UNIT – V

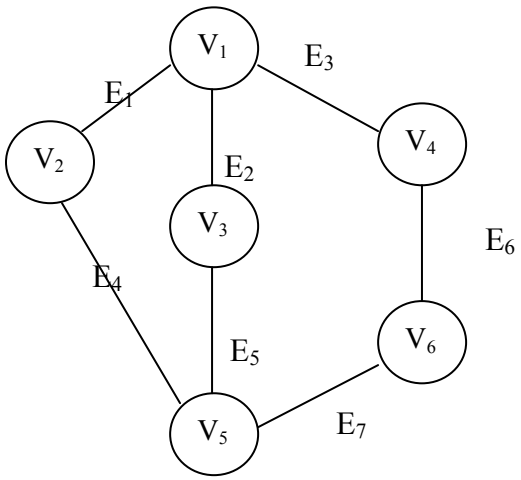
Graphs–Basic Terminology, Graph Representations- Adjacency matrix,Adjacency lists,Adjacency multilists,Graph traversals- DFS and BFS, Spanning trees-Minimum cost spanning trees,Kruskal's Algorithm for Minimum cost Spanning trees, Shortest paths- Single Source Shortest Path Problem,All Pairs Shortest Path Problem.
Text Processing - Pattern matching algorithms- The Knuth-Morris-Pratt algorithm,The Boyer-Moore algorithm,Tries- Standard Tries, Compressed Tries, Suffix tries.

GRAPHS

A graph is a collection of two sets V and E where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

Vertices are nothing but the nodes in the graph.
Two adjacent vertices are joined by edges.
Any graph is denoted as G = {V, E}

Eg:



G = {{ V1, V2, V3, V4, V5, V6}, {E1, E2, E3, E4, E5, E6, E7}}

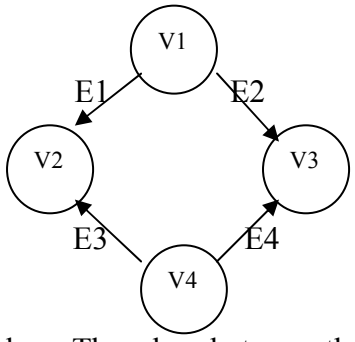
COMPARISON BETWEEN GRAPH AND TREE

S.No	Graph	Tree
1.	Graph is a non-linear data structure.	Tree is a non-linear data structure.
2.	It is a collection of vertices/nodes and edges.	It is a collection of nodes and edges.
3.	Each node can have any number of edges.	General trees consist of the nodes having any number of child nodes. But in case of binary trees every node can have at the most two child nodes.
4.	There is no unique node called root in graph.	There is a unique node called root in trees.
5.	A cycle can be formed.	There will not be any cycle.
6.	Application: For finding shortest path in networking is used	Application: For game trees, decision trees, the tree is used.

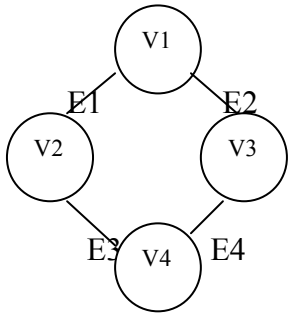
Types of Graphs

Basically graphs are of two types-

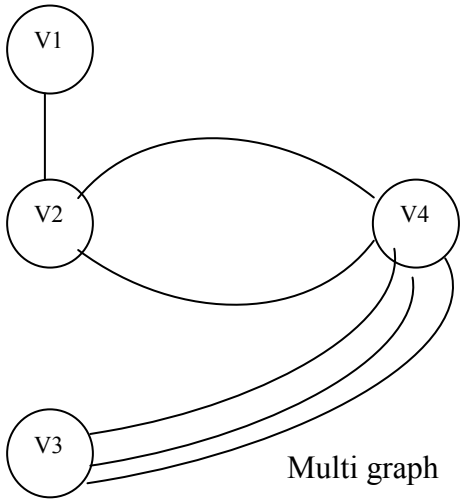
1. Directed graphs
2. Undirected graphs



In the directed graph the directions are shown on the edges. The edges between the vertices are ordered. In this type of graph, the edge E1 is in between the vertices V1 and V2. The V1 is called head and V2 is called the tail. Similarly for V1 head and the tail is V3 and so on.. We can say E1 is the set of (V1, V2) and not of (V2, V1).



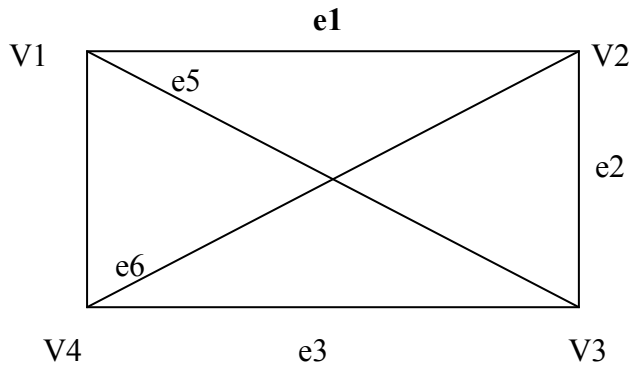
Similarly, in an undirected graph, the edges are not ordered. In this type of graph the edge E1 is set of (V1, V2) or (V2, V1).



Multi graph

BASIC TERMINOLOGIES

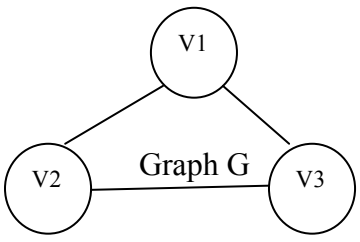
1. **Complete graph :** If an indirected graph of n vertices consists of n(n-1)/2 number of edges then that graph is called a complete graph.



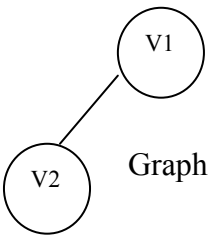
Here n = 4
e = n(n-1)/2 = 4(3)/2 = 6

Subgraph:

A subgraph G' of graph G is a graph such that the set of vertices and set of edges of G' are proper subset of the set of edges of G.

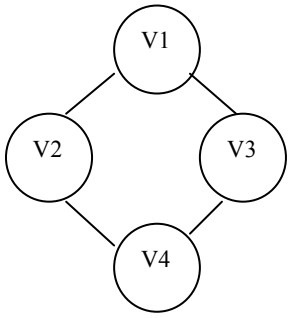


Graph G

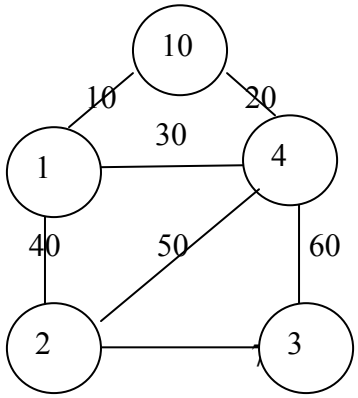


Graph G'

Connected Graph:
An undirected graph is said to be connected if for every pair of distinct vertices Vi and Vj in V(G) there is an edge Vi to Vj in G.

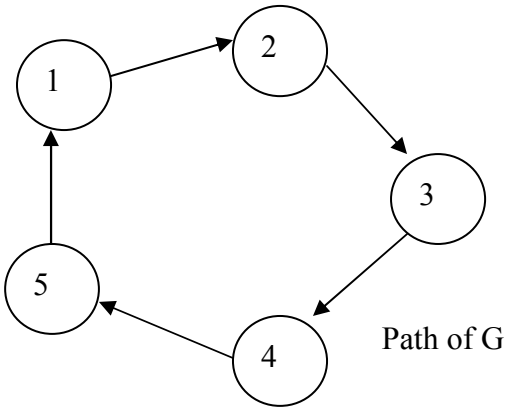


Weighted Graph:
A weighted graph is a graph which consists of weights along its edges.



Path:

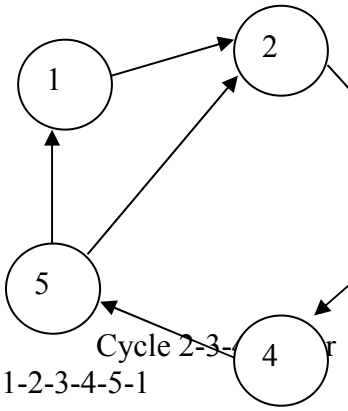
A path is denoted using sequence of vertices and there exists as edge from one vertex to the next vertex.



Path of G is 1-2-3-4-5

Cycle:

A closed walk through the graph with repeated vertices, mostly having the same starting and ending vertex is called a cycle.

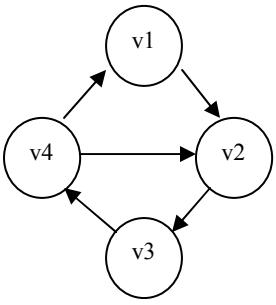


Cycle 2-3-4-5-2 or
1-2-3-4-5-1

In-degree and out-degree:

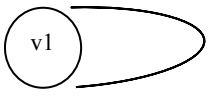
ices	In-degree	Out-degree
V1	1	1
V2	2	1
V3	1	1
V4	1	2

The degree vertex is the number of edges associated with the vertex.
In-degree of a vertex is the number of edges that incident to the vertex. Out-degree is the total number of edges that are going away
from the vertex.



Self loop:

Self loop is an edge that connects the same vertex to itself.



Representation of Graphs

Normally a graph can be represented by two representations and those are

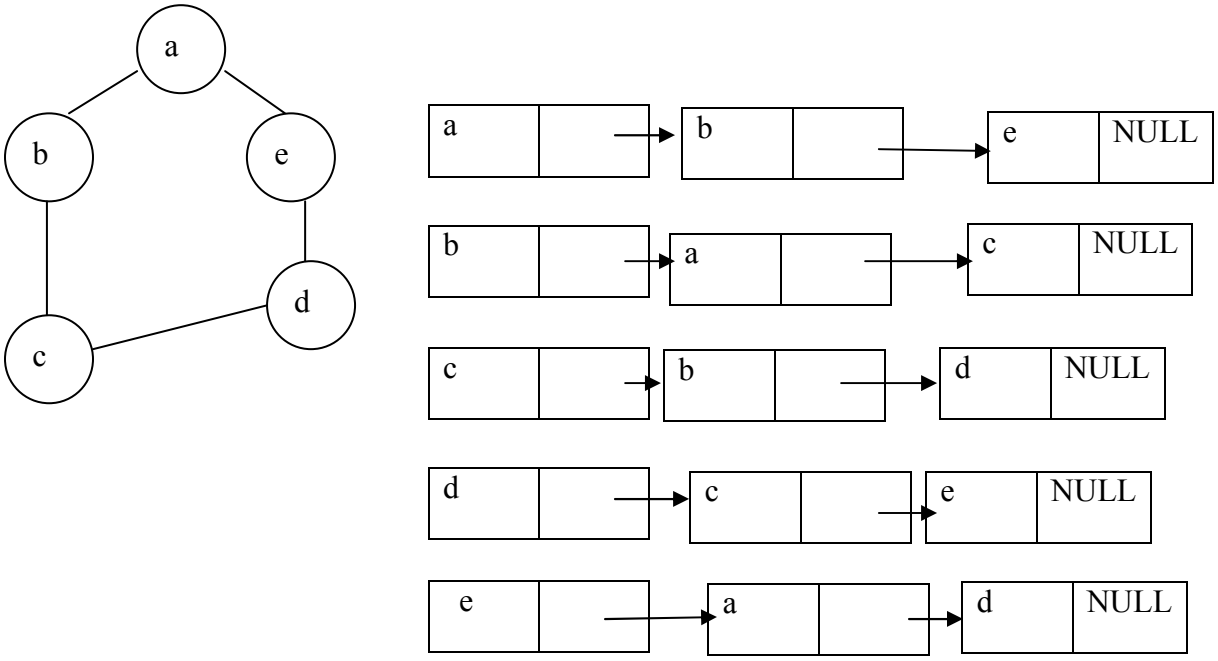
Adjacency matrix: In this representation, matrix or 2 dimensional array is used to represent the graph.

Consider a graph G of n vertices and the matrix M. If there is an edge present between vertices Vi and Vj then M[i][j] = 1 else M[i][j] = 0. For an undirected graph if M[i][j]= 1 then M[j][i] =1.

In the weighted graph, weights or distances are given along every edge. Hence in an adjacency matrix representation any edge which is present between vertices Vi and Vj is denoted by its weight. Hence M[i][j] = weight of edge.

If there is no edge between Vi and Vj then, M[i][j] = 0.

Adjacency list:
In this representation, a linked list is used to represent a graph.



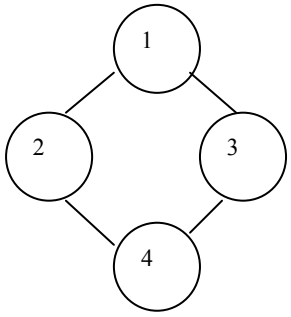
GRAPH TRAVERSAL

The graph can be displayed using

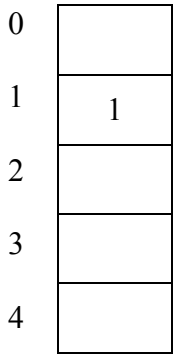
1. Breadth First Search
2. Depth First Search

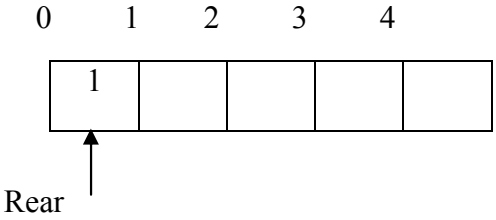
BREADTH FIRST SEARCH

In BFS the queue is maintained for storing the adjacent nodes and an array 'visited' is maintained for keeping the track of visited nodes. i.e. once a particular node is visited it should not be revisited again.



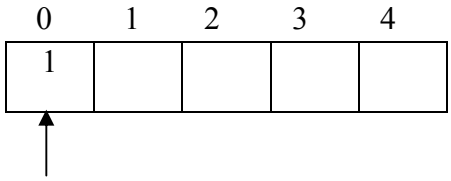
Step 1: Start with vertex 1





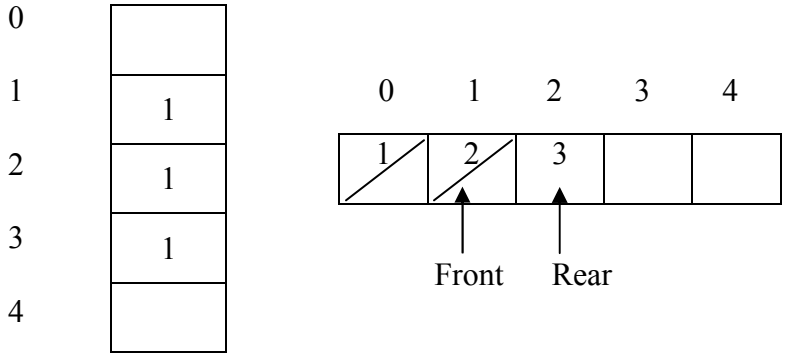
Inserted vertex 1 in queue and marked the index 1 of visited array by 1.

Step 2:



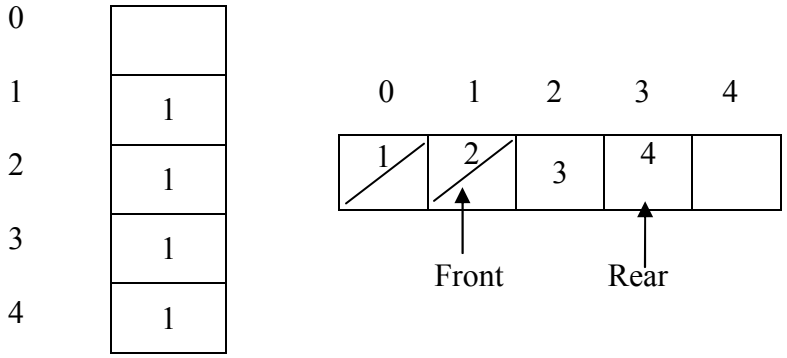
Delete 1 and print it so '1' gets printed.

Step 3: Find adjacent vertices of vertex 1 and mark them as visited, insert those in Queue.

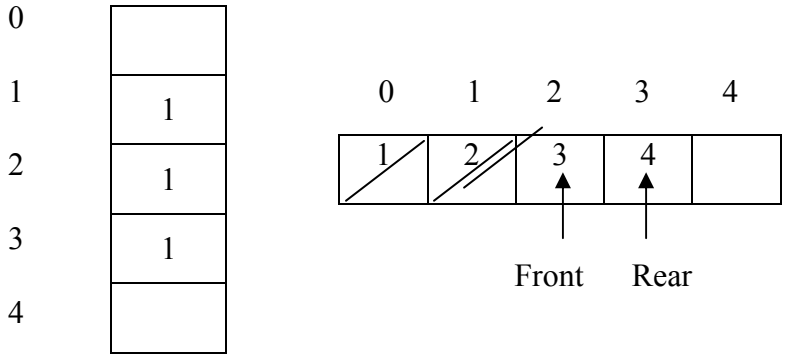


Increment front by 1 delete '2' from Queue and print it so '2' gets printed.

Step 4: Find adjacent to '2' and insert those nodes in Queue as well as mark them as visited.



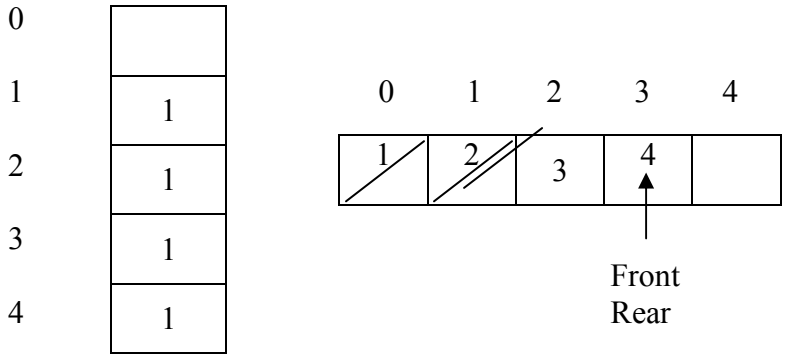
Step 5: Increment front and delete the node print it.



So '3' gets printed.

Step 6: Find adjacent to '3' i.e. 4 check whether it is marked as visited. If it is marked as visited do not insert in the queue.

Increment front, delete the node from Queue and print it.



So '4' gets printed since front = rear stop the procedure.

So output will be BFS for above graph as

1 2 3 4

Analysis of BFS

If the graph is created using adjacency matrix in BFS routine then while statement executes for n times. Inside this while statement there is a for loop which executes for all the vertices. Hence time complexity of BFS is $O(V^2)$.

If the graph is created using adjacency list then each vertex V is enqueued and dequeued at most once. Scanning for all the adjacent vertices takes O(|E|) time. Hence the time complexity of BFS is O(|V|+|E|).

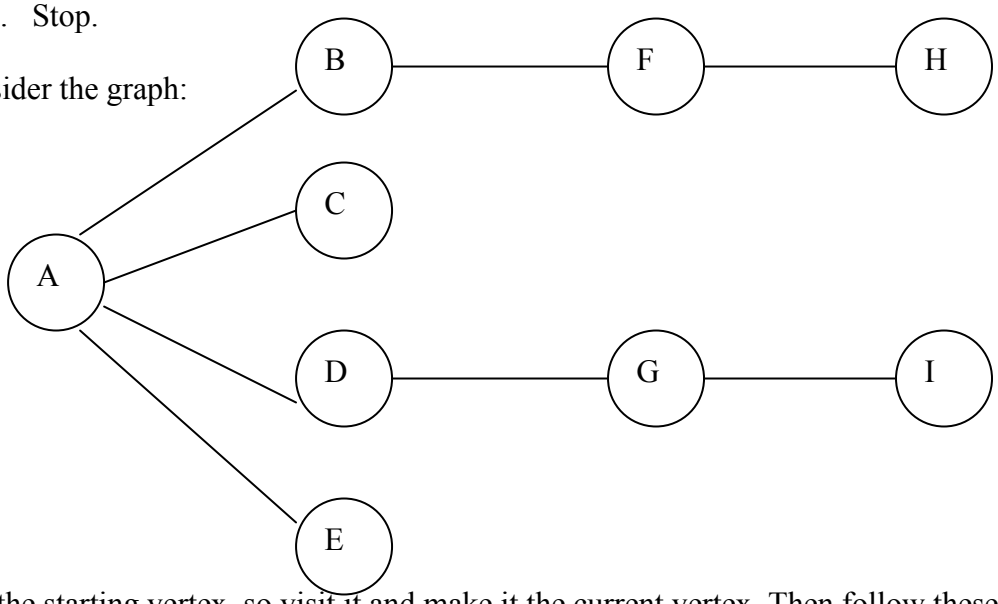
- In BFS we start from some vertex and find all the adjacent vertices of it. This process will be repeated for all the vertices so that the vertices lying on same breadth get printed.
- For avoiding repetition of vertices, we maintain array of **visited** nodes.
- A **queue** data structure is used to store adjacent vertices.

Algorithm:

1. Create a graph. Depending on the type of graph, i.e. directed or undirected set the value of the flag as either 0 or 1 respectively.
2. Read the vertex from which you want to traverse the graph say Vi.
3. Initialize the visited array to 1 at the index of Vi.
4. Insert the visited vertex Vi in the queue.
5. Visit the vertex which is at the front of the queue. Delete it from the queue and place its adjacent nodes in the queue.

6. Repeat the step 5, till the queue is not empty.
7. Stop.

Consider the graph:



A is the starting vertex, so visit it and make it the current vertex. Then follow these rules:

Rule 1: Visit the next unvisited vertex that's adjacent to the current vertex, mark it, and insert it onto the queue.

Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue and make it the current vertex.

Rule 3: If you can't carry Rule 2 because the queue is empty. You are finished.

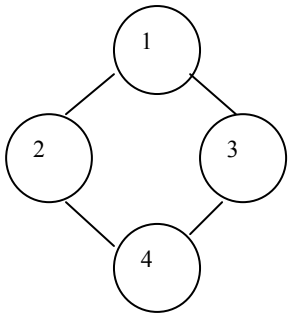
QUEUE CONTENTS

Event	Queue
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	

DEPTH FIRST SEARCH

In DFS we start from one vertex and traverse the path as deeply as we can go. When there is no vertex further, we traverse back and search for unvisited vertex.

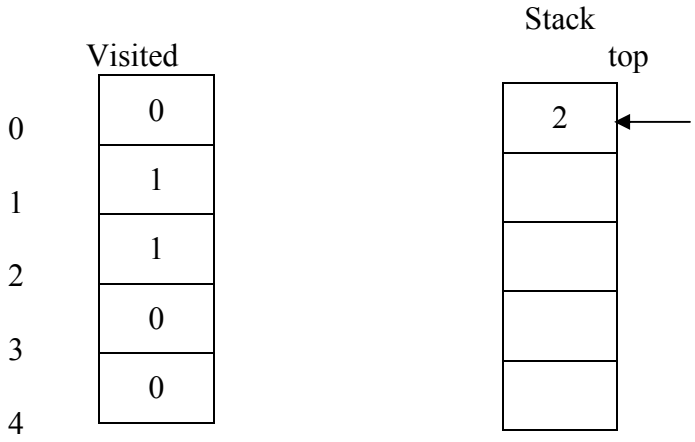
In DFS the basic data structure for storing the adjacent nodes is stack.



Step 1: Start with vertex 1, print it so '1' gets printed. Mark 1 as visited.

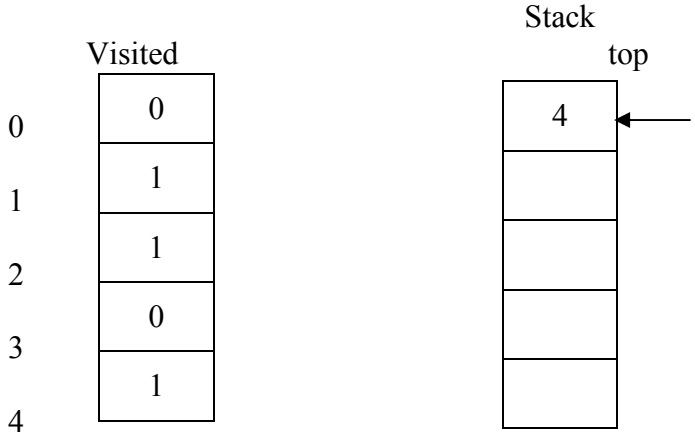
	Visited
0	0
1	1
2	0
3	0
4	0

Step 2: Find adjacent vertex to 1, say 2, if it is not visited, call DFS(2) i.e. 2 will get inserted into the stack, mark it as visited.



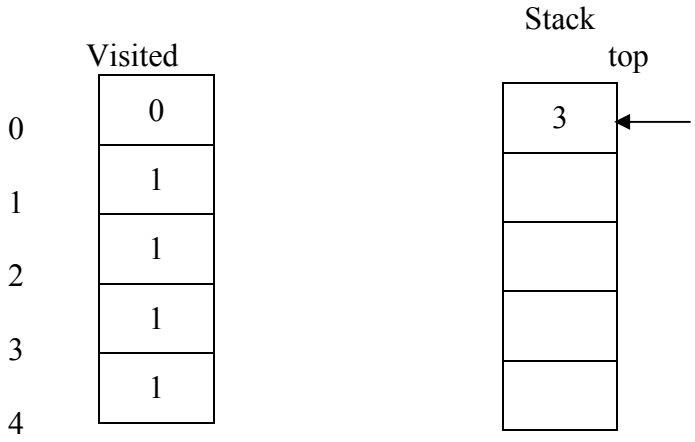
After exiting loop, 2 will be popped, print '2'

Step 3: Find adjacent to '2' i.e. vertex 4, if it is not visited call DFS(4) i.e. 4 will get pushed on to the stack mark it as visited.



After exiting the loop 4 will be popped and print '4'.

Step 4: Find adjacent to 4 i.e. vertex 3, if it is not visited call DFS(3), i.e 3 will be pushed on to the stack, mark it as visited.



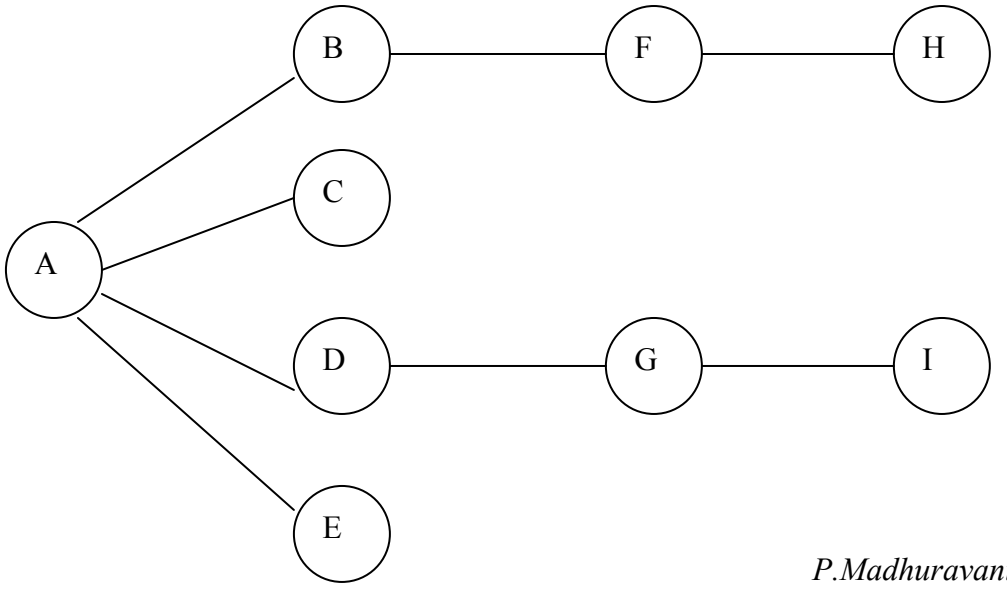
After exiting the loop 3 will be popped and print '3'.

Since all the nodes are covered stop the procedure.

Output of DFS : 1 – 2 – 4 – 3.

Algorithm:

1. Set visit(V) to 1
2. Display the visited vertex V
3. Loop for each adjacent vertex A of V
4. Check whether visit(A) is zero then call DFS(A).
5. end loop.
6. Stop.



To carry out depth first search, pick a starting point-in this eg, vertex A. Then do 3 things: visit this vertex, push it onto a stack, and mark it so won't visit it again.

Next go to any vertex adjacent to A that hasn't been visited. Assume the vertices are selected in alphabetical order, so that brings up B. Visit B, mark it, and push it on the stack.

At B, do the same thing as before: go to an adjacent vertex that hasn't been visited. This leads to F. Call this process Rule 1.

Rule 1: If possible, visit an adjacent unvisited vertex, mark it, and push it on the Stack.

Applying Rule 1 again leads to H. At this point, there are no unvisited vertices adjacent to H. Now apply Rule 2.

Rule 2: If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

Following this rule, pop H off the stack, which brings back to F. F has no unvisited adjacent vertices, so pop it. Ditto B. Now only A is left on the stack.

A have unvisited adjacent vertices, so visit the next one C. But C is the end of the line again, so pop it and back to A. Visit D, G and I and then pop them all when you reach the dead end I. Now we are back to A. Visit E, and again back to A.

This time A has no unvisited neighbors, so pop it off the stack. But now there is nothing left to pop, which brings up Rule 3.

Rule 3: If we can't follow Rule 1 or Rule 2, we are finished.

The order in which we visit the vertices is ABFHCDGIE

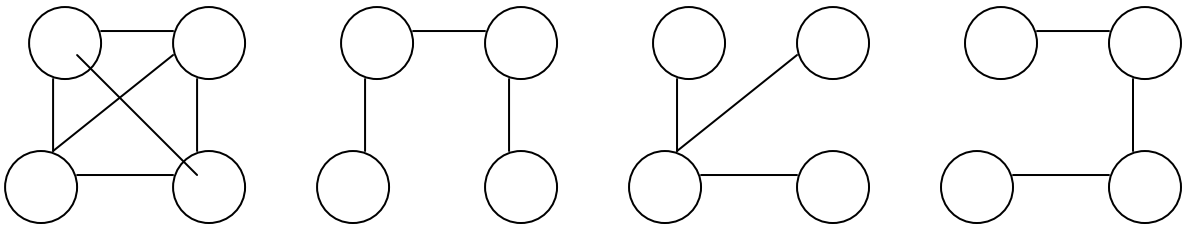
STACK CONTENTS

Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	

MINIMUM COST SPANNING TREES

Definition: Let G = (V, E) be an undirected connected graph. A sub graph t = (V, E') of G is a spanning tree of G iff t is a tree.

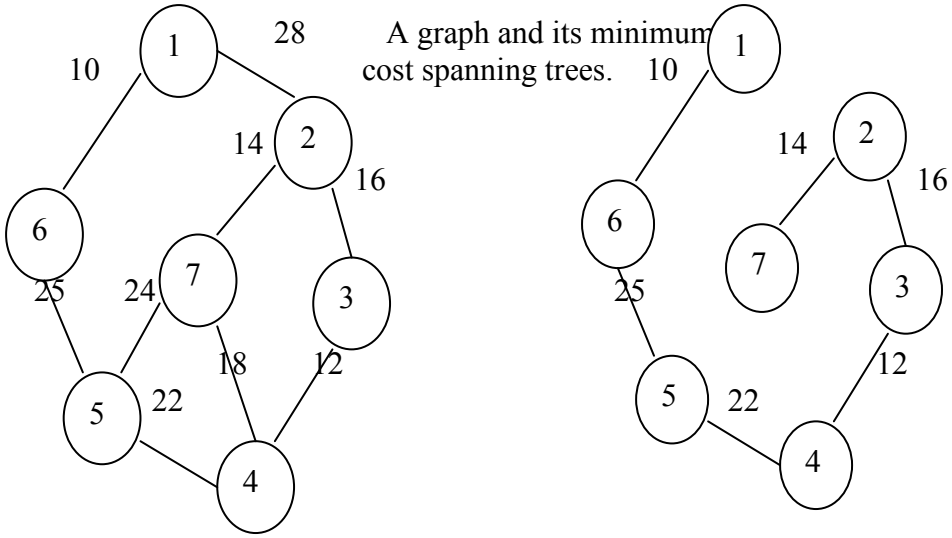
Example:



An undirected graph and three of its spanning trees.

Spanning trees have many applications. An application of spanning trees arises from the property that a spanning tree is a minimal sub graph G' of G such that V(G') = V(G) and G' is connected. (A minimal sub graph is one with the fewest number of edges). Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is n-1. The spanning trees of G represent all feasible choices.

In practical, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree. If this is not so, then the selection of links contain a cycle. Removal of any one of the links on this cycle results in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. The cost of a spanning tree is the sum of the costs of the edges in that tree.



A graph and its minimum cost spanning trees.

There are two possible ways to interpret this criterion.

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

Prim's Algorithm

A greedy method to obtain a minimum cost spanning tree builds the tree edge by edge. The next edge to include is chosen according to some optimization criterion. The criterion to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected from a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that AU{(u, n)} is also a tree.

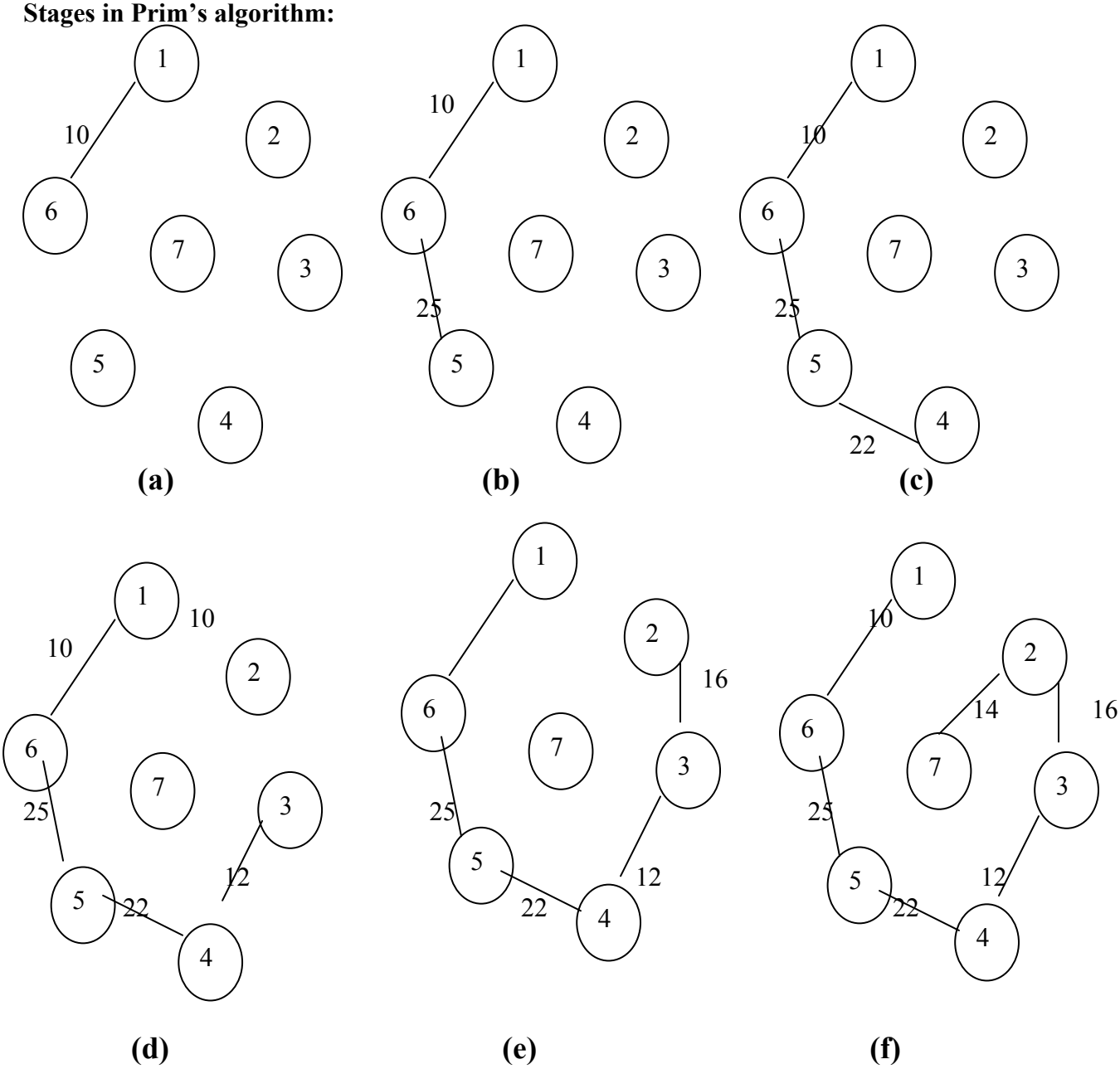
Let us obtain a pseudo code algorithm to find a minimum-cost spanning tree using this method. The algorithm will start with a tree that includes only a minimum cost edge of G. Then edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j), cost[i, j] is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value near[j]. The value near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j]. We define near[j] = 0 for all vertices j that are already in the tree. The next edge to include is defined by the vertex j such that near[j]≠0 (j not already in the tree) and cost[j, near[j]] is minimum.

Algorithm: line procedure PRIM(E, COST, n, T, mincost)

```
real COST(n, n), mincost;
integer NEAR(n), n, i, j, k, l, T(1:n-1, 2(;
(k,l) ←edge with minimu cost
mincost ← COST(k, l)
(T(1, 1), T(1, 2)) ← (k, l)
for i←1 to n do
if COST(i, l) < COST(i, k) then NEAR(i) ← l
else NEAR(i) ← k endif
repeat
NEAR(k) ← NEAR(l) ← 0;
for i←2 to n-1 do
let j be an index such that NEAR(j)≠0 and COST(j, NEAR(j)) is minimum
(T(1, 1), T(1, 2)) ← (j, NEAR(j))
mincost ← mincost + COST(j, NEAR(j))
NEAR(j) ← 0
for k←1 to n do
if NEAR(k)≠0 and COST(k, NEAR(k)) > COST(k, j) then NEAR(k) ← j
endif
repeat
```

repeat
if mincost≥∞ then print (“no spanning tree”) endif
end PRIM

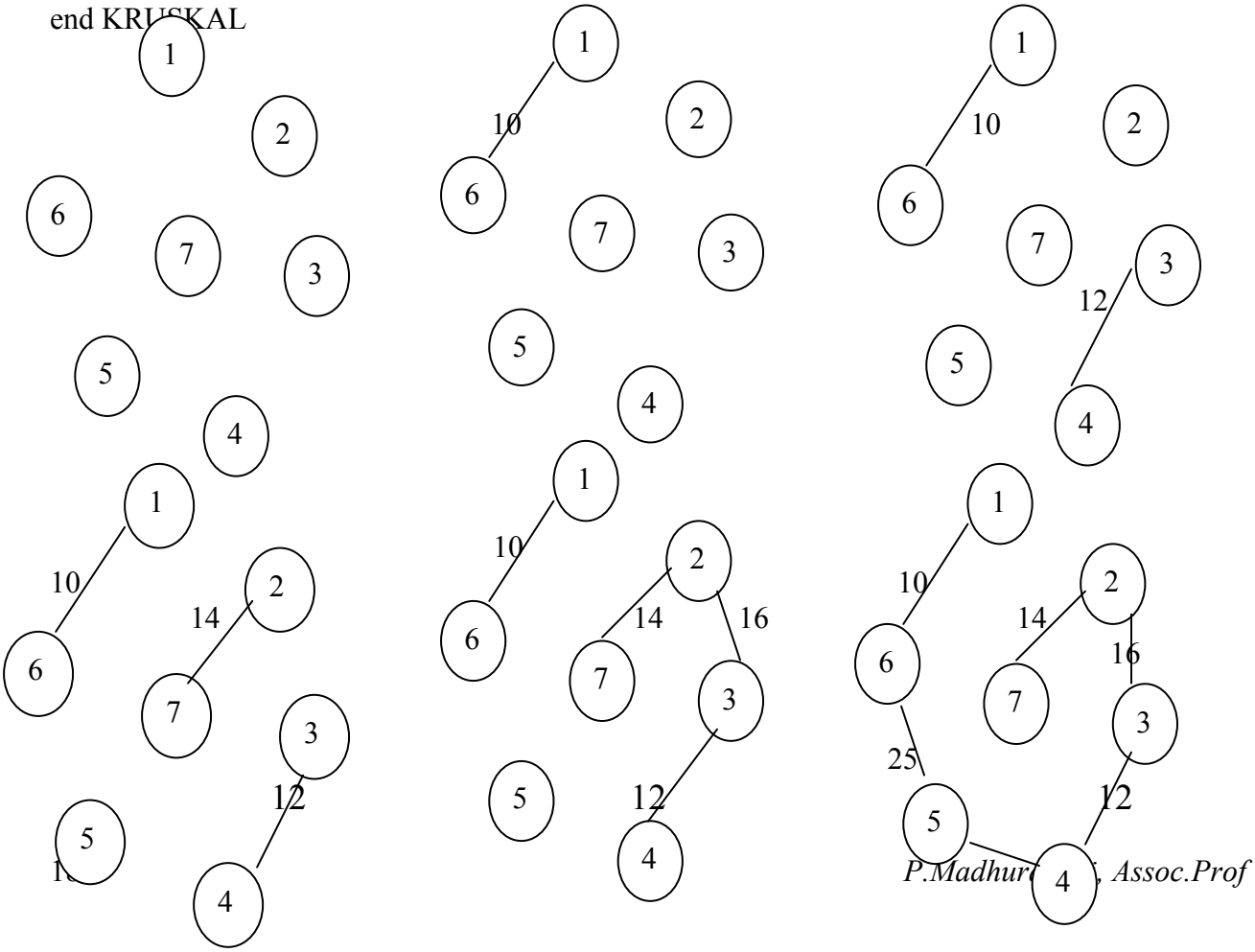
Stages in Prim’s algorithm:



Kruskal's Algorithm

There is a second possible interpretation of optimization criteria mentioned earlier in which the edges of the graph are considered in non decreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to complete t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t.

Algorithm: line procedure KRUSKAL(E, COST, n, T, mincost)
real mincost, COST(1:n, 1:n),
INTEGER parent(1:n), T(1:n-1, 2), n
construct a heap out of the edge costs using HEAPIFY
PARENT ← -1
i←mincost←0
while i<n-1 and heap not empty do
delete a minimum cost edge (u, v) from the heap and reheapify using ADJUST
j←FIND(u); k←FIND(v);
if j≠k then i←i+1
T(i, 1) ← u; T(1, 2) ←v
mincost←mincost+COST(u, v)
call UNION(j, k)
endif
repeat
if i≠n-1 then print ("no spanning tree") endif
return
end KRUSKAL



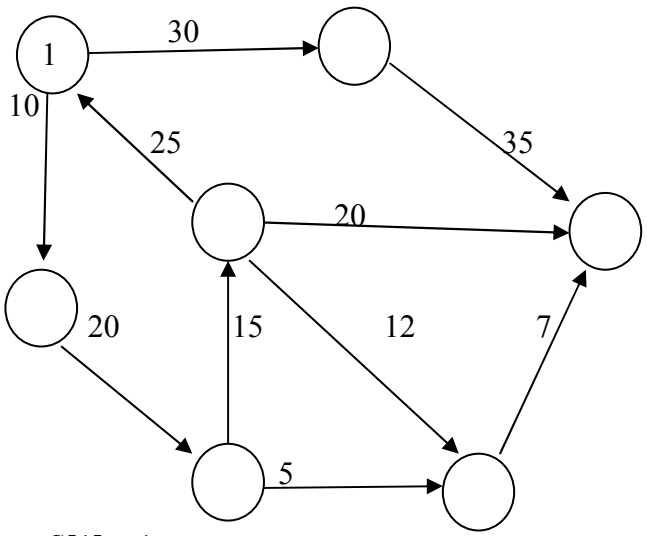
SINGLE SOURCE SHORTEST PATH PROBLEM

Many times, Graph is used to represent the distance between two cities. Everybody is often interested in moving from one city to another as quickly as possible. The single source shortest path is based on this.

In single source shortest path problem the shortest distance from a single vertex called source is obtained.

Let G(V, E) be a graph, then in single source shortest path the shortest paths from vertex v_0 to all remaining vertex is obtained. The vertex v_0 is then called as source and the last vertex is called destination. It is assumed that all the distances are positive.

Example: Consider a graph G as given below:



start from vertex 1. Hence set S[1] = 1.

Now shortest distance from vertex 1 is 10. i.e 1→2 = 10. Hence {1, 2} and min = 10.

From vertex 2 the next vertex chosen is 3.

{1, 2} = 10
{1, 3} = ∞
{1, 5} = ∞
{1, 6} = ∞
{1, 7} = ∞

Now
{1, 2, 3} = 30
{1, 2, 4} = ∞
{1, 2, 7} = ∞
{1, 2, 5} = ∞
{1, 2, 6} = ∞

Hence select 3.

S[3] = 1

{1, 2, 3, 4} = 45
{1, 2, 3, 5} = 35
{1, 2, 3, 6} = ∞
{1, 2, 3, 7} = ∞

Hence select next vertex as 5.
S[5] = 1

Now
{1, 2, 3, 5,6} = ∞
{1, 2, 3, 5,7} = 42

Hence vertex 7 will be selected. In single source shortest path if destination vertex is 7 then we have achieved shortest path 1 – 2 – 3 – 5 – 7 with path length 42. The single source shortest path from each vertex is summarized as below -

1, 2	10
1, 2, 3	30
1, 2, 3, 4	45
1, 2, 3, 5	35
1, 2, 3, 4, 7	65
1, 2, 3, 5, 7	42
1, 6	30
1, 6, 7	65

Algorithm ShortestPaths(v, cost, dist, n)
//dist[j], 1≤j≤n, is set to the length of the shortest path from vertex v to vertex j in a digraph G //with n vertices. dist[v] is set to zero. G is represented by its cost adjacency matrix cost [1:n, 1:n]

```
{
        for i:=1 to n do
        {
                S[i] := false; dist[i] := cost[v, i];
        }
        S[v] := true; dist[v] := 0.0;  // Put v in S
        for num := 2 to n-1 do
        {
                //Determine n-1 paths from v.
                Choose u from among those vertices not in S such that dist[u] is minimum;
                S[u] := true;  //Put u in S.
```

```
            for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
            dist[w] := dist[u] + cost[u, w];
        }
}
```

ALL PAIRS SHORTEST PATH PROBLEM

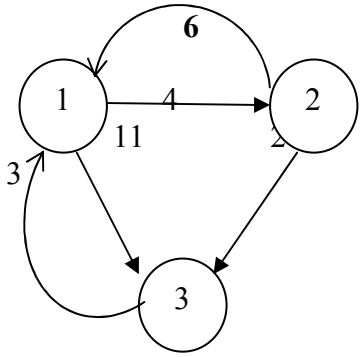
Let G = (V, E) be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that *cost*(i, i) = 0, $1 \le i \le n$. Then *cost*(i, j) is the length (or cost) of edge (i, j) if (i, j)$\in$E(G) and *cost*(i, j) = α if $i \neq j$ and (i, j) $\notin$ E(G). The all pairs shortest path problem is to determine a matrix A such that A(i, j) is the length of a shortest path from i to j.

We will apply dynamic programming to solve the all pairs shortest path
Step 1: We will decompose the given problem into sub problems. Let, A^k (i, j) be the length of shortest path from node i to j such that the label for evry intermediate node will be $\le$k. We will compute A^k for k=1 … n for n nodes.
Step 2: For solving all pair shortest path, the principle of optimality is used. That means any sub path of shortest path is a shortest path between the end nodes. Divide the paths from node i to j for every intermediate node k. Then there arises two cases:

i) Path going from i to j via k.
ii) Path which is not going via k. Select only shortest path from two cases.

Step 3: The shortest path can be computed using bottom up computation method. Following is recursion method.
Initially $A^0(i, j) = cost(i, j)$
A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$
If it does not, then no intermediate vertex has index greater than k-1 Hence , $A^k(i, j) = A^{k-1}(i, j)$
combining, , $A^k(i, j) = \min \{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}$, $k \ge 1$

Example:



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	α	0

a) A^0

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

b) A^1

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

c)A^2

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

d) A^3

Algorithm:

Algorithm AllPaths(cost, A, n)
//cost [1:n, 1:n] is the cost adjacency matrix of a graph with n vertices; A[i, j] is the cost of a
//shortest path from vertex i to vertex j. cost[i, i] = (0, 0) for 1≤i≤n.

```
{
        for i := 1 to n do
                for j := 1 to n do
                A[i, j] := cost[i, j];
        for k := 1 to n do
                for i := 1 to n do
                        for j := 1 to n do
                        A[i, j] := min (A[i, j], A[i, k] + A[k, j]);
}
```

Analysis:

Time complexity : $O(n^3)$

PATTERN MATCHING

A string is a sequence of characters.
Let text[0,..,n-1] be the string of length n, and pattern[0,…,m-1] be some substring of length m, then pattern matching is a technique of finding the substring in text which is equal to pattern.
The pattern matching problem is also called as PMP.

Application of pattern matching

1. Pattern matching technique is used in text editors.
2. Search engines use the pattern matching algorithm for matching the query submitted by the user.
3. In biological research pattern matching algorithm is used.

PATTERN MATCHING ALGORITHMS

There are various algorithms used to implement the pattern matching problem. Some of them are

1. Brute Force
2. Boyer – Moore
3. Knuth-Morris-Pratt(KMP)

BRUTE FORCE PATTERN MATCHING ALGOITHM

The Brute Force pattern matching algorithm compares the pattern P with text T for consecutive values of i such that i ranges from n-m where n is total length of string T and m is the length of pattern P. This search continues until

a) A match is found.
b) All placements of the pattern have been tried and no match has been found.

The Brute Force algorithm is as given below-

```
Algorithm Brute-Force(T[0..n], P[0..m])
for i <- 0 to n-m-1 do
j<-0;
while (j<m) and (T[i+j] = P[j])
{
j <- j + 1;
if (j=m) then
return(i);
return (-1);
}
```

Example:

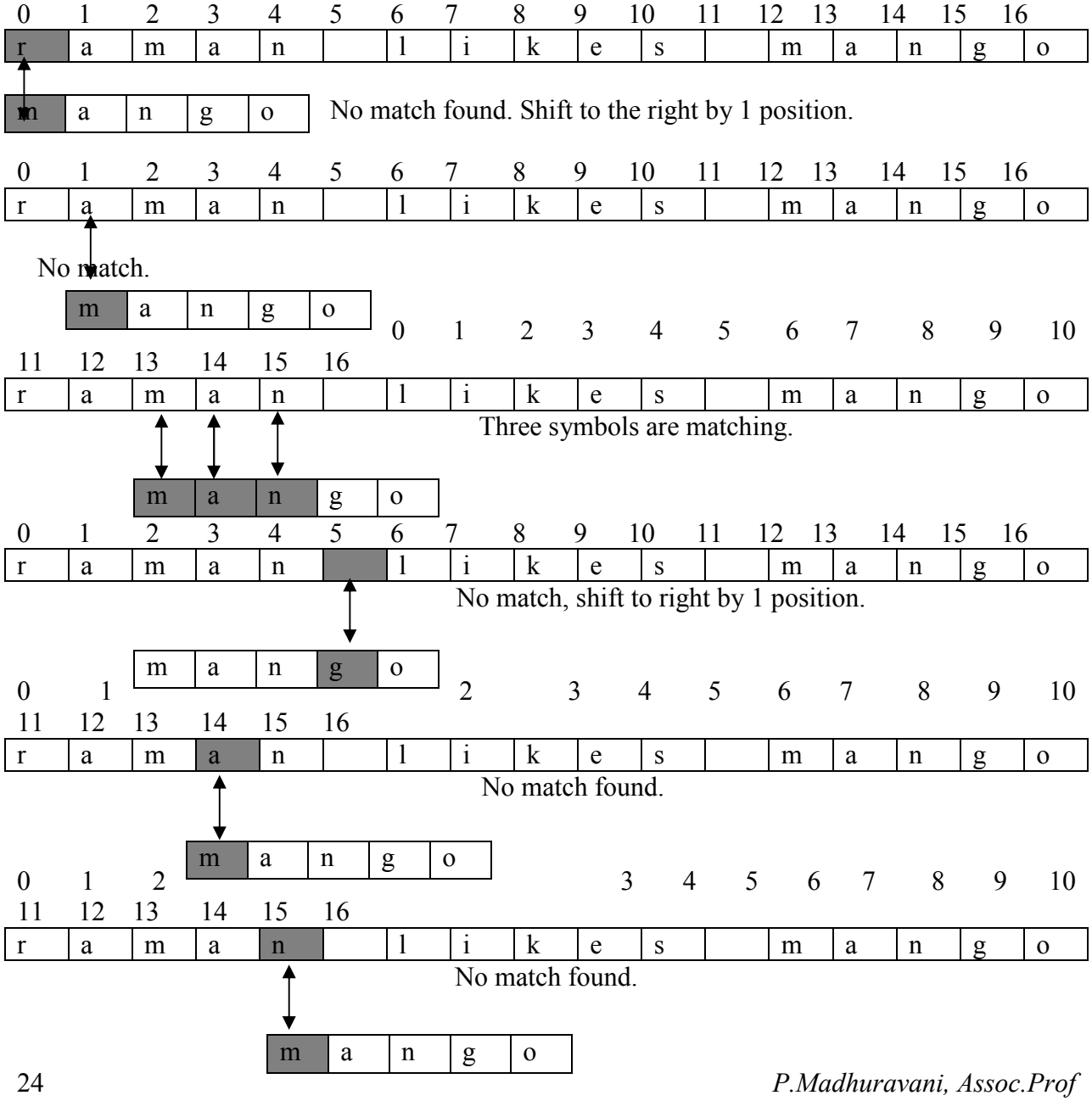
Text

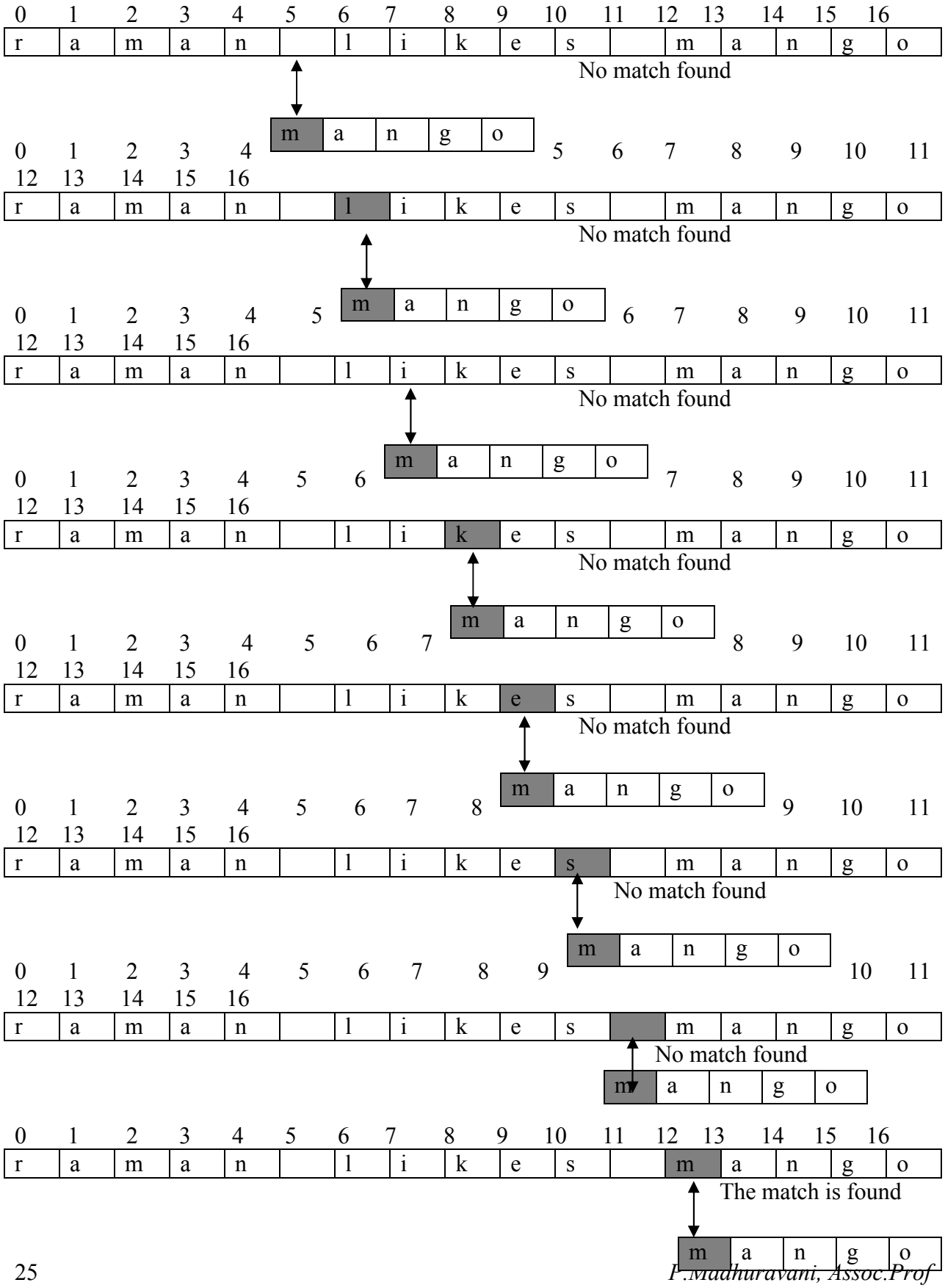
r	a	m	a	n		l	i	k	e	s		m	a	n	g	o
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Pattern

m	a	n	g	o
0	1	2	3	4

We will start finding for pattern from 0^{th} location in Text. If the match is not found the shift to the right by 1 position.




0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
r a m a n l i k e s m a n g o
No match found
m a n g o
No match found
m a n g o
No match found
m a n g o
No match found
m a n g o
No match found
m a n g o
No match found
m a n g o
No match found
m a n g o
The match is found
m a n g o

Hence return index 12, because a match with the pattern is found from that location in text.

Analysis

If a match is not found then shift the pattern to right by 1 position i.e., almost always we are shifting the pattern to the right. The worst case occurs when we have to make all the m comparisons. This results **worst case** time complexity of ɵ(mn). For a typical word search in natural language the **average case** time complexity is ɵ(n).

BOYER-MOORE PATTERN MATCHING ALGOITHM

The Boyer-Moore algorithm was invented by **Boyer** and **Moore**. Hence is the name. The Boyer-Moore scans the characters of the search pattern from right to left. If a match is not found then a shift is made by some number of characters. This algorithm is also called "**looking glass heuristic**".

Algorithm Boyer-Moore(T[0..n], P[0..n])

```
i <- m-1;
j<- m-1;

while i<n   // loop to the end of text

if P[i] = T[i] then    // if both characters match

if j = 0 then // reached end of P
return i;  // found a match

else
//go to next char
i <- i-1;
j <- i-1;

else
//skip over the whole word or shift to last occurance
i <- i + m – min(j, 1+last[T[i]]);
j <- m-1;

return -1; //no match
```

Analysis

The worst case time complexity of Boyer-Moore is O(mn+|∑|). This algorithm works faster than Brute-Force algorithm. This algorithm works efficiently when the text length is long.

Example

consider a text T = XYXZXXYXTZXYXZXXYXXYXXYY

to match against the pattern P = XYXZXY

We first build the last table using last (c) function where c represents characters from T.

Consider the pattern XYXZXY

X	Y	X	Z	X	Y
0	1	2	3	4	5

right most occurrence of Z last (Z) = 3

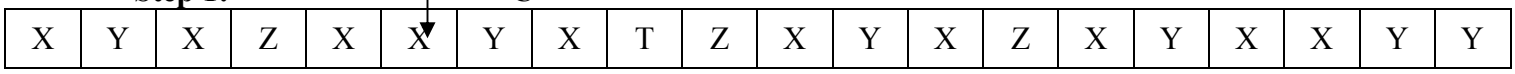
right most occurrence of X last (X) = 4

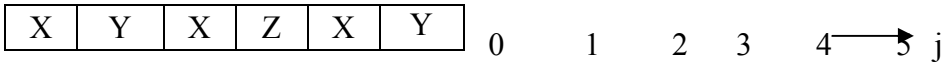
right most occurrence of Y last(Y) = 5

The remaining character in text string is T which is not present in pattern.
Its last (T) = -1.

C	X	Y	Z	T
last(c)	4	5	3	-1

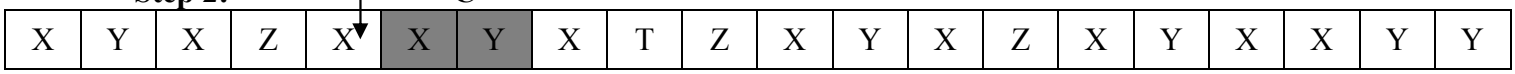
Step 1:

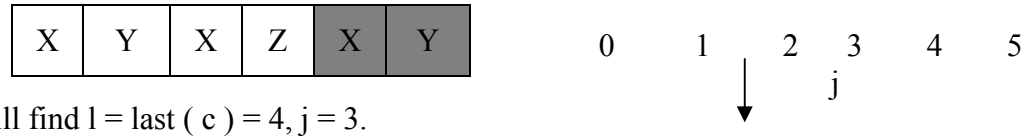




We will find l = last (c) = 4, j = 5.
As 1 + *l* ≤ j i.e. 1 + 4 ≤ 5 we shift the pattern by j – *l* i.e. 5 – 4 = 1

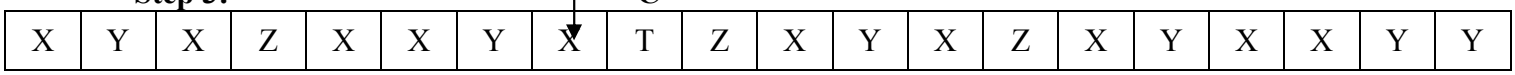
Step 2:





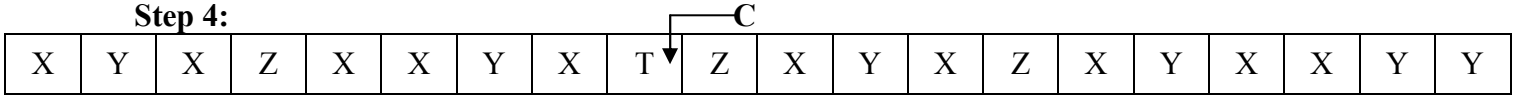
We will find l = last (c) = 4, j = 3.
As j < *l* +1 i.e. 3 < 4 + 1 we shift the pattern by 1

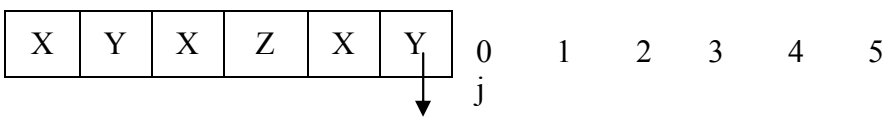
Step 3:



We will find l = last (c) = last (X) = 4, j = 5
As *l* +1 ≤ j, we shift the pattern by j – *l* i.e. 5 – 4 = 1

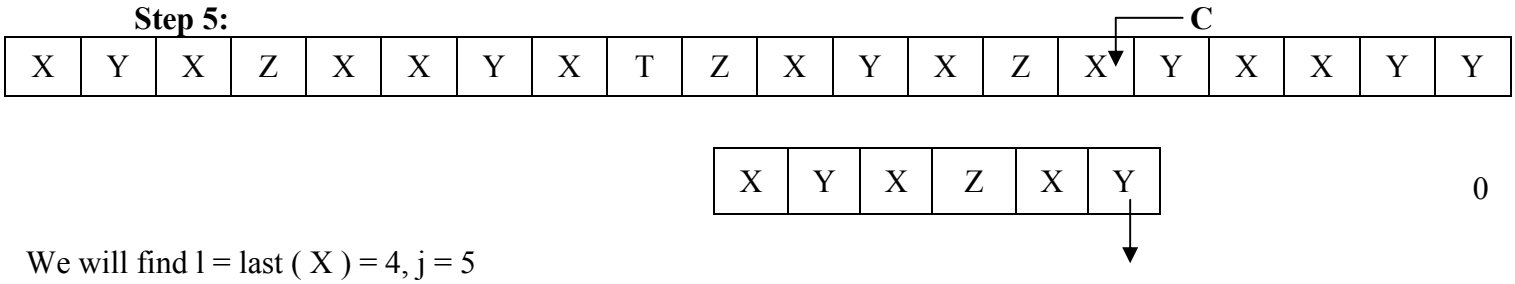
Step 4:





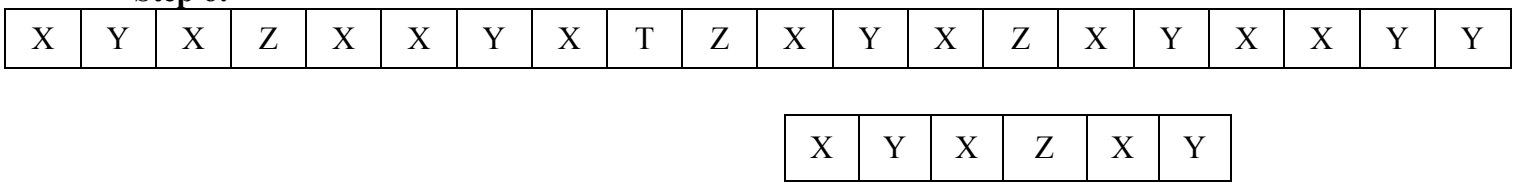
We will find l = last (c) = last (T) = -1, j = 5
As l +1 ≤ j, we shift the pattern by j – l i.e. 5 – (-1) = 6

Step 5:



We will find l = last (X) = 4, j = 5
As l +1 ≤ j, we shift the pattern by j – l i.e. 5 – 4 = 1

Step 6:



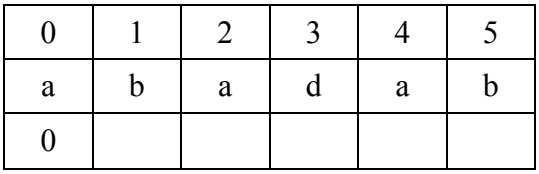
Now the match for the given pattern is found in the given string.

KNUTT MORRIS PRATT PATTERN MATCHING ALGORITHM

In the pattern matching algorithms like naïve method or Boyer-Moore, we often compare the pattern characters that do not match in the text, and on occurrence of mismatch we simply throw a way the information and restart the comparison, for another set of characters from the text. Thus again and again with next incremental position of text, the characters from pattern are matched. This ultimately reduces the efficiency of pattern matching algorithm. Hence the Knuth-Morris-Pratt algorithm came up which avoids the repeated comparison of characters. This algorithm is named after the scientists Knuth, Morris and Pratt.

The basic idea behind the algorithm is to build a **prefix array.** Some times this array is also called ∏ array. This prefix array is built using the prefix and suffix information of pattern. The overlapping prefix and suffix is used in KMP algorithm.

The KMP algorithm achieves the efficiency of O(m+n) which is optimal in worst case.
Suppose we have a pattern "abadab". The prefix array for this pattern can be built as follows:

Initially we will put 0 in 0^{th} location of prefix array.	

Consider a string **ab**
No match of prefix and suffix. Hence we will put 0 in prefix array at 1st location

Prefix : €, a
Suffix : €, b

0	1	2	3	4	5
a	b	a	d	a	b
0	0				

Consider a string **aba**
The length of matching prefix and suffix is 1. Hence make entry 1 in prefix table.

Prefix : €, a, ab
Suffix : €, a, ba

0	1	2	3	4	5
a	b	a	d	a	b
0	0	1			

Consider a string **abad**
The length of matching prefix and suffix is 0. Hence make entry 0 in prefix table.

Prefix : €, a, ab, aba
Suffix : €, d, ad, bad

0	1	2	3	4	5
a	b	a	d	a	b
0	0	1	0		

Consider a string **abada**
The length of matching prefix and suffix is 1. Hence make entry 1 in prefix table.

Prefix : €, a, ab, aba, abad
Suffix : €, a, da, ada, bada

0	1	2	3	4	5
a	b	a	d	a	b
0	0	1	0	1	

Consider a string **abadab**
The length of matching prefix and suffix is 2. Hence make entry 2 in prefix table.

Prefix : €, a, ab, aba, abad, abada
Suffix : €, d, ab, dab, adab, badab

0	1	2	3	4	5
a	b	a	d	a	b
0	0	1	0	1	2

Thus we have completed the prefix table before preceding for the actual algorithm. The pseudo code for computing prefix array is as given below:

Algorithm Compute_Prefix(char p[size])

```
prefix_table[0] <- 0
for(q <- 1 to m) do
{
while(k > 0 AND p[k] != p[q])
k <- prefix_table [k-1];
if(p[k] = p[q]) then
k <- k+1
prefix_table[q] <- k;
}
```

Now the next step is to compare the characters of pattern against text. It can be well understood with following example:

Consider

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b
P	a	b	a	b	a	d	a								

Prefix table for the pattern ababada

0	1	2	3	4	5	6
a	b	a	b	a	d	a
0	0	1	2	3	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	b	a	d	b	a	b	a	b	a	b	a	d	a	a	b
P	a	b	a	b	a	d	a								

Compare b and a, as it is not matching. We will compare Text[1] with Pattern[0].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	a	d	b	a	b	a	b	a	b	a	d	a	a	b
a	b	a	b	a	d	a								

As Text[1] is not matching with Pattern[0], we will now compare Text[2] with Pattern[1].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	a	d	b	a	b	a	b	a	b	a	d	a	a	b
a	b	a	b	a	d	a								

As Text[2] is not matching with Pattern[1], we will now backtrack on Pattern and compare Pattern[0] with Text[3]. Because we consult prefix table[1] which is 0. Hence Pattern[0] is compared with Text[3].

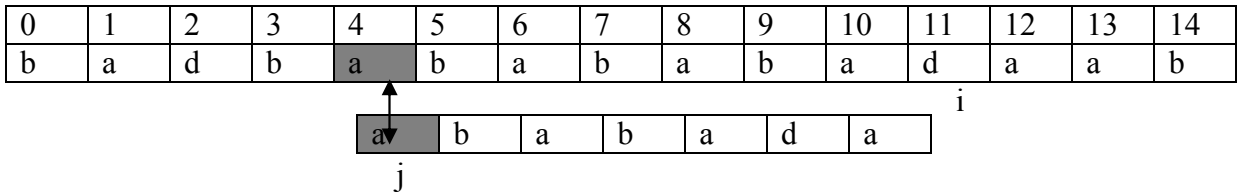
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
b	a	d	b	a	b	a	b	a	b	a	d	a	a	b

i

a	b	a	b	a	d	a

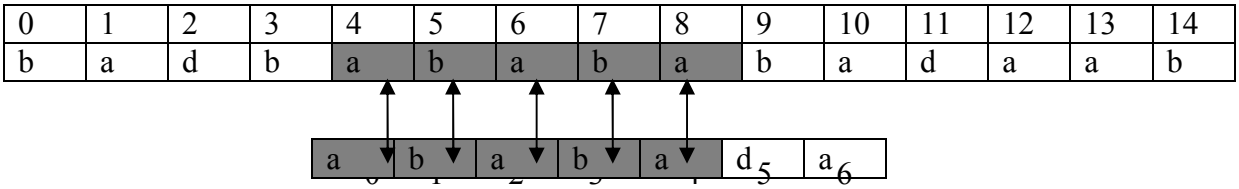
j

Again Text[3] is not matching with Pattern[0]. We will then ask prefix table [0] for the location of pattern. As prefix table[0] is 0, we will compare Pattern[0] with Text[4]

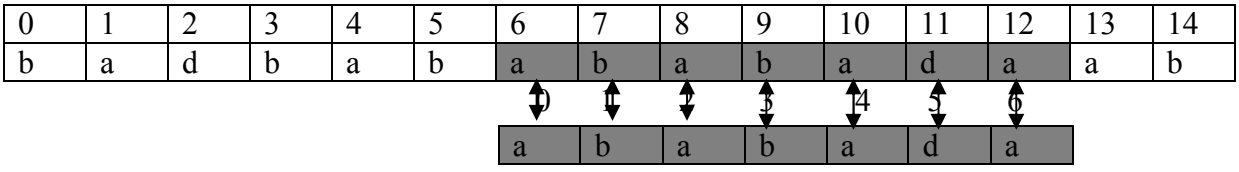


Text[4] matches with Pattern[0]. Increment i and j
Text[5] matches with Pattern[1]. Increment i and j
Text[6] matches with Pattern[2]. Increment i and j
Text[7] matches with Pattern[3]. Increment i and j
Text[8] matches with Pattern[4]. Increment i and j

But Text[9] is not matching with Pattern[5]



Hence we must backtrack on pattern array. That means j will be positioning on location 4. Consult prefix_table[4] which denotes the value 3. That indicates, compare pattern[3] with current i positiontext array character. Hence we will compare Text[9] with Pattern[3], which is matching.
Text[10] with Pattern[4], matching. Increment i and j.
Text[11] with Pattern[5], matching. Increment i and j
Text[12] with Pattern[6], matching. Increment i and j.
Thus we have reached on the last character of pattern, at the same time i is positioned at location 12 in the text array. The last character of pattern is also matching with Text[12], hence we can declare that a match of pattern is found in the text array at i – length of pattern + 1.
i.e. 12-7+1
i.e 6
Hence



Thus required pattern matches at location 6 in text array using KMP algorithm.

Algorithm KMP_Match(char t[50], char p[10])
j <- 0;
n = strlen(t);
m = strlen(p);

Prefix_table = create+prefix_table(p);
for (i = 0 to i < n) do
{

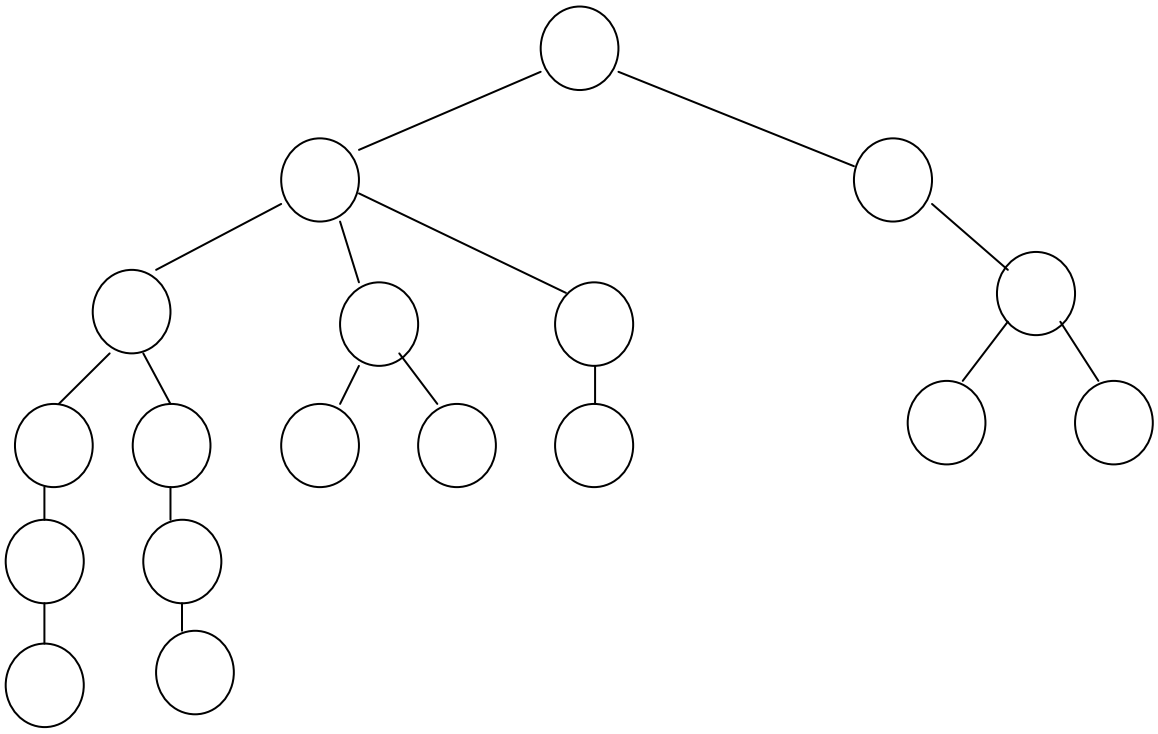
```
while (j > 0 AND p[j] != t[i]) do
j <- prefix_table[j-1];
if(p[j] = = t[i]) then
j++;
if (j==m) then
{
write “pattern present in the text at”;
write(i-m+1);
j <- prefix_table[j-1];
}
}
```

Analysis:

The time complexity of the above is O(n+m). The n represents the length of text and m represents the length of pattern.

8.2 TRIES

A trie derived from retrieval is a multiway tree data structure used for storing strings over an alphabet. It is used to store a large amount of strings. The pattern matching can be done efficiently using tries.



The above trie shows words like allot, alone, ant, and, are, bat, bad. The idea is that all strings sharing common prefix should come from a common node. The tries are used in spell checking programs.

Advantages of tries:

1. In tries the keys are searched using common prefixes. Hence it is faster. The lookup of keys depends upon the height in case of binary search tree.
2. Tries take less space when they contain large number of short strings. As nodes are shared between the keys.
3. Tries help with longest prefix matching, when we want to find the key.

Comparison of tries with hash table:

1. Looking up data in a trie is faster in worst case as compared to imperfect hash table.
2. There are no collisions of different keys in a trie.
3. In trie if single key is associated with more than one value then it resembles in hash table.
4. There is no hash function in trie.
5. Some times data retrieval from tries is very much slower than hashing.
6. Representation of keys a string is complex. For eg, representing floating point numbers using strings is really complicated in tries.
7. Tries always take more space than hash tables.
8. Tries are not available in programming tool it. Hence implementation of trie has to be done from scratch.

Applications of tries:

1. Tries has an ability to insert, delete or search for the entries. Hence they are used in building dictionaries such as entries for telephone numbers, English words etc..
2. Tries are also used in spell checking softwares.

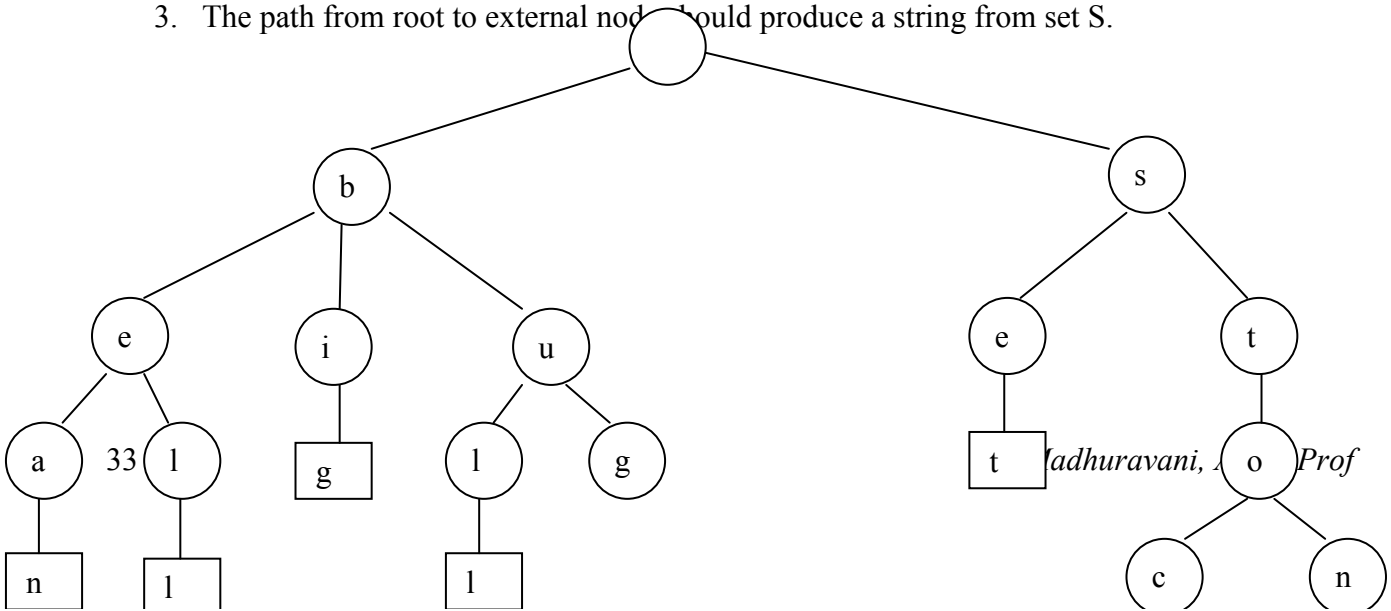
DIFFERENT TYPES OF TRIES:

1. Standard tries.
2. Compressed tries
3. Suffix tries

STANDARD TRIE

The standard trie is an ordered tree for building the strings of set S such that

1. Each node is labeled with an alphabet except root node.
2. The children of a node are alphabetically arranged.
3. The path from root to external node should produce a string from set S.



S = {bean, bell, big, bull, set, stock, stone}

Space complexity – O(n)

Disadvantages:

1. Use more number of nodes than other tries(space complexity increases).
2. Not suitable for matching non-word patters, such as string with white spaces.
3. Can compare a single character at a time.

COMPRESSED TRIE

Compressed trie or Patricia tries are similar to the standard tries and are like the tree structure, but shape of the node depends on the data it holds.

A node in the standard trie will have only one character, but compressed trie can have multiple characters.

Compressed trie will have at least two children at each of its internal nodes.

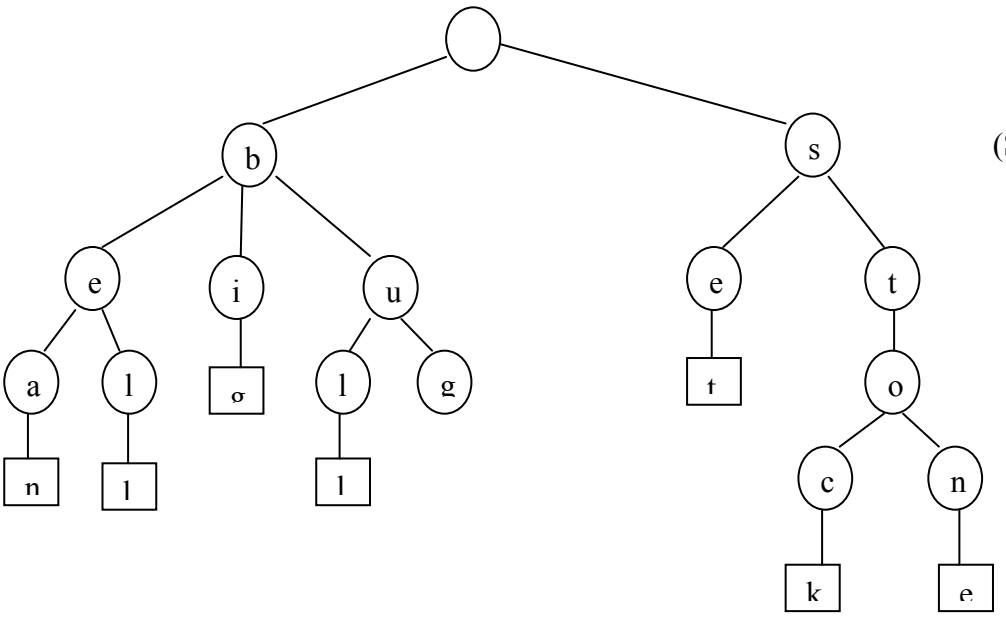
Properties:

Let S be a set of strings with an alphabet set of size d,

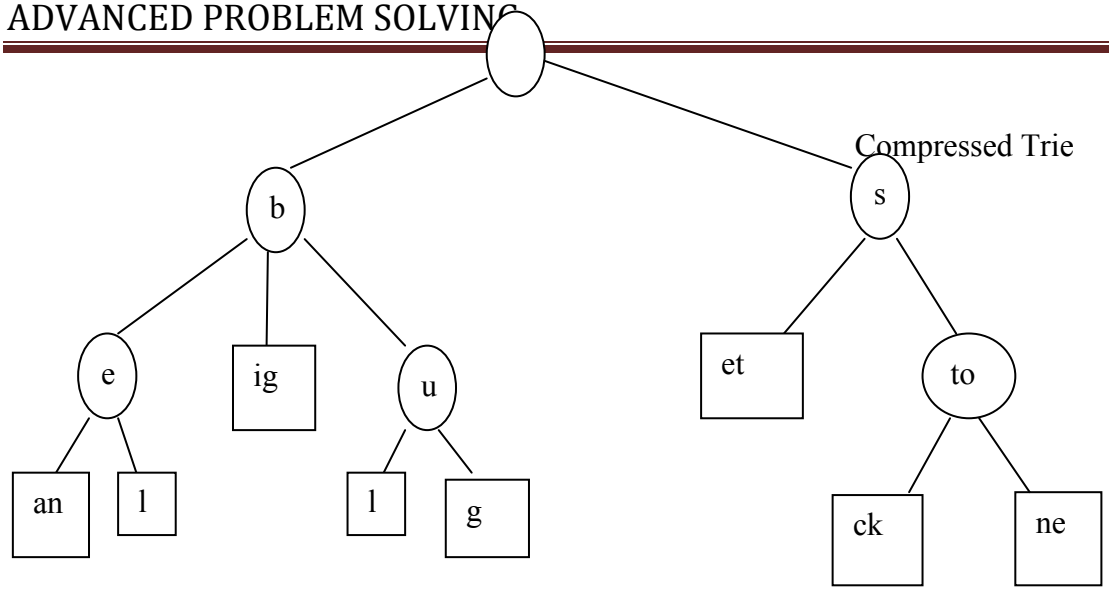
1. All the internal nodes in the compressed trie should have at least two children and at most d children.
2. If all the strings of set S do not have the same start suffix character then the compressed trie should have s number of external nodes where s is the number of strings in S.
3. The number of nodes of T is O(s).

Disadvantages:

1. Inefficient when too many duplicates are present in the patterns.
2. Each node occupies at least one byte.
3. Updating the node is difficult.



(Standard Trie)



Space complexity – O(n)

SUFFIX TRIE

Suffix trie is a tree which has the suffix symbols of each string from a branch. Suffix trie can be called a position tree or a suffix tree.

This procedure is better suited for finding the largest repetitive substring from the given large pattern string. This procedure completely avoids the unnecessary comparisons.

Example:

Let the pattern P = "banana"

Step1: Find all possible substrings

T1 = bananas
T2 = ananas
T3 = nanas
T4 = anas
T5 = nas
T6 = as
T7 = s

Step2: Sort all the substrings alphabetically

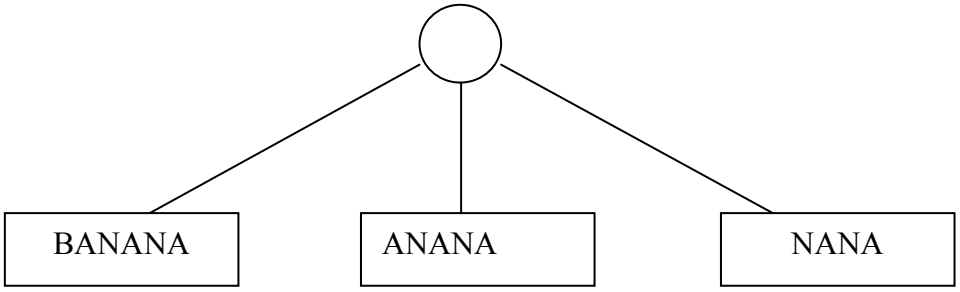
T6 = a
T4 = ana
T2 = anana
T1 = banana
T5 = na
T3 = nana

Step3: Find the prefixes shared by substrings.

Here the prefix of T2 is shared by T6 and T4. T1 is the only node that starts with the prefix, and finally the prefix of T3 is shared by T5.

So, the found major possible strings are T2, T1 and T3.

Step4: Construct the tree.



APPLICATION OF TRIES

1. Tries in the web search engines
2. Symbol table maintenance
3. Domain name servers
4. Indexing a book
5. Document similarity check
6. Tries in the Internet routers.