Programming Assignment 8 (Sets and Maps)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are essentially going to:

- **Learn how to use ordered sets and maps**

  These are essentially balanced binary search trees (such as AVL tree). In C++, these are called `set` and `map`. In Java, these are called `TreeSet` and `TreeMap`. These are ordered because the values are logically stored and can be retrieved in a sorted order.

- **Learn how to use unordered sets and maps**

  These are essentially hash tables. In C++, these are called `unordered_set` and `unordered_map`. In Java, these are called `HashSet` and `HashMap`.

To this end, **your task is to compelte the following class and methods**:

- `alphabet`, and `zeroSumSubArray` methods in `Ordered.h/Ordered.java`
- `findDuplicates`, and `areAnagram` methods in `Unordered.h/Unordered.java`

The project also contains additional files (which you do not need to modify).
Use `TestCorrectness.cpp/TestCorrectness.java` to test your code.
Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

2 What are sets and maps?

A set is essentially a set in the typical Math terminology – contains unique items. Sets are implemented as a balanced search tree (aka ordered sets) or as a hash table (aka unordered sets).

You have already used unordered maps in the Trie assignment. In general, a map entry comprises of a *key* and *value* pair, whereby you can search the map for a particular key, and if it exists, retrieve the corresponding value back. Maps can be implemented as a balanced search tree (aka ordered maps) or as a hash table (aka unordered maps).

Operations on an ordered set/map are typically supported in $O(\log n)$ time, where n is the number of items in the set/map. Operations on an unordered set/map are typically supported in $O(1)$ expected time but in the worst-case it may take $O(n)$ time, where n is the number of items in the set/map.

2.1 C++

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out C++'s documentation.

Ordered and Unordered Set

- To create an integer ORDERED set: `set<int, less<int> > mySet;`

  To create an integer UNORDERED set: `unordered_set<int> mySet;`

  Note that you may need to change the types of values stored according to the application.

- To get the size of the set: `mySet.size();`

- To insert an integer x: `mySet.insert(x);`

  This will add x only if it is not present, else no change.

- To check if an integer x is already in the set: `if(mySet.find(x) != mySet.end())`

  The statement inside the if evaluates to true if and only if x is already in the set.

- To create an iterator on an ORDERED set and print all values:

```
for (ordered_set<int>::iterator it = mySet.begin(); it != mySet.end(); ++it)
    cout << *it << " ";
```

- To create an iterator on an UNORDERED set and print all values:

```
for (unordered_set<int>::iterator it = mySet.begin(); it != mySet.end(); ++it)
    cout << *it << " ";
```

Ordered and Unordered Map

- To create an ORDERED map with integer keys and integer values: `map<int, int> myMap;`

  To create an UNORDERED map with integer keys and integer values: `unordered_map<int, int> myMap;`

  Note that you may need to change the types of keys and values according to the application.

- To get the size of the map: `myMap.size();`

- To insert an integer *key-value* pair: `myMap[key] = value;`

  This will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.

- To check if the map already contains a particular *key*: `if(myMap.find(key) != myMap.end())`

  The statement inside the if evaluates to true if and only if *key* is already in the map.

- To retrieve the value corresponding to a particular *key*: `int value = myMap[key];`

  If *key* is not present, since this method won't typically return a NULL, to avoid unexpected results, you should first check if *key* is present before using this.

2.2 Java

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out Java's documentation.

Ordered and Unordered Set

- To create an integer ORDERED set: `TreeSet<Integer> mySet = new TreeSet<Integer>();`

  To create an integer UNORDERED set: `HashSet<Integer> mySet = new HashSet<Integer>();`

  Note that you may need to change the types of values stored according to the application.

- To get the size of the set: `mySet.size();`

- To insert an integer x: `mySet.add(x);`

  This will add x only if it is not present, else no change.

- To check if an integer x is already in the set: `if(mySet.contains(x))`

  The statement inside the if evaluates to true if and only if x is already in the set.

- To create an iterator on a set and print all values:

```
Iterator<Integer> it = mySet.iterator();
while (it.hasNext())
    System.out.print(it.next() + " ");
```

Ordered and Unordered Map

- To create an ORDERED map with integer keys and integer values: `TreeMap<Integer, Integer> myMap = new TreeMap<Integer, Integer>();`

  To create an UNORDERED map with integer keys and integer values: `HashMap<Integer, Integer> myMap = new HashMap<Integer, Integer>();`

  Note that you may need to change the types of keys and values according to the application.

- To get the size of the map: `myMap.size();`

- To insert an integer *key-value* pair: `myMap.put(key, value);`

  This will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.

- To check if the map already contains a particular *key*: `if(myMap.contains(key))`

  The statement inside the if evaluates to true if and only if *key* is already in the map.

- To retrieve the value corresponding to a particular *key*: `Integer value = myMap.get(key);`

  The method returns *null* if *key* is not present.

3 Ordered Sets and Maps

You will write two methods to understand how ordered sets and maps work. One of the methods simply return the alphabet of a given array of numbers, i.e., this method returns the distinct numbers in the array. The other one checks if an array contains a subarray whose sum is zero. If such a subarray exists, then it returns the start and end indexes of the array, else returns null.

3.1 Alphabet Finder

To find the alphabet, use the following idea and fill up the `alphabet` method:

Alphabet Finder

- Note that the ordered set will essentially solve this straightaway. So, create an ordered set and just take all the numbers from the array and insert them one-by-one into the ordered set.
- Since sets will only keep unique numbers, you are pretty much done – all duplicates have been removed. Just an use iterator on the set to retrieve the numbers one-by-one, and insert them into a dynamic array.
- Once all numbers have been added, return the dynamic array.

3.2 Is there a subarray summing to zero?

Let's understand the idea first. Pick an example array: $[12, -26, 1, 8, 9, -6, 4, -12, -3, 12]$. Note that this array contains a sub-array that sums to zero; specifically, the numbers: $8, 9, -6, 4, -12$ and -3. On the other hand, $[-6, 4, -12, -3, 12]$ does not have a sub-array summing to zero.

One obvious approach is to find sum of all subarrays and see if any one of them is zero, but this will take $O(n^2)$ time. We will see how to solve this in $O(n \log n)$ time.

Let's define $prefixSum(i) = arr[0] + arr[1] + arr[2] + \cdots + arr[i]$, i.e., $prefixSum(i)$ is the sum of all numbers from index 0 to index i. Note that if there is a subarray from x to y such that $A[x] + A[x+1] + \cdots + A[y] = 0$, then

$$\begin{aligned} prefixSum(y) &= A[0] + A[1] + A[2] + \cdots + A[x-1] + A[x] + A[x+1] + \cdots + A[y] \\ &= prefixSum(x-1) + A[x] + A[x+1] + \cdots + A[y] \\ &= prefixSum(x-1) \end{aligned}$$

In the following example, notice that the red-colored prefix sum -13 has occurred twice: once at index 2 and then at index 8. You can check that the subarray shaded in blue from index 3 to index 8 has a sum of zero. This makes sense as a zero-sum subarray won't change the prefix sum.

i	0	1	2	3	4	5	6	7	8	9
arr[i]	12	-26	1	8	9	-6	4	-12	-3	12
prefixSum(i)	12	-14	**-13**	-5	4	-2	2	-10	**-13**	19

On the other hand there is no repeated occurrence of a prefix sum for the second example:

i	0	1	2	3	4
arr[i]	-6	4	-12	3	-12
prefixSum(i)	-6	-2	-14	-11	-23

So, all we need to do is keep the prefix sum and whenever we find a prefix sum that we have seen before, we know there is a sub-array summing to zero – this subarray starts at the index immediately after last occurrence of the prefix sum and ends at the current index. To check for duplicate prefix sums, we use an ordered map.

Use this idea and the following sketchy pseudo-code to fill up the `zeroSumSubArray` method:

Zero-Sum Subarray Finder

- Create an integer-integer ordered map.
- Initialize $prefixSum$ to 0
- Traverse through the array and do the following:
  - Update $prefixSum$ by adding the current number in the array to it
  - If $prefixSum$ is zero, then there is obviously a zero-sum array starting at index 0 and ending at the current index. So, simply return these indexes in an array of length 2 – first 0 and then the current index.
  - Now, check if the ordered map contains the prefix sum as a key. If it does, then retrieve the value from the ordered map (which is essentially the index of the last occurrence of the same prefix sum). Also, return the appropriate indexes in an array of length 2. Read the section before the pseudo-code to figure out what "appropriate indexes" means.
  - A match with prefix sum was not found. So, insert the current index as value with the prefix sum as key into the ordered set.
- You never found a zero-sum sub-array; so, return null.

4 Unordered Sets and Maps

Here again you'll solve two problems but this time using unordered sets and maps. In the first problem, you will find the duplicates in two arrays, i.e., the common elements – arrays are not sorted. In the second problem, you will verify if two strings are anagrams of each other.

4.1 Duplicate Finder

To find the duplicates, use the following idea and fill up the `findDuplicates` method:

Find Duplicates among two arrays

- This is similar to the alphabet finder stuff. The difference is that you will use an unordered set and duplicates are across arrays.
- First create two integer unordered sets – `checker` and `duplicates`.
- Take all the numbers from one of the arrays and insert them one-by-one to `checker`
- Now, loop over the other array. If a number in this array is present in `checker`, then this number is a duplicate; so, insert it into `duplicates`.
- Once you are done, `duplicates`, being a set, will contain a duplicate number once. So, at this point, iterate through `duplicates` and add the numbers to a dynamic array.
- Finally, return the dynamic array.

4.2 Anagram Checker

Two strings are an anagram of each other if one can be changed to the other by shuffling characters. Thus, *integral* is an anagram of *triangle* and *estrange* is an anagram of *sergeant*, but *cattle* is not an anagram of *eclat*. Notice that if two strings are anagrams of each other, then it means that they have the same set of characters and the same frequencies for each character.

So, we first write a method that accepts a string as argument that computes the frequency of each character in the string. This method signature is not given to you – you have to write it. It returns the character to frequency mapping in an unordered map.

Compute Frequencies

- First create a character-integer unordered map.
- Now, loop over the string.
  - If the current character in the string is not in the map as a key, then insert it into the map as key with value one. This implies that this is the first time you are seeing the character in the string, and so its frequency is one.
  - Else, get the stored frequency (i.e., value) of the current character (as key) from the map. Add one to this value and reinsert into the map (as value once again) with key as the current character. This implies that you have seen the current character in the string prior to this occurrence, and so its frequency should be increased by one.
- At this point, all frequencies have been computed and are stored in the map as a character to frequency map. So, return the map.

Now, fill in the code for the `areAnagrams` using the following idea:

Anagram Checker

- If two strings are of different length, they cannot be anagrams of each other. So, return $false$ if they are of unequal lengths.
- Call the frequency method that you have written to obtain the frequencies of each character in each string and store them in two unordered maps.
- Now, loop over first string.
  - Use the two maps to get the frequencies (values) of the current character (as key) in both the strings.
  - If these frequencies are different, then the strings are not anagrams of each other. So, return $false$
- Once the loop has terminated, you have ensured that all characters in the first string also occur in the second string; moreover, they have the same frequencies in both the strings. Also, the strings are of equal length; so, the second string does not contain a character which is not present in the first string. So, the strings must be anagrams of each other – return *true*.