БК №536
КУРС «ПРОГРАММИРОВАНИЕ В ЗАДАЧАХ РАДИОЛОКАЦИИ»
РАЗДЕЛ IV «СТРУКТУРЫ ДАННЫХ»

ЛЕКЦИЯ №6

«АССОЦИАТИВНЫЕ МАССИВЫ. ХЭШ-ТАБЛИЦЫ»

Hash

Функция, отображающая объект ключа в целое число (обычно битовое представление числа имеет заранее заданную длину).

$$hash: Key \rightarrow \mathbb{Z}$$

где Key — множество ключей

Обязательное требование к хэш-функциии:

- детерменированность

Желаемые качества хэш-функции:

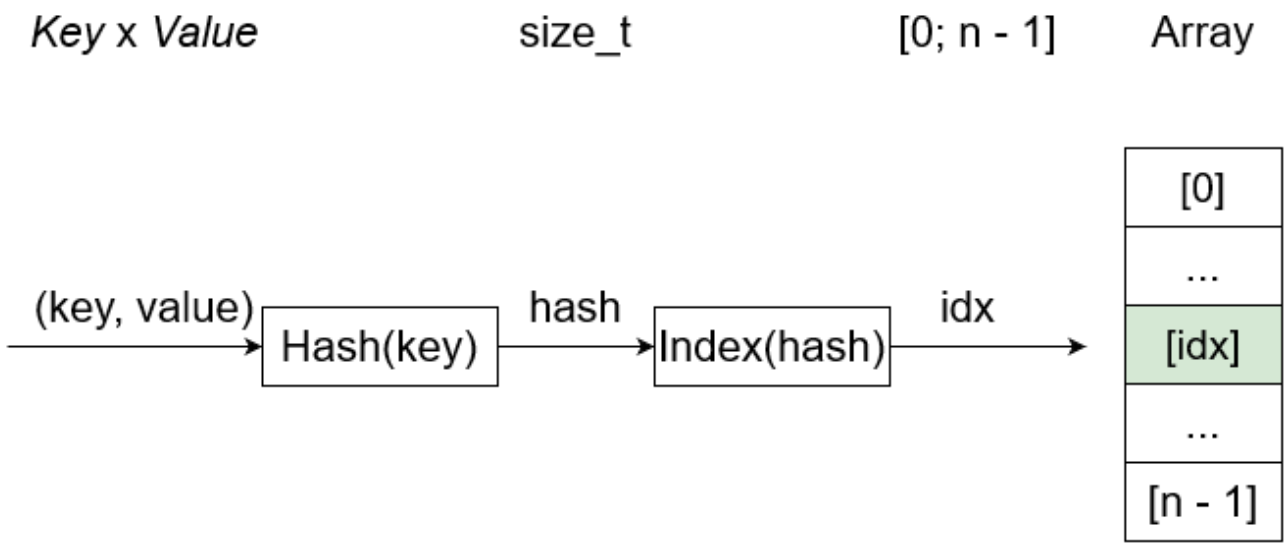
- быстрое вычисление;
- равномерность.

Для использования в таблице размера n необходимо сужать область значений:

$$hash: Key \rightarrow \mathbb{Z}_n$$

Hash-table

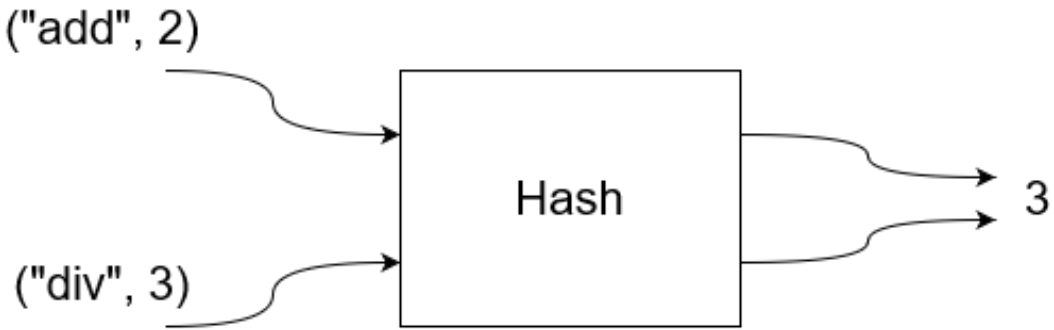
- ассоциативная;
- строится на динамическом массиве размера n;
- хэши — индексы массива;
- неотсортированная.



Коллизии

Коллизия — два ключа, один хэш

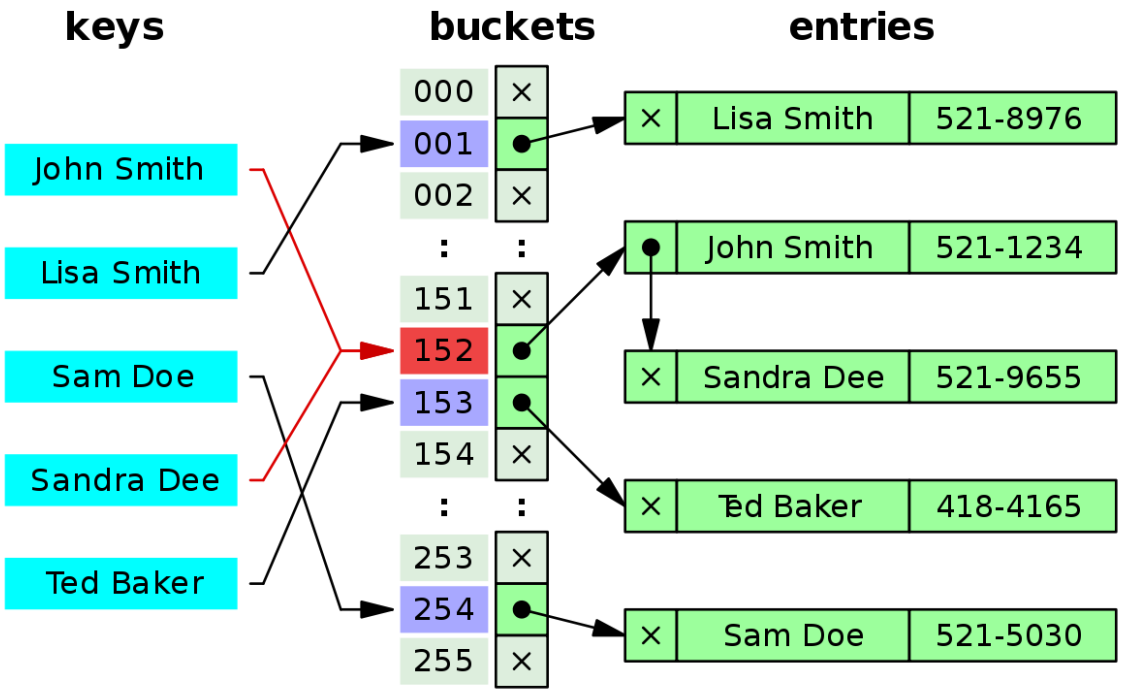
- на практике возникают почти всегда;
- их необходимо разрешать, но сами по себе не являются ошибкой.



Метод цепочек

Альтернативные названия — закрытая адресация, открытое хэширование

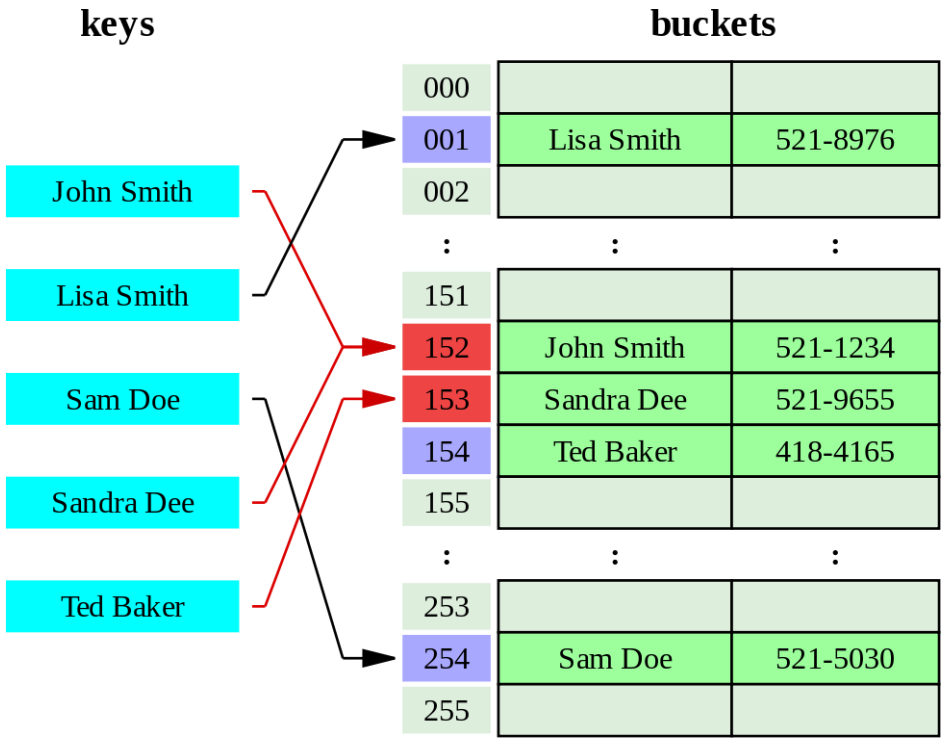
- элементы массива — обычно связные списки;
- элементы связного списка — узлы с данными ключ-значение.



Пробирование

Альтернативные названия — открытая адресация, закрытое хэширование

- элементы массива — узлы с данными ключ-значение, а также индикатором, удален ли узел;
- при коллизии ищется свободный слот, следующий после слота, соответствующего хэшу;
- поиск может осуществляться с различными интервалами.



Характеристики таблицы

- Коэффициент загрузки — нужен для определения, когда перестраивать таблицу

  $loadFactor = \frac{size}{capacity}$,

  где $size$ — текущее количество элементов, $capacity$ — размер таблицы
- Частота коллизий — нужна для оценки хэш-функции

  $f = \frac{colisionCount}{addCount}$,

  где $colisionCount$ — количество коллизий, $addCount$ — общее число добавлений

	Низкая f	Высокая f
Низкий loadFactor	Все нормально	Вероятно, плохая хэш-функция
Высокий loadFactor	Нужно перестраивать таблицу	Нужно перестраивать таблицу

Получение индекса

Остаток от деления

```
def index(key):
    # получаем целое число
    hashCode = hash(key)
    # ужимаем до индекса таблицы
    return hashCode % capacity
```

- легко реализуется, понятный;
- желательно, чтобы размер таблицы был простым числом;
- при расширении таблицы необходимо знать следующий подходящий размер.

Мультипликативный метод

```
def index(key):
    # получаем целое число
    hashCode = hash(key)
    # ужимаем до индекса таблицы
    r = key * coef - int(key * coef)
    return int(capacity * r)
```

- нет спец. требований к размеру таблицы, но обычно берут степень двойки;
- coef — число [0..1], обычно золотое сечение.

Добавление

Пусть coef — заранее заданное значение [0..1], data — массив, k — коэффициент пробирования

Закрытая адресация

```
def add(key, value):
    if loadFactor >= coef:
        resize()
    idx = index(key)
    bucket = data[index]
    bucket.pushFront(Node(key, value))
```

Открытая адресация

```
def add(key, value):
    if loadFactor >= coef:
        resize()
    idx = index(key)
    while not data[idx].empty():
        idx += k #k – коэф. проб.
    data[idx] =
         Node(key, value, false)
    # false – флаг, ячейка заполнена
    # true – ячейка удалена
```

Поиск

Пусть data — массив, k — коэффициент пробирования

Закрытая адресация

```
def find(key) -> Value:
    idx = index(key)
    return data[idx].find(key)
```

Открытая адресация

```
def remove(key) -> Value:
    idx = index(key)
    i = 0
    while data[idx].empty() ||
          data[idx].key != key:
        idx += k
        ++i
        if i >= size: NodeNotFind
    return data[idx].value
```

Удаление

Пусть data — массив, k — коэффициент пробирования

Закрытая адресация

```
def remove(key):
    idx = index(key)
    data[idx].remove(key)
```

Открытая адресация

```
def remove(key):
    idx = index(key)
    while data[idx].key != key:
        idx += k
    data[idx].empty = true
```

Сложности операций

	Лучший случай	Средний случай	Худший случай
Добавление	Amort(O(1))	Amort(O(1))	Amort(O(1)) — цепочки O(n) — пробирование
Поиск	Θ(1)	O(1)	O(n)
Удаление	Θ(1)	O(1)	O(n)

Важно: сложности представлены без учета сложности подсчета хэша!

Достоинства и недостатки

Достоинства

- простота идеи и реализации;
- константная сложность основных операций в среднем случае.

Недостатки

- сложность подбора хэш-функции, особенно для пользовательских типов;
- деградация сложности до линейной для основных операций при высокой загрузке или плохой хэш-функции;
- нельзя использовать в задачах с постоянным потоком данных и требуемой постоянной сложностью операций;
- требуется время на инициализацию.

Применение

- когда известен примерный объем данных, который будет содержаться в таблице, и он достаточно велик;
- когда заранее известны ключи и/или для них можно подобрать хорошую хэш-функцию;

Особенности реализации

Передача функции как параметра

```
template<typename Key>
size_t index(const Key& key, size_t (*hash)(const Key&)) {
    return hash(key) % SIZE;
}
```

Используя std::function (заголовочный файл functional)

```
template<typename Key>
size_t index(const Key& key, std::function<size_t(const Key&)>& hash) {
    return hash(key) % SIZE;
}
```