Find All Unique Elements

> *CS3334 - Data Structures*

> *Kaur Kirandeep -* [redacted]

Table of Contents…

Project Background…

As the name shows the concept is **identifying and extracting** unique elements from a list of numbers. For instance, let's consider my student id as the list {5, 7, 1, 5, 3, 6, 8, 0}. The outcome to obtain a new list containing only the unique elements: {7, 1, 3, 4, 8, 0}. The straightforward way to achieve this is to loop through the list and compare each number on by one. If no match is not found during the iteration, it is considered unique and can be added to a separate list. Let's take a look at the code for this approach:

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

const int MAX_SIZE = 10000; // Maximum size 10^4

void findUnique(int numbers[], int size, int unique[], int& uniqueSize) {
    for (int i = 0; i < size; ++i) {
        int count = 0;
        for (int j = 0; j < size; ++j) {
            if (i != j && numbers[i] == numbers[j]) {
                count++;
                break;
            }
        }
        if (count == 0) {
            unique[uniqueSize] = numbers[i];
            ++uniqueSize;
        }
```

```
    }
}

int main() {
    int numbers[MAX_SIZE] = {0}; // Array to store numbers
    int size = 0; // size of the array

    string line;
    while (getline(cin, line)) {
        istringstream iss(line);
        int num;

        while (iss >> num && size < MAX_SIZE) {
            numbers[size] = num;
            ++size;
        }

        int unique[MAX_SIZE] = {0}; // Array to store unique elements
        int uniqueSize = 0; // size of the unique array

        findUnique(numbers, size, unique, uniqueSize);

        for (int i = 0; i < uniqueSize; ++i) {
          if(i==uniqueSize-1){
            cout << unique[i] ;
          }
          else{
            cout << unique[i] << " ";
          }
        }
        cout << endl;

        size = 0;
    }

    return 0;
}
```

Time complexity

The code has 2 for loops, both of which iterates `size` times. Making the total number of iterations to `size^2` . Let's denote `size` as `n` . Thus, the time complexity of the code can be expressed as `O(n^2)` . As the input size (`n`) increases, the execution time will also grow.

Problem

Let's consider the worst-case scenario, where all elements in the list are unique. In such a case, for each element, we would need to perform `n-1` comparisons. For a list of size 100, we would need to perform 99 comparisons for each element. Since this approach is not efficient with a large size of list, below we will go through various approaches using data structures to achieve the same result.

Test Cases …

Basic Cases:

1. **Case 1**: All unique numbers
   - Input: `1 2 3 4 5`
   - Expected output: `1 2 3 4 5`
2. **Case 2**: All numbers are duplicates
   - Input: `1 1 1 1 1`
   - Expected output: `1`
3. **Case 3**: Mixed numbers with duplicates
   - Input: `1 2 3 2 1`
   - Expected output: `3`

Edge Case:

1. **Case 4**: Negative numbers
   - Input: `1 -2 -3 -4 -5`
   - Expected output: `1 -2 -3 -4 -5`
2. **Case 5**: Large numbers
   - Input: `1000 2000 3000 4000`
   - Expected output: `1000 2000 3000 4000`
3. **Case 6**: All zeros
   - Input: `0 0 0 0 0 0`
   - Expected output: `0`

Tricky Cases:

1. **Case 7**: Mixed numbers with negative , positive and zero duplicates
   - Input: `1 2 0 -3 4 0 -5 2 4`
   - Expected output: `1 -3 -5`
2. **Case 8**: Empty input
   - Input:
   - Expected output:
3. **Case 9**: Single number
   - Input: `5`
   - Expected output: `5`

Boundary Cases:

1. **Case 10**: Maximum allowed input size
   - Input: `1 2 3 4 ... 1000`
   - Expected output: `1 2 3 4 ... 1000`
2. **Case 11**: Mix of large numbers and zero
   - Input: `0 2000 -3000 0 5000`
   - Expected output: `2000 -3000 5000`
3. **Case 12**: Maximum and minimum integers
   - Input: `2147483648 2147483647`
   - Expected output: `2147483648 2147483647`

> Note: Please note that the `main()` function is the same for each approach , so we will only focus on examining the implementation of `findUnique()` in that approach.

01. Hashing …

Hashing have many different approaches and in this project i have used **Direct Addressing** and **Separate Chaining.** While the implementation of Hashing is different but the `findUnique()` stays the same for both.

```
void findUnique(int numbers[], int size, int unique[], int& uniqueSize) {
    Hash hashSet; // Hash set to store visited numbers
    Hash duplicateSet; // Hash set to store duplicate numbers

    for (int i = 0; i < size; ++i) {
        int value = numbers[i];

        // Check if the value is already in the hash set
        if (hashSet.contains(value)) {
```

```
            // Add the value to the duplicate set
            duplicateSet.insert(value);
        } else {
            // Add the value to the hash set
            hashSet.insert(value);
        }
    }

    // Populate unique array with values that are not duplicates
    for (int i = 0; i < size; ++i) {
        int value = numbers[i];

        if (!duplicateSet.contains(value)) {
            unique[uniqueSize] = value;
            ++uniqueSize;
        }
    }
}
```

1. **Initialisation**:
   - Two instances of the `Hash` class are created: `hashSet` and `duplicateSet`.
     - `hashSet`: Stores numbers that have been visited only once.
     - `duplicateSet`: Stores numbers that have been seen more than once (duplicates).
2. **Finding Duplicate Numbers**:
   - **Iterate through the Input Array**:
     - For each number in the input array (`numbers[]`), check its presence in `hashSet`.
     - **If the Number is Already in `hashSet`**:
       - Insert the number into `duplicateSet` because it's a duplicate.
     - **If the Number is Not in `hashSet`**:
       - Insert the number into `hashSet` to mark it as seen for the first time.
3. **Populating Unique Array**
   - **Iterate through the Input Array Again**:
     - For each number in the input array (`numbers[]`), check its presence in `duplicateSet`.
     - **If the Number is Not in `duplicateSet`**:
       - Insert the number into the `unique[]` array at the current position indicated by `uniqueSize`.
       - Increment `uniqueSize` to indicate that a unique number has been added to the output array.

Direct Addressing

Key-value pairs are stored in a collection using direct addressing hashing, occasionally referred to as array-based hashing, where the keys are represented by the **array indices**. Using this approach, a key is assigned directly to an array index via the hash function. The `Hash` class is designed to implement a simple hash set using an array. This hash set is used to efficiently store and check the presence of integers.

Member Variables:

- `set[HASH_SIZE]` : A boolean array where each index corresponds to a possible integer value. If `set[i]` is `true`, it means the integer `i` is present in the hash set.
- `values[HASH_SIZE]` : An integer array to store the actual values that are inserted into the hash set. This allows us to store the integer value associated with each index.

Methods:

1. **Constructor (`Hash()`)**:
   - Initializes both the `set` and `values` arrays. Initially, all values are set to `false` in `set`, and all values are set to `0` in `values`.
2. **hash(int key)**:
   - This method takes an integer key and computes its hash value using modulo operation to ensure the hash value remains within the range `[0, HASH_SIZE-1]`.
3. **insert(int key)**:
   - Inserts an integer `key` into the hash set. It first computes the hash value for the key and then sets the corresponding index in `set` to `true` and stores the key in the `values` array at that index.
4. **remove(int key)**:
   - Removes an integer `key` from the hash set. It computes the hash value for the key and sets the corresponding index in `set` to `false`.
5. **contains(int key)**:
   - Checks if an integer `key` is present in the hash set. It computes the hash value for the key and returns `true` if `set[index]` is `true` and `values[index]` is equal to the key.

Separate Chaining

A different type of hashing is called separate chaining, in which a **linked list** or is stored within every array index to **handle collisions**. The newly created key-value pair might be added to the current linked list at the relevant index if the t**wo keys hash to the same index** (collision). The `Node` class represents a node in a linked list used for chaining within the hash table. The `Hash` class implements a hash table using chaining, where each entry in the array (`table`) is a pointer to the head of a linked list of nodes.

Node Class:

- `key` : An integer representing the value stored in the node.
- `next` : A pointer to the next `Node` in the linked list.
- The constructor initializes a `Node` with a given key and sets the `next` pointer to `NULL`.

Hash Class:

- `table` : An array of pointers to `Node` objects, representing the hash table

1. **Constructor (`Hash()`)**:
   - Initializes the hash table by setting all entries to `NULL`.
2. **hash(int key)**:
   - Calculates the hash index for a given key using the modulo operation.
3. **insert(int key):**
   - Inserts a new `key` into the hash table.
   - Calculates the hash index for the key.
   - Creates a new `Node` with the key.
   - Inserts the new node at the beginning of the linked list at the calculated hash index.

4. **remove(int key)**:
    - Removes a `key` from the hash table.
    - Calculates the hash index for the key.
    - Traverses the linked list at the hash index to find the node with the given key.
    - Removes the node from the linked list.
5. **contains(int key):**
    - Checks if a key exists in the hash table.
    - Calculates the hash index for the key.
    - Traverses the linked list at the hash index to find the node with the given key.
    - Returns `true` if the key is found, otherwise `false`.

Time Complexity Analysis

Method 1: Hash Set with Direct Addressing

1. Hash Set Operations:

- **insert**:
    - Average Time Complexity: O(1)
    - Worst-Case Time Complexity: O(1)
- **remove**:
    - Average Time Complexity: O(1)
    - Worst-Case Time Complexity: O(1)
- **contains**:
    - Average Time Complexity: O(1)
    - Worst-Case Time Complexity: O(1)

2. Finding Unique Elements:

- The first loop iterates through the `numbers` array of size `size`. Inside this loop, `hashSet.contains(value)` and `duplicateSet.insert(value)` are called. Both operations have O(1) worst-case time complexity.
- The second loop also iterates through the `numbers` array of size `size`. Inside this loop, `duplicateSet.contains(value)` is called. The `contains` operation has O(1) worst-case time complexity.

Method 2: Hash Set with Chaining (Linked List)

1. Hash Set Operations:

1. Hash Set Operations with Chaining:

- **insert**:
    - Average Time Complexity: O(1)
    - Worst-Case Time Complexity: O(n), where n is the number of elements in the linked list.
- **remove**:
    - Average Time Complexity: O(n)

- Worst-Case Time Complexity: O(n)

- **contains**:
  - Average Time Complexity: O(n)
  - Worst-Case Time Complexity: O(n)

2. **Finding Unique Elements:**

- The first loop iterates through the `numbers` array of size `size`. Inside this loop, `hashSet.contains(value)` and `duplicateSet.insert(value)` are called. Both operations have O(n) worst-case time complexity.
- The second loop also iterates through the `numbers` array of size `size`. Inside this loop, `duplicateSet.contains(value)` is called. The `contains` operation has O(n) worst-case time complexity.

— —

Summary of Hashing

The array implementation of the hash set offers a **constant time complexity O(size)** for its operations in **both average and worst-case scenarios**. On the other hand, the chaining method using linked lists might face a **linear time complexity O(size*n)** in the **worst-case due to potential collisions**. Choosing the appropriate method depends on the specific requirements and characteristics of the data. If the hash function distributes keys evenly and collisions are rare, the array implementation might be more efficient. However, if collisions are common, the chaining method might be more suitable despite its higher worst-case complexity.

---

02. Binary Search Tree …

The BST is made up of nodes, each of which includes the integer value (`data`), a count of instances (count), pointers to left and right child nodes (`left` and `right`), and an index (`index`) to keep track of the insertion order.

1. **Insertion (`insert` Function)**:
   - Inserts a new node with a given value and index into the BST.
   - If the value already exists, it increments the count of occurrences.
   - The insertion operation maintains the binary search tree property, where values in the left subtree are less than the parent node, and values in the right subtree are greater.
2. **Node Retrieval by Index (`getNodeByIndex` Function)**:
   - Retrieves a node from the BST based on its index.
   - Recursively searches for the index in the left and right subtrees until the node with the matching index is found or all nodes are traversed.

```
void findUnique(int numbers[], int size, int unique[], int& uniqueSize) {
    BST bst; // BST initialisation

    // Insert each number into the BST
    for (int i = 0; i < size; ++i) {
        bst.insert(numbers[i]);
    }

    // Print the unique elements in the order of insertion
    uniqueSize = 0;
    for (int i = 1; i <= size; ++i) {
        Node* node = bst.getNodeByIndex(i);
        if (node != NULL && node->count == 1) {
```

```
            unique[uniqueSize] = node->data;
            ++uniqueSize;
        }
    }
}
```

1. **BST Initialisation**:
   - Creates an instance of the BST to facilitate the storage and management of unique integers.
2. **Insertion of Integers into BST**:
   - Iterates through the input array of integers.
   - Inserts each integer into the BST using the `insert` function of the BST class.
   - The insertion process ensures that only unique integers are stored in the BST.
3. **Identification of Unique Integers**:
   - Iterates through the BST nodes in the order of their insertion using the `getNodeByIndex` function.
   - Checks the `count` of each node to identify unique integers (nodes with a count of 1).
   - Retrieves and stores the unique integers in an output array.

Time Complexity Analysis

1. **Insertion of Integers into BST:**
   - **Average Case**:
     - In a balanced BST, each insertion takes O(log n) time.
     - For `n` insertions, the total time complexity becomes O(n * log n).
   - **Worst Case**:
     - If the tree becomes skewed, meaning it resembles a linked list, each insertion could take O(n) time.
     - In the worst-case scenario, the total time complexity for `n` insertions would be O(n^2).
2. **Identification of Unique Integers:**
   - **Traversal**:
     - Traversing `n` nodes in the BST to identify unique integers takes O(n) time.

Combining the complexities from both steps, While the average case of the algorithm is efficient with **O(n * log n)** time complexity, it's crucial to recognise the potential worst-case scenario, resulting in a quadratic time complexity of **O(n^2)**.

---

03. Graph Adjacency Matrix …

An effective way to visualise the relationships between numbers is with an adjacency matrix of a graph. It makes it possible to quickly identify duplicates by **looking at the matrix cells,** making the process of finding unique numbers simple.

```
void findUnique(int numbers[], int size, int adjMatrix[][MAX_SIZE], int unique[], int& un
iqueSize) {
    uniqueSize = 0;
```

```
    // Initialize adjacency matrix to 0
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            adjMatrix[i][j] = 0;
        }
    }

    // Build adjacency matrix
    for (int i = 0; i < size; ++i) {
        for (int j = i + 1; j < size; ++j) {
            if (numbers[i] == numbers[j]) {
                adjMatrix[i][j] = 1;
                adjMatrix[j][i] = 1;
            }
        }
    }

    // Find unique numbers
    for (int i = 0; i < size; ++i) {
        bool isUnique = true;

        for (int j = 0; j < size; ++j) {
            if (adjMatrix[i][j] == 1) {
                isUnique = false;
                break;
            }
        }

        if (isUnique) {
            unique[uniqueSize] = numbers[i];
            ++uniqueSize;
        }
    }
}
```

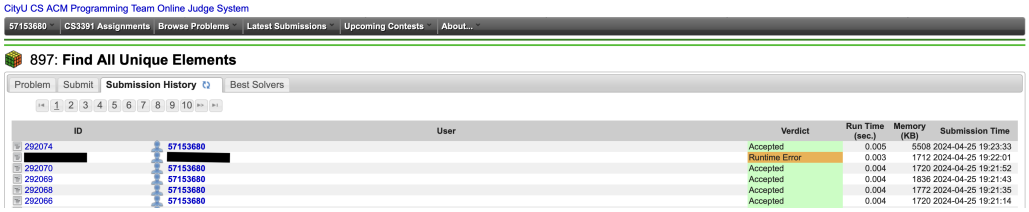
1. **Initialisation**:
   - `uniqueSize` is initialised to 0, which keeps track of the number of unique elements found.
2. **Adjacency Matrix Initialisation**:
   - `adjMatrix` is initialised to zeros, representing no relationships between numbers initially.
3. **Building the Adjacency Matrix**:
   - The matrix is filled based on the presence of duplicate numbers in the sequence. A cell `(i, j)` is set to 1 if `numbers[i]` equals `numbers[j]`.
4. **Finding Unique Numbers**:
   - iterates through the numbers and checks their corresponding rows in the adjacency matrix.
   - Numbers without duplicates (i.e., all cells in their rows are 0) are identified as unique and stored in the `unique` array.

Time Complexity Analysis

1. **Adjacency Matrix Initialisation**:
   - Initialising an $n \times n$ matrix with zeros takes $O(n^2)$ time.
2. **Building the Adjacency Matrix**:
   - Checking for duplicates among n numbers takes $O(n^2)$ time in the worst case.
3. **Finding Unique Numbers**:
   - Iterating over the adjacency matrix and checking each number's row to identify unique numbers takes $O(n^2)$ time in the worst case.

Combining the operations, while the worst-case time complexity is $O(n^2)$, the average-case time complexity also remains $O(n^2)$ since the BST algorithm always constructs and checks the adjacency matrix regardless of input distribution.

Online Judge System (Outputs) …



Approach	Submission ID
Basic Array with Nested Loop	292066
Hashing with Direct Addressing	292068
Hashing with Separate Chaining	292069
Binary Search Tree (BST)	292070
Graph Adjacency Matrix	292074

Conclusion …

The project explores various approaches to identify and extract unique elements from a list of numbers.

1. **Basic Array with Nested Loop**: This approach, though simple, suffers from inefficiency for larger datasets due to its quadratic time complexity of **O(n^2)** caused by **nested loops**.
2. **Hashing with Direct Addressing**: Offers a simple implementation with **O(1)** average and worst-case time complexity for insert, remove, and contains operations. However, it requires a large array, making it **memory-intensive**.
3. **Hashing with Separate Chaining**: Efficiently handles collisions and requires less memory than direct addressing but may result in longer lookup times due to linked list traversal. Its worst-case time complexity is **O(n)** for insert, remove, and contains operations.
4. **Binary Search Tree (BST)**: Maintains sorted elements with an average time complexity of **O(log n)** for insertions in balanced trees. However, unbalanced trees can lead to a worst-case time complexity of **O(n)** and require extra memory for each node.
5. **Graph Adjacency Matrix**: Offers a straightforward method for quick lookup and duplicate identification but is memory-intensive with a quadratic time complexity of **O(n^2)** for building and identifying unique numbers.

Choosing the most effective method depends on several needs, like **memory limits**, **input size**, and **output sorting needs**. It may be better to use BST or direct addressing for hashing smaller datasets in which memory is not an issue. On the other hand, hashing using

separate chaining or the graph adjacency matrix may be more appropriate for larger datasets that may have memory limitations. Every method has compromises between memory utilisation, time complexity, and simplicity of implementation; the best option will depend on the particulars of the data being processed.