Day 2

1.Majority Elements

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

int majorityElement(vector<int>& nums) {
    int candidate = nums[0];  // Initial candidate
    int count = 1;  // Initial count for the first element

    // Traverse through the array to find the candidate for majority element
    for (int i = 1; i < nums.size(); ++i) {
        if (count == 0) {
            candidate = nums[i];
            count = 1;
        } else if (nums[i] == candidate) {
            count++;
        } else {
            count--;
        }
    }

    // Return the candidate (majority element)
    return candidate;
}

int main() {
    vector<int> nums = {3, 2, 3};
    cout << "The majority element is: " << majorityElement(nums) << endl;
    return 0;
}
```

```
The majority element is: 3


=== Code Execution Successful ===
```

2. Single Number

```
#include <iostream>
#include <vector>
```

```
using namespace std;

int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;  // XOR the current number with result
    }
    return result;  // result will hold the number that appears only once
}

int main() {
    vector<int> nums = {4, 1, 2, 1, 2};
    cout << "The single number is: " << singleNumber(nums) << endl;
    return 0;
}
```

Output

```
The single number is: 4


=== Code Execution Successful ===
```

3.Linked List Cycle

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

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

bool hasCycle(ListNode *head) {
    if (head == NULL) {
        return false;
    }

    ListNode *slow = head;      // Tortoise (moves 1 step at a time)
    ListNode *fast = head;      // Hare (moves 2 steps at a time)

    while (fast != NULL && fast->next != NULL) {
```

```
        slow = slow->next;          // Move slow by 1 step
        fast = fast->next->next;    // Move fast by 2 steps

        if (slow == fast) {         // Cycle detected
            return true;
        }
    }

    return false;  // No cycle
}

int main() {
    // Create a test linked list: 1 -> 2 -> 3 -> 4 -> 5
    ListNode *head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    // Introduce a cycle for testing: 5 -> 2 (cycle)
    head->next->next->next->next->next = head->next;  // 5 -> 2 creates a cycle

    // Test for cycle
    if (hasCycle(head)) {
        cout << "The linked list has a cycle." << endl;
    } else {
        cout << "The linked list does not have a cycle." << endl;
    }

    return 0;
}
```

Output

```
The linked list has a cycle.


=== Code Execution Successful ===
```

4. **Pascal's Triangle**

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

vector<vector<int>> generate(int numRows) {
    vector<vector<int>> triangle;  // This will hold the entire triangle
```

```
    // Generate each row
    for (int i = 0; i < numRows; ++i) {
        vector<int> row(i + 1, 1);  // Initialize a row with `i + 1` elements, all set to 1

        // Fill in the values for the interior elements
        for (int j = 1; j < i; ++j) {
            row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];  // Sum of the two elements above
        }

        triangle.push_back(row);  // Add the row to the triangle
    }

    return triangle;  // Return the generated Pascal's triangle
}

void printTriangle(const vector<vector<int>>& triangle) {
    for (const auto& row : triangle) {
        for (int num : row) {
            cout << num << " ";
        }
        cout << endl;
    }
}

int main() {
    int numRows = 5;  // Example input
    vector<vector<int>> triangle = generate(numRows);  // Generate the triangle
    printTriangle(triangle);  // Print the triangle

    return 0;
}
```

Output

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1


=== Code Execution Successful ===
```

5. Remove Element

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

int removeDuplicates(vector<int>& nums) {
    if (nums.empty()) {
        return 0;  // No elements in the array
    }

    int i = 1;  // Pointer to place the next unique element

    // Traverse the array starting from the second element
    for (int j = 1; j < nums.size(); ++j) {
        // If the current element is different from the previous one
        if (nums[j] != nums[j - 1]) {
            nums[i] = nums[j];  // Place the unique element at index i
            i++;  // Increment the index for the next unique element
        }
    }

    return i;  // The number of unique elements is i
}

void printArray(const vector<int>& nums, int k) {
    for (int i = 0; i < k; ++i) {
        cout << nums[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> nums = {1, 1, 2, 2, 3, 3, 4};

    int k = removeDuplicates(nums);  // Remove duplicates and get the number of unique
elements
    cout << "Number of unique elements: " << k << endl;

    // Print the modified array with unique elements
    cout << "Array after removing duplicates: ";
    printArray(nums, k);

    return 0;
```

}

Output

```
Number of unique elements: 4
Array after removing duplicates: 1 2 3 4


=== Code Execution Successful ===
```

6. . Insert Greatest Common Divisors in Linked List

```
#include <iostream>
#include <algorithm>  // For std::gcd
using namespace std;

// Definition for singly-linked list node.
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* insertGCDNodes(ListNode* head) {
    if (!head || !head->next) {
        return head;  // If the list has 0 or 1 node, return as is
    }

    ListNode* current = head;

    while (current && current->next) {
        // Calculate the GCD of the current node and the next node
        int gcd_val = gcd(current->val, current->next->val);

        // Create a new node with the GCD value
        ListNode* newNode = new ListNode(gcd_val);

        // Insert the new node between current and current->next
        newNode->next = current->next;
        current->next = newNode;

        // Move to the next original node (skip the new node)
        current = current->next->next;
```

```
    }

    return head;
}

// Function to print the linked list
void printList(ListNode* head) {
    ListNode* current = head;
    while (current) {
        cout << current->val << " ";
        current = current->next;
    }
    cout << endl;
}

int main() {
    // Create a linked list: 1 -> 2 -> 3 -> 4
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);

    cout << "Original List: ";
    printList(head);

    // Insert GCD nodes
    head = insertGCDNodes(head);

    cout << "List after inserting GCD nodes: ";
    printList(head);

    return 0;
}
```

```
Original List: 1 2 3 4
List after inserting GCD nodes: 1 1 2 1 3 1 4
```

7. **Merge Two Sorted Lists**

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

// Definition for singly-linked list node.
struct ListNode {
```

```
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
    // Create a dummy node to simplify the merge process
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;  // Pointer to the current node in the merged list

    // Traverse through both lists and merge them
    while (list1 != nullptr && list2 != nullptr) {
        if (list1->val < list2->val) {
            current->next = list1;
            list1 = list1->next;  // Move to the next node in list1
        } else {
            current->next = list2;
            list2 = list2->next;  // Move to the next node in list2
        }
        current = current->next;  // Move the current pointer to the next node
    }

    // If there are remaining nodes in list1 or list2, append them
    if (list1 != nullptr) {
        current->next = list1;
    } else if (list2 != nullptr) {
        current->next = list2;
    }

    // Return the merged list starting from dummy->next
    ListNode* mergedListHead = dummy->next;
    delete dummy;  // Free the dummy node
    return mergedListHead;
}

// Helper function to print the linked list
void printList(ListNode* head) {
    ListNode* current = head;
    while (current != nullptr) {
        cout << current->val << " ";
        current = current->next;
    }
    cout << endl;
}
```

```
int main() {
    // Create list1: 1 -> 2 -> 4
    ListNode* list1 = new ListNode(1);
    list1->next = new ListNode(2);
    list1->next->next = new ListNode(4);

    // Create list2: 1 -> 3 -> 4
    ListNode* list2 = new ListNode(1);
    list2->next = new ListNode(3);
    list2->next->next = new ListNode(4);

    // Merge the two lists
    ListNode* mergedList = mergeTwoLists(list1, list2);

    // Print the merged list
    cout << "Merged List: ";
    printList(mergedList);

    return 0;
}
```

Output

```
Merged List: 1 1 2 3 4 4


=== Code Execution Successful ===
```

8. **Convert Sorted Array to Binary Search Tree**

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

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Helper function to construct the BST
TreeNode* sortedArrayToBSTHelper(const vector<int>& nums, int left, int right) {
    if (left > right) {
```

```
        return nullptr;  // Base case: no elements left
    }

    // Choose the middle element to maintain balance
    int mid = left + (right - left) / 2;
    TreeNode* root = new TreeNode(nums[mid]);

    // Recursively build the left and right subtrees
    root->left = sortedArrayToBSTHelper(nums, left, mid - 1);
    root->right = sortedArrayToBSTHelper(nums, mid + 1, right);

    return root;
}

// Main function to convert the sorted array to a balanced BST
TreeNode* sortedArrayToBST(const vector<int>& nums) {
    return sortedArrayToBSTHelper(nums, 0, nums.size() - 1);
}

// Helper function to print the tree (in-order traversal)
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}

int main() {
    vector<int> nums = {-10, -3, 0, 5, 9};

    // Convert sorted array to balanced BST
    TreeNode* root = sortedArrayToBST(nums);

    // Print the in-order traversal of the BST
    cout << "In-order Traversal of the BST: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Output

```
In-order Traversal of the BST: -10 -3 0 5 9


=== Code Execution Successful ===
```

9. Reverse Linked List

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

// Definition for singly-linked list node.
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

// Function to reverse the linked list iteratively
ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;
    ListNode* next = nullptr;

    while (current != nullptr) {
        // Store the next node
        next = current->next;

        // Reverse the current node's pointer
        current->next = prev;

        // Move pointers one position ahead
        prev = current;
        current = next;
    }

    // prev will be the new head of the reversed list
    return prev;
}

// Helper function to print the linked list
void printList(ListNode* head) {
    ListNode* current = head;
    while (current != nullptr) {
```

```
        cout << current->val << " ";
        current = current->next;
    }
    cout << endl;
}

int main() {
    // Create a simple linked list: 1 -> 2 -> 3 -> 4 -> 5
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    cout << "Original List: ";
    printList(head);

    // Reverse the linked list
    head = reverseList(head);

    cout << "Reversed List: ";
    printList(head);

    return 0;
}
```

Output

```
Original List: 1 2 3 4 5
Reversed List: 5 4 3 2 1


=== Code Execution Successful ===
```

10. Maximum Twin Sum of a Linked List

```
#include <iostream>

#include <stack>

using namespace std;



// Definition for singly-linked list node.

struct ListNode {
```

```
    int val;

    ListNode* next;

    ListNode(int x) : val(x), next(nullptr) {}

};


// Function to find the maximum twin sum of a linked list

int pairSum(ListNode* head) {

    // Step 1: Find the midpoint of the list using the fast and slow pointer technique

    ListNode* slow = head;

    ListNode* fast = head;


    // Move slow pointer one step and fast pointer two steps until fast reaches the end

    while (fast && fast->next) {

        slow = slow->next;

        fast = fast->next->next;

    }


    // Step 2: Reverse the second half of the list

    ListNode* prev = nullptr;

    ListNode* current = slow;

    while (current) {

        ListNode* next = current->next;

        current->next = prev;

        prev = current;

        current = next;

    }
```

```
    // Step 3: Calculate the maximum twin sum
    int maxTwinSum = 0;
    ListNode* firstHalf = head;
    ListNode* secondHalf = prev;  // This is the reversed second half

    // Compare corresponding nodes from both halves
    while (secondHalf) {
        maxTwinSum = max(maxTwinSum, firstHalf->val + secondHalf->val);
        firstHalf = firstHalf->next;
        secondHalf = secondHalf->next;
    }

    return maxTwinSum;
}

// Helper function to print the linked list
void printList(ListNode* head) {
    ListNode* current = head;
    while (current) {
        cout << current->val << " ";
        current = current->next;
    }
    cout << endl;
}

int main() {
    // Create a sample linked list: 5 -> 4 -> 2 -> 1
```

```
    ListNode* head = new ListNode(5);

    head->next = new ListNode(4);

    head->next->next = new ListNode(2);

    head->next->next->next = new ListNode(1);



    // Print the original linked list

    cout << "Original List: ";

    printList(head);



    // Find the maximum twin sum

    int result = pairSum(head);

    cout << "Maximum Twin Sum: " << result << endl;



    return 0;

}
```

Output

```
Original List: 5 4 2 1
Maximum Twin Sum: 6


=== Code Execution Successful ===
```