## Question 1: Sales Report Analysis

**Difficulty:** Easy
**Topics:** Arrays, Loops

### *Problem Description:*

A store records daily sales for a given period. Management wants to analyze:

- The day with the **highest sales** and its value.
- The day with the **lowest sales** and its value.
- The **average sales** for the given days.

Write a program that takes the number of days and their sales values as input and outputs these details.

### *Example:*

Input:

```
n = 5
sales = [1200, 4500, 3200, 1500, 6000]
```

Output:

```
Highest Sales: Day 5 with 6000
Lowest Sales: Day 1 with 1200
Average Sales: 3280
```

### *Constraints:*

- $1 \leq n \leq 31$
- $1 \leq sales[i] \leq 10^6$
- Do not use built-in functions for max, min, or average.

### *C++ Solution:*

```cpp
// Language: C++
// Question 1: Sales Report Analysis

#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main() {
    int n;
    cout << "Enter number of days: ";
    cin >> n;

    // Validate constraint: 1 ≤ n ≤ 1000
    if (n < 1 || n > 1000) {
        cout << "Invalid number of days! Must be between 1 and 1000.\n";
        return 0;
    }

    int sales[1000];  // Fixed size array to satisfy constraint
    cin.ignore(); // To ignore newline after n

    cout << "Enter sales for each day (separated by space or comma): ";
    string inputLine;
    getline(cin, inputLine); // Read full line of sales

    // Parse input manually to handle both spaces and commas
    stringstream ss(inputLine);
    string temp;
    int i = 0;
    while (getline(ss, temp, ' ') && i < n) {
        // Remove any trailing comma
        if (temp.back() == ',') {
            temp.pop_back();
        }
        sales[i] = stoi(temp);
        i++;
    }

    // Check if we got enough values
    if (i != n) {
        cout << "Error: Expected " << n << " sales values.\n";
        return 0;
    }

    // Initialize max, min, and total
    int maxSales = sales[0], minSales = sales[0];
    int maxDay = 1, minDay = 1;
    long long total = 0;
```

```
    // Process sales to find max, min, and total
    for (int j = 0; j < n; j++) {
        if (sales[j] > maxSales) {
            maxSales = sales[j];
            maxDay = j + 1;
        }
        if (sales[j] < minSales) {
            minSales = sales[j];
            minDay = j + 1;
        }
        total += sales[j];
    }

    double average = (double)total / n;

    // Output results clearly
    cout << "Highest Sales: Day " << maxDay << " with " << maxSales <<
endl;
    cout << "Lowest Sales: Day " << minDay << " with " << minSales <<
endl;
    cout << "Average Sales: " << average << endl;

    return 0;
}
```

---

**Approach:**

1. Read n (number of days) and validate constraint (1 ≤ n ≤ 1000).
2. Read all sales in a single line, manually parse spaces and commas.
3. Loop through array to find max, min, and total sales.
4. Compute average and print results.

Time Complexity: O(n)  // One pass through sales array
Space Complexity: O(n) // For storing sales values

programiz.com/cpp-programming/online-compiler/

Error

YouTube | www.google.com | AI Text Classifier - O... | chat gpt | LPU Live | Women in IAFSearc... | Call Bomber In - Be... | New Tab | LPU_Colab | LPU_Colab | View skill tests | All Bookmarks

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

```cpp
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main() {
    int n;
    cout << "Enter number of days: ";
    cin >> n;

    // Validate constraint: 1 ≤ n ≤ 1000
    if (n < 1 || n > 1000) {
        cout << "Invalid number of days! Must be between 1 and 1000.\n";
        return 0;
    }

    int sales[1000];  // Fixed size array to satisfy constraint
    cin.ignore(); // To ignore newline after n

    cout << "Enter sales for each day (separated by space or comma): ";
    string inputLine;
    getline(cin, inputLine); // Read full line of sales

    // Parse input manually to handle both spaces and commas
    stringstream ss(inputLine);
    string temp;
    int i = 0;
    while (getline(ss, temp, ' ') && i < n) {
        // Remove any trailing comma
        if (temp.back() == ',') {
            temp.pop_back();
        }
        sales[i] = stoi(temp);
        i++;
    }
```

Output

Clear

```
Enter number of days: 5
Enter sales for each day (separated by space or comma): 1200, 4500, 3200, 1500, 6000
Highest Sales: Day 5 with 6000
Lowest Sales: Day 1 with 1200
Average Sales: 3280


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

programiz.com/cpp-programming/online-compiler/

Error

YouTube | www.google.com | AI Text Classifier - O... | chat gpt | LPU Live | Women in IAFSearc... | Call Bomber In - Be... | New Tab | LPU_Colab | LPU_Colab | View skill tests | All Bookmarks

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

```cpp
    }

    // Check if we got enough values
    if (i != n) {
        cout << "Error: Expected " << n << " sales values.\n";
        return 0;
    }

    // Initialize max, min, and total
    int maxSales = sales[0], minSales = sales[0];
    int maxDay = 1, minDay = 1;
    long long total = 0;

    // Process sales to find max, min, and total
    for (int j = 0; j < n; j++) {
        if (sales[j] > maxSales) {
            maxSales = sales[j];
            maxDay = j + 1;
        }
        if (sales[j] < minSales) {
            minSales = sales[j];
            minDay = j + 1;
        }
        total += sales[j];
    }

    double average = (double)total / n;

    // Output results clearly
    cout << "Highest Sales: Day " << maxDay << " with " << maxSales << endl;
    cout << "Lowest Sales: Day " << minDay << " with " << minSales << endl;
    cout << "Average Sales: " << average << endl;

    return 0;
}
```

Output

Clear

```
Enter number of days: 5
Enter sales for each day (separated by space or comma): 1200, 4500, 3200, 1500, 6000
Highest Sales: Day 5 with 6000
Lowest Sales: Day 1 with 1200
Average Sales: 3280


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

Output

```
Enter number of days: 5
Enter sales for each day (separated by space or comma): 1200, 4500, 3200, 1500, 6000
Highest Sales: Day 5 with 6000
Lowest Sales: Day 1 with 1200
Average Sales: 3280


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

## Question 2: Student Ranking

### *Scenario*

A teacher wants to arrange student marks in **descending order** without using built-in sort functions.

### *Problem*

Given an array of marks, sort them using **Selection Sort** and display the sorted list.

### *Constraints*

- 1 ≤ number of students ≤ 1000
- Marks are integers between 0 and 100
- **No built-in sort functions allowed.**

### *Input Format*

- First line: An integer n (number of students)
- Second line: n integers representing marks of students

### *Output Format*

- Sorted marks in descending order.

### *Example:*

**Input:**

```
Enter number of students: 5
Enter marks: 78 92 56 88 70
```

**Output:**

```
Sorted marks in descending order: 92 88 78 70 56
```

### *C++ Solution*

```
// Question 2: Student Ranking
// Language: C++

#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Enter number of students: ";
    cin >> n;

    int marks[n];
    cout << "Enter marks: ";
    for (int i = 0; i < n; i++) {
        cin >> marks[i];
    }

    // Selection sort in descending order
    for (int i = 0; i < n - 1; i++) {
        int maxIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (marks[j] > marks[maxIndex]) {
                maxIndex = j;
            }
        }
        // Swap elements
        int temp = marks[i];
        marks[i] = marks[maxIndex];
        marks[maxIndex] = temp;
    }

    cout << "Sorted marks in descending order: ";
    for (int i = 0; i < n; i++) {
        cout << marks[i] << " ";
    }
    cout << endl;
```

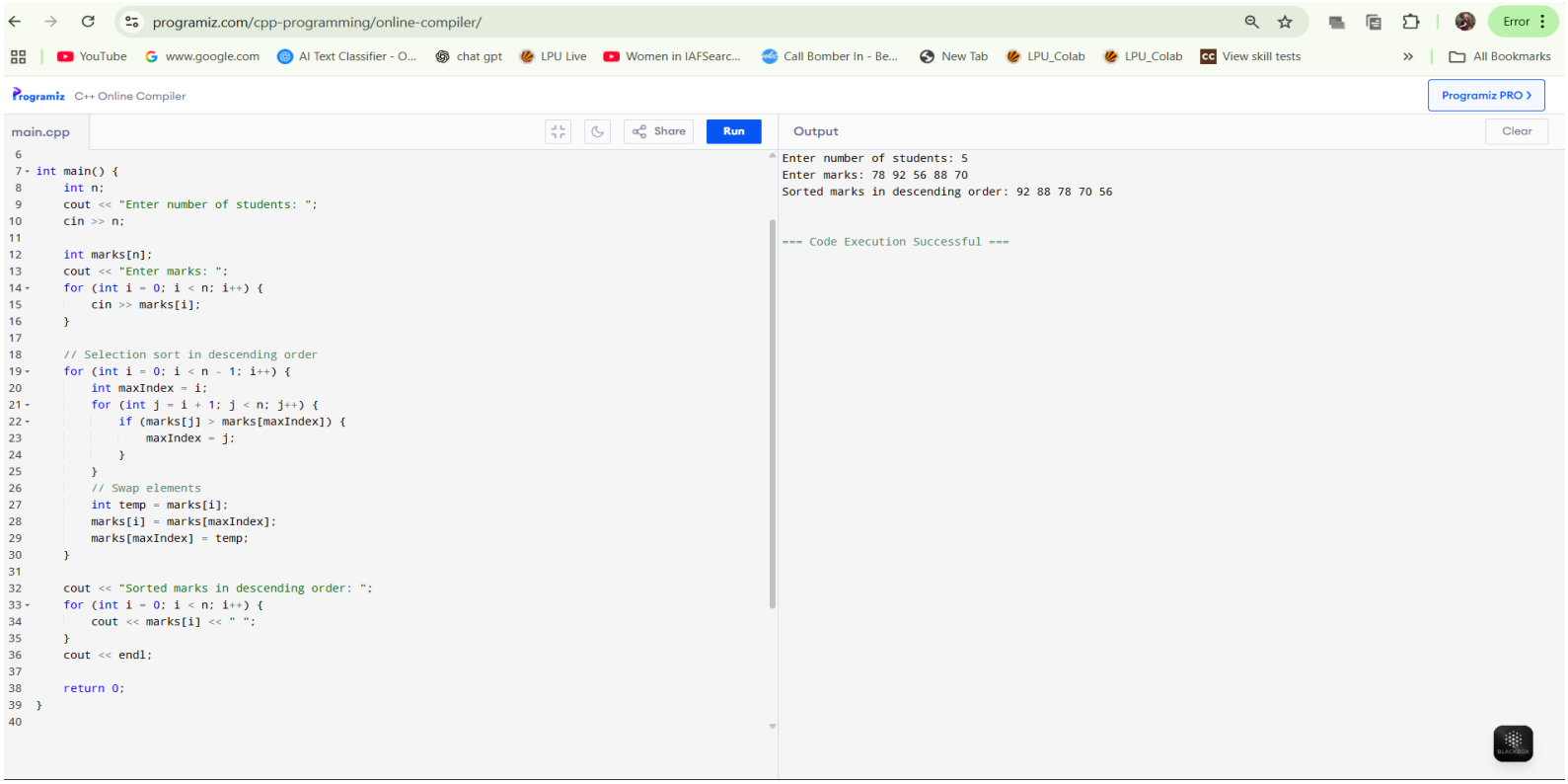
```
    return 0;
}
```

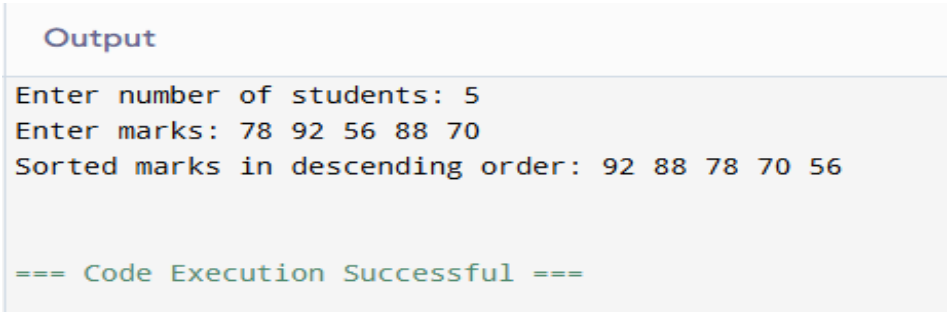
---

**Approach:**

- Read number of students and their marks.
- Apply Selection Sort to arrange marks in descending order.
- Print the sorted list.

Time Complexity: O(n^2) (nested loops for selection sort)
Space Complexity: O(1) (sorting in-place)



## Question 3: Rotate an Array

### *Scenario*

In a game leaderboard, the top **k** players are moved to the end after each round.

### *Problem*

Write a program to rotate the array of player IDs by **k positions** without using extra space.

### *Constraints*

- 1 ≤ n ≤ 10^6
- 0 ≤ k < n
- **O(1) extra space**, O(n) time complexity

### *Input Format*

- First line: An integer n (number of players)
- Second line: n integers representing player IDs
- Third line: An integer k (number of positions to rotate)

### *Output Format*

- Rotated array after shifting first **k** elements to the end.

### *Example:*

**Input:**

```
Enter number of players: 6
Enter player IDs: 10 20 30 40 50 60
Enter k: 2
```

**Output:**

```
Rotated Array: 30 40 50 60 10 20
```

### *C++ Solution:*

```cpp
// Question 3: Rotate an Array
// Language: C++
#include <iostream>
using namespace std;
// Function to reverse part of the array
void reverseArray(int arr[], int start, int end) {
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
int main() {
    int n, k;
    cout << "Enter number of players: ";
    cin >> n;

    int arr[n];
    cout << "Enter player IDs: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "Enter k: ";
    cin >> k;

    // Normalize k in case it's larger than n
    k = k % n;

    // Rotate using reversal algorithm (O(1) extra space)
    reverseArray(arr, 0, k - 1);
    reverseArray(arr, k, n - 1);
    reverseArray(arr, 0, n - 1);

    cout << "Rotated Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
```

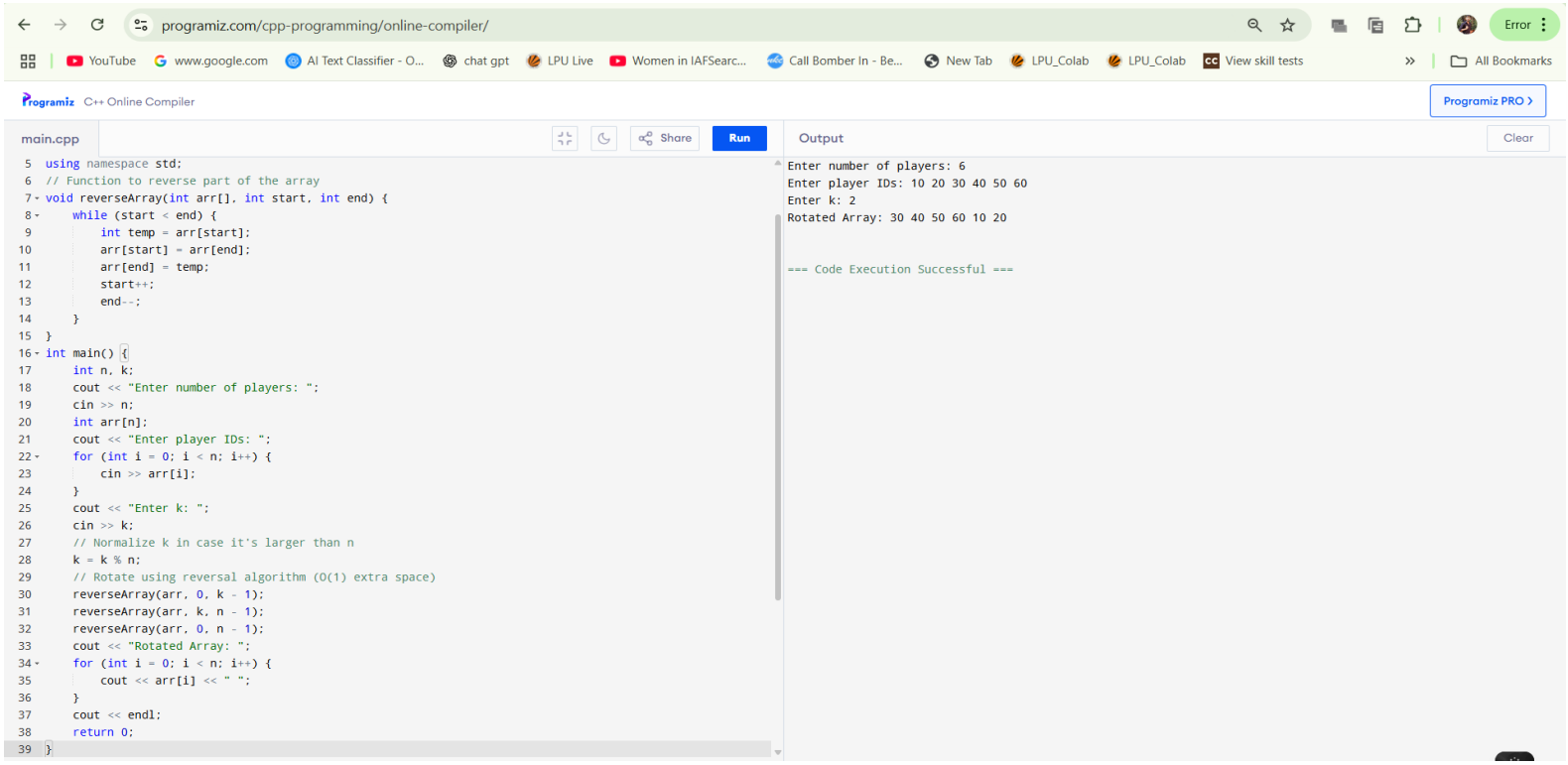
```
}
```

---

**Approach:**

- Use the reversal algorithm:
    1. Reverse the first k elements.
    2. Reverse the remaining elements.
    3. Reverse the entire array.
- This ensures O(n) time and O(1) space.

Time Complexity: O(n)
Space Complexity: O(1)




```
Output

Enter number of players: 6
Enter player IDs: 10 20 30 40 50 60
Enter k: 2
Rotated Array: 30 40 50 60 10 20


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

## Question 4: Find K Smallest Elements (Heap)

***Scenario:***

You are given the prices of items in a shop. You need to find the **k cheapest items.**

***Problem Statement:***

Given an array of item prices of size n and an integer k, write a program to find the **k smallest elements using a manually implemented Min Heap.**

***Constraints:***

- 1 ≤ n ≤ 10^5
- 1 ≤ k ≤ n
- No built-in heap or priority_queue allowed.
- Must manually implement Min Heap.
- **Time Complexity:** O(n log n)

***Input Format:***

- First line: integer n (number of items)
- Second line: n integers representing item prices
- Third line: integer k

***Output Format:***

- Print k smallest elements in ascending order

***Example:***

**Input:**

```
Enter number of items: 6
Enter prices: 40 10 20 50 30 5
Enter k: 3
```

**Output:**

```
K smallest elements: 5 10 20
```

***C++ Solution:***

```
// Question 4: Find K Smallest Elements using Min Heap
// Language: C++

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

// Function to swap two elements in the heap
void swapElements(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Function to heapify a subtree at index i in Min Heap
void minHeapify(vector<int> &heap, int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && heap[left] < heap[smallest]) {
        smallest = left;
    }
    if (right < n && heap[right] < heap[smallest]) {
        smallest = right;
    }

    if (smallest != i) {
        swapElements(heap[i], heap[smallest]);
        minHeapify(heap, n, smallest);
    }
}

// Function to build Min Heap
void buildMinHeap(vector<int> &heap, int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
```

```
        minHeapify(heap, n, i);
    }
}

// Function to extract minimum element from heap
int extractMin(vector<int> &heap, int &n) {
    if (n <= 0) return -1;

    int root = heap[0];
    heap[0] = heap[n - 1];
    n--; // Reduce size
    minHeapify(heap, n, 0);
    return root;
}

int main() {
    int n, k;
    cout << "Enter number of items: ";
    cin >> n;

    vector<int> prices(n);
    cout << "Enter prices: ";
    for (int i = 0; i < n; i++) {
        cin >> prices[i];
    }

    cout << "Enter k: ";
    cin >> k;

    // Build Min Heap
    buildMinHeap(prices, n);

    cout << "K smallest elements: ";
    for (int i = 0; i < k; i++) {
        cout << extractMin(prices, n) << " ";
    }
    cout << endl;

    return 0;
```

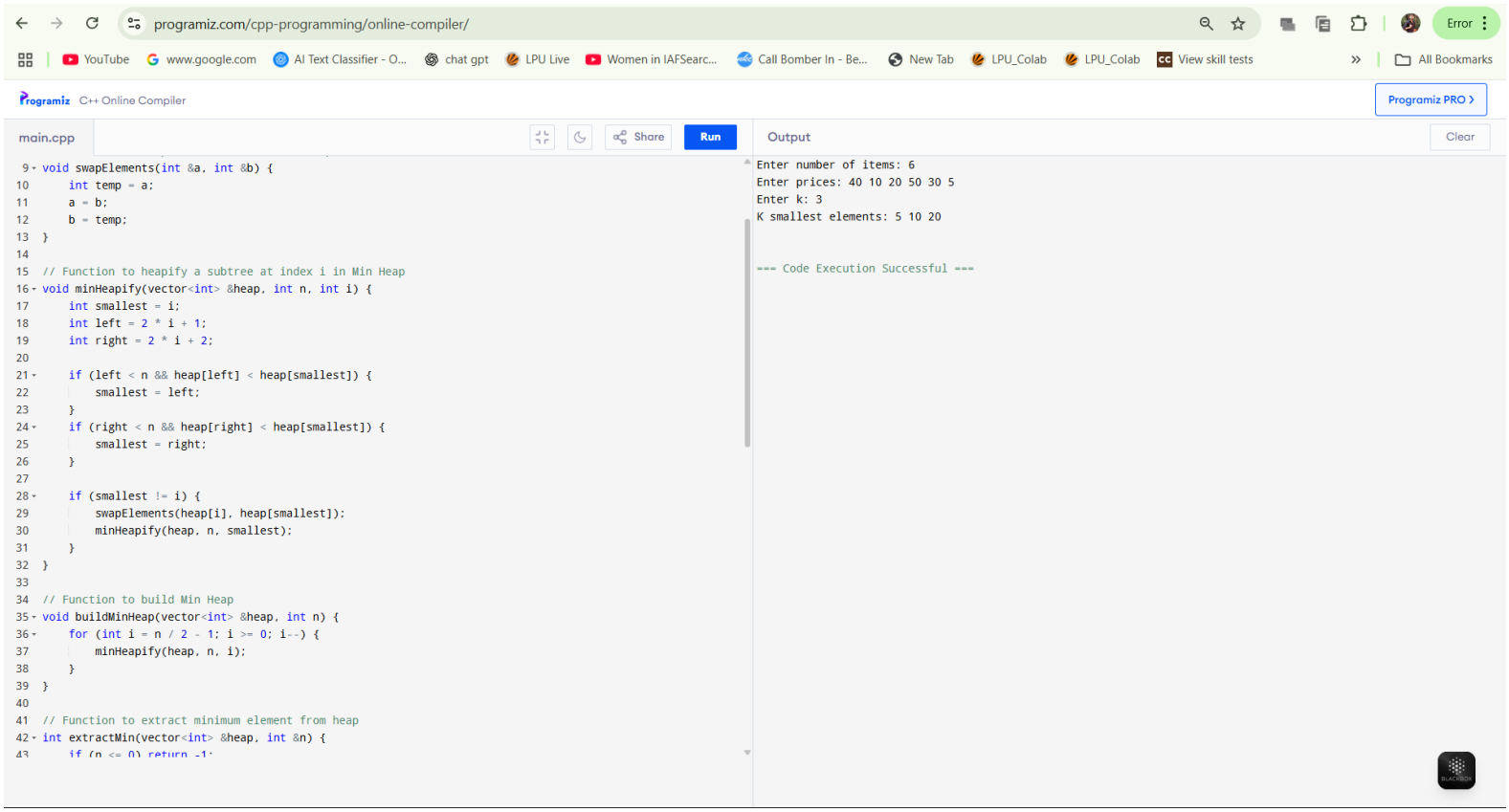
```
}
```

---

**Approach:**

1. Build a Min Heap from the input array (O(n)).
2. Extract the minimum element k times (O(k log n)).
3. Total time complexity: O(n + k log n), which satisfies O(n log n) for large k.
4. No extra space except for input array (O(1) additional space).

```
// Time Complexity: O(n + k log n)
// Space Complexity: O(1) (In-place heap implementation)
```



programiz.com/cpp-programming/online-compiler/

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

```cpp
int extractMin(vector<int> &heap, int &n) {
    if (n <= 0) return -1;

    int root = heap[0];
    heap[0] = heap[n - 1];
    n--; // Reduce size
    minHeapify(heap, n, 0);
    return root;
}

int main() {
    int n, k;
    cout << "Enter number of items: ";
    cin >> n;

    vector<int> prices(n);
    cout << "Enter prices: ";
    for (int i = 0; i < n; i++) {
        cin >> prices[i];
    }

    cout << "Enter k: ";
    cin >> k;

    // Build Min Heap
    buildMinHeap(prices, n);

    cout << "K smallest elements: ";
    for (int i = 0; i < k; i++) {
        cout << extractMin(prices, n) << " ";
    }
    cout << endl;

    return 0;
}
```

Output | Clear

```
Enter number of items: 6
Enter prices: 40 10 20 50 30 5
Enter k: 3
K smallest elements: 5 10 20


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

## Output

```
Enter number of items: 6
Enter prices: 40 10 20 50 30 5
Enter k: 3
K smallest elements: 5 10 20


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

## Question 5: Username Validator

**Scenario:**
A website has strict rules for creating usernames to ensure consistency and avoid invalid entries.

**Rules:**

- Username must start with a **letter (A-Z or a-z)**.
- It must have at least **5 characters**.
- It can contain only **letters and digits** (no spaces, symbols, or special characters).

***Problem Statement:***
Given a string representing a username, check if it meets the given rules.

- If valid → print **"Valid Username"**
- If invalid → print **"Invalid Username"**

***Constraints:***

- 5 ≤ length ≤ 20
- No use of **built-in regex functions**.
- Only loops and **basic character checks** are allowed.

***Input Format:***

- A single string: the username

***Output Format:***

- **Valid Username** or **Invalid Username**

***Example:***

**Input 1:**

Enter username: John123

**Output 1:**
Valid Username

**Input 2:**

Enter username: 123John

**Output 2:**

Invalid Username

***C++ Solution:***

```
// Question 5: Username Validator
// Language: C++

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

// Function to check if a character is a letter
bool isLetter(char ch) {
    return ( (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') );
}

// Function to check if a character is a digit
bool isDigit(char ch) {
    return (ch >= '0' && ch <= '9');
}

// Function to validate the username
bool isValidUsername(string username) {
    int length = username.length();

    // Rule 1: Length should be between 5 and 20
```

```
    if (length < 5 || length > 20) return false;
    // Rule 2: First character must be a letter
    if (!isLetter(username[0])) return false;

    // Rule 3: All characters must be letters or digits
    for (int i = 0; i < length; i++) {
        if (!isLetter(username[i]) && !isDigit(username[i])) {
            return false;
        }
    }
    return true;
}

int main() {
    string username;
    cout << "Enter username: ";
    cin >> username;

    if (isValidUsername(username)) {
        cout << "Valid Username" << endl;
    } else {
        cout << "Invalid Username" << endl;
    }

    return 0;
}
```

---

**Approach:**

1. Check if the username length is between 5 and 20.
2. Ensure the first character is a letter.
3. Traverse the string and confirm all characters are letters or digits.
4. If all conditions pass → Valid, else Invalid.

```
// Time Complexity: O(n) where n = length of username
// Space Complexity: O(1) (only a few extra variables)
```

programiz.com/cpp-programming/online-compiler/

YouTube | www.google.com | AI Text Classifier - O... | chat gpt | LPU Live | Women in IAFSearc... | Call Bomber In - Be... | New Tab | LPU_Colab | LPU_Colab | View skill tests

Programiz C++ Online Compiler

main.cpp

Share | Run

```cpp
// Function to check if a character is a letter
bool isLetter(char ch) {
    return ( (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') );
}
// Function to check if a character is a digit
bool isDigit(char ch) {
    return (ch >= '0' && ch <= '9');
}
// Function to validate the username
bool isValidUsername(string username) {
    int length = username.length();
    // Rule 1: Length should be between 5 and 20
    if (length < 5 || length > 20) return false;
    // Rule 2: First character must be a letter
    if (!isLetter(username[0])) return false;
    // Rule 3: All characters must be letters or digits
    for (int i = 0; i < length; i++) {
        if (!isLetter(username[i]) && !isDigit(username[i])) {
            return false;
        }
    }
    return true;
}
int main() {
    string username;
    cout << "Enter username: ";
    cin >> username;
    if (isValidUsername(username)) {
        cout << "Valid Username" << endl;
    } else {
        cout << "Invalid Username" << endl;
    }
    return 0;
}
/*
```

Output

```
Enter username: 123John
Invalid Username


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

Output

```
Enter username: 123John
Invalid Username


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

Output

```
Enter username: John123
Valid Username


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

## Question 6: Word Frequency Counter

***Scenario:***

In a chat application, the system needs to analyze messages and determine how many times each word appears. This helps with spam detection, trending word analysis, and keyword tracking.

The counting must be **case-insensitive** (e.g., "Hello" and "hello" should be treated as the same word).

***Problem:***

Given a sentence, count how many times each unique word appears, ignoring case.

***Restrictions:***

- You cannot use **built-in map/dictionary** data structures.
- Implement the counting manually using **arrays or a linked list**.

***Constraints:***

- Sentence length ≤ 1000 characters
- Words are separated by spaces
- **Case-insensitive comparison**
- **No built-in map / unordered_map / dictionary allowed**

***Input Format:***

- A single line containing a sentence

***Output Format:***

- Each unique word and its frequency on a separate line

***Example:***

**Input:**

Enter sentence: Hello hello world World

**Output:**

```
hello : 2
world : 2
```

***C++ Solution:***

```
// Question 6: Word Frequency Counter
// Language: C++
#include <iostream>
#include <string>
using namespace std;

// Convert all characters in a word to lowercase
string toLowerCase(string str) {
    for (int i = 0; i < str.length(); i++) {
        if (str[i] >= 'A' && str[i] <= 'Z') {
            str[i] = str[i] + 32; // convert uppercase to lowercase
        }
    }
    return str;
}

int main() {
    string sentence;
    cout << "Enter sentence: ";
    getline(cin, sentence);

    // Split the sentence into words manually and store in an array
    string words[200];  // Max 200 words assumption
    int wordCount = 0;
    string temp = "";

    for (int i = 0; i <= sentence.length(); i++) {
        if (sentence[i] == ' ' || sentence[i] == '\0') {
            if (!temp.empty()) {
                words[wordCount++] = toLowerCase(temp);
                temp = "";
            }
        } else {
            temp += sentence[i];
```

```
        }
    }

    // Arrays for unique words and their counts
    string uniqueWords[200];
    int counts[200] = {0};
    int uniqueCount = 0;

    // Count frequencies manually
    for (int i = 0; i < wordCount; i++) {
        bool found = false;
        for (int j = 0; j < uniqueCount; j++) {
            if (words[i] == uniqueWords[j]) {
                counts[j]++;
                found = true;
                break;
            }
        }
        if (!found) {
            uniqueWords[uniqueCount] = words[i];
            counts[uniqueCount] = 1;
            uniqueCount++;
        }
    }

    // Display the results
    for (int i = 0; i < uniqueCount; i++) {
        cout << uniqueWords[i] << " : " << counts[i] << endl;
    }

    return 0;
}
```

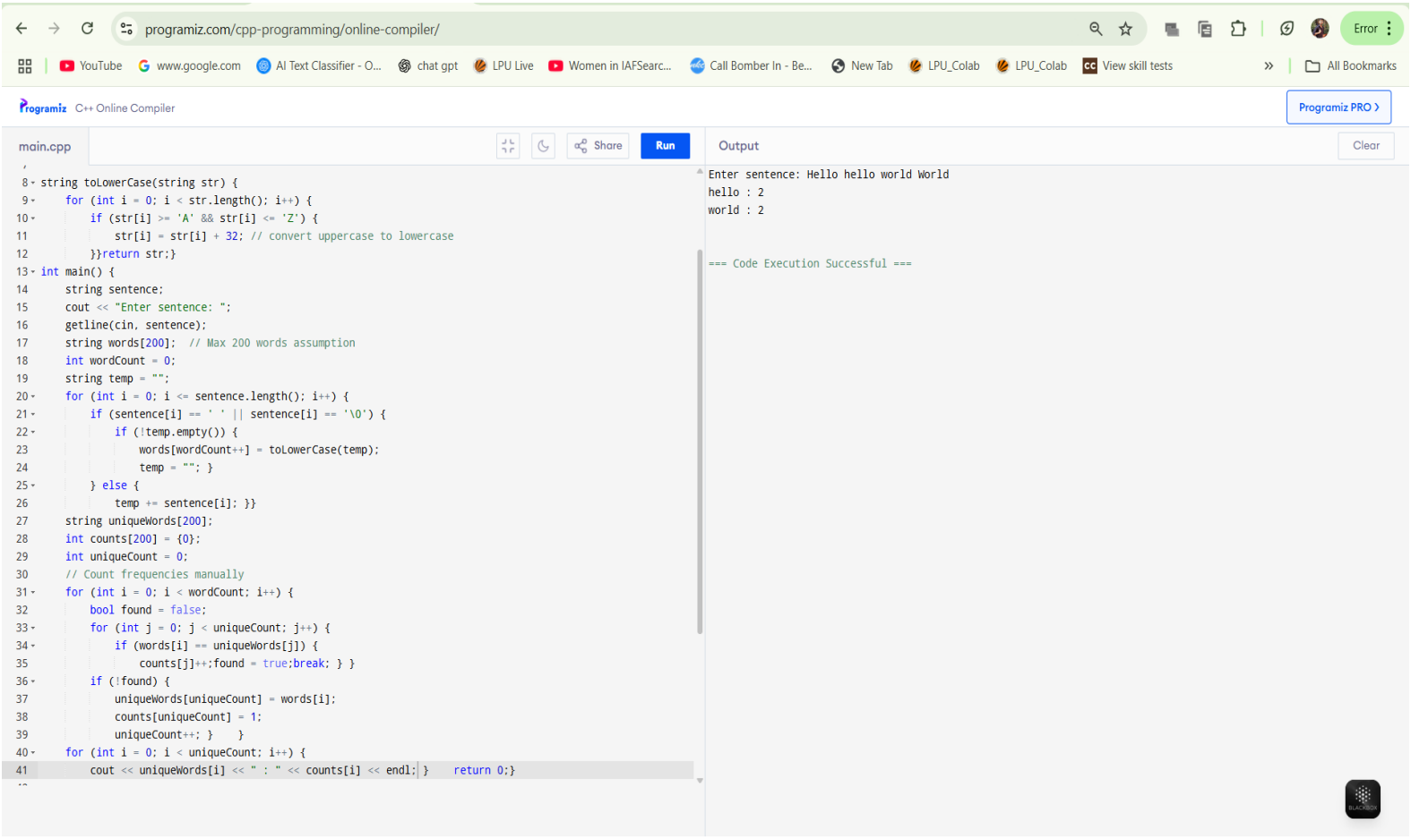
---

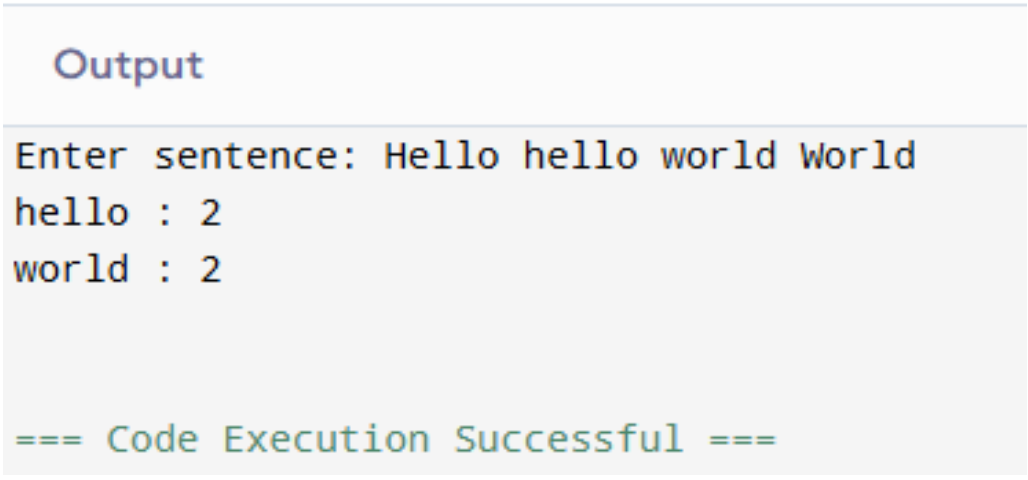
## Approach:

1. Read the entire sentence and split it into words manually using a loop.
2. Convert each word to lowercase for case-insensitive comparison.
3. Use two arrays: one for unique words and another for their frequencies.
4. Count occurrences of each word without using any built-in

map/dictionary.
5. Print each word with its frequency.

// Time Complexity: O(n * m) where n = number of words, m = unique words
// Space Complexity: O(n) for storing words and count



## Question 7: Minimum Window Substring Finder

***Scenario:***

A search engine's query optimizer must highlight the smallest section of text that contains all the characters from the search query. This needs to be efficient because the text can be very large (up to 100,000 characters).

***Problem Statement:***

Given:

- **T** → The text string
- **S** → The set of characters to find

Find the smallest substring of T containing **all characters in S, case-insensitive.**

If no such substring exists, output **"No valid window found".**

***Constraints:***

- 1 ≤ |T| ≤ 10^5
- 1 ≤ |S| ≤ 52
- **No built-in substring search (like find() or regex)**
- Must use **custom sliding window logic**
- **Optimize to O(n) time complexity**

***Input Format:***

Enter text: (string T)
Enter characters to find: (string S)

***Output Format:***

Minimum window substring: <substring>

Or:

No valid window found

***Example:***

**Input:**

```
Enter text: ADOBECODEBANC
Enter characters to find: ABC
```

**Output:**

```
Minimum window substring: BANC
```

***C++ Solution:***

```
// Question 7: Minimum Window Substring Finder
// Language: C++

#include <iostream>


#include <string>
using namespace std;

// Convert character to lowercase manually
char toLowerChar(char ch) {
    if (ch >= 'A' && ch <= 'Z') return ch + 32;
    return ch;
}

int main() {
    string text, pattern;
    cout << "Enter text: ";
    getline(cin, text);
    cout << "Enter characters to find: ";
    getline(cin, pattern);

    // Convert both text and pattern to lowercase manually
    for (int i = 0; i < text.length(); i++) text[i] =
toLowerChar(text[i]);
    for (int i = 0; i < pattern.length(); i++) pattern[i] =
toLowerChar(pattern[i]);

    // Frequency arrays (26 letters + case handled by lowercase
conversion)
```

```
    int freqPattern[256] = {0};
    int freqWindow[256] = {0};

    // Fill pattern frequency
    for (int i = 0; i < pattern.length(); i++) {
        freqPattern[(int)pattern[i]]++;
    }

    int required = 0; // Number of unique characters in pattern
    for (int i = 0; i < 256; i++) {
        if (freqPattern[i] > 0) required++;
    }

    int left = 0, right = 0;
    int formed = 0;
    int minLen = text.length() + 1;
    int startIndex = -1;

    // Sliding window logic
    while (right < text.length()) {
        char ch = text[right];
        freqWindow[(int)ch]++;

        // Check if this character count matches pattern requirement
        if (freqPattern[(int)ch] > 0 && freqWindow[(int)ch] ==
freqPattern[(int)ch]) {
            formed++;
        }

        // Shrink window from left if all required characters are
included
        while (formed == required && left <= right) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                startIndex = left;
```

```
            }

            char leftChar = text[left];
            freqWindow[(int)leftChar]--;
            if (freqPattern[(int)leftChar] > 0 &&
freqWindow[(int)leftChar] < freqPattern[(int)leftChar]) {
                formed--;
            }
            left++;
        }
        right++;
    }

    if (startIndex == -1) {
        cout << "No valid window found" << endl;
    } else {
        cout << "Minimum window substring: ";
        for (int i = startIndex; i < startIndex + minLen; i++) {
            cout << text[i];
        }
        cout << endl;
    }

    return 0;
}
```

---

## Approach:

1. Convert both text and pattern to lowercase for case-insensitive comparison.
2. Use frequency arrays (size 256) to track required characters and current window.
3. Apply sliding window: expand right until all required chars are present, then shrink from left.
4. Keep track of the smallest valid window length and starting index.
5. Print result if found; else print "No valid window found".

```
// Time Complexity: O(n) where n = length of text
// Space Complexity: O(1) (constant size frequency arrays)
```

programiz.com/cpp-programming/online-compiler/

Programiz C++ Online Compiler

main.cpp

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

// Convert character to lowercase manually
char toLowerChar(char ch) {
    if (ch >= 'A' && ch <= 'Z') return ch + 32;
    return ch;
}

int main() {
    string text, pattern;
    cout << "Enter text: ";
    getline(cin, text);
    cout << "Enter characters to find: ";
    getline(cin, pattern);

    // Convert both text and pattern to lowercase manually
    for (int i = 0; i < text.length(); i++) text[i] = toLowerChar(text[i]);
    for (int i = 0; i < pattern.length(); i++) pattern[i] = toLowerChar(pattern[i]);

    // Frequency arrays (26 letters + case handled by lowercase conversion)
    int freqPattern[256] = {0};
    int freqWindow[256] = {0};

    // Fill pattern frequency
    for (int i = 0; i < pattern.length(); i++) {
        freqPattern[(int)pattern[i]]++;
    }

    int required = 0; // Number of unique characters in pattern
    for (int i = 0; i < 256; i++) {
        if (freqPattern[i] > 0) required++;
    }

```

Output

```
Enter text: ADOBECODEBANC
Enter characters to find: ABC
Minimum window substring: banc


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

programiz.com/cpp-programming/online-compiler/

Programiz C++ Online Compiler

main.cpp

```cpp

    int left = 0, right = 0;
    int formed = 0;
    int minLen = text.length() + 1;
    int startIndex = -1;
    while (right < text.length()) {
        char ch = text[right];
        freqWindow[(int)ch]++;
                if (freqPattern[(int)ch] > 0 && freqWindow[(int)ch] == freqPattern[(int)ch]) {
            formed++;
        }
        while (formed == required && left <= right) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                startIndex = left;
            }
            char leftChar = text[left];
            freqWindow[(int)leftChar]--;
            if (freqPattern[(int)leftChar] > 0 && freqWindow[(int)leftChar] < freqPattern[(int)leftChar])
                {
                formed--;
            }
            left++;}
        right++;}
    if (startIndex == -1) {
        cout << "No valid window found" << endl;
    } else {
        cout << "Minimum window substring: ";
        for (int i = startIndex; i < startIndex + minLen; i++) {
            cout << text[i];
        }
        cout << endl;
    }
    return 0;
}
```

Output

```
Enter text: ADOBECODEBANC
Enter characters to find: ABC
Minimum window substring: banc


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

```
Output

Enter text: ADOBECODEBANC
Enter characters to find: ABC
Minimum window substring: banc


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

## Question 8: Longest Repeating Character Replacement

***Scenario:***
A text compression algorithm can replace **at most k characters in a substring** to make all characters identical. We need to figure out the maximum possible length of such a substring.

***Problem Statement:***
Given:

- `str` → a string consisting of **uppercase letters A-Z**
- k → maximum number of characters you can replace

**Find:**
The length of the **longest substring** where replacing at most k characters will make all characters the same.

***Constraints:***

- `1 ≤ |str| ≤ 10^5`
- Only uppercase letters allowed (A–Z)
- **O(n)** or **O(n log n)** solution required — brute force **not allowed**

***Input Format:***

```
Enter string: (only uppercase letters)
Enter k: (integer)
```

### *Output Format:*

```
Longest substring length: <value>
```

### *Example:*

**Input:**

```
Enter string: AABABBA
Enter k: 1
```

**Output:**

```
Longest substring length: 4
```

**Explanation:** Replace one B with A → substring AABA or ABBA.

### *C++ Solution:*

```
// Question 8: Longest Repeating Character Replacement
// Language: C++

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

int main() {
    string str;
    int k;
    cout << "Enter string: ";
    cin >> str;
    cout << "Enter k: ";
    cin >> k;

    int n = str.length();
    int freq[26] = {0}; // Frequency of letters A-Z

    int left = 0, maxCount = 0, result = 0;
```

```
    // Sliding window approach
    for (int right = 0; right < n; right++) {
        freq[str[right] - 'A']++;

        // Track the most frequent character in the window
        if (freq[str[right] - 'A'] > maxCount) {
            maxCount = freq[str[right] - 'A'];
        }

        // If current window size - most frequent char > k, shrink
from left
        while ((right - left + 1) - maxCount > k) {
            freq[str[left] - 'A']--;
            left++;
        }

        // Update result
        if (right - left + 1 > result) {
            result = right - left + 1;
        }
    }

    cout << "Longest substring length: " << result << endl;

    return 0;
}
```

---

**Approach:**

1. Use a sliding window to track the current substring.
2. Maintain frequency of each character and the max frequency in the window.
3. If (window length - max frequency) > k, shrink the window from the left.
4. Keep track of the maximum valid window size.

```
// Time Complexity: O(n) where n = length of string
// Space Complexity: O(1) (constant array size of 26)
```

programiz.com/cpp-programming/online-compiler/

Programiz C++ Online Compiler

main.cpp

```cpp
int main() {
    string str;
    int k;
    cout << "Enter string: ";
    cin >> str;
    cout << "Enter k: ";
    cin >> k;

    int n = str.length();
    int freq[26] = {0}; // Frequency of letters A-Z

    int left = 0, maxCount = 0, result = 0;

    // Sliding window approach
    for (int right = 0; right < n; right++) {
        freq[str[right] - 'A']++;

        // Track the most frequent character in the window
        if (freq[str[right] - 'A'] > maxCount) {
            maxCount = freq[str[right] - 'A'];
        }

        // If current window size - most frequent char > k, shrink from left
        while ((right - left + 1) - maxCount > k) {
            freq[str[left] - 'A']--;
            left++;
        }// Update result
        if (right - left + 1 > result) {
            result = right - left + 1;
        }}
    cout << "Longest substring length: " << result << endl;

    return 0;
}
```

Output

```
Enter string: AABABBA
Enter k: 1
Longest substring length: 4


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

## Output

```
Enter string: AABABBA
Enter k: 1
Longest substring length: 4


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

## Question 9: Playlist Manager

### *Scenario:*

You're designing a music app where each playlist is stored as a **singly linked list**. Each node in the linked list represents a song, storing:

- The song's name
- A pointer to the next song

You need to implement basic playlist operations:

✓ Add song at the end

✓ Delete song by name

✓ Display all songs

### *Problem Statement:*

Write a program to manage a playlist using a **singly linked list** with the above operations.

### *Constraints:*

- `1 ≤ number of songs ≤ 10^4`
- Song names ≤ 50 characters
- No built-in linked list class allowed (manual implementation required)

### *Input Format:*

```
Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
```

### *Output Format:*

Songs displayed in the order they were added.

### *Example:*

**Input/Output Flow:**

```
1. Add song
Enter song name: ShapeOfYou
1. Add song
Enter song name: Perfect
1. Add song
Enter song name: Despacito
3. Display playlist
Playlist: ShapeOfYou -> Perfect -> Despacito
2. Delete song
Enter song name to delete: Perfect
3. Display playlist
Playlist: ShapeOfYou -> Despacito
4. Exit
```

### *C++ Solution:*

```
// Question 9: Playlist Manager
// Language: C++

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

```
// Node structure for the linked list
struct Node {
    string song;
    Node* next;

    Node(string name) {
        song = name;
        next = nullptr;
    }
};

// Playlist class with basic operations
class Playlist {
private:
    Node* head;
public:
    Playlist() {
        head = nullptr;
    }

    // Add a song at the end
    void addSong(string name) {
        Node* newNode = new Node(name);
        if (head == nullptr) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
        cout << "Song added: " << name << endl;
    }

    // Delete a song by name
    void deleteSong(string name) {
        if (head == nullptr) {
```

```
        cout << "Playlist is empty!" << endl;
        return;
    }

    // If the song to delete is the first node
    if (head->song == name) {
        Node* temp = head;
        head = head->next;
        delete temp;
        cout << "Song deleted: " << name << endl;
        return;
    }

    // Search for the song in the list
    Node* temp = head;
    Node* prev = nullptr;
    while (temp != nullptr && temp->song != name) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Song not found!" << endl;
    } else {
        prev->next = temp->next;
        delete temp;
        cout << "Song deleted: " << name << endl;
    }
}

// Display all songs
void display() {
    if (head == nullptr) {
        cout << "Playlist is empty!" << endl;
        return;
    }

    Node* temp = head;
    cout << "Playlist: ";
```

```
        while (temp != nullptr) {
            cout << temp->song;
            if (temp->next != nullptr) cout << " -> ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Playlist playlist;
    int choice;
    string name;

    do {
        cout << "\nMenu:\n1. Add song\n2. Delete song\n3. Display
playlist\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(); // Ignore newline character after number input

        switch (choice) {
            case 1:
                cout << "Enter song name: ";
                getline(cin, name);
                playlist.addSong(name);
                break;
            case 2:
                cout << "Enter song name to delete: ";
                getline(cin, name);
                playlist.deleteSong(name);
                break;
            case 3:
                playlist.display();
                break;
            case 4:
                cout << "Exiting Playlist Manager." << endl;
                break;
            default:
```

```
                cout << "Invalid choice! Try again." << endl;
            }
        } while (choice != 4);

        return 0;
    }
```

---

**Approach:**

1. Implemented a singly linked list manually (Node structure).
2. addSong(): Adds a song at the end by traversing the list.
3. deleteSong(): Deletes the node matching the given name.
4. display(): Prints all songs in the playlist.

```
// Time Complexity:
// Add Song: O(n) (traverse to end)
// Delete Song: O(n) (search for song)
// Display: O(n)
// Space Complexity: O(n) (one node per song)
```

programiz.com/cpp-programming/online-compiler/

Programiz C++ Online Compiler

main.cpp

```cpp
// Node structure for the linked list
struct Node {
    string song;
    Node* next;

    Node(string name) {
        song = name;
        next = nullptr;
    }
};

// Playlist class with basic operations
class Playlist {
private:
    Node* head;
public:
    Playlist() {
        head = nullptr;
    }

    // Add a song at the end
    void addSong(string name) {
        Node* newNode = new Node(name);
        if (head == nullptr) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
        cout << "Song added: " << name << endl;
    }
```

Output

```
Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: ShapeOfYou
Song added: ShapeOfYou

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: Perfect
Song added: Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 3
Playlist: ShapeOfYou -> Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice:
```

Programiz C++ Online Compiler

main.cpp

```cpp
    // Delete a song by name
    void deleteSong(string name) {
        if (head == nullptr) {
            cout << "Playlist is empty!" << endl;
            return;
        }

        // If the song to delete is the first node
        if (head->song == name) {
            Node* temp = head;
            head = head->next;
            delete temp;
            cout << "Song deleted: " << name << endl;
            return;
        }

        // Search for the song in the list
        Node* temp = head;
        Node* prev = nullptr;
        while (temp != nullptr && temp->song != name) {
            prev = temp;
            temp = temp->next;
        }

        if (temp == nullptr) {
            cout << "Song not found!" << endl;
        } else {
            prev->next = temp->next;
            delete temp;
            cout << "Song deleted: " << name << endl;
        }
    }

    // Display all songs
```

Output

```
Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: ShapeOfYou
Song added: ShapeOfYou

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: Perfect
Song added: Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 3
Playlist: ShapeOfYou -> Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice:
```

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

Output | Clear

```cpp
    void display() {
        if (head == nullptr) {
            cout << "Playlist is empty!" << endl;
            return;
        }

        Node* temp = head;
        cout << "Playlist: ";
        while (temp != nullptr) {
            cout << temp->song;
            if (temp->next != nullptr) cout << " -> ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Playlist playlist;
    int choice;
    string name;

    do {
        cout << "\nMenu:\n1. Add song\n2. Delete song\n3. Display playlist\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(); // Ignore newline character after number input

        switch (choice) {
            case 1:
                cout << "Enter song name: ";
                getline(cin, name);
                playlist.addSong(name);
                break;
            case 2:
```

```
Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: ShapeOfYou
Song added: ShapeOfYou

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: Perfect
Song added: Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 3
Playlist: ShapeOfYou -> Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice:
```

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

Output | Clear

```cpp
                playlist.addSong(name);
                break;
            case 2:
                cout << "Enter song name to delete: ";
                getline(cin, name);
                playlist.deleteSong(name);
                break;
            case 3:
                playlist.display();
                break;
            case 4:
                cout << "Exiting Playlist Manager." << endl;
                break;
            default:
                cout << "Invalid choice! Try again." << endl;
        }
    } while (choice != 4);

    return 0;
}

/*
Approach:
1. Implemented a singly linked list manually (Node structure).
2. addSong(): Adds a song at the end by traversing the list.
3. deleteSong(): Deletes the node matching the given name.
4. display(): Prints all songs in the playlist.
*/

// Time Complexity:
// Add Song: O(n) (traverse to end)
// Delete Song: O(n) (search for song)
// Display: O(n)
// Space Complexity: O(n) (one node per song)
```

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: ShapeOfYou
Song added: ShapeOfYou

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: Perfect
Song added: Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 3
Playlist: ShapeOfYou -> Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice:

Output

```
Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: ShapeOfYou
Song added: ShapeOfYou

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 1
Enter song name: Perfect
Song added: Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice: 3
Playlist: ShapeOfYou -> Perfect

Menu:
1. Add song
2. Delete song
3. Display playlist
4. Exit
Enter your choice:
```

## Question 10: Train Coach Arrangement

***Scenario:***

You're designing a **train management system** where each coach is connected to **both the previous and the next coach**, so the data structure should be a **doubly linked list.**

You need to allow the following operations:

✓ Add coach at the **front**

✓ Add coach at the **end**

✓ Remove a coach from the **middle** (given the coach number)

***Problem Statement:***

Create a **doubly linked list** that supports these operations:

- Add coach at front
- Add coach at end

- Remove a coach from middle (by coach number)

***Constraints:***

- Number of coaches ≤ 10^5
- Coach numbers are unique and positive integers

***Input Format:***

Menu:

1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit

***Output Format:***

Display coaches in order from front to end.

***Example:***

**Input/Output Flow:**

1. Add coach at front
Enter coach number: 101
2. Add coach at end
Enter coach number: 102
2. Add coach at end
Enter coach number: 103
4. Display train
Train coaches: 101 <-> 102 <-> 103
3. Remove coach
Enter coach number to remove: 102
4. Display train
Train coaches: 101 <-> 103
5. Exit

***C++ Solution:***

```
// Question 10: Train Coach Arrangement
// Language: C++

#include <iostream>
using namespace std;

// Node structure for doubly linked list
struct Node {
    int coachNumber;
    Node* prev;
    Node* next;

    Node(int num) {
        coachNumber = num;
        prev = nullptr;
        next = nullptr;
    }
};

class Train {
private:
    Node* head;
    Node* tail;
public:
    Train() {
        head = nullptr;
        tail = nullptr;
    }

    // Add coach at front
    void addFront(int num) {
        Node* newNode = new Node(num);
        if (head == nullptr) { // Empty list
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        cout << "Coach " << num << " added at front.\n";
    }

    // Add coach at end
```

```
void addEnd(int num) {
    Node* newNode = new Node(num);
    if (tail == nullptr) { // Empty list
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    cout << "Coach " << num << " added at end.\n";
}

// Remove a coach by number
void removeCoach(int num) {
    if (head == nullptr) {
        cout << "Train is empty!\n";
        return;
    }
    Node* temp = head;
    while (temp != nullptr && temp->coachNumber != num) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Coach not found!\n";
        return;
    }

    // If node to delete is head
    if (temp == head) {
        head = head->next;
        if (head != nullptr) head->prev = nullptr;
        else tail = nullptr; // List becomes empty
    }
    // If node to delete is tail
    else if (temp == tail) {
        tail = tail->prev;
        tail->next = nullptr;
    }
    // Node is in the middle
    else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
```

```
        }

        delete temp;
        cout << "Coach " << num << " removed.\n";
    }

    // Display all coaches
    void display() {
        if (head == nullptr) {
            cout << "Train is empty!\n";
            return;
        }
        Node* temp = head;
        cout << "Train coaches: ";
        while (temp != nullptr) {
            cout << temp->coachNumber;
            if (temp->next != nullptr) cout << " <-> ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Train train;
    int choice, num;

    do {
        cout << "\nMenu:\n1. Add coach at front\n2. Add coach at
end\n3. Remove coach by number\n4. Display train\n5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter coach number: ";
                cin >> num;
                train.addFront(num);
                break;
            case 2:
                cout << "Enter coach number: ";
                cin >> num;
                train.addEnd(num);
```

```
                break;
            case 3:
                cout << "Enter coach number to remove: ";
                cin >> num;
                train.removeCoach(num);
                break;
            case 4:
                train.display();
                break;
            case 5:
                cout << "Exiting Train Manager.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 5);

    return 0;
}
```

---

**Approach:**

1. Used a doubly linked list for bidirectional traversal.
2. addFront(): Inserts node at beginning.
3. addEnd(): Inserts node at the end.
4. removeCoach(): Searches for node and adjusts links to delete.
5. display(): Traverses list from head to tail.

Time Complexity:
Add at front/end: O(1)
Remove coach: O(n) (search)
Display: O(n)
Space Complexity: O(n)

programiz.com/cpp-programming/online-compiler/

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

```cpp
struct Node {
    int coachNumber;
    Node* prev;
    Node* next;
Node(int num) {
        coachNumber = num;
        prev = nullptr;
        next = nullptr;}};
class Train {
private:
    Node* head;
    Node* tail;
public:
    Train() {
        head = nullptr;
        tail = nullptr;}
    void addFront(int num) {
        Node* newNode = new Node(num);
        if (head == nullptr) { // Empty list
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;}
        cout << "Coach " << num << " added at front.\n";    }
    void addEnd(int num) {
        Node* newNode = new Node(num);
        if (tail == nullptr) { // Empty list
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;}
        cout << "Coach " << num << " added at end.\n";}    // Remove a coach by number
```

Output | Clear

```
Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 101
Coach 101 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 102
Coach 102 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 103
Coach 103 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
```

main.cpp

Share | Run

```cpp
    void removeCoach(int num) {
        if (head == nullptr) {
            cout << "Train is empty!\n";
            return;}
        Node* temp = head;
        while (temp != nullptr && temp->coachNumber != num) {
            temp = temp->next;
        }

        if (temp == nullptr) {
            cout << "Coach not found!\n";
            return;
        }

        // If node to delete is head
        if (temp == head) {
            head = head->next;
            if (head != nullptr) head->prev = nullptr;
            else tail = nullptr; // List becomes empty
        }
        // If node to delete is tail
        else if (temp == tail) {
            tail = tail->prev;
            tail->next = nullptr;
        }
        // Node is in the middle
        else {
            temp->prev->next = temp->next;
            temp->next->prev = temp->prev;
        }

        delete temp;
        cout << "Coach " << num << " removed.\n";
    }
```

Output | Clear

```
Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 101
Coach 101 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 102
Coach 102 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 103
Coach 103 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
```

main.cpp

Share | Run

```
    // Display all coaches
    void display() {
        if (head == nullptr) {
            cout << "Train is empty!\n";
            return;
        }

        Node* temp = head;
        cout << "Train coaches: ";
        while (temp != nullptr) {
            cout << temp->coachNumber;
            if (temp->next != nullptr) cout << " <-> ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    Train train;
    int choice, num;

    do {
        cout << "\nMenu:\n1. Add coach at front\n2. Add coach at end\n3. Remove coach by number\n4.
            Display train\n5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter coach number: ";
                cin >> num;
                train.addFront(num);
                break;
            case 2:
```

Output | Clear

```
Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 103
Coach 103 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 3
Enter coach number to remove: 102
Coach 102 removed.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 4
Train coaches: 103 <-> 101

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
```

main.cpp

Share | Run

```
            case 2:
                cout << "Enter coach number: ";
                cin >> num;
                train.addEnd(num);
                break;
            case 3:
                cout << "Enter coach number to remove: ";
                cin >> num;
                train.removeCoach(num);
                break;
            case 4:
                train.display();
                break;
            case 5:
                cout << "Exiting Train Manager.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 5);

    return 0;}
/*
Approach:
1. Used a doubly linked list for bidirectional traversal.
2. addFront(): Inserts node at beginning.
3. addEnd(): Inserts node at the end.
4. removeCoach(): Searches for node and adjusts links to delete.
5. display(): Traverses list from head to tail.

Time Complexity:
Add at front/end: O(1)
Remove coach: O(n) (search)
Display: O(n)
Space Complexity: O(n)
```

Output

```
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 103
Coach 103 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 3
Enter coach number to remove: 102
Coach 102 removed.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 4
Train coaches: 103 <-> 101

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 5
=== Session Ended. Please Run the code again ===
```

Output

```
Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 101
Coach 101 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 102
Coach 102 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 1
Enter coach number: 103
Coach 103 added at front.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 3
Enter coach number to remove: 102
Coach 102 removed.

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 4
Train coaches: 103 <-> 101

Menu:
1. Add coach at front
2. Add coach at end
3. Remove coach by number
4. Display train
5. Exit
Enter your choice: 5
=== Session Ended. Please Run the code again ===
```

## Question 11: Reverse a Linked List

***Scenario:***
You have a **music playlist** stored as a **singly linked list**, where each node represents a song and points to the next song in order. You need to **reverse the playlist in-place**, so the last song becomes the first and so on, **without using extra space**.

***Problem Statement:***
Reverse a singly linked list **in-place** (no extra array or list allowed).

***Constraints:***
- 1 ≤ nodes ≤ 10^5
- Only **O(1) extra space** is allowed.
- **O(n) time complexity**

***Example:***

**Input:**

Playlist: Song1 -> Song2 -> Song3 -> Song4

**Output:**

Reversed Playlist: Song4 -> Song3 -> Song2 -> Song1

***C++ Solution:***

```
// Question 11: Reverse a Linked List
// Language: C++

#include <iostream>
using namespace std;

// Node structure for singly linked list
struct Node {
    string song;
    Node* next;
```

```
    Node(string name) {
        song = name;
        next = nullptr;
    }
};
class Playlist {
private:
    Node* head;

public:
    Playlist() {
        head = nullptr;
    }

    // Add song at the end of playlist
    void addSong(string name) {
        Node* newNode = new Node(name);
        if (!head) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
        cout << "Added: " << name << "\n";
    }

    // Reverse the playlist in-place
    void reversePlaylist() {
        Node* prev = nullptr;
        Node* curr = head;
        Node* nextNode = nullptr;

        while (curr) {
            nextNode = curr->next; // Save next node
            curr->next = prev;     // Reverse pointer
            prev = curr;           // Move prev forward
            curr = nextNode;       // Move curr forward
        }

        head = prev;
```

```
        cout << "Playlist reversed successfully!\n";
    }

    // Display the playlist
    void display() {
        if (!head) {
            cout << "Playlist is empty!\n";
            return;
        }
        Node* temp = head;
        cout << "Playlist: ";
        while (temp) {
            cout << temp->song;
            if (temp->next) cout << " -> ";
            temp = temp->next;
        }
        cout << "\n";
    }
};

int main() {
    Playlist playlist;
    int choice;
    string songName;

    do {
        cout << "\nMenu:\n1. Add Song\n2. Display Playlist\n3. Reverse
Playlist\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter song name: ";
                cin >> songName;
                playlist.addSong(songName);
                break;
            case 2:
                playlist.display();
                break;
            case 3:
                playlist.reversePlaylist();
                break;
```

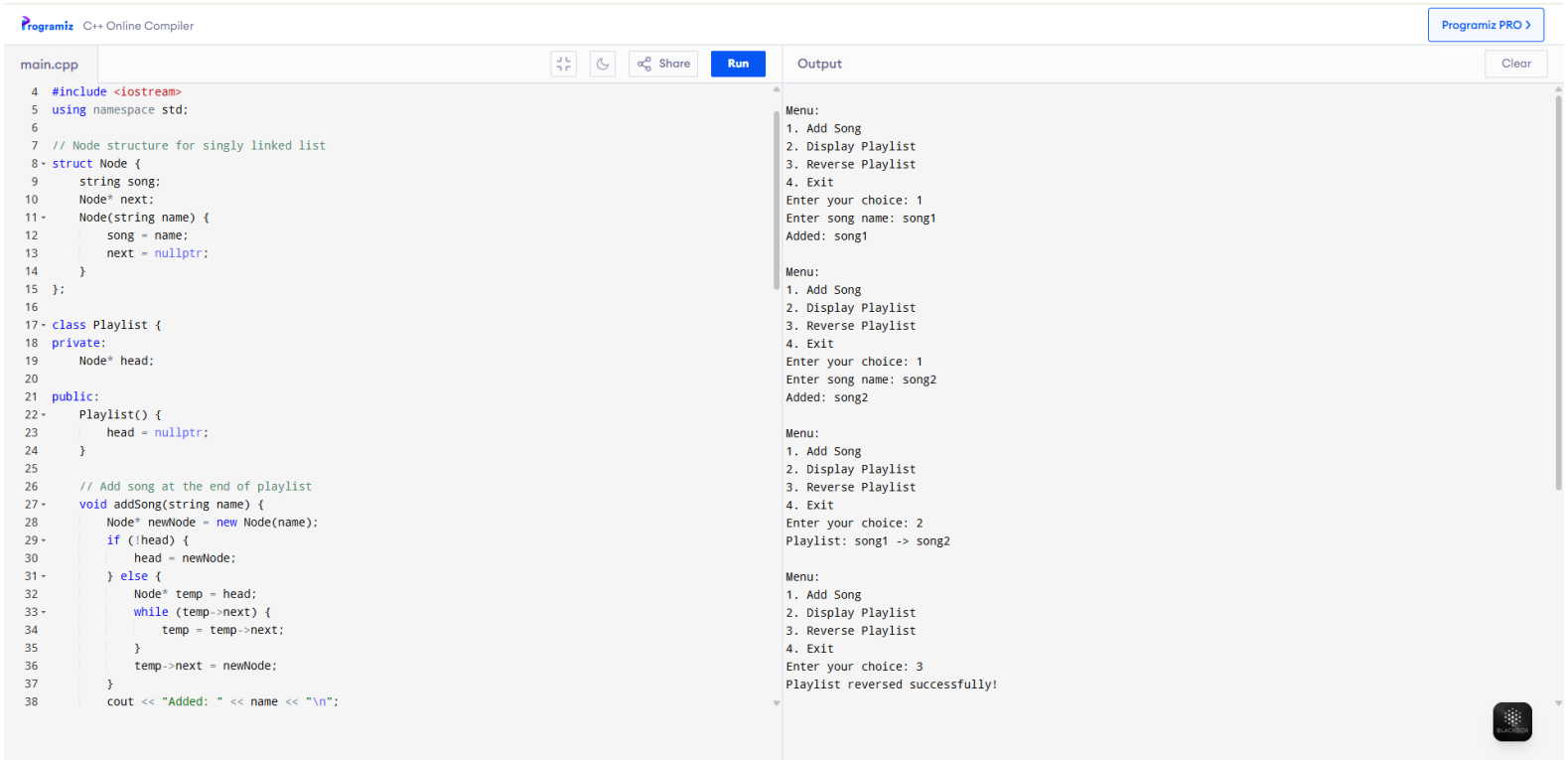
```
            case 4:
                cout << "Exiting Playlist Manager.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 4);

    return 0;
}
```

---

## Approach:

- Used three pointers: prev, curr, nextNode.
- Iterated through the linked list and reversed the links one by one.
- Finally, updated the head pointer to the last node.

Time Complexity: O(n) (traverse entire list once)
Space Complexity: O(1) (no extra space used)



```
        cout << "Added: " << name << "\n";
    }

    // Reverse the playlist in-place
    void reversePlaylist() {
        Node* prev = nullptr;
        Node* curr = head;
        Node* nextNode = nullptr;

        while (curr) {
            nextNode = curr->next; // Save next node
            curr->next = prev;     // Reverse pointer
            prev = curr;           // Move prev forward
            curr = nextNode;       // Move curr forward
        }

        head = prev;
        cout << "Playlist reversed successfully!\n";
    }

    // Display the playlist
    void display() {
        if (!head) {
            cout << "Playlist is empty!\n";
            return;
        }
        Node* temp = head;
        cout << "Playlist: ";
        while (temp) {
            cout << temp->song;
            if (temp->next) cout << " -> ";
            temp = temp->next;
        }
        cout << "\n";
    }
```

```
Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 1
Enter song name: song1
Added: song1

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 1
Enter song name: song2
Added: song2

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 2
Playlist: song1 -> song2

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 3
Playlist reversed successfully!
```

main.cpp

Share | Run | Output | Clear

```
};

int main() {
    Playlist playlist;
    int choice;
    string songName;

    do {
        cout << "\nMenu:\n1. Add Song\n2. Display Playlist\n3. Reverse Playlist\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter song name: ";
                cin >> songName;
                playlist.addSong(songName);
                break;
            case 2:
                playlist.display();
                break;
            case 3:
                playlist.reversePlaylist();
                break;
            case 4:
                cout << "Exiting Playlist Manager.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 4);

    return 0;
}
```

```
Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 2
Playlist: song1 -> song2

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 3
Playlist reversed successfully!

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 2
Playlist: song2 -> song1

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 4
Exiting Playlist Manager.


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

Output

```
Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 1
Enter song name: song1
Added: song1

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 1
Enter song name: song2
Added: song2

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 2
Playlist: song1 -> song2

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 3
Playlist reversed successfully!

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 2
Playlist: song2 -> song1

Menu:
1. Add Song
2. Display Playlist
3. Reverse Playlist
4. Exit
Enter your choice: 4
Exiting Playlist Manager.


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

## Question 12: Undo Feature in Text Editor

***Scenario:***
You're building a **text editor** where every time a user **types a word**, it's stored in a **stack**. When the user presses **Undo**, the last typed word is removed.

***Problem Statement:***
Implement a **stack** using an **array** where:

- **push** = type a word.
- **pop** = undo last word.

***Constraints:***

- 1 ≤ number of operations ≤ 10^5
- Word length ≤ 50 characters
- **No built-in stack allowed** – must use array implementation.

***Example:***

**Input:**

```
Operations:
1. Type "Hello"
2. Type "World"
3. Undo
4. Display
```

**Output:**

```
Current Text: Hello
```

***C++ Solution:***

```
// Question 12: Undo Feature in Text Editor
// Language: C++

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

```
class TextEditor {
private:
    string stack[100000]; // Array-based stack
    int top;

public:
    TextEditor() {
        top = -1; // Initialize empty stack
    }

    // Push operation (Type a word)
    void typeWord(string word) {
        if (top == 99999) {
            cout << "Stack Overflow! Cannot type more words.\n";
            return;
        }
        stack[++top] = word;
        cout << "Typed: " << word << "\n";
    }

    // Pop operation (Undo last word)
    void undo() {
        if (top == -1) {
            cout << "Nothing to undo!\n";
            return;
        }
        cout << "Undo: " << stack[top--] << "\n";
    }

    // Display current text
    void display() {
        if (top == -1) {
            cout << "Text is empty!\n";
            return;
        }
        cout << "Current Text: ";
        for (int i = 0; i <= top; i++) {
            cout << stack[i] << " ";
        }
        cout << "\n";
    }
};
```

```
int main() {
    TextEditor editor;
    int choice;
    string word;

    do {
        cout << "\nMenu:\n1. Type Word\n2. Undo\n3. Display Text\n4.
Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter word: ";
                cin >> word;
                editor.typeWord(word);
                break;
            case 2:
                editor.undo();
                break;
            case 3:
                editor.display();
                break;
            case 4:
                cout << "Exiting Text Editor.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
```

```
        }
    } while (choice != 4);

    return 0;
}
```

---

**Approach:**

- Implemented stack using an array and top pointer.
- push() adds a word to stack, pop() removes the last word.
- display() prints all typed words in order.

Time Complexity:

- push: O(1)
- pop: O(1)
- display: O(n)

Space Complexity: O(n) for storing words (array-based stack).

```
    // Push operation (Type a word)
    void typeWord(string word) {
        if (top == 99999) {
            cout << "Stack Overflow! Cannot type more words.\n";
            return;
        }
        stack[++top] = word;
        cout << "Typed: " << word << "\n";
    }
    // Pop operation (Undo last word)
    void undo() {
        if (top == -1) {
            cout << "Nothing to undo!\n";
            return;
        }
        cout << "Undo: " << stack[top--] << "\n";
    }
    // Display current text
    void display() {
        if (top == -1) {
            cout << "Text is empty!\n";
            return;
        }
        cout << "Current Text: ";
        for (int i = 0; i <= top; i++) {
            cout << stack[i] << " ";
        }
        cout << "\n";
    }
;
nt main() {
    TextEditor editor;
    int choice;
    string word;
    do {
        cout << "\nMenu:\n1. Type Word\n2. Undo\n3. Display Text\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter word: ";
                cin >> word;
                editor.typeWord(word);
                break;
            case 2:
                editor.undo();
                break;
            case 3:
                editor.display();
                break;
            case 4:
                cout << "Exiting Text Editor.\n";
                break;
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 4);
    return 0;
```

Output

```
Menu:
1. Type Word
2. Undo
3. Display Text
4. Exit
Enter your choice: 1
Enter word: Amit
Typed: Amit

Menu:
1. Type Word
2. Undo
3. Display Text
4. Exit
Enter your choice: 1
Enter word: Gupta
Typed: Gupta

Menu:
1. Type Word
2. Undo
3. Display Text
4. Exit
Enter your choice: 3
Current Text: Amit Gupta

Menu:
1. Type Word
2. Undo
3. Display Text
4. Exit
Enter your choice: 2
Undo: Gupta

Menu:
1. Type Word
2. Undo
3. Display Text
4. Exit
Enter your choice:
=== Session Ended. Please Run the code again ===
```

## Question 13: Circular Queue for Parking

***Scenario:***
You manage a parking lot with a fixed number of spots. Cars enter and leave in **FIFO order**, but since space is limited, the parking lot is modeled as a **circular queue**.

***Problem Statement:***
You need to support the following operations:

- **enqueue(carNumber)** → A car enters the parking lot.
- **dequeue()** → The first car leaves the parking lot.
- **display()** → Show all cars currently parked in order.

***Constraints:***
- Parking spots ≤ 1000
- **No built-in queue allowed** – implement your own **circular queue using an array**.

***Example:***

**Input:**

```
Operations:
1. enqueue(101)
2. enqueue(102)
3. enqueue(103)
4. dequeue()
5. display()
```

**Output:**

```
Car left: 101
Cars in parking: 102 103
```

***C++ Solution:***

```
// Question 13: Circular Queue for Parking
// Language: C++
```

```
#include <iostream>
using namespace std;
class CircularQueue {
private:
    int front, rear, size;
    int queue[1000]; // Max parking spots = 1000

public:
    CircularQueue(int n) {
        size = n;
        front = rear = -1;
    }

    // Enqueue - Car enters
    void enqueue(int carNumber) {
        if ((rear + 1) % size == front) {
            cout << "Parking Full! Cannot add car " << carNumber <<
"\n";
            return;
        }
        if (front == -1) front = 0; // First element
        rear = (rear + 1) % size;
        queue[rear] = carNumber;
        cout << "Car " << carNumber << " parked.\n";
    }

    // Dequeue - Car leaves
    void dequeue() {
        if (front == -1) {
            cout << "Parking Empty! No car to leave.\n";
            return;
        }
        cout << "Car left: " << queue[front] << "\n";
        if (front == rear) {
            front = rear = -1; // Queue empty
        } else {
            front = (front + 1) % size;
        }
    }

    // Display cars in parking
    void display() {
```

```
        if (front == -1) {
            cout << "Parking is empty!\n";
            return;
        }
        cout << "Cars in parking: ";
        int i = front;
        while (true) {
            cout << queue[i] << " ";
            if (i == rear) break;
            i = (i + 1) % size;
        }
        cout << "\n";
    }
};

int main() {
    int n;
    cout << "Enter number of parking spots: ";
    cin >> n;
    CircularQueue parking(n);

    int choice, carNumber;
    do {
        cout << "\nMenu:\n1. Car Enters\n2. Car Leaves\n3. Display
Parking\n4. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter car number: ";
                cin >> carNumber;
                parking.enqueue(carNumber);
                break;
            case 2:
                parking.dequeue();
                break;
            case 3:
                parking.display();
                break;
            case 4:
                cout << "Exiting system.\n";
                break;
```

```
            default:
                cout << "Invalid choice! Try again.\n";
        }
    } while (choice != 4);

    return 0;
}
```

---

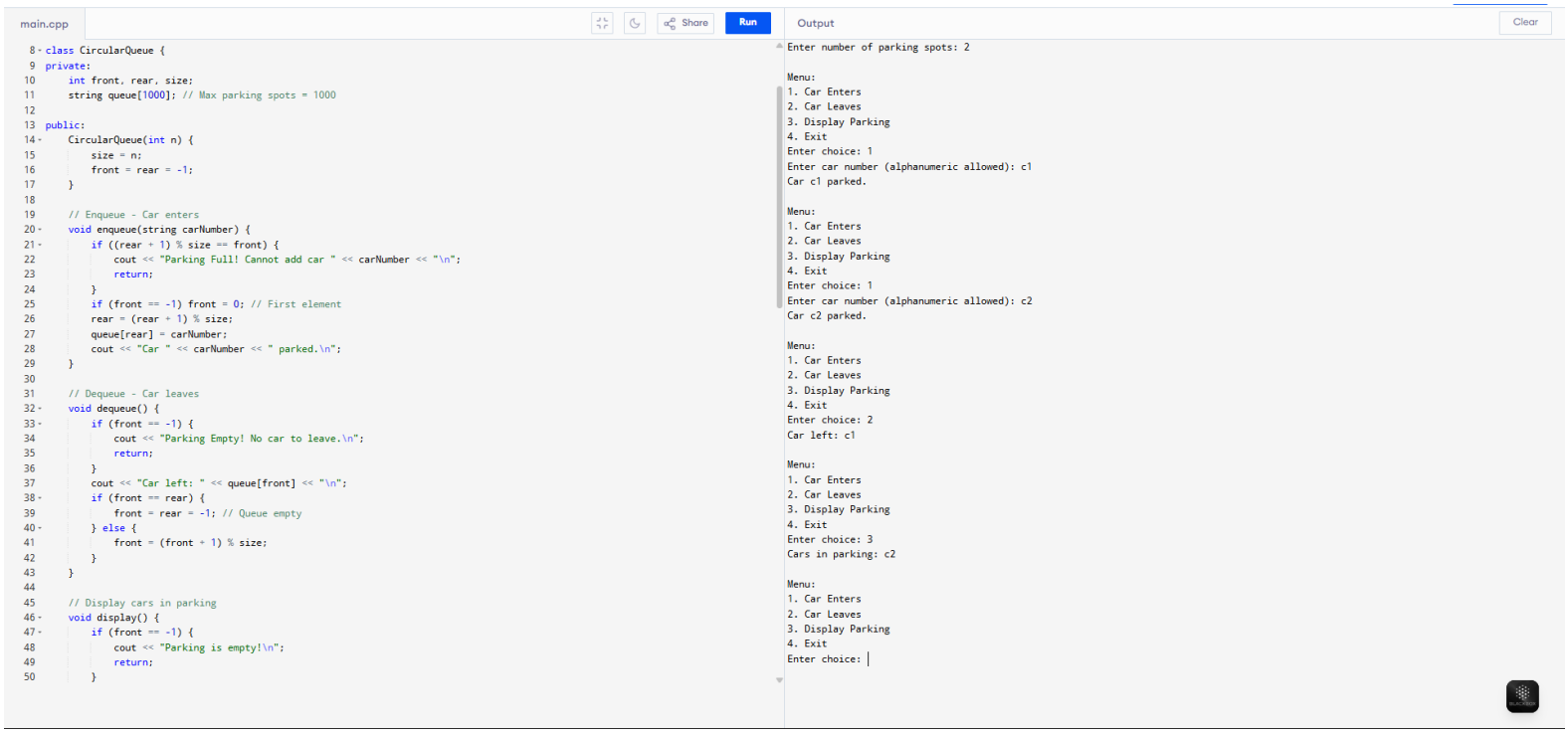
**Approach:**

- Implemented circular queue using array.
- Used modulo operator for circular behavior.
- Enqueue adds car, Dequeue removes the first car, Display shows current queue.

Time Complexity:

- Enqueue: O(1)
- Dequeue: O(1)
- Display: O(n)

Space Complexity: O(n) for queue array.



Output

```
Enter number of parking spots: 2

Menu:
1. Car Enters
2. Car Leaves
3. Display Parking
4. Exit
Enter choice: 1
Enter car number (alphanumeric allowed): c1
Car c1 parked.

Menu:
1. Car Enters
2. Car Leaves
3. Display Parking
4. Exit
Enter choice: 1
Enter car number (alphanumeric allowed): c2
Car c2 parked.

Menu:
1. Car Enters
2. Car Leaves
3. Display Parking
4. Exit
Enter choice: 2
Car left: c1

Menu:
1. Car Enters
2. Car Leaves
3. Display Parking
4. Exit
Enter choice: 3
Cars in parking: c2

Menu:
1. Car Enters
2. Car Leaves
3. Display Parking
4. Exit
Enter choice: 4
Exiting system.


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

## Question 14: Employee Hierarchy (Binary Tree)

***Scenario:***
A company wants to store **employee names** in a **binary tree** for hierarchy tracking.

***Problem Statement:***

- Create a **binary tree** where each node stores an employee name.
- Print the names using **Inorder Traversal (Left → Root → Right)**.

***Constraints:***

- Number of nodes (employees) ≤ 10^5
- Employee names ≤ 50 characters
- **No built-in tree libraries allowed** – implement manually.

***Example:***

**Input:**

Number of employees = 5
Names: CEO, Manager1, Manager2, Lead1, Lead2

**Output:**

Inorder Traversal: Lead1 Manager1 Lead2 CEO Manager2

***C++ Solution:***

```
// Question 14: Employee Hierarchy (Binary Tree)
// Language: C++

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

// Node structure for Binary Tree
struct Node {
    string name;
```

```
    Node* left;
    Node* right;

    Node(string empName) {
        name = empName;
        left = right = nullptr;
    }
};

// Insert node into binary tree based on lexicographical order
Node* insert(Node* root, string empName) {
    if (root == nullptr) {
        return new Node(empName);
    }

    if (empName < root->name) {
        root->left = insert(root->left, empName);
    } else {
        root->right = insert(root->right, empName);
    }
    return root;
}

// Inorder Traversal (Left -> Root -> Right) with comma separation
void inorder(Node* root, bool &first) {
    if (root != nullptr) {
        inorder(root->left, first);
        if (!first) cout << ", ";
        cout << root->name;
        first = false;
        inorder(root->right, first);
    }
}

int main() {
    int n;
    cout << "Enter number of employees: ";
    cin >> n;

    Node* root = nullptr;
    string empName;

    cout << "Enter employee names:\n";
    for (int i = 0; i < n; i++) {
        cin >> empName;
        root = insert(root, empName);
    }

    cout << "Inorder Traversal of Employee Hierarchy:\n";
    bool first = true;
```

```
    inorder(root, first);
    cout << "\n";

    return 0;
}
```

---

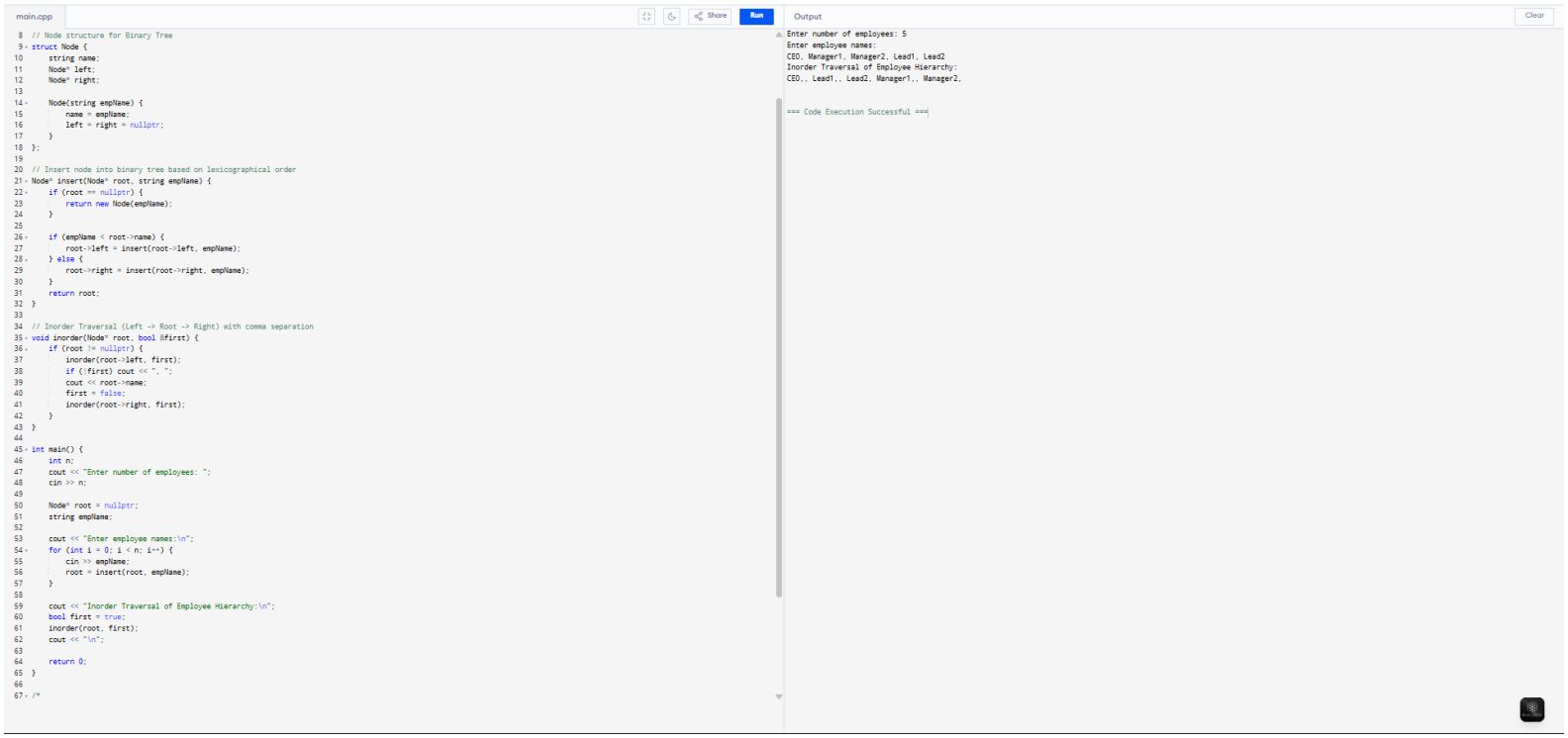
**Approach:**

- Implemented a Binary Search Tree (BST) for storing employee names.
- Inserted each employee name based on lexicographical order.
- Used inorder traversal to print names in sorted order with commas.

Time Complexity:

- Insertion: O(log n) average per node, total O(n log n) average
- Traversal: O(n)

Space Complexity: O(n) for tree nodes.



```
Enter number of employees: 5
Enter employee names:
CEO, Manager1, Manager2, Lead1, Lead2
Inorder Traversal of Employee Hierarchy:
CEO,, Lead1,, Lead2, Manager1,, Manager2,


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

## Question 15: Priority-Based Job Scheduling (Max Heap)

### *Scenario:*

You have a set of jobs, each with a **priority value**. Jobs must be executed from **highest priority to lowest priority**.

### *Problem Statement:*

Implement a **Max Heap manually** (without using built-in priority queue) to manage job scheduling. Jobs should be extracted in order of priority.

### *Constraints:*

- `Number of jobs ≤ 10^5`
- `Job priority values are positive integers`
- **No built-in priority queue or heap functions allowed** – implement heap manually using arrays.

### *Example:*

**Input:**

```
Number of jobs: 5
Priorities: 10 30 20 50 40
```

**Output:**

```
Jobs executed in order of priority: 50 40 30 20 10
```

### *C++ Solution:*

```
// Question 15: Priority-Based Job Scheduling (Max Heap)
// Language: C++

#include <iostream>
using namespace std;

// Max Heap implementation using array
class MaxHeap {
    int* arr;
    int size;
    int capacity;

public:
    MaxHeap(int cap) {
        capacity = cap;
        size = 0;
        arr = new int[cap];
    }

    // Insert a new job priority
    void insert(int val) {
        if (size == capacity) {
            cout << "Heap is full!\n";
            return;
        }
        size++;
        int i = size - 1;
        arr[i] = val;

        // Fix heap property (heapify up)
        while (i != 0 && arr[(i - 1) / 2] < arr[i]) {
            swap(arr[i], arr[(i - 1) / 2]);
            i = (i - 1) / 2;
        }
    }

    // Extract maximum element (highest priority)
    int extractMax() {
        if (size <= 0)
            return -1;
        if (size == 1) {
```

```
            size--;
            return arr[0];
        }

        int root = arr[0];
        arr[0] = arr[size - 1];
        size--;

        heapify(0); // fix heap property
        return root;
    }

    // Heapify down (maintain max heap)
    void heapify(int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < size && arr[left] > arr[largest])
            largest = left;
        if (right < size && arr[right] > arr[largest])
            largest = right;

        if (largest != i) {
            swap(arr[i], arr[largest]);
            heapify(largest);
        }
    }

    bool isEmpty() {
        return size == 0;
    }
};

int main() {
    int n;
    cout << "Enter number of jobs: ";
    cin >> n;

    MaxHeap heap(n);
    cout << "Enter job priorities:\n";
    for (int i = 0; i < n; i++) {
        int priority;
```

```
        cin >> priority;
        heap.insert(priority);
    }

    cout << "Jobs executed in order of priority:\n";
    while (!heap.isEmpty()) {
        cout << heap.extractMax() << " ";
    }
    cout << "\n";

    return 0;
}
```

---

**Approach:**

- Implemented a Max Heap using an array.
- Insert each priority and maintain heap property using heapify up.
- Extract jobs one by one using extractMax() which performs heapify down.

Time Complexity:
- Insertion: O(log n) per element
- Extraction: O(log n) per element
Total: O(n log n)

Space Complexity:
- O(n) for heap storage.

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

Output | Clear

```cpp
// Question 15: Priority-Based Job Scheduling (Max Heap)
// Language: C++

#include <iostream>
using namespace std;

// Max Heap implementation using array
class MaxHeap {
    int* arr;
    int size;
    int capacity;

public:
    MaxHeap(int cap) {
        capacity = cap;
        size = 0;
        arr = new int[cap];
    }

    // Insert a new job priority
    void insert(int val) {
        if (size == capacity) {
            cout << "Heap is full!\n";
            return;
        }
        size++;
        int i = size - 1;
        arr[i] = val;

        // Fix heap property (heapify up)
        while (i != 0 && arr[(i - 1) / 2] < arr[i]) {
            swap(arr[i], arr[(i - 1) / 2]);
            i = (i - 1) / 2;
        }
    }
```

```
Enter number of jobs: 5
Enter job priorities:
10 30 20 50 40
Jobs executed in order of priority:
50 40 30 20 10


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

Programiz C++ Online Compiler

Programiz PRO >

main.cpp

Share | Run

Output | Clear

```cpp
    int extractMax() {
        if (size <= 0)
            return -1;
        if (size == 1) {
            size--;
            return arr[0];}
        int root = arr[0];
        arr[0] = arr[size - 1];
        size--;
        heapify(0); // fix heap property
        return root;}
    // Heapify down (maintain max heap)
    void heapify(int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < size && arr[left] > arr[largest])
            largest = left;
        if (right < size && arr[right] > arr[largest])
            largest = right;

        if (largest != i) {
            swap(arr[i], arr[largest]);
            heapify(largest);}}
    bool isEmpty() {
        return size == 0;}};
int main() {
    int n;
    cout << "Enter number of jobs: ";
    cin >> n;
    MaxHeap heap(n);
    cout << "Enter job priorities:\n";
    for (int i = 0; i < n; i++) {
        int priority;
        cin >> priority;
        heap.insert(priority);
    }
```

Enter number of jobs: 5
Enter job priorities:
10 30 20 50 40
Jobs executed in order of priority:
50 40 30 20 10


=== Code Execution Successful ===

Output

```
Enter number of jobs: 5
Enter job priorities:
10 30 20 50 40
Jobs executed in order of priority:
50 40 30 20 10


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