

数据结构与算法 15- 哈希表

笔记本: 我的笔记

创建时间: 2020/10/11 11:08

更新时间: 2020/10/11 11:24

作者: liuhouer

标签: 算法

1> 哈希表本质是数组，按照索引可以O(1)找到相应的元素，所以哈希操作就是一个把原来元素转化为一个索引的操作

什么是哈希表

int[26] freq 就是一个哈希表!

每一个字符都和一个索引相对应

a → 0
b → 1
c → 2
.....
z → 25

index = ch - 'a'

哈希函数

f(ch) = ch - 'a'

O(1)的查找操作!

哈希函数 “键”转换为“索引”

f(ch) = ch - 'a'

一个班的学生学号: 1-30

身份证号 110108198512166666

字符串

浮点数

日期

哈希表

在哈希表上操作

解决哈希冲突

很难保证每一个“键”

通过哈希函数的转换

对应不同的“索引”

哈希函数的设计

大整数

一个简单的解决办法: 模一个素数

背后的数学理论超出课程范畴

10 % 4 → 2
20 % 4 → 0
30 % 4 → 2
40 % 4 → 0
50 % 4 → 2

10 % 7 → 3
20 % 7 → 6
30 % 7 → 2
40 % 7 → 4
50 % 7 → 1

字符串 转成整型处理

hash(code) = (c * B^3 + o * B^2 + d * B^1 + e * B^0) % M

hash(code) = (((((c * B) + o) * B + d) * B + e) % M

hash(code) = (((((c * M) * B + o) % M * B + d) % M * B + e) % M

哈希函数的设计

哈希冲突的处理 链地址法

0 → TreeMap
1 → TreeMap
2 → TreeMap
3 → TreeMap
4 → TreeMap
5 → TreeMap
... → TreeMap
M-1 → TreeMap

HashMap 就是一个 TreeMap 数组

HashSet 就是一个 TreeSet 数组

Java8之前, 每一个位置对应一个链表

Java8开始, 当哈希冲突达到一定程度

每一个位置从链表转成红黑树

哈希表 链地址法

0
1
2
3
4
5
...
M-1

总共有M个地址

如果放入哈希表的元素为N

如果每个地址是链表: O(N/M)

如果每个地址是平衡树: O(log(N/M))

哈希表的复杂度分析

对于哈希表来说, 元素数从 N 增加到 upperTol * N; 地址空间倍增

平均复杂度 O(1)

每个操作在O(lowerTol) ~ O(upperTol) → O(1)

细节同理

哈希表

哈希表: 均摊复杂度为O(1)

牺牲了什么? 顺序性

集合, 映射

有序集合, 有序映射

平衡树

无序集合, 无序映射

哈希表

开放地址法

0
1
2
3
4
5
...
M-1

平方探测

遇到哈希冲突

+1 +4 +9 +16

hash(x) = x % 10

2.实现Hash表【HashMap】

```
import java.util.Map;  
import java.util.TreeMap;  
  
public class HashTable<K, V> {  
  
    private static final int upperTol = 10;
```

```

private static final int lowerTol = 2;
private static final int initCapacity = 7;

private TreeMap<K, V>[] hashtable;
private int size;
private int M;

public HashTable(int M){
    this.M = M;
    size = 0;
    hashtable = new TreeMap[M];
    for(int i = 0 ; i < M ; i ++){
        hashtable[i] = new TreeMap<>();
    }

    public HashTable(){
        this(initCapacity);
    }

    private int hash(K key){
        //key.hashCode() & 0x7fffffff 消除正数位的符号的写法【绝对值】
        return (key.hashCode() & 0x7fffffff) % M;
    }

    public int getSize(){
        return size;
    }

    public void add(K key, V value){
        TreeMap<K, V> map = hashtable[hash(key)];
        if(map.containsKey(key))
            map.put(key, value);
        else{
            map.put(key, value);
            size ++;

            if(size >= upperTol * M)
                resize(2 * M);
        }
    }

    public V remove(K key){
        V ret = null;
        TreeMap<K, V> map = hashtable[hash(key)];
        if(map.containsKey(key)){
            ret = map.remove(key);
            size --;

            if(size < lowerTol * M && M / 2 >= initCapacity)
                resize(M / 2);
        }
        return ret;
    }

    public void set(K key, V value){
        TreeMap<K, V> map = hashtable[hash(key)];
        if(!map.containsKey(key))
            throw new IllegalArgumentException(key + " doesn't exist!");

        map.put(key, value);
    }

    public boolean contains(K key){

```

```

        return hashtable[hash(key)].containsKey(key);
    }

    public V get(K key){
        return hashtable[hash(key)].get(key);
    }

    private void resize(int newM){
        TreeMap<K, V>[] newHashTable = new TreeMap[newM];
        for(int i = 0 ; i < newM ; i ++){
            newHashTable[i] = new TreeMap<>();

            int oldM = M;
            this.M = newM;
            for(int i = 0 ; i < oldM ; i ++){
                TreeMap<K, V> map = hashtable[i];
                for(K key: map.keySet())
                    newHashTable[hash(key)].put(key, map.get(key));
            }

            this.hashtable = newHashTable;
        }
    }
}

```