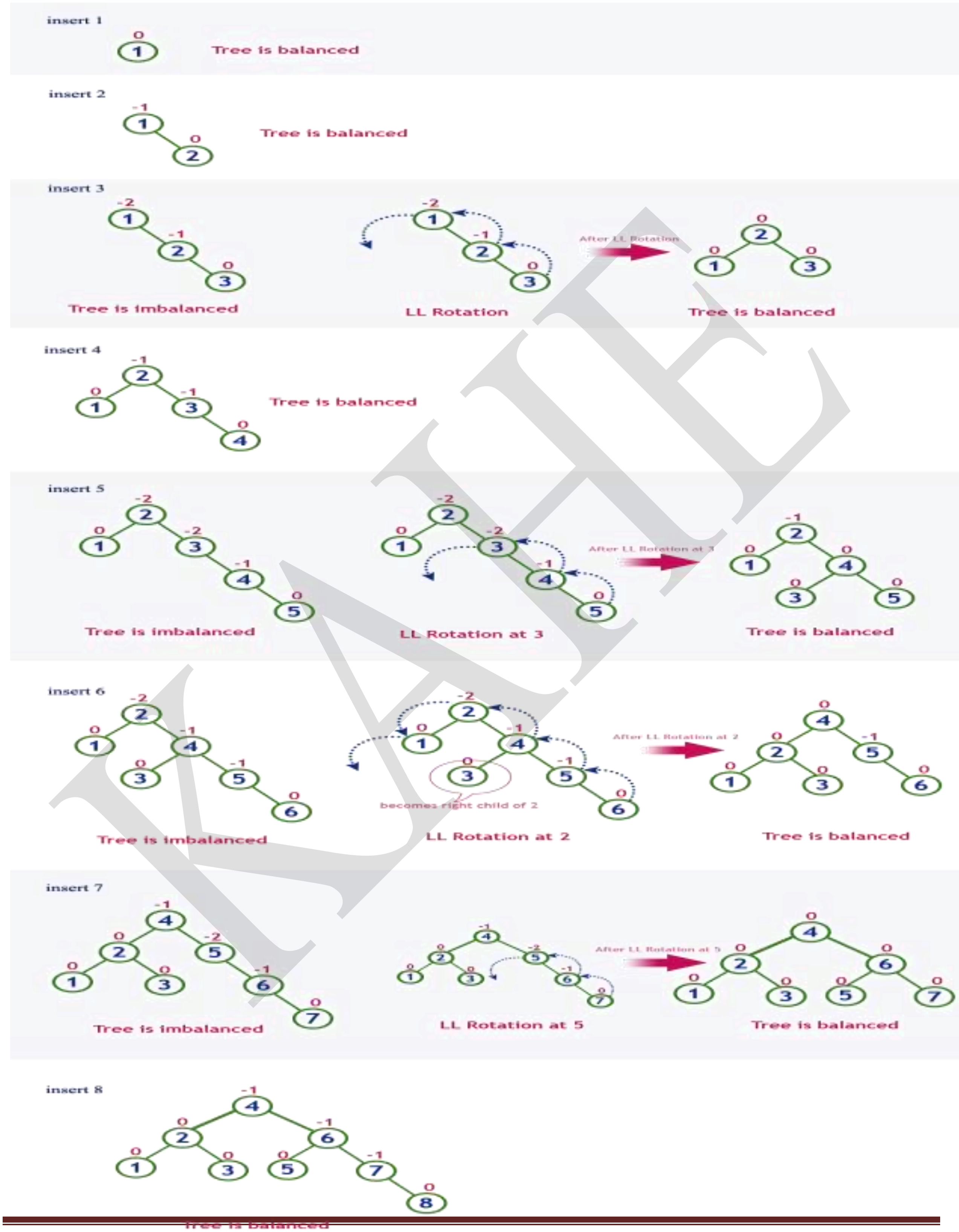
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

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

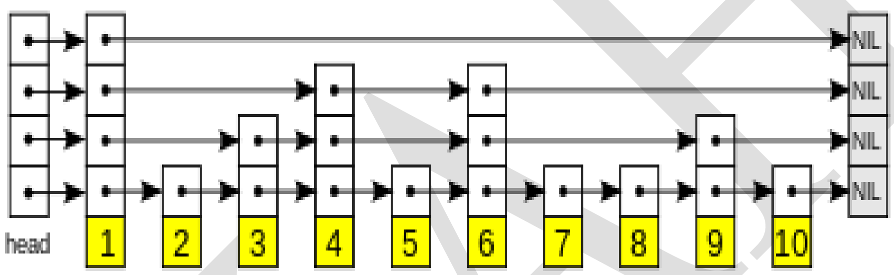
Deletion Operation in AVL Tree:

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.



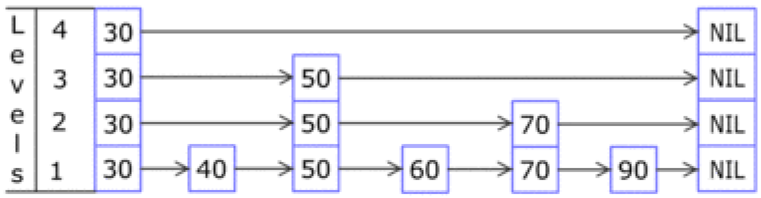
A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes (in yellow) at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the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

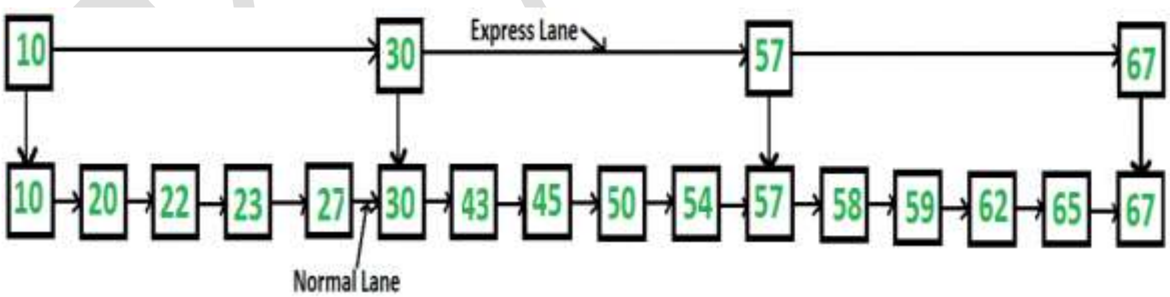
Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list.



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 Examinations)

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.

QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4			**ANSWER**
_______________ are genealogical charts which are used to present the data	Graphs	Pedigree and lineal chart	Line , bar chart	pie chart			**Pedigree and lineal chart**
A __ is a finite set of one or more nodes, with one root node and remaining form the disjoint sets forming the subtrees.	tree	graph	list	set			**tree**
A _________ is a graph without any cycle.	tree	path	set	list			**tree**
In binary trees there is no node with a degree greater than _____	zero	one	two	three			**two**
Which of this is true for a binary tree.	It may be empty	The degree of all nodes must be <=2	It contains a root node	All the options			**All the options**
The Number of subtrees of a node is called its _______.	leaf	terminal	children	degree			**degree**
Nodes that have degree zero are called ________.	end node	leaf nodes	subtree	root node			**leaf nodes**
A binary tree with all its left branches supressed is called a _________	balanced tree	left sub tree	full binary tree	right skewed tree			**right skewed tree**

All node except the leaf nodes are called________.	terminal node	percent node	non terminal	children node			**non terminal**
The roots of the subtrees of a node X, are the _______ of X.	Parent	Children	Sibling	sub tree			**Children**
X is a root then X is the ______ of its children.	sub tree	Parent	Sibilings	subordinate			**Parent**
The children of the same parent are called __________________.	sibiling	leaf	child	subtree			**sibiling**
___________of a node are all the nodes along the path form the root to that node.	Degree	sub tree	Ancestors	parent			**Ancestors**
The ______________of a tree is defined to be a maximum level of any node in the tree.	weight	length	breath	height			**height**
A___________ is a set of $n \geq 0$ disjoint trees	Group	forest	Branch	sub tree			**forest**
A tree with any node having at most two branches is called a _____________.	branched tree	sub tree	binary tree	forest			**binary tree**
A ___________of depth k is a binary tree of depth k having 2^K-1 nodes.	full binary tree	half binary tree	sub tree	n branch tree			**full binary tree**

Data structure represents the hierarchical relationships between individual data item is known as ___________.	Root	Node	Tree	Address			**Tree**
Node at the highest level of the tree is known as _______.	Child	Root	Sibiling	Parent			**Root**
The root of the tree is the _______of all nodes in the tree.	Child	Parent	Ancestor	Head			**Ancestor**
_____is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree			**Subtree**
A node with no children is called _________.	Root Node	Branch	Leaf Node	Null tree			**Leaf Node**
In a tree structure a link between parent and child is called _______	Branch	Root	Leaf	Subtree			**Branch**
Height – balanced trees are also referred as as ___________trees.	AVL trees	Binary Trees	Subtree	Branch Tree			**AVL trees**
Visiting each node in a tree exactly once is called _________	searching	travering	walk through	path			**travering**
In________traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order			**PreOrder**

In________traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order			**Inorder**
In________traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order			**PostOrder**
Inorder traversal is also sometimes called______	Symmetric Order	End Order	PreOrder	PostOrder			**Symmetric Order**
Postorder traversal is also sometimes called______	Symmetric Order	End Order	PreOrder	PostOrder			**End Order**
Nodes of any level are numbered from _________	Left to right	Right to Left	Top to Bottom	Bottom to Top			Left to right
________ search involves only addition and subtraction.	binary	fibonacci	sequential	non sequential			**fibonacci**
A________ is defined to be a complete binary tree with the property that the value of root node is at least as large as the value of its children node.	quick	radix	merge	heap			**heap**
Binary trees are used in ______ sorting.	quick sort	merge sort	heap sort	lrsort			**heap sort**
The ____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch			**Root**

In Threaded Binary Tree ,LCHILD(P) is a normal pointer When LBIT(P) = ____	1	2	3	0			**1**
In Threaded Binary Tree ,LCHILD(P) is a Thread When LBIT(P) = ____	1	2	3	0			**0**
In Threaded Binary Tree ,RCHILD(P) is a normal pointer When RBIT(P) = ____	2	1	3	0			**1**
In Threaded Binary Tree ,RCHILD(P) is a Thread When LBIT(P) = ____	1	2	0	4			**0**
Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	m-way search			**Binary search**
______ algorithm can be used only with sorted lists.	Linear search	Binary search	insertion sort	merge sort			**Binary search**
________ search involves comparision of the element to be found with every elements in a list.	Linear search	Binary search	fibonacci search	m-way search			**Linear search**
Binary search algorithm in a list of n elements takes only _______ time.	$O(\log_2 n)$	O(n)	$O(n^3)$	$O(n^2)$			**$O(\log_2 n)$**
_____ is used for decision making in eight coin problem.	trees	graphs	linked lists	array			**trees**

The Linear search algorithm in a list of n element takes ________ time to compare in worst case.	constant	linear	quadratic	exponential			**constant**
Which of these is an application of trees.	Finding minimum cost spanning tree	Decision tree	Storage management	Job sequencing			**Decision tree**
________ is an operation performed on sets	union	sort	rename	traverse			**union**
In sets _______ is used to find the set containing the element i	subset(i)	Disjoin(i)	Union(i)	Find(i)			**subset(i)**
Sets are represented as _____	arrays	linked lists	graphs	trees			**trees**
_________ is an example of application of trees in decision making.	Binay search	Optimal merge pattern	Eight Coins problem	Huffman's Message coding			**Eight Coins problem**
_________ is a internal sorting method.	sorting with disks	quick sort	balanced merge sort	sorting with tapes			**quick sort**
Quick sort reads _______ space to implement the recursion.	stack	queue	circular stacks	circular queue			**stack**
The most popular method for sorting on external storage devices is _____.	quick sort	radix sort	merge sort	heap sort			**merge sort**

The 2-way merge algorithm is almost identical to the ____________procedure.	quick	merge	heap	radix			**merge**
A _________ merge on m runs requires at most [log $_k$m] passes over the data.	n-way	m-way	k-way	q-way			**k-way**
Combining elements of two similar data structure into one is called ___________	Merging	Insertion	Searching	Sorting			**Merging**
Adding a new element into a data structure called ___________	Merging	Insertion	Searching	Sorting			**Insertion**
The Process of finding the location of the element with the given value or a record with the given key is ___________.	Merging	Insertion	Searching	Sorting			**Searching**
Arranging the elements of a data structure in some type of order is called ___________.	Merging	Insertion	Searching	Sorting			**Sorting**
In ________ search method the search begins by examining the record in the middle of the file.	sequential	fibonacci	binary	non-sequential			**binary**
__________ sort is done in graphs	Merge sort	Heap	Topological sort	Linear sort			**Topological sort**
The disadvantage of _____ sort is that is need a temporary array to sort.	Quick	Merge	Heap	Insertion			**Merge**

The cost of decoding a code word is -------------------- to the number of bits in the code	equal	not equal	proportional	inversely proportional			**proportional**
The solution of finding a binary tree with minimum weighted external path length has been given by	Huffman	Kruskal	Euler	Hamilton			**Huffman**
A ---tree is a binary tree in which external nodes represent messages	decode	uncode	extended	encode			**decode**
The identifier x is divided by some number m and the remainder is used as the hash address for x .Then f(x) is	m mod x	x mod m	m mod f	d mod x			**x mod** m
The identifier is folded at the part boundaries and digits falling into the same position are added together to obtain f(x).this method adding is called	folding at the boundaries	shift method	folding method	Tag method			**folding at the boundaries**

UNIT-IV

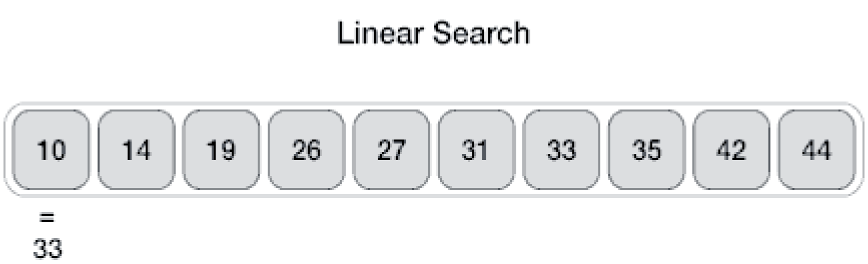
Searching and Sorting,Linear Search, Binary Search, Comparison of Linear and Binary Search, Selection Sort, Insertion Sort, Insertion Sort, Shell Sort, Comparison of Sorting Techniques

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
```

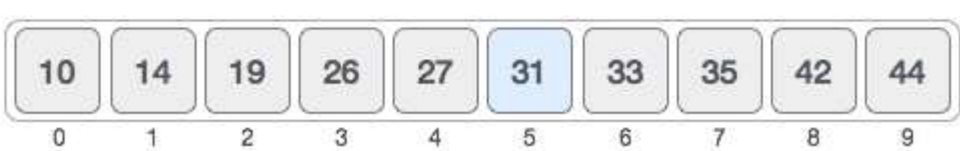
<u>BINARY SEARCH</u>:

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:

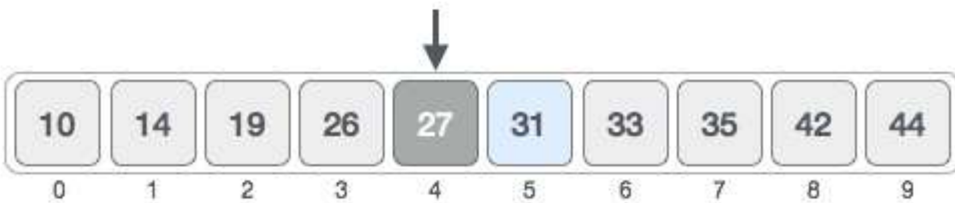
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



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.



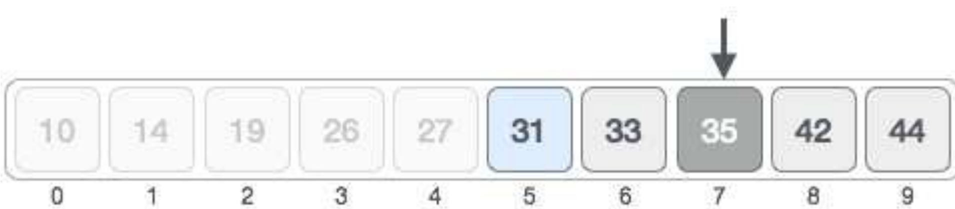
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

low = mid + 1

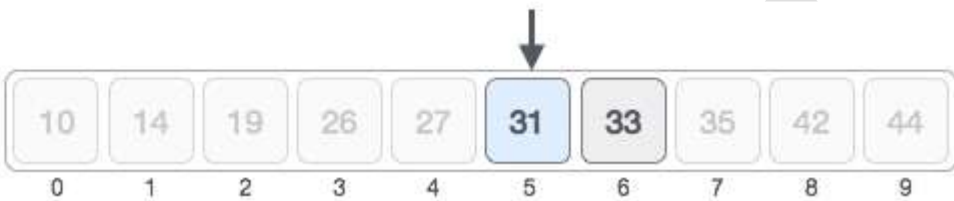
mid = low + (high - low) / 2

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

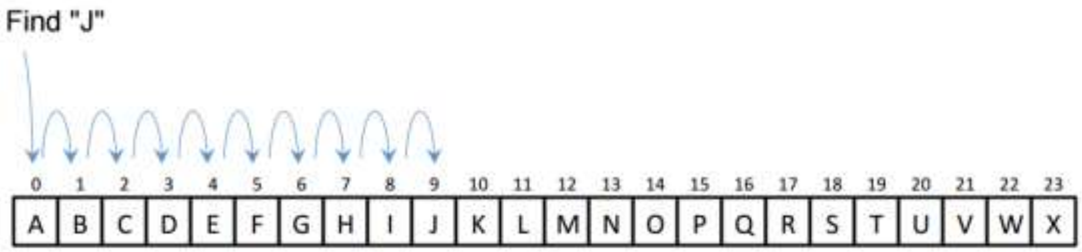
Linear search does the sequential access whereas Binary search access data randomly.

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

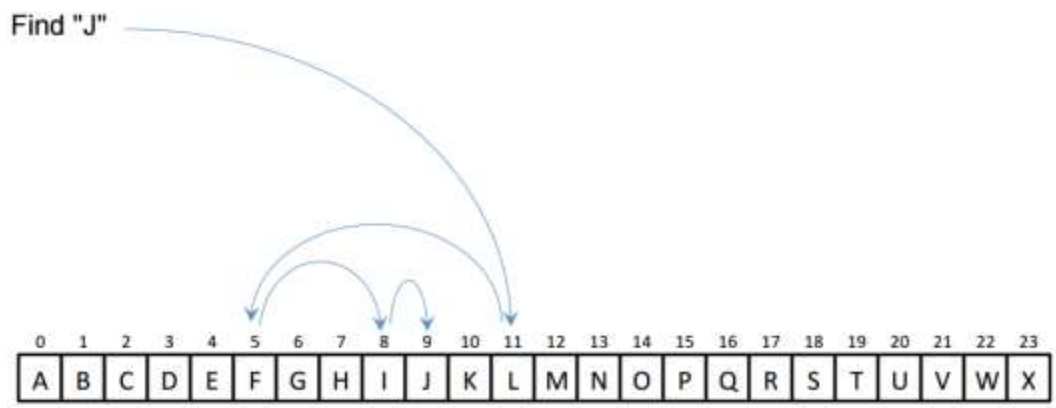
Let us look at an example to compare the two:

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

<u>SORTING</u>:

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.

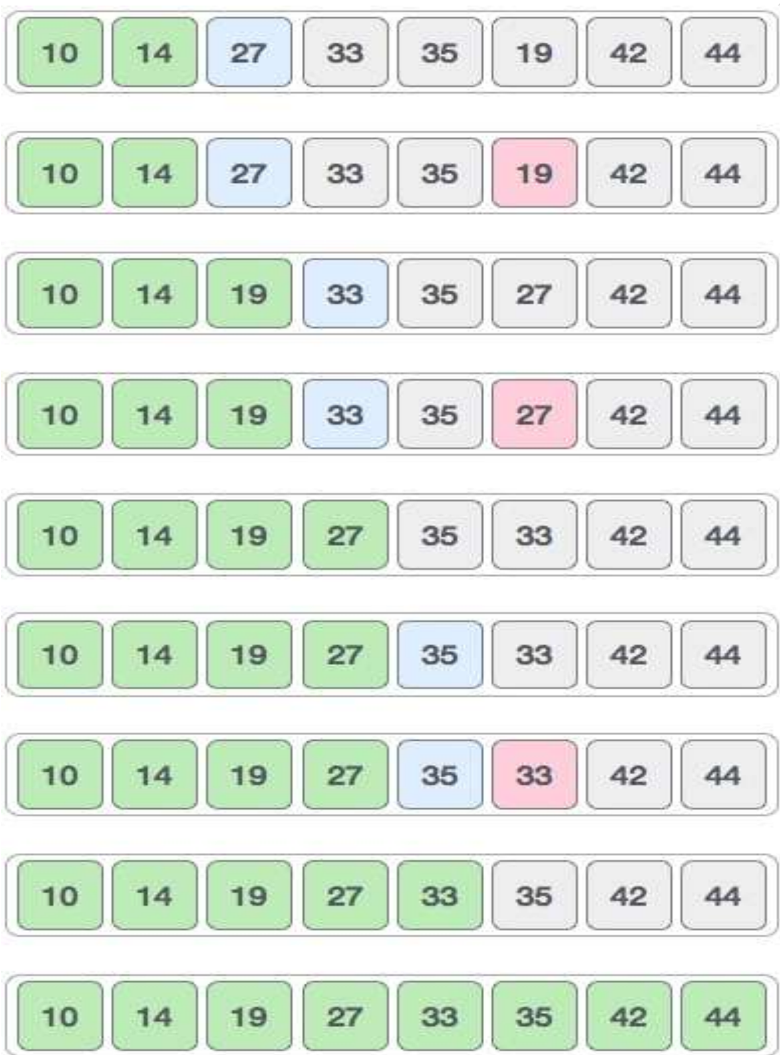
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

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:

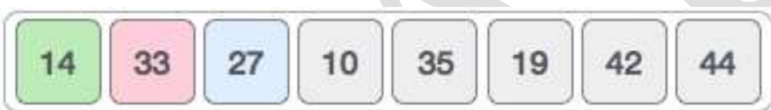
We take an unsorted array for our example.

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
```

<u>Shell Sort</u>:

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.

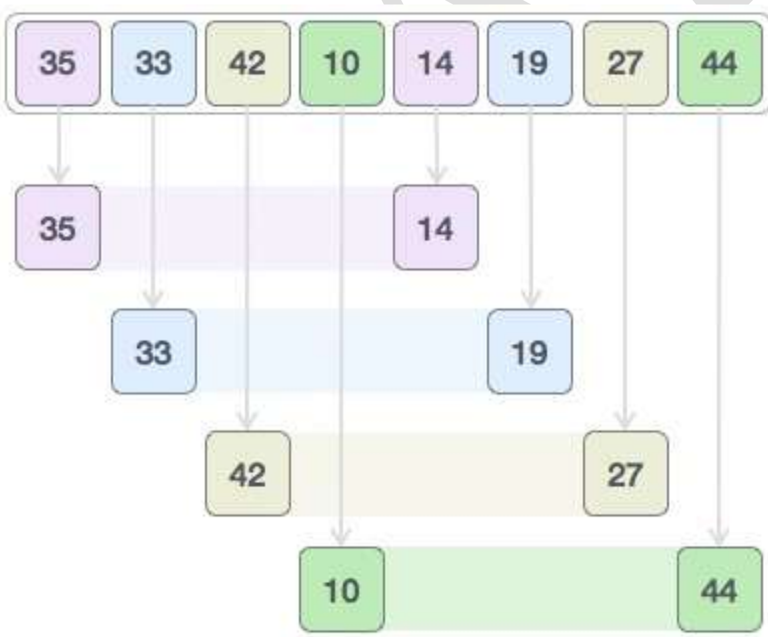
This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as −

Knuth's Formula

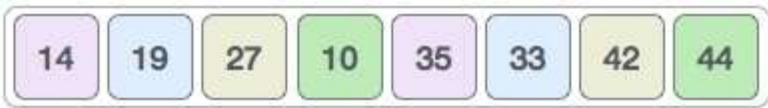
h = h * 3 + 1 where − h is interval with initial value 1

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.

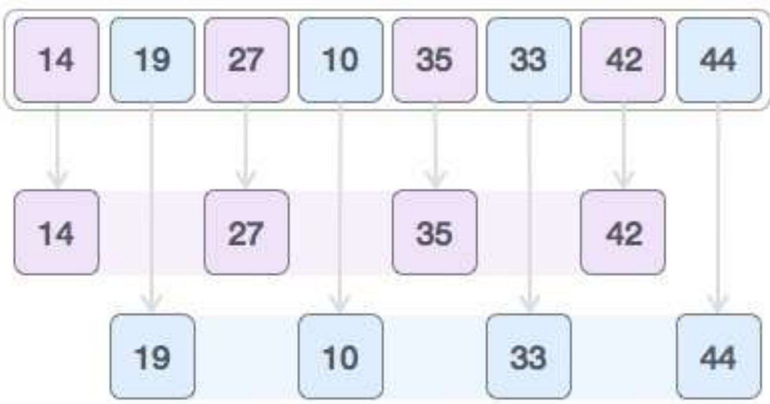
Shell Sort Works: Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions.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



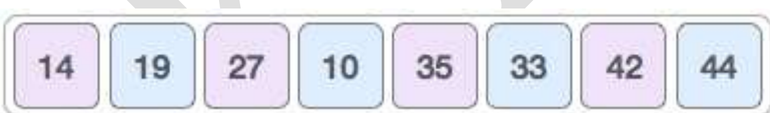
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



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 Examinations)

PART-B (2 MARKS)

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

PART-C (6 MARKS)

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

KARPAGAM ACADEMY OF HIGHER EDUCATION
Coimbatore - 641021.
(For the candidates admitted from 2017 onwards)
DEPARTMENT OF COMPUTER SCIENCE,CA & IT

UNIT IV :(Objective Type/Multiple choice Questions each Question carries one Mark)

Subject :Data Structures Sub Code:18CTU301

PART-A (Online Examination)

S. NO	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4	KEY
1	In a graph G(V,E), V is a finite non-empty set of	Nodes	Items	Vertices	Circles	Vertices
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	Kruskals path	Hamiltonian cycle	Eulerian walk	Koenisberg bridge	Eulerian walk
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.	connected	adjacent	neighbours	incident	connected
4	In a Graph G if there are n vertices the adjacency list	n/2	2n	n-1	n	n
5	In a Graph G if there are n vertices the adjacency Matrix of the graph consists of ________rows and	n/2	2n	n-1	n	n
6	In _________graph the pair of vertices joined by any	directed	undirected	sub	multi	undirected
7	In___________graph each edge is represented by the	undirected	multi	directed	sub	directed
8	The_________of a path is the number of edges on it.	tree	maximal	length	cycle	length
9	An n vertex undirected graph with exactly n(n-1)/2 distinct edges is said to be ___________	connected	complete	directed	cyclic	complete
10	A_________is a simple path in which the first & last vertices are the same.	Cycle	graph	component	matrix	Cycle
11	A connected component of an undirected graph is a _______connected subgraph.	tree	strongly	weekly	maximal	maximal

12	A _________is a connected acyclic graph.	graph	component	tree	list	tree
13	In a graph with n vertices the number of distinct unordered pairs(vi,vj) with vi not equal to vj is	n(n-1)/2	n-1	n(n-1)	n/2	n(n-1)/2
14	In a graph _____of a vertex is the number of edges incident to it.	path	degree	depth	height	degree
15	The _________of a vertex is defined as the number of edges for which v is the head.	out-degree	pre-degree	in-degree	post-degree	in-degree
16	The _________is defined to be the number of edges for which v is the tail	in-degree	out-degree	pre-degree	post-degree	out-degree
17	Directed graph is also called _______.	Line Graph	sub graph	connected graph	di graph	di graph
18	In a directed graph G(V,E) a vertex vj is _______vi iff there is an edge <vi,vj> in E	adjacent from	adjacency matrix	adjacent from	adjacency list	adjacent from
19	A vertex vi is adjacent to vj iff ________	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
20	An edge connected by any 2 vertices i & j can be determined by______	sparse matrix	adjacency matrix	linked lists	tree graph	adjacency matrix
21	An edge <vi,vj> is said to _________on two vertices vi and vj.	parallel	incident	degree	loop	incident
22	A ______from vertex vi to vj is a seqence of vertices from vi to vj with an edge connecting them in pairs	path	length	cycle	tree	path
23	The ______of a path is the number if edges on it.	degree	length	degree	height	length
24	The adjacency matrix of an undirect graph is always ______.	Unit Matrix	Diagonal Matix	Identity Matrix	SymetricMatrix	SymetricMa
25	A graph with weighted edge is called a ______.	strong graph	network	component	sub graph	network

26	Any tree consisting solely of edges in G and including all vertices in G is called _______.	graph	spanning tree	tri graph	bread the first spanning tree	spanning tree
27	The spanning tree resulting from a call to DFS is known as a ________spanning tree.	Independent	breadth first	depth first	dependent	depth first
28	When BFS is used the resulting spanning tree is called a ______spanning tree.	breadth first	depth first	independent	dependent	breadth first
29	_______and _______are the searching techniques used in graphs.	Inorder, preorder	firstfit, bestfit	depth first, breadth first	prefix, postfix	depth first, breadth
30	Starting from a vertex v and visiting all vertices adjacent to it before moving to the next vertex is called ________search	breadth first	depth first	pre order	first fit	breadth first
31	The all pairs shortest path problem calls for finding the _____paths between all pairs of vertices.	longest	shortest	minimal	maximal	shortest
32	A _________graph is a graph in which each vertex of G is adjacent to every other vertex in G.	Complete	Incomplete	connected	un connected	Complete
33	_________is a collection of records, each record having one or more fields.	data base	file	directory	address	file
34	A directed graph with no directed cycle is called ______	topological graph	subgraph	connected graph	acyclic graph	acyclic graph
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 _______	Heap sort	Merge sort	Topological sort	linear search	Topological sort
36	In a AOV network if i is a predecessor of j then j is the ____of i	root	successor	path	ancester	successor
37	A precedence relation which is both transitive and irreflexive is a _________	spanning tree	depth first searching	partial ordering	topological sorting	partial ordering

38	For a network with n vertices and e edges the asymptotic computing time of the topological ordering algorithm is _________	O(e+n)	O(e)	O(n)	O(en)	O(e+n)
39	For a network with n vertices and e edges the asymptotic computing time of the topological ordering algorithm is _________	linear	constant	quadratic	exponential	linear
40	A directed graph in which the vertecies represemt tasks or activities and edges represent precedence relation between tasks is ___________	activity on vertex network	topological network	complete network	precedence relation network	activity on vertex network
41	All connected graphs with n-1 edges are called ___	digraphs	cyclic	trees	subgraphs	trees
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.	Adjacency list	Inverse adjacency list	Adjacency multilist	adjacency matrix	Adjacency list
43	Introducing any one edge into a spanning tree will result in a _____	cycle	component	digraph	tree	cycle
44	The cost of spanning tree is the ______cost of the edges in that tree	maximum	average	sum	minimum	sum
45	_________assigned to the edges of a graph are called its cost or length of the link.	weights	size	depth	degree	weights
46	All algorithms using adjacency matrix will require atleast ______time.	$O(n^2)$	O(n)	$O(n^3)$	O(2n)	$O(n^2)$
47	The fields used to distinguish among the records are known as ________.	records	address	pointers	keys	keys
48	In ________search method the search begins by examining the record in the middle of the file.	sequential	fibonacci	binary	non-sequential	binary
49	_________search involves only addition and subtraction.	Binary	fibonacci	sequential	non sequential	fibonacci

50	Kruskal formulated a method to determine _________in graphs.	Breadth first search	connected component	adjacency matrix	minimum cost spanning tree	minimum cost spanning tree
51	_______sort is done in graphs	Merge sort	Heap	Topological sort	Linear sort	Topological sort
52	An __________requires that the collection of data fit entirely in the computer's main memory.	Internal sort	external sort	sorting	searching	Internal sort
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.	Internal sort	external sort	sorting	searching	external sort
54	___________are a specific type of Algorithms that specialize in taking in multiple sorted lists and merging them into a single sorted list.	Multiway	Merges	Multiway Merges	Dynamic Merges	Multiway Merges
55	A _________data structure, the size of the structure is fixed.	static	dynamic	non terminal	non sequential	static
56	A ________sort is one in which successive elements are selected.	insertion	deletion	Selection	shell	Selection
57	The straight selection sort is also known as ___________	push-down sort	selection	shell sort	shell	push-down sort
58	An ________is one that sorts a set of records by inserting records into an existing sorted file.	down sort	insertion sort	selection	shell	insertion sort
59	The ________sorts separate subfilesof the original file.	down sort	insertion sort	shell sort	shell	shell sort
60	A table of records in which a key is used for retrieval is called a___________.	insert table	delete table	records	search table	search table

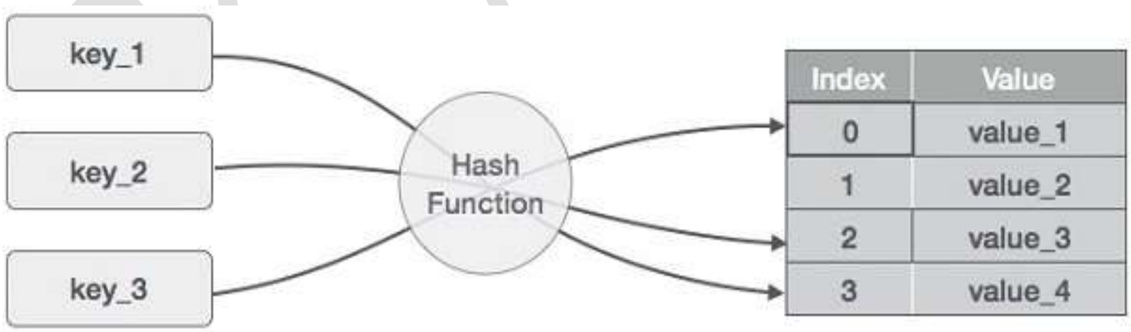
UNIT-V

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.

<u>RESOLVING COLLUSION</u> :

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

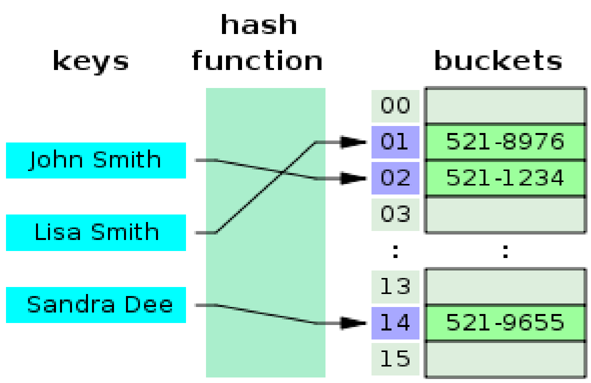
Hash function: key mod hash size

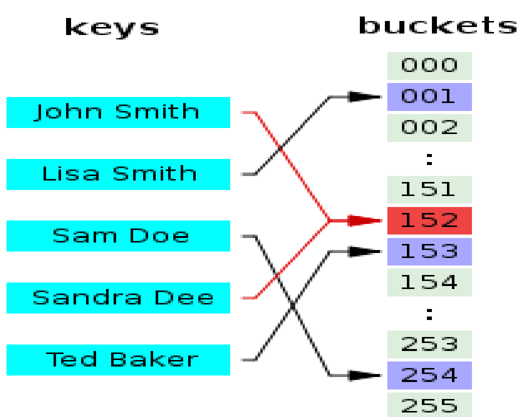
So, the new positions in the hash table are:

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):



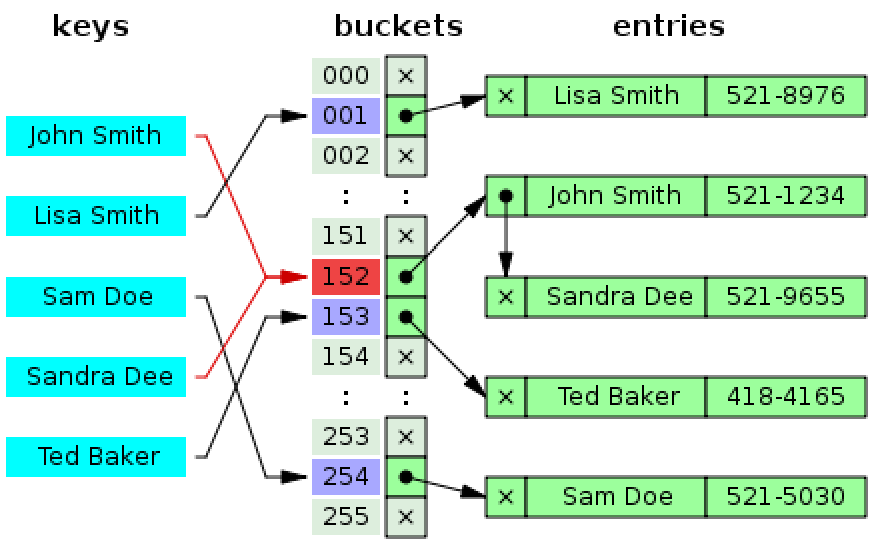


Here, the buckets for keys 'John Smith' and 'Sandra Dee' are the same. So, its a collision case.

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)

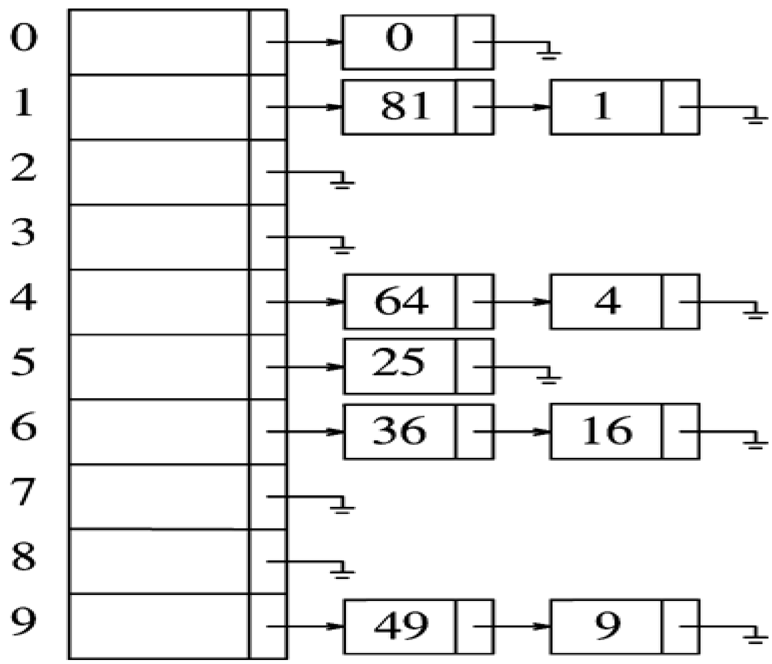
Hash table T is a vector of linked lists

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)

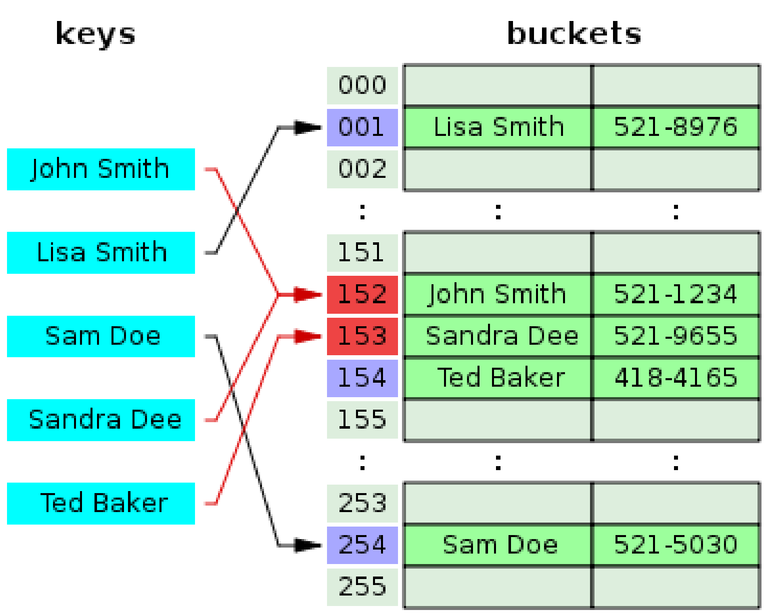
- Ideally, want λ ≤ 1 (not a function of N)

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

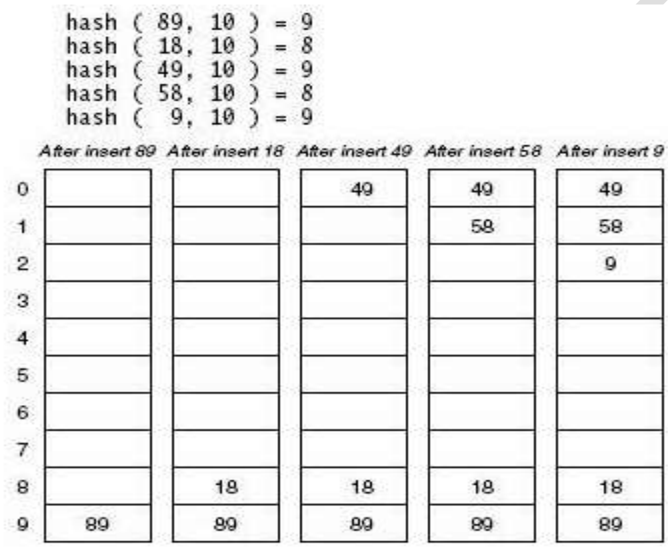
The next slot for the collided key is found in this method by using a technique called **"Probing".** It generates a probe sequence of slots in the hash table and we need to chose the proper slot for the key 'x'.

- 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



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.

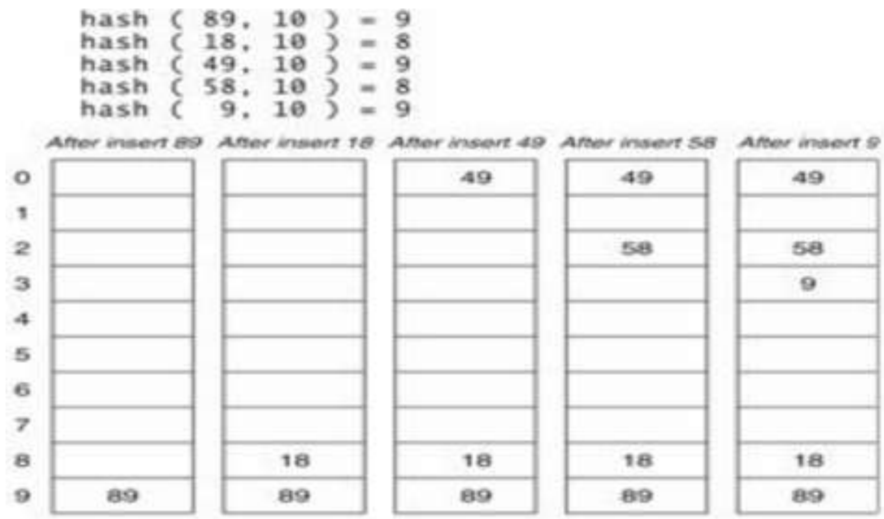
Issues with Linear Probing:

- 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: $f(i) = i^2$
- $h_i(x) = (h(x) + i^2) \bmod tablesize$

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.

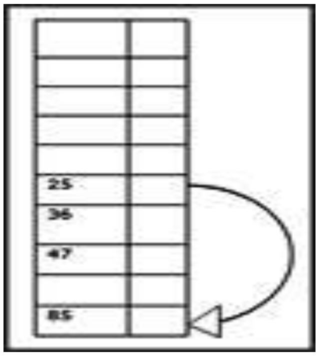
The solution to this problem is called coalesced hashing or coalesced chaining. This method is the hybrid of chaining and open addressing. Each index position in the table stores key value and a pointer to the next index position. The pointer generally points to the index position where the colliding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

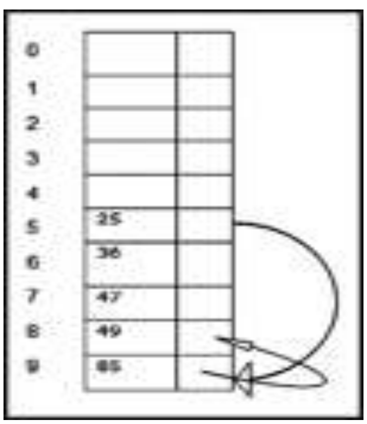
For Example, the values 25, 36, and 47 will be inserted thus in the table –

0		
1		
2		
3		
4		
5	25	
6	36	
7	47	
8		
9		

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)

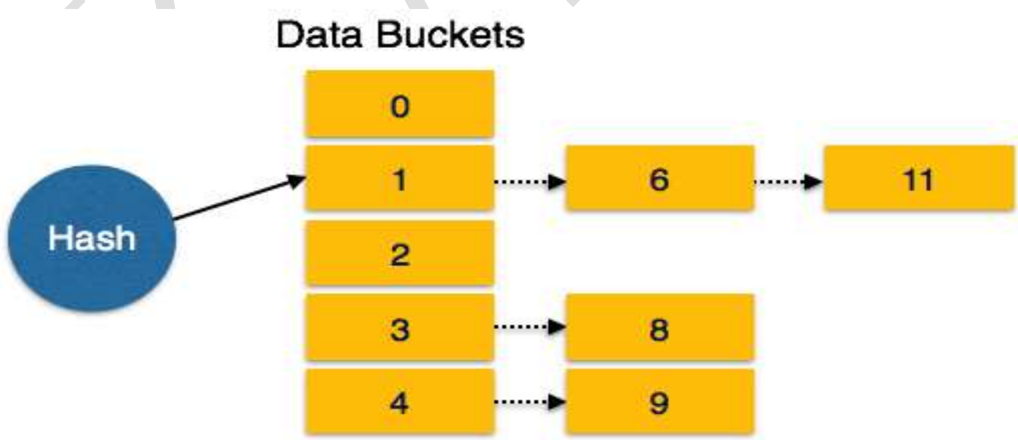
- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

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

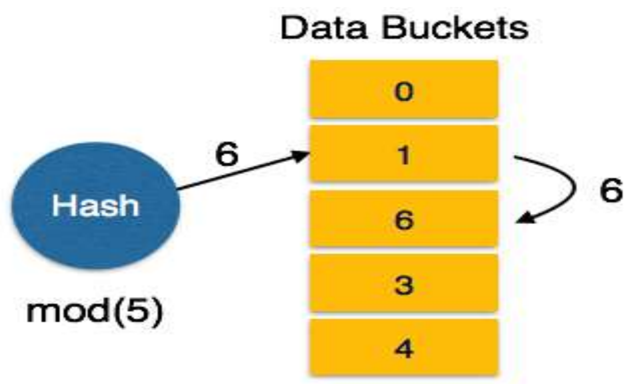
Bucket Overflow

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**.



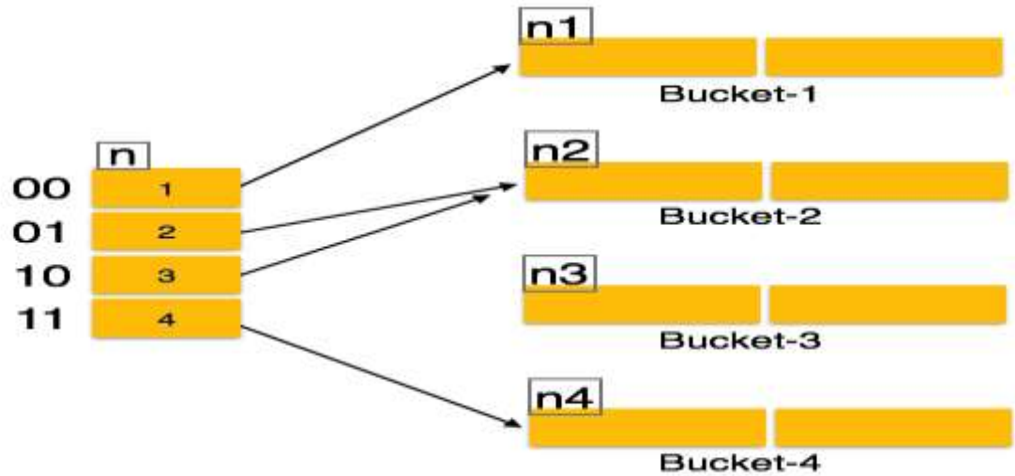
- **Linear Probing** − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



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 Examinations)

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.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore - 641021.

UNIT V :(Objective Type/Multiple choice Questions each Question carries one Mark)

Subject :Data Structures Sub Code:18CTU301

PART-A (Online Examination)

SNo	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4			ANSWER
1	____________ is a collection of records where each record consists of one or more fields.	file	record	storage	data			**file**
2	____________ may be retrieved by specifying values for some or all of these keys.	Update	Records	query	file			**Records**
3	File organization is concerned with representing data records on_____________________	Internal storage media	storage media	External storage	Records			**External storage media**
4	One of the important components of a file is the ___________	key	indexes	Records	Directory			**Directory**
5	A directory is a collection of ___________	records	indexes	query	storage			**indexes**
6	An index is the collection of key values and _____________ pairs	one	sequential	address	merging			**address**
7	A ___________ ia a collection of index.	Directory	Record	tables	files			**Directory**
8	____________are localized to lie within a cell.	Lists	Records	index	tables			**Lists**
9	__________files are similar to multilists.	Ordered	Inverted	index	linked			**Inverted**
10	______________are maintained as for multilists.	Fields	Records	Indexes	tables			**Indexes**

11	a hash that grows to handle more items is called.	optimal hashing	dynamic hashing	minimal perfect	none of thes			**dynamic hashing**
12	a hash table in which the hash function is the last few bits of the key and the table refers to buckets is called.	optimal hashing	dynamic hashing	minimal perfect	extendible hashing			extendible hashing
13	a dynamic hashing table that grows one slot at a time is called.	linear hashing	dynamic hashing	minimal perfect	extendible hashing			linear hashing
14	linear hashing is also known as.	decremental hashing	incremental hashing	extendible hashing	none of these			incremental hashing
15	a dictionary implemented with two hassh tables of equl size,T1and T2,and two different hash functions,h1and h2.	optimal hashing	dynamic hashing	minimal perfect	2-left hashing.			2-left hashing
16	a variant of a hah table in which keys are added by hashing with two hash functions is called	optimal hashing	2-choice hashing	minimalper fect	2-left hashing.			2-choice hashing
17	the average-case cost of a successful search is for 2-choice hashing where m is the number of keys and n is the size of the array.	o(2+(m-1)/n)	o(1+(m-1)/n)	o(3+(m-1)/n)	none of these			o(2+(m-1)/n)
18	a conceptual method of open addressing for a hah table is called	universal hashing	uniform hashing	optimal hashing	none of these			uniform hashing
19	the assumption or goal that items are equally likely to hash to any value is called.	simple uniform hashing	uniform hashing	optimal hashing	nonr of these			simple uniform
20	a method of open addressing for a hash table in which a collision is resolved by searching the table for an empty place at intervals given	double hashing	uniformhashing	optimalhas hing	none of these			double hashing
21	a scheme that choose randomly from a set of hash function is called	double hashing	uniform hashing	optimal hashing	universal hashing			universal hashing
22	a dictionary in which keys are mapped to array position by hash function is called.	both(a)and (b)	view table	and (b)	none of these			both(a)and (b)
23	a hash funcion that maps each different key to a distinct integer is called	perfect hashing	internal hashing	external hashing	none of these			perfect hashing
24	a hash table that uses a perfect hash has	1 collision	no collision	multipel collision	none of these			no collision
25	perfect hashing is also called	optimal hashing	dynam8ic hashing	cuckoo hashing	none of these			optimal hashing

26	a perfect hashing funciton that maps each different key to a distinct integer ans hs the same number of possible integers as keys is	optimal hashing	dynamic hashing	minimalper fect	none of these			minimal perfect
27	which of the following are not hash function.	division	coupling	folding	exteraction			folding
28	is a collision resolution technique that puts all elements that hash to the same slot in a linked list	chaining	open addressing	closed addressing	hashing			chaining
29	the collision resolution techniques and methods used in closed addressing include(1)linked list,(2)binart tree,(3)overflow block	(1)and(2)only	(1)only	(2)only	(1),(2),(3)			(1)and(2)only
30	suppose you place m items in a hash table with an array size of s.what is the correct formula for the load factor?	s+m	m-s	m*s	m/s			m/s
31	what is the best definition of a collision in a hash table.	two entries are indentical except	two entries with different	two entires with	two entries with the			two entries with
32	_____ symbol table allows insertion and deletion of names.	Hashed	Sorted	Static	Dynamic			**Dynamic**
33	The arithmetic functions used for Hashing is called _______	Logical operations	Rehashing	Mapping function	Hashing function			**Hashing function**
34	Each bucket of Hash table is said to have several _______	slots	nodes	fields	links			**slots**
35	A________ occurs when two non_identical identifiers are hashed in the same bucket.	collision	contraction	expansion	Extraction			**collision**
36	A hashing function f transforms an identifier x into a ___________ in the hash table	symbol name	bucket address	link field	slot number			**bucket address**
37	When a new identifier I is mapped or hashed by the function f into a full bucket then _______occurs	underflow	overflow	collision	rehashing			**overflow**
38	If f(I) and F(J) are equal then Identifiers I and J are called_________	synonyms	antonyms	hash functions	buckets			**synonyms**
39	In hash table, if the identifier x has an equal chance of hashing into any of the buckets, this function is called as	Equal hash function	uniform hash function	Linear hashing	unequal Hashing			**uniform hash**
40	Each head node is smaller than the other nodes because it has to retain	only a link	only a link and a record	only two link	only the record			**only a link**

41	Each chain in the hash tables will have a	tail node	link node	head node	null node			**head node**
42	Folding of identifiers fron end to end to obtain a hashing function is called ____	Shift folding	boundary folding	expanded folding	end to end folding			**boundary folding**
43	Average number of probes needed for searching can be obtained by ------------------- probing	quadratic	linear	rehashing	Sequential			**quadratic**
44	Rehashing is ______	series of hash function	linear probing	quadratic functions	Rebuild function			**series of hash**
45	_________ is a method of overflows handling.	linear open addressing	Adjacency lsit	sequential representat	Indexed address			**linear open addressing**
46	The number of _________ over the data can be reduced by using a higer order merge (k-way merge with k>2)	records	passes	tapes	merges			**passes**
47	A __________ is a binary tree where each node represents the smaller of its two children	search tree	decision tree	extended tree	selection tree			**selection tree**

KARPAGAM ACADEMY FOR HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2018 onwards)
B.Sc DEGREE EXAMINATION
Third Semester
COMPUTER TECHNOLOGY
DATA STRUCTURES

Class: II BSc (CT) **Maximum : 50**
MarksSub.Code:18CTU301 **Duration: 2 Hours**
Date:

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

1. ____sequence of instructions to accomplish particular task.
   a) Data structure b) Data c) Algorithm d) Stack
2. A _______ is a specialized format for organizing and storing data.
   a) Data b) Data structure c) Data item d) entity
3. The term ______ simply refers to a value or set of values.
   a) Entity b) Data c) Data item d) Record
4. A ______is a collection of related data items.
   a) Data b) Entity c) Record d) None
5. An ______ is a list of finite number of elements of same data type.
   a) Array b) Linear array c) Non-linear d) List
6. A linked list is a linear collection of data elements called ______
   a) Pointer b) List c) Data d) Nodes
7. ________ list allows traversing in only one direction.
   a)Singly linked list b) Doubly linked list c)Circular Linked List d) Ordered List
8. In singly linked list ,each node has _________field.
   a) One b) Two c) Three d) Five
9. In Doubly linked list ,each node has _________field.
   a) One b) Two c) Three d) Five
10. A stack also called a ----- system.
    a) LIFO b) FIFO c) LILO d) FILO
11. Each item in a node is called a_______.
    a) Field b) Data item c) Pointer d) Data
12. In stack the TOP decremented one by one is called______.
    a) Push b) Pop c) Add d) Delete
13. Which of the following is not structured data type?
    a) Union b) Array c) Pointer d) Queue
14. Forward pointer for each node will be referred as _________.
    a) Forward b) Flink c) Front d) Data
15. A linked list is a linear collection of data elements called -----
    a) Pointer b) List c) Data d) Nodes

16. _________ list allows traversing in only one direction.

a)Singly linked list   b) Doubly linked list   c)Circular Linked List   d) Ordered List

17. In singly linked list, each node has __________field.

a) One   b) Two   c) Three   d) Five

18. In Doubly linked list, each node has __________field.

a) One   b) Two   c) Three   d) Five

19. Convert Infix to Postfix A+B*C

a) AB+C*   b) ABC+*   c) A+B*C   d) AB+*C

20. Convert Infix to Prefix (A+B)/(C+D)

a) +AB/+CD   b) ++/ABC   c) /+AB+CD   d) +/+ABCD

PART-B(3 * 2= 6 Marks)
(Answer all the Questions)

21. What is sparse matrix?

22. List different applications of data structures.

23. Define Priority Queue with diagram

PART-C (3 *8=24 Marks)
(Answer all the Questions)

26. a) Define Stack .Explain different operations on stacks.

(or)

b) Write about the single dimensional and multi-dimensional array with example

27. a) Discuss about Singly Linked List and its operations.

(or)

b) Discuss about Doubly Linked List and its operations.

28. a) Explain array implementation of queue with an example.

(or)

b) Discuss circular and its operations

KARPAGAM ACADEMY FOR HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2018 onwards)
B.Sc DEGREE EXAMINATION
Third Semester
COMPUTER TECHNOLOGY
DATA STRUCTURES
Answer Key

Class: II BSc (CT) **Maximum: 50**
MarksSub.Code:18CTU301 **Duration: 2 Hours**
Date:

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

1. ____sequence of instructions to accomplish particular task.
   a) Data structure b) Data c) Algorithm d) Stack
2. A _______ is a specialized format for organizing and storing data.
   a) Data b) Data structure c) Data item d) entity
3. The term ______ simply refers to a value or set of values.
   a) Entity b) Data c) Data item d) Record
4. A ______is a collection of related data items.
   a) Data b) Entity c) Record d) None
5. An ______ is a list of finite number of elements of same data type.
   a) Array b) Linear array c) Non-linear d) List
6. A linked list is a linear collection of data elements called ______
   a) Pointer b) List c) Data d) Nodes
7. ________ list allows traversing in only one direction.
   a)Singly linked list b) Doubly linked list c)Circular Linked List d) Ordered List
8. In singly linked list ,each node has _________field.
   a) One b) Two c) Three d) Five
9. In Doubly linked list ,each node has _________field.
   a) One b) Two c) Three d) Five
10. A stack also called a ----- system.
    a) LIFO b) FIFO c) LILO d) FILO
11. Each item in a node is called a_______.
    a) Field b) Data item c) Pointer d) Data
12. In stack the TOP decremented one by one is called______.
    a) Push b) Pop c) Add d) Delete
13. Which of the following is not structured data type?
    a) Union b) Array c) Pointer d) Queue
14. Forward pointer for each node will be referred as _________.
    a) Forward b) Flink c) Front d) Data
15. A linked list is a linear collection of data elements called -----

a) Pointer b) List c) Data d) Nodes

16. _________ list allows traversing in only one direction.

a)Singly linked list b) Doubly linked list c)Circular Linked List d) Ordered List

17. In singly linked list, each node has __________field.

a) One b) Two c) Three d) Five

18. In Doubly linked list, each node has __________field.

a) One b) Two c) Three d) Five

19. Convert Infix to Postfix A+B*C

a) AB+C* b) ABC+* c) A+B*C d) AB+*C

20. Convert Infix to Prefix (A+B)/(C+D)

a) +AB/+CD b) ++/ABC c) /+AB+CD d) +/+ABCD

PART-B(3 * 2= 6 Marks)
(Answer all the Questions)

21. What is sparse matrix?

In a matrix most of the elements are non-zero

22. List different applications of data structures.

23. Define Priority Queue with diagram

In normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle.

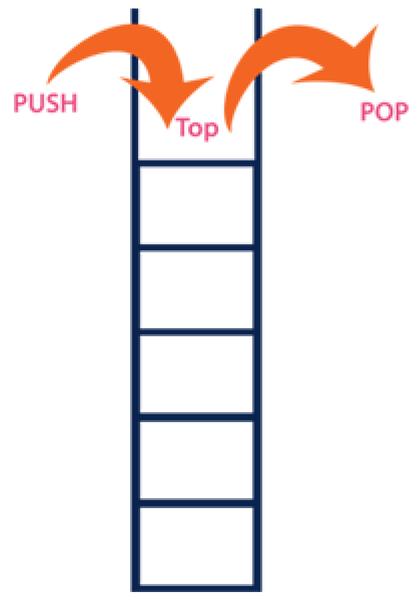
PART-C (3 *8=24 Marks)
(Answer all the Questions)

26. a) Define Stack .Explain different operations on stacks

5. STACKS IMPLEMENTING SINGLE / MULTIPLE STACK/S IN AN ARRAY

STACK:

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

Stack is a linear data structure in which the operations are performed based on LIFO principle.

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

The following operations are performed on the stack...

1. Push (To insert an element on to the stack)
2. Pop (To delete an element from the stack)
3. Display (To display elements of the stack)

Stack data structure can be implement in two ways. They are as follows...

1. Using Array
2. Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

PUSH(Stack, Top, MaxStk, Item)

1. If Top == MaxStk //check Stack already fill or not
   then print "Overflow" and return
2. Set Top = Top + 1 //increase top by 1
3. Stack[Top] = Item //insert item in new top position
4. Return

Pop operation on Stack

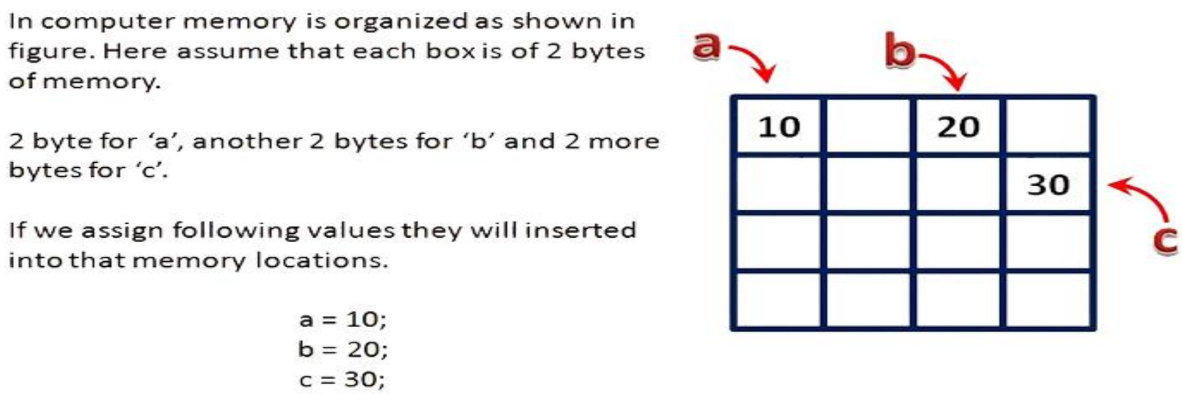
POP(Stack, top, Item)
1. If Top == Null //check Stack top element to be deleted is empty
   then print "Underflow" and return
2. Item = Stack[Top] //assign top element to item
3. Set Top = Top - 1 //decrease top by 1
4. Return

(or)

b) Write about the single dimensional and multi-dimensional array with example

An array is a variable which can store multiple values of same data type at a time.

"Collection of similar data items stored in continuous memory locations with single name".



<u>One-Dimensional Arrays:</u>

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

Syntax :type variable_name[size]

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

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

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

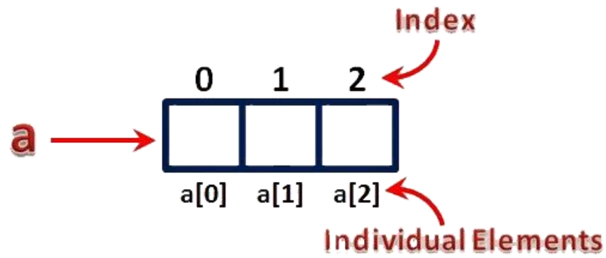
Examples:

int sample[10];

floatfloat_numbers[100];

charlast_name[40];

Now consider the following declaration...



Algorithm

1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

3. MULTIDIMENSIONAL ARRAYS:

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

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

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.

Example : int matrix[3][3];

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

27. a) Discuss about Singly Linked List and its operations

LINKED LIST:

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

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.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

(or)

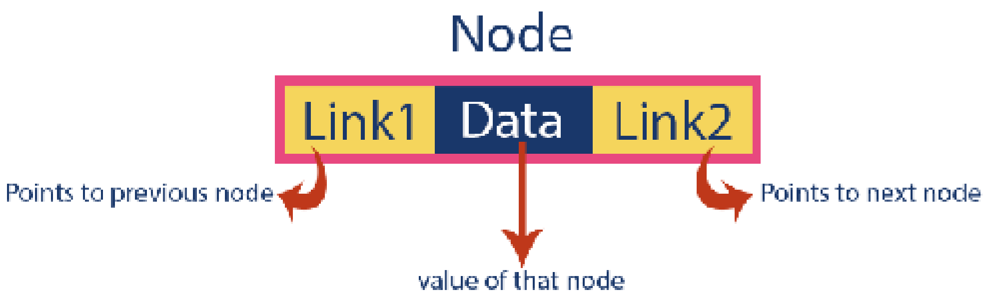
b) Discuss about Doubly Linked List and its operations

DOUBLY LINKED LIST:

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

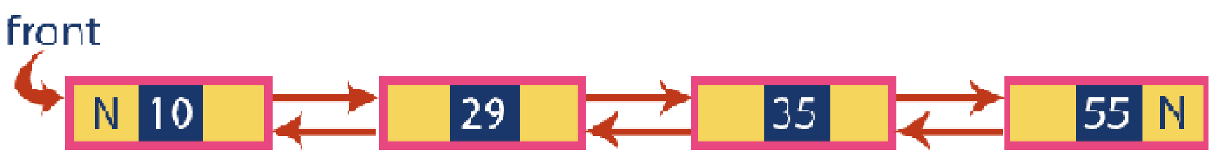
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



- In double linked list, the first node must be always pointed by **head**.
- Always the previous field of the first node must be **NULL**. Always the next field of the last node must be **NULL**.
- 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.

<u>IMPLEMENTING LISTS USING ARRAYS</u>:

Arrays are suitable for:

– Inserting/deleting an element at the end.

– Randomly accessing any element.

– Searching the list for a particular value.

Array representation of linked list

An array of linked list is an unique structure which combines a static structure (an array) and a dynamic structure (linked lists) to form a useful data structure. This type of data structure is useful for applications. Like, When you know the categories under a menu but have no idea about the sub-categories under categories. For instance, we can use an array of linked lists, where each list contains words starting with a specific letter in the alphabet. When you declare all variable, then

```
node* A[n] ;    // defines an array of n node pointers
for (i=0; i<n; i++)  A[i] = NULL;  // initializes the array to NULL
```

Now, to insert more nodes into the list.

A call to the function can be as follows.

```
node* ptr = (node*)malloc(sizeof(node));
ptr → size = 10;
ptr → name = (char*) malloc(strlen(“neha”+1));
strcpy(ptr → name, “neha\0”);
ptr → next = NULL;
insertnodes(&A, ptr, 3);   // insert node ptr to array location 3.
```

<u>Implementing Lists using Linked List</u>:

Linked lists are suitable for:

Inserting an element.

Deleting an element.

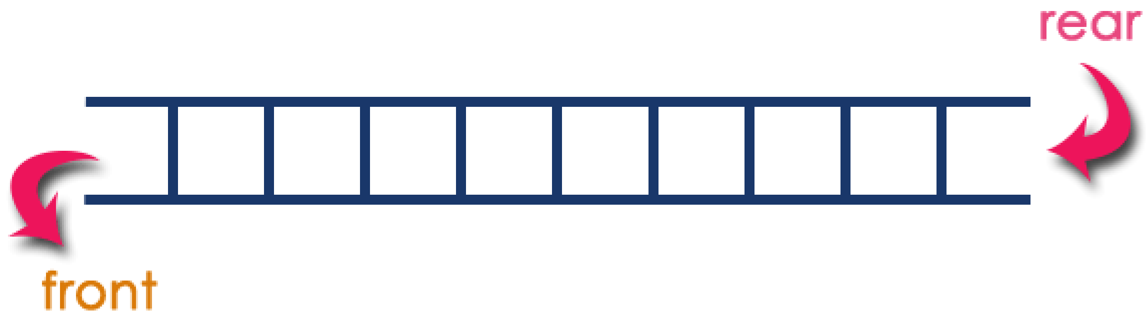
Applications where sequential access is required. In situations where the number of elements cannot be predicted beforehand.

28. a) Explain array implementation of queue with an example.

<u>**QUEUES**</u>

What is a Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".

Queue data structure can be defined as follows...

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

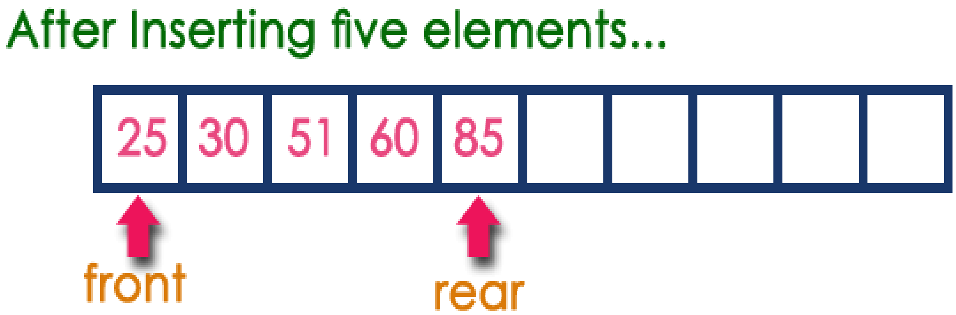
A queue can also be defined as

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Example

Queue after inserting 25, 30, 51, 60 and 85.

Operations on a



Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

ARRAY AND LINKED REPRESENTATION OF QUEUE

Queue Using Array

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)**

- **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

- **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

- **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

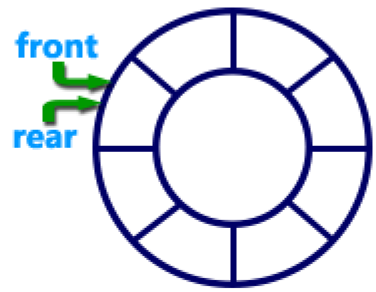
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 3:** Display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

(or)

b) Discuss circular and its operations

A Circular Queue can be defined as follows

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.



<u>DOUBLE ENDED QUEUE (DEQUEUE)</u>

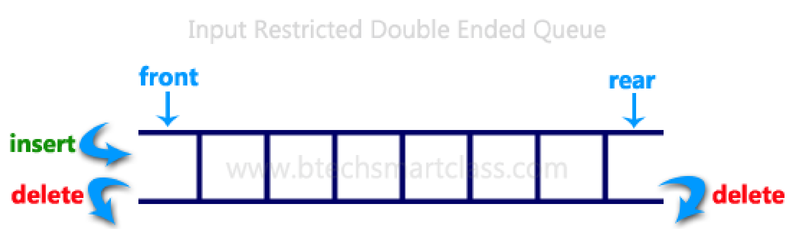
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front**and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

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

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

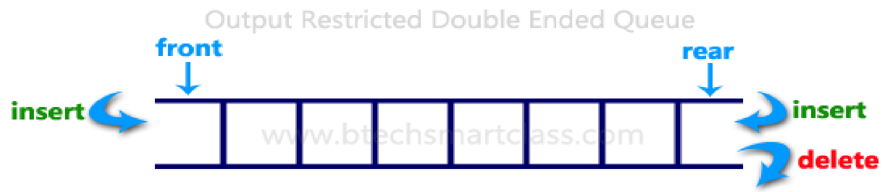
Input Restricted Double Ended Queue

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.

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.

KARPAGAM ACADEMY FOR HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore – 641021

SECOND INTERNAL EXAMINATION, AUGUST 2019
COMPUTER TECHNOLOGY
Third Semester
DATA STRUCTURES

Class: II BSc (CT) **Maximum: 50 Marks**
Sub.Code:18CTU301 **Duration: 2 Hours**
Date: 28.08.19 & FN

PART-A (20 * 1 = 20 Marks)
(Answer all the Questions)

1. ________is the process of collecting all used nodes and returning them to the available space

   a) compaction b) coalescing c) garbage d) deallocation

2. A________is a data structures that is used for storing a sorted list of items

   a) skip list b) self-organizing list c) self-list d) node list

3. Height balanced tree al so called _____

   a) skip list b) self-organizing list c) self-list d) node list

4.____is a subset of a tree that is itself a tree

   a) branch b) root c) leaf d) sub tree

5. In binary trees there is no node with a degree greater than _____

   a) AVL b) binary tee c) sub tree d) branch tree

6. The number of sub tree of a node is called its _______

   a) leaf b) terminal c) node d) degree

7. Which of the following non-linear data structures

   a) stack b) queue c) tree d) list

8. All nodes except the leaf nodes are called________

   a) terminal node b) terminal c) node d) non-terminal

9. The children of the same parent are called __________________.

   a) sibling b) leaf c) child d) sub tree

10. Height – balanced trees are also referred as ___________trees

    a) binary trees b) AVL tree c) sub tree d) thread tree

11. The operation of processing each element

    a) Traversing b) inserting c) deleting d) thread tree

12. Binary tree with threshed is called____

    a) tree b) threaded c) forest d) pointer tree

13. The post order traversal of a binary tree is DEBFCA: Convert pre-order

a)ABFDEC b) ADBFEC c) ABEECF d) ABECEF

14. In ______traverse , the node is visited between a sub trees

a) preorder b) postorder c) inorder d) endorder

15. Nodes that have degree zero are called _________.

a)root b) subtree c) leaf node d) binary tree

16. In search start at beginning of the list and check every element in list

a)hash search b) linear search c) binary search d) binary tree

17. A ____is a finite set of one or more nodes, with one root node and remaining form the disjoint sets forming the sub trees

a) list b) Graph c) tree d) set

18. The ______of path is the number of edges on it

a) degree b) length c) edges d) height

19. Visiting each node in a tree exactly once is called __________

a) searching b) traversing c) Walk through d)path

20.______ is the process of finding the location of given element in the linear array.

a) linear b) sorting c) searching d) binary

PART-B (3 * 2= 6 Marks)
(Answer all the Questions)

21. What is binary search tree?
22. List different applications of data structures.
23. Define Priority Queue with diagram

PART-C (3 *8=24 Marks)
(Answer all the Questions)

24. a) What is binary search tree? Convert the following post order and pre order tree traversal

10, 25, 36,28,19,36

(OR)

b) Explain Threaded binary tree and its types with an algorithm

25. a) Discuss AVL tree in details

(OR)

b) Write short note on i) self – organizing list ii) skip list

26. a) Explain array implementation of queue with an example.

(OR)

b) Write about iterative, traverse operations on binary search tree

KARPAGAM ACADEMY FOR HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore – 641021
SECOND INTERNAL EXAMINATION, AUGUST 2019
Third Semester
COMPUTER TECHNOLOGY
DATA STRUCTURES

Class: II BSc (CT) **Maximum: 50 Marks**
Sub.Code:18CTU301 **Duration: 2 Hours**
Date: Date: 28.08.19 & FN

PART-A (20 * 1 = 20 Marks)
(Answer all the Questions)

1. ________is the process of collecting all used nodes and returning them to the available space

    a) compaction b) coalescing **c) garbage** d) deallocation

2. A________is a data structures that is used for storing a sorted list of items

    a**) skip list** b) self-organizing list c) self-list d) node list

3. Height balanced tree al so called _____

    a) skip list b) self-organizing list c) self-list **d) AVL tree**

4.____is a subset of a tree that is itself a tree

    a) branch b) root c) leaf **d) sub tree**

5. In binary trees there is no node with a degree greater than _____

    a) AVL b) binary tee **c) sub tree** d) branch tree

6. The number of sub tree of a node is called its _______

    a) leaf b) terminal c) node **d) degree**

7. Which of the following non-linear data structures

    a) stack b) queue **c) tree** d) list

8. All nodes except the leaf nodes are called________

    a) terminal node b) terminal c) node **d) non-terminal**

9. The children of the same parent are called ___________________.

    a) sibling b) leaf c) child d) subtree

10. Height – balanced trees are also referred as as ___________trees

    a) binary trees **b) AVL tree** c) subtree d) thread tree

11. The operation of processing each element

    a) Traversing b) inserting c) deleting d) thread tree

12. Binary tree with threshed is called____

    a) tree **b) threaded** c) forest d) pointer tree

13. The post order traversal of a binary tree is DEBFCA: Convert pre-order

a)ABFDEC b) ADBFEC **c) ABEECF** d) ABECEF

14. In **_____**travese , the node is visted between a sub trees

a)preorder b) postorder **c) inorder** d) endorder

15. Nodes that have degree zero are called **________**.

a)root b) subtree **c) leaf node** d) binary tree

16. In search start at beginning of the list and check every element in list

a)hash search **b) linear search** c) binary search d) binary tree

17. A **__** is a finite set of one or more nodes, with one root node and remaining form the disjoint sets forming the subtrees

a)list b) Graph **c) tree** d) set

18. The **_____**of path is the number of edges on it

a) degree b) length c) edges d) height

19. Visiting each node in a tree exactly once is called **_________**

a) searching **b) traversing** c) Walk through d)path

20.**______** is the process of finding the location of given element in the linear array.

a) linear b) sorting **c) searching** d) binary

PART-B(3 * 2= 6 Marks)
(Answer all the Questions)

21. What is binary search tree?

Binary Search Tree is a node-based **binary tree** data structure which has the following properties: The left subtree of a node contains only nodes with keys lesser than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key.

22. List different applications of data structures.

- **Applications of linked list data structure**
- Implementation of stacks and queues.
- Implementation of graphs : Adjacency **list** representation of graphs is most popular which is **uses linked list** to store adjacent vertices.
- Dynamic memory allocation : We use **linked list** of free blocks.
- Maintaining directory of names.
- Performing arithmetic operations on long integers.

23. Define Priority Queue with diagram

In computer science, a **priority queue** is an abstract data type which is like a regular **queue** or stack data structure, but where additionally each element has a "**priority**" associated with it. In a **priority queue**, an element with high **priority** is served before an element with low **priority**.

PART-C (3 *8=24 Marks)
(Answer all the Questions)

26. a) What is binary search tree? Convert the following post order and pre order tree traversal

10, 25, 36,28,19, *Evaluating expressions*

A stack is used in two phases of evaluating an expression such as

3 * 2 + 4 * (A + B)

•Convert the infix form to postfix using a stack to store operators and then pop them in correct order of precedence.
•Evaluate the postfix expression by using a stack to store operands and then pop them when an operator is reached.

Infix to postfix conversion
Scan through an expression, getting one token at a time.

1 Fix a priority level for each operator. For example, from high to low:

3. - (unary negation)
2. * /
1. + - (subtraction)

Thus, high priority corresponds to high number in the table.

2 If the token is an operand, do not stack it. Pass it to the output.

3 If token is an operator or parenthesis, do the following:

-- Pop the stack until you find a symbol of lower priority number than the current one. An incoming left parenthesis will be considered to have higher priority than any other symbol. A left parenthesis on the stack will not be removed unless an incoming right parenthesis is found.
The popped stack elements will be written to output.

Example:

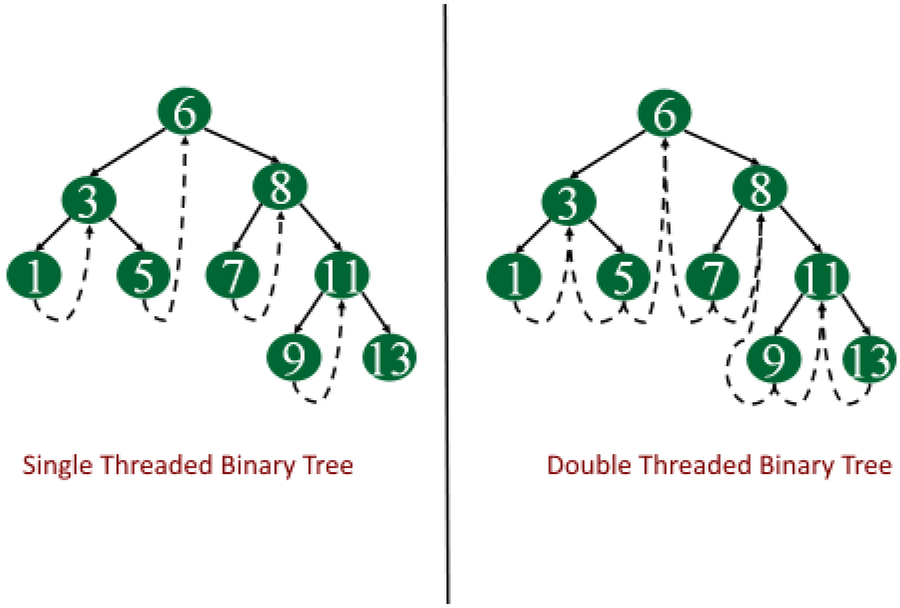
Convert A * (B + C) * D to postfix notation.

Move	Curren Ttoken	Stack	Output
1	A	empty	A

2	*	*	A
3	(	(*	A
4	B	(*	A B
5	+	+(*	A B
6	C	+(*	A B C
7	)	*	A B C +
8	*	*	A B C + *
9	D	*	A B C + * D
10		empty	

(OR)

b) Explain Threaded binary tree and its types with an algorithm



27. a) Discuss AVL tree in details

AVL Tree Datastructure

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

(OR)

b) Write short note on i) Self – organizing list ii) Skip 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**.

The **skip list** is a probabilisitc data structure that is built upon the general idea of a linked **list**. The **skip list** uses probability to build subsequent layers of linked **lists** upon an original linked **list**. Each additional layer of links contains fewer elements, but no new elements.

28. a) Explain array implementation of queue with an example.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
  return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
  return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

  if rear equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
  if(rear == MAXSIZE - 1)
    return true;
  else
    return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif
```

```
end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

```
bool isempty() {
  if(front < 0 || front > rear)
     return true;
  else
     return false;
}
```

(OR)

b) Write about iterative, traverse operations on binary search tree

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.

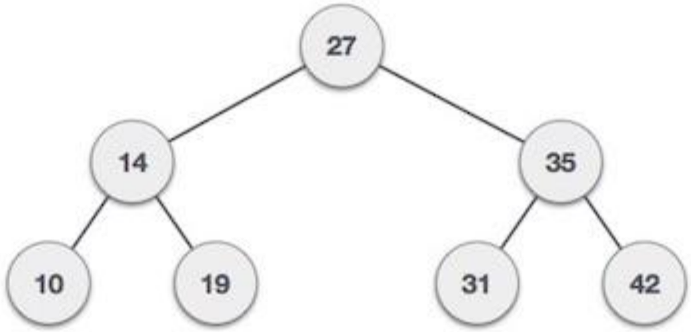
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

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 −



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.

KARPAGAM ACADEMY FOR HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore – 641021

THIRD INTERNAL EXAMINATION, OCTOBER 2019
INFORMATION TECHNOLOGY
Third Semester
DATA STRUCTURES

Class: II BSc (CT) **Maximum: 50 Marks**
Sub.Code:18CTU301 **Duration: 2 Hours**
Date: 14.10.19 & FN

PART-A (20 * 1 = 20 Marks)
(Answer all the Questions)

1. ______ is the process of finding the location of given element in the linear array
   a) Linear b) Sorting c) Searching d) Binary

2.________ search is used to find location of the given element
   a)Searching b)Sorting c)Linear d)Binary

3._______ techniques are used for sorting large files
   a) Topological sort b) External sorting c) Linear Sorting d) Heap sort

4. Which is diminishing increment sort?
   a) Selection sort b) Bubble sort c) Insertion sort d) Shell sort

5. All algorithms using adjacency matrix will require at least _____ time
   a) $O(n^2)$ b)O(n) c)$O(n^3)$ d) O(2n)

6. The linear search is also called ______
   a) Binery b)Sequential c)Unary d) Random

7. The fields used to distinguish among the records are known as ________
   a) Records b) Address c) Pointers d) Keys

8. Each name in the symbol tales is associated with an___
   a) Name value pairs b) Element c) Attribute d) Object

9. Each bucket of Hash table is said to have several ______.
   a) Slots b) Nodes c) Fields d) Links

10. Rehashing is ______
    a) Series of hash b) Linear probing c) Quadratic probing d) Rebuild

11. _____ symbol table allows insertion and deletion of names
    a) Hashed b) Sorted c) Static d) Dynamic

12. A_______ occurs when two non_identical identifiers are hashed in the same bucket.
    a) Expansion b) Contraction c) Collision d) Extraction

13. If the names are _________ in the symbol table, searching is easy

a) Sorted   b) Short   c) Bold   d) Upper case

14. A search in a hash table with n identifiers may take ________ time.

a) O(2)   b) O(1)   c) O(n)   d)O(2n)

15. A________ occurs when two non-identical identifiers are hashed in the same bucket

a) Collision   b) Contraction   c) Expansion   d) Extraction

16. __________ is a method of overflows handling

a) Linear open addressing   b) Adjacency list   c) Sequential representation   d) Indexed

17. A general method for resolving hash clashes called ______

a) Hash collision   b) Rehashing   c) Hashing   d) Collision

18. Each chain in the hash tables will have a __________

a) Tail node   b) Link node   c) Head node   d) Null node

19. In hash table, if the identifier x has an equal chance of hashing into any of the buckets, this function is called as

a) Equal hash function   b) Uniform hash function

c) Linear hashing function   d) Unequal Hashing function

20. No clashes occur under a ________ hash function

a) Folding   b) Perfect   c) Midsquare   d) Collision

PART-B (3 * 2= 6 Marks)
(Answer all the Questions)

21. Differentiate internal and external sorting
22. Compare linear search and binary search
23. Define hashing and hash table

PART-C (3 *8=24 Marks)
(Answer all the Questions)

24. a) What is searching? Explain about binary search

(OR)

b) Show how insertion sort and selection sort process the input 24, 12,46,5,20,18,35,10

25.a) Explain Shell sort with suitable example and algorithm

(OR)

b) Discuss about Hashing techniques

26. a) b) Write a note on i) Probing methods ii) Resolving collusion by open addressing

(OR)

b) Explain about Dynamic and Extendible hashing

KARPAGAM ACADEMY FOR HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore – 641021
THIRD INTERNAL EXAMINATION, OCTOBER 2019
COMPUTER TECHNOLOGY
Third Semester
DATA STRUCTURES

Class: II BSc (CT) **Maximum: 50 Marks**
Sub.Code:18CTU301 **Duration: 2 Hours**
Date: 28.08.19 & FN

PART-A (20 * 1 = 20 Marks)
(Answer all the Questions)

1. ______ is the process of finding the location of given element in the linear array
   a) Linear b) Sorting **c) Searching** d) Binary
2. ________ search is used to find location of the given element
   a)Searching b)Sorting c)Linear **d)Binary**
3. _______ techniques are used for sorting large files
   a) Topological sort **b) External sorting** c) Linear Sorting d) Heap sort
4. Which is diminishing increment sort?
   a) Selection sort b) Bubble sort c) Insertion sort **d) Shell sort**
5. All algorithms using adjacency matrix will require at least _____ time
   a) $O(n^2)$ b)O(n) c)$O(n^3)$ d) O(2n)
6. The linear search is also called ______
   a) Binary **b)Sequential** c)Unary d) Random
7. The fields used to distinguish among the records are known as ________
   a) Records b) Address c) Pointers **d) Keys**
8. Each name in the symbol tales is associated with an___
   a) Name value pairs b) Element **c) Attribute** d) Object
9. Each bucket of Hash table is said to have several ______.
   a) Slots b) Nodes c) Fields d) Links
10. Rehashing is ______
    a) Series of hash b) Linear probing c) Quadratic probing d) Rebuild
11. _____ symbol table allows insertion and deletion of names
    a) Hashed b) Sorted c) Static **d) Dynamic**

12. A_______ occurs when two non_identical identifiers are hashed in the same bucket.

a) Expansion b) Contraction **c) Collision** d) Extraction

13. If the names are ________ in the symbol table, searching is easy

a) Sorted b) Short c) Bold d) Upper case

14. A search in a hash table with n identifiers may take _______ time.

a) O(2) b) O(1) **c) O(n)** d)O(2n)

15. A_______ occurs when two non-identical identifiers are hashed in the same bucket

a) Collision b) Contraction c) Expansion d) Extraction

16. _________ is a method of overflows handling

a) Linear open addressing b) Adjacency list c) Sequential representation d) Indexed

17. A general method for resolving hash clashes called ______

a) Hhash collision **b) Rehashing** c) Hashing d) Collision

18. Each chain in the hash tables will have a _________

a) Tail node b) Link node **c) Head node** d) Null node

19. In hash table, if the identifier x has an equal chance of hashing into any of the buckets, this function is called as

a) Equal hash function **b) Uniform hash function**

c) Linear hashing function d) Unequal Hashing function

20. No clashes occur under a _______ hash function

a) Folding **b) Perfect** c) Midsquare d) Collision

PART-B (3 * 2= 6 Marks)
(Answer all the Questions)

21. Differentiate internal and external sorting

In **internal sorting** all the data to **sort** is stored in memory at all times while **sorting** is in progress. In **external sorting** data is stored outside memory (like on disk) and only loaded into memory in small chunks

22. Compare linear search and binary search

A **linear search** scans one item at a time, without jumping to any item. In contrast, **binary search** cuts down your **search** to half as soon as you find the middle of a sorted list

23. Define hashing and hash table

A **hash table** is a data structure that is used to store keys/value pairs. It uses a **hash function** to compute an index into an array in which an element will be inserted or searched.

PART-C (3 *8=24 Marks)
(Answer all the Questions)

24. a) What is searching? Explain about binary search

Searching means finding or locating some specific element or node within a data structure. However, **searching** for some specific node in **binary search tree** is pretty easy due to the fact that, element in **BST are** stored in a particular order. Compare the element with the root of the **tree**.

(OR)

b) Show how insertion sort and selection sort process the input 24, 12,46,5,20,18,35,10

To **perform an insertion sort**, begin at the left-most element of the array and invoke Insert to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined.

25.a) Explain Shell sort with suitable example and algorithm

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 of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

How 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

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

(OR)

b) Discuss about Hashing techniques

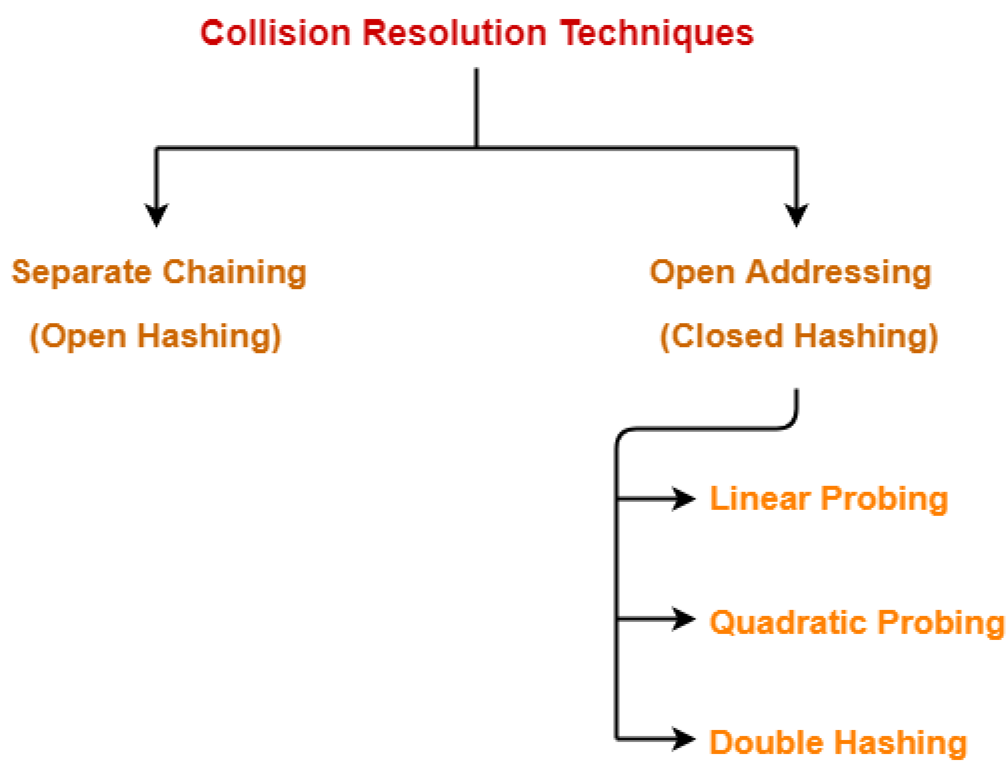
Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

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.

26. a) b) Write a note on i) Probing methods ii) Resolving collusion by open addressing

ave discussed-

- **Hashing** is a well-known searching technique.
- Collision occurs when hash value of the new key maps to an occupied bucket of the hash table.
- Collision resolution techniques are classified as-



(OR)

b) Explain about Dynamic and Extendible hashing

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".
Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.
Open Addressing is done following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.
let **hash(x)** be the slot index computed using hash function and **S** be the table size

```
If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
```