Introduction to Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It consists of two main components:

- **Base Case:** The simplest scenario where the function returns a result without further recursion.
- **Recursive Case:** The function calls itself with a modified input, moving closer to the base case.

Why Use Recursion?

- Simplifies code for problems with repetitive substructures (e.g., trees, graphs, mathematical sequences).
- Naturally models divide-and-conquer and backtracking algorithms.
- Often more elegant (but sometimes less efficient) than iterative solutions.

Key Concepts in Recursion

(A) Factorial: Iterative vs. Recursive

Problem: Compute $n!$ (product of all positive integers $\leq n$).

Iterative Approach (Loop-Based):

```
int factorial(int n) {
    int result = 1;
    for(int i = 1; i <= n; i++) result *= i;
    return result;
}
```

Pros:

- Efficient (O(n) time, O(1) space).

Cons:

- Less intuitive for mathematical definitions.

Recursive Approach:

```
int factorial(int n) {
    if (n <= 1) return 1; // base case
    return n * factorial(n - 1); // recursive call
}
```

Recursive Breakdown

$\text{factorial}(5) \to 5 \times \text{factorial}(4)$
$\to 5 \times 4 \times \text{factorial}(3)$
$\to 5 \times 4 \times 3 \times \text{factorial}(2)$
$\to 5 \times 4 \times 3 \times 2 \times \text{factorial}(1)$
$\to 5 \times 4 \times 3 \times 2 \times 1 = 120$

Pros:

- Mathematically elegant.

Cons:

- Higher memory usage (O(n) stack space).

(B) Fibonacci Sequence: Iterative vs. Recursive

Problem: Compute the nth Fibonacci number, where:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2) \quad \text{for } n \geq 2$$

Iterative Approach (Loop-Based)

```
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    int a = 0, b = 1, result;
    for (int i = 2; i <= n; i++) {
        result = a + b;
        a = b;
        b = result;
    }
    return b;
}
```

Pros:

- Time-efficient: $O(n)$
- Space-efficient: $O(1)$

Cons:

- Less aligned with the mathematical recurrence

Recursive Approach

```
int fibonacci(int n) {
    if (n <= 1) return n;  // base case
    return fibonacci(n - 1) + fibonacci(n - 2);  // recursive calls
}
```

Recursive Breakdown

$\text{fibonacci}(5) \rightarrow \text{fibonacci}(4) + \text{fibonacci}(3)$
$\rightarrow (\text{fibonacci}(3) + \text{fibonacci}(2)) + (\text{fibonacci}(2) + \text{fibonacci}(1))$
$\rightarrow ((\text{fibonacci}(2) + \text{fibonacci}(1)) + (\text{fibonacci}(1) + \text{fibonacci}(0))) + ((\text{fibonacci}(1) + \text{fibonacci}(0)) + 1)$
$\rightarrow ((1 + 1) + (1 + 0)) + ((1 + 0) + 1)$
$\rightarrow 2 + 1 + 1 + 1 = 5$

Pros:

- Clear and matches mathematical definition

Cons:

- Inefficient due to repeated calculations (exponential time)
- High stack usage: $O(n)$

Linked List Traversal

Theory: Why Use Recursion for Linked Lists?

Linked lists are linear data structures where each node points to the next. Recursion provides an elegant way to traverse them because:

- Each node's `next` pointer naturally leads to a smaller subproblem (the rest of the list).
- The base case (`node == NULL`) handles the end of the list.
- Recursive reversal mirrors the divide-and-conquer approach by breaking the problem into smaller steps.

Iterative Linked List Traversal & Reversal

```
void printListIterative(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
}
Node* reverseListIterative(Node* head) {
```

```
    Node* prev = nullptr;
    Node* current = head;
    while (current != nullptr) {
        Node* next = current->next;
        current->next = prev;
        prev = current;
        current = next; }
    return prev; } // New head after reversal
```

Recursive Linked List Traversal & Reversal

Recursion leverages the call stack to implicitly track the state as we move through the list. This leads to cleaner code but can be less memory-efficient for very large lists.

```
void printListRecursive(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    printListRecursive(node->next);
}

Node* reverseListRecursive(Node* head, Node* prev = nullptr) {
    if (head == nullptr) return prev;
    Node* next = head->next;
    head->next = prev;
    return reverseListRecursive(next, head);
}
```

Tree Traversals

Introduction to Trees

A tree is a hierarchical, non-linear data structure consisting of nodes connected by edges. Unlike arrays or linked lists, trees represent data in a parent-child relationship.

- The top node is called the **root**.
- Each node can have zero or more **child** nodes.
- Nodes with no children are called **leaves**.
- There is only one path between two nodes (no cycles).

Key Properties

- **Recursive Nature:** A tree is composed of smaller trees (subtrees), making it ideal for recursive algorithms.

- **Use Cases:** Hierarchies, file systems, compilers, AI decision trees, and more.

What is a Binary Search Tree (BST)?

A Binary Search Tree (BST) is a special type of binary tree where:

- Each node has at most two children: `left` and `right`.
- `Left child < Parent < Right child` — this is the BST property.

This structure allows efficient searching, insertion, and deletion.

Why Recursion Works Well in BSTs

- A BST is made of smaller BSTs (its subtrees).
- Recursive functions can naturally navigate and operate on BSTs by breaking the problem into similar subproblems.

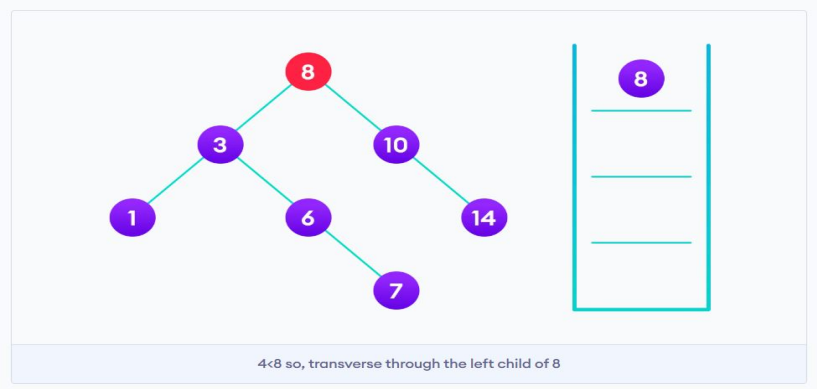
Example: Insertion in BST Using Recursion

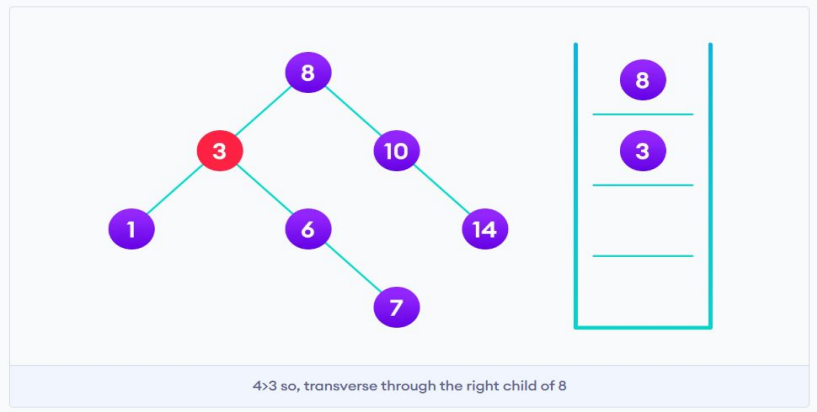
Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root. We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

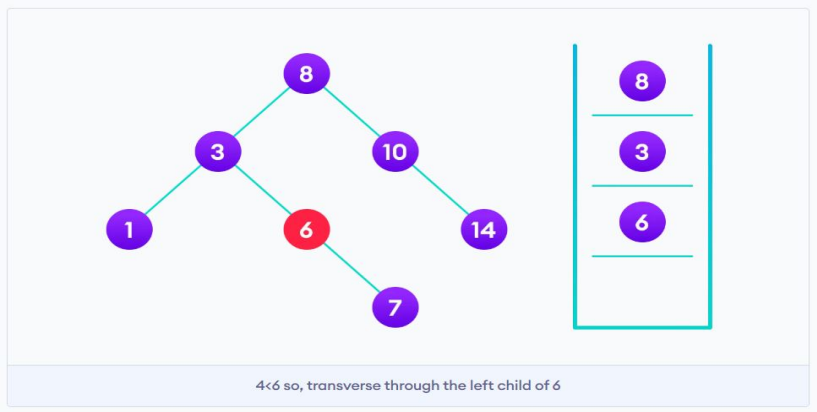
Algorithm:

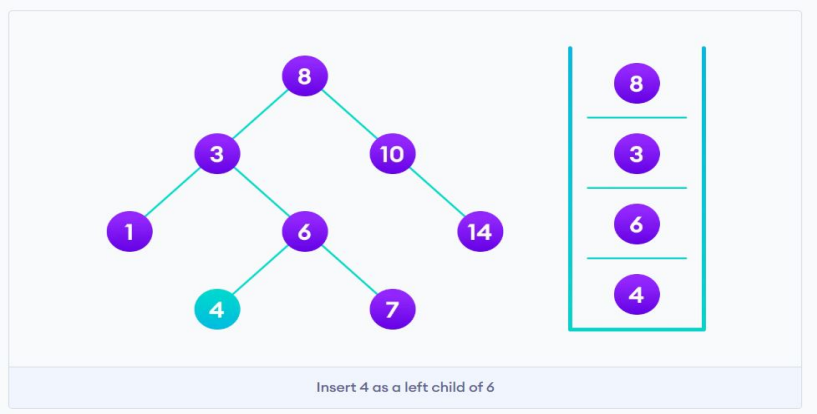
```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

The algorithm isn't as simple as it looks. Let's try to visualize how we add a number (4) to an existing BST.

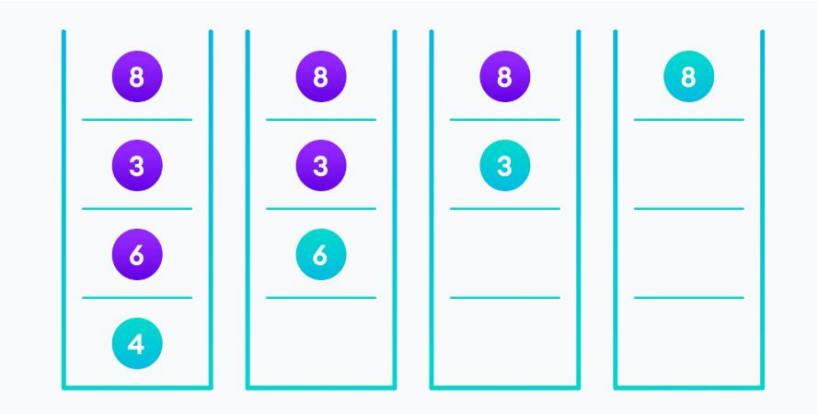








We have attached the node, but we still have to exit from the function without causing any damage to the rest of the tree. This is where the `return node;` at the end comes in handy. In the case of `NULL`, the newly created node is returned and attached to the parent node; otherwise, the same node is returned without any change as we go up, until we return to the root.
This ensures that as we move back up the tree, the other node connections remain unchanged.

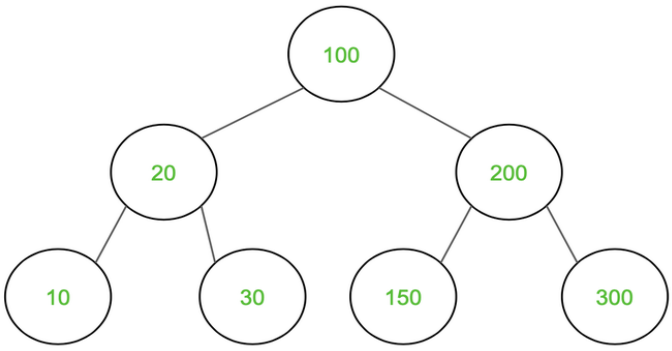


This image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

Tree Traversal Techniques (Using Recursion)

Traversal Type	Visit Order	Recursive Steps
Pre-order	Root → Left → Right	Visit current → Recurse left → Recurse right
In-order	Left → Root → Right	Recurse left → Visit current → Recurse right
Post-order	Left → Right → Root	Recurse left → Recurse right → Visit current

Example: BST Traversals



Binary Search Tree

Output

```
Preorder Traversal: 100 20 10 30 200 150 300
Inorder Traversal: 10 20 30 100 150 200 300
Postorder Traversal: 10 30 20 150 300 200 100
```

Binary Search

Theory: Divide-and-Conquer via Recursion

Binary Search is a fundamental algorithm used to efficiently find a target value within a **sorted array**. It operates by repeatedly dividing the search interval in half, making it ideal for recursive implementation.

Recursive Binary Search

Base Case: The search space is empty (`low > high`)
Recursive Step:

- Compare target with the middle element.
- If target $<$ mid $\rightarrow$ search the left half.
- If target $>$ mid $\rightarrow$ search the right half.

Why Recursive Binary Search?

- **Easy to Understand:** Naturally follows the logic of binary search.
- **Cleaner Code:** Eliminates loop constructs and focuses on logic.
- **Memory Usage:** Uses additional space on the call stack (O(log n)), unlike the iterative version.

Example: Search in Array

Input: `arr = [2, 5, 8, 12, 16, 23]`, `target = 12`

- Mid = 8 (index 2) $\quad 12 > 8 \rightarrow$ search right half.
- Mid = 16 (index 4) $\quad 12 < 16 \rightarrow$ search left half.
- Mid = 12 (index 3) $\quad$ found!

Implementations

Iterative Binary Search

```
int binarySearchIterative(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Recursive Binary Search

```
int binarySearchRecursive(const vector<int>& arr, int target, int left, int right) {
    if (left > right) return -1;  // Base case
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] < target)
        return binarySearchRecursive(arr, target, mid + 1, right);
    else
        return binarySearchRecursive(arr, target, left, mid - 1);
}
```

Comparison: Iterative vs Recursive

Feature	Iterative	Recursive
Time Complexity	O(log n)	O(log n)
Space Complexity	O(1)	O(log n) (stack)
Ease of Understanding	Moderate	High
Code Readability	Medium	High

Binary Search in Binary Search Trees (BST)

What is a BST?

A **Binary Search Tree** (BST) is a binary tree where:

- All values in the left subtree are less than the current node.
- All values in the right subtree are greater than the current node.

Why Use Recursion in BST Search?

- **Natural Fit:** Each node points to left and right — just like recursive subproblems.
- **Clean and Intuitive:** Simple logic — search left or search right.
- **Logical Stopping:** Stops when value is found or node is null.

How Recursive Search Works in a BST

Base Case: `node == nullptr` → value not found.
Recursive Cases:

- If `target < node->val` → search left subtree.
- If `target > node->val` → search right subtree.
- If `target == node->val` → value found.

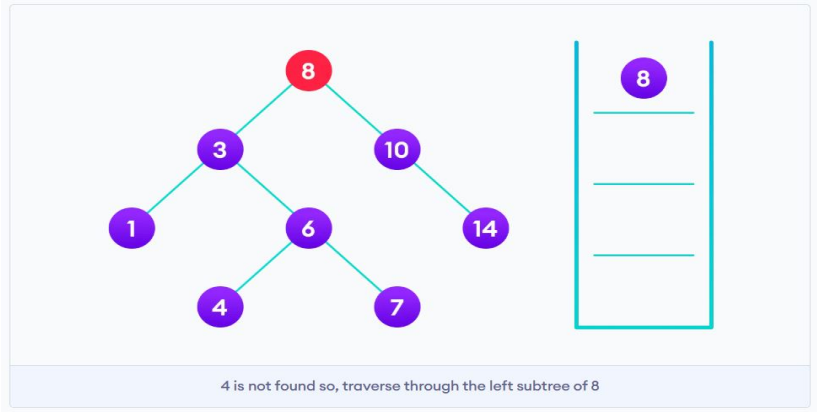
BST Search Operation

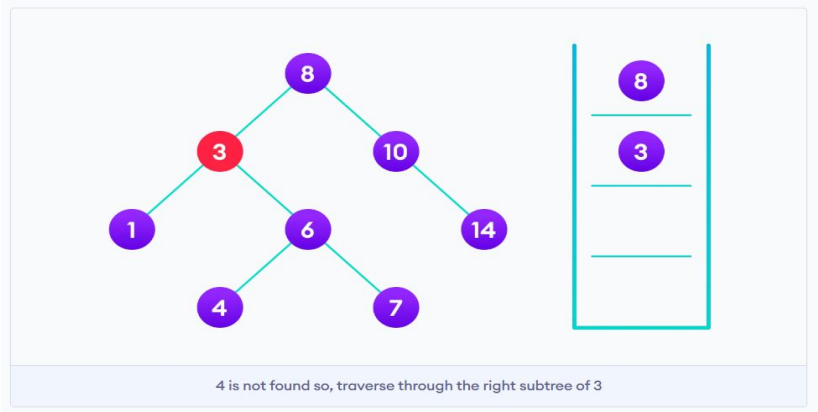
The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

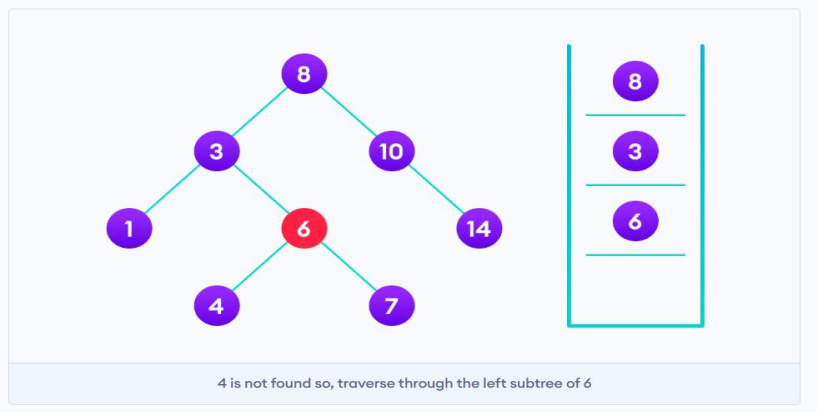
```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

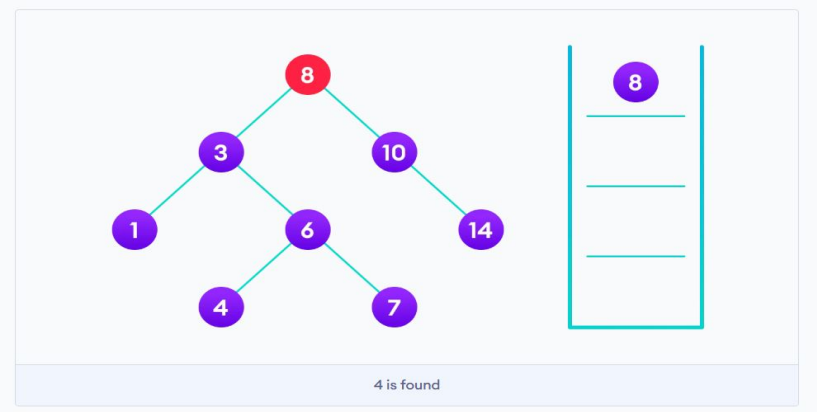
Example BST Structure

Let's try to visualize how we search a number (4) in an existing BST.

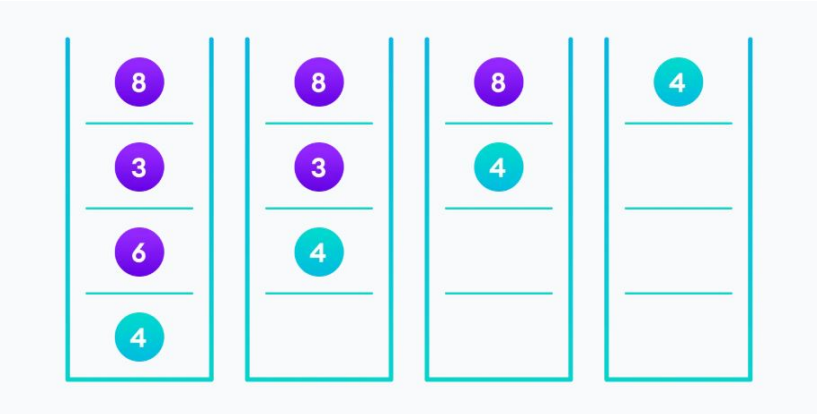








If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.
If you might have noticed, we have called return search(struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned. If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Array vs BST Binary Search

Feature	Array Binary Search	BST Binary Search
Data Structure	Sorted Array	Binary Search Tree
Time Complexity	O(log n)	O(h), where h = tree height
Requires Sorting	Yes	No
Implementation	Index-based	Pointer-based

Backtracking Algorithms

Backtracking is a general algorithmic technique used to solve problems by incrementally building a solution and abandoning it (or *"backtracking"*) as soon as it is determined that the solution is not viable. It is commonly used in puzzles, combinatorial optimization, and constraint satisfaction problems.

How Backtracking Works

- **Choice:** At each step, make a decision (choose an option).
- **Explore:** Recursively move forward with that decision.
- **Check Constraints:** If the decision is invalid, backtrack.
- **Backtrack:** Undo the last step and try another option.

The key idea is to construct a partial solution and, if it violates constraints, backtrack to try a different path.

Example: The N-Queens Problem

Place N queens on an $N \times N$ chessboard such that no two queens attack each other (i.e., no two queens share a row, column, or diagonal).

Backtracking Algorithm for N-Queens

1. Try placing a queen in a row.
2. Move to the next row and try a safe column.
3. If a conflict arises, backtrack and try the next column.

C++ Code Example (4-Queens)

```
#include <iostream>
#include <vector>
using namespace std;

bool isSafe(int row, int col, vector<int>& board, int n) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col || board[i] - i == col - row || board[i] + i == col + row)
```

```
            return false;
    }
    return true;
}

bool solveNQueens(int row, vector<int>& board, int n) {
    if (row == n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i] == j)
                    cout << "Q ";
                else
                    cout << ". ";
            }
            cout << endl;
        }
        cout << endl;
        return true;
    }

    for (int col = 0; col < n; col++) {
        if (isSafe(row, col, board, n)) {
            board[row] = col;
            if (solveNQueens(row + 1, board, n)) return true;
            board[row] = -1;
        } }
    return false;
}
int main() {
    int n = 4;
    vector<int> board(n, -1);
    solveNQueens(0, board, n);
    return 0; }
```

Explanation

- `isSafe(row, col, board, n)` checks if placing a queen at (row, col) violates any constraints.
- `solveNQueens(row, board, n)` places a queen row by row. If no column is safe in a row, it backtracks.

Output for $n = 4$

```
Q . . .
. . Q .
. . . Q
. Q . .

. . . Q
Q . . .
. Q . .
. . Q .
```

The algorithm finds all valid configurations of queens such that no two threaten each other.

Lab exercises

Exercise 1

Elimination Game: You have a list `arr` of all integers in the range $[1, n]$ sorted in a strictly increasing order. Apply the following algorithm on `arr`:

1. Starting from left to right, remove the first number and every other number afterward until you reach the end of the list.
2. Repeat the previous step again, but this time from right to left, remove the rightmost number and every other number from the remaining numbers.
3. Keep repeating the steps again, alternating left to right and right to left, until a single number remains.

Given the integer n, return the last number that remains in `arr`.

Example:

Input: `n = 9`

Output: `6`

Explanation

`arr` = [1, 2, 3, 4, 5, 6, 7, 8, 9]
$\rightarrow [2, 4, 6, 8]$
$\rightarrow [2, 6]$
$\rightarrow [6]$

Constraints

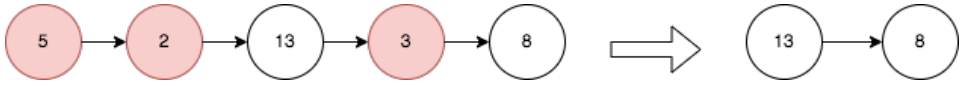
- $1 \leq n \leq 10^9$

Exercise 2

Remove Nodes From Linked List using Recursion: You are given the complete implementation of a singly linked list. Your task is to **implement only the `remove` function**. This function should **recursively** remove every node that has a node with a greater value anywhere to the right side of it.

You **must** solve this problem using **recursion**. The linked list file is already provided and attached to this question. Only implement the missing recursive `remove` function in the given file.

Example:



Input: `head = [5, 2, 13, 3, 8]`

Output: `[13, 8]`

Explanation: The nodes that should be removed are 5, 2, and 3.

- Node 13 is to the right of node 5.
- Node 13 is to the right of node 2.
- Node 8 is to the right of node 3.

Exercise 3

Insert a Node into a BST using Recursion: You are given the complete implementation of a Binary Search Tree (BST). Your task is to **implement only the `insertIntoBST` function**.

This function should recursively insert a new node with a given value into the BST.

A BST is a binary tree where for every node:

- The value of the left child is less than the value of the parent node.
- The value of the right child is greater than the value of the parent node.

You **must** solve this problem using **recursion**.

The BST file is already provided and attached to this question. Only implement the missing recursive `insertIntoBST` function in the given file.

Function Signature

```
TreeNode* insertIntoBST(TreeNode* root, int val);
```

Input

- A `TreeNode* root` pointing to the root of the binary search tree.
- An `int val` representing the value of the node to be inserted into the BST.

Output

- Return the root of the binary search tree after the insertion.

Constraints

- The tree will contain **unique values** (no duplicates).
- You can assume that the input tree is already a valid BST.

Exercise 4

Binary Search in a BST: You are given the complete implementation of a Binary Search Tree (BST). Your task is to **implement only the `searchInBST` function**.

This function should recursively determine whether a given target value exists in the BST.

A BST is a binary tree where:

- The value of the left child is less than the value of the parent node.
- The value of the right child is greater than the value of the parent node.

You **must** solve this problem using **recursion**.

The BST file is already provided and attached to this question. Only implement the missing recursive `searchInBST` function in the given file.

Function Signature

```
bool searchInBST(TreeNode* root, int val);
```

Input

- A `TreeNode* root` pointing to the root of the binary search tree.
- An integer `val` representing the target value to be searched in the BST.

Output

- Return `true` if the target value exists in the BST, and `false` otherwise.

Constraints

- The tree may have duplicate values. If the tree contains duplicates, only one instance will be present.
- The tree will be a valid binary search tree.

Exercise 5

Fibonacci Word: A Fibonacci Word is a specific sequence of binary digits (or symbols from any two-letter alphabet). The Fibonacci Word is formed by repeated concatenation in the same way that the Fibonacci numbers are formed by repeated addition.

Create a recursive function that takes a number n as an argument and returns the first n elements of the Fibonacci Word sequence. If $n < 2$, the function must return "invalid".

Examples

- `fiboWord(1)` → `"invalid"`
- `fiboWord(3)` → `"b, a, ab"`
- `fiboWord(7)` → `"b, a, ab, aba, abaab, abaababa, abaababaabaab"`