Experiment-2

Student Name Mayank Kumar
UI D 23BCS80013
Branch: BE-CSE
Section/Group 22BCS_IOT-627/A
Semester: 6th
Date of Performance 24/01/25
Subject Name: AP LAB-II
Subject Code: 22CSP-351

1. **Aim:** Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to target. Each input has exactly one solution, and you cannot use the same element twice.

2. **Algorithm:**

- Initialize an empty hash map (dict).
- Iterate through the nums array:
- For each element num, calculate the complement: complement = target - num.
- Check if the complement exists in the hash map:
  - If it does, return the indices of the complement and the current number.
  - If it doesn't, add the current number and its index to the hash map.
- Return the indices of the two numbers that add up to the target.

3. **Implementation/Code:**

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;
vector<int> twoSum(vector<int>& nums, int target)
 {
    unordered_map<int, int> seen;
    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if (seen.find(complement) != seen.end()) {
            return {seen[complement], i};
        }
        seen[nums[i]] = i;
    }
  return {};
}
int main() {
    vector<int> nums = {2, 7, 11, 15};
    int target = 9;
    vector<int> result = twoSum(nums, target);
```

```
    cout << "Indexes: [" << result[0] << ", " << result[1] << "]" << endl;
    return 0;
}
```

4. **Output:**

```
Indexes: [0, 1]


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

5. **Learning Outcome:**

- Learn how to use hash maps to efficiently solve problems that require quick lookups for complements or matches in arrays.
- Learn to optimize brute force approaches ($O(n^2)$) into linear time solutions ($O(n)$) by leveraging hash maps for complement checks.
- Develop the ability to break down a problem into logical steps, such as finding a complement and storing indices for reference.
- Gain insight into how data structures like arrays and hash maps can work together to solve real-world problems effectively

6. **Time Complexity:** O (n), where n is the number of elements in the nums array. This is because we only iterate through the array once.

7. **Space Complexity**: O (n), since we store each number and its index in the hash map.

Experiment-2.2

Student Name Mayank Kumar
UI D 23BCS80013
Branch: BE-CSE
Section/Group: 22BCS_IOT-627/A
Semester: 6th
Date of Performance: 24/01/25
Subject Name: AP LAB-II
Subject Code: 22CSP-351

1. **Aim:** Implement a FIFO queue using two stacks. The queue should support all the functions of a normal queue: push, peek, pop, and empty.

2. **Algorithm:**
   - Use two stacks: stack1 for pushing elements, stack2 for popping elements.
   - For push(x), simply push x onto stack1.
   - For pop(), if stack2 is empty, transfer all elements from stack1 to stack2. Then, pop the top element from stack2.
   - For peek(), if stack2 is empty, transfer all elements from stack1 to stack2. Then, return the top element of stack2.
   - For empty(), check if both stack1 and stack2 are empty.

3. **Implementation/Code:**

```
#include <iostream>
#include <stack>
using namespace std;
class MyQueue {
private:
    stack<int> inputStack, outputStack;
    void transfer() {
        while (!inputStack.empty()) {
            outputStack.push(inputStack.top());
            inputStack.pop();
      }    }
public:
    void push(int x) {
        inputStack.push(x);
    }
    int pop() {
        if (outputStack.empty()) {
            transfer();
        }
        int top = outputStack.top();
        outputStack.pop();
        return top;
```

```
    }
int peek() {
        if (outputStack.empty()) {
            transfer();
        }  return outputStack.top();
    }
    bool empty() {
        return inputStack.empty() && outputStack.empty();
    }
};
int main() {

    MyQueue q;
    q.push(1);
    q.push(2);
    cout << "Peek: " << q.peek() << endl;  // Outputs 1
    cout << "Pop: " << q.pop() << endl;    // Outputs 1
    cout << "Empty: " << q.empty() << endl;  // Outputs 0 (false)
    return 0;
}
```

4. Output:

```
Peek: 1
Pop: 1
Empty: 0
```

5. Learning Outcome:

- Learn the difference between a **queue** (FIFO) and a **stack** (LIFO).
- Understand how to manipulate stack operations (push, pop, top, empty) to simulate queue behavior.
- Develop the ability to break down a problem into logical steps, such as finding a complement and storing indices for reference.
- Gain insight into how data structures like arrays and hash maps can work together to solve real-world problems effectively

6. Time Complexity:

- push(x): O (1)
- pop (): O (n) in the worst case when elements are transferred from stack1 to stack2.
- Peek (): O (n) in the worst case.
- Empty (): O (1)