ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS

PREPARED BY

PEMMA RADHIKA

ANNAMACHARYA UNIVERSITY

EXCELLENCE IN EDUCATION; SERVICE TO SOCIETY

(ESTD, UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016)

Title of the Course:	Advanced Data Structures and Algorithm Analysis
Category:	Professional Core
Couse Code:	24ACSE31T
Year:	II B. Tech
Semester:	I Semester
Branch:	AI& DS, AI&ML, CSE, CSE(AI), CSE(AIML),CSE(DS) and CSE(IOTCSBT)

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	-	-	3

Course Objectives: This course will be able to

1. To understand and analyse the time and space complexity of algorithms using asymptotic notations.
2. To study advanced data structures like AVL trees, B-trees, heaps, and graphs, and apply them in real-world problems.
3. To explore algorithmic design paradigms including divide-and-conquer, greedy, and dynamic programming techniques.
4. To solve complex computational problems using backtracking and branch-and-bound strategies.
5. To introduce the concept of NP-completeness and understand the complexity of intractable problems.

Course Outcomes:

At the end of the course, the student will be able to

1. Analyze algorithm performance using time and space complexity and apply asymptotic notations.
2. Implement AVL trees, B-trees, heap trees, and graph traversals for efficient data handling.
3. Solve problems using divide-and-conquer, greedy, and dynamic programming techniques.
4. Apply backtracking and branch-and-bound algorithms to solve combinatorial and optimization problems.
5. Understand NP-hard and NP-complete problems and evaluate the computational complexity of algorithms.

Unit 1	**Introduction to Algorithms and advanced Data Structures**	10

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.
AVL Trees – Creation, Insertion, Deletion operations and Applications

B-Trees – Creation, Insertion, Deletion operations and Applications

Unit 2	**Advanced Data Structures and Divide and Conquer**	10

Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications
Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications

Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen‟s matrix multiplication, Convex Hull

Unit 3	**Greedy Method and Dynamic Programming**	10

Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths Dynamic Programming: General Method, All pairs shortest paths, Single Source Shortest Paths– General Weights (Bellman Ford Algorithm), Optimal Binary Search Trees, 0/1 Knapsack, String Editing, Travelling Salesperson problem		
Unit 4	**Backtracking & Branch and Bound**	10
Backtracking: General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, 0/1 Knapsack Problem Branch and Bound: The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem		
Unit 5	**NP Hard and NP Complete Problems**	10
NP Hard and NP Complete Problems: Basic Concepts, Cook's theorem. NP Hard Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP) NP Hard Scheduling Problems: Scheduling Identical Processors, Job Shop Scheduling		

Prescribed Textbooks:

1. Fundamentals of Data Structures in C++, Horowitz, Ellis; Sahni, Sartaj; Mehta, Dinesh, 2nd Edition Universities Press
2. Computer Algorithms in C++, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, 2nd Edition University Press

Reference Books:

1. Data Structures and program design in C, Robert Kruse, Pearson Education Asia
2. An introduction to Data Structures with applications, Trembley & Sorenson, McGraw Hill
3. The Art of Computer Programming, Vol.1: Fundamental Algorithms, Donald E Knuth, Addison-Wesley, 1997.
4. Data Structures using C & C++: Langsam, Augenstein & Tanenbaum, Pearson, 1995
5. Algorithms + Data Structures & Programs:, N.Wirth, PHI
6. Fundamentals of Data Structures in C++: Horowitz Sahni & Mehta, Galgottia Pub.
7. Data structures in Java:, Thomas Standish, Pearson Education Asia

Online Learning Resources:

1. https://www.tutorialspoint.com/advanced_data_structures/index.asp
2. http://peterindia.net/Algorithms.html
3. https://www.youtube.com/playlist?list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O

PERFORMANCE ANALYSIS

Efficiency of an algorithm can be decided by measuring the performance of an algorithm. The performance of an algorithm depends upon two factors.

1. Space Complexity
2. Time Complexity

1. Space Complexity

Space complexity of an algorithm is the amount of memory it needs to run to completion. Space needed by an algorithm is equal to the sum of the following two components

- ***A fixed part*** -that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent on the size of the problem.
- ***A variable part*** -is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity S(p) of any algorithm p is

$$S(p) = C + S_p$$

Where C is treated as the fixed part (Constant value) and Sp is treated as the variable part of the algorithm which depends on **instance characteristic-** it refer to the specific properties or attributes of the input data in the given algorithm. These characteristics are essential in understanding the behavior and performance of an algorithm.

Following is a simple example that tries to explain the concept

Example:1

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

The problem instance is characterized by the specific values of a, b, and c. The space needed by a, b & c is independent of the instance characteristics.

Consequently, Sp(instance characteristics) = 0.

Example:2

```
Algorithm Sum(a, n)
{
        s:= 0.0;
        for i:= 1 to n do
        s:=s+ a[i];
        return s;
}
```

The problem instances for Algorithms are characterized -The space needed by n is one word. The space needed by i is one word and The space needed by s is one word. So, total space required for fixed part variables are 3 words

But The space needed by array of variable a[ ] will be n words

Sum(n) ≥ (n + 3)

2. Time Complexity

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

T(p)= Compile Time + Execution Time

The compile time does not depend on the instance characteristics. This run time is denoted by tp (instance characteristics). Execution time is measured by step count

Step count Rules:

- Comments-0 steps
- Assignment Statement- 1 step
- Condition Statement - 1 step
- Loop Condition for n times- n+1 steps
- Body of loop - n steps
- Recursion- n+1

Here, the number of steps per execution(s/e) of the statement and the total number of times (i.e., frequency) each statement is executed By adding the contributions of all statements, the step count for the entire algorithm is obtained

Example 1: Step count for SUM OF INTEGERS algorithm

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn + 2m + 1$

Example 2: Step count for SUM OF INTEGERS algorithm using Recursion

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a, n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

ASYMPTOTIC NOTATIONS

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

The growth patterns of order notations have been listed below:

$O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \ldots < O(2^n)$.

The common name of few order notations is listed below:

Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(n log(n))	Linearithmic
$O(n^2)$	Quadratic
$O(c^n)$	Exponential
O(n!)	Factorial

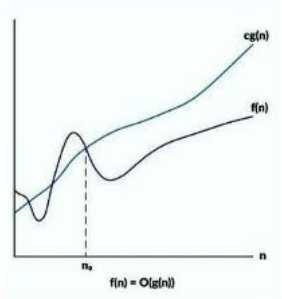
For example, In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case. But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case. When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

1. Big-O notation
2. Omega notation
3. Theta notation

1. Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Big-O gives the upper bound of a function

The function f(n)= O(g(n))

if There exist a positive constants c, Positive numbers n & n_0 such that

$0 \le f(n) \le cg(n)$ for all $n \ge n_0$, $C \ge 0$

The above expression can be described as a function f(n) belongs to the set O(g(n))

$f(n) \le O(g(n))$

$f(n) \le C.g(n)$

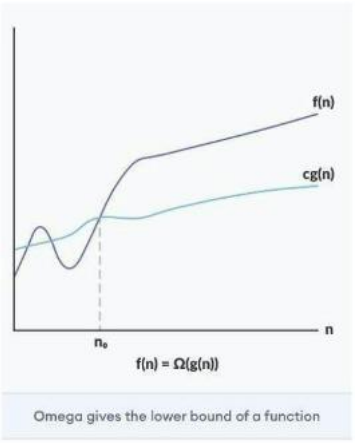
For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Examples

1. The function 3n + 2 = O(n) as 3n + 2 < =4n for all n >= 23n +3= O(n) as 3n + 3 <= 4n for all n >= 3.
2. 100n +6=O(n) as 100m +6 < =101n for all n ≥ 6.
3. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all n≥5.

2. Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the **best-case** complexity of an algorithm.



Omega gives the lower bound of a function

The function f(n) =Ω (g(m)) (read as "f of n is omega of q of n")if there exist positive constants c and n_0 such that

f(n) ≥ c* g(n) for all n, n ≥ n_0 , C≥0

The above expression can be described as a function f(n) belongs to the set

Ω(g(n))f(n) ≥ Ω (g(n))

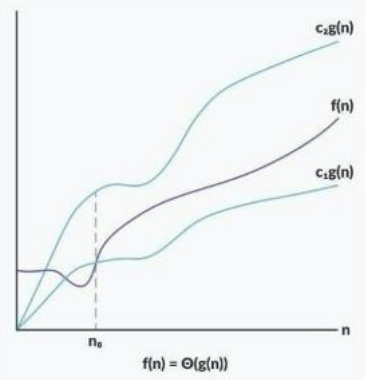
f(n) ≥ C.g(n)

Examples:

1. The function 3n + 2= Ω (n) as 3n + 2 ≥ 3n for n ≥ 1 3n +3 = Ω (n) as 3n + 3 ≥ 3n for n ≥ 1.
2. 100n + 6 = Ω (n) as 100n+ 6 ≥100n for n ≥ 1.
3. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for n ≥ 1.

3. Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the **average-case** complexity of an algorithm.



Theta bounds the function within constants factors

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n_0 such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all $n \ge n_0$ }

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all $n \ge n_0$, then f(n) is said to be asymptotically tight bound.

Examples:

1. The function 3n+2 = Θ(n)as 3n+2 ≥ 3n for all n ≥2 and 3n+2 ≤4n for all n≥ 2,so c1 = 3, C2= 4, and n_0 = 2.
2. 3n+3 = Θ (n),

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

o-notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

AVL Trees

To eliminate this drawback of BST, it is essential that during an insert or delete operation which can affect the structure of the tree and hence the height of the tree, which turns into **skewed**. We needa mechanism called **balanced trees** or **height balanced trees. AVL Trees** are the best example for **strict balanced** or **self balanced trees**. AVL trees were proposed by Adelson- Velski and Landis in 1962.

Definition:

An empty binary tree is an AVL tree. If non empty, the binary tree T is an AVL tree if

(i) $T_L$and T_R, the left and right subtrees of T are also AVL trees and

(ii) $|h(T_L) - h(T_R)| \leq 1$, where $h(T_L)$ and (T_R) are the heights of the left subtree and right subtree of T respectively.

For a node u, $bf(u) = (h(U_L) - h(U_R))$ where $h(U_L)$ and $h(U_R)$ are the heights of the left and right subtrees ofthe node u respectively, is known as the **balance factor (bf)** of the node u. In an AVL tree therefore, every node u has a balance factor bf(u) which may be either 0 or +1 or -1,

A binary search tree T which is an AVL tree is referred to as an AVL search tree. From the below diagram, the balance factor of each of the nodes is indicated by the side of the node within parentheses. The balance factors of the nodes in the AVL trees are either 0 or +1 or -1.

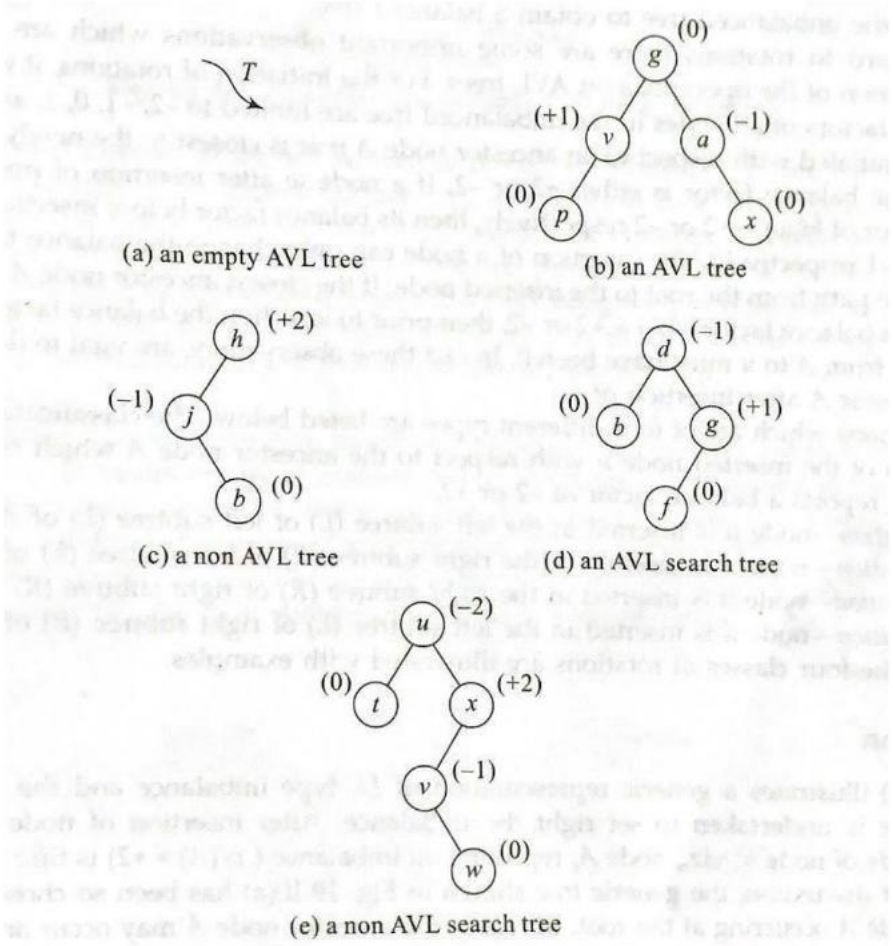


Fig. 10.9 *Examples of AVL trees and non AVL trees*

Operations on AVL Search trees

1. Retrieval from AVL search trees
2. Insertion into an AVL search tree
3. Deletion into an AVL search tree

1. Retrieval from AVL search trees

Let T be a AVL search tree. To retrieve a key u from T, u is first compared with the root key of T.

- If u= r then the search is done.
- If u < r then the search begins at the left subtree of T.
- If u> r then the search begins at the right subtree of T.

The search is repeated recursively in the left and right sub trees with u compared against the respective root keys, until the key u is either found or not found. If the key is found the search is termed **successful** and if not found, is termed **unsuccessful**

2. Insertion into an AVL search tree

The insertion of an element u into an AVL search tree T, proceeds exactly same as inserting u in a binary search tree. However, after insertion- the balance factors of any of the nodes turns out to be anything other than 0 or +1 or -1, then the tree is said to be **unbalanced**. To balance the tree we need do rotations. **Rotations** are mechanisms which shift some of the subtrees of the unbalanced tree to obtain a balanced tree.

For the initiation of rotations, it is required that the balance factors of all nodes in the unbalancedtree are limited to -2, -1, 0, 1, and +2. Also the rotation is initiated with respect to an ancestor node A thatis closest to the newly inserted node u and whose balance factor is either +2 or -2.

The rotations which are of four different types are listed below and the classification is based on the position of the inserted node u with respect to the ancestor node A which is closest to the node u and reports a balance factor of -2 or +2.

1. **LL rotation**- node u is inserted in the left subtree (L) of left subtree (L) of A
2. **LR rotation**- node u is inserted in the right subtree (R) of left subtree (L) of A
3. **RR rotation**- node u is inserted in the right subtree (R) of right subtree (R) of A
4. **RL rotation**- node u is inserted in the left subtree (L) of right subtree (R) of A

LL rotation

Figure 10.10 illustrates a generic representation **of LL type imbalance** After insertion of node u, the closest ancestor node of node u, viz., node A, reporting an imbalance **bf (A) = +2** is first found out. This implies to call for **LL rotation**. The AVL tree before insertion of u (Fig. 10.10(a)), the unbalanced tree after insertion of u (Fig. 10.10(b)) and the balanced tree after the LL rotation (Fig. 10.10(c)) have been illustrated.

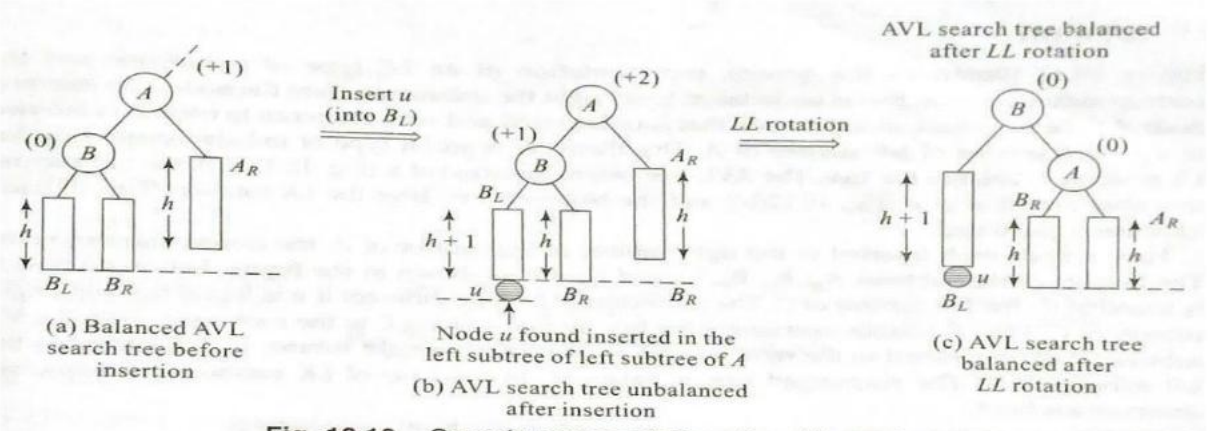


Fig. 10.10 *Generic representation of an LL rotation*

Here u is found inserted in the left subtree of B, viz., BL, where B is in the left subtree of A.bf(A) whichwas +1 before insertion of u ,changes to +2 after insertion. To balance the tree, the LL rotation pushes B up as the root of the AVL tree which results in node A slumping downwards to its left along with its right subtree AR. Now the tree is rearranged by shifting the right subtree of B, viz., BR to join A as its left subtree,

Example:

Consider the AVL search tree shown in Fig. 10.11(a). Let us insert C into the AVL search tree.

C is inserted in the left subtree of left subtree of M, the closest ancestor node of C that shows bf(M) = +2after insertion. The LL rotation pushes F up, to become the root of the tree and shifts the subtree with node K which was originally the right subtree of F, to the left subtree of M.

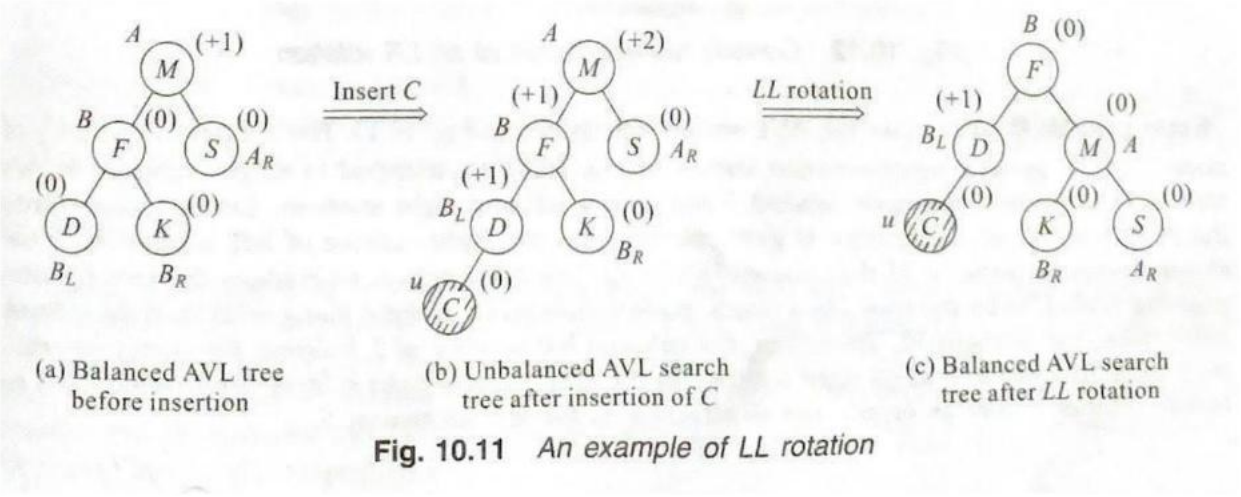


Fig. 10.11 *An example of LL rotation*

LR Rotation:

Here, after inserting the new node u at the left sub tree of right sub tree of A which yields to LR type of imbalance having the balancing factor (+2) . Therefore this type of imbalance calls for LR rotation to balance the tree. I'm the below diagrams we can see , The AVL tree before insertion of u (Fig. 10.12 (a)), the unbalanced tree after insertion of u (Fig. 10.12(b)) and the balanced tree after the LR rotation (Fig. 10.12(c))

To initiate the rotations, we need to follow these steps:

1. C to the root node.
2. Then the left subtree CL of C is shifted to the right subtree of B and
3. CR the right subtree of C is shifted to the C left subtree of A. The rearranged tree is balanced.

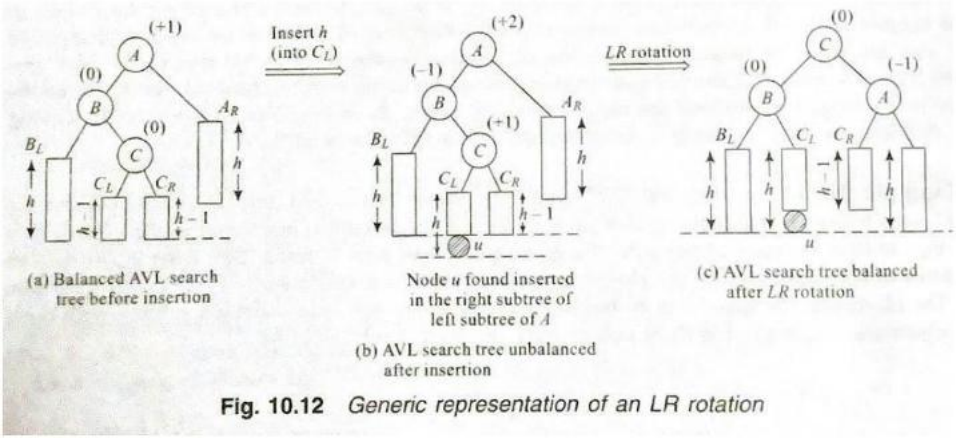


Fig. 10.12 *Generic representation of an LR rotation*

Example :

Considering the below example tree, from diagram (a) which is balanced AVL tree. From diagram (b),we can see the unbalanced AVL search tree after insertion of H at right sub tree of left sub tree of NodeS having the balancing factor of (+2) and yields to LR type of imbalance and calls for LR Rotation.

The LR rotation rearranges the tree by first pushing node L to be the root. As a result, node S slumps toits right along with its right subtree comprising the element W. Thereafter, the original left subtree of L holding the newly inserted node H is attached to F as its right subtree, which is shown in diagram (c).

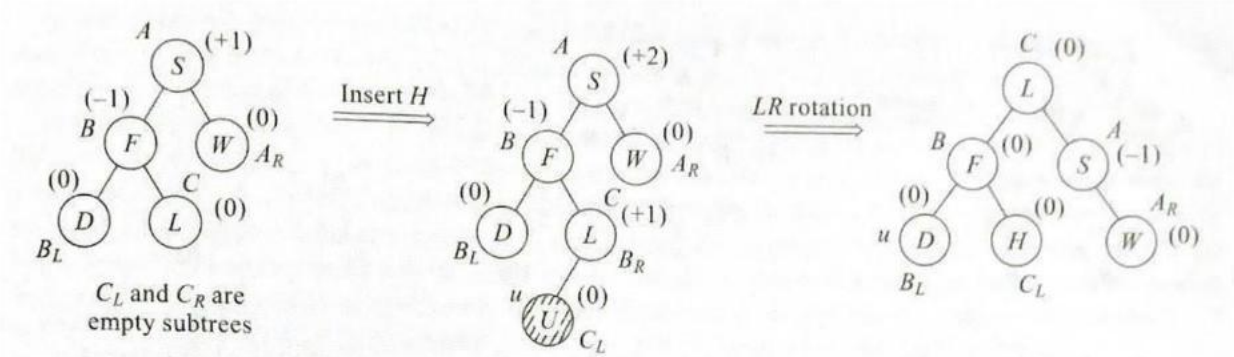


Fig. 10.13 *An example of LR rotation*

RR Rotation

The RR rotation is symmetric to the LL rotation. Figure 10.14 illustrates the generic representation of the RR rotation scheme. After inserting the node u at right subtree of right subtree of A, the closest ancestor of node u and which causes to unbalanced is node A and the rotation is merely a mirror image of the LL rotation scheme.

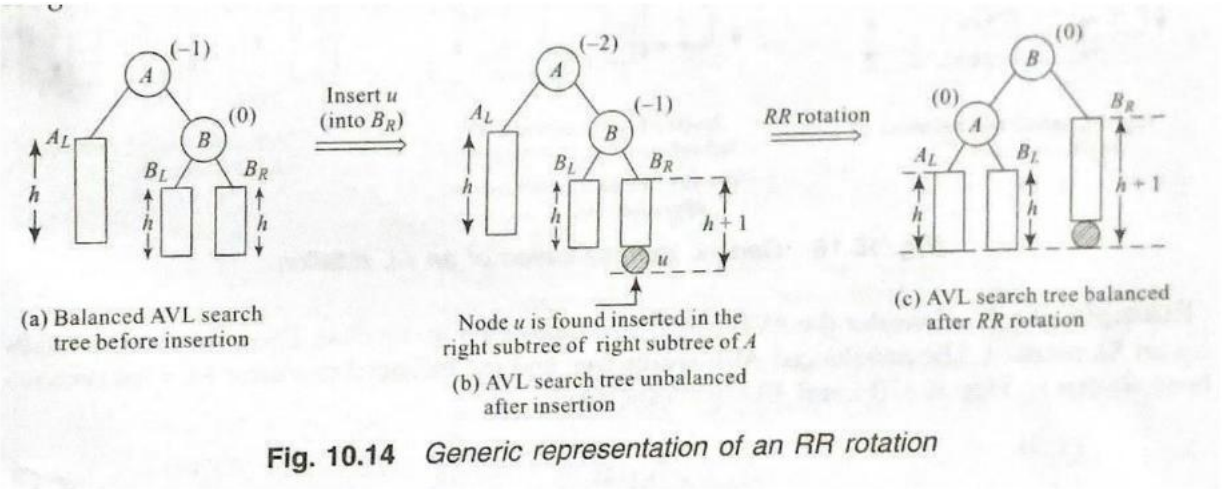


Fig. 10.14 *Generic representation of an RR rotation*

Example:

Consider the AVL search tree shown in Fig. 10.15. The insertion of Z calls for an RR rotation. The unbalanced AVL search tree and the balanced tree after RR rotation have been shown in Figs 10.15(b) and 10.15 (c) respectively.

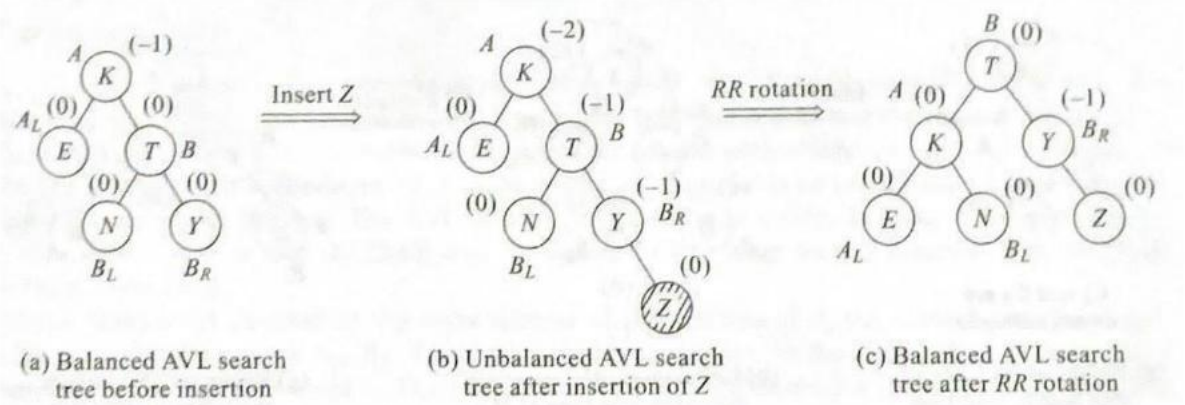


Fig. 10.15 *An example of RR rotation*

RL Rotation

RL rotation is symmetric to LR rotation. Figure 10.16 illustrates the generic representation of the RL rotation scheme. Here node u finds itself inserted in the left subtree of right subtree of node A which is the closest ancestor node A that is unbalanced.The RL rotation is the mirror image of the LR rotation scheme. The rotation procedure for RL remains the same as LR rotation. The procedure for RL rotationis:

1. C to the root node.
2. Then the left subtree CL of C is shifted to the right subtree of B and
3. CR the right subtree of C is shifted to the C left subtree of A. The rearranged tree is balanced.

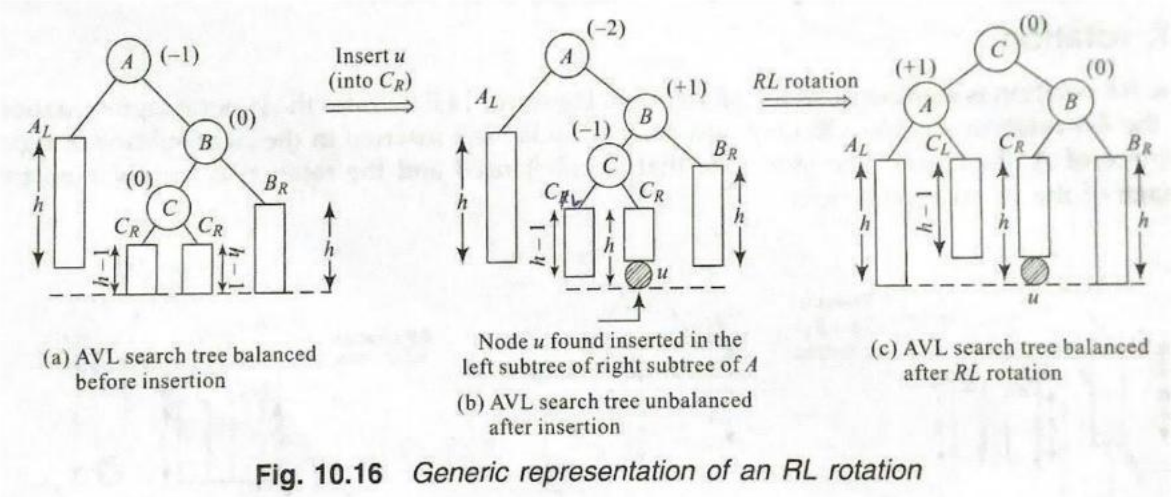
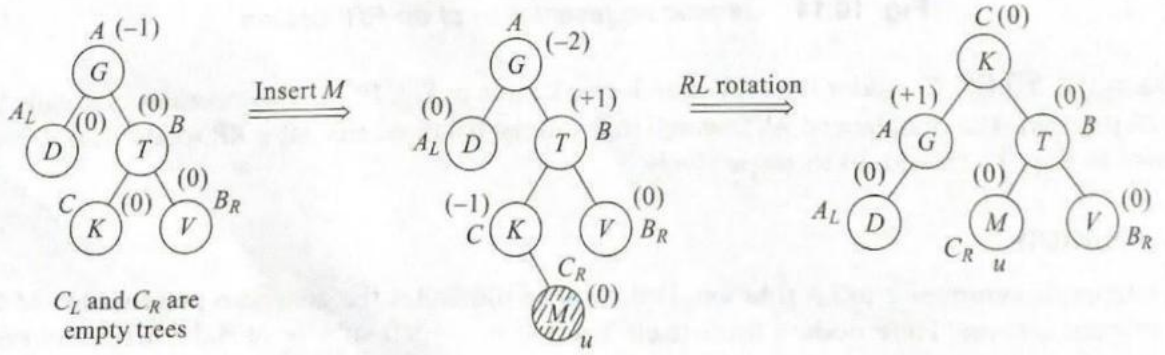


Fig. 10.16 *Generic representation of an RL rotation*

Example:

Consider the AVL search tree shown in Fig. 10.17(a). The insertion of M calls for an RL rotation. The unbalanced AVL search tree and the balanced tree after RL rotation have been shown in Figs 10.17(b) and 10.17(c) respectively.



(a) Balanced AVL search tree before insertion

(b) Unbalanced AVL search tree after insertion of *M*

(c) Balanced AVL search tree after *RL* rotation

Fig. 10.17 *An example of RL rotation*

Algorithm 10.3: Skeletal procedure to insert an ITEM into an AVL search tree *T*

```
procedure INSERT(T, ITEM)
                    /*Steps to insert an ITEM into an AVL search tree T.
                      The node structure comprises the fields LCHILD, DATA,
                      BF and RCHILD representing left child link, data,
                      balance factor and right child link*/
call GETNODE(X);          /* get ready new node X containing ITEM*/
DATA(X)=ITEM,
LCHILD(X)=RCHILD(X)=NIL and BF(X)=0;
                                        /* AVL search tree T is empty*/
if (T=NIL) then { Set T to X;
                  exit;
                  }
                  /* AVL search tree T is non empty and ITEM is distinct
                  from other elements in T */
Find node P where ITEM is to be inserted as either the left child or right
child of P by following a path from the root onwards. Also, while
traversing down the tree in search of the point of insertion of ITEM, take
note of the most recent ancestor node A whose BF(A)= ±1;
Insert node X carrying ITEM as the left or right child of node P;
                  /*if no ancestor node A is found the balance factors of
                  all nodes on the path from the root to the node
                  containing ITEM is 0. The tree will therefore remain
                  balanced even after insertion of ITEM. Merely update the
                  BF fields of all the nodes on the path from the root to
                  node P after insertion of ITEM and exit*/
if (node A not found) then {TEMP = T;
                  /* update BF field of node to +1 if ITEM is inserted in
                  its left subtree and to -1 if inserted in its right
                  subtree*/
                        while ( TEMP <> X) do
                          if (DATA(X) > DATA(TEMP))
```

```
                                then
                                    {BF(TEMP)=-1;
                                    TEMP= RCHILD(TEMP);
                                    }
                                else {BF(TEMP) = +1;
                                    TEMP= LCHILD(TEMP);
                                    }
                                endwhile
                                exit;
                                /* if node A exists and BF(A)= +1 with ITEM
                                inserted in the right subtree of A or BF(A)=
                                -1 with ITEM inserted in the left subtree of
                                A, then set BF(A)=0. Update the balance factors
                                of all nodes in the path from node A to the
                                inserted node X*/
if (node A found)
then
  { if (BF(A)= +1 and ITEM was inserted in the right subtree of A) or
       (BF(A)= -1 and ITEM was inserted in the left subtree of A)
  then {BF(A)=0;
       Update the balance factors of all nodes in the path from node A to
       the inserted node X;
       exit;
       }
}
else
            /* AVL search tree T is unbalanced. Classify the imbalance
            and perform the appropriate rotations*/
  {
       Identify the type of imbalance and apply the appropriate rotations.
       Update the balance factors of the nodes as required by the rotation
       scheme as well as reset the LCHILD and RCHILD links of the
       appropriate nodes.
  }
end INSERT
```

Here,

- LL and RR are called as single rotations and LR and RL are called as double rotations.
- An LR rotation is a combination of RR rotation followed by an LL rotation and
- RL rotation is a combination of LL rotation followed by an RR rotation.

Deletion from an AVL search tree

The deletion operation is based on whether the node t carrying the element to be deleted is either a leaf node or one with a single non empty
subtree or with two non empty subtrees. A delete operation just like an insert operation may also imbalance an AVL search tree after deletion . Just as LL/ LR/ RL/ RR rotations are called for to rebalance the tree after insertion, a delete operation also calls for **rotations categorized as L and R**. While the **L category** is further classified as L0,L1 and L - 1 rotations, the **R category** is further classified as
RO, R1 and R - 1 rotations.

RotaTion free deletion of a leaf node

In the case of deletion of node t which is a leaf node, we physically delete the node and make the child link of its parent, viz., node p null. update the balance factor of node p based on whether the deletion occurred to its right or left.If it had occurred in the right then we increase bf(node p) by 1 or else decrease bf(node p) by 1 of its closest ancestor node.

Example :

Consider the balanced AVL search tree shown in Fig. 10.18. Deletion of 30 is a case of deleting a leaf node. Figure 10.18(a) shows the balanced tree after deletion. Observe how the parent of node 30, viz., the node holding 25 resets its RCHILD link to NIL after the physical deletion of the node holding 30. Since the deletion of key 30 occurred to the right of key 25, bf(25) is increased by 1. Hence the new updated bf(25) is 0.Since the root key 20 is the only ancestor node and after updating the balancing factor bf(20) is increased by 1.

Rotation free deletion of a node having a single subtree:

In the case of deletion of node t with a single subtree, after deleting the node t we need to resetthe child link of the parent node, node p, to point to the child node of node t. The balance factors of node p and or its ancestor nodes are to be updated.

Example:

Deletion of key 25 illustrated in Fig. 10.18(b) is a case of deletion of a node 25 with a single subtree 30. The CHILD link of node 20 which is the root is reset to point to node 30, Since the deletion occurred to the right of node 20, bf(20) is updated to +1. Again the deletion does not call for any rotation to balance the tree.

Rotation free deletion of a node having a two subtree:

In the case of deletion node t having two sub-childrens we need to first replace the DATA(nodet) with *the smallest key of the right subtree of node t or with the largest key of the left subtree of node t*. The smallest key of the right subtree of node t can be obtained by moving right and then moving deep left until the left child link is NIL. Similarly, moving left and then moving deep right until an empty right child link is seen will yield the largest element in the left subtree of node t.Now we physically delete the node holding this replacement value.

Example:

Deletion of a key 15 illustrated in Fig. 10.18(c) is a case of deletion of a node 15 with two subtrees node 12 and node 18. The node 15 is replaced with the largest element in the left subtree node12 or smallest element in the right subtree node 18 both are valid. After deleting the node 15 physically and replacing it with node 18, node 12 becomes the left subtree of the node 18. After updating the balancing factor bf(18) updated to (+1) and no other ancestor nodes changed.

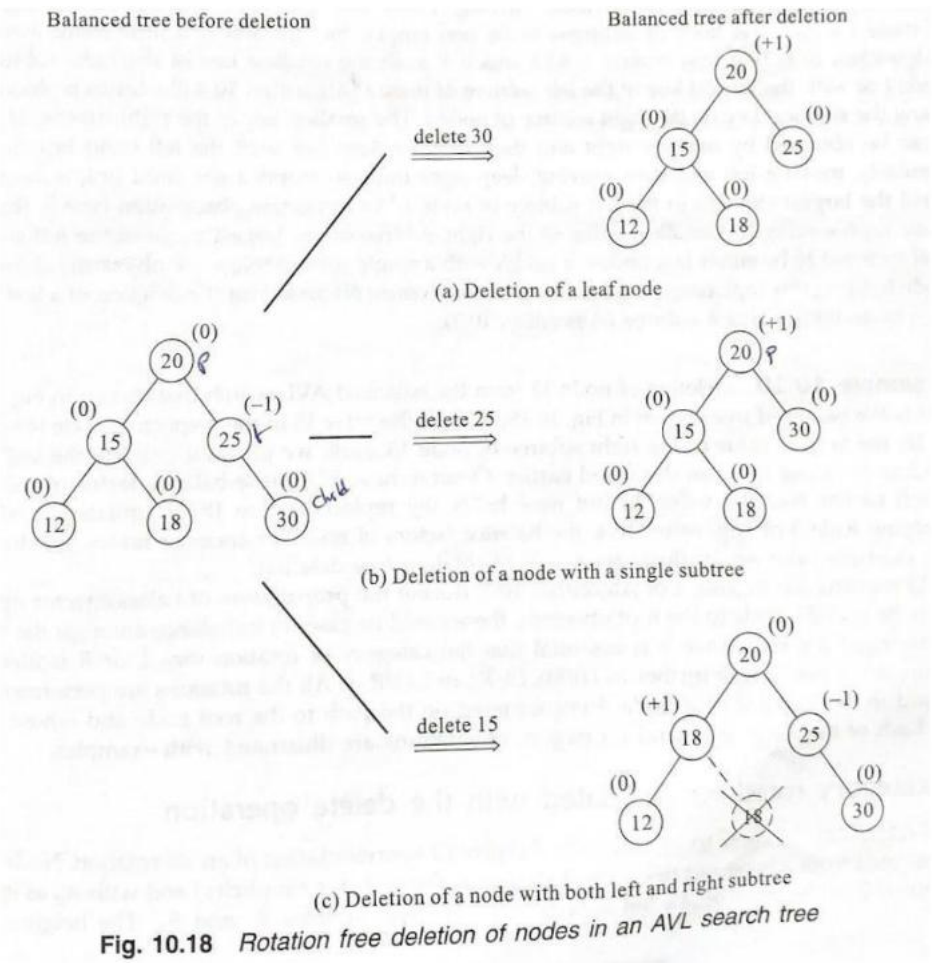


Fig. 10.18 *Rotation free deletion of nodes in an AVL search tree*

Classification of Rotations in Deletion operations of AVL Tree

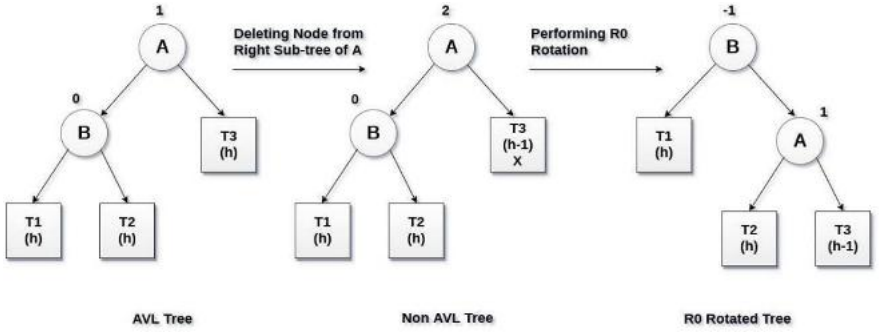
1. R category rotations
2. L category rotations

R category rotations:

R category rotations are further classified into 3 types- R0 rotation, R! Rotation and R-1 rotation.

R0 rotation:

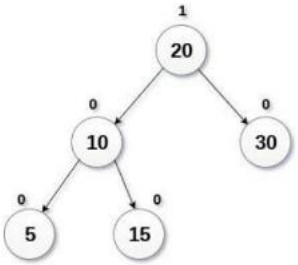
If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting thenode X, then the tree will be rebalanced by rotating tree using R0 rotation.



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.

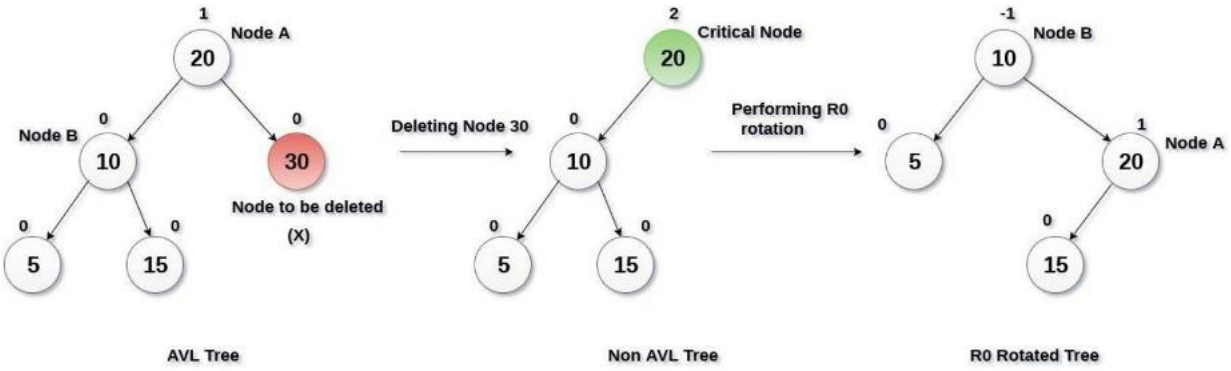
Example:

Delete the node 30 from the AVL tree



Solution

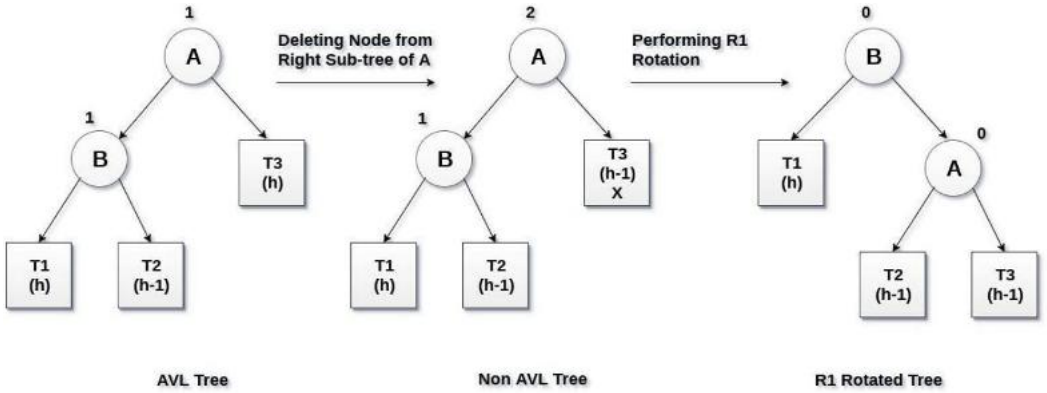
In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



R1 Rotation (Node B has balance factor 1)

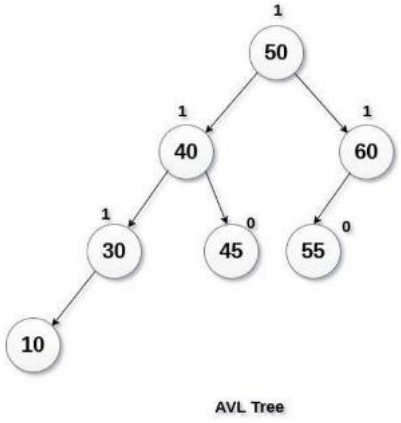
R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

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



Example

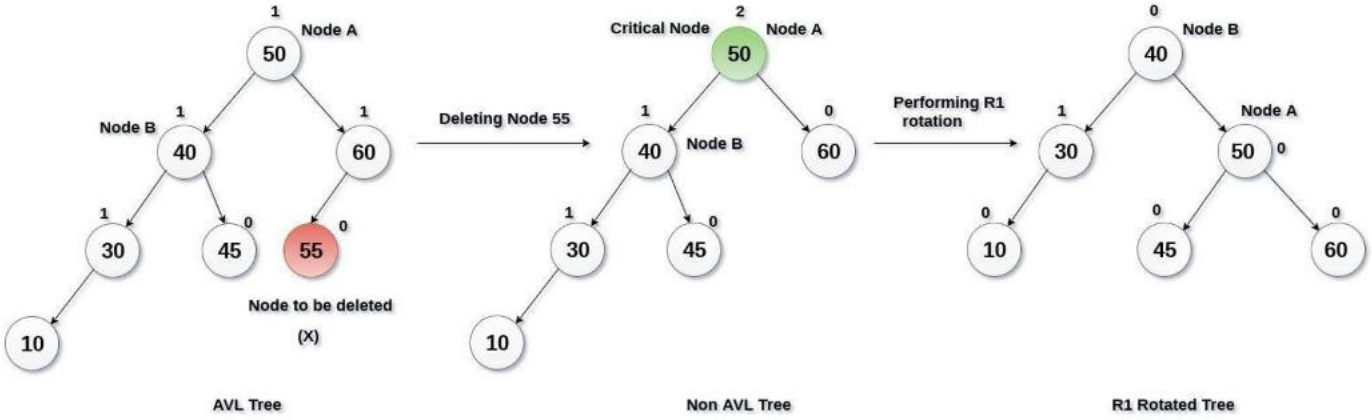
Delete Node 55 from the AVL tree shown in the following image.



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 (shownin 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.

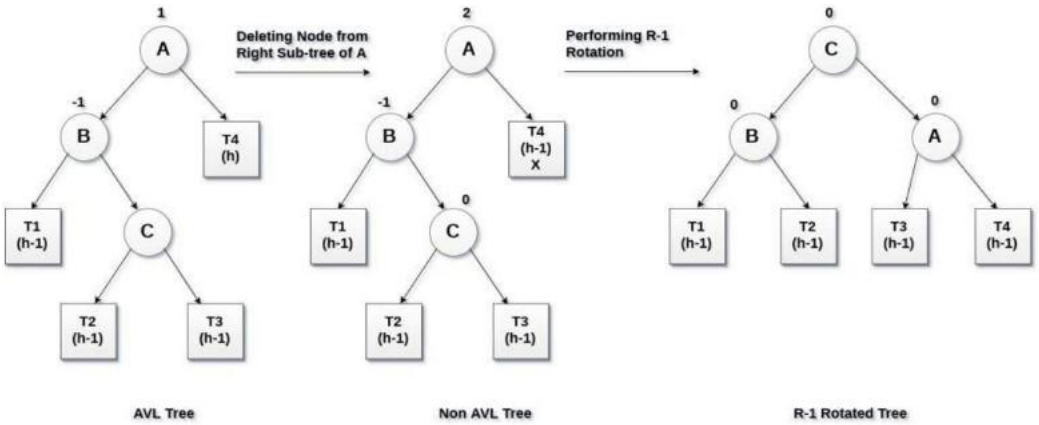


R-1 Rotation

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.

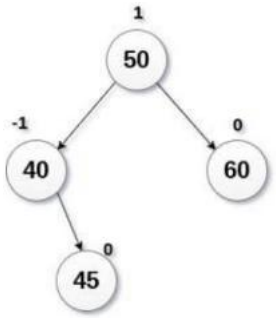
The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

The process involved in R-1 rotation is shown in the following image.



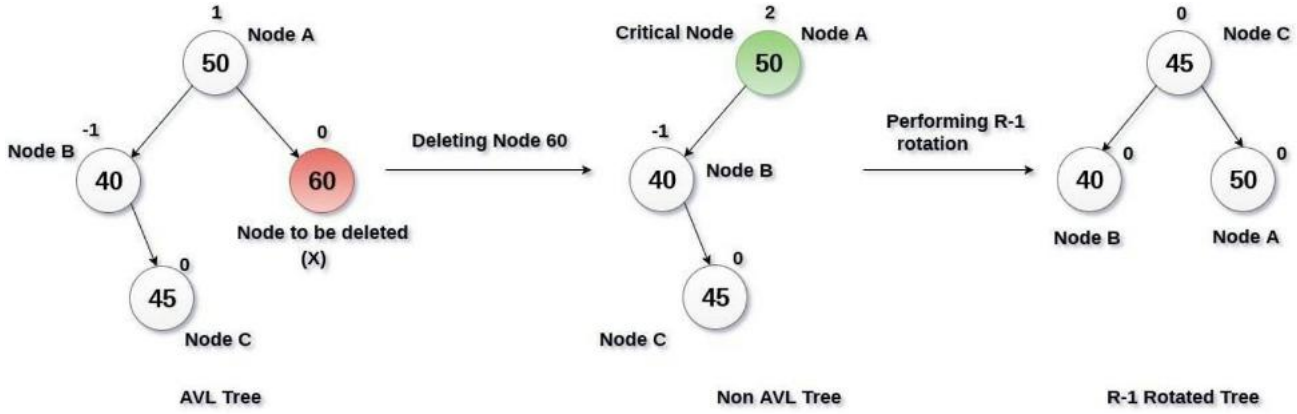
Example

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 AVL Trees

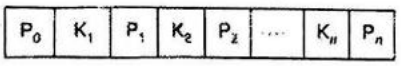
- Most in-memory sets and dictionaries are stored using AVL trees.
- Database applications, where insertions and deletions are less common but frequent data lookups are necessary, also frequently employ AVL trees.
- In addition to database applications, it is employed in other applications that call for better searching.
- A balanced binary search tree called an AVL tree uses rotation to keep things balanced.
- It may be used in games with plotlines as well.
- It is mostly utilized in business sectors where it is necessary to keep records on the employees that work there and their shift changes.

B-TREES

Definition of m-way tree:

Binary search tree (BST) uses the concept of tree indexing, where each node contains a key value, pointers to the left sub-tree and right sub-tree. Binary search tree is, in fact, a 2-way search tree and this concept of tree indexing can be generalized for an
m-way (m >= 2) search tree (m = 2 is the special case for BST) with the following
definitions:

1. An m-way search tree T is a tree in which all the nodes are of degree <=m.
2. Each node in the tree contains the following attributes:



where
$1 \le n < m$
K_i $(1 \le i \le n)$ are the key values in the node
P_i $(0 \le i \le n)$ are pointers to the sub-trees of T.

3. $K_i < K_{i+1}$, $1 \le i < n$
4. All the key values in the sub-tree pointed by P_i are less than the key values K_{i+1}, $0 \le i < n$.
5. All the key values in the sub-tree pointed by P_n is greater than K_n.
6. All the sub-trees pointed by P_i $(0 \le i \le n)$ are also the m-way search trees.

Figure 7.116 illustrates a 3-way search tree.

Definition of B-TREE

B Tree Indexing
B tree is an extension of the m-way search tree. In fact, in order to improve the search efficiency, the tree should be balanced, and if the m-way search tree is height Balanced then it is termed B tree. Following is the formal definition of a B tree.

Definition. A B tree T of order m is an m-way search tree, that is, either it is empty or it satisfies the following properties:

Figure 7.117 represents a B tree of order 3 which satisfies all the above-mentioned properties.

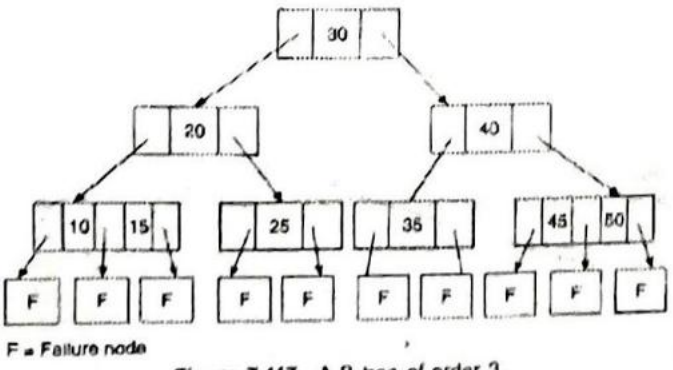


Figure 7.117 A B tree of order 3.

Properties of B-Tree:

1. All leaf nodes are at the same level, which ensures that the tree remains balanced, and the time complexity for operations like insertion, deletion, and search is logarithmic (O(log n)).
2. Internal node -Min and Max Keys Constraints
   - Each node must have at least t - 1 keys.
   - Every node can have at most 2t - 1 keys.
3. Internal node -Min and Max children Constraints
   - Each internal node must have at least t children.
   - Every internal node can have at most 2t children
4. Root node -Min and Max Keys Constraints
   - The root node can have as few as 1 key,
   - Like all other nodes, the root can have at most 2t - 1 keys.
5. Root node -Min and Max Keys Constraints
   - If the root node is not a leaf, it must have at least 2 children (if it has 1 key).
   - The root node can have at most 2t children
6. Keys in each node are stored in sorted order

In a B-Tree, the **m value** represents the **maximum number of children** that a node can have. The relationship between the **minimum degree t** and the **m value** is as follows:

m = 2t

Where:
t is the minimum degree of the B-Tree.
m is the maximum number of children a node can have

Example (For a B-Tree with t = 3):
- Minimum keys per node: t - 1 = 2
- Maximum keys per node: 2t - 1 = 5
- Minimum children per node: t = 3
- Maximum children per node: 2t = 6

1. Searching

Searching for a key value in a B tree is almost same as that of searching a key value in a binary search tree except that in a binary search tree we have to consider nodes which contain one key value whereas a B tree of order m contains at most m - 1 key values (and m children).

Suppose X be the item of search. Then in the case of binary search tree, this X is compared with the values in the node (that is, root node). If there is a match the search is successful, otherwise the search will be propagated to a sub-tree depending on the value of X and value in the current node. Or in other words, let there be a set of key values in a node, say, K1, K2, .., K,, n < m. First we have to perform a
search over it for some i such that Ki <=X < Ki+1;

The following two cases may occur:

Case 1:

X= Ki, that is, match occurs. Then the search is successful and complete.

Case 2:

Ki< X < Ki+1, that is, no match is found. In this case, the search has to be propagated in the sub-tree whose pointer is stored in Pi . If Pi is null then the
search is unsuccessful else repeat the same in the node which is pointed by Pi

As an illustration, let us consider the B tree shown in Below Figure. Consider the case of searching of 35.

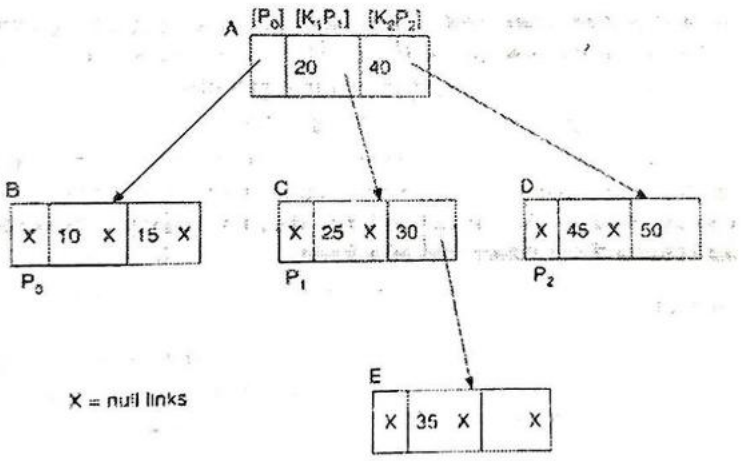


Figure 7.116 A 3-way search tree.

Here the root node is A. 35 is not here, and with 20 < 35 < 40, the next search should be in node C. No match in C also and we have 30 < 35 < ∞, the search therefore proceeds to node E. In node E, 35 is there, hence the search is successful.

As an another example, let us consider the case of searching 22. Searching in node A gives
that 20 < 22 < 40, so the next node to be considered is C. Now, in C it is found that - ∞ < 22 < 25, and P0 field of its points the next node to be searched, but the pointer value in Po in the current node C is null, hence the search is unsuccessful.

2. Insertion Operation in B-Tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

1. **Step 1** - Check whether tree is Empty.
2. **Step 2** - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.
3. **Step 3** - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
4. **Step 4** - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
5. **Step 5** - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
6. **Step 6** - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Alogithms:

```
B-TREE-SPLIT-CHILD(x, i)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i
 3  z.leaf = y.leaf
 4  z.n = t - 1
 5  for j = 1 to t - 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t
 9          z.c_j = y.c_{j+t}
10  y.n = t - 1
11  for j = x.n + 1 downto i + 1
12      x.c_{i+1} = x.c_i
```

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t - 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

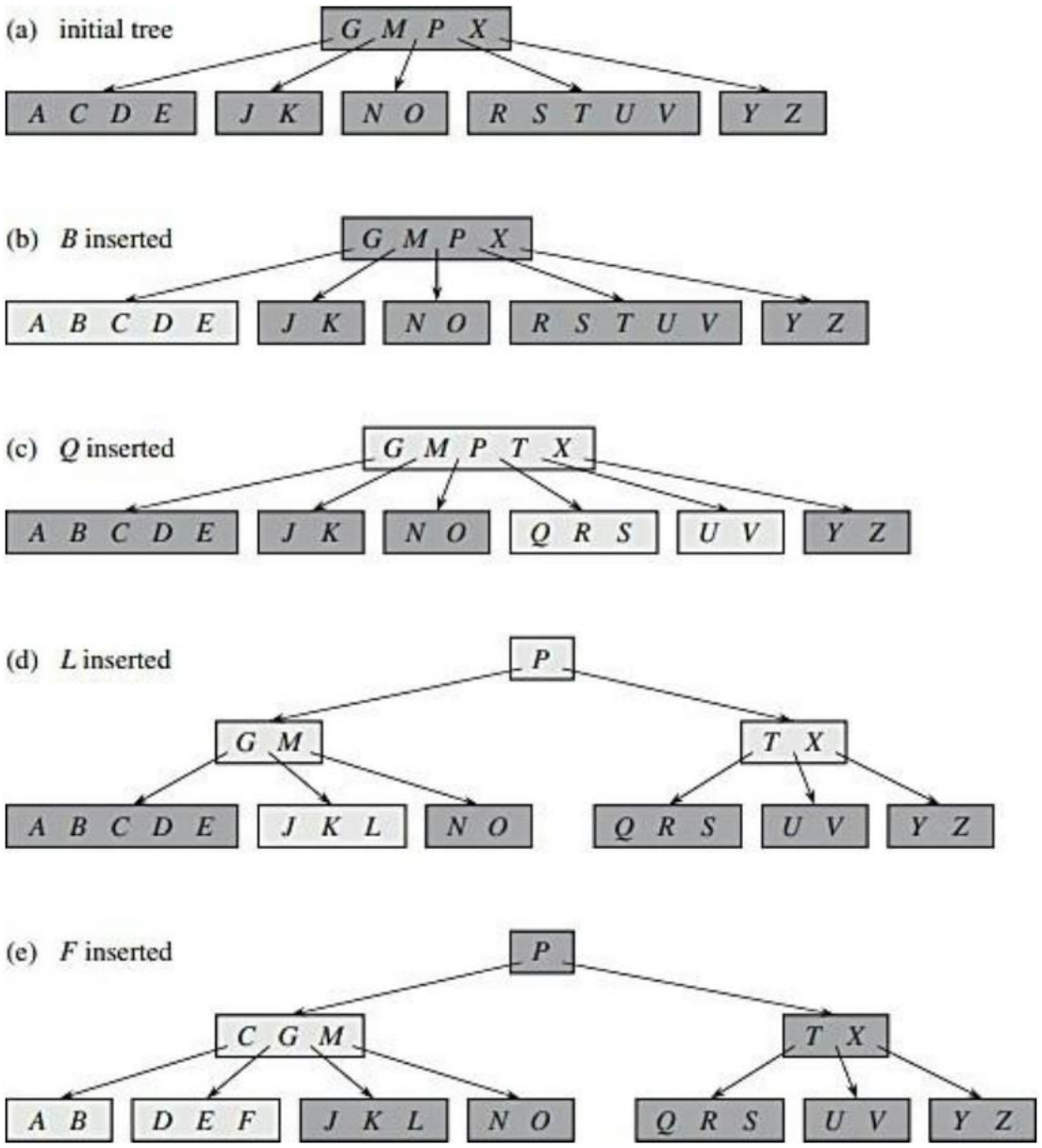


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. **(a)** The initial tree for this example. **(b)** The result of inserting B into the initial tree; this is a simple insertion into a leaf node. **(c)** The result of inserting Q into the previous tree. The node $RSTUV$ splits into two nodes containing RS and UV, the key T moves up to the root, and Q is inserted in the leftmost of the two halves (the RS node). **(d)** The result of inserting L into the previous tree. The root splits right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. **(e)** The result of inserting F into the previous tree. The node $ABCDE$ splits before F is inserted into the rightmost of the two halves (the DE node).

Deleting a key from a B-tree

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.The procedure B-TREE-DELETE deletes the key k from the subtree rooted at x.

Figure 18.8 illustrates the various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following:

   a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

   b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

   c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t - 1$ keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

   a. If $x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

   b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

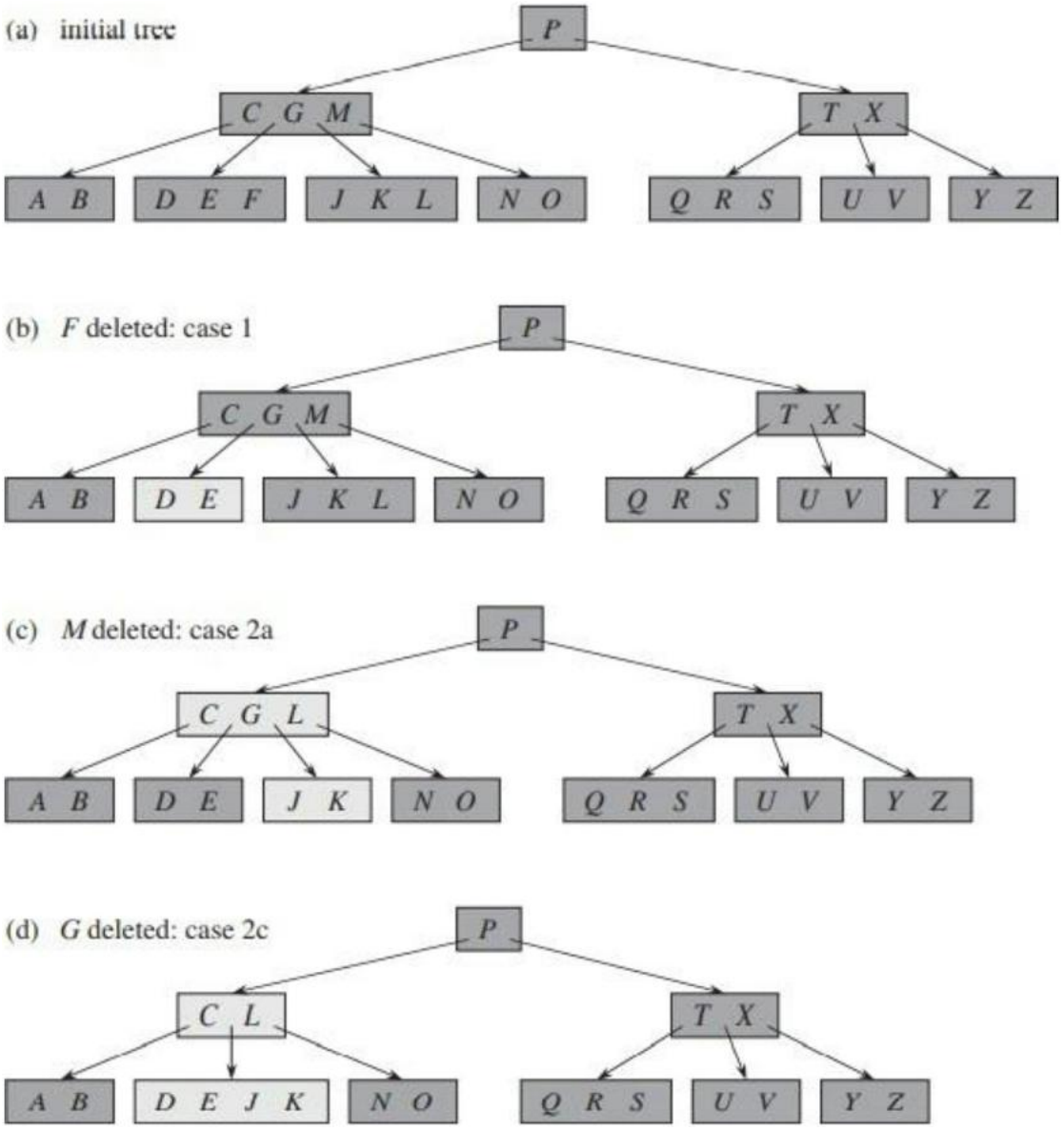


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M moves up to take M's position. **(d)** Deletion of G. This is case 2c: we push G down to make node $DEGJK$ and then delete G from this leaf (case 1).

Applications of B-Trees

- Large databases employ it to access information stored on discs.
- Using the B-Tree, finding data in a data set can be done in a great deal less time.
- Multilevel indexing is possible with the indexing feature.
- The B-tree method is also used by the majority of servers.
- In CAD systems, B-Trees are used to catalogue and search geometric data.
- Other applications of B-Trees include encryption, computer networks, and natural language processing.
- Since accessing values stored in a large database that is stored on a disc takes a long time, B trees are used to index the data and provide quick access to the actual data stored on the disks.
- In the worst case, it takes O(n) running time to search a database with n key values that is not sorted or indexed. However, if we use B Tree to index this database, it will be searched in O (log n) time in worst case.

PRIORITY QUEUES

Any data structure that supports the operations of search min (or max), insert, and delete min (or max, respectively) is called a priority queue

The simplest way to represent a priority queue is as an unordered linear list .Suppose that we have n elements in this queue then insertion and deletion operations takes O(n) in worst case and best case will be O(1). .When a max heap is used, both additions and deletions can be performed in O(log n) time

Heaps

Definition: A max (min)heap is a complete binary tree with the Property that the value at each node is at least as large as (or as small as) the values at its children (if they exist). Call this property the "heap property."

Operations in Heap

Insertion To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, great-grandparent, and so on, until it is less than or equal to one of these values.

Below example shows, how the **Insert** operation would add a new value into an existing heap of six elements. This shows the step-by-step process of placing the new element "at the bottom" of the heap, followed by adjustments to maintain the heap property.

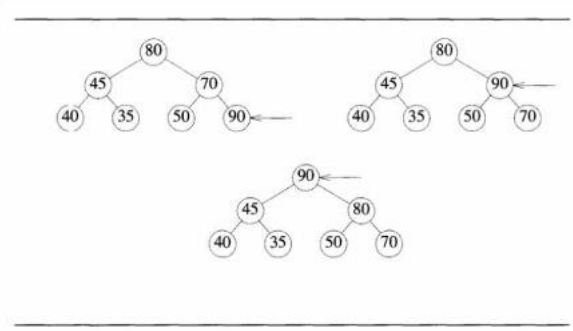


Figure 2.14 Action of Insert inserting 90 into an existing heap

Algorithm Insert (Algorithm 2.8) describes this process in detail

```
1   Algorithm Insert(a, n)
2   {
3       // Inserts a[n] into the heap which is stored in a[1 : n − 1].
4       i := n; item := a[n];
5       while ((i > 1) and (a[⌊i/2⌋] < item)) do
6       {
7           a[i] := a[⌊i/2⌋]; i := ⌊i/2⌋;
8       }
9       a[i] := item; return true;
10  }
```

Algorithm 2.8 Insertion into a heap

Deletion To delete the maximum key from the max heap, we use an algorithm called Adjust. Adjust takes as input the array a[]and the integers iii and n. It regards a[1:n] as a complete binary tree. If the subtrees rooted at 2i and 2i+1 are already max heaps, then Adjust will rearrange elements of a[] such that the tree rooted at i is also a max heap. The maximum element from the max heap a[1:n] can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, that is, a[n], is copied to the root, and finally, we call Adjust(a,1,n−1). Both Adjust and DelMax are described in Algorithm 2.9.

```
1   Algorithm Adjust(a, i, n)
2   // The complete binary trees with roots 2i and 2i + 1 are
3   // combined with node i to form a heap rooted at i. No
4   // node has an address greater than n or less than 1.
5   {
6       j := 2i; item := a[i];
7       while (j ≤ n) do
8       {
9           if ((j < n) and (a[j] < a[j + 1])) then j := j + 1;
10              // Compare left and right child
11              // and let j be the larger child.
12          if (item ≥ a[j]) then break;
13              // A position for item is found.
14          a[⌊j/2⌋] := a[j]; j := 2j;
15      }
16      a[⌊j/2⌋] := item;
17  }

1   Algorithm DelMax(a, n, x)
2   // Delete the maximum from the heap a[1 : n] and store it in x.
3   {
4       if (n = 0) then
5       {
6           write ("heap is empty"); return false;
7       }
8       x := a[1]; a[1] := a[n];
9       Adjust(a, 1, n − 1); return true;
10  }
```

Algorithm 2.9 Deletion from a heap

.Therefore, if there are n elements in a heap, deleting the maximum can be done in O(log n)time.

Sorting To sort n elements, it suffices to make n insertions followed by n deletions from a heap. Algorithm 2.10 has the details. Since insertion and deletion take O(log n) time each in the worst case, this sorting algorithm has a time complexity of 0(n log n).

```
1  Algorithm Sort(a, n)
2  // Sort the elements a[1 : n].
3  {
4      for i := 1 to n do Insert(a, i);
5      for i := n to 1 step -1 do
6      {
7          DelMax(a, i, x); a[i] := x;
8      }
9  }
```

Algorithm 2.10 A sorting algorithm

Heapify:

Heapify is an operation used in heap data structures (typically binary heaps) to maintain the heap property, ensuring that each parent node in the heap is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) its child nodes. This property allows efficient retrieval of the maximum or minimum element, depending on the type of heap.

How Heapify Works

Heapify is usually called on a node in the heap and operates as follows:

1. **Identify the Node to Heapify**: Start with a node that may violate the heap property due to insertion, deletion, or modification.
2. **Compare with Children**: Compare the node's value with its children's values. For a max heap, check if the node is less than either of its children. For a min heap, check if it's greater.
3. **Swap if Necessary**: If the heap property is violated, swap the node with its largest (in max heap) or smallest (in min heap) child.
4. **Recursively Heapify**: After swapping, recursively call Heapify on the child where the swap occurred, continuing downwards until the heap property is restored.

Types of Heapify Operations

1. **Upward Heapify (Bubble-Up)**: Used when a new element is added at the end of the heap. This element is moved up to maintain the heap property.
2. **Downward Heapify (Sink-Down)**: Used when the root or another node is replaced or removed, typically after deleting the maximum or minimum element. This operation "sinks" the node down to maintain the heap property.

```
1  Algorithm Heapify(a, n)
2  // Readjust the elements in a[1 : n] to form a heap.
3  {
4      for i := ⌊n/2⌋ to 1 step -1 do Adjust(a, i, n);
5  }
```

Algorithm 2.11 Creating a heap out of n arbitrary elements

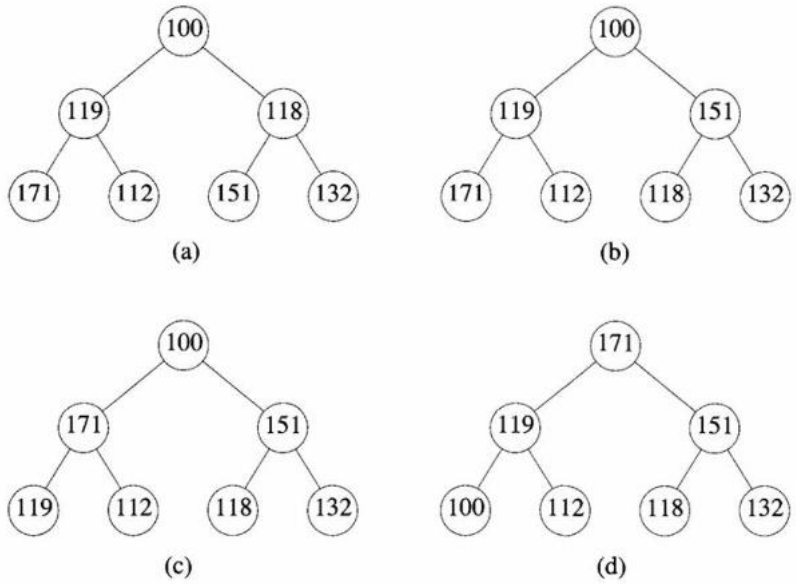


Figure 2.16 Action of Heapify$(a, 7)$ on the data (100, 119, 118, 171, 112, 151, and 132)

II. Graphs

Defination:- A Graph G consist of two sets

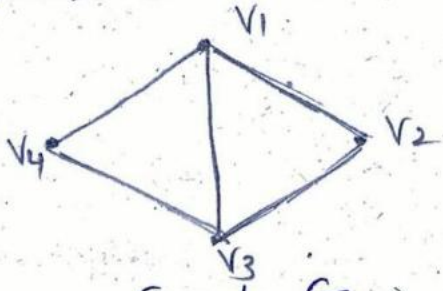
i) A set V, called the set of all vertices or nodes

ii) A set E, called the set of all edges or arcs. this set E is set of all pairs of elements from V.

For example, consider the Graph G1

$V = \{V_1, V_2, V_3, V_4\}$

$E = \{(V_1, V_2), (V_1, V_3), (V_1, V_4), (V_2, V_3), (V_3, V_4)\}$



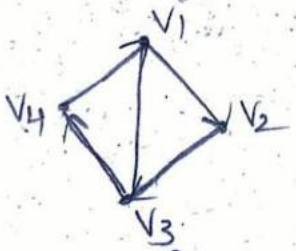
Graph G1

2M

Graph Terminologies:-

1. **Diagraph:-** A Diagraph is also called as directed graph G1. Such that $G1 = \{V, E\}$ where V is set of vertices and E is set of edges.

Ex:

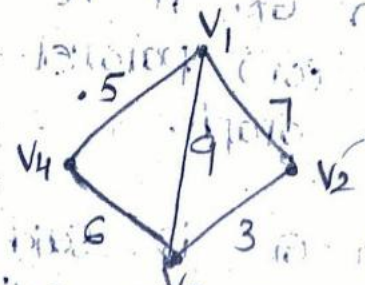


$V = \{V_1, V_2, V_3, V_4\}$

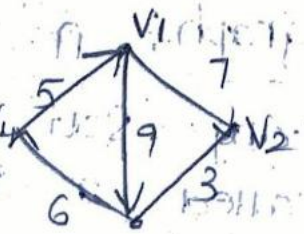
$E = \{(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_1, V_4), (V_1, V_3)\}$

2. **weighted graph:-** If all the edges Labelled with some weights then that graph is termed as weighted graph.

Ex:-



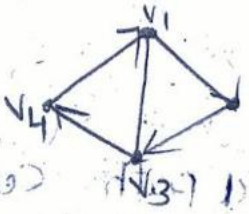
G1



G2

3. Adjacent Vertices:- A vertex V_i is adjacent to (Neighbours of) another vertex V_j, if there is an edge from V_i to V_j.

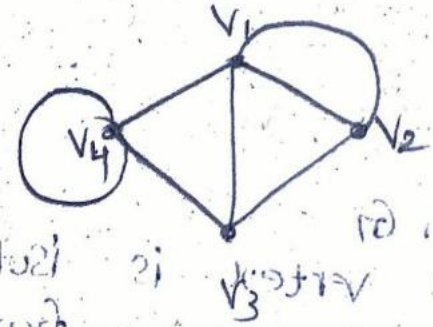
Ex:-



Ex:- V_3 adjacent node : V_4

4. Self loop:- If there is an edge whose starting and end vertices are same. That is in edge, then it is called as self loop. or simply loop.

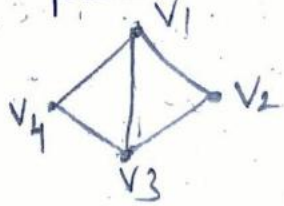
Ex:-



G2

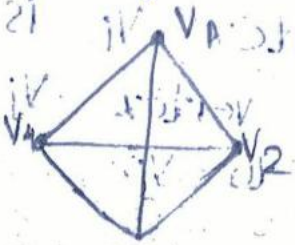
5. parallel edges:- If there is more than one edge between the same pair of vertices, then there are known as parallel edges.

Ex:- parallel edges : V_1 and V_2



Simple graph:- A graph G, if it doesn't have any self loop (or) parallel edges is called a simple graph.

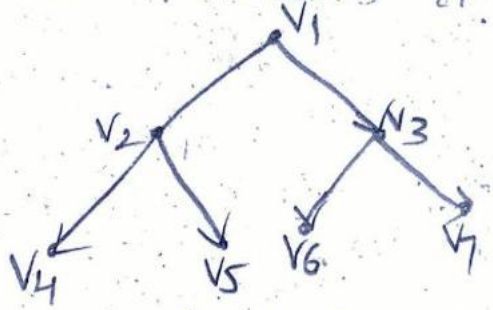
Complete graph:- A Guraph G is said to be complete if each vertex V_i is adjacent to every other vertex V_j.



G_{I_2}

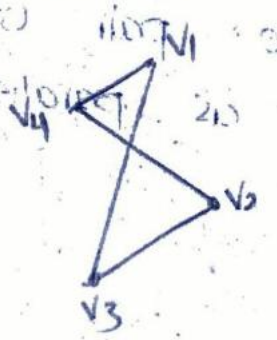
Acyclic path:- If there is a path containg one or more edges which starts from a vertex V_i and terminate into the some vertex then the path is known as cycle.

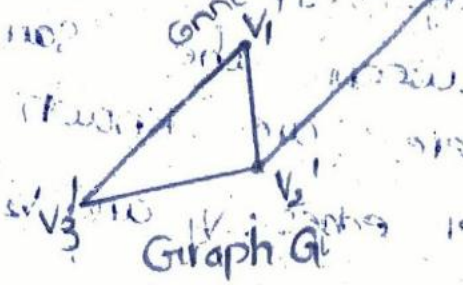
If a graph doesn't have any cycle then it is called Acyclic graph.



Giraph G1

a) Isolated Vertex:- A vertex is isolated If there is no edge connected from any other vertex to vertex.





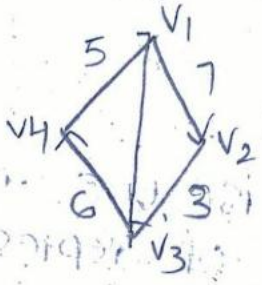
Graph G

10) Degree of Vertex:-

The number of edges connected with vertex is the Degree of Vertex. For a Diagraph there are two degrees. "Indegree and outdegree

Indegree is denoted by indegree (V_i) - number of edges incident into V_i outdegree (V_i) emanating from V_i

Eg:-

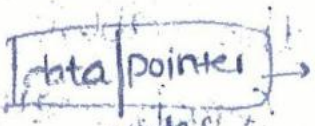


indegree (V_1) = 2
indegree (V_2) = 2
indegree (V_3) = 1
indegree (V_4) = 0

out degree (V_1) = 1
out degree (V_2) = 0
outdegree (V_3) = 2
out degree (V_4) = 2

11) pendent Vertex:- A Vertex V_i is pendent if its indegree of (V_i) equals to one and outdegree of (V_i) equals to zero.

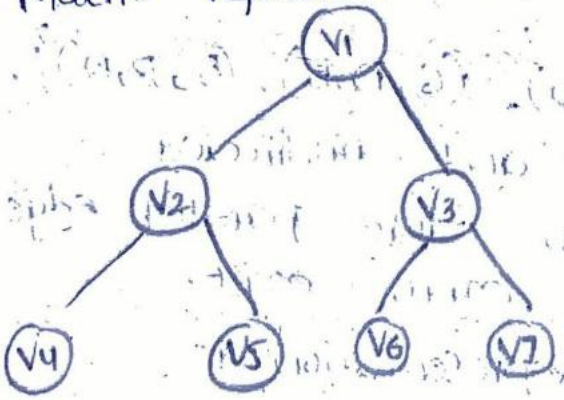
12) connected Graph:- In a Graph G Two vertices V_i and V_j are said to be connected if there is a path in G from V_i to V_j.



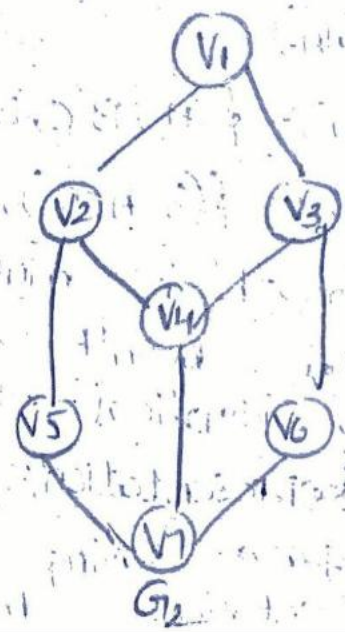
Node's collection of nodes

Representation of Graphs:-

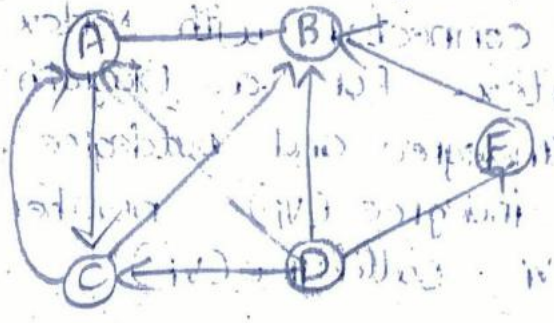
1. set representation
2. Linked representation
3. Matrix representation



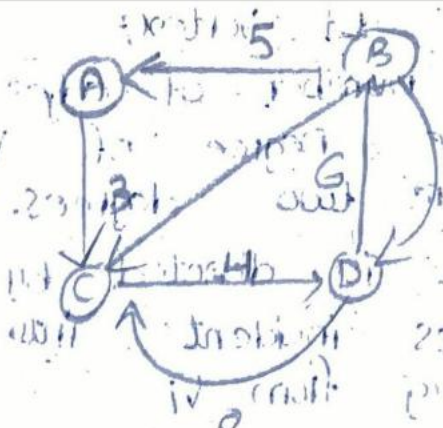
G1



G2



G_3



G_4

① Set Representation:- This is one of the straight forward method of representing a graph. In this method we have two sets

① V, set of Vertices

② E, set of Edges (VXV);

If the Graph is unpaited Graph then the set E consist of ordered collection of three tuples $E \subseteq V \times V \times V$

From the above Graph,

Graph G_1

$V(G_1) = \{V_1, V_2, V_3, V_4, V_5, V_6, V_7\}$

$E(G_1) = \{(V_1, V_2), (V_1, V_3), (V_2, V_4) \ldots\}$

Graph (G_4)

$V(G_4) = \{A, B, C, D\}$

$E(G_4) = \{(3, A, C), (4, C, D), (6, D, B), (5, B, A)\}$.

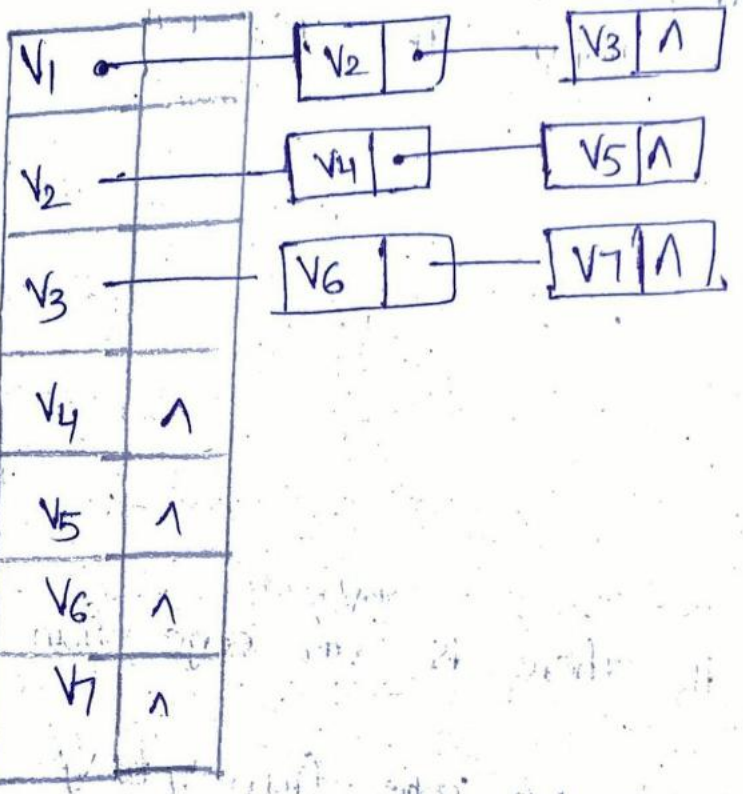
If the Graph is multigraph and undirected this method doesn't allow to store parallel edges and two identical elements cannot exist.

Linked Representation:- This Representation is another space saving way of representation. In this representation two types of node structures are there

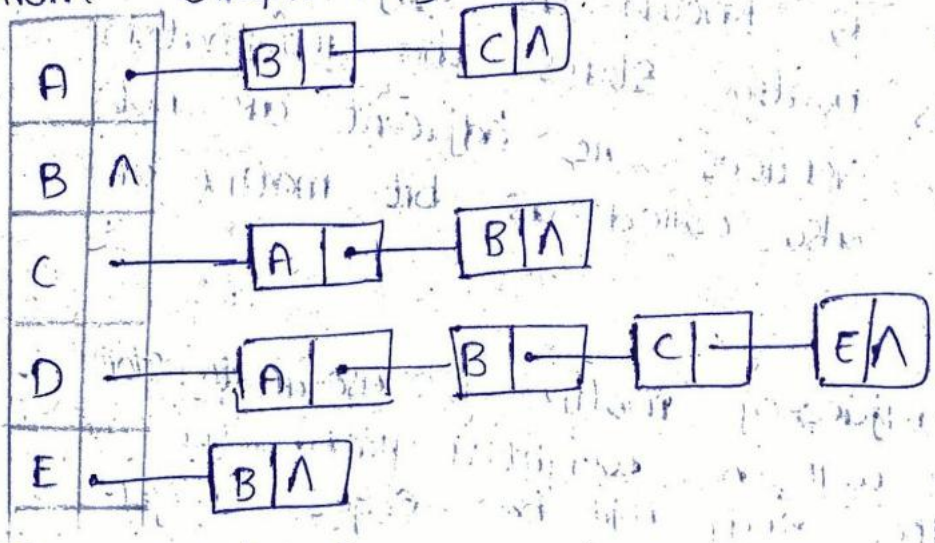
NODE - LABEL	ADJ - LIST

WEIGHT	NODE - LABEL	ADJ LIST

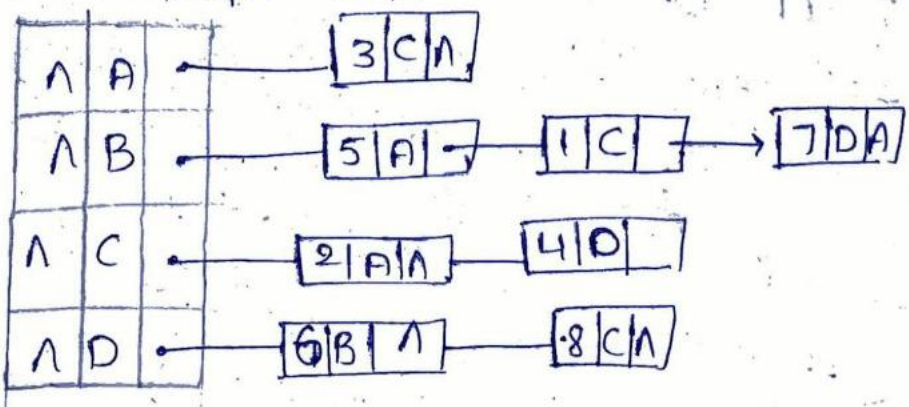
From the Graph G_1



from Graph G_3

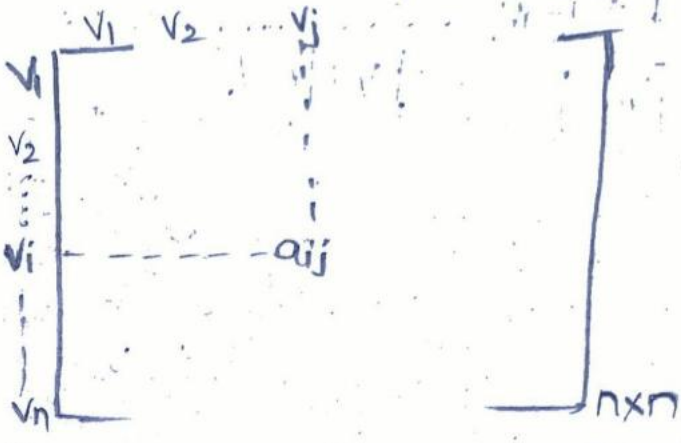


From Graph G_4



Matrix representation:-

Is the most useful way of representing any graph. This representation uses a squre Matrix of order $n \times n$, where n is the no. of vertices in a graph. The general representation is

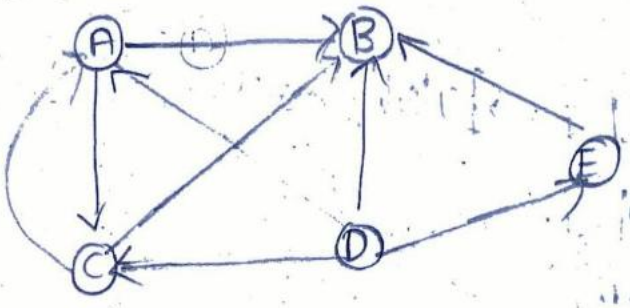


Here, $a_{ij} = 1$ if there is an edge from V_i to V_j.

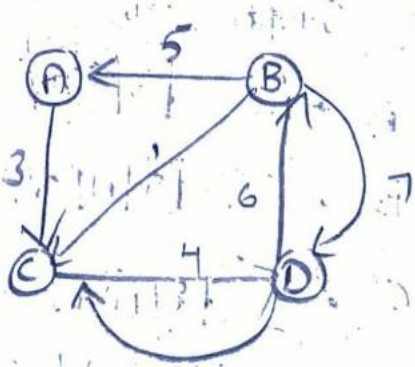
$a_{ij} = 0$. If there is no edge from V_i to V_j.

This matrix is known as, adjacency matrix because this matrix stores the information whether two vertices are adjacent are not. This matrix also called as bit matrix or Boolean matrix.

This adjacency matrix is useful to store multigraph as well as weighted graph. In case of multigraph, The entry will be edges weight instead of '1' or '0'.



G_1



G_2

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	0	0
C	1	1	0	0	0
D	1	1	0	0	1
E	0	1	0	0	0

G_{11}

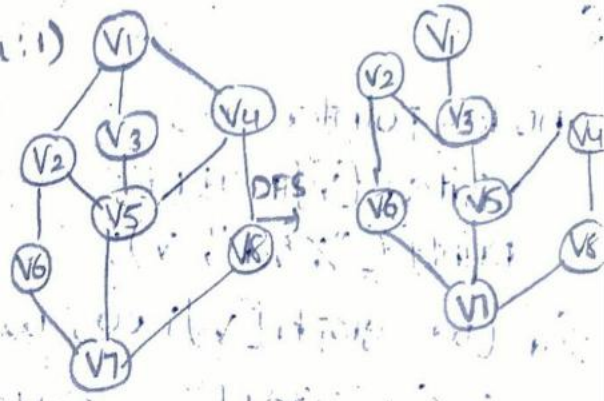
	A	B	C	D
A	0	0	3	0
B	5	0	1	9
C	0	0	0	4
D	7			

G_{12}

Operations on Graph:

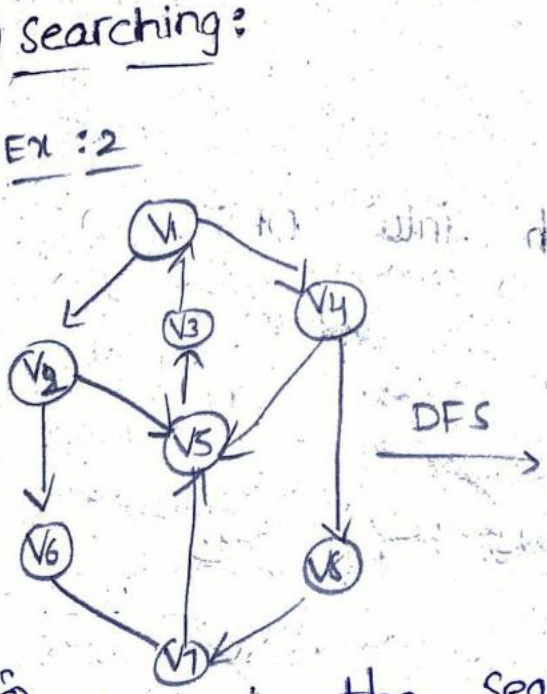
1) Searching
2) connected components.
3) Biconnected components.

Ex: 1)



1) Searching:

Ex: 2



① we begin the search by visiting the starting vertex.
② we select an unvisited vertex w from these adjacency list and carry out depth first search and w.
③ we preserve our current position in these adjacency list by placing it on a stack.
④ Finally our search reaches a vertex that has no adjacency vertex list.
⑤ At this point, we remove a vertex from the stack and continue processing its adjacency list.

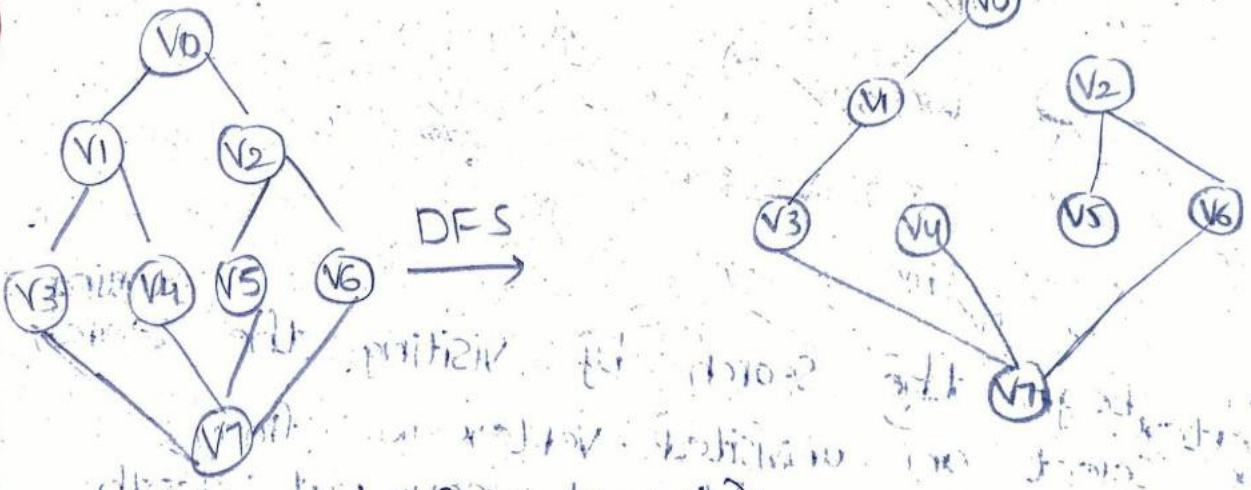
⑥ Then previously visited vertices are discarted and unvisited vertices are visited and placed on the stack.

Algorithm:

pointer:- pointer a variable which adresses the another variable

```
Void dfs(int V)
{
  node_pointer w;
    visited[V] = TRUE;
    printf("%5d", V);
 for(w = graph[V]; w; w = w→link)
    if(!visited[w→vertex])
        dfs(w→vertex);
}
```

Ex:- convert the Given Graph into DFS



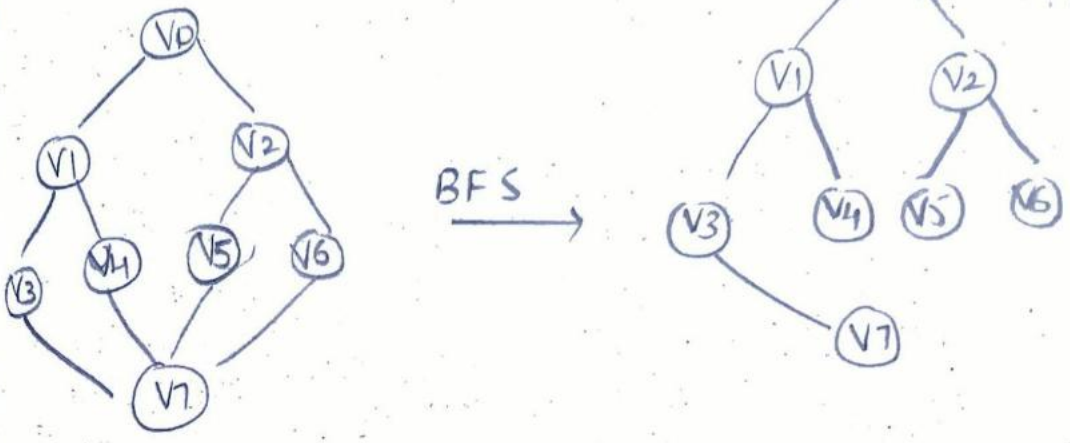
BFS:- (Breadth first search):

① BFS starts at vertex 'V' and mute it as visited ② visit each of the vertices of these V's adjacency list.

③ when we visit all the unvisited vertices that are adjacent to the first vertex of these V's adjcancy his adjacency list.

④ To implement this, after visiting each vertex we place it in a queue.

⑤ Search will be finished when the queue is empty.



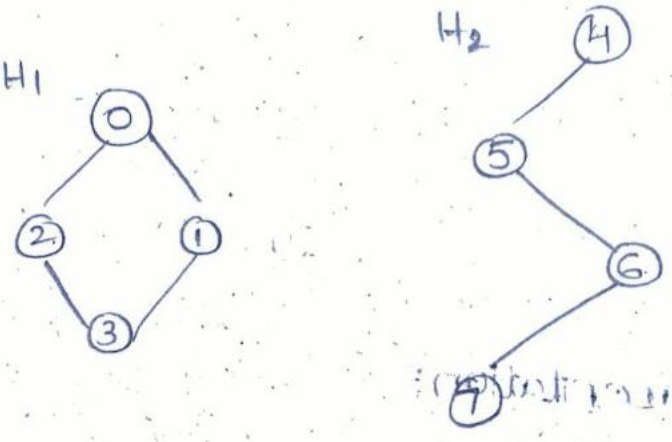
Algorithm (or) Implementation:

```
Void bfs (int v)
    node - pointer w;
    queue - pointer front, rear;
    front = rear = NULL;
    printf ("%5d", v);
    visited [v] = TRUE;
    addq (&Front, &rear, v);
    while (front)
    {
      v = deleteq (&front);
    for (w=graph[v]; w; w=w→Link)
    {
      if (!visited [w→Vertex])
        {
          printf ("%5d", w→vertex);
          addq (&front, &rear, w→vertex);
          visited [w→ vertex] = TRUE;
        }
    }
```

Connected components:-

Connected component is a subgraph which refers to a group of vertices that are connected to each other through edges, but not connected to other vertices outside the group.

Ex:-



Graph G1

We can implement this operation (whether (or) not an undirected graphs is connected) by simply calling either BFS or DFS and then determine If there are any unvisited vertices.

From the above example the connected components are two

$H_1 = \{0, 1, 2, 3\}$

$H_2 = \{4, 5, 6, 7\}$

∴ we can conclude that the Graph G1 is not connected. The computing time for this operation is $O(n+e)$

Implementation:-

```
Void connected (void)
{
    int i;
    for (i=0; i<n; i++)
```

```
    if (! visited [i])
    {
      dfs (i);
      printf ("\n");
    }
}
```

Biconnected Components & Articulation points

An articulation point is a vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph, G', that has at least two connected com ponents. For example, the connected graph of Figure 6.22 has four articulation points, vertices 1, 3, 5, and 7

A biconnected graph is a connected graph that has no articulation points. For example, the graph of Figure 6.19 is biconnected, while the graph of Figure 6.22 obviously is not. In many graph applications, articulation points are undesirable. For instance, suppose that the graph of Figure 6.22(a) represents a communication network.

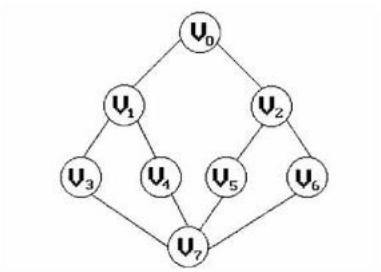


Figure 6.19 biconnected Graph

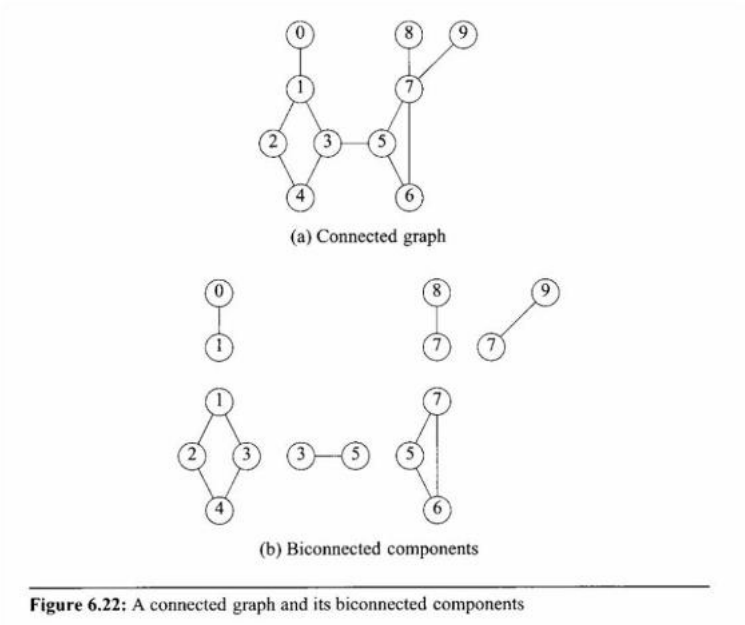


Figure 6.22: A connected graph and its biconnected components

In such graphs, the vertices represent communication stations and the edges represent communication links. Now suppose that one of the stations that is an articulation point fails. The result is a loss of communication not just to and from that single station, but also between certain other pairs of stations.

For example, the graph of Figure 6.22(a) contains the six biconnected components shown in Figure 6.22(b). The biconnected graph of Figure 6.19, however, contains just one biconnected component: the whole graph. It is easy to verify that two biconnected components of the same graph have no more than one vertex in common. This means that no edge can be in two or more biconnected components of a graph. Hence, the biconnected components of G partition the edges of G.

We can find the biconnected components of a connected undirected graph, G, by using any depth first spanning tree of G.

- For example, the function call dfs (3) applied to the graph of Figure 6.22(a) produces the spanning tree of Figure 6.23(a).
- We have redrawn the tree in Figure 6.23(b) to better reveal its tree structure. The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search.
- We call this number the depth first number, or dfn, of the vertex. For example, dfn (3) = 0, dfn (0) = 4, and dfn (9) = 8.

- Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower dfn than either of these vertices.
- Generally, if u and V are two vertices, and u is an ancestor of v in the depth first spanning tree, then dfn (w) < dfn (v)

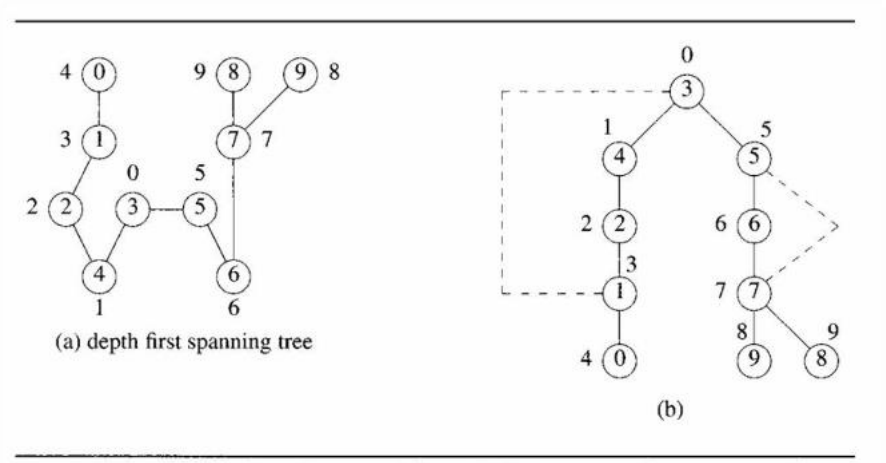


Figure 6.23: Depth first spanning tree of Figure 6.22(a)

Steps to find Articulation Points :

Step 1 : Perform Depth first search, starting at any vertex

Step 2 : Number the vertex as they are visited, as Num (v).

Step 3 : Compute the lowest numbered vertex for every vertex v in the Depth first spanning tree, which we call as low (w), that is reachable from v by taking zero or more tree edges and then possibly one back edge. By definition, Low(v) is the minimum of

(i) Num (v)

(ii) The lowest Num (w) among all back edges (v, w)

(iii) The lowest Low (w) among all tree edges (v, w)

Step 4 : (i) The root is an articulation if and only if it has more than two children.

(ii) Any vertex v other than root is an articulation point if and only if v has same child w such that Low (w) ≥ Num (v), The time taken to compute this algorithm an a graph is $0(|E|+|V|)$.

Note

For any edge (v, w) we can tell whether it is a tree edge or back edge merely by checking Num (v) and Num (w).

If Num (w) > Num (v) then the edge is a back edge.

Application of Graphs

- To study Dijkstra's Algorithm, Prims's Algorithm & Kruskal's Algorithm
- Graphs are used to define the **flow of computation**.
- Graphs are used to represent **networks of communication**.
- Graphs are used to represent **data organization**.
- Graph transformation systems work on rule-based in-memory manipulation of graphs. Graph databases ensure **transaction-safe, persistent storing and querying of graph structured data**.
- Graph theory is used to find **shortest path in road** or a network.

DIVIDE AND CONQUER

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 subproblems. These subproblems must be solved,and then a method must be found to combine subsolutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied until splitting is not possible in sub problems.

Example: Detecting a Counterfeit Coin

You are given a bag with 16 coins and told that one of these coins may be counterfeit. Further, you are told that counterfeit coins are lighter than genuine ones. Your task is to determine whether the bag contains a counterfeit coin. To aid you in this task, you have a machine that compares the weights of two sets of coins and tells you which set is lighter or whether both sets have the same weight.

Approach 1: We can compare the weights of coins 1 and 2. If coin 1 is lighter than coin 2, then coin 1is counterfeit and we are done with our task. If coin 2 is lighter than coin 1, then coin 2 is counterfeit. If both coins have the same weight , we compare coins 3 and 4. Proreeding in this way, we can determine whether the bag contains? counterfeit coin by making at most eight weight comparisons. This process also identifies the counterfeit coin.

Approach 2: the divide-and-conquer methodology. Suppose that our 16-coin instance is considered a large instance. So it is divided into two 8- coin instances A and B as above. By comparing the weights of these two instances, we determine whether or not a counterfeit coin is present. If not, the algorithm terminates. Otherwise, we continue with the subinstance known to have the counterfeit coin. Suppose B is the lighter set. It is divided into two sets of four coins each. Call these sets B1 and B2. The two sets are compared. One set of coins must be lighter. If B1 is lighter, the counterfeit coin is in B1 and B1 is divided into two sets of two coins each. Call these sets Bla and B1b. The two sets are compared, and we continue with the lighter set. Since the lighter set has only two coins, it is a small instance. Comparing the weights of the two coins in the lighter set, we can determine which is lighter. The lighter one is the counterfeit coin.

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
9       }
10  }
```

Algorithm 3.1 Control abstraction for divide-and-conquer

Performance analysis

Here, we can use a algorithm called **controlabstraction** based on divide- and-conquer. The DAndC is is initially invoked as DAndC(P), where P is the problem to be solved. Small(P) is a Boolean- valued function that determines whether the input size is small enough that the answer can be computed without splitting.

If this is so, the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These subproblems P1, P2,..., Pk. are solved by recursive applications of DAndC. Combine is a function that determines the solution to P using the solutions to the k subproblems. If the size of P is n and the sizes of the k subproblems are n1, n2,. .., nk, respectively, then the computing time of DAndC is described by the recurrence relation.

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

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

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where,a and b are known constants. We assume that T(1) is known and n is a power of b

Quick sort

Quick sort procedure formulated by C.A.R. Hoare belongs to the family of sorting by exchange ortransposition where elements that are out of order are exchanged amongst themselves to obtain thesorted list.

The procedure works on the principle of partitioning the unordered list into two sub lists at every stage of the sorting process based on what is called a ***pivot element.*** The two sub lists occur to the left and right of the pivot element. The pivot element is placed at appropriate position in the sorted list. That is all left side values are smaller to Pivot and all right-side values are greater value to the Pivot. After placing the element pivot element in proper place then we need to perform **divide- and conquer**

Partitioning

Consider an unordered list L=(K1, K2, K3,.. Kn). Let us choose K, to be the pivot element. Now K, compares itself with each of the keys on a left to rightencounter looking for the first key Ki <=K2 ≥ K3.

The pivot element is placed at appropriate position in the sorted list.

- That is all left side values are smaller to Pivot and
- All right side values are greater value to the Pivot.

Algorithm 16.7: Procedure for Partition

```
procedure PARTITION(L, first, last, loc )
              /* L[first:last] is the list to be partitioned. loc is the
                 position where the pivot element finally settles down*/
left = first;
right = last+1;
pivot_elt = L[first];   /* set the pivot element to the first
                           element in list L*/
while (left < right) do
    repeat
          left = left+1;     /* pivot element moves left to right*/
    until L[left] ≥ pivot_elt;
    repeat
          right = right -1; /* pivot element moves right to left*/
    until L[right] ≤ pivot_elt;
    if (left < right) then swap(L[left], L[right]); /*arrows face each
          other*/
end
loc = right
swap(L[first], L[right]);      /* arrows have crossed each other - exchange
                                  pivot element L[first]   with L[right]*/
end PARTITION.
```

Example:

Let us quick sort the list L= {5, 1, 26, 15, 76, 34, 15}.

The various phases of the sorting process are shown in. When the partitioned sub lists contain only one element then no sorting is done. Also, in phase 4 of Fig. below observe how the pivot element 34 exchanges with itself.

The final sorted list is (1, 5, 15, 15, 26, 34, 76)

Quick sort procedure:

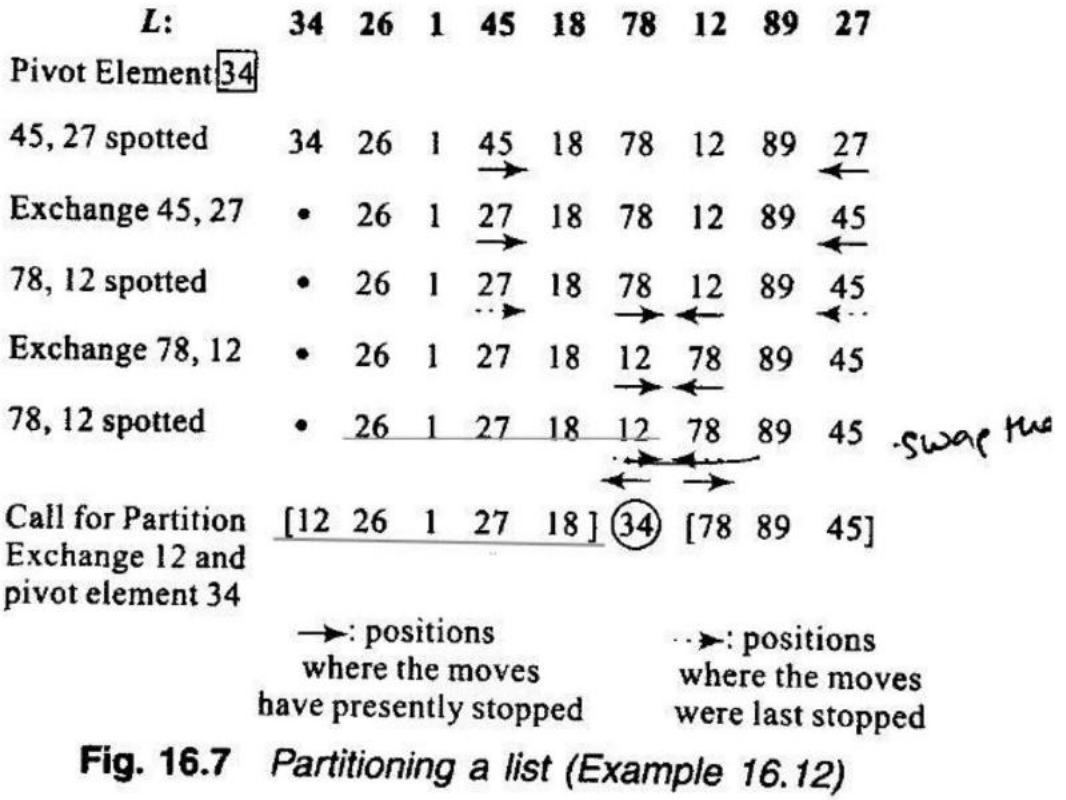


Fig. 16.7 *Partitioning a list (Example 16.12)*

Merge sort

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves by finding the mid value and, calls itself for the two halves and then merges the two sorted halves.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. Finally, the sorted pairs is merged into single element.

Given a sequence of n elements(also called keys) a[1],... ,a[n],the general idea is to imagine them splitinto two sets.Each set is individually sorted, and the resulting sorted sequences are merged to produce a singles ordered sequence of n element

The calling the **MergeSort( )** algorithm recursively we divide the list into two sub list until single enough

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Time Complexity

Best Case Time Complexity: **O(n*log n)**

Worst Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

The **MergeSort( )** function calls the **Merge( )** function recursively to sort and combine the given list. The Merge() algorithm is shown below

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

Example: Given input array a[]={6,2,4,5,1,3} after performing the merge sort we can get the sorted array.

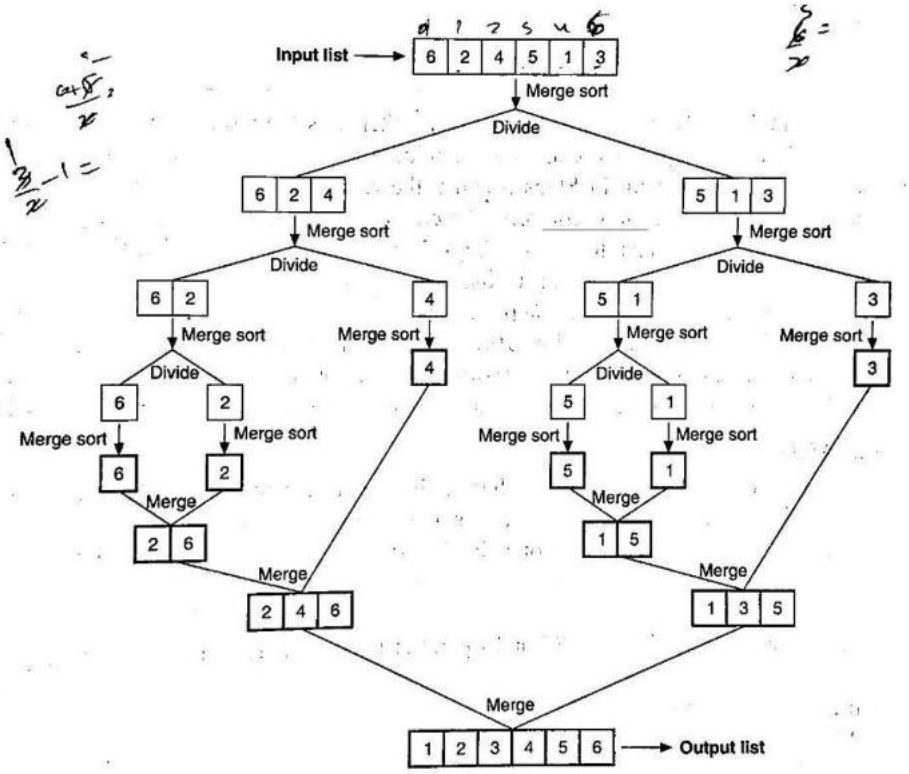


Figure 10.57 Illustration of the merge sort.

/*you need to write steps your own- how these steps, how we divided these list into two sub list(mid value) */

Strassen's Matrix Multiplication

Given two square matrices A and B of size n x n each, find their multiplication matrix.

Naive Method: Following is a simple way to multiply two matrices. A=(a_{ij})

$B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij}, for $i, j = 1, 2, \ldots, n$, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} . \qquad (4.8)$$

We must compute n^2 matrix entries, and each is the sum of n values. The following procedure takes $n \times n$ matrices A and B and multiplies them, returning their $n \times n$ product C. We assume that each matrix has an attribute *rows*, giving the number of rows in the matrix.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Time Complexity of above method is $O(N^3)$.

***Divide and Conquer**:*

Following is simple Divide and Conquer method to multiply two square matrices.

1. Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2. Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

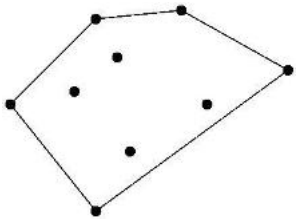
Generally Strassen's Method is not preferred for practical applications for following reasons.

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The submatrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in Naive Method

Chapter A1: Convex Hulls: An Example

A polygon is ***convex*** if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counterclockwise direction you always take left turns.

The ***convex hull*** of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A ***vertex*** is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull. Some other characterizations are given in the exercises.



We discuss three algorithms for finding a convex hull: Graham Scan, Jarvis March and Divide & Conquer. We present the algorithms under the ***assumption*** that:

- *no 3 points are collinear (on a straight line)*

A1.1 Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order.

Let x_0 be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from x_0 going counterclockwise: $x_1, x_2, \ldots, x_{n-1}$. Let $xn = x0$, the chosen point. (We're assuming no two points have the same angle from $x0$.)

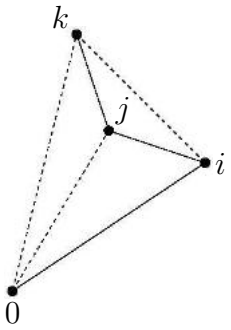
The algorithm is simple to state with a single stack:

Graham Scan

1. Sort points by angle from x_0
2. Push x_0 and x_1. Set i=2
3. While $i \leq n$ do:
   If x_i makes left turn w.r.t. top 2 items on stack
   then { push x^i; i++ }
   else { pop and discard }

To prove that the algorithm works, it suffices to argue that:

- *A discarded point is not in the convex hull.* If x_i is discarded, then for some $i < j < k$ the points $x^i \to x^j \to x^k$ form a right turn. So, x^j is inside the triangle x_0, x_i, x_k and hence is not on the convex hull.



- *What remains is convex.* This is immediate as every turn is a left turn.

The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most $2n$ times. There is a constant-time subroutine for checking, given three points in order, whether the angle is a left or a right turn (Exercise). This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$. (Recall that the notation $O(f(n))$, pronounced "order $f(n)$", means "asymptotically at most a constant times $f(n)$".)

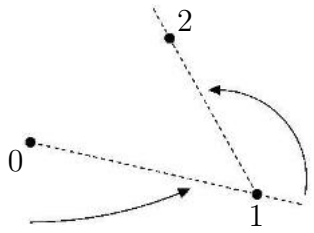
A1.2 Jarvis March

This is also called the ***wrapping algorithm***. This algorithm finds the points on the convex hull ***in the order*** in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many.

Let x_0 be the leftmost point. Let x_1 be the first point counterclockwise when viewed from x_0. Then x_2 is the first point counterclockwise when viewed from x_1, and so on.

Jarvis March

$i = 0$
while not done do
    x_{i+1} = first point counterclockwise from x_i



Finding x_{i+1} takes linear time. The while loop is executed at most n times. More specifically, the while loop is executed h times where h is the number of vertices on the convex hull. So Jarvis March takes time $O(nh)$.

The best case is $h = 3$. The worst case is $h = n$, when the points are, for example, arranged on the circumference of a circle.

A1.3 Divide and Conquer

Divide and Conquer is a popular technique for algorithm design. We use it here to find the convex hull. The first step is a Divide step, the second step is a Conquer step, and the third step is a Combine step.

The idea is to:

Divide and conquer

1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Part 2 is simply two ***recursive*** calls. Note that, if a point is in the overall convex hull, then it is in the convex hull of any subset of points that contain it. (Use characterization in exercise.) So the task is: given two convex hulls, find the convex hull of their union.

◊ *Combining two hulls*

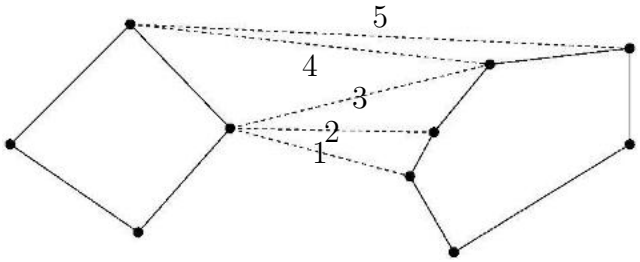
It helps to work with convex hulls that do not overlap. To ensure this, all the points are ***presorted*** from left to right. So we have a left and right half, and hence a left and right convex hull.

Define a ***bridge*** as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the ***upper*** and ***lower*** bridges. The following produces the upper bridge.

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.

2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.

3. If made no progress in Step 2 (cannot raise either side), then stop else repeat Step 2.



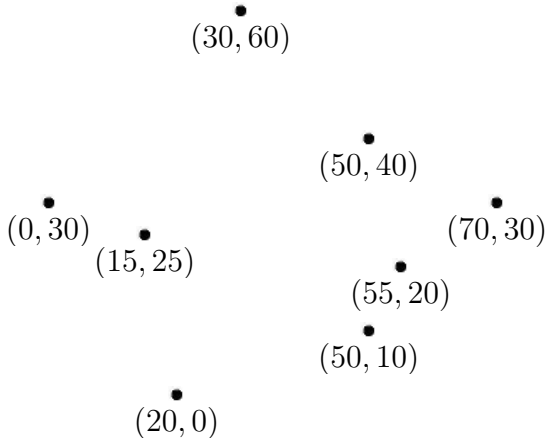
We need to be sure that one will eventually stop. Is this obvious?

Now, we need to determine the running time of the algorithm. The key is to perform Step 2 in constant time. For this it is sufficient that each vertex has a pointer to the next vertex going clockwise and going counterclockwise. Hence the choice of data structure: we store each hull using a ***doubly linked circular linked list***.

It follows that the total work done in a merge is proportional to the number of vertices. And as we shall see from a later chapter, this means that the overall algorithm takes time $O(n \log n)$.

Exercises

1. Find the convex hulls for the following list of points using the three algorithms presented.



GREEDY APPROACH

General Method

The greedy method is the straight forward technique to solve the given problem, and this method is used to solve optimisation problems. For the given n no. of inputs we need to find out the solutions which are satisfies the given constraints. These constraints are called as feasible solutions. n. We need to find a feasible solution that either maximizes or minimizes a given objective function. This solution is called as optimal solution.

These decisions are made regarding to given n inputs and considering one input at a time. The function **Select** selects an input from a[ ] and removes it. The selected input's value is assigned to x. Feasible is a Boolean-valued 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 look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Algorithm 4.1 Greedy method control abstraction for the subset paradigm

JOB SEQUENCE WITH DEADLINES

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution

Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines. Assume, deadline of i^{th} job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Initially, these jobs are ordered according to profit, i.e. p1⩾p2⩾p3⩾...⩾pn


```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

Algorithm 4.5 High-level description of job sequencing algorithm

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$.

Example: Given an array of jobs having a specific deadline and associated with a profit, provided the job is completed within the given deadline. The task is to maximize the profit by arranging the jobs in a schedule, such that only one job can be done at a time.

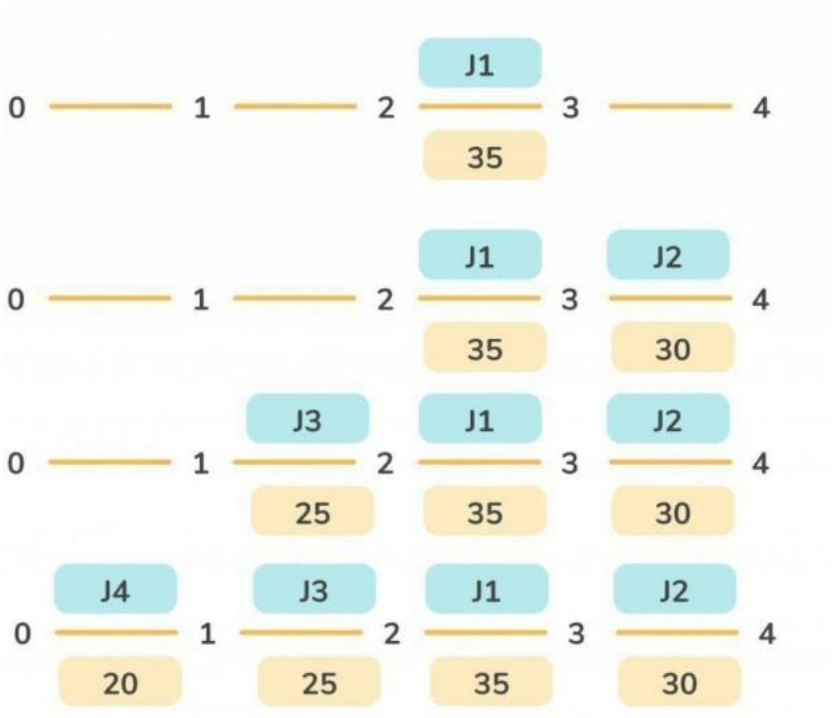
Input:

n=7

Jobs	J1	J2	J3	J4	J5	J6	J7
Profits	35	30	25	20	15	12	5
Deadlines	3	4	4	2	3	1	2

Output: J4 J3 J1 J2

Explanation:



Total profit – 20 + 25 + 35 + 30 = 110

KNAPSACK PROBLEM

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \tag{4.2}$$

$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Applications

In many cases of resource allocation along with some constraint, the problem canbe derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Ex:

objects	1	2	3	4	5	6	7
Profit	5	10	15	7	8	9	4
weight	1	3	5	4	1	3	2

w = 15, n = 7

We can solve the above problem in 3 ways.

1. considering the profit
2. Considering the wight
3. Considering the practional values.

1. Considering the profit:

objects	profit	weight	Remaing weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
4	$7 \times \frac{3}{4} = 5.25$	4	3 - 3 = 0
	Total = 47.25		

2. Considering the wight:

Objects	profit	weight	Remaing weight.
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
7	4	2	13 - 2 = 11
6	9	3	11 - 3 = 8
2	10	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	$15 \times \frac{1}{5} = 3$	1	1 - 1 = 0
	Total = 46		

3. Considering the fractional values:

objects	1	2	3	4	5	6	7
Profit	5	10	15	7	8	9	4
weight	1	3	5	4	1	3	2
p/w	5	3.3	3	1.75	8	3	2

objects	profit	weight	Remaing weight.
5	8	1	15-1 = 14
1	5	1	14-1 = 13
2	10	3	13-3 = 10
3	15	5	10-5 = 5
6	9	3	5-3 = 2
7	4	2	2-2 = 0
	Total: 51		

MINIMUM COST SPANNING TREE

Definition:

Let G = (V, E) be a connected graph in which each edge (u, v) belongs to E has an associatedcost C(u, v).

- A Spanning Tree for G is a subgraph of G that it is a free tree connecting all vertices in V. Thecost of a spanning tree is the sum of costs on its edges.
- An MST of G is a spanning tree of G having a minimum cost.

Properties

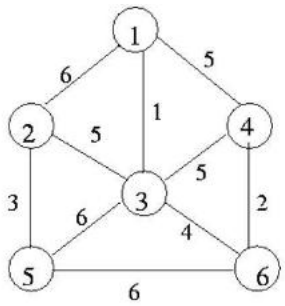
- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

The minimum spanning tree from a graph is found using the following algorithms:

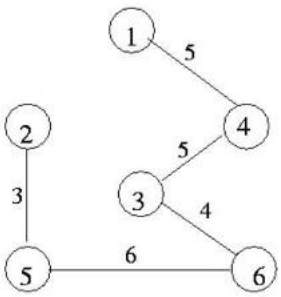
1. Prim's Algorithm
2. Kruskal's Algorithm

See the below, several examples.

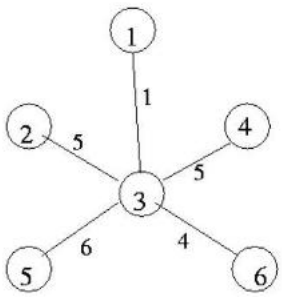
Spanning trees in a connected graph



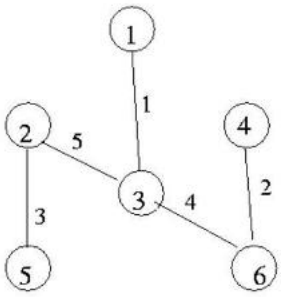
A connected graph.



A spanning tree with cost = 23



Another spanning tree with cost 21



MST, cost = 15

Prim's Algorithm

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Working of prim's algorithm

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the newvertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Algorithm 4.8 Prim's minimum-cost spanning tree algorithm

Example:

Kruskal's Algorithm

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

Working of Kruskal's algorithm

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges untilthe goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to beadded creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree iscreated.

The applications of Kruskal's algorithm are -

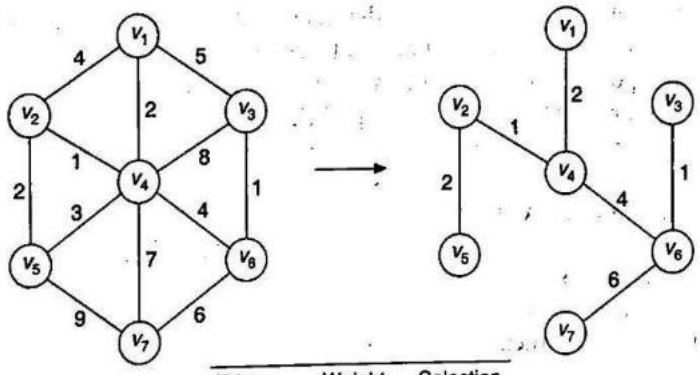
- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Algorithm 4.9 Early form of minimum-cost spanning tree algorithm due to Kruskal

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1) and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

Algorithm 4.10 Kruskal's algorithm

Let us consider an example for the illustration of the above algorithm. Figure 8.37 illustrates the algorithm for a graph *G*. Lists of edges with their weights are maintained in the form of a table. From this list, we have to select (✓) or reject (×) depending on whether it forms a cycle or not. The length of the spanning tree as obtained in Figure 8.38 can be calculated as 16.



Edge	Weight	Selection
V_2–V_4	1	✓
V_3–V_6	1	✓
V_2–V_5	2	✓
V_1–V_4	2	✓
V_4–V_5	3	×
V_1–V_2	4	×
V_4–V_6	4	✓
V_1–V_3	5	×
V_6–V_7	6	✓
V_4–V_7	7	
V_3–V_4	8	
V_5–V_7	9	

Figure 8.38 The minimum spanning tree using Kruskal's algorithm.

SINGLE-SOURCESHORTESTPATHS: GENERAL WEIGHTS

The **Single-Source Shortest Paths (SSSP)** problem is about finding the shortest paths from a given source vertex to all other vertices in a weighted graph. A dynamic programming approach can be used effectively to solve this problem, especially for **Directed Acyclic Graphs (DAGs)** or in special cases like **Bellman-Ford** for general graphs with negative weights.

Definition: Given a weighted directed graph G(V,E) where:

- V is the set of vertices,
- E is the set of edges, where each edge (u,v)∈E has a weight w(u,v)w
- A source vertex s∈V.

The goal is to find the shortest path from SSS to every other vertex in V. One of the most common methods to apply dynamic programming is through **Bellman and Ford Algorithm.**

Let $dist^l[u]$be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most l edges. Then, $dist^1[u]$= cost[v,u], 1<n. s. Hence, $dist^{n-1}[u]$is the length of an un restricted shortest path from v to u.

Our goal then is to compute $dist^{n-1}[u]$for all u. This can be done using the dynamic programming methodology by following the recurrence dist:

$$dist^k[u] = \min\{dist^{k-1}[u], \min_i \{dist^{k-1}[i] + cost[i,u]\}\}$$

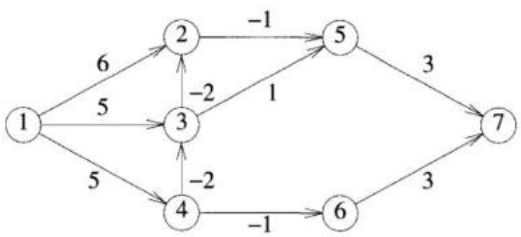
This recurrence can be used to compute $dist^k$ from $dist^{k-1}$, for k-2,3,4,….. n-1

Example:

Below Figure gives a seven-vertex graph, together with the arrays $dist^k$,k = 1,... ,6. These arrays were computed using the equation just given. For instance, $dist^k[l]$= 0 for all k since 1 is the source node. Also, $dist^1[2]$=6,$dist^1[3]$= 5, and $dist^1[4]$ = 5, since there are edges from 1 to these nodes. The distance $dist^1[]$ is oo for the nodes 5,6,and 7 since there are no edges to these from 1.

$$\begin{aligned} dist^2[2] &= \min\{dist^1[2], \min_i dist^1[i] + cost[i,2]\} \\ &= \min\{6, 0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty\} = 3 \end{aligned}$$

Here the terms $0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty$, and $\infty+\infty$ correspond to a choice of $i = 1, 3, 4, 5, 6$, and 7, respectively. The rest of the entries are computed in an analogous manner. □



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

Bellman and Ford algorithm to compute shortest paths

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Time complexity:

Each iteration of the for loop of lines7 to 12takes0(n^2) time if adjacency matrices are used and O(e) time if adjacency lists are used. Heree is the number of edges in the graph. The overall complexity is O(n^3) when adjacency matrices are used and 0(ne)when adjacency lists are used

DYNAMIC PROGRAMMING

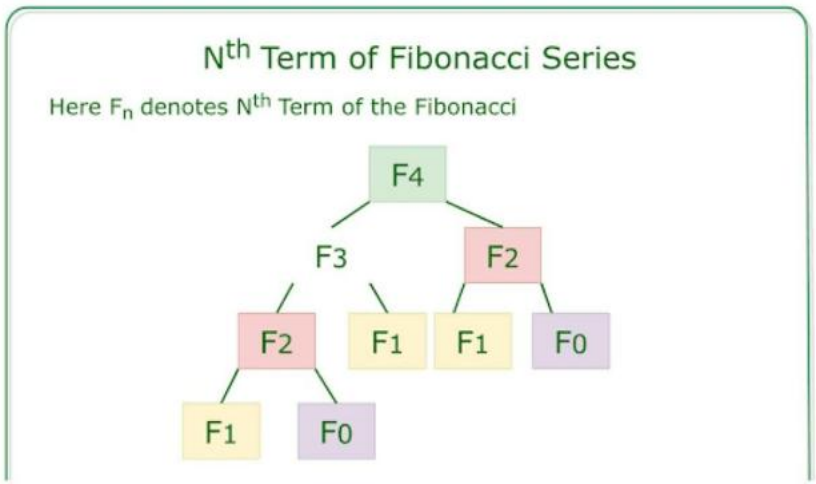
THE GENERAL METHOD

Dynamic programming (DP) is a problem-solving method used in computer science for solving complex problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. It is particularly useful for optimization problems, where we need to find the minimum, maximum, or optimal solution.

Example 1: Consider the problem of finding the Fibonacci sequence:

Fibonacci sequence: *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, .*

Dynamic Programming Approach:



Fibonacci Series using Dynamic Programming

- **Subproblems:** F(0), F(1), F(2), F(3), ...
- **Store Solutions:** Create a table to store the values of F(n) as they are calculated.
- **Build Up Solutions:** For F(n), look up F(n-1) and F(n-2) in the table and add them.
- **Avoid Redundancy:** The table ensures that each subproblem (e.g., F(2)) is solved only once.

By using DP, we can efficiently calculate the Fibonacci sequence without having to recompute subproblems.

ALL-PAIRS SHORTEST PATHS

The All-Pairs Shortest Path (APSP) problem is a classic problem in graph theory where we want to find the shortest paths between every pair of vertices in a given weighted graph. One of the most well-known dynamic programming approaches to solve APSP is **Floyd-Warshall Algorithm**. It works for both directed and undirected graphs and can handle negative weights.

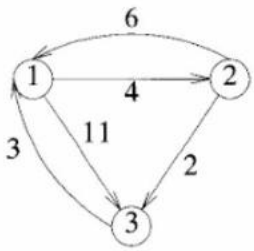
The algorithm uses a dynamic programming approach where it builds up the shortest paths by considering each vertex as an intermediate vertex and checking if a shorter path exists through this vertex. The **all-pairs shortest-path problem** is to determine a matrix **A** such that A(i,j) is the length of a shortest path from i to j.

- Let A[][] be the distance matrix where A[i][j] is the weight of the edge from vertex i to vertex j.
- If there is no edge between vertices i and j, then set A[i][j] = INF
- The distance from any vertex to itself is always 0, so A[i][i] = 0.
- For each pair of vertices (i, j), check if an intermediate vertex k can provide a shorter path from i to j. If so, update the distance:

$$A^k(i,j) = \min\ \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\},\quad k \geq 1$$

This is repeated for all pairs of vertices and all possible intermediate vertices.

Example 5.15 The graph of Figure 5.6(a) has the cost matrix of Figure 5.6(b). The initial A matrix, $A^{(0)}$, plus its values after 3 iterations $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ are given in Figure 5.6. □



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Figure 5.6 Directed graph and associated matrices

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Time Complexity:

The time complexity of the Floyd-Warshall algorithm is $O(n^3)$, where n is the number of vertices.

OPTIMAL BINARY SEARCH TREE

The **Optimal Binary Search Tree** (OBST) problem is a dynamic programming problem where the goal is to construct a binary search tree (BST) from a set of keys such that the total cost of searching for all keys is minimized. The cost of searching for a key is proportional to its depth in the tree, so we want to minimize the weighted sum of the depths of the nodes, where weights are the probabilities of accessing each key.

Problem Definition:

- A sorted array of n keys: $K_1, K_2, ..., K_n$
- Access probabilities for each key: $p_1, p_2, ..., p_n$
- Access probabilities for unsuccessful searches between keys (dummy nodes): $q_0, q_1, ..., q_n$

The objective is to construct a binary search tree that minimizes the expected search cost.

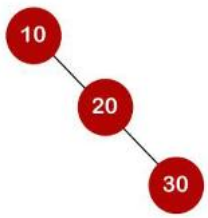
For example:

10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.
The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

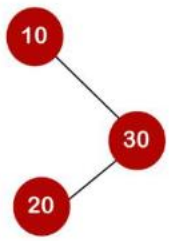
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



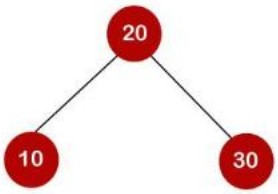
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



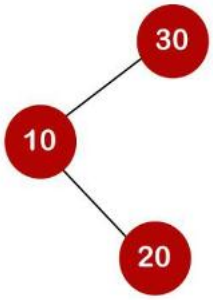
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



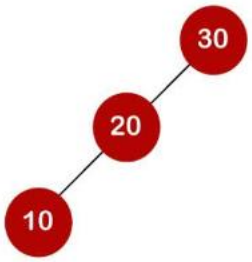
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach

Consider the below table, which contains the keys and frequencies.

	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3

i \ j	0	1	2	3	4
0					
1					
2					
3					
4					

First, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0
When i = 1, j=1, then j-i = 0
When i = 2, j=2, then j-i = 0
When i = 3, j=3, then j-i = 0
When i = 4, j=4, then j-i = 0
Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1
When j=2, i=1 then j-i = 1
When j=3, i=2 then j-i = 1
When j=4, i=3 then j-i = 1
Now to calculate the cost, we will consider only the jth value.
The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).
The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).
The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)
The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

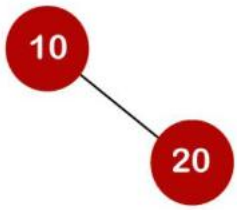
	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where j-i = 2

When j=2, i=0 then j-i = 2
When j=3, i=1 then j-i = 2

When j=4, i=2 then j-i = 2
In this case, we will consider two keys.

- When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: 4*1 + 2*2 = 8
In the second binary tree, cost would be: 4*2 + 2*1 = 10
The minimum cost is 8; therefore, c[0,2] = 8

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be: 1*2 + 2*6 = 14
In the second binary tree, cost would be: 1*6 + 2*2 = 10
The minimum cost is 10; therefore, c[1,3] = 10

- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

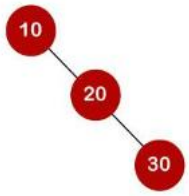
In the first binary tree, cost would be: 1*6 + 2*3 = 12
In the second binary tree, cost would be: 1*3 + 2*6 = 15
The minimum cost is 12, therefore, c[2,4] = 12

i \ 1	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

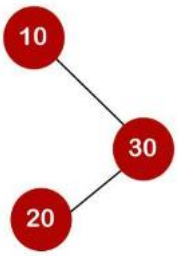
Now we will calculate the values when j-i = 3
When j=3, i=0 then j-i = 3
When j=4, i=1 then j-i = 3

- When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

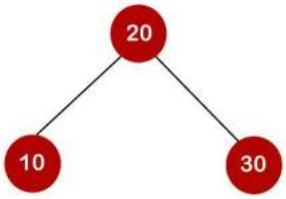
The following are the trees that can be made if 10 is considered as a root node.



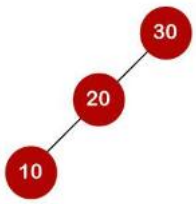
In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.
Cost would be: 1*4 + 2*2 + 3*6 = 26



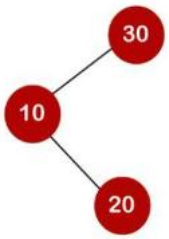
In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.
Cost would be: 1*4 + 2*6 + 3*2 = 22
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.
Cost would be: 1*2 + 4*2 + 6*2 = 22
The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.
Cost would be: 1*6 + 2*2 + 3*4 = 22



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.
Cost would be: 1*6 + 2*4 + 3*2 = 20
Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

When i=1 and j=4 then we will consider the keys 20, 30, 40

c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

= min{0+12, 2+3, 10+0}+ 11

= min{12, 5, 10} + 11

The minimum value is 5; therefore, c[1,4] = 5+11 = 16

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

- Now we will calculate the values when j-i = 4

When j=4 and i=0 then j-i = 4

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

w[0, 4] = 4 + 2 + 6 + 3 = 15

If we consider 10 as the root node then

C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]

= min {0 + 16} + 15= 31

If we consider 20 as the root node then

C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]

= min{4 + 12} + 15

= 16 + 15 = 31

If we consider 30 as the root node then,

C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]

= min {8 + 3} + 15

= 26

If we consider 40 as the root node then,

C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]

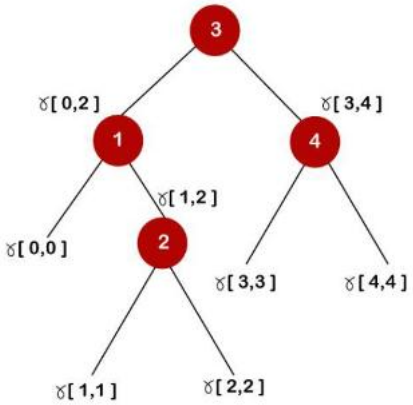
= min{20 + 0} + 15

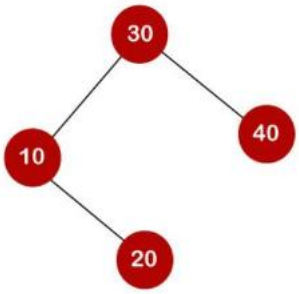
= 35

In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

The optimal binary search tree can be created as:
Root node: r[I,j]=k
Left node : r[I,k-1]
Right node: r[k,j]

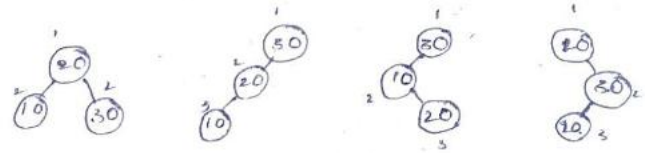




Time complexity:
The time complexity of the algorithm is $O(n^3)$ because we have three nested loops: one for the length of the subtree, one for the starting index, and one for the root selection.

Optimal Binary search tree:

10, 20, 30



$\frac{2 \times {}^{n}C_{n}}{n+1} = \frac{2 \cdot {}^{3}C_{3}}{n+1} = \frac{\frac{3!}{3!}}{4} = \frac{2/6}{4} = 5$

1. $\frac{1+2+3}{3} = 2$ 2. $\frac{1+2+3}{3} = 2$ 3. $\frac{1+2+3}{3} = 2$ 4. $\frac{1+2+3}{3} = 2$ 5. $\frac{1+2+3}{3} = 2$

	1	2	3	4
key	10	20	30	40
frequency	4	2	6	3

j \ i	0	1	2	3	4
0	0	4	8	20	26
1		0	2	10	16
2			0	6	12
3				0	3
4					0

$j-i=0$

$0-0=0$ c[0,0]
$1-1=0$ c[1,1]
$2-2=0$ c[2,2]
$3-3=0$ c[3,3]
$4-4=0$ c[4,4]

$j-i=1$ (1st iteration)

$1-0=1$ c(0,1) = 4
$2-1=1$ c(1,2) = 2
$3-2=1$ c(2,3) = 6
$4-3=1$ c(3,4) = 3

$j-i=2$ (2nd iteration)

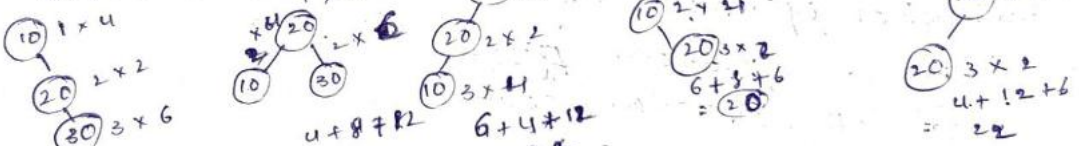
$2-0=2$ c[0,2] = 8
$3-1=2$ c[1,3] = 6
$4-2=2$ c[2,4] = 3

$j-i=3$

$3-0=3$
$4-1=3$
c[0,3]
c[1,4]

$j-i=3$

$3-0=3$ c[0,3]
$4-1=3$ c[1,4]



$j-i=4$

$4-0=4$ c[0,4]

$c[i,j] = c[i,k-1] + c[k,j] + w[i,j]$

$c[0,4] = c[0,0] + c[1,4] + 15$ (k=2)
$= c[0,1] + c[2,4] + 11$ (k=3)
$= c[0,2] + c[3,4] + 9$ (k=4)
$= c[0,3] + c[4,4] + 3$

Algorithm:

```
1   Algorithm OBST(p, q, n)
2   // Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities
3   // p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
4   // the cost c[i,j] of optimal binary search trees t_ij for identifiers
5   // a_{i+1}, ..., a_j. It also computes r[i,j], the root of t_ij.
6   // w[i,j] is the weight of t_ij.
7   {
8       for i := 0 to n - 1 do
9       {
10          // Initialize.
11          w[i,i] := q[i]; r[i,i] := 0; c[i,i] := 0.0;
12          // Optimal trees with one node
13          w[i,i+1] := q[i] + q[i+1] + p[i+1];
14          r[i,i+1] := i + 1;
15          c[i,i+1] := q[i] + q[i+1] + p[i+1];
16      }
17      w[n,n] := q[n]; r[n,n] := 0; c[n,n] := 0.0;
18      for m := 2 to n do  // Find optimal trees with m nodes.
19          for i := 0 to n - m do
20          {
21              j := i + m;
22              w[i,j] := w[i,j-1] + p[j] + q[j];
23              // Solve 5.12 using Knuth's result.
24              k := Find(c, r, i, j);
25                  // A value of l in the range r[i,j-1] ≤ l
26                  // ≤ r[i+1,j] that minimizes c[i,l-1] + c[l,j];
27              c[i,j] := w[i,j] + c[i,k-1] + c[k,j];
28              r[i,j] := k;
29          }
30      write (c[0,n], w[0,n], r[0,n]);
31  }

1   Algorithm Find(c, r, i, j)
2   {
3       min := ∞;
4       for m := r[i,j-1] to r[i+1,j] do
5           if (c[i,m-1] + c[m,j]) < min then
6           {
7               min := c[i,m-1] + c[m,j]; l := m;
8           }
9       return l;
10  }
```

Algorithm 5.5 Finding a minimum-cost binary search tree

0/1 KNAPSACK PROBLEM

The **0/1 Knapsack Problem** is a classic problem in combinatorial optimization. The objective is to maximize the total value of items that can be placed in a knapsack of fixed capacity, subject to the constraint that the total weight of the selected items cannot exceed the knapsack's capacity. Each item can either be taken (1) or not taken (0), hence the name "0/1 knapsack."

Definition:

- n items, each with a weight w[i] and a value v[i].
- A knapsack with a maximum weight capacity W.

The goal is to select items such that the total weight is less than or equal to W and the total value is maximized.

Example: Consider the problem having weights and profits are:
Weights: {3, 4, 6, 5}
Profits: {2, 3, 1, 4}
The weight of the knapsack is 8 kg
The number of items is 4
The above problem can be solved by using the following method:
x_i = {1, 0, 0, 1}
= {0, 0, 0, 1}
= {0, 1, 0, 1}
The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:
2^4 = 16;
So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.
First,
we create a matrix shown as below:

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

In the above matrix, columns represent the weight, i.e., 8. The rows represent the profits and weights of items. Here we have not taken the weight 8 directly, problem is divided into sub-problems, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8. The solution of the sub-problems would be saved in the cells and answer to the problem would be stored in the final cell. First, we write the weights in the ascending order and profits according to their weights shown as below:

w_i = {3, 4, 5, 6}
p_i = {2, 3, 4, 1}

The first row and the first column would be 0 as there is no item for w=0

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

When i =1, W =8

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 8; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][8] shown as below:

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

When i = 2, W = 8

The weight corresponding to the value 2 is 4, i.e., w_2 = 4. Since we have two items in the set having weights 3 and 4, and the weight of the knapsack is 7. We can put item of weight 4 and 3 in a knapsack and the profits corresponding to weights are 2 and 3; therefore, the total profit is 5, so we add 5 at M[2][7] shown as below:

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

When i = 3, W = 8

The weight corresponding to the value 3 is 5, i.e., $w_3 = 5$. Since we have three items in the set of weight 3, 4, and 5 respectively, and the weight of the knapsack is 8. In this case, we can keep both the items of weight 3 and 4, the sum of the profit would be equal to (2 + 3), i.e., 5, so we add 5 at M[3][8] shown as below:

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

When i = 4, W = 8

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 8. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., (2 + 3) equals to 5, so we add 5 at M[4][8] shown as below:

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

As we can observe in the above table that 5 is the maximum profit among all the entries. The pointer points to the last row and the last column having 5 value. Now we will compare 5 value with the previous row; if the previous row, i.e., i = 3 contains the same value 5 then the pointer will shift upwards.

X = {1, 1, 0, 0}

The profit corresponding to the weight is 2. Therefore, the remaining profit is 0. We compare 0 value with the above row. Since the above row contains a 0 value but the profit corresponding to this row is 0. In this problem, two weights are selected, i.e., 3 and 4 to maximize the profit.

can be either
as 0/1 knapsack problem.

Ex: weigh solve the below problem by using 0 or 1 knapsack Problem by using dynamic approch

weight {3, 4, 6, 5}

Profit {2, 3, 1, 4}

the weight of the knapsack is = 8 kg

the no. of items are = 4

i↓ w→	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	4	4	5	6
4	0	0	0	2	3	4	4	5	6

{1, 0, 0, 1}

weight: {3, 4, 6, 5}

Profit: {2, 3, 1, 4}

{1, 0, 0, 1}

3+5=8.

STRING EDITING

String editing using dynamic programming is the problem of converting one string to another string which involves finding the minimum number of operations (insert, delete, substitute) required to convert one string into another.

- We are given two strings X = $x_1,x_2,...,x_n$ and Y = $y_1,y_2,.....y_m$, where x_i, 1<*i*< *n, are members of a finite set of symbols known alphabet.*
- *We want to transform X into Y using a sequence of edit operations on X. The permissible edit operations are insert, delete, and change (a symbol of X into another) and there Is a cost associated with performing each.*
- *The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence.*
- *The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y.*

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_j)$ be the cost of inserting the symbol y_j into X, and $C(x_i,y_j)$ be the cost of changing the symbol x_i of X into y_j.

A dynamic programming solution for this problem can be obtained as follows. Define cost(i,j) to be the minimum cost of any edit sequence for transforming $x_1,x_2,...,x_i$ into $y_1,y_2,.....y_j$. Compute cost(i,j) for each i and j. Then cost(i, j) is the cost of an optimal edit sequence.

For i = j = 0, cost(i,j) = 0, since the two sequences are identical(and empty).Also,
If i != 0 and j!=0, $x_1,x_2,...,x_i$ into $y_1,y_2,.....y_j$ in one of three ways:

1. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_j$ using a minimum-cost edit sequence and then delete x_i. The corresponding cost is $cost(i-1, j) + D(x_i)$.

2. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then change the symbol x_i to y_j. The associated cost is $cost(i-1, j-1) + C(x_i, y_j)$.

3. Transform $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then insert y_j. This corresponds to a cost of $cost(i, j-1) + I(y_j)$.

The minimum cost of any edit sequence that transforms $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $i > 0$ and $j > 0$) is the minimum of the above three costs, according to the principle of optimality. Therefore, we arrive at the following recurrence equation for $cost(i, j)$:

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0,\ i > 0 \\ cost(0,j-1) + I(y_j) & i = 0,\ j > 0 \\ cost'(i,j) & i > 0,\ j > 0 \end{cases} \qquad (5.13)$$

$$\text{where } cost'(i,j) = \min \{ \begin{array}{l} cost(i-1,j) + D(x_i), \\ cost(i-1,j-1) + C(x_i,y_j), \\ cost(i,j-1) + I(y_j) \ \} \end{array}$$

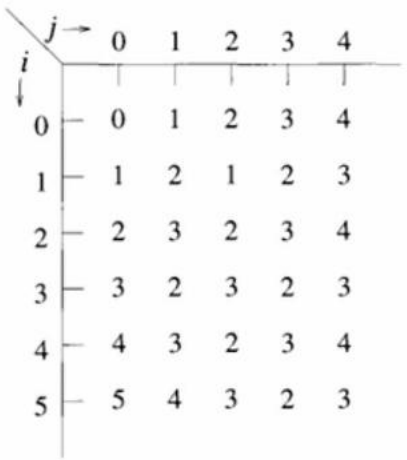
Example 5.20 Consider the string editing problem of Example 5.19. $X = a, a, b, a, b$ and $Y = b, a, b, b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i = 0, j > 1$, and $j = 0, i > 1$, $cost(i, j)$ can be computed first (Figure 5.18). Let us compute the rest of the entries in row-major order. The next entry to be computed is $cost(1, 1)$.

$$\begin{aligned} cost(1,1) &= \min \{cost(0,1) + D(x_1), cost(0,0) + C(x_1, y_1), cost(1,0) + I(y_1)\} \\ &= \min \{2, 2, 2\} = 2 \end{aligned}$$

Next is computed $cost(1, 2)$.

$$\begin{aligned} cost(1,2) &= \min \{cost(0,2) + D(x_1), cost(0,1) + C(x_1, y_2), cost(1,1) + I(y_2)\} \\ &= \min \{3, 1, 3\} = 1 \end{aligned}$$

The rest of the entries are computed similarly. Figure 5.18 displays the whole table. The value $cost(5, 4) = 3$. One possible minimum-cost edit sequence is delete x_1, delete x_2, and insert y_4. Another possible minimum cost edit sequence is change x_1 to y_2 and delete x_4. □



i \ j	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

TRAVELLING SALES PERSON PROBLEM

The Traveling Salesperson Problem (TSP) is a classic optimization problem in which a salesperson must visit a set of cities, starting from a specific city, visit each city exactly once, and return to the starting city. The goal is to minimize the total travel distance or cost.

Using dynamic programming, we can solve TSP efficiently for small to medium-sized sets of cities (typically around 20 or fewer). For larger numbers of cities, the exponential time complexity of dynamic programming for TSP makes it infeasible.

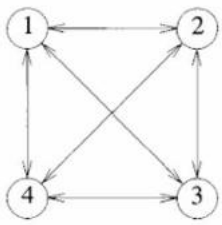
To solve the problem of travelling sales person problem, the equation for the given graph is

$$g(i,S) = \min_{j \in S}\{c_{ij} + g(j, S-\{j\})\}$$

Here,

i is the source vertex, s is the sub set, $c_{i,j}$ is the cost or weight of the i to j

Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).



(a)

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

(b)

Figure 5.21 Directed graph and edge length matrix c

Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

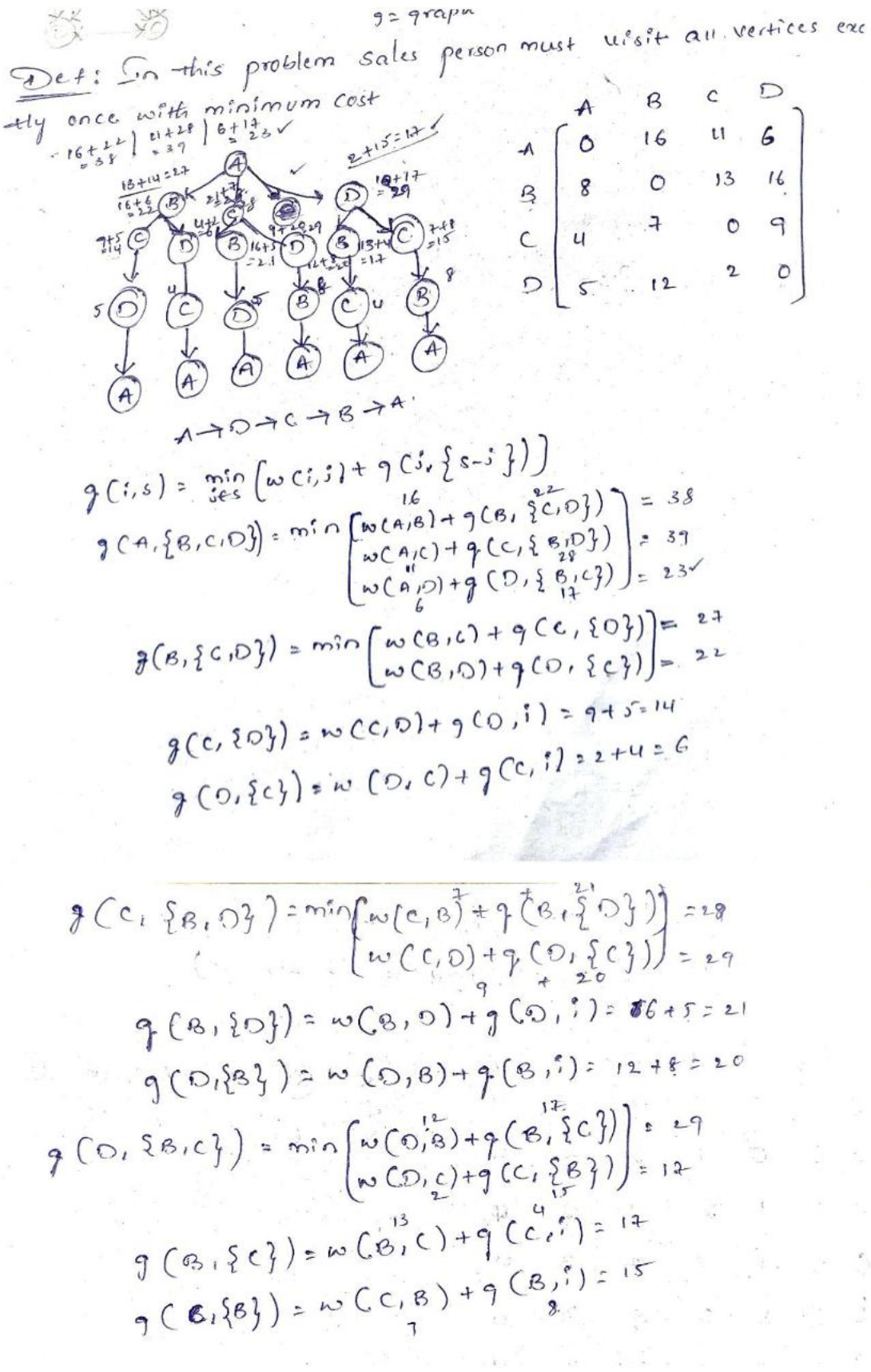
$$\begin{array}{lll} g(2,\{3\}) = c_{23} + g(3,\phi) = 15 & & g(2,\{4\}) = 18 \\ g(3,\{2\}) = 18 & & g(3,\{4\}) = 20 \\ g(4,\{2\}) = 13 & & g(4,\{3\}) = 15 \end{array}$$

Next, we compute $g(i,S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lll} g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} = 25 \\ g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = 25 \\ g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{aligned} g(1,\{2,3,4\}) &= \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$



Time Complexity

The time complexity of this approach is $O(2^n \times n^2)$, where 2^n represents the number of subsets, and n^2 arises from the inner loops for each subset and city.

BACKTRACKING

Definition: Backtracking is a powerful algorithmic technique whose goal is to use brute force (problem-solving technique that involves checking every possible solution until the correct one is found) to solve solutions by incrementally making choices, and when necessary, going back to those choices to explore alternative paths.

- This method is particularly useful for problems where there are multiple possible solutions, but not all of them meet certain constraints or criteria.
- Backtracking can be defined to search every possible combination in order to solve a computational problem.

For any problem these constraints can be divided into two categories: explicit and implicit

Explicit constraints are rules that restrict each x_i to take on values only from a given set. Common examples of explicit constraints are:

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} & S_i & = & \{0,1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

Where x_i is the chosen solution from some finite set S_i.

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the xi must relate to each other.

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

Algorithm 7.1 Recursive backtracking algorithm

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
11          {
12                  if (x[1], ..., x[k] is a path to an answer node)
13                      then write (x[1 : k]);
14                  k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

Algorithm 7.2 General iterative backtracking method

Terminologies

1. **Solution Space**: The set of all possible solutions that satisfy the problem's constraints.
2. **State Space Tree**: A conceptual tree used to represent the choices at each step in backtracking.
3. **Decision Point**: The stage at which a choice is made, deciding how to proceed with building the solution.
4. **Constraint**: A rule or condition that must be met for a solution to be valid.
5. **Pruning**: The process of eliminating parts of the solution space that do not satisfy constraints, thus avoiding unnecessary computations.
6. **Backtracking**: The process of returning to the previous decision point to try a different option when a path fails to meet constraints.
7. **Partial Solution**: A solution that is incomplete and is built upon step-by-step.
8. **Feasible Solution**: A partial solution that satisfies all constraints up to a certain point, though it may not be complete.
9. **Complete Solution**: A solution that meets all constraints fully and solves the problem.
10. **Dead End**: A situation where no further extensions of a partial solution can lead to a valid solution.
11. **Recursive Backtracking**: A technique where the backtracking process is implemented using recursive function calls.
12. **E-node** (or **Expanding Node**): is the current node in the **state space tree** where decisions are being made to explore further.

8-QUEENS PROBLEM

A classic combinatorial problem is to place eight queens on an 8 x 8 chess board so that no two attack that is, so that no two of them are on the same row, column, or diagonal.

- Let us number the rows and columns of the chessboard 1 through 8 .
- The queens can also be numbered 1 through 8. Since each queen must be on a different row, we can do without loss of generality assume queen i is to be placed on row i.
- All solutions to the 8-queens problem can therefore be represented as 8-tuples (x_1,…….x_8), where x_i is the column on which queen i is placed.
- The explicit constraints using this formulation are S_i= {1,2,3,4,5,6,7,8}, 1 ≤ i ≤ 8.
- Therefore, the solution space consists of 8^8 8-tuples.
- The implicit constraints for this problem are that no two x_1's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.
- The first of these two constraints implies that all solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8). This realization reduces the size of the solution space from 8^8 tuples to 8! tuples.

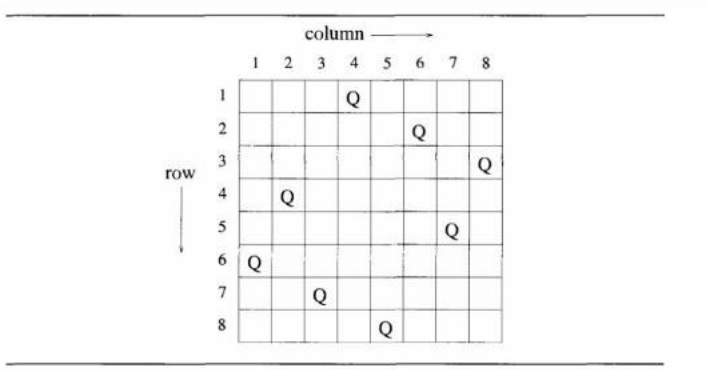


Figure 7.1 One solution to the 8-queens problem

```
1   Algorithm NQueens(k,n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k,i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1,n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

```
1   Algorithm Place(k,i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] − i) = Abs(j − k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

The time complexity of the backtracking algorithm for the 8 queens problem is O(N!). This is because in the worst case, the algorithm will try to place a queen in each row of the board, one column at a time. This results in a time complexity of O(N * (N-1) * (N-2) * … * 1) = O(N!)

SUM OF SUBSETS PROBLEM

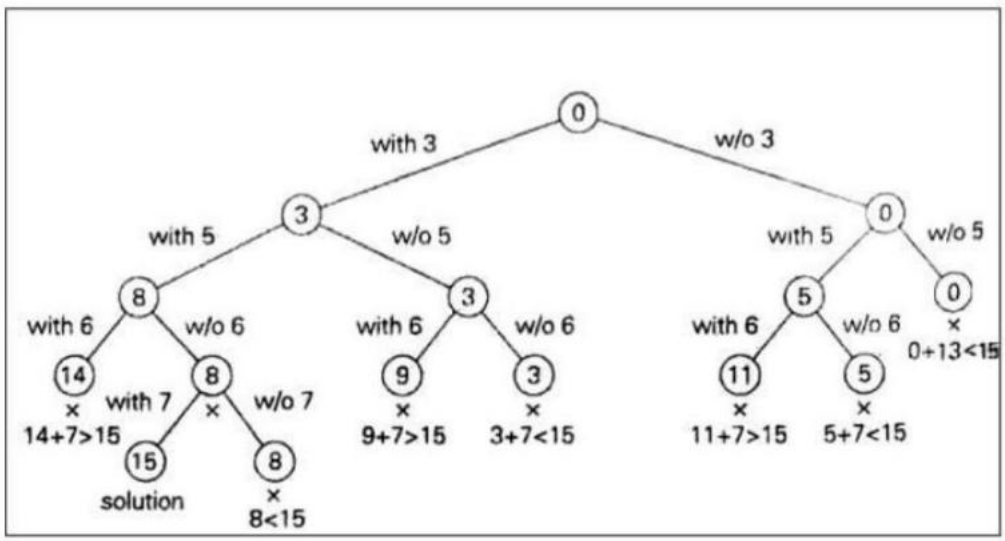
Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number. The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem.

A subset A of n positive integers and a value sum(d) is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

Steps:

1. Start with an empty set
2. Add the next element from the list to the set
3. If the subset is having sum M, then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible (sum of subset < d) then go to step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Example: S = {3,5,6,7} and d = 15, Find the sum of subsets by using backtracking



Solution {3,5,7}

GRAPH COLORING

Definition: Let G be a graph and m be a given positive integer. We want to discover whether 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. This is termed the *m-colorability* decision problem. if d is the degree of the given graph, then it can be colored with d + 1 colors. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.

For example, the graph of Figure7.11 can be colored with three colors 1,2,and 3.The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

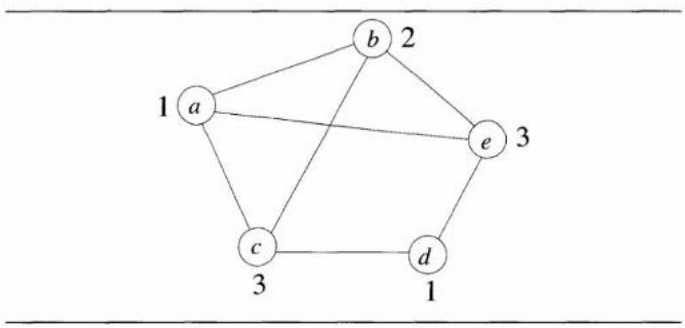


Figure 7.11 An example graph and its coloring

Suppose we represent a graph by its adjacency matrix G[1:n, 1:n],where G[i, j] = 1, if (i,j) is an edge of G, and G[i, j]= 0 otherwise. The colors are represented by the integers 1,2,...,m. and the solutions are given by the n tuple(x_1,..,x_n),where X_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is m-Coloring (Algorithm 7.7)

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

Algorithm 7.7 Finding all m-colorings of a graph

Time Complexity: The time complexity is **O(m^n)**, where:

- **m** is the number of colors.
- **n** is the number of vertices. This is because, for each vertex, the algorithm tries all **m** colors, leading to an exponential number of possibilities in the worst case.

- The algorithm tries all possible color assignments for the current vertex x[k] by calling NextValue(k), which assigns a legal color to x[k].
- **Check for Validity**:
  - If x[k] = 0, it means no legal color was found for vertex k, so the algorithm returns, backtracking to the previous state.
  - If k=n =, it means all n vertices have been successfully colored with at most m colors, and the current coloring x[1:n]is printed.
- **Recursive Call**:
  - If all vertices haven't been colored yet, the algorithm recursively calls mColoring(k + 1) to color the next vertex.
- This repeat loop continues until no more colors are left to try for the current vertex, leading to backtracking if a coloring solution isn't found.

```
1   Algorithm NextValue(k)
2   // x[1], ..., x[k-1] have been assigned integer values in
3   // the range [1, m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0, m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17              // If (k, j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

Algorithm 7.8 Generating a next color

- **Line 10**:
  - x[k] := (x[k] + 1) mod (m + 1): This statement increments x[k] by one to get the next color.
  - The modulo operation wraps the color assignment back to 1 when it reaches m+1. If x[k] is set to 0, it means all colors have been used.
- **Line 11**:
  - If x[k] = 0, it means no valid color was found for vertex k, so the function returns, indicating failure to find a color.
- **Line 13-20**:
  - A for loop checks if the color assigned to x[k] conflicts with any adjacent vertices.
  - **Line 16**: if (G[k, j] ≠ 0) and (x[k] = x[j]): Checks if vertex j is adjacent to k (G[k, j] ≠ 0) and if x[k] has the same color as x[j]. If both conditions are true, it means there's a conflict with an adjacent vertex.
  - **Line 18**: break; If a conflict is found, the algorithm breaks out of the for loop to try a different color.
- **Line 21**:
  - If j = n + 1, it means the color assigned to x[k] is valid (no conflicts with adjacent vertices). The function then returns, indicating a valid color was found.
- The repeat loop continues until a valid color is found or all options are exhausted.

Example Walkthrough with NextValue(k)

We start the coloring process with an initial array xxx representing the colors assigned to each vertex:
x=[0,0,0,0]x = [0, 0, 0, 0]x=[0,0,0,0]
Here, x[i]=0x[i] = 0x[i]=0 means that Vertex iii has not been assigned a color yet.
Let's go through the steps of NextValue(k) to assign a color to each vertex one by one.
Step 1: Color Vertex 1 (k = 1)

1. Initial Assignment:
   - x[1] := (x[1] + 1) mod (m + 1) = (0 + 1) mod 4 = 1
   - Vertex 1 is tentatively assigned color 1.
2. Conflict Check:
   - The function checks all vertices adjacent to Vertex 1 (Vertex 2 and Vertex 3).
   - Since no adjacent vertices have been colored yet, there is no conflict.
3. Valid Color:
   - Color 1 is valid for Vertex 1.
   - Result: x=[1,0,0,0]x = [1, 0, 0, 0]x=[1,0,0,0]

Step 2: Color Vertex 2 (k = 2)

1. Initial Assignment:
   - x[2] := (x[2] + 1) mod (m + 1) = (0 + 1) mod 4 = 1
   - Vertex 2 is tentatively assigned color 1.
2. Conflict Check:
   - The function checks all vertices adjacent to Vertex 2 (Vertex 1 and Vertex 4).
   - Vertex 1 has color 1, which is the same as the tentative color for Vertex 2.
3. Conflict Detected:
   - Since there is a conflict with Vertex 1, the algorithm breaks out of the for loop.
4. Next Color Attempt:
   - x[2] := (x[2] + 1) mod (m + 1) = (1 + 1) mod 4 = 2
   - Vertex 2 is now tentatively assigned color 2.
5. Conflict Check:
   - The function checks adjacent vertices again.
   - Vertex 1 has color 1, which is different from the tentative color 2 for Vertex 2.
6. Valid Color:
   - Color 2 is valid for Vertex 2.
   - Result: x=[1,2,0,0]x = [1, 2, 0, 0]x=[1,2,0,0]

Step 3: Color Vertex 3 (k = 3)

1. Initial Assignment:
   - x[3] := (x[3] + 1) mod (m + 1) = (0 + 1) mod 4 = 1
   - Vertex 3 is tentatively assigned color 1.
2. Conflict Check:
   - The function checks all vertices adjacent to Vertex 3 (Vertex 1 and Vertex 4).
   - Vertex 1 has color 1, which is the same as the tentative color for Vertex 3.
3. Conflict Detected:
   - Since there is a conflict with Vertex 1, the algorithm breaks out of the for loop.
4. Next Color Attempt:
   - x[3] := (x[3] + 1) mod (m + 1) = (1 + 1) mod 4 = 2
   - Vertex 3 is now tentatively assigned color 2.
5. Conflict Check:
   - The function checks adjacent vertices again.
   - Vertex 2 has color 2, which is the same as the tentative color for Vertex 3.
6. Conflict Detected:
   - Since there is a conflict with Vertex 2, the algorithm tries another color.
7. Next Color Attempt:
   - x[3] := (x[3] + 1) mod (m + 1) = (2 + 1) mod 4 = 3
   - Vertex 3 is now tentatively assigned color 3.
8. Conflict Check:
   - The function checks adjacent vertices.
   - Both Vertex 1 (color 1) and Vertex 2 (color 2) have different colors from Vertex 3 (color 3).

9. Valid Color:
    - Color 3 is valid for Vertex 3.
    - Result: x=[1,2,3,0]x = [1, 2, 3, 0]x=[1,2,3,0]

Step 4: Color Vertex 4 (k = 4)

1. Initial Assignment:
    - x[4] := (x[4] + 1) mod (m + 1) = (0 + 1) mod 4 = 1
    - Vertex 4 is tentatively assigned color 1.
2. Conflict Check:
    - The function checks all vertices adjacent to Vertex 4 (Vertex 2 and Vertex 3).
    - Vertex 2 has color 2 and Vertex 3 has color 3, so there is no conflict.
3. Valid Color:
    - Color 1 is valid for Vertex 4.
    - Result: x=[1,2,3,1]x = [1, 2, 3, 1]x=[1,2,3,1]

0/1 KNAPSACK PROBLEM

In the 0/1 knapsack optimization problem, Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized}$$

The goal is to maximize the total value of selected items without exceeding a given weight capacity. Each item can either be included in the knapsack (taken) or not (hence the "0/1" name).

0/1 Knapsack Problem Using Backtracking Approach

1. **Backtracking Decision Tree**:
   - For each item, we have two choices:
     - **Include the item**: Add the item's weight and value to the knapsack.
     - **Exclude the item**: Skip the item.
   - Recursively make these choices until all items have been considered or the weight limit has been reached.
2. **Base Case**:
   - If the current weight exceeds the knapsack capacity W, stop further exploration along this path.
   - If all items have been considered, check if the current selection is the best found so far.
3. **Backtrack to Explore All Paths**:
   - After choosing an item (either to include or exclude), backtrack by unselecting it and moving on to the next choice.
   - This allows exploring all possible subsets of items.

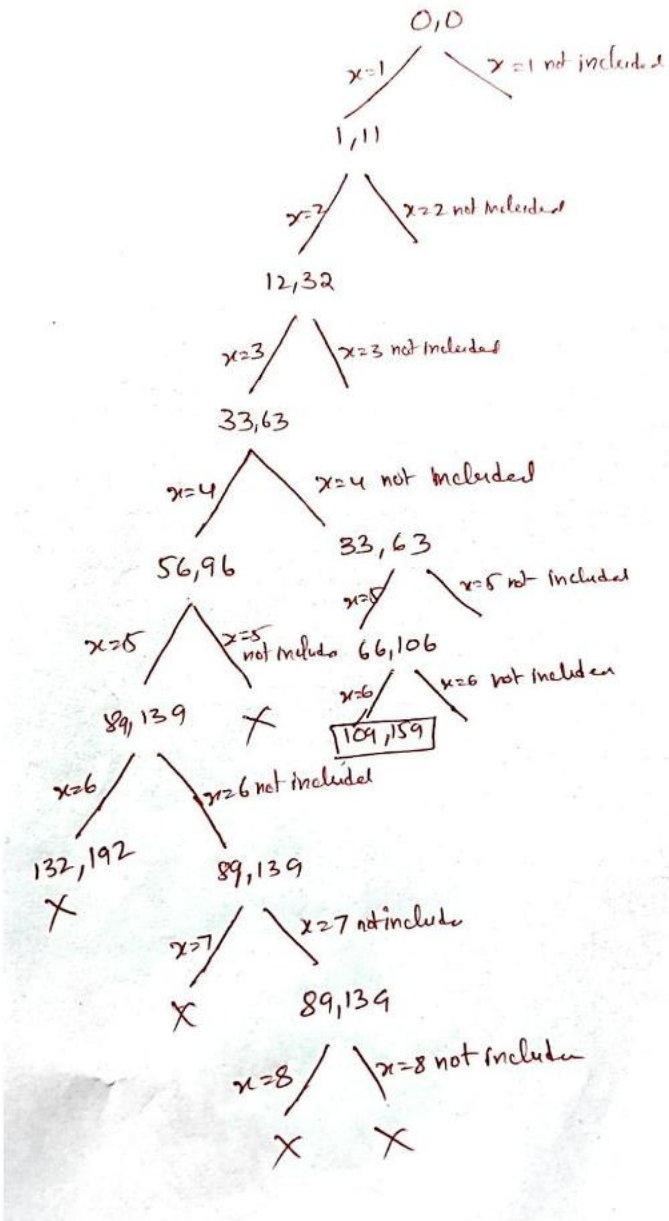
```
1   Algorithm BKnap(k, cp, cw)
2   // m is the size of the knapsack; n is the number of weights
3   // and profits. w[ ] and p[ ] are the weights and profits.
4   // p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
5   // knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
6   // is not in the knapsack; else x[k] = 1.
7   {
8       // Generate left child.
9       if (cw + w[k] ≤ m) then
10      {
11          y[k] := 1;
12          if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
13          if ((cp + p[k] > fp) and (k = n)) then
14          {
15              fp := cp + p[k]; fw := cw + w[k];
16              for j := 1 to k do x[j] := y[j];
17          }
18      }
19      // Generate right child.
20      if (Bound(cp, cw, k) ≥ fp) then
21      {
22          y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
23          if ((cp > fp) and (k = n)) then
24          {
25              fp := cp; fw := cw;
26              for j := 1 to k do x[j] := y[j];
27          }
28      }
29  }
```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

Example

Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: p={11,21,31,33,43,53,55,65}, w={1,11,21,23,33,43,45,55} m=110, and n=8.

The State Space Teee



Selected Items for Maximum Profit of 159

1. **Item 1**: Profit = 11, Weight = 1
2. **Item 2**: Profit = 21, Weight = 11
3. **Item 3**: Profit = 31, Weight = 21
4. **Item 4** Profit = 33, Weight = 23
5. **Item 5**: Profit = 43, Weight = 33
6. **Item 6**: Profit = 53, Weight = 43

Time Complexity

- The time complexity of the backtracking solution can be summarized as: $O(2^n)$
- In the worst case, the algorithm may have to explore all combinations of items, leading to exponential growth in the number of calls.

BRANCH-AND-BOUND

The General Method

Terminologies

- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.
- **Branch-an-bound** refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.
- **Heuristic** is method or trick to improving problem solving performance & Heuristics do not guarantee optimal solutions,

Definition: Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node.

- However branch and Bound ***differs*** from backtracking in two important manners:
  1. ***It has a branching function,*** which can be a depth first search, breadth first search or based on bounding function.
  2. ***It has a bounding function***, which prune efficiently the search tree.
- Branch and Bound is the ***generalization*** of both graph search strategies,
  1. **A BFS** like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
  2. **A D-search** like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

Least Cost (LC) Search

- The search for an answer node can be speeded by using an "intelligent" ranking function c( ) for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:
  - c(x) = f(h(x)) + g(x)
  - where, c(x) is the cost of x.
  - h(x) is the cost of reaching x from the root and
  - f(.) is any non-decreasing function.
  - g (x) is an estimate of the additional effort needed to reach an answer node from x.
- A search strategy that uses a cost function c(x) = f(h(x) + g(x) to select the next E-node with least c(.) is called a **LC-search** (Least Cost search)
- BFS and D-search are special cases of LC-search.
  - If g(x) = 0 and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS.
  - If f(h(x)) = 0 and g(x) >g(y) whenever y is a child of x, then the search is essentially a D-search.
- An LC-search coupled with bounding functions is called an **LC-branch and bound search** .

Control Abstraction for LC-Search

- Let t be a state space tree and c() a ***cost function*** for the nodes in t. If x is a node in t, then c(x) is ***the minimum cost of any answer node*** in the sub tree with root x.
- Thus, c(t) is the cost of a minimum-cost answer node in t. LC search uses ĉto find an answer node. The algorithm uses two functions.

1. **Least-cost()**: Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.
2. **Add_node():** to delete and add a live node from or to the list of live nodes.
3. **Add_node(x):** adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

```
Listnode = record
{
        Listnode * next, *parent; float cost;
}

Algorithm LCSearch(t)
{       //Search t for an answer node
        if *t is an answer node then output *t and return;
        E := t;         //E-node.
        initialize the list of live nodes to be empty;
        repeat
        {
                for each child x of E do
                {
                        if x is an answer node then output the path from x to t and return;
                        Add  (x);                       //x is a new live node.
                        (x → parent) := E;              // pointer for path to root
                }
                if there are no more live nodes then
                {
                        write ("No answer node");
                        return;
                }
                E := Least();
        } until (false);
}
```

Bounding

- A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.
- A ***good bounding*** helps to prune efficiently the tree, leading to a faster exploration of the solution space.
- Branch and bound ***algorithms*** are used for optimization problem where we deal directly only with minimization problems.

Three common search strategies are FIFO, LIFO, and LC. These three search methods differ only in the selection rule used to obtain the next E-node

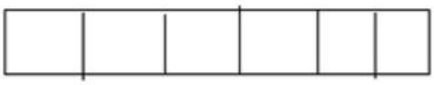
FIFO Branch and Bound

FIFO Branch and Bound is a heuristic search algorithm used to solve complex optimization problems. It is a variant of the Branch and Bound algorithm, which is a popular method for solving *discrete optimization problems.*

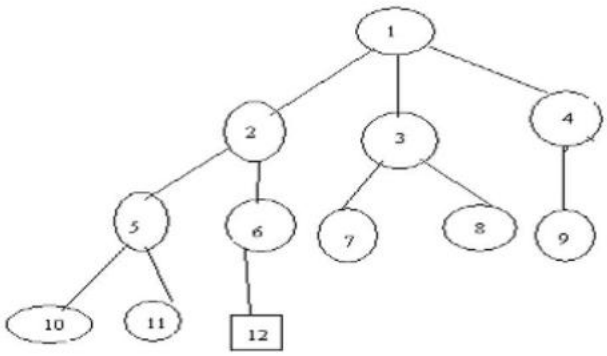
Working:

The FIFO Branch and Bound algorithm works by recursively breaking down a complex optimization problem into smaller sub-problems. Each sub-problem is solved using a bounding function, which provides an estimate of the optimal solution. The algorithm then uses a branching strategy to select the most promising sub-problems and solves them recursively. The process continues until the optimal solution is found.

For this we will use a data structure called Queue. Initially Queue is empty.



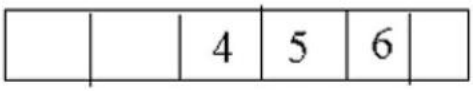
Example:



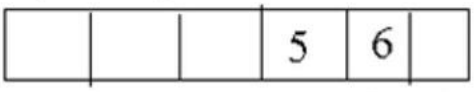
Assume the node 12 is an answer node (solution)
In FIFO search, first we will take E-node as a node 1.
Next we generate the children of node 1. We will place all these live nodes in a queue.

Now we will delete an element from queue, i.e. node 2, next generate children of node 2 and place in this queue.

Next, delete an element from queue and take it as E-node, generate the children of node 3, 7, 8 are children of 3 and these live nodes are killed by bounding functions. So we will not include in the queue.



Again delete an element from queue. Take it as E-node, generate the children of 4. Node 9 is generated and killed by boundary function.

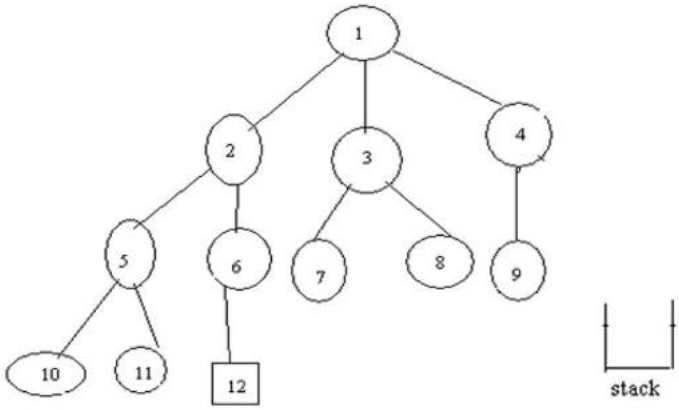


Next, delete an element from queue. Generate children of nodes 5, i.e., nodes 10 and 11 are generated and by boundary function, last node in queue is 6. The child of node 6 is 12 and it satisfies the conditions of the problem, which is the answer node, so search terminates.

LIFO Branch and Bound

For this we will use a data structure called stack. Initially stack is empty.

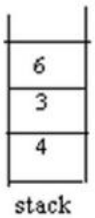
Example:



Generate children of node 1 and place these live nodes into stack.

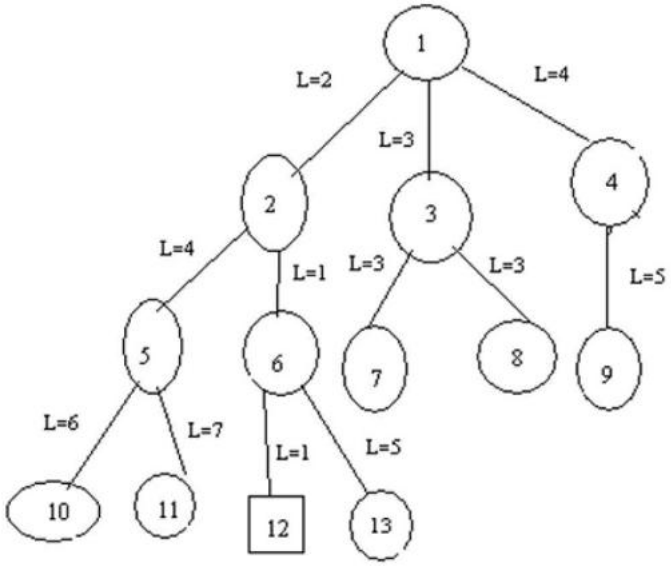
Remove element from stack and generate the children of it, place those nodes into stack. 2 is removed from stack. The children of 2 are 5, 6. The content of stack is,

Again remove an element from stack, i.,e node 5 is removed and nodes generated by 5 are 10, 11 which are killed by bounded function, so we will not place 10, 11 into stack.



LC (Least Cost) Branch and Bound Search

An LC-search coupled with bounding functions is called an **LC-branch and bound search**



Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4. By using ranking function we will calculate the cost of 2, 3, 4 nodes is $\hat{c}$ =2, $\hat{c}$ =3, $\hat{c}$ =4 respectively. Now we will select a node which has minimum cost i.,e node 2. For node 2, the children are 5, 6. Between 5 and 6 we will select the node 6 since its cost minimum. Generate children of node 6 i.,e 12 and 13. We will select node 12 since its cost ($\hat{c}$ =1) is minimum. More over 12 is the answer node. So, we terminate search process.

APPLICATION: 0/1 KNAPSACK PROBLEM (LCBB)

There are n objects given and capacity of knapsack is M. Select some objects to fill the knapsack in such a way that it should not exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is maximization problem. It means we will always seek for maximum p_1x_1 (where p_1 represents profit of object x_1).

A branch bound technique is used to find solution to the knapsack problem. But we cannot directly apply the branch and bound technique to the knapsack problem. Because the branch bound deals only the minimization problems. We modify the knapsack problem to the minimization problem. The modifies problem is,

$$\text{minimize} \quad -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

```
Algorithm Reduce(p, w, n, m, I1, I2)
// Variables are as described in the discussion.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1], 1 ≤ i < n.
{
     I1 := I2 := ∅;
     q := Lbb(∅, ∅);
     k := largest j such that w[1] + ··· + w[j] < m;
     for i := 1 to k do
     {
          if (Ubb(∅, {i}) < q) then I1 := I1 ∪ {i};
          else if (Lbb(∅, {i}) > q) then q := Lbb(∅, {i});
     }
     for i := k + 1 to n do
     {
          if (Ubb({i}, ∅) < q) then I2 := I2 ∪ {i};
          else if (Lbb({i}, ∅) > q) then q := Lbb({i}, ∅);
     }
}
```

Algorithm: KNAPSACK PROBLEM

Example: Consider the instance M=15, n=4, (p_1, p_2, p_3, p_4) = 10, 10, 12, 18 and (w_1, w_2, w_3, w_4)=(2, 4, 6, 9).

Solution: knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Arrange the item profits and weights with respect of profit by weight ratio. After that, place the first item in the knapsack. Remaining weight of knapsack is 15-2=13. Place next item w_2 in knapsack and the remaining weight of knapsack is 13-4=9. Place next item w_3, in knapsack then the remaining weight of knapsack is 9-6=3. No fraction are allowed in calculation of upper bound so w_4, cannot be placed in knapsack.

Profit= p_1+p_2+ p_3,=10+10+12

So, Upper bound=32

To calculate Lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound=10+10+12+ (3/9*18)=32+6=38

Knapsack is maximization problem but branch bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, upper bound (U) =-32

Lower bound (L)=-38

We choose the path, which has minimized difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.

Now we will calculate upper bound and lower bound for nodes 2, 3

For node 2, x_1=1, means we should place first item in the knapsack.

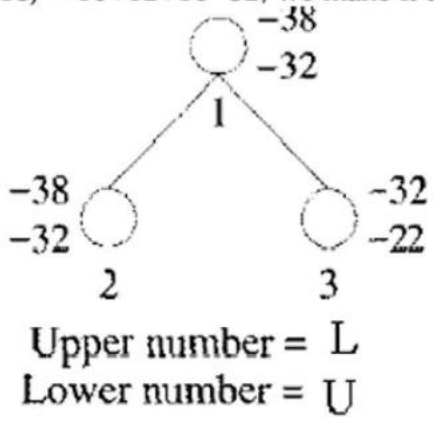
U=10+10+12=32, make it as -32

L=10+10+12+ (3/9*18) = 32+6=38, we make it as -38

For node 3, x_1=0, means we should not place first item in the knapsack.

U=10+12=22, make it as -22

L=10+12+ (5/9*18) = 10+12+10=32, we make it as -32

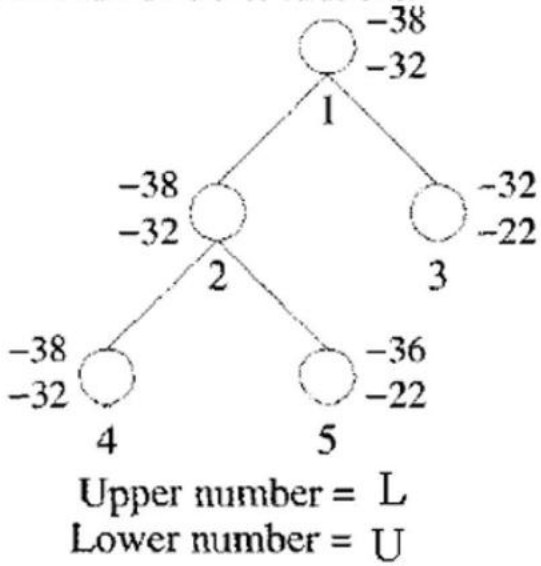


Next we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U-L=-32+38=6

For node 3, U-L=-22+32=10

Choose node 2, since it has minimum difference value of 6.

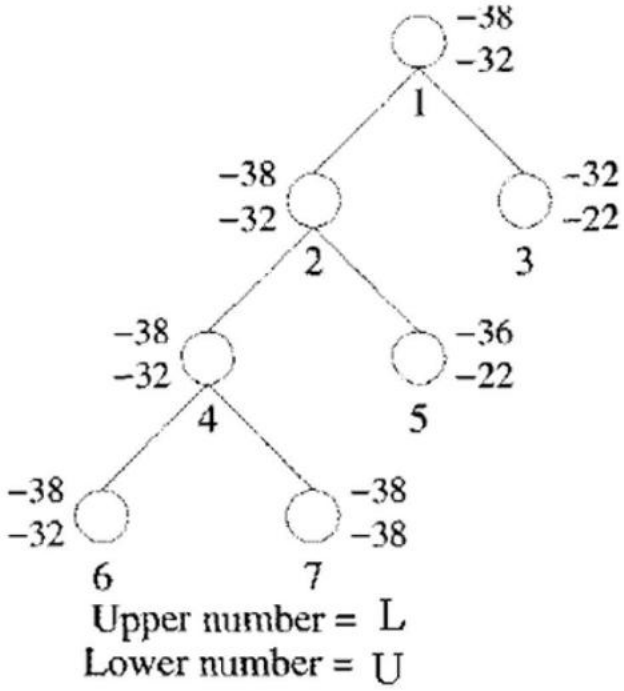


Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U-L=-32+38=6

For node 5, U-L=-22+36=14

Choose node 4, since it has minimum difference value of 6

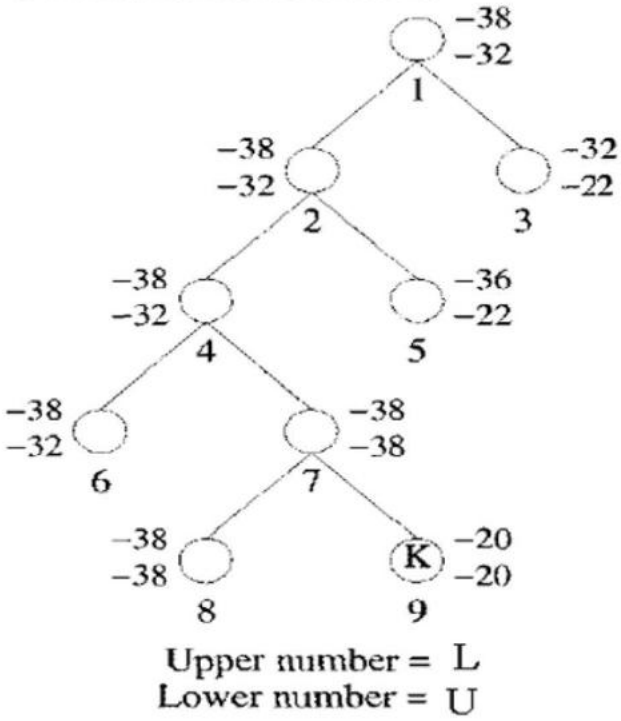


Now we will calculate lower bound and upper bound of node 6 and 7. Calculate difference of lower and upper bound of nodes 6 and 7.

For node 6, U-L=-32+38=6

For node 7, U-L=-38+38=0

Choose node 7, since it has minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 8, U-L=-38+38=0

For node 9, U-L=-20+20=0

Here, the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1→2→4→7→8

$X_1=1$

$X_2=1$

$X_3=0$

$X_4=1$

The solution for 0/1 knapsack problem is $(x_1, x_2, x_3, x_4)=(1, 1, 0, 1)$

Maximum profit is:

$\sum pixi=10*1+10*1+12*0+18*1$

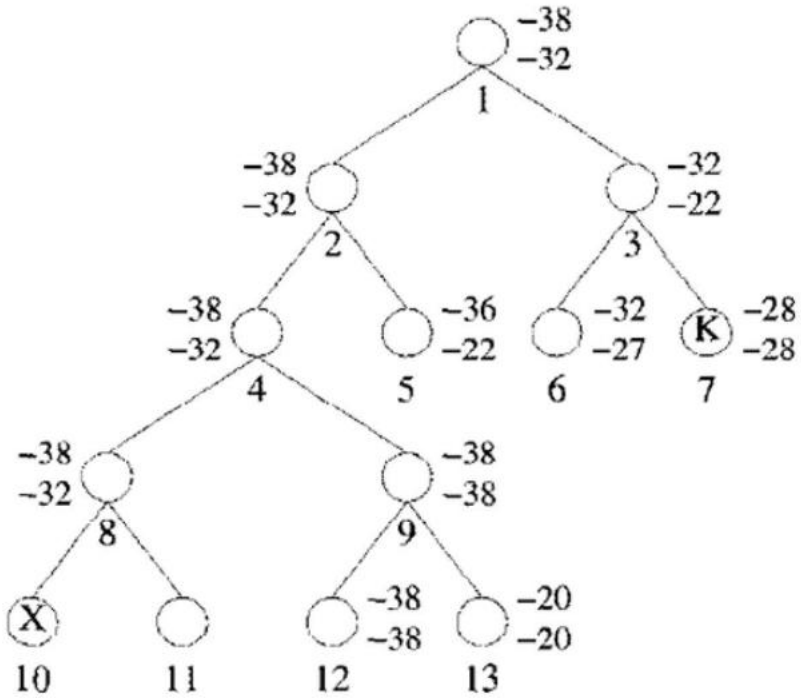
$10+10+18=38.$

FIFO Branch-and-Bound Solution

Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in above Example. Initially the root node, node 1 of following Figure, is the E-node and the queue of live nodes is empty. Since this is not a solution node, upper is initialized to u(l) = -32. We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of upper remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue.

Node 3, the next-node, is expanded. Its children nodes are generated; Node 6 gets added to the queue. Node 7 is immediately killed as L (7) > upper. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then Upper is updated to u(9) = -38, Nodes 5 and 6 are the next two nodes to become B-nodes. Neither is expanded as for each, L > upper. Node 8 is the next E-node. Nodes 10 and 11 are generated; Node 10 is infeasible and so killed. Node 11 has L (11) > upper and so is also killed. Node 9 is expanded next.

When node 12 is generated, 'Upper and ans are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as L (13) > upper. The only remaining live node is node 12. It has no children and the search terminates. The value of upper and the path from node 12 to the root is output. So solution is $X_1=1$, X2=1, X3=0, X4=1.



upper number = L
lower number = U

APPLICATON: TRAVELLING SALES PERSON PROBLEM

Let G = (V', E) be a directed graph defining an instance of the traveling salesperson problem. Let Cij equal the cost of edge (i, j), Cij = ∞ if (i, j) != E, and let IVI = n, without loss of generality, we can assume that every tour starts and ends at vertex 1.

Procedure for solving travelling sales person problem

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. This can be done as follows:

   Row Reduction:

   a) Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
   b) Find the sum of elements, which were subtracted from rows.
   c) Apply column reductions for the matrix obtained after row reduction.

   Column Reduction:

   d) Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.
   e) Find the sum of elements, which were subtracted from columns.
   f) Obtain the cumulative sum of row wise reduction and column wise reduction.

   Cumulative reduced sum=Row wise reduction sum + Column wise reduction sum.

   Associate the cumulative reduced sum to the starting state as lower bound and α as upper bound.

2. Calculate the reduced cost matrix for every node.
   a) If path (i,j) is considered then change all entries in row **i** and column **j** of A to α.
   b) Set A(j,1) to α.
   c) Apply row reduction and column reduction except for rows and columns containing only α. Let **r** is the total amount subtracted to reduce the matrix.
   d) Find ĉ(S)= ĉ(R)+A(i,j)+r.

Repeat step 2 until all nodes are visited.

Example: Find the LC branch and bound solution for the travelling sales person problem whose cost matrix is as follows.

The cost matrix is

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Step 1: Find the reduced cost matrix

Apply now reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd t row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix =

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Row wise reduction sum = 10+2+2+3+4=21.

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 2nd column.

The resulting column wise reduced cost matrix (A) =

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	3	12	∞

Column wise reduction sum = 1+0+3+0+0=4.
Cumulative reduced sum = row wise reduction + column wise reduction sum.
=21+ 4 =25.

This is the cost of a root i.e. node 1, because this is the initially reduced cost matrix.
The lower bound for node is 25 and upper bound is ∞.
Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1,3), (1, 4), (1,5).
The tree organization up to this as follows;
Variable **i** indicate the next node to visit.

Step 2:

Now consider the path (1, 2)

Change all entries of row 1 and column 2 of A to ∞ and also set A (2, 1) to ∞.

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞. Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0= 0
Cumulative reduction(r) = 0 + 0=0
Therefore, as ĉ(S)= ĉ(R)+A(1,2)+r ➔ ĉ(S)= 25 + 10 + 0 = 35.

Now consider the path (1, 3)

Change all entries of row 1 and column 3 of A to ∞ and also set A (3, 1) to ∞.

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2
15	3	∞	∞	0
11	0	∞	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction(r) = 0 +11=11
Therefore, as ĉ(S)= ĉ(R)+A(1,3)+r
ĉ(S)= 25 + 17 +11 = 53.

Now consider the path (1, 4)

Change all entries of row 1 and column 4 of A to ∞ and also set A(4,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(1,4)+r

ĉ(S)= 25 + 0 +0 = 25.

Now Consider the path (1, 5)

Change all entries of row 1 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	2	∞
0	3	∞	0	∞
15	3	12	∞	∞
∞	0	0	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

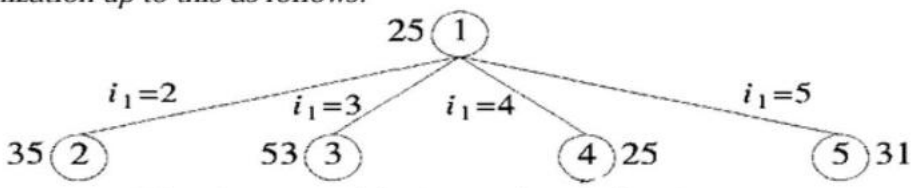
Then the resultant matrix is =

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction(r) = 5 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(1,5)+r

ĉ(S)= 25 + 1 +5 = 31.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25, (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A = \begin{matrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{matrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Now consider the path (4, 2)

Change all entries of row 4 and column 2 of A to ∞ and also set A(2,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{matrix}$$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as $\hat{c}(S) = \hat{c}(R) + A(4,2) + r$

$\hat{c}(S) = 25 + 3 + 0 = 28.$

Now consider the path (4, 3)

Change all entries of row 4 and column 3 of A to ∞ and also set A(3,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{matrix}$$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction(r) = 2 +11=13

Therefore, as ĉ(S)= ĉ(R)+A(4,3)+r

ĉ(S)= 25 + 12 +13 = 50.

.Now consider the path (4, 5)

Change all entries of row 4 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Row reduction sum =11

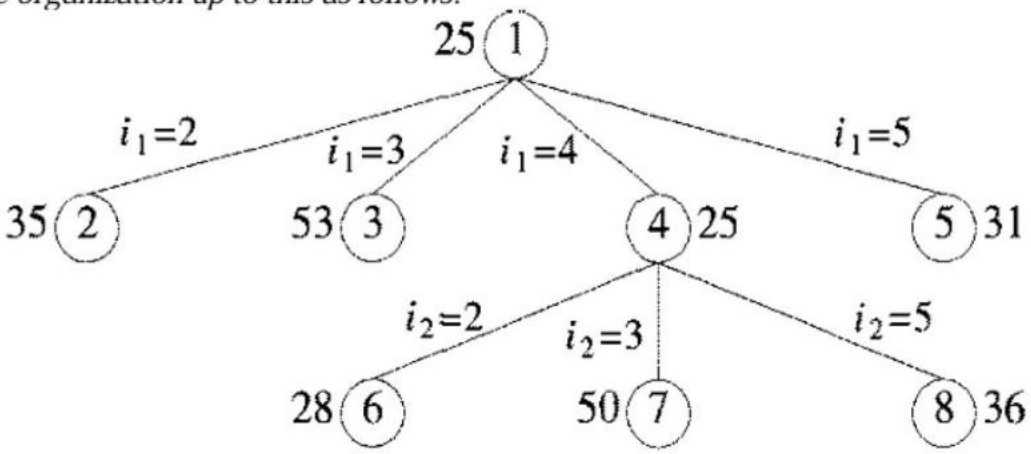
Column reduction sum = 0

Cumulative reduction(r) = 11 +0=11

Therefore, as ĉ(S)= ĉ(R)+A(4,5)+r

ĉ(S)= 25 + 0 +11 = 36.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (4, 2) = 28, (4, 3) = 50, (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

A =

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

The new possible paths are (2, 3) and (2, 5).

Now Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A(3,1) to ∞.

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	∞	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

Row reduction sum =13
Column reduction sum = 0
Cumulative reduction(r) = 13 +0=13
Therefore, as ĉ(S)= ĉ(R)+A(2,3)+r
ĉ(S)= 28 + 11 +13 = 52.

Now Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5,1) to ∞.

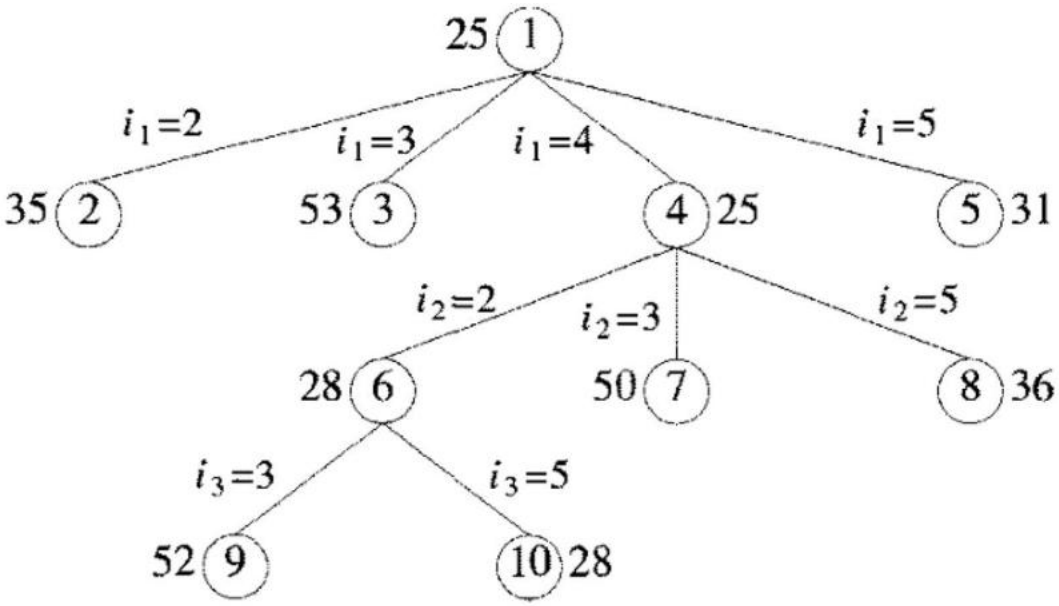
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

Row reduction sum =0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(2,5)+r ➔ ĉ(S)= 28 + 0 +0 = 28.
The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

$$A = \begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{matrix}$$

The new possible path is (5, 3).

Now c***onsider the path (5, 3):***

Change all entries of row 5 and column 3 of A to ∞ and also set A(3,1) to ∞. Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{matrix}$$

Row reduction sum =0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as $\hat{c}(S) = \hat{c}(R)+A(5,3)+r$

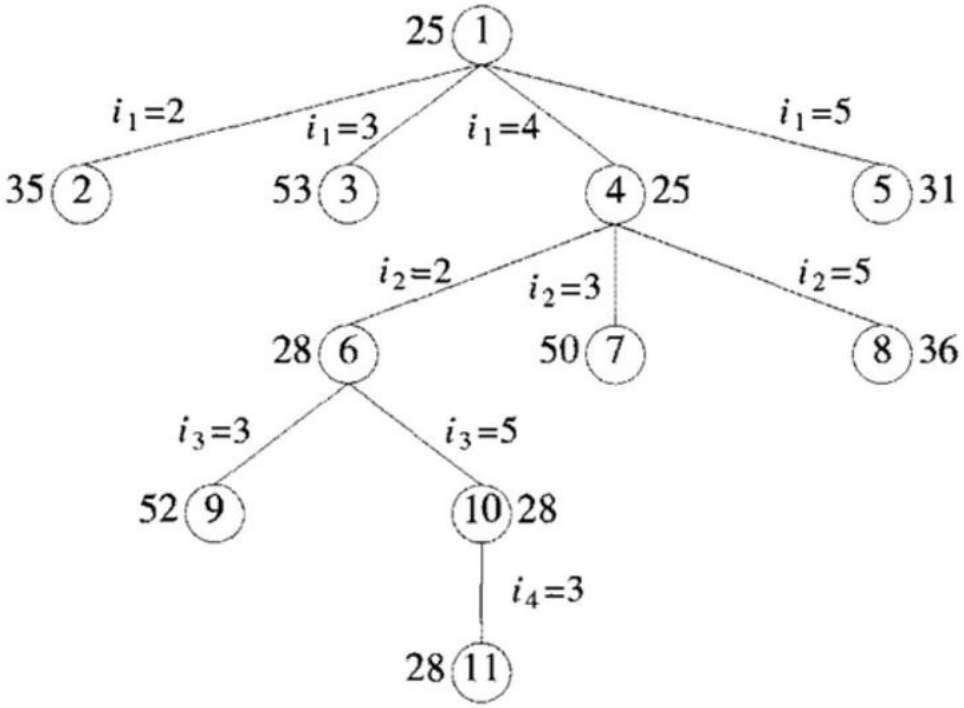
$\hat{c}(S) = 28 + 0 + 0 = 28.$

The path travelling sales person problem is:

1→4→2→5→3→1:

The minimum cost of the path is: 10+2+6+7+3=28.

The overall tree organization is as follows:



Numbers outside the node are $\hat{c}$ values

```
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);
}
```

5.7.NP-HARD AND NP-COMPLETE PROBLEMS:

5.7.1Basic 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.

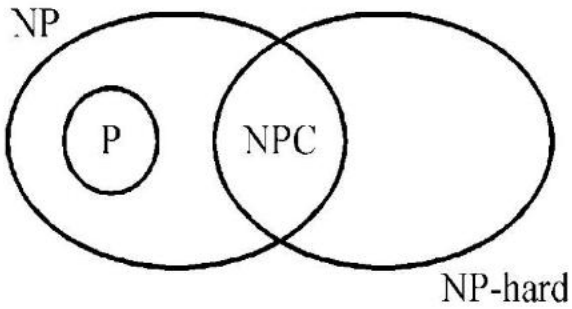
Examples: 1. Path in a graph

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



5.7.2.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

5.7.3.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)

5.8. 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

5.9.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 ($\wedge$) & or ($\vee$)

→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:

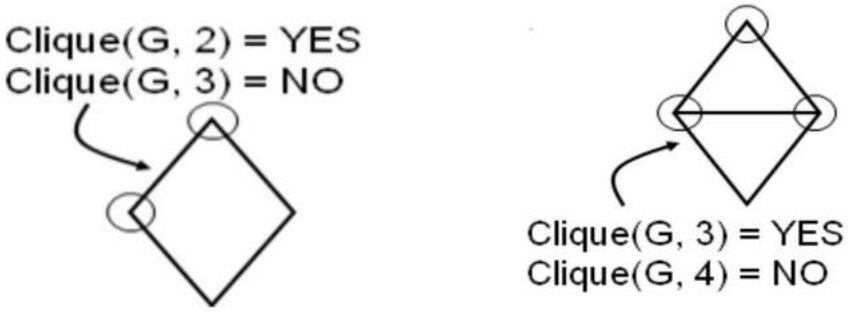
Optimization problems can be cast into decision problems by imposing a bound on output or solution. Decision problem is assumed to be easier (or no harder) to solve compared to the optimization problem. Decision problem can be solved in polynomial time if and only if the corresponding optimization problem 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();
}
```

N**on-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();
}
```

5.11.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" );
}
```

5.12.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.

Cook's Theorem Proof:

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"