Algorithm

midterm – multiset

Jyun-Ao Lin

Anagrams

Two words are said to be anagrams if they are made up of the same set of letters. Thus, `CHEATER` and `TEACHER` are anagrams, because they are made up of the letters $\{\!\{\mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{E}, \mathtt{H}, \mathtt{R}, \mathtt{T}\}\!\}$.

Problem:
Given a file F containing words (for example, your spell checker file), we propose to find all the anagrams.

multiset

multiset

A multiset is a set variant where the same element can appear multiple times.

example: $\{\!\{a, a, a, b, c, c\}\!\}$ is the multiset that contains three occurrences of a, one occurrence of b and two occurrences of c.

If m is a multiset, we denote by $m(x)$ the number of occurrences of x in m, called its multiplicity.

example: If $m = \{\!\{a, a, a, b, c, c\}\!\}$, we have $m(a) = 3$, $m(b) = 1$ and $m(c) = 2$.

An element x belongs to m if and only if $m(x) > 0$.

interface

```
class MSet<E> {
      MSet<E>()                       // new multiset, empty
      int size()                      // size
      int mult(E x) //                // multiplicity
      boolean contains(E x)           // membership
      void add(E x)                   // add an element
      void remove(E x)                // remove an element
      boolean include(MSet<E> that) // is 'this' included in 'that
      boolean equal(Object obj)
}
```

MSet

```
class MSet<E> {
  private HashMap<E, Integer> mult;
  private int size;

  MSet() { this.size = 0; this.mult = new HashMap<>(); }
```

- the `mult` field verifies the invariant

$$\text{for each } (x, n) \in \texttt{mult}, \ \text{we have } n > 0 \tag{1}$$

- the `size` field verifies the invariant

$$\texttt{size} = \sum_{(x,n) \in \texttt{mult}} n \tag{2}$$

encapsulation

The `private` feature enables the principle of encapsulation.
Without `private`, it is impossible to maintain the invariants (1) and (2).

Indeed, code outside the class could modify `size` without affecting `mult`, or even store negative or zero values in the `mult` table.

methods

- return multiplicity

```
int mult(E x) {
    if (this.mult.containsKey(x))
      return this.mult.get(x);
    else return 0;
}
```

- check membership

```
boolean contains(E x) { return mult(x) > 0; }
```

- add an element

```
void add(E x) {
  this.mult.put(x, 1 + mult(x));
  this.size++;
}
```

remove

```
void remove(E x) {
  int n = mult(x);
  if (n == 0) return;
  if (n == 1) this.mult.remove(x);
  else this.mult.put(x, n-1);
  this.size--;
}
```

note: in order to preserve (1), the case when multiplicity 1 use `remove`

include

```
boolean included(MSet<E> that) {
    for (E x: this.mult.keySet())
      if (this.mult(x) > that.mult(x))
        return false;
    return true;
}
```

Hash function

equality

```
public boolean equals(Object obj) {
    MSet<E> that = (MSet<E>)obj;
    return this.size == that.size && this.included(that);
}
```

- if two sets are equal, then they have the same cardinality and the same multiplicity for any element, and so `equals` returns `true`.
- conversely, if `equals` returns `true`, it means that $m_1(x) \leq m_2(x)$ for all x and that

$$\sum_{x \in U} m_1(x) = \sum_{x \in U} m_2(x).$$

  Therefore, $m_1(x) = m_2(x)$ for all x because the multiplicities are positive or zero.

hash code

We would like to be able to use multisets as keys in the hash tables.
To do this, we need to equip our class `MSet` of a suitable `hashCode` method. We propose to add it to the `MSet` class like this, with a traversal of all the keys of the `mult` table:

```
public int hashCode() {
    int h = 0;
    for (E x : this.mult.keySet()) {
        int hx = x.hashCode();
        ...
    }
    return h;
}
```

Options:

1. `h = h + hx;`
2. `h = h + mult(x) * hx * hx;`
3. `h = 31 * h + mult(x) * hx;`

hash code

1. `h = h + hx;`
   Acceptable.
   - But inefficient because it takes values that are too small.
   - Furthermore, it does not use multiplicity, and therefore does not differentiate between $\{\!\{\mathtt{a}\}\!\}$ and $\{\!\{\mathtt{a}, \mathtt{a}\}\!\}$.
   - Finally, the linear nature of this function will give the same value for $\{\!\{\mathtt{a}, \mathtt{d}\}\!\}$ and $\{\!\{\mathtt{b}, \mathtt{c}\}\!\}$, which is regrettable.
2. `h = h + mult(x) * hx * hx;`
   Acceptable. And probably effective because it has none of the three flaws of the previous version.
3. `h = 31 * h + mult(x) * hx;`
   Incorrect, because it depends on the order of traversal of the elements, which is not fixed for a hash table. Thus, for the same multiset $\{\!\{\mathtt{a}, \mathtt{b}\}\!\}$, we can obtain either $31m(a)h(a) + m(b)h(b)$, or $31m(b)h(b) + m(a)h(b)$, which are not equal.

multiset of a word

```
static MSet<Character> msOfWord(String s) {
    MSet<Character> ms = new MSet<>();
    for (int i = 0; i < s.length(); i++)
      ms.add(s.charAt(i));
    return ms;
}
```

- the construction of `ms` is in $O(1)$.
- the `add` method performs an insertion into a hash table, which is an amortized $O(1)$ complexity,
- and an increment of `size` in $O(1)$.

hence a total in $O(\texttt{s.length()})$ amortized.

back to the initial problem

building `word` dictionary

Problem:
Given a file F containing words (for example, your spell checker file), we propose to find all the anagrams.

To do this, we will fill a `words` dictionary of type

```
HashMap< MSet<Character>, Vector<String> >
```

which associates a multiset of letters with the list of words that correspond to it.

example: with English words, we will have something like this:

$$\begin{array}{rcl} \{\!\{\mathtt{A},\mathtt{C},\mathtt{E},\mathtt{E},\mathtt{H},\mathtt{R},\mathtt{T}\}\!\} & \mapsto & [\mathtt{CHEATER},\mathtt{TEACHER},\text{etc}] \\ \{\!\{\mathtt{A},\mathtt{A},\mathtt{N},\mathtt{S},\mathtt{T}\}\!\} & \mapsto & [\mathtt{SANTA},\mathtt{SATAN},\text{etc.}] \\ & \text{etc} & \end{array}$$

building `word` dictionary

We construct the `words` dictionary as follows.

For each word s in the file F, we start by calculating its multiset of characters m with the `msOfWord` method.

- if m is not in `words`, we create a new entry with a list containing the single word s.
- otherwise, we add s to the end of the list associated with m, with the `add` method of the `Vector` class.

building `word` dictionary

For each word w of F,

- we construct its multiset m, in amortized $O(|w|)$ time (previous question);
- we search for m in `words` in constant time;
- and,
    - if it is the first time that we encounter m, we add an entry in `words` in amortized $O(1)$ time;
    - if m is already associated with a list ℓ, we add w to ℓ in amortized $O(1)$ time (`Vector.add`).

Hence a total in $O(|F|)$ amortized.

Extension

Given two words from file F, for example `HASH` and `TABLE`, we can find other pairs of words from F which are anagrams of them: `HEALTH ABS`, `BATHS HEAL`, etc.

Propose an algorithm that takes two words from F as input and prints all word pairs that are anagrams of them.

addAll

adds all elements of the multiset that to the multiset this

```
void addAll(MSet<E> that) {
    for (E x: that.mult.keySet())
      this.mult.put(x, this.mult(x) + that.mult.get(x));
    this.size += that.size;
}
```

Note the use of the this.mult *method* to protect against the case where x is not in this. Be careful not to forget to update size.

diff

Write a method `MSet<E> diff(MSet<E> that)` that returns a new multiset where, for each element, its multiplicity is the difference between its multiplicity in `this` and its multiplicity in `that`.

```java
MSet<E> diff(MSet<E> that) {
    MSet<E> ms = new MSet<>();
    for (E x: this.mult.keySet()) {
        int n = this.mult(x) - that.mult(x);
        if (n > 0) { ms.mult.put(x, n); ms.size += n; }
    }
    return ms;
}
```

back to the problem

1. Let w_1 and w_2 be the two words and u the multiset of all their characters (obtained with `addAll`).
2. For all m in `words`,
    - if $m \subseteq u$, then display `words`$[m]$ and `words`$[u \setminus m]$.

The complexity is $O(N)$ (assuming the operations $\subseteq$ and $\setminus$ in constant time) and therefore quite realistic with N of the order of 10^5.

On the other hand, it would not be reasonable to consider all pairs m_1, m_2 of elements of `words` and then examine whether the union of their multiset is equal u. (This would be in $O(N^2)$ this time.)

Questions?