ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΜΑΤΙΚΗΣ

Open Addressing Linear Probing Hash Table

Νικόλαος Γουρνάκης , Νικόλαος Χάσκαρης
It22023 , it22118

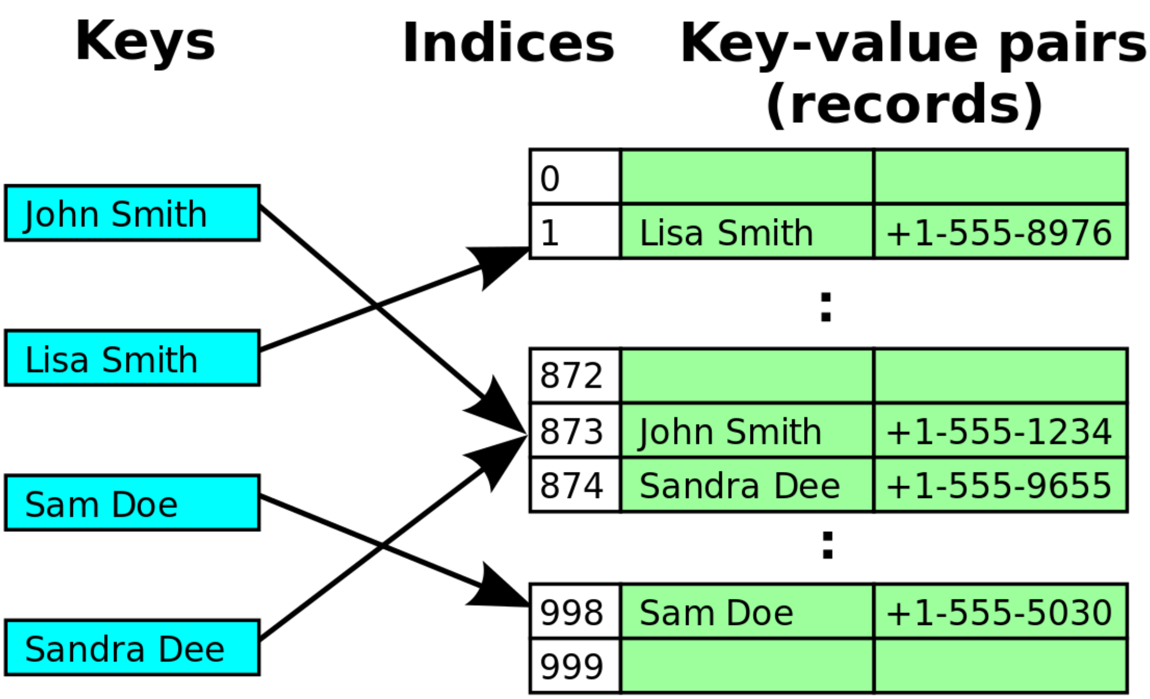


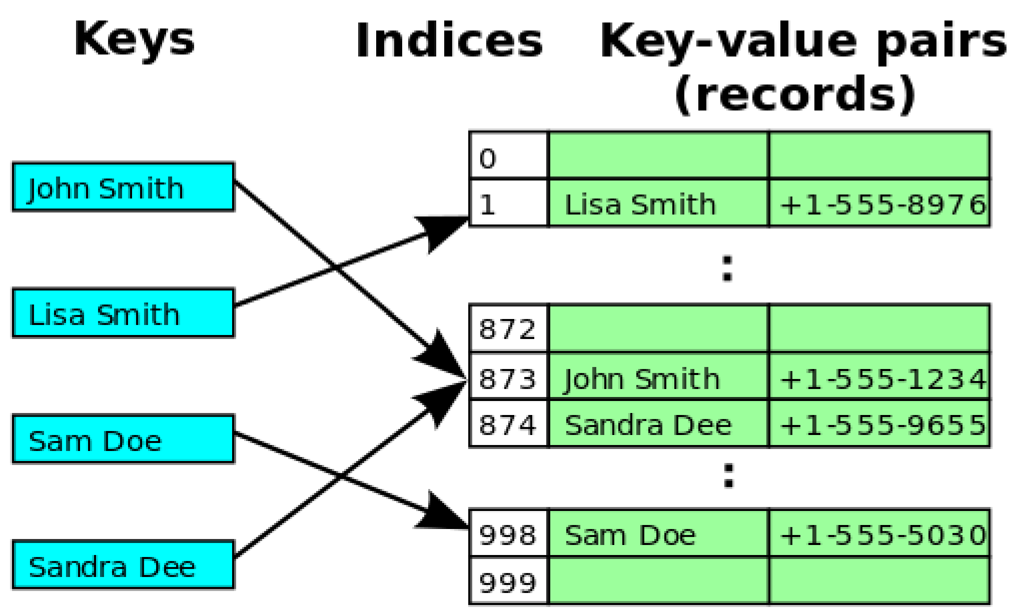
Table of Contents

The Purpose of the Project

- ## Goal of the Project

  The goal of this project is to create a Hashing table. This project implements the open-addressing method of solving collisions, furthermore we are going to succeed in that by using the submethod linear-probing.

- ## Example of Open-addressing and Linear-probing



Implementation

```java
public class OpenAddressingHashTable<K, V> implements Dictionary<K, V>
```

The OpenAddressingHashTable class is a **generic** class that implements the given Dictionary class from the assignment document.

Fields

```java
    private static final int DEFAULT_CAPACITY = 64;

    private Entry<K, V>[] table;
    private int           size;
    private Byte[][]      hashingTable;
    private int           b;
```

The fields and their purpose is as follows:

1. `private static final int DEFAULT_CAPACITY = 64;`
   DEFAULT_CAPACITY is a constant and it's used only to initialize the HashTable if no size is given or if the input given is not a power of 2.
2. `private Entry<K, V>[] table;`
   The table field is used to save instances of Entry objects into the HashTable. It is initialized with the constructor at either a given number that is a power of 2 or at DEFAULT_SIZE this array is also pseudo-dynamic, meaning that if it runs out of space for elements, it doubles in size. Also if the array gets too small, specifically ¼ of its capacity, it gets halved in size.

3. `private int size;`
   Size is equal to the amount of not null elements in the HashTable.

4. `private Byte[][] hashingTable;`
   The hashingTable is a 2d array of only 0's and 1's, it is assigned an array with the createNewHashingTable function and it is used in the creation of individual hash codes for every entry using the matrix method.

5. `private int b;`

   b is the amount of bits required to create hash codes for new entries.

   According to the lectures and hashing theory $2^b = m$ *, where* ***m*** *equals* *table.length* *, therefore we can find b by doing =>* $b = \log_2 m$.

   This field is used to specify one of the dimensions of the hashingTable and therefore when we require a new hashingTable, because the table length has been halved or doubled, b also needs to be updated.

Constructors

```java
    @SuppressWarnings("unused")
    public OpenAddressingHashTable()
    {
        this(DEFAULT_CAPACITY);
    }

    @SuppressWarnings("unchecked")
    public OpenAddressingHashTable(int capacity)
    {
        int n = isPower(capacity) && capacity != 0 ?
                capacity :
                (int) Math.pow(2 , Math.ceil(Math.log(capacity) / Math.log(2)));

        table        = (Entry<K, V>[]) Array.newInstance(Entry.class , n);
        size         = 0;
        b            = (int) (Math.log(table.length) / Math.log(2));
        hashingTable = createNewHashingTable();

    }
```

There are two constructors that are being used:

1. The first constructor is being called when the user doesn't specify an integer for the length of the array. It calls the second constructor with the DEFAULT_CAPACITY as the size of the HashTable.
2. The second constructor accepts an integer as a parameter to be used as the capacity of the HashTable. It is responsible for initializing the fields. If capacity is zero or if it is not a power of two then it calculates a capacity that is a power of 2 by finding the $\log_2$(capacity), getting the smallest integer that is greater than or equal from $\log_2$(capacity) and raising that to the power of 2 to get the closest capacity that is a power of 2, or else it uses the parameter given as the capacity of the table. It also creates an object array of length n and casts it as an array of Entry<K,V>. Size is set to 0 , as the Hashtable is empty , and b is calculated and assigned as explained here. We also initialize the hashingTable.

Methods

```java
public static boolean isPower(int n)
{
    double x = Math.log(n) / Math.log(2);
    return (int) (Math.ceil(x)) == (int) (Math.floor(x));
}
```

1. `public static boolean isPower(int n)`

   isPower is a method that returns true if the number given is a power of 2 and false if not. It is used to check whether the size given in the constructor is a valid number. We succeed this by first finding the $\log_2 n$ using this mathematical formula

$$\log_b(a) = \frac{\log_x(a)}{\log_x(b)}$$

   [formula explanation]

   Math.log is a function that implements log with base e, that’s why we need to change the base to 2. Then we check that the ceiling and the floor of the number are the same , meaning that it is an integer value , therefore it is a power of 2.

```java
@Override
@TestedAndFunctional
public void put(K key , V value
{
    rehashIfNecessary();
    put_us(key , value);
}
```

2. `public void put(K key , V value)`
   Put is a method that calls rehashIfNecessary method to re-hash the hash table if it's needed and afterwards calls put_us to insert a new element. The reason behind the existence of put & put_us is that put_us is used in rehashIfNecessary, so if we had everything in one method for example put, then when the re-hashing was going to happen an endless loop would be created.

```java
@TestedAndFunctional
private void put_us(K key , V value)
{
    int index = hash(key);

    while(table[index] != null) {
        if(table[index].getKey().equals(key)) {
            table[index].setValue(value);
            return;
        }
        index = (index + 1) % table.length;
    }


    table[index] = new Entry<>(key, value);
    size++;
}
```

3. `private void put_us(K key , V value)`

   Put_us (stands for Put_unsafe) is a method that is responsible for inserting a new element in the hash table. First we acquire the position to save the new element, by calling the method hash. Then we check if an element with the same key already exists in the hash table so we can update the value of it to the new one. This is being achieved with a while loop that stops when it finds a null element. If an element with the same key was not found in the hash table then we simply save the new element at the position that the null element was found and increase the size of the contents of the hash table by one.

```java
    @Override
    @TestedAndFunctional
    public V remove(K key)
    {
        rehashIfNecessary();
        if (! this.contains(key))
        {
            throw new NoSuchElementException();
        }

        int index = hash(key);

        while(!table[index].getKey().equals(key)) {
            index = (index +1) % table.length;
        }

        V returnValue = table[index].getValue();

        table[index] = null;

        int j = (index + 1) % table.length;

        while(table[j] != null) {
            int pos = hash(table[j].getKey());

            if(pos <= index) {
                Entry<K, V> temp = table[j];
                table[j] = table[index];
                table[index] = temp;
                index = j;
            }

            j = (j + 1) % table.length;
        }

        size--;
        return returnValue;
    }
```

4. `public V remove(K key)`

   Remove is a method responsible for removing an element from the hash table with the key given from the parameter. Firstly it calls the rehashIfNecessary method to rehash the hash table, if it's needed, to a smaller length. Afterwards it calls the contains method that checks if an element with the specified key exists inside of the hash table. If an element isn't found then it throws an exception for the user to handle. Then we acquire the position of the element by calling the hash method and save it to **index**. Since we use linear probing all we have to do to get the actual position of the element is to move the index to the right till we have an element with the same key as the key of the element we want to remove. Then we save the value of the element so we can return it at the end of the function. We set the element at the position that we found earlier to null which means the item was deleted and a new index is created called **j**, so we can fix any problems that were created by deleting the element. If the element table[j] is null then we don't do anything. If not then we get the position of the table[j] element's key using hash. If that position is lower or equal to the position of the element that was removed then we swap the values of table[j] and table[index] and we set the value of **index** to be **j**. The last part is essential because we want all the elements at the right of the removed element until we find a null element. For that purpose a while loop is being used that exits when a null element is found. Lastly we reduce the size of the contents of the table by 1 and return the removed element's value.

```java
@Override
@TestedAndFunctional
public V get(K key) throws NoSuchElementException
{

    for (Dictionary.Entry<K, V> item : this)
    {
        if (item.getKey().equals(key))
        {
            return item.getValue();
        }
    }

    throw new NoSuchElementException();
}
```

5. `public V get(K key) throws NoSuchElementException`

   Get is a method that returns the value of an element given a specific key if found. If not found then it throws an exception that the user must handle. In order to implement this method a custom iterator was made. The iterator loops the table and if it finds a key that is the same as the specified key, it returns the value stored in the hash table.

```java
@Override
@TestedAndFunctional
public boolean contains(K key)
{
    try
    {
        get(key);
        return true;
    } catch (Exception e)
    {
        return false;
    }
}
```

6. `public boolean contains(K key)`
   Contains is a method that returns true or false, if an element with the specified key exists or not. For that purpose , get method is being used. If the operation of get was successful, true is being returned but if an exception is thrown then false is being returned.

```java
@Override
@TestedAndFunctional
public boolean isEmpty()
{
    return size == 0;
}
```

7. `public boolean isEmpty()`
   The isEmpty method checks whether size is equal to 0 to determine if the hash table has any elements or not.

```java
@Override
@TestedAndFunctional
public int size()
{
    return size;
}
```

8. `public int size()`

   Size is a method that returns the [size](#) of the table. Practically the number of non null elements in the table.

```java
@Override
@TestedAndFunctional
public void clear()
{
    Arrays.fill(table , val: null);
    size = 0;
}
```

9. `public void clear()`

   Clear is a method that is responsible for clearing the hash table from all the elements. It fills the table with null elements and sets the [size](#) to 0.

```java
@Override
@TestedAndFunctional
public Iterator<Dictionary.Entry<K, V>> iterator()
{
    return new HashIterator();
}
```

10. `public Iterator<Dictionary.Entry<K, V>> iterator()`

    Iterator is a method that is responsible for the creation of a custom iterator from the inner class HashIterator for the hash table.

```java
@TestedAndFunctional
public int hash(K key)
{
    String temp = Integer.toBinaryString(key.hashCode());
    String keyHashBits_s = String.format("%32s" , temp).replace( oldChar: ' ' ,  newChar: '0');
    Byte[] keyHashBits = new Byte[32];
    for (int i = 0; i < keyHashBits.length; i++)
    {
        keyHashBits[i] = Byte.parseByte(Character.toString(keyHashBits_s.charAt(i)));
    }
    String returnValue_s = "";

    //noinspection ForLoopReplaceableByForEach
    for (int i = 0; i < this.hashingTable.length; i++)
    {
        int sum = 0;
        for (int j = 0; j < this.hashingTable[i].length; j++)
        {
            sum += this.hashingTable[i][j] * keyHashBits[j];
        }
        //noinspection StringConcatenationInLoop
        returnValue_s += sum % 2;
    }

    return Integer.parseInt(returnValue_s ,  radix: 2);
}
```

11. `public int hash(K key)`

Hash is a method that is responsible for finding the position of an element in the hash table. Firstly we get the hashCode of the key given, we do this because this will always return an Integer which has a fixed bit length. Then we convert that to binary and then to a string, we convert it to a string to fill in all the leading 0 of the binary form so we can perform accurate bitwise operations upon that number. Then we create a Byte array with size of 32 to save the individual bit of the string(which as said before is the binary representation of the key's hashCode). {We are using the Byte class because it is the smallest number type in java}. That is being accomplished with the use of a for loop, and transforming each character to a Byte and adding it to its corresponding position in the Byte array. Then we do matrix multiplication with the Byte array and the hashingTable that is filled with random numbers. and we append the result from each row multiplication to a string, thus constructing a string that is the binary representation of the key's hash in our hash table. Lastly we parse the string as a signed integer in base 2, since the string has the form of a binary number, and we return that integer.

```java
@TestedAndFunctional
private void rehashIfNecessary()
{
    int newSize;
    if (size >= table.length)
    {
        newSize = table.length * 2;
    } else if (size <= table.length / 4 && table.length > 2)
    {
        newSize = table.length / 2;
    } else
    {
        return;
    }
    OpenAddressingHashTable<K, V> newTable = new OpenAddressingHashTable<>(newSize);

    for (Dictionary.Entry<K, V> item : this)
    {
        newTable.put_us(item.getKey() , item.getValue());
    }

    this.b            = newTable.b;
    this.hashingTable = newTable.hashingTable;
    this.table        = newTable.table;
    this.size         = newTable.size;
}
```

12. `private void rehashIfNecessary()`

rehashIfNecessary is a method responsible for increasing and decreasing the size of the hash table when needed. If the size is bigger or equal to the length of the hash table then an increase is needed and the new size is set to double its old length. If the size is less or equal to ¼ of the length of the hash table and the length of the hash table is at least bigger than 2 then a decrease is needed and the new size is set to half of the old length. If none of the above criterias are met then a rehash is not needed. If a rehash is needed, then a new hash table table is created with the new size and the elements from the old table are inserted into the new one with the method put_us , we use the method put_us because we know that a rehash won't be needed in that new hash table. Lastly we change the old b, hashing table, table and size with the new ones.

```java
    private Byte[][] createNewHashingTable()
    {
        Byte[][] hashingTable = new Byte[b][32];
        for (int i = 0; i < hashingTable.length; i++)
        {
            for (int j = 0; j < hashingTable[i].length; j++)
            {
                hashingTable[i][j] = (byte) (Math.random() * 2);
            }
        }
        return hashingTable;

    }
```

13. `private Byte[][] createNewHashingTable()`

createNewHashingTable is responsible for the creation of a new hashingTable. Firstly creates a new Byte array with b rows and 32 columns. Then it fills the array with 0 and 1. The process of selecting 0 and 1 when inserting to the array is random. Lastly it returns the created Byte array filled with random 0 and 1s.

Test Methods

```java
//region Test Methods
public int getLength()
{
    return table.length;
}
```

1. `public int getLength()`
   getLength is responsible for returning the length of the current hash table.

```java
public int getIndex(K key) {
    int length = getLength();
    for (int i = 0; i < length; i++) {
        if(table[i] != null) {
            if (table[i].getKey().equals(key)) return i;
        }
    }

    return -1;
```

2. `public int getIndex(K key)`
   getIndex is a method responsible for returning the index that the specified key was found in the hash table. If no such element exists then it returns -1.

```java
public String hash_debug(K key)
{
    String temp = Integer.toBinaryString(key.hashCode());
    String keyHashBits_s = String.format("%32s" , temp).replace( oldChar: ' ' ,  newChar: '0');
    Byte[] keyHashBits = new Byte[32];
    for (int i = 0; i < keyHashBits.length; i++)
    {
        keyHashBits[i] = Byte.parseByte(Character.toString(keyHashBits_s.charAt(i)));
    }
    String returnValue_s = "";

    //noinspection ForLoopReplaceableByForEach
    for (int i = 0; i < this.hashingTable.length; i++)
    {
        int sum = 0;
        for (int j = 0; j < this.hashingTable[i].length; j++)
        {
            sum += this.hashingTable[i][j] * keyHashBits[j];
        }
        //noinspection StringConcatenationInLoop
        returnValue_s += sum % 2;
    }

    return returnValue_s;
}
```

3. `public String hash_debug(K key)`

   Hash_debug is a method that does exactly what the hash method does , but instead of returning the hash it returns the binary representation of the hash with a string. it is being used for testing purposes at the third test.

Testing

First Test

```java
@Test
void testBasicFunctionality()
{
    int size = 16;
    OpenAddressingHashTable<Integer, Integer> table = new OpenAddressingHashTable<>(size);
    assertTrue(table.isEmpty());
    assertEquals( expected: 0, table.size());

    for (int i = 0; i < size; i++)
    {
        table.put(i, i + 1);
        table.contains(i);
        assertEquals( expected: i + 1, table.size());
    }

    for (int i = 0; i < size; i++)
    {
        assertEquals( expected: i + 1, table.get(i));
    }
    assertFalse(table.isEmpty());

    for (int i = 0; i < size; i++)
    {
        Integer x = table.remove(i);
        assertEquals( expected: i + 1, x);
        assertEquals( expected: size - i - 1, table.size());
    }
    assertTrue(table.isEmpty());
}
```

Explanation

The first test is responsible for testing the basic functionality of the Hashtable. Some of the methods used are *isEmpty*, *size*, *put*, *contains*, *get* and *remove*. We create a table of `int size = 16;`
Firstly we evaluate if the table isEmpty and if the size of the table is 0. Then a loop is used to insert `int size = 16;` elements into the table. On every iteration we insert one element , with a key of i and value of i + 1 , in the table. We check if the table contains the added element and lastly if the size of the table is correct. Afterwards with the help of another loop, we get the inserted elements to check if they are in the correct order. The last loop removes the elements one by one, check if they are the correct elements and lastly if the size was reduced correctly. Lastly isEmpty is used to prove that the table was emptied.

Second Test

```java
@Test
void testLinearProbing()
{
    int size = 15;
    OpenAddressingHashTable<String, Integer> table = new OpenAddressingHashTable<>(size);

    assertFalse(OpenAddressingHashTable.isPower(size));
    assertTrue(OpenAddressingHashTable.isPower(table.getLength()));

    assertNotEquals( unexpected: "Aa",  actual: "BB");
    assertEquals("Aa".hashCode(), "BB".hashCode());

    table.put("Aa", 1);
    table.put("BB", 2);

    assertEquals(table.hash( key: "Aa"), table.hash( key: "BB"));
    assertEquals( expected: 2, table.size());

    int indexAa = table.getIndex( key: "Aa") , indexBB = table.getIndex( key: "BB");

    assertEquals( expected: (indexAa + 1) % table.getLength(), indexBB);

}
```

Explanation

The second test is responsible for proving that the Hashtable uses linear probing.It starts by initializing a table with `int size = 15;`
Then we use a public static method that evaluates if `int size = 15;` is a power of 2 or not. We do the same for the length of the table. The purpose behind this activity is to show that it will accept any int size and if it is not a power of 2 it will find the next nearest number that is a power of 2. The next 2 assertions prove that those two specific strings are different but have the same hashCode. Then these 2 elements are inserted into the table and it is shown that both of them return the same hash. Afterwards we find the positions of the 2 elements and we show that the position of the “BB” element is positioned after the “Aa” element because they have the same hashcode but they collide, therefore we use linear probing to resolve that collision.

Third Test

```java
@Test
void testRehashing()
{
    int size = 16;
    OpenAddressingHashTable<Integer, Integer> table = new OpenAddressingHashTable<>(size);

    for (int i = 0; i < size; i++)
    {
        table.put(i, i + 1);
    }

    int[] hashCodesBinaryRepresentationStringLength = new int[size];
    for (int i = 0; i < size; i++)
    {
        hashCodesBinaryRepresentationStringLength[i] = table.hash_debug(i).length();
    }

    table.put(size, size + 1);
    for (int i = 0; i < size; i++)
    {
        assertEquals( expected: hashCodesBinaryRepresentationStringLength[i] + 1, table.hash_debug(i).length());
    }

    assertEquals(table.getLength(), actual: 2 * size);

    table.clear();

    assertTrue(table.isEmpty());
    assertEquals( expected: 0, table.size());

}
```

Explanation

The third test is responsible for testing the functionality of rehashing the table whenever it is full or when it doesn't contain a specific number of elements. It starts by pushing `int size = 16;` amount of elements inside the table. For the purpose of this test a public method was created that calculates the hashcode of a key. Practically a duplicate method of the one used internally. We can determine if the rehash was successful if we count the number of bits of the hashed key before the table has been rehashed and after it has been rehashed to find that the amount of bits is different. Then we added a new element that leads to a rehash. All we have to do now is check whether the length of the hash code of the new table's elements is bigger, than the lengths saved on the array, by 1.

For example if we insert 16 elements into the table, the length of the table will also be $16 = 2^4$ which means that the length of the hash will be $\log_2(2^4) = 4$. So if we add another element the length of the table becomes: ($32 = 2^5$) and the length of the hash: ($\log_2(2^5) = 5$).