

Java数据结构和算法（十三）——哈希表

目录

- 1、哈希函数的引入
- 2、冲突
- 3、开放地址法
 - ①、线性探测
 - ②、装填因子
 - ③、二次探测
 - ④、再哈希法
- 4、链地址法
- 5、桶
- 6、总结

Hash表也称散列表，也有直接译作哈希表，Hash表是一种根据关键字值（key - value）而直接进行访问的数据结构。它基于数组，通过把关键字映射到数组的某个下标来加快查找速度，但是又和数组、链表、树等数据结构不同，在这些数据结构中查找某个关键字，通常要遍历整个数据结构，也就是O(N)的时间级，但是对于哈希表来说，只是O(1)的时间级。

注意，这里有个重要的问题就是如何把关键字转换为数组的下标，这个转换的函数称为哈希函数（也称散列函数），转换的过程称为哈希化。

[回到顶部](#)

1、哈希函数的引入

大家都用过字典，字典的优点是我们可以通过前面的目录快速定位到所要查找的单词。如果我们想把一本英文字典的每个单词，从 a 到 zzzzyva(这是牛津字典的最后一个单词)，都写入计算机内存，以便快速读写，那么哈希表是个不错的选择。

这里我们将范围缩小点，比如想在内存中存储5000个英文单词。我们可能想到每个单词会占用一个数组单元，那么数组的大小是5000，同时可以用数组下标存取单词，这样设想很完美，但是数组下标和单词怎么建立联系呢？

首先我们要建立单词和数字（数组下标）的关系：

我们知道 ASCII 是一种编码，其中 a 表示97，b表示98，以此类推，一直到122表示z，而每个单词都是由这26个字母组成，我们可以不用 ASCII 编码那么大的数字，自己设计一套类似ASCII的编码，比如a表示1，b表示2，依次类推，z表示26，那么表示方法我们就知道了。

接下来如何把单个字母的数字组合成代表整个单词的数字呢？

①、把数字相加

首先第一种简单的方法就是把单词的每个字母表示的数字相加，得到的和便是数组的下标。

比如单词 cats 转换成数字：

cats = 3 + 1 + 20 + 19 = 43

那么单词 cats 存储在数组中的下标为43，所有的英文单词都可以用这个办法转换成数组下标。但是这个办法真的可行吗？

昵称：YSOcean
园龄：2年
粉丝：2067
关注：13
[+加关注](#)

<	2019年3月							>
	日	一	二	三	四	五	六	
24	25	26	27	28	1	2		
3	4	5	6	7	8	9		
10	11	12	13	14	15	16		
17	18	19	20	21	22	23		
24	25	26	27	28	29	30		
31	1	2	3	4	5	6		

我的标签

- Linux系列教程(25)
- 深入理解计算机系统(24)
- Java数据结构和算法(16)
- MyBatis详解系列(11)
- JDK源码解析(11)
- Maven系列教程(8)
- Redis详解(8)
- Spring入门系列(8)
- Java IO详解系列(7)
- Java高并发设计(7)
- 更多

随笔分类

- Java SE(22)
- JavaWeb(34)
- Java高并发
- Java关键字(6)
- Java数据结构和算法(15)

假设我们约定一个单词最多有 10 个字母，那么字典的最后一个单词为 zzzzzzzzzz，其转换为数字：

$zzzzzzzzzz = 26 \times 10 = 260$

那么我们可以得到单词编码的范围是从1-260。很显然，这个范围是不够存储5000个单词的，那么肯定有一个位置存储了多个单词，每个数组的数据项平均要存储192个单词（5000除以260）。

对于上面的问题，我们如何解决呢？

第一种方法：考虑每个数组项包含一个子数组或者一个子链表，这个办法存数据项确实很快，但是如果我们想要从192个单词中查找到其中一个，那么还是很慢。

第二种方法：为啥要让那么多单词占据同一个数据项呢？也就是说我们没有把单词分的足够开，数组能表示的元素太少，我们需要扩展数组的下标，使其每个位置都只存放一个单词。

对于上面的第二种方法，问题产生了，我们如何扩展数组的下标呢？

②、幂的连乘

我们将单词表示的数拆成数列，用适当的 27 的幂乘以这些位数（因为有26个可能的字符，以及空格，一共27个），然后把乘积相加，这样就得出了每个单词独一无二的数字。

比如把单词cats 转换为数字：

$cats = 3 \times 27^3 + 1 \times 27^2 + 20 \times 27^1 + 19 \times 27^0 = 59049 + 729 + 540 + 19 = 60337$

这个过程会为每个单词创建一个独一无二的数，但是注意的是我们这里只是计算了 4 个字母组成的单词，如果单词很长，比如最长的10个字母的单词 zzzzzzzzzz，仅仅是 27^9 结果就超出了7000000000000，这个结果是很巨大的，在实际内存中，根本不可能为一个数组分配这么大的空间。

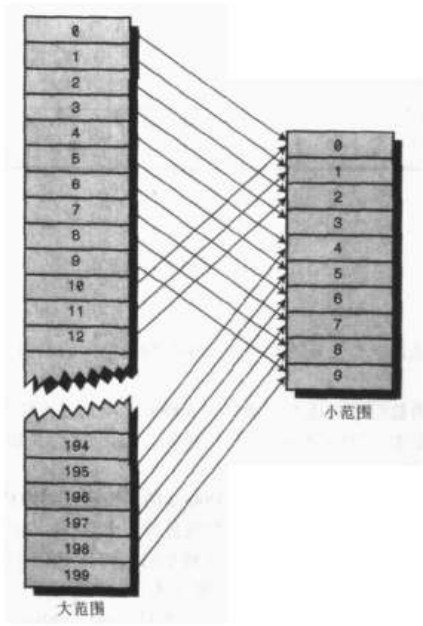
所以这个方案的问题就是虽然为每个单词都分配了独一无二的下标，但是只有一小部分存放了单词，很大一部分都是空着的。那么现在就需要一种方法，把数位幂的连乘系统中得到的巨大的整数范围压缩到可接受的数组范围中。

对于英语字典，假设只有5000个单词，这里我们选定容量为10000 的数组空间来存放（后面会介绍为啥需要多出一倍的空间）。那么我们就需要将 0 到超过 7000000000000 的范围，压缩到从0到10000的范围。

第一种方法：取余，得到一个数被另一个整数除后的余数。首先我们假设要把从0-199的数字（用largeNumber表示），压缩为从0-9的数字（用smallNumber表示），后者有10个数，所以变量smallRange 的值为10，这个转换的表达式为：

$smallNumber = largeNumber \% smallRange$

当一个数被 10 整除时，余数一定在0-9之间，这样，我们就把从0-199的数压缩为从0-9的数，压缩率为 20 :1。



我们也可以用类似的方法把表示单词唯一的数压缩成数组的下标：

Java虚拟机
JDK源码解析(11)
Linux(9)
Linux详解(24)
Nginx详解(4)
Redis详解(8)
TCP/IP协议
编程小技巧(1)
查找算法(1)
大数据(3)
工具使用(15)
计算机系统与结构(24)
计算机组成与系统结构
浪潮之巅(1)
排序算法
前端(5)
日常工作问题(7)
设计模式(1)
算法分析(1)
消息中间件(6)
邮件服务(4)
积分与排名
积分 - 441370
排名 - 415
阅读排行榜
1. Tomcat 部署项目的三种方法(152987)
2. Java 集合详解(74896)
3. Java数据结构和算法（一）——简介(62792)
4. MyBatis 详解（一对一，一对多，多对多）(62095)

arrayIndex = largerNumber % smallRange

这也就是哈希函数。它把一个较大范围的数字哈希（转化）成一个小范围的数字，这个小范围的数对应着数组的下标。使用哈希函数向数组插入数据后，这个数组就是哈希表。

[回到顶部](#)

2、冲突

把巨大的数字范围压缩到较小的数字范围，那么肯定会有几个不同的单词哈希化到同一个数组下标，即产生了**冲突**。

冲突可能会导致哈希化方案无法实施，前面我们说指定的数组范围大小是实际存储数据的两倍，因此可能有一半的空间是空着的，所以，当冲突产生时，一个方法是通过系统的方法找到数组的一个空位，并把这个单词填入，而不再用哈希函数得到数组的下标，这种方法称为**开放地址法**。比如加入单词 cats 哈希化的结果为5421，但是它的位置已经被单词parsnip占用了，那么我们会考虑将单词 cats 存放在parsnip后面的一个位置 5422 上。

另一种方法，前面我们也提到过，就是数组的每个数据项都创建一个子链表或子数组，那么数组内不直接存放单词，当产生冲突时，新的数据项直接存放到这个数组下标表示的链表中，这种方法称为**链地址法**。

[回到顶部](#)

3、开放地址法

开发地址法中，若数据项不能直接存放在由哈希函数所计算出来的数组下标时，就要寻找其他的位置。分别有三种方法：线性探测、二次探测以及再哈希法。

①、线性探测

在线性探测中，它会线性的查找空白单元。比如如果 5421 是要插入数据的位置，但是它已经被占用了，那么就使用5422，如果5422也被占用了，那么使用5423，以此类推，数组下标依次递增，直到找到空白的位置。这就叫做线性探测，因为它沿着数组下标一步一步顺序的查找空白单元。

完整代码：

```
1 package com.js.hash;
2
3 public class MyHashTable {
4     private DataItem[] hashArray; //DataItem类，表示每个数据项信息
5     private int arraySize;//数组的初始大小
6     private int itemNum;//数组实际存储了多少项数据
7     private DataItem nonItem;//用于删除数据项
8
9     public MyHashTable(int arraySize){
10         this.arraySize = arraySize;
11         hashArray = new DataItem[arraySize];
12         nonItem = new DataItem(-1);//删除的数据项下标为-1
13     }
14     //判断数组是否存储满了
15     public boolean isFull(){
16         return (itemNum == arraySize);
17     }
18
19     //判断数组是否为空
20     public boolean isEmpty(){
21         return (itemNum == 0);
22     }
23
24     //打印数组内容
25     public void display(){
26         System.out.println("Table:");
27         for(int j = 0 ; j < arraySize ; j++){
28             if(hashArray[j] != null){
29                 System.out.print(hashArray[j].getKey() + " ");
30             }else{
```

5. Java数据结构和算法（七）
——链表(46203)

评论排行榜

1. 深入理解计算机系统（1.1）-----Hello World 是如何运行的(27)

2. SpringMVC详解（四）-----SSM三大框架整合之登录功能实现(23)

3. 深入理解计算机系统（序章）-----谈程序员为什么要懂底层计算机结构(20)

4. Tomcat 部署项目的三种方法(18)

5. Java数据结构和算法（十）
——二叉树(18)

```
31         System.out.print("** ");
32     }
33 }
34 }
35 //通过哈希函数转换得到数组下标
36 public int hashFunction(int key){
37     return key%arraySize;
38 }
39
40 //插入数据项
41 public void insert(DataItem item){
42     if(isFull()){
43         //扩展哈希表
44         System.out.println("哈希表已满，重新哈希化...");
45         extendHashTable();
46     }
47     int key = item.getKey();
48     int hashVal = hashFunction(key);
49     while(hashArray[hashVal] != null && hashArray[hashVal].getKey() != -1){
50         ++hashVal;
51         hashVal %= arraySize;
52     }
53     hashArray[hashVal] = item;
54     itemNum++;
55 }
56 /**
57  * 数组有固定的大小，而且不能扩展，所以扩展哈希表只能另外创建一个更大的数组，然后把旧数组中的
58  * 但是哈希表是根据数组大小计算给定数据的位置的，所以这些数据项不能再放在新数组中和老数组相同
59  * 因此不能直接拷贝，需要按顺序遍历老数组，并使用insert方法向新数组中插入每个数据项。
60  * 这个过程叫做重新哈希化。这是一个耗时的过程，但如果数组要进行扩展，这个过程是必须的。
61  */
62 public void extendHashTable(){
63     int num = arraySize;
64     itemNum = 0; //重新计数，因为下面要把原来的数据转移到新的扩张的数组中
65     arraySize *= 2; //数组大小翻倍
66     DataItem[] oldHashArray = hashArray;
67     hashArray = new DataItem[arraySize];
68     for(int i = 0 ; i < num ; i++){
69         insert(oldHashArray[i]);
70     }
71 }
72
73 //删除数据项
74 public DataItem delete(int key){
75     if(isEmpty()){
76         System.out.println("Hash Table is Empty!");
77         return null;
78     }
79     int hashVal = hashFunction(key);
80     while(hashArray[hashVal] != null){
81         if(hashArray[hashVal].getKey() == key){
82             DataItem temp = hashArray[hashVal];
83             hashArray[hashVal] = nonItem; //nonItem表示空Item,其key为-1
84             itemNum--;
85             return temp;
86         }
87         ++hashVal;
88         hashVal %= arraySize;
89     }
90     return null;
91 }
92
93 //查找数据项
94 public DataItem find(int key){
95     int hashVal = hashFunction(key);
96     while(hashArray[hashVal] != null){
97         if(hashArray[hashVal].getKey() == key){
```

```
98         return hashArray[hashVal];
99     }
100     ++hashVal;
101     hashVal %= arraySize;
102 }
103 return null;
104 }
105
106 public static class DataItem{
107     private int iData;
108     public DataItem(int iData){
109         this.iData = iData;
110     }
111     public int getKey(){
112         return iData;
113     }
114 }
115
116 }
```

需要注意的是，当哈希表变得太满时，我们需要扩展数组，但是需要注意的是，数据项不能放到新数组中和老数组相同的位置，而是要根据数组大小重新计算插入位置。这是一个比较耗时的过程，所以一般我们要确定数据的范围，给定好数组的大小，而不再扩容。

另外，当哈希表变得比较满时，我们每插入一个新的数据，都要频繁的探测插入位置，因为可能很多位置都被前面插入的数据所占用了，这称为聚集。数组填的越满，聚集越可能发生。

这就像人群，当某个人在商场晕倒时，人群就会慢慢聚集。最初的人群聚过来是因为看到了那个倒下的人，而后面聚过来的人是因为它们想知道这些人聚在一起看什么。人群聚集的越大，吸引的人就会越多。

②、装填因子

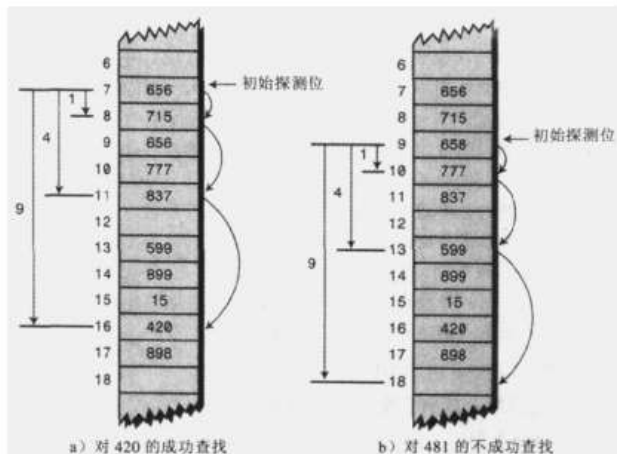
已填入哈希表的数据项和表长的比率叫做装填因子，比如有10000个单元的哈希表填入了6667个数据后，其装填因子为 $2/3$ 。当装填因子不太大时，聚集分布的比较连贯，而装填因子比较大时，则聚集发生的很大了。

我们知道线性探测是一步一步的往后面探测，当装填因子比较大时，会频繁的产生聚集，那么如果我们探测比较大的单元，而不是一步一步的探测呢，这就是下面要讲的二次探测。

③、二次探测

二次探测是防止聚集产生的一种方式，思想是探测相距较远的单元，而不是和原始位置相邻的单元。

线性探测中，如果哈希函数计算的原始下标是 x ，线性探测就是 $x+1, x+2, x+3$ ，以此类推；而在二次探测中，探测的过程是 $x+1, x+4, x+9, x+16$ ，以此类推，到原始位置的距离是步数的平方。二次探测虽然消除了原始的聚集问题，但是产生了另一种更细的聚集问题，叫二次聚集：比如讲184, 302, 420和544依次插入表中，它们的映射都是7，那么302需要以1为步长探测，420需要以4为步长探测，544需要以9为步长探测。只要有一项其关键字映射到7，就需要更长步长的探测，这个现象叫做二次聚集。二次聚集不是一个严重的问题，但是二次探测不会经常使用，因为还有好的解决方法，比如再哈希法。



④、再哈希法

为了消除原始聚集和二次聚集，我们使用另外一种方法：再哈希法。

我们知道二次聚集的原因是，二测探测的算法产生的探测序列步长总是固定的：1,4, 9,16 以此类推。那么我们想到的是需要产生一种依赖关键字的探测序列，而不是每个关键字都一样，那么，不同的关键字即使映射到相同的数组下标，也可以使用不同的探测序列。

方法是把关键字用不同的哈希函数再做一遍哈希化，用这个结果作为步长。对于指定的关键字，步长在整个探测中是不变的，不过不同的关键字使用不同的步长。

第二个哈希函数必须具备如下特点：

一、和第一个哈希函数不同

二、不能输出0（否则，将没有步长，每次探测都是原地踏步，算法将陷入死循环）。

专家们已经发现下面形式的哈希函数工作的非常好： $stepSize = constant - key \% constant$ ；其中constant是质数，且小于数组容量。

再哈希法要求表的容量是一个质数，假如表长度为15(0-14)，非质数，有一个特定关键字映射到0，步长为5，则探测序列是0,5,10,0,5,10,以此类推一直循环下去。算法只尝试这三个单元，所以不可能找到某些空白单元，最终算法导致崩溃。如果数组容量为13, 质数，探测序列最终会访问所有单元。即0,5,10,2,7,12,4,9,1,6,11,3,一直下去，只要表中有一个空位，就可以探测到它。

完整再哈希法代码：

```
1 package com.js.hash;
2
3 public class HashDouble {
4     private DataItem[] hashArray;    //DataItem类，表示每个数据项信息
5     private int arraySize;//数组的初始大小
6     private int itemNum;//数组实际存储了多少项数据
7     private DataItem nonItem;//用于删除数据项
8
9     public HashDouble(){
10         this.arraySize = 13;
11         hashArray = new DataItem[arraySize];
12         nonItem = new DataItem(-1);//删除的数据项下标为-1
13     }
14     //判断数组是否存储满了
15     public boolean isFull(){
16         return (itemNum == arraySize);
17     }
18
19     //判断数组是否为空
20     public boolean isEmpty(){
21         return (itemNum == 0);
22     }
23
24     //打印数组内容
25     public void display(){
26         System.out.println("Table:");
27         for(int j = 0 ; j < arraySize ; j++){
28             if(hashArray[j] != null){
29                 System.out.print(hashArray[j].getKey() + " ");
30             }else{
31                 System.out.print("*** ");
32             }
33         }
34     }
35     //通过哈希函数转换得到数组下标
36     public int hashFunction1(int key){
37         return key%arraySize;
38     }
39
40     public int hashFunction2(int key){
41         return 5 - key%5;
```

```
42     }
43
44     //插入数据项
45     public void insert(DataItem item){
46         if(isFull()){
47             //扩展哈希表
48             System.out.println("哈希表已满，重新哈希化...");
49             extendHashTable();
50         }
51         int key = item.getKey();
52         int hashVal = hashFunction1(key);
53         int stepSize = hashFunction2(key); //用第二个哈希函数计算探测步数
54         while(hashArray[hashVal] != null && hashArray[hashVal].getKey() != -1){
55             hashVal += stepSize;
56             hashVal %= arraySize; //以指定的步数向后探测
57         }
58         hashArray[hashVal] = item;
59         itemNum++;
60     }
61
62     /**
63      * 数组有固定的大小，而且不能扩展，所以扩展哈希表只能另外创建一个更大的数组，然后把旧数组中的
64      * 但是哈希表是根据数组大小计算给定数据的位置的，所以这些数据项不能再放在新数组中和老数组相同
65      * 因此不能直接拷贝，需要按顺序遍历老数组，并使用insert方法向新数组中插入每个数据项。
66      * 这个过程叫做重新哈希化。这是一个耗时的过程，但如果数组要进行扩展，这个过程是必须的。
67      */
68     public void extendHashTable(){
69         int num = arraySize;
70         itemNum = 0; //重新计数，因为下面要把原来的数据转移到新的扩张的数组中
71         arraySize *= 2; //数组大小翻倍
72         DataItem[] oldHashArray = hashArray;
73         hashArray = new DataItem[arraySize];
74         for(int i = 0; i < num; i++){
75             insert(oldHashArray[i]);
76         }
77     }
78
79     //删除数据项
80     public DataItem delete(int key){
81         if(isEmpty()){
82             System.out.println("Hash Table is Empty!");
83             return null;
84         }
85         int hashVal = hashFunction1(key);
86         int stepSize = hashFunction2(key);
87         while(hashArray[hashVal] != null){
88             if(hashArray[hashVal].getKey() == key){
89                 DataItem temp = hashArray[hashVal];
90                 hashArray[hashVal] = nonItem; //nonItem表示空Item,其key为-1
91                 itemNum--;
92                 return temp;
93             }
94             hashVal += stepSize;
95             hashVal %= arraySize;
96         }
97         return null;
98     }
99
100     //查找数据项
101     public DataItem find(int key){
102         int hashVal = hashFunction1(key);
103         int stepSize = hashFunction2(key);
104         while(hashArray[hashVal] != null){
105             if(hashArray[hashVal].getKey() == key){
106                 return hashArray[hashVal];
107             }
108             hashVal += stepSize;
```



```

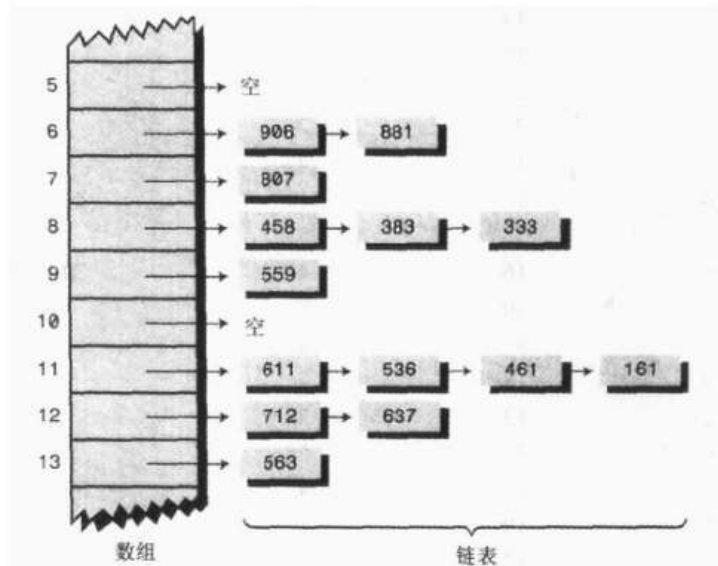
109         hashVal %= arraySize;
110     }
111     return null;
112 }
113 public static class DataItem{
114     private int iData;
115     public DataItem(int iData){
116         this.iData = iData;
117     }
118     public int getKey(){
119         return iData;
120     }
121 }
122 }

```

[回到顶部](#)

4、链地址法

在开放地址法中，通过再哈希法寻找一个空位解决冲突问题，另一个方法是在哈希表每个单元中设置链表（即链地址法），某个数据项的关键字值还是像通常一样映射到哈希表的单元，而数据项本身插入到这个单元的链表中。其他同样映射到这个位置的数据项只需要加到链表中，不需要在原始的数组中寻找空位。



有序链表：

```

1  package com.js.hash;
2
3  public class SortLink {
4      private LinkNode first;
5      public SortLink(){
6          first = null;
7      }
8      public boolean isEmpty(){
9          return (first == null);
10     }
11     public void insert(LinkNode node){
12         int key = node.getKey();
13         LinkNode previous = null;
14         LinkNode current = first;
15         while(current != null && current.getKey() < key){
16             previous = current;
17             current = current.next;
18         }
19         if(previous == null){
20             first = node;
21         }else{

```



```

22         previous.next = node;
23     }
24     node.next = current;
25 }
26 public void delete(int key){
27     LinkNode previous = null;
28     LinkNode current = first;
29     if(isEmpty()){
30         System.out.println("Linked is Empty!!!");
31         return;
32     }
33     while(current != null && current.getKey() != key){
34         previous = current;
35         current = current.next;
36     }
37     if(previous == null){
38         first = first.next;
39     }else{
40         previous.next = current.next;
41     }
42 }
43
44 public LinkNode find(int key){
45     LinkNode current = first;
46     while(current != null && current.getKey() <= key){
47         if(current.getKey() == key){
48             return current;
49         }
50         current = current.next;
51     }
52     return null;
53 }
54
55 public void displayLink(){
56     System.out.println("Link(First->Last)");
57     LinkNode current = first;
58     while(current != null){
59         current.displayLink();
60         current = current.next;
61     }
62     System.out.println("");
63 }
64 class LinkNode{
65     private int iData;
66     public LinkNode next;
67     public LinkNode(int iData){
68         this.iData = iData;
69     }
70     public int getKey(){
71         return iData;
72     }
73     public void displayLink(){
74         System.out.println(iData + " ");
75     }
76 }
77 }

```

链地址法:

```

1 package com.js.hash;
2
3 import com.js.hash.SortLink.LinkNode;
4
5 public class HashChain {
6     private SortLink[] hashArray;//数组中存放链表
7     private int arraySize;
8     public HashChain(int size){

```

```
9         arraySize = size;
10        hashArray = new SortLink[arraySize];
11        //new 出每个空链表初始化数组
12        for(int i = 0 ; i < arraySize ; i++){
13            hashArray[i] = new SortLink();
14        }
15    }
16
17    public void displayTable(){
18        for(int i = 0 ; i < arraySize ; i++){
19            System.out.print(i + ": ");
20            hashArray[i].displayLink();
21        }
22    }
23
24    public int hashFunction(int key){
25        return key%arraySize;
26    }
27
28    public void insert(LinkNode node){
29        int key = node.getKey();
30        int hashVal = hashFunction(key);
31        hashArray[hashVal].insert(node);//直接往链表中添加即可
32    }
33
34    public LinkNode delete(int key){
35        int hashVal = hashFunction(key);
36        LinkNode temp = find(key);
37        hashArray[hashVal].delete(key);//从链表中找到要删除的数据项，直接删除
38        return temp;
39    }
40
41    public LinkNode find(int key){
42        int hashVal = hashFunction(key);
43        LinkNode node = hashArray[hashVal].find(key);
44        return node;
45    }
46
47 }
```

链地址法中，装填因子（数据项数和哈希表容量的比值）与开放地址法不同，在链地址法中，需要有N个单元的数组中转入N个或更多的数据项，因此装填因子一般为1，或比1大（有可能某些位置包含的链表中包含两个或两个以上的数据项）。

找到初始单元需要O(1)的时间级别，而搜索链表的时间与M成正比，M为链表包含的平均项数，即O(M)的时间级别。

[回到顶部](#)

5、桶

另外一种方法类似于链地址法，它是在每个数据项中使用子数组，而不是链表。这样的数组称为桶。

这个方法显然不如链表有效，因为桶的容量不好选择，如果容量太小，可能会溢出，如果太大，又造成性能浪费，而链表是动态分配的，不存在此问题。所以一般不使用桶。

[回到顶部](#)

6、总结

哈希表基于数组，类似于key-value的存储形式，关键字值通过哈希函数映射为数组的下标，如果一个关键字哈希化到已占用的数组单元，这种情况称为冲突。用来解决冲突的有两种方法：开放地址法和链地址法。在开放地址法中，把冲突的数据项放在数组的其它位置；在链地址法中，每个单元都包含一个链表，把所有映射到同一数组下标的数据项都插入到这个链表中。

作者：[YSOcean](#)

出处：<http://www.cnblogs.com/ysocan/>
 本文版权归作者所有，欢迎转载，但未经作者同意不能转载，否则保留追究法律责任的权利。

分类：[Java数据结构和算法](#)

标签：[Java数据结构和算法](#)


好文要顶

关注我

收藏该文







YSOcean

关注 - 13

粉丝 - 2067

+加关注

8

0

« 上一篇：[Java数据结构和算法（十二）——2-3-4树](#)
 » 下一篇：[Java数据结构和算法（十四）——堆](#)

posted @ 2018-01-26 22:56 YSOcean 阅读(5914) 评论(4) 编辑 收藏

评论列表

- #1楼 2018-11-20 19:11 梦想从不缺席

学习，打卡

支持(0) 反对(0)
- #2楼 2018-11-30 22:57 一杯热咖啡AAA

厉害，博主。

支持(0) 反对(0)
- #3楼 2019-01-18 14:26 Zoro、

有序链表的代码有两处错误：

19行的 if 里应该加上：

1 | node.next = current;

46行的while循环应该在结尾加上：

1 | current = current.next;

支持(0) 反对(0)
- #4楼[楼主] 2019-01-21 10:57 YSOcean

@ Zoro、
感谢，确实这两处是有错误的，已更正！！

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

- 【课程】开学季给程序员们送福利啦！限量X-box等你来拿！
- 【推荐】超50万C++/C#源码：大型实时仿真组态图形源码
- 【推荐】百度云“猪”你开年行大运，红包疯狂拿
- 【活动】2019开源技术盛宴(6.24~26上海世博中心)
- 【推荐】55K刚面完Java架构师岗，这些技术你必须掌握



相关博文：

- 关于哈希，分布式哈希表，一致性哈希
- 哈希表
- 哈希表
- 哈希表
- 哈希冲突，哈希函数

香港云服务器CN2高速直连仅29元
亿速云香港服务器免备案,20+行业领袖视频推荐 免备案低延时
CN2高速带宽每月低至 29元 亿速云

打开

最新新闻：

- 采访Facebook产品设计师：我是如何从零开始转行成功的？
- 游戏陪练，电竞行业造血者？
- 露露事件背后是腾讯资产的流失
- 电池和续航，可能是苹果AirPods便利性的「最大受害者」
- 收购爱康国宾，阿里巴巴图什么？
- » 更多新闻...