Module 3

Backtracking: Control Abstraction, N-Queens problem, Sum of Subsets Problem

Branch and Bound: Control Abstraction, 8- Puzzle problem

Lower Bounds: The Decision Tree method, Lower Bounds for Comparison based Sort and Searching (Analysis not required)

Backtracking

- Backtracking is one of the techniques that can be used to solve the problem.
- It uses the **Brute force approach** "Try out all the possible solutions and pick out the best solution from all the desired solutions".
- This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems.
- Backtracking is used when we have multiple solutions, and we require all those solutions.

Backtracking

- 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.
- In backtracking, we represent the solution in the form of tree called **solution tree or state space tree** and the constraint applied to find the solution is called **bounding function**
- **Bounding function** will be used to kill live nodes without generating all their children if it does not lead to a feasible solution.

Backtracking:example

3 chairs , 3 students.

↓

2 Boys , 1 Girl.

* We arrange them in 3! ways.

Constraint :-

"Girl should not sit in the middle".

Backtracking:example

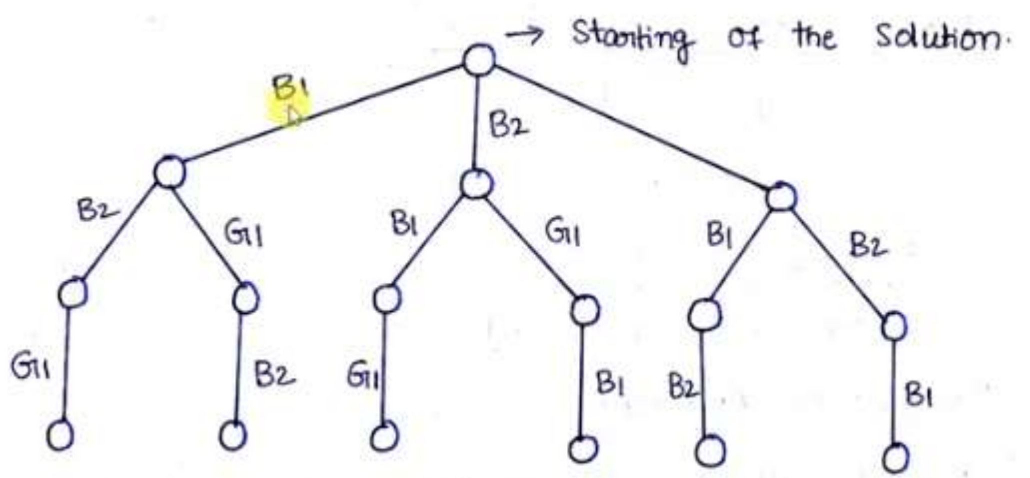


fig: State Space Tree.

* 6 Possibilities.

Backtracking:example

→ Considering the constraint, the possible solutions are:-

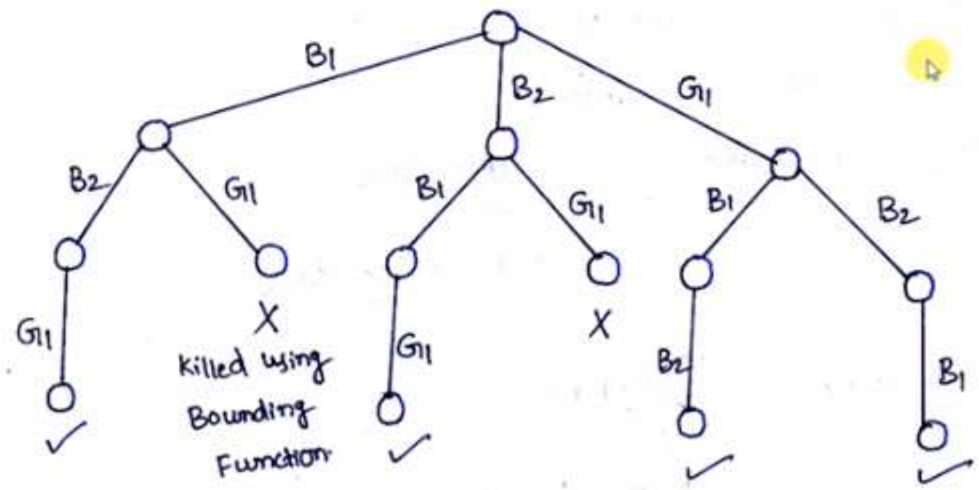


fig: State Space Tree.

If we reach the last level, we get solution. Total 4 solution

Backtracking

- Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:
- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Explicit Constraints

- Explicit constraints are rules that **restrict each x_i to take one value only from a given set.**
- **Example 8-queens**
  - The explicit constraints Si={ 1,2,3,4,5,6,7,8}
- The explicit constrains depends on the particular instance **I** of the problem being solved. All tuples that satisfy the explicit constraints define a possible ***solution space*** for **I**.

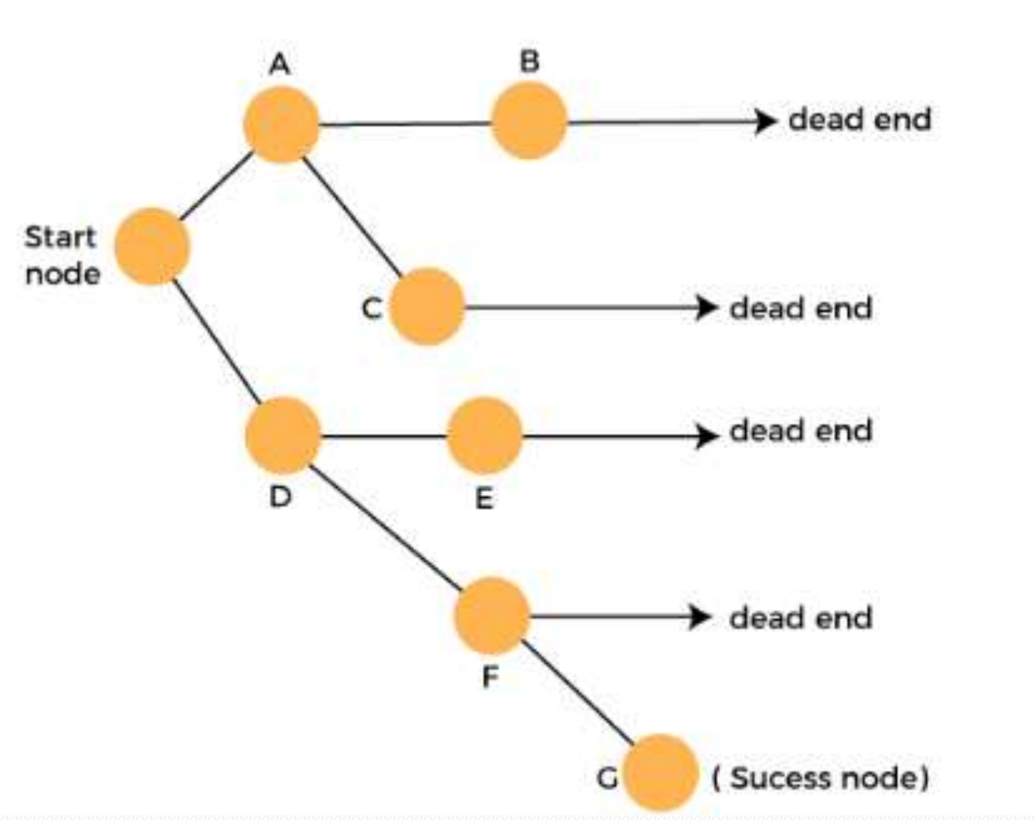
Implicit Constraints

- The implicit constraints are rules that determine which of the tuples in the *solution space* of an instance **I** of a problem ***satisfy the criterion function.***
- The implicit constraints describe the way in which the x_i must relate to each other.
- **Example 8-queens**
  - Implicit constraints : no two x_i's can be the same column and no two queens can be on the same diagonal

Applications of Backtracking

- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

Backtracking-Tree organization



Terminology

- Backtracking determines the solution by *systematically searching* the solution space for the given problem instance
- This is done by using a *tree organization*
- For a given problem many tree organization may be possible
- Each node in the tree defines *a problem state*

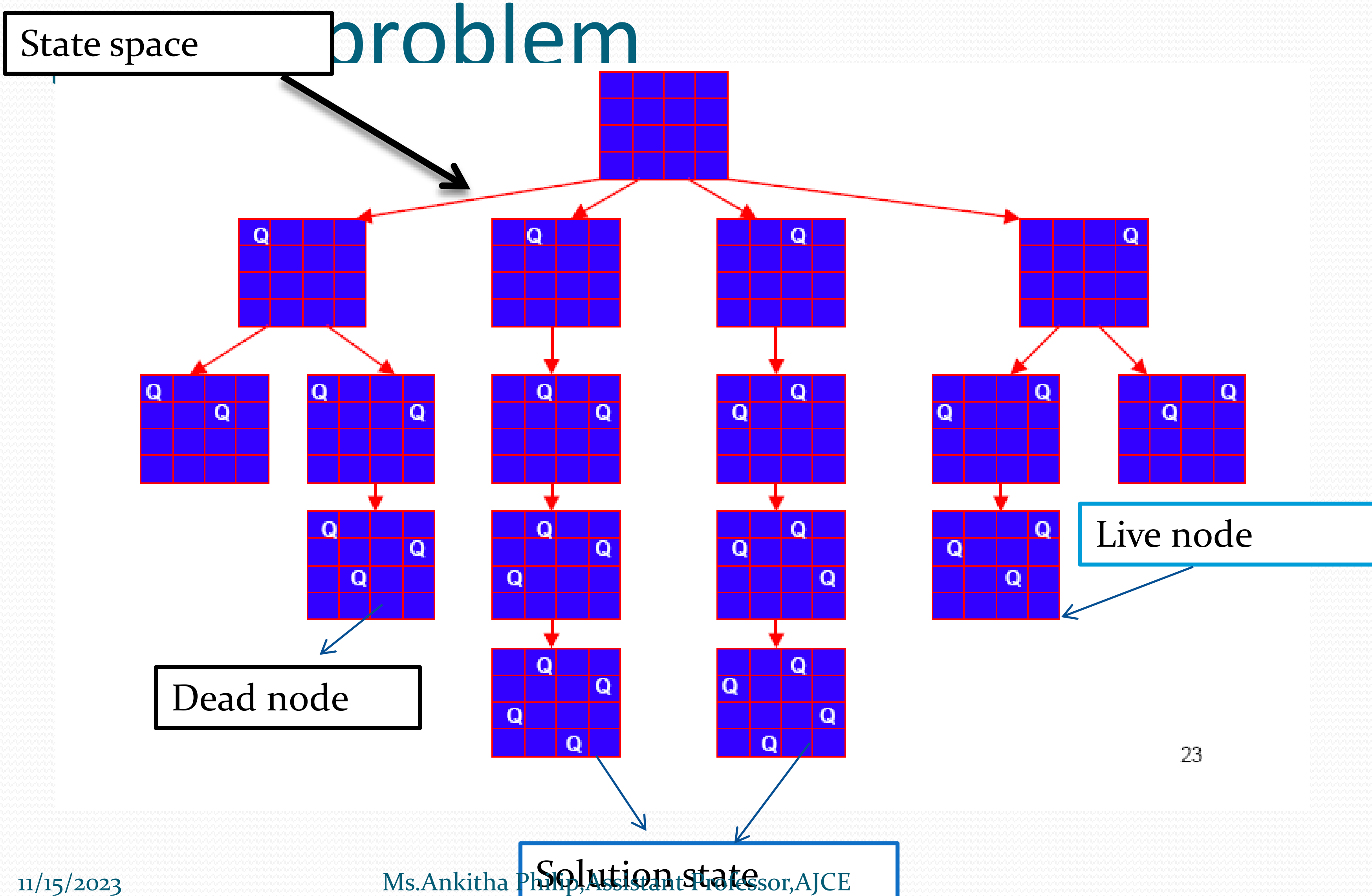
Terminology

- All paths from the root to other node defines the *state space* of the problem.
- *Solution state* of the problem state s is the path from the root to s defines a tuple in the solution space
- The tree organization of the solution space is referred as the *state space tree*

Terminology

- **Live node:** The nodes which has been generated are known as live nodes.
- **E node:** The live nodes whose children are being generated.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

State Space Tree of the Four-problem



General Iterative Backtracking Method Algorithm

```
Algorithm IBacktrack(n)
//This schema describes the backtracking process.
//All solutions are generated in x[1:n] and printed as soon as they are
// determined.
{
        k=1;
        while(k ≠ 0) do
        {
            if(there remains an untried x[k] € T(x[1],x[2],…….x[k-1] )
                    and Bk (x[1],…………x[k]) is true then
                    {
                            if(x[1],………x[k] is a path to an answer node)
                                    Then write (x[1 : k]);
                            K=k + 1;// consider the next set
                    }
                    else k = k -1; // backtrack to the previous set.
        }
}
```

General Recursive Backtracking Algorithm

```
Algorithm Backtrack(k)
//This schema describes the backtracking process using recursion.
//On entering, the  first k-1 values x[1],x[2]……..x[k-1] of
//the solution vector x[1:n] have been assigned.
//X[] and n are global.
{
          for (each  x[k] € T(x[1],………….,x[k-1])do
          {
                    if(B_k( x[1],x[2],………..x[k]) !=0)then
                    {
                              if (x[1],x[2],………………….,x[k] is a path to an answer node)
                                        then write (x[1:k]);
                              if (k<n) then Backtrack(k + 1);
                    }
          }
}
```

The n-Queen problem

- Place n queens on an n by n chess board so that no two of them are on the same row, column, or diagonal
- NOTES: A queen can attack horizontally, vertically, and on both diagonals, so it is pretty hard to place several queens on one board so that they don't attack each other

The n-queens problem and solution

- In implementing the n – queens problem we imagine the chessboard as a two-dimensional array A (1 : n, 1 : n).
- The condition to test whether two queens, at positions (i, j) and (k, l) are on the same row or column is simply to check $i = k$ or $j = l$
- The conditions to test whether two queens are on the same diagonal or not are to be found

The n-queens problem and solution contd..

Observe that

i) For the elements in the the upper left to lower Right diagonal, the column values are same or row- column = 0, e.g. 1-1=2-2=3-3=4-4=0

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	row + (3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

ii) For the elements in the upper right to the lower left diagonal, row + column value is the same e.g. 1+4=2+3=3+2=4+1=5

The n-queens problem and solution contd..

- Thus two queens are placed at positions (i, j) and (k, l), then they are on the same diagonal only if

  $i - j = k - l$ or $i + j = k + l$

  or $j - l = i - k$ or $j - l = k - i$

- Two queens lie on the same diagonal if and only if

  $$|j - l| = |i - k|$$

Algorithm

```
Algorithm nqueen(k,n)
//this procedure prints all possible
// placement of n queue on an n*n
//chess board 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 nqueen(k+1,n);
   }
   }
   }
```

```
Algorithm place(k,i)
//return true if a queen can be placed in
//kth row ith column. Else it return false.
// X[] is a global array. abs (r ) returns
//absolute value of r
{
for j =1 to k –1 do
if ((x[j] =i) //same column
or (abs(x[j] –i)= abs (j-k))//same diagonal
then return false;
return true;
}
```

N Queen problem- Algorithm

Algorithm NQueens(k,n)

*//using backtracking , this procedure prints all possible placements of n queens on an 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);
                }
        }
}
```

N Queen problem- Algorithm

Algorithm Place (k,i)

//returns true if a queen can be placed in the k^{th} row and i^{th} column. Ow //it returns false. X[] is a global array whose first (k-1) values have been //set. ABS (r) retur

two are in the same column

in the same diagonal

```
{
        for j ← 1 to k-1 do
                if ([X(j] = i )  or (ABS(X[j] – i) = ABS(j-k)) )
                        then Return (false)
        return (true)
}
```

4-queen solution

- NQ(1,4)
- i=1 k=1
- Place(1,1)=true
- X[1]=1
- NQ(2,4)
- i=1,k=2,x[1]=1
- Place(2,1)=false
- Place(2,2)=false
- Place(2,3)=true
- X[2]=3
- NQ(3,4)
- Place(3,1)=false
- Place(3,2)=false
- Place(3,3)=false
- Place(3,4)=false // backtrack

- NQ(2,4)
- Place(2,4)=true
- X[2]=4
- NQ(3,4)
- Place(3,1)=false
- Place(3,2)=true
- X[3]=2;
- NQ(4,4)
- Place(41)=false
- Place(42)=false
- Place(43)=false
- Place(44)=false//backtrack
- NQ(3,4)
- Place(3,3)=false
- Place(3,4)=false//backtrack
- NQ(2,4) //backtrack

- NO(1,4)
- Place(1,2)==true
- X[2]=1
- NQ(2,4)
- Place(2,4)=true
- X[2]=4
- NQ(3,4)
- Place(3,1)=true
- X[3]=1
- NQ(4,4)
- Place(4,3)=true
- X[4]=3

BACKTRACKING (Contd..)

Example : 4 Queens problem

1			

1			
.	.	2	

1			
			2

1			
			2
	3		
.	.	.	.

	1		

	1		
			2
3			
.	,	4	

BRANCH AND BOUND

"Branch and bound is a state space search method in which all the children of a E-node are generated before any other live node(active node) can become E-node "

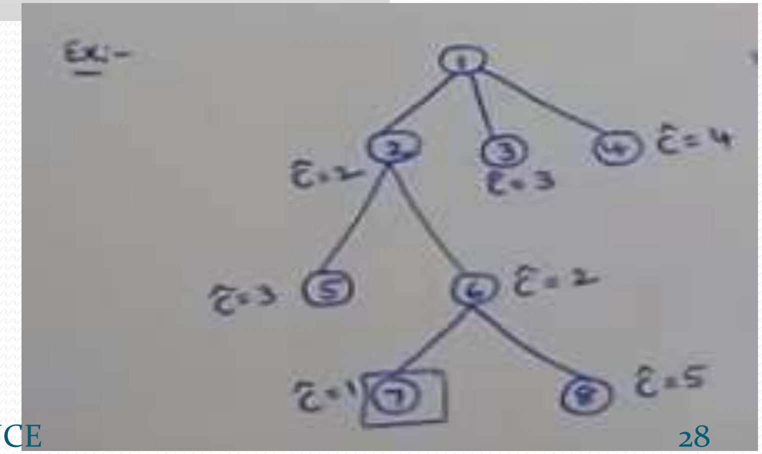
BRANCH AND BOUND

Terminologies Used

Active node - is a node that has been generated but whose children have not yet been generated.

E-node - is an active node whose children are currently being explored.

dead node - is a generated node that should not be expanded or explored further. All the children of a dead node have already been expanded



Branch and Bound

- 3 methods
- LIFO B&B(Last in first out)
  - Also known as DFS(Depth First Search) B&B
  - Implemented using stack
- FIFO(First In First Out)
  - Also known as BFS(Breadth First Search) B&B
  - Implemented using queue
- LC(Least Cost)
  - Implemented using priority queue
  - Eg: 15 puzzle problem

FIFO B&B

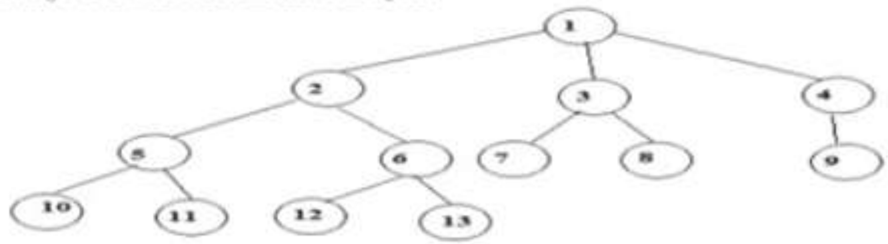
FIFO B&B:

FIFO Branch & Bound is a BFS.

In this, children of E-Node (or Live nodes) are inserted in a queue.

Implementation of list of live nodes as a queue

- ✓ Least()→ Removes the head of the Queue
- ✓ Add()→ Adds the node to the end of the Queue



Assume that node '12' is an answer node in FIFO search, 1st we take E-node has '1'

1

2	3	4

3	4	5	6

4	5	6

5	6

6

LIFO B&B

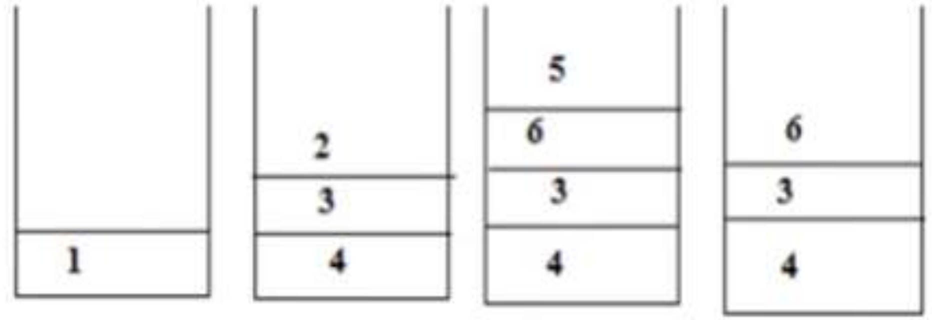
LIFO B&B:

LIFO Brach & Bound is a D-search (or DFS).

In this children of E-node (live nodes) are inserted in a stack

Implementation of List of live nodes as a stack

- ✓ Least()→ Removes the top of the stack
- ✓ ADD()→Adds the node to the top of the stack.



LC B&B

Least Cost (LC) Search

- The selection rule for the next E-node in FIFO or LIFO is sometimes "blind".
- The search for an answer node can be speeded by using an "intelligent" ranking function called an approximate cost function "$\hat{C}$".
- E-node - is the live node with the best $\hat{C}$ value.
- $\hat{C}=g(X)+H(X)$.

  Where

  - g(X) - is an additional effort needed to reach an answer node from x.
  - H(X) - is the cost of reaching x from the root

Control Abstraction-LC Search

```
Algorithm LCSearch(t)
// search t for an answer node
{
        if  t is the answer node then output t  and return;
        E = t    //E-node
        initialize the list of live nodes to be empty;
        repeat
        {
            for each child x of E do
                 {
                          if x is an answer node then output the
                          path from x to t and return;
```

```
        Add(x);     // x is a live node.
        (x → parent) =E   //Pointer for path to root
        }
        if there are no more live node then
        {
                write(“No answer node”); return;
        }
        E:= Least();
    } until (false)
}
```

Least()--- find a live node with least c(). This node is deleted from the list of live nodes and returned.

Lower Bound Theory

- Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.
- Lower Bound Theory uses a number of methods/techniques to find out the lower bound.
- **Concept/Aim:** The main aim is to calculate a minimum number of comparisons required to execute an algorithm.

Lower Bound Theory

The techniques which are used by lower Bound Theory are:

- Comparisons Trees.
- Oracle and adversary argument
- State Space Method

Comparison trees:

- In a comparison sort, we use only comparisons between elements to gain order information about an input sequence (a1; a2……an).
- **Given a_i,a_j from (a_1, a_2…..a_n)We Perform One of the Comparisons**
- $a_i < a_j$ less than
- $a_i \le a_j$ less than or equal to
- $a_i > a_j$ greater than
- $a_i \ge a_j$ greater than or equal to
- $a_i = a_j$ equal to

Comparison trees:

- Consider sorting three numbers a1, a2, and a3. There are 3! = 6 possible combinations:
- (a1, a2, a3), (a1, a3, a2),
- (a2, a1, a3), (a2, a3, a1)
- (a3, a1, a2), (a3, a2, a1)
- The Comparison based algorithm defines a decision tree.

Decision Tree:

- A decision tree is a full binary tree that shows the comparisons between elements that are executed by an appropriate sorting algorithm operating on an input of a given size.
- Control, data movement, and all other conditions of the algorithm are ignored.
- In a decision tree, there will be an array of length n.
- So, total leaves will be **n!** (I.e. total number of comparisons)

<u>Example of comparing a1, a2, and a3</u>.

- Left subtree will be true condition i.e. $a_i \leq a_j$
- Right subtree will be false condition i.e. $a_i > a_j$

Decision Tree:

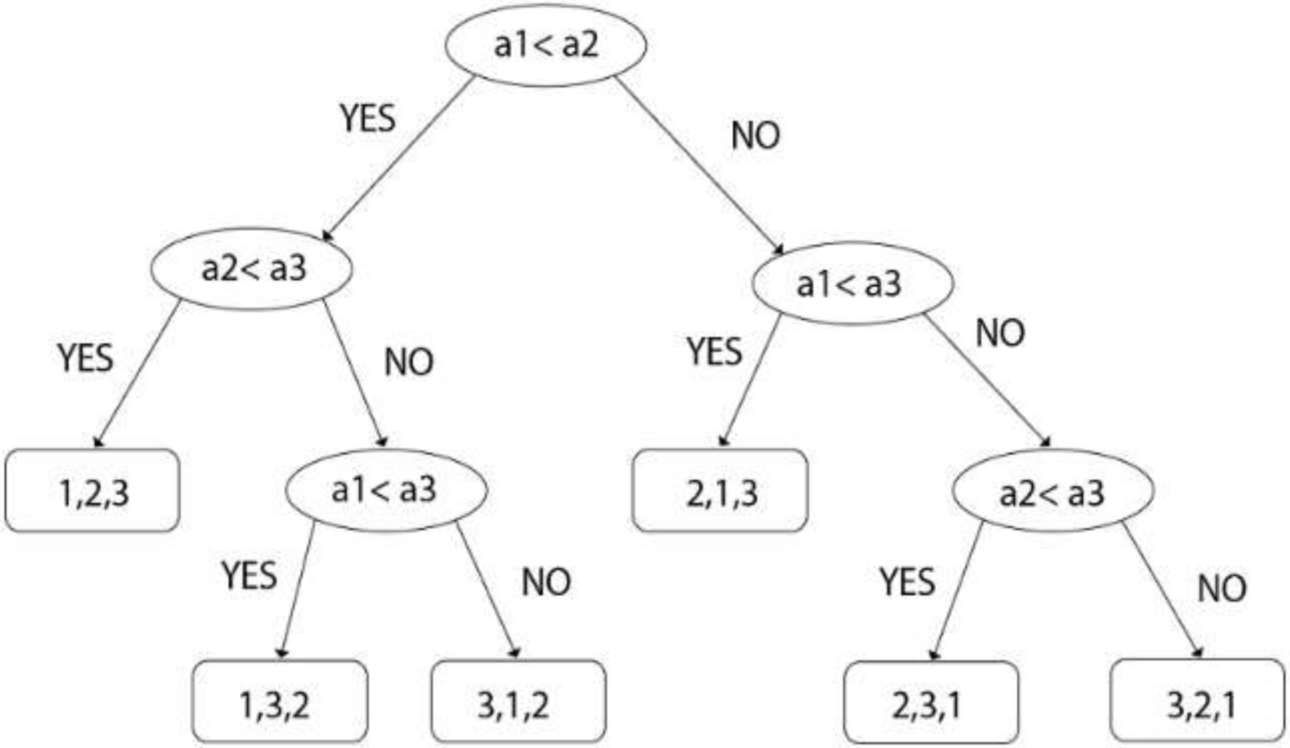


Fig: Decision Tree

Comparison trees:

- What is the lower bound of the time complexity of Comparison based sorting algorithms?
- **O(nlogn)**

If tree height is h, then surely

$n! \leq 2^n$ (tree will be binary)

Taking Log both sides

Log n! ≤ h log 2

Ignoring the Constant terms

$h \geq n \log_2 n$

Comparison tree for Binary Search:

- **Example:** Suppose we have a list of items according to the following Position:

1,2,3,4,5,6,7,8,9,10,11,12,13,14

Mid = $\left[\left(\frac{1+14}{2}\right)\right] = \frac{15}{2} = 7.5 = \mathbf{7}$

Note: Choose the greatest integer

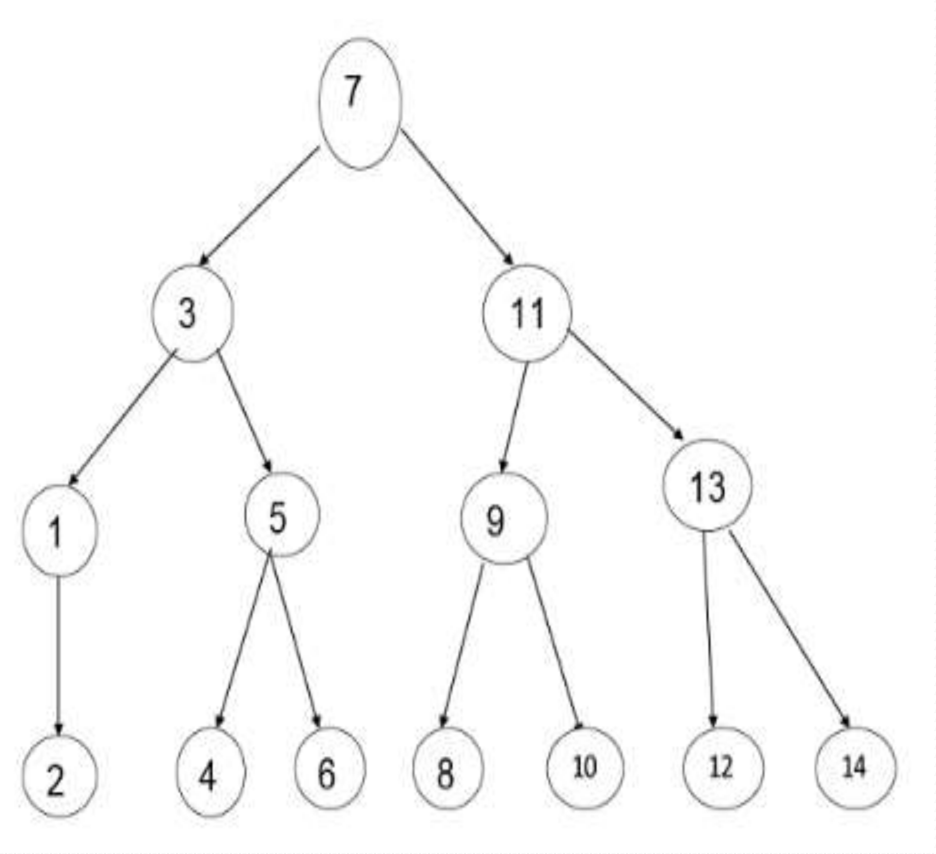
1, 2, 3, 4, 5, 6	8, 9, 10, 11, 12, 13, 14
Mid=$\left(\frac{1+6}{2}\right)$=**3**	Mid =$\left(\frac{8+14}{2}\right)$ = **11**

1, 2	4, 5, 6	8, 9, 10	12, 13, 14
Mid=$\left(\frac{1+2}{2}\right)$=**1**	Mid=$\left(\frac{4+6}{2}\right)$=**5**	Mid= $\left(\frac{8+10}{2}\right)$=**9**	Mid=$\left(\frac{12+14}{2}\right)$=**13**

And the last midpoint is:

2, 4, 6, 8, 10, 12, 14

Comparison tree for Binary Search:



Complexity of binary search

Step1: Maximum number of nodes up to k level of the internal node is 2^k-1

For Example

```
2^k-1
2^3-1= 8-1=7
Where k = level=3
```

Step2: Maximum number of internal nodes in the comparisons tree is n!

Note: Here Internal Nodes are Leaves.

Step3: From Condition1 & Condition 2 we get

```
N! ≤ 2^k-1
14 < 15
Where N = Nodes
```

Step4: Now, $n+1 \leq 2^k$

Here, Internal Nodes will always be less than 2^k in the Binary Search.

Step5:

$$n+1 <= 2^k$$

$$\text{Log } (n+1) = k \log 2$$

$$k >= \frac{\log(n+1)}{\log 2}$$

$$k >= \log_2(n+1)$$

Step6:

T (n) = k

Step7:

$$T(n) >= \log_2(n+1)$$