Задачи за задължителна самоподготовка

по

Структури от данни и програмиране

email: kalin@fmi.uni-sofia.bg

5 декември 2017 г.

1. Да се дефинира метод `HashMap::efficiency()`, който изчислява ефективността на хеш таблицата като отношението $\frac{all-coliding}{all}$, където *coliding* е броят на ключовете, записани при колизия, а *all* е броят на всички записани ключове.

2. Да се дефинира оператор `<<` за клас `HashMap`, който отпечатва в поток всички двойки ключ-стойност в Хеш таблицата.

3. Да се напише програма, която въвежда от клавиатурата две текста с произволна големина t_1 и t_2. Програмата да извежда броя на всички срещания на думи в t_2, които се срещат и в t_1.

   Пример: за следните текстове

   In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

   и

   Ideally, the hash function will assign each key to a unique bucket, but this situation is rarely achievable in practice (usually some keys will hash to the same bucket)

   Този брой е 10, съставен от думите *the* (2 срещания във втория текст), *a* (1 срещане), *hash* (2), *function* (1), *to* (2), *is* (1), *keys* (1).

4. Да се напише програма, която въвежда от клавиатурата две текста с произволна големина t_1 и t_2. Програмата да извежда броя на уникалните думи в t_2, които се срещат и в t_1.

   Пример: за двата текста от предишната задача, този брой е 7, съставен от думите *the*, *a*, *hash*, *function*, *to*, *is*, *keys*.

5. Да се напише програма, която прочита от входа даден текст с произволна големина и намира такава дума с дължина повече от 3 букви, която се среща най-често в текста. Пример: за текста

   In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

   Най-често срещаната дума е *hash*.

6. От клавиатурата да се въведе цялото положително число n, следвано от $2 \times n$ цели положителни числа $a_1, b_1, a_2, b_2, ..., a_n, b_n$. Програмата да печата на екрана ‘‘`Yes`’’, ако изображението, дефинирано като $h(a_i) = b_i, i = 1, ..., n$ е добре дефинирана функция. Т.е. програмата да проверява дали има два различни индекса i и j, за които е изпълнено $a_i = a_j$, но $b_i \neq b_j$.

7. Да се дефинира `operator *` на шаблона на хеш-таблицата. Хеш-таблицата `c`, която се получава при `c = a * b`, да съдържа като множество от ключове сечението на множествата на ключове на `a` и `b`, като стойноста на всеки ключ е двойка (`std::pair`) от съответните стойности от `a` и `b`. Хеш-функцията на `c` да е същата като на `b`.

8. Да се дефинира `operator +` на шаблона на хеш-таблицата. Хеш-таблицата `c`, която се получава при `c = a + b`, да съдържа като ключове симетричната разлика на ключовете на `a` и `b`, със съответните им стойности от `a` и `b`. Хеш-функцията на `c` да е същата като на `b`.

   Симетрична разлика на множествата A и B наричаме множеството $C = A \Delta B = A \cup B - A \cap B$, съдържащо тези елементи на A, които не са елементи на B и тези елементи на B, които не са елементи на A.

9. Да се дефинира метод

   ```
   void map (void (*f) (ValueType&))
   ```

на хеш-таблицата, който прилага функцията `f` над всички стойности в хеш-таблицата.

10. Да се дефинира метод

    `void mapKeys (KeyType (*f) (const KeyType&))`

    на хеш-таблицата, който замества всеки ключ `key` на хеш-таблицата с `f(key)`, като се запазва старата му стойност.

    Упътване: Да се извърши съответното ре-хеширане на елемента и той да се премести на съответния нов индекс в таблицата.

11. Да се дефинира `operator *` на шаблона на хеш-таблицата. Хеш-таблицата `c`, която се получава при `c = a * b`, да съдържа като множество от ключове сечението на множествата на ключове на `a` и `b`, като стойноста на всеки ключ е:

    - Вектор с два елемента – стойността на ключа от `a` и стойността на ключа от `b`, ако тези стойности *не са вектори*.
    - Конкатенацията на стойността на ключа от `a` и стойността на ключа от `b`, ако тези стойности *са вектори*.

    Хеш-функцията на `c` да е като хеш-функцията на `b`.

Пример: Операторът да удовлетворява следния тест:

```
HashMap<string,double> m(5,stringhash1);
m["Kalin"] = 1.85; m["Ivan"] = 1.86;
HashMap<string,double> m1(3,stringhash1);
m1["Kalin"] = 2; m1["Petar"] = 2;
HashMap<string,vector<double>> mult = m * m1;
mult = mult * mult;

assert (mult.containsKey("Kalin"));
assert (!mult.containsKey("Ivan"));
assert (!mult.containsKey("Petar"));
assert (mult["Kalin"].size() == 4);
```

Решение.

Можем да реализираме глобален шаблон на оператор за умножение на хеш-таблици:

```
template <class KeyType, class ValueType>
```

```
HashMap<KeyType,vector<ValueType>>
  operator * (const HashMap<KeyType,ValueType> &hm1,
              const HashMap<KeyType,ValueType> &hm2)
```

Както се вижда, аргументите на оператора са две хеш-таблици с тип на стойностите `ValueType`, а стойността на оператора е от типа `vector<ValueType>`. Един лесен начин да реализираме тялото на оператора е:

```
HashMap<KeyType,vector<ValueType>>
  result(hm2.size(),hm2.getHashFunction());
for (const KeyType &key : hm1)
{
    //key е в сечението на ключовете
    if (hm2.containsKey (key))
    {
        result[key].push_back(hm1[key]);
        result[key].push_back(hm2[key]);
    }
}
return result;
```

Очевидно е, че ако типа `ValueType` се случи да е самият той *вектор*, ефектът от горната реализация ще бъдат стойности, които са *вектори от вектори*, а това не е търсеният резултат. Следователно нашата реализация трябва да "знае" кога `ValueType` е вектор и да вземе съответните мерки да конкатенира векторите, които са стойности на ключа `key` в двете таблици `hm1` и `hm2`.

От друга страна, няма как по време на изпълнението на програмата да определим в тялото на оператора дали `ValueType` е вектор или не е. Дори да използваме някакъв `RTTI` трик и все пак да разберем в *run time*, че `ValueType` е вектор, ще се наложи да приложим още трикове, с които да преобразуваме `ValueType` така, че да можем да използваме методите на `std::vector` за достъп до елементите и да извършим конкатенацията. Би се получило неелегантно и неразбираемо решение.

За щастие, шаблоните в `C++` ни позволяват да направим частен случай на шаблона на оператора `*`, който да се предпочете от компилатора тогава, когато хеш-таблицата е със стойност вектори. Необходимо е единствено да добавим още един шаблон:

```
template <class KeyType, class ValueType>
```

```
HashMap<KeyType,vector<ValueType>>
  operator * (const HashMap<KeyType,vector<ValueType>> &hm1,
              const HashMap<KeyType,vector<ValueType>> &hm2)
```

Забележете, че хеш-таблиците са дефинирани с тип на стойностите `vector<ValueType>`, което е частен случай на простото `ValueType` в предишния оператор `*`, но е достатъчно на компилатора да подбере именно този оператор при опит да умножим две хеш-таблици, чиито стойности са от тип вектор. Съответната реализация на оператора е:

```
HashMap<KeyType,vector<ValueType>>
  result(hm2.size(),hm2.getHashFunction());
for (const KeyType &key : hm1)
{
    //key е в сечението на ключовете
    if (hm2.containsKey (key))
    {
        result[key] = append (hm1[key], hm2[key]);
    }
}
return result;
```

Тук `append` е помощна функция за конкатенация на вектори.