Name: Archi bansal Date: 20/12/2024

UID: 22BCS15264 Section: 620 - A

Q 1 : Majority Elements

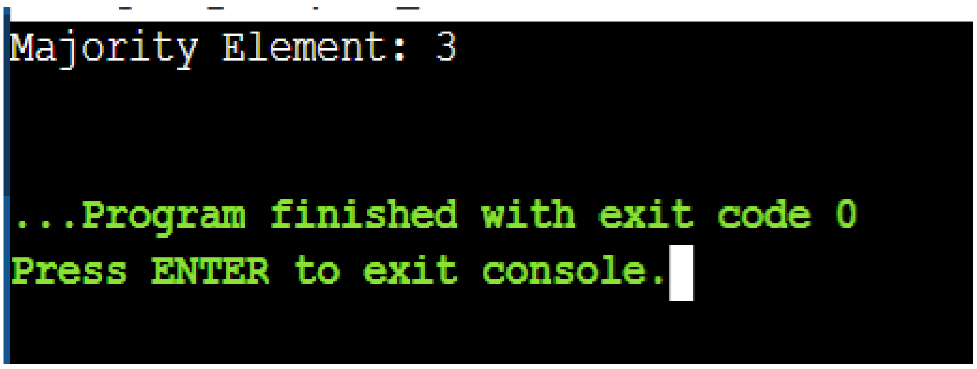
Given an array nums of size n, return the majority element.

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.

Soln:

```
#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; // Set new candidate
        }
        count += (num == candidate) ? 1 : -1; // Adjust count
    }
    return candidate; // Return the majority element
}
int main() {
    vector<int> nums = {3, 2, 3};
    cout << "Majority Element: " << majorityElement(nums) << endl;
    return 0;
}
```

Output:



```
Majority Element: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

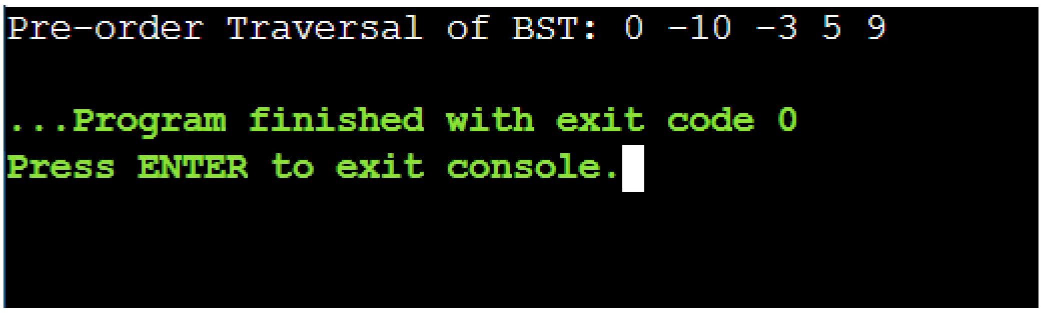
Question 2 Convert Sorted Array to Binary Search Tree

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

Example 1:

Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Soln:

```
#include <iostream>
#include <vector>
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) {}
};
TreeNode* sortedArrayToBST(vector<int>& nums, int left, int right) {
    if (left > right) return nullptr;
    int mid = left + (right - left) / 2; // Find the middle element
    TreeNode* root = new TreeNode(nums[mid]);
    root->left = sortedArrayToBST(nums, left, mid - 1); // Construct left subtree
    root->right = sortedArrayToBST(nums, mid + 1, right); // Construct right subtree
    return root;
}
TreeNode* sortedArrayToBST(vector<int>& nums) {
    return sortedArrayToBST(nums, 0, nums.size() - 1);
}
// Helper function for testing
void preOrderTraversal(TreeNode* root) {
    if (!root) return;
    cout << root->val << " ";
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}
int main() {
    vector<int> nums = {-10, -3, 0, 5, 9};
    TreeNode* root = sortedArrayToBST(nums);
    cout << "Pre-order Traversal of BST: ";
    preOrderTraversal(root);
    return 0;
}
```



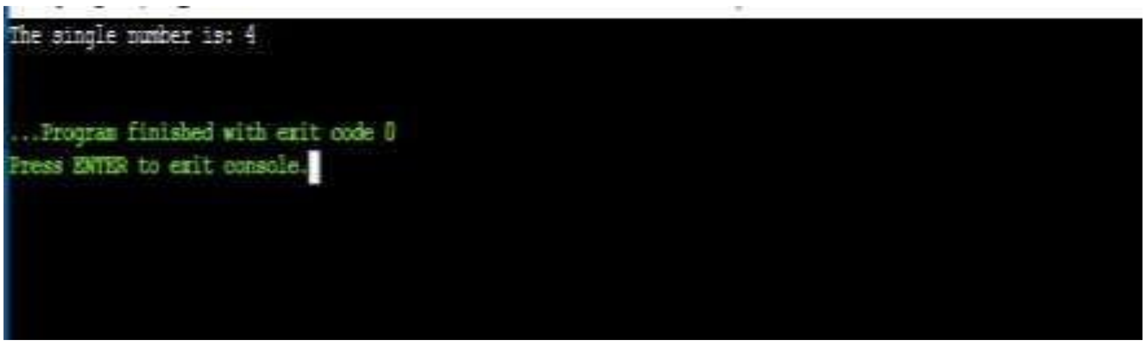
```
Pre-order Traversal of BST: 0 -10 -3 5 9

...Program finished with exit code 0
Press ENTER to exit console.
```

2. 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.

Code:

```
#include <iostream>
using namespace std;
int singleNumber(int nums[], int n) {
int result = 0;
for (int i = 0; i < n; i++) {
result ^= nums[i]; // XOR operation
}
return result;
}
int main() {
int nums[] = {4, 1, 2, 1, 2};
int n = sizeof(nums) / sizeof(nums[0]);
cout << "The single number is: " << singleNumber(nums, n) << endl;
return 0;
}
```

Output:



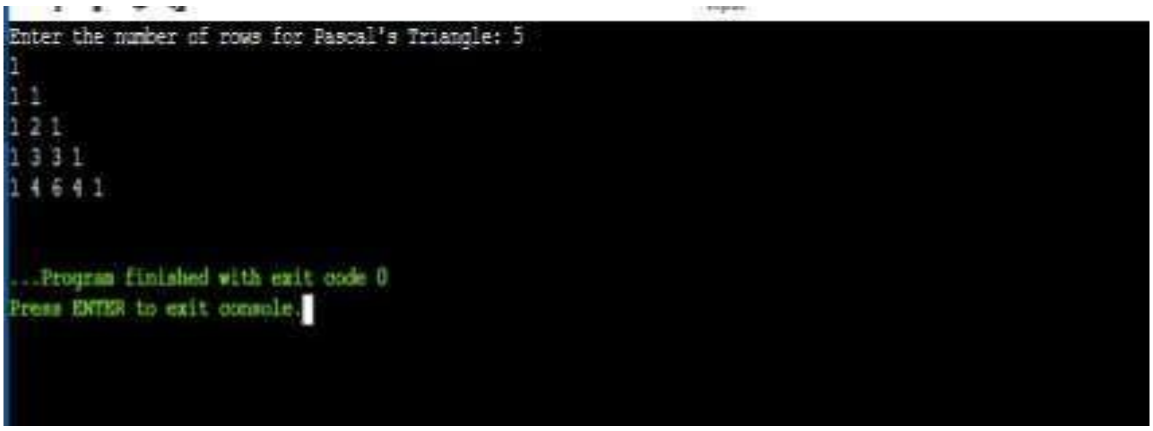
```
The single number is: 4

...Program finished with exit code 0
Press ENTER to exit console.
```

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

Code:

```
#include <iostream>
using namespace std;
// Function to generate Pascal's Triangle
void generatePascalsTriangle(int numRows) {
int triangle[numRows][numRows];
// Build the Pascal's Triangle
for (int i = 0; i < numRows; i++) {
for (int j = 0; j <= i; j++) {
// The first and last elements in each row are 1
if (j == 0 || j == i) {
triangle[i][j] = 1;
} else {
// Every other element is the sum of the two elements above it
triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
}
}
}
// Print Pascal's Triangle
for (int i = 0; i < numRows; i++) {
for (int j = 0; j <= i; j++) {
cout << triangle[i][j] << " ";
}
cout << endl;
}
}
int main() {
int numRows;
cout << "Enter the number of rows for Pascal's Triangle: ";
cin >> numRows;
generatePascalsTriangle(numRows);
return 0;
}
```

Output:



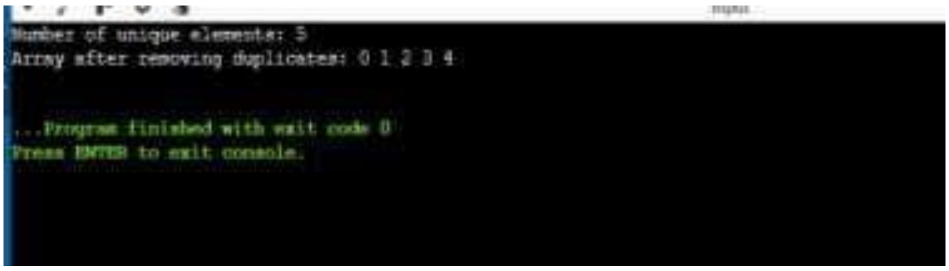
```
Enter the number of rows for Pascal's Triangle: 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1


...Program finished with exit code 0
Press ENTER to exit console.
```

5. 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.

Code:

```
#include <iostream>
using namespace std;
int removeDuplicates(int nums[], int n) {
if (n == 0) return 0; // If the array is empty, return 0
int k = 1; // The first unique element is at index 0
for (int i = 1; i < n; i++) {
if (nums[i] != nums[k - 1]) { // Compare with the last unique element
nums[k] = nums[i]; // Update the position for the next unique element
k++;
}
}
return k; // Number of unique elements
}
int main() {
int nums[] = {0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
int n = sizeof(nums) / sizeof(nums[0]);
int k = removeDuplicates(nums, n);
cout << "Number of unique elements: " << k << endl;
cout << "Array after removing duplicates: ";
for (int i = 0; i < k; i++) {
cout << nums[i] << " ";
}
cout << endl;
return 0;
}
```

Output:



```
Number of unique elements: 5
Array after removing duplicates: 0 1 2 3 4

...Program finished with exit code 0
Press ENTER to exit console.
```

6. 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*.

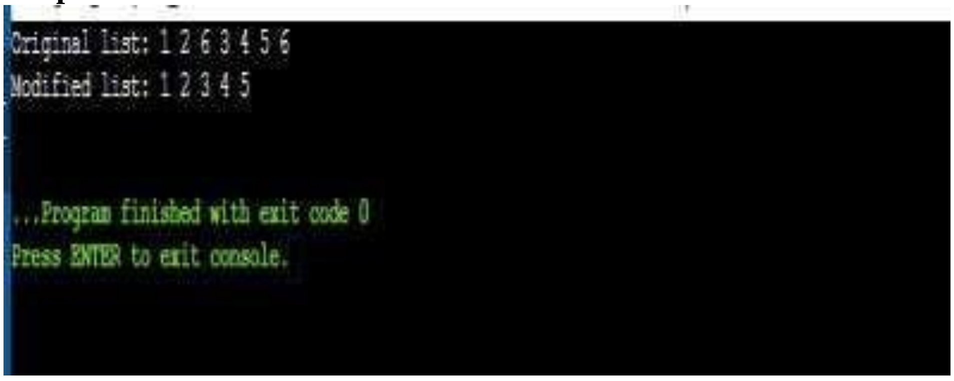
Code:

```
#include <iostream>
using namespace std;
// Definition for a singly-linked list node
struct ListNode {
```

```
int val;
ListNode* next;
ListNode(int x) : val(x), next(nullptr) {}
};
// Function to remove all nodes with the given value
ListNode* removeElements(ListNode* head, int val) {
// Handle the case where the head itself needs to be removed
while (head != nullptr && head->val == val) {
ListNode* temp = head;
head = head->next;
delete temp; // Free memory of the removed node
}
// Pointer to traverse the list
ListNode* current = head;
// Traverse the list and remove nodes with the given value
while (current != nullptr && current->next != nullptr) {
if (current->next->val == val) {
ListNode* temp = current->next;
current->next = current->next->next; // Bypass the node
delete temp; // Free memory of the removed node
} else {
current = current->next; // Move to the next node
}
}
return head;
}
// Function to print the linked list
void printList(ListNode* head) {
while (head != nullptr) {
cout << head->val << " ";
head = head->next;
}
cout << endl;
}
// Main function
int main() {
// Create a sample linked list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6
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);
cout << "Original list: ";
printList(head);
int val = 6;
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


...Program finished with exit code 0
Press ENTER to exit console.
```

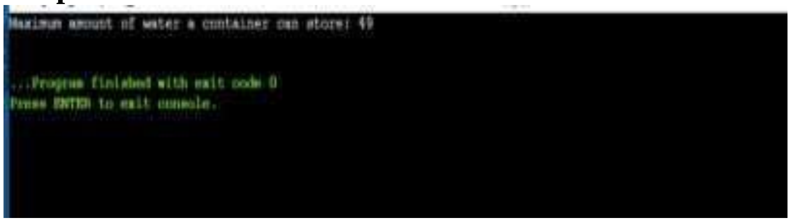
Medium

7. 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.

Code:

```
#include <iostream>
using namespace std;
int maxArea(int height[], int n) {
int left = 0, right = n - 1; // Two pointers
int max_area = 0;
while (left < right) {
// Calculate the current area
int width = right - left;
int h = (height[left] < height[right]) ? height[left] : height[right];
int current_area = width * h;
// Update the maximum area
if (current_area > max_area) {
max_area = current_area;
}
// Move the pointer pointing to the shorter line
if (height[left] < height[right]) {
left++;
} else {
right--;
}
}
return max_area;
}
int main() {
int height[] = {1, 8, 6, 2, 5, 4, 8, 3, 7};
int n = sizeof(height) / sizeof(height[0]);
int result = maxArea(height, n);
cout << "Maximum amount of water a container can store: " << result << endl;
return 0;
}
```

Output:



```
Maximum amount of water a container can store: 49


...Program finished with exit code 0
Press ENTER to exit console.
```

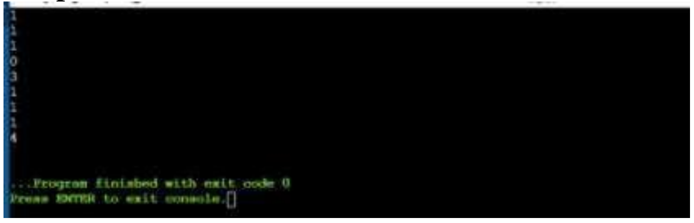
8. Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

Code:

```
#include <iostream>
using namespace std;class MyCircularQueue {
private:
int *queue; // Array to store elements
int front; // Index of the front element
int rear; // Index of the rear element
int size; // Maximum size of the queue
int count; // Current number of elements in the queue
public:
// Constructor to initialize the circular queue with a size of k
MyCircularQueue(int k) {
size = k;
queue = new int[size];
front = 0;
rear = -1;
count = 0;
}
// Destructor to clean up allocated memory
~MyCircularQueue() {
delete[] queue;
}
// Inserts an element into the circular queue
bool enQueue(int value) {
if (isFull()) {
return false; // Queue is full
}
rear = (rear + 1) % size; // Move rear to the next position
queue[rear] = value;
count++;
return true;
}
// Deletes an element from the circular queue
bool deQueue() {
if (isEmpty()) {
```

```
return false; // Queue is empty
}
front = (front + 1) % size; // Move front to the next position
count--;
return true;
}
// Gets the front item from the queue
int Front() {
if (isEmpty()) {
return -1; // Queue is empty
}
return queue[front];
}
// Gets the last item from the queue
int Rear() {
if (isEmpty()) {
return -1; // Queue is empty
}
return queue[rear];
}
// Checks whether the circular queue is empty
bool isEmpty() {
return count == 0;
}
// Checks whether the circular queue is full
bool isFull() {
return count == size;
}
};
// Main function to test the implementation
int main() {
MyCircularQueue q(3); // Initialize a circular queue with size 3
cout << q.enQueue(1) << endl; // Returns true
cout << q.enQueue(2) << endl; // Returns true
cout << q.enQueue(3) << endl; // Returns true
cout << q.enQueue(4) << endl; // Returns false (queue is full)
cout << q.Rear() << endl; // Returns 3
cout << q.isFull() << endl; // Returns true
cout << q.deQueue() << endl; // Returns true
cout << q.enQueue(4) << endl; // Returns true
cout << q.Rear() << endl; // Returns 4
return 0;
}
```

Output:



```
1
1
1
0
3
1
1
1
4


...Program finished with exit code 0
Press ENTER to exit console.
```

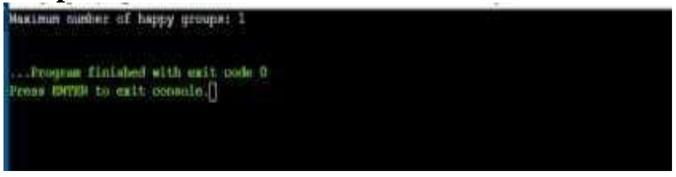
Hard

9. Maximum Number of Groups Getting Fresh Donuts

Code:

```
#include <iostream>
using namespace std;
int maxHappyGroups(int batchSize, int groups[], int n) {
// Sort groups manually in descending order (bubble sort for simplicity)
for (int i = 0; i < n - 1; ++i) {
for (int j = 0; j < n - i - 1; ++j) {
if (groups[j] < groups[j + 1]) {
// Swap the elements
int temp = groups[j];
groups[j] = groups[j + 1];
groups[j + 1] = temp;
} } }
int happyGroups = 0; // To store the number of happy groups
int remainingDonuts = 0; // To store the remaining donuts after serving groups
// Process each group in sorted order
for (int i = 0; i < n; ++i) {
int group = groups[i];
// If the group size is less than or equal to the remaining donuts, serve it fresh
if (group <= remainingDonuts) {
happyGroups++;
remainingDonuts -= group; // Decrease the number of remaining donuts
} else {
// Otherwise, we can't serve the group fresh donuts
remainingDonuts = batchSize - group; // Store the leftover donuts after
serving
} }
return happyGroups;
}
int main() {
int batchSize = 3;
int groups[] = {1, 2, 3, 4, 5, 6};
int n = sizeof(groups) / sizeof(groups[0]);
int result = maxHappyGroups(batchSize, groups, n);
cout << "Maximum number of happy groups: " << result << endl;
return 0;
}
```

Output:



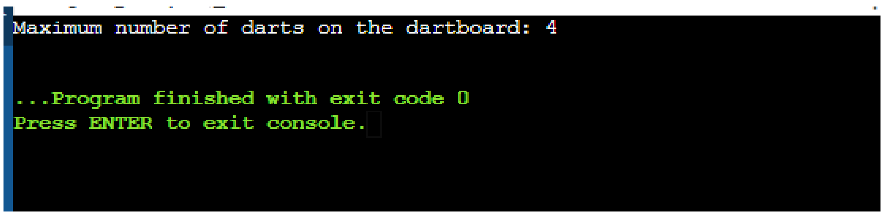
```
Maximum number of happy groups: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

10. Maximum number of Darts Inside of a Circular Dartboard

Code:

```
#include <iostream>
using namespace std;
// Function to check if a point is within the circle
bool isInCircle(int x, int y, int cx, int cy, int r) {
// Calculate the squared distance to avoid using sqrt
return ( (x - cx) * (x - cx) + (y - cy) * (y - cy) ) <= r * r;
}
// Function to calculate the maximum darts that can be inside a circle of radius r
int maxDartsInCircle(int darts[][2], int n, int r) {
int maxDarts = 0;
// Check every pair of darts to find possible circle centers
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];
// Find the midpoint of the two darts
int midX = (x1 + x2) / 2;
int midY = (y1 + y2) / 2;
// Count how many darts are within the circle with center (midX, midY)
int count = 0;
for (int k = 0; k < n; k++) {
if (isInCircle(darts[k][0], darts[k][1], midX, midY, r)) {
count++;
}
}
maxDarts = maxDarts > count ? maxDarts : count; // Update max darts if
needed
}
}
return maxDarts;
}
int main() {
// Input: array of dart positions and radius of the dartboard
int darts[][2] = {{-2,0},{2,0},{0,2},{0,-2}};
int n = 4; // Number of darts
int r = 2; // Radius of the dartboard
// Call the function and display the result
int result = maxDartsInCircle(darts, n, r);
cout << "Maximum number of darts on the dartboard: " << result << endl;
return 0;
}
```

Output



```
Maximum number of darts on the dartboard: 4


...Program finished with exit code 0
Press ENTER to exit console.
```