unordered_set in C++ STL

An **unordered_set** is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered_set take constant time O(1) on an average which can go up to linear time O(n) in the worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

The **unordered_set** can contain key of any type – predefined or user-defined data structure but when we define key of type user-defined type, we need to specify our comparison function according to which keys will be compared.

Header File: The unordered_set is declared in the header file "unordered_set".

Sets vs Unordered Sets

Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered. Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal).

The time complexity of set operations is O(log N) while for unordered_set, it is O(1).

Common methods on unoredered_set:

- **insert()**– Inserts a new {element} in the unordered_set container.
- **begin()**– Returns an iterator pointing to the first element in the unordered_set container.
- **end()**– Returns an iterator pointing to the past-the-end-element.
- strong>clear()– Removes all of the elements from an unordered_set and empties it.
- **size()**– Return the number of elements in the unordered_set container.

Traversal: The unordered_set works simply like a Hash Table and does not maintain any ordering of the elements. We can traverse an unordered_set just like any other container using iterators, but the elements will be printed in any random order every time. There is no such particular ordering of elements inside the unordered_set container.

Below program illustrate the above methods and traversal of unordered_set:

```cpp
// C++ program to illustrate unordered_set

#include<iostream>
#include<algorithm>
#include<unordered_set>

using namespace std;

int main()
{
    unordered_set<int> s;

    // Inserting elements using
    // insert() function
    s.insert(10);
    s.insert(5);
    s.insert(15);
    s.insert(20);

    // Traversing the unordered_set
    for(auto it = s.begin(); it!=s.end(); it++)
    {
        cout<<(*it)<<" ";
    }
    cout<<endl;

    cout<<"Size of unordered_set: "<<s.size()<<endl;

    // Clear all elements
```

Run

Output:

```
20 15 10 5
Size of unordered_set: 4
Size after clearing unordered_set: 0
```

Some more important functions on unordered_set

- **find()**: The find() function returns an iterator to end() if key is not there in set, otherwise an iterator to the key position is returned. The iterator works as a pointer to the key values so that we can get the key by dereferencing them using * operator.
- **count()**: The count() function works similar to the find() function. It is used to return the count of a key present in the unordered_set, but as unordered_set doesnot allows duplicates, the count function returns 1 if the key is present in the set or 0 if it is not present.
- **erase()**: The erase() function is used to remove either a single element of a range of elements ranging from start(inclusive) to end(exclusive).

Below program illustrate the above functions:

```cpp

// C++ program to illustrate unordered_set

#include<iostream>
#include<algorithm>
#include<unordered_set>

using namespace std;

int main()
{
    unordered_set<int> s;

    // Inserting elements using
    // insert() function
    s.insert(10);
    s.insert(5);
    s.insert(15);
    s.insert(20);

    // Check if 20 is present using
    // find() function
    // If the iterator returned by find() does
    // not points to end() iterator then 20 is
    // present otherwise not
    if(s.find(20)==s.end())
        cout<<"20 not found\n";
    else
        cout<<"20 found\n";

```

Run

Output:

```
20 found
20 found
Size before erasing: 4
Size after erasing: 3
Size: 0
```

Time Complexity and Internal Working

The unordered_set internally uses HashMap for implementation. The time complexity of operations like search(), insert(), erase(), count() works on O(1) time on average depending on the hash function that how uniformly it is distributing the elements in different sections of the HashTable.

The functions like begin(), end(), rbegin(), rend(), size(), empty() works in constant time, i.e. O(1).

Sample Problem: Print Unique Elements of an Array

Problem: Given an unsorted integer array, print all distinct elements in an array. The given array may contain duplicates and the output should print every element only once.

```
Input: arr[] = {7, 2, 9, 4, 3, 2, 10, 4}
Output: {10 3 4 7 9 2}

Input: arr[] = {1, 2, 16, 16, 2, 3, 16}
Output: {3, 1, 16, 2}
```

Explanation: The repetitions have been removed.

Approach:

Step 1: Create a unordered_set or HashSet of integers

Step 2: Traverse the given array from left to right and keep track of visited elements in a hash table.

Step 3: Finally print the elements of hashset or unordered_set

Time Complexity: The time taken for insertion is **O(N)** and accessing an individual element from HashSet is O(1). In the worst case, it can take O(N) for traversing the HashSet if all the elements in the array are unique.

```cpp
// C++ program to print the
// distinct elements of an array
#include <bits/stdc++.h>
using namespace std;

// Function to print unique elements
void printUniqueElememt(int arr[], int n)
{
    // Creating an unordered_set
    unordered_set<int> unique_set;

    // Inserting elements into set
    for (int i = 0; i < n; i++) {
        unique_set.insert(arr[i]);
    }

    // Displaying the set
    for (
        auto it = unique_set.begin();
        it != unique_set.end(); it++) {
        cout << *it << " ";
    }
}

// Driving Method
int main()
{
    int arr[] = {7, 2, 9, 4, 3, 2, 10, 4};
    int n = 7;
```

Run

Output:

```
10 3 4 7 9 2
```

Sample Problem: Print duplicate elements of array

Problem: Given an array of integers, find and print all those elements of an array which have appeared twice in the array.

```
Input: arr[] = {7, 2, 5, 9, 4, 2, 7, 10}
Output: {2, 7}

Input: arr[] = {1, 1, 2, 2, 3, 4, 4}
Output: {1, 2, 4}

Explanation: All the duplicate
elements have been printed
```

Approach:

Step 1: Create an unordered_set of integers

Step 2: Traverse the array and print the element if it is not found in the set otherwise insert in the set. This will take care of the repetitions and will print the element if it again occurs in the set. Only, the first time due to the emptiness of the set, the first element will be inserted by default.

Time complexity: The above approach takes O(N) time in the worst possible case.

```cpp

// C++ program to print the
// duplicate elements of array
#include <bits/stdc++.h>
using namespace std;

// Function to print the duplicate
void printDuplicate(int arr[], int n)
{
    // Creating an unordered_set
    unordered_set<int> st;
    for (int i = 0; i < n; i++) {
        // If repeititions are found
        if (st.find(arr[i]) != st.end())
            cout << arr[i] << " ";

        else
            st.insert(arr[i]);
    }
}

// Driver method
int main()
{
    int arr[] = { 7, 2, 5, 9, 4, 2, 7, 10 };
    int n = 8;
    printDuplicate(arr, n);
    return 0;
}
```

Run

Output:

```
2 7
```

Sample Problem : Check for a Pair Sum

Problem: Check whether any pair's sum in an array matches to the given 'sum' or not using Unordered Set. Return true or 1 if a match is found else return false or 0.

Examples:

```
Input: arr[] = {5, 9, 8, 13, 2, 4}
       n = 6
       sum = 7
Output: True
Explanation: 5 + 2 = 7

Input: arr[] = {5, 9, 8, 13, 2, 4}
       n = 6
       sum = 3
Output: False
Explanation: No pairs sum = 3
```

Approach: The problems could have been easily solved using nested for loops but that would result in increased time complexity. Using an Unordered Set, one can optimise the time complexity for this problem. To do this first create an empty unordered_set. Traverse the array and perform the following operations. Find the difference between the sum and the current_element of the array and check whether the difference is present in the unordered_set or not. Also, add the current_element into the unordered_set for upcoming verifications. This process can be precisely broken down into the following steps:

down into the following steps:

Step 1: Create an unordered_set.

Step 2: Find the complement i.e., sum - arr[current_element].

Step 3: If the complement is present in the unordered_set, return true.

Step 4: Insert the current_element into the unordered_set.

Step 5: Repeat Step 2 to Step 4 until full array is traversed.

Time Complexity: Since we are using only one loop to traverse throughout the array, the time complexity would be O(N). The time complexity to search for an element in an unordered set is O(1) or constant. Therefore the total time complexity would be **O(N)**.

Space Complexity: As we are creating an unordered set for this problem, we would have a space complexity of **O(N)**.

```cpp

// C++ program to check for the pair's sum
// matching to given sum

#include <bits/stdc++.h>
using namespace std;

// Function to check for the pair
bool checkForPair(int arr[], int n, int sum)
{

    // Creating an unordered_set
    unordered_set <int> set;

    // Loop traversing the array
    for(int i = 0; i < n; i++)
    {
        // Finding the complement
        int comp = sum - arr[i];

        // Checking for the complement in
        // the set
        if(set.find(comp)!=set.end())
            return true;

        // Inserting the i-th
        // element into the set
        set.insert(arr[i]);
    }

```

Run

Output:

```
1
```

Sample Problem: Union of two unsorted arrays

Problem: Find the Union of two unsorted arrays. Assume that the arrays are unique and any single array does not contain repeated items.

Example:

```
Input: arr1[] = {2, 7, 13, 10}
       arr2[]: {10, 9, 4, 5, 7, 20}
Output: {2, 7, 13, 10, 9, 4, 5, 20}

Input: arr1[] = {7, 1, 5, 2, 3, 6}
       arr2[] = {3, 8, 6, 20, 7}
Output: {1, 2, 3, 5, 6, 7, 8, 20}
```

Approaches and Time Complexity

Approach 1:

This involves printing all the elements of the first array(arr1[m]) and then print all those elements of the second array(arr2[n]) that is not present in the first array. To search for the elements of arr2[] in arr1[] we will implement the method of linear search. The pseudo-code for the following approach is here.

```

void printUnion(int arr1[], int arr2[], int m, int n)
{
    // Printing arr1[]
    for (i = 0; i < m; i++) {
        cout << arr[i];
    }

    // Printing arr2[]
    for (i = 0; i < n; i++) {
        // Searching for elements
        if (!search(arr2[i], arr1[], m)) {
            cout << arr2[i];
        }
    }
}

// Linear search for arr2[i] in arr1[]
bool search(int x, int arr1[], int m)
{
    for (i = 0; i < m; i++) {
        if (arr[i] == x)
            return true;
        return false;
    }
}

```

Time complexity: The printing of arr1[] will take O(m) times. The printing of arr2[] will take O(n) times. The search function being a linear search would take O(m) times. Therefore the print and search operation of second array will take a total of O(n*m) times. The overall time complexity of the program will be O(m) + O(n*m) which is equivalent to O(m*n)

Approach 2:

This involves sorting of the first array using the method of QuickSort. As the array is already sorted, so to search for an element of arr2[] in arr1[] we can use the method of binary search.

```

void printUnion(int arr1[], int arr2[], int m, int n)
{
    // Printing arr1[]
    for (i = 0; i < m; i++) {
        cout << arr[i];
    }

    // Sort using QuickSort
    sort(arr1, arr1 + m)

        // Printing arr2[]
        for (i = 0; i < n; i++)
    {
        // Searching for elements
        if (!search(arr2[i], arr1[], m)) {
            cout << arr2[i];
        }
    }
}

// Binary search for arr2[i] in arr1[]
search(int x, int arr1[], int m)
{
    for (i = 0; i < m; i++) {
        if (arr[i] == x)
            return true;
        return false;
    }
}
```

Time Complexity: The printing of arr1[] will take O(m) times. The sorting of the arr1[] being a QuickSort will take O(mlogm) times. The printing of arr2[] will take O(n) times. The search function being a binary search would take O(log m) times. Therefore the print and search operation of the second array will take a total of O(nlogm) times. The overall time complexity of the program will be O(mlogm) + O(nlogm) which is equivalent to O((m+n)logm) times. This is better than Approach 1.

Approach 3:

As we have seen in the previous approaches that the trick to solve the problem efficiently lies in finding the best data structure that has a minimum time complexity in search and sort operations. This is going to be the most efficient method to solve the following problem where we will use the concept of unordered_set which takes O(1) time for searching. The step involves creating and printing an unordered_set from arr1[], then searching for arr2[] elements in arr1[] using the find() function and printing only those elements that are not present in arr1. Here is the full working code for this approach.

```
1
2  // C++ code to find the union of
3  // two unsorted arrays
4  #include <bits/stdc++.h>
5  using namespace std;
6
7  // Created an unordered_set
8  unordered_set<int> s;
9
10 // Function to print the union of
11 // two unsorted arrays
12 void printUnion(int arr1[], int arr2[],
13                 int m, int n)
14 {
15     // Inserting arr1 elements
16     // into an unordered_set
17     for (int i = 0; i < m; i++) {
18         cout << arr1[i] << " ";
19         s.insert(arr1[i]);
20     }
21
22     // searching and printing the
23     // elements of arr2[] that is
24     // absent in arr1[]
25     for (int i = 0; i < n; i++) {
26         if (s.find(arr2[i]) == s.end()) {
27             cout << arr2[i] << " ";
28         }
29     }
30 }
```

Run

Output:

```
7 6 13 10 9 4 5 20
```

Time Complexity: The printing of arr1[] will take O(m) times and inserting elements into an unordered_set from arr1[] will take O(1) time. The printing of arr2[] will take O(n) times. The search function will take O(1) times as this is an unordered_set. Therefore the print and search operation of the second array will take a total of O(n) times. The overall time complexity of the program will be O(m) + O(n) which is equivalent to O(m+n) times. This is better than Approach 1 and Approach 2 as this performs the operation in linear time.

Space Complexity: Creation of an unordered set will take O(m) time.

- Sample Problem: Intersection of two unsorted arrays

Problem: Find the Intersection of two unsorted arrays. Assume that the arrays are unique and any single array does not contain repeated items.

Example:

```
Input: arr1[] = {7, 2, 9, 15, 10}
       arr2[]: {5, 10, 7, 3, 2, 20, 9}
Output: {7, 10, 2, 9}

Input: arr1[] = {7, 1, 5, 2, 3, 6}
       arr2[] = {3, 8, 6, 20, 7}
Output: {7, 3, 6}
```

Output: {7, 5, 8}

Approaches and Time Complexity

Approach 1:

This involves searching for an element in arr1[m] in arr2[n] using the method of linear search. The pseudo-code for the following approach is here.

```

void printIntersect(int arr1[], int arr2[], int m, int n)
{
    // Beginning the search operation
    for (i = 0; i < m; i++) {
        if (search(arr1[i], arr2[], n)) {
            cout << arr1[i];
        }
    }
}

// Linear search for arr2[i] in arr1[]
bool search(int x, int arr1[], int m)
{
    for (i = 0; i < m; i++) {
        if (arr[i] == x)
            return true;
        return false;
    }
}

```

Time complexity: The first loop will take O(m) times and the search function being a linear search would take O(n) times. Therefore the overall time complexity of the program will be O(m*n) times

Approach 2:

This involves sorting of the first array using the method of QuickSort. As the array is already sorted, so to search for an element of arr2[] in arr1[] we can use the method of binary search.

```

void printIntersect(int arr1[], int arr2[], int m, int n)
{
    // Sorting the second array
    sort(arr2, arr2+n);

    // Beginning the search operation
    for (i = 0; i < m; i++) {
        if (search(arr1[i], arr2[], n)) {
            cout << arr1[i];
        }
    }
}

// Binary search for arr2[i] in arr1[]
bool search(int x, int arr1[], int m)
{
    for (i = 0; i < m; i++) {
        if (arr[i] == x)
            return true;
        return false;
    }
}

```

Time Complexity: The sorting of the arr2[] being a QuickSort will take O(nlogn) times. The search function being a binary search would take O(log n) times. Therefore the search and print operation of the loop will take a total of O(mlogn) times. The overall time complexity of the program will be O(mlogn) + O(nlogn) which is equivalent to O((m+n)logn) times. This is better than Approach 1.

Approach 3:

As we have seen in the previous approaches that the trick to solve the problem efficiently lies in finding the best data structure that has a minimum time complexity in search and sort operations. This is going to be the most efficient method to solve the following problem where we will use the concept of unordered_set which takes O(1) time for searching. The step involves creating and printing an unordered_set from arr1[], then searching for arr2[] elements in arr1[] using the find() function and printing only those elements that are not present in arr1. Here is the full working code for this approach.

```

// C++ code to find the intersection of
```

```cpp
// two unsorted arrays
#include<bits/stdc++.h>
using namespace std;

// Created an unordered_set
unordered_set<int> s;

// Function to print the intersection
// of two unsorted arrays
void printIntersect(int arr1[], int arr2[],
int m, int n)
{
    // Inserting arr2 elements
    // into an unordered_set
    for(int i = 0; i < n; i++)
    {
        s.insert(arr2[i]);
    }

    // Searching for arr1[] in arr2[]
    // Printing the common elements
    for(int i = 0; i < m; i++)
    {
        if(s.find(arr1[i]) != s.end())
        {
            cout<<arr1[i]<<" ";
        }
    }
```

Run

Time Complexity: The loop used to insert elements into an unordered_set from arr2[] will take O(n) time. The second loop used to find and print the elements of arr1[] will take O(m) times. The overall time complexity of the program will be O(m) + O(n) which is equivalent to O(m+n) times. This is better than Approach 1 and Approach 2 as this performs the operation in linear time.

Space Complexity: Creation of an unordered set will take O(n) time.

- Sample Problem: Distribute Candies

Problem: Given an array of candies, the goal is to distribute the candies between a brother and a sister such that the sister gets the maximum types of candies. Assume that there is an even number of candies and the sister must not get the minimum type of candies in comparison the brother. *Note:* If all the candies are distinct, then distribute an equal number of candies between the brother and the sister, as that would mean equal distribution of type too.

Example:

Input: {1, 1, 2, 2, 3, 3}

Output: 3

Explanation: In the array, there are 6 candies

and there are 3 types i.e., Type 1, Type 2 and Type 3. So the

sister can get maximum 3 types of candies.

brother = {1, 2, 3}

sister = {1, 2, 3}

Input: {1, 1, 1, 5}

Output: 2

Explanation: In the array, there are 4 candies

```
and there are 2 types i.e., Type 1 and Type 5. So the sister

can get maximum 2 types of candies.

brother = {1, 1}

sister = {1, 5}
```

```
Input: {1, 1, 1, 1}

Output: 1

Explanation: In the array, there are 4 candies

and there are 1 type i.e., Type 1. So the sister can get

maximum 1 type of candies.

brother = {1, 1}

sister = {1, 1}
```

```
Input: {1, 2, 7, 4}

Output: 1

Explanation: In the array, there are 4 candies

and there are 4 types i.e., Type 1, Type 2, Type 7 and Type 4.

So the sister can get maximum 2 type of candies.

brother = {1, 2}

sister = {7, 4}
```

Approach: The approach is quite simple. What we can do is check whether the types of candies is greater than or equal to or less than half the number of candies. If the types are greater than or equal to half the number of candies, then we can return the half the size of the array, which will naturally be equal to the maximum type of candies that can be offered to the sister. For example,

```
array =  {1, 1, 2, 2, 3, 3}

n(size of array) = 6

n/2 = 3

type = 3

Check whether type(3) is greater than or

equal to n/2. Since it is valid, the max

type of candies that can be offered to

the sister is 3.
```

```
1
2  // C++ program to illustrate
3  // Candies distribution problem
4  #include <bits/stdc++.h>
5  using namespace std;
6
7  // Function to return max candies
8  int distributeCandies(int arr[], int n)
9  {
10
11     // Creating an unordered_set
```

```
12      unordered_set<int> set;
13
14      // Inserting the elements into
15      // the set
16      for (int i = 0; i < n; i++)
17          set.insert(arr[i]);
18
19      // Size of the set
20      int types = set.size();
21      if (types >= n / 2)
22          return n / 2;
23      else
24          return types;
25  }
26
27  // Driving Method
28  int main()
29  {
30      // Arrays of candies
```

Run

Output:

```
3
```