1a **Back Tracking:**

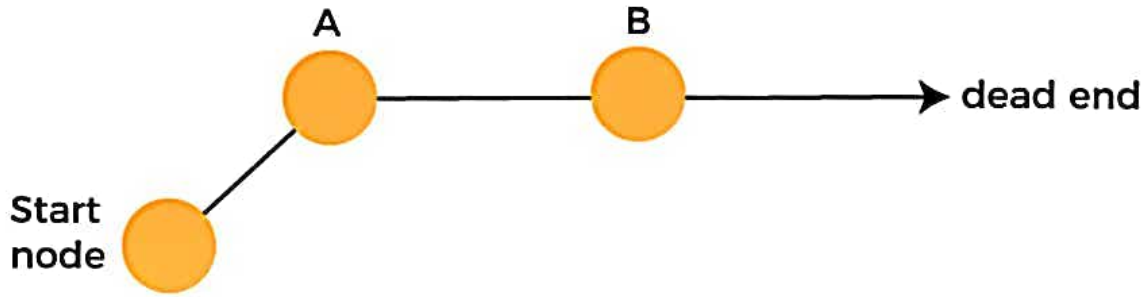
Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

- A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
- Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

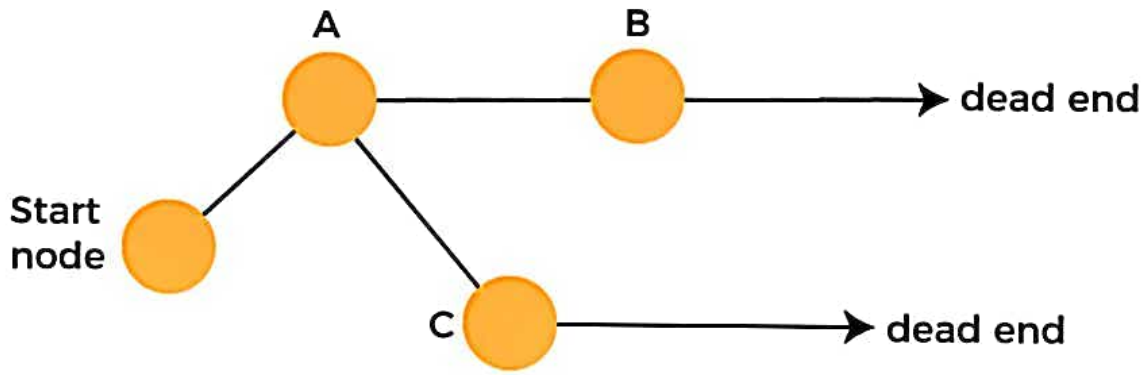
Backtracking is a systematic method of trying out various sequences of decisions until you find out that works.

Ex:

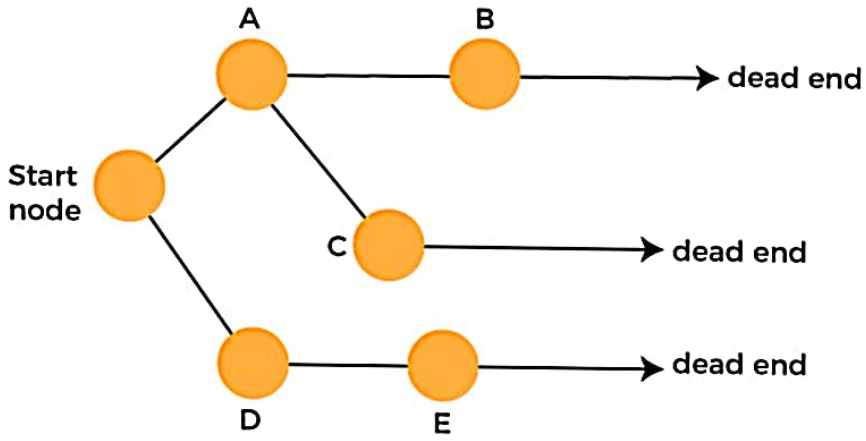
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



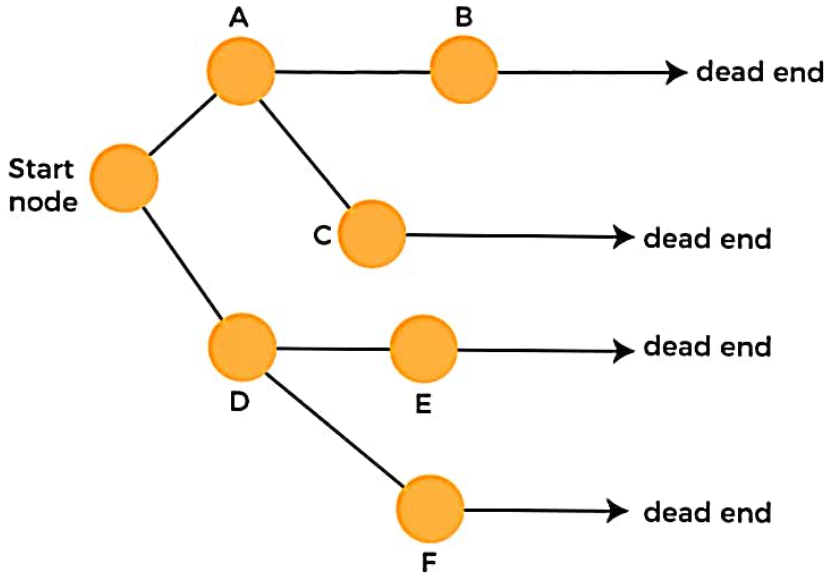
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



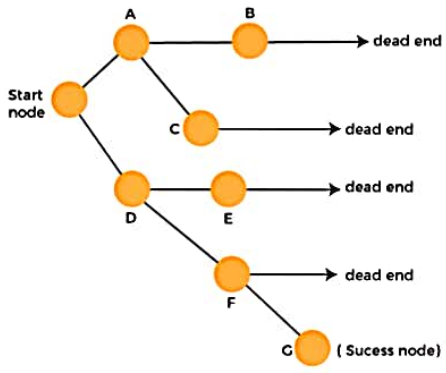
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

Terminology:

Problem state is each node in the depth-first search tree

State space is the set of all paths from root node to other nodes

Solution states are the problem states **s** for which the path from the root node to **s**

Answer states are that solution states s for which the path from root node to s defines a tuple that is a member of the set of solutions
State space tree is the tree organization of the solution space

Live node is a generated node for which all of the children have not been generated yet.

E-node is a live node whose children are currently being generated or explored

Dead node is a generated node that is not to be expanded any further

---

1b **Algorithm:**

```
Algorithm mColoring(k)
// the graph is represented by its Boolean adjacency matrix G[1:n,1:n]. All
//assignments of 1,2,.....,m to the vertices of the graph such that adjacent vertices
//are assigned distinct integers are printed. 'k' is the index of the next vertex to color.
{
     repeat
    {
    // generate all legal assignment for X[k].
         Nextvalue(k);   // Assign to X[k] a legal color.
         if (X[k]=0) then return;          // No new color possible.
         if (k=n) then    // Almost 'm' colors have been used to color the 'n' vertices
           write(x[1:n]);
         else mcoloring(k+1);
     }until(false);
}
```

Example:

Consider a simple graph G with the following vertices and edges:

Vertices: {A, B, C, D, E, F}

Edges: (A, B), (A, C), (B, D), (C, D), (D, E), (D, F), (E, F)

The question is, can we color the vertices of this graph with, let's say, 3 colors (m = 3), in such a way that no two adjacent vertices share the same color?

Solution:

1. Start with an empty color assignment for each vertex.

Vertex	Color
A	
B	
C	
D	
E	
F	

2. Begin by picking an arbitrary vertex, let's say A, and color it with color 1.

Vertex	Color
A	1
B	
C	
D	
E	
F	

3. Move to the next uncolored vertex (B) and color it with the lowest available color that's not used by its neighbors (A in this case). So, B gets colored with color 2.

Vertex	Color
A	1
B	2
C	
D	
E	
F	

4. Continue this process for the remaining uncolored vertices. For instance, you can color C with color 3, and then proceed to D, which has neighbors A and B with colors 1 and 2. Since all colors are taken, you'll need to use a new color (color 3) for D.

Vertex	Color
A	1
B	2
C	3
D	3
E	
F	

5. Continue this process until all vertices are colored.

Vertex	Color
A	1
B	2
C	3
D	3
E	1
F	2

2a

```
Algorithm RBackTrack (k)
// Global n :integer;
//          x :array[1..n] of items;
// On entering, the first k-1 values have been assigned

{
    for each x[k] ∈ T(x[1],...,x[k-1]) do
    {
        if (B_k(x[1],...,x[k]) = true) then
        {
            if (x[1],...,x[k]) is a path to an answer node then
                print (x[1],...,x[k]);
            if (k < n) then RBackTrack(k+1);
        }
    }
}
```

Recursive backtracking is a general algorithmic technique used to solve problems that involve making a series of choices. It is a depth-first search algorithm that explores all possible choices and can backtrack when it reaches a dead end. The general method for a recursive backtracking algorithm can be explained in the following steps:

1. **Define the Problem:** Begin by clearly defining the problem you want to solve. Identify the decision points, constraints, and goals of the problem. You should have a clear understanding of what constitutes a valid solution.
2. **Create a Recursive Function:** Implement a recursive function that will explore the possible choices and make progress toward a solution. This function should take as arguments the current state or configuration of the problem, as well as any other necessary parameters.
3. **Define Base Cases:** Every recursive function should have one or more base cases. Base cases define the conditions under which the recursion stops. These are typically points where you either find a solution or determine that there are no valid solutions from the current state. The base cases are crucial for the termination of the recursion.
4. **Explore Choices:** Within the recursive function, explore the available choices at the current state. For each choice, make a recursive call, effectively moving to the next step in the problem. These choices may involve selecting an option, making a move, or setting a variable.
5. **Backtrack:** If a choice leads to a dead end (e.g., it violates constraints or does not lead to a solution), you need to backtrack. Backtracking involves returning from the current function call, undoing the last choice, and continuing with other possible choices. This process continues until a valid solution is found or all possibilities are exhausted.
6. **Maintain State:** Throughout the recursion, maintain the state and data structures that represent the problem. Update the state as you make choices and revert it during backtracking. It's crucial to ensure that the state accurately reflects the current configuration of the problem.
7. **Repeat the Process:** Continue the process of exploring choices, making recursive calls, and backtracking until you've found a valid solution or have determined that no valid solutions exist.
8. **Return the Solution:** If a valid solution is found, you can return it or store it as needed. Depending on the problem, you may be looking for one solution, all possible solutions, or some other specific outcome.

backtracking algorithm is to structure your code to make

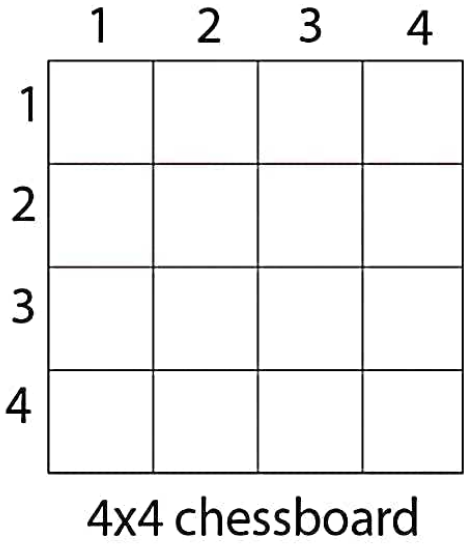
3a

n-Queens Problem:

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

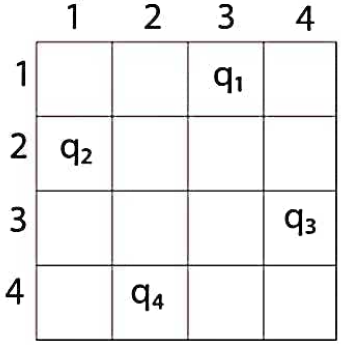


4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."
Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other.
We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely.
So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2).

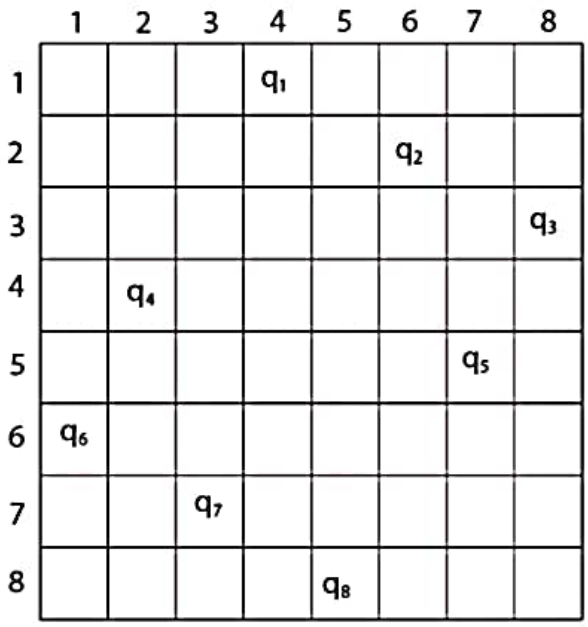
[redacted]

But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely.
Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3).
That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem.
For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:



1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

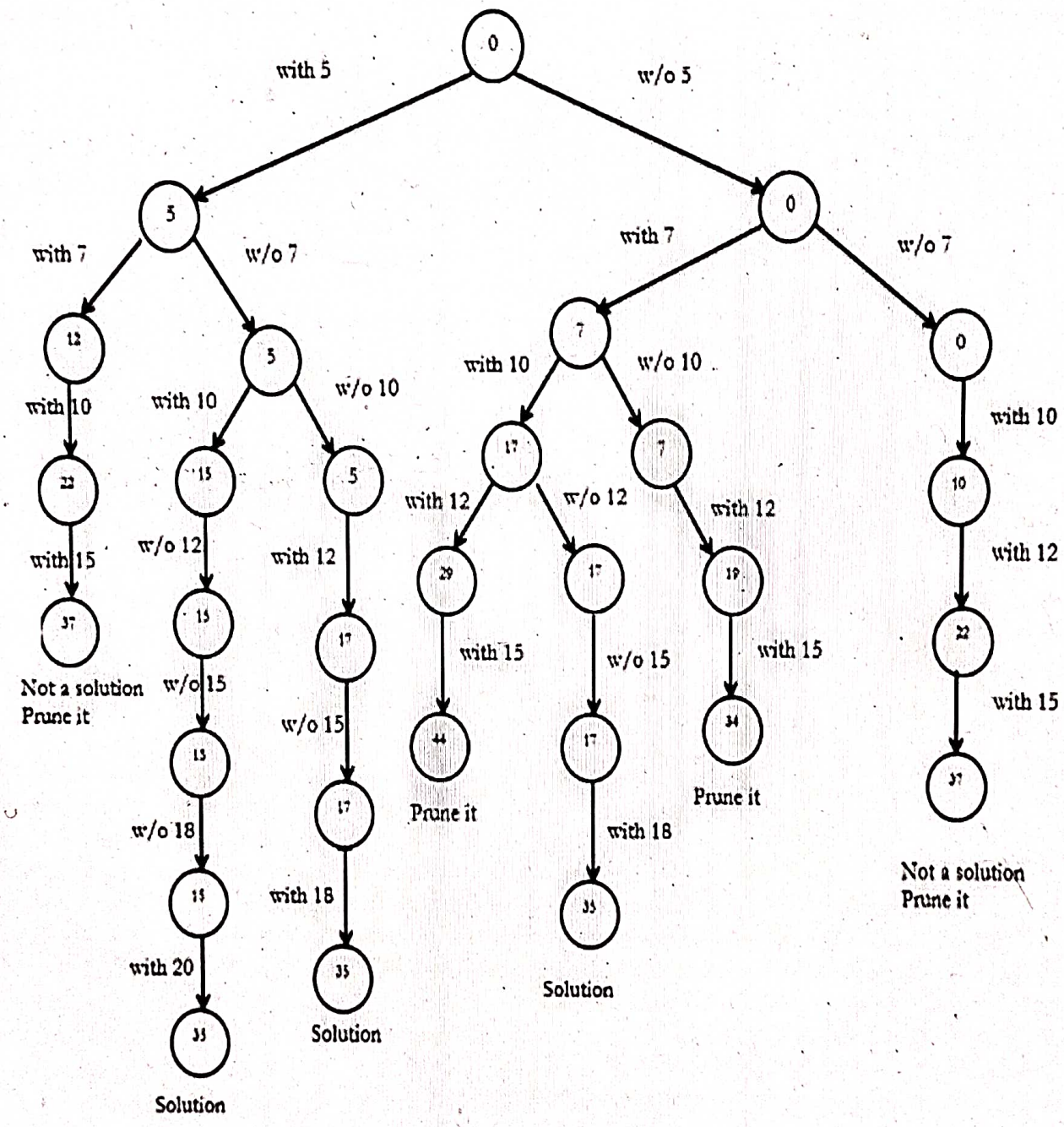
Example:

Solve following problem and draw portion of state space tree M = 35, W = (5, 7, 10, 12, 15, 18, 20)

Solution: 3b

Initially subset = {}	Sum = 0	Description
5	5	Then add next element.
5, 7	12,i.e. 12 < 35	Add next element.
5, 7, 10	22,i.e. 22 < 35	Add next element.
5, 7, 10, 12	34,i.e. 34 < 35	Add next element.
5, 7, 10, 12, 15	49	Sum exceeds M = 35. Hence backtrack.
5, 7, 10, 12, 18	52	Sum exceeds M = 35. Hence backtrack.
5, 7, 10, 12, 20	54	Sum exceeds M = 35. Hence backtrack.
5, 12, 15	32	Add next element.
5, 12, 15, 18	50	Not feasible. Therefore backtrack
5, 12, 18	35	Solution obtained as M = 35

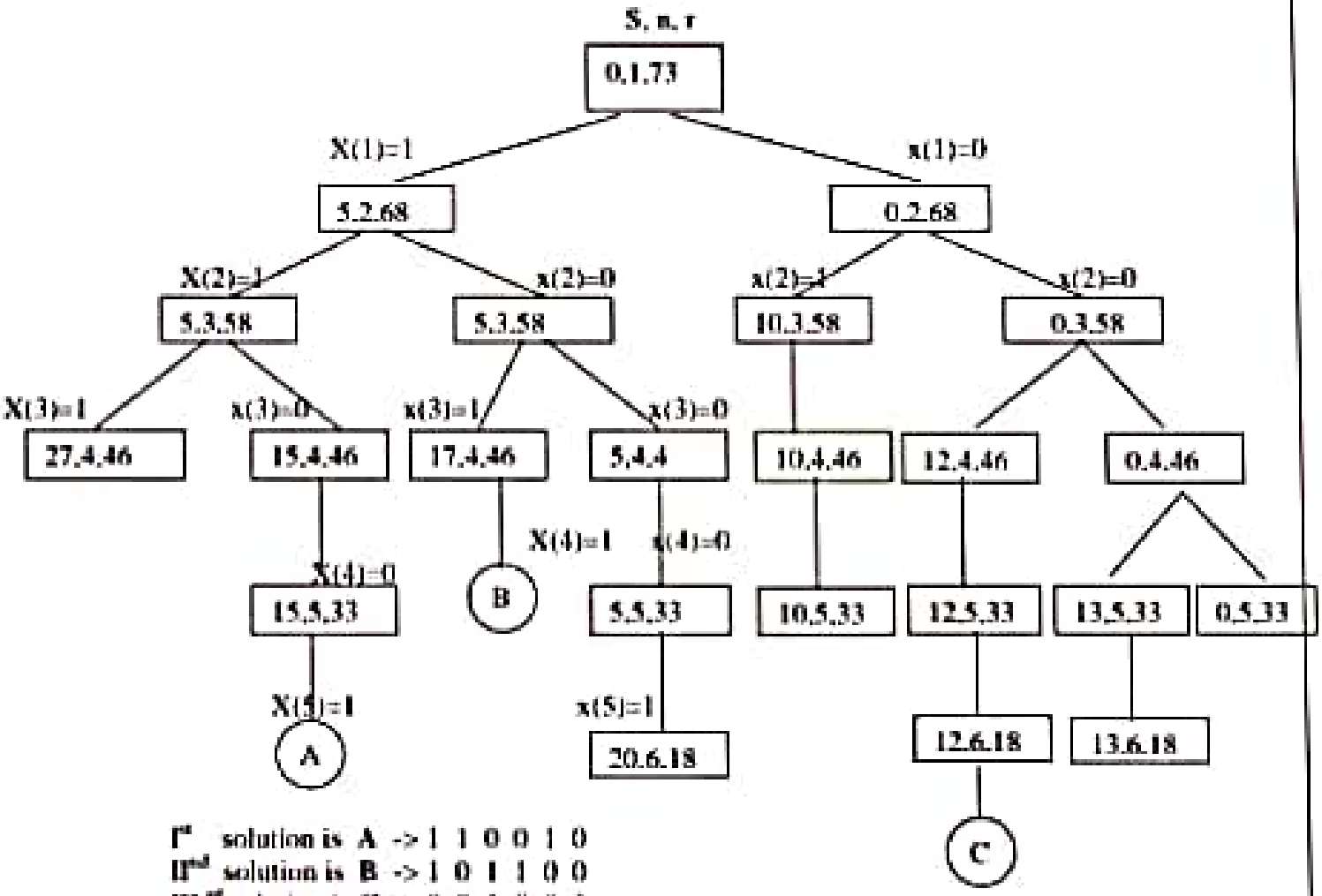
ıe state space tree is shown as below in figure 8. (5, 7, 10, 12, 15, 18, 20)



- In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets, 'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

  $s = \sum_{j=1}^{k-1} w_j x_j$ and $r = \sum_{j=k}^{n} w_j$

- Portion of the state space tree for the given problem is,



I^st solution is A -> 1 1 0 0 1 0
II^nd solution is B -> 1 0 1 1 0 0
III^rd solution is C -> 0 0 1 0 0 1

- In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeled with the values of Xi, which is either 0 or 1.
- The left sub tree of the root defines all subsets containing Wi.
- The right subtree of the root defines all subsets, which does not include Wi.
- The bounding function is $B_k(x_1, \ldots, x_k)$ = true iff

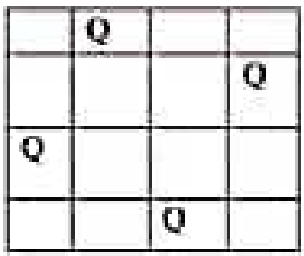
$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m \text{ and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

4a

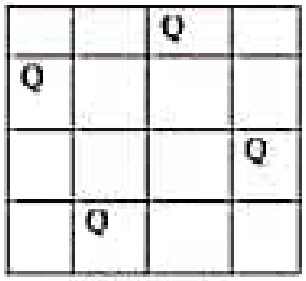
Algorithm Nqueens (k,n) *// recursive algorithm*
//using backtracking it prints all possible positions of n queens in 'n*n' chessboard so
//that they are non attacking.

```
{
   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)   ;
        }
   }
}
```

Example: 4 queens.
Two possible solutions are



Solutin-1
(2 4 1 3)

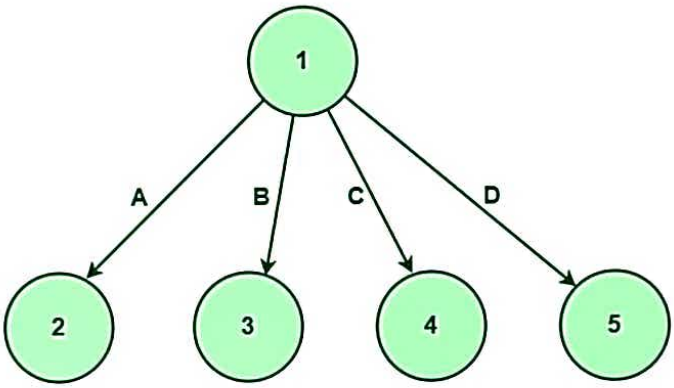


Solution 2
(3 1 4 2)

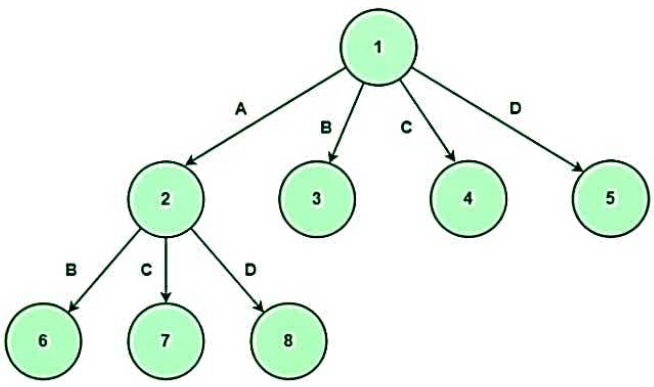
5a

FIFO Branch and Bound:

FIFO Branch and Bound
First-In-First-Out is an approach to the branch and bound problem that uses the queue approach to create a state-space tree. In this case, the breadth-first search is performed, that is, the elements at a certain level are all searched, and then the elements at the next level are searched, starting with the first child of the first node at the previous level.
For a given set {A, B, C, D}, the state space tree will be constructed as follows :

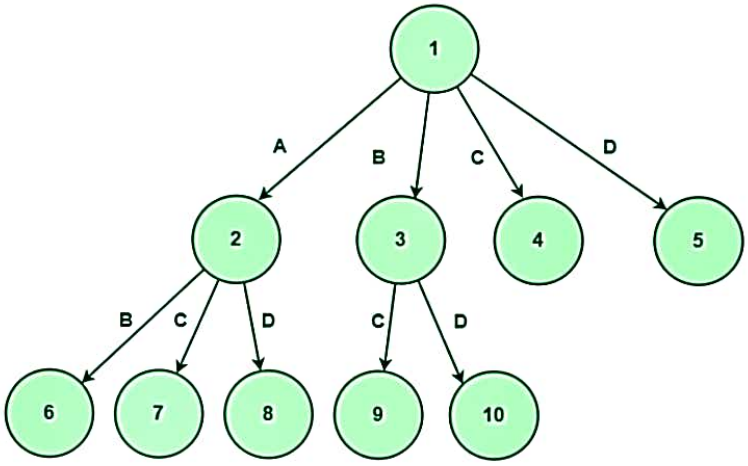


State Space tree for set {A, B, C, D}
The above diagram shows that we first consider element A, then element B, then element C and finally we'll consider the last element which is D. We are performing BFS while exploring the nodes.
So, once the first level is completed. We'll consider the first element, then we can consider either B, C, or D. If we follow the route then it says that we are doing elements A and D so we will not consider elements B and C. If we select the elements A and D only, then it says that we are selecting elements A and D and we are not considering elements B and C.



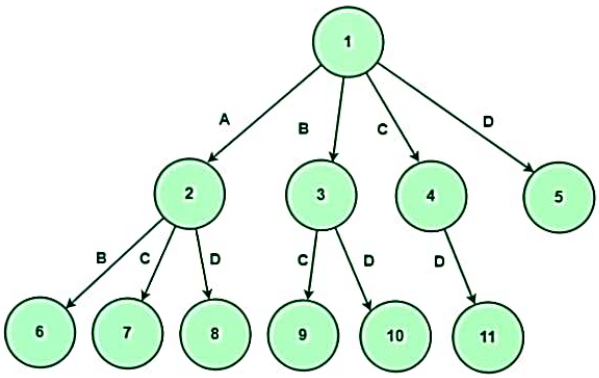
Selecting element A

Now, we will expand node 3, as we have considered element B and not considered element A, so, we have two options to explore that is elements C and D. Let's create nodes 9 and 10 for elements C and D respectively.



Considered element B and not considered element A

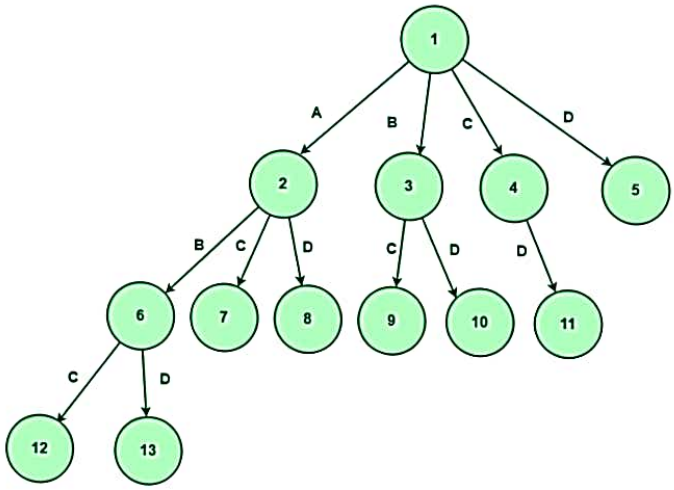
Now, we will expand node 4 as we have only considered elements C and not considered elements A and B, so, we have only one option to explore which is element D. Let's create node 11 for D.



Considered elements C and not considered elements A and B

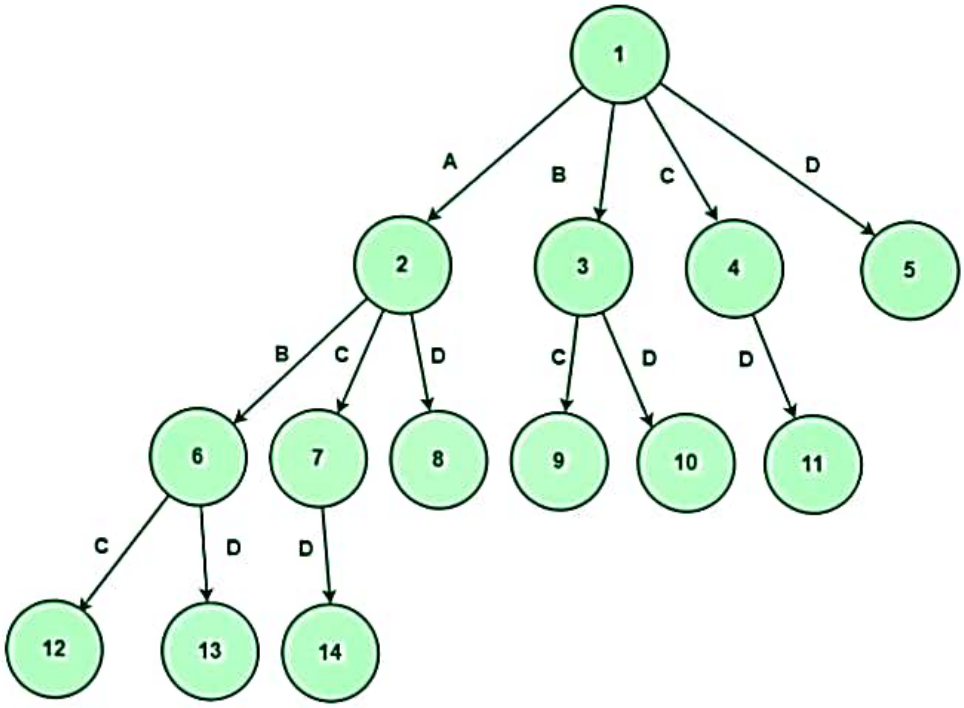
Till node 5, we have only considered elements D, and not selected elements A, B, and C. So, We have no more elements to explore, Therefore on node 5, there won't be any expansion.

Now, we will expand node 6 as we have considered elements A and B, so, we have only two option to explore that is element C and D. Let's create node 12 and 13 for C and D respectively.



Expand node 6

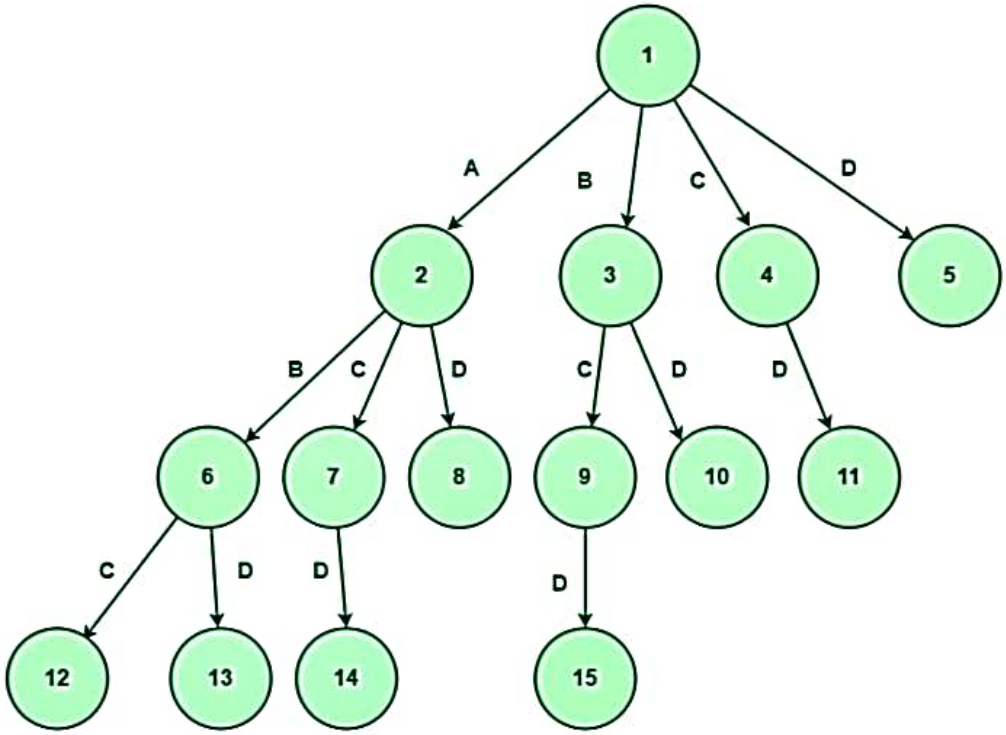
Now, we will expand node 7 as we have considered elements A and C and not consider element B, so, we have only one option to explore which is element D. Let's create node 14 for D.



Expand node 7

Till node 8, we have considered elements A and D, and not selected elements B and C, So, We have no more elements to explore, Therefore on node 8, there won't be any expansion.

Now, we will expand node 9 as we have considered elements B and C and not considered element A, so, we have only one option to explore which is element D. Let's create node 15 for D.



Expand node 9

6a LC-search (Least Cost Search) is **a search strategy that uses a heuristic cost function to select the next E-node**. The node with the least value of the cost function is selected as the next E-node.

The LC-search algorithm:

- Uses a ranking function to calculate the cost of each node
- Generates the children of the E-node
- Selects the node with the minimum cost

The LC-search algorithm is considered the most intelligent because it selects the next node based on a heuristic cost function.

The LC-search algorithm uses the following formula to assign a rank to a node:

- $c(x) = f(h(x)) + g(x)$:

Where, $c(x)$ is the cost of x.

BFS and DFS are special cases of LC-search.

```
1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```