Абстрактен тип **Map**

Хеширане

Какво е абстракцията **Map**?

- В компютърните науки **map** (**associative array**, **dictionary**) , на български **съответствие (асоциативен масив, речник)** е абстрактен тип представящ колекция от двойки **(key, value) (ключ, стойност) .** Всеки ключ се среща най-много веднъж.

- Реализацията на **map** поставя класическа задача в компютърните науки (така наречения **dictionary problem**):
  Да се създаде структура от данни, която поддържа множество от стойности с ефективни операции **търсене, изтриване и вмъкване.**

- Двете класически решения на тази задача са:
  - балансирани бинарни дърветата за търсене
  - хеш-таблиците

- **STL** класа **map** се базира на балансирани бинарни дървета за търсене

Методи и оператори в опростена версия на класа **Map**

Constructors

Map<*key type*, *value type*>**()**	Creates an empty map associating keys and values.

Methods

size()	Returns the number of key/value pairs contained in the map.
isEmpty()	Returns **true** if the map is empty.
put(*key*, *value*)	Associates the specified key and value in the map. If *key* has no previous definition, a new entry is added; if a previous association exists, the old value is discarded and replaced by the new one.
get(*key*)	Returns the value currently associated with *key* in the map. If *key* is not defined, **get** returns the default value for the value type.
remove(*key*)	Removes *key* from the map along with any associated value. If *key* does not exist, this call leaves the map unchanged.
containsKey(*key*)	Checks to see whether *key* is associated with a value. If so, this method returns **true**; if not, it returns **false**.
clear()	Removes all the key/value pairs from the map.

Operators

map[*key*]	The **Map** class overloads the square bracket operator so that a map acts as an *associative array* indexed by the key value. If the key does not exist in the map, the square bracket operator creates a new entry and sets its value to the default for that type.

Map като асоциативен масив

- Можем да разглеждаме обикновения масив като **map** (съответствие) между целите числа **i =0,1,2,3**,…, (индексите на масива) и стойностите на масива **a[i]**. **Map** може да се разглежда като обобщение на понятието масив, при който стойност от някакъв произволен тип се **асоциира** (свързва) с ключ от друг произволен тип.

- Можем да предефинираме **selection оператора []** и да пишем **m[key]=value** вместо **map.put(key,value)** или да връщаме **m[key]** вместо **map.get(key)**

Проста реализация на **StringMap**, при която двойките (ключ, стойност) се записват във вектор

```
public:

    StringMap();
   ~StringMap();
    std::string get(const std::string& key) const;
    void put(const std::string & key, const std::string& value);
    bool containsKey(const std::string& key);
private:
    /*
     * Type: KeyValuePair
     * ------------------
     * This type combine a key and a value into a single structure.
     */

     struct KeyValuePair
     {
         std::string key;
         std::string value;
     };

     /*Instance variables */

     Vector<KeyValuePair> bindings;
     int findKey(const std::string& key) const;
```

- Задача 1: Напишете програма, която прочита трибуквен код на международно летище от потребителя и връща неговото разположение. Данните се вземат от текстовия файл **AirportCodes.txt**, който съдържа няколко хиляди летища по света.

- Задача 2: Напишете програма, която прочита трицифрен телефонен код в **САЩ** и връща в кой щат е кодът и обратно, при даден щат връща кои са телефонните кодове в този щат. Данните се вземат от текстовия файл **AreaCodes.txt**

- Задача 3:

  Пощенските кодове на 50-те щата в САЩ са двубуквени.

  Кодовете и съответните щати **(key, value)** можем да прочетем от файл или да ги въведем чрез отделна функция, тъй като са малко на брой.

  Напишете програма, която използва такова представяне на двойките **(key, value)** в паметта, че за даден ключ стойността се достъпва директно, а не след обхождане.

- **Кое е фундаменталното свойство на масива?**

  Колкото и да е голям масивът, всеки негов елемент може да се достъпи за константно време, защото адресът на всеки елемент може да се изчисли като се знае началния адрес на масива и колко елемента трябва да се прескочат.

- В частния случай на двубуквен код, да се направи операцията за търсене за константно време, не е трудно. Всичко от което се нуждаем е да запомним името на щата за даден код в двумерен масив **string table[26][26]** от символни низове, в който всяка буква от кода се използва за изчисление на индексите в двумерния масив.

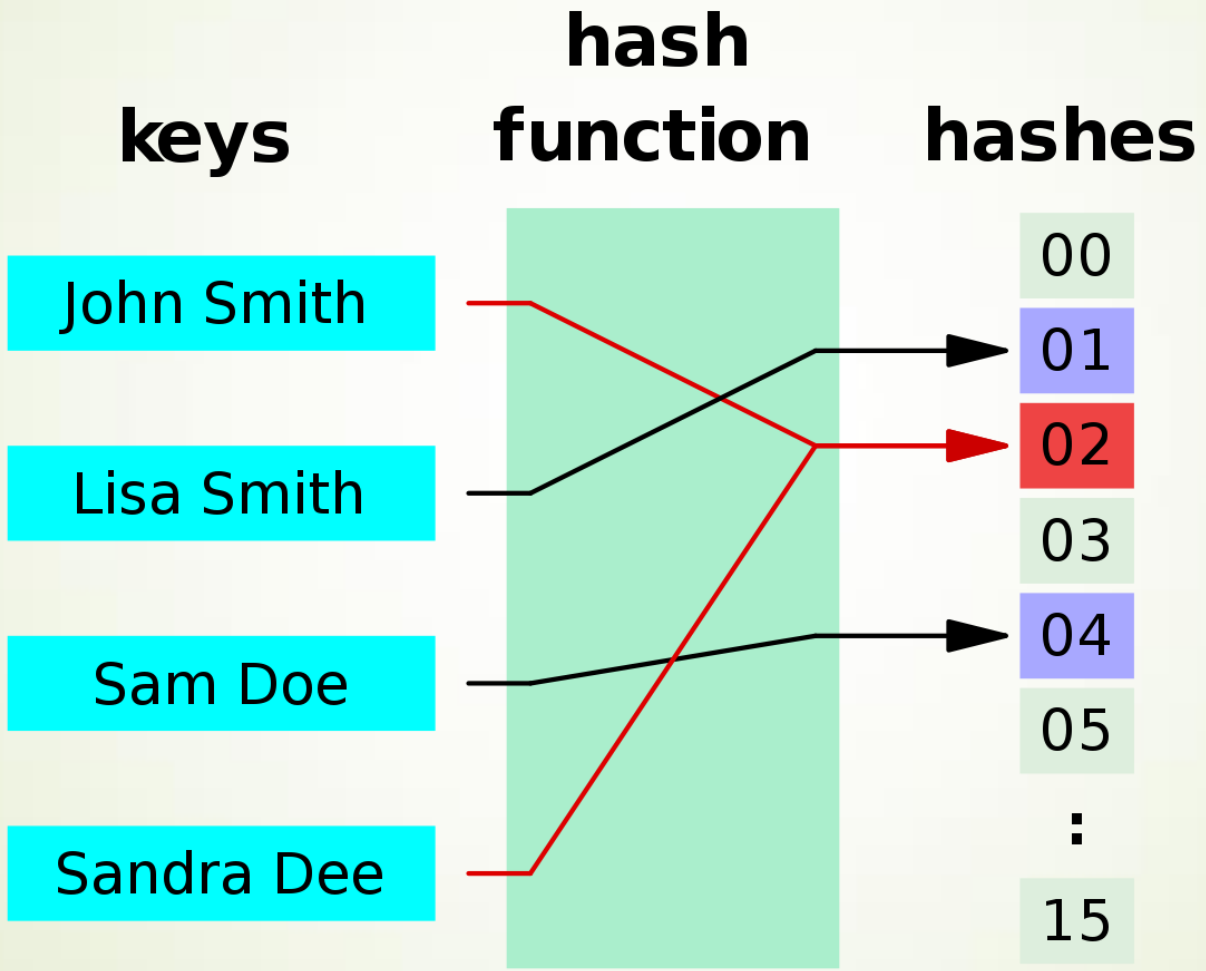
- Това е типичен пример за таблица за търсене (**lookup table**), която е програмна структура която позволява да се извлече стойност от тази таблица с изчислението на индекс от нея.

Хеширане (Hashing)

- Идеята от таблицата за търсене може да се усъвършенства със стратегията за **хеширане**, която работи така:
  1) Избираме функция **f**, която трансформира ключа (key) в цяло число. Това число се нарича **хеш-код** на ключа. Функцията **f** се нарича **хеш-функция**.
  2) Използваме **хеш-кода** като отправна точка на търсене на ключа в **хеш-таблицата**. Хеш-таблицата се представя като масив от свързани списъци.

- Реализацията на абстракцията **map** по този начин се нарича **хеш-таблица.**

Хеш-функция


keys
hash function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

- Типично хеш-кодът се използва да се изчисли индекса в масив от свързани списъци. Всеки списък съдържа всички двойки **(key, value)**, които имат един и същи код. Всеки такъв списък се нарича **bucket (кофа).**

  В повечето приложения броят на **хеш-кодовете** е по-голям от броя на **кофите**. Тогава **кофата** се изчислява така:

  int backet = hashCode(key) % nBuckets;

Изисквания за хеш-фунцията

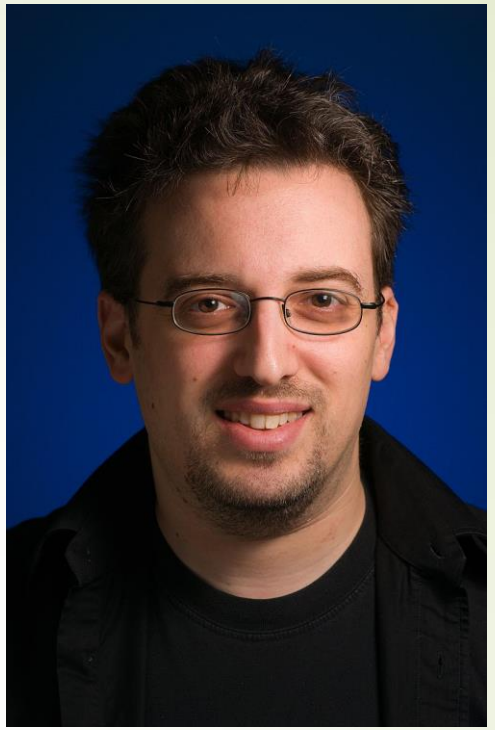
- 1) Функцията трябва да бъде евтина (малко изчислително време)
- 2) Функцията трябва да разпределя ключовете, колкото се може по-равномерно в множеството на целите числа.

Макар кодът за хеш-функциите обикновено да е кратък, той съвсем не е прост и изисква сериозна математическа подготовка.

Отношението $\lambda = \frac{N_{entries}}{N_{buckets}}$ се нарича **load factor**

Може да се обоснове математически, че **load factor <=0.7** ни осигурява сложност на операциите търсене, вмъкване, триене в средния случай ***O(1)***. В най-лошия случай сложността си остава ***O(N)*.**

Имплементация на хеш-функция за символни низове от проф. Deniel J. Bernstein



```
const int HASH_SEED = 5381;                  /* Starting point for first cycle */
const int HASH_MULTIPLIER = 33;              /* Multiplier for each cycle      */
const int HASH_MASK = unsigned(-1) >> 1;     /* All 1 bits except the sign     */

int hashCode(const string & str) {
   unsigned hash = HASH_SEED;
   int n = str.length();
   for (int i = 0; i < n; i++) {
      hash = HASH_MULTIPLIER * hash + str[i];
   }
   return int(hash & HASH_MASK);
}
```

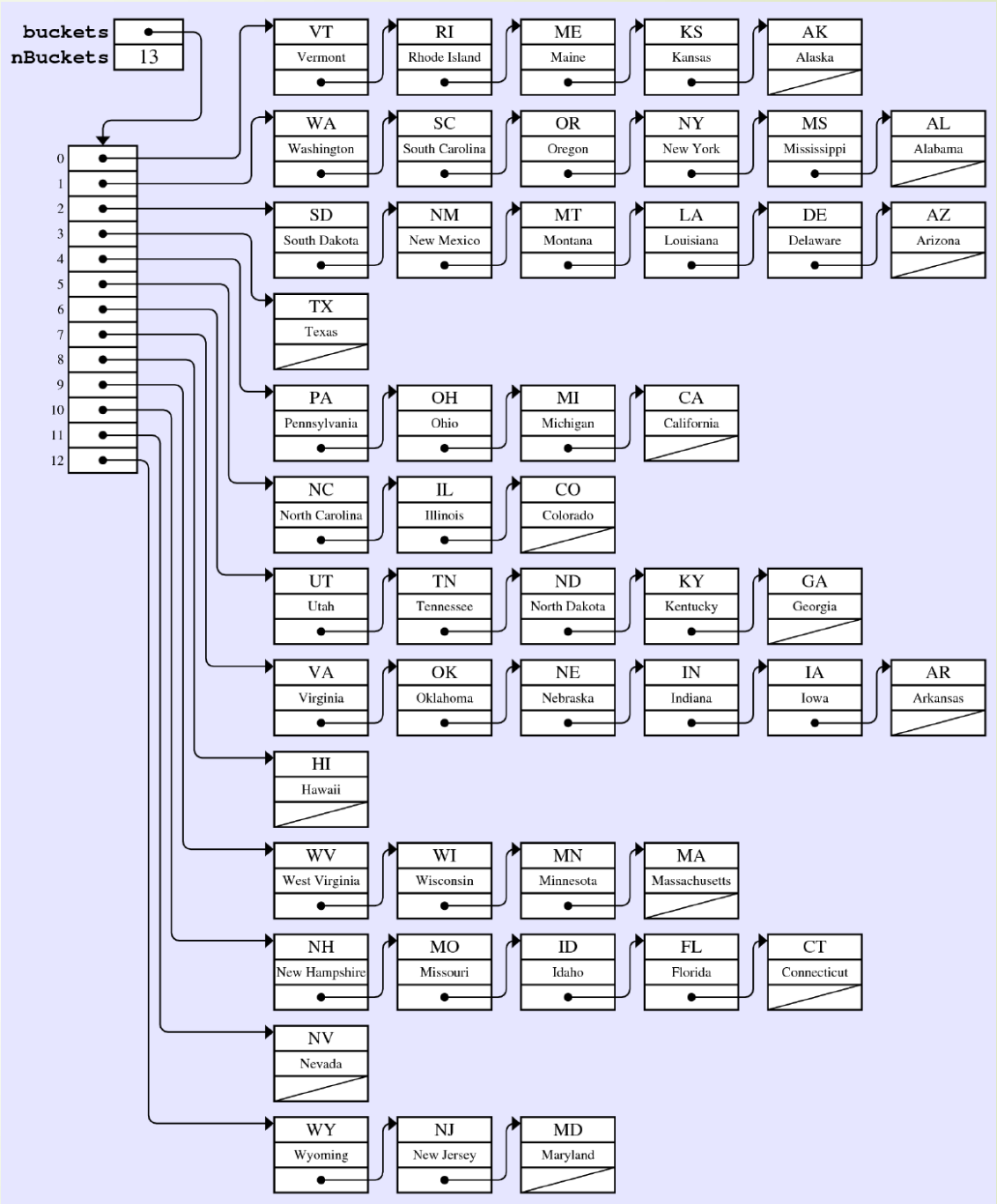
- Задача 4:
  Напишете клас **StringMap**, реализиран чрез хеш-таблица представена чрез масив от свързани списъци и която използва хеш-функцията от предишния слайд.

- Задача 5:
  Пренапишете програмата от **Задача 3**, като използвате новия клас **StringMap.**

- **Забележка**:
  Реализацията, която направихме, използва масив от фиксиран брой кофи. Може да се направи реализация с разширение на масива, след като отношението между броят елементите в хеш-таблицата и броят на кофите надвиши някаква стойност, например 0.7.
  Да отбележим, че при разширение на масива се изисква задължително рехеширане (преизчисляване на хеш-кодовете).

Хеш-таблицата за пощенските кодове в САЩ представена като масив от свързани списъци.

Забележка:
Вместо масив от списъци, може да се използва по-ефективната от гледна точка на количеството памет стратегия наречена ***open addressing,*** при която двойките ***(key, value)*** се записват директно в масив:
https://en.wikipedia.org/wiki/Open_addressing



Сравнение между двете реализации на абстракцията **Map**

- Двете класически реализации на абстракцията **Map** са:

  - балансирани бинарни дърветата за търсене.

  - хеш-таблиците.

- При бинарните дървета елементите са наредени (сортирани по ключ) и могат да се обхождат в естествения ред на ключовете. Обхождането при хеш-таблиците е невъзможно да става в естествения ред на ключовете, то е разбъркано.
- Сложността на операциите при хеш-таблиците са **O(1)** в средния случай и **O(N)** в най-лошия. При дърветата, и в средния и в най-лошия случай сложността е **log(N).**

Използвани източници

- http://web.stanford.edu/dept/cs_edu/BXReader-Beta-2012.pdf (стр. 664-688)