KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore-641 021
(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHMS **SEMESTER : IV**

SUBJECT CODE: 17CSU401 **CLASS: II- B. Sc (CS)**

COURSE OBJECTIVE:

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

COURSE OUTCOME:

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

UNIT-I

Introduction: Basic Design and Analysis techniques of Algorithms, Correctness of Algorithm. **Algorithm Design Techniques:** Iterative techniques, Divide and Conquer, Dynamic Programming, Greedy Algorithms.

UNIT-II

Sorting and Searching Techniques: Elementary sorting techniques–Bubble Sort, Insertion Sort, Merge Sort, Advanced Sorting techniques - Heap Sort, Quick Sort, Sorting in Linear Time - Bucket Sort, Radix Sort and Count Sort, Searching Techniques, Medians & Order Statistics, complexity analysis;

UNIT-III

Lower Bounding Techniques: Decision Trees **Balanced Trees:** Red-Black Trees

UNIT-IV
Advanced Analysis Technique: Amortized analysis **Graphs:** Graph Algorithms–Breadth First Search, Depth First Search and its Applications, Minimum Spanning Trees.

UNIT-V
String Processing: String Matching, KMP Technique.

Suggested Readings

1. Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.
2. Sarabasse., Gelder, A.V. (1999). Computer Algorithm – Introduction to Design and Analysis (3rd ed.). New Delhi: Pearson

ESE MARKS ALLOCATION

1.	**Section A** 20 X1 = 20 (Online Examination)	20
2.	**Section B** 5 x 2 = 10 (Answer all the questions)	10
3.	**Section C** 5 X 6 = 30 (Either ‘A’ or ‘B’ Choice)	30
4.	**Total**	**60**

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021.

LECTURE PLAN
DEPARTMENT OF COMPUTER SCIENCE

STAFF NAME: S.A.SATHYA PRABHA
SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHMS SUB.CODE : 17CSU401
SEMESTER: IV CLASS : II B.SC CS B

Sl.No	**Lecture Duration (Periods)**	**Topics to be covered**	**Support Materials**
UNIT- I			
1	1	Introduction, Basic Design and Analysis techniques of Algorithms	W1
2	1	Correctness of Algorithm	T1:2
3	1	Algorithm Design Techniques: Itérative techniques	W2
4	1	Divide and Conquer	T1: 12-14
5	1	Dynamic Programming	T1: 301-307
6	1	Greedy Algorithms	T1: 329-332
7	1	Recapitulation and Discussion of Possible Questions	
		Total No. Of Hours Planned for unit I	**07**

TEXT BOOK:		**T1:Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.**	
WEB SITES		**W1: https://www.tutorialspoint.com/design_and_analysis_of_algorithm** **W2: https://en.wikipedia.org/wiki/Iterative_method**	
Sl.No	**Lecture Duration (Periods)**	**Topics to be covered**	**Support Materials**
UNIT- II			
1	1	Sorting and Searching Techniques: Elementary sorting techniques- Bubble Sort	W3
2	1	Bubble Sort	W3
3	1	Insertion Sort	T1: 2-5
4	1	Merge Sort	T1: 12-15
5	1	Advanced Sorting techniques – Heap Sort	T1: 140-152
6	1	Quick Sort	T1: 153-163
7	1	Sorting in Linear Time – Bucket Sort	T1: 180-183
8	1	Radix Sort, Count Sort	T1: 175-180
9	1	Searching Techniques- Medians & Order Statistics, complexity analysis	W4, W5
10	1	Contd.. Searching Techniques	W4, W5
11	1	Recapitulation and Discussion of Possible Questions	
		Total No. Of Hours Planned for unit II:	**11**

TEXT BOOK:		**T1:Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.**	
WEB SITES		**W3: https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm** **W4: https://www.w3schools.in/data-structures-tutorial/searching-techniques/** **W5: https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/**	
Sl.No	**Lecture Duration (Periods)**	**Topics to be covered**	**Support Materials**
		UNIT- III	
1	1	Lower Bounding Techniques: Decision Trees	W6
2	1	Contd.. Decision Trees	W6
3	1	Contd.. Decision Trees	W6
4	1	Balanced Trees: Red-Black Trees	T1: 263-265
5	1	Contd.. Red-Black Trees	T1: 265-270
6	1	Contd.. Red-Black Trees	T1: 270-272
7	1	Recapitulation and Discussion of Possible Questions	
		Total No. Of Hours Planned for unit III:	**07**
TEXT BOOK:		**T1:Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.**	

WEB SITES		**W6: https://en.wikipedia.org/wiki/Decision_tree**	
Sl.No	**Lecture Duration (Periods)**	**Topics to be covered**	**Support Materials**
UNIT- IV			
1	1	Advanced Analysis Technique: Amortized analysis	T1: 356-367
2	1	Graphs: Graph Algorithms–Breadth First Search	T1: 465-467
3	1	Contd.. Breadth First Search	T1: 468-477
4	1	Depth First Search	T1: 477-480
5	1	Contd.. Depth First Search	T1: 480-485
6	1	Minimum Spanning Trees	T1: 498-505
7	1	Contd.. Minimum Spanning Trees	T1: 505-512
8	1	Recapitulation and Discussion of Possible Questions	
		Total No. Of Hours Planned for unit III:	**08**
TEXT BOOKS:		**T1:Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.**	
Sl.No	**Lecture Duration (Periods)**	**Topics to be covered**	**Support Materials**
UNIT- V			
1	1	String Processing: String Matching	T1: 853-860
2	1	KMP Technique	T1: 861-875
3	1	Contd.. KMP Technique	T1: 875-883
4	1	Recapitulation and Discussion of Possible Questions	
5	1	Discussion of Previous ESE Question Paper	

6	1	Discussion of Previous ESE Question Paper	
7	1	Discussion of Previous ESE Question Paper	
		Total No. Of Hours Planned for unit V:	**07**
Overall Planned Hours : 40			
TEXT BOOKS:		**T1:Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.**	

<u>TEXT BOOK</u>

1. Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.

2. Sarabasse., Gelder, A.V. (1999). Computer Algorithm – Introduction to Design and Analysis (3rd ed.). New Delhi: Pearson.

WEBSITES

1. https://www.tutorialspoint.com/design_and_analysis_of_algorithm
2. https://en.wikipedia.org/wiki/Iterative_method
3. https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm
4. https://www.w3schools.in/data-structures-tutorial/searching-techniques/
5. https://www.w3schools.in/data-structures-tutorial/searching-techniques/
6. https://en.wikipedia.org/wiki/Decision_tree

UNIT I

SYLLABUS

Introduction: Basic Design and Analysis techniques of Algorithms, Correctness of Algorithm. Algorithm Design Techniques: Iterative techniques, Divide and Conquer, Dynamic Programming, Greedy Algorithms.

Introduction: Basic Design and Analysis Techniques of Algorithms

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the algorithm is independent from any programming languages.

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Problem Development Steps

The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm

- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

Characteristics of Algorithms

The main characteristics of algorithms are as follows:

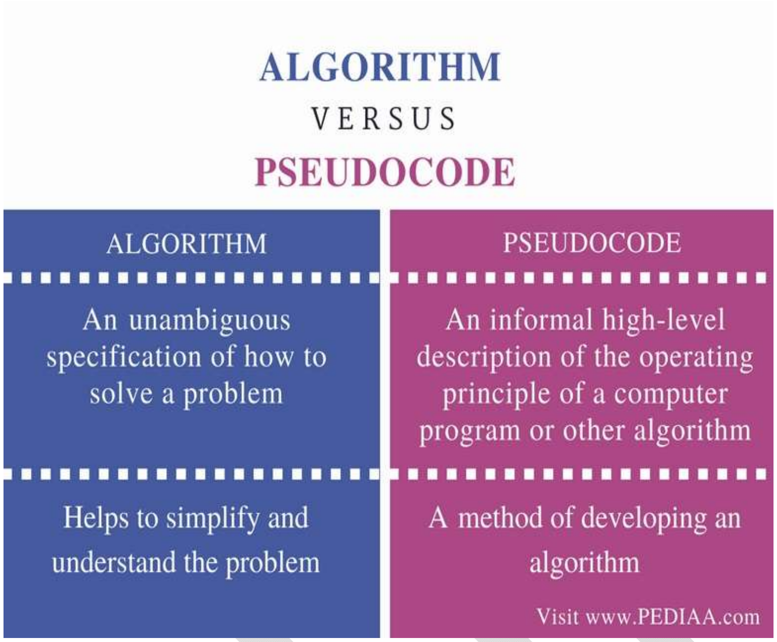
- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

Pseudocode

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

The running time can be estimated in a more general manner by using Pseudocode to represent the algorithm as a set of fundamental operations which can then be counted.

Difference between Algorithm and Pseudocode



ALGORITHM VERSUS PSEUDOCODE

ALGORITHM	PSEUDOCODE
An unambiguous specification of how to solve a problem	An informal high-level description of the operating principle of a computer program or other algorithm
Helps to simplify and understand the problem	A method of developing an algorithm

Visit www.PEDIAA.com

An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task. Generally, the word "algorithm" can be used to describe any high level task in computer science.

On the other hand, pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.

Algorithm: Insertion-Sort

```
Input: A list L of integers of length n
Output: A sorted list L1 containing those integers present in L
Step 1: Keep a sorted list L1 which starts off empty
Step 2: Perform Step 3 for each element in the original list L
 Step 3: Insert it into the correct position in the sorted list L1.
Step 4: Return the sorted list
Step 5: Stop
```

Here is a pseudocode which describes how the high level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

```
for i ← 1 to length(A) x ←
        A[i]
        j ← i
        while j > 0 and A[j-1] > x A[j]
                ← A[j-1]
                j ← j - 1 A[j]
        ← x
```

In this tutorial, algorithms will be presented in the form of pseudocode, which is similar in many respects to C, C++, Java, Python, and other programming languages.

Analysis of algorithms

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objectives of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input.

Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis:

- **Worst-case**: The maximum number of steps taken on any instance of size **a**.
- **Best-case**: The minimum number of steps taken on any instance of size **a**.
- **Average case**: An average number of steps taken on any instance of size **a**.
- **Amortized**: A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or maybe vice-versa. In this context, if we compare bubble sort and merge sort. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Correctness of Algorithm

When an algorithm is designed it should be analyzed at least from the following points of view:

Correctness. This means to verify if the algorithm leads to the solution of the problem (hopefully after a finite number of processing steps).

Efficiency. This means to establish the amount of resources (memory space and processing time) needed to execute the algorithm on a machine (a formal one or a physical one).

Basic steps in algorithms correctness verification

To verify if an algorithms really solves the problem for which it is designed we can use one of the following strategies:

Experimental analysis (testing). We test the algorithm for different instances of the problem (for different input data). The main advantage of this approach is its simplicity while the main disadvantage is the fact that testing cannot cover always all possible instances of input data (it is difficult to know how much testing is enough). However the experimental analysis allows sometimes to identify situations when the algorithm doesn't work.

Formal analysis (proving). The aim of the formal analysis is to prove that the algorithm works for any instance of data input. The main advantage of this approach is that if it is rigourously applied it guarantee the correctness of the algorithm. The main disadvantage is the difficulty of finding a proof, mainly for complex algorithms. In this case the algorithm is decomposed in subalgorithms and the analysis is focused on these (simpler) subalgorithms. On the other hand the formal approach could lead to a better understanding of the algorithms. This approach is called formal due to the use of formal rules of logic to show that an algorithm meets its specification.

The main steps in the formal analysis of the correctness of an algorithm are:
Identification of the properties of input data (the so-called *problem's preconditions*).
Identification of the properties which must be satisfied by the output data (the so called *problem's postconditions*).

Proving that starting from the preconditions and executing the actions specified in the algorithms one obtains the postconditions.

When we analyze the correctness of an algorithm a useful concept is that of *state*.

The algorithm's state is the set of the values corresponding to all variables used in the algorithm.

The state of the algorithm changes (usually by variables assignments) from one processing step to another processing step. The basic idea of correctness verification is to establish which should be the state corresponding to each processing step such that at the end of the algorithm the postconditions are satisfied. Once we established these intermediate states is sufficient to verify that each processing step ensures the transformation of the current state into the next state.

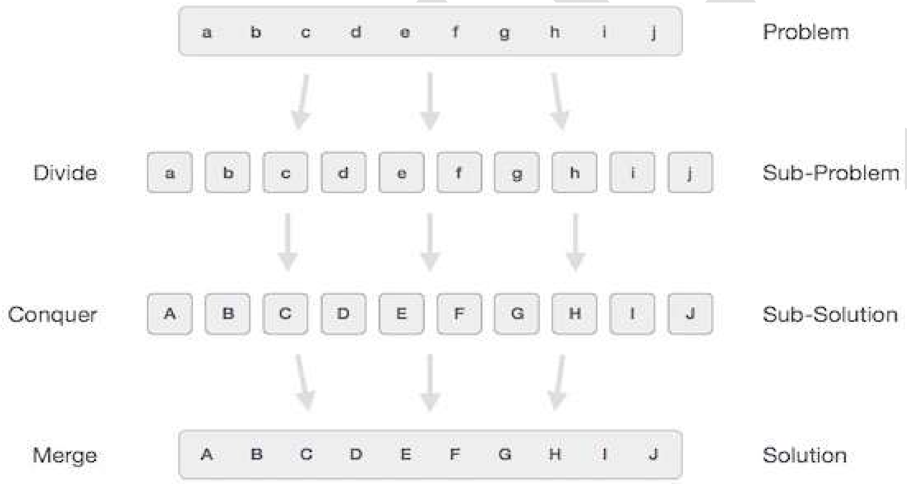
When the processing structure is a sequential one (for example a sequence of assignments) then the verification process is a simple one (we must only analyze the effect of each assignment on the algorithm's state).

Difficulties may arise in analyzing loops because there are many sources of errors: the initializations may be wrong, the processing steps inside the loop may be wrong or the stopping condition may be wrong. A formal method to prove that a loop statement works correctly is the mathematical induction method.

Algorithm Design Techniques: Iterative Techniques

Divide and Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

For example,

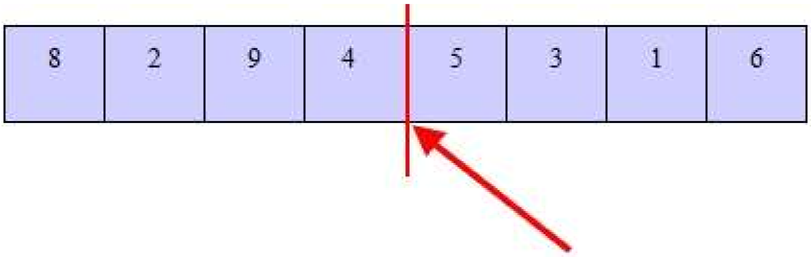
The following computer algorithms are based on divide-and-conquer programming approach −

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)

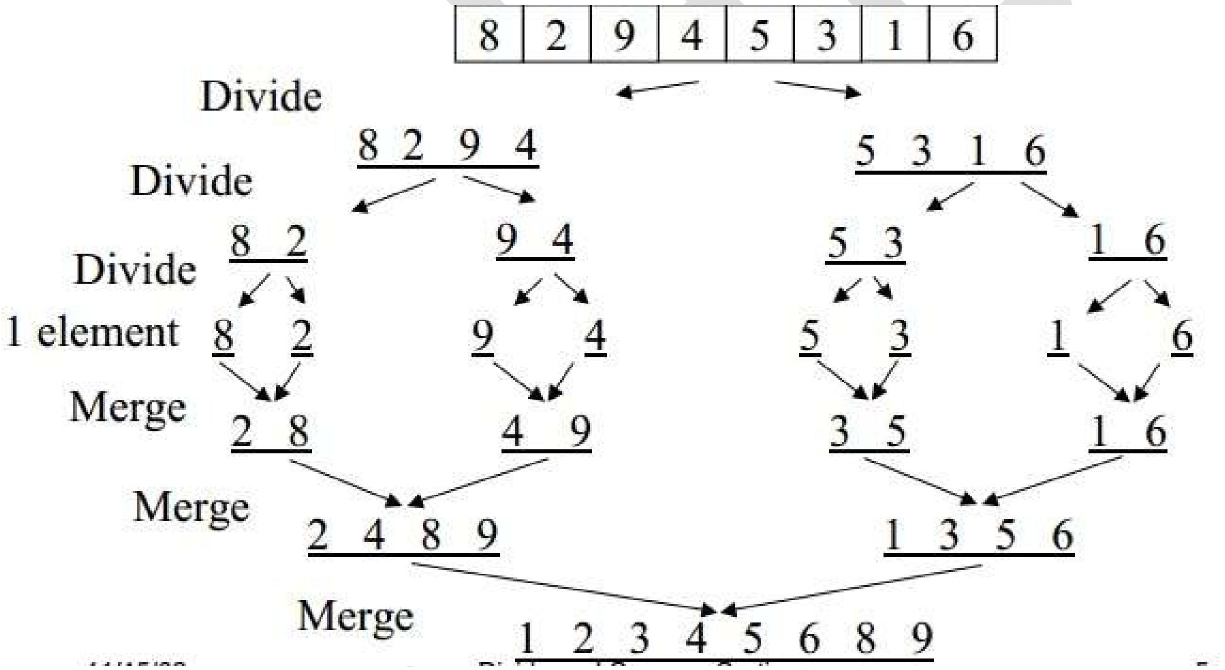
Example for Divide and Conquer

- Idea 1: Divide array into two halves, recursively sort left and right halves,then mergetwohalvesknownasMergesort
- Idea 2 : Partition array into small items and large items, then recursively sort the two sets known asQuicksort

Merge Sort Example



- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halve together



Dynamic Programming

Dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those sub problems just once, and storing their solutions. The next time the same sub problem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

The following computer problems can be solved using dynamic programming approach −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

Fibonacci (n) = 1; if n = 0
Fibonacci (n) = 1; if n = 1
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21**...** and so on!

A code for it using pure recursion:

```
int fib (int n) {
    if (n < 2)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

Using Dynamic Programming approach with memoization:

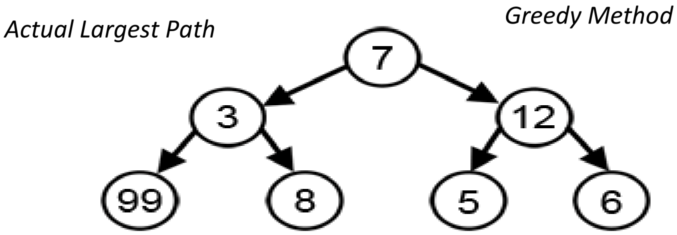
```
void fib () {
    fibresult[0] = 1;
    fibresult[1] = 1;
    for (int i = 2; i<n; i++)
       fibresult[i] = fibresult[i-1] + fibresult[i-2];
}
```

Greedy Algorithms

Some optimization problems can be solved using a greedy algorithm. A greedy algorithm builds a solution iteratively. At each iteration the algorithm uses a greedy rule to make its choice. Once a choice is made the algorithm never changes its mind or looks back to consider a different perhaps better solution; the reason the algorithm is called greedy.

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's Algorithm, which is used to find the shortest path through a graph.

However, in many problems, a greedy strategy does not produce an optimal solution. For example, in the animation below, the greedy algorithm seeks to find the path with the largest sum. It does this by selecting the largest available number at each step. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, and without regard to the overall problem.

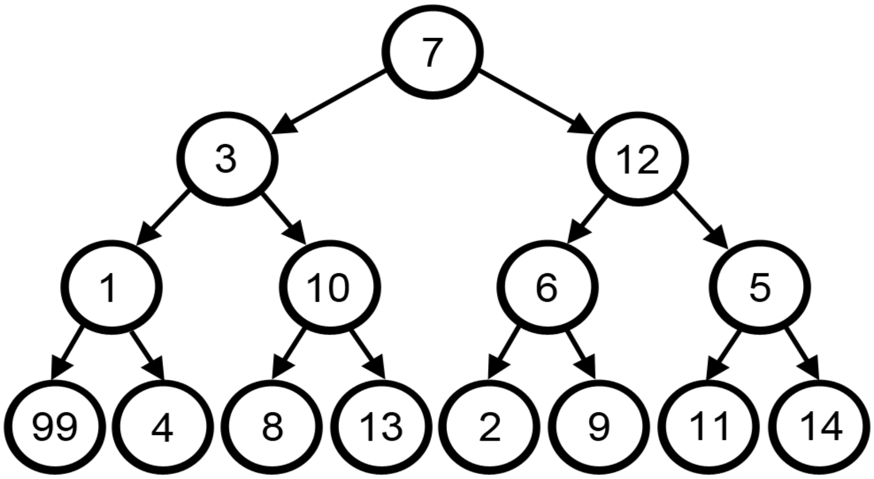


Limitations of Greedy Algorithms

Sometimes greedy algorithms fail to find the globally optimal solution because they do not consider all the data. The choice made by a greedy algorithm may depend on choices it has made so far, but it is not aware of future choices it could make.

Example: 1

In the graph below, a greedy algorithm is trying to find the longest path through the graph (the number inside each node contributes to a total length). To do this, it selects the largest number at each step of the algorithm. With a quick visual inspection of the graph, it is clear that this algorithm will not arrive at the correct solution.



Solution:

The correct solution for the longest path through the graph is 7, 3, 1, 99. This is clear to us because we can see that no other combination of nodes will come close to a sum of 99, so whatever path we choose, we know it should have 99 in the path. There is only one option that includes 99 is 7, 3, 1, 99.

The greedy algorithm fails to solve this problem because it makes decisions purely based on what the best answer at the time is: at each step it *did* choose the largest number. However, since there could be some huge number that the algorithm hasn't seen yet, it could end up selecting a path that does not include the huge number. The solutions to the subproblems for finding the largest sum or longest path do not necessarily appear in the solution to the total problem. The optimal substructure and greedy choice properties don't hold in this type of problem.

Example : 2 – Knapsack Problem

Here, we will look at one form of the knapsack problem. The knapsack problem involves deciding which subset of items you should take from a set of items if you want to optimize some value: perhaps the worth of the items, the size of the items, or the ratio of worth to size.

In this problem, we will assume that we can either take an item or leave it (we cannot take a fractional part of an item). In this problem we will assume that there is only one of each item. Our knapsack has a fixed size, and we want to optimize the worth of the items we take, so we must choose the items we take with care.

Our knapsack can hold at most 25 units of space.

Here is the list of items and their worth.

Item	**Size**	**Price**
Laptop	22	12
Playstation	10	9
Textbook	9	9
Basketball	7	6

Which items do we choose to optimize for price?

Solution:

There are two greedy algorithms we could propose to solve this. One has a rule that selects the item with the largest price at each step, and the other has a rule that selects the smallest sized item at each step.

Largest Price Algorithm: At the first step, we take the laptop. We gain 12 units of worth, but can now only carry 25 – 22 = 3 units of additional space in the knapsack. Since no items that remain will fit into the bag, we can only take the laptop and have a total of 12 units of worth.

Smallest Sized Item Algorithm: At the first step, we will take the smallest sized item: the basketball. This gives us 6 units of worth, and leaves us with 25-7=18 units of space in our bag. Next, we select the next smallest item, the textbook. This gives us a total of 6+9=15 units of worth, and leaves us with 18-9=9 units of space. Since no remaining items are 9 units of space or less, we can take no more items.

The greedy algorithms yield solutions that give us 12 units of worth and 15 units of worth. But neither of these are the optimal solution. Inspect the table yourself and see if you can determine a better selection of items.

Taking the textbook and the play station yields 9+9=18 units of worth and takes up 10+9=19 units of space. This is the optimal answer, and we can see that a greedy algorithm will not solve the knapsack problem since the greedy choice and optimal substructure properties do not hold.

In problems where greedy algorithms fail, dynamic programming might be a better approach.

Drawback of Greedy

A greedy algorithm works by choosing the best possible answer in each step and then moving on to the next step until it reaches the end, without regard for the overall solution. It only hopes that the path it takes is the globally optimum one, but as proven time and again, this method does not often come up with a globally optimum solution. In fact, it is entirely possible that the most optimal short-term solutions lead to the worst possible global outcome.

POSSIBLE QUESTIONS

UNIT-I

2 Mark Questions:

1. Define an Algorithm.

2. What is meant by Time Complexity?

3. Define Analysis of an Algorithm.

4. What do you mean by Correctness of an algorithm?

5. State Divide and Conquer approach.

6. What is known as Efficiency of an algorithm?

7. Define the concept of Dynamic Programming.

8. What is the limitation of Greedy algorithm?

9. Define Knapsack problem.

10. List out the types of Algorithm analysis.

6 Mark Questions:

1. Discuss the Basic Design and Analysis Techniques of algorithms.

2. Explain about Correctness of algorithms.

3. Explain Divide and Conquer with example.

4. Explain the concept of Dynamic Programming.

5. Describe in detail about Iterative techniques.

6. Differentiate Pseudocode and Algorithm with example.

7. Explain about Algorithm Design.

8. Discuss in detail about Dynamic Programming.

9. Explain the correctness of an algorithm.

10. Elaborate Greedy technique.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act, 1956)

Coimbatore-641021

Department of Computer Science

II B.Sc(CS) (BATCH 2017-2020)

Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS **ONE MARK QUESTIONS**

UNIT-1

S.NO	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	All the above	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	All the above	effectiveness
5	_______ criteria of an algorithm ensures that each step of the algorithm must be clear and unambiguous.	effectiveness	finiteness	definiteness	All the above	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	effectiveness	finiteness	definiteness	performance
8	__________ is a set of steps of operations to solve a problem.	Sub-Problem	Sub-Task	Algorithm	Process	Algorithm
9	__________ from any programming languages.	independent	dependent	Concept	Based	independent
10	Pseudocode is a ______ description of an algorithm.	Low-level	Middle-level	High-level	Bottom-level	High-level

11	An algorithm is a _______ definition with some specific characteristics.	Informal	Formal	Descriptive	Customized	Formal
12	Bubble sort does not require additional ________ when compared to merge sort.	Complexity	Memory	Time	Time and Memory	Memory
13	Identification of the properties of input data is called as _______.	Problem’s preconditi	Problem’s postcondition	Problem’s in buttons	Problem’s outcondition	Problem’s precondition
14	the problem into sub-problems	Divide	Break	Both a & b	combine	Both a & b
15	Divide and Conquer is a ______ step process.	5	4	3	2	3
16	Tree represents the nodes connected by ________.	Edges	Arrows	Tables	Squares	Edges
17	Binary Tree is a special ________ used for data storage purposes.	Element	data structure	Object	Representation	data structure
18	can have a maximum of ________.	three nodes	Two Pairs	two children	Both a & b.	two children
19	________ refers to the sequence of nodes along the edges of a tree.	Edge	Root	Path	Traversal	Path
20	_____ number is the minimum number of colors required to color a graph.	Vertex	Node	Chromatic	non-Chromatic	Chromatic
21	A ______ is an equation that describes a function in terms of its value.	Binary search	Recurrence	Tree	Graph	Recurrence
22	Interconnected objects in a graph are called _____.	Trees	Graphs	Entities	Vertices	Vertices
23	Recurrences are generally used in _______ paradigm.	Interface	Graph theory	Divide-and-conquer	Both a & b	Divide-and-conquer
24	procedure can be interpreted graphically as sliding a _____	template	Granules	Recursive	Interface	template
25	String-matching algorithms are used for______	Graphics	Characters	Pattern searching	Aggregation	Pattern searching
26	Which of the following technique performs pre-processing?	Naive	Rabin	KMP	Morris	Naive

27	The zero-length string is denoted as _______.	Null string	Empty string	Void string	Exit	Empty string
28	When the maximum entries of (m*n) matrix are zeros then it is called as _______.	Transpose matrix	Sparse Matrix	Inverse Matrix	None of the above.	Sparse Matrix
29	A matrix of the form (row, col, n) is otherwise known as _______.	Transpose matrix	Inverse Matrix	Sparse Matrix	None of the above.	Sparse Matrix
30	Which of the following is a valid linear data structure.	Stacks	Records	Trees	Graphs	Stacks
31	Which of the following is a valid non - linear data structure.	Stacks	Trees	Queues	Linked list.	Trees
32	A list of finite number of homogeneous data elements are called as _________	Stacks	Records	Arrays	Linked list.	Arrays
33	No of elements in an array is called the _________ of an array.	Structure	Height	Width	Length.	Length.
34	_________ is the art of creating sample data upon which to run the program	Testing	Designing	Analysis	Debugging	Testing
36	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
37	A _________ is a collection of elements such that each element has been assigned a	Priority Queue	De Queue	Circular Queue	En Queue	Priority Queue
38	A _________ is made up of Operators and Operands.	Stack	Expression	Linked list	Queue	Expression
39	A _____________ is a procedure or function which calls itself.	Stack	Recursion	Queue	Tree	Recursion
40	An example for application of stack is __________.	Time sharing computer	Waiting Audience	Processing of subroutines	None of the above	Processing of subroutines

41	An example for application of queue is __________.	Stack of coins	Stack of bills	Processing of subroutines	Job Scheduling in TimeSharing computers	Job Scheduling in TimeSharing computers
42	Combining elements of two similar data structure into one is called __________	Merging	Insertion	Searching	Sorting	Merging
43	Adding a new element into a data structure called	Merging	Insertion	Searching	Sorting	Insertion
44	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
45	Arranging the elements of a data structure in some type of order is called	Merging	Insertion	Searching	Sorting	Sorting
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 mathematical	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 key	Place	Location	Value	Operand	Location
50	Length of an array is defined as ___________ of elements in it.	Structure	Height	Size	Number	Number
51	In _______ search method the search begins by examining the record in the middle of the file	sequential	fibonacci	binary	non-sequential	binary
52	________ is a internal sorting method.	sorting with disks	quick sort	balanced merge sort	sorting with tapes	quick sort

53	Quick sort reads _______ space to implement the recursion	stack	queue	circular stacks	circular queue	stack
54	The most popular method for sorting on external storage devices is _____.	quick sort	radix sort	merge sort	heap sort	merge sort
55	The 2-way merge algorithm is almost identical to the ___________procedure.	quick	merge	heap	radix	merge
56	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
57	Associating an element of an ordered list A_i with an index i is called __________	linear	sequential	ordered	indexed	linear
58	________ is a set of pairs, index and value.	stack	queue	Arrays	Set	queue
59	The design approach where the main task is decomposed into subtasks and each subtask is further decomposed into simpler solutions is called ________	top down approach	bottom up approach	hierarchical approach	merging approach	top down approach
60	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	top down approach

UNIT II

SYLLABUS

Sorting and Searching Techniques: Elementary sorting techniques–Bubble Sort, Insertion Sort, Merge Sort, Advanced Sorting techniques - Heap Sort, Quick Sort, Sorting in Linear Time - Bucket Sort, Radix Sort and Count Sort, Searching Techniques, Medians & Order Statistics, complexity analysis;

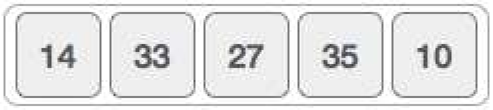
Sorting and Searching Techniques: Elementary sorting techniques

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. 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.

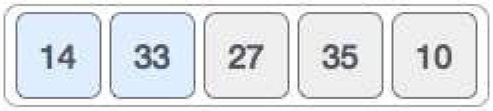
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes O(n^2) time so we're keeping it short and precise.



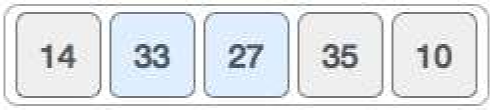
14	33	27	35	10

Bubble sort starts with very first two elements, comparing them to check which one is greater.



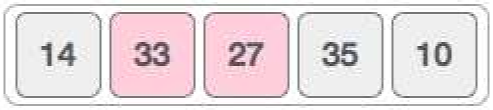
14	33	27	35	10

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



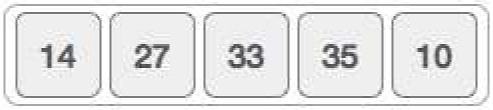
14	33	27	35	10

We find that 27 is smaller than 33 and these two values must be swapped.



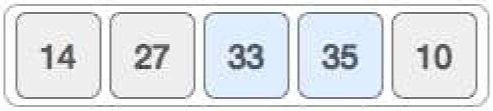
14	33	27	35	10

The new array should look like this −



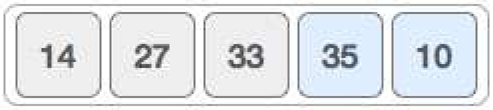
14	27	33	35	10

Next we compare 33 and 35. We find that both are in already sorted positions.



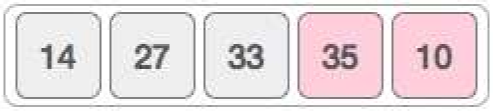
14	27	33	35	10

Then we move to the next two values, 35 and 10.



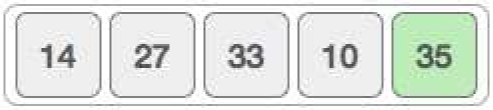
14	27	33	35	10

We know then that 10 is smaller 35. Hence they are not sorted.



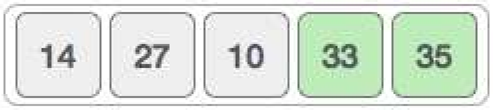
14	27	33	35	10

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



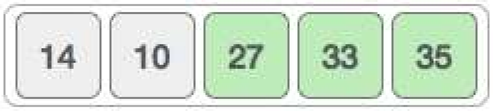
14	27	33	10	35

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



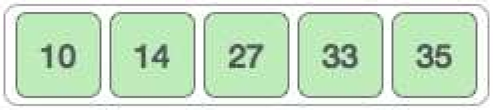
14	27	10	33	35

Notice that after each iteration, at least one value moves at the end.



14 10 27 33 35

And when there's no swap required, bubble sorts learns that an array is completely sorted.



10 14 27 33 35

Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap**function swaps the values of the given array elements.

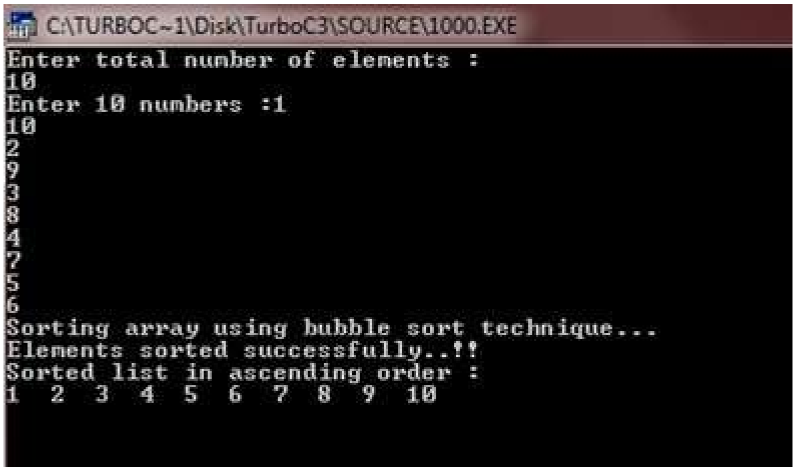
```
beginBubbleSort(list)

for all elements of list

if list[i]> list[i+1]

        swap(list[i], list[i+1])

endif

endfor

return list

endBubbleSort
```

Program-Bubble Sort

```
/* C++ Program - Bubble Sort */

#include<iostream.h>
#include<conio.h>
void main()
{
                clrscr();
                int n, i, arr[50], j, temp;
```

```
    cout<<"Enter total number of elements :";
    cin>>n;
    cout<<"Enter "<<n<<" numbers :";
    for(i=0; i<n; i++)
    {
        cin>>arr[i];
    }
    cout<<"Sorting array using bubble sort technique...\n";
    for(i=0; i<(n-1); i++)
    {
        for(j=0; j<(n-i-1); j++)
        {
            if(arr[j]>arr[j+1])
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
    cout<<"Elements sorted successfully..!!\n";
    cout<<"Sorted list in ascending order :\n";
    for(i=0; i<n; i++)
    {
        cout<<arr[i]<<" ";
    }
    getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:



```
Enter total number of elements :
10
Enter 10 numbers :1
10
2
9
3
8
4
7
5
6
Sorting array using bubble sort technique...
Elements sorted successfully..!!
Sorted list in ascending order :
1  2  3  4  5  6  7  8  9  10
```

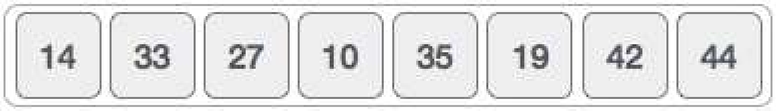
Insertion Sort

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.

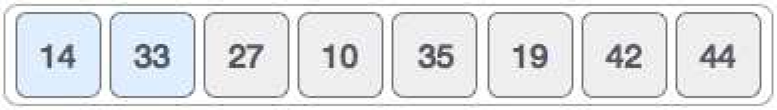
How Insertion Sort Works?

We take an unsorted array for our example.



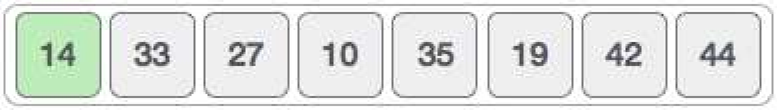
14	33	27	10	35	19	42	44

Insertion sort compares the first two elements.



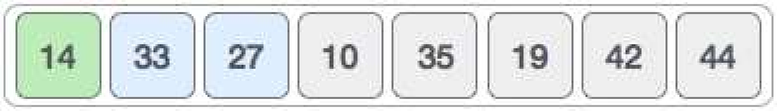
14	33	27	10	35	19	42	44

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



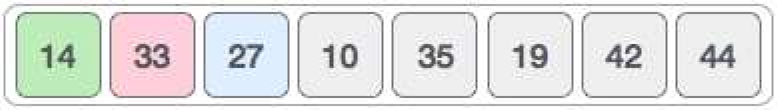
14	33	27	10	35	19	42	44

Insertion sort moves ahead and compares 33 with 27.



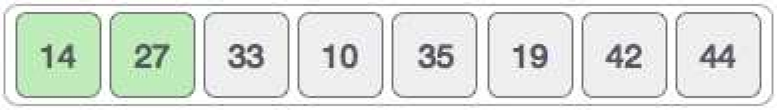
14	33	27	10	35	19	42	44

And finds that 33 is not in the correct position.



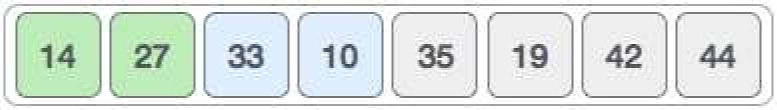
14	33	27	10	35	19	42	44

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.



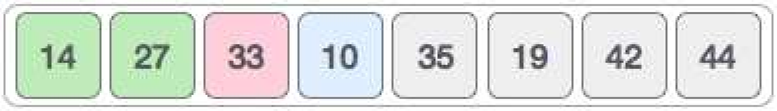
14	27	33	10	35	19	42	44

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



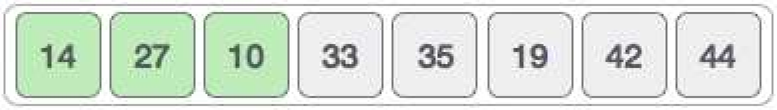
14	27	33	10	35	19	42	44

These values are not in a sorted order.



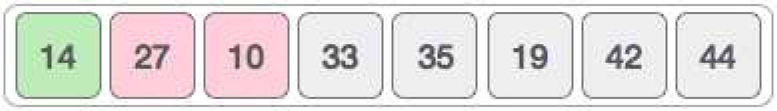
14	27	33	10	35	19	42	44

So we swap them.



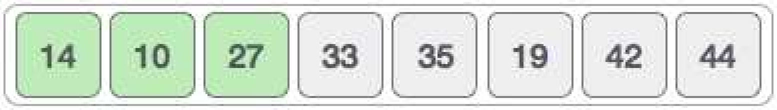
14	27	10	33	35	19	42	44

However, swapping makes 27 and 10 unsorted.



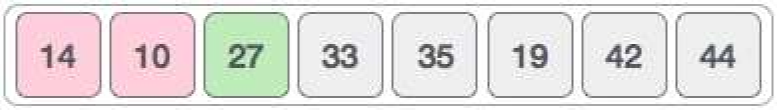
14	27	10	33	35	19	42	44

Hence, we swap them too.



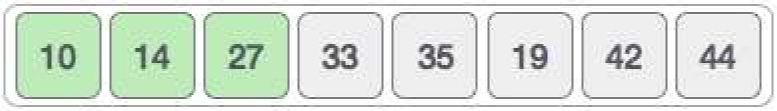
14	10	27	33	35	19	42	44

Again we find 14 and 10 in an unsorted order.



14	10	27	33	35	19	42	44

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



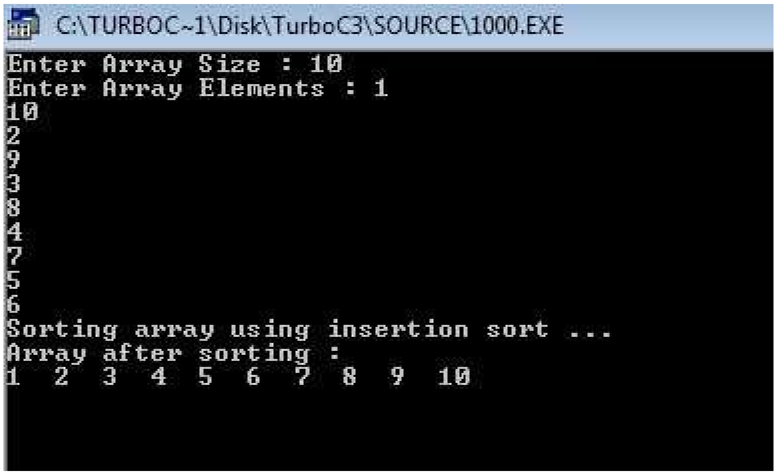
This process goes on until all the unsorted values are covered in a sorted sub-list.

Program for Insertion Sort

Following C++ program ask to the user to enter array size and array element to sort the array using insertion sort technique, then display the sorted array on the screen:

```
/* C++ Program - Insertion Sort */

#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int size, arr[50], i, j, temp;
        cout<<"Enter Array Size : ";
        cin>>size;
        cout<<"Enter Array Elements : ";
        for(i=0; i<size; i++)
        {
                cin>>arr[i];
        }
        cout<<"Sorting array using selection sort ... \n";
        for(i=1; i<size; i++)
        {
                temp=arr[i];
                j=i-1;
                while((temp<arr[j]) && (j>=0))
                {
                        arr[j+1]=arr[j];
                        j=j-1;
                }
                arr[j+1]=temp;
        }
        cout<<"Array after sorting : \n";
        for(i=0; i<size; i++)
        {
                cout<<arr[i]<<" ";
        }
        getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:



```
Enter Array Size : 10
Enter Array Elements : 1
10
2
9
3
8
4
7
5
6
Sorting array using insertion sort ...
Array after sorting :
1  2  3  4  5  6  7  8  9  10
```

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

Problem Description

1. Merge-sort is based on an algorithmic design pattern called divide-and-conquer.
2. It forms tree structure.
3. The height of the tree will be log(n).
4. we merge n element at every level of the tree.
5. The time complexity of this algorithm is O(n*log(n)).

Problem Solution

1. Split the data into two equal half until we get at most one element in both half.
2. Merge Both into one making sure the resulting sequence is sorted.
3. Recursively split them and merge on the basis of constraint given in step 1.

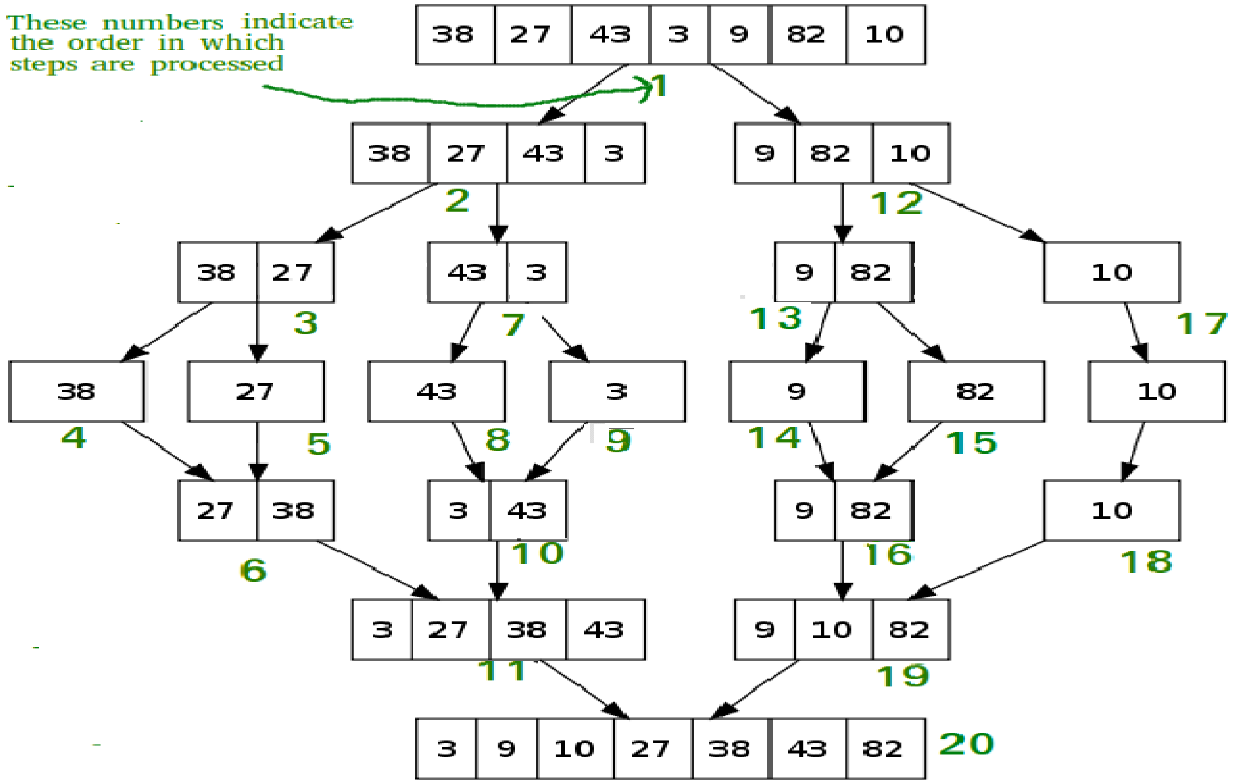
4. Display the result.
5. Exit.

Implementation using c++

```
MergeSort(arr[], l,  r)

If r > l

1. Find the middle point to divide the array into two halves:

          middle m = (l+r)/2

 2. Call mergeSort for first half:

          Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

          Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

          Call merge(arr, l, m, r)
```

The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

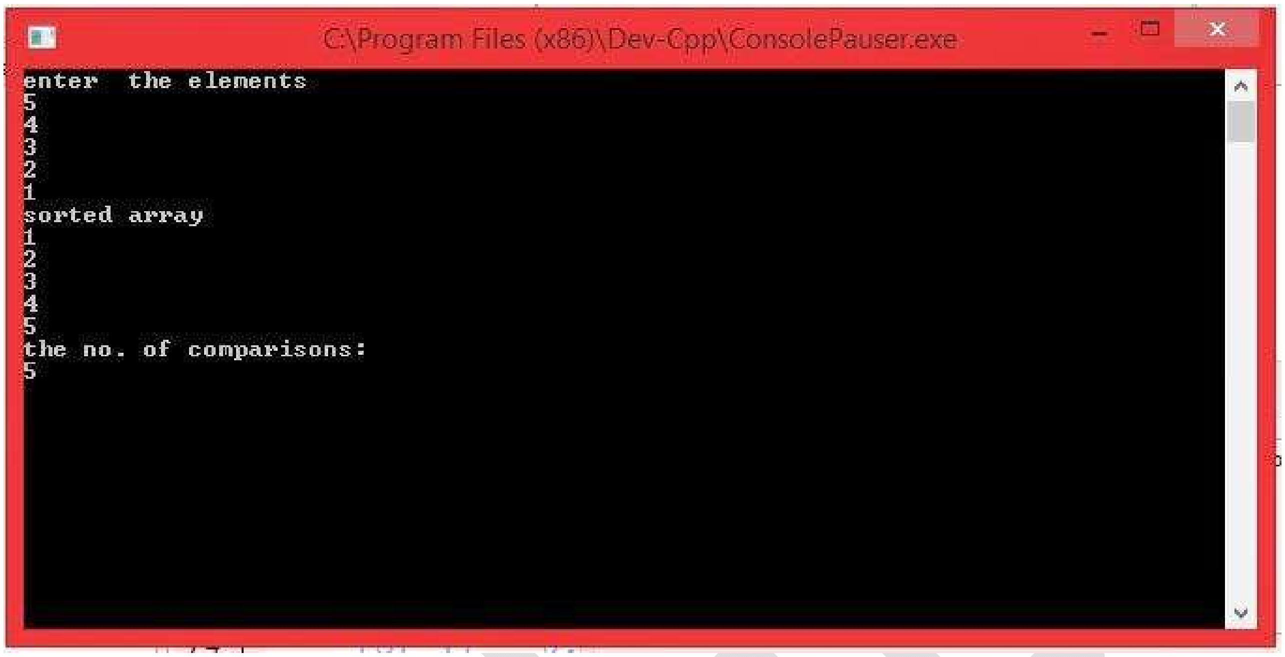


Program for Merge Sort:

```
#include <iostream>
using namespace std;
#include <conio.h>
int comp=0;
void merge(int *,int, int , int );
void mergesort(int *a, int low, int high)
{
   int mid;
   if (low < high)
   {
      mid=(low+high)/2;
```

```
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,high,mid);
    }
    return;
}
void merge(int *a, int low, int high, int mid)
{
    inti, j, k, c[50];
    i = low;
    k = low;
    j = mid + 1;
    while (i<= mid && j <= high)
    {
        if(a[i] < a[j])
        {
            c[k] = a[i];
            k++;
            i++;
            comp++;
        }
        else
        {
            c[k] = a[j];
            k++;
            j++;
            comp++;
        }
    }
    while (i<= mid)
    {
        c[k] = a[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        c[k] = a[j];
        k++;
        j++;
    }
    for (i = low; i< k; i++)
    {
        a[i] = c[i];
    }
}
int main()
{
    int a[20], i, b[20];
    cout<<"enter  the elements\n";
    for (i = 0; i< 5; i++)
    {
        cin>>a[i];
    }
    mergesort(a, 0, 4);
    cout<<"sorted array\n";
    for (i = 0; i< 5; i++)
    {
        cout<<a[i]<<"\n";
    }
    cout<<"the no. of comparisons:\n"<<comp<<endl;
    getch();
}
```

Output:



```
enter  the elements
5
4
3
2
1
sorted array
1
2
3
4
5
the no. of comparisons:
5
```

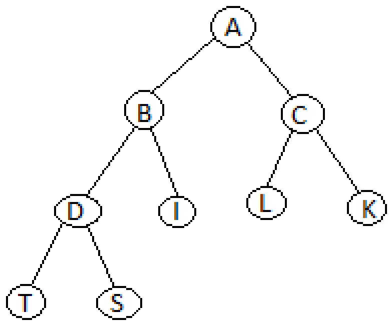
Heap Sort

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. The heapsort algorithm has O(n log n) time complexity. Heap sort algorithm is divided into two basic parts :

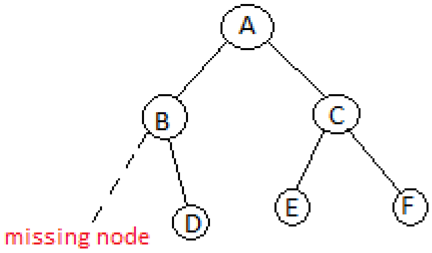
- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Heap is a special tree-based data structure that satisfies the following special heap properties:

1. **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.



Complete Binary Tree

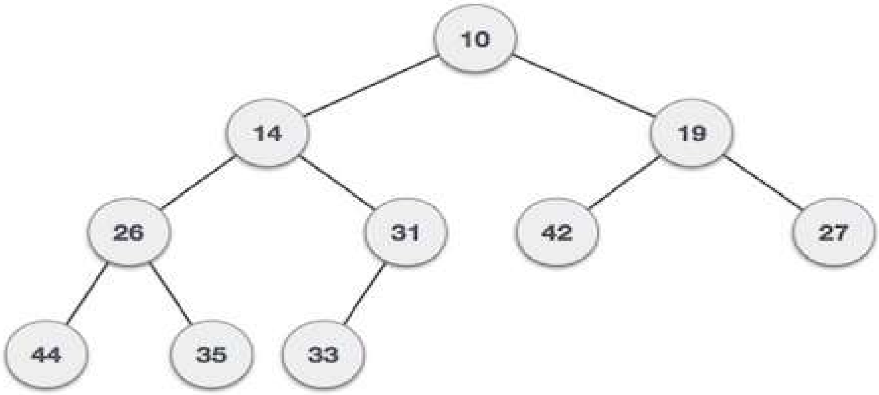


In-Complete Binary Tree

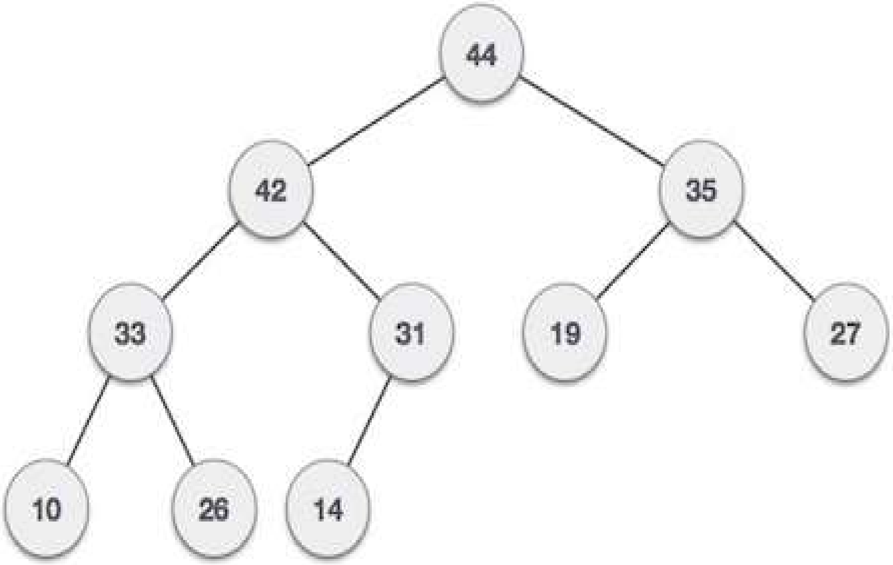
2. **Heap Property :** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



An Example of Heapsort:

Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort.

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
2. Start deleteMin operations, storing each deleted element at the end of the heap array.

After performing step 2, the order of the elements will be opposite to the order in the heap tree. Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree in descending order - the greatest element will have the highest priority.

Note that we use only one array , treating its parts differently:

a. when building the heap tree, part of the array will be considered as the heap, and the rest part - the original array.
b. when sorting, part of the array will be the heap, and the rest part - the sorted array.

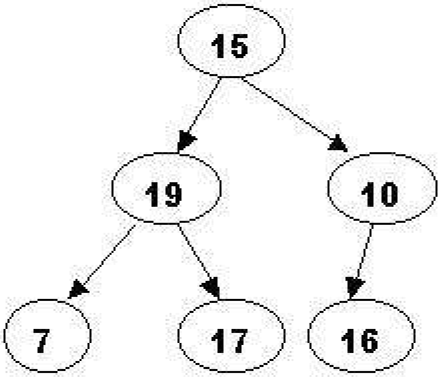
This will be indicated by colors: white for the original array, blue for the heap and red for the sorted array

Here is the array: 15, 19, 10, 7, 17, 6

A. Building the heap tree

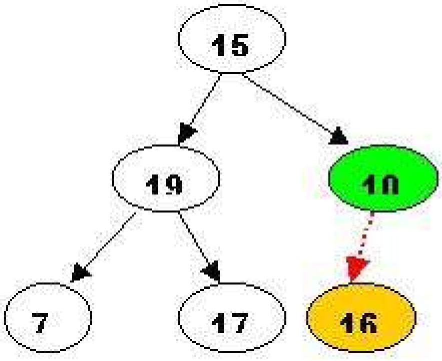
The array represented as a tree, complete but not ordered:

15	19	10	7	17	16

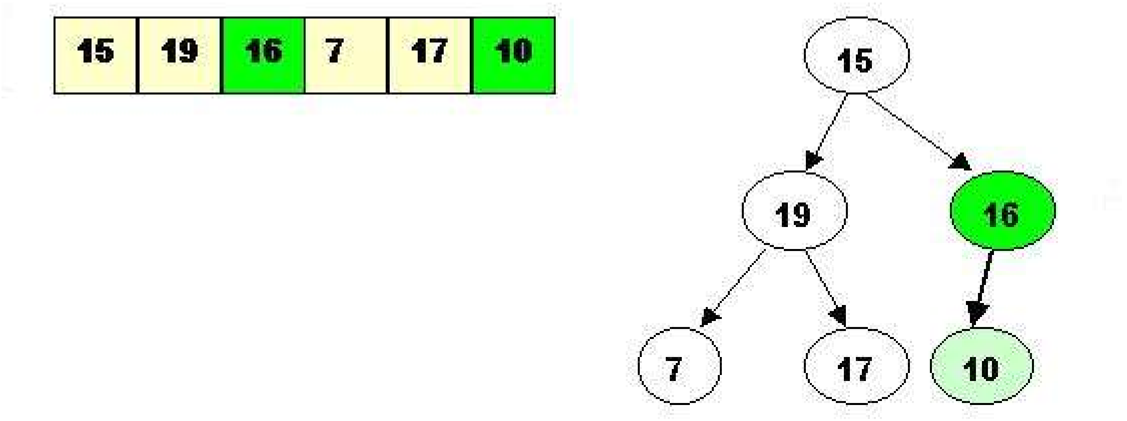


Start with the rightmost node at height 1 - the node at position 3 = Size/2.
It has one greater child and has to be percolated down:

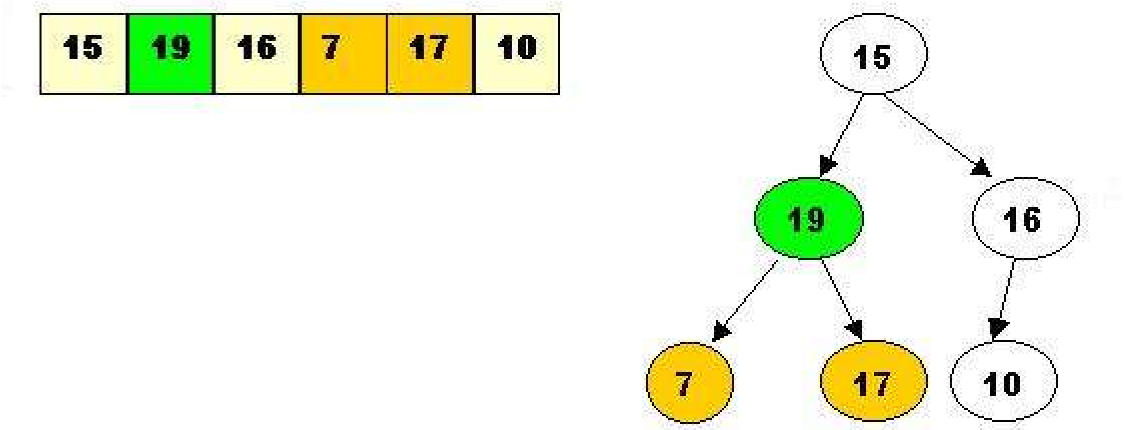
15	19	10	7	17	16



After processing array[3] the situation is:

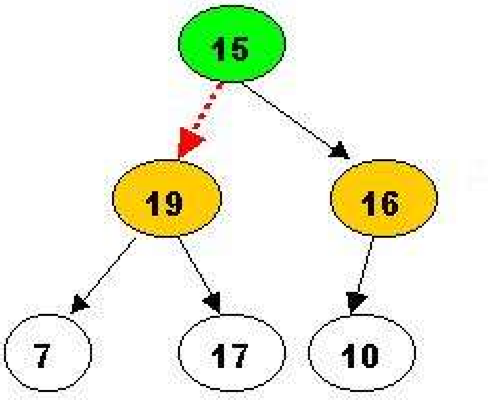


Next comes array[2]. Its children are smaller, so no percolation is needed.



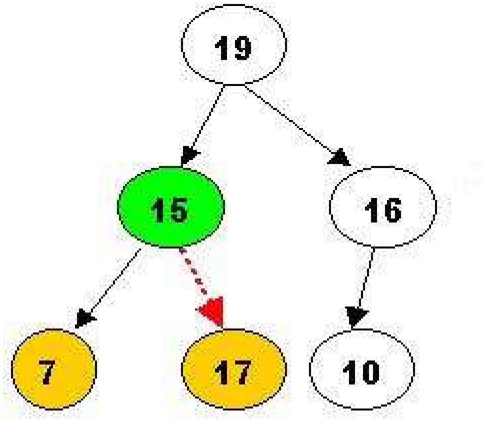
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array[1] has to be percolated down to the left, swapped with array[2].

15	19	16	7	17	10

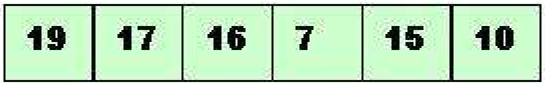


As a result the situation is:

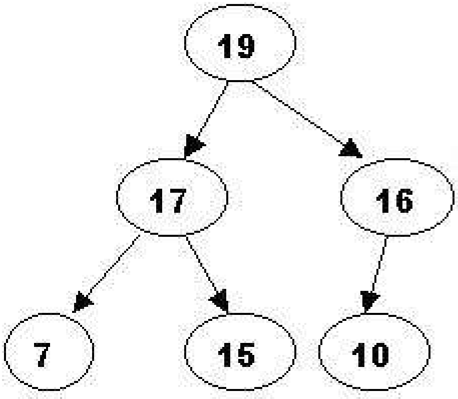
19	15	16	7	17	10



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].



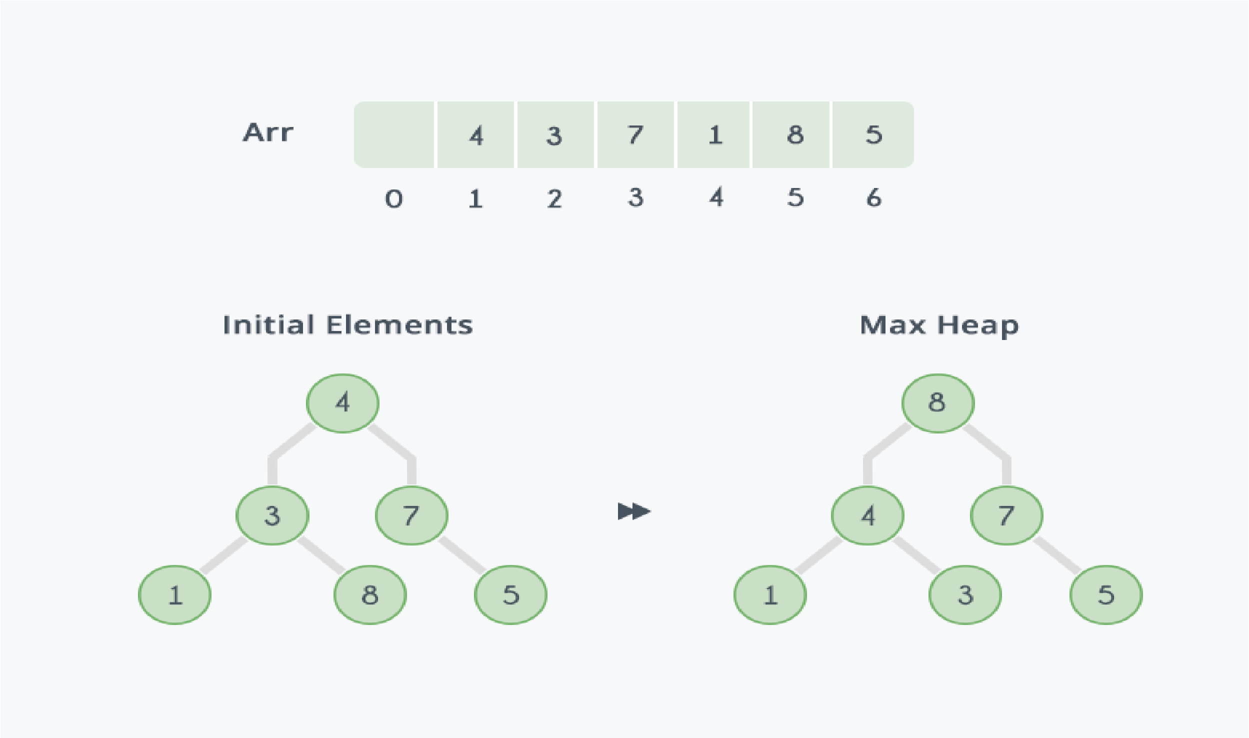
19	17	16	7	15	10



Now the tree is ordered, and the binary heap is built.

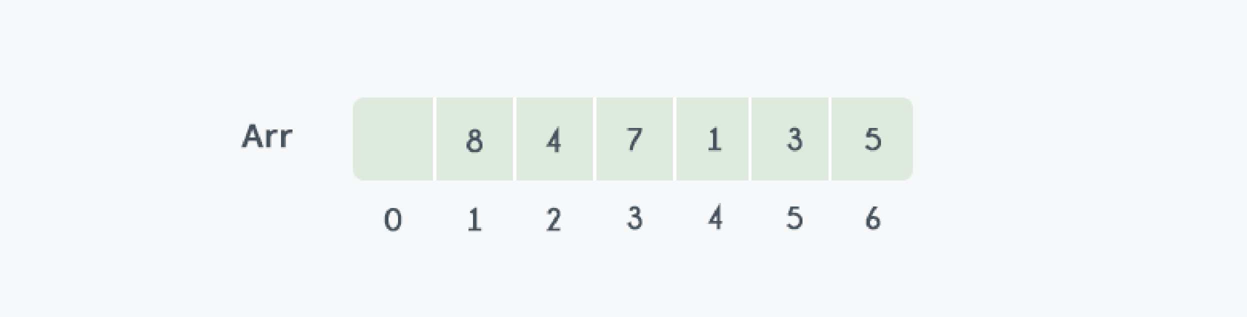
Example:

In the diagram below, initially there is an unsorted array Arr having 6 elements and then max-heap will be built.



Arr		4	3	7	1	8	5
	0	1	2	3	4	5	6

After building max-heap, the elements in the array Arr will be:



Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6

Step 1: 8 is swapped with 5.

Step 2: 8 is disconnected from heap as 8 is in correct position now and.

Step 3: Max-heap is created and 7 is swapped with 3.

Step 4: 7 is disconnected from heap.

Step 5: Max heap is created and 5 is swapped with 1.

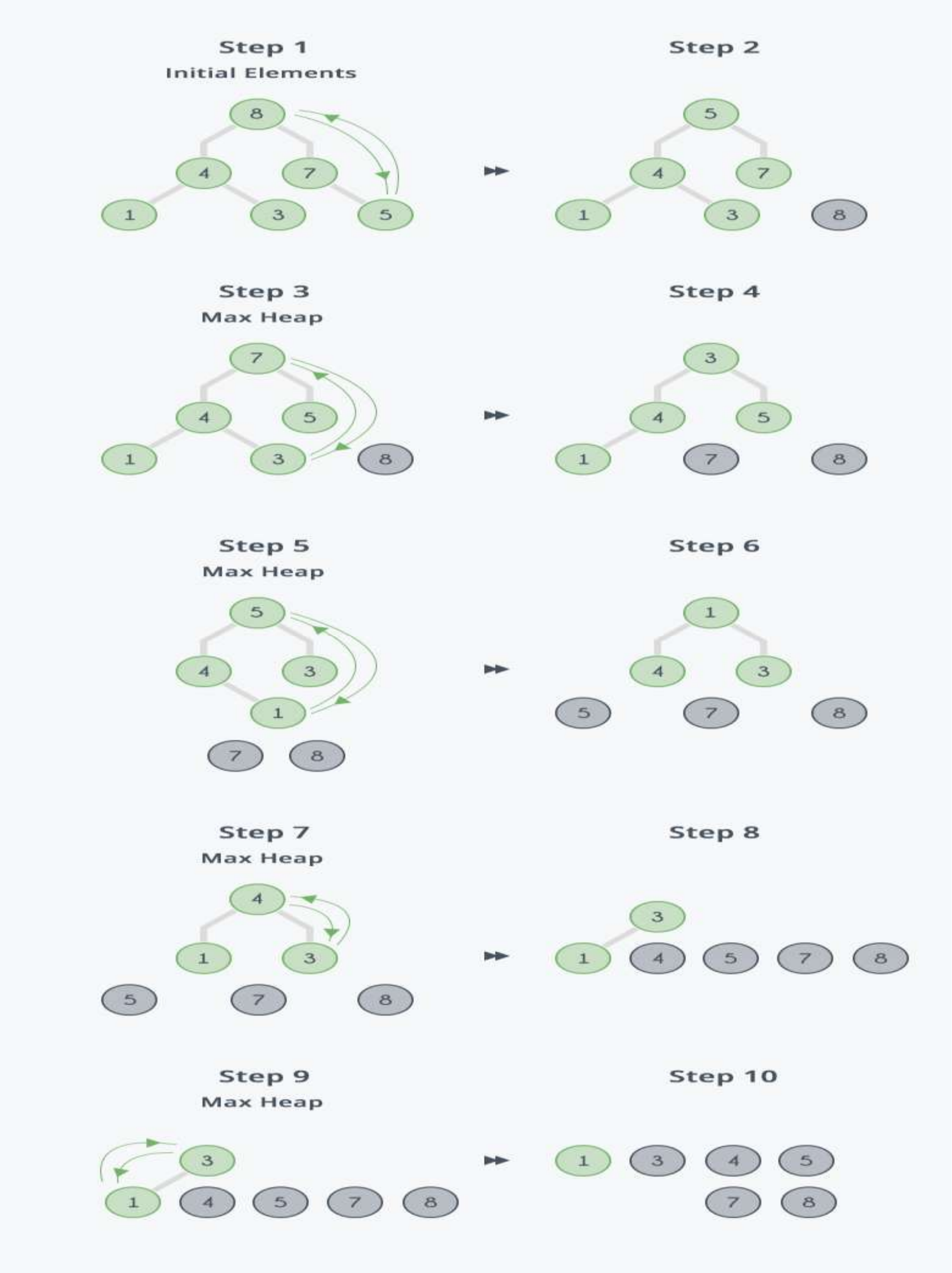
Step 6: 5 is disconnected from heap.

Step 7: Max heap is created and 4 is swapped with 3.

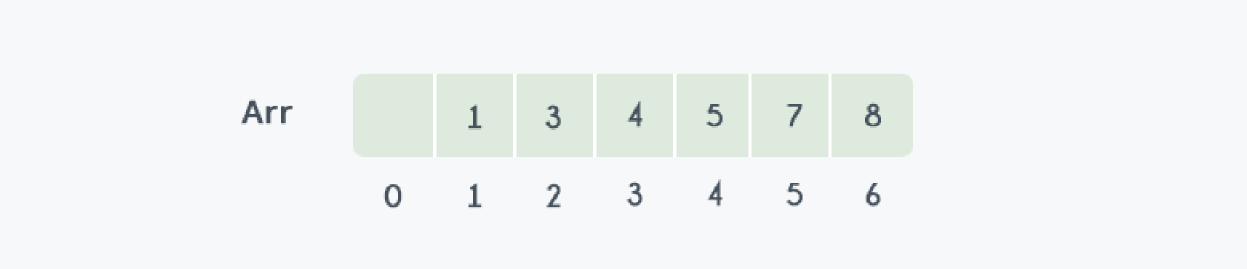
Step 8: 4 is disconnected from heap.

Step 9: Max heap is created and 3 is swapped with 1.

Step 10: 3 is disconnected.


Step 1
Initial Elements
8
4
7
1
3
5
Step 2
5
4
7
1
3
8
Step 3
Max Heap
7
4
5
1
3
8
Step 4
3
4
5
1
7
8
Step 5
Max Heap
5
4
3
1
7
8
Step 6
1
4
3
5
7
8
Step 7
Max Heap
4
1
3
5
7
8
Step 8
3
1
4
5
7
8
Step 9
Max Heap
3
1
4
5
7
8
Step 10
1
3
4
5
7
8

After all the steps, we will get a sorted array.



Arr		1	3	4	5	7	8
	0	1	2	3	4	5	6

Program for Heap Sort

```
#include <iostream>

using namespace std;


void max_heapify(int *a, inti, int n)

{

   int j, temp;

   temp = a[i];

   j = 2*i;

   while (j <= n)

   {

      if (j < n && a[j+1] > a[j])

         j = j+1;

      if (temp > a[j])

         break;

      else if (temp <= a[j])

      {
            a[j/2] = a[j];
            j = 2*j;
        }
    }
    a[j/2] = temp;
    return;
}
void heapsort(int *a, int n)
{
    inti, temp;
    for (i = n; i>= 2; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        max_heapify(a, 1, i - 1);
    }
}
void build_maxheap(int *a, int n)
{
    inti;
    for(i = n/2; i>= 1; i--)
    {
        max_heapify(a, i, n);
    }
```

```
}
int main()
{
   int n, i, x;
   cout<<"Enter no of elements of array\n";
   cin>>n;
   int a[20];
   for (i = 1; i<= n; i++)
   {
      cout<<"Enter element"<<(i)<<endl;
      cin>>a[i];
   }
   build_maxheap(a,n);
   heapsort(a, n);
   cout<<"\n\nSorted Array\n";
   for (i = 1; i<= n; i++)
   {
      cout<<a[i]<<endl;
   }
   return 0;
}
```

Output:

```
Enter no of elements of array

5

Enter element1

3

Enter element2

8

Enter element3

9

Enter element4

3

Enter element5

2

Sorted Array

2

3

3

8

9
```

QuickSort

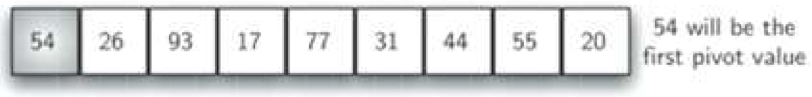
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)

3. Pick a random element as pivot.
4. Pick median as pivot.

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

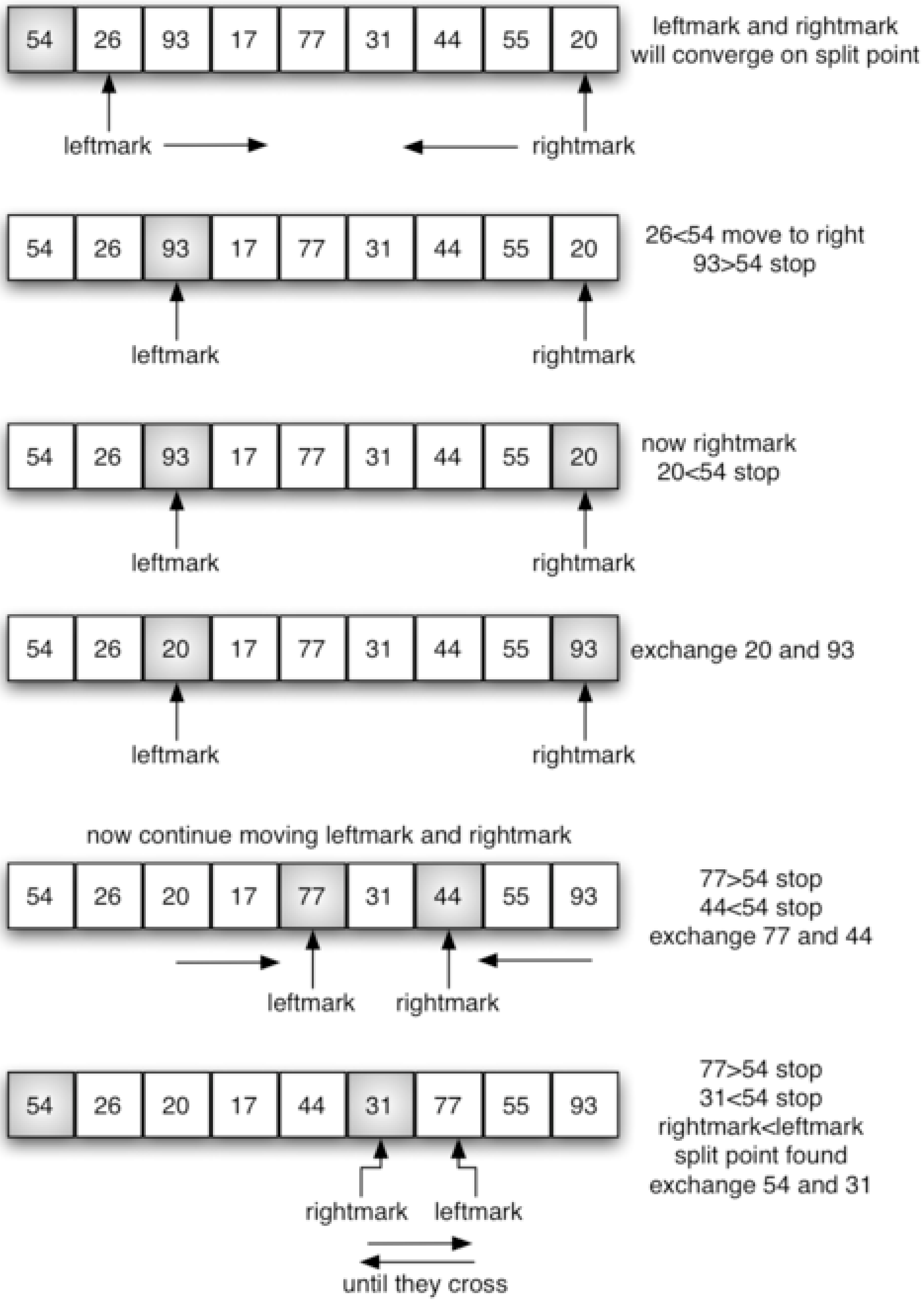
Figure 12 shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.



54	26	93	17	77	31	44	55	20

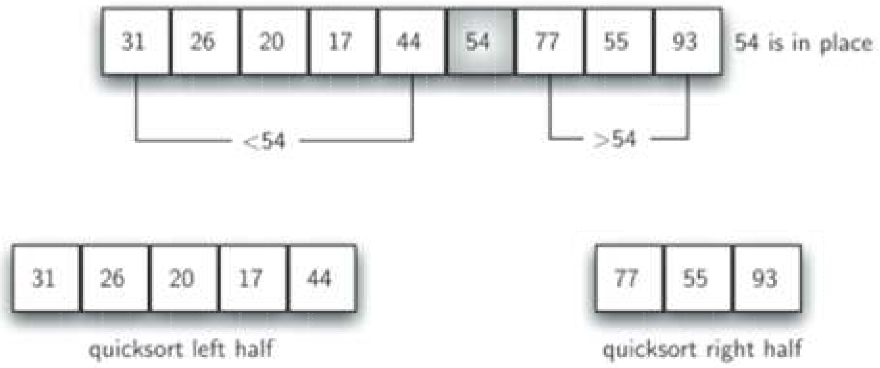
54 will be the first pivot value

Partitioning begins by locating two position markers—let’s call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure 13). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure 13 shows this process as we locate the position of 54.



We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (Figure 14). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



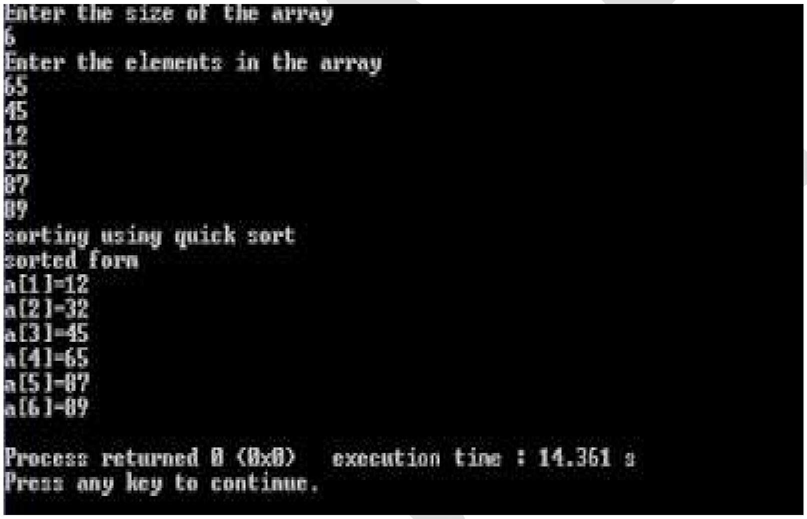
Algorithm:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
         at right place */
      pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);  // Before pi
quickSort(arr, pi + 1, high); // After pi
   }
}
```

Program:

```
#include<iostream>
usingnamespace std;

voidQUICKSORT(int [],int ,int );
intPARTITION(int [],int,int );

intmain()
{
int n;
cout<<"Enter the size of the array"<<endl;
cin>>n;
int a[n];
cout<<"Enter the elements in the array"<<endl;
for(inti=1;i<=n;i++)
    {
cin>>a[i];
    }
cout<<"sorting using quick sort"<<endl;
int p=1,r=n;
   QUICKSORT(a,p,r);
cout<<"sorted form"<<endl;
for(inti=1;i<=n;i++)
   {
cout<<"a["<<i<<"]="<<a[i]<<endl;
   }
return0;
}

voidQUICKSORT(int a[],intp,int r)
    {
int q;
if(p<r)
       {
        q=PARTITION(a,p,r);
        QUICKSORT(a,p,q-1);
        QUICKSORT(a,q+1,r);
       }
    }

intPARTITION(int a[],intp,int r)
    {
inttemp,temp1;
int x=a[r];
inti=p-1;
for(int j=p;j<=r-1;j++)
        {
if(a[j]<=x)
            {

i=i+1;
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
        temp1=a[i+1];
        a[i+1]=a[r];
        a[r]=temp1;
return i+1;
    }
```

Output:



```
Enter the size of the array
6
Enter the elements in the array
65
45
12
32
87
89
sorting using quick sort
sorted form
a[1]=12
a[2]=32
a[3]=45
a[4]=65
a[5]=87
a[6]=89

Process returned 0 (0x0)   execution time : 14.361 s
Press any key to continue.
```

Bucket Sort

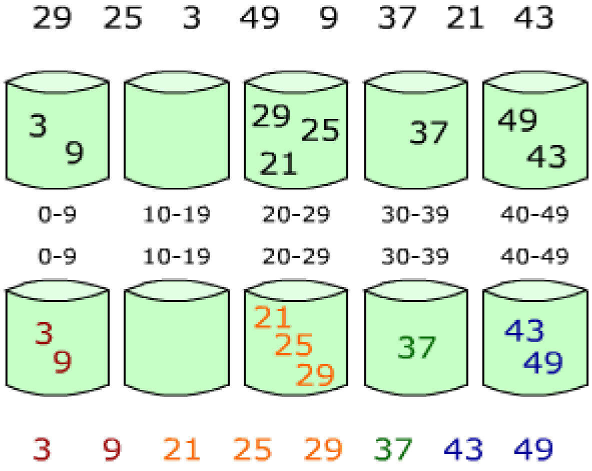
Bucket sort is a sorting algorithm that works by partitioning an array into a number of buckets.

In bucket sort algorithm the array elements are distributed into a number of buckets. Then. each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. The computational complexity estimates involve the number of buckets.

Bucket sort works as follows:

1. Set up an array of initially empty buckets.
2. Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Visit the buckets in order and put all elements back into the original array.

Example-1



Algorithm

```
bucketSort(arr[], n)

1) Create n empty buckets (Or lists).

2) Do following for every array element arr[i].

.......a) Insert arr[i] into bucket[n*array[i]]
```

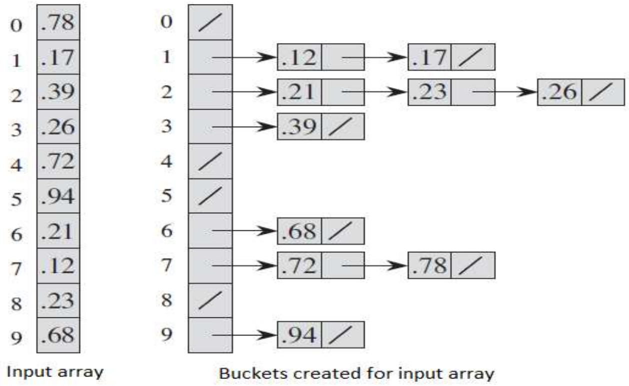
3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

Bucket sort is *mainly useful when input is uniformly distributed over a range. For example, Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range.*

Example-2

Following diagram demonstrates working of bucket sort.



```
// C++ program to sort an array using bucket sort

#include <iostream>

#include <algorithm>

#include <vector>

usingnamespacestd;


// Function to sort arr[] of size n using bucket sort

voidbucketSort(floatarr[], intn)

{
    // 1) Create n empty buckets
    vector<float> b[n];


    // 2) Put array elements in different buckets
    for(inti=0; i<n; i++)
    {
      intbi = n*arr[i]; // Index in bucket
      b[bi].push_back(arr[i]);
    }


    // 3) Sort individual buckets
    for(inti=0; i<n; i++)
      sort(b[i].begin(), b[i].end());


    // 4) Concatenate all buckets into arr[]
    intindex = 0;
    for(inti = 0; i< n; i++)
       for(intj = 0; j < b[i].size(); j++)
         arr[index++] = b[i][j];
}
 intmain()
{
    floatarr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
```

```
   intn = sizeof(arr)/sizeof(arr[0]);
   bucketSort(arr, n);


   cout<< "Sorted array is \n";
   for(inti=0; i<n; i++)
     cout<<arr[i] << " ";
   return0;
}
```

Output:

```
Sorted array is
0.1234 0.3434 0.565 0.656 0.665 0.897
```

Radix Sort

Radix Sort puts the elements in order by comparing the **digits of the numbers**. Radix sort works by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Algorithm: Radix-Sort (list, n)

```
shift = 1
for loop = 1 to keysize do
  for entry = 1 to n do
bucketnumber = (list[entry].key / shift) mod 10
    append (bucket[bucketnumber], list[entry])
  list = combinebuckets()
  shift = shift * 10
```

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist

0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385

4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

Disadvantages

The speed of Radix Sort largely depends on the inner basic operations, and **if** the operations are not efficient enough, **Radix Sort *can* be slower than some other algorithms** such as Quick Sort and Merge Sort. These operations include the insert and delete functions of the sublists and the process of isolating the digit you want.

In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.

Radix Sort can also take up more **space** than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sublist for **each** of the possible digits or letters.

Example-2

Following example shows how Radix sort operates on seven 3-digits number.

Input	1st Pass	2nd Pass	3rd Pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the above example, the first column is the input. The remaining columns show the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in an array ***A*** of ***n***elements has ***d*** digits, where digit ***1*** is the lowest-order digit and ***d*** is the highest-order digit.

Program for Radix sort

```
#include <iostream>


usingnamespacestd;


// Get maximum value from array.
```

```
intgetMax(intarr[], int n)
{
        int max =arr[0];
        for(inti=1;i< n;i++)
                if(arr[i]> max)
                        max =arr[i];
        return max;
}


// Count sort of arr[].
voidcountSort(intarr[], int n, intexp)
{
        // Count[i] array will be counting the number of array values having that 'i' digit at their (exp)th place.
        int output[n], i, count[10]={0};


        // Count the number of times each digit occurred at (exp)th place in every input.
        for(i=0;i< n;i++)
                count[(arr[i]/exp)%10]++;


        // Calculating their cumulative count.
        for(i=1;i<10;i++)
                count[i]+= count[i-1];
```

```
    // Inserting values according to the digit '(arr[i] / exp) % 10' fetched into count[(arr[i] / exp) % 10].
    for(i= n -1;i>=0;i--)
    {
        output[count[(arr[i]/exp)%10]-1]=arr[i];
        count[(arr[i]/exp)%10]--;
    }

    // Assigning the result to the arr pointer of main().
    for(i=0;i< n;i++)
        arr[i]= output[i];
}

// Sort arr[] of size n using Radix Sort.
voidradixsort(intarr[], int n)
{
    intexp, m;
    m =getMax(arr, n);

    // Calling countSort() for digit at (exp)th place in every input.
    for(exp=1; m/exp>0;exp*=10)
        countSort(arr, n, exp);
}
```

```
intmain()
{
        int n, i;
        cout<<"\nEnter the number of data element to be sorted: ";
        cin>>n;

        intarr[n];
        for(i=0;i< n;i++)
        {
                cout<<"Enter element "<<i+1<<": ";
                cin>>arr[i];
        }

        radixsort(arr, n);

        // Printing the sorted data.
        cout<<"\nSorted Data ";
        for(i=0;i< n;i++)
                cout<<"->"<<arr[i];
        return0;
}
```

Output:

```
Enter the number of data element to be sorted: 5
Enter element

25

14

26

78

10

Sorted Data

10 ->14 ->25-> 26 ->78
```

Searching Techniques

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

- Linear Search or Sequential Search
- Binary 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 goes to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudocode

```
procedure linear_search (list, value)

  for each item in the list


    if match item == value

      return the item's location

    end if

  end for

end procedure
```

Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

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

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

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

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

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

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

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

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

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

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

  Set lowerBound = 1
  Set upperBound = n

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


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


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


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


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

Medians & Order Statistics

- The ith order statistic of a set of n elements is the ith smallest element.
- The minimum is the first order statistic (i =1).
- The maximum is the nth order statistic (i = n).
- A median is the “halfway point” of the set.
- When n is odd, the median is unique, at i = (n + 1)/2.
- When n is even, there are two medians

The selection problem

How can we find the ith order statistic of a set and what is the running time?

- Input: A set A of n (distinct) number and a number i, with 1 ≤ i ≤ n.
- Output: The element x A that is larger than exactly i–1 other elements of A.
- The selection problem can be solved in O (n log n) time.

Finding minimum

We can easily obtain an upper bound of n−1 comparisons for finding the minimum of a set of n elements.

- Examine each element in turn and keep track of the smallest one.
- The algorithm is optimal, because each element, except the minimum, must be compared to a smaller element at least once.

```
MINIMUM(A)
min ← A[1]
for i ← 2 to length[A]
    do if min > A[i]
        then min ← A[i]
return min
```

Selection in expected linear time

In fact, selection of the ith smallest element of the array A can be done in Θ(n) time.

We first present a randomized version in this section and then present a deterministic version in the next section.

The function RANDOMIZED - SELECT: ` is a divide -and -conquer algorithm, ` uses RANDOMIZED - PARTITION from the quicksort algorithm.

RANDOMIZED-SELECT procedure

```
RANDOMIZED-SELECT(A, p, r, i)
if p = r
    then return A[p]
```

```
q ← RANDOMIZED-PARTITION(A, p, r)
k ← q − p + 1
if i = k /* the pivot value is the answer */
then return A[q]
elseif i < k
then return RANDOMIZED□SELECT(A, p, q − 1, i)
else return RANDOMIZED□SELECT(A, q, r, i − k)
```

Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
   statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
   for(j=0; j < N;j++)
   {
   statement;
   }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
   mid = (low + high) / 2;
   if (target < list[mid])
      high = mid - 1;
   else if (target > list[mid])
      low = mid + 1;
   else break;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
   int pivot = partition(list, left, right);
   quicksort(list, left, pivot - 1);
   quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

Notations of Time Complexity

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

POSSIBLE QUESTIONS

UNIT II

2 Mark Questions:

1. Define Bubble Sort.
2. What is meant by insertion sort?
3. What is Divide-and-Conquer method?
4. Define Heap sort.
5. Define Radix sort.
6. What is known as Merge sort?
7. Define Complexity analysis.
8. Define Median.

6 Mark Questions:

1. Explain Bubble sort with example program.
2. Explain Insertion sort with example.
3. Explain Merge sort with suitable example.
4. Elaborate Heap sort technique with example.
5. Describe in detail about Quick sort.
6. Explain about Bucket sort.
7. Explain in detail about Radix sort.
8. Elaborate Count sort technique with example.
9. Explain about medians and other statistics.
10.Explain about Complexity analysis.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act, 1956)

Coimbatore-641021

Department of Computer Science

II B.Sc(CS) (BATCH 2017-2020)

Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS ONE MARK QUESTIONS

UNIT-2

S.NO	QUESTIONS	OPT1	OPT2	OPT3	OPT4	ANSWER
1	Algorithms must have _____ name.	Common	Unique	Different	Multiple	Unique
2	The term 'Analysis of Algorithms' was coined by ___	Prism	Donald David	Donald Ruth	Donald Knuth	Donald Knuth
3	Analysis of Algorithms is the determination of the ________ resources.	Time and Space	Time	Time and Analysis	All of these	Time and Space
4	There are _______ types of analysis performed in an algorithm.	5	4	2	3	4
5	Bubble sort is a _______ algorithm.	Sorting	Matching	Processing	Conquering	Sorting
6	Quick sort is an example for ________ technique.	Divide and Conqu	Knapsack	Tower of Hanoi	Greedy	Divide and Conquer
7	Dijkstra's algorithm is an example for ________	Greedy	Dynamic	Dynamic Program	All of these	Dynamic Programming
8	Sub-list is maintained in _______	Insertion	Bubble	heap	None of these	Insertion
9	The node at the top of the tree is called _____.	Edge	Child	Element	Root	Root
10	Any node except the root node has one edge upward to a node called ______.	Parent	Child	Element	Both a & b	Parent
11	The node below a given node connected by its edge downward is called ______.	Element	Identity	Child	Root	Child
12	The node which does not have any child node is called the _____ node.	Parent	Child	Root	Leaf	Leaf
13	_______ technique is used to find the complexity of a recurrence relation.	Greedy	Knapsack	Master's	Analysis	Master's Theorem

14	______ algorithm traverses a graph in a breadth ward motion.	Breadth First Search	Greedy	Aggregate	Amortized	Breadth First Search
15	______ provides a bound on the actual cost of the entire sequence.	Tree	Graph	Amortized	Aggregation	Amortized analysis
16	Vertices are also known as _____	Queue	Edges	Nodes	Stack	Nodes
17	In_____, each character is scanned atmost once.	Naive	Rabin	Karp	finite automat	finite automata
18	_______ method is applied for two-dimensional pattern matching.	Greedy	Kruskal	Prims	Rabin and	Rabin and Karp
19	The average-case running time of Rabin and Karp is _______.	Worse	high	Average	All of these	high
20	Knuth-Morris-Pratt algorithm is a ______ time string-matching algorithm.	Linear	non-linear	Directive	non-directive	Linear
21	Finding a free block whose size is as close as possible to the size of the program (N), but not less than N is called _____ allocation strategy.	Near fit	First fit	Best fit	Next Fit	Best fit
22	________ strategy distributes the small nodes evenly and searching for a new node starts from the node where the previous allocation was made.	Best Fit	First Fit	Worst Fit	Next Fit	Next Fit
23	Problem in _________ allocation stratery is all small nodes collect in the front of the av-list.	Best Fit	First Fit	Worst Fit	Next Fit	First Fit
24	________ is the storage allocation method that fits the program into the largest block available.	Best Fit	First Fit	Worst Fit	Next Fit	Worst Fit
25	The back pointer for each node will be referred as _________.	Blink	Break	Back	Clear	Blink
26	Forward pointer for each node will be referred as _________.	Forward	Flink	Front	Data	Flink
27	A___________is a linked list in which last node of the list points to the first node in the list.	Linked list	Singly linked circular list	Circular list	Insertion node	Singly linked circular list
28	A________in which each node has two pointers, a forward link and a Backward link.	Doubly linked circular	Circular list	Singly linked circular	Linked list	Doubly linked circular

29	In sparse matrices each nonzero term was represented by a node with ______ fields.	Five	Six	Three	Four	Three
30	We want to represent n stacks with 1 ≤ i ≤ n then T(i)_______	Top of the i^{th} stack	Top of the $(i + 1)^{th}$ stack	Top of the $(i – 1)^{th}$ stack	Top of the $(i - 2)^{th}$ stack	Top of the i^{th} stack
31	We want to represent m queues with 1 ≤ i ≤ m then F(i)_______	Front of the $(i + 1)^{th}$	Front of the i^{th} Queue	Front of the $(i – 1)^{th}$ Queue	Front of the $(i - 2)^{th}$ Queue	Front of the i^{th} Queue
32	We want to represent m queues with 1 ≤ i ≤ m then R(i)_______	Rear of the $(i + 1)^{th}$	Rear of the i^{th} Queue	Rear of the $(i – 1)^{th}$ Queue	Rear of the $(i - 2)^{th}$ Queue	Rear of the i^{th} Queue
33	In Linked representation of Sparse Matrix, DOWN field used to link to the next nonzero element in the same _________	Row	List	Column	Diagonal	Column
34	In Linked representation of Sparse Matrix, RIGHT field used to link to the next nonzero element in the same _________	Row	Matrix	Column	Diagonal	Row
35	The time complexity of the MREAD algorithm that reads a sparse matrix of n rows, n columns and r nonzero terms is ____	O(max {n, m, r})	O(m * n * r)	O(m + n + r)	O(max {n, m})	O(m + n + r)
36	In Available Space list combining the adjacent free blocks is called _________	Defragmenting	Coalescing	Joining	Merging	Coalescing
37	In Available Space list, the first and last word of each block are reserved for ___________	Data	Allocation Information	Link	Value	Allocation Information
38	In Storage management, in the Available Space List, the first word of each free block has ________fields.	4	3	2	1	4
39	In Available Space list, the last word of each free block has ________fields.	4	3	2	1	2

40	The first and last nodes of each block have tag fields, this system of allocation and freeing is called the _________.	Tag Method	Boundary Method	Free Method	Boundary Tag Method	Boundary Tag Method
41	In Available Space list ,Tag field has the value one when the block is _________	Allocated	Coalesced	Free	Merge	Allocated
42	Available Space list ,Tag field has the value Zero when the block is _________	Allocated	Coalesced	Free	Merged	Free
43	The ______field of each storage block indicates if the block is free are in-use.	rlink	tag	size	uplink	tag
44	In storage management the ________ field of the free block points to the start of the block	rlink	llink	uplink	top	rlink
45	__________ is the process of collecting all unused nodes and returning them to the available space.	Compaction	Coalescing	Garbage collection	Deallocation	Garbage collection
46	Moving all free nodes aside to form a single contiguous block of memeory is called __________	Compaction	Coalescing	Garbage collection	Deallocation	Compaction
47	__________ of disk space reduces the average retrieval time of allocation.	Compaction	Coalescing	Garbage collection	Deallocation	Compaction
48	___________ is done in two phases 1) marking used nodes and 2) returning all unmarked nodes to available space list.	Compaction	Coalescing	Garbage collection	Deallocation	Garbage collection
49	Which of these sorting algorithm uses the Divide and Conquere technique for sorting	selection sort	insertion sort	merge sort	heap sort	merge sort
50	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	None of the above	Binary search
51	The disadvantage of _____ sort is that is need a temporary array to sort.	Quick	Merge	Heap	Insertion	Merge
52	A __________ is a set of characters is called a string.	Array	String	Heap	List	String

53	The straight forward find operation for pattern matching,pat of size m in string of size n needs _________ time.	O(mn)	O(n^2)	O(m^2)	O(m+n)	O(mn)
54	Knuth,Morris and Pratt's method of pattern matching in strings takes ________ time, if pat is of sixe m and string is size n.	O(mn)	O(n^2)	O(m^2)	O(m+n)	O(m+n)
55	_________ representation always need extensive data movement.	Linked	sequential	tree	graph	sequential
56	Which of these representations are used for strings.	sequential representation	Linked representation with fixed sized blocks	Linked representation with variable sized blocks	All the above	All the above

UNIT III

SYLLABUS

Lower Bounding Techniques: Decision Trees **Balanced Trees:** Red-Black Trees

Lower Bounds

- Any algorithm that sorts by removing at most one inversion per comparison does at least n(n-1)/2 comparisons in the worst case.
- Any comparison based sorting algorithm must do at least Ω(n lg n) comparisons.

Decision Tree

A decision tree is a structure that includes a root node, branches, and leaf nodes. Each internal node denotes a test on an attribute, each branch denotes the outcome of a test, and each leaf node holds a class label. The topmost node in the tree is the root node.

The following decision tree is for the concept buy computer that indicates whether a customer at a company is likely to buy a computer or not. Each internal node represents a test on an attribute. Each leaf node represents a class.

The benefits of having a decision tree are as follows −

- It does not require any domain knowledge.
- It is easy to comprehend.
- The learning and classification steps of a decision tree are simple and fast.

Tree Pruning

Tree pruning is performed in order to remove anomalies in the training data due to noise or outliers. The pruned trees are smaller and less complex.

Tree Pruning Approaches

There are two approaches to prune a tree −

- **Pre-pruning** − The tree is pruned by halting its construction early.
- **Post-pruning** - This approach removes a sub-tree from a fully grown tree.

Cost Complexity

The cost complexity is measured by the following two parameters −

- Number of leaves in the tree, and
- Error rate of the tree.

Basic Explanation

A decision tree is a tree-like graph with nodes representing the place where we pick an attribute and ask a question; edges represent the answers to the question; and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface.

Decision trees classify the examples by sorting them down the tree from the root to some leaf node, with the leaf node providing the classification to the example. Each node in the tree acts as a test case for some attribute, and each edge descending from that node corresponds to one of the possible answers to the test case. This process is recursive in nature and is repeated for every subtree rooted at the new nodes.

Let's illustrate this with help of an example. Let's assume we want to play badminton on a particular day — say Saturday — how will you decide whether to play or not. Let's say you go out and check if it's hot or cold, check the speed of the wind and humidity, how the weather is, i.e. is it sunny, cloudy, or rainy. You take all these factors into account to decide if you want to play or not.

So, you calculate all these factors for the last ten days and form a lookup table like the one below.

Day	Weather	Temperature	Humidity	Wind	Play?

Day	Weather	Temperature	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Cloudy	Hot	High	Weak	Yes
3	Sunny	Mild	Normal	Strong	Yes
4	Cloudy	Mild	High	Strong	Yes
5	Rainy	Mild	High	Strong	No
6	Rainy	Cool	Normal	Strong	No
7	Rainy	Mild	High	Weak	Yes
8	Sunny	Hot	High	Strong	No
9	Cloudy	Hot	Normal	Weak	Yes

Day	Weather	Temperature	Humidity	Wind	Play?
10	Rainy	Mild	High	Strong	No

Table 1.Observations of the last ten days.

Now, you may use this table to decide whether to play or not. But, what if the weather pattern on Saturday does not match with any of rows in the table? This may be a problem. A decision tree would be a great way to represent data like this because it takes into account all the possible paths that can lead to the final decision by following a tree-like structure.

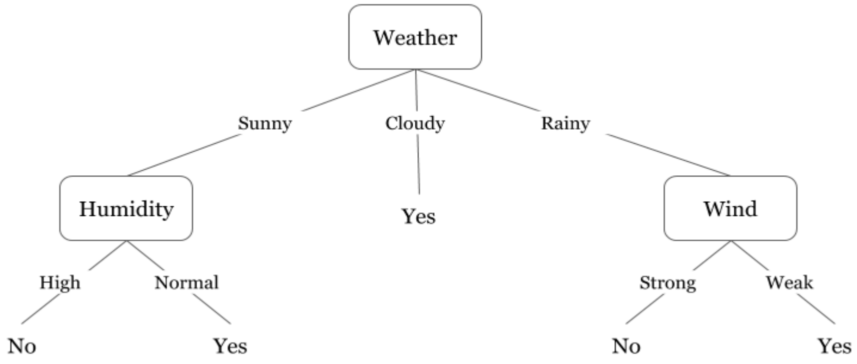


Fig 1. A decision tree for the concept Play Badminton

Fig 1.illustrates a learned decision tree. We can see that each node represents an attribute or feature and the branch from each node represents the outcome of that node. Finally, its the leaves of the tree where the final decision is made. If features are continuous, internal nodes can test the value of a feature against a threshold (see Fig. 2).

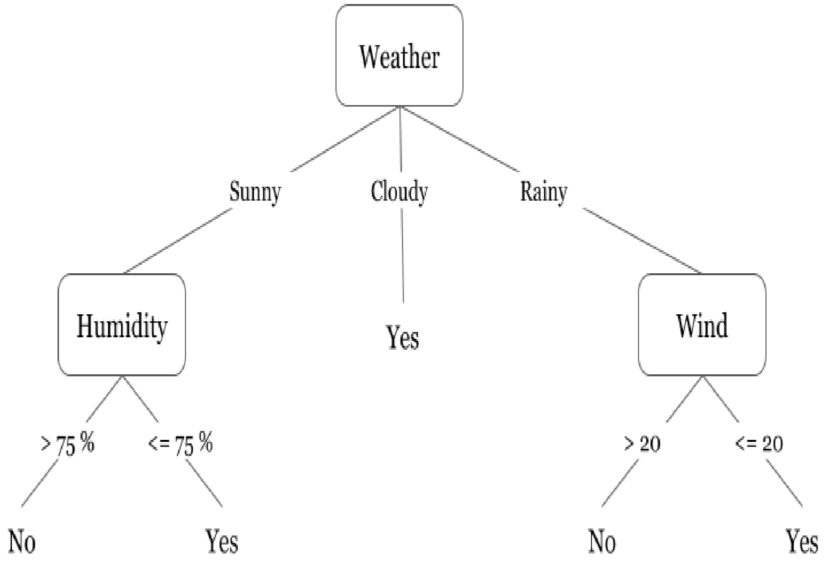


Fig 2. A decision tree for the concept Play Badminton (when attributes are continuous)

A general algorithm for a decision tree can be described as follows:

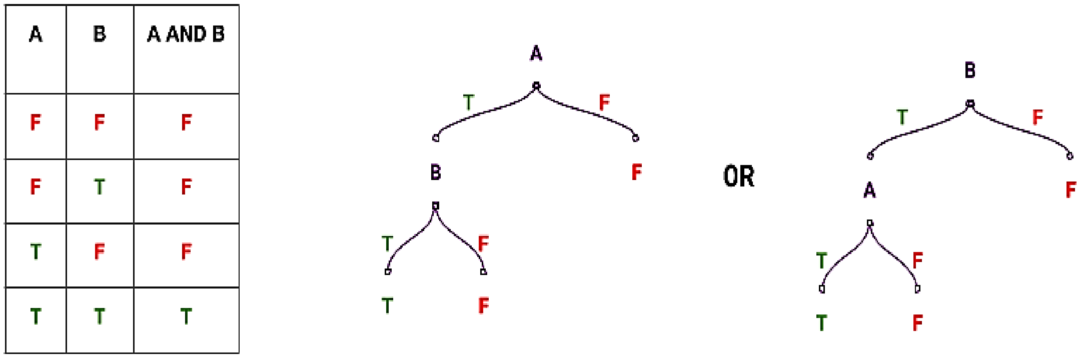
1. Pick the best attribute/feature. The best attribute is one which best splits or separates the data.
2. Ask the relevant question.
3. Follow the answer path.
4. Go to step 1 until you arrive to the answer.

The best split is one which separates two different labels into two sets.

Expressiveness of decision trees

Decision trees can represent any Boolean function of the input attributes. Let's use decision trees to perform the function of three Boolean gates AND, OR and XOR.

Boolean Function: AND

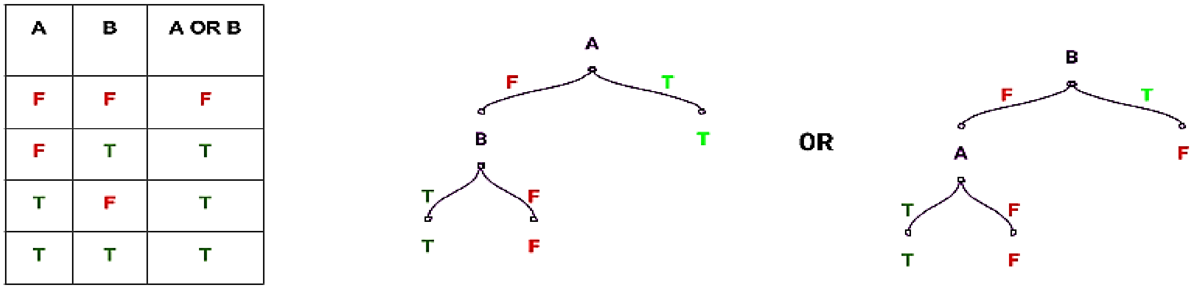


A	B	A AND B
F	F	F
F	T	F
T	F	F
T	T	T

Fig 3. Decision tree for an AND operation.

In Fig 3., we can see that there are two candidate concepts for producing the decision tree that performs the AND operation. Similarly, we can also produce a decision tree that performs the boolean OR operation.

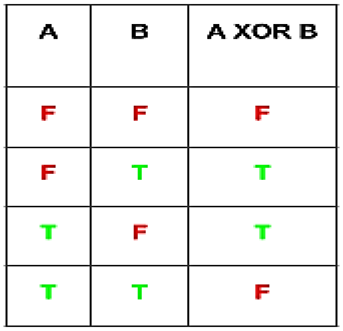
Boolean Function: OR



A	B	A OR B
F	F	F
F	T	T
T	F	T
T	T	T

Fig 4. Decision tree for an OR operation

Boolean Function: XOR



A	B	A XOR B
F	F	F
F	T	T
T	F	T
T	T	F

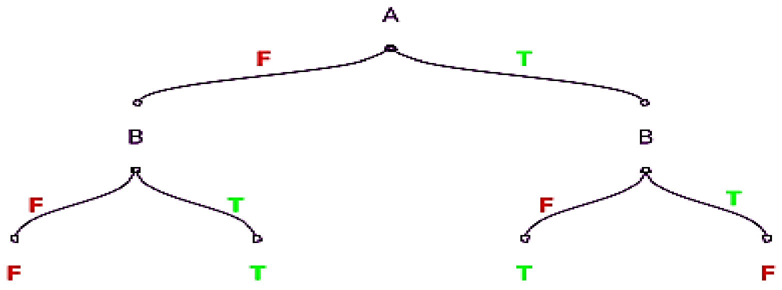


Fig 5.Decision tree for an XOR operation.

Let's produce a decision tree performing XOR functionality using 3 attributes:

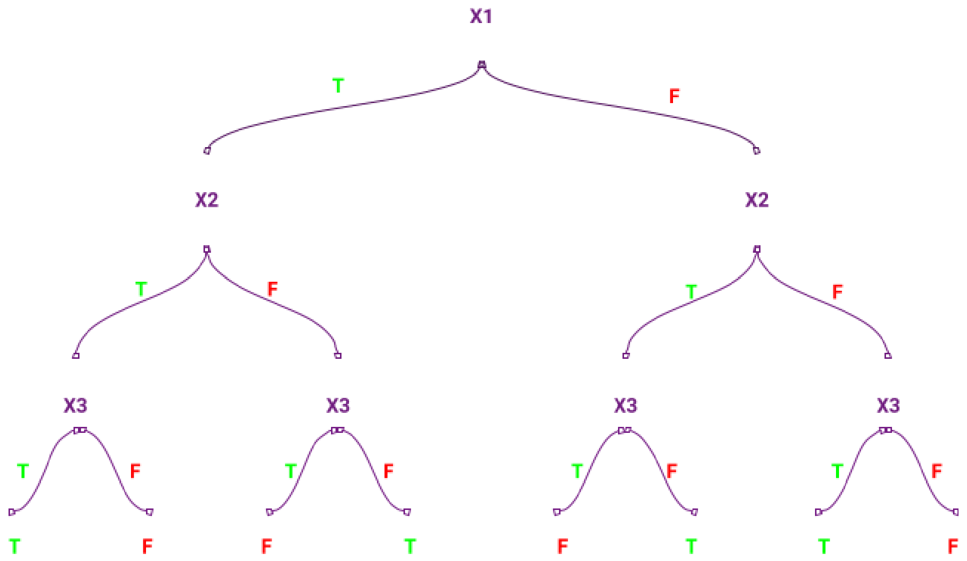


Fig 6. Decision tree for an XOR operation involving three operands

In the decision tree, shown above (Fig 6.), for three attributes there are 7 nodes in the tree, i.e., for $n = 3$, number of nodes = 2^3-1. Similarly, if we have n attributes, there are 2^n nodes (approx.) in the decision tree. So, the tree requires exponential number of nodes in the worst case.

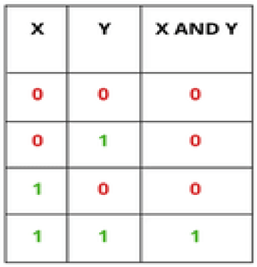
We can represent boolean operations using decision trees. But, what other kind of functions can we represent and if we search over the various possible decision trees to find the right one, how many decision trees do we have to worry about. Let's answer this question by finding out the possible number of decision trees we can generate given N different attributes (assuming the attributes are boolean). Since a truth table can be transformed into a decision tree, we will form a truth table of N attributes as input.

X1	X2	X3	....	XN	OUTPUT
T	T	T	...	T	
T	T	T	...	F	
...	...	...	...	...	
...	...	...	...	...	
...	...	...	...	...	
F	F	F	...	F	

The above truth table has 2^n rows (i.e. the number of nodes in the decision tree), which represents the possible combinations of the input attributes, and since each node can a hold a binary value, the number of ways to fill the values in the decision tree is ${2^{2^n}}$. Thus, the space of decision trees, i.e, the hypothesis space of the decision tree is very expressive because there are a lot of different functions it can represent. But, it also means one needs to have a clever way to search the best tree among them.

Decision tree boundary

Decision trees divide the feature space into axis-parallel rectangles or hyperplanes. Let’s demonstrate this with help of an example. Let’s consider a simple AND operation on two variables (see Fig 3.). Assume X and Y to be the coordinates on the x and y axes, respectively, and plot the possible values of X and Y (as seen the table below). Fig 7.represents the formation of the decision boundary as each decision is taken. We can see that as each decision is made, the feature space gets divided into smaller rectangles and more data points get correctly classified.



X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

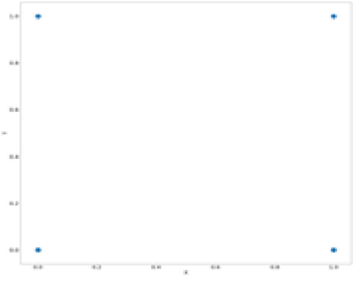


Fig 7. Animation showing the formation of the decision tree boundary for AND operation

The decision tree learning algorithm

The basic algorithm used in decision trees is known as the ID3 (by Quinlan) algorithm. The ID3 algorithm builds decision trees using a top-down, greedy approach. Briefly, the steps to the algorithm are: - Select the best attribute → A - Assign A as the decision attribute (test case) for the **NODE**. - For each value of A, create a new descendant of the **NODE**. - Sort the training examples to the appropriate descendant node leaf. - If examples are perfectly classified, then STOP else iterate over the new leaf nodes.

Now, the next big question is how to choose the best attribute. For ID3, we think of the best attribute in terms of which attribute has the most information gain, a measure that expresses how well an attribute splits that data into groups based on classification.

Pseudocode: ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples or until all attributes have been used.

The pseudocode assumes that the attributes are discrete and that the classification is binary. Examples are the training example. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Finally, it returns a decision tree that correctly classifies the given Examples.

ID3(Examples, Target_attribute, Attributes): - Create a root node for the tree. - If all Examplesare positive, return the single-node tree root, with positive labels. - If all Examples are negative, return the single-node tree root, with negative labels. - If Attributes is empty, return the single-node tree root, with the most common labels of the Target_attribute in Examples. - Otherwise, begin - A ← the attribute from Attributes that best* classifies Examples - The decision attribute for root ← A - For each possible value v_i, of A, - Add a new tree branch below root, corresponding to the test A = v_i - Let Examples_vi be the subset of Examples that have value v_i for A. - If Examples_vi is empty - Then, below this new branch add a leaf node with the labels having the most common value of Target_attribute in Examples. - Else, below this new branch add the subtree(or call the function) - ID3(Examples_vi, Target_attribute, Attributes-{A}) - End - Return root

Red-Black Tree

Introduction

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

In Red Black Tree, the color of a node is decided based on the properties of Red Black Tree. Every Red Black Tree has the following properties.

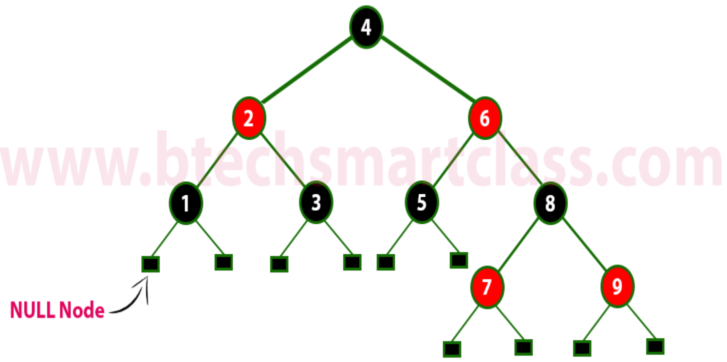
Properties of Red Black Tree

- **Property #1:** Red - Black Tree must be a Binary Search Tree.
- **Property #2:** The ROOT node must be colored BLACK.
- **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.

- **Property #5:** Every new node must be inserted with RED color.
- **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

Example

Following is a Red Black Tree which is created by inserting numbers from 1 to 9.



The above tree is a Red Black tree where every node is satisfying all the properties of Red Black Tree.

Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

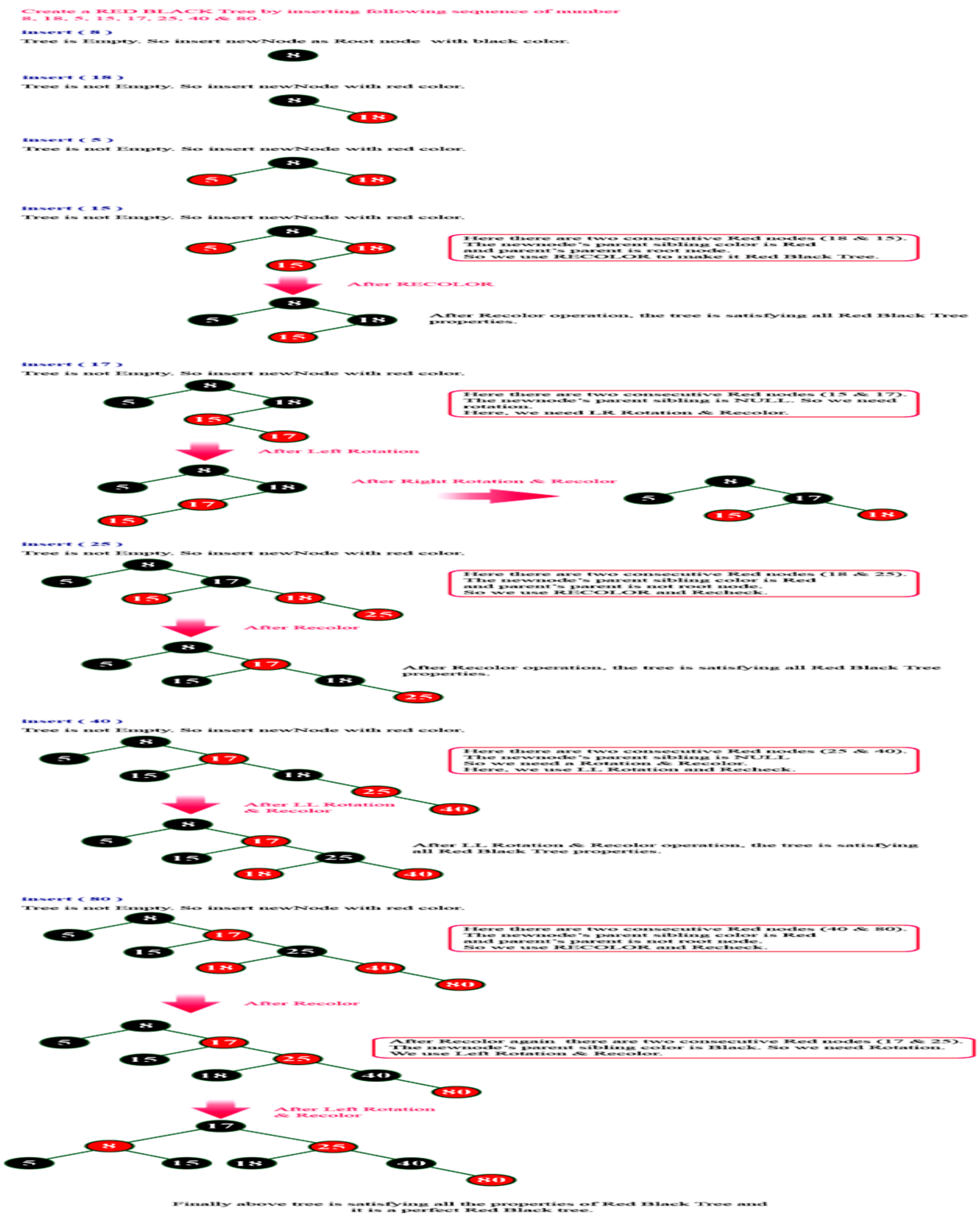
In a Red Black Tree, every new node must be inserted with color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform following operation to make it Red Black Tree.

- **1. Recolor**
- **2. Rotation**
- **3. Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node with color Red.
- **Step 4 -** If the parent of newNode is Black then exit from the operation.
- **Step 5 -** If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
- **Step 6 -** If it is colored Black or NULL then make suitable Rotation and Recolor it.
- **Step 7 -** If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Example



Create a RED BLACK Tree by inserting following sequence of number 8, 18, 5, 15, 17, 25, 40 & 80.

insert (8)

Tree is Empty. So insert newNode as Root node with black color.

insert (18)

Tree is not Empty. So insert newNode with red color.

insert (5)

Tree is not Empty. So insert newNode with red color.

insert (15)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (18 & 15). The newnode's parent sibling color is Red and parent's parent is root node. So we use RECOLOR to make it Red Black Tree.

After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (17)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (15 & 17). The newnode's parent sibling is NULL. So we need rotation. Here, we need LR Rotation & Recolor.

insert (25)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (18 & 25). The newnode's parent sibling color is Red and parent's parent is not root node. So we use RECOLOR and Recheck.

After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (40)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (25 & 40). The newnode's parent sibling is NULL So we need a Rotation & Recolor. Here, we use LL Rotation and Recheck.

After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (40 & 80). The newnode's parent sibling color is Red and parent's parent is not root node. So we use RECOLOR and Recheck.

After Recolor again there are two consecutive Red nodes (17 & 25). The newnode's parent sibling color is Black. So we need Rotation. We use Left Rotation & Recolor.

Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

After RECOLOR
After Left Rotation
After Right Rotation & Recolor
After Recolor
After LL Rotation & Recolor
After Recolor
After Left Rotation & Recolor

Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation we need to check with the Red Black Tree properties. If any of the properties is violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Red-black tree retrieval:

Retrieving a node from a red-black tree doesn’t require more than the use of the BST procedure, which takes O(log n) time.

Program for Red Black Trees

```
#include<iostream>

using namespace std;

struct node
{
       int key;
       node *parent;
       char color;
       node *left;
       node *right;
};
class RBtree
{
      node *root;
      node *q;
   public :
      RBtree()
      {
             q=NULL;
             root=NULL;
      }
      void insert();
      void insertfix(node *);
```

```
    void leftrotate(node *);
    void rightrotate(node *);
    void del();
    node* successor(node *);
    void delfix(node *);
    void disp();
    void display( node *);
    void search();
};
void RBtree::insert()
{
   int z,i=0;
   cout<<"\nEnter key of the node to be inserted: ";
   cin>>z;
   node *p,*q;
   node *t=new node;
   t->key=z;
   t->left=NULL;
   t->right=NULL;
   t->color='r';
   p=root;
   q=NULL;
   if(root==NULL)
   {
       root=t;
       t->parent=NULL;
   }
   else
   {
      while(p!=NULL)
      {
          q=p;
          if(p->key<t->key)
             p=p->right;
          else
             p=p->left;
```

```
        }
        t->parent=q;
        if(q->key<t->key)
            q->right=t;
        else
            q->left=t;
    }
    insertfix(t);
}
void RBtree::insertfix(node *t)
{
    node *u;
    if(root==t)
    {
        t->color='b';
        return;
    }
    while(t->parent!=NULL&&t->parent->color=='r')
    {
         node *g=t->parent->parent;
         if(g->left==t->parent)
         {
                    if(g->right!=NULL)
                    {
                         u=g->right;
                         if(u->color=='r')
                         {
                             t->parent->color='b';
                             u->color='b';
                             g->color='r';
                             t=g;
                         }
                    }
                    else
                    {
                       if(t->parent->right==t)
```

```
                   {
                       t=t->parent;
                       leftrotate(t);
                   }
                   t->parent->color='b';
                   g->color='r';
                   rightrotate(g);
                }
        }
        else
        {
                if(g->left!=NULL)
                {
                    u=g->left;
                    if(u->color=='r')
                    {
                        t->parent->color='b';
                        u->color='b';
                        g->color='r';
                        t=g;
                    }
                }
                else
                {
                   if(t->parent->left==t)
                   {
                         t=t->parent;
                         rightrotate(t);
                   }
                   t->parent->color='b';
                   g->color='r';
                   leftrotate(g);
                }
        }
        root->color='b';
    }
```

```
}

void RBtree::del()
{
    if(root==NULL)
    {
         cout<<"\nEmpty Tree." ;
         return ;
    }
    int x;
    cout<<"\nEnter the key of the node to be deleted: ";
    cin>>x;
    node *p;
    p=root;
    node *y=NULL;
    node *q=NULL;
    int found=0;
    while(p!=NULL&&found==0)
    {
         if(p->key==x)
            found=1;
         if(found==0)
         {
              if(p->key<x)
                 p=p->right;
              else
                 p=p->left;
         }
    }
    if(found==0)
    {
          cout<<"\nElement Not Found.";
          return ;
    }
    else
    {
```

```
    cout<<"\nDeleted Element: "<<p->key;
    cout<<"\nColour: ";
    if(p->color=='b')
 cout<<"Black\n";
else
 cout<<"Red\n";

    if(p->parent!=NULL)
         cout<<"\nParent: "<<p->parent->key;
    else
         cout<<"\nThere is no parent of the node.  ";
    if(p->right!=NULL)
         cout<<"\nRight Child: "<<p->right->key;
    else
         cout<<"\nThere is no right child of the node.  ";
    if(p->left!=NULL)
         cout<<"\nLeft Child: "<<p->left->key;
    else
         cout<<"\nThere is no left child of the node.  ";
    cout<<"\nNode Deleted.";
    if(p->left==NULL||p->right==NULL)
        y=p;
    else
        y=successor(p);
    if(y->left!=NULL)
        q=y->left;
    else
    {
        if(y->right!=NULL)
            q=y->right;
        else
            q=NULL;
    }
    if(q!=NULL)
        q->parent=y->parent;
    if(y->parent==NULL)
```

```
            root=q;
         else
         {
            if(y==y->parent->left)
               y->parent->left=q;
            else
               y->parent->right=q;
         }
         if(y!=p)
         {
            p->color=y->color;
            p->key=y->key;
         }
         if(y->color=='b')
            delfix(q);
      }
}

void RBtree::delfix(node *p)
{
   node *s;
   while(p!=root&&p->color=='b')
   {
         if(p->parent->left==p)
         {
                s=p->parent->right;
                if(s->color=='r')
                {
                       s->color='b';
                       p->parent->color='r';
                       leftrotate(p->parent);
                       s=p->parent->right;
                }
                if(s->right->color=='b'&&s->left->color=='b')
                {
                       s->color='r';
            p=p->parent;
        }
        else
        {
           if(s->right->color=='b')
           {
                s->left->color=='b';
                s->color='r';
                rightrotate(s);
                s=p->parent->right;
           }
           s->color=p->parent->color;
           p->parent->color='b';
           s->right->color='b';
           leftrotate(p->parent);
           p=root;
        }
    }
    else
    {
        s=p->parent->left;
        if(s->color=='r')
        {
            s->color='b';
            p->parent->color='r';
            rightrotate(p->parent);
            s=p->parent->left;
        }
        if(s->left->color=='b'&&s->right->color=='b')
        {
            s->color='r';
            p=p->parent;
        }
        else
        {
            if(s->left->color=='b')
```

```
                    {
                         s->right->color='b';
                         s->color='r';
                         leftrotate(s);
                         s=p->parent->left;
                    }
                    s->color=p->parent->color;
                    p->parent->color='b';
                    s->left->color='b';
                    rightrotate(p->parent);
                    p=root;
               }
         }
      p->color='b';
      root->color='b';
    }
}

void RBtree::leftrotate(node *p)
{
    if(p->right==NULL)
         return ;
    else
    {
         node *y=p->right;
         if(y->left!=NULL)
         {
               p->right=y->left;
               y->left->parent=p;
         }
         else
               p->right=NULL;
         if(p->parent!=NULL)
             y->parent=p->parent;
         if(p->parent==NULL)
             root=y;
```

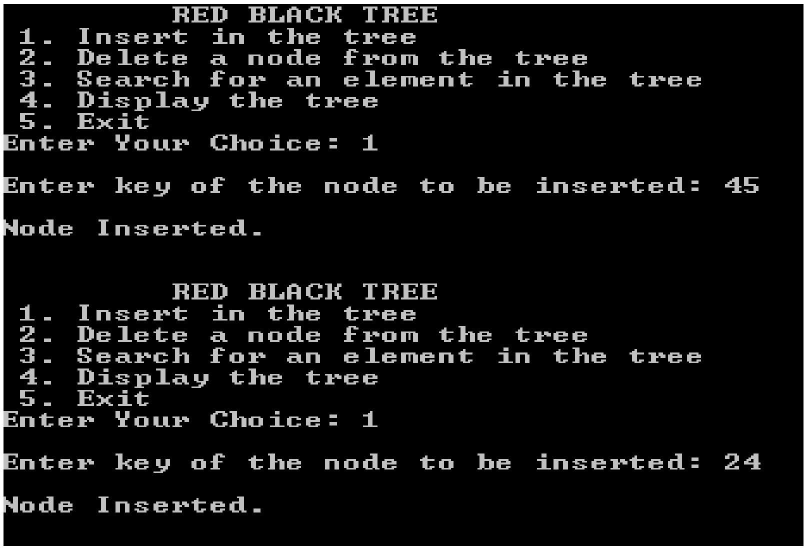
```
           else
           {
              if(p==p->parent->left)
                     p->parent->left=y;
              else
                     p->parent->right=y;
           }
           y->left=p;
           p->parent=y;
      }
}
void RBtree::rightrotate(node *p)
{
     if(p->left==NULL)
          return ;
     else
     {
         node *y=p->left;
         if(y->right!=NULL)
         {
                  p->left=y->right;
                  y->right->parent=p;
         }
         else
                 p->left=NULL;
         if(p->parent!=NULL)
                 y->parent=p->parent;
         if(p->parent==NULL)
              root=y;
         else
         {
             if(p==p->parent->left)
                   p->parent->left=y;
             else
                   p->parent->right=y;
         }
```

```
        y->right=p;
        p->parent=y;
     }
}

node* RBtree::successor(node *p)
{
      node *y=NULL;
     if(p->left!=NULL)
     {
        y=p->left;
        while(y->right!=NULL)
             y=y->right;
     }
     else
     {
        y=p->right;
        while(y->left!=NULL)
             y=y->left;
     }
     return y;
}

void RBtree::disp()
{
     display(root);
}
void RBtree::display(node *p)
{
     if(root==NULL)
     {
          cout<<"\nEmpty Tree.";
          return ;
     }
     if(p!=NULL)
     {
            cout<<"\n\t NODE: ";
            cout<<"\n Key: "<<p->key;
            cout<<"\n Colour: ";
   if(p->color=='b')
    cout<<"Black";
   else
    cout<<"Red";
            if(p->parent!=NULL)
                  cout<<"\n Parent: "<<p->parent->key;
            else
                  cout<<"\n There is no parent of the node.  ";
            if(p->right!=NULL)
                  cout<<"\n Right Child: "<<p->right->key;
            else
                  cout<<"\n There is no right child of the node.  ";
            if(p->left!=NULL)
                  cout<<"\n Left Child: "<<p->left->key;
            else
                  cout<<"\n There is no left child of the node.  ";
            cout<<endl;
   if(p->left)
   {
             cout<<"\n\nLeft:\n";
    display(p->left);
   }
   /*else
    cout<<"\nNo Left Child.\n";*/
   if(p->right)
   {
    cout<<"\n\nRight:\n";
             display(p->right);
   }
   /*else
    cout<<"\nNo Right Child.\n"*/
    }
}
```

```
void RBtree::search()
{
    if(root==NULL)
    {
        cout<<"\nEmpty Tree\n" ;
        return  ;
    }
    int x;
    cout<<"\n Enter key of the node to be searched: ";
    cin>>x;
    node *p=root;
    int found=0;
    while(p!=NULL&& found==0)
    {
         if(p->key==x)
            found=1;
         if(found==0)
         {
             if(p->key<x)
                 p=p->right;
             else
                 p=p->left;
         }
    }
    if(found==0)
        cout<<"\nElement Not Found.";
    else
    {
            cout<<"\n\t FOUND NODE: ";
            cout<<"\n Key: "<<p->key;
            cout<<"\n Colour: ";
   if(p->color=='b')
    cout<<"Black";
   else
    cout<<"Red";
            if(p->parent!=NULL)
```

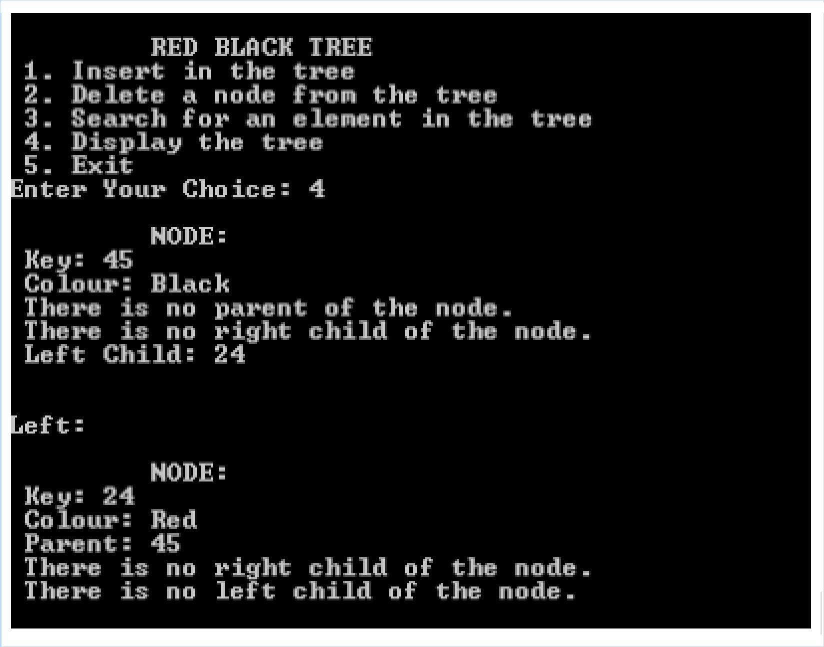
```
                cout<<"\n Parent: "<<p->parent->key;
            else
                cout<<"\n There is no parent of the node.  ";
            if(p->right!=NULL)
                cout<<"\n Right Child: "<<p->right->key;
            else
                cout<<"\n There is no right child of the node.  ";
            if(p->left!=NULL)
                cout<<"\n Left Child: "<<p->left->key;
            else
                cout<<"\n There is no left child of the node.  ";
            cout<<endl;

     }
}
int main()
{
   int ch,y=0;
   RBtree obj;
   do
   {
            cout<<"\n\t RED BLACK TREE " ;
            cout<<"\n 1. Insert in the tree ";
            cout<<"\n 2. Delete a node from the tree";
            cout<<"\n 3. Search for an element in the tree";
            cout<<"\n 4. Display the tree ";
            cout<<"\n 5. Exit " ;
            cout<<"\nEnter Your Choice: ";
            cin>>ch;
            switch(ch)
            {
                  case 1 : obj.insert();
                        cout<<"\nNode Inserted.\n";
                        break;
                  case 2 : obj.del();
                        break;
                  case 3 : obj.search();
                           break;
                  case 4 : obj.disp();
                           break;
                  case 5 : y=1;
                           break;
                  default : cout<<"\nEnter a Valid Choice.";
            }
            cout<<endl;

    }while(y!=1);
    return 1;
}
```

// Output of the above program.



```
               RED BLACK TREE
 1. Insert in the tree
 2. Delete a node from the tree
 3. Search for an element in the tree
 4. Display the tree
 5. Exit
Enter Your Choice: 1

Enter key of the node to be inserted: 45

Node Inserted.


               RED BLACK TREE
 1. Insert in the tree
 2. Delete a node from the tree
 3. Search for an element in the tree
 4. Display the tree
 5. Exit
Enter Your Choice: 1

Enter key of the node to be inserted: 24

Node Inserted.
```

Red Black Tree Using C++



```
          RED BLACK TREE
 1. Insert in the tree
 2. Delete a node from the tree
 3. Search for an element in the tree
 4. Display the tree
 5. Exit
Enter Your Choice: 4

          NODE:
 Key: 45
 Colour: Black
 There is no parent of the node.
 There is no right child of the node.
 Left Child: 24


Left:

          NODE:
 Key: 24
 Colour: Red
 Parent: 45
 There is no right child of the node.
 There is no left child of the node.
```

POSSIBLE QUESTIONS

UNIT III

2 Mark Questions:

1. What is a Decision tree?

2. Define Construction of Decision Tree.

3. How to represent a Decision tree?

4. Define Red-black tree.

5. What are the properties of red-black tree?

6 Mark Questions:

1. Explain about Decision Tree with example.

2. How to build a Decision Tree. Explain with example.

3. Explain about Representation of Decision Tree.

4. Describe about strengths and weakness of decision tree approach.

5. Explain about Decision Tree Rules.

6. Explain about Red-Black tree with example.

7. Explain about implementation of inserting a node using Red-black tree.

8. Explain about implementation of deleting a node using Red-black tree.

9. Describe the Red-black tree rules with example.

10. Explain in detail about the properties of Red-black tree.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act, 1956)

Coimbatore-641021

Department of Computer Science

II B.Sc(CS) (BATCH 2017-2020)

Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS **ONE MARK QUESTIONS**

UNIT-3

S.NO	QUESTIONS	OPT1	OPT2	OPT3	OPT4	ANSWER
1	Algorithms should have explicitly defined set of _______	Inputs	Outputs	Inputs and Outputs	Instructions	Inputs and Outputs
2	_______ means to establish the amount of resources.	Efficiency	Module	Program	Process	Efficiency
3	_______ is to verify if the algorithm leads to the solution of the problem.	Efficiency	Flexibility	Durability	Correctness	Correctness
4	_______ stage recursively combines the sub problems.	Merge	Divide	Break	Solve	Merge
5	_______ sort is a comparison based algorithm.	Bubble	Insertion	a & b	None of these	a & b
6	The minimum number of steps taken on any instance is called as _____.	Best-case	Worst-case	Average-case	a & b	Best-case
7	The algorithm's ______ is the set of values corresponding to all the variables.	Process	State	Definition	d. Technique	State
8	Bubble sort is not suitable for _______	larger data set	Smaller data set	Medium data set	Both b & c	larger data set
9	______ means passing through nodes in a specific order.	Interchanging	Traversing	Dividing	Splitting	Traversing
10	Level of a node represents the _______ of a node.	Priority	Object	Generation	Both a & b	Generation
11	______ represents a value of a node.	Key	Instance	Type	Variable	Key

12	Red-Black tree is one of the _______ binary search tree.	Balanced	Unbalanced	Different	Similar	Balanced
13	The method gives a global view of a problem.	Manual	Analysis	aggregate	Recurrence	aggregate
14	In _____ method different charges are assigned to different operations.	Accounting	Recurrence	Aggregation	All of these	Accounting
15	The spanning tree cannot be _____.	Union	Intersection	Disconnected	Updated	Disconnected
16	_______ method represents the prepaid work as potential energy.	Graph theory	Potential	Aggregation	Accounting	Potential
17	The ______ string-matching procedure can be interpreted graphically.	naive	Pattern	KMP	Automative	naive
18	Rabin and Karp method is applied for _______ pattern matching.	3D	1D	2D	4D	2D
19	_______ algorithm is a Linear time string-matching algorithm.	Greedy	b. Dynamic	Descriptive	KMP	KMP
20	The _____ case running time of Rabin and Karp is high.	Average	Minimum	Maximum	Worst	Average
21	X is a root then X is the ______ of its children.	sub tree	Parent	Sibilings	subordinate	Parent
22	The children of the same parent are called ___________________.	sibiling	leaf	child	subtree	sibiling
23	___________of a node are all the nodes along the path form the root to that node.	Degree	sub tree	Ancestors	parent	Ancestors
24	The ______________of a tree is defined to be a maximum level of any node in the tree.	weight	length	breath	height	height

25	A___________ is a set of n ≥ 0 disjoint trees	Group	forest	Branch	sub tree	forest
26	A tree with any node having at most two branches is called a _____________.	branched tree	sub tree	binary tree	forest	binary tree
27	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
28	Data structure represents the hierarchical relationships between individual data item is known as __________.	Root	Node	Tree	Address	Tree
29	Node at the highest level of the tree is known as _______.	Child	Root	Sibiling	Parent	Root
30	The root of the tree is the _______of all nodes in the tree.	Child	Parent	Ancestor	Head	Ancestor
31	_____is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree	Subtree
32	A node with no children is called _________.	Root Node	Branch	Leaf Node	Null tree	Leaf Node
33	In a tree structure a link between parent and child is called _______	Branch	Root	Leaf	Subtree	Branch
34	Height – balanced trees are also referred as as ___________trees.	AVL trees	Binary Trees	Subtree	Branch Tree	AVL trees
35	Visiting each node in a tree exactly once is called _________	searching	travering	walk through	path	travering
36	In________traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order	PreOrder

37	In________traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order	Inorder
38	In________traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order	PostOrder
39	Inorder traversal is also sometimes called______	Symmetric Order	End Order	PreOrder	PostOrder	Symmetric Order
40	Postorder traversal is also sometimes called______	Symmetric Order	End Order	PreOrder	PostOrder	End Order
41	Nodes of any level are numbered from _________	Left to right	Right to Left	Top to Bottom	Bottom to Top	Left to right
42	________ search involves only addition and subtraction.	binary	fibonacci	sequential	non sequential	fibonacci
43	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
44	Binary trees are used in ______ sorting.	quick sort	merge sort	heap sort	lrsort	heap sort
45	The ____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch	Root
46	In Threaded Binary Tree ,LCHILD(P) is a normal pointer When LBIT(P) = ____	1	2	3	0	1
47	In Threaded Binary Tree ,LCHILD(P) is a Thread When LBIT(P) = ____	1	2	3	0	0
48	In Threaded Binary Tree ,RCHILD(P) is a normal pointer When RBIT(P) = ____	2	1	3	0	1

49	In Threaded Binary Tree ,RCHILD(P) is a Thread When LBIT(P) = ____	1	2	0	4	0
50	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	None of the above	Binary search
51	______ algorithm can be used only with	Linear search	Binary search	insertion sort	merge sort	Binary search
52	________ search involves comparision	Linear search	Binary search	fibonacci search	None of the above	Linear search
53	Binary search algorithm in a list of n	$O(\log_2 n)$	$O(n)$	$O(n^3)$	$O(n^2)$	$O(\log_2 n)$
54	_____ is used for decision making in	trees	graphs	linked lists	array	trees
55	The Linear search algorithm in a list of n element takes ________ time to compare in worst case.	constant	linear	quadratic	exponential	linear
56	Which of these is an application of trees.	Finding minimum cost spanning tree	Decision tree	Storage management	Job sequencing	Finding minimum cost spanning tree
57	________ is an operation performed	union	sort	rename	traverse	union
58	In sets _______ is used to find the set containing the element i	subset(i)	Disjoin(i)	Union(i)	Find(i)	Find(i)
59	Sets are represented as _____	arrays	linked lists	graphs	trees	linked lists
60	_________ is an example of application of trees in decision making.	Binay search	Optimal merge pattern	Eight Coins problem	Huffman's Message coding	Binay search

UNIT IV

SYLLABUS

Advanced Analysis Technique: Amortized analysis **Graphs:** Graph Algorithms– Breadth First Search, Depth First Search and its Applications, Minimum Spanning Trees.

Amortized Analysis

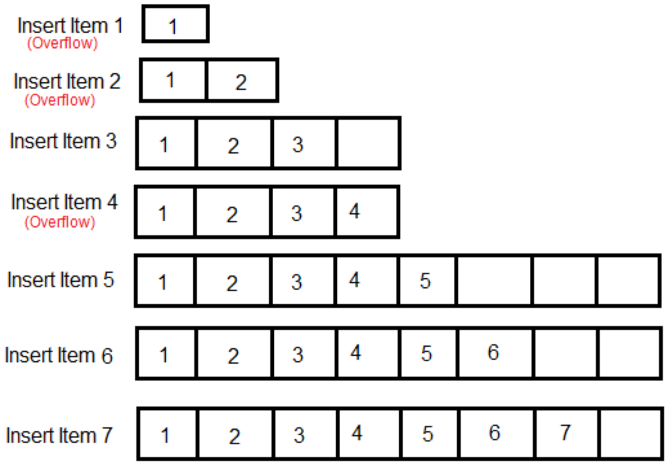
Amortized analysis is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time per operation, rather than per algorithm, can be too pessimistic.

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

Initially table is empty and size is 0



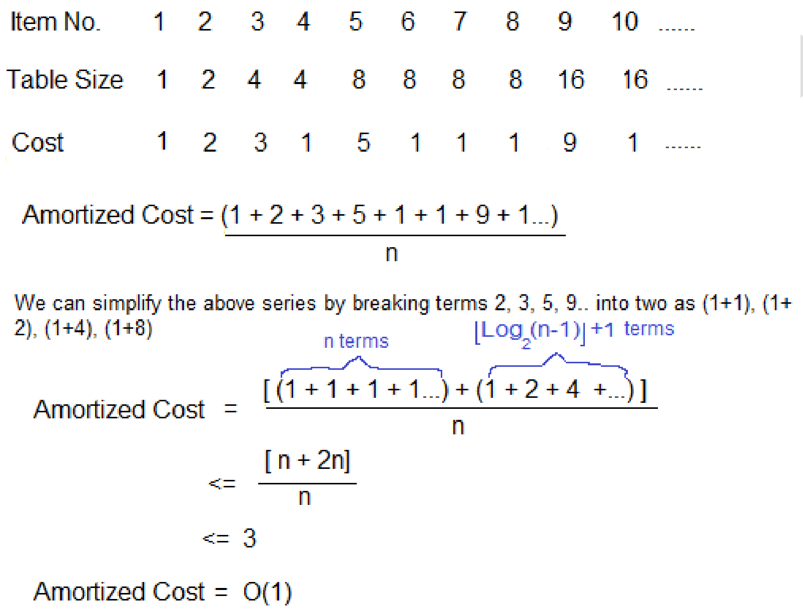
Next overflow would happen when we insert 9, table size would become 16

The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.

1) Allocate memory for a larger table of size, typically twice the old table.
2) Copy the contents of old table to new table.
3) Free the old table.

If the table has space available, we simply insert new item in available space.

What is the time complexity of n insertions using the above scheme?
If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is O(n^2). This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take Θ(n) time.



Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1...)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{n\ \text{terms}} + \overbrace{(1 + 2 + 4\ +...)}^{\lfloor Log_2(n-1)\rfloor+1\ \text{terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++, ArrayList in Java.

Following are few important notes:

1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can

spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

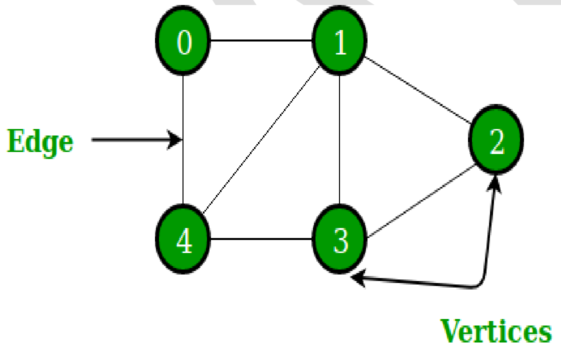
2) The above Amortized Analysis done for Dynamic Array example is called Aggregate Method. There are two more powerful ways to do Amortized analysis called Accounting Method and Potential Method. We will be discussing the other two methods in separate posts.

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing.

Graphs:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes



.

In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

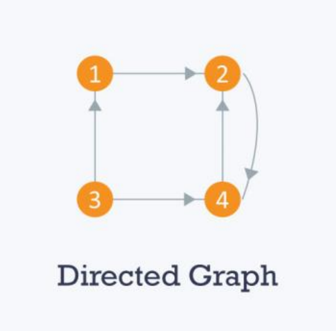
Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

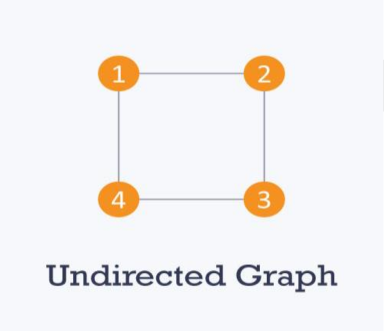
A graph is an abstract notation used to represent the connection between pairs of objects. A graph consists of −

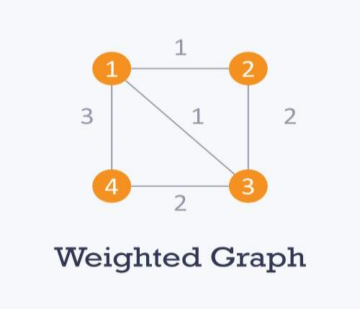
- Vertices − Interconnected objects in a graph are called vertices. Vertices are also known as nodes.
- Edges − Edges are the links that connect the vertices.

There are three types of graphs −

- Directed graph − In a directed graph, edges have direction, i.e., edges go from one vertex to another.
- Undirected graph − In an undirected graph, edges have no direction.
- Weighted: In a weighted graph, each edge is assigned a weight or cost.







Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

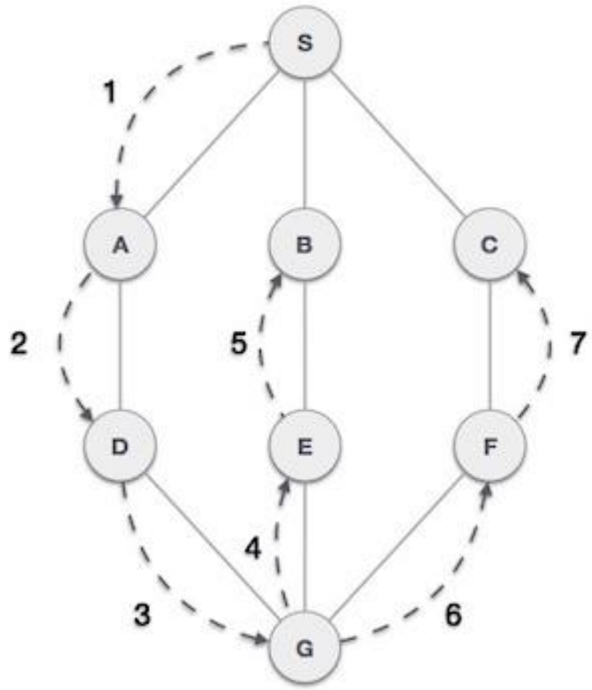
During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

The breadth first search (**BFS**) and the depth first search (**DFS**) are the two **algorithms** used for traversing and searching a node in a graph.

Depth First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

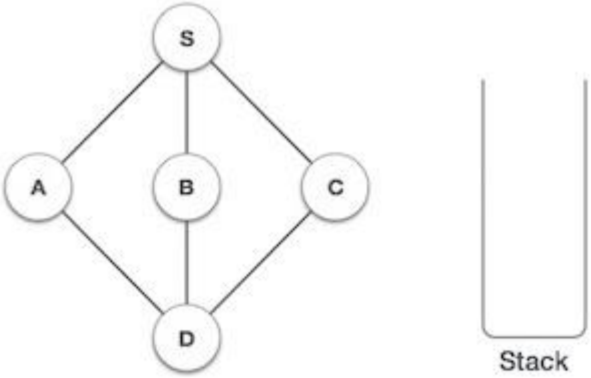
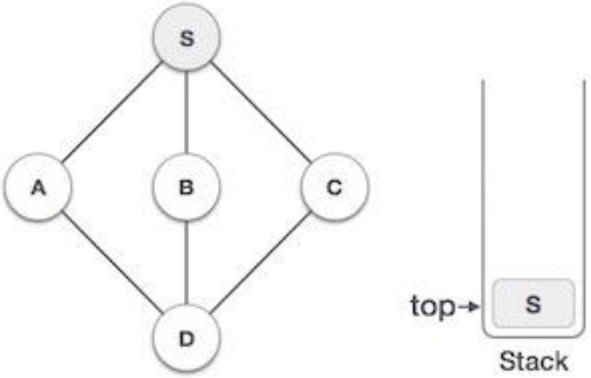
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

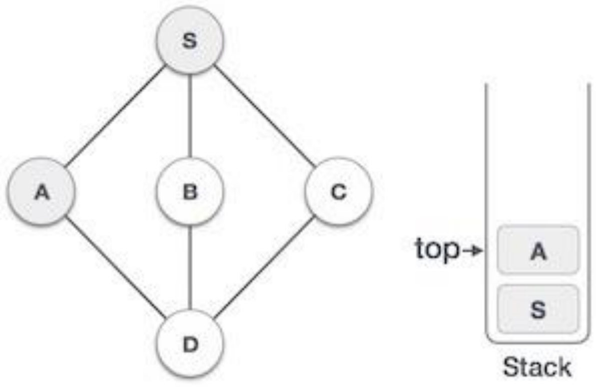
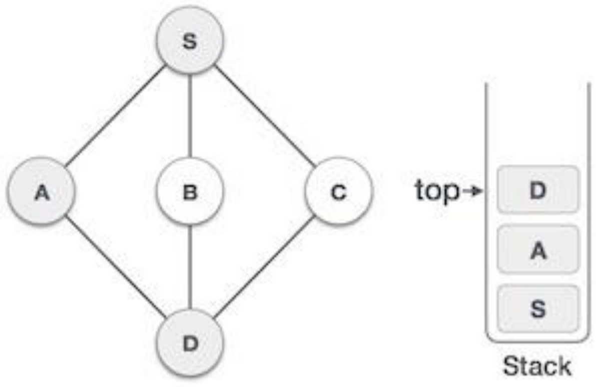
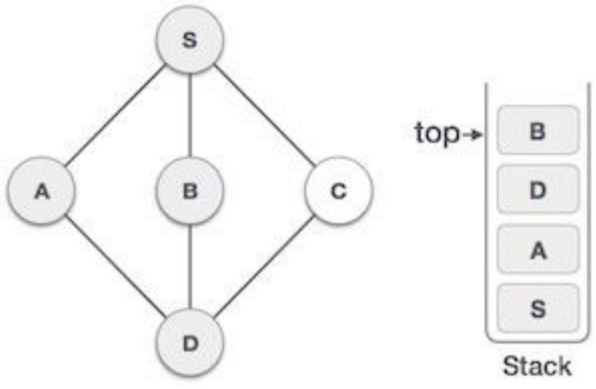


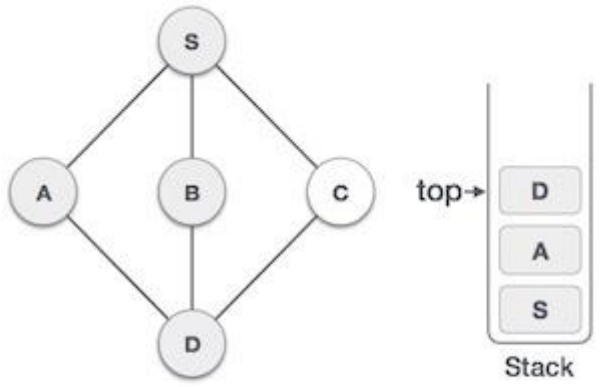
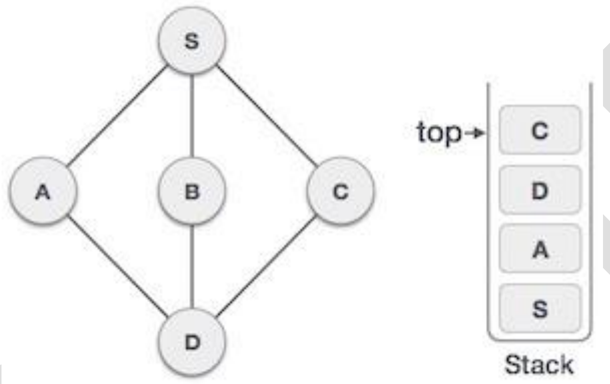
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B**and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Pseudocode

```
DFS-iterative (G, s):                                   //Where G is graph and s is source vertex
  let S be stack
  S.push( s )            //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
```

```
    for all neighbours w of v in Graph G:
       if w is not visited :
              S.push( w )
             mark w as visited
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
       if w is not visited:
          DFS-recursive(G, w)
```

Applications of Depth First Search:

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

1) **For an unweighted graph**, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) **Detecting cycle in a graph**
A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) **Path Finding**
We can specialize the DFS algorithm to find a path between two given vertices u and z.
i) Call DFS(G, u) with u as the start vertex.
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
iii) As soon as destination vertex z is encountered, return the path as the
contents of the stack

4) **Topological Sorting**
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers.

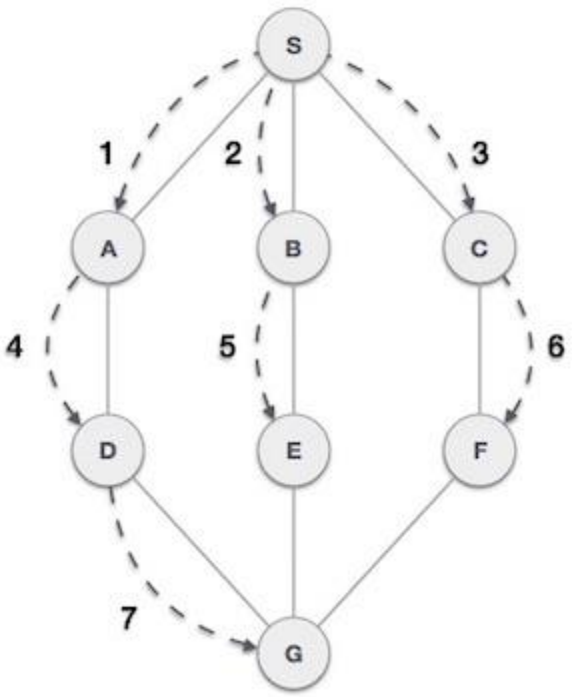
5) **To test if a graph is bipartite**
we can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) **Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based algo for finding Strongly Connected Components)

7) **Solving puzzles** with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

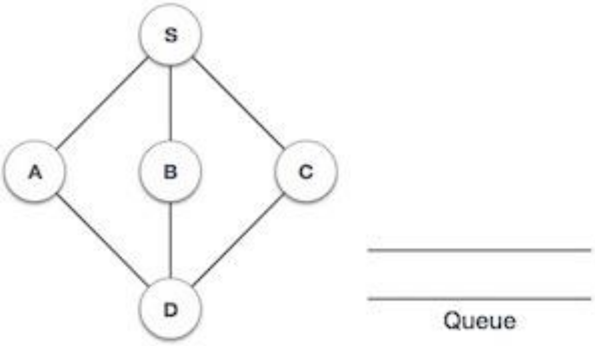
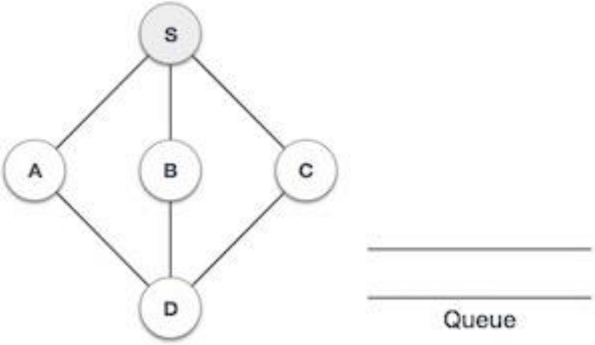
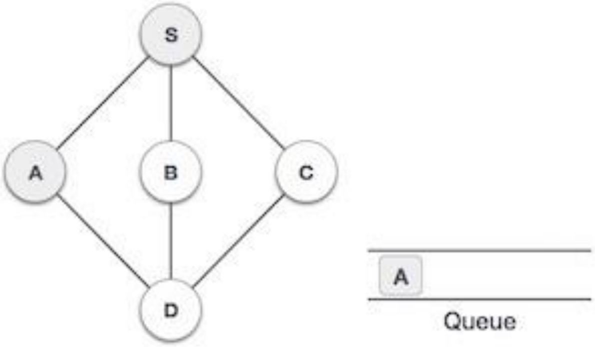
Breadth First Search:

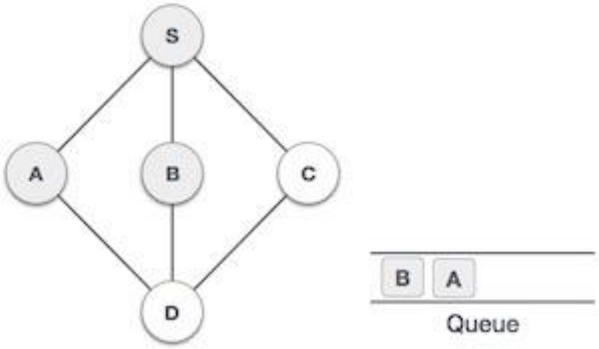
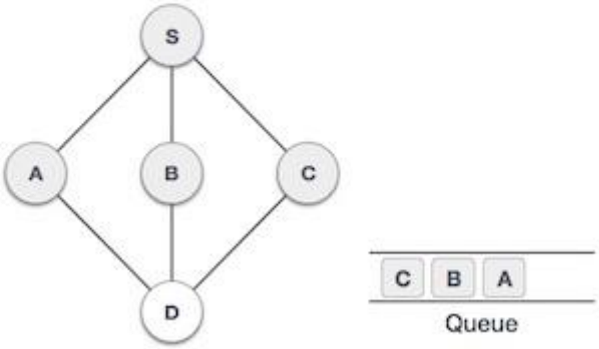
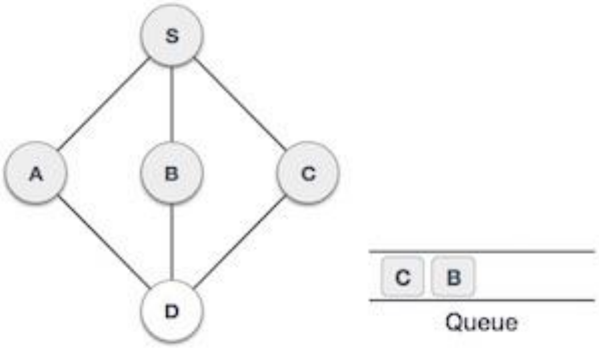
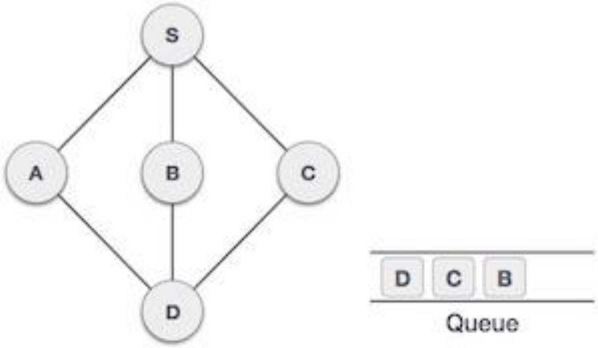
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.
- Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S(starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.

4		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Pseudocode

```
BFS (G, s)                   //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v  =  Q.dequeue( )

       //processing all the neighbours of v
       for all neighbours w of v in Graph G
           if w is not visited
                  Q.enqueue( w )            //Stores w in Q to further visit its neighbour
                  mark w as visited.
```

Applications of Breadth First Traversal:

1) **Shortest Path and Minimum Spanning Tree for unweighted graph** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2**) Peer to Peer Networks**. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) **Crawlers in Search Engines**: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

4) **Social Networking Websites**: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) **GPS Navigation systems**: Breadth First Search is used to find all neighboring locations.

6) **Broadcasting in Network**: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) **In Garbage Collection**: Breadth First Search is used in copying garbage collection using Cheney's algorithm.

8) **Cycle detection in undirected graph**: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) **Ford–Fulkerson algorithm** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) **To test if a graph is Bipartite** We can either use Breadth First or Depth First Traversal.

11) **Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) **Finding all nodes within one connected component**: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Minimum Spanning Tree

What is a Spanning Tree?

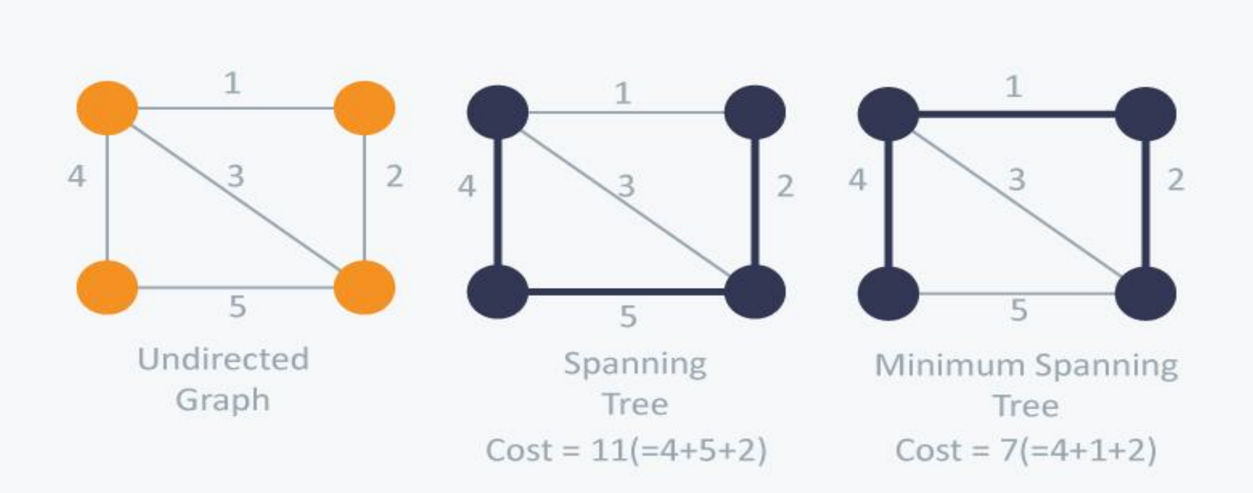
Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G(that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal’s Algorithm

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

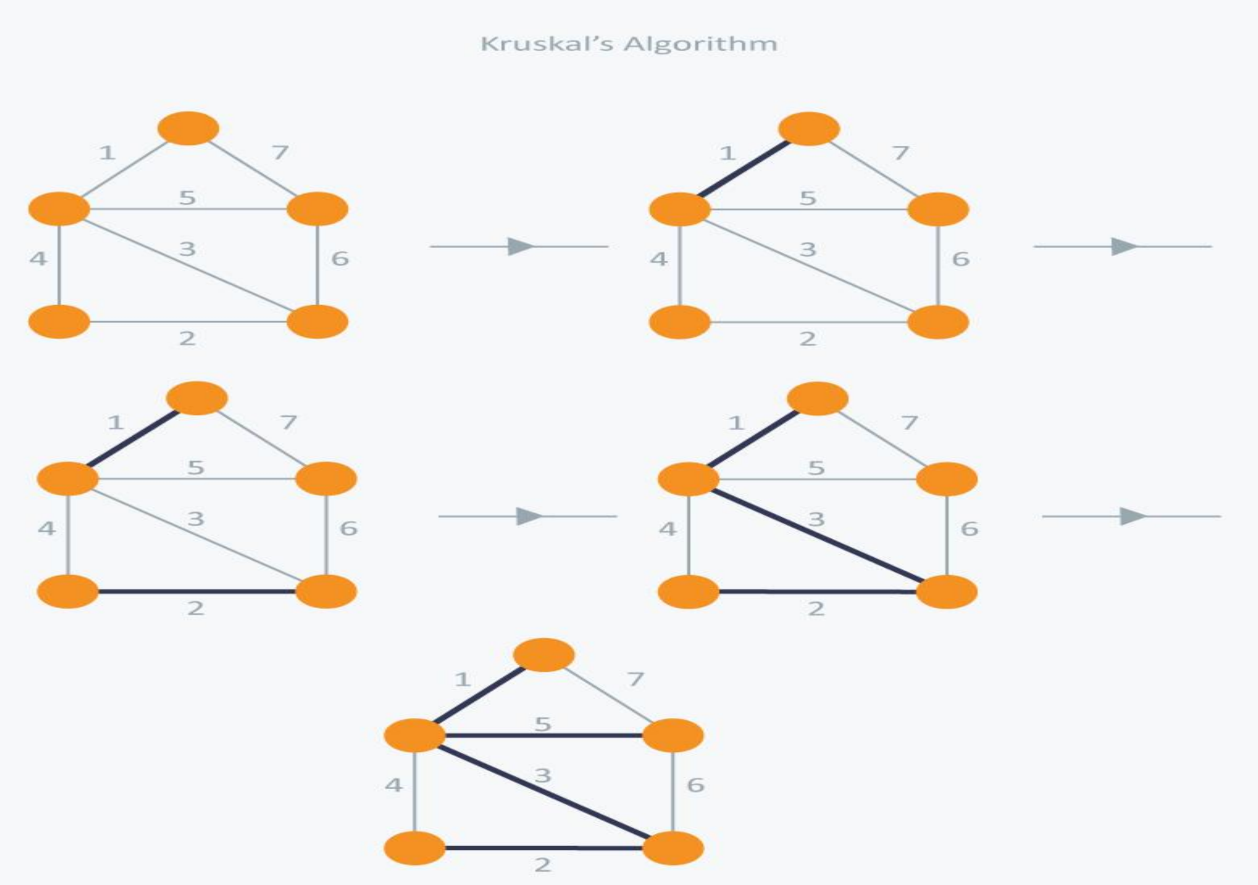
- Sort the graph edges with respect to their weights.

- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **"Disjoint Sets":**
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:



In Kruskal’s algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can’t have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Implementation:

```
#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>

using namespace std;

const int MAX = 1e4 + 5;

int id[MAX], nodes, edges;

pair <long long, pair<int, int> > p[MAX];

void initialize()

{

    for(int i = 0;i < MAX;++i)

        id[i] = i;

}
```

```
int root(int x)
{
    while(id[x] != x)
    { id[x] = id[id[x]];
        x = id[x];    }
    return x;}
void union1(int x, int y){
    int p = root(x);
    int q = root(y);
    id[p] = id[q];}
long long kruskal(pair<long long, pair<int, int> > p[]){
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)    {
        // Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))  {
            minimumCost += cost;

            union1(x, y);        }        }    return minimumCost;}

int main()

{ int x, y;

    long long weight, cost, minimumCost;

    initialize();

    cin >> nodes >> edges;

    for(int i = 0;i < edges;++i)    {

        cin >> x >> y >> weight;

        p[i] = make_pair(weight, make_pair(x, y));    }

    // Sort the edges in the ascending order

    sort(p, p + edges);

    minimumCost = kruskal(p);

    cout << minimumCost << endl;    return 0;}
```

Time Complexity:
In Kruskal’s algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

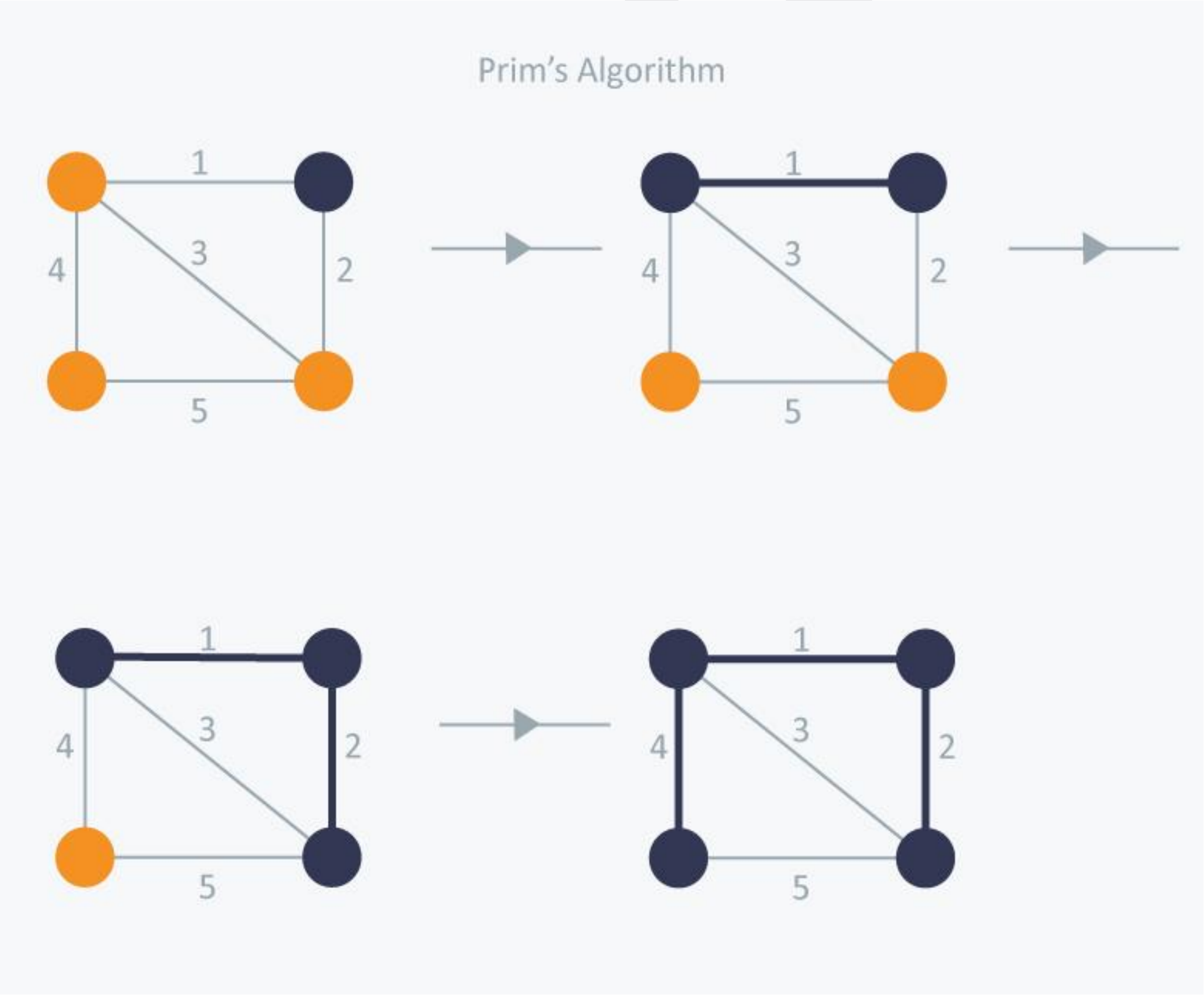
Prim’s Algorithm

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Example

Using Prim's algorithm, we can start from any vertex, let us start from vertex **1**.

Vertex **3** is connected to vertex **1** with minimum edge cost, hence edge **(1, 2)** is added to the spanning tree.

Next, edge **(2, 3)** is considered as this is the minimum among edges **{(1, 2), (2, 3), (3, 4), (3, 7)}**.

In the next step, we get edge **(3, 4)** and **(2, 4)** with minimum cost. Edge **(3, 4)** is selected at random.

In a similar way, edges **(4, 5), (5, 7), (7, 8), (6, 8)** and **(6, 9)** are selected. As all the vertices are visited, now the algorithm stops.

The cost of the spanning tree is (2 + 2 + 3 + 2 + 5 + 2 + 3 + 4) = 23. There is no more spanning tree in this graph with cost less than **23**.

Implementation:

```
#include <iostream>

#include <vector>

#include <queue>

#include <functional>

#include <utility>

using namespace std;
```

```
const int MAX = 1e4 + 5;

typedef pair<long long, int> PII;

bool marked[MAX];

vector <PII> adj[MAX];

long long prim(int x){

    priority_queue<PII, vector<PII>, greater<PII> > Q;

    int y;

    long long minimumCost = 0;

    PII p;

    Q.push(make_pair(0, x));

    while(!Q.empty())    {

        // Select the edge with minimum weight

        p = Q.top();

        Q.pop();

        x = p.second;

        // Checking for cycle

        if(marked[x] == true)

            continue;

        minimumCost += p.first;

        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);        }    }
    return minimumCost;}
int main(){
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;}
```

Time Complexity:

The time complexity of the Prim's Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

POSSIBLE QUESTIONS

UNIT IV

2 Mark Questions:

1. What is the use of Amortized analysis?

2. Define aggregate method.

3. What is a Breadth-First search approach?

4. What is a Depth-First search approach?

5. Write a short note on Spanning tree.

6 Mark Questions:

1. Explain the concepts of Amortized analysis.

2. Explain about Aggregate method with example.

3. Explain about Potential method with example.

4. Explain about accounting method with example.

5. Elaborate Breadth-First search approach with example.

6. Elaborate Depth-First search approach with example.

7. Explain in detail about Minimum spanning tree.

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
Coimbatore-641021
Department of Computer Science
II B.Sc(CS) (BATCH 2017-2020)
Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS **ONE MARK QUESTIONS**

UNIT-4

S.NO	QUESTIONS	OPT1	OPT2	OPT3	OPT4	ANSWER
1	Experimental analysis is otherwise called as________.	Testing	Implementing	Proving	Scheduling	Testing
2	Algorithms are well ordered with _____ operations.	Efficient	Multiple	Unambiguous	Infinite	Unambiguous
3	An algorithm is a _______ definition.	Formal	Informal	Basic	Complete	Formal
4	_______ involves the solution of sub-problems.	Merge	Combine	Divide	Conquer	Conquer
5	Swapping is used to _______ the values.	Remove	Add	interchange	Integrate	interchange
6	A sub-list is maintained in _______ sort.	Bubble	Insertion	Heap	Both a & c	Insertion
7	Example for Divide and conquer is _______	Merge sort	Quick sort	Bubble sort	All of these	Merge sort
8	Heap data structure is always a _______	Radix	complete binary tree	Heap sort	Both a & b	complete binary tree
9	Red-Black tree is one of the _______ binary search tree.	Balanced	Unbalanced	Different	Similar	Balanced
10	In Red-Black tree every node is either _______.	Black or blue	Black or orange	red or black	Both b & c	red or black
11	In Red-Black tree the root is _______.	Red	Black	Blue	Purple	Red
12	In Red-Black tree if a node is _______, then both its children are black.	Black	Blue	Yellow	Red	Red
13	All paths from the node have the ______ black height in Red-Black tree.	Same	Different	Null	Variable	Same
14	______ algorithm traverses a graph in a depthward motion.	Depth First Search	Greedy	Analysis	Both a & c	Depth First Search

15	A ______ is a notation used to represent the connection between pairs of objects.	Graph	Tree	Binary Tree	Stack	Graph
16	Edges are the links that connect the vertices.	Nodes	Edges	Trees	Both a & c	Edges
17	In a directed graph, edges have ______.	Scalar	Vector	Direction	Time	Direction
18	The ______ string is denoted as empty string.	zero-length	Nil	Empty	max-length	zero-length
19	Naive technique performs _______.	Union	Intersect	Post-processing	Pre-processing	Pre-processing
20	The average-case running time of ______ is high.	Rabin and Karp	Prims	Naïve	Kruskal	Rabin and Karp
21	_______________ are genealogical charts which are used to present the data	Graphs	Pedigree and lineal chart	Line , bar chart	pie chart	Pedigree and lineal chart
22	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
23	A _________ is a graph without any cycle.	tree	path	set	list	tree
24	In binary trees there is no node with a degree greater than _____	zero	one	two	three	two
25	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 above	All the above
26	The Number of subtrees of a node is called its _______.	leaf	terminal	children	degree	degree
27	Nodes that have degree zero are called ________.	end node	leaf nodes	subtree	root node	leaf nodes
28	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
29	All node except the leaf nodes are called________.	terminal node	percent node	non terminal	children node	non terminal
30	The roots of the subtrees of a node X, are the _______ of X.	Parent	Children	Sibling	sub tree	Children

31	X is a root then X is the ______ of its children.	sub tree	Parent	Sibilings	subordinate	Parent
32	The children of the same parent are called ___________________.	sibiling	leaf	child	subtree	sibiling
33	___________of a node are all the nodes along the path form the root to that node.	Degree	sub tree	Ancestors	parent	Ancestors
34	The ______________of a tree is defined to be a maximum level of any node in the tree.	weight	length	breath	height	height
35	A___________ is a set of $n \geq 0$ disjoint trees	Group	forest	Branch	sub tree	forest
36	A tree with any node having at most two branches is called a _____________.	branched tree	sub tree	binary tree	forest	binary tree
37	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
38	Data structure represents the hierarchical relationships between individual data item is known as __________.	Root	Node	Tree	Address	Tree
39	Node at the highest level of the tree is known as _______.	Child	Root	Sibiling	Parent	Root
40	The root of the tree is the _______of all nodes in the tree.	Child	Parent	Ancestor	Head	Ancestor
41	_____is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree	Subtree
42	A node with no children is called _________.	Root Node	Branch	Leaf Node	Null tree	Leaf Node
43	In a tree structure a link between parent and child is called _______	Branch	Root	Leaf	Subtree	Branch
44	Height – balanced trees are also referred as as ___________trees.	AVL trees	Binary Trees	Subtree	Branch Tree	AVL trees
45	Visiting each node in a tree exactly once is called _________	searching	traversing	walk through	path	traversing

46	In________traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order	PreOrder
47	In________traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order	Inorder
48	In________traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order	PostOrder
49	Inorder traversal is also sometimes called______	Symmetric Order	End Order	PreOrder	PostOrder	Symmetric Order
50	Postorder traversal is also sometimes called______	Symmetric Order	End Order	PreOrder	PostOrder	End Order
51	Nodes of any level are numbered from _________	Left to right	Right to Left	Top to Bottom	Bottom to Top	Left to right
52	________ search involves only addition and subtraction.	binary	fibonacci	sequential	non sequential	fibonacci
53	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
54	Binary trees are used in ______ sorting.	quick sort	merge sort	heap sort	lrsort	heap sort
55	The ____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch	Root
56	Which of these searching algorithm uses the Divide and Conquere technique for sorting	Linear search	Binary search	fibonacci search	None of the above	Binary search
57	______ algorithm can be used only with sorted lists.	Linear search	Binary search	insertion sort	merge sort	Binary search
58	________ search involves comparision of the element to be found with every elements in a list.	Linear search	Binary search	fibonacci search	None of the above	Linear search
59	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($\log_2 n$)
60	_____ is used for decision making in eight coin problem.	trees	graphs	linked lists	array	trees

UNIT V

SYLLABUS

String Processing: String Matching, KMP Technique.

String Matching

String-searching algorithms, sometimes called string-matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

The object of string searching is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.). As with most algorithms, the main considerations for string searching are speed and efficiency.

A **search string** is the combination of characters and words that make up the **search** being conducted. For example, if you were **searching** for computer help and typed "computer help" in Google, an Internet **search** engine that would be your **search string**.

There are a number of string searching algorithms in existence today they are as follows,

String Matching Algorithms:

There are many types of string matching algorithms like:

1. The Naïve string-matching algorithm
2. The Rabin-krap algorithm
3. String matching with finite automata
4. The Knuth-Morris-Pratt algorithm

String-matching algorithms are also used, for example, to search for particular patterns in DNA sequences.

The Naive string-matching algorithm

Naïve pattern searching is the simplest method among other pattern searching algorithms. It checks for all character of the main string to the pattern. This algorithm is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.

The time complexity of Naïve Pattern Search method is O(m*n). The m is the size of pattern and n is the size of the main string.

Input and Output

Input:

Main String: “ABAAABCDBBABCDDEBCABC”, pattern: “ABC”

Output:

Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18

Algorithm

naivePatternSearch(pattern, text)

Input: The text and the pattern

Output: location, where patterns are present in the text

```
Begin
   patLen := pattern Size
   strLen := string size

   for i := 0 to (strLen - patLen), do
      for j := 0 to patLen, do
         if text[i+j] ≠ pattern[j], then
            break the loop
      done

      if j == patLen, then
         display the position i, as there pattern found
   done
End
```

Source Code (C++)

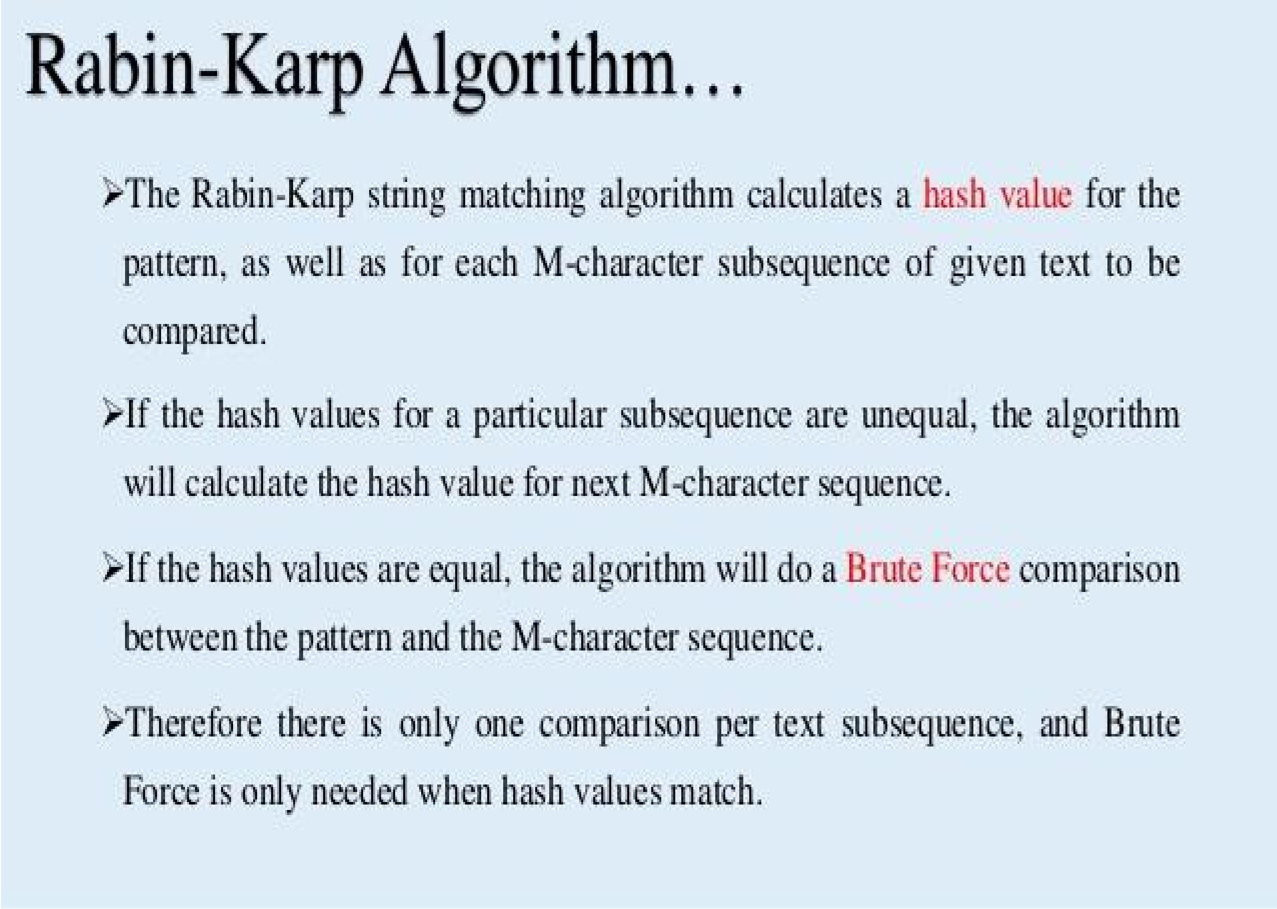
```
#include<iostream>
using namespace std;

void naivePatternSearch(string mainString, string pattern, int array[], int *index) {
   int patLen = pattern.size();
   int strLen = mainString.size();

   for(int i = 0; i<=(strLen - patLen); i++) {
      int j;
      for(j = 0; j<patLen; j++) {      //check for each character of pattern if it is matched
         if(mainString[i+j] != pattern[j])
            break;
      }

      if(j == patLen) {     //the pattern is found
         (*index)++;
         array[(*index)] = i;      }   }   }

int main() {
   string mainString = "ABAAABCDBBABCDDEBCABC";
   string pattern = "ABC";
   int locArray[mainString.size()];
```

```
  int index = -1;
  naivePatternSearch(mainString, pattern, locArray, &index);

  for(int i = 0; i <= index; i++) {
    cout << "Pattern found at position: " << locArray[i]<<endl;
  }
}
```

Output

```
Pattern found at position: 4
Pattern found at position: 10
Pattern found at position: 18
```

The Rabin-Karp algorithm



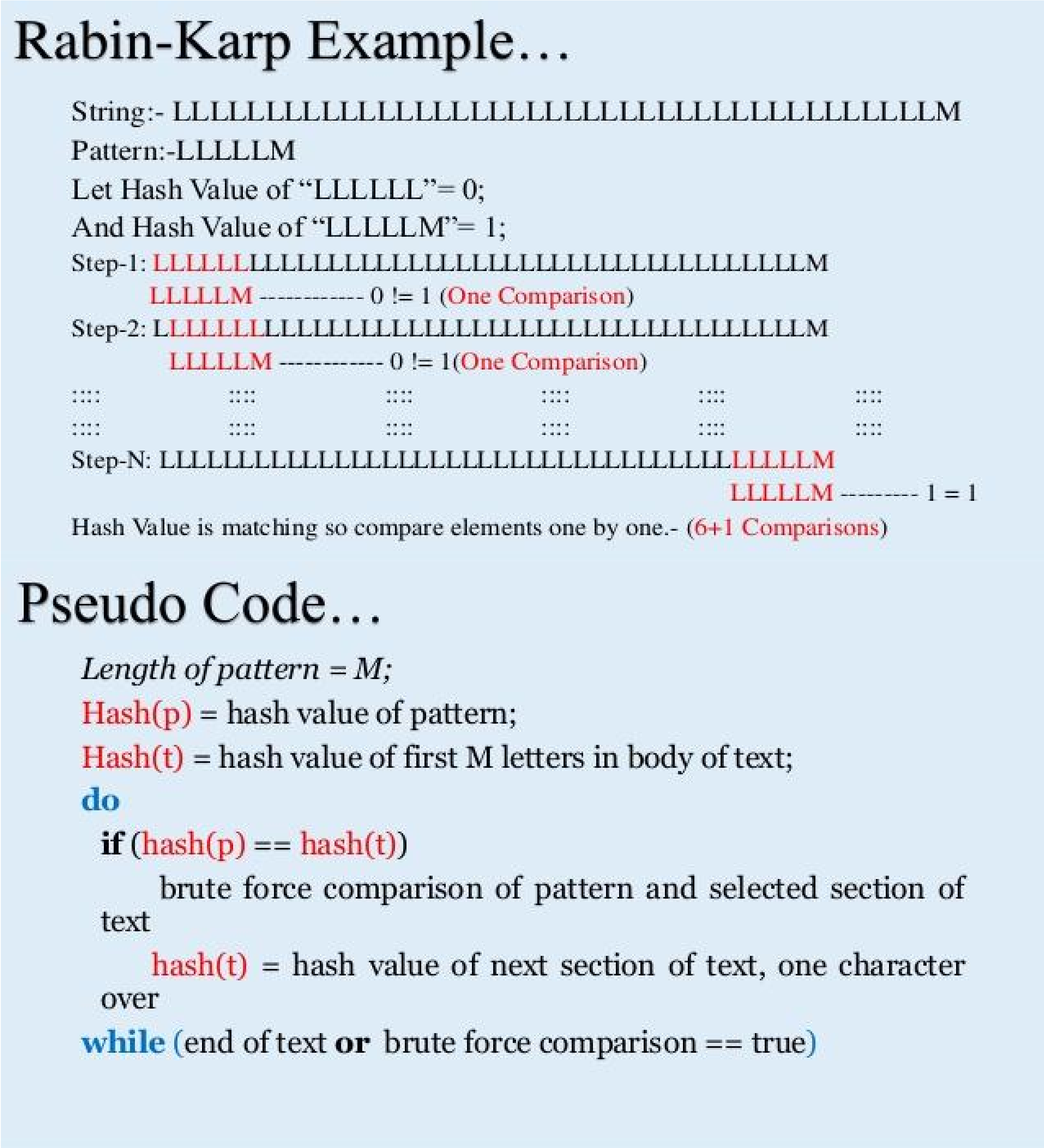
Rabin-Karp Algorithm…

- The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequence of given text to be compared.
- If the hash values for a particular subsequence are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.
- Therefore there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

The **Brute Force** algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

- Compared characters are italicized.
- Correct matches are in boldface type.

The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.



Rabin-Karp Example…

String:- LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM

Pattern:-LLLLLM

Let Hash Value of "LLLLLL"= 0;

And Hash Value of "LLLLLM"= 1;

Step-1: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM
LLLLLM ------------ 0 != 1 (One Comparison)

Step-2: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM
LLLLLM ------------ 0 != 1(One Comparison)

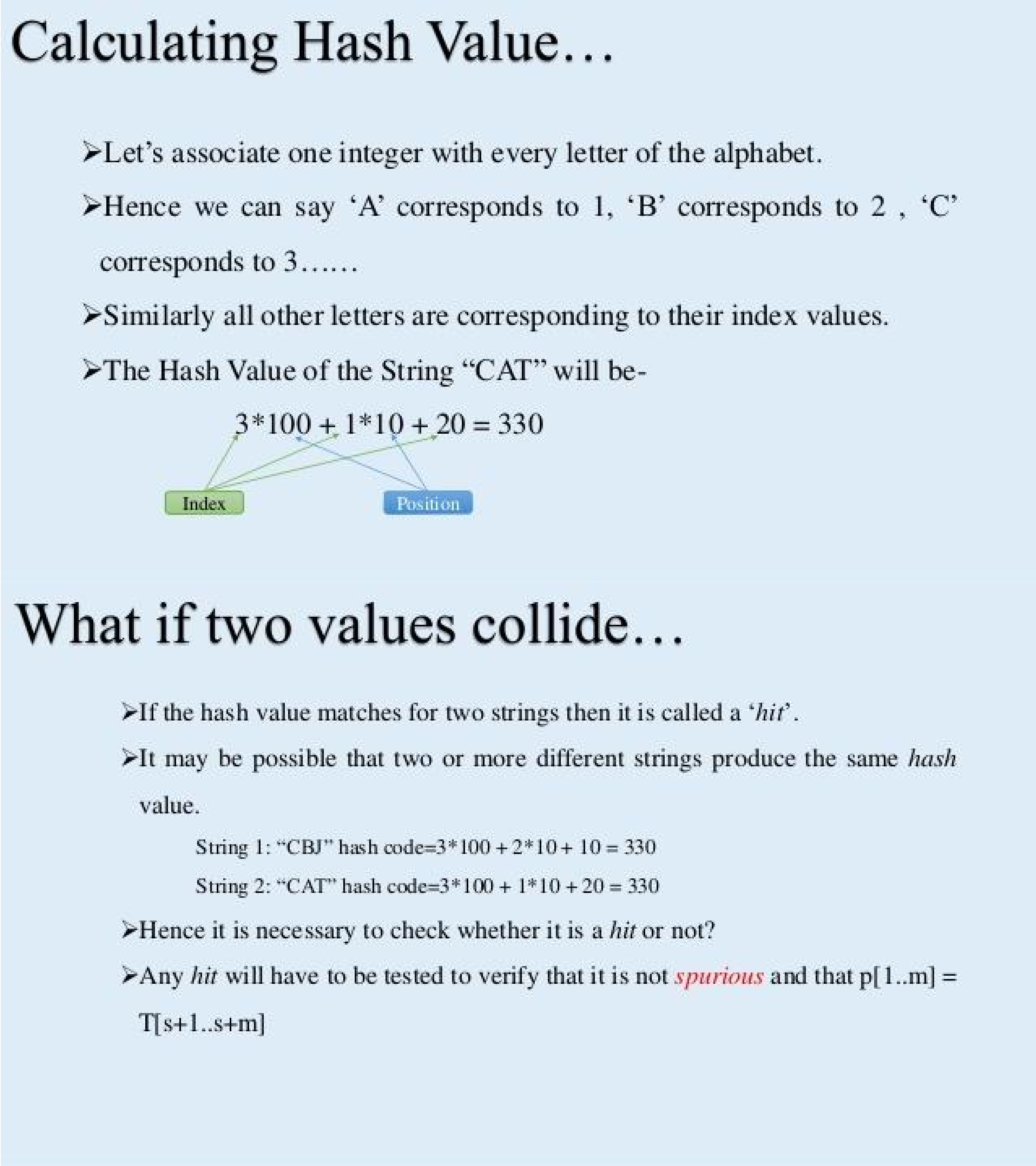
:::: :::: :::: :::: :::: ::::
:::: :::: :::: :::: :::: ::::

Step-N: LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLM
LLLLLM --------- 1 = 1

Hash Value is matching so compare elements one by one.- (6+1 Comparisons)

Pseudo Code…

```
Length of pattern = M;
Hash(p) = hash value of pattern;
Hash(t) = hash value of first M letters in body of text;
do
  if (hash(p) == hash(t))
      brute force comparison of pattern and selected section of
  text
      hash(t) = hash value of next section of text, one character
  over
while (end of text or brute force comparison == true)
```



Calculating Hash Value…

- ➢Let’s associate one integer with every letter of the alphabet.
- ➢Hence we can say ‘A’ corresponds to 1, ‘B’ corresponds to 2 , ‘C’ corresponds to 3……
- ➢Similarly all other letters are corresponding to their index values.
- ➢The Hash Value of the String “CAT” will be-

$$3*100 + 1*10 + 20 = 330$$

Index

Position

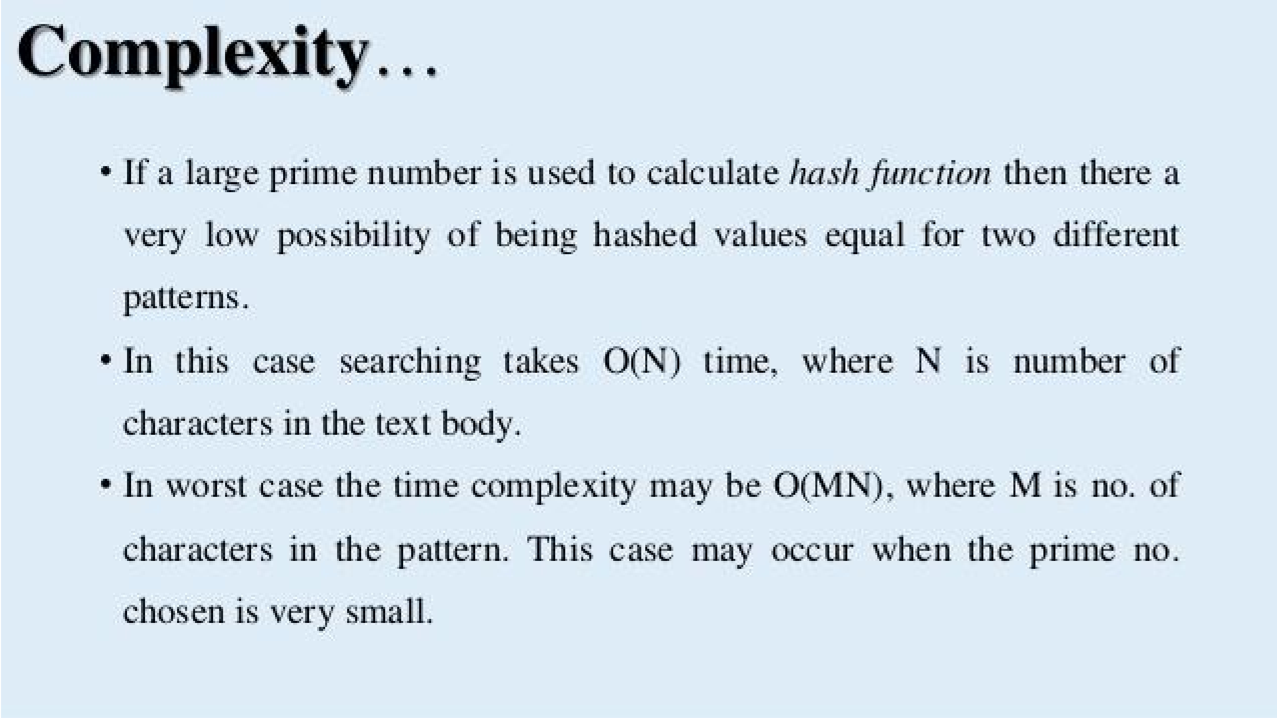
What if two values collide…

- ➢If the hash value matches for two strings then it is called a ‘*hit*’.
- ➢It may be possible that two or more different strings produce the same *hash* value.

  String 1: “CBJ” hash code=3*100 + 2*10 + 10 = 330

  String 2: “CAT” hash code=3*100 + 1*10 + 20 = 330

- ➢Hence it is necessary to check whether it is a *hit* or not?
- ➢Any *hit* will have to be tested to verify that it is not *spurious* and that p[1..m] = T[s+1..s+m]



Complexity…

- If a large prime number is used to calculate *hash function* then there a very low possibility of being hashed values equal for two different patterns.
- In this case searching takes O(N) time, where N is number of characters in the text body.
- In worst case the time complexity may be O(MN), where M is no. of characters in the pattern. This case may occur when the prime no. chosen is very small.

String matching with finite automata

Many string-matching algorithms build a finite automaton that scans the text string *T* for all occurrences of the pattern *P*. This section presents a method for building such an automaton. These string-matching automata are very efficient: they examine each text character *exactly once*, taking constant time per text character.

The matching time used—after preprocessing the pattern to build the automaton—is therefore *_(n)*. The time to build the automaton, however, can be large if _ is large. We begin this section with the definition of a finite automaton.

We then examine a special string-matching automaton and show how it can be used to find occurrences of a pattern in a text. This discussion includes details on how to simulate the behavior of a string-matching automaton on a given text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

Finite automata

A ***finite automaton*** *M* is a 5-tuple *(Q, q*0*, A,_, δ)*, where
- *Q* is a finite set of ***states***,
- *q*0 *Q* is the ***start state***,
- *A Q* is a distinguished set of ***accepting states***,
- _ is a finite ***input alphabet***,
- *δ* is a function from *Q* ?_ into *Q*, called the ***transition function*** of *M*. The finite automaton begins in state *q*0 and reads the characters of its input string one at a time. If the automaton is in state *q* and

reads input character *a*, it moves (“makes a transition”) from state *q* to state *δ(q, a)*. Whenever its current state *q* is a member of *A*, the machine *M* is said to have ***accepted*** the string read so far.

```
FINITE-AUTOMATON-MATCHER(T, δ,m)
n ← length[T ]
q ← 0
for i ← 1 to n
    do q ← δ(q, T [i ])
        if q = m
            then print “Pattern occurs with shift” i − m
```

The simple loop structure of FINITE-AUTOMATON-MATCHER implies that its matching time on a text string of length *n* is _(n). This matching time, however, does not include the preprocessing time required to compute the transition function *δ*. We address this problem later, after proving that the procedure FINITEAUTOMATON-MATCHER operates correctly.

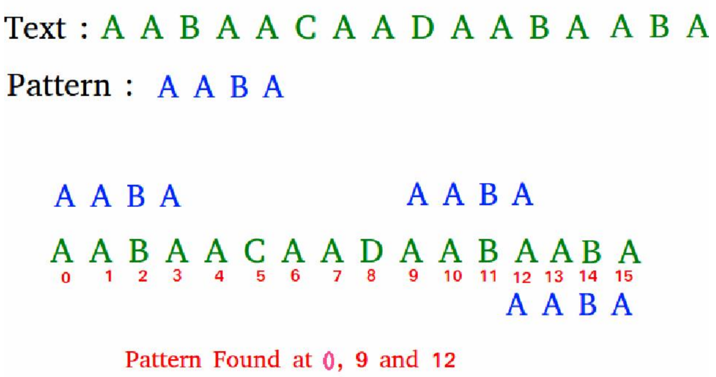
KMP (Knuth-Morris-Pratt) algorithm

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that *n > m*.

Examples:

Input: txt[] = "THIS IS A TEST TEXT"
pat[] = "TEST"
Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"
pat[] = "AABA"
Output: Pattern found at index 0
Pattern found at index 9
Pattern found at index 12


Text : A A B A A C A A D A A B A A B A
Pattern : A A B A
A A B A
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A A B A
Pattern Found at 0, 9 and 12

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed Naive pattern searching algorithm in the previous post. The worst case complexity of the Naive algorithm is O(m(n-m+1)). The time complexity of KMP algorithm is O(n) in the worst case.

KMP (Knuth Morris Pratt) Pattern Searching:

The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

Matching Overview

txt = "AAAAABAAABA"
pat = "AAAA"

We compare first window of **txt** with **pat**

txt = "**AAAA**ABAAABA"
pat = "**AAAA**" [Initial position]

We find a match. This is same as Naive String Matching.

In the next step, we compare next window of **txt** with **pat**.

txt = "**AAAAA**BAAABA"
pat = "**AAAA**" [Pattern shifted one position]

This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array lps[] that tells us the count of characters to be skipped.

Preprocessing Overview:

- KMP algorithm preprocesses pat[] and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.

- **name lps indicates longest proper prefix which is also suffix.**. A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].
- lps[i] = the longest proper prefix of pat[0..i]
  which is also a suffix of pat[0..i].

Note : lps[i] could also be defined as longest prefix which is also proper suffix. We need to use properly at one place to make sure that the whole substring is not considered.

Examples of lps[] construction:
For the pattern "AAAA",
lps[] is [0, 1, 2, 3]

For the pattern "ABCDE",
lps[] is [0, 0, 0, 0, 0]

For the pattern "AABAACAABAA",
lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern "AAACAAAAAC",
lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern "AAABAAA",
lps[] is [0, 1, 2, 0, 1, 2, 3]

Searching Algorithm:

Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use lps[] to decide next positions (or to know a number of characters to be skipped)?

- We start comparison of pat[j] with j = 0 with characters of current window of text.
- We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
- When we see a **mismatch**
  - We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
  - We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.

- From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider above example to understand this.

txt[] = "**AAAA**ABAAABA"
pat[] = "**AAAA**"
lps[] = {0, 1, 2, 3}

i = 0, j = 0
txt[] = "**AAAA**ABAAABA"
pat[] = "**AAAA**"
txt[i] and pat[j] match, do i++, j++

i = 1, j = 1
txt[] = "**AAAA**ABAAABA"
pat[] = "**AAAA**"
txt[i] and pat[j] match, do i++, j++

i = 2, j = 2
txt[] = "**AAAA**ABAAABA"
pat[] = "**AAAA**"
pat[i] and pat[j] match, do i++, j++

i = 3, j = 3
txt[] = "**AAAA**ABAAABA"
pat[] = "**AAAA**"
txt[i] and pat[j] match, do i++, j++

i = 4, j = 4
Since j == M, print **pattern found** and reset j,
j = lps[j-1] = lps[3] = 3

Here unlike Naive algorithm, we do not match first three characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.
i = 4, j = 3
txt[] = "A**AAAA**BAAABA"
pat[] = "**AAAA**"
txt[i] and pat[j] match, do i++, j++

i = 5, j = 4
Since j == M, print **pattern found** and reset j,
j = lps[j-1] = lps[3] = 3

Again unlike Naive algorithm, we do not match first three

characters of this window. Value of lps[j-1] (in above step) gave us index of next character to match.
i = 5, j = 3
txt[] = "AA**AAAB**AAABA"
pat[] = "**AAAA**"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[2] = 2

i = 5, j = 2
txt[] = "AAA**AABA**AABA"
pat[] = "**AAAA**"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[1] = 1

i = 5, j = 1
txt[] = "AAAA**ABAA**ABA"
pat[] = "**AAAA**"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[0] = 0

i = 5, j = 0
txt[] = "AAAAA**BAAA**BA"
pat[] = "**AAAA**"
txt[i] and pat[j] do NOT match and j is 0, we do i++.

i = 6, j = 0
txt[] = "AAAAAB**AAABA**"
pat[] = "**AAAA**"
txt[i] and pat[j] match, do i++ and j++

i = 7, j = 1
txt[] = "AAAAAB**AAAB**A"
pat[] = "**AAAA**"
txt[i] and pat[j] match, do i++ and j++

We continue this way...

POSSIBLE QUESTIONS

UNIT V

2 Mark Questions:

1. Define String matching.
2. What is Naive string matching method?
3. Define Rabin-Karp approach.
4. Define Fine-automaton method.
5. Define KMP technique.

6 Mark Questions:

1. Explain the Concept of String matching technique.
2. Explain about naive string-matching algorithm.
3. Discuss the Rabin-Karp algorithm.
4. Explain in detail about the prefix function for a pattern.
5. Discuss about Running-time analysis.
6. Elaborate KMP technique.
7. Explain about Correctness of the KMP algorithm.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
Coimbatore-641021
Department of Computer Science
II B.Sc(CS) (BATCH 2017-2020)
Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS **ONE MARK QUESTIONS**

UNIT-5

S.NO	QUESTIONS	OPT1	OPT2	OPT3	OPT4	ANSWER
1	Writing a Pseudocode has _______ of styles.	Restriction	Protocols	Condition	No restriction	No restriction
2	Analysis of algorithms is the determination of _______.	Time and Space	Time	Space	Distance	Time and Space
3	The algorithm’s state is the set of values corresponding to all _______.	Instances	Factors	Variables	Objects	Variables
4	_______ involves division of problems into sub problems.	Combine	Merge	Conquer	Break	Break
5	Heap data structure is always a _______	Radix	complete binary tree	Heap sort	Both a & b	complete binary tree
6	Bucket sort is a sorting algorithm that works by _______ an array.	Partitioning	Adding	Updating	Combining	Partitioning
7	A quick sort first selects a value, which is called the _______.	Constant	Variable	Pivot value	None of these	Pivot value
8	The speed of Radix Sort largely depends on the _______ basic operations.	Outer	Input	Output	Inner	Inner
9	A _______ is an operation that preserves in-order traversal key ordering.	Alteration	Reversal	Resolution	Rotation	Rotation
10	The ______ of the tree is the maximum depth of any node in the tree.	Row	Column	Height	Width	Height
11	The number of children emanating from a given node is referred to as its ______	Depth	Degree	Height	Width	Degree
12	In binary search trees, traversing from ______ is known as inorder-tree traversal.	Right to Left	Top to Bottom	Left to right	Both a & b	Left to right

13	Edges are the links that connect the vertices.	Nodes	Edges	Trees	Both a & c	Edges
14	In a directed graph, edges have ______.	Scalar	Vector	Direction	Time	Direction
15	The spanning tree does not have ____.	Loops	Graph	Aggregation	Recursion	Loops
16	In an _____ graph, edges have no direction.	Undirected	Tree	Recurrence	Modules	Undirected
17	The running time of naïve string matcher is _____ to its matching time.	Not-equal	Equal	Minimum	Maximum	Equal
18	_____algorithm makes use of elementary number-theoretic notions.	Rabin-Karp	Naive	KMP	Both a & c	Rabin-Karp
19	To specify the string-matching automaton, we first define a ______ function.	Suffix	Prefix	Theoretical	String	Suffix
20	______algorithm avoids the computation of the transition function.	KMP	Naive	Rabin	Karp	KMP
21	The _____node are not a part of original tree and are represented as square nodes.	internal node	external node	intermediate node	terminal node	external node
22	The external nodes are in a binary search tree are also known as _________ nodes	internal	search	failure	round	round
23	A binary tree with external nodes added is an --------------- binary tree	extended	expanded	internal	external	extended
24	______ is a set of name-value pairs.	Symbol table	Graph	Node	Record	Symbol table
25	Each name in the symbol tales is associated with an___	name value pairs	element	attribute	entries	attribute
26	This is not an operation perform on the symbol table.	insert a new name and its value.	retrieve the attribute of a name.	search if a name is already present	Add or subtract two values	Add or subtract two values
27	If the identifiers are known in advance and no deletion/insertions are allowed then this symbol table is _______	static	empty	dynamic	automatic	static

28	The cost of decoding a code word is ------------------- to the number of bits in the code	equal	not equal	proportional	inversely proportional	proportional
29	The solution of finding a binary tree with minimum weighted external path length has been given by	Huffman	Kruskal	Euler	Hamilton	Huffman
30	_____ symbol table allows insertion and deletion of names.	Hashed	Sorted	Static	Dynamic	Dynamic
31	________ is an application of Binary trees with minimal weighted external path lengths.	Finding optimal merge patterns	Storage compaction	Recursive Procedure calls	Job Scheduling	Storage compaction
32	If hl and hr are the heights of the left and right subtrees of a tree respectively and if |hl-hr|<=1 then this tree is called _____	extended binary tree	binary search tree	skewed tree	height balanced tree	height balanced tree
33	If hl and hr are the heights of the left and right subtrees of a tree respectively then |hl-hr| is called its _____	Average height	minimal depth	Maximum levels	Balance factor	Balance factor
34	For an AVL Tree the balance factor is =____	0	-1	1	Any of the above	Any of the above
35	If the names are ________ in the symbol table, searching is easy.	sorted	short	bold	upper case	sorted
36	_________ allocation is not desirable for dynamic tables, where insertions and deletions are allowed.	Linear	Sequential	Dynamic	None	Sequential
37	A search in a hash table with n identifiers may take ------ time	O(n)	O(1)	O(2)	O(2n)	O(n)
38	_________ data structure is used to implement symbol tables	directed graphs	binary search trees	circular queue	None	binary search trees
39	Every binary search tree wth n nodes has ______ sqare node (external nodes).	n/2	n+1	n-1	2^n	n+1

40	In a Hash table the address of the identifier x is obtained by applying	sequence of comparisions	binary searching	arithmetic function	collision	arithmetic function
41	The partitions of the hash table are called ________	Nodes	Buckets	Roots	Fields	Buckets
42	The arithmetic functions used for Hashing is called	Logical operations	Rehashing	Mapping function	Hashing function	Hashing function
43	Each bucket of Hash table is said to have several ______	slots	nodes	fields	links	slots
44	A_______ occurs when two non_identical identifiers are hashed in the same bucket.	collision	contraction	expansion	Extraction	collision
45	A hashing function f transforms an identifier x into a __________ in the hash table	symbol name	bucket address	link field	slot number	bucket address
46	When a new identifier I is mapped or hashed by the function f into a full bucket then ______occurs	underflow	overflow	collision	rehashing	overflow
47	If f(I) and F(J) are equal then Identifiers I and J are called________	synonyms	antonyms	hash functions	buckets	synonyms
48	A ---tree is a binary tree in which external nodes represent messages	decode	uncode	extended	none	decode
49	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	none of these	x mod m
50	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
51	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 function	unequal Hashing function	uniform hash function

52	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
53	Each chain in the hash tables will have a	tail node	link node	head node	null node	head node
54	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
55	Average number of probes needed for searching can be obtained by -------------------probing	quadratic	linear	rehashing	Sequential	quadratic
56	Rehashing is ______	series of hash function	linear probing	quadratic functions	Rebuild function	series of hash function
57	_________ is a method of overflows handling.	linear open addressing	Adjacency lsit	sequential representation	Indexed address	linear open addressing
58	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
59	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
60	In External sorting data are stored in _________	RAM memory	Cache memory	selection tree	Buffers	selection tree
61	The ______ string-matching procedure can be interpreted graphically.	naïve	KMP	Pattern	Automative	naïve
62	The _____ case running time of Rabin and Karp is high.	Average	Minimum	Maximum	Worst	Average
63	The ______ string is denoted as empty string.	zero-length	Nil	Empty	max-length	zero-length
64	Naive technique performs _______.	Union	Intersect	Post-processing	Pre-processing	Pre-processing
65	The average-case running time of ______ is high.	Rabin and Karp	Prims	Naïve	Kruskal	Rabin and Karp
66	The spanning tree does not have ____.	Loops	Graph	Aggregation	Recursion	Loops
67	Recurrences are generally used in _______ paradigm.	Interface	Graph theory	Divide-and-conquer	Both a & b	Divide-and-conquer

68	The naive string-matching procedure can be interpreted graphically as sliding a _____	template	Granules	Recursive	Interface	template
69	String-matching algorithms are used for______	Graphics	Characters	Pattern searching	Aggregation	Pattern searching
70	Which of the following technique performs pre-processing?	Naive	Rabin	KMP	Morris	Naive
71	The zero-length string is denoted as _______.	Null string	Empty string	Void string	Exit	Empty string
72	A ______ is an equation that describes a function in terms of its value.	Binary search	Recurrence	Tree	Graph	Recurrence
73	Knuth-Morris-Pratt algorithm is a ______ time string-matching algorithm.	Linear	non-linear	Directive	non-directive	Linear