Mar24 : Day 2

Kiran Waghmare

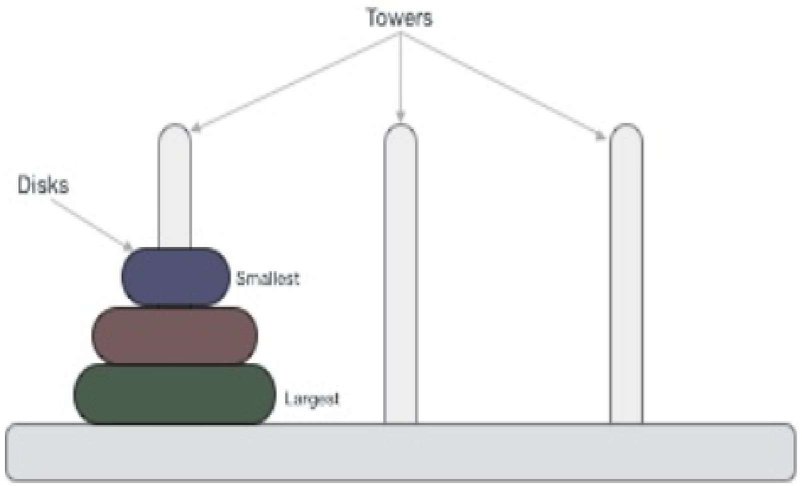
CDAC Mumbai

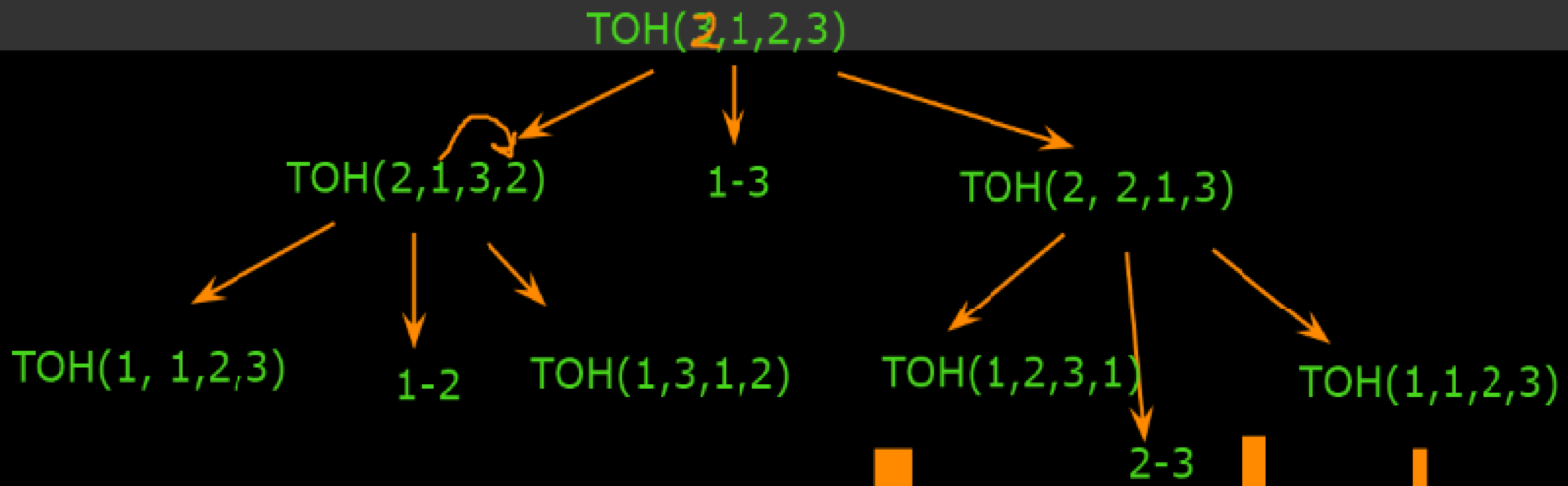
What is Tower of Hanoi?

- A mathematical puzzle consisting of three towers and more than one ring is known as Tower of Hanoi.

- Tower of Hanoi

- The rings are of different sizes and are stacked in ascending order, i.e., the smaller one sits over the larger one. In some of the puzzles, the number of rings may increase, but the count of the tower remains the same.




TOH(~~3~~2,1,2,3)

TOH(2,1,3,2)

1-3

TOH(2, 2,1,3)

TOH(1, 1,2,3)

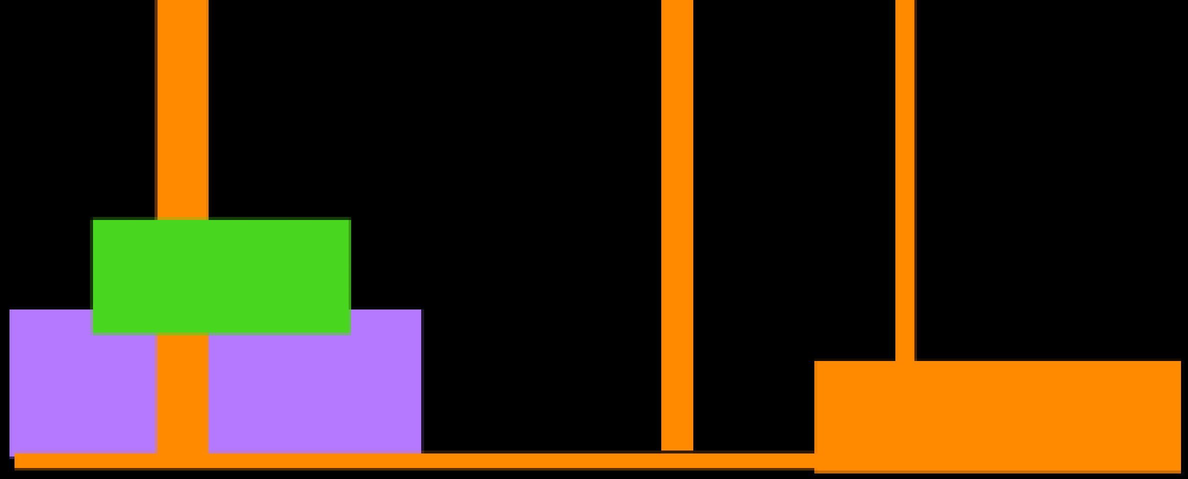
1-2

TOH(1,3,1,2)

TOH(1,2,3,1)

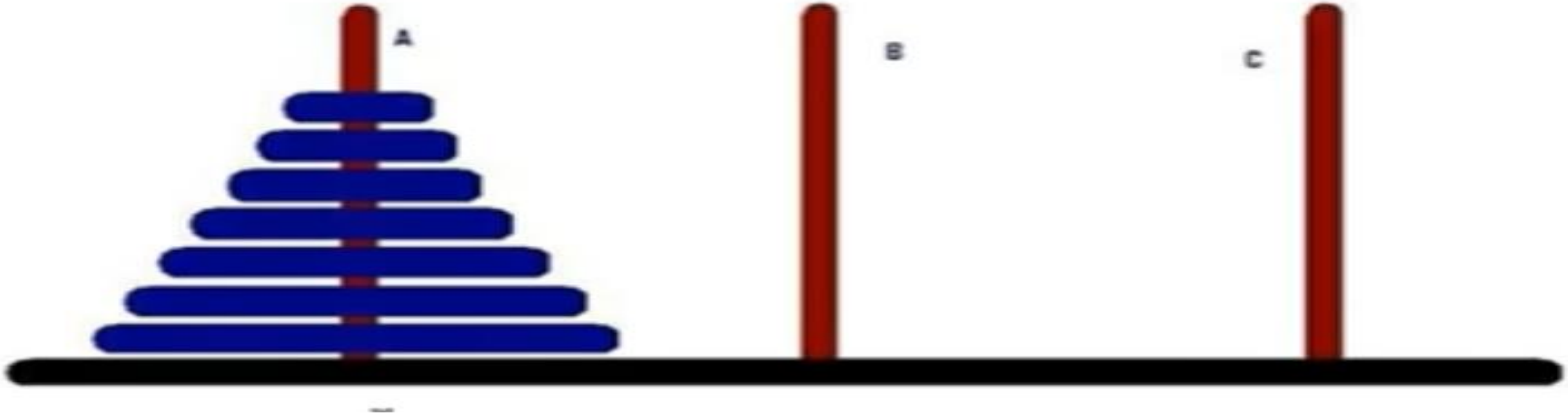
2-3

TOH(1,1,2,3)



Disk(n)	Moves
n=1	1-3
n=2	1-2,1-3,2-3
n=3	1-3,1-2,3-2,1-3,2-1,2-3,1-3

Tower of Hanoi

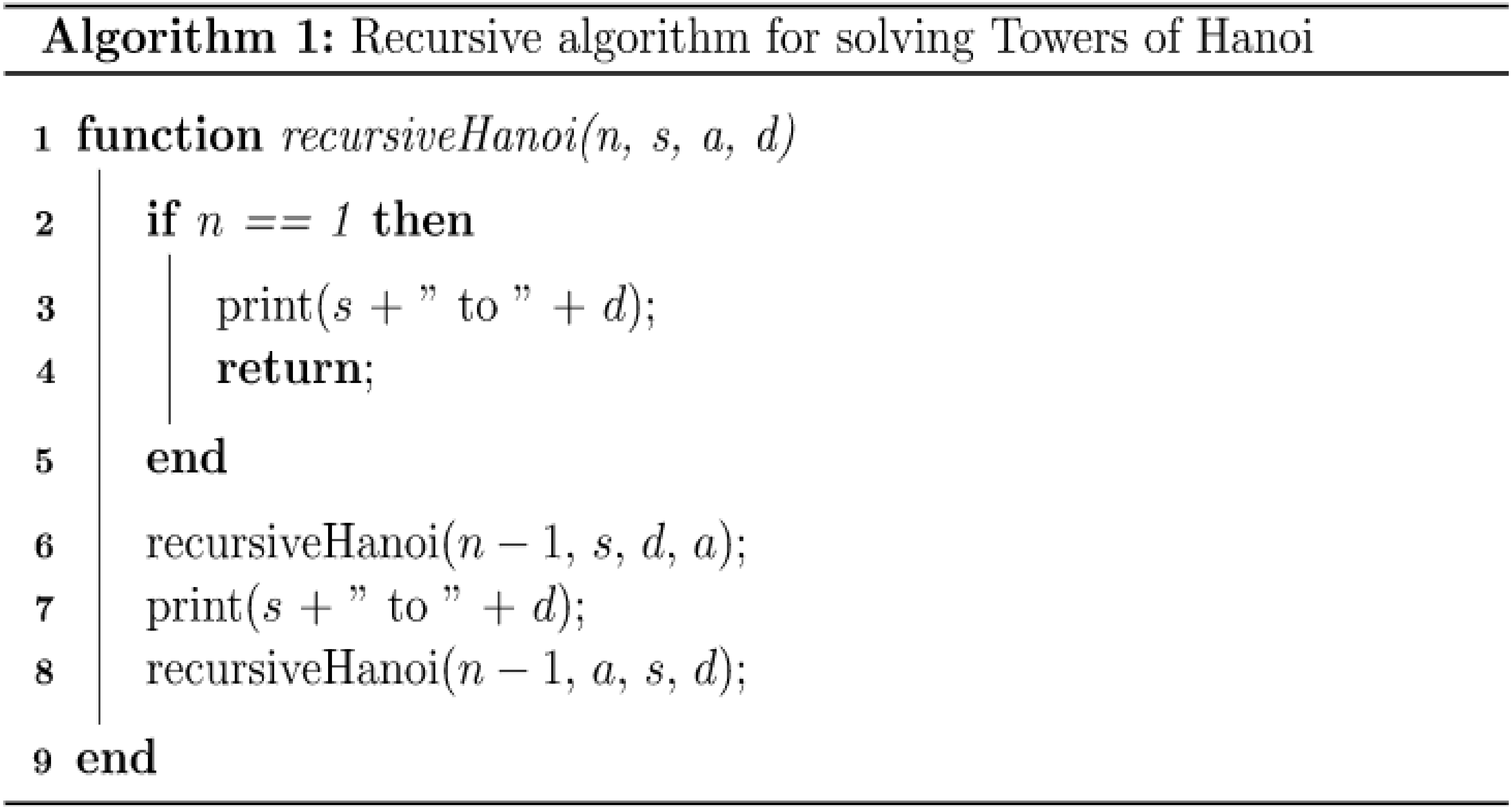


What are the rules to be followed by Tower of Hanoi?

- **The Tower of Hanoi puzzle is solved by moving all the disks to another tower by not violating the sequence of the arrangements.**

The rules to be followed by the Tower of Hanoi are -

1. Only one disk can be moved among the towers at any given time.
2. Only the "top" disk can be removed.
3. No large disk can sit over a small disk.



Algorithm 1: Recursive algorithm for solving Towers of Hanoi

```
function recursiveHanoi($n$, $s$, $a$, $d$)
    if $n == 1$ then
        print($s$ + " to " + $d$);
        return;
    end
    recursiveHanoi($n - 1$, $s$, $d$, $a$);
    print($s$ + " to " + $d$);
    recursiveHanoi($n - 1$, $a$, $s$, $d$);
end
```

Home Work

- Implement Tower of Hanoi Program
- No of Disk=3
- No of Disk=5
- No of Disk=n

Problem 1

Recursive program to find the Sum of the series 1 – 1/2 + 1/3 – 1/4 … 1/N
Given a positive integer N, the task is to find the sum of the series 1 – (1/2) + (1/3) – (1/4) +…. (1/N) using recursion.

Examples:

Input: N = 3
Output: 0.8333333333333333
Explanation:
1 – (1/2) + (1/3) = 0.8333333333333333

Input: N = 4
Output: 0.5833333333333333
Explanation:
1- (1/2) + (1/3) – (1/4) = 0.5833333333333333

Problem 2

Recursive Program to print multiplication table of a number

Given a number N, the task is to print its multiplication table using recursion.

Examples

Input: N = 5

Output:

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Input: N = 8

Output:

```
8 * 1 = 8
8 * 2 = 16
8 * 3 = 24
8 * 4 = 32
8 * 5 = 40
8 * 6 = 48
8 * 7 = 56
8 * 8 = 64
8 * 9 = 72
8 * 10 = 80
```

Problem 3

Recursive program to print formula for GCD of n integers

Given a function gcd(a, b) to find GCD (Greatest Common Divisor) of two number. It is also known that GCD of three elements can be found by gcd(a, gcd(b, c)), similarly for four element it can find the GCD by gcd(a, gcd(b, gcd(c, d))). Given a positive integer n. The task is to print the formula to find the GCD of n integer using given gcd() function.
Examples:

Input : n = 3
Output : gcd(int, gcd(int, int))

Input : n = 5
Output : gcd(int, gcd(int, gcd(int, gcd(int, int))))

Ackermann's function

$A(0, n) = n + 1$

$A(m, 1) = A(m+1, 0)$

$A(m+1, n+1) = A(m, A(m+1, n))$

This function build a VERY deep stack very quickly

Problem 4

Java Program to Reverse a Sentence Using Recursion

A sentence is a sequence of characters separated by some delimiter. This sequence of characters starts at the 0th index and the last index is at len(string)-1. By reversing the string, we interchange the characters starting at 0th index and place them from the end. The first character becomes the last, the second becomes the second last, and so on.

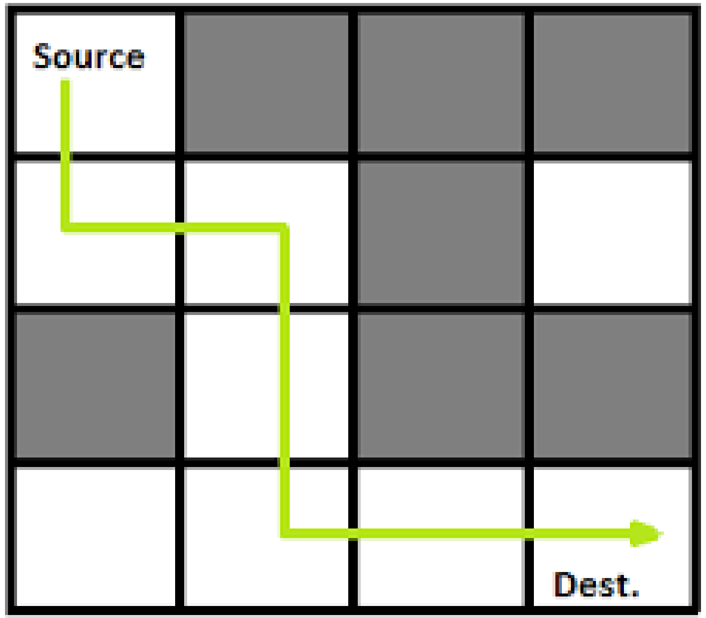
Example:

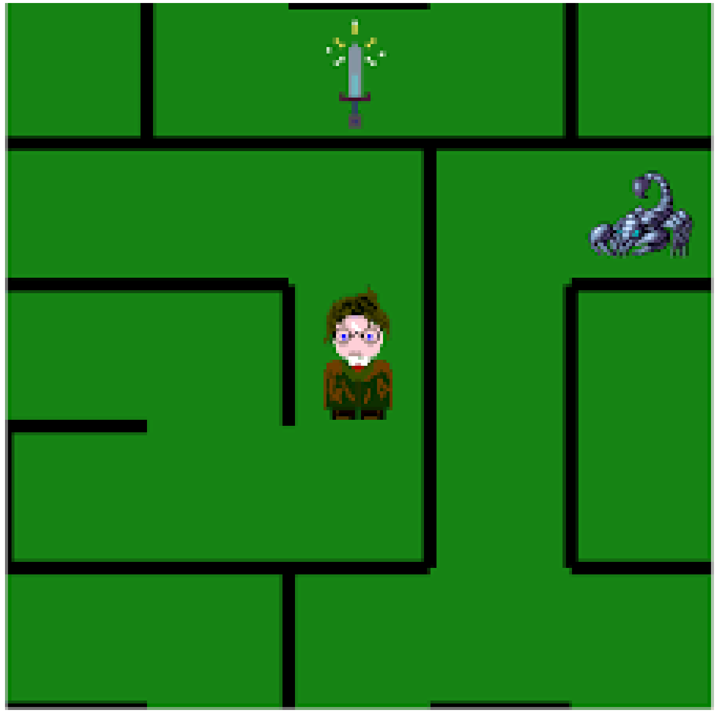
Input : CDACMumbai
Output: iabmuMCADC

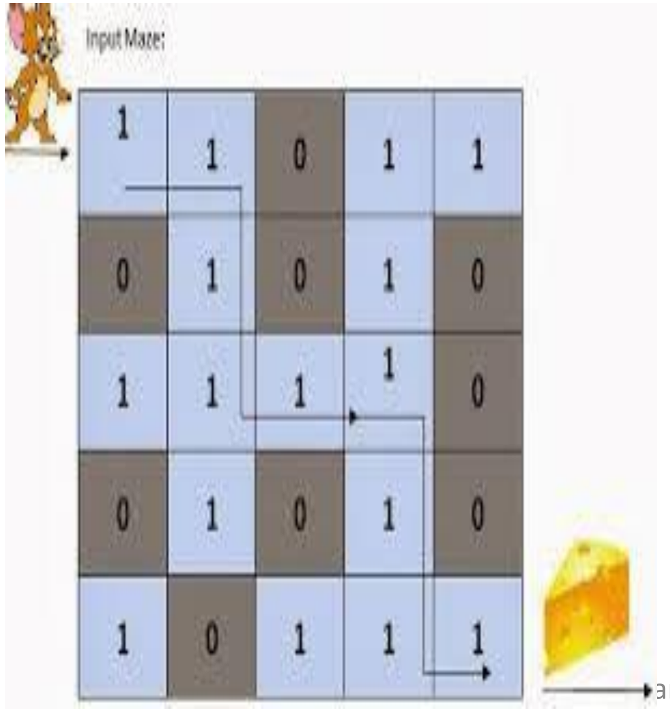
Input : Alice
Output: ecilA
Approach:

Check if the string is empty or not, return null if String is empty.
If the string is empty then return the null string.
Else return the concatenation of sub-string part of the string from index 1 to string length with the first character of a string. e.g. return substring(1)+str.charAt(0); which is for string “Mayur” return will be “ayur” + “M”.

Backtracking



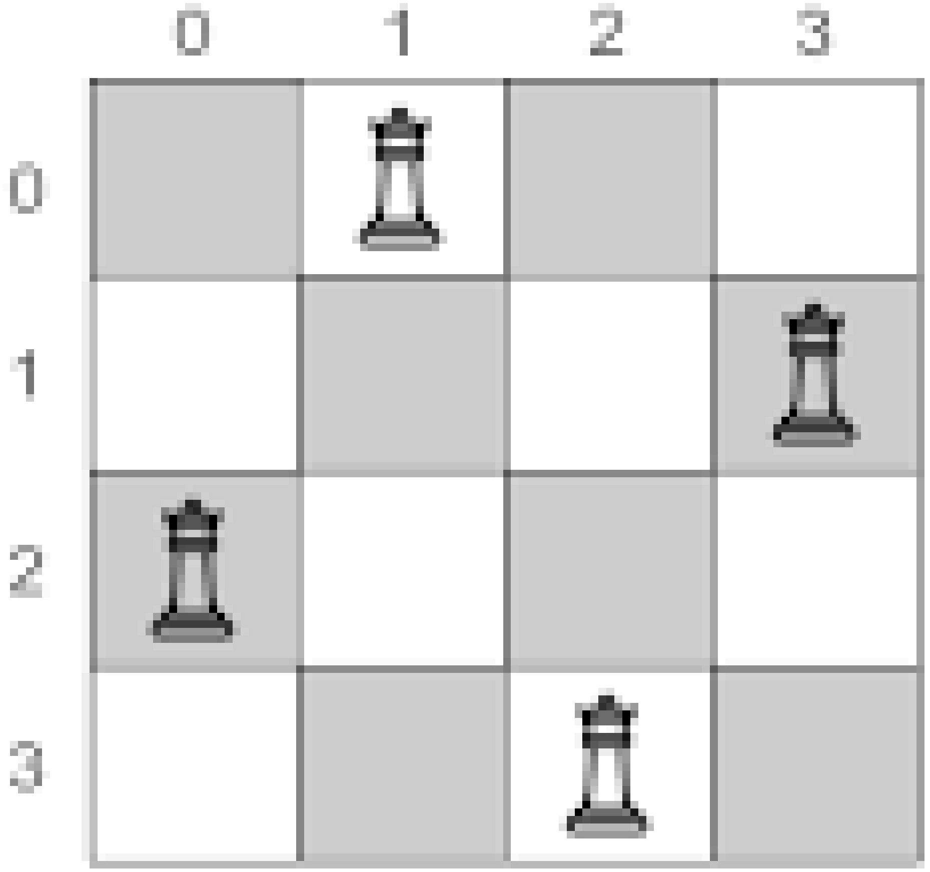




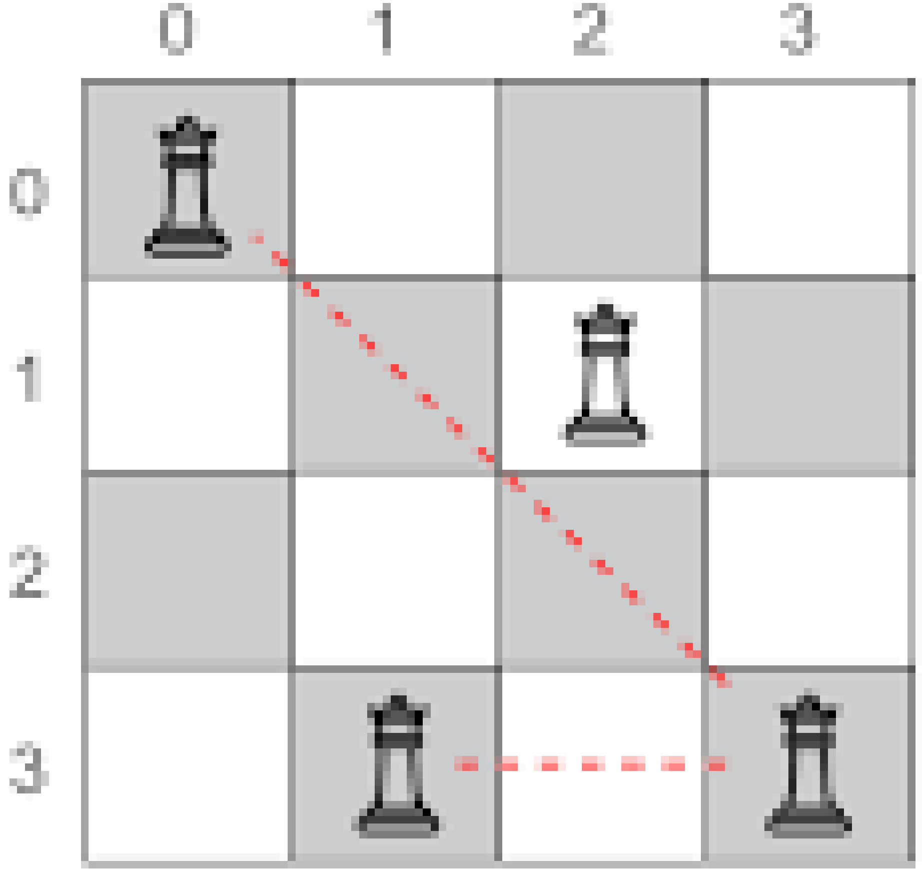
Input Maze:

1	1	0	1	1
0	1	0	1	0
1	1	1	1	0
0	1	0	1	0
1	0	1	1	1

are

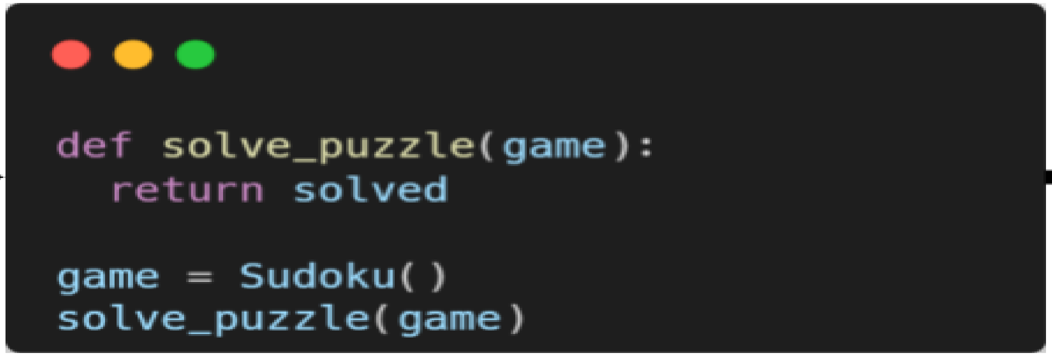


Valid queen positions



Invalid queen positions

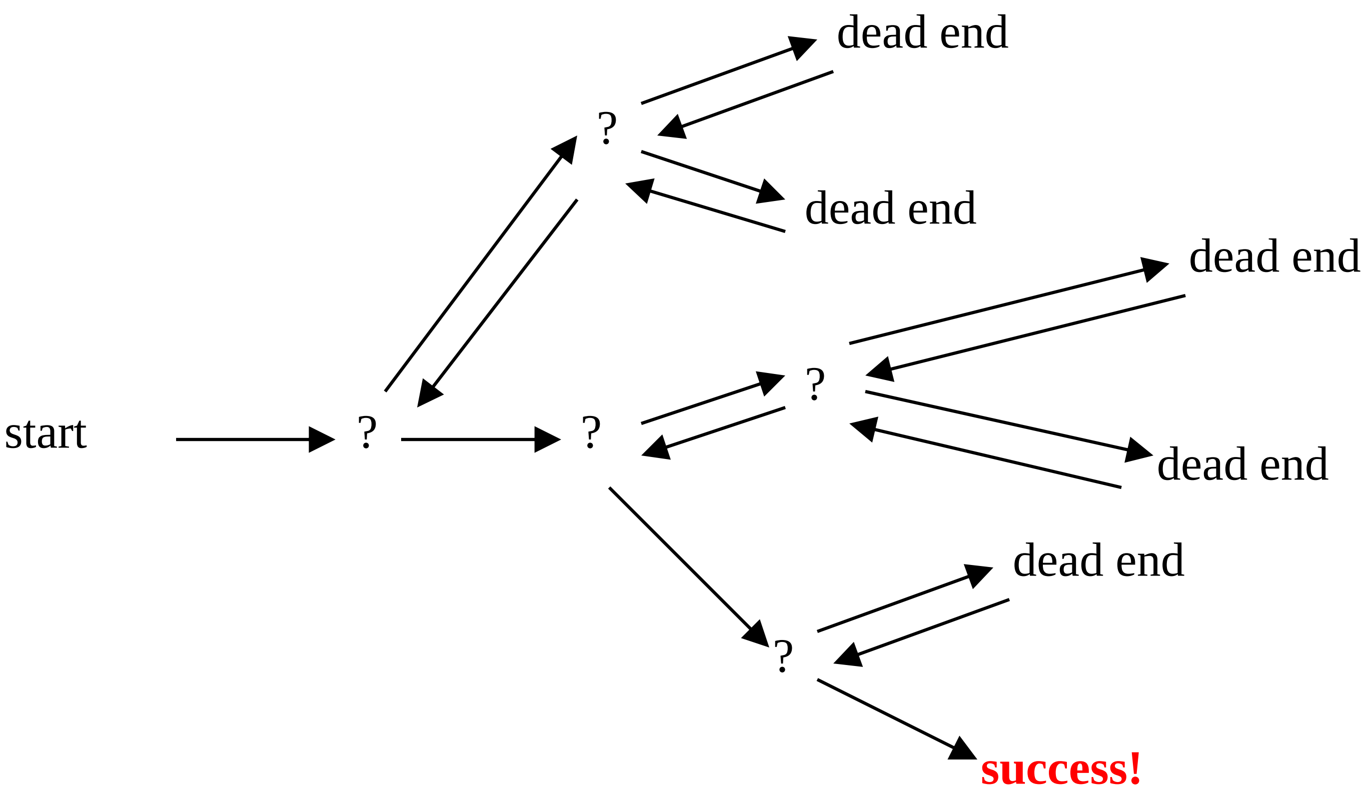
		3			2	6	1	4
		2	6	4	1			8
	1	6		3	5		7	
			3				9	7
6	5				7		3	1
	3	7		5	4			
	7	9			6	2		
			5	8	3	7		
8	4			2			6	



```
def solve_puzzle(game):
  return solved

game = Sudoku()
solve_puzzle(game)
```

5	8	3	9	7	2	6	1	4
7	9	2	6	4	1	3	5	8
4	1	6	8	3	5	9	7	2
1	2	4	3	6	8	5	9	7
6	5	8	2	9	7	4	3	1
9	3	7	1	5	4	8	2	6
3	7	9	4	1	6	2	8	5
2	6	1	5	8	3	7	4	9
8	4	5	7	2	9	1	6	3

Backtracking (animation)



start

?

?

?

dead end

dead end

?

dead end

dead end

?

dead end

success!

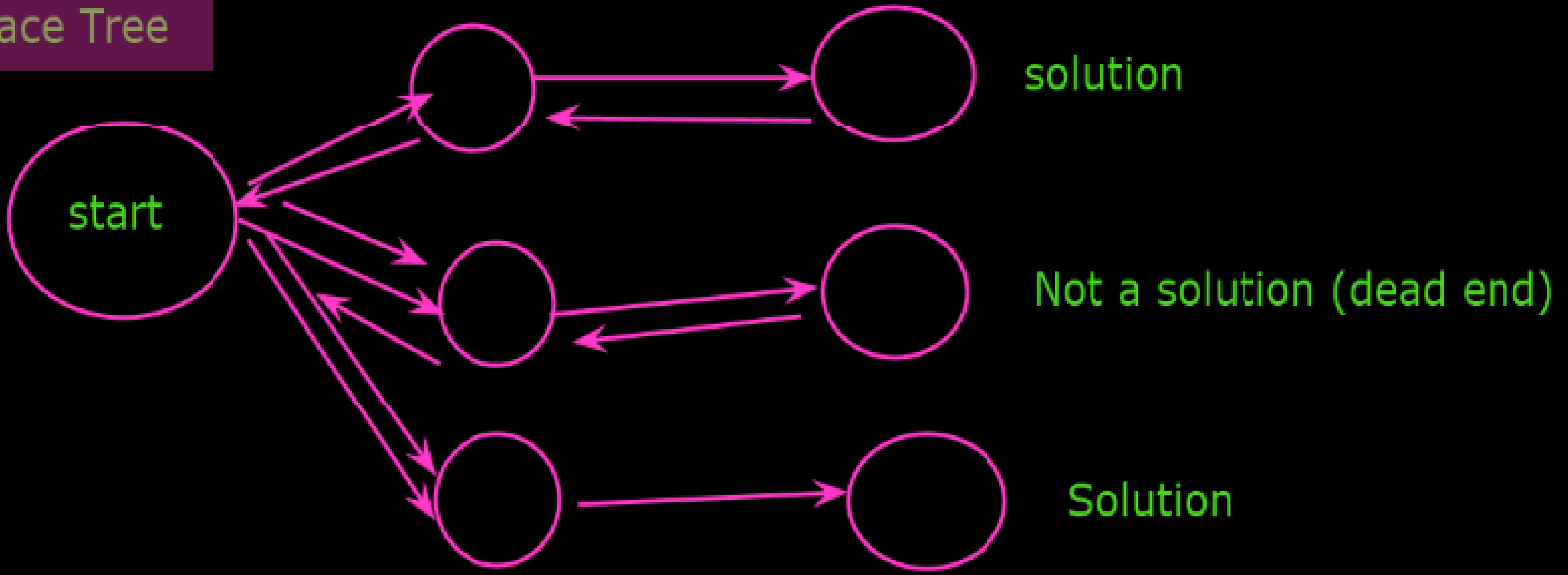
start
explore
backtrack
repeat
constraint check

Backtracking:
-It is a problem solving technique.
    -Backtracking: all possible results.
    -Dynamic Progrqamming: (Optimised result)
    -Greedy Tech : (best of all result)

B1 B2 G1

3!=6

State space Tree

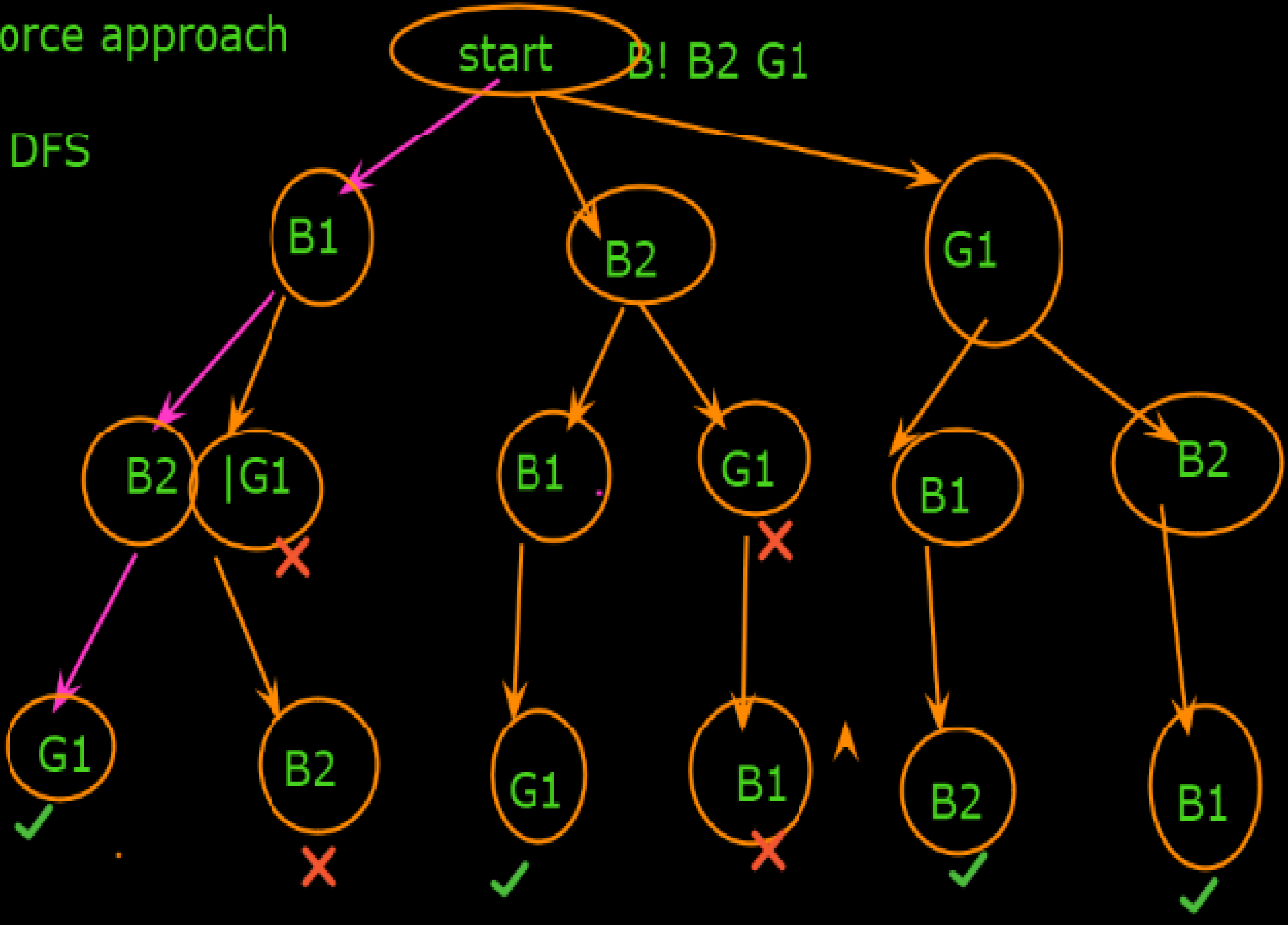


Backtracking

- Backtracking is a **problem-solving technique.**
- It involves systematically **exploring different paths** to find a solution.
- When faced with **multiple choices, backtracking tries each option.**
- It backtracks when it reaches **a dead end**.
- It's akin to navigating through a complex maze.
- **Wrong turns lead to retracing steps until the correct path** is found.
- Backtracking **enables the exploration of various possibilities**.
- It's a **powerful tool for tackling** challenging problems.

Brute-Force approach

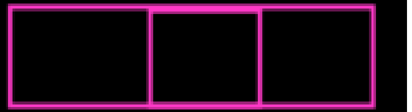
Depth: DFS



start
explore
backtrack
repeat
constraint check

B1 B2 G1

B1 G1 B2

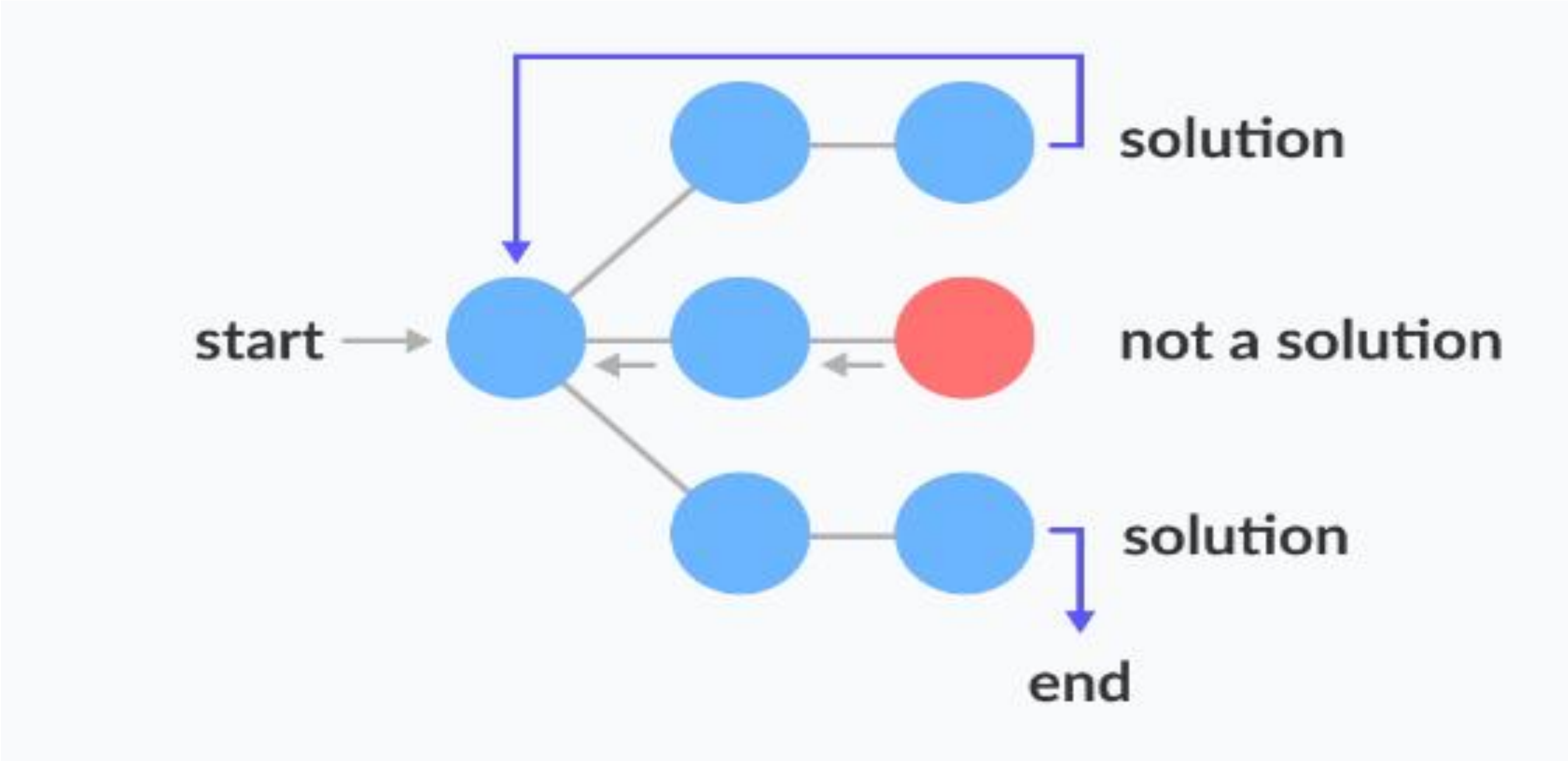


3!=6

Constraint: No girl in center place

State Space Tree

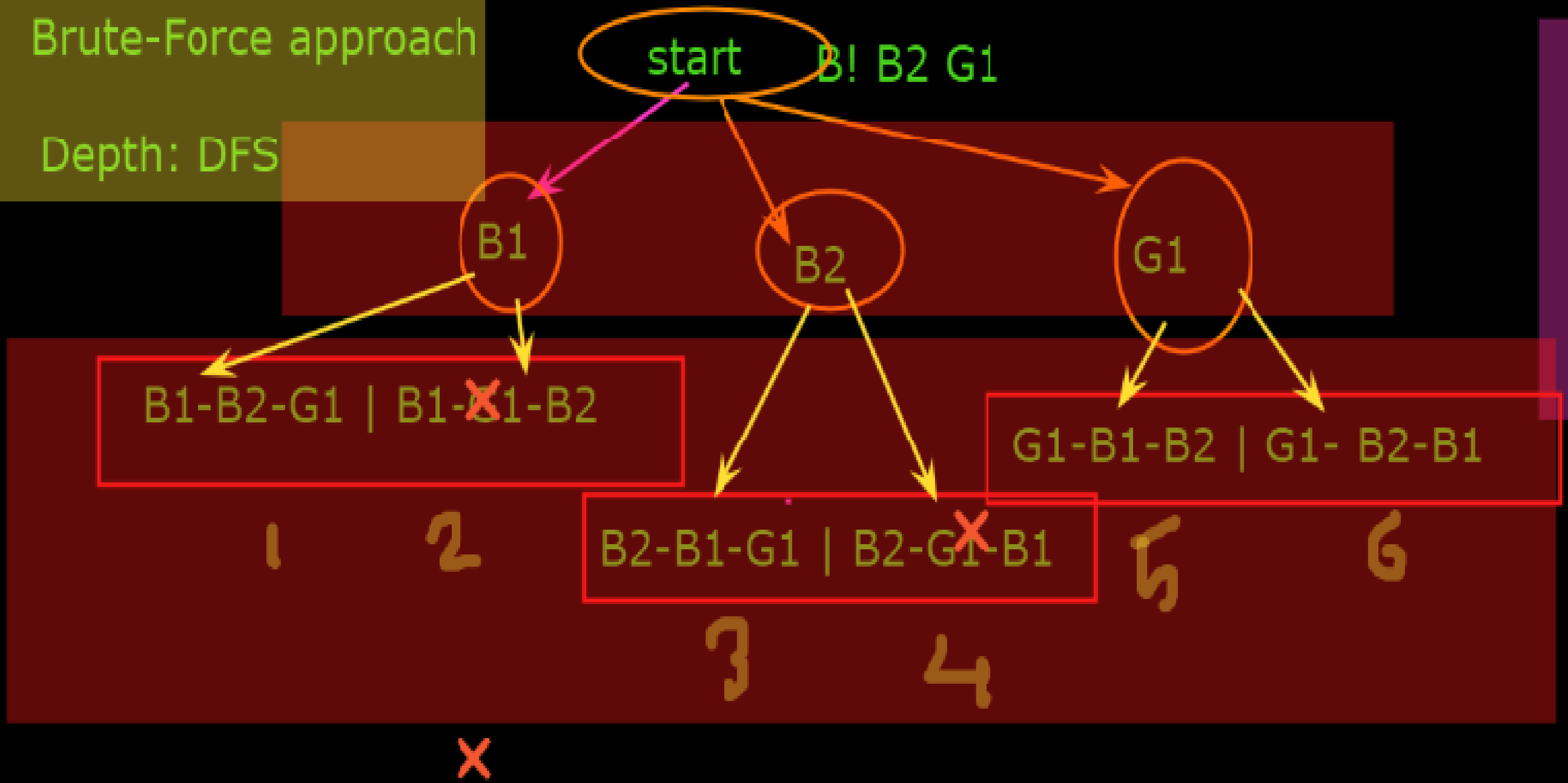
- A space state tree is a tree representing **all the possible states (solution or nonsolution) of the problem** from the root as an initial state to the leaf as a terminal state.



Greedy Tech : (best of all result)

Brute-Force approach

Depth: DFS



start
explore
backtrack
repeat
constraint check

B1 B2 G1

B1 G1 B2

3!=6

Constraint: No girl in center place

Branch and Bound approach

Level wise: BFS

- **Start**: Begin with an initial solution candidate or state.
- **Explore**: Examine all possible next steps or choices from the current state.
- **Constraint check**: Verify whether the current solution candidate satisfies the problem constraints or conditions.
- **Recursion**: If the current candidate satisfies the constraints, recursively explore further by making a choice and moving to the next state.
- **Backtrack**: If the current candidate does not satisfy the constraints, backtrack to the previous state and try a different choice or explore a different path.
- **Repeat**: Repeat steps 2-5 until a valid solution is found or all possible candidates have been explored.

- **Backtracking Algorithm**

```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

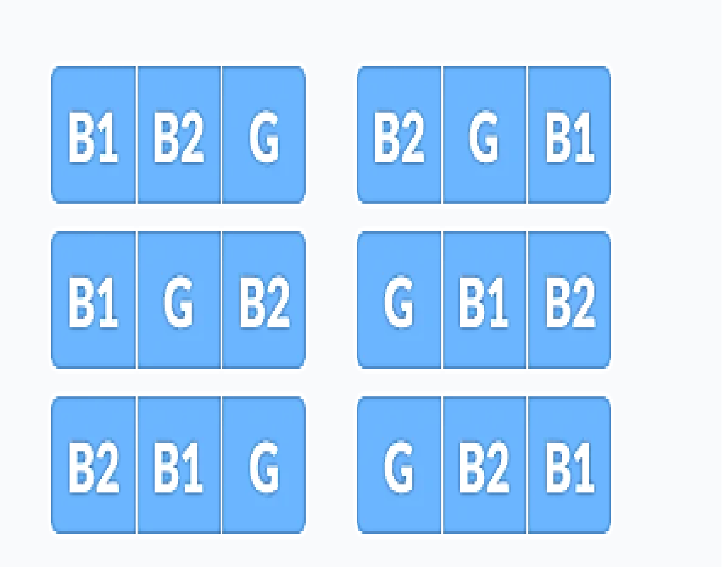
- Backtracking **uses recursive algorithms** to explore potential solutions.
- **Each recursive call makes a choice and explores** further.
- If a dead end is reached, the algorithm backtracks to the previous state.

- Backtracking is commonly **used in problem-solving** scenarios.
- It's utilized in **constraint satisfaction problems, combinatorial optimization, sudoku solving, N-queens problem, graph traversal**, etc.

- Backtracking **efficiently searches through a large solution space**.
- It avoids unnecessary computations by pruning branches that cannot lead to a valid solution.

Example Backtracking Approach

- Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. Constraint: Girl should not be on the middle bench.



B1 B2 G

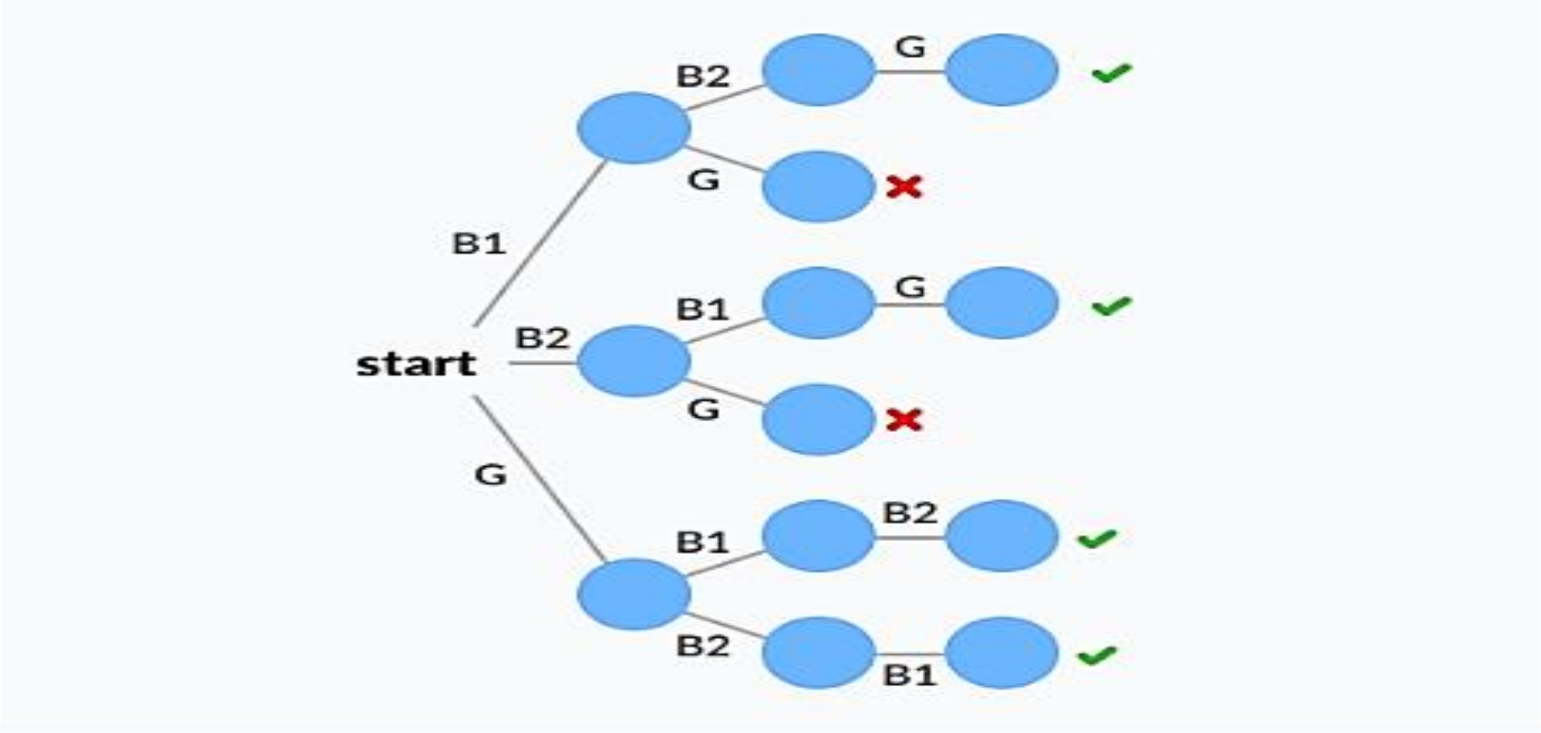
B2 G B1

B1 G B2

G B1 B2

B2 B1 G

G B2 B1

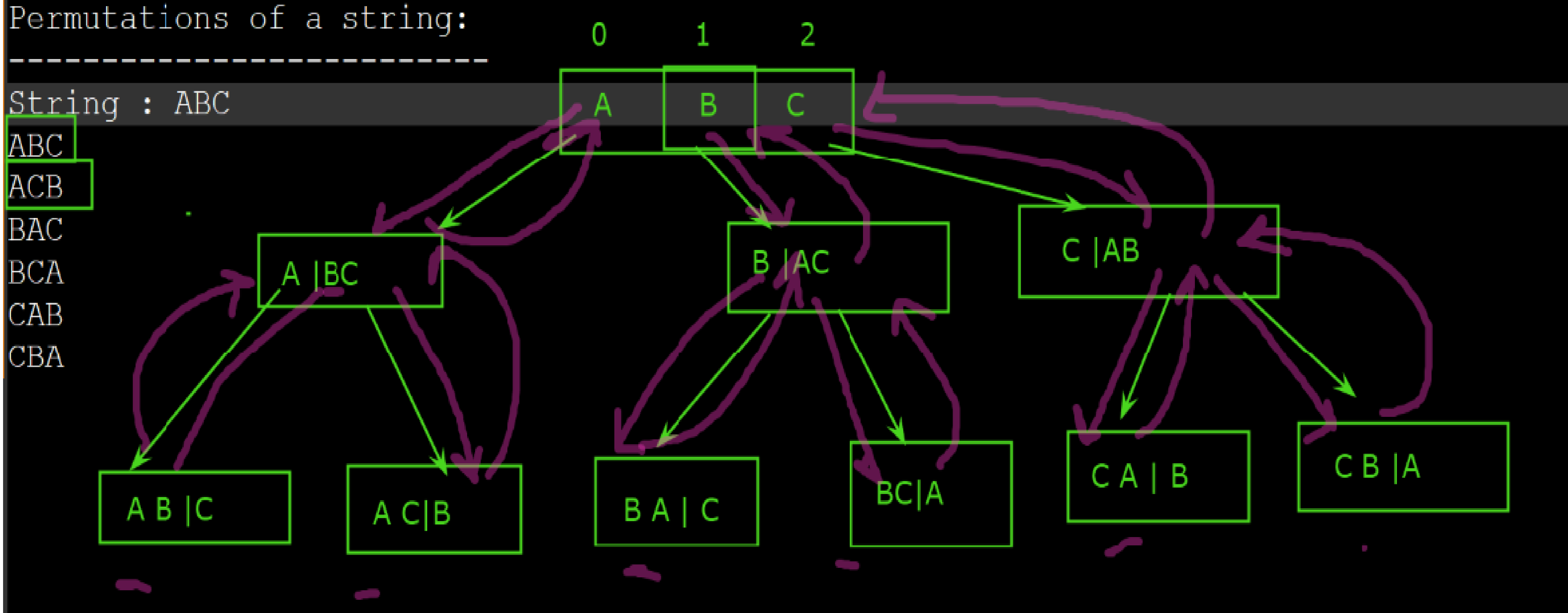


State tree with all the solutions

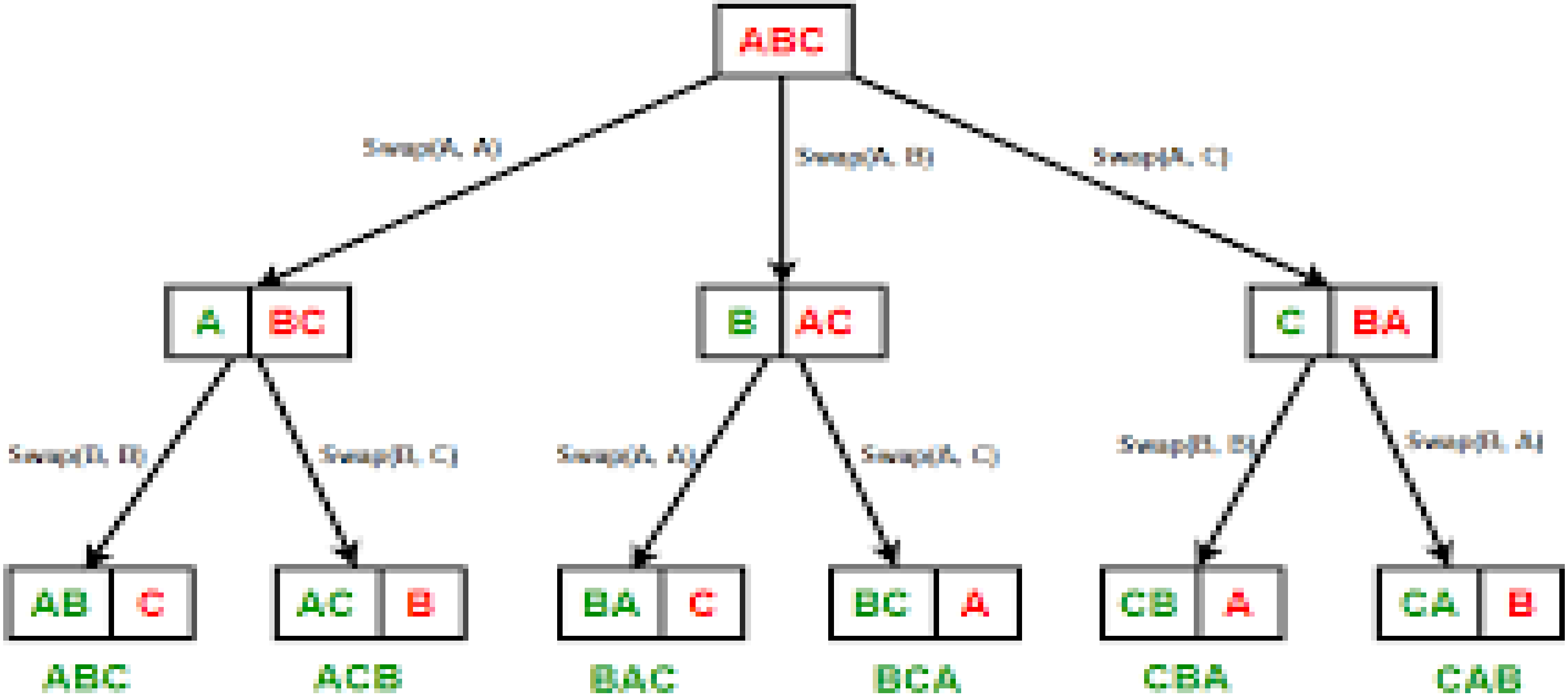
- **Backtracking Algorithm Applications**
- To find all Hamiltonian Paths present in a graph.
- To solve the N Queen problem.
- Maze solving problem.
- The Knight's tour problem.

Complexity Analysis of Backtracking

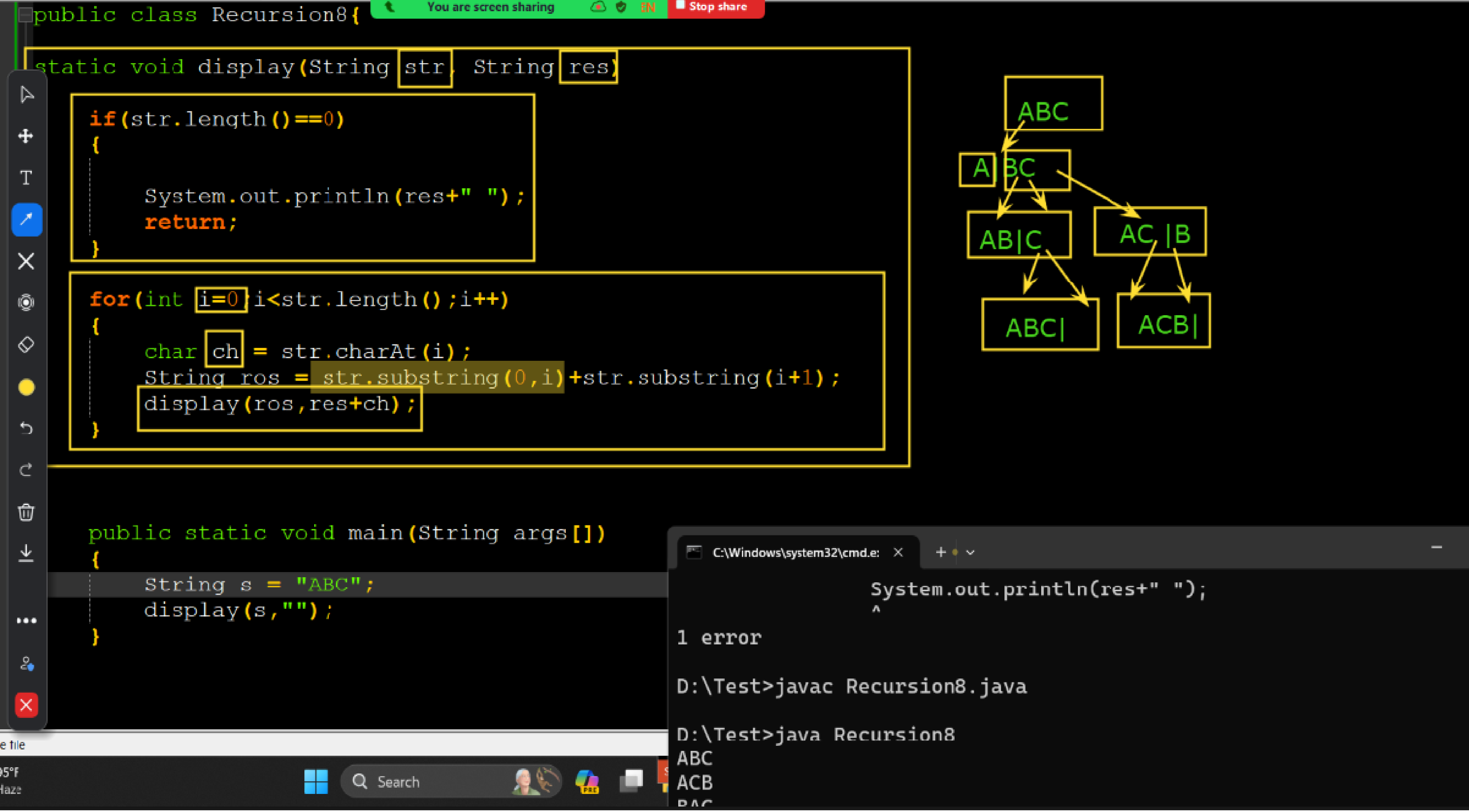
- Since backtracking algorithm is purely brute force therefore in terms of time complexity, it performs very poorly. Generally backtracking can be seen having below mentioned time complexities:
- Exponential (O(K^N))
- Factorial (O(N!))



```
Permutations of a string:
--------------------------
String : ABC
ABC
ACB
BAC
BCA
CAB
CBA
```



Recursion Tree for string "ABC"



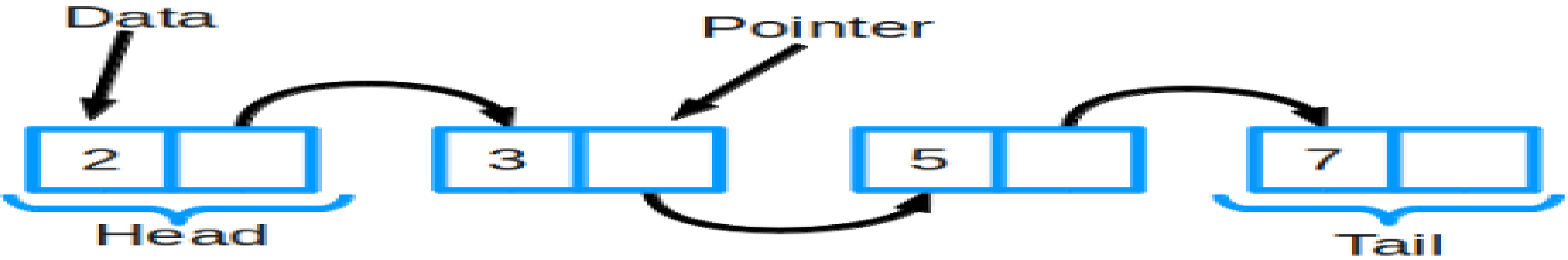
```
public class Recursion8{
static void display(String str, String res)
    if(str.length()==0)
    {

        System.out.println(res+" ");
        return;
    }
    for(int i=0;i<str.length();i++)
    {
        char ch = str.charAt(i);
        String ros = str.substring(0,i)+str.substring(i+1);
        display(ros,res+ch);
    }


    public static void main(String args[])
    {
        String s = "ABC";
        display(s,"");
    }
```

```
                System.out.println(res+" ");
                ^
1 error

D:\Test>javac Recursion8.java

D:\Test>java Recursion8
ABC
ACB
BAC
```

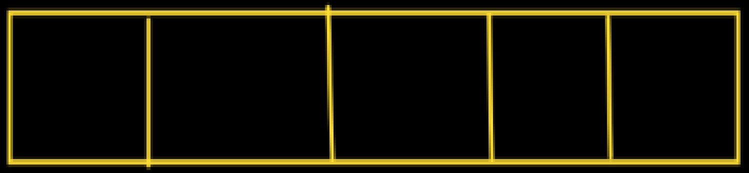
Branch and Bound

- **Branch and Bound Algorithm**:
  - Method used in combinatorial optimization problems to find the best solution.
  - Divides the problem into smaller subproblems (branches) and eliminates certain branches based on bounds on the optimal solution.
  - Continues until the best solution is found or all branches have been explored.
- **Exploration of Search Space**:
  - Identifies possible candidates for solutions step-by-step through the exploration of the entire search space.
- **Optimality and NP-Hard Problems**:
  - Provides an optimal solution to NP-Hard problems by exploring the entire search space.
- **Search Techniques**:
  - LC search (Least Cost Search): Uses a heuristic cost function to compute bound values at each node, selecting nodes with the least cost as next E-nodes.
  - BFS (Breadth First Search): Maintains live nodes in a FIFO order (queue) and searches nodes in FIFO order.
  - DFS (Depth First Search): Maintains live nodes in a LIFO order (stack) and searches nodes in LIFO order.
- **Applications**:
  - Commonly used in problems like the traveling salesman and job scheduling.
- **Efficiency**:
  - Offers an efficient way to explore large solution spaces by eliminating unpromising branches

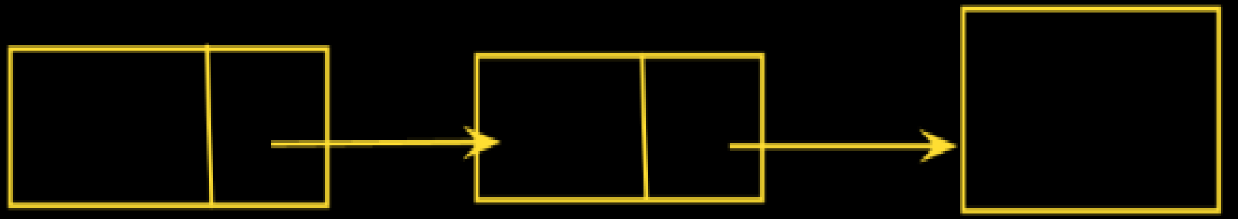
Linked list



```
Linked List:
--------------
```



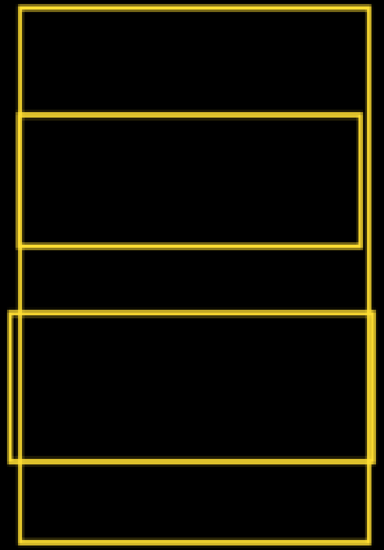
Array

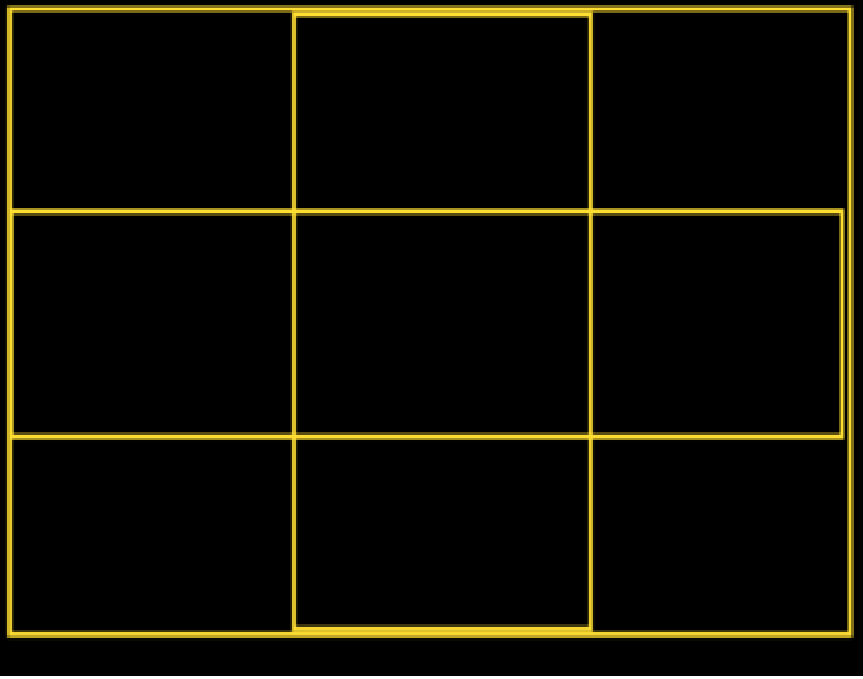


Linked List

Array-> List
Stack->List
Queue-> List
Tree-> DLL
Graph-> List

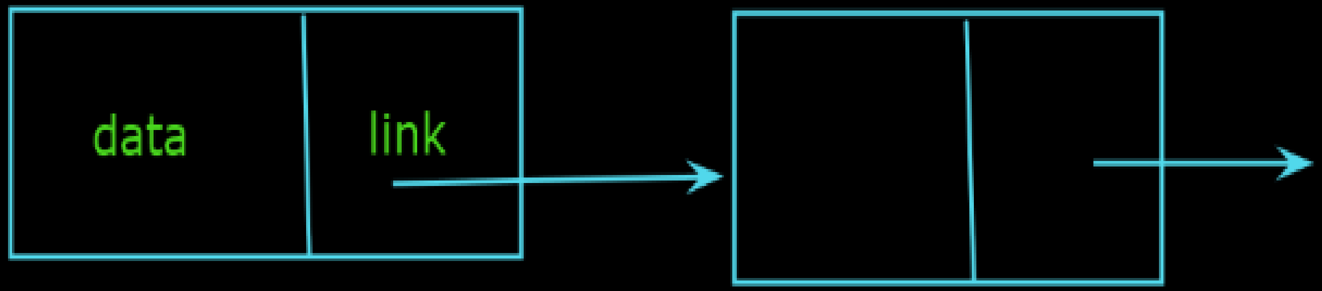
Dyanmic
Implementation





Linked List:

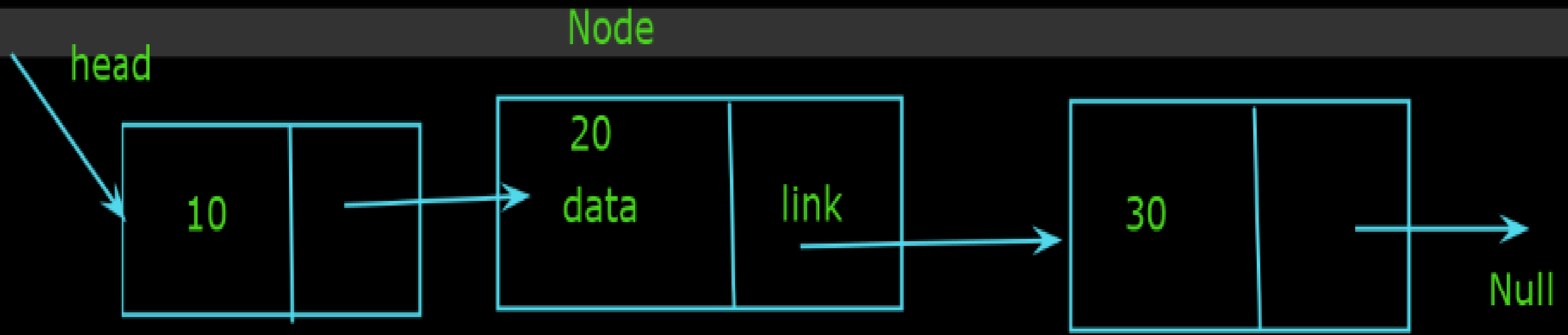
Node



Data :value (int,float,char,String)

link:connection (reference= address(next node ka address store karta hai .)

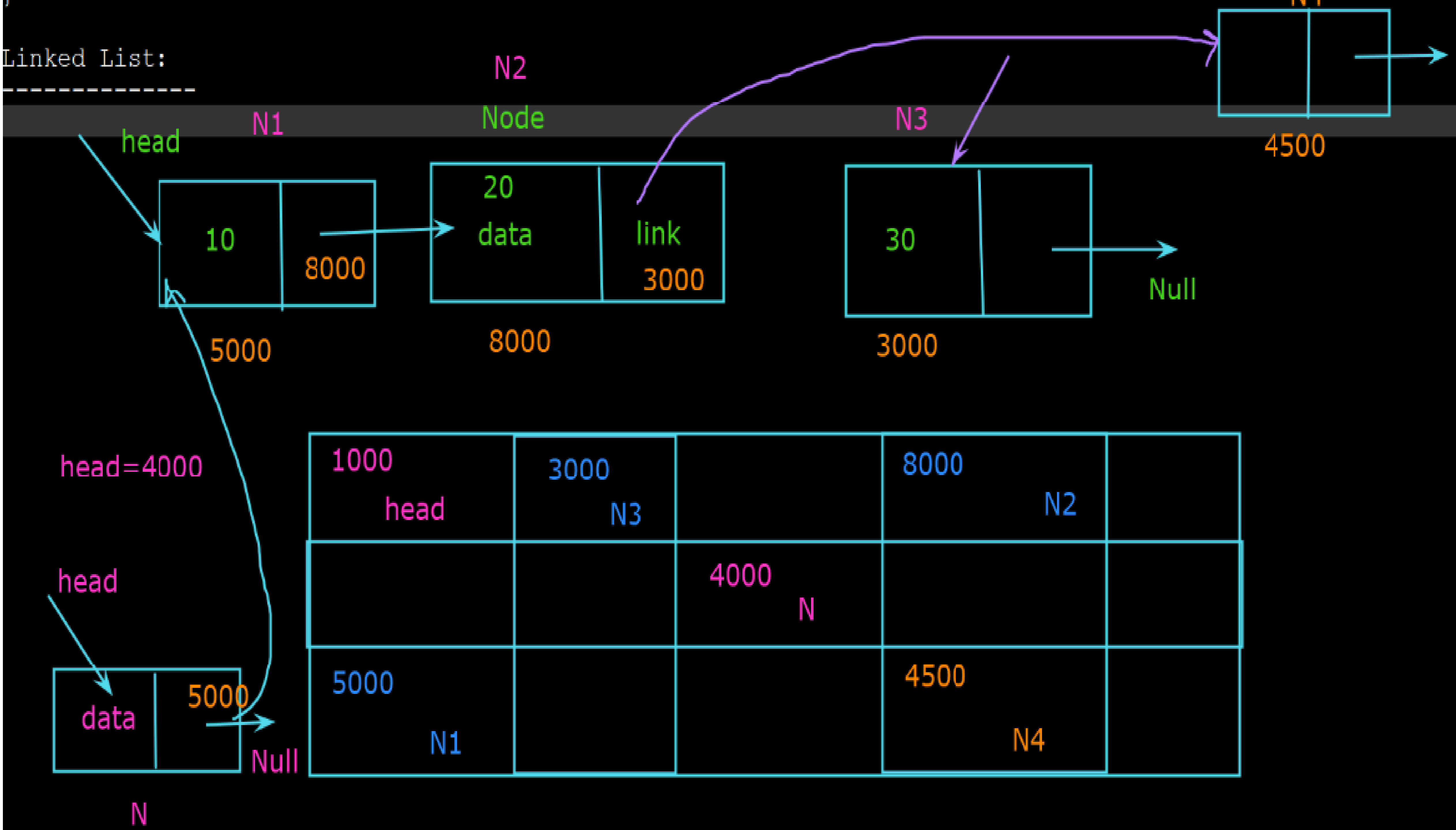
Linked List:



Data :value (int,float,char,String)

link:connection (reference= address(next node ka address store karta hai .)

```
Linked List:
--------------
```



1000 head	3000 N3		8000 N2	
		4000 N		
5000 N1			4500 N4	

Linked List

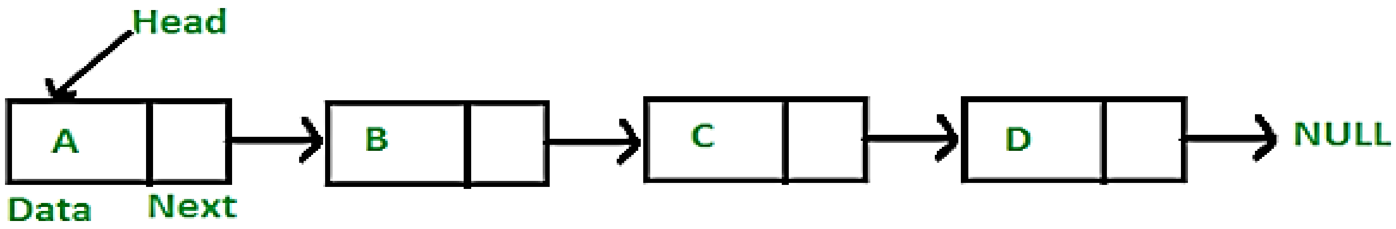
- A linked list is a sequence of data structures, which are connected together via links.

- Linked List is a sequence of links which contains items.
- Each link contains a connection to another link.
- Linked list is the second most-used data structure after array.
- Following are the important terms to understand the concept of Linked List.

1. **Link** − Each link of a linked list can store a data called an **element**.
2. **Next** − Each link of a linked list contains a link to the next link called **Next.**
3. **LinkedList** − A Linked List contains the **connection link** to the first link called **First**.

Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.



- As per the above illustration, following are the important points to be considered.

1. Linked List contains a **link element** called **first.**
2. Each link carries a **data field(s)** and a **link field** called **next.**
3. Each link is **linked with its next link** using its **next link.**
4. **Last link carries a link as null** to mark the end of the list.

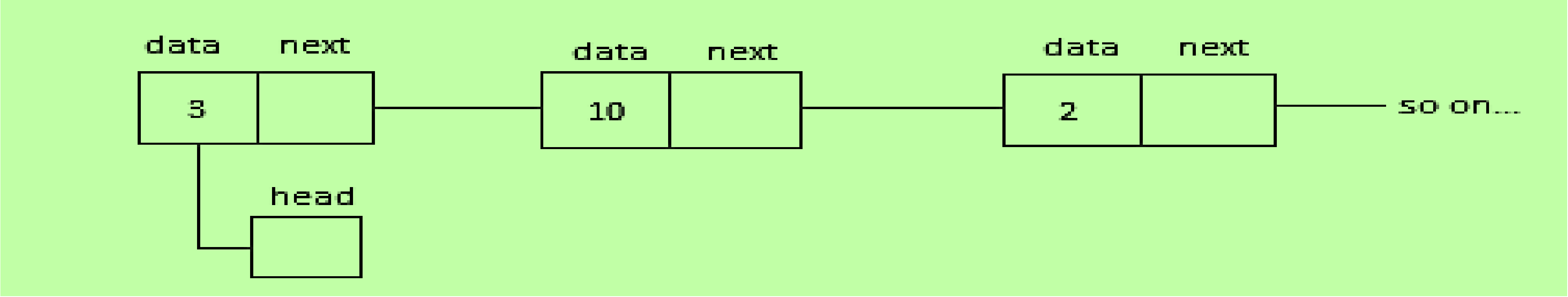
Types of Linked List

- **Following are the various types of linked list.**
  1. **Simple Linked List** − Item navigation is forward only.

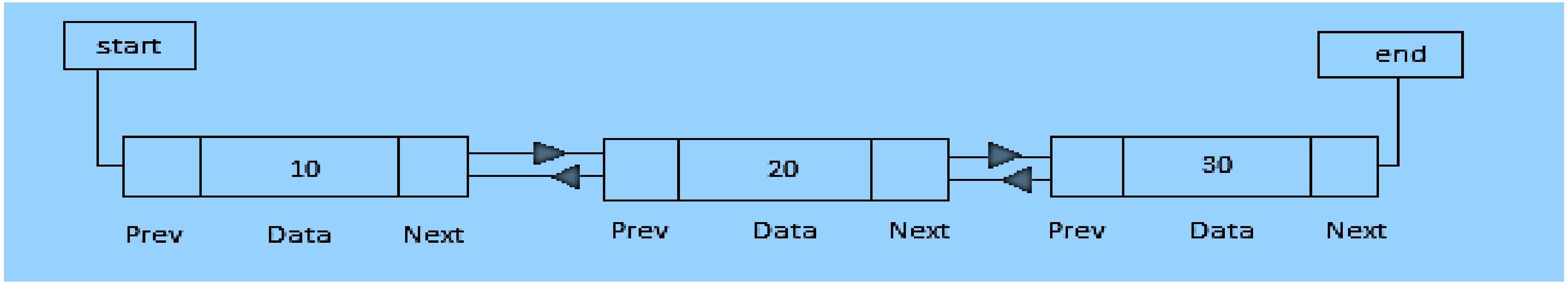
  2. **Doubly Linked List** − Items can be navigated forward and backward.

  3. **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

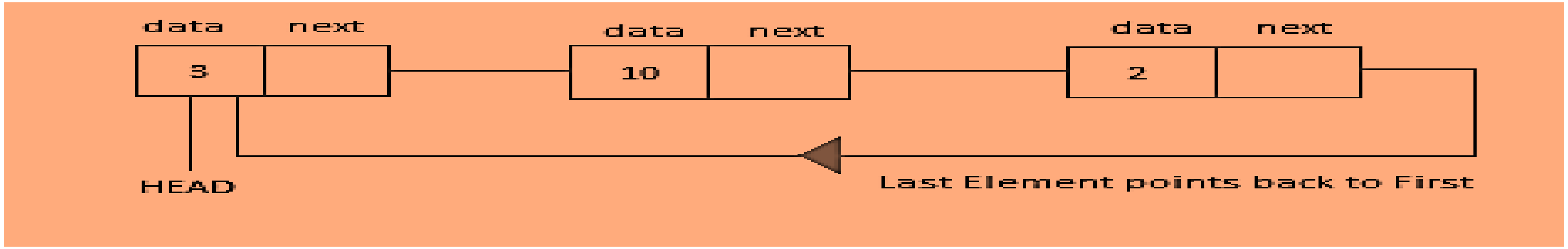
- **Simple Linked List**



- **Doubly Linked List**

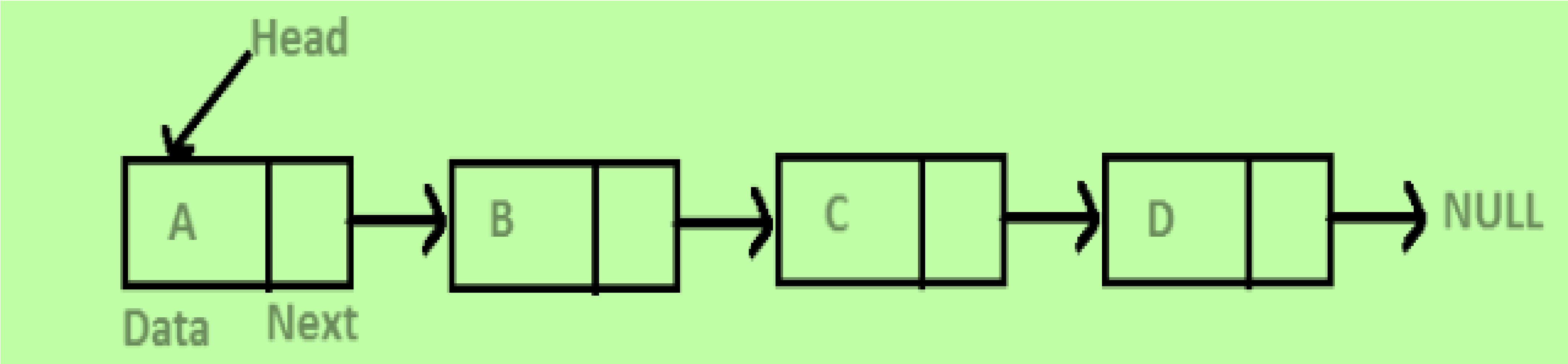


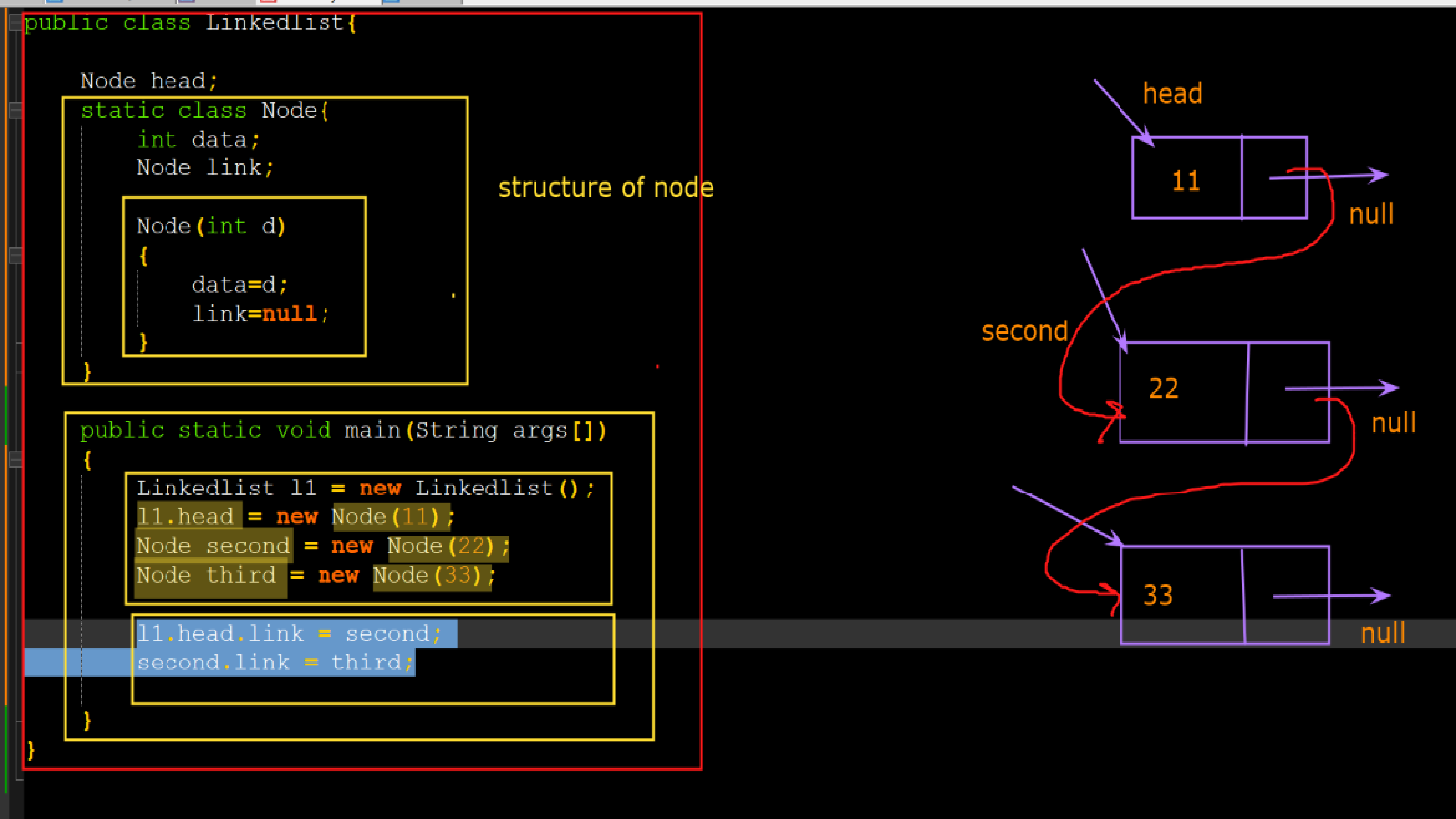
- **Circular Linked List**



Singly Linked List

- Singly Linked Operations: Insert, Delete, Traverse, search, Sort, Merge





```
public class Linkedlist{

    Node head;
    static class Node{
        int data;
        Node link;

        Node(int d)
        {
            data=d;
            link=null;
        }
    }

    public static void main(String args[])
    {
        Linkedlist l1 = new Linkedlist();
        l1.head = new Node(11);
        Node second = new Node(22);
        Node third = new Node(33);

        l1.head.link = second;
        second.link = third;

    }
}
```

structure of node

Advantages of Linked Lists

1. They are a dynamic in nature which allocates the memory when required.
2. Insertion and deletion operations can be easily implemented.
3. Stacks and queues can be easily executed.
4. Linked List reduces the access time.

Disadvantages of Linked Lists

1. The memory is wasted as pointers require extra memory for storage.
2. No element can be accessed randomly; it has to access each node sequentially.
3. Reverse Traversing is difficult in linked list.

Applications of Linked Lists

1. **Linked lists are used to implement stacks, queues, graphs, etc.**
2. **Linked lists let you insert elements at the beginning and end of the list.**
3. **In Linked Lists we don't need to know the size in advance.**

Basic Operations

- **Following are the basic operations supported by a list.**
  1. **Insertion** − Adds an element at the beginning of the list.
  2. **Deletion** − Deletes an element at the beginning of the list.
  3. **Display** − Displays the complete list.
  4. **Search** − Searches an element using the given key.
  5. **Delete** − Deletes an element using the given key.

Problem Statement 1 : Delete a Linked List node at a given position.

Given a singly linked list and a position, delete a linked list node at the given position.

Example:

Input: position = 1, Linked List = 18->12->13->11->17
Output: Linked List = 18->13->11->17

Input: position = 0, Linked List = 98->24->32->17->74
Output: Linked List = 24->32->17->74

Thanks