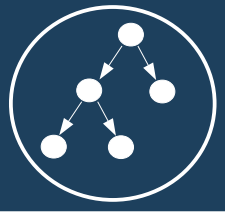
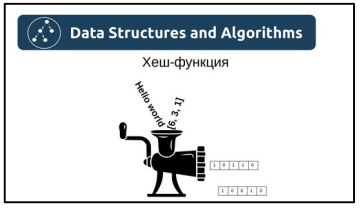


Data Structures and Algorithms

Структуры данных. Ассоциативный массив на основе хеш-таблиц

Список лекций необходимых для занятия

Перед просмотром этого занятия нужно просмотреть следующие лекции.



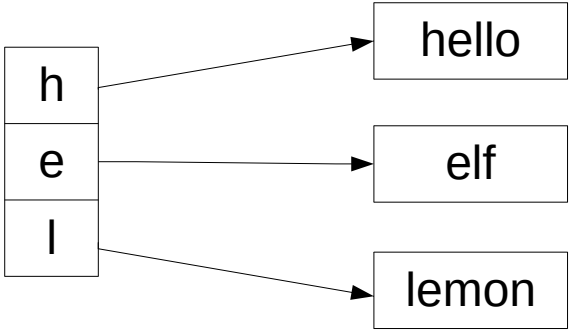
Хеш-функция

Ассоциативный массив

Ассоциативный массив — абстрактный тип данных, позволяющий хранить пары вида «(ключ, значение)». Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

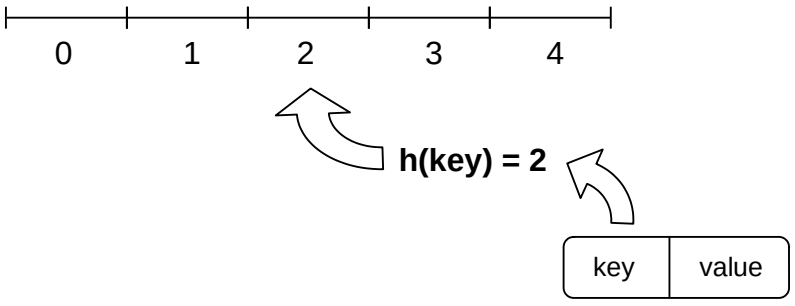
Поддерживаемые операции:

- Добавление пары
- Удаление пары
- Поиск пары и поиск значения по ключу
- Получение размера ассоциативного массива



Хеш-таблица

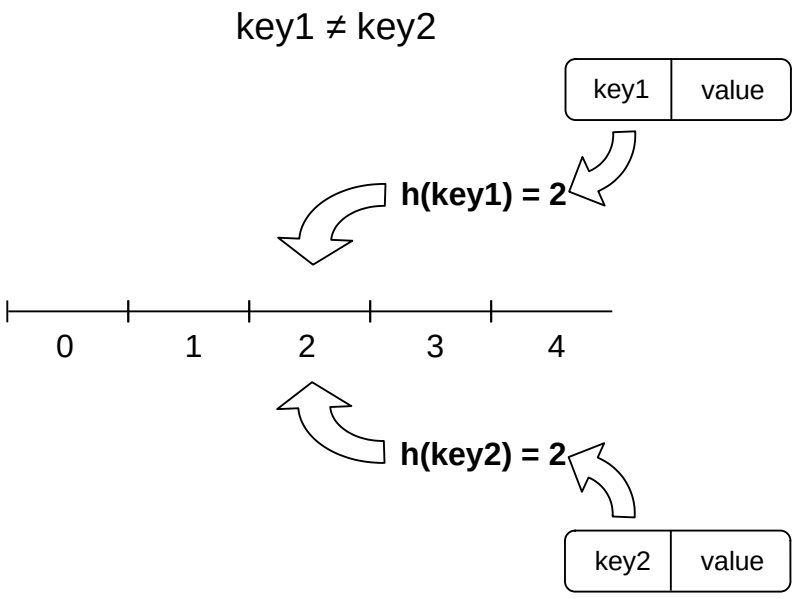
В качестве хеш-таблицы выступает массив (или список) в котором элементы хранятся парами ключ-значение. Индекс для хранения пары определяется хеш-кодом ключа. При этом хеш-функция выбирается такая, что бы генерировать хеш-код в диапазоне индексов этого массива.



Разрешение коллизий

Диапазон всех ключей гораздо больше диапазона индексов в хеш таблице. Таким образом возможны коллизии. Это означает, что для разных ключей генерируется один и тот же хеш-код и как следствие они должны быть размещены по одному и тому же индексу.

Пример коллизии

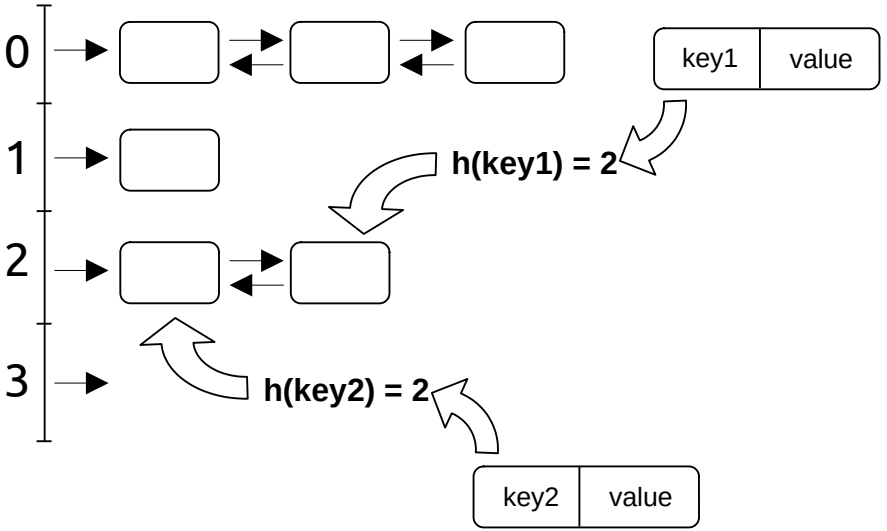


Способы разрешения коллизий

1) Метод списков
2) Открытая адресация

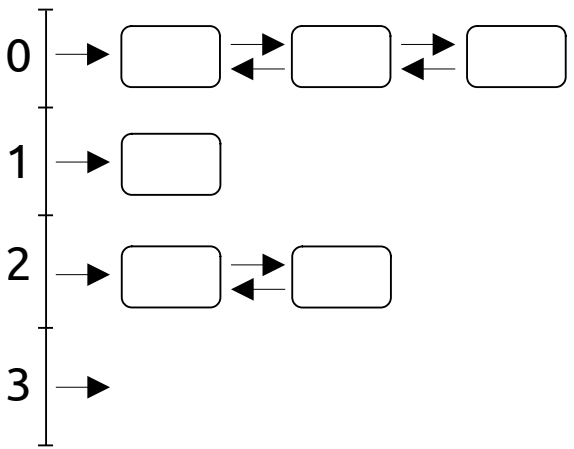
Метод списков

Одним из самых простых способов разрешения коллизий является метод списков. В таком случае элементами хеш-таблицы выступают списки содержащие пары ключ-значение. При коллизии пара ключ-значение добавляется в соответствующий список.



Некоторые термины

Hash Table (хеш-таблица) — массив для хранения пар ключ-значение.
Capacity (емкость) — количество элементов в массиве.
Load Factor(коэффициент загрузки) — вещественное значение (по умолчанию 0.75) используемое в алгоритме работы хеш-таблицы.
Threshold (Порог) — пороговое значение количества добавленных элементов после достижения которого происходит увеличение хеш-таблицы. Обычно равен capacity * load factor.
Size (количество элементов) — количество элементов во всех списках хеш-таблицы.

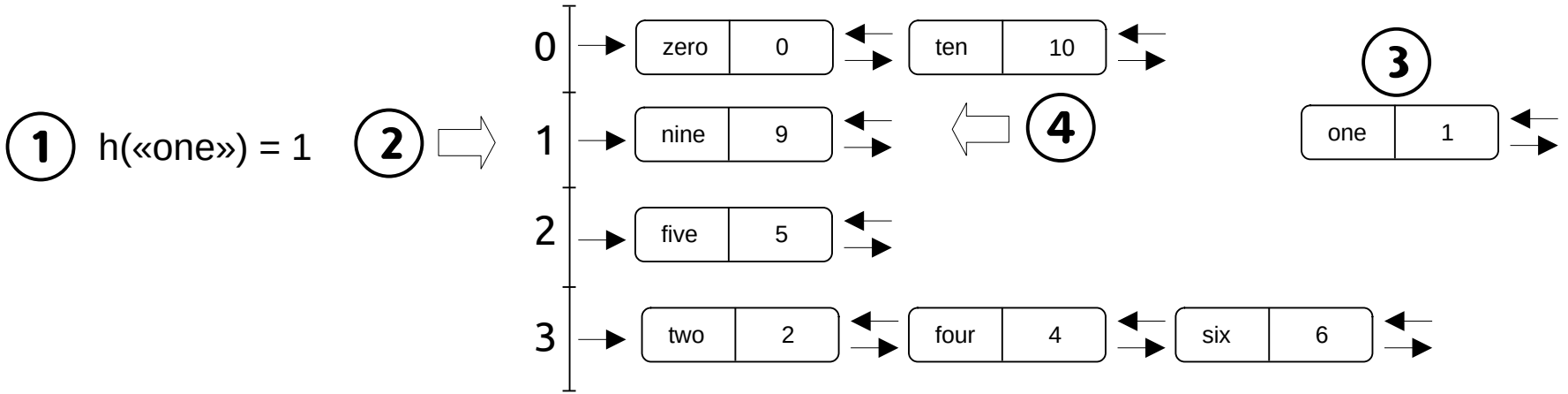


Добавление пары в хеш-таблицу

1) Вычисляется хеш-код на основе ключа. Должна использоваться хеш-функция генерирующая хеш в диапазоне от [0..capacity-1].
2) Полученный хеш используется как индекс в массиве. Начинают итерации по списку расположенном на этом индексе.
    1) Если при итерации оказалось, что элемент с таким ключом уже есть в списке тогда обновляют найденный элемент. Обновление заключается в замене значения на добавляемое.
    2) Если в списке нет элемента с таким ключом, тогда формируется новый узел списка и добавляется в его конец.

Добавление пары в хеш-таблицу (ключа нет в карте)

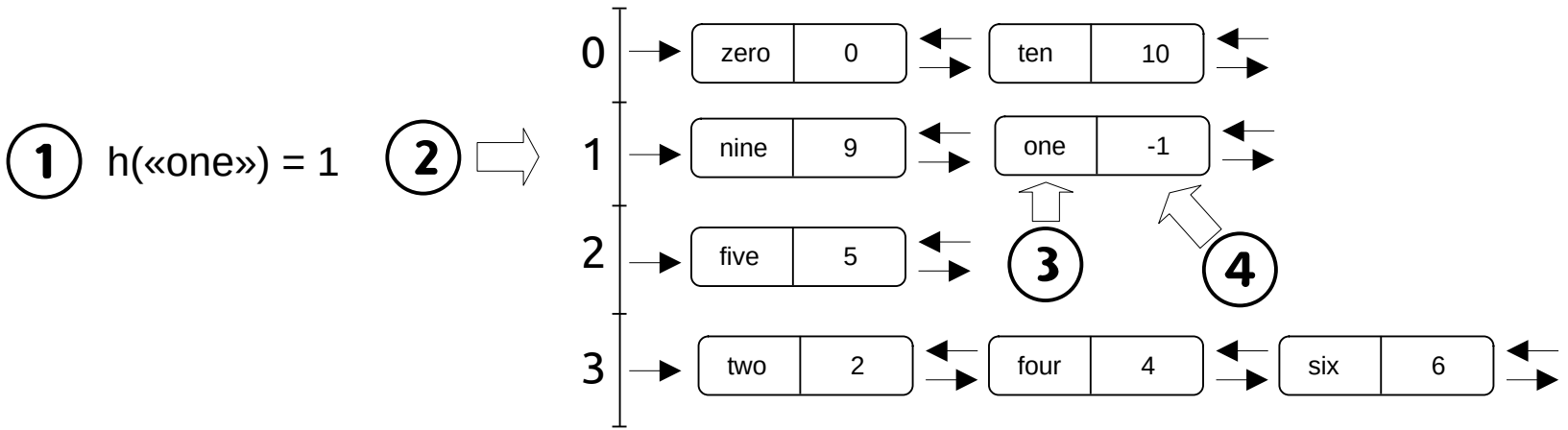
Предположим добавляется пара элементов (ключ - «one», значение — 1). Текущий размер хеш-таблицы примем равный 4. Процесс добавления выглядит следующим образом:



1) Вычисляем хеш-код ключа
2) Переходим к списку по индексу равному полученному хеш-коду. Проходим по списку. В списке нет элементов с таким ключом.
3) Создаем новый элемент хранящий пару ключ-значение
4) Добавляем полученный элемент в конец списка

Добавление пары в хеш-таблицу (ключа есть в карте)

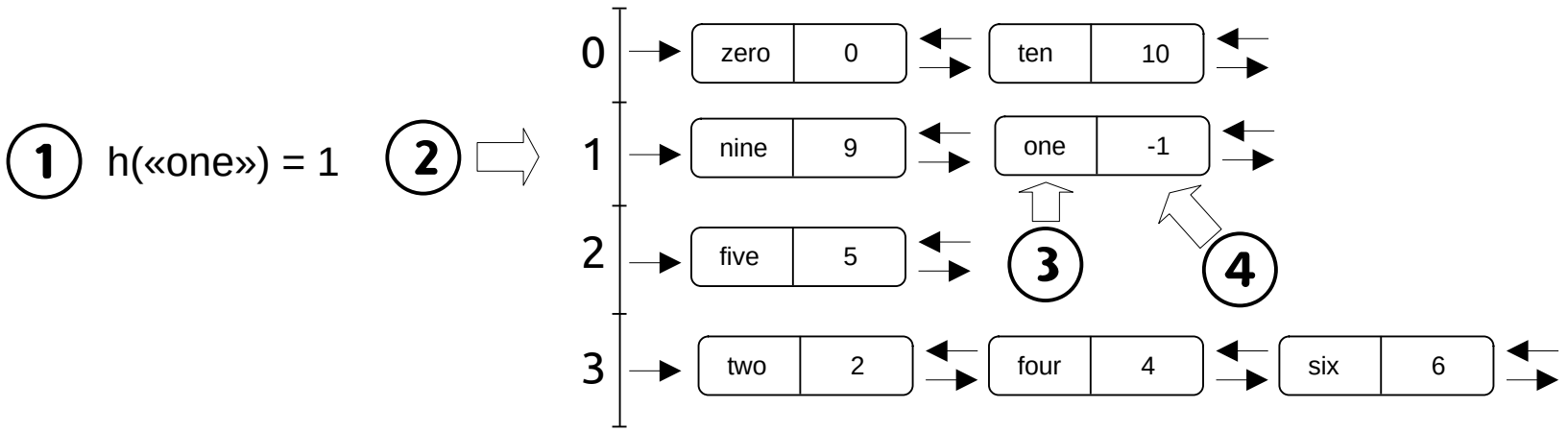
Предположим добавляется пара элементов (ключ - «one», значение — -1). Текущий размер хеш-таблицы примем равный 4. Процесс добавления выглядит следующим образом:



1) Вычисляем хеш-код ключа
2) Переходим к списку по индексу равному полученному хеш-коду. Проходим по списку. В списке есть элемент с таким ключом.
3) Переходим к найденному элементу
4) Заменяем значение на добавляемое

Поиск значения по ключу

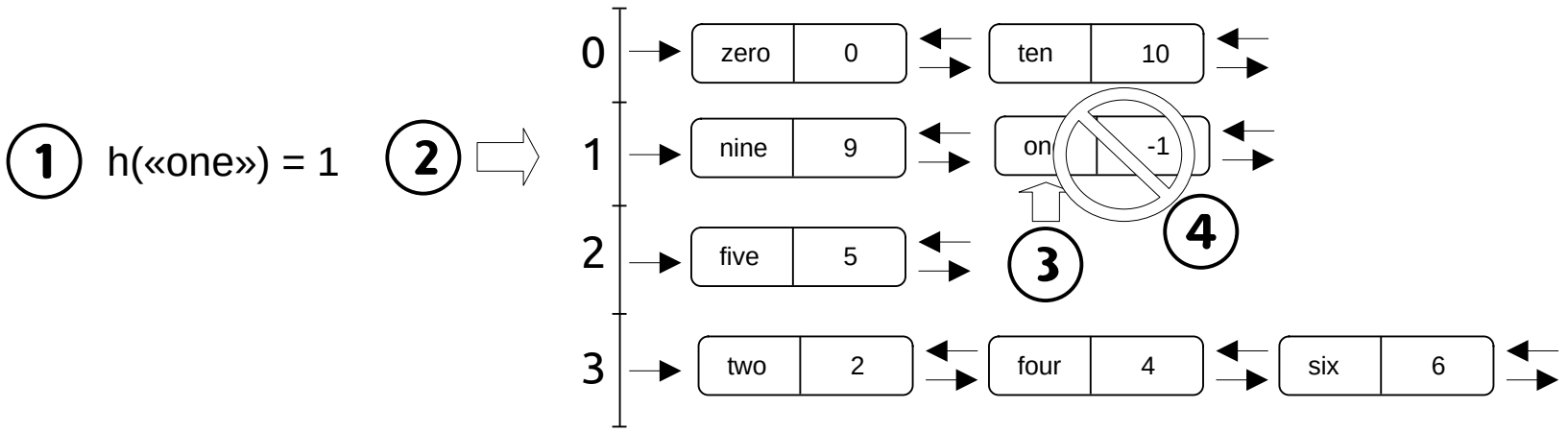
Выполняем поиск ключа - «one». Поиск выглядит так:



1) Вычисляем хеш-код ключа
2) Переходим к списку по индексу равному полученному хеш-коду. Проходим по списку. В списке есть элемент с таким ключом.
3) Переходим к найденному элементу
4) Возвращаем значение записанное в этом элементе

Удаление значения по ключу

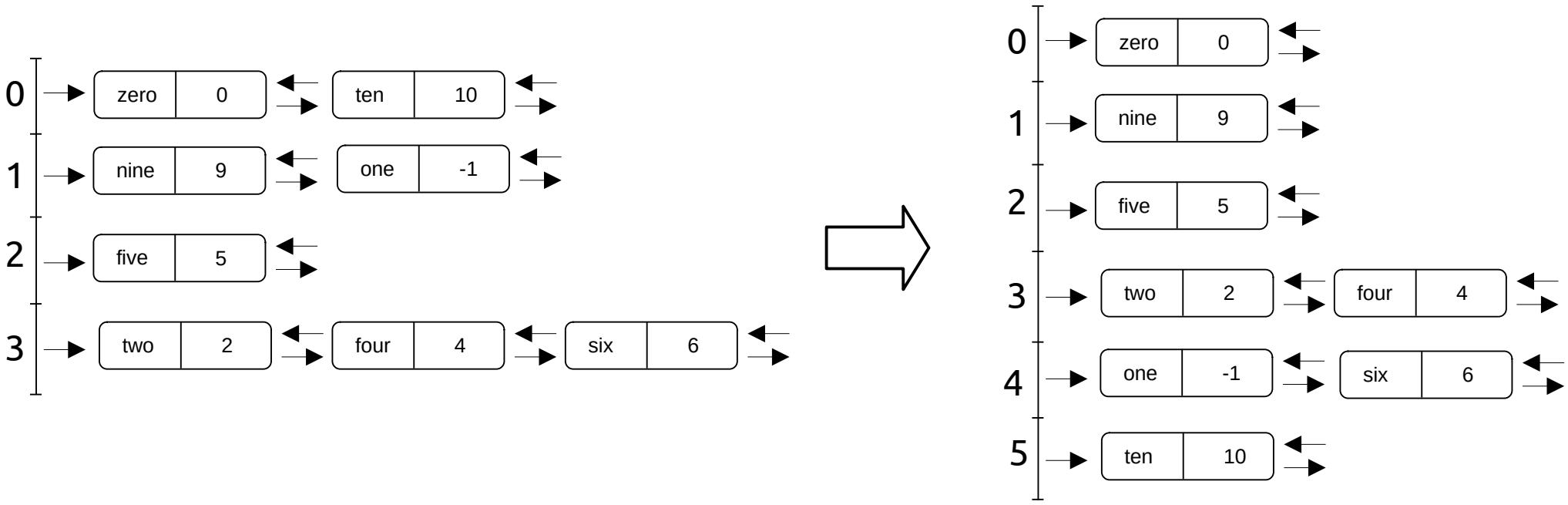
Удаляем пару по ключу - «one». Удаление выглядит так:



1) Вычисляем хеш-код ключа
2) Переходим к списку по индексу равному полученному хеш-коду. Проходим по списку. В списке есть элемент с таким ключом.
3) Переходим к найденному элементу
4) Удаляем найденный элемент

Увеличение размера хеш-таблицы

Как только количество элементов превышает пороговое значение вызывается алгоритм увеличения размера хеш-таблицы. Его суть сводиться к созданию новой хеш-таблицы большего размера. После чего добавляем все элементы из старой хеш-таблицы в новый. После этого удаляем старую хеш-таблицу.



Получение размера ассоциативного массива

Возможны два варианта получения количества элементов.

1) Объявление дополнительной переменной с начальным значением равной 0. При добавлении пары ключ значение увеличиваем значение на единицу. При удалении уменьшаем. В таком случае в любой момент времени значение этой переменной равно количеству элементов.
2) Пройти по всем элементам хеш-таблицы. Просуммировать количество элементов во всех списках.

Реализация на Python

Описание хеш-таблицы

```
class HashTable:

    class Node:
        def __init__(self, key, value):
            self.key = key
            self.value = value

    def __init__(self, load_factor=0.75):
        self.capacity = 16
        self.size = 0
        self.load_factor = load_factor
        self.hash_table = [[] for i in range(self.capacity)]
```

Методы добавления и увеличения размера

```
def put(self, key, value):
        n = hash(key) % self.capacity
        for node in self.hash_table[n]:
            if node.key == key:
                node.value = value
                break
        else:
            self.hash_table[n].append(HashTable.Node(key, value))
            self.size += 1
        if self.size > self.capacity * self.load_factor:
            self.size_up()

def size_up(self):
        self.capacity *= 2
        new_hash_table = [[] for i in range(self.capacity)]
        for node_list in self.hash_table:
            for node in node_list:
                n = hash(node.key) % self.capacity
                new_hash_table[n].append(node)
        self.hash_table = new_hash_table
```

Метод получения по ключу

```
def get(self, key):
    n = hash(key) % self.capacity
    for node in self.hash_table[n]:
        if node.key == key:
            return node.value
    return None
```

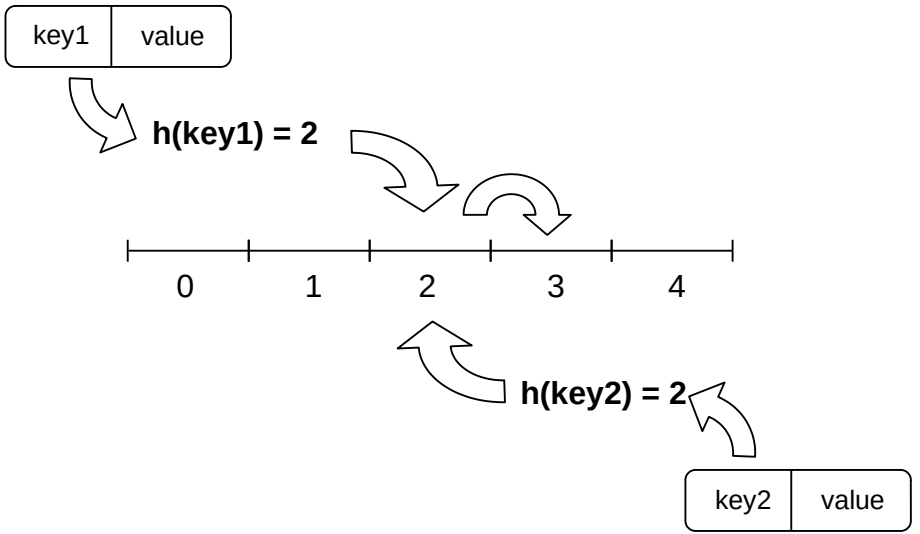
Метод удаления по ключу

```
def delete(self, key):
    n = hash(key) % self.capacity
    for i in range(len(self.hash_table[n])):
        if self.hash_table[n][i].key == key:
            del self.hash_table[n][i]
            self.size -= 1
```

Метод открытой адресации

В хеш-таблице хранятся сами пары ключ-значение. Алгоритм вставки элемента проверяет ячейку массива в некотором порядке до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент.

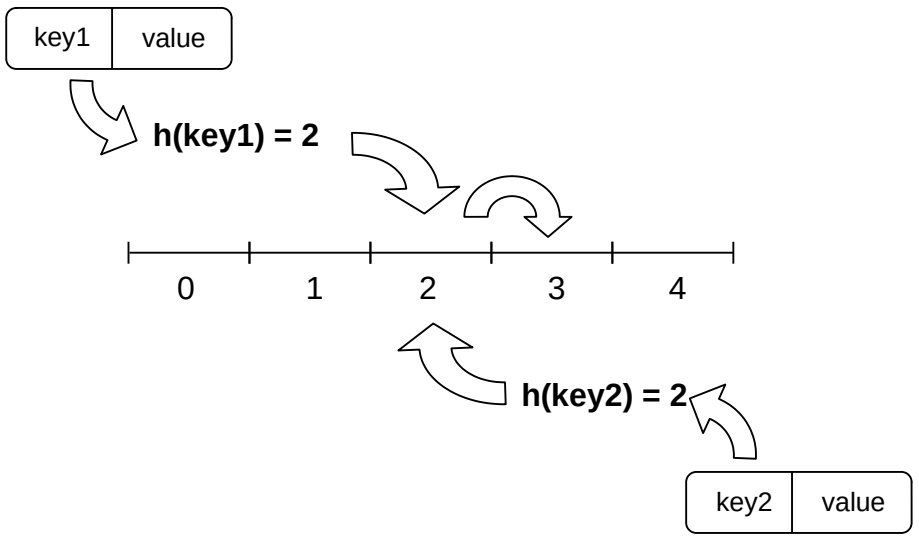
key1 ≠ key2



Линейное пробирование

Последовательность, в которой просматриваются ячейки хеш-таблицы, называется последовательностью проб. Одна из самых простых (и в то же время эффективная) это линейное пробирование. Его суть сводиться к просмотрю индексов (при коллизии) по порядку с шагом в 1 до нахождения первого свободного места.

key1 ≠ key2



Последовательность проб

$$h(key, i) = (h(key) + i) \bmod N$$

N - размер хеш-таблицы

i - номер попытки

Используемая хеш-функция

Хеш-таблица с открытой адресацией (линейное пробирование) чувствительна к виду используемой хеш-функции. И хотя работать она будет с любой, для оптимальной производительности стоит использовать семейство функций k - независимого хеширования. Одним из таких является псевдослучайное полиномиальное хеширование.

Идея хеширование следующая. Выбирается простое число p, также используют значение $p=2^n$. Выбираются как минимум k=5 случайных чисел (которые фиксируются) из диапазона [0..p-1]. Эти числа выступают коэффициентами полинома. В таком случае хеш-код вычисляется следующим образом.

$$h(key)=\left(\sum_{i=0}^{k-1} A_i \cdot n(key)^i\right) mod\ p$$

$n(key)$ - числовое представление ключа

A_i - случайное число выступающее как коэффициент полинома

Пример псевдослучайного полиномиального хеширования

Выбираем значение $p = 2^4 = 16$

Генерируем 5 случайных чисел в диапазоне от 0..15

i	0	1	2	3	4
A_i	7	2	13	5	11

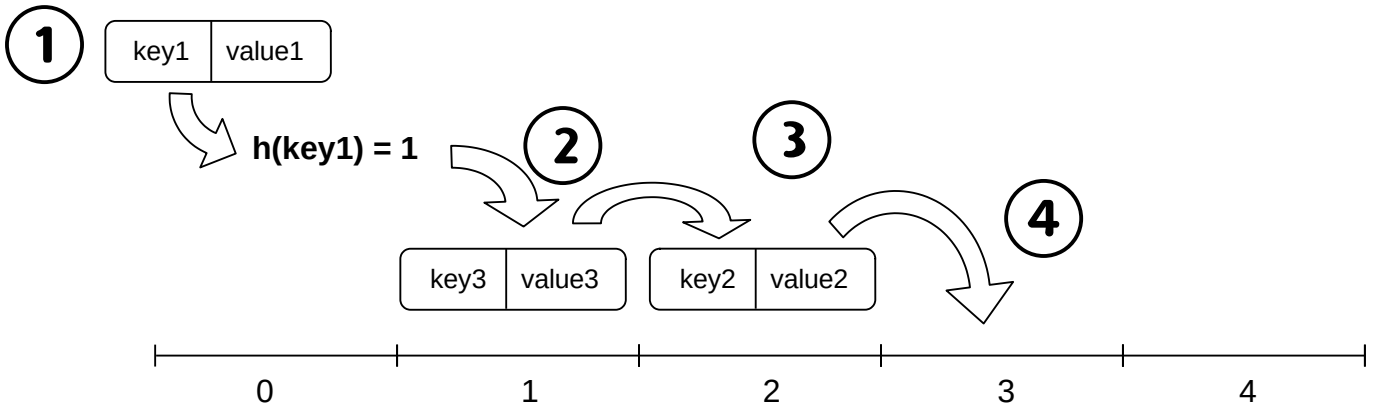
Полученная хеш-функция равна

$$h(key) = \left(\sum_{i=0}^{k-1} A_i \cdot n(key)^i \right) mod\ p = \left(A_0 + A_1 \cdot n(key) + A_2 \cdot n(key)^2 + A_3 \cdot n(key)^3 + A_4 \cdot n(key)^4 \right) mod\ 16$$

$$h(key) = \left(7 + 2 \cdot n(key) + 13 \cdot n(key)^2 + 5 \cdot n(key)^3 + 11 \cdot n(key)^4 \right) mod\ 16$$

Добавление пары (ключа нет в хеш-таблице)

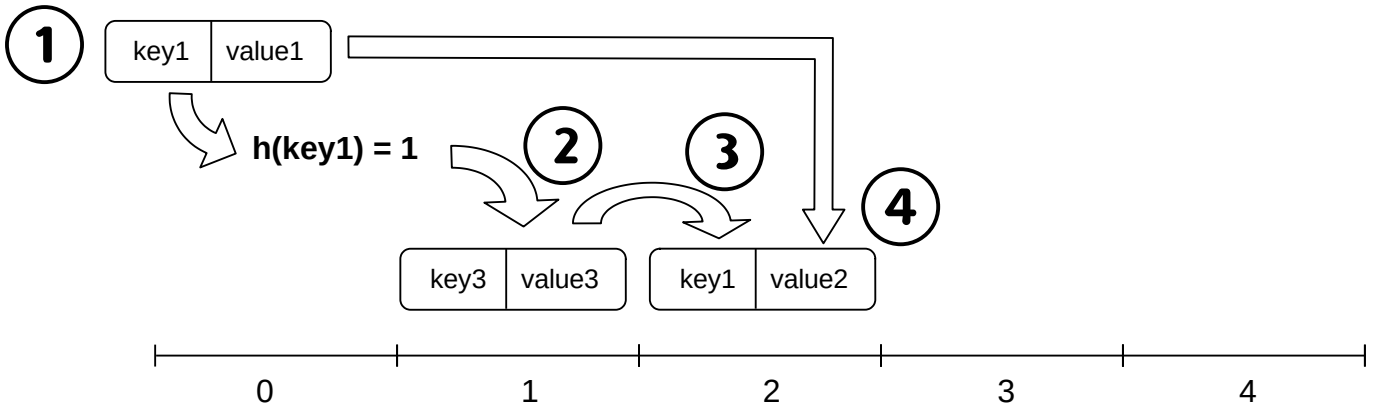
При добавлении пары вычисляем индекс на который попадет элемент, если на этом индексе нет элемента то добавляем пару ключ значение и заканчиваем. Если же этот индекс уже занят, то переходим к следующему элементу и так далее.



1) Вычисляем хеш-код ключа
2) Переходим к индексу равному полученному хеш-коду. Этот индекс уже занят. Переходим к следующему индексу.
3) Этот индекс также занят. Переходи к следующему.
4) Этот индекс свободен. Записываем пару ключ-значение на полученный индекс.

Добавление пары (ключ есть в хеш-таблице)

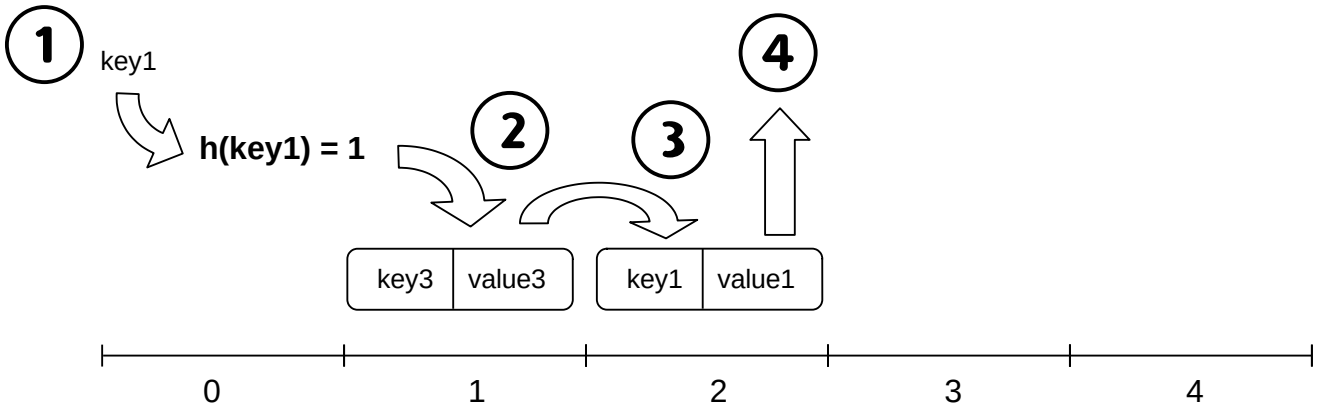
При добавлении пары вычисляем индекс на который попадет элемент, если на этом индексе есть элемент с равным ключом заменяем значение на добавляемое и заканчиваем.



1) Вычисляем хеш-код ключа
2) Переходим к индексу равному полученному хеш-коду. Этот индекс уже занят. Переходим к следующему индексу.
3) Этот индекс также занят. Хеш-код пары равен добавляемому.
4) Заменяем значение пары, на добавляемое значение.

Поиск пары по ключу

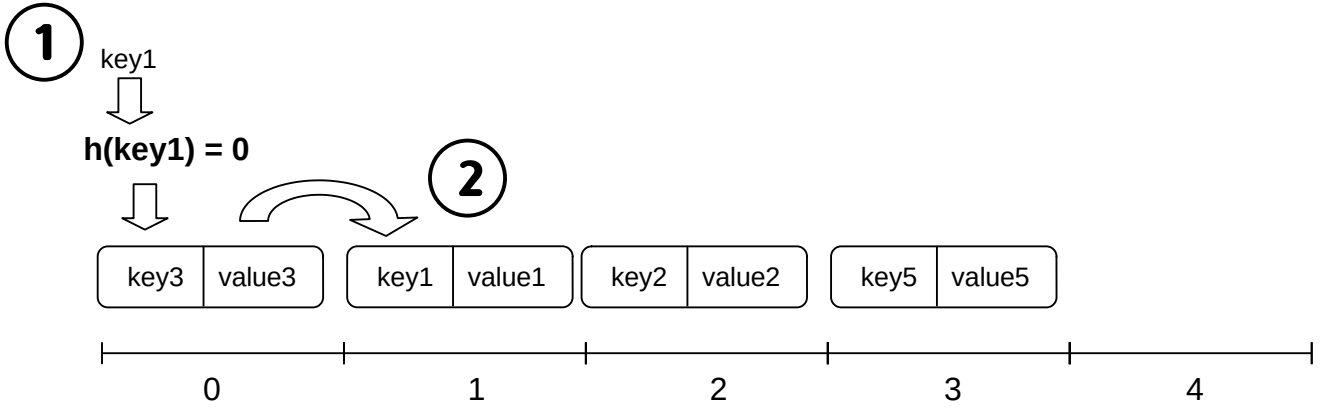
При поиске пары по ключу вычисляем хеш-код ключа. Переходим по индексу равному полученному хеш-коду. Если на индексе нет пары, то поиск неудачен. Если ключ пары на индексе равен искомому, возвращаем значение, если нет переходим к следующему индексу.



1) Вычисляем хеш-код ключа
2) Переходим к индексу равному полученному хеш-коду. Там есть пара. Ее ключ не равен искомому. Переходим к следующему индексу.
3) На этом индексе есть пара. Ее ключ равен искомому.
4) Возвращаем значение найденной пары.

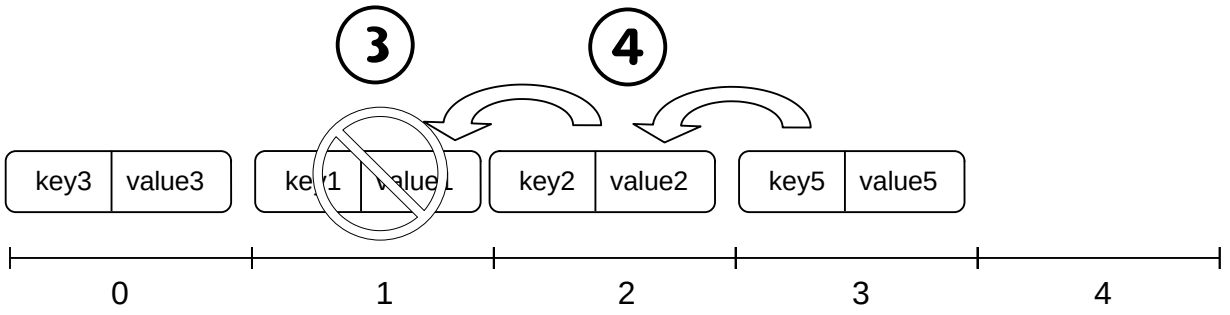
Удаление пары по ключу

Вычисляем хеш-код ключа. Переходим по индексу равному полученному хеш-коду. Если на индексе нет пары, заканчиваем такой пары нет. Методом проб находим пару ключ которой равен удаляемому. Удаляем пару. После чего выполняем смещение все последующих пар (до следующей пустой позиции) на одну позицию влево.



1) Вычисляем хеш-код ключа
2) Переходим к индексу равному полученному хеш-коду. Тут есть пара. Ключ не равен удаляемому. Переходим к следующему элементу. Ключ пары равен удаляемому.

Удаление пары по ключу



3) Удаляем найденный элемент.
4) Производим сдвиг пар (до следующего отсутствующего элемента) на одну позицию влево.

Увеличение размера хеш-таблицы

В случае использования линейного пробирования увеличение размера стоит выполнять при коэффициенте заполнения 0.5. Увеличивают размер массива в два раза и просто последовательно добавляют пары (используя тот же алгоритм добавления) из старого массива в новый. После окончания работы освобождают память (при необходимости) занимаемую старым массивом. В качестве хеш-таблицы используем новый.

Получение размера ассоциативного массива

Возможны два варианта получения количества элементов.

1) Объявление дополнительной переменной с начальным значением равной 0. При добавлении пары ключ значение увеличиваем значение на единицу. При удалении уменьшаем. В таком случае в любой момент времени значение этой переменной равно количеству элементов.
2) Пройти по всем элементам хеш-таблицы. Просуммировать количество существующих элементов.

Реализация на Java

Описание класса хеш-таблицы и пары

```
class HashTable {
     private class Pair {
          public String key;
          public Object value;

          public Pair(String key, Object value) {
               this.key = key;
               this.value = value;
          }
     }

     private Pair[] pairArray;
     private int capacity = 16;
     private int[] polyCoeff = new int[5];
     private int size = 0;

     public HashTable() {
          super();
          pairArray = new Pair[capacity];
          calculatePolyCoeff();
     }
```

Методы для генерации коэффициентов полинома и вычисления хеш-кода на его основе

```
private final void calculatePolyCoeff() {
          Random rn = new Random();
          for (int i = 0; i < polyCoeff.length; i++) {
               polyCoeff[i] = rn.nextInt(capacity);
          }
     }

private int calculateNewHash(int oldHash) {
     int newHash = polyCoeff[0];
     for (int i = 0; i < polyCoeff.length - 1; i++) {
          newHash = newHash * oldHash + polyCoeff[i + 1];
     }
     return Math.abs(newHash % capacity);
}
```

Метод добавления пары и увеличения размера

```
public void addPair(String key, Object value) {
        int index = calculateNewHash(key.hashCode());
        for (;;) {
                if (pairArray[index] == null) {
                        pairArray[index] = new Pair(key, value);
                        size += 1;
                        break;
                } else if (pairArray[index].key.equals(key)) {
                        pairArray[index].value = value;
                        break;
                } else {
                        index = (index + 1) % capacity;
                }
        }
        if (size > capacity / 2) {
                upResize();
        }
}

private void upResize() {
        int newSize = capacity * 2;
        if (newSize < 0) {
                throw new IllegalArgumentException("");
        }
        Pair[] oldPairArray = pairArray;
        pairArray = new Pair[newSize];
        capacity = newSize;
        calculatePolyCoeff();
        for (Pair pair : oldPairArray) {
                if (pair != null) {
                        addPair(pair.key, pair.value);
                }
        }
}
```

Метод получения значения по ключу

```
public Object get(String key) {
     int index = calculateNewHash(key.hashCode());
     for (;;) {
          if (pairArray[index] == null) {
               return null;
          } else if (pairArray[index].key.equals(key)) {
               return pairArray[index].value;
          } else {
               index = (index + 1) % capacity;
          }
     }
}
```

Метод удаления пары по ключу

```
public boolean remove(String key) {
     int index = calculateNewHash(key.hashCode());
     for (;;) {
          if (pairArray[index] == null) {
               return false;
          } else if (pairArray[index].key.equals(key)) {
               pairArray[index] = null;
               for (;;) {
                    pairArray[index] = pairArray[(index + 1) % capacity];
                    if (pairArray[index] == null) {
                         break;
                    }
                    index = (index + 1) % capacity;
               }
               size = size - 1;
               return true;
          } else {
               index = (index + 1) % capacity;
          }
     }
}
```

Метод для получения размера

```
public long getLength() {
     long length = 0;
     Node currentNode = head.next;
     for (; currentNode != tail;) {
          length += 1;
          currentNode = currentNode.next;
     }
     return length;
}
```

Двойное хеширование

Как уже было сказано алгоритм открытой адресации чувствителен к выбранному виду хеш-функции. Одним из лучших способов решение этой проблемы является использование двойного хеширования. При двойном хешировании используется хеш-функция вида

$$h(key, i) = \left(h_1(key) + i \cdot h_2(key)\right) mod\ p$$

i - номер попытки

$h_1(key), h_2(key)$ - вспомогательные хеш-функции выбор которых зависит от выбора p

К хеш-функции h_2 выдвигается ряд требований:

1) Она не должна возвращать 0
2) Она должна возвращать значения для циклического прохода по таблице
3) Желательно быстро вычисляться
4) Любое полученное значение должно быть взаимно простым с p

Двойное хеширование

Одним из способов определить вспомогательные хеш функции является следующее. p — простое число.

$$h_1(key) = n(key) \bmod p$$
$$h_2(key) = 1 + n(key) \bmod (p-1)$$

$n(key)$ - числовое представление ключа

«Ленивое» удаление

При интенсивной работе с ассоциативным массивом можно уменьшить нагрузку на подсистему выделения и освобождения памяти (и как следствие поднять производительность) путем использования «ленивого» удаления. Под «ленивым» удалением понимают фиктивное удаление элемента, заключающееся просто в пометке его как удаленного. Для этого к паре ключ-значение добавляют элемент, выступающий в роли маркера существует пара или удалена. Фактически удаления пары не происходит, поэтому при вставке на ее место можно просто заметить ее ключ и значение на добавляемые. Таким образом не нужно многократно выделять и освобождать память занимаемую парами.

Реализация на Fortran

Описание пары и хеш-таблицы

```
type Pair
    character(len = 30)::key
    integer::value
    logical::is_present
end type Pair


type HashTable
    type(Pair), allocatable::pair_table(:)
    integer::prime_number(10)
    integer::prime_index = 1
    integer::capacity
    integer:: h_size= 0
    contains
        procedure,pass::init
        procedure,pass::put
        procedure,pass::calculate_hash
        procedure,pass::up_resize
        procedure,pass::get
        procedure,pass::remove
        procedure,pass::show
        procedure,pass::clear
        procedure,pass::destroy

end type HashTable
```

Процедура добавления пары

```
subroutine put(this, add_key, add_value,op_result)
    class(HashTable)::this
    character(len=*), intent(in)::add_key
    integer, intent(in)::add_value
    logical,intent(inout)::op_result
    integer::add_index, i
    integer::key_number_value
    integer::p

    if(this%h_size > this%capacity / 2) then
        call this%up_resize(op_result)
        if (.not. op_result) then
            return
        end if
    end if

    p = this%capacity
    key_number_value = this%calculate_hash(add_key)
    i = 0
    do
        add_index = mod((mod(key_number_value,p) + i * (1 + mod(key_number_value,p - 1))), p)
        if ( .not. this%pair_table(add_index)%is_present) then
            this%pair_table(add_index)%key = add_key
            this%pair_table(add_index)%value = add_value
            this%pair_table(add_index)%is_present = .true.
            this%h_size = this%h_size + 1
            op_result = .true.
            exit
        end if

        if (this%pair_table(add_index)%is_present) then
            if(this%pair_table(add_index)%key == add_key) then
                this%pair_table(add_index)%value = add_value
                op_result = .true.
                exit
            end if
        else
            i = i + 1
        end if
    end do
end subroutine put
```

Процедура увеличения размера

```
subroutine up_resize(this, op_res)
    class(HashTable)::this
    logical::op_res
    integer::i
    type(Pair), allocatable::temp_pair_table(:)
    if (this%prime_index == size(this%prime_number)) then
       op_res = .false.
       return
    end if
    this%prime_index = this%prime_index + 1
    temp_pair_table = this%pair_table
    this%capacity = this%prime_number(this%prime_index)
    allocate(this%pair_table(this%capacity))
    this%pair_table%is_present = .false.
    do i = 1, size(temp_pair_table)
        if (temp_pair_table(i)%is_present) then
            call this%put(temp_pair_table(i)%key, temp_pair_table(i)%value, op_res)
        end if
    end do
    deallocate(temp_pair_table)
    op_res = .true.
end subroutine up_resize
```

Функция для вычисления хеш-кода ключа

```
function calculate_hash(this, add_key)
    class(HashTable)::this
    character(len=*), intent(in)::add_key
    integer::calculate_hash
    integer::i
    calculate_hash = iachar(add_key(1:1))
    do i = 1, len_trim(add_key) - 1
        calculate_hash = calculate_hash * 31 + iachar(add_key(i+1:i+1))
    end do
    calculate_hash = abs(calculate_hash)
end function calculate_hash
```

Получение значения по ключу

```
subroutine get(this, key, get_value, op_result)
    class(HashTable)::this
    character(len=*), intent(in)::key
    integer, intent(inout)::get_value
    logical, intent(inout)::op_result
    integer::get_index, p, key_number_value, i
    p = this%capacity
    key_number_value = this%calculate_hash(key)
    i = 0
    do
        get_index = mod((mod(key_number_value,p) + i * (1 + mod(key_number_value,p - 1))), p)
        if ( .not. this%pair_table(get_index)%is_present) then
            op_result = .false.
            exit
        end if

        if (this%pair_table(get_index)%is_present) then
            if(this%pair_table(get_index)%key == key) then
                get_value = this%pair_table(get_index)%value
                op_result = .true.
                exit
            end if
        else
            i = i + 1
        end if
    end do
end subroutine get
```

Процедура для удаления по ключу

```
subroutine remove(this, key)
    class(HashTable)::this
    character(len=*), intent(in)::key
    integer::remove_index, p, key_number_value, i, next_remove_index
    p = this%capacity
    key_number_value = this%calculate_hash(key)
    i = 0
    do
        remove_index = mod((mod(key_number_value,p) + i * (1 + mod(key_number_value,p - 1))), p)
        if ( .not. this%pair_table(remove_index)%is_present) then
            exit
        end if
        if (this%pair_table(remove_index)%is_present) then
            if(this%pair_table(remove_index)%key == key) then
                this%pair_table(remove_index)%is_present = .false.
                this%h_size = this%h_size - 1
                do
                    i = i + 1
                    next_remove_index = mod((mod(key_number_value,p) + i * (1 + mod(key_number_value,p - 1))), p)
                    if (.not. this%pair_table(next_remove_index)%is_present) then
                        exit
                    end if
                    this%pair_table(remove_index) = this%pair_table(next_remove_index)
                    remove_index = next_remove_index
                end do
                exit
            end if
        else
            i = i + 1
        end if
    end do
end subroutine remove
```

Процедура для инициализации и очистки

```
subroutine init(this)
    class(HashTable)::this
    this%prime_number = [17, 37, 101, 239, 571, 1019, 5171, 15031, 50033, 100003]
    this%capacity =  this%prime_number(this%prime_index)
    allocate(this%pair_table(this%capacity))
    this%pair_table%is_present = .false.
end subroutine init

subroutine clear(this)
    class(HashTable)::this
    deallocate(this%pair_table)
    this%prime_index = 1
    this%capacity =  this%prime_number(this%prime_index)
    allocate(this%pair_table(this%capacity))
    this%pair_table%is_present = .false.
end subroutine clear
```

Список литературы

1) Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн // Алгоритмы: построение и анализ 3-е издание. — М.: «Вильямс», 2013. — С. 1328. ISBN 978-5-8459-1794-2
2) Роберт Седжвик, Кевин Уэйн «Алгоритмы на java 4-е издание» Пер. с англ. - М. : ООО "И.Д. Вильямс", 2013. ISBN 978-5-8459-1781-2.