Хеш-таблиці

Марія Любарська

24 квітня 2020

Принцип роботи хеш-таблиць

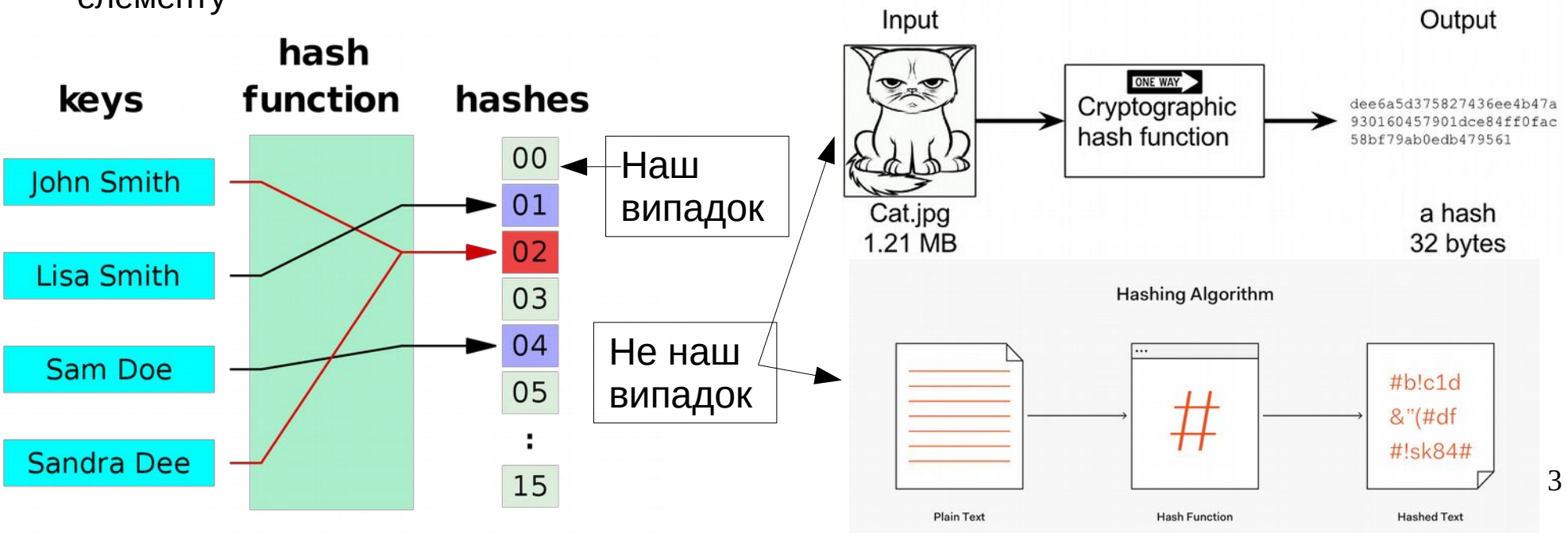
- *Хеш-таблиці* – це одна з реалізацій **асоціативного масиву**
- *Асоціативний масив* – це абстрактна структура даних, що дозволяє зберігати пари **ключ-значення** та яка підтримує такі операції:

  1) Операція додавання пари: INSERT(key, value)
  2) Операція пошуку за ключем: FIND(key)
  3) Операція видалення за ключем: REMOVE(key)

- Таким чином, *хеш-таблиця* – це масив, у якому зберігаються пари ключ-значення і який **використовує ключ для пошуку індексу елементу у масиві** (для подальшого додавання, пошуку або видалення відповідного елемента)
- Особливість хеш-таблиць полягає у тому, що відповідність між ключем та індексом у масиві встановлюється за допомогою **хеш-функції**

Хеш-функції

- *Хеш-функція* – у загальному, довільна функція, що на основі вхідного набору даних генерує відповідне значення певного формату
- Використовуються не тільки для хеш-таблиць!
- **У випадку хеш-таблиць** хеш-функція генерує індекс елементу масиві на основі ключа даного елементу



Два типи хеш-таблиць

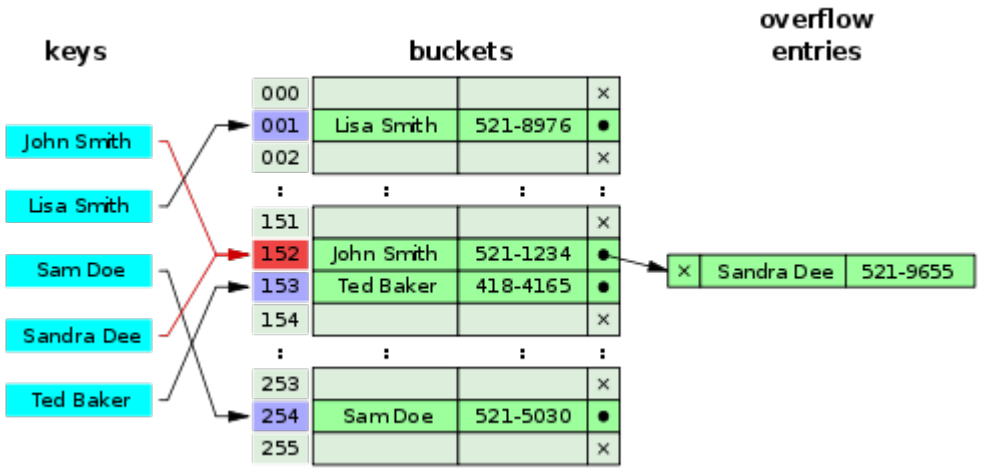
- За малої кількості пар ключ-значення (особливо якщо ключі відомі заздалегідь) буває можливим придумати **ідеальну хеш-функцію**, що для будь-якого ключа генеруватиме унікальний індекс
- Проте, зазвичай це не є можливим
- У такому випадку, у хеш-таблиці потенційно можуть виникати **колізії**
- *Колізія* – це випадок, коли два різних елемента у результаті виконання хеш-функції отримують однаковий індекс
- Хеш-таблиця має мати **механізм вирішення колізій**

- Хеш-таблиці поділяються на дві групи за типом механізмів вирішення колізій:

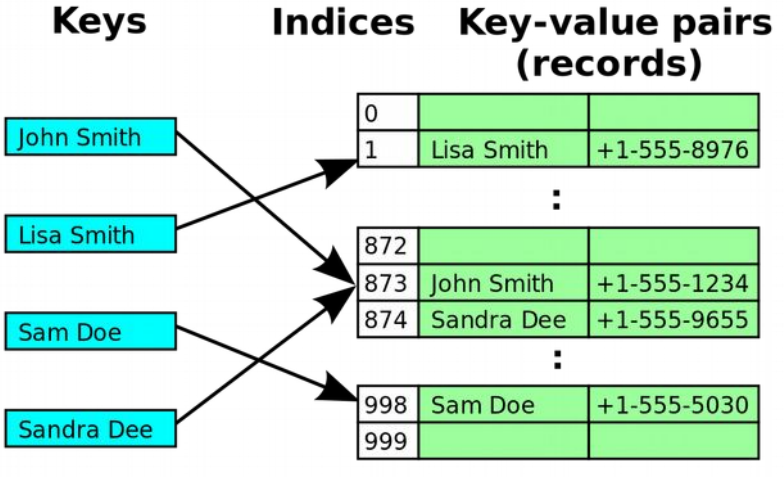
  1) **Хеш-таблиці з ланцюжками (chaining)**
  2) **Хеш-таблиці з відкритою адресацією (open adressing)**

Два типи хеш-таблиць

Хеш-таблиці з ланцюжками (chaining)



Хеш-таблиці з відкритою адресацією (open adressing)



Часова складність хеш-таблиць у порівнянні з іншими структурами даних

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)

std::map →

- Хеш-таблиці мають **гарну часову складність** для операцій пошуку, додавання та видалення **у середньому**
- Проте, вони **стають повільними** по всім операціям **у найгіршому випадку** (коли всі елементи мають однакове значення хешу і доводиться їх перебирати)

std::map (https://en.cppreference.com/w/cpp/container/map)

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

(1)

```
namespace pmr {
    template <class Key, class T, class Compare = std::less<Key>>
    using map = std::map<Key, T, Compare,
                         std::pmr::polymorphic_allocator<std::pair<const Key,T>>>
}
```

(2) (since C++17)

`std::map` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function `Compare`. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees.

Everywhere the standard library uses the *Compare* requirements, uniqueness is determined by using the equivalence relation. In imprecise terms, two objects a and b are considered equivalent (not unique) if neither compares less than the other: `!comp(a, b) && !comp(b, a)`.

`std::map` meets the requirements of *Container*, *AllocatorAwareContainer*, *AssociativeContainer* and *ReversibleContainer*.