Day-2

Arrays & Linked List

Student Name: Riya Sharma **UID: 22BCS11911**
Branch: BE - CSE **Section/Group: FL_IOT-603-A**
Semester: 5th **Date of Performance:20-12-24**

Problem 1 - Majority Elements

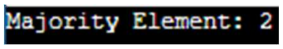
Aim- Given an array nums of size n, return the majority element.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Solution-

```
#include <iostream>

#include <vector>


using namespace std;


int majorityElement(vector<int>& nums) {

    int candidate = 0, count = 0;


    for (int num : nums) {

        if (count == 0) {

            candidate = num;

        }

        count += (num == candidate) ? 1 : -1;

    }


    return candidate;
}

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

    int result = majorityElement(nums);

    cout << "Majority Element: " << result << endl;

    return 0;
}
```

Output-



```
Majority Element: 2
```

Problem 2 - Single Number

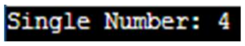
Aim- Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Solution-

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

using namespace std;

int singleNumber(vector<int>& nums) {
    int result = 0;
    for (int num : nums) {
        result ^= num;
    }

    return result;
}

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

    int result = singleNumber(nums);

    cout << "Single Number: " << result << endl;

    return 0;
}
```

Output-



```
Single Number: 4
```

Problem 3- Convert Sorted Array to Binary Search Tree

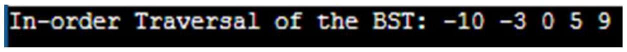
Aim-Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Solution-

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

using namespace std;
```

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

TreeNode* sortedArrayToBST(vector<int>& nums, int left, int right) {
    if (left > right) return nullptr;

    int mid = left + (right - left) / 2;

    TreeNode* root = new TreeNode(nums[mid]);

    root->left = sortedArrayToBST(nums, left, mid - 1);
    root->right = sortedArrayToBST(nums, mid + 1, right);

    return root;
}

TreeNode* sortedArrayToBST(vector<int>& nums) {
    return sortedArrayToBST(nums, 0, nums.size() - 1);
}

void inorderTraversal(TreeNode* root) {
    if (!root) return;

    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}

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

    TreeNode* root = sortedArrayToBST(nums);

    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
```

Problem 4: Merge Two Sorted Lists

Aim- You are given the heads of two sorted linked lists list1 and list2.

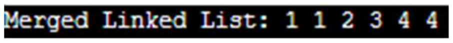
Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Solution-

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

using namespace std;

struct ListNode {
    int val;
    ListNode* next;

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

ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;

    while (list1 != nullptr && list2 != nullptr) {
        if (list1->val < list2->val) {
            current->next = list1;
            list1 = list1->next;
        } else {
            current->next = list2;
            list2 = list2->next;
        }
        current = current->next;
    }

    if (list1 != nullptr) {
        current->next = list1;
    } else {
        current->next = list2;
    }

    return dummy->next;
}

void printList(ListNode* head) {
    while (head != nullptr) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}

ListNode* createList(const vector<int>& nums) {
    ListNode* head = nullptr;
    ListNode* tail = nullptr;

    for (int num : nums) {
        ListNode* newNode = new ListNode(num);
        if (head == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = tail->next;
        }
    }

    return head;
}

int main() {
    ListNode* list1 = createList({1, 2, 4});
    ListNode* list2 = createList({1, 3, 4});

    ListNode* mergedList = mergeTwoLists(list1, list2);

    cout << "Merged Linked List: ";
    printList(mergedList);

    return 0;
}
```

Output-



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

Problem 5: Linked List Cycle

Aim:

Given head, the head of a linked list, determine if the linked list has a cycle in it.

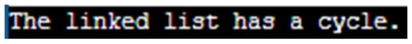
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.

Solution:

```
#include <iostream>

#include <vector>

using namespace std;


struct ListNode {

    int val;

    ListNode* next;


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

};


bool hasCycle(ListNode* head) {

    if (head == nullptr) return false;


    ListNode* slow = head;

    ListNode* fast = head;


    while (fast != nullptr && fast->next != nullptr) {

        slow = slow->next;

        fast = fast->next->next;


        if (slow == fast) {
            return true;
        }
    }

    return false;
}

ListNode* createList(const vector<int>& nums) {
    ListNode* head = nullptr;
    ListNode* tail = nullptr;

    for (int num : nums) {
        ListNode* newNode = new ListNode(num);
        if (head == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = tail->next;
        }
    }

    return head;
}

void createCycle(ListNode* head, int pos) {
    if (head == nullptr) return;
    ListNode* cycleNode = nullptr;
    ListNode* temp = head;
    int index = 0;

    while (temp->next != nullptr) {
        if (index == pos) {
            cycleNode = temp;
        }
        temp = temp->next;
        index++;
    }

    if (cycleNode != nullptr) {
        temp->next = cycleNode;
    }
}

int main() {
    ListNode* head = createList({3, 2, 0, -4});

    createCycle(head, 1);

    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.
```

Problem 6- Pascal's Triangle

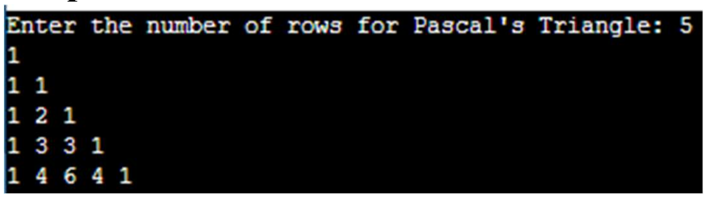
Aim- Given an integer numRows, return the first numRows of Pascal's triangle.

In Pascal's triangle, each number is the sum of the two numbers directly

Solution:

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

using namespace std;

vector<vector<int>> generate(int numRows) {
    vector<vector<int>> triangle;

    if (numRows == 0) {
        return triangle;
    }

    triangle.push_back({1});
    for (int i = 1; i < numRows; i++) {
        vector<int> row(i + 1, 1);

        for (int j = 1; j < i; j++) {
            row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
        }

        triangle.push_back(row);
    }

    return triangle;
}

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

int main() {
    int numRows;
    cout << "Enter the number of rows for Pascal's Triangle: ";
    cin >> numRows;
    vector<vector<int>> pascalTriangle = generate(numRows);
    printTriangle(pascalTriangle);

    return 0;
}
```

Output-



```
Enter the number of rows for Pascal's Triangle: 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Problem 7- Remove Element

Aim-Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.

Return k.

Custom Judge:

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

for (int i = 0; i < k; i++) {

assert nums[i] == expectedNums[i];

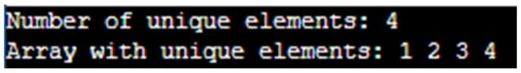
}

If all assertions pass, then your solution will be accepted.

Solution-

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

using namespace std;

// Function to remove duplicates from the sorted array in-place
int removeDuplicates(vector<int>& nums) {
    // If the array is empty, return 0 (no unique elements)
    if (nums.empty()) {
        return 0;
    }

    int i = 0;  // Pointer to track the position of unique elements

    // Iterate through the array using pointer j
    for (int j = 1; j < nums.size(); j++) {
        // If nums[j] is not equal to nums[i], it's a new unique element
        if (nums[j] != nums[i]) {
            i++;  // Move pointer i to the next position
            nums[i] = nums[j];  // Update nums[i] with the unique element
        }
    }

    // The number of unique elements is i + 1
    return i + 1;
}

// Function to print the array
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};

    // Remove duplicates and get the number of unique elements
    int k = removeDuplicates(nums);

    // Print the unique elements in the array
    cout << "Number of unique elements: " << k << endl;
    cout << "Array with unique elements: ";
    printArray(nums, k);

    return 0;
}
```

Output:



```
Number of unique elements: 4
Array with unique elements: 1 2 3 4
```

Problem 8- Baseball Game

Aim- **You are keeping the scores for a baseball game with strange rules. At the beginning of the game, you start with an empty record.**

You are given a list of strings operations, where operations[i] is the ith operation you must apply to the record and is one of the following:

An integer x.

Record a new score of x.

'+'.

Record a new score that is the sum of the previous two scores.

'D'.

Record a new score that is the double of the previous score.

'C'.

Invalidate the previous score, removing it from the record.

Return the sum of all the scores on the record after applying all the operations.

The test cases are generated such that the answer and all inter

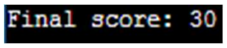
mediate calculations fit in a 32-bit integer and that all operations are valid.

Solution:

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

```
#include <string>


using namespace std;


int calPoints(vector<string>& operations) {
    vector<int> record;


    for (const string& op : operations) {
        if (op == "+") {
            int size = record.size();
            record.push_back(record[size - 1] + record[size - 2]);
        } else if (op == "D") {
            record.push_back(2 * record.back());
        } else if (op == "C") {
            record.pop_back();
        } else {
            record.push_back(stoi(op));
        }
    }


    int totalSum = 0;
    for (int score : record) {
        totalSum += score;
    }


    return totalSum;
}

int main() {
    vector<string> operations = {"5", "2", "C", "D", "+"};
    int result = calPoints(operations);
    cout << "Final score: " << result << endl;
    return 0;
}
```

Output:



```
Final score: 30
```

Problem 9- Remove Linked List Elements

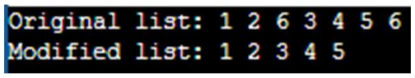
Aim: **Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.**

Solution:

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


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

```
ListNode* removeElements(ListNode* head, int val) {
    ListNode* dummy = new ListNode(-1);
    dummy->next = head;
    ListNode* current = dummy;

    while (current->next) {
        if (current->next->val == val) {
            ListNode* temp = current->next;
            current->next = current->next->next;
            delete temp;
        } else {
            current = current->next;
        }
    }

    head = dummy->next;
    delete dummy;
    return head;
}

void printList(ListNode* head) {
    while (head) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}


int main() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(6);
    head->next->next->next = new ListNode(3);
    head->next->next->next->next = new ListNode(4);
    head->next->next->next->next->next = new ListNode(5);
    head->next->next->next->next->next->next = new ListNode(6);


    int val = 6;
    cout << "Original list: ";
    printList(head);


    head = removeElements(head, val);
    cout << "Modified list: ";

    printList(head);


    return 0;

}
```

Output:



```
Original list: 1 2 6 3 4 5 6
Modified list: 1 2 3 4 5
```

Problem 10 - Reverse Linked List

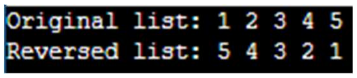
Aim: Given the head of a singly linked list, reverse the list, and return the reversed list.

Solution:

```
#include <iostream>

using namespace std;


struct ListNode {

    int val;

    ListNode* next;

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

};


ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;

    ListNode* current = head;

    ListNode* next = nullptr;



    while (current) {

        next = current->next;

        current->next = prev;

        prev = current;

        current = next;

    }



    return prev;

}



void printList(ListNode* head) {

    while (head) {

        cout << head->val << " ";

        head = head->next;

    }

    cout << endl;

}
```

```
int main() {
    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);

    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
```

Problem 11: Container With Most Water

Aim: **You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).**

Find two lines that together with the x-axis form a container, such that the container contains the most water.

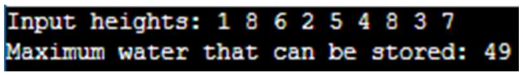
Return the maximum amount of water a container can store.

Notice that you may not slant the container.

Solution:

```
#include <iostream>

#include <vector>

using namespace std;


int maxArea(vector<int>& height) {

    int left = 0, right = height.size() - 1;

    int maxWater = 0;


    while (left < right) {

        int width = right - left;

        int h = min(height[left], height[right]);

        maxWater = max(maxWater, width * h);


        if (height[left] < height[right]) {

            left++;

        } else {

            right--;

        }
    }

    return maxWater;
}

int main() {
    vector<int> height = {1, 8, 6, 2, 5, 4, 8, 3, 7};

    cout << "Input heights: ";
    for (int h : height) {
        cout << h << " ";
    }
    cout << endl;

    int result = maxArea(height);

    cout << "Maximum water that can be stored: " << result << endl;

    return 0;
}
```

Output:



```
Input heights: 1 8 6 2 5 4 8 3 7
Maximum water that can be stored: 49
```

Problem 12: Valid Sudoku

Aim: **Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:**

Each row must contain the digits 1-9 without repetition.

Each column must contain the digits 1-9 without repetition.

Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

A Sudoku board (partially filled) could be valid but is not necessarily solvable.

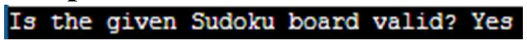
Only the filled cells need to be validated according to the mentioned rules

Solution:

```
#include <iostream>

#include <vector>

#include <unordered_set>

using namespace std;



bool isValidSudoku(vector<vector<char>>& board) {

    vector<unordered_set<char>> rows(9), cols(9), boxes(9);



    for (int i = 0; i < 9; i++) {

        for (int j = 0; j < 9; j++) {

            char num = board[i][j];

            if (num == '.') continue;



            if (rows[i].count(num) || cols[j].count(num) || boxes[(i / 3) * 3 + j / 3].count(num))
```

```
                return false;


            rows[i].insert(num);

            cols[j].insert(num);

            boxes[(i / 3) * 3 + j / 3].insert(num);

        }

    }


    return true;

}


int main() {

    vector<vector<char>> board = {

        {'5', '3', '.', '.', '7', '.', '.', '.', '.'},

        {'6', '.', '.', '1', '9', '5', '.', '.', '.'},

        {'.', '9', '8', '.', '.', '.', '.', '6', '.'},

        {'8', '.', '.', '.', '6', '.', '.', '.', '3'},

        {'4', '.', '.', '8', '.', '3', '.', '.', '1'},

        {'7', '.', '.', '.', '2', '.', '.', '.', '6'},

        {'.', '6', '.', '.', '.', '.', '2', '8', '.'},

        {'.', '.', '.', '4', '1', '9', '.', '.', '5'},
        {'.', '.', '.', '.', '8', '.', '.', '7', '9'}
    };

    cout << "Is the given Sudoku board valid? " << (isValidSudoku(board) ? "Yes" : "No") <<
endl;

    return 0;
}
```

Output:



```
Is the given Sudoku board valid? Yes
```

Problem 13: Jump Game II

Aim:

You are given a 0-indexed array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

0 <= j <= nums[i] and

i + j < n

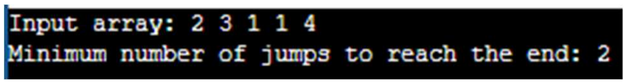
Return the minimum number of jumps to reach nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

Solution:

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

```
int jump(vector<int>& nums) {
    int n = nums.size();
    int jumps = 0, currentEnd = 0, farthest = 0;

    for (int i = 0; i < n - 1; i++) {
        farthest = max(farthest, i + nums[i]);

        if (i == currentEnd) {
            jumps++;
            currentEnd = farthest;
        }
    }

    return jumps;
}

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

    cout << "Input array: ";
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;
    int result = jump(nums);

    cout << "Minimum number of jumps to reach the end: " << result << endl;

    return 0;
}
```

Output:



```
Input array: 2 3 1 1 4
Minimum number of jumps to reach the end: 2
```

Problem 14 : Populating Next Right Pointers in Each Node

Aim: **You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:**

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

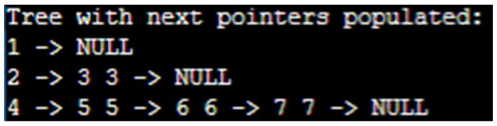
Solution:

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

```
struct Node {
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node(int _val) : val(_val), left(nullptr), right(nullptr), next(nullptr) {}
};

Node* connect(Node* root) {
    if (!root) return nullptr;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        Node* prev = nullptr;

        for (int i = 0; i < size; i++) {
            Node* curr = q.front();
            q.pop();

            if (prev) prev->next = curr;
            prev = curr;
            if (curr->left) q.push(curr->left);
            if (curr->right) q.push(curr->right);
        }
    }

    return root;
}

void printTreeWithNext(Node* root) {
    Node* levelStart = root;

    while (levelStart) {
        Node* curr = levelStart;
        while (curr) {
            cout << curr->val << " -> ";
            if (curr->next) {
                cout << curr->next->val << " ";
            } else {
                cout << "NULL ";
            }
            curr = curr->next;
        }
        cout << endl;
        levelStart = levelStart->left;
    }
```

```
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    root = connect(root);

    cout << "Tree with next pointers populated:" << endl;
    printTreeWithNext(root);

    return 0;
}
```

Output:



```
Tree with next pointers populated:
1 -> NULL
2 -> 3 3 -> NULL
4 -> 5 5 -> 6 6 -> 7 7 -> NULL
```

Problem 15: Design Circular Queue

Aim:

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the MyCircularQueue class:

- **MyCircularQueue(k) Initializes the object with the size of the queue to be k.**
- **int Front() Gets the front item from the queue. If the queue is empty, return -1.**
- **int Rear() Gets the last item from the queue. If the queue is empty, return -1.**
- **boolean enQueue(int value) Inserts an element into the circular queue. Return true if the operation is successful.**
- **boolean deQueue() Deletes an element from the circular queue. Return true if the operation is successful.**
- **boolean isEmpty() Checks whether the circular queue is empty or not.**
- **boolean isFull() Checks whether the circular queue is full or not.**

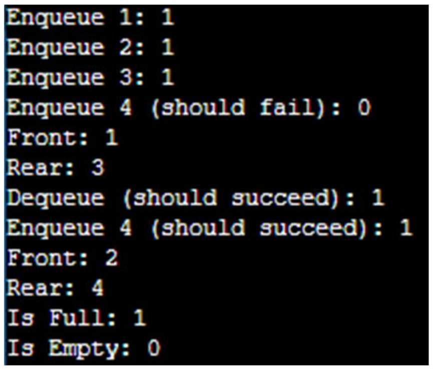
You must solve the problem without using the built-in queue data structure in your programming language.

Solution:

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

class MyCircularQueue {
private:
    vector<int> queue;
    int front, rear, size;

public:
    MyCircularQueue(int k) {
        size = k;
        queue.resize(k);
        front = rear = -1;
    }

    bool enQueue(int value) {
        if (isFull()) return false;
        if (isEmpty()) front = 0;
        rear = (rear + 1) % size;
        queue[rear] = value;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % size;
        }
        return true;
    }

    int Front() {
        if (isEmpty()) return -1;
        return queue[front];
    }
```

```
    int Rear() {
        if (isEmpty()) return -1;
        return queue[rear];
    }

    bool isEmpty() {
        return front == -1;
    }

    bool isFull() {
        return (rear + 1) % size == front;
    }
};

int main() {
    MyCircularQueue queue(3);

    cout << "Enqueue 1: " << queue.enQueue(1) << endl;
    cout << "Enqueue 2: " << queue.enQueue(2) << endl;
    cout << "Enqueue 3: " << queue.enQueue(3) << endl;
    cout << "Enqueue 4 (should fail): " << queue.enQueue(4) << endl;

    cout << "Front: " << queue.Front() << endl;
    cout << "Rear: " << queue.Rear() << endl;

    cout << "Dequeue (should succeed): " << queue.deQueue() << endl;
    cout << "Enqueue 4 (should succeed): " << queue.enQueue(4) << endl;


    cout << "Front: " << queue.Front() << endl;

    cout << "Rear: " << queue.Rear() << endl;

    cout << "Is Full: " << queue.isFull() << endl;

    cout << "Is Empty: " << queue.isEmpty() << endl;


    return 0;

}
```

Output:



```
Enqueue 1: 1
Enqueue 2: 1
Enqueue 3: 1
Enqueue 4 (should fail): 0
Front: 1
Rear: 3
Dequeue (should succeed): 1
Enqueue 4 (should succeed): 1
Front: 2
Rear: 4
Is Full: 1
Is Empty: 0
```

Problem 16: Maximum Number of Groups Getting Fresh Donuts

Aim:

There is a donuts shop that bakes donuts in batches of batchSize. They have a rule where they must serve all of the donuts of a batch before serving any donuts of the next batch. You are given an integer batchSize and an integer array groups, where groups[i] denotes that there is a group of groups[i] customers that will visit the shop. Each customer will get exactly one donut.

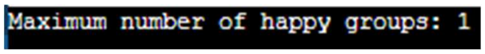
When a group visits the shop, all customers of the group must be served before serving any of the following groups. A group will be happy if they all get fresh donuts. That is, the first customer of the group does not receive a donut that was left over from the previous group.

You can freely rearrange the ordering of the groups. Return the maximum possible number of happy groups after rearranging the groups.

Solution:

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


int maxHappyGroups(int batchSize, vector<int>& groups) {

    int n = groups.size();

    vector<int> dp(1 << n, -1);

    dp[0] = 0;

    int totalDonuts = 0;


    for (int i = 0; i < n; ++i) {

        totalDonuts += groups[i];

    }


    int maxGroups = 0;


    for (int mask = 0; mask < (1 << n); ++mask) {

        int currentDonuts = 0;

        int groupCount = 0;


        for (int i = 0; i < n; ++i) {

            if (mask & (1 << i)) {

                currentDonuts += groups[i];

                if (currentDonuts % batchSize == 0) {
                    ++groupCount;
                }
            }
        }

        maxGroups = max(maxGroups, groupCount);
    }

    return maxGroups;
}

int main() {
    vector<int> groups = {2, 3, 5, 8};
    int batchSize = 6;

    cout << "Maximum number of happy groups: " << maxHappyGroups(batchSize, groups) <<
endl;

    return 0;
}
```

Output:



```
Maximum number of happy groups: 1
```

Problem 17 : Cherry Pickup II

Aim: **You are given a rows x cols matrix grid representing a field of cherries where grid[i][j] represents the number of cherries that you can collect from the (i, j) cell.**

You have two robots that can collect cherries for you:

Robot #1 is located at the top-left corner (0, 0), and

Robot #2 is located at the top-right corner (0, cols - 1).

Return the maximum number of cherries collection using both robots by following the rules below:

From a cell (i, j), robots can move to cell (i + 1, j - 1), (i + 1, j), or (i + 1, j + 1).

When any robot passes through a cell, It picks up all cherries, and the cell becomes an empty cell.

When both robots stay in the same cell, only one takes the cherries.

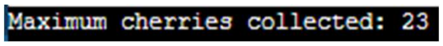
Both robots cannot move outside of the grid at any moment.

Both robots should reach the bottom row in grid.

Solution:

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


int cherryPickup(vector<vector<int>>& grid) {
    int rows = grid.size(), cols = grid[0].size();
    vector<vector<vector<int>>> dp(rows, vector<vector<int>>(cols, vector<int>(cols, -1)));


    dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1];


    for (int i = 1; i < rows; ++i) {
        for (int j1 = 0; j1 < cols; ++j1) {
            for (int j2 = 0; j2 < cols; ++j2) {
                if (dp[i - 1][j1][j2] == -1) continue;
                int maxCherries = dp[i - 1][j1][j2];
                for (int dj1 = -1; dj1 <= 1; ++dj1) {
                    for (int dj2 = -1; dj2 <= 1; ++dj2) {
                        int nj1 = j1 + dj1, nj2 = j2 + dj2;
                        if (nj1 >= 0 && nj1 < cols && nj2 >= 0 && nj2 < cols) {
                            int cherries = (j1 == j2 ? grid[i][nj1] : grid[i][nj1] + grid[i][nj2]);
                            dp[i][nj1][nj2] = max(dp[i][nj1][nj2], maxCherries + cherries);
                        }
                    }
                }
            }
        }
    }

    int result = 0;
    for (int j1 = 0; j1 < cols; ++j1) {
        for (int j2 = 0; j2 < cols; ++j2) {
            result = max(result, dp[rows - 1][j1][j2]);
        }
    }

    return result;
}

int main() {
    vector<vector<int>> grid = {
        {3, 1, 1},
        {2, 5, 4},
        {1, 5, 1}
    };

    cout << "Maximum cherries collected: " << cherryPickup(grid) << endl;

    return 0;
}
```

Output:



```
Maximum cherries collected: 23
```

Problem 18: Maximum Number of Darts Inside of a Circular Dartboard

Aim: **Alice is throwing n darts on a very large wall. You are given an array darts where darts[i] = [xi, yi] is the position of the ith dart that Alice threw on the wall.**

Bob knows the positions of the n darts on the wall. He wants to place a dartboard of radius r on the wall so that the maximum number of darts that Alice throws lie on the dartboard.

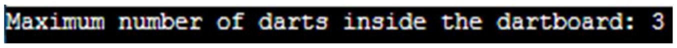
Given the integer r, return the maximum number of darts that can lie on the dartboard.

Solution:

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <cmath>

using namespace std;
```

```
int maxPoints(vector<vector<int>>& darts, int r) {
    int n = darts.size();
    int result = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            int x1 = darts[i][0], y1 = darts[i][1];
            int x2 = darts[j][0], y2 = darts[j][1];

            double midX = (x1 + x2) / 2.0;
            double midY = (y1 + y2) / 2.0;
            double dist = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
            double radius = dist / 2;

            if (radius > r) continue;

            int count = 0;
            for (int k = 0; k < n; ++k) {
                int x3 = darts[k][0], y3 = darts[k][1];
                double d = sqrt(pow(x3 - midX, 2) + pow(y3 - midY, 2));
                if (d <= r) count++;
            }

            result = max(result, count);
        }
    }

    return result;
}

int main() {
    vector<vector<int>> darts = {{1, 2}, {2, 3}, {3, 4}, {5, 6}};
    int radius = 2;

    cout << "Maximum number of darts inside the dartboard: " << maxPoints(darts, radius) <<
endl;

    return 0;
}
```

Output:



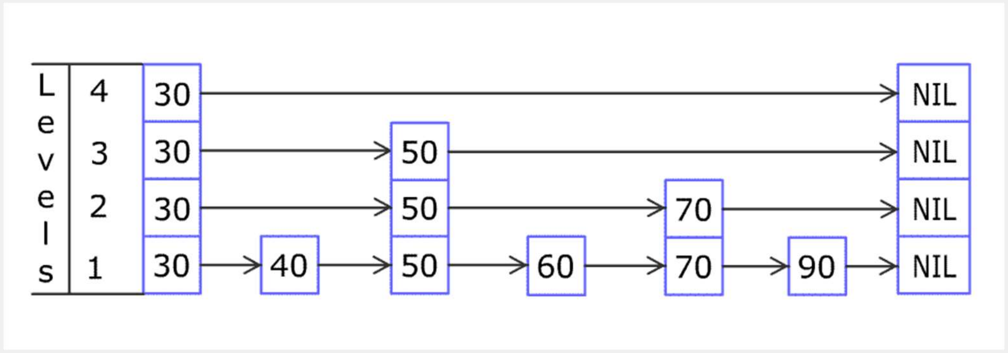
```
Maximum number of darts inside the dartboard: 3
```

Problem 19: Design Skiplist

Aim: **Design a Skiplist without using any built-in libraries.**

A skiplist is a data structure that takes O(log(n)) time to add, erase and search. Comparing with treap and red-black tree which has the same function and performance, the code length of Skiplist can be comparatively short and the idea behind Skiplists is just simple linked lists.

For example, we have a Skiplist containing [30,40,50,60,70,90] and we want to add 80 and 45 into it. The Skiplist works this way:



Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

You can see there are many layers in the Skiplist. Each layer is a sorted linked list. With the help of the top layers, add, erase and search can be faster than O(n). It can be proven that the average time complexity for each operation is O(log(n)) and space complexity is O(n).

See more about Skiplist: https://en.wikipedia.org/wiki/Skip_list

Implement the Skiplist class:

- Skiplist() Initializes the object of the skiplist.
- bool search(int target) Returns true if the integer target exists in the Skiplist or false otherwise.
- void add(int num) Inserts the value num into the SkipList.
- bool erase(int num) Removes the value num from the Skiplist and returns true. If num does not exist in the Skiplist, do nothing and return false. If there exist multiple num values, removing any one of them is fine.

Note that duplicates may exist in the Skiplist, your code needs to handle this situation.

Solution:

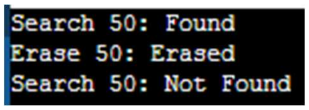
```
#include <iostream>

#include <cstdlib>

#include <vector>

#include <ctime>



using namespace std;



class Skiplist {

private:

    struct Node {

        int val;

        vector<Node*> next;



        Node(int value, int level) : val(value), next(level, nullptr) {}

    };



    int maxLevel;

    Node* head;



    int randomLevel() {
        int level = 1;
        while (rand() % 2 && level < maxLevel) {
            level++;
        }
        return level;
    }

public:
    Skiplist() : maxLevel(16), head(new Node(-1, maxLevel)) {
        srand(time(0));
    }

    bool search(int target) {
        Node* node = head;
        for (int level = maxLevel - 1; level >= 0; --level) {
            while (node->next[level] && node->next[level]->val < target) {
                node = node->next[level];
            }
        }
        node = node->next[0];
        return node && node->val == target;
```

```
}


void add(int num) {
    Node* node = head;
    vector<Node*> update(maxLevel, nullptr);


    for (int level = maxLevel - 1; level >= 0; --level) {
        while (node->next[level] && node->next[level]->val < num) {
            node = node->next[level];
        }
        update[level] = node;
    }


    int newLevel = randomLevel();
    Node* newNode = new Node(num, newLevel);
    for (int level = 0; level < newLevel; ++level) {
        newNode->next[level] = update[level]->next[level];
        update[level]->next[level] = newNode;
    }
}
```

```
    bool erase(int num) {
        Node* node = head;
        vector<Node*> update(maxLevel, nullptr);
        bool found = false;

        for (int level = maxLevel - 1; level >= 0; --level) {
            while (node->next[level] && node->next[level]->val < num) {
                node = node->next[level];
            }
            update[level] = node;
        }

        node = node->next[0];
        if (node && node->val == num) {
            found = true;
            for (int level = 0; level < maxLevel; ++level) {
                if (update[level]->next[level] == node) {
                    update[level]->next[level] = node->next[level];
                }
            }
            delete node;
        }

        return found;
    }
};

int main() {
    Skiplist skiplist;

    skiplist.add(30);
    skiplist.add(40);
    skiplist.add(50);
    skiplist.add(60);
    skiplist.add(70);
    skiplist.add(90);

    cout << "Search 50: " << (skiplist.search(50) ? "Found" : "Not Found") << endl;
    cout << "Erase 50: " << (skiplist.erase(50) ? "Erased" : "Not Found") << endl;
    cout << "Search 50: " << (skiplist.search(50) ? "Found" : "Not Found") << endl;

    return 0;
```

}

Output:



```
Search 50: Found
Erase 50: Erased
Search 50: Not Found
```