Sets for incomparable objects

There is library ‘**set**’ in **c++** which implements set and it has inbuilt functions for unions of sets and intersections of sets and many more.

But it is only defined for the objects which are comparable, if we go by its properties it says ***Sets are containers that store unique elements following a specific order.*** This can be used for only those objects in which there exists an order between them, which can be compared.

If we want to use elements which are not comparable, that is as such there is no ordering between them. We in our project implemented those by first simply using a vector in c++, then using a hash table so that the time complexity get reduced by significant amount.

CASE 1:

In this case we used a vector of strings just for the sake of simplicity, we didn’t defined any ordering in them (we can use any object of any class in similar manner).

We defined some sets of operations on them, we defined it for two sets only, but it can be extended to any number of sets.

Functions:

Contains(Set a, Element e):

Return Type:

Boolean

Parameter:

Set a, which was a vector in our case. Element e, which was a string in our case.

Objective:

Our objective for this function was to check if Element e present in the set a or not.

Algorithm:

It was just a brute force algorithm we just linearly search through all the element of a and compared to e if it was found returned true or returned false.

Complexity:

O(n), as we have to search through all the elements of a in case it was not there.

Equal(Set a, Set b):

Return Type:

Boolean

Parameter:

Set a, which was a vector in our case. set b, which was also a vector.

Objective:

Our objective for this function was to check if both sets are equal or not.

Algorithm:

It was also a brute force algorithm, we just checked if the sizes of both of the sets are equal or not, in case it was equal we just linearly verified if elements of set a are in set b or not. If all of them were present then we returned true or false.

Complexity:

O(n*n), as we have to search through all the elements of a and then use **contains** for all elements of a to check if they are in b.

Union(Set a, Set b):

Return Type:

Set(vector<string>)

Parameter:

Set a, which was a vector in our case. Set b, which was also a vector.

Objective:

Our objective for this function was to find the union of both sets.

Algorithm:

We took another set c and firstly we put all the elements of a in c then we used **contains** function to check if an element is already present in c for every element of b, if it is not there we put it in c and finally returned c.

Complexity:

It too was roughly O(n*n), if we assume both sets contains n elements each, then for each elements of set b I have to check if it was already in c or not, which roughly took O(n) and then for n elements it becomes O(n*n).

Intersection(Set a, Set b):

Return Type:

Set(vector<string>)

Parameter:

Set a, which was a vector in our case. Set b, which was also a vector.

Objective:

Our objective for this function was to find the intersection of both sets.

Algorithm:

For every element of a I just checked if it was in b or not, if it was also in b I just put it in c(a set), else not and finally I returned c.

Complexity:

It too was roughly O(n*n), if we assume both sets contains n elements each, then for each elements of set a I have to check if it was in b or not, which roughly took O(n) and then for n elements it becomes O(n*n).

A\B(Set a, Set b):

Return Type:

Set(vector<string>)

Parameter:

Set a, which was a vector in our case. Set b, which was also a vector.

Objective:

Our objective for this function was to find the set which is equal to A\B.

Algorithm:

For every element of ‘a’ I just checked if it was in ‘b’ or not, if it was not in b I put that in c else not.

Complexity:

It has a time complexity of O(n*n), same reason as I had to check for element in a if it is present in b or not.

These were the set of operations defined by us is case 1 of our code.

We have attached the code with the report.

We can also use **issubset()** by just taking intersection of two sets and comparing it with smaller one, if both are equal then it is subset of other set or else not.

Similarly **issuperset()** can be implemented using union and equal function.

CASE 2:

In this case the functions implemented were same as the previous ones, except we used a hash table to reduce the time complexity of searching for an element in a set.

For this purpose I defined another function hash() with following properties:

Hash(String s):

Return Type:

Integer in range(0,999)

Parameter:

A string s.

Objective:

To map the string s to an integer in range (0,999).

Algorithm:

I initially choose the value of sum to be 7 and then added the ASCII value of every character and multiplied it (sum) with a prime number (31) every time for every character and then taken the mod with 1000 and returned the sum.

Complexity:

It has a time complexity of length of the string.

I used this function as, whenever I had to take union of two sets I first put all the elements of a and put that in c and simultaneously I also kept and array of 1000 vectors (I had to use vectors in case of collision) and mapped those element via the hash function in range(0,999) and put that element in the vector corresponding to that index.

Now when I had to search if that element is present in the set already or not, I just checked by using that element and searching in the place where it is supposed to be i.e. in the hash table now our searching reduced to basically **O(1)**,in best case or **O(logn/(log(logn))),** in case of collision. Here I have assumed that my hash function mapping the string in range (0,999) uniformly.

Which is very less as compared to O(n) in our earlier case.

Similarly I used this function in intersection and A\B.

My others functions were modified accordingly as to incorporate the hash functionality and the **contains** part now I used modified **contains,** which basically first calculates the hash function and then search in the vectors corresponding to that index and returns true and false in very less time.

I have attached the code corresponding to same also with the report.

Conclusion: