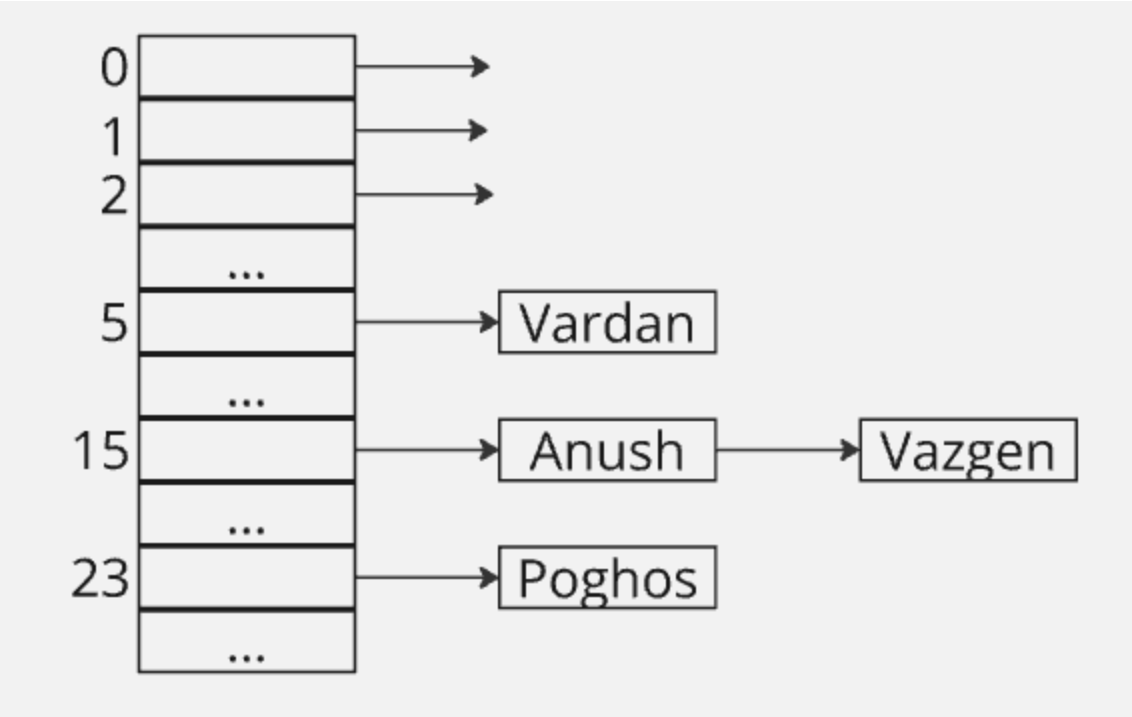
20. Definition of a hash function and a hash table, closed addressing, complexity analysis, std::unordered_set, std::unordered_map, std::unordered_multiset, std::unordered_multimap, examples of hash functions.

Пусть имеем следующую задачу: имеем N количество имён людей и их возраста. Распределим их в таблицу.
5 - Vardan
15 - Anush
23 - Poghos
15 - Vazgen




```
#include <iostream>
#include <vector>
#include <list>
#include <string>

int main() {
```

```
        int n; // количество людей
        std::cin >> n;
        std::vector<std::list<std::string>> buckets (130);
        // максимальный возраст 130

        for(int i = 0; i<n; i++){
                int age;
                std::string name;
                std::cin >> age >> name;
                buckets[age].push_back(name);
        }

    int k;
    std::cin >> k;
    for(const auto& name: backets[k]){
           std::cout << name << " ";
    }
    //печатаем list под k-ым индексом в массиве buckets (имена людей с одним
возрастом)
}
```

Данный алгоритм называется *bucket sort* (не алгоритм сортировки).

Для того, чтобы найти человека в buckets, если имеем возраст, будем искать его по индексу возраста, что займет в худшем случае O(n). Если имена распределены равномерно по таблице, то n/130.

Пусть на множестве объектов M задано отображение

$$h : M \to \mathbb{Z}^+$$

h называется *Hash function*, если h нильпотентна
h *нильпотентна*, т.е.

$$\text{Если } x_1 = x_2, \text{ то } h(x_1) = h(x_2)$$

Хэш-таблица — это структура данных, которая позволяет эффективно хранить и извлекать элементы, используя их ключи.
Метод разрешения коллизий, при котором все элементы с одинаковым хэшем хранятся в связанном списке (или в другой структуре) внутри одной ячейки таблицы называется *закрытой адресацией*.

Hash table с закрытой адесацией - это bucket sort где индексы определяются при помощи hash function h.

таблица сложностей операций

поиск	**в худшем случае O(n) / если хеш функция хорошая, то вероятностно амортизированная оценка - O(n/m = alpha)**
вставка	в худшем случае O(n) / амортизировано O(1)
удаление, если элемент найден	O(1)

Для улучшения поиска элемента введем следующее определение:
Хеш функция называется *хорошей*, если

$$\forall x_1 \neq x_2,\ h(x_1) \neq h(x_2) \text{ почти всегда.}$$

тир функция получает значение хеша и возвращает индекс bucket-a.

$$m(h) = h \bmod m, \text{ где m - это количество ячеек в массиве}$$

$$m : \mathbb{Z}^+ \to [0, m), \text{ где второе m - количество ячеек массива}$$

Для строк лучше всего использовать следующую хеш функцию:

$$h(s) = s0 \cdot p0 + s1 \cdot p1 + \cdots + sk - 1 \cdot pk - 1, \text{ где p - просто число, а si - i-ая буква строки}$$

Если значения map функции совпадают, это называется *коллизией*.

O(n/m = alpha) - оценка поиска элемента, где alpha называется *фактором балансировки*.

Rehashing

Сделаем так, чтобы

$$\alpha = \frac{n}{m} \leq 0.2$$

Пусть имеем m bucket-ов и n = 0.
В какой-то момент добавляется элемент и n = 1, alpha = 1/m; потом второй - n = 2, alpha = 2/m и тд. И в какой-то момент alpha > 0.2.
Тогда произведём следующие действия:

- Создадим новую хеш таблицу, которая например в 2 раза больше исходной

- Для каждого элемента из исходной таблицы применим новую map функцию

$$m(h) = h \bmod 2m$$

- Поставим элементы в соответствующие ячейки
- Сотрём исходную хеш таблицу

Перехеширование выполняется за O(n) операций, амортизировано - O(1).

std::unordered_set

- Контейнер для хранения уникальных элементов.
- Использует хэш-таблицу для быстрого доступа.
- Операции поиска, вставки, удаления имеют амортизированную сложность O(1).

Пример:

```
#include <unordered_set>
#include <iostream>

int main() {
    std::unordered_set<int> uset = {1, 2, 3, 4, 5};

    uset.insert(6);
    if (uset.find(3) != uset.end()) {
        std::cout << "Элемент найден\n";
    }

    return 0;
}
```

std::unordered_map

- Контейнер для хранения пар "ключ-значение".
- Ключи уникальны, значения могут повторяться.
- Операции поиска, вставки, удаления имеют амортизированную сложность O(1).

Пример:

```
#include <unordered_map>
#include <iostream>
```

```
int main() {
    std::unordered_map<std::string, int> umap = {{"Alice", 25}, {"Bob", 30}};

    umap["Charlie"] = 35;  // Вставка
    std::cout << "Возраст Bob: " << umap["Bob"] << std::endl;

    return 0;
}
```

`std::unordered_multiset`

- Аналог `std::unordered_set`, но позволяет хранить **неуникальные элементы**.
- Используется для хранения данных, где допустимы повторы, но порядок не важен.

Пример:

```
#include <unordered_multiset>
#include <iostream>

int main() {
    std::unordered_multiset<int> umset = {1, 1, 2, 3, 4};

    umset.insert(1);  // Повторяющиеся элементы допустимы

    for (int val : umset) {
        std::cout << val << " ";
    }

    return 0;
}
```

`std::unordered_multimap`

- Аналог `std::unordered_map`, но позволяет **неуникальные ключи**.
- Используется для сценариев, где одному ключу может соответствовать несколько значений.

Пример:

```
#include <unordered_multimap>
#include <iostream>

int main() {
    std::unordered_multimap<std::string, int> ummap;

    ummap.insert({"Alice", 25});
    ummap.insert({"Alice", 30});  // Два значения для одного ключа

    for (auto& [key, val] : ummap) {
        std::cout << key << ": " << val << std::endl;
    }

    return 0;
}
```