

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

Coimbatore-641 021

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT**SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHMS****SEMESTER : IV****SUBJECT CODE: 16CSU401****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 C 5X8 = 40 (Either 'A' or 'B' Choice)	40
3.	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: Dr. T. GENISH

SUBJECT NAME: DESIGN AND ANALYSIS OF ALGORITHMS

SUB.CODE : 16CSU401

SEMESTER: IV

CLASS : II B.SC CS

Sl.No	Lecture Duration (Periods)	Topics to be covered	Support Materials
UNIT- I			
1	1	Introduction	
2	1	Basic Design and Analysis techniques of Algorithms	W1
3	1	Contd..Analysis Techniques	W1
4	1	Correctness of Algorithm.	T1:2
5	1	Algorithm Design Techniques: Iterative techniques	W2
6	1	Contd.. Iterative techniques	W2
7	1	Divide and Conquer	T1: 12-14
8	1	Dynamic Programming	T1: 301-307
9	1	Contd.. Dynamic Programming	T1: 308-320
10	1	Contd.. Dynamic Programming	T1: 320-328
11	1	Greedy Algorithms	T1: 329-332
12	1	Contd.. Greedy Algorithms	T1: 333-340

13	1	Contd.. Greedy Algorithms	T1: 340-355
14	1	Recapitulation and Possible Questions Discussion	
15	1	Recapitulation and Possible Questions Discussion	
		Total No. Of Hours Planned for unit I	15
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	Contd.. Bubble Sort	W3
4	1	Insertion Sort	T1: 2-5
5	1	Merge Sort	T1: 12-15
6	1	Advanced Sorting techniques – Heap Sort	T1: 140-145
7	1	Contd.. Heap Sort	T1: 145-152
8	1	Quick Sort	T1: 153-159
9	1	Contd.. Quick Sort	T1: 159-163
10	1	Sorting in Linear Time – Bucket Sort	T1: 180-183

11	1	Radix Sort	T1: 178-180
12	1	Count Sort	T1: 175-178
13	1	Searching Techniques- Medians & Order Statistics	W4
14	1	Medians & Order Statistics	W4
15	1	complexity analysis	W5
16	1	Recapitulation and Possible Questions Discussion	
17	1	Recapitulation and Possible Questions Discussion	
18	1	Recapitulation and Possible Questions Discussion	
Total No. Of Hours Planned for unit II:			18
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 Possible Questions Discussion	
8	1	Recapitulation and Possible Questions Discussion	
		Total No. Of Hours Planned for unit III:	08
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: 356-360
2	1	Contd.. Amortized analysis	T1: 360-367
3	1	Graphs: Graph Algorithms–Breadth First Search	T1: 465-467
4	1	Contd.. Breadth First Search	T1: 468-470
5	1	Contd.. Breadth First Search	T1: 470-477
6	1	Depth First Search	T1: 477-480
7	1	Contd.. Depth First Search	T1: 480-482
8	1	Contd.. Applications of Depth First Search	T1: 482-485
9	1	Minimum Spanning Trees	T1: 498-505
11	1	Contd.. Minimum Spanning Trees	T1: 505-512
12	1	Recapitulation and Possible Questions Discussion	

13	1	Recapitulation and Possible Questions Discussion	
		Total No. Of Hours Planned for unit III:	13
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: 853-856
2	1	Contd.. String Matching	T1: 856-860
3	1	KMP Technique	T1: 861-875
4	1	Contd.. KMP Technique	T1: 875-883
5	1	Recapitulation and Possible Questions Discussion	
6	1	Recapitulation and Possible Questions Discussion	
		Total No. Of Hours Planned for unit V:	6
TEXT BOOKS:		T1:Cormen, T.H., Charles, E. Leiserson., Ronald, L. Rivest. (2009). Clifford Stein Introduction to Algorithms(3rd ed.). New Delhi: PHI.	

TEXT BOOK

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

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, that 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.

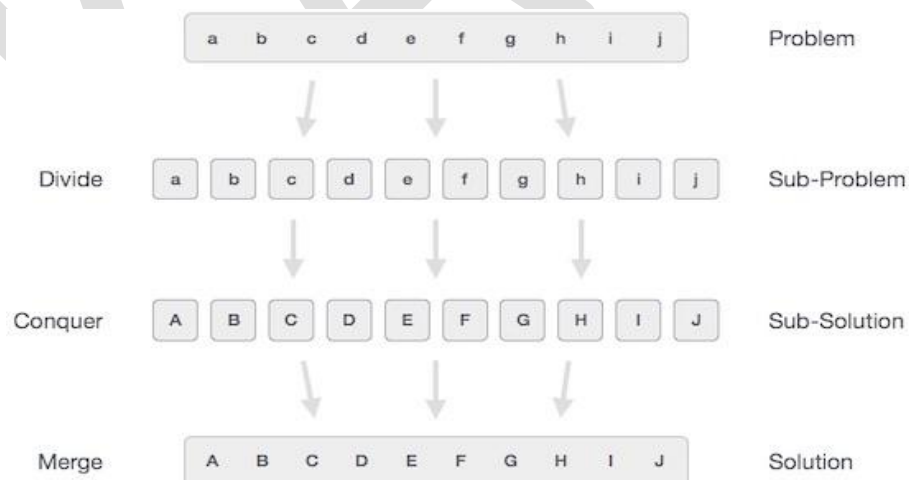
When the processing structure is a sequential one (for example a sequence of assignments) then the verification process is a simple one (we must only analyze the effect of each assignment on the algorithm's state).

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 work so close that they appear as one.

For example,

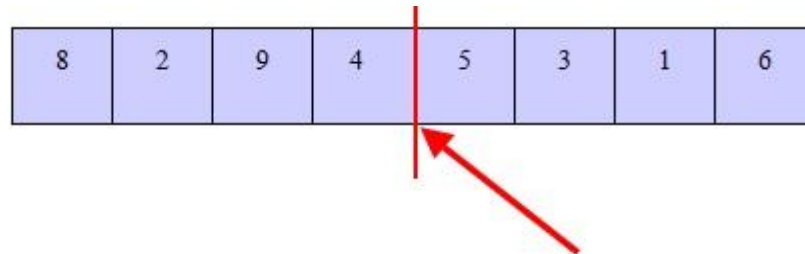
The following computer algorithms are based on divide-and-conquer programming approach –

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

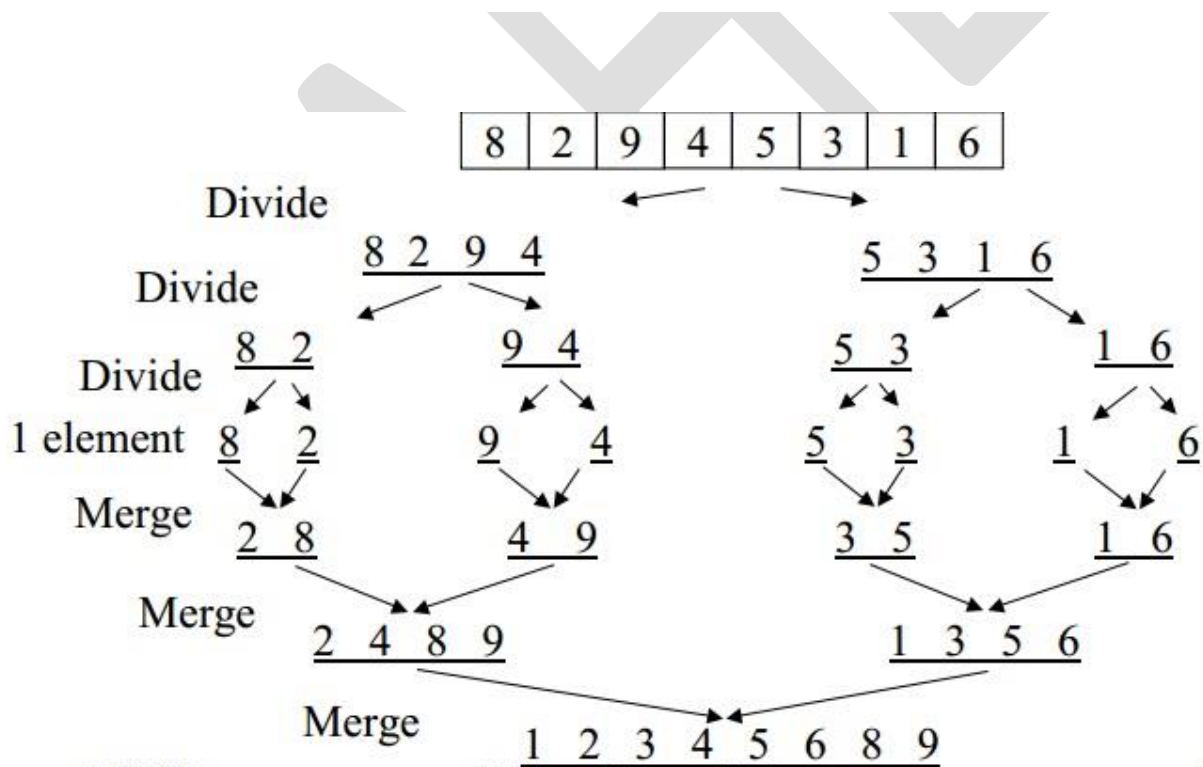
Example for Divide and Conquer

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

Merge Sort Example



- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves 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 subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem 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:

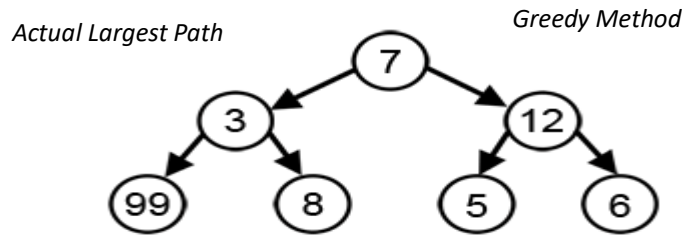
```
void fib () {
    fibresult[0] = 1;
    fibresult[1] = 1;
    for (int i = 2; i < n; i++)
        fibresult[i] = fibresult[i-1] + fibresult[i-2];
}
```

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.

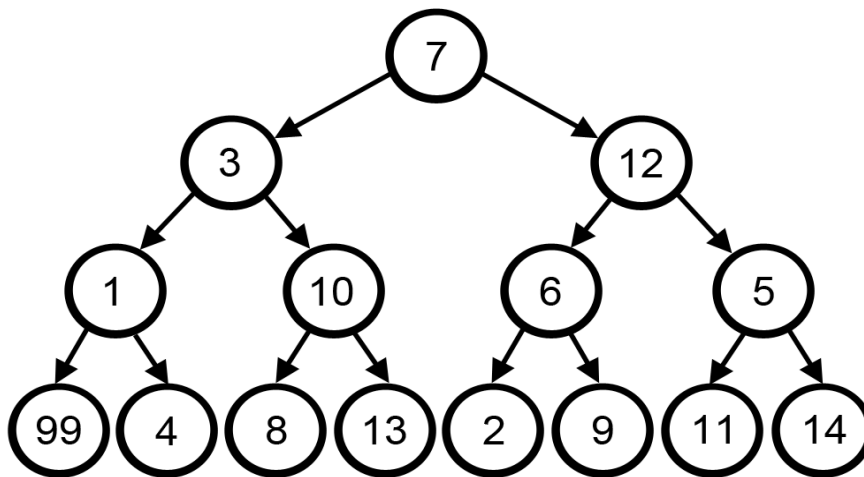


Limitations of Greedy Algorithms

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

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT I: INTRODUCTION

BATCH: 2016-2019

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 playstation 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 2016-2019)

Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARK QUESTIONS

S.NO	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4
1	_____ is a sequence of instructions to accomplish a particular task	Data Structure	Algorithm	Ordered List	Queue
2	_____ criteria of an algorithm ensures that the algorithm terminate after a particular number of steps.	effectiveness	finiteness	definiteness	All the above
3	An algorithm must produce _____ output(s)	many	only one	atleast one	zero or more
4	_____ criteria of an algorithm ensures that the algorithm must be feasible.	effectiveness	finiteness	definiteness	All the above
5	_____ criteria of an algorithm ensures that each step of the algorithm must be clear and unambiguous.	effectiveness	finiteness	definiteness	All the above
6	The logical or mathematical model of a particular data organization is called as _____	Data Structure	Software Engineering	Data Mining	Data Ware Housing
7	An algorithms _____ is measured in terms of computing time ad space consumed by it.	performance	effectiveness	finiteness	definiteness
8	_____ is a set of steps of operations to solve a problem.	Sub-Problem	Sub-Task	Algorithm	Process
9	_____ from any programming languages.	independent	dependent	Concept	Based

10	Pseudocode is a _____ description of an algorithm.	Low-level	Middle-level	High-level	Bottom-level
11	An algorithm is a _____ definition with some specific characteristics.	Informal	Formal	Descriptive	Customized
12	Bubble sort does not require additional _____ when compared to merge sort.	Complexity	Memory	Time	Time and Memory
13	Identification of the properties of input data is called as _____.	Problem's precondition	Problem's postcondition	Problem's in buttons	Problem's outcondition
14	_____ involves breaking the problem into sub-problems	Divide	Break	Both a & b	combine
15	Divide and Conquer is a _____ step process.	5	4	3	2
16	Tree represents the nodes connected by _____.	Edges	Arrows	Tables	Squares
17	Binary Tree is a special _____ used for data storage purposes.	Element	data structure	Object	Representation
18	In a binary tree each node can have a maximum of _____.	three nodes	Two Pairs	two children	Both a & b.
19	_____ refers to the sequence of nodes along the edges of a tree.	Edge	Root	Path	Traversal
20	_____ number is the minimum number of colors required to color a graph.	Vertex	Node	Chromatic	non-Chromatic
21	A _____ is an equation that describes a function in terms of its value.	Binary search	Recurrence	Tree	Graph
22	Interconnected objects in a graph are called _____.	Trees	Graphs	Entities	Vertices
23	Recurrences are generally used in _____ paradigm.	Interface	Graph theory	Divide-and-conquer	Both a & b
24	The naive string-matching procedure can be interpreted graphically as sliding a _____	template	Granules	Recursive	Interface

25	String-matching algorithms are used for _____	Graphics	Characters	Pattern searching	Aggregation
26	Which of the following technique performs pre-processing?	Naive	Rabin	KMP	Morris
27	The zero-length string is denoted as _____.	Null string	Empty string	Void string	Exit
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.
29	A matrix of the form (row, col, n) is otherwise known as _____.	Transpose matrix	Inverse Matrix	Sparse Matrix	None of the above.
30	Which of the following is a valid linear data structure.	Stacks	Records	Trees	Graphs
31	Which of the following is a valid non - linear data structure.	Stacks	Trees	Queues	Linked list.
32	A list of finite number of homogeneous data elements are called as _____	Stacks	Records	Arrays	Linked list.
33	No of elements in an array is called the _____ of an array.	Structure	Height	Width	Length.
34	_____ is the art of creating sample data upon which to run the program	Testing	Designing	Analysis	Debugging
35	If a program fail to respond corectly then _____ is needed to determine what is wrong and how to correct it.	Testing	Designing	Analysis	Debugging
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
37	A _____ is a collection of elements such that each element has been assigned a _____	Priority Queue	De Queue	Circular Queue	En Queue

38	A _____ is made up of Operators and Operands.	Stack	Expression	Linked list	Queue
39	A _____ is a procedure or function which calls itself.	Stack	Recursion	Queue	Tree
40	An example for application of stack is _____.	Time sharing computer system	Waiting Audience	Processing of subroutines	None of the above
41	An example for application of queue is _____.	Stack of coins	Stack of bills	Processing of subroutines	Job Scheduling in TimeSharing computers
42	Combining elements of two similar data structure into one is called _____	Merging	Insertion	Searching	Sorting
43	Adding a new element into a data structure called	Merging	Insertion	Searching	Sorting
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
45	Arranging the elements of a data structure in some type of order is called _____.	Merging	Insertion	Searching	Sorting
46	The size or length of an array = _____.	UB – LB + 1	LB + 1	UB - LB	UB – 1
47	The _____ model of a particular data organization is called as Data Structure	software Engineering	logical or mathematical	Data Mining	Data Ware Housing
48	Combining elements of two _____ data structure into one is called Merging	Similar	Dissimilar	Even	Un Even
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

50	Length of an array is defined as _____ of elements in it.	Structure	Height	Size	Number
51	In _____ search method the search begins by examining the record in the middle of the file	sequential	fibonacci	binary	non-sequential
52	_____ is a internal sorting method.	sorting with disks	quick sort	balanced merge sort	sorting with tapes
53	Quick sort reads _____ space to implement the recursion	stack	queue	circular stacks	circular queue
54	The most popular method for sorting on external storage devices is _____.	quick sort	radix sort	merge sort	heap sort
55	The 2-way merge algorithm is almost identical to the _____ procedure.	quick	merge	heap	radix
56	A _____ merge on m runs requires at most $\lceil \log_k m \rceil$ passes over the data.	n-way	m-way	k-way	q-way
57	Associating an element of an ordered list A_i with an index i is called _____	linear	sequential	ordered	indexed
58	_____ is a set of pairs, index and value.	stack	queue	Arrays	Set
58	The design approach where the main task is decomposed into subtasks and each subtask is further decomposed into simpler subtasks	top down approach	bottom up approach	hierarchical approach	merging 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

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.



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



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



KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

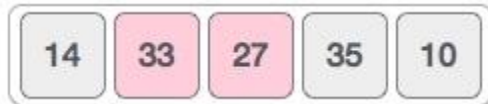
COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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



The new array should look like this –



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



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



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



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



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 –



KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

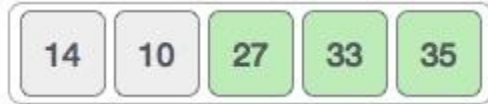
COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

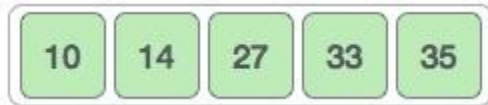
UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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



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



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()
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
{
    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:

```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
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.



Insertion sort compares the first two elements.



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

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



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



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



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



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



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

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++)
    {
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

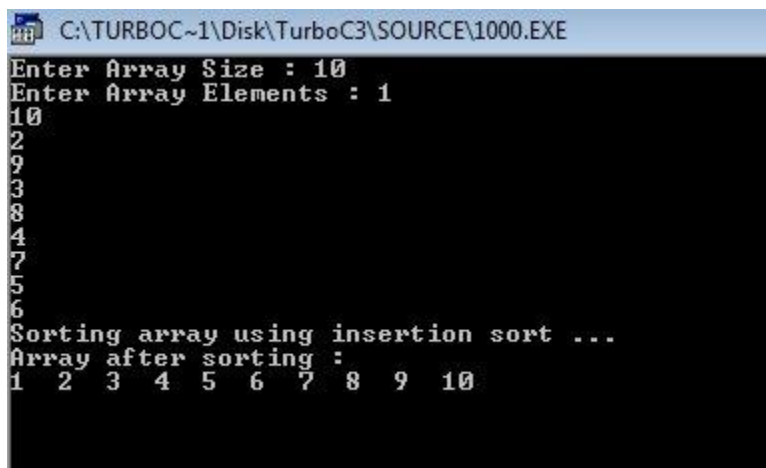
COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
        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:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
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:

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

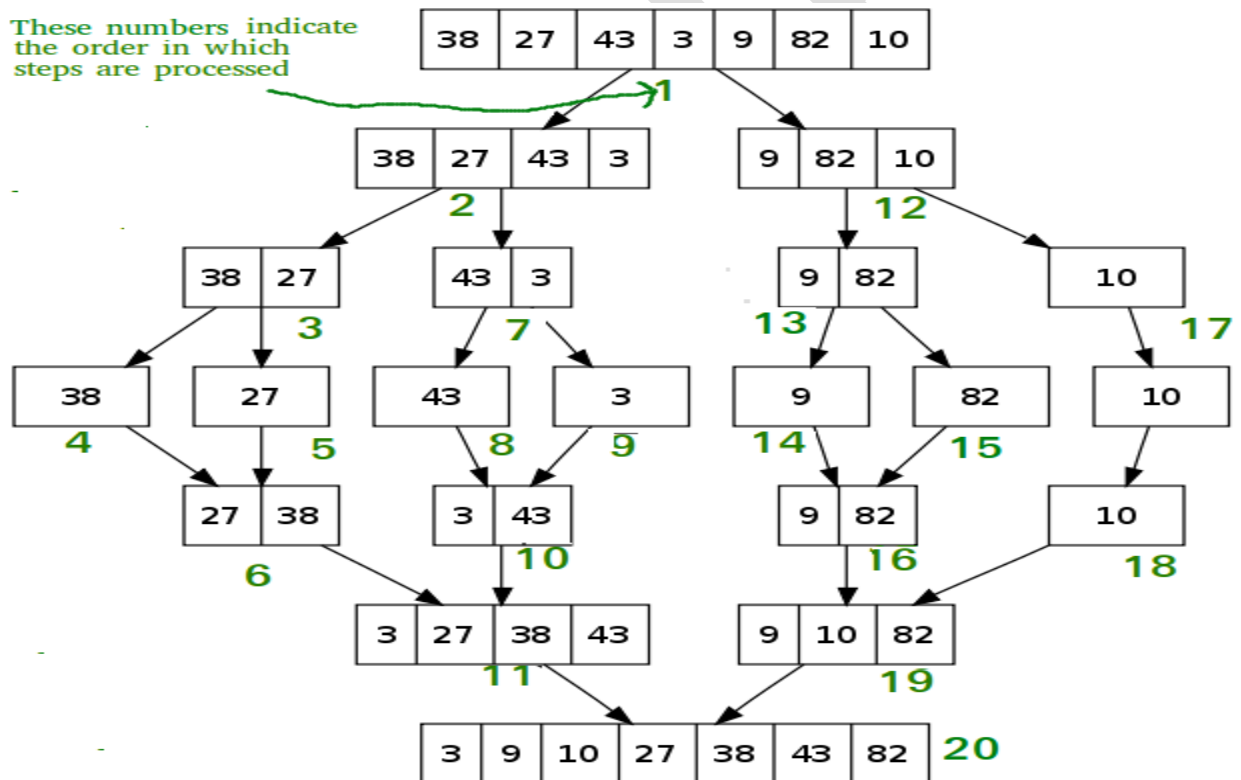
UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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)
{
    int i, 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];
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
        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:



```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe  
enter the elements  
5  
4  
3  
2  
1  
sorted array  
1  
2  
3  
4  
5  
the no. of comparisons:  
5
```

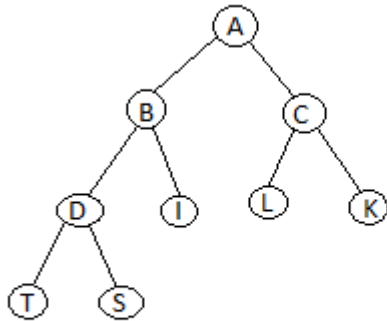
Heap Sort

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 :

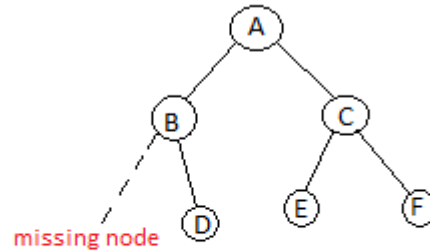
- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

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

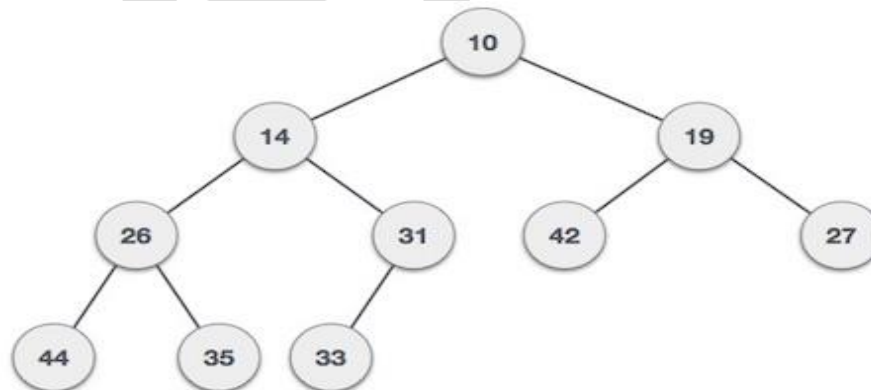


In-Complete Binary Tree

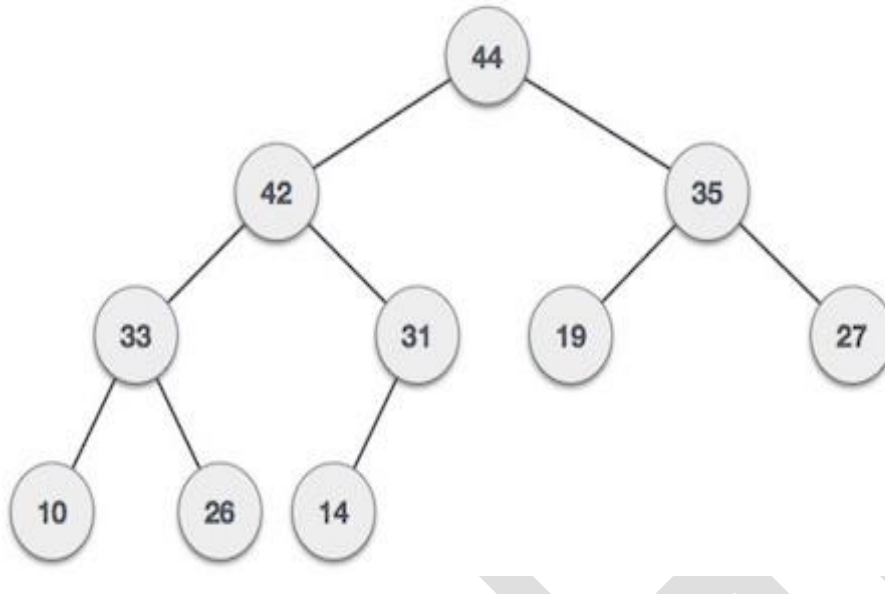
2. **Heap Property** : All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

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:

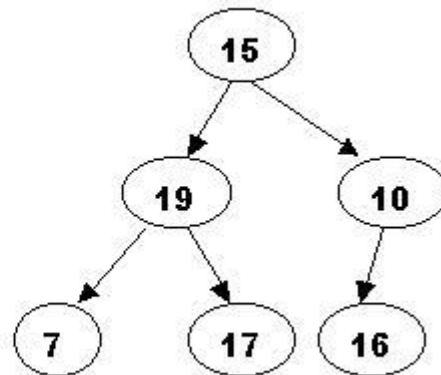
- a. when building the heap tree, part of the array will be considered as the heap, and the rest part - the original array.
- b. when sorting, part of the array will be the heap, and the rest part - the sorted array.

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

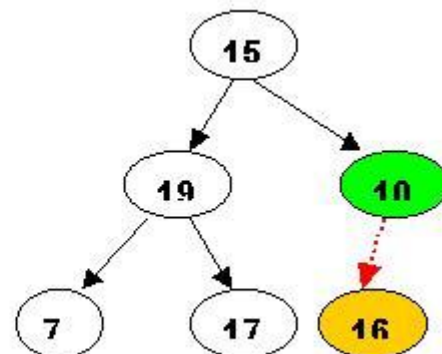
Here is the array: 15, 19, 10, 7, 17, 6

A. Building the heap tree

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



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:



After processing array[3] the situation is:

KARPAGAM ACADEMY OF HIGHER EDUCATION

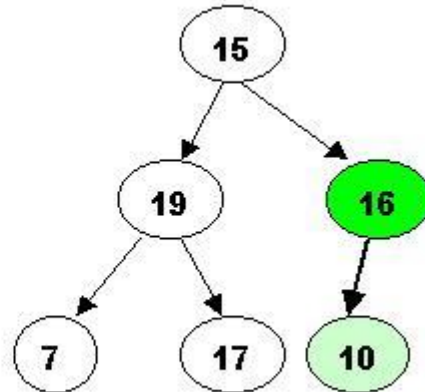
CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

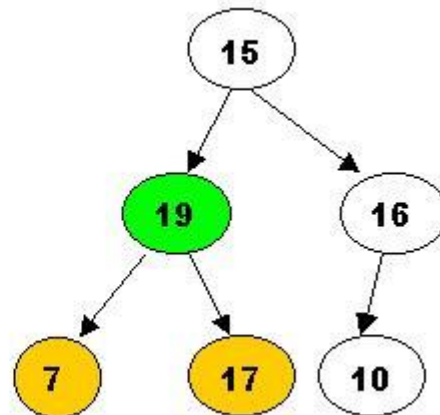
COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019



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].

KARPAGAM ACADEMY OF HIGHER EDUCATION

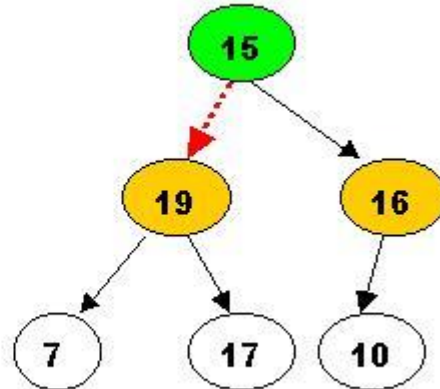
CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

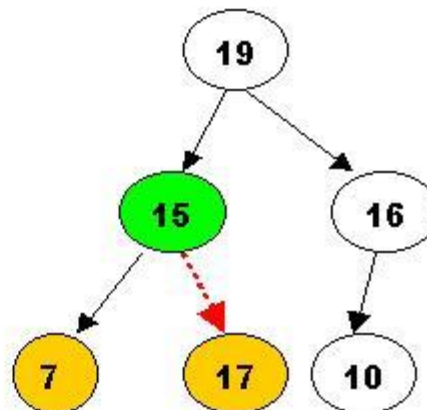
COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019



As a result the situation is:



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

KARPAGAM ACADEMY OF HIGHER EDUCATION

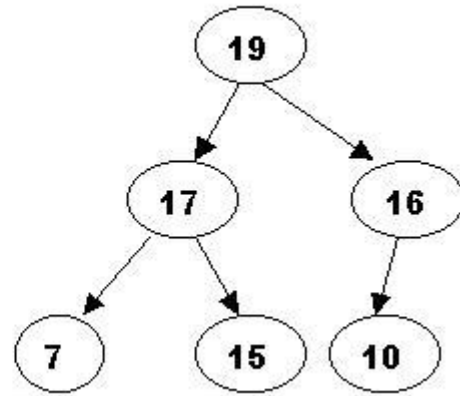
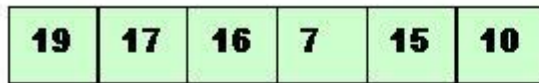
CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019



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

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])
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
        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;
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
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;
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

}

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

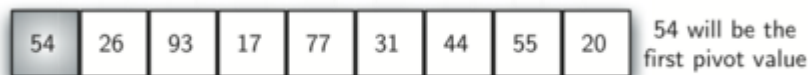
QuickSort

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.



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.

KARPAGAM ACADEMY OF HIGHER EDUCATION

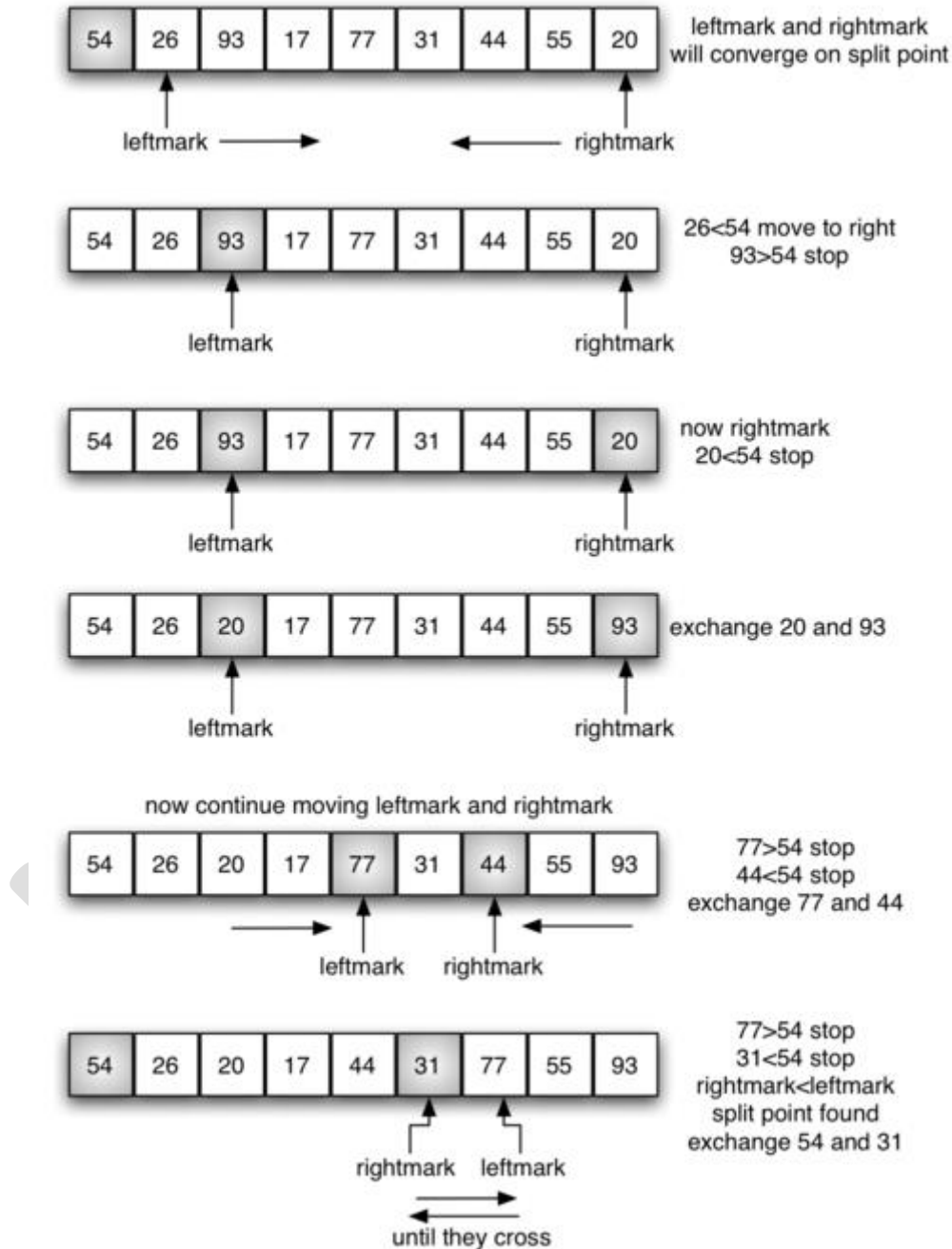
CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

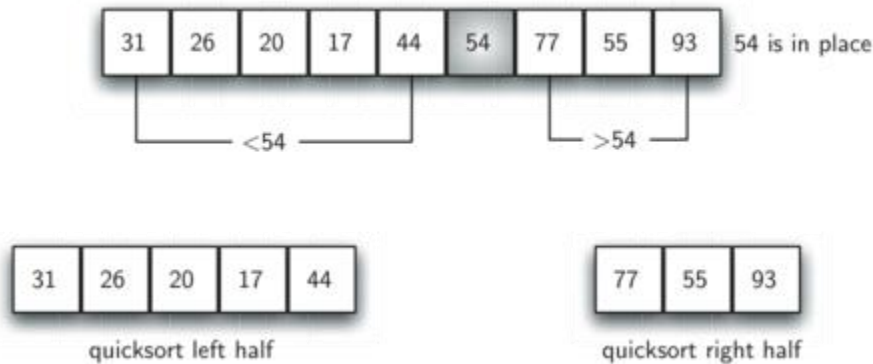
BATCH: 2016-2019



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
```


KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
}  
  
}
```

Program:

```
#include<iostream>  
using namespace std;  
  
void QUICKSORT(int [],int ,int );  
int PARTITION(int [],int,int );  
  
int main()  
{  
    int n;  
    cout<<"Enter the size of the array"<<endl;  
    cin>>n;  
    int a[n];  
    cout<<"Enter the elements in the array"<<endl;  
    for(int i=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(int i=1;i<=n;i++)  
    {  
        cout<<"a["<<i<<"]="<<a[i]<<endl;  
    }  
    return 0;  
}  
  
void QUICKSORT(int a[],int p,int r)  
{  
    int q;  
    if(p<r)  
    {  
        q=PARTITION(a,p,r);  
        QUICKSORT(a,p,q-1);  
        QUICKSORT(a,q+1,r);  
    }
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
    }  
  }  
  
int PARTITION(int a[], int p, int r)  
{  
    int temp, temp1;  
    int x = a[r];  
    int i = 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

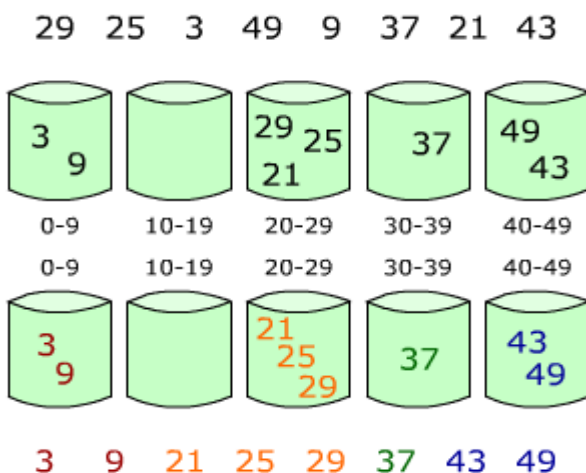
Bucket sort is a sorting algorithm that works by partitioning an array into a number of buckets.

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]]

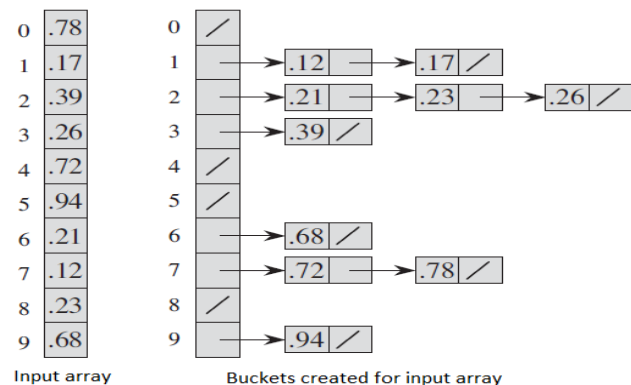
3) Sort individual buckets using insertion sort.

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>
```

```
using namespace std;
```

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

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
void bucketSort(float arr[], int n)
{
    // 1) Create n empty buckets
    vector<float> b[n];

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

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

    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for(int i = 0; i < n; i++)
        for(int j = 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
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
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

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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
-------	---------

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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.

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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	1 st Pass	2 nd Pass	3 rd 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>
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
using namespace std;
```

```
// Get maximum value from array.
```

```
int getMax(int arr[], int n)
```

```
{
```

```
    int max = arr[0];
```

```
    for(int i=1; i<n; i++)
```

```
        if(arr[i] > max)
```

```
            max = arr[i];
```

```
    return max;
```

```
}
```

```
// Count sort of arr[].
```

```
void countSort(int arr[], int n, int exp)
```

```
{
```

```
    // 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]++;
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
// 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);
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
// 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 ";
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

```
    for(i=0;i<n;i++)  
        cout<<"->"<<arr[i];  
  
    return 0;  
  
}
```

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 find 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 techniques that are being followed in data structure are 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 go to step 6
Step 4: Set i to $i + 1$
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)
    for each item in the list
        if match item == value
            return the item's location
    end if
```

```
end for  
end procedure
```

Binary Search

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

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

How Binary Search Works?

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

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

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

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 $\text{mid} + 1$ and find the new mid value again.

$$\begin{aligned}\text{low} &= \text{mid} + 1 \\ \text{mid} &= \text{low} + (\text{high} - \text{low}) / 2\end{aligned}$$

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

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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 i th order statistic of a set of n elements is the i th smallest element.
- The minimum is the first order statistic ($i = 1$).
- The maximum is the n th 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 i th order statistic of a set and what is the running time?

- Input: A set A of n (distinct) number and a number i , with $1 \leq i \leq n$.
- Output: The element $x \in A$ 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)

1. $\text{min} \leftarrow A[1]$
2. for $i \leftarrow 2$ to $\text{length}[A]$
3. do if $\text{min} > A[i]$
4. then $\text{min} \leftarrow A[i]$
5. return min

Selection in expected linear time

In fact, selection of the i th smallest element of the array A can be done in $\Theta(n)$ 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

1. RANDOMIZED-SELECT(A, p, r, i)
2. if $p = r$
3. then return $A[p]$
4. $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$
5. $k \leftarrow q - p + 1$
6. if $i = k$ /* the pivot value is the answer */
7. then return $A[q]$

8. elseif $i < k$
9. then return RANDOMIZED-SELECT($A, p, q - 1, i$)
10. 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 at least 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 let's 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;

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT II: SORTING AND SEARCHING TECHNIQUES

BATCH: 2016-2019

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 $N \cdot \log(N)$. 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

$O(\text{expression})$ 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.

$\Omega(\text{expression})$ 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.

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

POSSIBLE QUESTIONS

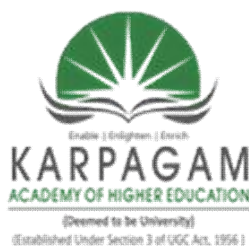
UNIT II

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 2016-2019)

Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARK QUESTIONS

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

13	_____ technique is used to find the complexity of a recurrence relation.	Greedy	Knapsack	Master's Theorem	Analysis
14	_____ algorithm traverses a graph in a breadth ward motion.	Breadth First Search	Greedy	Aggregate	Amortized
15	_____ provides a bound on the actual cost of the entire sequence.	Tree	Graph	Amortized analysis	Aggregation
16	Vertices are also known as _____	Queue	Edges	Nodes	Stack
17	In _____, each character is scanned atmost once.	Naive	Rabin	Karp	finite automata
18	_____ method is applied for two-dimensional pattern matching.	Greedy	Kruskal	Prims	Rabin and Karp
19	The average-case running time of Rabin and Karp is _____.	Worse	high	Average	All of these
20	Knuth-Morris-Pratt algorithm is a _____ time string-matching algorithm.	Linear	non-linear	Directive	non-directive
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
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
23	Problem in _____ allocation stratergy is all small nodes collect in the front of the av-list.	Best Fit	First Fit	Worst Fit	Next Fit
24	_____ is the storage allocation method that fits the program into the largest block available.	Best Fit	First Fit	Worst Fit	Next Fit
25	The back pointer for each node will be referred as _____.	Blink	Break	Back	Clear
26	Forward pointer for each node will be referred as _____.	Forward	Flink	Front	Data
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

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

39	In Available Space list, the last word of each free block has _____ fields.	4	3	2	1
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
41	In Available Space list ,Tag field has the value one when the block is _____	Allocated	Coalesced	Free	Merge
42	Available Space list ,Tag field has the value Zero when the block is _____	Allocated	Coalesced	Free	Merged
43	The _____ field of each storage block indicates if the block is free are in-use.	rlink	tag	size	uplink
44	In storage management the _____ field of the free block points to the start of the block	rlink	llink	uplink	top
45	_____ is the process of collecting all unused nodes and returning them to the available space.	Compaction	Coalescing	Garbage collection	Deallocation
46	Moving all free nodes aside to form a single contiguous block of memory is called _____	Compaction	Coalescing	Garbage collection	Deallocation
47	_____ of disk space reduces the average retrieval time of allocation.	Compaction	Coalescing	Garbage collection	Deallocation
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
49	Which of these sorting algorithm uses the Divide and Conquer technique for sorting	selection sort	insertion sort	merge sort	heap sort
50	Which of these searching algorithm uses the Divide and Conquer technique for sorting	Linear search	Binary search	fibonacci search	None of the above
51	The disadvantage of _____ sort is that it needs a temporary array to sort.	Quick	Merge	Heap	Insertion
52	A _____ is a set of characters is called a string.	Array	String	Heap	List

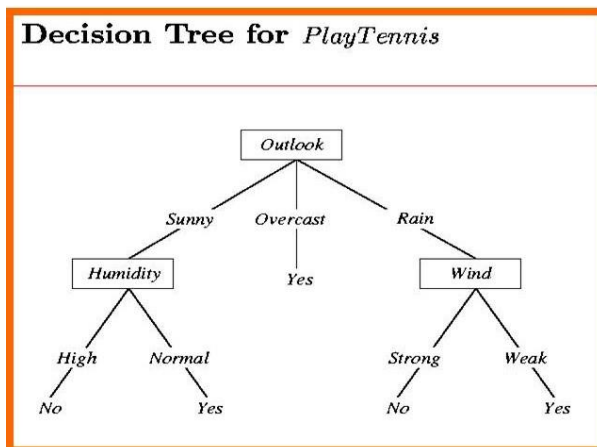
53	The straight forward find operation for pattern matching, pat of size m in string of size n needs _____ time.	O(mn)	$O(n^2)$	$O(m^2)$	$O(m+n)$
54	Knuth, Morris and Pratt's method of pattern matching in strings takes _____ time, if pat is of size m and string is size n.	$O(mn)$	$O(n^2)$	$O(m^2)$	O(m+n)
55	_____ representation always need extensive data movement.	Linked	sequential	tree	graph
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

UNIT III
SYLLABUS

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

Decision Tree

Decision tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.



A decision tree for the concept PlayTennis.

Construction of Decision Tree :

A tree can be “*learned*” by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called *recursive partitioning*. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. The construction of decision tree classifier does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. In general decision tree classifier has good accuracy. Decision tree induction is a typical inductive approach to learn knowledge on classification.

Decision Tree Representation :

Decision trees classify instances by sorting them down the tree from the root to some leaf node,

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

which provides the classification of the instance. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute as shown in the above figure. This process is then repeated for the subtree rooted at the new node.

The decision tree in above figure classifies a particular morning according to whether it is suitable for playing tennis and returning the classification associated with the particular leaf. (in this case Yes or No).

For example, the instance

(Outlook = Rain, Temperature = Hot, Humidity = High, Wind = Strong)

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance.

In other words we can say that decision tree represent a disjunction of conjunctions of constraints on the attribute values of instances.

(Outlook = Sunny ^ Humidity = Normal) v (Outlook = Overcast) v (Outlook = Rain ^ Wind = Weak)

Strengths and Weakness of Decision Tree approach

The strengths of decision tree methods are:

- Decision trees are able to generate understandable rules.
- Decision trees perform classification without requiring much computation.
- Decision trees are able to handle both continuous and categorical variables.
- Decision trees provide a clear indication of which fields are most important for prediction or classification.

The weaknesses of decision tree methods :

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- Decision trees are prone to errors in classification problems with many class and relatively small number of training examples.
- Decision tree can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.

A decision tree consists of three types of nodes:^[1]

1. Decision nodes – typically represented by squares
2. Chance nodes – typically represented by circles

3. End nodes – typically represented by triangles

Decision trees are commonly used in operations research and operations management. If, in practice, decisions have to be taken online with no recall under incomplete knowledge, a decision tree should be paralleled by a probability model as a best choice model or online selection model algorithm. Another use of decision trees is as a descriptive means for calculating conditional probabilities.

Decision rules

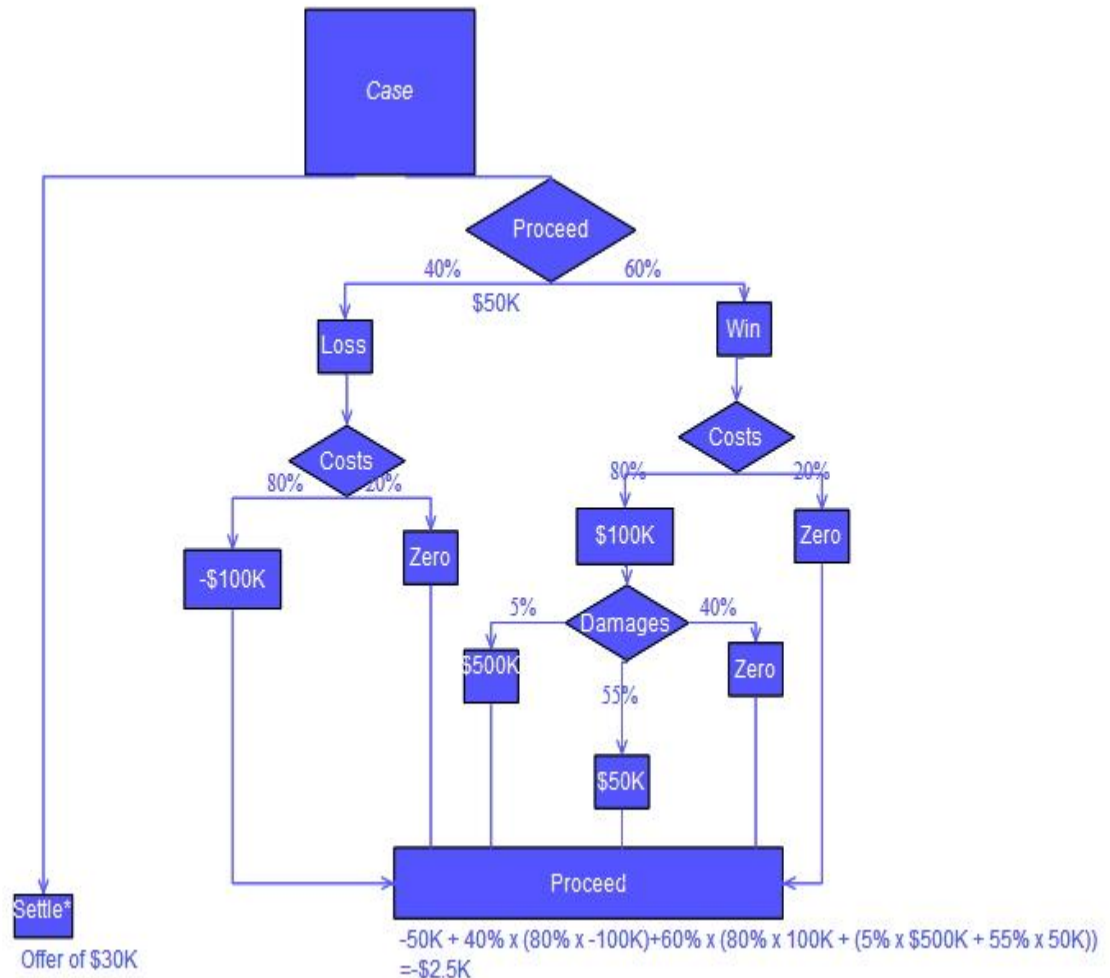
The decision tree can be linearized into **decision rules**, where the outcome is the contents of the leaf node, and the conditions along the path form a conjunction in the if clause. In general, the rules have the form:

if condition1 and condition2 and condition3 then outcome.

Decision rules can be generated by constructing association rules with the target variable on the right. They can also denote temporal or causal relations.

Decision tree using flowchart symbols[edit]

Commonly a decision tree is drawn using flowchart symbols as it is easier for many to read and understand.



Decision tree learning

Decision tree learning uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). It is one of the predictive modelling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a discrete set of values are called **classification trees**; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called **regression trees**.

In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. In data mining, a decision tree describes data (but the resulting

classification tree can be an input for decision making). This page deals with decision trees in data mining.

Decision tree learning is a method commonly used in data mining. The goal is to create a model that predicts the value of a target variable based on several input variables. An example is shown in the diagram at right. Each interior node corresponds to one of the input variables; there are edges to children for each of the possible values of that input variable. Each leaf represents a value of the target variable given the values of the input variables represented by the path from the root to the leaf.

A decision tree is a simple representation for classifying examples. For this section, assume that all of the input features have finite discrete domains, and there is a single target feature called the "classification". Each element of the domain of the classification is called a *class*. A decision tree or a classification tree is a tree in which each internal (non-leaf) node is labeled with an input feature. The arcs coming from a node labeled with an input feature are labeled with each of the possible values of the target or output feature or the arc leads to a subordinate decision node on a different input feature. Each leaf of the tree is labeled with a class or a probability distribution over the classes.

Left: A partitioned two-dimensional feature space. These partitions could not have resulted from recursive binary splitting. Middle: A partitioned two-dimensional feature space with partitions that did result from recursive binary splitting. Right: A tree corresponding to the partitioned feature space in the middle. Notice the convention that when the expression at the split is true, the tree follows the left branch. When the expression is false, the right branch is followed.

A tree can be "learned" by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. See the examples illustrated in the figure for spaces that have and have not been partitioned using recursive partitioning, or recursive binary splitting. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This process of *top-down induction of decision trees* (TDIDT) is an example of a greedy algorithm, and it is by far the most common strategy for learning decision trees from data.

In data mining, decision trees can be described also as the combination of mathematical and computational techniques to aid the description, categorization and generalization of a given set of data.

Decision tree types

- **Classification tree** analysis is when the predicted outcome is the class to which the data belongs.
- **Regression tree** analysis is when the predicted outcome can be considered a real number (e.g. the price of a house, or a patient's length of stay in a hospital).

The term **Classification And Regression Tree (CART)** analysis is an umbrella term used to refer to both of the above procedures, first introduced by Breiman et al. Trees used for regression and trees used for classification have some similarities - but also some differences, such as the procedure used to determine where to split.

Some techniques, often called *ensemble* methods, construct more than one decision tree:

- **Boosted trees** Incrementally building an ensemble by training each new instance to emphasize the training instances previously mis-modeled. A typical example is AdaBoost. These can be used for regression-type and classification-type problems.
- **Bootstrap aggregated** (or bagged) decision trees, an early ensemble method, builds multiple decision trees by repeatedly resampling training data with replacement, and voting the trees for a consensus prediction.
- A **random forest** classifier is a specific type of bootstrap aggregating
- **Rotation forest** - in which every decision tree is trained by first applying principal component analysis (PCA) on a random subset of the input features.

A special case of a decision tree is a decision list, which is a one-sided decision tree, so that every internal node has exactly 1 leaf node and exactly 1 internal node as a child (except for the bottommost node, whose only child is a single leaf node). While less expressive, decision lists are arguably easier to understand than general decision trees due to their added sparsity, permit non-greedy learning methods and monotonic constraints to be imposed.

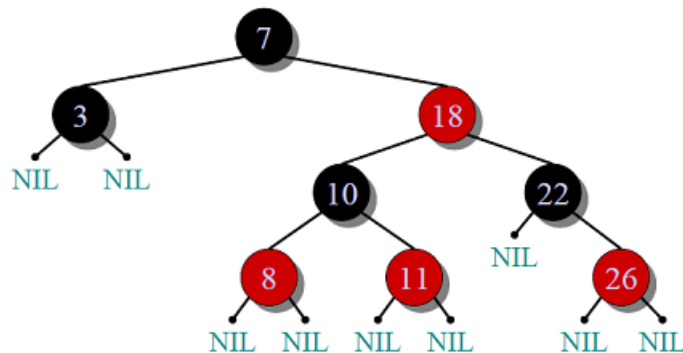
Decision tree learning is the construction of a decision tree from class-labeled training tuples. A decision tree is a flow-chart-like structure, where each internal (non-leaf) node denotes a test on an attribute, each branch represents the outcome of a test, and each leaf (or terminal) node holds a class label. The topmost node in a tree is the root node.

There are many specific decision-tree algorithms. Notable ones include:

- ID3 (Iterative Dichotomiser 3)
- C4.5 (successor of ID3)
- CART (Classification And Regression Tree)
- CHAID (CHi-squared Automatic Interaction Detector). Performs multi-level splits when computing classification trees.
- MARS: extends decision trees to handle numerical data better.
- Conditional Inference Trees. Statistics-based approach that uses non-parametric tests as splitting criteria, corrected for multiple testing to avoid overfitting. This approach results in unbiased predictor selection and does not require pruning.

Red-Black Tree

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.



- 1) Every node has a color either red or black.
- 2) Root of tree is always black.
- 3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- 4) Every path from root to a NULL node has same number of black nodes.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

Comparison with AVL Tree

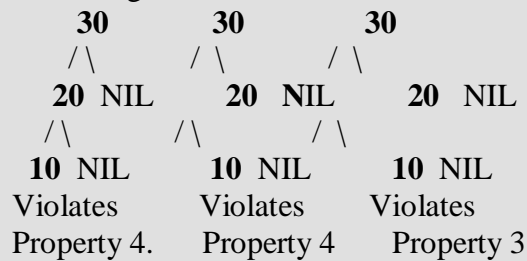
The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

How does a Red-Black Tree ensure balance?

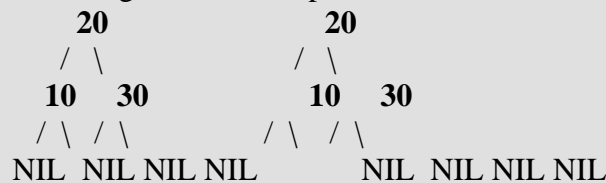
A simple example to understand balancing is, a chain of 3 nodes is not possible in red black tree. We can try any combination of colors and see all of them violate Red-Black tree property.

A chain of 3 nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Black Height of a Red-Black Tree :

Black height is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, **a node of height h has black-height $\geq h/2$.**

Every Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

This can be proved using following facts:

- 1) For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k \leq 2\log_2(n+1)$
- 2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
- 3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lfloor n/2 \rfloor$ where n is total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

Insertion

In AVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

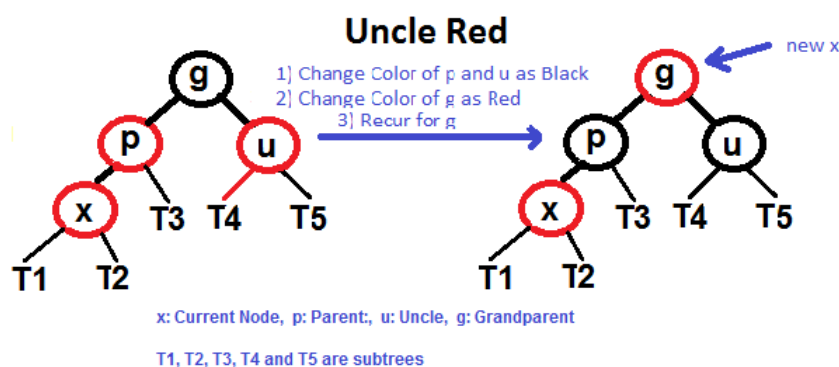
- 1) Recoloring
- 2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

- 1) Perform standard BST insertion and make the color of newly inserted nodes as RED.
- 2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
- 3) Do following if color of x 's parent is not BLACK or x is not root.
 -a) If x 's uncle is RED (Grand parent must have been black from property 4)
 -(i) Change color of parent and uncle as BLACK.
 -(ii) color of grand parent as RED.
 -(iii) Change $x = x$'s grandparent, repeat steps 2 and 3 for new x .



....b) If x 's uncle is BLACK, then there can be four configurations for x , x 's parent (p) and x 's grandparent (g) (This is similar to AVL Tree)

-i) Left Left Case (p is left child of g and x is left child of p)
-ii) Left Right Case (p is left child of g and x is right child of p)

.....iii) Right Right Case (Mirror of case a)

.....iv) Right Left Case (Mirror of case c)

Deletion

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, *we check color of sibling* to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

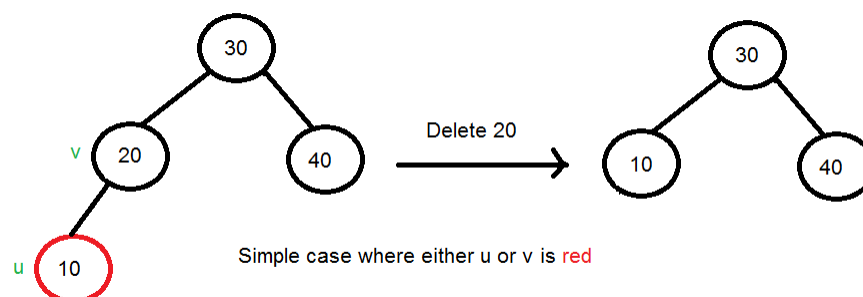
Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as *double black*. The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

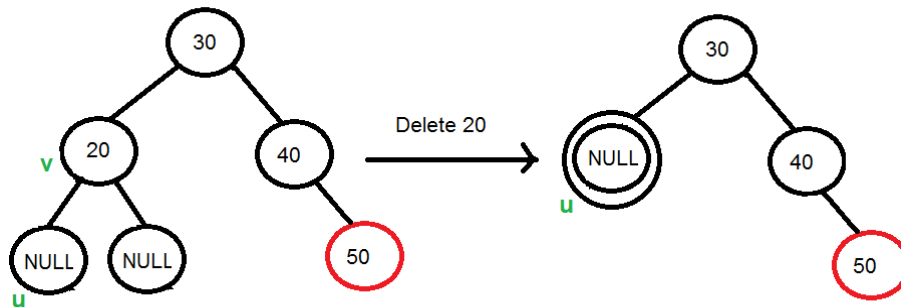
1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) **Simple Case: If either u or v is red**, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.

Note that deletion is not done yet, this double black must become single black

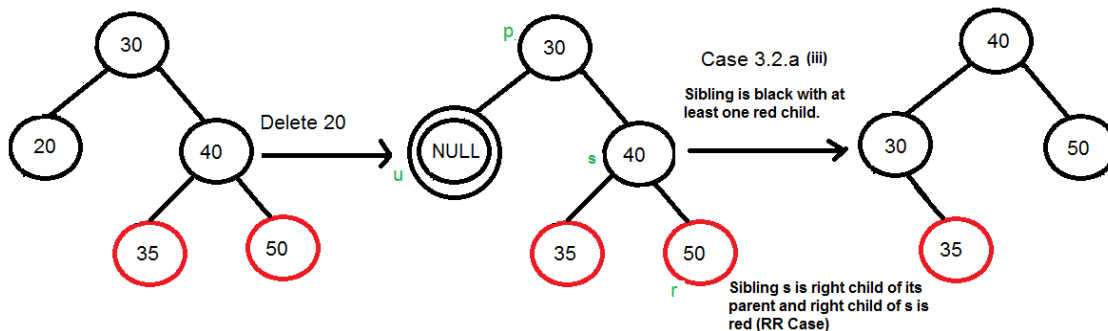
3.2) Do following while the current node u is double black and it is not root. Let sibling of node be s .

....(a): **If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be r . This case can be divided in four subcases depending upon positions of s and r .

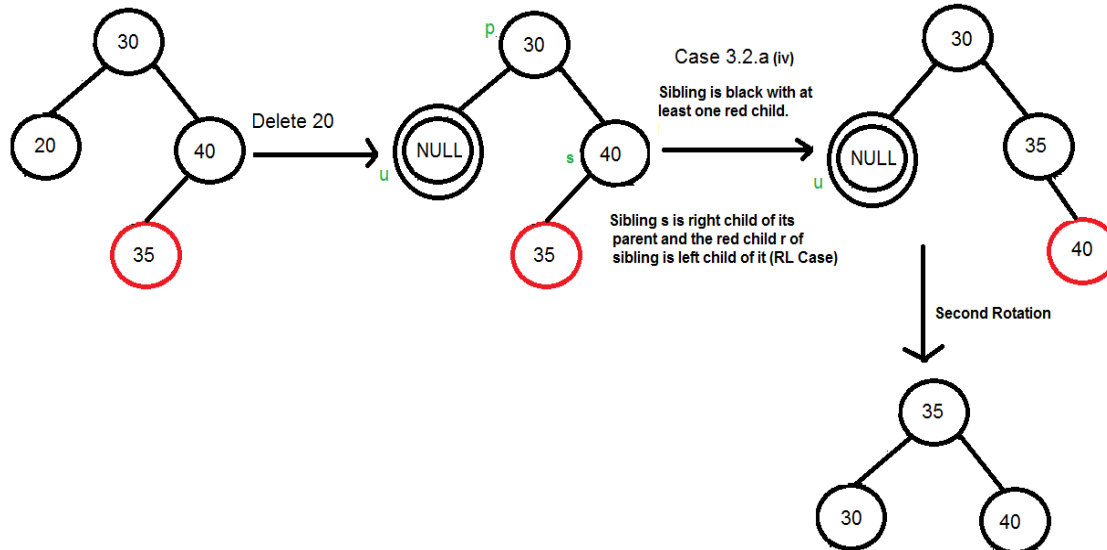
.....(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

.....(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

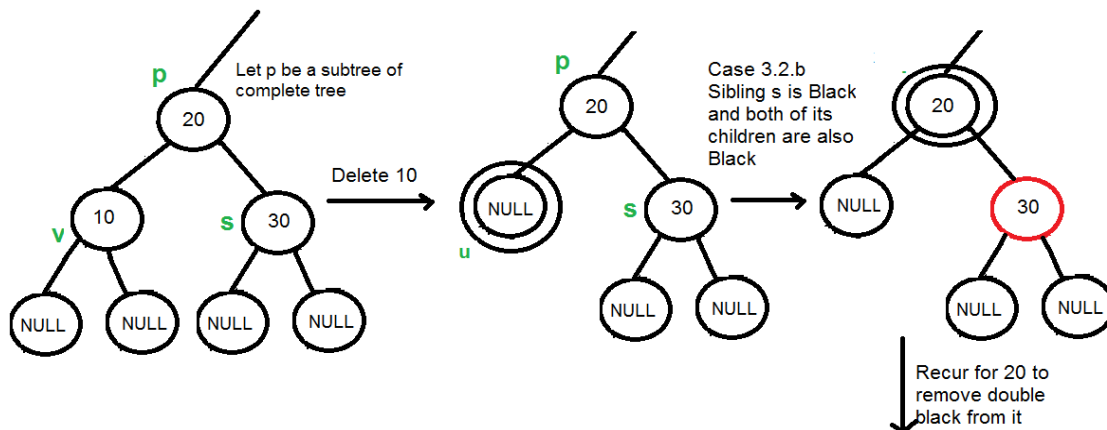
.....(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



.....(iv) Right Left Case (s is right child of its parent and r is left child of s)



.....(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

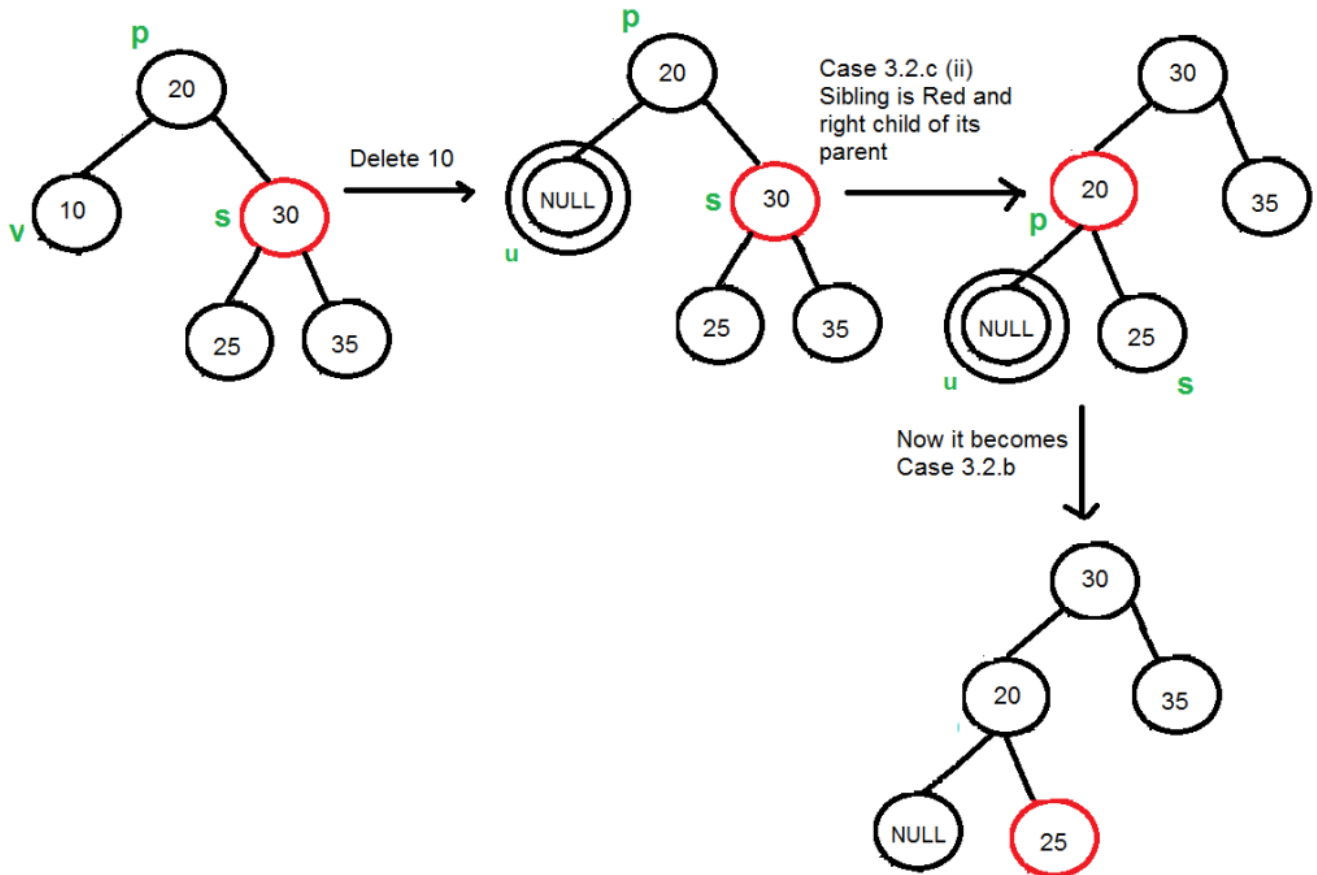


In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

.....(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

.....(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

.....(iii) Right Case (s is right child of its parent). We left rotate the parent p.



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;
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
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)
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
        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;
            rightrightrotate(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;
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
        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;
                }
            }
        }
    }
}
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
        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 ;
}
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
        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;
        }
    }
}
```


KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
        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)
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
        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)
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
                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)
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

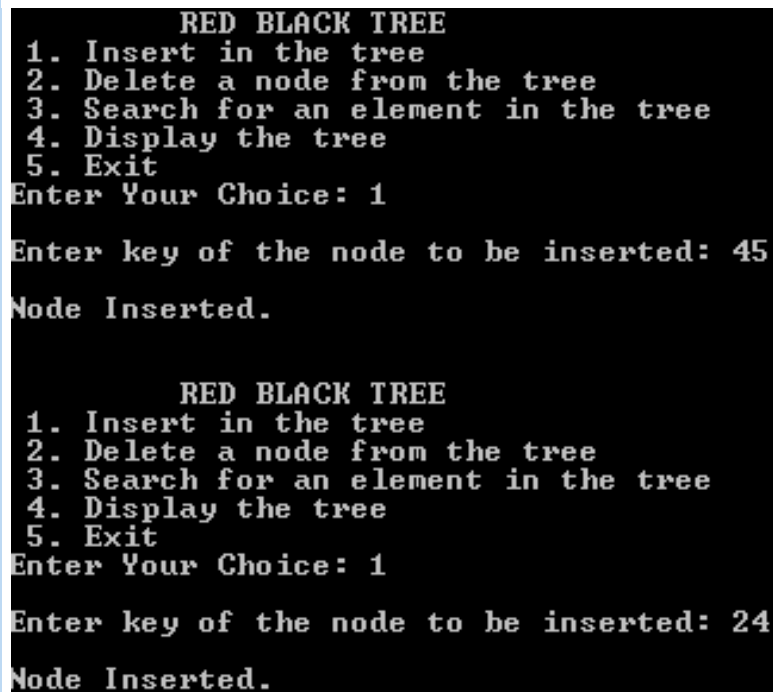
UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
{
    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++

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT III: LOWER BOUNDING TECHNIQUES

BATCH: 2016-2019

```
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 2016-2019)
Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARK QUESTIONS

S.NO	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4
1	Algorithms should have explicitly defined set of _____	Inputs	Outputs	Inputs and Outputs	Instructions
2	_____ means to establish the amount of resources.	Efficiency	Module	Program	Process
3	_____ is to verify if the algorithm leads to the solution of the problem.	Efficiency	Flexibility	Durability	Correctness
4	_____ stage recursively combines the sub problems.	Merge	Divide	Break	Solve
5	_____ sort is a comparison based algorithm.	Bubble	Insertion	a & b	None of these
6	The minimum number of steps taken on any instance is called as _____.	Best-case	Worst-case	Average-case	a & b
7	The algorithm's _____ is the set of values corresponding to all the variables.	Process	State	Definition	d. Technique
8	Bubble sort is not suitable for _____	larger data set	Smaller data set	Medium data set	Both b & c
9	_____ means passing through nodes in a specific order.	Interchanging	Traversing	Dividing	Splitting
10	Level of a node represents the _____ of a node.	Priority	Object	Generation	Both a & b
11	_____ represents a value of a node.	Key	Instance	Type	Variable
12	Red-Black tree is one of the _____ binary search tree.	Balanced	Unbalanced	Different	Similar
13	The method gives a global view of a problem.	Manual	Analysis	aggregate	Recurrence
14	In _____ method different charges are assigned to different operations.	Accounting	Recurrence	Aggregation	All of these
15	The spanning tree cannot be _____.	Union	Intersection	Disconnected	Updated

16	_____ method represents the prepaid work as potential energy.	Graph theory	Potential	Aggregation	Accounting
17	The _____ string-matching procedure can be interpreted graphically.	naive	Pattern	KMP	Automotive
18	Rabin and Karp method is applied for _____ pattern matching.	3D	1D	2D	4D
19	_____ algorithm is a Linear time string-matching algorithm.	Greedy	b. Dynamic	Descriptive	KMP
20	The _____ case running time of Rabin and Karp is high.	Average	Minimum	Maximum	Worst
21	X is a root then X is the _____ of its children.	sub tree	Parent	Siblings	subordinate
22	The children of the same parent are called _____.	sibling	leaf	child	subtree
23	_____ of a node are all the nodes along the path from the root to that node.	Degree	sub tree	Ancestors	parent
24	The _____ of a tree is defined to be a maximum level of any node in the tree.	weight	length	breadth	height
25	A _____ is a set of $n \geq 0$ disjoint trees	Group	forest	Branch	sub tree
26	A tree with any node having at most two branches is called a _____.	branched tree	sub tree	binary tree	forest
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
28	Data structure represents the hierarchical relationships between individual data item is known as _____.	Root	Node	Tree	Address
29	Node at the highest level of the tree is known as _____.	Child	Root	Sibling	Parent
30	The root of the tree is the _____ of all nodes in the tree.	Child	Parent	Ancestor	Head
31	_____ is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree
32	A node with no children is called _____.	Root Node	Branch	Leaf Node	Null tree

33	In a tree structure a link between parent and child is called _____	Branch	Root	Leaf	Subtree
34	Height – balanced trees are also referred as as _____ trees.	AVL trees	Binary Trees	Subtree	Branch Tree
35	Visiting each node in a tree exactly once is called _____	searching	travering	walk through	path
36	In _____ traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order
37	In _____ traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order
38	In _____ traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order
39	Inorder traversal is also sometimes called _____	Symmetric Order	End Order	PreOrder	PostOrder
40	Postorder traversal is also sometimes called _____	Symmetric Order	End Order	PreOrder	PostOrder
41	Nodes of any level are numbered from _____	Left to right	Right to Left	Top to Bottom	Bottom to Top
42	_____ search involves only addition and subtraction.	binary	fibonacci	sequential	non sequential
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
44	Binary trees are used in _____ sorting.	quick sort	merge sort	heap sort	lrsort
45	The _____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch
46	In Threaded Binary Tree ,LCHILD(P) is a normal pointer When LBIT(P) = _____	1	2	3	0
47	In Threaded Binary Tree ,LCHILD(P) is a Thread When LBIT(P) = _____	1	2	3	0
48	In Threaded Binary Tree ,RCHILD(P) is a normal pointer When RBIT(P) = _____	2	1	3	0
49	In Threaded Binary Tree ,RCHILD(P) is a Thread When LBIT(P) = _____	1	2	0	4

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

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

In an amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 18.1 starts with the aggregate method, in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The amortized cost per operation is then $T(n)/n$.

When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. The credit is used later in the sequence to pay for operations that are charged less than they actually cost.

We shall use two examples to examine these three models. One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They should not appear in the code. If, for example, a credit is assigned to an object x when using the accounting method, there is no need to assign an appropriate amount to some attribute $credit[x]$ in the code.

The aggregate method

In the aggregate method of amortized analysis, we show that for all n , a sequence of n operations takes worst-case time $T(n)$ in total. In the worst case, the average cost, or amortized cost, per operation is therefore $T(n) / n$. Note that this amortized cost applies to each operation, even when

there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

In our first example of the aggregate method, we analyze stacks that have been augmented with a new operation. Section 11.1 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S .

POP(S) pops the top of stack S and returns the popped object.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n , and the actual running time for n operations is therefore $\Theta(n)$.

The situation becomes more interesting if we add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S , or pops the entire stack if it contains less than k objects. In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

MULTIPOP(S, k)

```
1 while not STACK-EMPTY( $S$ ) and  $k \neq 0$ 
2   do POP( $S$ )
3    $k \leftarrow k - 1$ 
```

The accounting method

In the *accounting method* of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its *amortized cost*. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as *credit*. Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. This is very different from the aggregate method, in which all operations have the same amortized cost.

One must choose the amortized costs of operations carefully. If we want analysis with amortized costs to show that in the worst case the average cost per operation is small, the total amortized

cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. Moreover, as in the aggregate method, this relationship must hold for all sequences of operations. Thus, the total credit associated with the data structure must be nonnegative at all times, since it represents the amount by which the total amortized costs incurred exceed the total actual costs incurred. If the total credit were ever allowed to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. Recall that the actual costs of the operations were

PUSH 1 ,
POP 1 ,
MULTIPOP $\min(k,s)$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH 2 ,
POP 0 ,
MULTIPOP 0 .

The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the **potential method** of amortized analysis represents the prepaid work as "potential energy," or just "potential," that can be released to pay for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows. We start with an initial data structure D_0 on which n operations are performed. For each $i = 1, 2, \dots, n$, we let c_i be the actual cost of the i th operation and D_i be the data structure that results after applying the i th operation to data structure D_{i-1} . A **potential function** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the **potential** associated with data structure D_i .

Breadth First Search

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.

Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

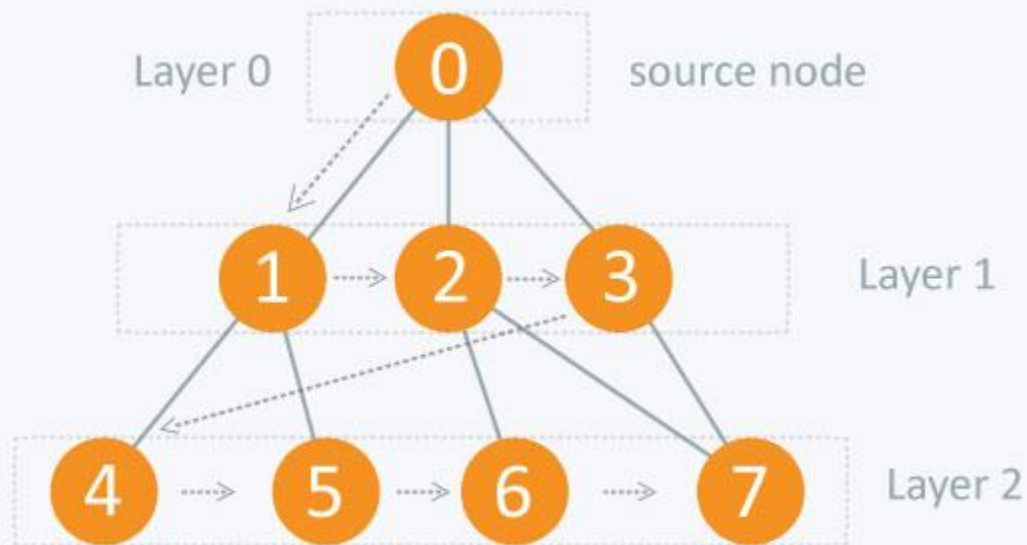
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.

Consider the following diagram.



The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

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.
```

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V; // No. of vertices

    // Pointer to an array containing adjacency
```


KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT IV: ADVANCED ANALYSIS TECHNIQUES

BATCH: 2016-2019

```
// lists
list<int> *adj;
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
```

```
{
    // Dequeue a vertex from queue and print it
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

    // Get all adjacent vertices of the dequeued
    // vertex s. If a adjacent has not been visited,
    // then mark it visited and enqueue it
    for (i = adj[s].begin(); i != adj[s].end(); ++i)
    {
        if (!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}
}
```

// Driver program to test methods of graph class

```
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Output:

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

Depth First Search

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

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)
```

Program for Depth first search

```
// C++ program to print DFS traversal from  
// a given vertex in a given graph  
#include<iostream>  
#include<list>  
using namespace std;  
  
// Graph class represents a directed graph  
// using adjacency list representation  
class Graph  
{  
    int V; // No. of vertices  
  
    // Pointer to an array containing  
    // adjacency lists  
    list<int> *adj;  
  
    // A recursive function used by DFS  
    void DFSUtil(int v, bool visited[]);  
public:  
    Graph(int V); // Constructor  
  
    // function to add an edge to graph  
    void addEdge(int v, int w);  
  
    // DFS traversal of the vertices  
    // reachable from v  
    void DFS(int v);  
};
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT IV: ADVANCED ANALYSIS TECHNIQUES

BATCH: 2016-2019

```
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main()
{
```

```
// Create a graph given in the above diagram
Graph g(4);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

cout << "Following is Depth First Traversal"
      " (starting from vertex 2) \n";
g.DFS(2);

return 0;
}
```

Output:

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

Applications of Breadth First Traversal

- 1) Shortest Path and Minimum Spanning Tree for unweighted graph** In 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 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 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. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

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.

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.)

Minimum Spanning Trees

A spanning tree is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.

- Any vertex can be reached from any other vertex.

Example

In the following graph, the highlighted edges form a spanning tree.

Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are n number of vertices, the spanning tree should have $n - 1$ number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.

In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is $(5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38$.

We will use Prim's algorithm to find the minimum spanning tree.

Prim's Algorithm

Prim's algorithm is a greedy approach to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.

Algorithm: MST-Prim's (G, w, r)

for each $u \in G.V$

$u.key = \infty$

$u.\Pi = \text{NIL}$

$r.key = 0$

$Q = G.V$

while $Q \neq \Phi$

$u = \text{Extract-Min}(Q)$

 for each $v \in G.\text{adj}[u]$

 if each $v \in Q$ and $w(u, v) < v.key$

$v.\Pi = u$

$v.key = w(u, v)$

The function Extract-Min returns the vertex with minimum edge cost. This function works on min-heap.

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**.

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 2016-2019)
Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARK QUESTIONS

S.NO	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4
1	Experimental analysis is otherwise called as _____.	Testing	Implementing	Proving	Scheduling
2	Algorithms are well ordered with _____ operations.	Efficient	Multiple	Unambiguous	Infinite
3	An algorithm is a _____	Formal	Informal	Basic	Complete
4	_____ involves the solution of sub-problems.	Merge	Combine	Divide	Conquer
5	Swapping is used to _____ the values.	Remove	b. Add	interchange	c. Integrate
6	A sub-list is maintained in _____ sort.	Bubble	Insertion	Heap	Both a & c
7	Example for Divide and conquer is _____	Merge sort	Quick sort	Bubble sort	All of these
8	Heap data structure is always a _____	Radix	complete binary tree	Heap sort	Both a & b
9	Red-Black tree is one of the _____ binary search tree.	Balanced	Unbalanced	Different	Similar
10	In Red-Black tree every node is either _____.	Black or blue	Black or orange	red or black	Both b & c
11	In Red-Black tree the root is _____.	Red	Black	Blue	Purple
12	In Red-Black tree if a node is _____, then both its children are black.	Black	Blue	Yellow	Red
13	All paths from the node have the _____ black height in Red-Black tree.	Same	Different	Null	Variable
14	_____ algorithm traverses a graph in a depthward motion.	Depth First Search	Greedy	Analysis	Both a & c
15	A _____ is a notation used to represent the connection between pairs of objects.	Graph	Tree	Binary Tree	Stack
16	Edges are the links that connect the vertices.	Nodes	Edges	Trees	Both a & c
17	In a directed graph, edges have _____.	Scalar	Vector	Direction	Time

18	The _____ string is denoted as empty string.	zero-length	Nil	Empty	max-length
19	Naive technique performs _____.	Union	Intersect	Post-processing	Pre-processing
20	The average-case running time of _____ is high.	Rabin and Karp	Prims	Naïve	Kruskal
21	_____ are genealogical charts which are used to present the data	Graphs	Pedigree and lineal chart	Line , bar chart	pie 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
23	A _____ is a graph without any cycle.	tree	path	set	list
24	In binary trees there is no node with a degree greater than _____	zero	one	two	three
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
26	The Number of subtrees of a node is called its _____.	leaf	terminal	children	degree
27	Nodes that have degree zero are called _____.	end node	leaf nodes	subtree	root node
28	A binary tree with all its left branches suppressed is called a _____	balanced tree	left sub tree	full binary tree	right skewed tree
29	All node except the leaf nodes are called _____.	terminal node	percent node	non terminal	children node
30	The roots of the subtrees of a node X, are the _____ of X.	Parent	Children	Sibling	sub tree
31	X is a root then X is the _____ of its children.	sub tree	Parent	Siblings	subordinate
32	The children of the same parent are called _____.	sibling	leaf	child	subtree
33	_____ of a node are all the nodes along the path from the root to that node.	Degree	sub tree	Ancestors	parent
34	The _____ of a tree is defined to be a maximum level of any node in the tree.	weight	length	breath	height
35	A _____ is a set of $n \geq 0$ disjoint trees	Group	forest	Branch	sub tree

36	A tree with any node having at most two branches is called a _____.	branched tree	sub tree	binary tree	forest
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
38	Data structure represents the hierarchical relationships between individual data item is known as _____.	Root	Node	Tree	Address
39	Node at the highest level of the tree is known as _____.	Child	Root	Sibling	Parent
40	The root of the tree is the _____ of all nodes in the tree.	Child	Parent	Ancestor	Head
41	_____ is a subset of a tree that is itself a tree.	Branch	Root	Leaf	Subtree
42	A node with no children is called _____.	Root Node	Branch	Leaf Node	Null tree
43	In a tree structure a link between parent and child is called _____.	Branch	Root	Leaf	Subtree
44	Height – balanced trees are also referred as _____ trees.	AVL trees	Binary Trees	Subtree	Branch Tree
45	Visiting each node in a tree exactly once is called _____.	searching	travering	walk through	path
46	In _____ traversal ,the current node is visited before the subtrees.	PreOrder	PostOrder	Inorder	End Order
47	In _____ traversal ,the node is visited between the subtrees.	PreOrder	PostOrder	Inorder	End Order
48	In _____ traversal ,the node is visited after the subtrees.	PreOrder	PostOrder	Inorder	End Order
49	Inorder traversal is also sometimes called _____.	Symmetric Order	End Order	PreOrder	PostOrder
50	Postorder traversal is also sometimes called _____.	Symmetric Order	End Order	PreOrder	PostOrder
51	Nodes of any level are numbered from _____.	Left to right	Right to Left	Top to Bottom	Bottom to Top
52	_____ search involves only addition and subtraction.	binary	fibonacci	sequential	non sequential

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
54	Binary trees are used in _____ sorting.	quick sort	merge sort	heap sort	lrsort
55	The _____ of the heap has the largest key in the tree.	Node	Root	Leaf	Branch
56	Which of these searching algorithm uses the Divide and Conquer technique for sorting	Linear search	Binary search	fibonacci search	None of the above
57	_____ algorithm can be used only with sorted lists.	Linear search	Binary search	insertion sort	merge sort
58	_____ search involves comparison of the element to be found with every elements in a list.	Linear search	Binary search	fibonacci search	None of the above
59	Binary search algorithm in a list of n elements takes only _____ time.	$O(\log_2 n)$	$O(n)$	$O(n^3)$	$O(n^2)$
60	_____ is used for decision making in eight coin problem.	trees	graphs	linked lists	array

UNIT V
SYLLABUS

String Processing: String Matching, KMP Technique.

String Matching

Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem can greatly aid the responsiveness of the text-editing program.

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

We formalize the string-matching problem as follows. We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ . For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \dots, z\}$. The character arrays P and T are often called strings of characters.

We say that pattern P occurs with shift s in text T (or, equivalently, that pattern P occurs beginning at position $s + 1$ in text T) if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$ (that is, if $T[s + j] = P[j]$, for $1 \leq j \leq m$).

If P occurs with shift s in T , then we call s a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T .

Except for the naive brute-force algorithm, each string-matching algorithm in this chapter performs some preprocessing based on the pattern and then finds all valid shifts; we will call this latter phase “matching.”

The total running time of each algorithm is the sum of the preprocessing and matching times. Although the $\Theta((n - m + 1)m)$ worst-case running time of this algorithm is no better than that of the naive method, it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems.

A string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern P in a text.

The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s+1..s+m]$ for each of the $n-m+1$ possible values of s .

NAIVE-STRING-MATCHER(T, P)

```
1  $n \leftarrow \text{length}[T]$ 
2  $m \leftarrow \text{length}[P]$ 
3 for  $s \leftarrow 0$  to  $n - m$ 
4 do if  $P[1..m] = T[s+1..s+m]$ 
5 then print “Pattern occurs with shift”  $s$ 
```

The naive string-matching procedure can be interpreted graphically as sliding a “template” containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s .

Procedure NAIVE-STRING-MATCHER takes time $O((n-m+1)m)$, and this bound is tight in the worst case. For example, consider the text string an (a string of n a’s) and the pattern am . For each of the $n-m+1$ possible values of the shift s , the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus $\Theta((n-m+1)m)$, which is $\Theta(n^2)$ if $m = \Theta(n)$. The running time of NAIVE-STRING-MATCHER is equal to its matching time, since there is no preprocessing.

As we shall see, NAIVE-STRING-MATCHER is not an optimal procedure for this problem. Indeed, in this chapter we shall show an algorithm with a worst-case preprocessing time of $\Theta(m)$ and a worst-case matching time of $\Theta(n)$. The naïve string-matcher is inefficient because information gained about the text for one value of s is entirely ignored in considering other values of s . Such information can be very valuable, however. For example, if $P = aaab$ and we find that $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $T[4] = b$.

The Rabin-Karp algorithm

Rabin and Karp have proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$. Based on certain assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number.

For expository purposes, let us assume that $\Sigma = \{0, 1, 2, \dots, 9\}$, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix- d

notation, where $d = |_|.)$ We can then view a string of k consecutive characters as representing a length- k decimal number. The character string 31415 thus corresponds to the decimal number 31,415. Given the dual interpretation of the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

Given a pattern $P[1 \dots m]$, we let p denote its corresponding decimal value. In a similar manner, given a text $T[1 \dots n]$, we let ts denote the decimal value of the length- m substring $T[s + 1 \dots s + m]$, for $s = 0, 1, \dots, n - m$. Certainly, $ts = p$ if and only if $T[s + 1 \dots s + m] = P[1 \dots m]$; thus, s is a valid shift if and only if $ts = p$. If we could compute p in time $_ (m)$ and all the ts values in a total of $_ (n - m + 1)$ time, then we could determine all valid shifts s in time $_ (m) + _ (n - m + 1) = _ (n)$ by comparing p with each of the ts 's. (For the moment, let's not worry about the possibility that p and the ts 's might be very large numbers.)

RABIN-KARP-MATCHER(T, P, d, q)

```

1  $n \leftarrow \text{length}[T]$ 
2  $m \leftarrow \text{length}[P]$ 
3  $h \leftarrow dm - 1 \bmod q$ 
4  $p \leftarrow 0$ 
5  $t0 \leftarrow 0$ 
6 for  $i \leftarrow 1$  to  $m$   $\bowtie$ Preprocessing.
7 do  $p \leftarrow (dp + P[i]) \bmod q$ 
8  $t0 \leftarrow (dt0 + T[i]) \bmod q$ 
9 for  $s \leftarrow 0$  to  $n - m$   $\bowtie$ Matching.
10 do if  $p = ts$ 
11 then if  $P[1 \dots m] = T[s + 1 \dots s + m]$ 
12 then print "Pattern occurs with shift"  $s$ 
13 if  $s < n - m$ 
14 then  $ts+1 \leftarrow (d(ts - T[s + 1])h) + T[s + m + 1]) \bmod q$ 
```

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix- d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high-order digit position of an m -digit window. Lines 4–8 compute p as the value of $P[1 \dots m] \bmod q$ and $t0$ as the value of $T[1 \dots m] \bmod q$. The **for** loop of lines 9–14 iterates through all possible shifts s , maintaining the following invariant:

Whenever line 10 is executed, $ts = T[s + 1 \dots s + m] \bmod q$. If $p = ts$ in line 10 (a "hit"), then we check to see if $P[1 \dots m] = T[s + 1 \dots s + m]$ in line 11 to rule out the possibility of a spurious hit.

Any valid shifts found are printed out on line 12. If $s < n - m$ (checked in line 13), then the **for** loop is to be executed at least one more time, and so line 14 is first executed to ensure that the loop invariant holds when line 10 is again reached. Line 14 computes the value of $ts+1 \bmod q$ from the value of $ts \bmod q$ in constant time directly.

RABIN-KARP-MATCHER takes $_ (m)$ preprocessing time, and its matching time is $_ ((n - m + 1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift.

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 $(Q, q_0, A, _, \delta)$, where

- Q is a finite set of **states**,
- q_0 is the **start state**,
- A is a distinguished set of **accepting states**,
- $_$ is a finite **input alphabet**,
- δ is a function from $Q \times _$ 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 $\delta(q, a)$. 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)

```
1  $n \leftarrow \text{length}[T]$ 
2  $q \leftarrow 0$ 
3 for  $i \leftarrow 1$  to  $n$ 
4 do  $q \leftarrow \delta(q, T[i])$ 
5 if  $q = m$ 
```

6 **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

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. Their algorithm avoids the computation of the transition function δ altogether, and its matching time is $_ (n)$ using just an auxiliary function $\pi[1 \dots m]$ precomputed from the pattern in time $_ (m)$. The array π allows the transition function δ to be computed efficiently (in an amortized sense) “on the fly” as needed.

Roughly speaking, for any state $q = 0, 1, \dots, m$ and any character a , the value $\pi[q]$ contains the information that is independent of a and is needed to compute $\delta(q, a)$. (This remark will be clarified shortly.) Since the array π has only m entries, whereas δ has $_ (m _)$ entries, we save a factor of $_$ in the preprocessing time by computing π rather than δ .

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid testing useless shifts in the naive pattern-matching algorithm or to avoid the precomputation of δ for a string-matching automaton.

Consider the operation of the naive string matcher. For this example, $q = 5$ of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing these q text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift $s + 1$ is necessarily invalid, since the first pattern character (a) would be aligned with a text character that is known to match with the second pattern character (b). The shift $s_ = s + 2$ shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match.

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER. It is mostly modeled after FINITE- AUTOMATONMATCHER, as we shall see. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π .

KMP-MATCHER(T, P)

1 $n \leftarrow \text{length}[T]$

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS : II B.SC CS

COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE: 16CSU401

UNIT V: STRING PROCESSING

BATCH: 2016-2019

```
2  $m \leftarrow \text{length}[P]$ 
3  $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4  $q \leftarrow 0$  //Number of characters matched.
5 for  $i \leftarrow 1$  to  $n$  //Scan the text from left to right.
6 do while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7 do  $q \leftarrow \pi[q]$  //Next character does not match.
8 if  $P[q + 1] = T[i]$ 
9 then  $q \leftarrow q + 1$  //Next character matches.
10 if  $q = m$  //Is all of  $P$  matched?
11 then print "Pattern occurs with shift"  $i - m$ 
12  $q \leftarrow \pi[q]$  //Look for the next match.
COMPUTE-PREFIX-FUNCTION( $P$ )
1  $m \leftarrow \text{length}[P]$ 
2  $\pi[1] \leftarrow 0$ 
3  $k \leftarrow 0$ 
4 for  $q \leftarrow 2$  to  $m$ 
5 do while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
6 do  $k \leftarrow \pi[k]$ 
7 if  $P[k + 1] = P[q]$ 
8 then  $k \leftarrow k + 1$ 
9  $\pi[q] \leftarrow k$ 
10 return  $\pi$ 
```

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 2016-2019)
Design and Analysis of Algorithms

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARK QUESTIONS

S.NO	QUESTIONS	OPTION 1	OPTION 2	OPTION 3	OPTION 4
1	Writing a Pseudocode has _____ of styles.	Restriction	Protocols	Condition	No restriction
2	Analysis of algorithms is the determination of _____.	Time and Space	Time	Space	Distance
3	The algorithm's state is the set of values corresponding to all _____.	Instances	Factors	Variables	Objects
4	_____ involves division of problems into sub problems.	Combine	Merge	Conquer	Break
5	Heap data structure is always a _____.	Radix	complete binary tree	Heap sort	Both a & b
6	Bucket sort is a sorting algorithm that works by _____ an array.	Partitioning	Adding	Updating	Combining
7	_____ value, which is called the _____.	Constant	Variable	Pivot value	None of these
8	The speed of Radix Sort largely depends on the _____ basic operations.	Outer	Input	Output	Inner
9	A _____ is an operation that preserves in-order traversal key ordering.	Alteration	Reversal	Resolution	Rotation
10	The _____ of the tree is the maximum depth of any node in the tree.	Row	Column	Height	Width
11	The number of children emanating from a given node is referred to as its _____.	Depth	Degree	Height	Width
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
13	Edges are the links that connect the vertices.	Nodes	Edges	Trees	Both a & c
14	In a directed graph, edges have _____.	Scalar	Vector	Direction	Time
15	The spanning tree does not have _____.	Loops	Graph	Aggregation	Recursion

16	In an _____ graph, edges have no direction.	Undirected	Tree	Recurrence	Modules
17	The running time of naïve string matcher is _____ to its matching time.	Not-equal	Equal	Minimum	Maximum
18	_____ algorithm makes use of elementary number-theoretic notions.	Rabin-Karp	Naive	KMP	Both a & c
19	To specify the string-matching automaton, we first define a _____ function.	Suffix	Prefix	Theoretical	String
20	_____ algorithm avoids the computation of the transition function.	KMP	Naive	Rabin	Karp
21	The _____ node are not a part of original tree and are represented as square nodes.	internal node	external node	intermediate node	terminal node
22	The external nodes are in a binary search tree are also known as _____ nodes	internal	search	failure	round
23	A binary tree with external nodes added is an ----- binary tree	extended	expanded	internal	external
24	_____ is a set of name-value pairs.	Symbol table	Graph	Node	Record
25	Each name in the symbol table is associated with an _____	name value pairs	element	attribute	entries
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
27	If the identifiers are known in advance and no deletion/insertions are allowed then this symbol table is _____	static	empty	dynamic	automatic
28	The cost of decoding a code word is ----- to the number of bits in the code	equal	not equal	proportional	inversely proportional
29	The solution of finding a binary tree with minimum weighted external path length has been given by	Huffman	Kruskal	Euler	Hamilton
30	_____ symbol table allows insertion and deletion of names.	Hashed	Sorted	Static	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
32	If hl and hr are the heights of the left and right subtrees of a tree respectively and if $ hl-hr \leq 1$ then this tree is called _____	extended binary tree	binary search tree	skewed 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
34	For an AVL Tree the balance factor is = _____	0	-1	1	Any of the above
35	If the names are _____ in the symbol table, searching is easy.	sorted	short	bold	upper case
36	_____ allocation is not desirable for dynamic tables, where insertions and deletions are allowed.	Linear	Sequential	Dynamic	None
37	A search in a hash table with n identifiers may take ----- time	O(n)	O(1)	O(2)	O(2n)
38	_____ data structure is used to implement symbol tables	directed graphs	binary search trees	circular queue	None
39	Every binary search tree with n nodes has _____ square node (external nodes).	n/2	n+1	n-1	2^n
40	In a Hash table the address of the identifier x is obtained by applying _____	sequence of comparisons	binary searching	arithmetic function	collision
41	The partitions of the hash table are called _____	Nodes	Buckets	Roots	Fields
42	The arithmetic functions used for Hashing is called _____	Logical operations	Rehashing	Mapping function	Hashing function
43	Each bucket of Hash table is said to have several _____	slots	nodes	fields	links
44	A _____ occurs when two non-identical identifiers are hashed in the same bucket.	collision	contraction	expansion	Extraction
45	A hashing function f transforms an identifier x into a _____ in the hash table	symbol name	bucket address	link field	slot number

46	When a new identifier I is mapped or hashed by the function f into a full bucket then _____ occurs	underflow	overflow	collision	rehashing
47	If f(I) and F(J) are equal then Identifiers I and J are called _____	synonyms	antonyms	hash functions	buckets
48	A ---tree is a binary tree in which external nodes represent messages	decode	unicode	extended	none
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
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
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
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
53	Each chain in the hash tables will have a _____	tail node	link node	head node	null node
54	Folding of identifiers from end to end to obtain a hashing function is called _____	Shift folding	boundary folding	expanded folding	end to end folding
55	Average number of probes needed for searching can be obtained by ----- probing	quadratic	linear	rehashing	Sequential
56	Rehashing is _____	series of hash function	linear probing	quadratic functions	Rebuild function
57	_____ is a method of overflows handling.	linear open addressing	Adjacency list	sequential representation	Indexed address
58	The number of _____ over the data can be reduced by using a higher order merge (k-way merge with k>2)	records	passes	tapes	merges

59	A _____ is a binary tree where each node represents the smaller of its two children	search tree	decision tree	extended tree	selection tree
60	In External sorting data are stored in _____	RAM memory	Cache memory	secondary storage devices	Buffers

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is an Algorithm?
2. Write an algorithm to compute the greatest common divisor of two numbers.
3. Devise an algorithm to make for 1655 using the Greedy strategy. The coins available are {1000, 500, 100, 50, 20, 10, 5}.
4. What is closest-pair problem?
5. State the general principle of greedy algorithm.
6. What do you mean by dynamic programming?
7. What do you mean by 'perfect matching' in bipartite graphs?
8. State: Planar coloring graph problem.
9. What is an articulation point in a graph?
10. Define 'P' and 'NP' problems.

11. (a) Briefly explain the mathematical analysis of recursive and non-recursive algorithm. (13)

Or

- (b) Explain briefly Big oh Notation, Omega Notation and Theta Notations. Give examples. (13)

12. (a) What is divide and conquer strategy and explain the binary search with suitable example problem. (13)

Or

- (b) Solve the following using Brute-Force algorithm: (13)

Find whether the given string follows the specified pattern and return 0 or 1 accordingly.

Examples :

- (i) Pattern: "abba", input: "redblueredblue" should return 1
 (ii) Pattern: "aaaa", input: "asdasdasdasd" should return 1
 (iii) Pattern: "aabb" input: "xyzabcxzyabc" should return 0.
13. (a) Solve the following instance of the 0/1, knapsack problem given the knapsack capacity in $W = 5$ using dynamic programming and explain it. (13)

Items	Weight	Value
1	4	10
2	3	20
3	2	15
4	5	25

Or

- (b) Write the Huffman's Algorithm. Construct the Huffman's tree for the following data and obtain its Huffman's Code. (13)

Character	A	B	C	D	E	-
Probability	0.5	0.35	0.5	0.1	0.4	0.2

14. (a) Describe in detail the simplex algorithm methods. (13)

Or

- (b) Explain KMP string matching algorithm for finding a pattern on a text, and analyze the algorithm. (13)

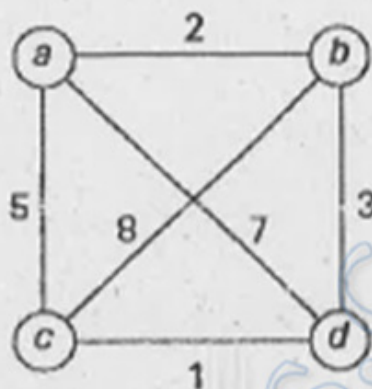
15. (a) Discuss the approximation algorithm for NP- hard problems. (13)

Or

- (b) Describe the backtracking solution to solve 8-queens problem. (13)

PART C — (1 × 15 = 15 marks)

16. (a) Apply Branch and Bound algorithm to solve the Travelling Salesman Problem for (15)



Or

- (b) Write an algorithm for quick sort and write its time complexity with example list are 5, 3, 1, 9, 8, 2, 4, 7. (15)

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER SCIENCE

ANSWER KEY INTERNAL-1

SUBJECT: DESIGN AND ANALYSIS OF ALGORITHMS

SUB.CODE : 16CSU401

2 marks

21. Algorithm

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.

22. 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 subproblems, solving each of those subproblems just once, and storing their solutions.

23. 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.

8 Marks

24. a. Basic Design and Analysis techniques of Algorithm

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

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.

24.b. 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.

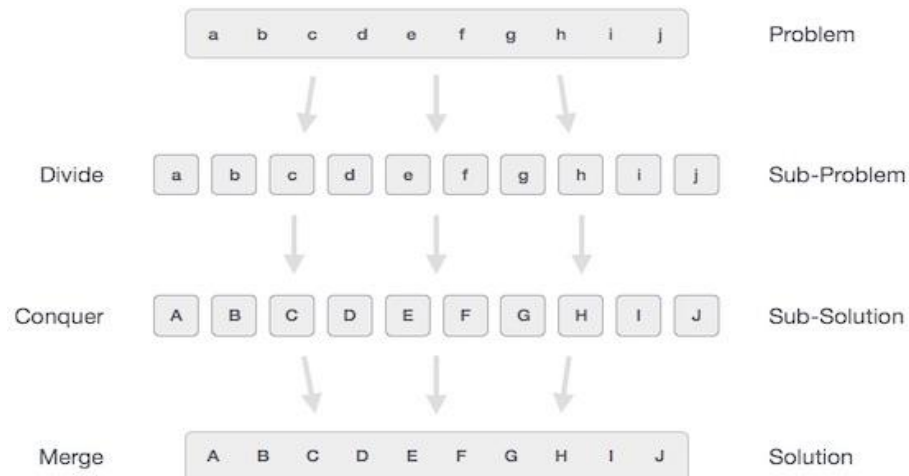
When the processing structure is a sequential one (for example a sequence of assignments) then the verification process is a simple one (we must only analyze the effect of each assignment on the algorithm's state).

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 work so close that they appear as one.

For example,

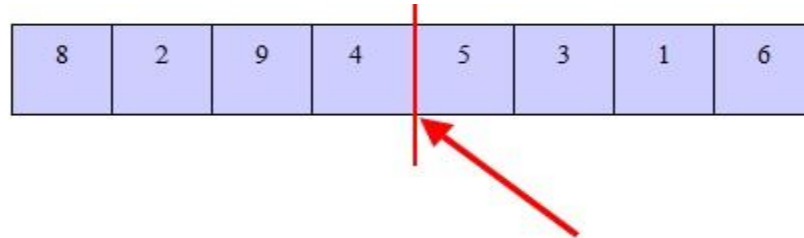
The following computer algorithms are based on divide-and-conquer programming approach –

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

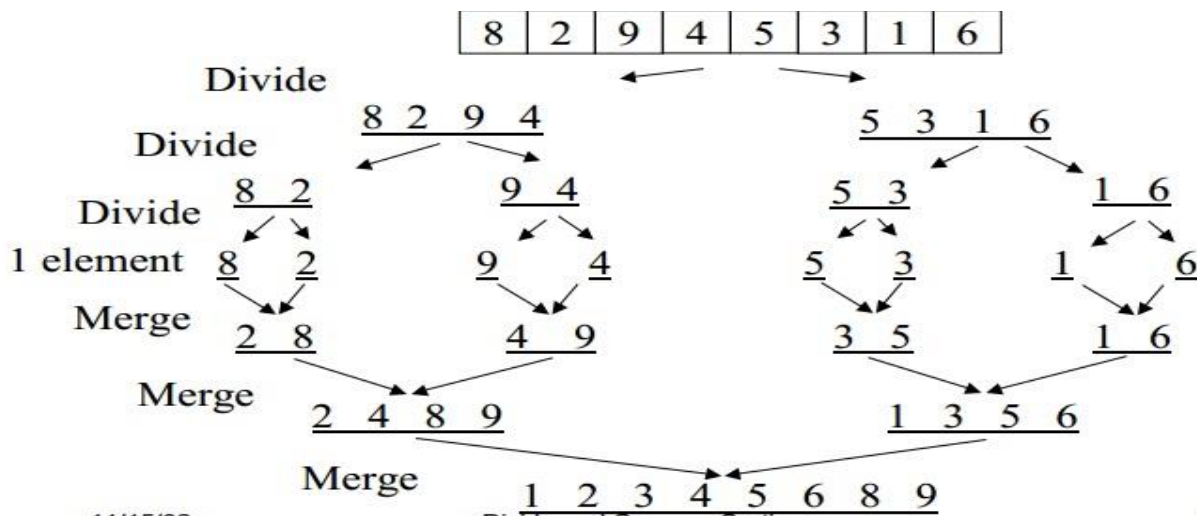
Example for Divide and Conquer

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

Merge Sort Example



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



25. a. 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 subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem 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];

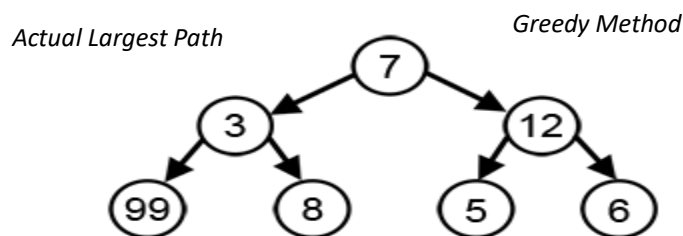
}
```

25. b. Greedy Technique

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.

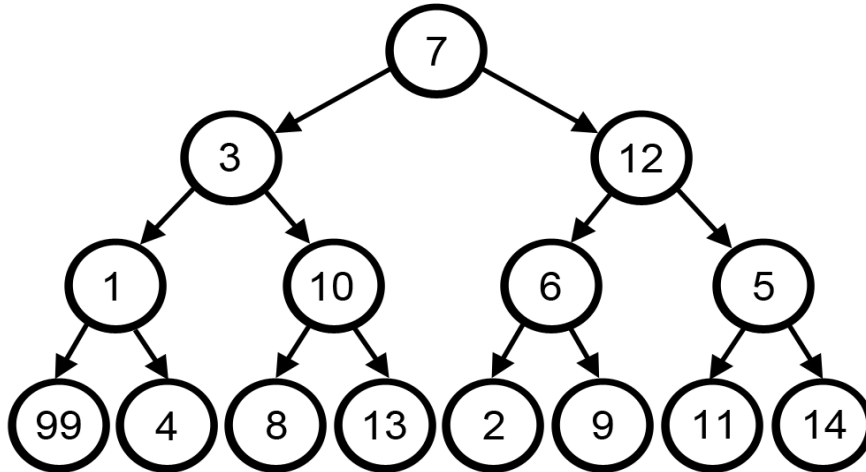


Limitations of Greedy Algorithms

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 playstation 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.

26. a. 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.



Insertion sort compares the first two elements.



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



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



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



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



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



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



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

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:

```

C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
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

```

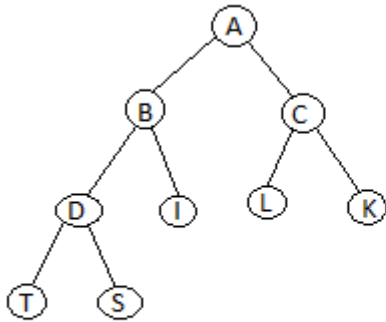
26. b. Heap Sort

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 :

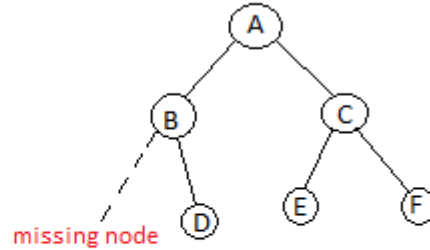
- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

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

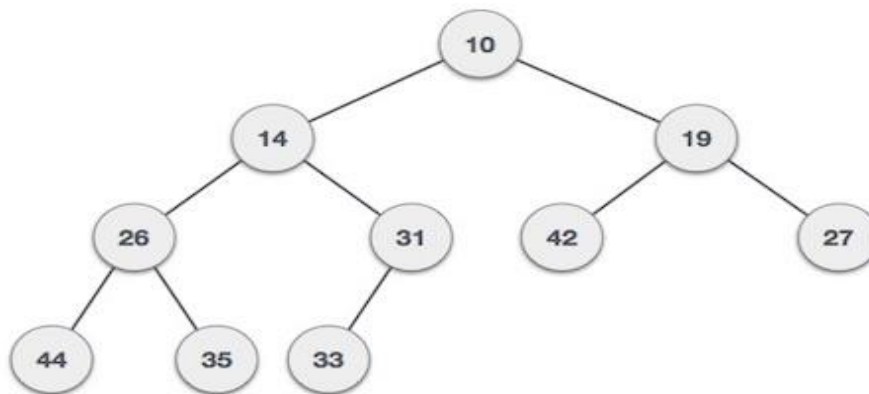


In-Complete Binary Tree

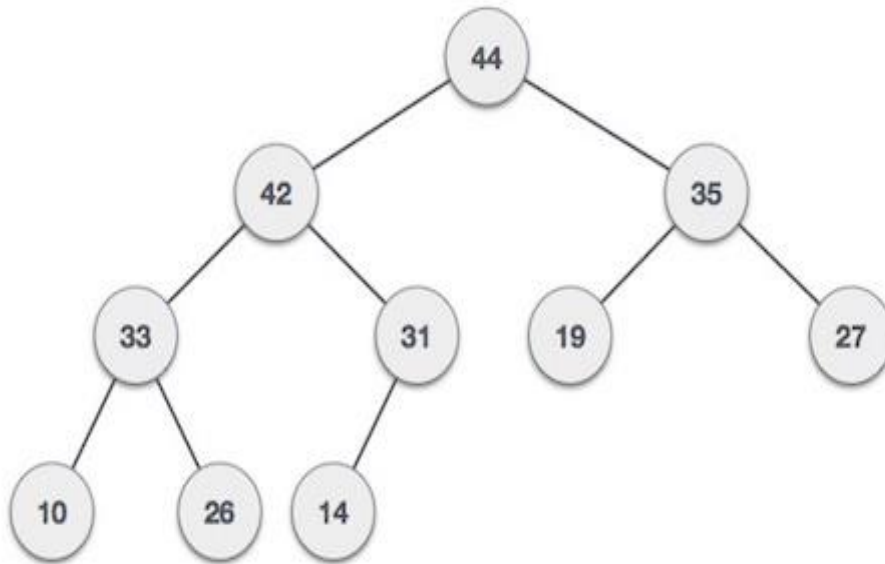
2. **Heap Property** : All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

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.

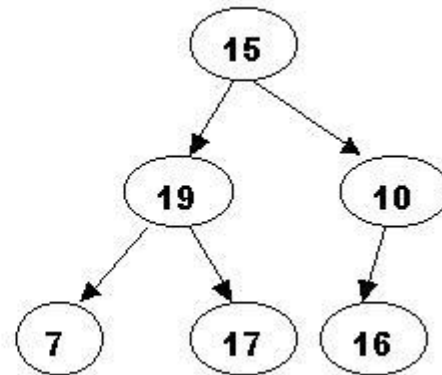
This will be indicated by colors: white for the original array, blue for the heap and red for the sorted array

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

A. Building the heap tree

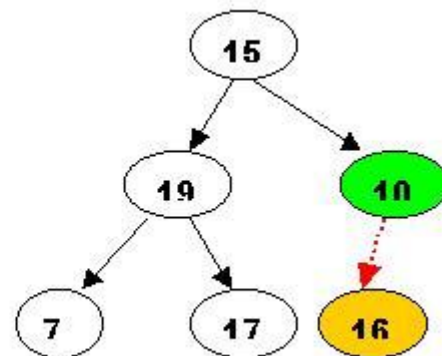
The array represented as a tree, complete but not ordered:

15	19	10	7	17	16
----	----	----	---	----	----



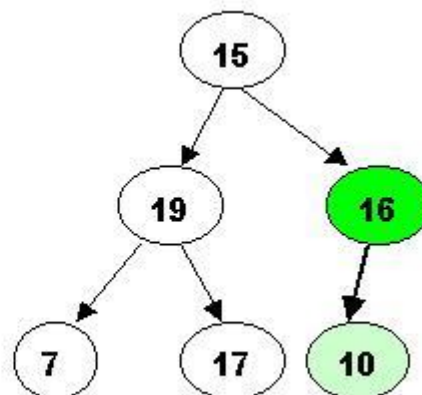
Start with the rightmost node at height 1 - the node at position 3 = Size/2.
It has one greater child and has to be percolated down:

15	19	10	7	17	16
----	----	----	---	----	----



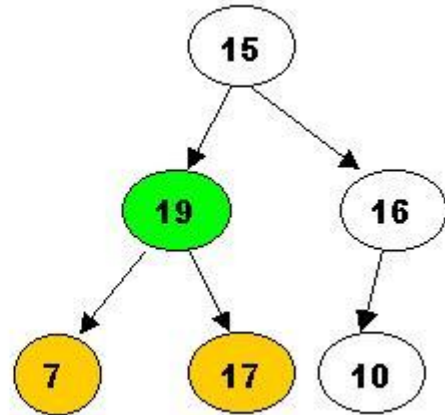
After processing array[3] the situation is:

15	19	16	7	17	10
----	----	----	---	----	----



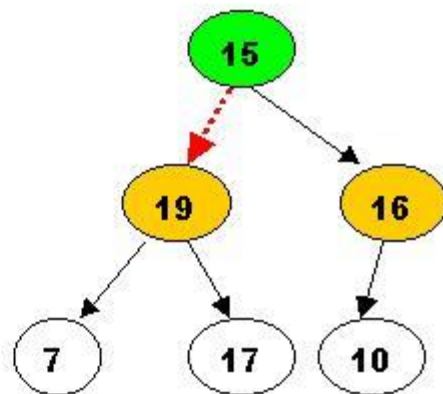
Next comes array[2]. Its children are smaller, so no percolation is needed.

15	19	16	7	17	10
----	----	----	---	----	----



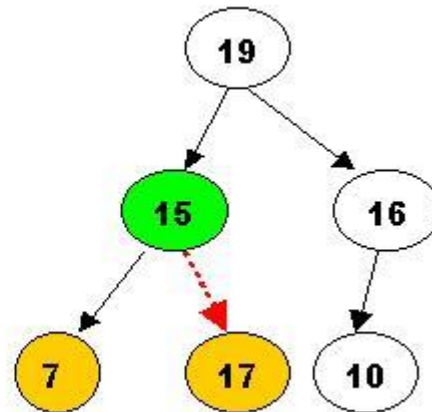
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array[1] has to be percolated down to the left, swapped with array[2].

15	19	16	7	17	10
----	----	----	---	----	----



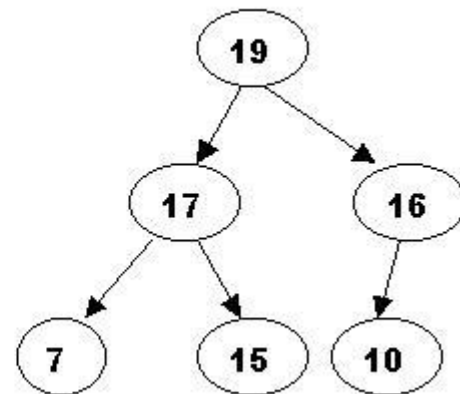
As a result the situation is:

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.

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