Project 3: MultiSet Design and Implementation

Learning Objectives

- Design and implement a multiset (also known as a bag) abstract data type (ADT)
- Analyze trade-offs between different data structures, and justify choices in a design document
- Produce design document that communicates implementation plans, reasoning, and anticipated challenges

Overview

In this two-part project, you will design and implement a `MultiSet` class. A multiset is like a set; however, in a multiset, elements may appear more than once. For our multiset, the elements will be C++ `std::string`.

You will first write a design document with explanations of how you will implement the `MultiSet`. This will include your choice of the underlying data structure, which will either be your hash table or AVL tree from the previous projects. You also have the option of designing your own custom data structure, with the only restriction being you cannot use any of the C++ containers such as vector, list, map, set, or multiset.

You will then implement standard multiset operations (insert, remove, contains, count, etc.) and support set-like operations such as union, intersection, difference, and symmetric difference. You will also implement an extension feature from a list of options.

Part 1: Design Document

Write a 3–5-page document in which you:

- Explain how you will implement the multiset
- Choose your internal data structure: HashTable, AVL Tree, or third custom structure (not a standard C++ container)
- Justify your choice
- Describe how you plan to implement core multiset operations
- Describe how you will implement the required set operations
- Identify any challenges anticipated and how you plan to address them

Part 2: Implementation

Once you have completed your design document, you will implement your `MultiSet` class in C++. You will be provided with a header file with the `MultiSet` class declaration. You will implement all the methods contained within the file, as well as any you determine necessary. The default constructor, copy constructor, assignment operator, and destructor are all declared, but it is up to you to determine if you must implement them.

Core MultiSet Operations

Each of the core `MultiSet` operations listed below should be implemented using the method signatures given. For each operation, your approach for implementing that operation should be detailed in your design document.

`bool insert(const std::string& key, size_t num = 1)`

Insert the key into the multiset. The optional parameter `num` is default to 1, and insert will add `num` number of keys to the multiset. Return true if element(s) was/were inserted, which will always be the case as we aren't rejecting duplicates.

`bool remove(const std::string& key, size_t num = 1)`

Remove the `key` from the multiset. The optional parameter `num` is default to 1. The method will attempt to remove `num` instances of the given `key`. If there are fewer instances of the given key in the set than `num`, remove will not modify the multiset and return false. Otherwise it will return true to indicate successful removal.

`std::vector<std::string> remove(size_t num = 1)`

An overload of `remove`, this version will remove `num` arbitrary elements from the multiset and return a vector containing those elements. You may choose to remove random elements, or just remove the first num elements encountered, or any other method of your choosing.

`bool contains(const std::string& key) const`

Return true if the `key` is in the multiset, otherwise false.

`size_t count(const std::string& key) const`

Return the number of instances of the given `key` are currently in the multiset.

`std::vector<std::string> keys() const`

Return a vector containing every element in the multiset. For example, if the key "`a`" appears in the multiset 4 times, there will be 4 "`a`" strings in the return vector.

`std::vector<std::string> uniqueKeys() const`

In contrast with `keys()`, `uniqueKeys()` will return a vector with one copy of each key in the multiset. There will be no duplicates in the result.

`bool empty()`

Determine if the multiset is empty, meaning the size is zero, or if there are elements in the multiset.

`size_t size() const`

Return the total number of elements in the multiset, counting all duplicates.

`size_t uniqueSize() const`

Return the number of unique keys in the multiset, with duplicate keys only counted once.

`void clear()`

Remove all elements from the multiset, making the size zero.

`std::ostream& operator<< (std::ostream& os, const MultiSet& ms)`

Output a representation of the multiset. One possible approach could be to print out pairs of `{key: count}` or print out every individual element.

Set Operations

You will also implement four set specific operations: union, intersection, difference, and symmetric difference. Figure 1 Diagram of Set Operations diagrams these operations.

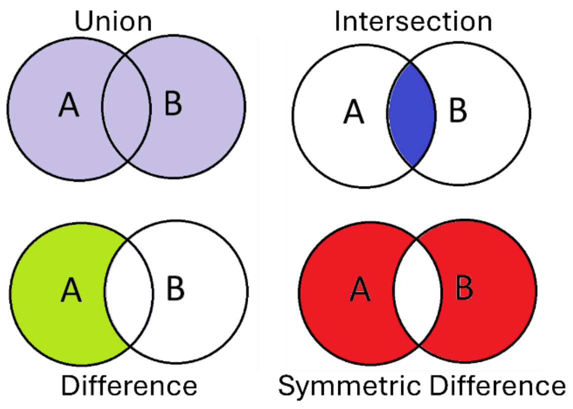

FIGURE 1 DIAGRAM OF SET OPERATIONS

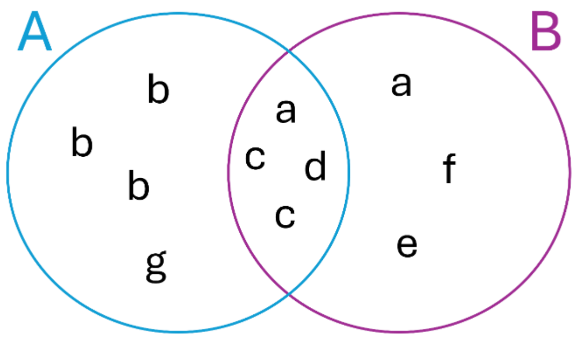

FIGURE 2 VENN DIAGRAM OF MULTISETS A AND B

`MultiSet unionWith(const MultiSet& other) const;`

The union of two sets, denoted $A \cup B$, is the combination of all elements in each set. For example, if the sets A and B, diagramed in Figure 2 Venn Diagram of Multisets A and B, are:

$$A = \{a, b, b, b\ c, c, d, g\}$$
$$B = \{a, a, c, c, d, e, f\}$$

then the union of A and B is

$$A \cup B = \{a, a, b, b, b, c, c, d, e, f, g\}$$

`MultiSet intersectionWith(const MultiSet& other) const;`

The intersection of two sets, denoted $A \cap B$, contains the elements that appear in both sets, so using A and B from above,

$$A \cap B = \{a, c, c, d\}$$

`MultiSet differenceWith(const MultiSet& other) const;`

The difference of two sets, denoted $A - B$ or $A \setminus B$ is the elements that appears in the first set that do not appear in the second set (or the elements in the first set minus the elements in the second set). The difference between A and B is,

$$A - B = \{\ b, b, b, g\}$$

`MultiSet symmetricDifferenceWith(const MultiSet& other) const;`

The symmetric difference, denoted as $A \Delta B$, of two sets contains the elements that appear in only one set, but not both. The symmetric difference of A and B is:

$$A \Delta B = \{\ a, b, b, b, e, f, g\}$$

The symmetric difference is also defined as the union of the difference of each set, or

$$A \Delta B = (A - B) \cup (B - A)$$

Extension Features (Pick at Least One)

You will also implement at least one of the flowing extension features. You may implement a second feature for up to +5 bonus points on the assignment.

Note: These features should be fully integrated into your `MultiSet` class and documented in your design document.

1. **Serialization/Deserialization**
   Add support for saving and loading a multiset to/from a file
   - Format can be plain text, binary, JSON, or another of your choice
   - You should be able to reconstruct an identical multiset after deserialization
2. **Generic Multiset (Templated Class)**
   Make your multiset a C++ template class so it can store any type (not just string)
   - Ensure your underlying data structure also supports generic types
   - For hash table, address how hash functions will be handled
   - For AVL tree, address how elements will be compared
3. **Iterator Support**
   Implement iterators or allow your multiset to be used in range-based for loops
   - You can choose whether iteration yields all elements (including duplicates), distinct ones, or pairs of <key, count>
4. **Equality and Comparison**
   Implement methods to compare to multisets
   - `operator==` returns true if the two multisets have the same elements with the same counts
   - `operator<=>` compares based on lexicographic order of (element, count) pairs
   - `isSubsetOf(other)` and `isSuperSetOf(other)`
5. **Cardinality Analytics**
   Add a method that returns a sorted list of (element, count) pairs
   - Sorted by most frequent -> least frequent
   - Address how keys with the same count will be sorted

Turn in and Grading

You will make two separate submissions: one for the design stage and one for the implementation. Upload the appropriate files to the corresponding Dropbox for Part 1 and Part 2 in Pilot. I recommend submitting your design document along with your implementation to the Dropbox for Part2, as well as the Dropbox designated for Part 1. In total, you will need to submit:

- A PDF, MS Word, or equivalent file with your design explanation as well as which optional features were implemented.
- MultiSet.h/MultiSet.cpp (you may implement the MultiSet class solely in the header file)
- A main.cpp with driver code that demonstrates your multiset with tests

This project is worth 50 points, distributed as follows:

	Task	Points
Part 1	Design document contains plan for implementing MultiSet, choice of data structure and extension feature(s) to use	10
Part 2	Core MultiSet methods implemented and operate correctly	12
	Set operations correctly implemented	12
	Required extension feature implemented and operates correctly	6
	Test cases/driver program demonstrates thorough testing of MultiSet operations	5
	Code is well organized, documented, and formatted according to the course Coding Guidelines.	5
	Bonus extension feature (optional)	+5
	Total	50