*Disjoint sets:

# UNIT 2

→ If the sets doesn't contain common elements then such types of sets known as disjoint sets.

Ex: when $n=10$, the elements can be partitioned into 3 disjoint sets.

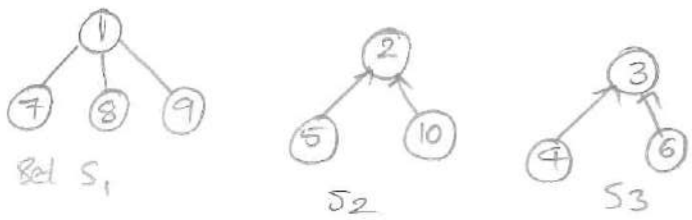
$S_1 = \{1, 7, 8, 9\}$

$S_2 = \{2, 5, 10\}$

$S_3 = \{3, 4, 6\}$

→ This sets can be represented in tree format.

to




These are the possible tree representation of sets is:

* Disjoint sets operations:

→ The following operations can be performed on disjoint set.

1. Disjoint set union
2. Find (i)

1. Disjoint set union:

To obtain the union of two sets, set the root node of one tree as child node of another tree.

*Disjoint sets:

→ If the sets doesn't contain common elements then such type of sets known as disjoint sets.

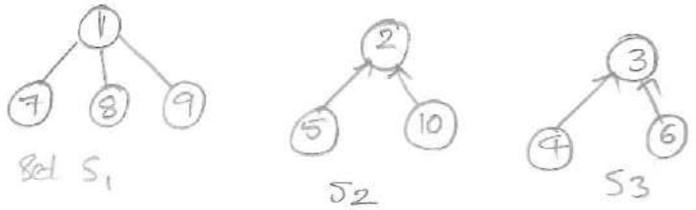
Ex: when $n = 10$, the elements can be partitioned into 3 disjoint sets.

$S_1 = \{1, 7, 8, 9\}$

$S_2 = \{2, 5, 10\}$

$S_3 = \{3, 4, 6\}$

→ This sets can be represented in tree format.




These are the possible tree representation of sets is:

* Disjoint sets operations:

→ The following operations can be performed on disjoint set.

1. Disjoint set union
2. Find (i)

1. Disjoint set union:

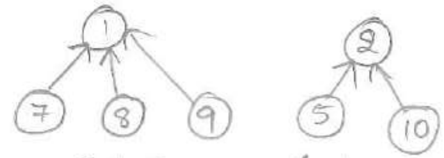
To obtain the union of two sets, set the root node of one tree as child node of another tree.

(or)

If $S_i$ and $S_j$ are two disjoint sets then their union $S_i \cup S_j$ = all elements '$x$' such that '$x$' is in either $S_i$ or $S_j$.

Ex: $S_1 = \{1, 7, 8, 9\}$

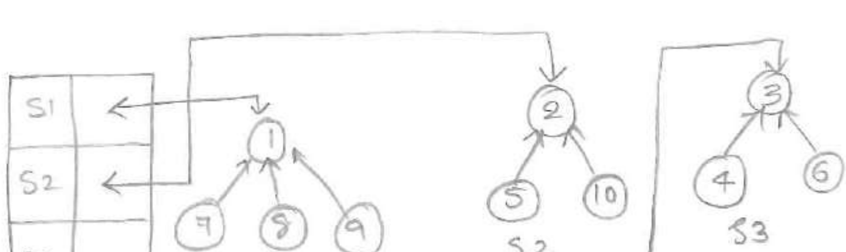
$S_2 = \{2, 5, 10\}$



Set $S_1$    Set $S_2$

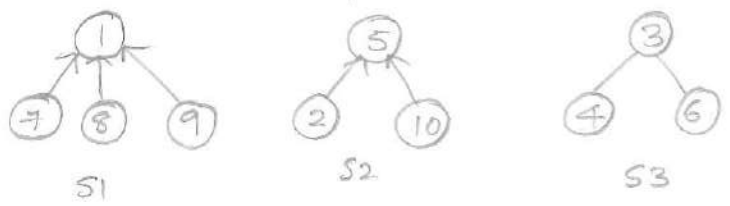
Ans:

→ Data representation for $S_1, S_2$ and $S_3$




→ Each root node has a pointer to the set name. To determine which set an element is currently in, we follow parent links to the root node of its tree.

* Array representation of $S_1$, $S_2$ and $S_3$ sets:




| i | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
|---|---|---|---|---|---|---|---|---|---|---|
| P | -1 | 5 | -1 | 3 | -1 | 3 | 1 | 1 | 1 | 5 |

→ We represent the tree nodes using an array P[1.....n] where n is maximum number of elements. The ith element of this array represents the tree node that contains element 'i'. This array element gives ~~represents~~ the ~~tree node that~~ parent pointer of the corresponding tree node.

→ Notice that root nodes ~~are~~ have a parent of '-1'.

Ex: $i = 2$

$P[i] = P[2] = 5.$

* Simple union algorithm:

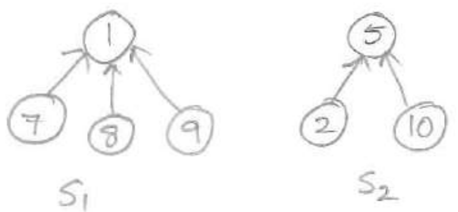
→ In this algorithm, root node of one tree becomes child node for another tree

Algorithm Simpleunion (i, j)

```
{
 P[i]:=j;
}
```

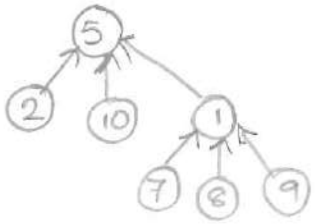
where i, j are root nodes of different trees.

Ex:




Here $i=1$, $j=5$.

when we perform Simple union (1, 5) on the trees $S_1$ and $S_2$. Root node of $S_1$ tree becomes child node for root node of $S_2$ tree.




| i | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
|---|---|---|---|---|---|---|---|---|---|---|
| P | 5 | 5 | | | -1 | | 5 | 5 | 5 | 5 |

→ Now let us process the following sequence of union operations union (1,2),

union (2,3), union (3,4)...... union(n-1,n). This sequence of union operations produces the following tree called as degenerate tree


→ The n-1 union operations can be processed in time O(n)

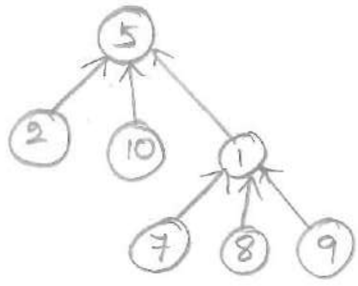
* Find operation:

Find (i) determines the root node of the tree containing the element 'i'

Simple find algorithm:

```
Algorithm SimpleFind(i)
{
  while (P[i] ≥ 0) do
     i := P[i];
  return i;
}
```

Ex:




Find(7) = 5

Find (10) = 5

Find(5) = -1

→ The total amount of time required to process the 'n' finds is $O(n^2)$. For 'i find' operation at level i time complexity is $O(i)$.

*Note:

→ we can improve the performance of union and find algorithm by avoiding the creation of degenerate trees.

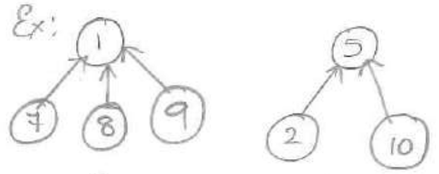
→ To perform this we make use of weighting rule for union (i,j)

*Weighting rule for union (i,j):

→ If the no. of the nodes in the tree with root 'i' is less than the no. of nodes in the tree with root 'j' then make 'j' the parent of 'i' otherwise make 'i' the parent of 'j'.

```
Algorithm weighted union (i,j)
// P[i]:=-count[i] and P[j]=-count[j]
{
temp:= P[i] + P[j];
 if (P[i] > P[j]) then
  {
  P[i]:= j;
  P[j]:= temp;
  }
 else
  {
  P[j]:= i;
  P[i]:= temp;
  }
}
```

Ex:



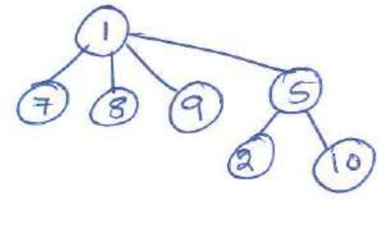
$S_1$ $S_2$

i = 1 j = 5

P[i] = -count [i] , P[j] = - count [j]

↳ -count [i] means number of nodes which returns the tree 'i'

P[i] = -4 , P[j] = -3



→ The number of nodes with root 'i' is 4

→ The no. of nodes in tree with root 'j' is 3. So make 'i' as a parent of 'j'.

* Collapsing rule:

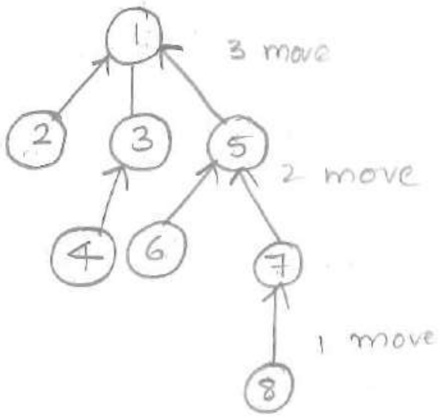
→ If 'j' is a node on the path from 'i' to its root and P[i] ≠ root [i] then set. P[j] to root[i].

* description:

→ Process the following 8 find operations.

Find(8), find(8), ... 8 times.

→ If simplefind algorithm is used each find(8) requires 3 moves in the following tree.




find(8) = 1.

→ 24 moves required to process all 8 find operations

→ when collapsing find algorithm is used the first find(8) requires going up 3 links and then resetting 3 links.

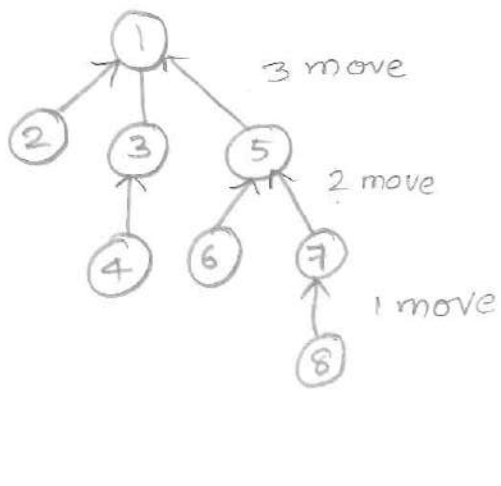
→ Each of the remaining 7 find operations requires going up only 1 link field.

→ The total cost is 6+1+1+1+1+1+1+1 = 13

Algorithm:

```
Algorithm collapsingfind (i)
{
  r:=i;
  while (P[r] >0) do
    r= P[r];
  while (i ≠ r)
  {
    s:= P[i];
    P[i]:= r;
     i:= s;
  }
  return r;
}
```

Ex:



Before execution

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| P | -1 | 1 | 1 | 3 | 1 | 5 | 5 | 7 |

After execution

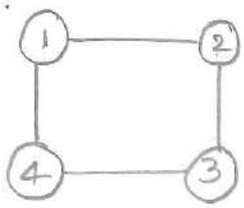
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| P | -1 | 1 | 1 | 3 | 1 | 5 | 1 | 1 |

total cost = 6 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 13

* Spanning tree:

Spanning tree is subgraph of given graph $G = \{V, E\}$ and contains all vertices of graph 'G' and no cycles

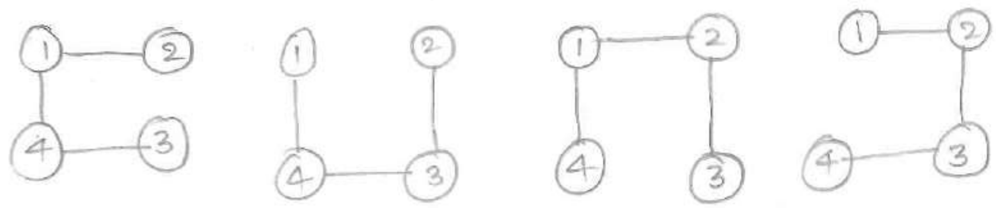
Ex: 1.




$G = \{V, E\}$

$V = \{1, 2, 3, 4\}$

$E = \{(1,2), (2,3), (3,4), (4,1)\}$



# BACK TRACKING

→ Many problems which deals with searching for a set of solutions are which ask for an optimal solution satisfying some constraints can be solved using back-tracking method.

→ Dynamic programming design method uses ~~root~~ brute force approach to find the optimal solution. Here bruite force approach will use ~~always~~ all possible ways to find an optimal solution.

→ we can use backtracking method to find the optimal solution with less no.of steps. In this method, if we are trying to find the Soln in one path and unable to get the solution then stop there, come back to previous step and choose alternate path to find the optimal solution.

→ many of the problems we solve using back-tracking method require that all the solutions satisfies a complex set of constraints.

→ For any problem these constraints can be divided into two catogories.

1. Explicit constraint
2. Implicit constraint.

1. Explicit constraint:

→ Explicit constraints are the rules that restricts each '$x_i$' to take on values only from a given set.

→ Common examples of Explicit constraints are $x_i \geq 0$, $x_i = 0$ or $1$

2. Implicit constraint:

→ Implicit constraints are the rules that determine which of the tuples in the solution space of 'I' satisfies the criterian function.

* Applications of backtracking:

1. n-queens problem
2. Sum of Subsets
3. Graph colouring
4. Hamiltonean cycle.

1. n-queens problems:

→ In this problem, n-queens are to be placed on an $n \times n$ chess board in such a way that no two queens are placed on same row, same column and same diagonal.

Ex: 4-queens problem

8-queens problem

*4-queens problem:

→ In this problem, 4-queens are to be placed on $4 \times 4$ chess board so that no two queens are placed on same row, column and diagonal.

Explicit constraint:

→ Let us consider ~~us~~ the number the rows and columns of chess board as 1, 2, 3, 4. Queens can also be numbered 1, 2, 3, 4. So the values of set $S_i = \{1, 2, 3, 4\}$. Therefore solution space contains '$4^4$' ways.

Implicit constraint:

→ No two queens are placed on same row, same column and same diagonal.

Step-1: Place queen 1 ($q_1$) on first row, first column.

| | | | |
|---|---|---|---|
| $q_1$ | | | |
| | | | |
| | | | |
| | | | |

Step-2: Queen 2 ($q_2$) cannot be placed on second row first column and also second column. So place on third column.

| | | | |
|---|---|---|---|
| $q_1$ | | | |
| • | • | $q_2$ | |
| | | | |
| | | | |

Step-3: Queen 3($q_3$) cannot be placed on any one of the columns

| | | | |
|---|---|---|---|
| $q_1$ | | | |
| | | $q_2$ | |
| • | • | • | • |
| | | | |

Backtrack to the previous step and place $q_2$ in another possible column i.e 4th column.

and place $q_3$ in $2^{nd}$ column

| | | | |
|---|---|---|---|
| $q_1$ | | | |
| | | | $q_2$ |
| | $q_3$ | | |
| | | | |

Step-4: Queen 4 ($q_4$) cannot be placed on any of the columns. So backtrack to previous step $q_3$. All possible ways are completed for $q_3$ so backtrack to $q_2$. All possible ways are completed for $q_2$ also so backtrack to $q_1$. and change the position of $q_1$ from $1^{st}$ column to second column.

| | | | |
|---|---|---|---|
| $q_1$ | | | |
| | | | $q_2$ |
| | $q_3$ | | |
| • | •• | • | • |

| | | | |
|---|---|---|---|
| | $q_1$ | | |
| | | | |
| | | | |
| | | | |

Step-5: Place $q_2$ on fourth column

| | | | |
|---|---|---|---|
| | $q_1$ | | |
| | | | $q_2$ |
| | | | |
| | | | |

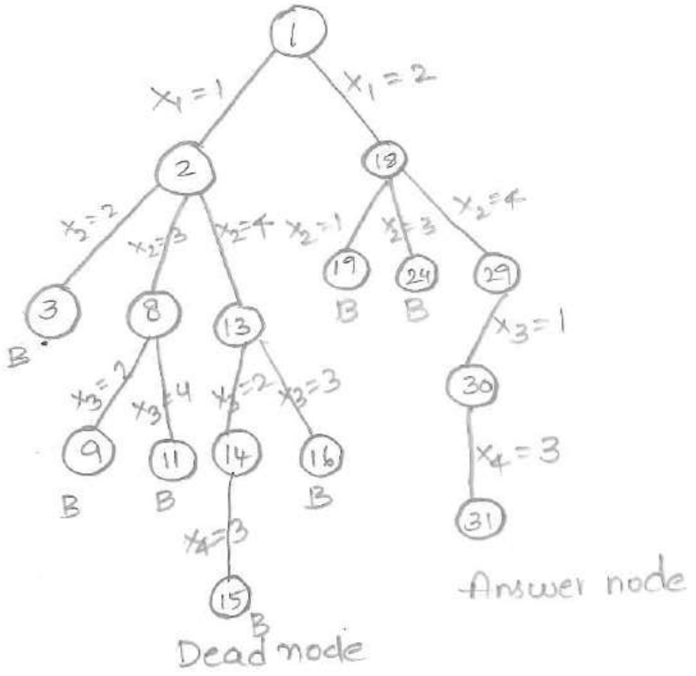
Step-6: Place $q_3$ on $1^{st}$ column & $q_4$ on $3^{rd}$ column.

| | | | |
|---|---|---|---|
| | $q_1$ | | |
| | | | $q_2$ |
| $q_3$ | | | |
| | | $q_4$ | |

Optimal solution:

$\{x_1, x_2, x_3, x_4\} = \{2, 4, 1, 3\}$

State Space tree for 4-queens problem:




* Problem state:

Each node in state space tree defines a problem state

* State space:

All the paths from root node to other nodes define the state space of the problem.

* Solution state:

→ Solution state are problem states for which the path from the root to the other nodes defines a tuples in a solution space.

* Answer State:

→ Answer states are the solution states for which the path from root to other nodes defines a tuples that is a member of the set of the solutions of the problem.

* State Space tree:

→ A tree organisation of solution space is represented as state space tree.

* Live node:

→ A node which has been generated and whose children have not yet been generated is called a live node

* E-node:

→ The live node whose children are currently been generated is called E-node.

* Dead node:

→ A dead node is a generated node which is not to be expanded further

* Bounding function:

→ These are used to kill live nodes without generating all their children.

## * 8-queen's problem:

8-queen's prob is an objective function or criterian function. In this problem 8 queens are to be placed on 8x8 chess board so that no 2 queens placed on same row, same column and same diagonal.

### Explicit constraint:

8x8 chess board contains 8 rows & 8 columns. The queens can also be numbered 1 to 8. So $S_i = \{1, 2, \ldots 8\}$ Therefore the solution space contains $8^8$ tuples.

### Implicit constraints:

No two queens are placed on same row, same column, same diagonal. One of solution's of 8 queen's problem.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | | | | | |
| | | | | | 2 | | |
| | 3 | | | | | | |
| | | | | | | 4 | |
| 5 | | | | | | | |
| | | | 6 | | | | |
| | | | | | | | 7 |
| | | | | 8 | | | |

$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (3, 6, 2, 7, 1, 4, 8, 5)$

Algorithm:

```
Algorithm place(k, i)
{
for j=1 to k-1 do
 if ((x[j]= i) or (Abs(x[j]-i)= Abs(j-k)))
 then return false;
return true
 }
```

```
Algorithm NQueen(k, n)
{
for i:=1 to n do
{
if place(k, i) then
{
 x[k]= i;
 if (k==n) then write(x[1:n]);
 else NQueens(k+1, n);
 }
 }
}
```

* Sum of subsets problem:

→ Suppose we are given n-distinct positive numbers and we need to find all combinations of this numbers whose sums are 'm'. This is called the sum of subsets problem.

Bounding function:

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Example: $n=4$ $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$

$m = 31$

Find all combination of the number whose sum is m.

Sol:-

| Subset | Sum | Remarks |
|---|---|---|
| { } | 0 | |
| {11} | 11 | |
| {11, 13} | 11+13=24 | |
| {11, 13, 24} | 11+13+24=48 | Exceeds m value backtrack |
| {11, 13} | - | - |
| {11, 13, 7} | 11+13+7=31 | optimal soln |

→ The optimal solution

$(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

$11 + 13 + 7 = 31$ [whose sum = m]

→ One more optimal solution is

$(x_1, x_2, x_3, x_4) = (0, 0, 1, 1)$

$24 + 7 = 31$ [whose sum = m]

* Recursive backtracking algorithm for [illegible]

problem:

```
Algorithm sumof sub (s, i, sum)
{
x[i] := 1;
if (s+w[i]= m) then write (x[1:n])
else if (s+w[i] + w [i+1] ≤ m)
then sumofsub (s+w[i], i+1, sum-w[i]);
if((s+sum +w[i] ≥ m) and (s+ w [i+1] ≤ m)) then
{
x[i] := 0;
sumofsub (s, i+1, sum - w[i]);
}
}
```

→ This algorithm

where s is values of subsets

i is position of values or elements

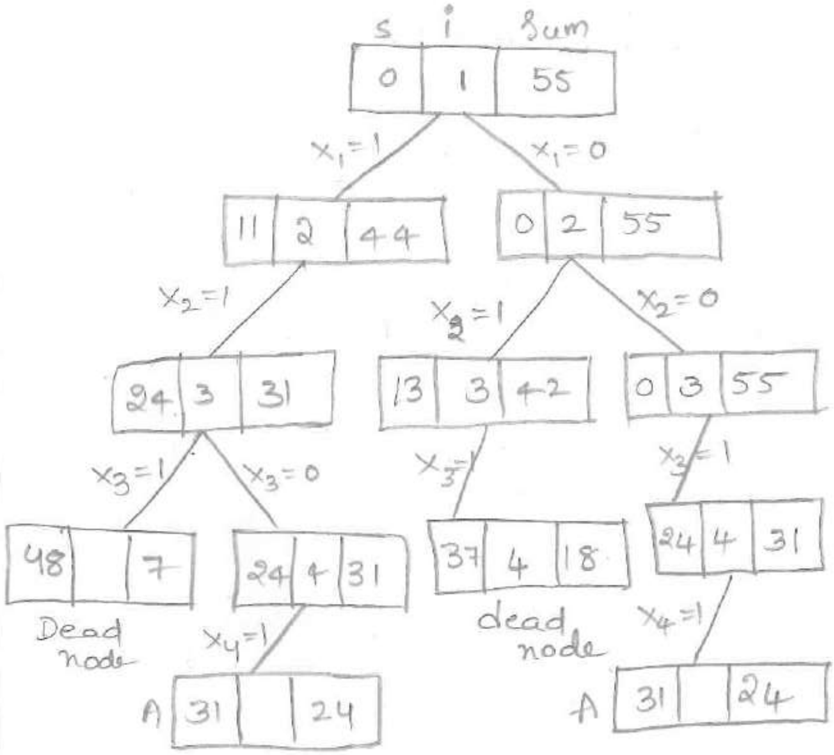
sum is sum of all elements of the given set

<u>for example</u>: 1) $n = 4$

$$(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$$

$$m = 31$$

Draw the state space tree using recursive backtracking algorithm.




we got 2 optimal solutions subset $(x_1, x_2, x_3, x_4)$

$= (1, 1, 0, 1)$

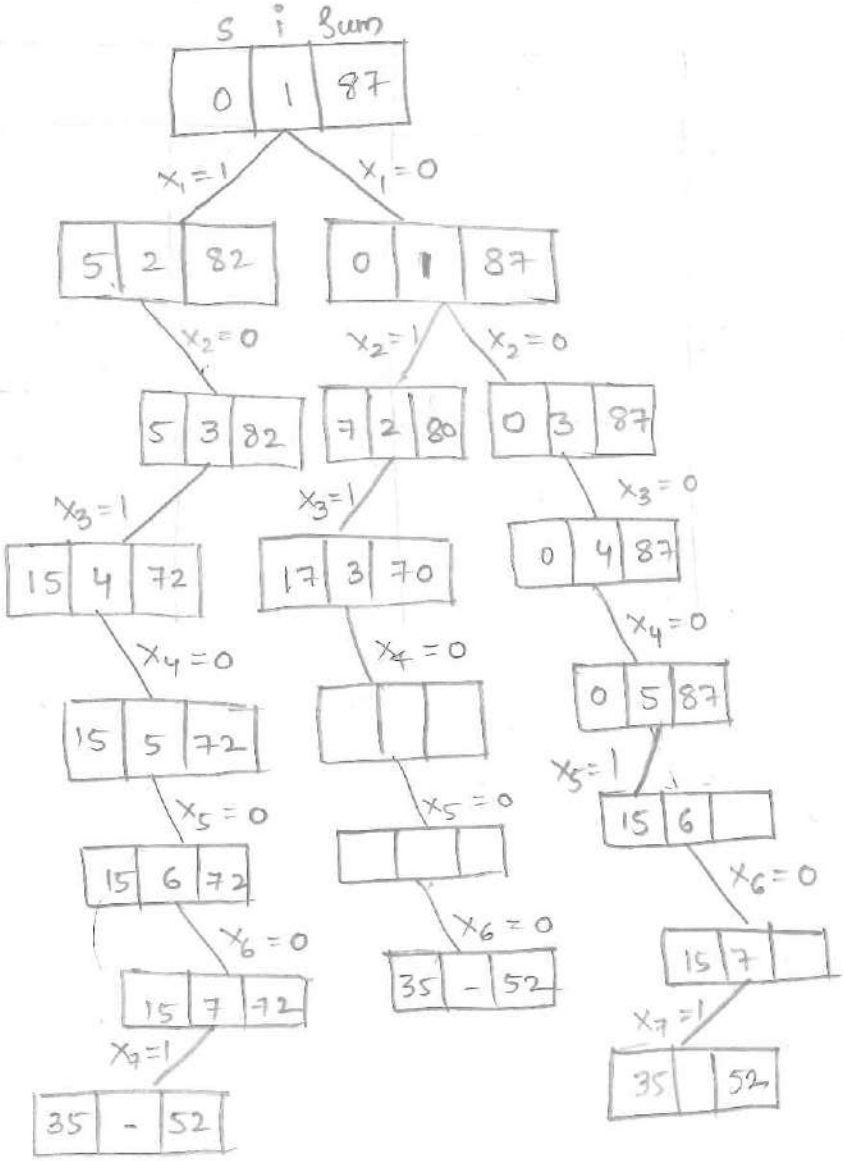
$11 + 13 + 7 = 31$ [whose sum = m]

subset 2 = $(x_1, x_2, x_3, x_4) = (0, 0, 1, 1)$

$24 + 7 = 31$

2) Let $w = (5, 7, 10, 12, 15, 18, 20)$ and $m = 35$ find all possible subsets of $w$ that sum is $m$. Draw the portion of state space tree

Sol:-



* Graph coloring:

→ The process of coloring all vertices of a given graph in such a way that no two adjacent vertices have same colour yet only 'm' colours are used.

→ Note that if 'd' is the degree of the given graph, then it can be coloured with 'd+1' colours.

→ The m-colarability optimization problem asks for the smallest integer 'm' for which the graph 'G' can be coloured. This integer is referred to as the chromatic number of the graph.

→ A graph is said to be planar graph iff it can be drawn in a plane in such a way that no two edges cross each other.

Ex:1) Draw a state space tree for m colouring $n=3$, $m=3$. $n$ = no. of vertices
$m$ = no. of colours.

Sol:- $n=3$
$m=3$

raph

the

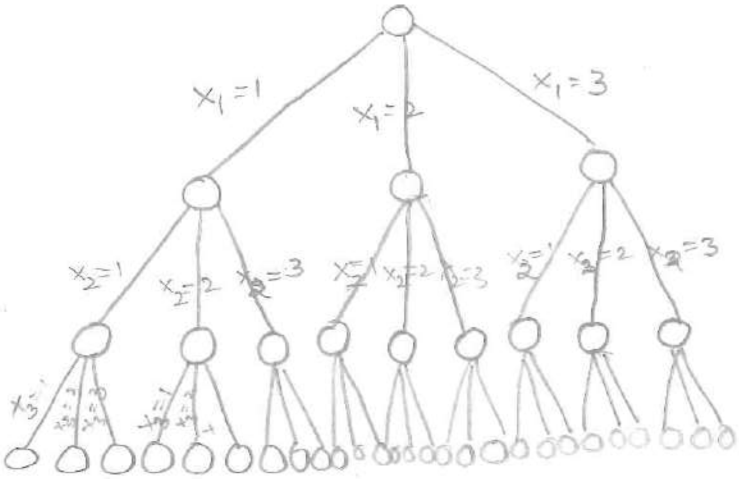
be

matic

can

two

n=3,




ces.

Algorithm:

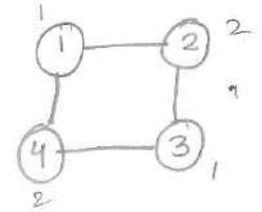
finding all m-colorings of graph

```
Algorithm mcoloring(k)
{
repeat
{
Nextvalue(k);
if(x[k]=0) then return;
if (k=n) then
write (x[1:n]);
else mcoloring(k+1);
}until(false);
}
```

```
Algorithm Nextvalue(k)
{
repeat
{
x[k]:= x[k]+1 mod(m+1);
if(x[k]=0) then return;
```

```
for j:=1 to n do
{
if((G[k,j]≠0) and (x[k] = x[j]))
then break;
}
if(j=n+1) then return
}until(false);
}
```

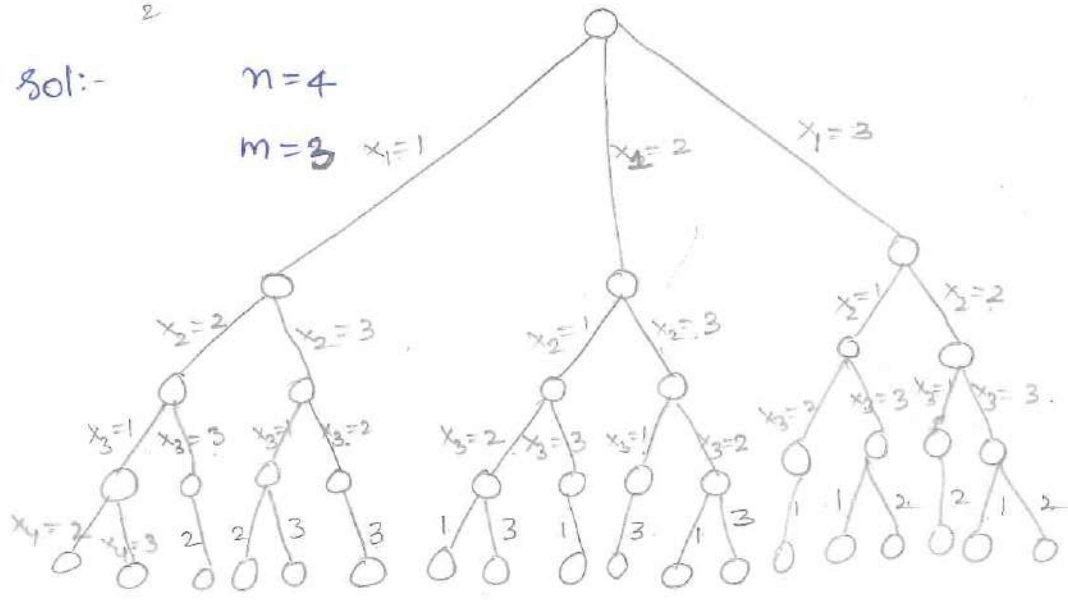
2) Color the given graph with m=3 colours




Draw the state space tree.

Sol:- $n=4$

$m=3$



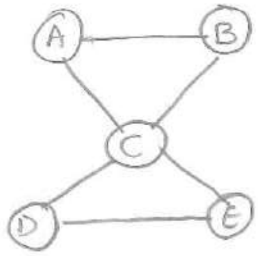
* Hamiltanion cycle:

→ Let $G = (V, E)$ be a connected graph with m-vertices.

* Hamiltonian path:

→ It is a path in a connected graph that visits each Vertex exactly once.

Ex:



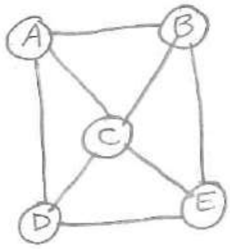

→ Hamiltanion path is ABCED (or) ABCDE.

→ Non-hamiltonion path is ABCEDC (or) ABCD.

* Hamiltanion cycle:

→ Hamiltanion cycle is a cycle in a connected graph that visits each Vertex exactly once and starting vertex, ending vertex must be same.

Ex:



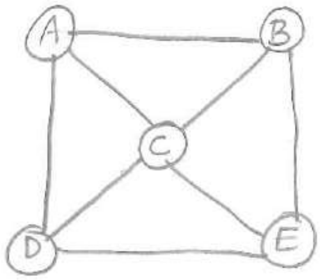

→ Hamiltanian cycle is ABCEDA (or) ABECDA

→Not Hamiltanian cycle is ABCEBA

* Hamiltanian graph:

→Hamiltonian graph is a graph that contains hamiltonian cycle.

Ex:

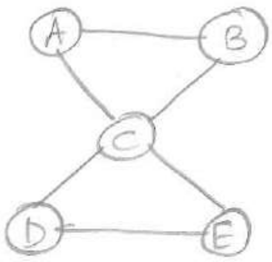



It consists of hamiltonian cycle. So, it is a hamiltonian graph.

*Traceable graph:

→Traceable graph is a graph that contains hamiltonian path is called traceable graph.

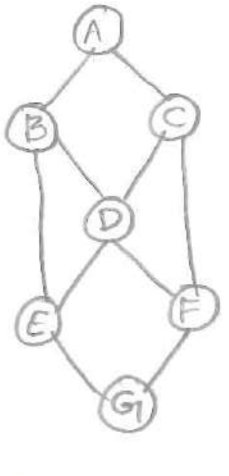
Ex:




It consists of hamiltonian ~~graph~~ path so it is a traceable graph.

Problem-1:

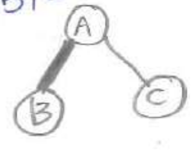
Check whether this graph consists of hamiltonian cycle or not?




Sol:- Step-1: Visit the source vertex A.

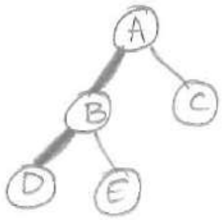

Step-2: Find all adjacent vertices of A and visit the nearest node

A – B, C




Step-3: Find adjacent vertices of B and visit the nearest node. 'D'

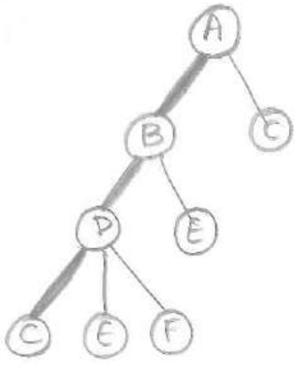
B – D, E.




Step-4: Find all adjacent vertices of D (i.e) D – B, C, E, F but B is already visited so visit next nearest

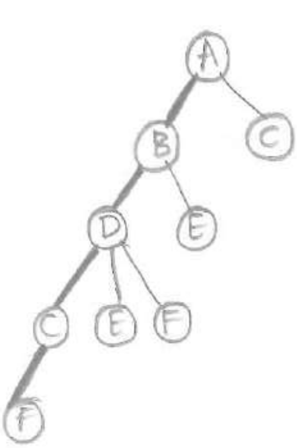
onian

Vertex C.



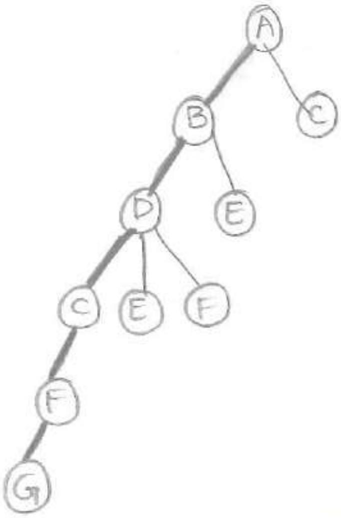


Step-5: Find all adjacent vertices of C i.e C - A, D, F. Since A and D are already visited visit F.



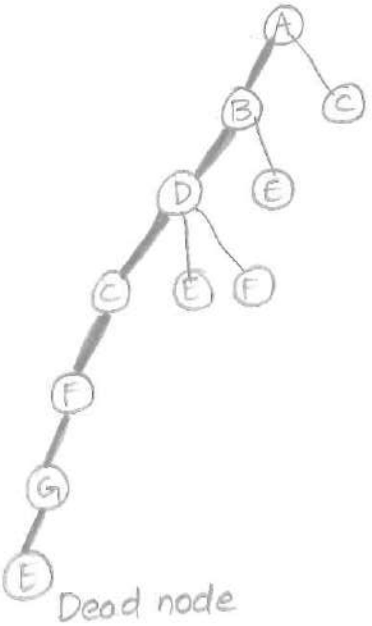


Step-6: Find all adjacent vertices of F i.e F - C, D, G. Since C and D are already visited visit 'G'.

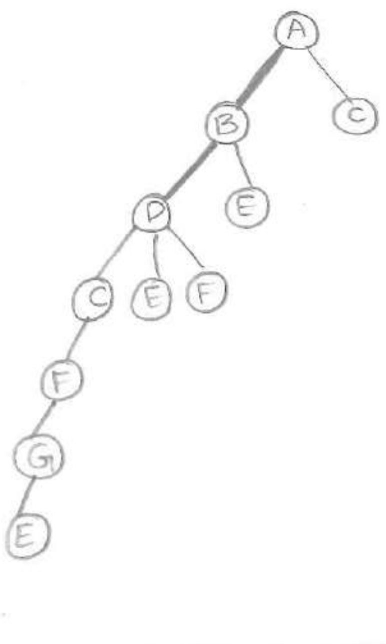




Step-7: Find all adjacent vertices of G (i.e) G - E, F. F is already visited. visit E.

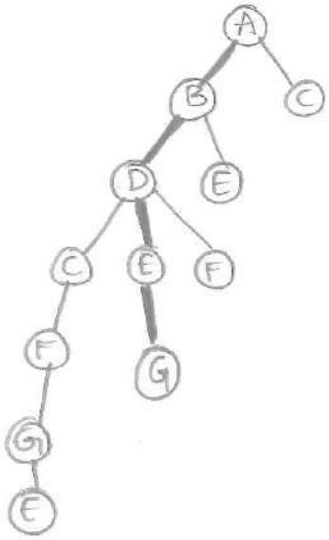



Step-8: Find all adjacent vertices of E i.e E – B, D, G. These vertices already visited and unable to generate child node. This node is known as dead node. Backtrack to vertex G there is no alternate path for G. So back track to F. no alternate path for vertex F also so backtrack to C. Since there is no alternate path for C backtrack to D. Now, choose alternate path and visit vertex E discard previous paths.

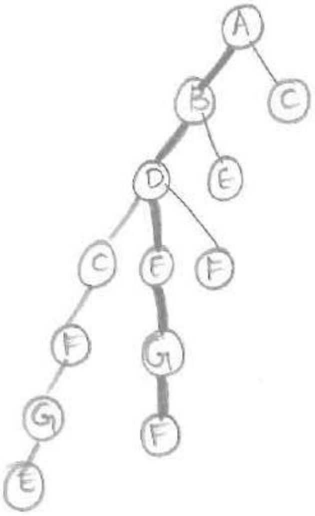


Step-9: Find all adjacent vertices of E (i.e) E - B, D, G

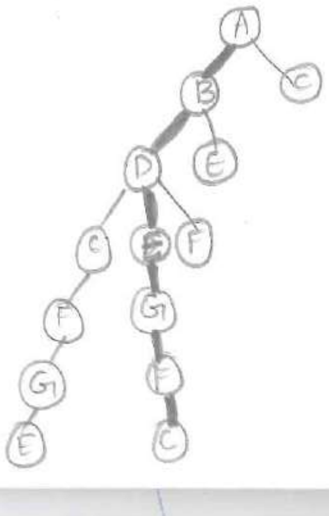
visit G.




Step-10: Find all adjacent vertices of G (i.e) E, F. E is already visited so visit F.

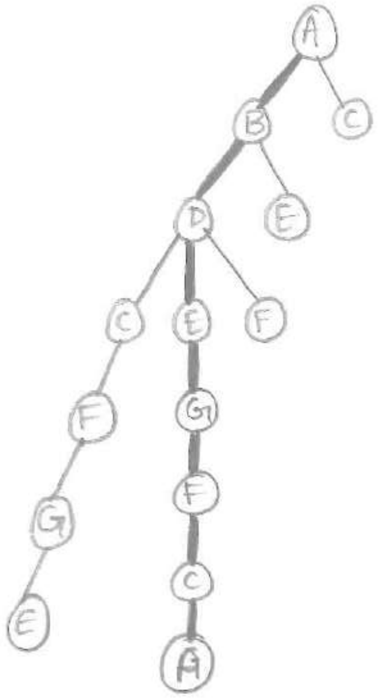



Step-11: Find all adjacent vertices of F - G, C then visit C.



Step-12: Similarly find all adjacent vertices of C – A, D, F

Since D, F are already visited visit A.




Hamiltonian cycle: A B D E G F C A.