Course Code	Advanced Data Structures & Algorithms (Common to CSE, IT, CSE(DS), CSE (IoT), CSE (AI), CSE (AI & ML) and AI & DS)		L	T	P	C
20A05301T			3	0	0	3
Pre-requisite	**Data Structures**	**Semester**	**III**			

Course Objectives:

- Learn asymptotic notations, and analyze the performance of different algorithms.
- Understand and implement various data structures.
- Learn and implement greedy, divide and conquer, dynamic programming and backtracking algorithms using relevant data structures.
- Understand non-deterministic algorithms, polynomial and non-polynomial problems.

Course Outcomes (CO):

After completion of the course, students will be able to

- Analyze the complexity of algorithms and apply asymptotic notations.
- Apply non-linear data structures and their operations.
- Understand and apply greedy, divide and conquer algorithms.
- Develop dynamic programming algorithms for various real-time applications.
- Illustrate Backtracking algorithms for various applications.

UNIT - I	**Introduction to Algorithms**	9 Hrs

Introduction to Algorithms:
Algorithms, Pseudocode for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation, Polynomial Vs Exponential Algorithms, Average, Best and Worst Case Complexities, Analysing Recursive Programs.

UNIT - II	**Trees Part-I**	8 Hrs

Trees Part-I
Binary Search Trees: Definition and Operations, AVL Trees: Definition and Operations, Applications.
B Trees: Definition and Operations.

UNIT - III	**Trees Part-II**	8 Hrs

Trees Part-II
Red-Black Trees, Splay Trees, Applications.
Hash Tables: Introduction, Hash Structure, Hash functions, Linear Open Addressing, Chaining and Applications.

UNIT - IV	**Divide and conquer, Greedy method**	9 Hrs

Divide and conquer: General method, applications-Binary search, Finding Maximum and minimum, Quick sort, Merge sort, Strassen's matrix multiplication.
Greedy method: General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

UNIT - V	**Dynamic Programming & Backtracking**	9 Hrs

Dynamic Programming: General method, applications- 0/1 knapsack problem, All pairs shortest path problem, Travelling salesperson problem, Reliability design.
Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.
Introduction to NP-Hard and NP-Complete problems: Basic Concepts.

Textbooks:

1. Data Structures and algorithms: Concepts, Techniques and Applications, G A V Pai.
2. Fundamentals of Computer Algorithms, Ellis Horowitz, Sartaj Sahni and Rajasekharam, Galgotia publications Pvt. Ltd.

Reference Books:

1. Classic Data Structures by D. Samanta, 2005, PHI
2. Design and Analysis of Computer Algorithms by Aho, Hopcraft, Ullman 1998, PEA.
3. Introduction to the Design and Analysis of Algorithms by Goodman, Hedetniemi, TMG.

Online Learning Resources:

https://www.tutorialspoint.com/advanced_data_structures/index.asp
http://peterindia.net/Algorithms.html

UNIT –I

Introduction to Algorithms:

Algorithms, Pseudocode for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh, Omega, Theta notation and Little oh notation, Polynomial Vs Exponential Algorithms, Average, Best and Worst Case Complexities, Analysing Recursive Programs.

INTRODUCTION TO ALGORITHMS: WHAT IS AN ALGORITHM? Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

- **INPUT** → Zero or more quantities are externally supplied.
- **OUTPUT** → At least one quantity is produced.
- **DEFINITENESS** → Each instruction is clear and unambiguous.
- **FINITENESS** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **EFFECTIVENESS** → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

1. How to device or design an algorithm → creating and algorithm.
2. How to express an algorithm → definiteness.
3. How to analysis an algorithm → time and space complexity.
4. How to validate an algorithm → fitness.
5. Testing the algorithm → checking for error.

Algorithm Specification:

Algorithm can be described in three ways.

1. **Natural language like English:**

When this way is choused care should be taken, we should ensure that each & every statement is definite.

2. **Graphic representation called flowchart:**

   This method will work well when the algorithm is small& simple.

3. **Pseudo-code Method:**

   In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

PSEUDO-CODE FOR EXPRESSING AN ALGORITHM:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
  data type – 1  data-1;
        .
        .
        .
  data type – n data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

   <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

   → Logical Operators AND, OR, NOT

   →Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

   For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
              .
              .
              .
        <statement-n>          }
```

For Loop:

```
For variable: = value-1 to value-2 step step do
{
  <statement-1>
        .
        .
        .
<statement-n>
}
```

repeat-until:

```
        repeat
                <statement-1>
                        .
                        .
                        .
                <statement-n>
        until<condition>
```

8. A conditional statement has the following forms.

   → If <condition> then <statement>

   → If <condition> then <statement-1>

   Else <statement-1>

   Case statement:

```
Case
{
        : <condition-1> : <statement-1>
                        .
                        .
                        .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

9. Input and output are done using the instructions read & write.
10. There is only one type of procedure: Algorithm, the heading takes the form,

    Algorithm Name (Parameter lists)

→ As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

```
1. algorithm Max(A,n)
2. // A is an array of size n
```

```
{
 Result := A[1];
 for I:= 2 to n do
    if A[I] > Result then
          Result :=A[I];
   return Result;
}
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

→ **Next we present 2 examples to illustrate the process of translation problem into an algorithm.**

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution given by the following.
- (From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

```
For i:= 1 to n do
{
            Examine a[I] to a[n] and suppose the smallest element is at a[j];
            Interchange a[I] and a[j];
}
```

→ Finding the smallest element (sat a[j]) and interchanging it with a[i]

- We can solve the latter problem using the code,

```
t    := a[i];
a[i]:=a[j];
a[j]:=t;
```

- The first subtask can be solved by assuming the minimum is a[I];checking a[I] with a[I+1],a[I+2]…….,and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.
- Putting all these observations together, we get the algorithm Selection sort.

Theorem: Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

Selection Sort:

Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position. This procedure is repeated till the entire list has been studied.

Example: List L = 3,5,4,1,2

1 is selected , → 1,5,4,3,2

2 is selected, →1,2,4,3,5

3 is selected, →1,2,3,4,5

4 is selected, →1,2,3,4,5

Proof:

- We first note that any I, say I=q, following the execution of lines 6 to 9,it is the case that a[q] Þ a[r],q<r<=n.
- Also observe that when 'i' becomes greater than q, a[1:q] is unchanged. Hence, following the last execution of these lines (i.e. I=n).We have a[1] <= a[2] <=……a[n].
- We observe this point that the upper limit of the for loop in the line 4 can be changed to n-1 without damaging the correctness of the algorithm.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3.{
4.      for I:=1 to n do
5.      {
6.              j:=I;
7.              for k:=i+1 to n do
8.                      if (a[k]<a[j])
9.                      t:=a[I];
10.                     a[I]:=a[j];
11.                     a[j]:=t;
12.     }
13. }
```

PERFORMANCE ANALYSIS:

1. **Space Complexity:**

   The space complexity of an algorithm is the amount of money it needs to run to compilation.

2. **Time Complexity:**

   The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity:

Space Complexity Example:

```
Algorithm abc(a,b,c)
{
return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

→ The Space needed by each of these algorithms is seen to be the sum of the following component.

1.**A fixed part** that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

a. The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics) Where 'c' is a constant.

Example : Algorithm sum(a,n)

```
{
    s=0.0;
    for I=1 to n do
    s= s+a[I];
```

```
        return s;
    }
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
- The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
- This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
- So,we obtain Ssum(n)>=(n+s) [n for a[ ],one each for n,I a& s]

Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

→The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

→ The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments → 0 steps.

Assignment statements → 1 steps. [Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until→ Control part of the statement.

1. **We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.**

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm:

```
Algorithm sum(a,n)
    {
      s= 0.0;
      count = count+1;
      for I=1 to n do
```

```
{
 count =count+1;
s=s+a[I];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}
```

→ If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

→First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

→By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	**S/e**	**Frequency**	**Total**
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

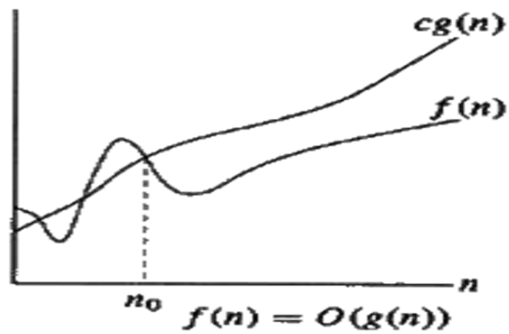
ASYMPTOTIC NOTATIONS

There are different kinds of mathematical notations used to represent time complexity. These are called Asymptotic notations. They are as follows:

1. Big oh(O) notation
2. Omega(Ω) notation
3. Theta(ɵ) notation

1. Big oh(O) notation:

- Big oh(O) notation is used to represent upperbound of algorithm runtime.
- Let f(n) and g(n) are two non-negative functions
- The function f(n) = O(g(n)) if and only if there exists positive constants c and n_0 such that $f(n) \le c*g(n)$ for all n , $n \ge n_0$.



Example:

If f(n)=3n+2 then prove that f(n) = O(n)

Let f(n) =3n+2, c=4, g(n) =n

if n=1 $3n+2 \le 4n$

$3(1)+2 \le 4(1)$

$3+2 \le 4$

$5 \le 4$ (F)

if n=2 $3n+2 \le 4n$

$3(2)+2 \le 4(2)$

$8 \le 8$ (T)

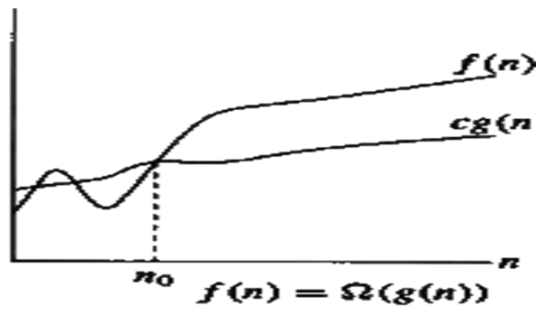
$3n+2 \le 4n$ for all $n \ge 2$

This is in the form of $f(n) \le c*g(n)$ for all $n \ge n_0$, where c=4, n_0 =2

Therefore, f(n) = O(n),

2. Omega(Ω) notation:

- Big oh(O) notation is used to represent lowerbound of algorithm runtime.
- Let f(n) and g(n) are two non-negative functions
- The function f(n) = Ω(g(n)) if and only if there exists positive constants c and n_0 such that $f(n) \ge c*g(n)$ for all n , $n \ge n_0$.



Example

f(n)=3n+2 then prove that f(n) = Ω(g(n))

Let f(n) =3n+2, c=3, g(n) =n

if n=1 $3n+2 \geq 3n$

$3(1)+2 \geq 3(1)$

$5 \geq 3$ (T)

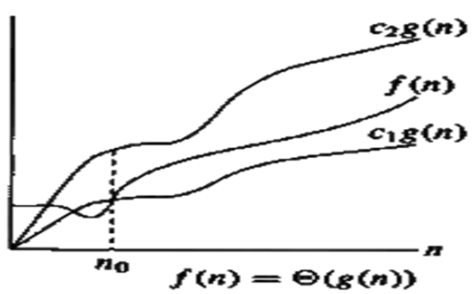
$3n+2 \geq 4n$ for all $n \geq 1$

This is in the form of $f(n) \geq c*g(n)$ for all $n \geq n_0$, where c=3, $n_0 = 1$

Therefore, f(n) = Ω(n).

3. Theta(ɵ) notation:

- Theta(ɵ) notation is used to represent the running time between upper bound and lower bound.
- Let f(n) and g(n) be two non-negative functions.
- The function f(n) = θ(g(n)) if and only if there exists positive constants c_1 , c_2 and n_0 such that $c_1*g(n) \leq f(n) \leq c_2* g(n)$ for all n, $n \geq n_0$.



Example:

f(n)=3n+2 then Prove that f(n) = θ(g(n))

Lower bound = $3n+2 \geq 3n$ for all $n \geq 1$

$c_1=3, g(n)=n, n_0=1$

Upper Bound = $3n+2 \leq 4n$ for all $n \geq 2$

$c2=4$, $g(n)=n$, $n0=2$

$3(n) \leq 3n+2 \leq 4(n)$ for all n, $n \geq 2$

This is in the form of $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$ Where $c_1=3$, $c_2=4$, $g(n)=n$, $n_0=2$

Therefore $f(n)= \theta(n)$

POLYNOMIAL VS EXPONENTIAL ALGORITHMS

The **time complexity**(generally referred as running time) of an algorithm is expressed as **the amount of time taken by an algorithm for some size of the input to the problem**. **Big O notation** is commonly used to express the time complexity of any algorithm as this suppresses the lower order terms and is described asymptotically. Time complexity is estimated by counting the operations(provided as instructions in a program) performed in an algorithm. Here each operation takes a fixed amount of time in execution. **Generally time complexities are classified as constant, linear, logarithmic, polynomial, exponential etc**. Among these the **polynomial and exponential are the most prominently considered** and defines the complexity of an algorithm. These two parameters for any algorithm are always influenced by size of input.

Polynomial Running Time

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the input. Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

All basic arithmetic operations ((i.e.) Addition, subtraction, multiplication,division), comparison operations, sort operations are considered as polynomial time algorithms.

Exponential Running Time

The set of problems which can be solved by an exponential time algorithms, but for which no polynomial time algorithms is known.

An algorithm is said to be exponential time, if $T(n)$ is upper bounded by $2^{\text{poly}(n)}$, where poly(n) is some polynomial in n. More formally, an algorithm is exponential time if $T(n)$ is bounded by $O(2^{nk})$ for some constant k.

Algorithms which have exponential time complexity grow much faster than polynomial algorithms.

The difference you are probably looking for happens to be where the variable is in the equation that expresses the run time. Equations that show a **polynomial time** complexity have variables in the bases of their terms.

Examples: $n^3 + 2n^2 + 1$. Notice n is in the base, NOT the exponent.

In exponential equations, the variable is in the exponent.

Examples: 2^n. As said before, exponential time grows much faster. If n is equal to 1000 (a reasonable input for an algorithm), then notice 1000^3 is 1 billion, and 2^{1000} is simply huge! For a reference, there are about 2^{80} hydrogen atoms in the sun, this is much more than 1 billion.

AVERAGE, BEST AND WORST CASE COMPLEXITIES

Best case: This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Worst case: This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Average case: This type of analysis results in average running time over every type of input.

Complexity: Complexity refers to the rate at which the storage time grows as a function of the problem size.

ANALYSING RECURSIVE PROGRAMS.

For every recursive algorithm, we can write recurrence relation to analyse the time complexity of the algorithm.

Recurrence relation of recursive algorithms

A recurrence relation is an equation that defines a sequence where any term is defined in terms of its previous terms.

The recurrence relation for the time complexity of some problems are given below:

Fibonacci Number

$T(N) = T(N-1) + T(N-2)$

Base Conditions: $T(0) = 0$ and $\mathbf{T}(1) = 1$

Binary Search

$T(N) = T(N/2) + C$

Base Condition: $T(1) = 1$

Merge Sort

$T(N) = 2\,T(N/2) + CN$

Base Condition: $T(1) = 1$

Recursive Algorithm: Finding min and max in an array

$T(N) = 2\,T(N/2) + 2$

Base Condition: $T(1) = 0$ and $\mathbf{T}(2) = 1$

Quick Sort

$T(N) = T(i) + T(N-i-1) + CN$

The time taken by quick sort depends upon the distribution of the input array and partition strategy. T(i) and T(N-i-1) are two smaller subproblems after the partition where i is the number of elements that are smaller than the pivot. **CN** is the time complexity of the partition process where **C** is a constant. .

Worst Case: This is a case of the unbalanced partition where the partition process always picks the greatest or smallest element as a pivot(Think!).For the recurrence relation of the worst case scenario, we can put i = 0 in the above equation.

$$T(N) = T(0) + T(N-1) + CN$$

which is equivalent to

$$\mathbf{T}(N) = T(N-1) + CN$$

Best Case: This is a case of the balanced partition where the partition process always picks the middle element as pivot. For the recurrence relation of the worst case scenario, put i = N/2 in the above equation.

$$T(N) = T(N/2) + T(N/2-1) + CN$$

which is equivalent to

$$\mathbf{T}(N) = 2T(N/2) + CN$$

Average Case: *For* average case analysis, we need to consider all possible permutation of input and time taken by each permutation.

$$T(N) = (\textbf{for } i = 0 \text{ to } N-1) \sum (T(i) + T(N-i-1)) / N$$

Note: This looks mathematically complex but we can find several other intuitive ways to analyse the average case of quick sort.

Analyzing the Efficiency of Recursive Algorithms

Step 1: Identify the number of sub-problems and a parameter (or parameters) indicating an input's size of each sub-problem (function call with smaller input size)

Recursive Problems	Input Size	Number of Subproblems	Input size of Subproblems
Finding nth Fibonacci	N	2	(N-1) and (N-2)
Binary Search	N	1	N/2
Merge Sort	N	2	N/2 each
Recursive: Finding min and max	N	2	N/2 each
Karastuba algorithm	N (Total number of digits in each Integer)	3	N/2 each
Quick Sort	N	2	(i) and (n-i-1)
Strassen's Matrix Multiplication	N (Size of each matrix)	7	N/2 each
Recursive: Longest Common Subsequence	(N, M) length of both the subsequences	3	(N-1, M-1), (N-1, M) and (N, M-1)

Step 2: Add the time complexities of the sub-problems and the total number of basic operations performed at that stage of recursion.

Step3: Set up a recurrence relation, with a correct base condition, for the number of times the basic operation is executed.

Recursive Problems	Time Complexity of the subproblems	Total Number of basic operations	Recurrence Relation
Finding nth Fibonacci	T(N-1) + T (N-2)	O(1) for one addition	T(N) = T(N-1) + T(N-2) + C
Binary Search	T(N/2)	O(1) for one comparison	T(N) = T(N/2) + C
Merge Sort	2 T(N/2)	O(N) for merging two sorted halves	T(N) = 2 T(N/2) + CN
Finding min and max (Recursive)	2 T(N/2)	O(1) for one comparison	T(N) = 2 T(N/2) + C
Karatsuba algorithm	3T(N/2)	O(N) (How? Think)	T(N) = 3 T(N/2) + CN
Quick Sort	T(i) + T(N - i - 1)	O(N) for partition process	T(N) = T(i) + T(N - i - 1) + CN
Strassen's Matrix Multiplication	7 T(N/2)	O(N^2) for addition and subtraction of two matrices	T(N) = 7 T(N/2) + CN^2
Recursive: Longest Common Subsequence	T(N-1,M-1), if (A[M-1] = B[N-1]) T(N-1,M)+T(N, M-1), otherwise	O(1) for one comparison	T(N,M) = T(N-1,M-1)+O(1), if (X[M-1] = Y[N-1]) = T(N-1,M)+T(N, M-1) +O(1), otherwise

Step4: Solve the recurrence or, at least, ascertain the order of growth of its solution. There are several ways to analyse the recurrence relation but we are discussing here two popular approaches of solving recurrences:

- **Method 1**: Recursion Tree Method
- **Method 2**: Master Theorem

Method 1: Recursion Tree Method

A recurrence tree is a tree where each node represents the cost of a certain recursive subproblem. We take the sum of each value of nodes to find the total complexity of the algorithm.

Steps for solving a recurrence relation

1. Draw a recursion tree based on the given recurrence relation.
2. Determine the number of levels, cost at each level and cost of the last level.
3. Add the cost of all levels and simplify the expression.

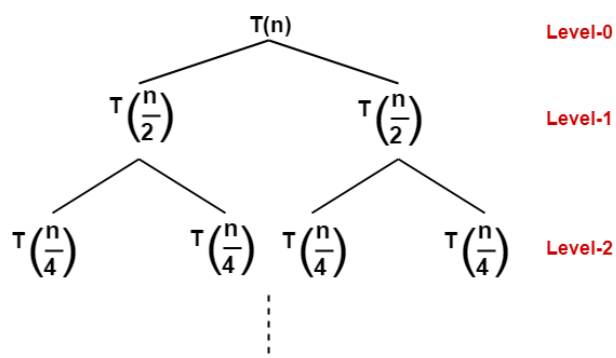
Let us solve the given recurrence relation by Recurrence Tree Method

$$T(N) = 2*T(N/2) + CN$$

From the above recurrence relation, we can find that

1. The problem of size N is divided into two sub-problems of size N/2.
2. The cost of dividing a sub-problem and then combining its solution of size N is CN.
3. Each time, the problem will be divided into half, until the size of the problem becomes 1.

The recursion tree for the above relation will be



Method 2: Master theorem

Master theorem states that for a recurrence relation of form

$T(N) = aT(N/b) + f(N)$ where $a >= 1$ and $b > 1$

If **f**$(N) = O(N^k)$ and $k \geq 0$, then

Case 1: T(N) = O(N^logb(a)), **if** k < logb(a).

Case 2: T(N) = O((N^k)*logN), **if** k = logb(a).

Case 3: T(N) = O(N^k), **if** k > logb(a)

Example 1

T(N) = T(N/2) + C

The above recurrence relation is of binary search. Comparing this with master theorem, we get a = 1, b = 2 and k = 0 because f(N) = C = C(N^0)

Here logb(a) = k, so we can apply case 2 of the master theorem.

T(n) = (N^0*log(N)) = O(logN).

Example 2

T(N) = 2*T(N/2) + CN

The above recurrence relation is of **merge sort**. Comparing this with master theorem,a = 2, b = 2 and f(N) = CN. Comparing left and right sides of f(N), we get k = 1.

logb(a) = log2(2) = 1 = K

So, we can apply the case 2 of the master theorem.

=> T(N) = O(N^1*log(N)) = O(NlogN).

PART-A (2 Marks)

1. What is performance measurement?

Ans. Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

2. What is an algorithm?

Ans. An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

3. **What are the characteristics of an algorithm?**

Ans. 1) Input

2) Output

3) Definiteness

4) Finiteness

5) Effectiveness

4. What is recursive algorithm?

Ans. An algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is direct recursive. Algorithm A is said to be indeed recursive if it calls another algorithm, which in turn calls A.

5. What is space complexity?

Ans. The space complexity of an algorithm is the amount of memory it needs to run to completion.

6. What is time complexity?

Ans. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

7. Define the asymptotic notation "Big Oh" (O) ,"Omega" (Ω) and "theta" (ɵ)

Ans. Big Oh(O) : The function f(n) = O(g(n)) iff there exist positive constants C and no such that f(n) ≤ C * g(n) for all n, n ≥ n0.

Omega (Ω) : The function f(n) =Ω(g(n)) iff there exist positive constant C and no such that f(n) ≥ C * g(n) for all n, n ≥ n0.

theta(ɵ) :The function f(n) = ɵ (g(n)) iff there exist positive constant C1, C2, and no such that C1 *g(n) ≤ f(n) ≤ C2* g(n) for all n, n ≥ n0.

PART-B (10 Marks)

1. Write the merge sort algorithm. Find out the best, worst and average cases of this algorithm. Sort the following numbers using merge sort:

   10, 12, 1, 5, 18, 28, 38, 39, 2, 4, 7

2. What is asymptotic notation? Explain different types of notations with example.

3. **Solve** the following recurrence relation $T(n) = 7T(n/2)+cn^2$

4. **Solve** the following recurrence relation $T(n) = \left\{2T\left(\frac{n}{2}\right) + 1, \quad and\ T(1) = 2\right.$

5. **Define** the term algorithm and state the criteria the algorithm should satisfy.

6. If $f(n)=5n^2 + 6n + 4$, then **prove** that f(n) is $O(n^2)$.

7. **Use** step count method and analyze the time complexity when two n×n matrices are added.

8. **Describe** the role of space complexity and time complexity of a program ?

9. **Discuss** various the asymptotic notations used for best case average case and worst case analysis of algorithms.

UNIT –II

Binary Search Trees: *Definition and Operations,* ***AVL Trees:*** *Definition and Operations, Applications.*
B Trees: *Definition and Operations.*

INTRODUCTION

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

A binary tree has the following time complexities...

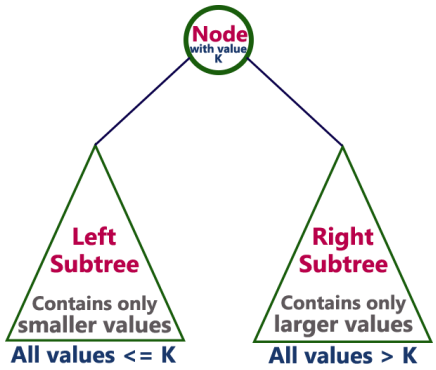
1. **Search Operation - O(n)**
2. **Insertion Operation - O(1)**
3. **Deletion Operation - O(n)**

BINARY SEARCH TREE

To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

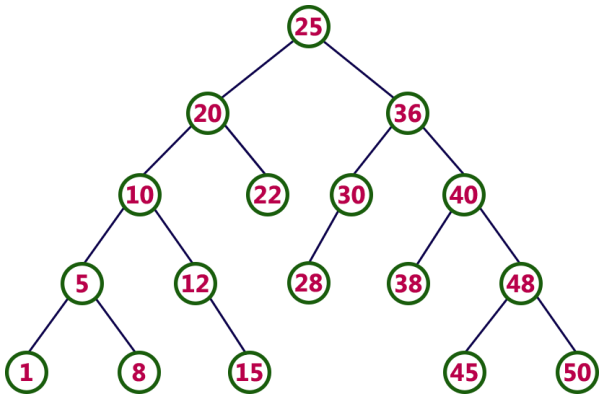
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...

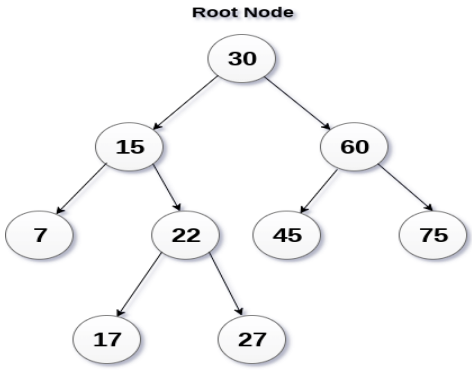


Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every binary search tree is a binary tree but every binary tree need not to be binary search tree.



Binary Search Tree

Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.

2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2$n) time. In worst case, the time it takes to search an element is 0(n).

3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

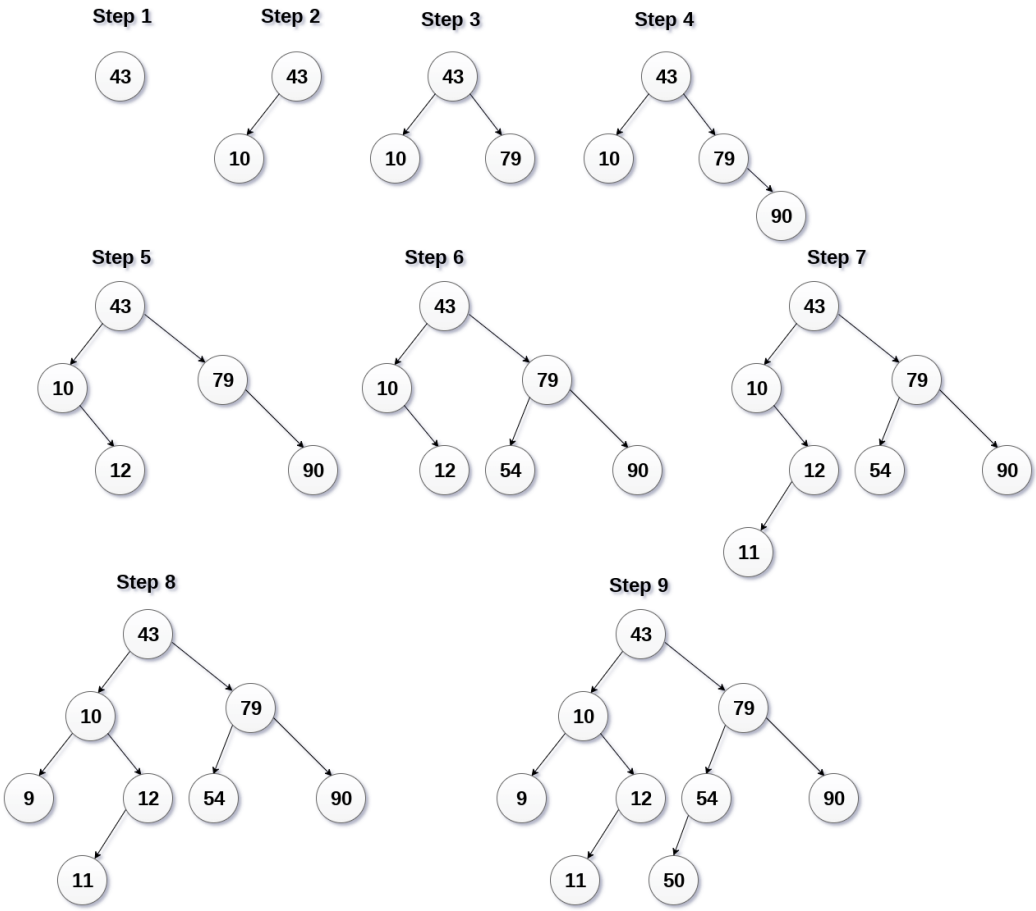
Example1:

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

   The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Example2

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

OPERATIONS ON A BINARY SEARCH TREE

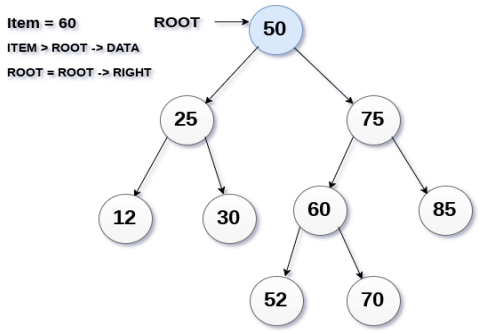
The following operations are performed on a binary search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

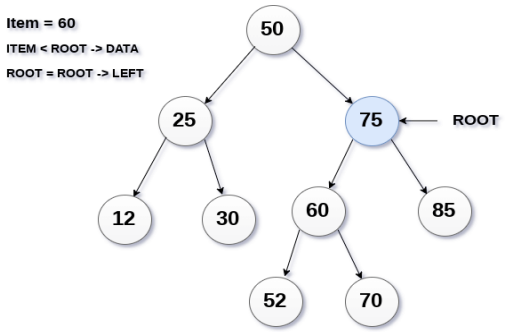
1. Search Operation in BST

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

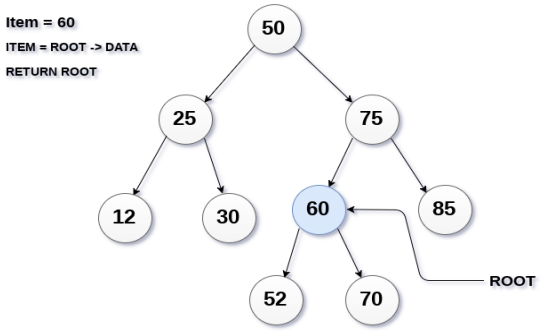
1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.



STEP 1



STEP 2



STEP 3

Algorithm:

Search (ROOT, ITEM)

- **Step 1:** IF ROOT -> DATA = ITEM OR ROOT = NULL
  Return ROOT
  ELSE
  IF ROOT < ROOT -> DATA
  Return search(ROOT -> LEFT, ITEM)
  ELSE
  Return search(ROOT -> RIGHT,ITEM)
  [END OF IF]
  [END OF IF]

- **Step 2:** END

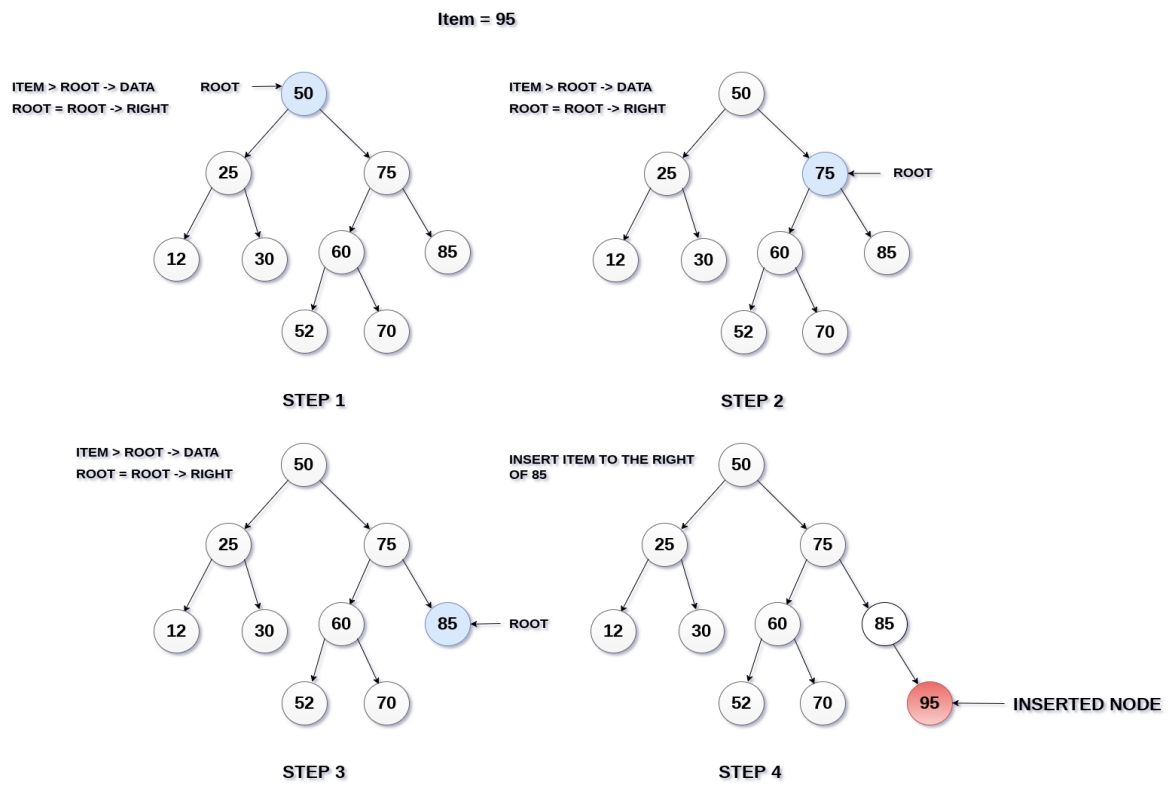
2. Insert Operation in BST

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

Insert (TREE, ITEM)

- Step 1: IF TREE = NULL
  Allocate memory for TREE
  SET TREE -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
  IF ITEM < TREE -> DATA
  Insert(TREE -> LEFT, ITEM)
  ELSE
  Insert(TREE -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
- Step 2: END



3. Delete Operation in BST

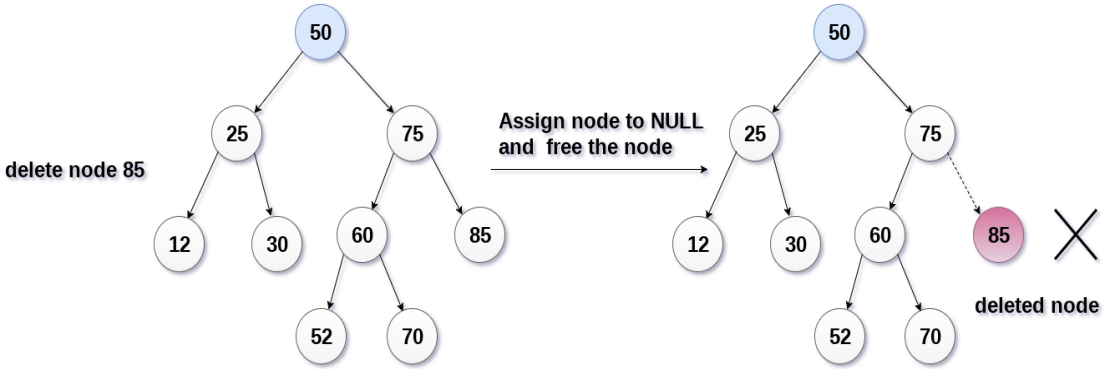
Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.

There are **three situations of deleting a node** from binary search tree.

a) The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

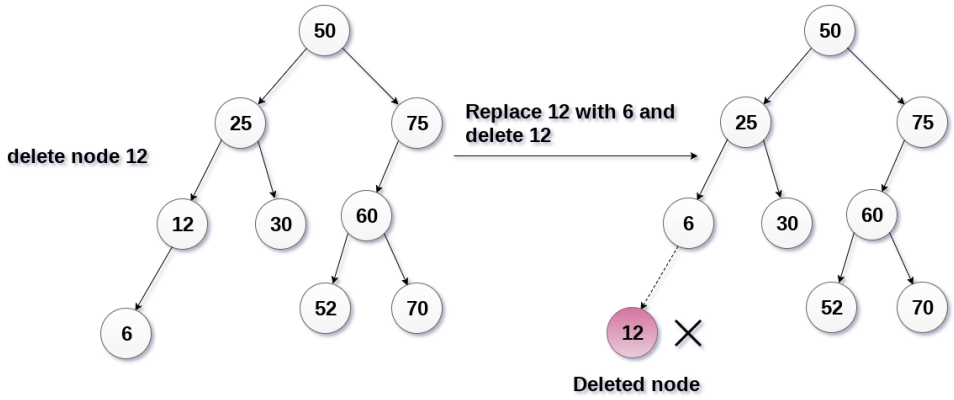
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



b) The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



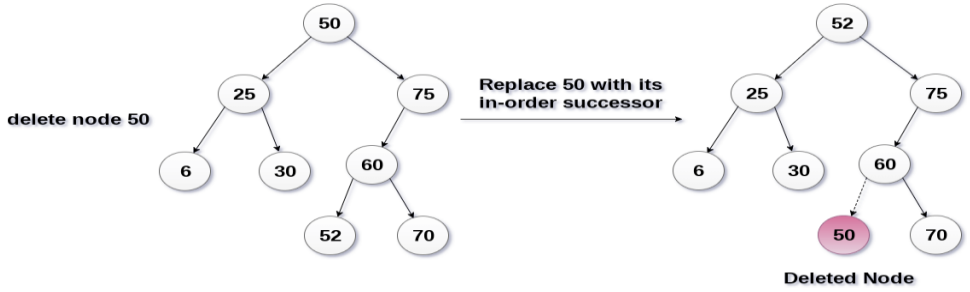
c) The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Algorithm Delete (TREE, ITEM)

```
Step1: IFTREE=NULL
            Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
            Delete(TREE->LEFT,ITEM)
       ELSE IF ITEM>TREE->DATA
            Delete(TREE->RIGHT,ITEM)
       ELSE IF TREE->LEFT AND TREE->RIGHT
             SET TEMP = findLargestNode(TREE -> LEFT)
            SET TREE -> DATA = TEMP -> DATA
            Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
            SET TEMP = TREE
       IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            SET TREE = NULL
       ELSE IF TREE -> LEFT != NULL
             SET TREE = TREE -> LEFT
        ELSE
            SET TREE = TREE -> RIGHT
       [END OF IF]
       FREE TEMP
       [END OF IF]
Step 2: END
```

AVL TREES

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

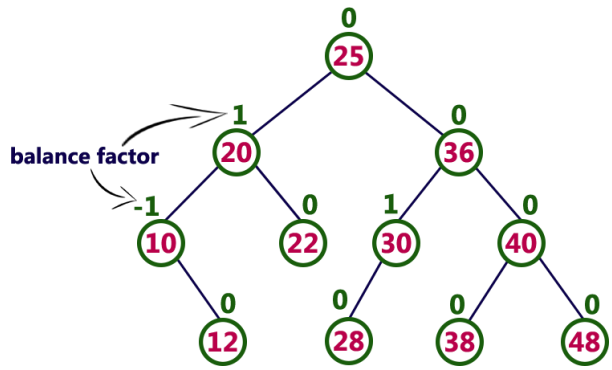
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of**

right subtree (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

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

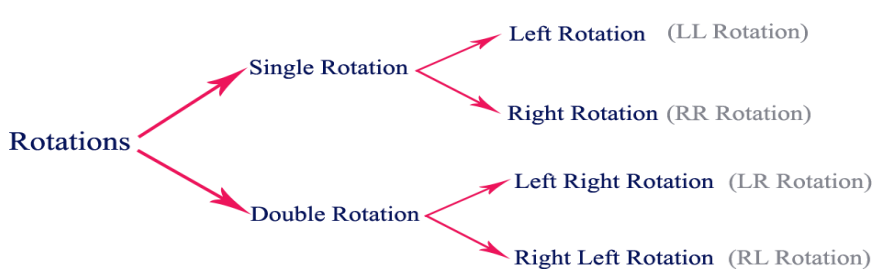
AVL Tree Rotations

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

Rotation operations are used to make the tree balanced.

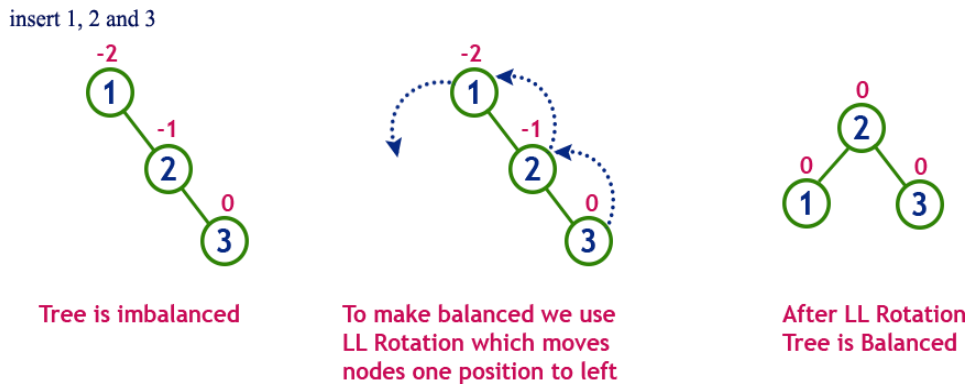
Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.



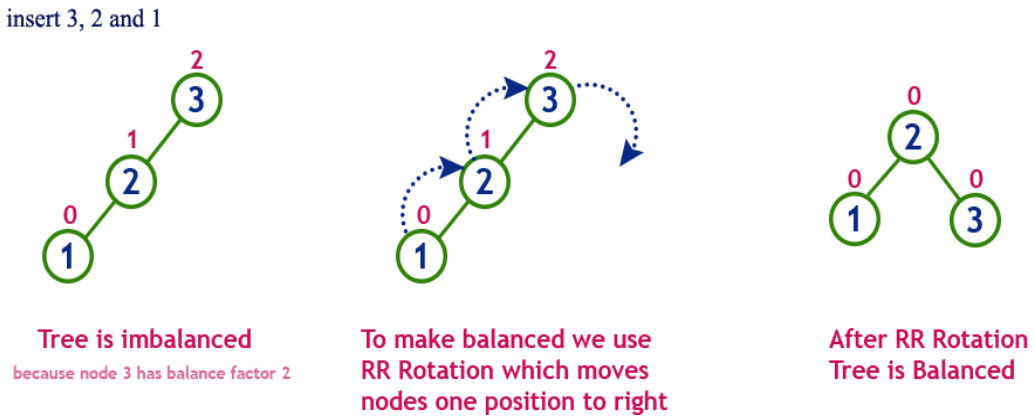
i) Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...



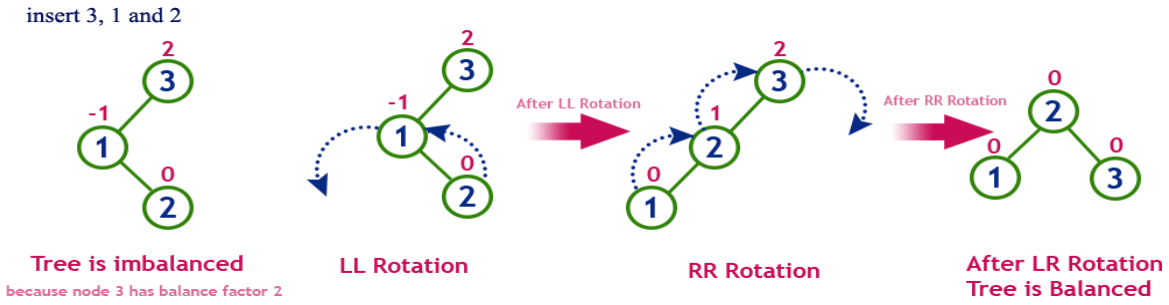
ii) Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



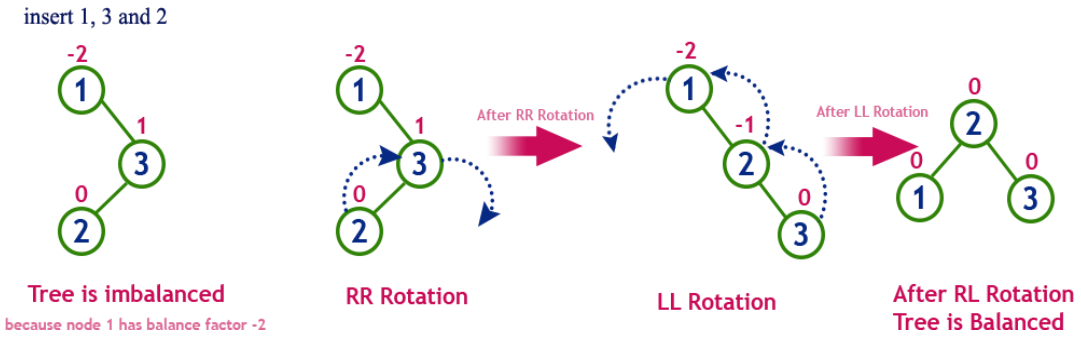
iii) Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



iv) Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

i) Search Operation in AVL Tree

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

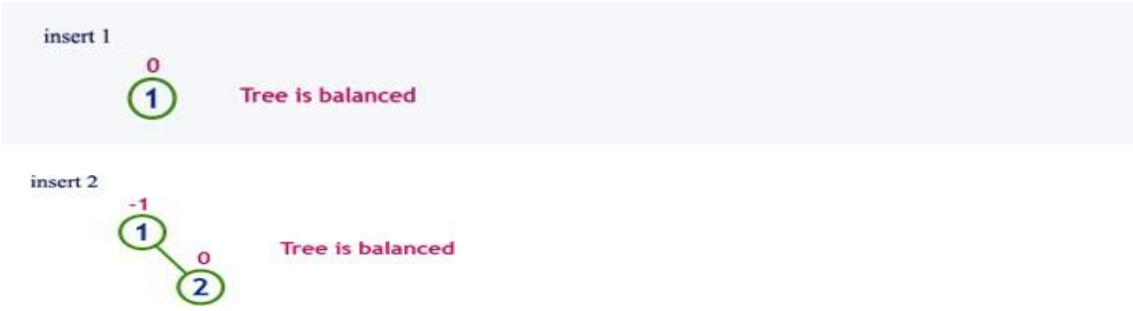
- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then continue the search process in right subtree.
- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

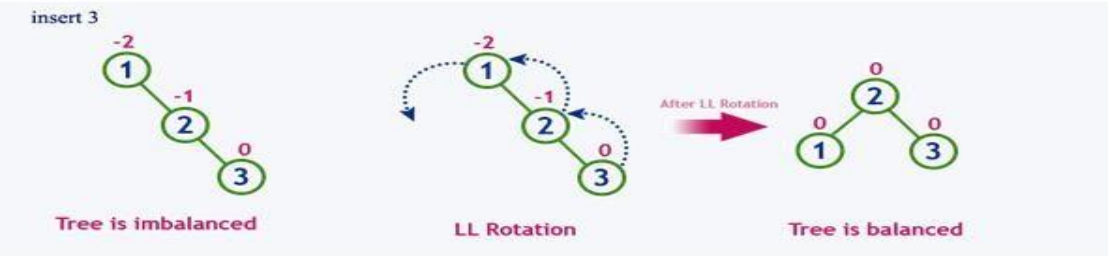
ii) Insertion Operation in AVL Tree

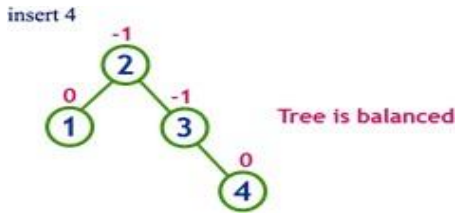
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

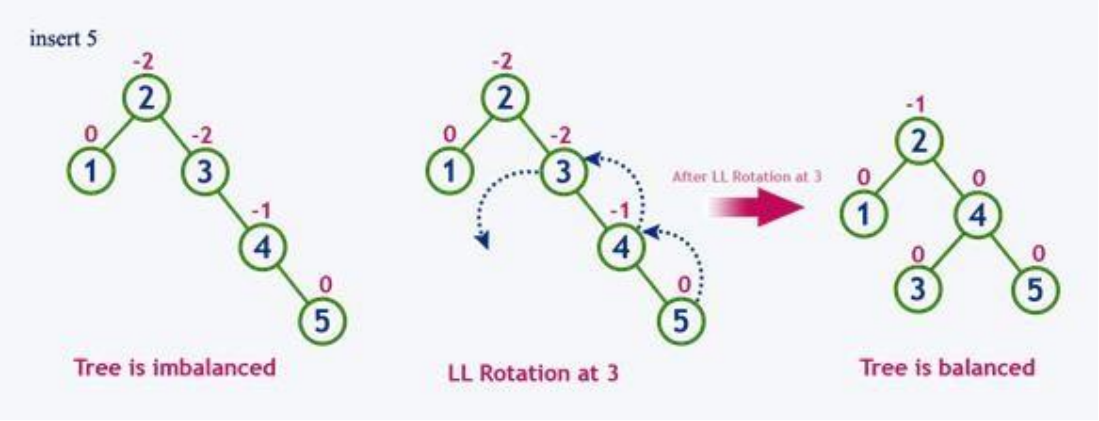
- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

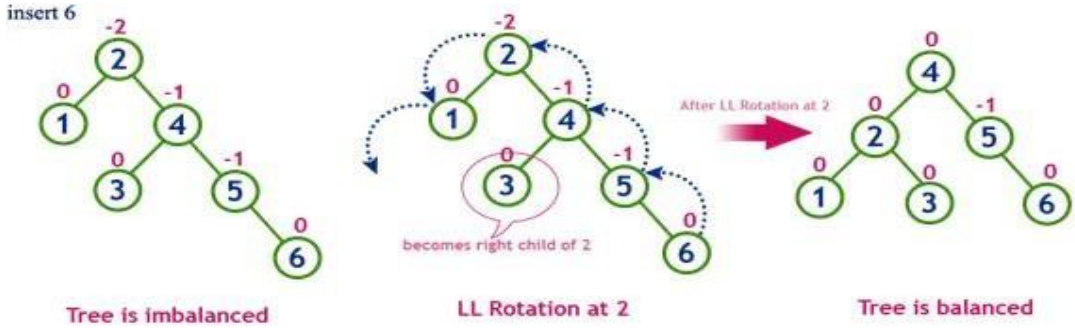
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

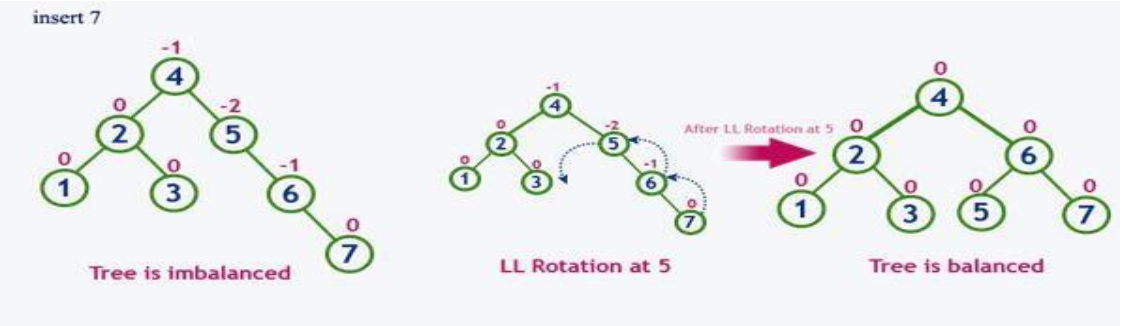


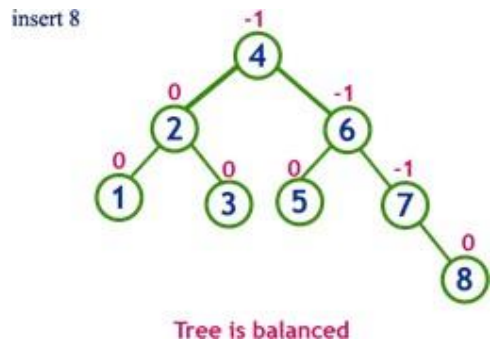

insert 3
Tree is imbalanced
LL Rotation
After LL Rotation
Tree is balanced


insert 4
Tree is balanced


insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced


insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced


insert 7
Tree is imbalanced
LL Rotation at 5
After LL Rotation at 5
Tree is balanced



iii) Deletion Operation in AVL Tree

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

The two types of rotations are **L rotation** and **R rotation.** Here, we will discuss R rotations. L rotations are the mirror images of them.

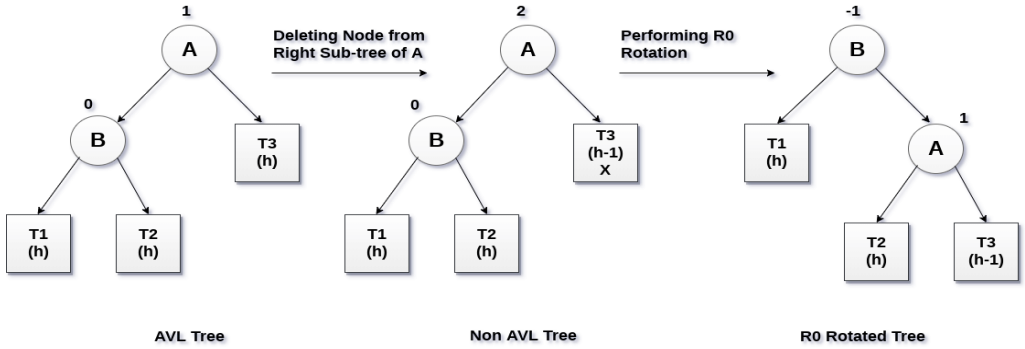
If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation will be applied.

Let us consider that, A is the critical node and B is the root node of its left sub-tree. If node X, present in the right sub-tree of A, is to be deleted, then there can be three different situations:

a) R0 rotation (Node B has balance factor 0)

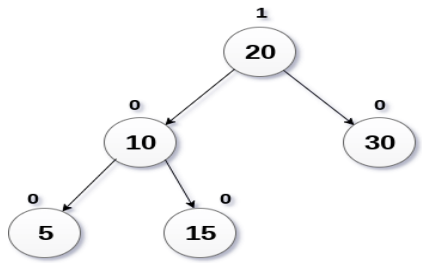
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A. the process involved in R0 rotation is shown in the following image.



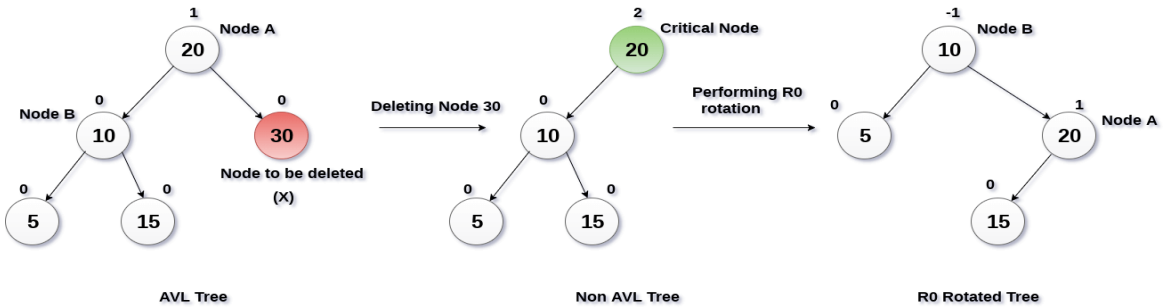
Example:

Delete the node 30 from the AVL tree shown in the following image.



Solution

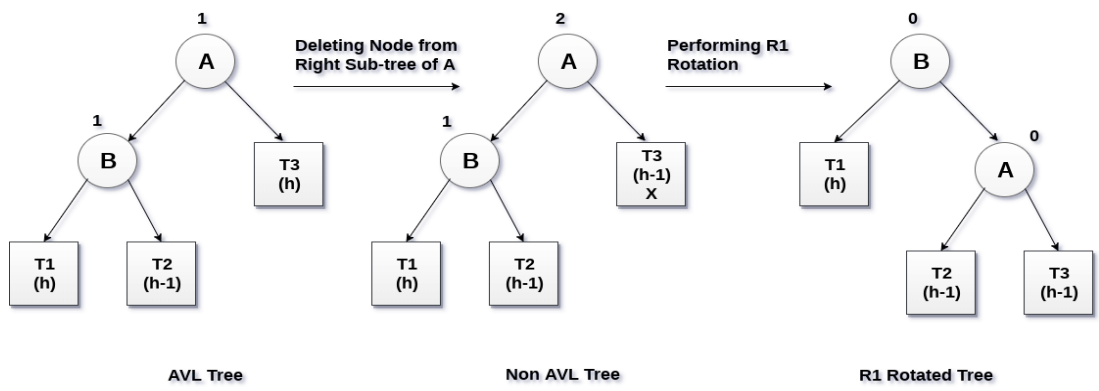
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



b) R1 Rotation (Node B has balance factor 1)

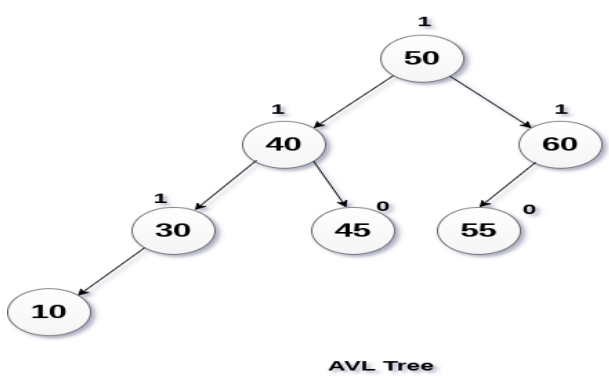
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

The process involved in R1 rotation is shown in the following image.



Example

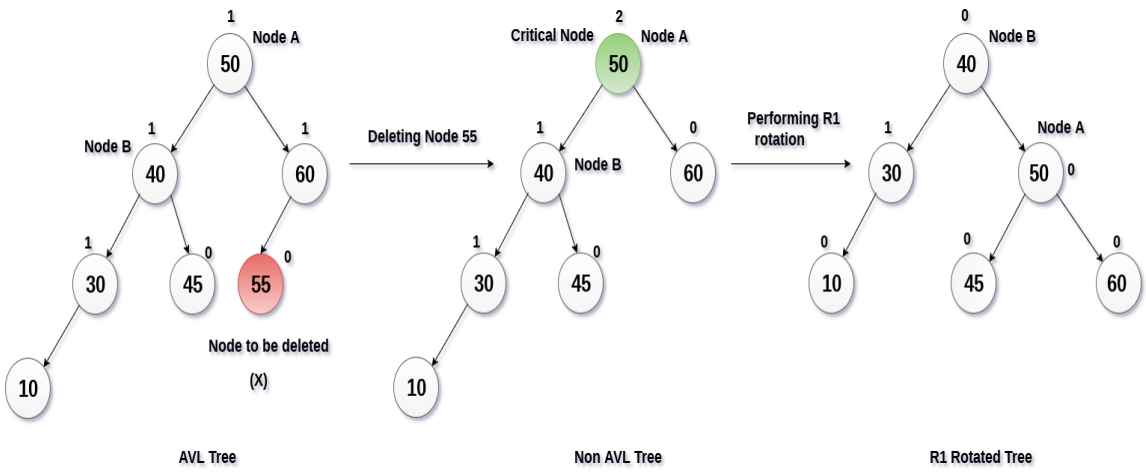
Delete Node 55 from the AVL tree shown in the following image.



Solution :

Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).

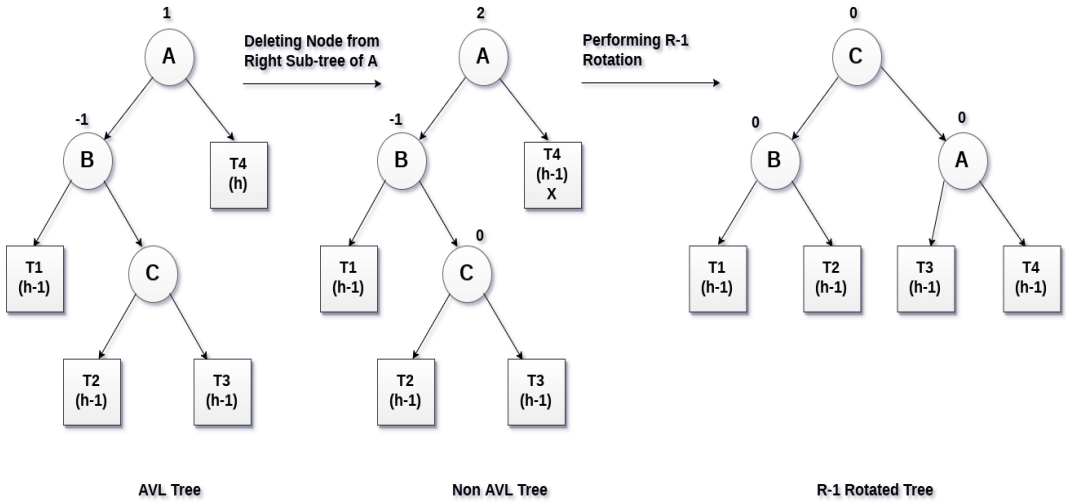
The process involved in the solution is shown in the following image.



c)R-1 Rotation (Node B has balance factor -1)

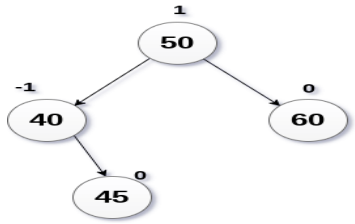
R-1 rotation is to be performed if the node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively.

The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A. The process involved in R-1 rotation is shown in the following image.



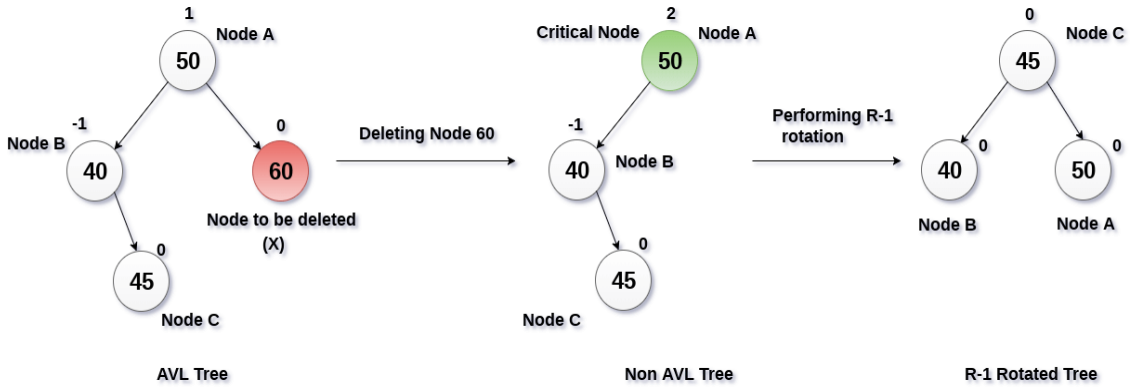
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.



Applications of A VL Trees

AVL trees are applied in th e following situations:

- There are few insertion and deletion operations
- Short search time is needed
- Input data is sorted or nearl y sorted

AVL tree structures can be used in situations which require fast searching. But, the large cost of re balancing may limit the usefulness.

B Tree

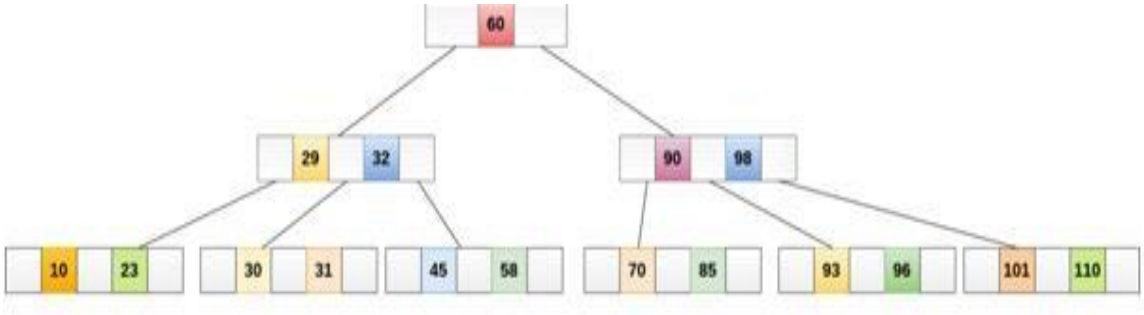
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations of B Trees

1.Searching

2.Insertion

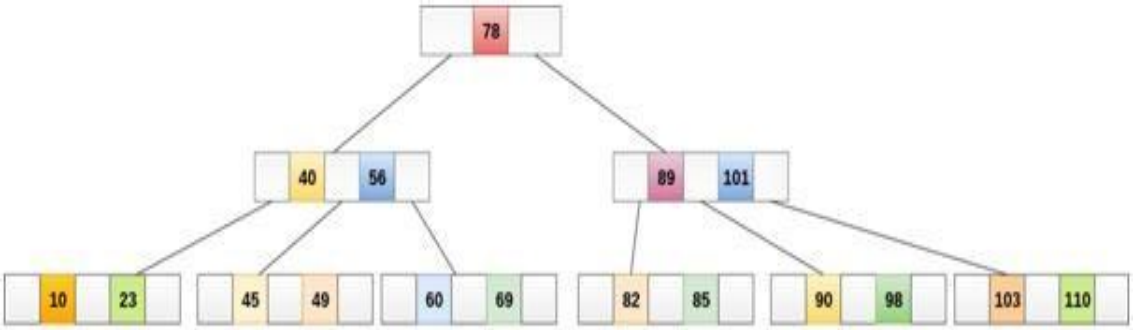
3.Deletion

i) Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



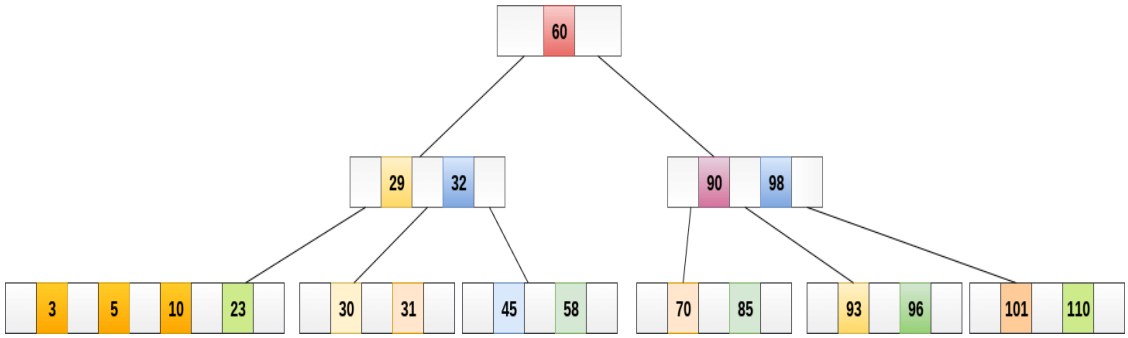
ii) Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

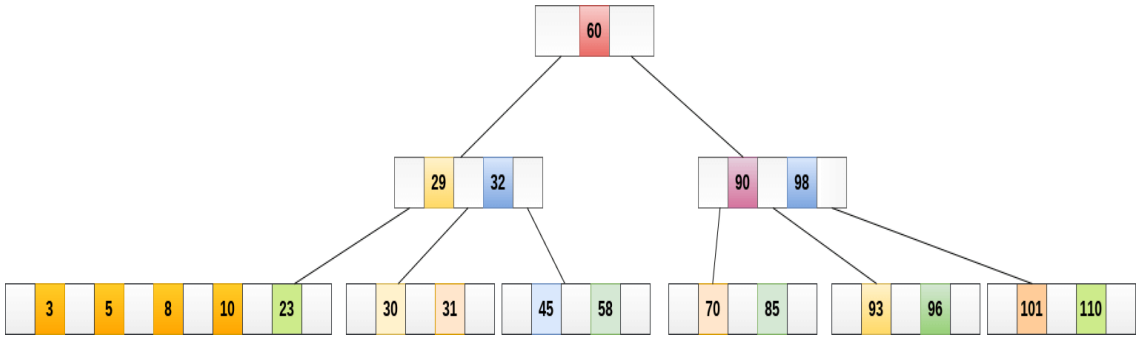
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

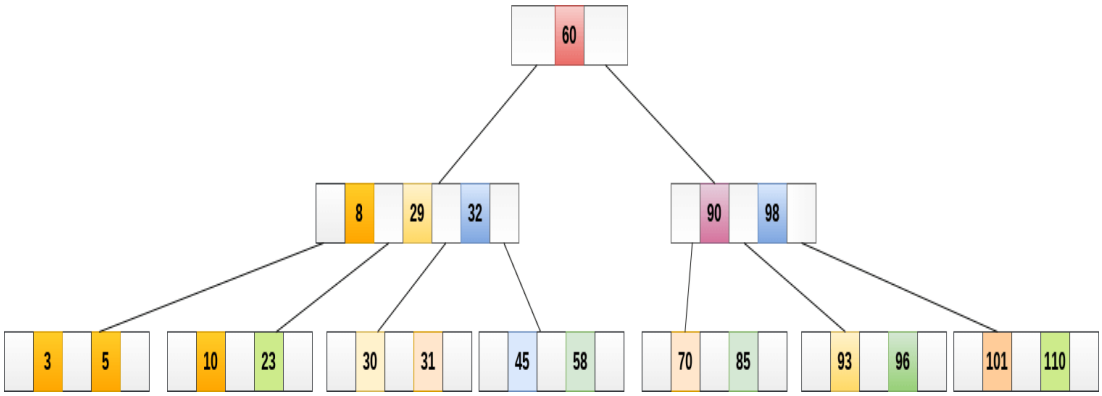
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



iii) Deletion

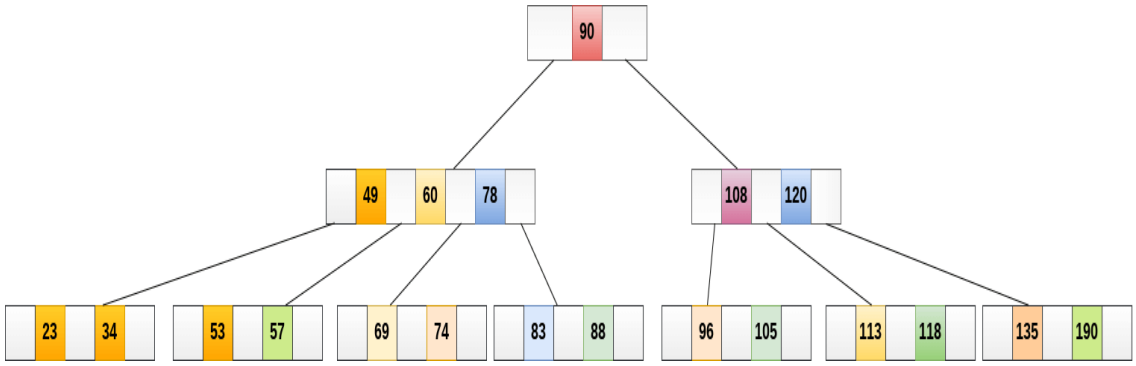
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

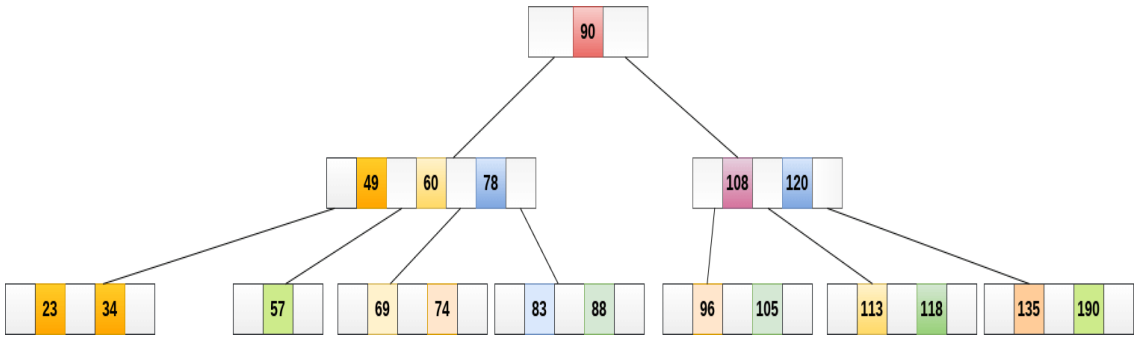
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

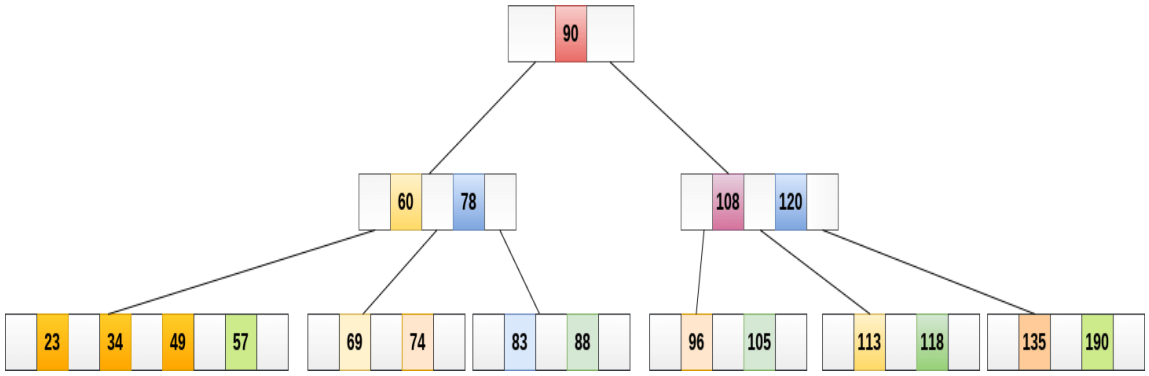


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

PART-A (2 Marks)

1. **Define a binary tree**

   Ans : A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub-tree and right sub-tree.

2. **Define a binary search tree**

   Ans : A binary search tree is a special binary tree, which is either empty or it should satisfy the following characteristics:

   Every node has a value and no two nodes should have the same value i.e) the values in the binary search tree are distinct

   • The values in any left sub-tree is less than the value of its parent node

   • The values in any right sub-tree is greater than the value of its parent node

   • The left and right sub-trees of each node are again binary search trees

3. **Define AVL Tree.**

   Ans: AVL stands for Adelson-Velskii and Landis. An AVL tree is a binary search tree which has the following properties:

   1. The sub-trees of every node differ in height by at most one.

   2. Every sub-tree is an AVL tree.

   Search time is O(logn). Addition and deletion operations also take O(logn) time.

4. **What do you mean by balanced trees?**

   Ans: Balanced trees have the structure of binary trees and obey binary search tree properties. Apart from these properties, they have some special constraints, which differ from one data structure to another. However, these constraints are aimed only at reducing the height of the tree, because this factor determines the time complexity. Eg: AVL trees, Splay trees.

5. **What are the categories of AVL rotations?**

   Ans: Let A be the nearest ancestor of the newly inserted nod which has the balancing factor ±2. Then the rotations can be classified into the following four categories:

   Left-Left: The newly inserted node is in the left subtree of the left child of A.

   Right-Right: The newly inserted node is in the right subtree of the right child of A.

   Left-Right: The newly inserted node is in the right subtree of the left child of A.

   Right-Left: The newly inserted node is in the left subtree of the right child of A.

6. **What do you mean by balance factor of a node in AVL tree?**

   Ans: The height of left subtree minus height of right subtree is called balance factor of a node in AVL tree.The balance factor may be either 0 or +1 or -1.The height of an empty tree is -1.

7. **List out the steps involved in deleting a node from a binary search tree.**

   Ans:
   - Deleting a node is a leaf node (ie) No children
   - Deleting a node with one child.
   - Deleting a node with two Childs.

8. **What is 'B' Tree?**

   Ans: A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

PART-B (10 Marks)

1. Explain the AVL tree insertion and deletion with suitable example.
2. Describe the algorithms used to perform single and double rotation on AVL tree.
3. Explain about B+ trees with suitable algorithm.
4. How to insert and delete an element into a binary search tree and write down the code for the insertion routine with an example.
5. What are binary search tree? Write an algorithm for deleting a node in a binary search tree.
6. Create a binary search tree for the following numbers start from an empty binary search tree. 45,26,10,60,70,30,40 Delete keys 10,60 and 45 one after the other and show the trees at each

UNIT –III

Red-Black Trees, *Splay Trees, Applications.* ***Hash Tables:*** *Introduction, Hash Structure, Hash functions, Linear Open Addressing, Chaining and Applications.*

Red - Black Tree

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

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

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

Properties of Red Black Tree

- **Property #1:** Red - Black Tree must be a Binary Search Tree.
- **Property #2:** The ROOT node must be colored BLACK.
- **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.
- **Property #5:** Every new node must be inserted with RED color.
- **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

Example

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

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

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

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree.

If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

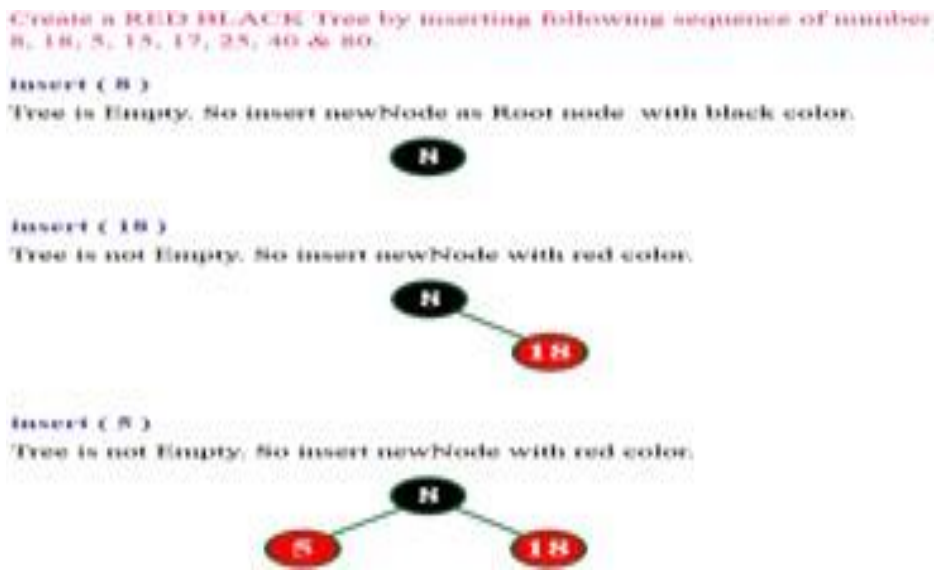
- **1. Recolor**
- **2. Rotation**
- **3. Rotation followed by Recolor**

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

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

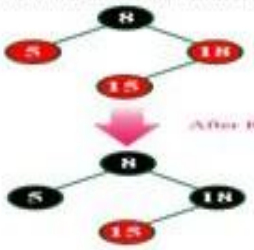
Deletion Operation in Red Black Tree

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



Insert (15)

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



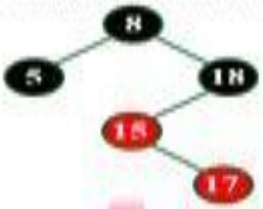
Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After RECOLOR

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

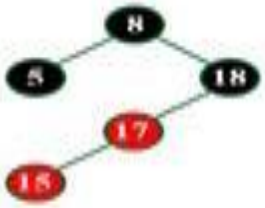
Insert (17)

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

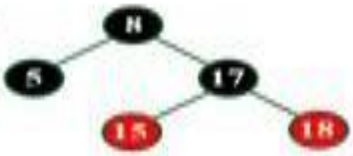


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

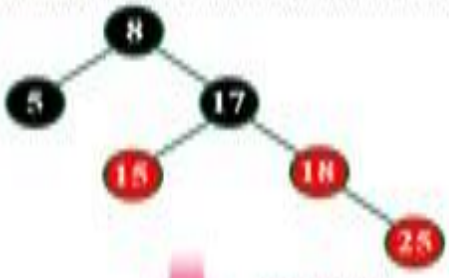


After Right Rotation & Recolor



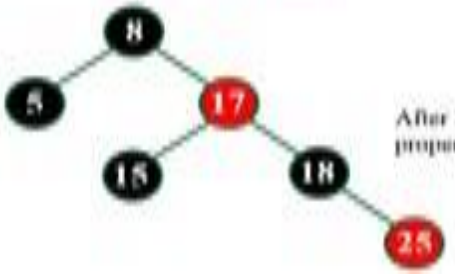
Insert (25)

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



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

After Recolor



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

Splay Tree Data structure

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called "Splaying".

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations. In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree. By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements. . Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process .In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

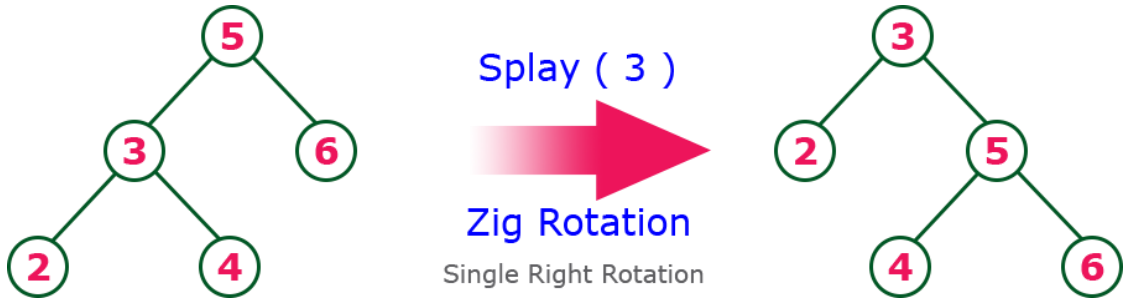
- **1. Zig Rotation**
- **2. Zag Rotation**
- **3. Zig - Zig Rotation**
- **4. Zag - Zag Rotation**
- **5. Zig - Zag Rotation**
- **6. Zag - Zig Rotation**

Example

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position.

Consider the following example...

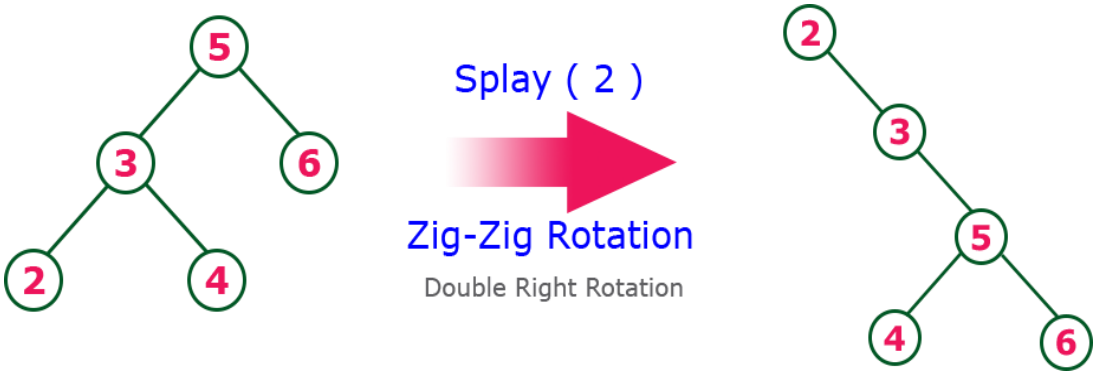


Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

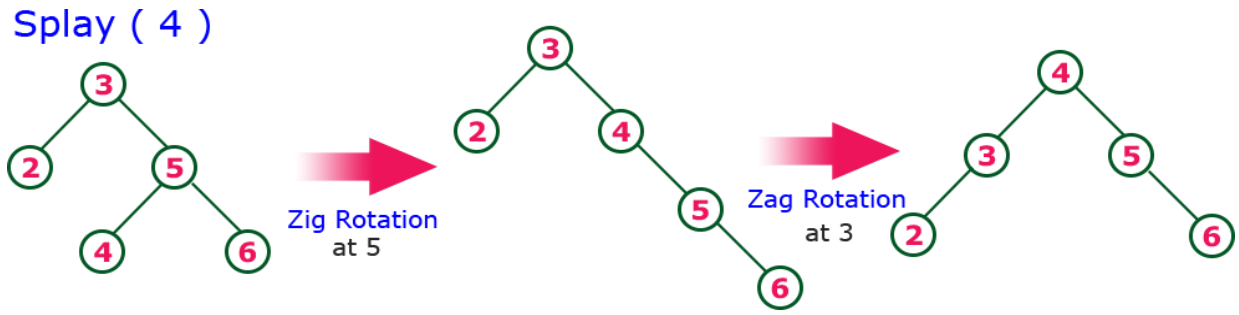


Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...

Zig-Zag Rotation

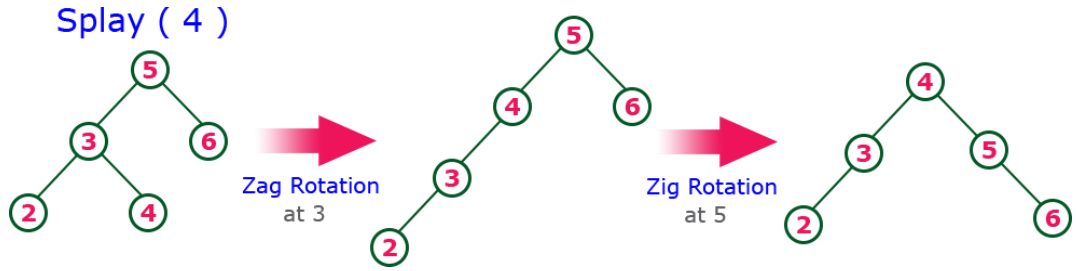
The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.

Consider the following example...



Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Applications:

Decision-based algorithm is used in machine learning which works upon the algorithm of tree. Databases also uses tree data structures for indexing. **Domain Name Server(DNS)** also uses tree structures. File explorer/my computer of mobile/any computer.

Some applications of the trees are:

1. XML Parser uses tree algorithms.
2. Decision-based algorithm is used in machine learning which works upon the algorithm of tree.
3. Databases also uses tree data structures for indexing.
4. Domain Name Server(DNS) also uses tree structures.
5. File explorer/my computer of mobile/any computer
6. BST used in computer Graphics
7. Posting questions on websites like Quora, the comments are child of questions

Hash Tables :

Introduction:

We've seen searches that allow you to look through data in $O(n)$ time, and searches that allow you to look through data in $O(logn)$ time, but imagine a way to find exactly what you want in $O(1)$ time. Think it's not possible? Think again! Hash tables allow the storage and retrieval of data in an average time of $O(1)$.

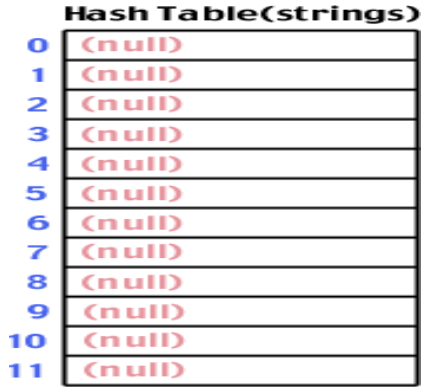
At its most basic level, a hash table data structure is just an array. Data is stored into this array at specific indices designated by a hash function. A hash function is a mapping between the set of input data and a set of integers.

With hash tables, there always exists the possibility that two data elements will hash to the same integer value. When this happens, a collision results (two data members try to occupy the same place in the hash table array),

and methods have been devised to deal with such situations. In this guide, we will cover two methods, linear probing and separate chaining, focusing on the latter.

A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array. In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

Let's take a simple example. First, we start with a hash table array of strings (we'll use strings as the data being stored and searched in this example). Let's say the hash table size is 12:



Next we need a hash function. There are many possible ways to construct a hash function. We'll discuss these possibilities more in the next section. For now, let's assume a simple hash function that takes a string as input. The returned hash value will be the sum of the ASCII characters that make up the string mod the size of the table:

*int hash(char *str, int table_size) { int sum; /* Make sure a valid string passed in */ if (str==NULL) return -1; /**

*Sum up all the characters in the string */ for(; *str; str++) sum += *str; /* Return the sum mod the table size */ return sum % table_size; }*

We run "Steve" through the hash function, and find that hash("Steve",12) yields 3:

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	"Steve"
4	(null)
5	(null)
6	(null)
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure %: The hash table after inserting "Steve"

Let's try another string: "Spark". We run the string through the hash function and find that hash("Spark",12) yields 6. Fine. We insert it into the hash table:

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	"Steve"
4	(null)
5	(null)
6	"Spark"
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure %: The hash table after inserting "Spark"

Let's try another: "Notes". We run "Notes" through the hash function and find that hash("Notes",12) is 3. Ok. We insert it into the hash table:

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	"Steve" "Notes"
4	(null)
5	(null)
6	"Spark"
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure %: A hash table collision

What happened? A hash function doesn't guarantee that every input will map to a different output. There is always the chance that two inputs will hash to the same output. This indicates that both elements should be inserted at the same place in the array, and this is impossible. This phenomenon is known as a collision.

There are many algorithms for dealing with collisions, such as linear probing an d separate chaining. While each of the methods has its advantages, we will only discuss separate chaining here.

Separate chaining requires a slight modification to the data structure. Instead of storing the data elements right into the array, they are stored in linked lists. Each slot in the array then points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. Because a linked list has no limit on length, collisions are no longer a problem. If more than one element hashes to the same value, then both

are stored in that linked list.

Let's look at the above example again, this time with our modified data structure:

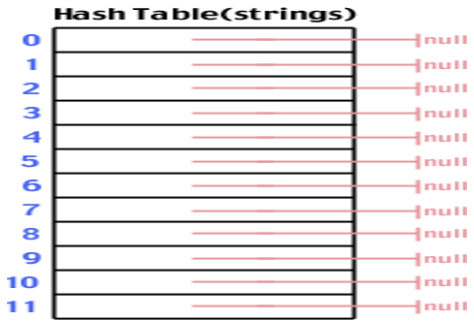


Figure %: Modified table for separate chaining

Again, let's try adding "Steve" which hashes to 3:

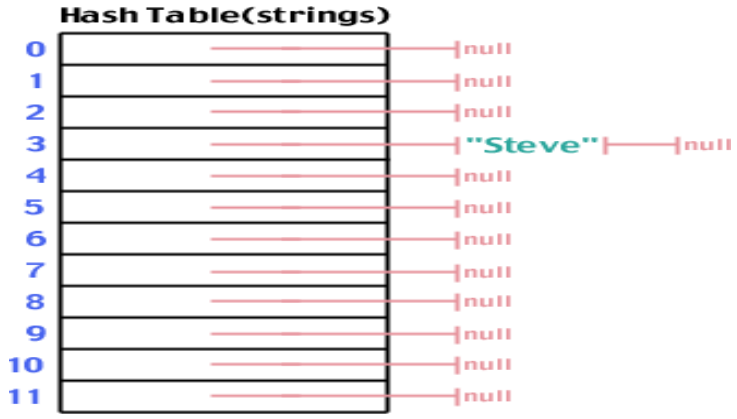


Figure %: After adding "Steve" to the table And "Spark" which hashes to 6:

Problem : How does a hash table allow for $O(1)$ searching? What is the worst case efficiency of a look up in a hash table using separate chainging?

A hash table uses hash functions to compute an integer value for data. This integer value can then be used as an index into an array, giving us a constant time access to the requested data. However, using separate chaining, we won't always achieve the best and average case efficiency of $O(1)$. If we have too small a hash table for the data set size and/or a bad hash function, elements can start to build in one index in the array. Theoretically, all n element could end up in the same linked list. Therefore, to do a search in the worst case is equivalent to looking up a data element in a linked list, something we already know to be $O(n)$ time. However, with a good hash function and a well created hash table, the chances of this happening are, for all intents and purposes, ignorable. **Problem :** The bigger the ratio between the size of the hash table and the number of data elements, the less chance there is for collision. What is a drawback to making the hash table big enough so the chances of collision is ignorable?
Wasted memory space

Problem : How could a linked list and a hash table be combined to allow someone to run through the list from item to item while still maintaining the ability to access an individual element in $O(1)$ time?

Hash Functions

As mentioned briefly in the previous section, there are multiple ways for constructing a hash function. Remember that hash function takes the data as input (often a string), and return s an integer in the range of possible indices into the hash table. Every hash function must do that, including the bad ones. So what makes for a good hash function?

Characteristics of a Good Hash Function

There are four main characteristics of a good hash function:
1) The hash value is fully determined by the data being hashed.
2) The hash function uses all the input data.
3) The hash function "uniformly" distributes the data across the entire set of possible hash values.
4) The hash function generates very different hash values for similar strings. Let's examine why each of these is important:

Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.

Rule 2: If the hash function doesn't use all the data would

cause an inappropriate number of similar hash values resulting in too many collisions.

Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

Rule 4: In real world applications, many data sets contain very similar data elements.

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

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

Hashing

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

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

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

Linear Probing

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

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

Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

delete − Deletes an element from a hash table.

DataItem

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

```
struct DataItem {
int data;
  int key;
};
```

Hash Method

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

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

Search Operation

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

Insert Operation

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

Delete Operation

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

OpenAddressing

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

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

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

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

Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as seen in the example below.

Let **hash(x)** be the slot index computed using a hash function and **S** be the table size

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

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

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

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Challenges in Linear Probing :

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i^2'th slot in i'th iteration.

let hash(x) be the slot index computed using hash function.

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

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

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

c) **Double Hashing** We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Comparison:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute. Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	**Separate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

PerformanceofOpenAddressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

Applications of Hashing

- Message Digest.
- Password Verification.
- Data Structures(Programming Languages)
- Compiler Operation.
- Rabin-Karp Algorithm.
- Linking File name and path together.

PART-A (2 Marks)

1. Define splay tree.

Ans: Splay tree is a binary search tree in which restructuring is done using a scheme called splay. The splay is a heuristic method which moves a given vertex v to the root of the splay tree using a sequence of rotations.

2. What is the idea behind splaying?

Ans: Splaying reduces the total accessing time if the most frequently accessed node is moved towards the root. It does not require to maintain any information regarding the height or balance factor and hence saves space and simplifies the code to some extent

3. Define hashing function.

Ans: A hashing function is a key-to-transformation, which acts upon a given key to compute the relative position of the key in an array. A simple hash function HASH(KEY_Value)= (KEY_Value)mod(Table-size).

4. What is open addressing?

Ans: Open addressing is also called closed hashing, which is an alternative to resolve the collisions with linked lists. In this hashing system, if a collision occurs, alternative cells are tired until an empty cell is found.

5. What are the collision resolution methods?

Ans: The following are the collision resolution methods:

- Separate chaining
- Open addressing
- Multiple hashing

6. Define separate chaining It is an open hashing technique.

Ans: A pointer field is added to each record location, when an overflow occurs, this pointer is set to point to overflow blocks making a linked list. In this method, the table can never overflow, since the linked lists are only extended upon the arrival of new keys.

7. Define Hashing.

Ans: Hashing is the transformation of string of characters into a usually shorter fixed length

value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the short hashed key than to find it using the original value.

8. What do you mean by hash table?

Ans: The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value. Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell.

9. Write the importance of hashing.

Ans: • Maps key with the corresponding value using hash function.

• Hash tables support the efficient addition of new entries and the time spent on searching for the required data is independent of the number of items stored.

10. What do you mean by collision in hashing?

Ans: When an element is inserted, it hashes to the same value as an already inserted element, and then it produces collision.

11. What are the collision resolution methods?

Ans: • Separate chaining or External hashing

• Open addressing or Closed hashing.

12. What do you mean by open addressing?

Ans: Open addressing is a collision resolving strategy in which, if collision occurs alternative cells are tried until an empty cell is found. The cells h0(x), h1(x), h2(x),…. are tried in succession, where hi(x)=(Hash(x)+F(i))mod Tablesize with F(0)=0. The function F is the collision resolution strategy.

13. List the limitations of linear probing.

Ans: • Time taken for finding the next available cell is large.

• In linear probing, we come across a problem known as clustering.

14. Mention one advantage and disadvantage of using quadratic probing.

Ans: Advantage: The problem of primary clustering is eliminated.

Disadvantage: There is no guarantee of finding an unoccupied cell once the table is nearly half

full.

15. what are the properties of red black tree?

Ans: **Each tree node is colored either red or black. The root node of the tree is always black**. Every path from the root to any of the leaf nodes must have the same number of black nodes. No two red nodes can be adjacent, i.e., a red node cannot be the parent or the child of another red node.

PART-B (10 Marks)

1. Explain Operations on Red Black tree.
2. Explain Operations on splay tree.
3. Explain Chaining method and Open Addressing with an example
4. Explain double hashing and Bucket hashing with an example
5. Explain difference between Linear Probing and Quadratic Probing with an example
6. Explain Rehashing and Extendible hashing with an example

UNIT –IV

Divide and conquer: *General method, applications-Binary search, Finding Maximum and minimum, Quick sort, Merge sort, Strassen's matrix multiplication*

Greedy method: *General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.*

4.1.General method:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.
- For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
- D And C(Algorithm) is initially invoked as D and C(P), where 'p' is the problem to be solved.
- Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting.
- If this so, the function 'S' is invoked.
- Otherwise, the problem P is divided into smaller sub problems.
- These sub problems P1, P2 …Pk are solved by recursive application of D And C.
- Combine is a function that determines the solution to p using the solutions to the 'k' sub problems.
- If the size of 'p' is n and the sizes of the 'k' sub problems are n1, n2 ….nk, respectively, then the computing time of D And C is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n1)+T(n2)+\ldots\ldots\ldots\ldots+T(nk)+f(n); & \text{otherwise.} \end{cases}$$

Where T(n) → is the time for D And C on any I/p of size 'n'.

g(n) → is the time of compute the answer directly for small I/ps.

f(n) → is the time for dividing P & combining the solution to sub problems.

```
Algorithm D And C(P)
{
if small(P) then return S(P);
else
{
divide P into smaller instances
        P1, P2… Pk, k>=1;
 Apply D And C to each of these sub problems;
return combine (D And C(P1), D And C(P2),…….,D And C(Pk));
  }
}
```

- The complexity of many divide-and-conquer algorithms is given by recurrences of the form

  $$T(n) = \begin{cases} T(1) & n=1 \\ AT(n/b)+f(n) & n>1 \end{cases}$$

  → Where a & b are known constants.

  → We assume that T(1) is known & 'n' is a power of b(i.e., n=b^k)
- One of the methods for solving any such recurrence relation is called the substitution method.
- This method repeatedly makes substitution for each occurrence of the function. T is the Right-hand side until all such occurrences disappear.

Example:

1) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n.

   We have,

   T(n) = 2T(n/2)+n

   = 2[2T(n/2/2)+n/2]+n

   = [4T(n/4)+n]+n

   = 4T(n/4)+2n

   = 4[2T(n/4/2)+n/4]+2n

= 4[2T(n/8)+n/4]+2n

= 8T(n/8)+n+2n

= 8T(n/8)+3n

*

*

*

- In general, we see that T(n)=2^iT(n/2^i)+in., for any log n >=I>=1.

→ T(n) =2^log n T(n/2^log n) + n log n

Thus, T(n) = 2^log n T(n/2^log n) + n log n

= n. T(n/n) + n log n

= n. T(1) + n log n [since, log 1=0, 2^0=1]

= 2n + n log n

BINARY SEARCH:

```
Algorithm Bin search(a,n,x)
// Given an array a[1:n] of elements in non-decreasing
//order, n>=0,determine whether 'x' is present and
// if so, return 'j' such that x=a[j]; else return 0.
{
low:=1; high:=n;
while (low<=high) do
{
    mid:=[(low+high)/2];
    if (x<a[mid]) then high;
    else if(x>a[mid]) then
        low=mid+1;
  else return mid;
 }
    return 0;
}
```

Algorithm, describes this binary search method, where Binsrch has 4I/ps a[], I , l & x.

It is initially invoked as Binsrch (a,1,n,x)

A non-recursive version of Binsrch is given below.

This Binsearch has 3 i/ps a,n, & x.The while loop continues processing as long as there are more elements left to check. At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x. We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.

Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.

Example: Let us select the 14 entries.

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

→ Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of 'x'.

→ Only the variables, low, high & mid need to be traced as we simulate the algorithm.

→ We try the following values for x: 151, -14 and 9.

for 2 successful searches & 1 unsuccessful search.

Table. Shows the traces of Bin search on these 3 steps.

X=151

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14
		Found

x=-14

low	high	mid
1	14	7
1	6	3
1	2	1
2	2	2
2	1	Not found

x=9

low	high	mid
1	14	7
1	6	3
4	6	5
		Found

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof:

We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

- Initially low =1, high= n,n>=0, and a[1]<=a[2]<=..... <=a[n].
- If n=0, the while loop is not entered and is returned.
- Otherwise we observe that each time thro' the loop the possible elements to be checked of or equality with x and a[low], a[low+1],……..,a[mid],……a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the search.
- If low becomes > than high, then 'x' is not present & hence the loop is exited.

FINDING THE MAXIMUM AND MINIMUM:

- Let us consider another simple problem that can be solved by the divide-and-conquer technique.
- The problem is to find the maximum and minimum items in a set of 'n' elements.
- In analyzing the time complexity of this algorithm, we once again concentrate on the no. of element comparisons.
- More importantly, when the elements in a[1:n] are polynomials, vectors, very large numbers, or strings of character,

  the cost of an element comparison is much higher than the cost of the other operations.
- Hence, the time is determined mainly by the total cost of the element comparison.

Algorithm straight MaxMin(a,n,max,min)

// set max to the maximum & min to the minimum of a[1:n]

{

```
        max:=min:=a[1];
        for I:=2 to n do
        {
            if(a[I]>max) then max:=a[I];
           if(a[I]<min) then min:=a[I];
         }
    }
```

Algorithm: Straight forward Maximum & Minimum

Straight MaxMin requires 2(n-1) element comparison in the best, average & worst cases.

An immediate improvement is possible by realizing that the comparison a[I]<min is necessary only when a[I]>max is false. Hence we can replace the contents of the for loop by,

```
If(a[I]>max) then max:=a[I];
Else if (a[I]<min) then min:=a[I];
```

Now the best case occurs when the elements are in increasing order.

→ The no. of element comparison is (n-1).

The worst case occurs when the elements are in decreasing order.

→ The no. of elements comparison is 2(n-1)

The average no. of element comparison is < than 2(n-1)

On the average a[I] is > than max half the time, and so, the avg. no. of comparison is 3n/2-1.

A divide- and conquer algorithm for this problem would proceed as follows:

→ Let P=(n, a[I] ,…….,a[j]) denote an arbitrary instance of the problem.

→ Here 'n' is the no. of elements in the list (a[I],….,a[j]) and we are interested in finding the maximum and minimum of the list.

If the list has more than 2 elements, P has to be divided into smaller instances.

For example , we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2],a[[n/2]+1],…..,a[n])

After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm: Recursively Finding the Maximum & Minimum

```
Algorithm MaxMin (I,j,max,min)
//a[1:n] is a global array, parameters I & j
//are integers, 1<=I<=j<=n.The effect is to
//set max & min to the largest & smallest value
//in a[I:j], respectively.
{
if(I=j) then max:= min:= a[I];
else if (I=j-1) then // Another case of small(p)
{
if (a[I]<a[j]) then
{
max:=a[j];
min:=a[I];
}
else
{
max:=a[I];
min:=a[j];
}
}
else
{
// if P is not small, divide P into subproblems.
// find where to split the set mid:=[(I+j)/2];
//solve the subproblems
MaxMin(I,mid,max.min);
MaxMin(mid+1,j,max1,min1);
//combine the solution
if (max<max1) then max=max1;
if(min>min1) then min = min1;
```

}

}

The procedure is initially invoked by the statement,

MaxMin(1,n,x,y)

Suppose we simulate MaxMin on the following 9 elements

A:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made.

For this Algorithm, each node has 4 items of information: I, j, max & imin.Examining fig: we see that the root node contains 1 & 9 as the values of I &j corresponding to the initial call to MaxMin. This execution produces 2 new calls to MaxMin, where I & j have the values 1, 5 & 6, 9 respectively & thus split the

set into 2 subsets of approximately the same size.From the tree, we can immediately see the maximum depth of recursion is 4. (including the 1st call)They include no.s in the upper left corner of each node represent the order in which max & min are assigned values.

No. of element Comparison:

If T(n) represents this no., then the resulting recurrence relations is

T(n)={ T([n/2]+T[n/2]+2 n>2

n=2

n=1

→ When 'n' is a power of 2, n=2^k for some +ve integer 'k', then

T(n) = 2T(n/2) +2

= 2(2T(n/4)+2)+2

= 4T(n/4)+4+2

*

*

= 2^k-1T(2)+

= 2^k-1+2^k-2

= 2^k/2+2^k-2

= n/2+n-2

= (n+2n)/2)-2

T(n)=(3n/2)-2

*Note that (3n/3)-3 is the best-average, and worst-case no. of comparisons when 'n' is a power of 2.

QUICK SORT

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j
  between (m+1) & n for some m, 1<=m<=n.
- Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.
- No merge is needed. This rearranging is referred to as partitioning.
- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]
- It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be
- defined and must be greater than or equal to all elements in a[1:n]
- The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning
  element than the first item in the set are better in practice.
- The function interchange (a,I,j) exchanges a[I] with a[j].

Algorithm: Partition the array a[m:p-1] about a[m]

```
Algorithm Partition(a,m,p)
//within a[m],a[m+1],…..,a[p-1] the elements
// are rearranged in such a manner that if
//initially t=a[m],then after completion
//a[q]=t for some q between m and
//p-1,a[k]<=t for m<=k<q, and
//a[k]>=t for  q<k<p. q is returned
//Set a[p]=infinite.
{
v=a[m];I=m;j=p;
repeat
{
repeat
    I=I+1;
 until(a[I]>=v);
repeat
     j=j-1;
until(a[j]<=v);
if (I<j) then interchange(a,i.j);
}until(I>=j);
a[m]=a[j]; a[j]=v;
retun j;
}

Algorithm Interchange(a,I,j)
//Exchange a[I] with a[j]
{
p=a[I];
a[I]=a[j];
a[j]=p;
```

}

Algorithm*:* Sorting by Partitioning

```
Algorithm Quicksort(p,q)
//Sort the elements a[p],….a[q] which resides
//is the global array a[1:n] into ascending
//order; a[n+1] is considered to be defined
// and must be >= all the elements in a[1:n]
{
if(p<q) then // If there are more than one element
{
// divide p into 2 subproblems
j=partition(a,p,q+1);
//'j' is the position of the partitioning element.
//solve the subproblems.
quicksort(p,j-1);
quicksort(j+1,q);
//There is no need for combining solution.
}
    }
```

Record Program: Quick Sort

```
#include  <stdio.h>
#include <conio.h>
int a[20];
main()
  {
      int n,I;
      clrscr();
      printf("QUICK SORT");
      printf("\n Enter the no. of elements ");
      scanf("%d",&n);
```

```
    printf("\nEnter the array elements");
     for(I=0;I<n;I++)
        scanf("%d",&a[I]);
     quicksort(0,n-1);
    printf("\nThe array elements are");
    for(I=0;I<n;I++)
    printf("\n%d",a[I]);
    getch();
}
quicksort(int p, int q)
{
   int j;
   if(p,q)
   {
      j=partition(p,q+1);
      quicksort(p,j-1);
      quicksort(j+1,q);
    }
}

Partition(int m, int p)
{
    int v,I,j;
    v=a[m];
    i=m;
    j=p;
    do
    {
        do
            i=i+1;
            while(a[i]<v);
       if (i<j)
             interchange(I,j);
    } while (I<j);
a[m]=a[j];
a[j]=v;
return j;
}

Interchange(int I, int j)
{
```

```
    int p;
    p= a[I];
    a[I]=a[j];
    a[j]=p;
}
```

Output:

Enter the no. of elements 5

Enter the array elements

3

8

1

5

2

The sorted elements are,

1

2

3

5

8

MERGE SORT

- As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n)
- This algorithm is called merge sort
- We assume throughout that the elements are to be sorted in non-decreasing order.
- Given a sequence of 'n' elements a[1],…,a[n] the general idea is to imagine then split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n].
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.
- Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm For Merge Sort:

```
Algorithm MergeSort(low,high)
//a[low:high] is a global array to be sorted
//Small(P) is true if there is only one element
//to sort. In this case the list is already sorted.
{
if (low<high) then //if there are more than one element
{
//Divide P into subproblems
//find where to split the set
mid = [(low+high)/2];
//solve the subproblems.
mergesort (low,mid);
mergesort(mid+1,high);
//combine the solutions .
merge(low,mid,high);
}
}
```

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
Algorithm merge(low,mid,high)
//a[low:high] is a global array containing
//two sorted subsets in a[low:mid]
//and in a[mid+1:high].The goal is to merge these 2 sets into
//a single set residing in a[low:high].b[] is an auxiliary global array.
{
h=low; I=low; j=mid+1;
while ((h<=mid) and (j<=high)) do
{
if (a[h]<=a[j]) then
{
   b[I]=a[h];
```

```
    h = h+1;
}
else
{
    b[I]= a[j];
     j=j+1;
}
I=I+1;
}
if (h>mid) then
   for k=j to high do
    {
        b[I]=a[k];
        I=I+1;
     }
 else
     for k=h to mid do
     {
         b[I]=a[k];
         I=I+1;
      }
     for k=low to high do a[k] = b[k];
}
```

- Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)
- Algorithm Mergesort begins by splitting a[] into 2 sub arrays each of size five (a[1:5] and a[6:10]).
- The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2(a[4:5])
- Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])
- The 2 values in a[1:2} are split to find time into one-element sub arrays, and now the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

→ Where vertical bars indicate the boundaries of sub arrays.

→Elements a[I] and a[2] are merged to yield,

(285, 310|179|652, 351| 423, 861, 254, 450, 520)

→ Then a[3] is merged with a[1:2] and

(179, 285, 310| 652, 351| 423, 861, 254, 450, 520)

→ Next, elements a[4] & a[5] are merged.

(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)

→ And then a[1:3] & a[4:5]

(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

→ Repeated recursive calls are invoked producing the following sub arrays.

(179, 285, 310, 351, 652| 423| 861| 254| 450, 520)

→ Elements a[6] &a[7] are merged.

→Then a[8] is merged with a[6:7]

(179, 285, 310, 351, 652| 254,423, 861| 450, 520)

→ Next a[9] &a[10] are merged, and then a[6:8] & a[9:10]

(179, 285, 310, 351, 652| 254, 423, 450, 520, 861)

→ At this point there are 2 sorted sub arrays & the final merge produces the fully sorted result.

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

- **If the time for the merging operations is proportional to 'n', then the computing time for merge sort is described by the recurrence relation.**

  T(n) = { a n=1,'a' a constant

  2T(n/2)+cn n>1,'c' a constant.

  → When 'n' is a power of 2, n= 2^k, we can solve this equation by successive substitution.

  T(n) =2(2T(n/4) +cn/2) +cn

  = 4T(n/4)+2cn

  = 4(2T(n/8)+cn/4)+2cn

  *

  *

  = 2^k T(1)+kCn.

  = an + cn log n.

→ It is easy to see that if s^k<n<=2^k+1, then T(n)<=T(2^k+1). Therefore,

T(n)=O(n log n)

STRASSEN'S MATRIX MULTIPLICAION

1. Let A and B be the 2 n*n Matrix. The product matrix C=AB is calculated by using the formula,

   C (i ,j)= A(i,k) B(k,j) for all 'i' and and j between 1 and n.

2. The time complexity for the matrix Multiplication is O(n^3).

3. Divide and conquer method suggest another way to compute the product of n*n matrix.

4. We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero
   can be added to both A and B .SO that the resulting dimension are the powers of two.

5. If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of
   A & B.

6. If n>2,Then the elements are partitioned into sub matrix n/2*n/2..since 'n' is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub
   matrix until 'N" become suitable small(n=2) so that the product is computed directly .

7. The formula are

$$\begin{pmatrix} A11 & A12 \\ A21 & A21 \end{pmatrix} * \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix} = \begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix}$$

C11 = A11 B11 + A12 B21

C12 = A11 B12 + A12 B22

C21 = A21 B11 + A22 B21

C22 = A21 B12 + A22 B22

For EX:

$$4 * 4 = \begin{matrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{matrix} * \begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{matrix}$$

The Divide and conquer method

$$\begin{matrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{matrix} * \begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{matrix} = \begin{matrix} 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 \end{matrix}$$

8. To compute AB using the equation we need to perform 8 multiplication of n/2*n/2 matrix and from 4 addition of n/2*n/2 matrix.
9. Ci,j are computed using the formula in equation →4
10. As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
11. The Cij are required addition 8 addition or subtraction.

$$T(n) = \begin{cases} b & n<=2 \\ 7T(n/2)+an^2 & n>2 \end{cases}$$

a &b are constant

Finally we get T(n) =O(n ^log27)

Example

$$\begin{matrix} 4 & 4 \\ 4 & 4 \end{matrix} * \begin{matrix} 4 & 4 \\ 4 & 4 \end{matrix}$$

P=(4*4)+(4+4)=64

Q=(4+4)4=32

R=4(4-4)=0

S=4(4-4)=0

T=(4+4)4=32

U=(4-4)(4+4)=0

V=(4-4)(4+4)=0

C11=(64+0-32+0)=32

C12=0+32=32

C21=32+0=32

C22=64+0-32+0=32

So the answer c(i,j) is $\begin{vmatrix} 32 & 32 \\ 32 & 32 \end{vmatrix}$

since n/2n/2 &matrix can be can be added in Cn for some constant C, The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

T(n)= | b n<=2 a &b are
| 8T(n/2)+cn^2 n>2 constant

That is T(n)=O(n^3)

* Matrix multiplication are more expensive then the matrix addition O(n^3).We can attempt to reformulate the equation for Cij so as to have fewer multiplication and possibly more addition .

12. Stressen has discovered a way to compute the Cij of equation (2) using only 7 multiplication and 18 addition or subtraction.
13. Strassen's formula are

P= (A11+A12)(B11+B22)

Q= (A12+A22)B11

R= A11(B12-B22)

S= A22(B21-B11)

T= (A11+A12)B22

U= (A21-A11)(B11+B12)

V= (A12-A22)(B21+B22)

C11=P+S-T+V

C!2=R+t

C21=Q+T

C22=P+R-Q+V

4.7 GREEDY METHOD

Greedy method is the most straightforward designed technique.

As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

DEFINITION:

A problem with N inputs will have some constraints .any subsets that satisfy these constraints are called a feasible solution.

A feasible solution that either maximize can minimize a given objectives function is called an optimal solution.

Control algorithm for Greedy Method:

```
Algorithm Greedy (a,n)
//a[1:n] contain the 'n' inputs
{
solution =0;//Initialise the solution.
For i=1 to n do
{
x=select(a);
if(feasible(solution,x))then
solution=union(solution,x);
}
return solution;
}
```

* The function select an input from a[] and removes it. The select input value is assigned to X.

Feasible is a Boolean value function that determines whether X can be included into the solution vector.

The function Union combines X with The solution and updates the objective function.The function Greedy describes the essential way that a greedy algorithm will once a particular problem is chosen and the function subset, feasible & union are properly implemented.

Example

Suppose we have in a country the following coins are available :

Dollars(100 cents)

Quarters(25 cents)

Dimes(10 cents)

Nickel(5 Cents)

Pennies(1 cent)

Our aim is paying a given amount to a customer using the smallest possible number of coins.

For example if we must pay 276 cents possible solution then,

→ 1 doll+7 q+ 1 pen➔9 coins

→ 2 doll +3Q +1 pen➔6 coins

→ 2 doll+7dim+1 nic +1 pen➔11 coins.

JOB SCHEDULING WITH DEAD LINES

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadlines, and it should yield a maximum profit.

Points To remember:

To complete a job, one has to process the job or a action for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.If we select a job at that time ,

→Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

→So the waiting time and the processing time should be less than or equal to the dead line of the job.

ALGORITHM:

```
Algorithm JS(d,j,n)
//The job are ordered such that p[1]>p[2]…>p[n]
//j[i] is  the ith job in the optimal solution
// Also at terminal d [ J[ i]<=d[ J {i+1],1<i<k
 {
  d[0]= J[0]=0;
J[1]=1;
```

```
K=1;
For I =1 to n do
{ // consider jobs in non increasing  order of P[I];find the position for I and check feasibility insertion
r=k;
while((d[J[r]]>d[i] )and
  (d[J[r]] =  r)do r =r-1;
if (d[J[r]]<d[I])and (d[I]>r))then
{
for q=k to (r+1) step –1 do J [q+1]=j[q]
J[r+1]=i;
K=k+1;
}
}
return k;
}
```

Example :

1. n=5 (P1,P2,…P5)=(20,15,10,5,1)
   (d1,d2….d3)=(2,2,1,3,3)

Feasible solution	***Processing Sequence***	***Value***
(1)	(1)	20
(2)	(2)	15
(3)	(3)	10
(4)	(4)	5
(5)	(5)	1
(1,2)	(2,1)	35
(1,3)	(3,1)	30
(1,4)	(1,4)	25
(1,5)	(1,5)	21
(2,3)	(3,2)	25
(2,4)	(2,4)	20

(2,5)	(2,5)	16
(1,2,3)	(3,2,1)	45
(1,2,4)	(1,2,4)	40

The Solution 13 is optimal

n=4 (P1,P2,…P4)=(100,10,15,27)

(d1,d2….d4)=(2,1,2,1)

Feasible solution	***Processing Sequence***	***Value***
(1,2)	(2,1)	110
(1,3)	(1,3)	115
(1,4)	(4,1)	127
(2,3)	(9,3)	25
(2,4)	(4,2)	37
(3,4)	(4,3)	42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

The solution 3 is optimal.

KNAPSACK PROBLEM

we are given n objects and knapsack or bag with capacity M object I has a weight Wi where I varies from 1 to N.

The problem is we have to fill the bag with the help of N objects and the resulting profit has to be maximum.

Formally the problem can be stated as

Maximize xipi subject to XiWi<=M

Where Xi is the fraction of object and it lies between 0 to 1.

There are so many ways to solve this problem, which will give many feasible solution for which we have to find the optimal solution. But in this algorithm, it will generate only one solution which is going to be feasible as well as optimal. First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios. Select an object with highest p/w

ratio and check whether its height is lesser than the capacity of the bag. If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.

Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected .in this case place a fraction of the object and come out of the loop. Whenever you selected.

ALGORITHM:

```
1.Algorityhm Greedy knapsack (m,n)
2//P[1:n] and the w[1:n]contain the profit
3.// & weight res'.of the n object ordered.
4.//such that p[i]/w[i] >=p[i+1]/W[i+1]
5.//n is the Knapsack size and x[1:n] is the solution vertex.
6.{
7.for I=1 to n do a[I]=0.0;
8.U=n;
9.For I=1 to n do
10.{
11.if (w[i]>u)then break;
13.x[i]=1.0;U=U-w[i]
14.}
15.if(i<=n)then x[i]=U/w[i];
16.}
```

Example:

Capacity=20

N=3 ,M=20

Wi=18,15,10

Pi=25,24,15

Pi/Wi=25/18=1.36,24/15=1.6,15/10=1.5

Descending Order ➔ Pi/Wi➔1.6 1.5 1.36

Pi = 24 15 25

Wi = 15 10 18

Xi = 1 5/10 0

PiXi=1*24+0.5*15➔31.5

The optimal solution is ➔31.5

X1	***X2***	***X3***	***WiXi***	***PiXi***
½	1/3	¼	16.6	24.25
1	2/5	0	20	18.2
0	2/3	1	20	31
0	1	½	20	31.5

Of these feasible solution Solution 4 yield the Max profit .As we shall soon see this solution is optimal for the given problem instance.

MINIMUM COST SPANNING TREE

Let G(V,E) be an undirected connected graph with vertices 'v' and edge 'E'.

A sub-graph t=(V,E') of the G is a Spanning tree of G iff 't' is a tree.3

The problem is to generate a graph G'= (V,E) where 'E' is the subset of E,G' is a Minimum spanning tree.

Each and every edge will contain the given non-negative length .connect all the nodes with edge present in set E' and weight has to be minimum.

NOTE:

We have to visit all the nodes.

The subset tree (i.e) any connected graph with 'N' vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph. A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

METHODS OFMINIMUM COST SPANNING TREE:

The cost of a spanning tree is the sum of cost of the edges in that trees.

There are 2 method to determine a minimum cost spanning tree are

1. Kruskal's Algorithm
2. Prom's Algorithm.

KRUSKAL'S ALGORITHM:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

- In this algorithm, a minimum cost-spanning tree 'T' is built edge by edge.
- Edge are considered for inclusion in 'T' in increasing order of their cost.

An edge is included in 'T' if it doesn't form a cycle with edge already in T.

To find the minimum cost spanning tree the edge are inserted to tree in increasing order of their cost

Algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G has 'n' vertices.
//cost[u,v]→cost of edge (u,v).t→set of edge in minimum cost spanning tree
// the first cost is returned.
{
for i=1 to n do parent[I]=-1;
I=0;mincost=0.0;
While((I<n-1)and (heap not empty)) do
{
j=find(n);
k=find(v);
if(j not equal k) than
{
```

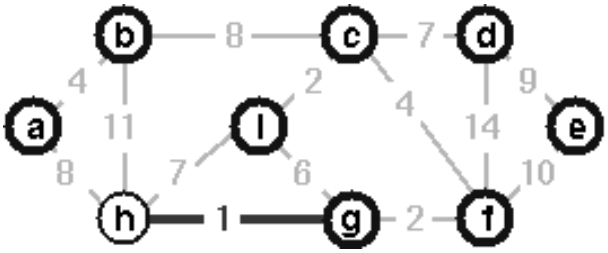
```
i=i+1
t[i,1]=u;
t[i,2]=v;
mincost=mincost+cost[u,v];
union(j,k);
    }
  }
if(i notequal n-1) then write("No spanning tree")
else return minimum cost;
}
```

Analysis

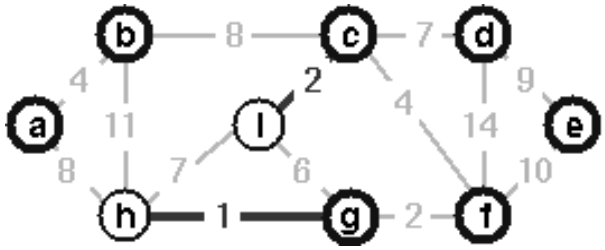
14. The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|), →where E is the edge set of G.

<u>Example: Step by Step operation of Kurskal algorithm.</u>

Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Lets choose vertex g arbitrarily.

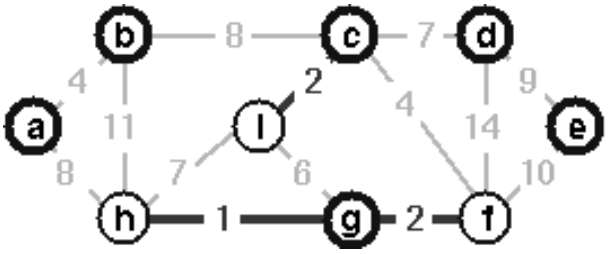


Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.

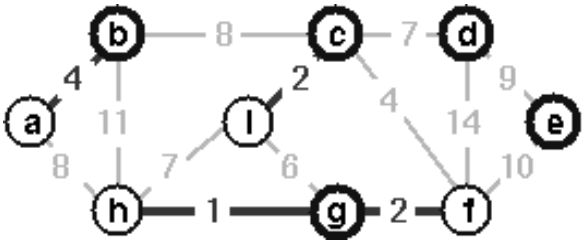


Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as

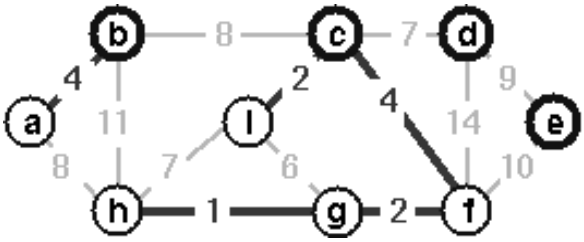
representative.



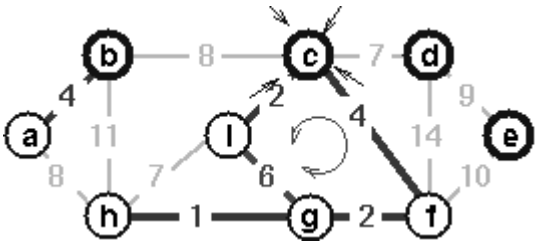
Step 4. Edge (a, b) creates a third tree.



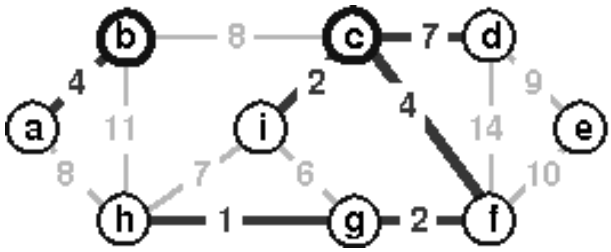
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



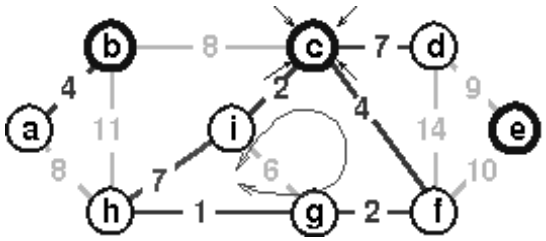
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



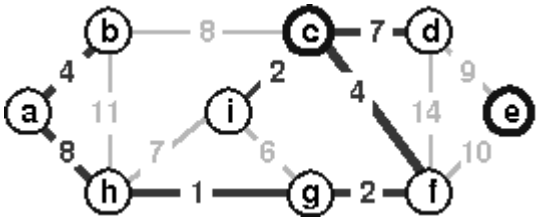
Step 7. Instead, add edge (c, d).



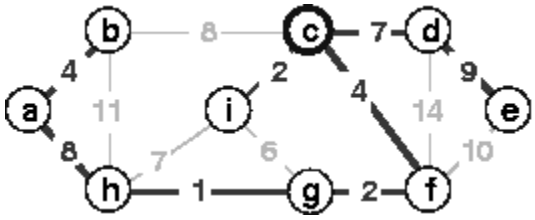
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).

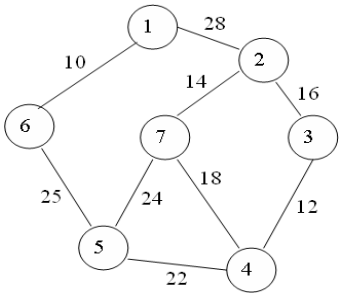


Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



PRIM'S ALGORITHM

Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached.




```
Algorithm prims(e,cost,n,t)
{
  Let (k,l) be an edge of minimum cost in E;
  Mincost :=cost[k,l];
  T[1,1]:=k; t[1,2]:=l;
  For I:=1 to n do
     If (cost[i,l]<cost[i,k]) then near[i]:=l;
     Else near[i]:=k;
    Near[k]:=near[l]:=0;
     For i:=2 to n-1 do
    {
  Let j be an index such that near[j]≠0 and
  Cost[j,near[j]] is minimum;
  T[i,1]:=j; t[i,2]:=near[j];
  Mincost:=mincost+ Cost[j,near[j]];
  Near[j]:=0;
  For k:=0 to n do
     If near((near[k]≠0) and (Cost[k,near[k]]>cost[k,j])) then
    Near[k]:=j;
 }
 Return mincost;
}
```

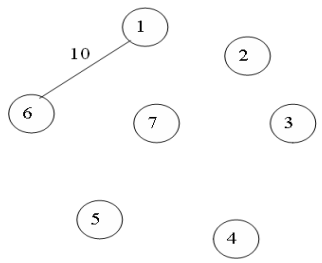
15. The prims algorithm will start with a tree that includes only a minimum cost edge of G.
16. Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a

vertex included

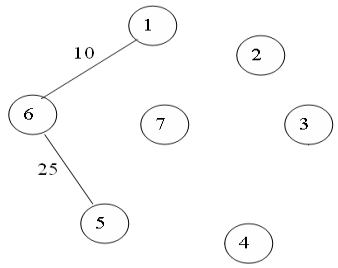
in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.

17. The working of prims will be explained by following diagram

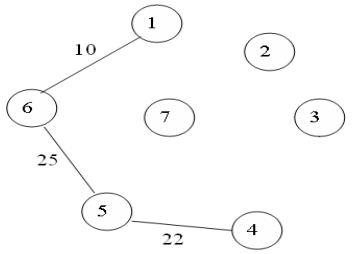
Step 1:



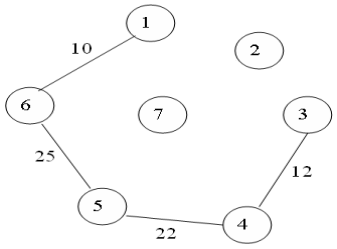
Step 2:



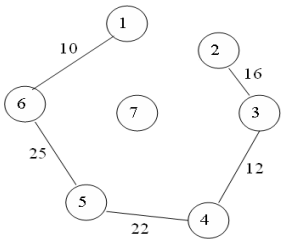
Step 3:



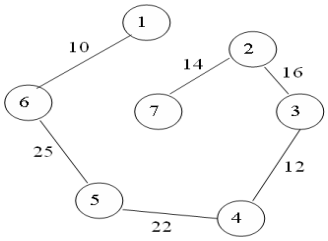
Step 4:



Step 5:



Step 6:



SINGLE-SOURCE SHORTEST PATH:

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B

would be interested in answers to the following questions:

- Is there a path from A to B?
- If there is more than one path from A to B? Which is the shortest path?

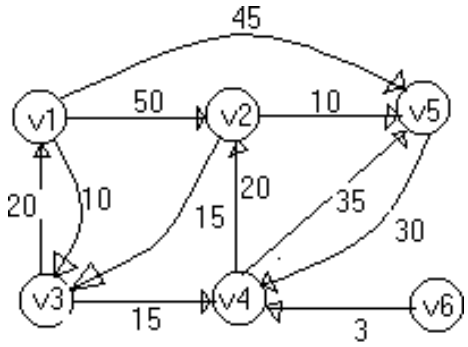


Fig 7.1

The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of fig 7-1. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

- v1 v2 = 50 units
- v1 v3 v4 v2 = 10+15+20=45 units
- v1 v5 v4 v2 = 45+30+20= 95 units
- v1 v3 v4 v5 v4 v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1 v3 v4 v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1 v2 that costs 50 units. One can also

notice that, it is not possible to travel to v6 from any other node.

To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.

A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows,

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for fig 7.1 is given below

	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf
V3	20	Inf	-	15	inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1. In the above table the minimum in row v1 is 10.

Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.

Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain only row v1. The second row is computed by adding 10 to all values of row v3.

The resulting matrix is

	V2	V4	V5	V6
V1 Vw	50	Inf	45	Inf
V1 V3 Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf

Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.

The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1 Vw	50	45	Inf
V1 V3 V4 Vw	25+20	25+35	25+inf
Minimum	45	45	inf

	V5	V6
V1 Vw	45	Inf
V1 V3 V4 V2 Vw	45+10	45+inf
Minimum	45	Inf

	V6
V1 Vw	Inf
V1 V3 V4 V2 V5 Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1 V3 V4 V2 V5.

PART-A (2 Marks)

1. What is meant by feasible solution?

Ans. Given n inputs and we are required to form a subset such that it satisfies some given constraints then such a subset is called feasible solution.

2. Write any two characteristics of Greedy Algorithm?

Ans. To solve a problem in an optimal way construct the solution from given set of candidates. As the algorithm proceeds, two other sets get accumulated among this one set contains the candidates that have been already considered and chosen while the other set contains the candidates that have been considered but rejected.

3.What is greedy method?

Ans. Greedy method is the most important design technique, which makes a choice that looks best at that moment. A given 'n' inputs are required us to obtain a subset that satisfies some constraints that is the feasible solution. A greedy method.

4. Define optimal solution?

Ans. A feasible solution either maximizes or minimizes the given objective function is called as optimal solution.

5. What is Knapsack problem?

Ans. A bag or knapsack is given capacity n and n objects are given. Each object has weight wi and profit pi .Fraction of object is considered as xi (i.e) 0<=xi<=1 .If fraction is 1 then entire object is put into sack. When we place this fraction into the sack we get wi xi and pixi.

6. Define optimal solution for Job sequencing with deadlines.

Ans. Feasible solution with maximum profit is optimal solution for Job sequencing with deadlines.

7. Define dynamic programming.

Ans. Dynamic programming is an algorithm design method that can be used when a solution to the problem is viewed as the result of sequence of decisions.

8. What are the drawbacks of dynamic programming?

Ans. Time and space requirements are high, since storage is needed for all level.

Optimality should be checked at all levels.

9. Define principle of optimality.

Ans. It states that an optimal sequence of decisions has the property that whenever the initial stage or decisions must constitute an optimal sequence with regard to stage resulting from the first decision.

10. Define Purging rule.

Ans. If Si contains two pairs (Pj ,Wj) and (Pk , Wk) with the property that Pj ≤ Pk and Wj ≥ Wk , then the pair (Pj ,Wj) can be discarded since, it's a duplicate tuple. Purging rule is also known as "Dominance rule".

PART-B (10 Marks)

1. Explain the solution to the problem of job sequencing with deadlines by Greedy technique.
2. Explain how travelling salesman problem is solved by using dynamic programming with an example.
3. Describe Kanpsack problem. Find an optimal solution for the Greedy Knapsack instance for n=3, m=6, profits are (P1, P2, P3)=(1, 2, 5) weights are (w1, w2, w3)=(2,3,4).
4. (a)Solve for Knapsack problem using Greedy method for the instance n = 7, profits (p1...p7) = (10,5,15,7,6,18,3) and weights (W1...W7) = (2,3,5,7,1,4,1) and capacity of sack is m = 15.

   (b) Explain the problem Job sequencing with Deadline and solve for n = 5, (P1...P5) = (20, 15, 10, 5, 1) and Deadlines (D1...D5) = (2, 2, 1, 3, 3).
5. Write the merge sort algorithm. Find out the best, worst and average cases of this algorithm. Sort the following numbers using merge sort: 10, 12, 1, 5, 18, 28, 38, 39, 2, 4, 7
6. Explain quick sort algorithm with an example. Derive its time complexity.
7. Mention the advantage of Stressan's matrix multiplication. Explain how the Stressan's matrix multiplication works.
8. Write an algorithm for finding the Maximum and Minimum values using Divide and Conquer method and solve for: 22, 13, -5, -8, 15, 60, 17, 31, 47.
9. Explain Binary search algorithm and derive its time complexity.

UNIT –V

Dynamic Programming: *General method, applications- 0/1 knapsack problem, All pairs shortest path problem, Travelling salesperson problem, Reliability design.*

Backtracking*: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.*

Introduction to NP-Hard and NP-Complete problems*: Basic Concepts*

5.1 DYNAMIC PROGRAMING GENERAL METHOD:

1. The idea of dynamic programming is thus quit simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
2. Divide and conquer is a top-down method.
3. When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub-instances as the algorithm progresses.
4. Dynamic programming on the other hand is a bottom-up technique.
5. We usually start with the smallest and hence the simplest sub- instances.
6. By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances.
7. The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated.
8. In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

APPLICATIONS:

0/1 KNAPSACK PROBLEM:

1. This problem is similar to ordinary knapsack problem but we may not take a fraction of an object.

2. We are given ‘ N ‘ object with weight W_i and profits P_i where I varies from l to N and also a knapsack with capacity ‘ M ‘.

3. The problem is, we have to fill the bag with the help of ‘ N ‘ objects and the resulting profit has to be maximum.

4. Formally, the problem can be started as, maximize $\sum_{i=l}^{n} X_i P_i$

$$\text{subject to} \sum_{i=l}^{n} X_i W_i \text{ L } M$$

5. Where X_i are constraints on the solution $X_i \in \{0,1\}$. (u) X_i is required to be 0 or 1. if the object is selected then the unit in 1. if the object is rejected than the unit is 0. That is why it is called as 0/1, knapsack problem.

6. To solve the problem by dynamic programming we up a table T[1…N, 0…M] (ic) the size is N. where 'N' is the no. of objects and column starts with 'O' to capacity (ic) 'M'.

7. In the table T[i,j] will be the maximum valve of the objects i varies from 1 to n and j varies from O to M.

RULES TO FILL THE TABLE:-

1. If i=l and j < w(i) then T(i,j) =o, (ic) o pre is filled in the table.

2. If i=l and j ≥ w (i) then T (i,j) = p(i), the cell is filled with the profit p[i], since only one object can be selected to the maximum.

3. If i>l and j < w(i) then T(i,l) = T (i-l,j) the cell is filled the profit of previous object since it is not possible with the current object.

4. If i>l and j ≥ w(i) then T (i,j) = {f(i) +T(i-l,j-w(i)),. since only 'l' unit can be selected to the maximum. If is the current profit + profit of the previous object to fill the remaining capacity of the bag.

5. After the table is generated, it will give details the profit.

ES TO GET THE COMBINATION OF OBJECT:

- Start with the last position of i and j, T[i,j], if T[i,j] = T[i-l,j] then no object of 'i' is required so move up to T[i-l,j].
- After moved, we have to check if, T[i,j]=T[i-l,j-w(i)]+ p[I], if it is equal then one unit of object 'i' is selected and move up to the position T[i-l,j-w(i)]

- Repeat the same process until we reach T[i,o], then there will be nothing to fill the bag stop the process.
- Time is 0(nw) is necessary to construct the table T.
- Consider a Example,

M = 6,
N = 3
$W_1 = 2$, $W_2 = 3$, $W_3 = 4$
$P_1 = 1$, $P_2 = 2$, $P_3 = 5$

i ⟶ 1 to N
j ⟶ 0 to 6

i=l, j=o (ic) i=l & j < w(i)

o<2 ⟶ T_1,o =0

i=l, j=l (ic) i=l & j < w(i)

l<2 ⟶ T_1,1 =0 (Here j is equal to w(i) ⟶ P(i)

i=l, j=2
2 o,= T_1,2 = l.

i=l, j=3
3>2,= T_1,3 = l.

i=l, j=4
4>2,= T_1,4 = l.

i=l, j=5
5>2,= T_1,5 = l.

i=l, j=6
6>2,= T_1,6 = l.

=> i=2, j=o (ic) i>l,j<w(i)
o<3= T(2,0) = T(i-l,j) = T(2)
T 2,0 =0

i=2, j=1
l<3= T(2,1) = T(i-l)
T 2,1 =0

ALL PAIR SHORTEST PATH

Let G=<N,A> be a directed graph 'N' is a set of nodes and 'A' is the set of edges.

1. Each edge has an associated non-negative length.
2. We want to calculate the length of the shortest path between each pair of nodes.
3. Suppose the nodes of G are numbered from 1 to n, so N={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i & j, and L(i,j)=infinity, if the edge (i,j) does not exist.
4. The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.
5. First, create a cost adjacency matrix for the given graph.
6. Copy the above matrix-to-matrix D, which will give the direct distance between nodes.
7. We have to perform N iteration after iteration k.the matrix D will give you the distance between nodes with only (1,2...,k)as intermediate nodes.
8. At the iteration k, we have to check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k.

COST ADJACENCY MATRIX:

$$D0 = L = \begin{vmatrix} 0 & 5 & \propto & \propto \\ 50 & 0 & 15 & 5 \\ 30 & \propto & 0 & 15 \\ 15 & \propto & 5 & 0 \end{vmatrix}$$

$$\begin{vmatrix} 1 & 7 & 5 & \propto & \propto \\ 2 & 7 & \propto & \propto & 2 \\ 3 & \propto & 3 & \propto & \propto \\ 4 & 4 & \propto & 1 & \propto \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & - & - \\ 21 & - & - & 24 \\ - & 32 & - & - \\ 41 & - & 43 & - \end{vmatrix}$$

vertex 1:

$$\begin{vmatrix} 7 & 5 & \propto & \propto \\ 7 & \mathbf{12} & \propto & 2 \\ \propto & 3 & \propto & \propto \\ 4 & \mathbf{9} & 1 & \propto \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & - & - \\ 21 & \mathbf{212} & - & 24 \\ - & 32 & - & - \\ 41 & \mathbf{412} & 43 & - \end{vmatrix}$$

vertex 2:

$$\begin{vmatrix} 7 & 5 & \propto & \mathbf{7} \\ 7 & 12 & \propto & 2 \\ \mathbf{10} & 3 & \propto & \mathbf{5} \\ 4 & 9 & 1 & \mathbf{11} \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & - & \mathbf{124} \\ 21 & 212 & - & 24 \\ \mathbf{321} & 32 & - & \mathbf{324} \\ 41 & 412 & 43 & \mathbf{4124} \end{vmatrix}$$

vertex 3:

$$\begin{vmatrix} 7 & 5 & \propto & 7 \\ 7 & 12 & \propto & 2 \\ 10 & 3 & \propto & 5 \\ 4 & \mathbf{4} & 1 & \mathbf{6} \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & - & 124 \\ 21 & 212 & - & 24 \\ 321 & 32 & - & 324 \\ 41 & \mathbf{432} & 4 & \mathbf{4324} \end{vmatrix}$$

vertex 4:

$$\begin{vmatrix} 7 & 5 & \mathbf{8} & 7 \\ \mathbf{6} & \mathbf{6} & \mathbf{3} & 2 \\ \mathbf{9} & 3 & \mathbf{6} & 5 \\ 4 & 4 & 1 & 6 \end{vmatrix} \quad \begin{vmatrix} 11 & 12 & \mathbf{1243} & 124 \\ \mathbf{241} & \mathbf{2432} & \mathbf{243} & 24 \\ \mathbf{3241} & 32 & \mathbf{3243} & 324 \\ 41 & 432 & 43 & 4324 \end{vmatrix}$$

1. At 0th iteration it nil give you the direct distances between any 2 nodes

$$D0 = \begin{vmatrix} 0 & 5 & \propto & \propto \\ 50 & 0 & 15 & 5 \\ 30 & \propto & 0 & 15 \\ 15 & \propto & 5 & 0 \end{vmatrix}$$

2. At 1st iteration we have to check the each pair(i,j) whether there is a path through node 1.if so we have to check whether it is minimum than the previous value and if I is so than the distance through 1 is the value of d1(i,j).at the same time we have to solve the intermediate node in the matrix position p(i,j).

$$D1 = \begin{vmatrix} 0 & 5 & \propto & \propto \\ 50 & 0 & 15 & 5 \\ 30 & \mathbf{35} & 0 & 15 \\ 15 & \mathbf{20} & 5 & 0 \end{vmatrix} \quad \begin{matrix} p[3,2]= 1 \\ p[4,2]= 1 \end{matrix}$$

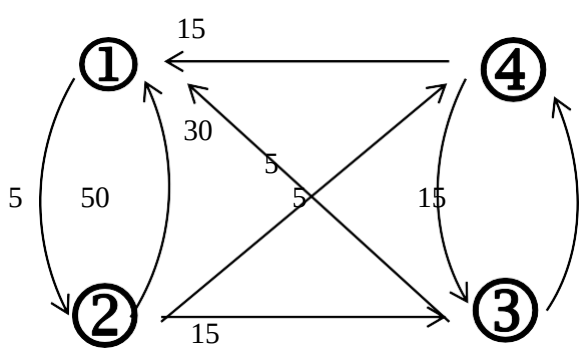


Fig: floyd's algorithm and work

3. likewise we have to find the value for N iteration (ie) for N nodes.

$$D2= \begin{vmatrix} 0 & 5 & \mathbf{20} & \mathbf{10} \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{vmatrix} \qquad \begin{matrix} P[1,3] = 2 \\ P[1,4] = 2 \end{matrix}$$

$$D3= \begin{vmatrix} 0 & 5 & 20 & 10 \\ \mathbf{45} & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{vmatrix} \qquad P[2,1]=3$$

$$D4= \begin{vmatrix} 0 & 5 & \mathbf{15} & 10 \\ 20 & 0 & \mathbf{10} & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{vmatrix} \qquad \begin{matrix} P[1,3]=4 \\ P[2,3]=4 \end{matrix}$$

4. D4 will give the shortest distance between any pair of nodes.
5. If you want the exact path then we have to refer the matrix p.The matrix will be,

$$P= \begin{vmatrix} 0 & 0 & 4 & 2 \\ 3 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} \qquad 0 \longrightarrow \text{direct path}$$

- Since,p[1,3]=4,the shortest path from 1 to3 passes through 4.
- Looking now at p[1,4]&p[4,3] we discover that between 1 & 4, we have to go to node 2 but that from 4 to 3 we proceed directly.
- Finally we see the trips from 1 to 2, & from 2 to 4, are also direct.
- The shortest path from 1 to 3 is 1,2,4,3.

ALGORITHM :

Function Floyd (L[1..r,1..r]):array[1..n,1..n]

array D[1..n,1..n]

D = L

For k = 1 to n do

For i = 1 to n do
For j = 1 to n do
D [i , j] = min (D[i, j], D[i, k] + D[k, j]

Return D

ANALYSIS:
This algorithm takes a time of θ (n^3)

MULTISTAGE GRAPH

1. A multistage graph G = (V,E) is a directed graph in which the vertices are portioned into K > = 2 disjoint sets Vi, 1 <= i<= k.
2. In addition, if < u,v > is an edge in E, then u < = Vi and V ∑ Vi+1 for some i, 1<= i < k.
   If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
3. Let 's' and 't' be the source and destination respectively.
4. The cost of a path from source (s) to destination (t) is the sum of the costs of the edger on the path.
5. The *MULTISTAGE GRAPH* problem is to find a minimum cost path from 's' to 't'.
6. Each set Vi defines a stage in the graph. Every path from 's' to 't' starts in stage-1, goes to stage-2 then to stage-3, then to stage-4, and so on, and terminates in stage-k.

This *MULISTAGE GRAPH* problem can be solved in 2 ways.

- Forward Method.
- Backward Method.

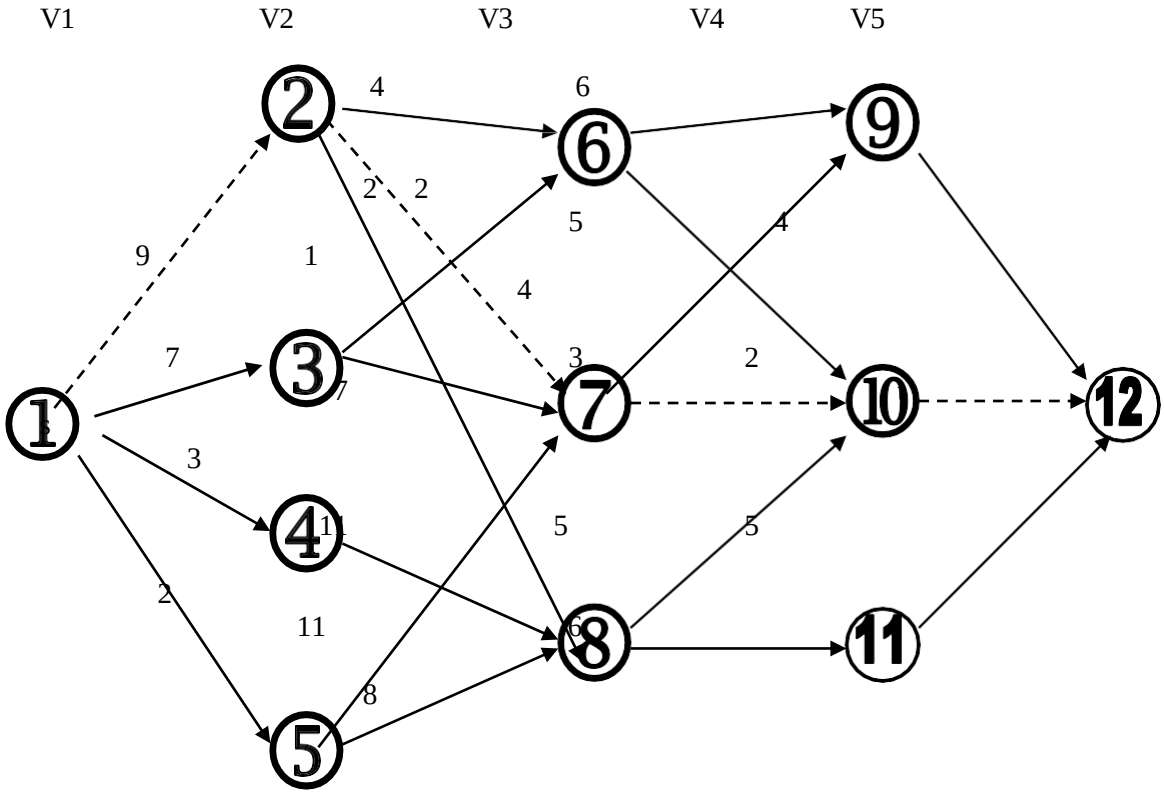
FORWARD METHOD

Assume that there are 'k' stages in a graph.

In this *FORWARD* approach, we will find out the cost of each and every node starling from the 'k' th stage to the 1st stage.

We will find out the path (i.e.) minimum cost path from source to the destination (ie) [Stage-1 to Stage-k].

PROCEDURE:



1. Maintain a cost matrix cost (n) which stores the distance from any vertex to the destination.
2. If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path, will be stored in the distance array 'D'.
3. In this way we will find out the minimum cost path from each and every vertex.
4. Finally cost(1) will give the shortest distance from source to destination.
5. For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the Destination.
6. For a 'k' stage graph, there will be 'k' vertex in the path.
7. In the above graph V1…V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

STEPS: -	**DESTINATION, D**
Cost (12)=0	⟶ D (12)=0
Cost (11)=5	⟶ D (11)=12

Cost (10)=2 $\longrightarrow$ D (10)=12

Cost (9)=4 $\longrightarrow$ D (9)=12

1. For forward approach,

$$\text{Cost}(i,j) = \min_{\substack{l \in V_{i+1} \\ (j,l) \in E}} \{C(j,l) + \text{Cost}(i+1,l)\}$$

Cost(8) = min {C (8,10) + Cost (10), C (8,11) + Cost (11) }

= min (5 + 2, 6 + 5)

= min (7,11)

= 7

cost(8) =7 =>D(8)=10

cost(7) = min(c (7,9)+ cost(9),c (7,10)+ cost(10))

(4+4,3+2)

= min(8,5)

= 5

cost(7) = 5 =>D(7) = 10

cost(6) = min (c (6,9) + cost(9),c (6,10) +cost(10))

= min(6+4 , 5 +2)

= min(10,7)

= 7

cost(6) = 7 =>D(6) = 10

cost(5) = min (c (5,7) + cost(7),c (5,8) +cost(8))

= min(11+5 , 8 +7)

= min(16,15)

= 15

cost(5) = 15 =>D(5) = 18

cost(4) = min (c (4,8) + cost(8))

= min(11+7)

= 18

cost(4) = 18 =>D(4) = 8

cost(3) = min (c (3,6) + cost(6),c (3,7) +cost(7))

= min(2+7 , 7 +5)

= min(9,12)

= 9

cost(3) = 9 =>D(3) = 6

cost(2) = min (c (2,6) + cost(6),c (2,7) +cost(7) ,c (2,8) +cost(8))

= min(4+7 , 2+5 , 1+7)

= min(11,7,8)

= 7

cost(2) = 7 =>D(2) = 7

cost(1) = min (c (1,2)+cost(2) ,c (1,3)+cost(3) ,c (1,4)+cost(4) ,c(1,5)+cost(5))

= min(9+7 , 7 +9 , 3+18 , 2+15)

= min(16,16,21,17)

= 16

cost(1) = 16 =>D(1) = 2

.....→ The path through which you have to find the shortest distance.

(i.e.) ① → ② → ⑦ → ⑩ → ⑫

Start from vertex - 2

D (1) = 2

D (2) = 7

D (7) = 10

D (10) = 12

So, the minimum –cost path is,

① —9→ ② —2→ ⑦ —3→ ⑩ —2→ ⑫

∴ The cost is 9+2+3+2+=16

ALGORITHM: FORWARD METHOD

Algorithm FGraph (G,k,n,p)

// The I/p is a k-stage graph G=(V,E) with 'n' vertex.

// Indexed in order of stages E is a set of edges.

// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.

{

```
    cost[n]=0.0;
     for j=n-1 to 1 step-1 do
  {
     //compute cost[j],
    //  let 'r' be the vertex such that <j,r> is an edge of 'G' &
   //  c[j,r]+cost[r] is minimum.

   cost[j] = c[j+r] + cost[r];
    d[j] =r;
   }
  // find a minimum cost path.

       P[1]=1;
       P[k]=n;
       For j=2 to k-1 do
       P[j]=d[p[j-1]];
}
```

ANALYSIS:

The time complexity of this forward method is O(V + E)

BACKWARD METHOD

if there one 'K' stages in a graph using back ward approach. we will find out the cost of each & every vertex starting from 1st
stage to the kth stage.

We will find out the minimum cost path from destination to source (ie)[from stage k to stage 1]

PROCEDURE:

- It is similar to forward approach, but differs only in two or three ways.
- Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
- Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
- To find out the path star from vertex 'k', then the distance array D (k) will give the minimum

cost neighbor vertex which in turn gives the next nearest neighbor vertex and proceed till we reach the destination.

STEP:

Cost(1) = 0 => D(1)=0

Cost(2) = 9 => D(2)=1

Cost(3) = 7 => D(3)=1

Cost(4) = 3 => D(4)=1

Cost(5) = 2 => D(5)=1

Cost(6) =min(c (2,6) + cost(2),c (3,6) + cost(3))

=min(13,9)

cost(6) = 9 =>D(6)=3

Cost(7) =min(c (3,7) + cost(3),c (5,7) + cost(5) ,c (2,7) + cost(2))

=min(14,13,11)

cost(7) = 11 =>D(7)=2

Cost(8) =min(c (2,8) + cost(2),c (4,8) + cost(4) ,c (5,8) +cost(5))

=min(10,14,10)

cost(8) = 10 =>D(8)=2

Cost(9) =min(c (6,9) + cost(6),c (7,9) + cost(7))

=min(15,15)

cost(9) = 15 =>D(9)=6

Cost(10)=min(c(6,10)+cost(6),c(7,10)+cost(7)),c (8,10)+cost(8)) =min(14,14,15)

cost(10)= 14 =>D(10)=6

Cost(11) =min(c (8,11) + cost(8))

cost(11) = 16 =>D(11)=8

cost(12)=min(c(9,12)+cost(9),c(10,12)+cost(10),c(11,12)+cost(11))

=min(19,16,21)

cost(12) = 16 =>D(12)=10

PATH:

Start from vertex-12

D(12) = 10

D(10) = 6

D(6) = 3

D(3) = 1

So the minimum cost path is,

$1 \xrightarrow{7} 3 \xrightarrow{2} 6 \xrightarrow{5} 10 \xrightarrow{2} 12$

The cost is 16.

ALGORITHM : BACKWARD METHOD

Algorithm BGraph (G,k,n,p)

```
//  The I/p is a k-stage graph G=(V,E) with 'n' vertex.
//  Indexed in order of stages E is a set of edges.
// and c[i,J] is the cost of<i,j>,p[1:k] is a minimum cost path.
{
     bcost[1]=0.0;
      for j=2 to n do
   {
      //compute bcost[j],
     //  let 'r' be the vertex such that <r,j> is an edge of 'G' &
    //  bcost[r]+c[r,j] is minimum.

    bcost[j] = bcost[r] + c[r,j];
     d[j] =r;
   }
  // find a minimum cost path.

        P[1]=1;
        P[k]=n;
        For j= k-1 to 2 do
        P[j]=d[p[j+1]];
}
```

5.4.TRAVELLING SALESMAN PROBLEM

- Let G(V,E) be a directed graph with edge cost c_{ij} is defined such that c_{ij} >0 for all i and j and c_{ij} =∝ ,if <i,j> ∉ E.

Let V ⊨n and assume n>1.

- The traveling salesman problem is to find a tour of minimum cost.
- A tour of G is a directed cycle that include every vertex in V.
- The cost of the tour is the sum of cost of the edges on the tour.
- The tour is the shortest path that starts and ends at the same vertex (ie) 1.

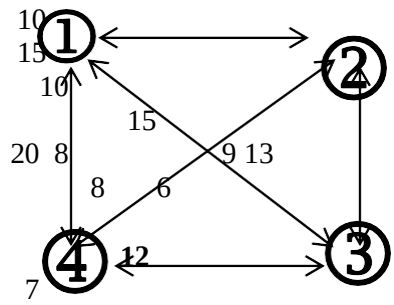
APPLICATION :

- Suppose we have to route a postal van to pick up mail from the mail boxes located at 'n' different sites.
- An n+1 vertex graph can be used to represent the situation.
- One vertex represent the post office from which the postal van starts and return.
- Edge <i,j> is assigned a cost equal to the distance from site 'i' to site 'j'.
- the route taken by the postal van is a tour and we are finding a tour of minimum length.
- every tour consists of an edge <1,k> for some k ∈ V-{} and a path from vertex k to vertex 1.
- the path from vertex k to vertex 1 goes through each vertex in V-{1,k} exactly once.
- the function which is used to find the path is

  $g(1,V-\{1\}) = \min\{ c_{ij} + g(j,s-\{j\})\}$
- g(i,s) be the length of a shortest path starting at vertex i, going through all vertices in S,and terminating at vertex 1.
- the function g(1,v-{1}) is the length of an optimal tour.

STEPS TO FIND THE PATH:

1. Find $g(i,\Phi) = c_{i1}$, 1<=i<n, hence we can use equation(2) to obtain g(i,s) for all s to size 1.
2. That we have to start with s=1,(ie) there will be only one vertex in set 's'.
3. Then s=2, and we have to proceed until |s| <n-1.
4. for example consider the graph.



Cost matrix

$$\begin{vmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{vmatrix}$$

g(i,s) → set of nodes/vertex have to visited.

↓

starting position

$g(i,s) = \min\{c_{ij} + g(j, s-\{j\})$

STEP 1:

$g(1,\{2,3,4\}) = \min\{c_{12}+g(2\{3,4\}), c_{13}+g(3,\{2,4\}), c_{14}+g(4,\{2,3\})\}$

$\min\{10+25, 15+25, 20+23\}$

$\min\{35,35,43\}$

$=35$

STEP 2:

$g(2,\{3,4\}) = \min\{c_{23}+g(3\{4\}), c_{24}+g(4,\{3\})\}$

$\min\{9+20, 10+15\}$

$\min\{29,25\}$

$=25$

$g(3,\{2,4\}) = \min\{c_{32}+g(2\{4\}), c_{34}+g(4,\{2\})\}$

$\min\{13+18, 12+13\}$

$\min\{31,25\}$

$=25$

$g(4,\{2,3\}) = \min\{c_{42}+g(2\{3\}), c_{43}+g(3,\{2\})\}$

$\min\{8+15, 9+18\}$

$\min\{23,27\}$

$=23$

STEP 3:

1. $g(3,\{4\}) = \min\{c_{34} + g\{4,\Phi\}\}$

   $12+8 = 20$

2. $g(4,\{3\}) = \min\{c_{43} + g\{3,\Phi\}\}$

   $9+6 = 15$

3. $g(2,\{4\}) = \min\{c_{24} + g\{4,\Phi\}\}$

   $10+8 = 18$

4. $g(4,\{2\}) = \min\{c_{42} + g\{2,\Phi\}\}$

   $8+5 = 13$

5. $g(2,\{3\}) = \min\{c_{23} + g\{3,\Phi\}\}$

   $9+6=15$

6. $g(3,\{2\}) = \min\{c_{32} + g\{2,\Phi\}\}$

   $13+5=18$

STEP 4:

$g\{4,\Phi\} = c_{41} = 8$

$g\{3,\Phi\} = c_{31} = 6$

$g\{2,\Phi\} = c_{21} = 5$

$|s| = 0.$

i =1 to n.

$g(1,\Phi) = c_{11} \Rightarrow 0$

$g(2,\Phi) = c_{21} \Rightarrow 5$

$g(3,\Phi) = c_{31} \Rightarrow 6$

$g(4,\Phi) = c_{41} \Rightarrow 8$

$|s| = 1$

i =2 to 4

$g(2,\{3\}) = c_{23} + g(3,\Phi)$

$= 9+6 = 15$

$g(2,\{4\}) = c_{24} + g(4,\Phi)$

$= 10+8 = 18$

$g(3,\{2\}) = c_{32} + g(2,\Phi)$

$= 13+5 = 18$

$g(3,\{4\}) = c_{34} + g(4,\Phi)$

$= 12+8 = 20$

$g(4,\{2\}) = c_{42} + g(2,\Phi)$

$= 8+5 = 13$

$g(4,\{3\}) = c_{43} + g(3,\Phi)$

$= 9+6 = 15$

$|s| = 2$

$i \neq 1$, $1 \in s$ and $i \in s$.

$g(2,\{3,4\}) = \min\{c_{23}+g(3\{4\}),c_{24}+g(4,\{3\})\}$

$\min\{9+20,10+15\}$

$\min\{29,25\}$

$=25$

$g(3,\{2,4\}) = \min\{c_{32}+g(2\{4\}),c_{34}+g(4,\{2\})\}$

$\min\{13+18,12+13\}$

$\min\{31,25\}$

$=25$

$g(4,\{2,3\}) = \min\{c_{42}+g(2\{3\}),c_{43}+g(3,\{2\})\}$

$\min\{8+15,9+18\}$

$\min\{23,27\}$

$=23$

$|s| = 3$

$g(1,\{2,3,4\})=\min\{c_{12}+g(2\{3,4\}),c_{13}+g(3,\{2,4\}),c_{14}+g(4,\{2,3\})\}$

$\min\{10+25,15+25,20+23\}$

$\min\{35,35,43\}$

$=35$

optimal cost is 35

the shortest path is,

g(1,{2,3,4}) = c_{12} + g(2,{3,4}) => 1->2

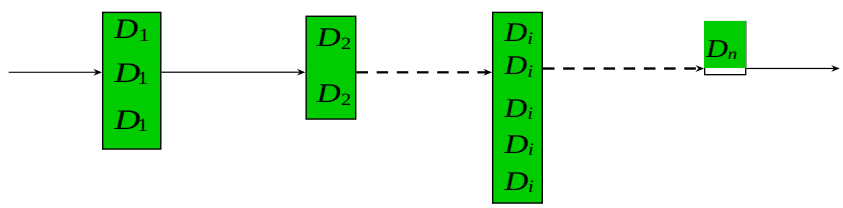
g(2,{3,4}) = c_{24} + g(4,{3}) => 1->2->4

g(4,{3}) = c_{43} + g(3{Φ}) => 1->2->4->3->1

so the optimal tour is **1 → 2 → 4 →3 → 1**

Reliability Design

- Input:
  - A system composed of several devices in serial
  - Each device (*D*) has a fixed reliability rating (*r*)
  - Multiple copies of the same device can be used in parallel to increase reliability
- Output:
  - A system with highest reliability rating subject to a cost constraint



m_i copies of devices D_i at stage i, r_i : *say*, 90%

with a reliability rating of $\Phi_i = 1-(1-r_i)^{m_i}$ ← Connected in parallel, At least one should work

$\max \prod_{1\le i\le n} \Phi_i(m_i)$ ← Connected in series, All of them have to work

subject to $\sum_{1\le i\le n} c_i m_i \le C$ and $m_i \ge 1, 1 \le i \le n$

- **Greedy method may not be applicable**
  - **Strategy to maximize reliability: Buy more less reliable units (Costs may be high)**
  - **Strategy to minimize cost: Buy more less expensive units (Reliability may not improve significantly)**
- **Divide-and-Conquer may fail**

Comparison

- A knapsack of capacity C
- Objects of size *ci* and profit *pi*
- Fill up the knapsack with 0 or 1 copy of *i*
- Maximize profit

- Total expenditure of C
- Stages of cost *ci* and reliability *ri*
- Construct a system with 1 or more copies of *i*
- Maximize reliability

$$u_i = 1 + \left\lfloor \frac{C - \sum_{j=1}^{n} c_j}{c_i} \right\rfloor$$

- How to build solutions recursively?
  - One stage and one device at a time
- How does principle of optimality apply?

$m_1 m_2 \cdots m_i \cdots m_n$

$(1, x \cdots x \cdots x) : (x \cdots x \cdots x)$ must be optimal for $C - c_1$

$(2, y \cdots y \cdots y) : (y \cdots y \cdots y)$ must be optimal for $C - 2 \times c_1$

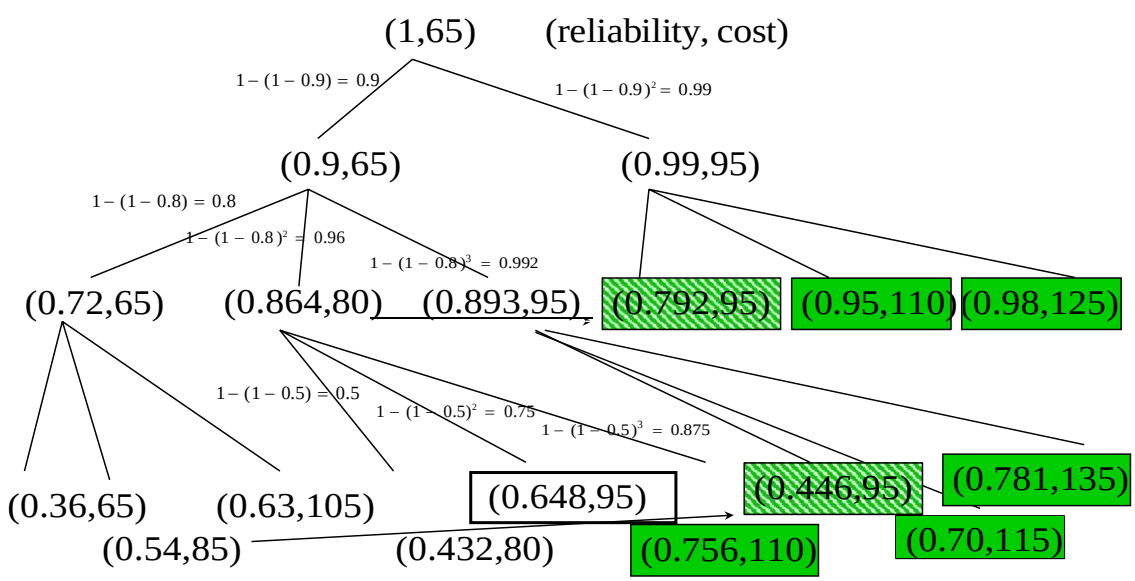
$\cdots$

$(u_1, y \cdots y \cdots y) : (y \cdots y \cdots y)$ must be optimal for $C - u_1 \times c_1$

- How to identify sub-optimal solutions?
  - if two solutions: (*m1*,...,*mi*,x,...,x), and (*n1*,...,*ni*,x,..., x) are such that one achieves higher reliability with a smaller cost, then the other cannot be optimal

- How to build table?

r=(0.9,0.8,0.5), c=(30,15,20),C=105



BACKTRACKING

- It is one of the most general algorithm design techniques.
- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.
- The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).
- The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (m_{i+1} ……m_n) possible test vectors may be ignored entirely.
- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
- These constraints are classified as:

  i) Explicit constraints.

  ii) Implicit constraints.

- Explicit constraints:

  Explicit constraints are rules that restrict each Xi to take values only from a given set.

  Some examples are,

$Xi \geq 0$ or Si = {all non-negative real nos.}

Xi =0 or 1 or Si={0,1}.

$Li \leq Xi \leq Ui$ or Si= {a: $Li \leq a \leq Ui$}

- All tupules that satisfy the explicit constraint define a possible solution space for I.
- Implicit constraints:

  The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

Algorithm IBacktracking (n)

// This schema describes the backtracking procedure .All solutions are generated in X[1:n]

//and printed as soon as they are determined.

```
{
  k=1;
  While (k ≠ 0) do
```

```
    {
      if (there remains all untried
      X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
      {
        if(X[1],……X[k] )is the path to the answer node)
       Then write(X[1:k]);
       k=k+1;              //consider the next step.
    }
  else k=k-1;                  //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.
- T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.
- B_k(X[1]……X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.

Applications:

- N-Queens problem.
- Sum of subsets.
- Graph coloring.
- Hamiltonian cycle.

N-Queens problem:

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th row where I th queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.

- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.
- Also, every element on the same diagonal that runs from lower right to upper left has the same value.
- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

STEPS TO GENERATE THE SOLUTION:

- Initialize x array to zero and start by placing the first queen in k=1 in the first row.
- To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Of queens.
- If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have to check whether there is any queen in the same column or diagonal.
- For this considers the previous position, which had already, been found out. Check whether X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
- If any one of the conditions is true then return false indicating that k th queen can't be placed in position X (k).
- For not possible condition increment X (k) value by one and precede d until the position is found.
- If the position X (k) $\leq$ n and k=n then the solution is generated completely.
- If k<n, then increment the 'k' value and find position of the next queen.
- If the position X (k)>n then k th queen cannot be placed as the size of the matrix is 'N*N'.
- So decrement the 'k' value by one i.e. we have to back track and after the position of the previous queen.

Algorithm:

Algorithm place (k,I)

//return true if a queen can be placed in k th row and I th column. otherwise it returns //

//false .X[] is a global array whose first k-1 values have been set. Abs® returns the //absolute value of r.

```
{
  For j=1 to k-1 do
    If ((X [j]=I)            //two in same column.
```

```
    Or (abs (X [j]-I)=Abs (j-k)))
 Then return false;
 Return true;
 }
```

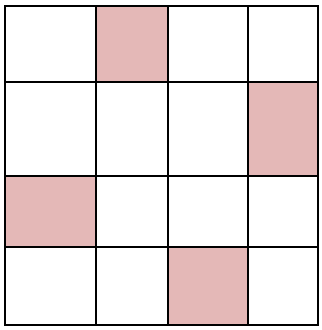
Algorithm Nqueen (k,n)

```
//using backtracking it prints all possible positions of n queens in 'n*n' chessboard. So
 //that they are non-tracking.
{
    For I=1 to n do
 {
   If place (k,I) then
     {
       X [k]=I;
        If (k=n) then write (X [1:n]);
         Else nquenns(k+1,n)    ;
      }
   }
}
```

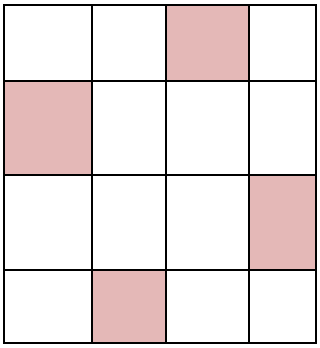
Example: 4 queens.

Two possible solutions are

Solutin-1
(2 4 1 3)



Solution 2
(3 1 4 2)



SUM OF SUBSETS:

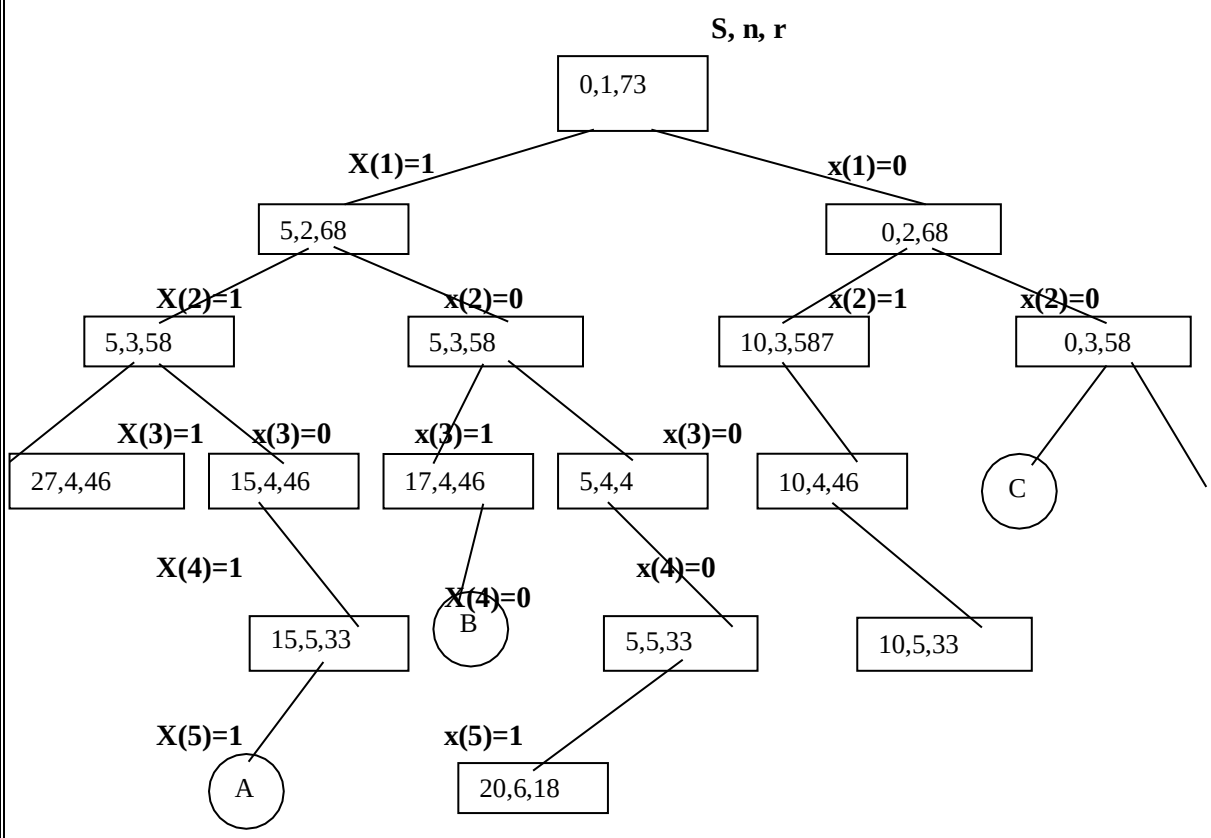
1) We are given 'n' positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.
2) If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.
3) If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and
4) right to X(i)=0.

Example:

a. Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.

b. In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

c. The state space tree for the given problem is,



Ist solution is A -> 1 1 0 0 1 0

IInd solution is B -> 1 0 1 1 0 0

III rd solution is C -> 0 0 1 0 0 1

In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeled with the values of Xi, which is either 0 or 1.

The left sub tree of the root defines all subsets containing Wi.

The right subtree of the root defines all subsets, which does not include Wi.

3.5.GENERATION OF STATE SPACE TREE:

Maintain an array X to represent all elements in the set.

The value of Xi indicates whether the weight Wi is included or not.

Sum is initialized to 0 i.e., s=0.

We have to check starting from the first node.

Assign X(k)<- 1.

- If S+X(k)=M then we print the subset b'coz the sum is the required output.
- If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.
- After generating the left sub tree we have to generate the right sub tree, for this we have to check S+W(k+1)<=M.B'coz W(k) is omitted and W(k+1) has to be selected.
- Repeat the process and find all the possible combinations of the subset.

Algorithm:

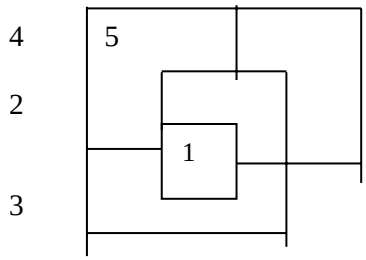
```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is true.
X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as W[j]>0,1<=j<=n.
Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r- W[k]);
//generate right child and evaluate Bk.
```

```
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{
  X{k]=0;
  sum of sub (S, k+1, r- W[k]);
}
}
```

GRAPH COLORING:

- Let ‘G’ be a graph and ‘m’ be a given positive integer. If the nodes of ‘G’ can be colored in such a way that no two adjacent nodes have the same color. Yet only ‘M’ colors are used. So it’s called M-color ability decision problem.
- The graph G can be colored using the smallest integer ‘m’. This integer is referred to as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.

2 is adjacent to 1, 3, 4, 5

3 is adjacent to 1, 2, 4

4 is adjacent to 1, 2, 3, 55 is adjacent to 2, 4

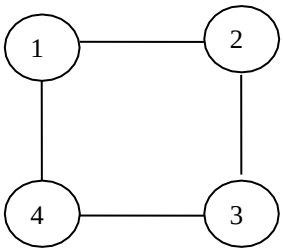
Steps to color the Graph:

1. First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.
2. The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),.....,X(n) ,X(index) is the color, index is the node.
3. He formula is used to set the color is,

   X(k) = (X(k)+1) % (m+1)
4. First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.
5. Repeat the procedure until all possible combinations of colors are found.
6. The function which is used to check the adjacent nodes and same color is,

If((Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4

M= 3

Adjacency Matrix:

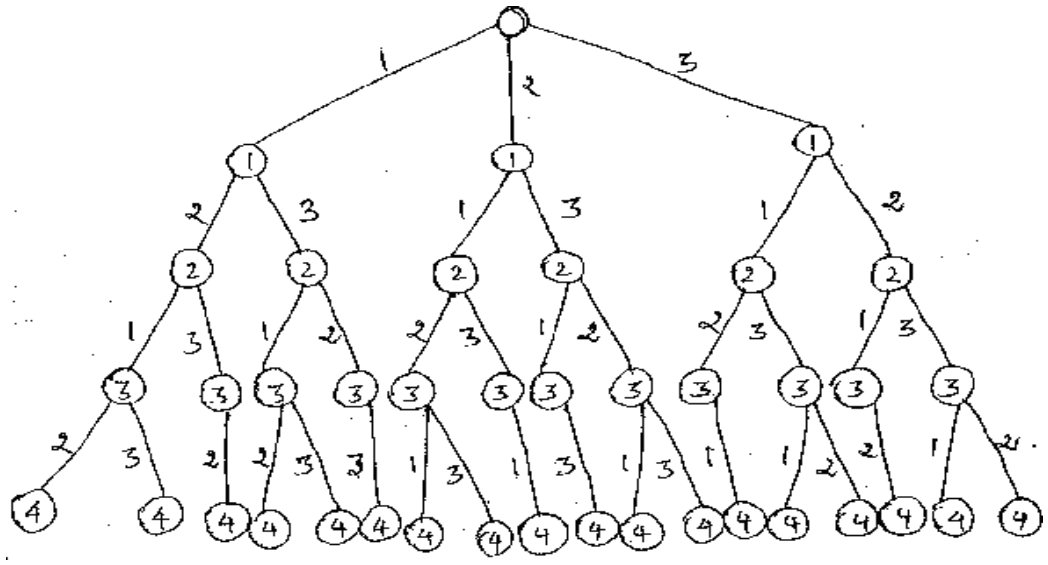
0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

→ Problem is to color the given graph of 4 nodes using 3 colors.

→Node-1 can take the given graph of 4 nodes using 3 colors.

→ The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments //of 1,2,…….,m to the vertices of the graph such that adjacent vertices are assigned //distinct integers are printed. 'k' is the index of the next vertex to color.

```
{
repeat
{
   // generate all legal assignment for X[k].
  Nextvalue(k);    // Assign to X[k] a legal color.
        If (X[k]=0) then return;          // No new color possible.
       If (k=n) then                  // Almost 'm' colors have been used to color the 'n' vertices
                Write(x[1:n]);
     Else mcoloring(k+1);
}until(false);
}
```

Algorithm Nextvalue(k)

// X[1],……X[k-1] have been assigned integer values in the range[1,m] such that //adjacent values have distinct integers. A value for X[k] is determined in the //range[0,m].X[k] is assigned the next

highest numbers color while maintaining //distinctness form the adjacent vertices of vertex K. If no such color exists, then X[k] is 0.

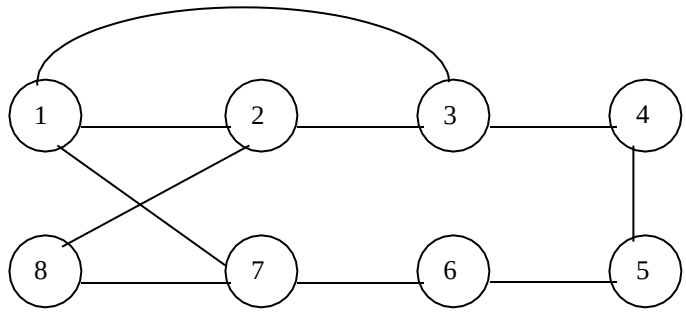
```
{
  repeat
   {
        X[k] = (X[k]+1)mod(m+1);    // next highest color.
       If(X[k]=0) then return;            //All colors have been used.
         For j=1 to n do
        {
           // Check if this color is distinct from adjacent color.
        If((G[k,j] ≠ 0)and(X[k] = X[j]))
           // If (k,j) is an edge and if adjacent vertices have the same color.
        Then break;
        }
      if(j=n+1) then return;     //new color found.
   } until(false);    //otherwise try to find another color.
}
```

→ The time spent by Nextvalue to determine the children is θ (mn)

→Total time is = θ (m^n n**).**

HAMILTONIAN CYCLES:

- Let G=(V,E) be a connected graph with 'n' vertices. A HAMILTONIAN CYCLE is a round trip path along 'n' edges of G which every vertex once and returns to its starting position.
- If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertex are visited in the order of V1,V2…….Vn+1,then the edges are in E,1<=I<=n and the Vi are distinct except V1 and Vn+1 which are equal.
- Consider an example graph G1.



The graph G1 has Hamiltonian cycles:

->1,3,4,5,6,7,8,2,1 and

->1,2,8,7,6,5,4,3,1.

- The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

- Define a solution vector X(Xi……..Xn) where Xi represents the I th visited vertex of the proposed cycle.
- Create a cost adjacency matrix for the given graph.
- The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.
- Now we have to find the second vertex to be visited in the cycle.
- The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,

  1.There should be a path from previous visited vertex to current vertex.

  2.The current vertex must be distinct and should not have been visited earlier.

6. When these two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.
7. When the nth vertex is visited we have to check, is there any path from nth vertex to first vertex. if no path, the go back one step and after the previous visited node.
8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:(Finding all Hamiltonian cycle)

```
Algorithm Hamiltonian (k)
{
 Loop
        Next value (k)
```

```
If (x (k)=0) then return;
{
  If k=n then
      Print (x)
Else
Hamiltonian (k+1);
End if
}
Repeat
}
```

Algorithm Nextvalue (k)

```
{
 Repeat
{
  X [k]=(X [k]+1) mod (n+1); //next vertex
  If (X [k]=0) then return;
  If (G [X [k-1], X [k]] ≠ 0) then
{
  For j=1 to k-1 do if (X [j]=X [k]) then break;
  // Check for distinction.
  If (j=k) then          //if true then the vertex is distinct.
    If ((k<n) or ((k=n) and G [X [n], X [1]] ≠ 0)) then return;
}
} Until (false);
}
```

NP-HARD AND NP-COMPLETE PROBLEMS:

Basic concepts:

→ **Tractability:** Some problems are *tractable*: that is the problems are solvable in reasonable amount of time called polynomial *time*. Some problems are *intractable*: that is as problem grow large, we are unable to solve them in reasonable amount of time called polynomial *time*.

→**Polynomial Time Complexity:** An algorithm is of **Polynomial Complexity,** if there exists a polynomial p() such that the computing time is O(p(n)) for every input size of 'n'. Polynomial time is the worst-case running time required to an algorithm to process an input of size *n* the is O(n^k) for some constant *k*

→ Polynomial time: O(n^2), O(n^3), O(n log n)

→ Not in polynomial time: O(2^n), O(n^n), O(n!) →Exponential Time

→Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

Decision Problems: Computational problem with produces output of "yes" or "no", 1 or 0 are decision problems.

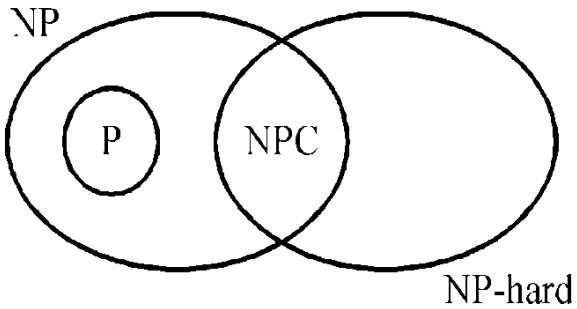
Examples: 1. Path in a graph

2. Minimum Spanning Tree whose cost is less than some value w.

Optimization Problems:___Computational problem where we try to maximize or minimize some value that is identifying optimal solution to problem

Examples: 1. Shortest-path in a graph.

2. Minimum Spanning Tree



CLASS P PROBLEMS:

- Class P problems are the set of decision problems solvable by deterministic algorithms in polynomial-time.
- A deterministic algorithm is (essentially) one that always computes the correct answer
- **Examples:** Fractional Knapsack, MST, Single-source shortest path

CLASS NP PROBLEMS:

- NP problems are set of decision problems solvable by non-deterministic algorithms in polynomial-time.
- A nondeterministic algorithm is one that can "guess" the right answer or solution
- **Examples:** Hamiltonian Cycle (Traveling Sales Person), Conjunctive Normal Form (CNF)

NP-Complete Problems:

- A problem 'x' is a NP class problem and also NP-Complete if and only if every other problem

in NP can be reducible (solvable) using non-deterministic algorithm in polynomial time.

- The class of problems which are NP-hard and belong to NP.
- The NP-Complete problems are always decision problems only.
- **Example :** TSP, Vertex covering problem

Examples of NP-complete problems:

- Packing problems: SET-PACKING, INDEPENDENT-SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3-COLOR, CLIQUE.
- Constraint satisfaction problems: SAT, 3-SAT.

Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK

NP-Hard Problems:

- A problem 'x' is a NP class problem and also NP-Hard if and only if every other problem in NP can be reducible (solvable) using non-deterministic algorithm in exponential time.
- The class of problems to which every NP problem reduces.
- The NP-Hard problems are decision problems and sometimes may be optimization problems.
- **Example :** Integer Linear Programming.

Nondeterministic Algorithms:

Deterministic Algorithms:

- Let A be an algorithm to solve problem P. A is called deterministic if it has only one choice in each step throughout its execution. Even if we run algorithm A again and again, there is no change in output.
- Deterministic algorithms are identified with uniquely defined results in terms of output for a certain input.

Nondeterministic Algorithms:

- Let A be a nondeterministic algorithm for a problem P. We say that algorithm A accepts an instance of P if and only if, there exists a guess that leads to a yes answer.
- In non deterministic algorithms, there is no uniquely defined result in terms of output for a

certain input.

- Nondeterministic algorithms are allowed to contain operations whose outcomes are limited to a given set of instances of P, instead of being uniquely defined results.
- A Non-deterministic algorithm A on input x consists of two phases:

– **Guessing:** An arbitrary "string of characters" is generated in polynomial time. It may

→Correspond to a solution

→Not correspond to a solution

→Not be in proper format of a solution

→Differ from one run to another

– **Verification:** A deterministic algorithm verifies

→The generated "string of characters" is in proper format

→Whether it is a solution in polynomial time

- The Nondeterministic algorithm uses three basic procedures, whose time complexity is O(1).

1. **CHOICE(1,*n*) or CHOICE(*S*) :** This procedure chooses and returns an arbitrary element, in favor of the algorithm, from the closed interval [1,*n*] or from the set *S*.
2. **SUCCESS :** This procedure declares a successful completion of the algorithm.
3. **FAILURE :** This procedure declares an unsuccessful termination of the algorithm.
   - Non deterministic algorithm terminates unsuccessfully if and only if there is no set of choices leading to successful completion of algorithm
   - Non deterministic algorithm terminates successfully if and only if there exists set of choices leading to successful completion of algorithm

Nondeterministic Search Algorithm: The following algorithm enables nondeterministic search of x in an unordered array A with n elements. It determines an index j such that A[j] = x or j = −1 if x does not belongs to A.

```
Algorithm nd_search ( A, n, x )
{
int j = choice ( 0, n-1 );
if ( A[j] == x )
{
cout << j;
success();
}
cout << -1;
failure();
}
```

By the definition of nondeterministic algorithm, the output is -1 iff there is no j such that A[j] = x Since A is not ordered, every deterministic search algorithm is of complexity O(n), whereas the nondeterministic algorithm has the complexity as O(1).

Nondeterministic Sort Algorithm: The following algorithm sorts 'n' positive integers in non-decreasing order and produces output in sorted order. The array B[] is an auxiliary array initialized to 0 and is used for convenience.

Algorithm nd_sort (A, n)

```
{
for ( i = 0; i < n; B[i++] = 0; );
for ( i = 0; i < n; i++ )
{
j = choice ( 0, n - 1 );
if ( B[j] != 0 ) failure();
B[j] = A[i];
}
// Verify order
for ( i = 0; i < n-1; i++ )
if ( B[i] > B[i+1] ) failure();
write ( B );
success();
}
```

The time complexity of nd_sort is O(n). Best-known deterministic sorting algorithm like binary search has a complexity of (n log n).

5.10 Satisfiability: (SAT Problem)

→ Let x1, x2 . . . denote a set of Boolean variables and xi denote the complement of $\bar{x}_i$.

→A variable or its complement is called a literal

→A formula in propositional calculus is an expression that is constructed by connecting literals using the operations and (∧) & or (∨)

→Examples of formulas in propositional calculus

$(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$

$(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$

→Conjunctive normal form (CNF): A Boolean formula is said to be in conjunctive normal form (CNF) if it is the conjunction of formulas.

Example: $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_5)$

→Disjunctive normal form (DNF) : A Boolean formula is said to be in disjunctive normal form (CNF) if it is the disjunction of formulas.

Example: $(x_1 \wedge \bar{x}_2) \vee (x_1 \wedge \bar{x}_5)$

→ **Satisfiability problem** is to determine whether a formula is true for some assignment of truth values to the variables

→ CNF--satisfiability is the satisfiability problem for CNF formulas

→ DNF--satisfiability is the satisfiability problem for DNF formulas

→ Polynomial time nondeterministic algorithm that terminates successfully iff a given propositional formula E(x1, . . . , xn) is satisfiable

→ Non deterministically choose one of the 2^n possible assignments of truth values to (x1, . . . , xn) and verify that E(x1, . . . , xn) is true for that assignment

Algorithm eval (E, n)

```
{
// Determine whether the propositional formula E is satisfiable.
Here variable are x1, x2, ..., xn
 for ( i = 1; i <= n; i++ )
x(i) = choice ( true, false );
if ( E ( x1, ..., xn ) )
success();
else
failure();
}
```

→The nondeterministic time to choose the truth value is O(n)

→The deterministic evaluation of the assignment is also done in O(n) time

Decision Problem Vs Optimization Problem:

Decision Problem and Algorithm : Any problem for which the answer is either zero or one is called a decision problem. An algorithm for a decision problem is termed a decision algorithm.

- A decision algorithm will output 0 or 1
- Implicit in the signals success() and failure()
- Output from a decision algorithm is uniquely defined by input parameters and algorithm specification.

Optimization Problem and Algorithm: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

- An optimization problem may have many feasible solutions
- The problem is to find out the feasible solution with the best associated value
- NP-completeness applies directly not to optimization problems but to decision problems.

Casting (Conversion) of Optimization Problem into Decision Problem:

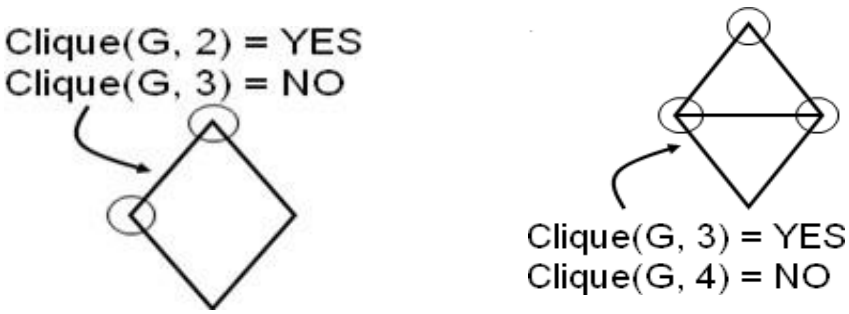
Optimization problems can be cast into decision problems by imposing a bound on output or solution. Decision problem is assumed to be easier (or no harder) to solve compared to the optimization problem. Decision problem can be solved in polynomial time if and only if the corresponding optimization problem can be solved in polynomial time. If the decision problem cannot be solved in polynomial time, the optimization problem cannot be solved in polynomial time either.

For example consider, shortest path problem. Optimization problem is to find a shortest path between two vertices in an undirected weighted graph, so that shortest path consists least number of edges. Whereas the decision problem is to determine that given an integer k, whether a path exists between two specified nodes consisting of at most k edges.

Maximal Clique:

Clique is a maximal complete sub-graph of a graph G = (V,E), that is a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph).

Example:



The **Size of a clique** is the number of vertices in it. The Maximal clique problem is an optimization problem that has to determine the size of a largest clique in G. A decision problem is to determine whether G has a clique of size at least 'k'.

Input for Maximal clique problem: Input can be provided as a sequence of edges. Each edge in E(G) is a pair of vertices (i, j) .The size of input for each edge (i, j) in binary representation is

$\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$

And input size of any instance is given by

$$n = \sum_{\substack{(i,j) \in E(G) \\ i < j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$$

Where 'k' is the number to indicate the clique size

Maximal clique problem as Decision Problem:

Let us denote the deterministic decision algorithm for the clique decision problem as dclique(G, k). If |V | = n, then the size of a maximal clique can be found by

for (k = n; dclique (G, k) != 1; k--);

If time complexity of dclique is f(n), size of maximal clique can be found in time g(n) <= n.f(n). Therefore, the decision problem can be solved in time g(n)

Note that Maximal clique problem can be solved in polynomial time if and only if the clique decision problem can be solved in polynomial time.

Non deterministic Clique Algorithm:

```
    Algorithm DCK(G, n, k)
{
    S=0; // Empty set.
    for i=1 to k  do
    {
            t = Choice(1,n);
            if t ε S then Failure();
            S= SU{ t };
    }
    for all pairs (i,j) such that i ε S, j ε S and i!=j do
            if (i, j) is not an edge of G then Failure();
    Success();
}
```

Non-deterministic knapsack problem:

- It is a non-deterministic polynomial time complexity algorithm.
- The for loop selects or discards each of the n items It also re-computes the total weight and profit corresponding to the selection
- The if statement checks to see the feasibility of assignment and whether the profit is above a lower bound r
- The time complexity of the algorithm is O(n) . If the input length is q in binary, then O(q).

Algorithm nd_knapsack (p, w, n, m, r, x)

```
{
W = 0; P = 0;
for ( i = 1; i <= n; i++ )
{
x[i] = choice ( 0, 1 );
W += x[i] * w[i];
```

```
P += x[i] * p[i];
}
if ( ( W > m ) || ( P < r ) )
failure();
else
success();
}
```

SUM OF SUBSETS PROBLEM:

- Bits are numbered from 0 to m from right to left
- Bit i will be 0 if and only if no subsets of A[j], 1 _ j _ n sums to i
- Bit 0 is always 1 and bits are numbered 0, 1, 2, . . . ,m right to left
- Number of steps for this algorithm is O(n)
- Each step moves m + 1 bits of data and would take O(m) time on a conventional computer
- Assuming one unit of time for each basic operation for a fixed word size, the complexity of deterministic algorithm is O(nm)
- Consider the deterministic decision algorithm to get sum of subsets

Algorithm Sum_of_subsets (A, n, m)

```
{
// A[n] is an array of integers
s = 1 // s is an m+1 bit word
// bit 0 is always 1
for i = 1 to n
s |= ( s << A[i] ) // shift s left by A[i] bits
if bit m in s is 1
write ( "A subset sums to m" );
else
write ( "No subset sums to m" );
}
```

COOK'S THEOREM:

- We know that, Class **P** problems are the set of all decision problems solvable by deterministic algorithms in polynomial time. Similarly **Class NP** problems are set of all decision problems solvable by nondeterministic algorithms in polynomial time.
- Since deterministic algorithms are a special case of nondeterministic algorithms, P NP
- Cook formulated the following question: Is there any single problem in NP such that if we showed it to be in P, then that would imply that P = NP? This led to Cook's theorem as :

Satisfiability is in P if and only if P = NP.

<u>Cook's Theorem Proof:</u>

Consider Z - denotes a deterministic polynomial algorithm

A - denotes a non- deterministic polynomial algorithm

I - denotes input instance of algorithm

n - denotes length of input instance

Q - denotes a formulae

m - denotes length of formulae

→ Now, the formula 'Q' is satisfiable if and only if the non-deterministic algorithm 'A' has a successful termination with input 'I'.

→ If the time complexity of 'A' is p(n) for some polynomial p(), then the time needed to construct the formula 'Q' by algorithm 'A' is given by $O(p^3(n)\log n)$.

Therefore complexity of non-deterministic algorithm 'A' is $O(p^3(n)\log n)$. (NP)

→ Similarly, the formula 'Q' is satisfiable if and only if the deterministic algorithm 'Z' has a successful termination with input 'I'.

→ If the time complexity of 'Z' is q(m) for some polynomial q(), then the time needed to construct the formula 'Q' by algorithm 'Z' is given by $O(p^3(n)\log n + q(p^3(n)\log n))$.

Therefore complexity of deterministic algorithm 'Z' is $O(p^3(n)\log n + q(p^3(n)\log n))$. (P)

→ If satisfiability is in P, then q(m) is a polynomial function and the complexity of 'Z' becomes O(r(n)) for some polynomial r(n).

→ Hence, P is satisfiable, then for every non-deterministic algorithm 'A' in NP can obtain a deterministic algorithm 'Z' in P.

→ So, the above construction shows that "if satisfiability is in P, then P=NP"

PART - A(2 Marks)

1. Write the general procedure of dynamic programming.

Ans: The development of dynamic programming algorithm can be broken into a sequence of 4 steps.

- Characterize the structure of an optimal solution.
- Recursively define the value of the optimal solution.
- Compute the value of an optimal solution in the bottom-up fashion.
- Construct an optimal solution from the computed information.

2. Define principle of optimality.

Ans: It states that an optimal sequence of decisions has the property that whenever the initial stage or decisions must constitute an optimal sequence with regard to stage resulting from the first decision.

3. Write the difference between the Greedy method and Dynamic programming.

Greedy method

1. Only one sequence of decision is generated.
2. It does not guarantee to give an optimal solution always.

Dynamic programming

1. Many number of decisions are generated.
2. It definitely gives an optimal solution always.

4. What are the drawbacks of dynamic programming?

Ans: Time and space requirements are high, since storage is needed for all level.

Optimality should be checked at all levels.

5. What are the features of dynamic programming?

Optimal solutions to sub problems are retained so as to avoid recomputing their values.

Decision sequences containing subsequences that are sub optimal are not considered.

It definitely gives the optimal solution always.

6. What is meant by n-queen Problem?

Ans: The problem is to place n queens on an n-by-n chessboard so that no two queens attack each

other by being in the same row or in the same column or in the same diagonal.

7. Define Backtracking

Ans: Backtracking is used to solve problems with tree structures. Even problems seemingly remote to trees such as a walking a maze are actually trees when the decision \'back-left-straight-right\' is considered a node in a tree. The principle idea is to construct solutions one component at a time and evaluate such partially constructed candidates

8. What is the Aim of Backtracking?

Ans:Backtracking is the approach to find a path in a tree. There are several different aims to be achieved : • just a path • all paths • the shortest path.

9. Define the Implementation considerations of Backtracking?

Ans: The implementation bases on recursion. Each step has to be reversible; hence the state has to be saved somehow. There are two approaches to save the state: • As full state on the stack • As reversible action on the stack

10. List out the implementation procedure of Backtracking

Ans: As usual in a recursion, the recursive function has to contain all the knowledge. The standard implementaion is :

1. check if the goal is achieved REPEAT
2. check if the next step is possible at all
3. check if the next step leads to a known position - prevent circles
4. do this next step UNTIL (the goal is achieved) or (this position failed) .

11. Define Subset-Sum Problem?

Ans:This problem find a subset of a given set S={s1,s2,……,sn} of n positive integers whose sum is equal to a given positive integer d.

12. Define Traveling Salesman Problem?

Ans: Given a complete undirected graph G=(V, E) that has nonnegative integer cost c(u, v) associated with each edge (u, v) in E, the problem is to find a hamiltonian cycle (tour) of G with minimum cost.

13. Define Knapsack Problem

Ans: Given n items of known weight wi and values vi=1,2,..,n and a knapsack of capacity w, find the most valuable subset of the items that fit in the knapsack.

14. . What is a state space tree?

Ans:The processing of backtracking is implemented by constructing a tree of choices being made. This is called the state-space tree. Its root represents a initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of the solution, the nodes in the second level represent the choices for the second component and so on.

15.Define nondeterministic Polynomial

Ans: Class NP is the class of decision problems that can be solved by nondeterministic Polynomial algorithms. This class of problems is called nondeterministic Polynomial.

16. Define NP-Complete

Ans: An NP-Complete problem is a problem in NP that is as difficult as any other problem in this class because any other problem in NP can be reduced to it in Polynomial time.

17.Define Polynomial reducible

Ans: A Decision problem D1 is said to be polynomial reducible to a decision problem D2 if there exists a function t that transforms instances of D2 such that o T maps all yes instances of D1 to yes instances of D2 and all noninstances of D1 to no instance of D2 o T is computable by a Polynomial-time algorithm

18.What is the difference between tractable and intractable?

Problems that can be solved in polynomial time are called tractable and the problems that cannot be solved in Polynomial time are called intractable.

19.Define undecidable Problem

Some decision problem that cannot be solved at all by any algorithm is called undecidable algorithm.

20 .Define Heuristic Generally speaking, a heuristic is a "rule of thumb," or a good guide to follow when making decisions.

In computer science, a heuristic has a similar meaning, but refers specifically to algorithms.

PART - B (10 Marks)

1) Write an algorithm to estimate the efficiency of backtracking.
2) Explain 4-queen problem using backtracking.
3) How many solutions are there to the eight queens problem? How many distinct solutions are there if we do not distinguish solution that can be transformed into one another by rotations and reflections?
4) Explain about graph coloring and Hamiltonian cycles with examples.
5) Explain Sum of subsets problem with an example.
6) Explain traveling sales person problem?
7) Explain about optimal binary search tress?
8) Take an example problem and solve All-Pairs shortest path?
9) Explain and give an example for single source shortest path?
10) Prove that any two NP complete problems are polynomial turning equivalent.
11) Give brief description about the cook's theorem and prove with example.
12) Give out the relation between NP hard and NP completeness problems.
13) Discuss NP hard and NP complete problems.
14) Discuss in detail the different classes in NP hard and NP complete.