Map和Set

本节目标

> - 掌握 Map/Set 及实际实现类 HashMap/TreeMap/HashSet/TreeSet 的使用
> - 掌握 TreeMap 和 TreeSet 背后的数据结构搜索树的原理和简单实现
> - 掌握 HashMap 和 HashSet 背后的数据结构哈希表的原理和简单实现

1. 搜索

1.1 概念及场景

搜索，查找，也可称检索，是在大量的数据元素中找到某个特定的数据元素而进行的工作。

实际生活中其实很多案例对应的都是搜索，比如：

1. 根据姓名查询考试成绩
2. 通讯录，即根据姓名查询联系方式
3. 不重复集合，即需要先搜索关键字是否已经在集合中

等等。

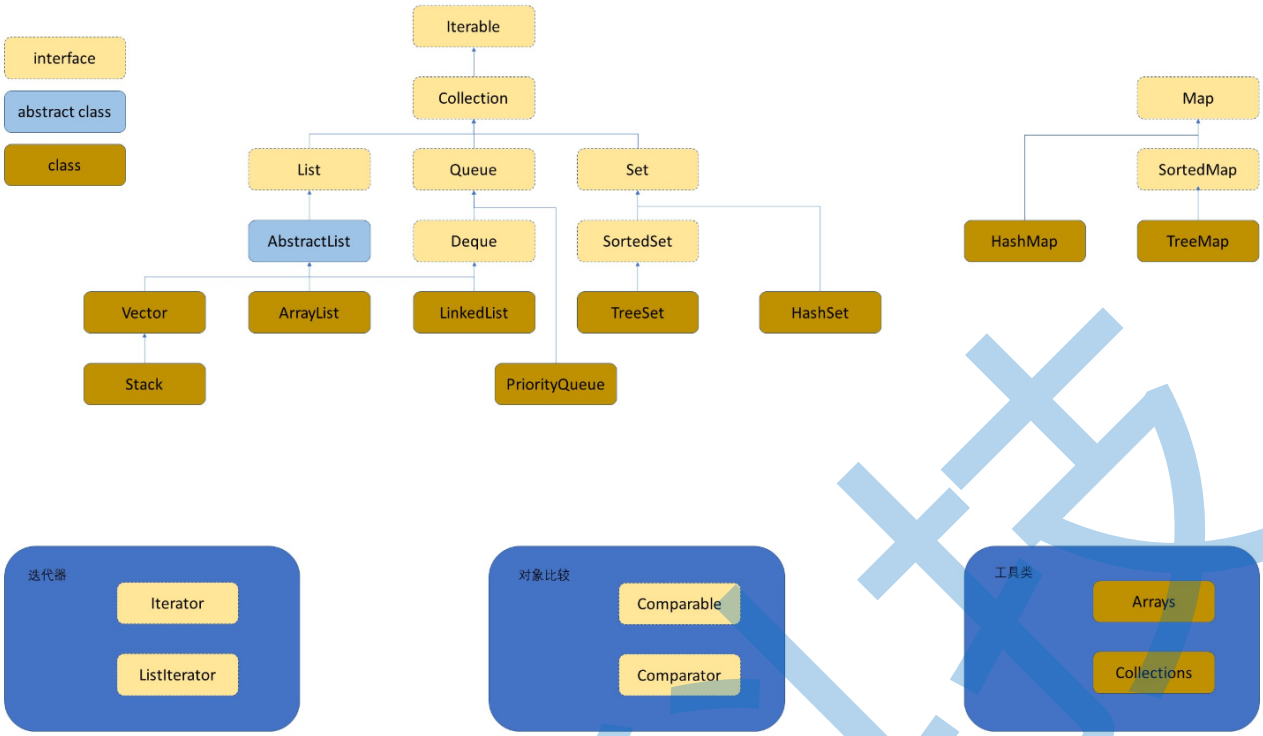
1.2 模型

一般把搜索的数据称为关键字（Key），和关键字对应的称为值（Value），所以模型会有两种：

1. **纯 key 模型**，即我们 **Set** 要解决的事情，只需要判断关键字在不在集合中即可，没有关联的 value；
2. **Key-Value 模型**，即我们 **Map** 要解决的事情，需要根据指定 Key 找到关联的 Value。

2. Map 的使用

Map 的官方文档



2.1 常见方法

Map.Entry<K, V> 即 Map 中定义的 K 类型的 key 和 V 类型的 value 的映射关系的类。

方法	解释
K getKey()	返回 entry 中的 key
V getValue()	返回 entry 中的 value

Map 的常见方法

方法	解释
V get(Object key)	返回 key 对应的 value
V getOrDefault(Object key, V defaultValue)	返回 key 对应的 value，key 不存在，返回默认值
V put(K key, V value)	设置 key 对应的 value
V remove(Object key)	删除 key 对应的映射关系
Set<K> keySet()	返回所有 key 的不重复集合
Collection<V> values()	返回所有 value 的可重复集合
Set<Map.Entry<K, V>> entrySet()	返回所有的 key-value 映射关系
boolean containsKey(Object key)	判断是否包含 key
boolean containsValue(Object value)	判断是否包含 value

2.2 示例

```
import java.util.HashMap;
import java.util.Map;

public class TestDemo {
    public static void main(String[] args) {
        Map<Integer,String> map = new HashMap<>() ;
        map.put(1,"hello") ;
        // key重复
        map.put(1,"Hello") ;
        map.put(3,"Java") ;
        map.put(2,"Bit") ;
        System.out.println(map) ;
        // 根据key取得value
        System.out.println(map.get(2));
        // 查找不到返回null
        System.out.println(map.get(99));

        // 打印所有的 key
        for (Integer key : map.keySet()) {
            System.out.println(key);
        }

        // 打印所有的 value
        for (String value : map.values()) {
            System.out.println(value);
        }

        // 按 key-value 映射关系打印
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }
```

```
    }
}
```

运行结果

```
{1=Hello, 2=Bit, 3=Java}
Bit
null
1
2
3
Hello
Bit
Java
1 = Hello
2 = Bit
3 = Java
```

3. Set 的使用

Set 的官方文档

3.1 常见方法

方法	解释
boolean add(E e)	添加元素，但重复元素不会被添加成功
void clear()	清空集合
boolean contains(Object o)	判断 o 是否在集合中
Iterator<E> iterator()	返回迭代器
boolean remove(Object o)	删除集合中的 o

3.2 示例

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class TestDemo {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>() ;
        set.add("Hello");
        // 重复元素
        set.add("Hello");
        set.add("Bit");
        set.add("Hello");
        set.add("Java");
```

```
        System.out.println(set);

        Iterator<String> it = set.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

运行结果

```
[Java, Hello, Bit]
Java
Hello
Bit
```

4 面试题练习

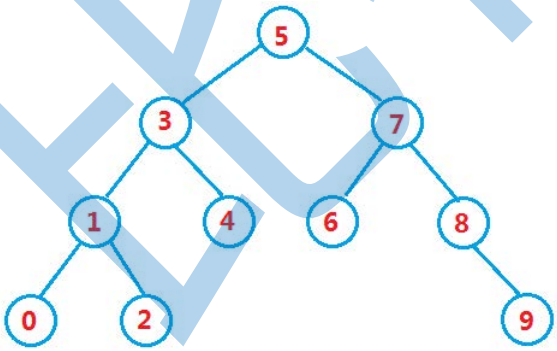
1. 只出现一次的数字
2. 复制带随机指针的链表
3. 宝石与石头
4. 坏键盘打字
5. 前K个高频单词

5. 搜索树

5.1 概念

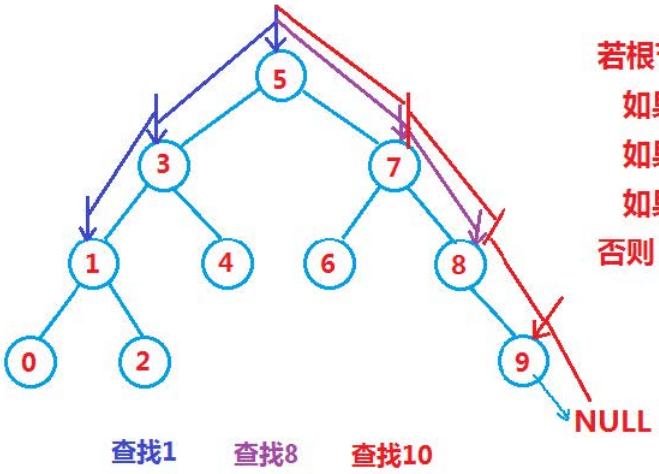
二叉搜索树又称二叉排序树，它或者是一棵空树**，或者是具有以下性质的二叉树:

- **若它的左子树不为空，则左子树上所有节点的值都小于根节点的值**
- **若它的右子树不为空，则右子树上所有节点的值都大于根节点的值**
- **它的左右子树也分别为二叉搜索树**



int a [] = {5,3,4,1,7,8,2,6,0,9};

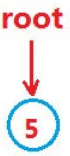
5.2 操作-查找



若根节点不为空：
如果根节点key==查找key 返回true
如果根节点key > 查找key 在其左子树查找
如果根节点key < 查找key 在其右子树查找
否则 返回false

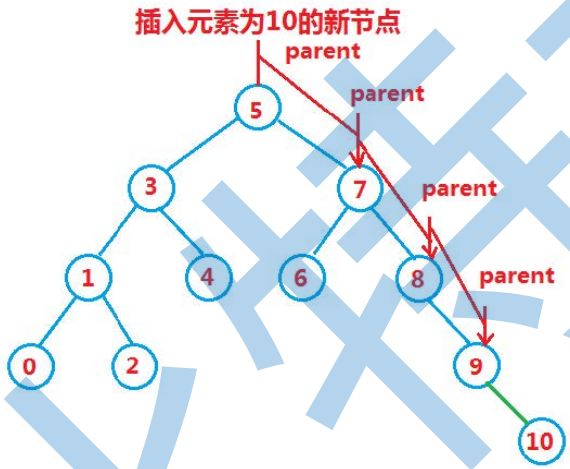
5.3 操作-插入

1. 如果树为空树，即根 == null，直接插入



如果是空树，直接插入，然后返回true

2. 如果树不是空树，按照查找逻辑确定插入位置，插入新结点



1. 按照二叉搜索树的性质，查找到插入结点的位置
root-->5 5<10 root = root->right parent = root
root-->7 7<10 root = root->right parent = root
root-->8 8<10 root = root->right parent = root
root-->9 9<10 root = root->right parent = root

2. 插入新节点

5.4 操作-删除（难点）

设待删除结点为 cur, 待删除结点的双亲结点为 parent

1. **cur.left == null**
    1. cur 是 root，则 root = cur.right
    2. cur 不是 root，cur 是 parent.left，则 parent.left = cur.right
    3. cur 不是 root，cur 是 parent.right，则 parent.right = cur.right
2. **cur.right == null**
    1. cur 是 root，则 root = cur.left
    2. cur 不是 root，cur 是 parent.left，则 parent.left = cur.left
    3. cur 不是 root，cur 是 parent.right，则 parent.right = cur.left
3. **cur.left != null && cur.right != null**

1. 需要使用**替换法**进行删除，即在它的右子树中寻找中序下的第一个结点(关键码最小)，用它的值填补到被删除节点中，再来处理该结点的删除问题

5.5 实现

```
public class BinarySearchTree {
    public static class Node {
        int key;
        Node left;
        Node right;

        public Node(int key) {
            this.key = key;
        }
    }

    private Node root = null;

    /**
     * 在搜索树中查找 key，如果找到，返回 key 所在的结点，否则返回 null
     * @param key
     * @return
     */
    public Node search(int key) {
        Node cur = root;
        while (cur != null) {
            if (key == cur.key) {
                return cur;
            } else if (key < cur.key) {
                cur = cur.left;
            } else {
                cur = cur.right;
            }
        }

        return null;
    }

    /**
     * 插入
     * @param key
     * @return true 表示插入成功，false 表示插入失败
     */
    public boolean insert(int key) {
        if (root == null) {
            root = new Node(key);
            return true;
        }

        Node cur = root;
        Node parent = null;
        while (cur != null) {
            if (key == cur.key) {
```

```
                return false;
            } else if (key < cur.key) {
                parent = cur;
                cur = cur.left;
            } else {
                parent = cur;
                cur = cur.right;
            }
        }

        Node node = new Node(key);
        if (key < parent.key) {
            parent.left = node;
        } else {
            parent.right = node;
        }
        return true;
    }

    /**
     * 删除成功返回 true，失败返回 false
     * @param key
     * @return
     */
    public boolean remove(int key) {
        Node cur = root;
        Node parent = null;
        while (cur != null) {
            if (key == cur.key) {
                // 找到，准备删除
                removeNode(parent, cur);
                return true;
            } else if (key < cur.key) {
                parent = cur;
                cur = cur.left;
            } else {
                parent = cur;
                cur = cur.right;
            }
        }

        return false;
    }

    private void removeNode(Node parent, Node cur) {
        if (cur.left == null) {
            if (cur == root) {
                root = cur.right;
            } else if (cur == parent.left) {
                parent.left = cur.right;
            } else {
                parent.right = cur.right;
            }
```

```
        } else if (cur.right == null) {
            if (cur == root) {
                root = cur.left;
            } else if (cur == parent.left) {
                parent.left = cur.left;
            } else {
                parent.right = cur.left;
            }
        } else {
            Node goatParent = cur;
            Node goat = cur.right;
            while (goat.left != null) {
                goatParent = goat;
                goat = goat.left;
            }

            cur.key = goat.key;
            //cur.value = goat.value;

            if (goat == goatParent.left) {
                goatParent.left = goat.right;
            } else {
                goatParent.right = goat.right;
            }
        }
    }

    public static void main(String[] args) {
        // 1. 创建搜索树
        // 2. 随机插入一些数据
        // 3. 打印前序 + 中序遍历
        // 4. 查找
        BinarySearchTree tree = new BinarySearchTree();
        int[] keys = { 3, 9, 7, 4, 1, 6, 2, 8, 5 };
        for (int key : keys) {
            System.out.println(tree.insert(key));
        }
        System.out.println("插入重复数据");
        System.out.println(tree.insert(7));

        System.out.println("前序遍历");
        preOrder(tree.root);
        System.out.println("中序遍历");
        inOrder(tree.root);

        System.out.println(tree.search(7).key);
        System.out.println(tree.search(8).key);
        System.out.println(tree.search(5).key);
    }

    private static void inOrder(Node node) {
        if (node != null) {
            inOrder(node.left);
```

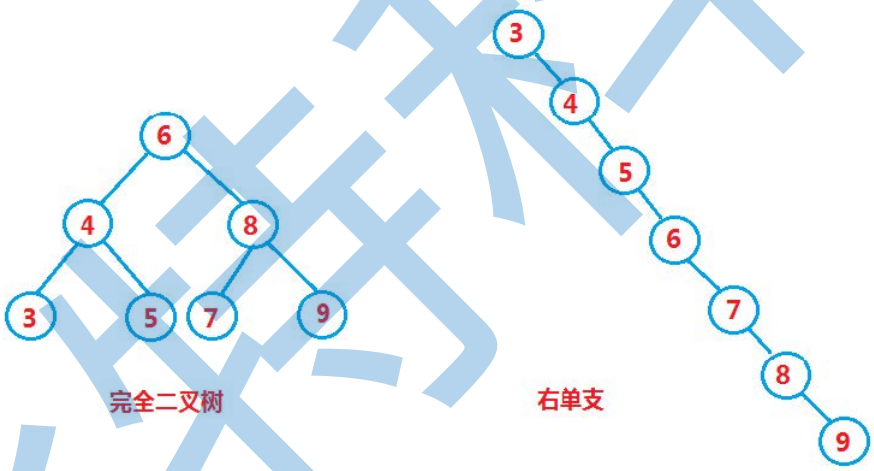
```
            System.out.println(node.key);
            inOrder(node.right);
        }
    }

    private static void preOrder(Node node) {
        if (node != null) {
            System.out.println(node.key);
            preOrder(node.left);
            preOrder(node.right);
        }
    }
}
```

5.6 性能分析

插入和删除操作都必须先查找，查找效率代表了二叉搜索树中各个操作的性能。

对有n个结点的二叉搜索树，若每个元素查找的概率相等，则二叉搜索树平均查找长度是结点在二叉搜索树的深度的函数，即结点越深，则比较次数越多。

但对于同一个关键码集合，如果各关键码插入的次序不同，可能得到不同结构的二叉搜索树：



最优情况下，二叉搜索树为完全二叉树，其平均比较次数为：$log_2 N$

最差情况下，二叉搜索树退化为单支树，其平均比较次数为：$\frac{N}{2}$

问题：如果退化成单支树，二叉搜索树的性能就失去了。那能否进行改进，不论按照什么次序插入关键码，都可以是二叉搜索树的性能最佳？

5.7 和 java 类集的关系

TreeMap 和 TreeSet 即 java 中利用搜索树实现的 Map 和 Set；实际上用的是红黑树，留给我们以后再去学习。

6. 哈希表

6.1 概念

顺序结构以及平衡树中，元素关键码与其存储位置之间没有对应的关系，因此在**查找一个元素时，必须要经过关键码的多次比较。顺序查找时间复杂度为O(N)，平衡树中为树的高度，即O($log_2 N$)**，搜索的效率取决于搜索过程中元素的比较次数。

理想的搜索方法：可以**不经过任何比较，一次直接从表中得到要搜索的元素。 如果构造一种存储结构，通过某种函数(hashFunc)使元素的存储位置与它的关键码之间能够建立一一映射的关系，那么在查找时通过该函数可以很快找到该元素。**

当向该结构中：

- **插入元素**

  根据待插入元素的关键码，以此函数计算出该元素的存储位置并按此位置进行存放

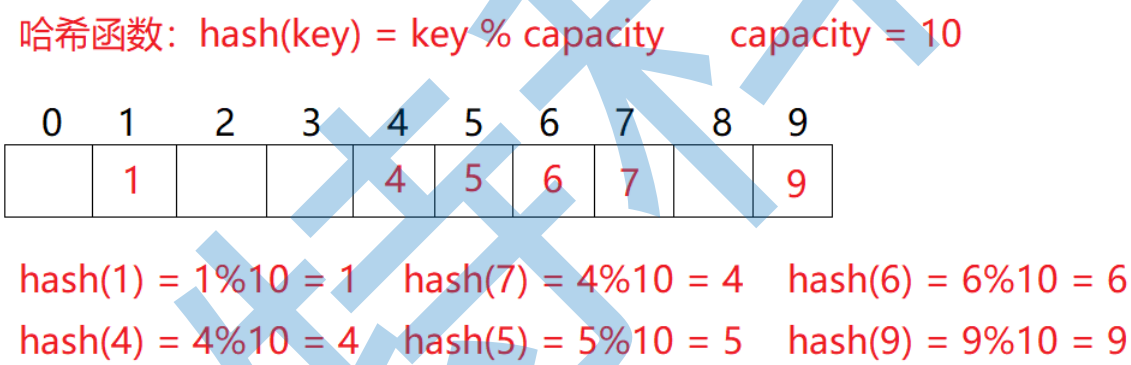
- **搜索元素**

  对元素的关键码进行同样的计算，把求得的函数值当做元素的存储位置，在结构中按此位置取元素比较，若关键码相等，则搜索成功

该方式即为哈希(散列)方法，**哈希方法中使用的转换函数称为哈希(散列)函数，构造出来的结构称为哈希表(Hash Table)(或者称散列表)**

例如：数据集合{1，7，6，4，5，9};

哈希函数设置为：**hash(key) = key % capacity**; capacity为存储元素底层空间总的大小。



用该方法进行搜索不必进行多次关键码的比较，因此搜索的速度比较快 问题：按照上述哈希方式，向集合中插入元素44，会出现什么问题?

6.2 冲突-概念

对于两个数据元素的关键字k_i和 k_j(i != j)，有k_i != k_j，但有：Hash(k_i) == Hash(k_j)，即：**不同关键字通过相同哈希哈数计算出相同的哈希地址，该种现象称为哈希冲突或哈希碰撞。**

把具有不同关键码而具有相同哈希地址的数据元素称为“同义词”。

6.3 冲突-避免

首先，我们需要明确一点，由于我们哈希表底层数组的容量往往是小于实际要存储的关键字的数量的，这就导致一个问题，**冲突的发生是必然的**，但我们能做的应该是尽量的**降低冲突率。**

6.4 冲突-避免-哈希函数设计

引起哈希冲突的一个原因可能是：**哈希函数设计不够合理。 哈希函数设计原则**：

- 哈希函数的定义域必须包括需要存储的全部关键码，而如果散列表允许有m个地址时，其值域必须在0到m-1之间
- 哈希函数计算出来的地址能均匀分布在整个空间中
- 哈希函数应该比较简单

常见哈希函数

1. **直接定制法**--(常用)

   取关键字的某个线性函数为散列地址：Hash（Key）= A*Key + B 优点：简单、均匀 缺点：需要事先知道关键字的分布情况 使用场景：适合查找比较小且连续的情况 面试题：字符串中第一个只出现一次字符

2. **除留余数法**--(常用)

   设散列表中允许的**地址数为m，取一个不大于m，但最接近或者等于m的质数p作为除数，按照哈希函数：Hash(key) = key% p(p<=m),将关键码转换成哈希地址**

3. **平方取中法**--(了解)

   假设关键字为1234，对它平方就是1522756，抽取中间的3位227作为哈希地址； 再比如关键字为4321，对它平方就是18671041，抽取中间的3位671(或710)作为哈希地址 **平方取中法比较适合：不知道关键字的分布，而位数又不是很大的情况**

4. **折叠法**--(了解)

   折叠法是将关键字从左到右分割成位数相等的几部分(最后一部分位数可以短些)，然后将这几部分叠加求和，并按散列表表长，取后几位作为散列地址。

   折叠法适合事先不需要知道关键字的分布，适合关键字位数比较多的情况

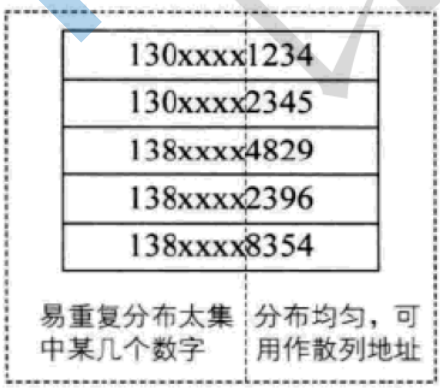
5. **随机数法**--(了解)

   选择一个随机函数，取关键字的随机函数值为它的哈希地址，即H(key) = random(key),其中random为随机数函数。

   通常应用于关键字长度不等时采用此法

6. **数学分析法**--(了解)

   设有n个d位数，每一位可能有r种不同的符号，这r种不同的符号在各位上出现的频率不一定相同，可能在某些位上分布比较均匀，每种符号出现的机会均等，在某些位上分布不均匀只有某几种符号经常出现。可根据散列表的大小，选择其中各种符号分布均匀的若干位作为散列地址。例如：



   假设要存储某家公司员工登记表，如果用手机号作为关键字，那么极有可能前7位都是 相同的，那么我们可以选择后面的四位作为散列地址，如果这样的抽取工作还容易出现 冲突，还可以对抽取出来的数字进行反转(如1234改成4321)、右环位移(如1234改成4123)、左环移位、前两数与后两数叠加(如1234改成12+34=46)等方法。

数字分析法通常适合处理关键字位数比较大的情况，如果事先知道关键字的分布且关键字的若干位分布较均匀的情况

注意：哈希函数设计的越精妙，产生哈希冲突的可能性就越低，但是无法避免哈希冲突

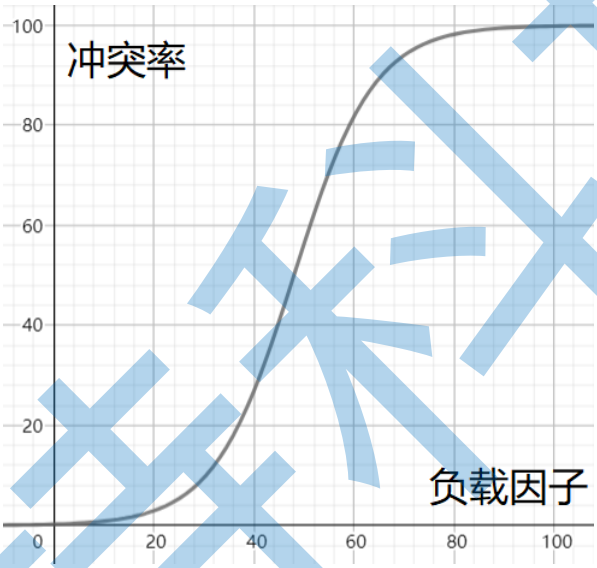
6.5 冲突-避免-负载因子调节（重点掌握）

散列表的载荷因子定义为：α = 填入表中的元素个数 / 散列表的长度

α是散列表装满程度的标志因子。由于表长是定值，α与“填入表中的元素个数”成正比，所以，α越大，表明填入表中的元素越多，产生冲突的可能性就越大；反之，α越小，标明填入表中的元素越少，产生冲突的可能性就越小。实际上，散列表的平均查找长度是载荷因子α的函数，只是不同处理冲突的方法有不同的函数。

对于开放定址法，荷载因子是特别重要因素，应严格限制在0.7-0.8以下。超过0.8，查表时的CPU缓存不命中（cache missing）按照指数曲线上升。因此，一些采用开放定址法的hash库，如Java的系统库限制了荷载因子为0.75，超过此值将resize散列表。

负载因子和冲突率的关系粗略演示



所以当冲突率达到一个无法忍受的程度时，我们需要通过降低负载因子来变相的降低冲突率。

已知哈希表中已有的关键字个数是不可变的，那我们能调整的就只有哈希表中的数组的大小。

6.6 冲突-解决

解决哈希冲突两种常见的方法是：**闭散列**和**开散列**

6.7 冲突-解决-闭散列

闭散列：也叫开放定址法，当发生哈希冲突时，如果哈希表未被装满，说明在哈希表中必然还有空位置，那么可以把key存放到冲突位置中的“下一个” 空位置中去。那如何寻找下一个空位置呢?

1. **线性探测**

   比如上面的场景，现在需要插入元素44，先通过哈希函数计算哈希地址，下标为4，因此44理论上应该插在该位置，但是该位置已经放了值为4的元素，即发生哈希冲突。

   线性探测：从发生冲突的位置开始，依次向后探测，直到寻找到下一个空位置为止。

   - 插入
     - 通过哈希函数获取待插入元素在哈希表中的位置

- 如果该位置中没有元素则直接插入新元素，如果该位置中有元素发生哈希冲突，使用线性探测找到下一个空位置，插入新元素

哈希函数：hash(key) = key % capacity　　capacity = 10

0	1	2	3	4	5	6	7	8	9
	1			4	5	6	7	44	9

hash(1) = 1%10 = 1　hash(7) = 4%10 = 4　hash(6) = 6%10 = 6
hash(4) = 4%10 = 4　hash(5) = 5%10 = 5　hash(9) = 9%10 = 9

- 采用闭散列处理哈希冲突时，不能随便物理删除哈希表中已有的元素，若直接删除元素会影响其他元素的搜索。比如删除元素4，如果直接删除掉，44查找起来可能会受影响。因此线性探测采用标记的伪删除法来删除一个元素。

2. **二次探测**

线性探测的缺陷是产生冲突的数据堆积在一块，这与其找下一个空位置有关系，因为找空位置的方式就是挨着往后逐个去找，因此二次探测为了避免该问题，找下一个空位置的方法为：H_i = (H_0 + i^2)% m，或者：H_i = (H_0 - i^2)% m。其中：i = 1,2,3…， H_0是通过散列函数Hash(x)对元素的关键码 key 进行计算得到的位置，m是表的大小。 对于2.1中如果要插入44，产生冲突，使用解决后的情况为：

哈希函数：hash(key) = key % capacity　　capacity = 10

0	1	2	3	4	5	6	7	8	9
	1			4	5	6	7	44	9

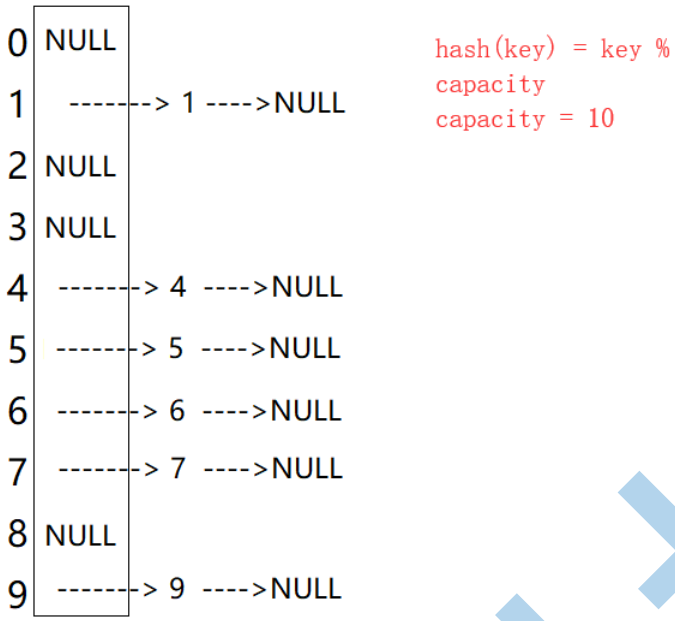
hash(1) = 1%10 = 1　hash(7) = 4%10 = 4　hash(6) = 6%10 = 6
hash(4) = 4%10 = 4　hash(5) = 5%10 = 5　hash(9) = 9%10 = 9

研究表明：当表的长度为质数且表装载因子a不超过0.5时，新的表项一定能够插入，而且任何一个位置都不会被探查两次。因此只要表中有一半的空位置，就不会存在表满的问题。在搜索时可以不考虑表装满的情况，但在插入时必须确保表的装载因子a不超过0.5，如果超出必须考虑增容。

因此：比散列最大的缺陷就是空间利用率比较低，这也是哈希的缺陷。

6.8 冲突-解决-开散列/哈希桶（重点掌握）

开散列法又叫链地址法(开链法)，首先对关键码集合用散列函数计算散列地址，具有相同地址的关键码归于同一子集合，每一个子集合称为一个桶，各个桶中的元素通过一个单链表链接起来，各链表的头结点存储在哈希表中。



从上图可以看出，开散列中每个桶中放的都是发生哈希冲突的元素。

开散列，可以认为是把一个在大集合中的搜索问题转化为在小集合中做搜索了。

6.9 冲突严重时的解决办法

刚才我们提到了，哈希桶其实可以看作将大集合的搜索问题转化为小集合的搜索问题了，那如果冲突严重，就意味着小集合的搜索性能其实也时不佳的，这个时候我们就可以将这个所谓的小集合搜索问题继续进行转化，例如：

1. 每个桶的背后是另一个哈希表
2. 每个桶的背后是一棵搜索树

6.10 实现

```
// key-value 模型
public class HashBucket {
    private static class Node {
        private int key;
        private int value;
        Node next;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private Node[]  array;
    private int size;    // 当前的数据个数
    private static final double LOAD_FACTOR = 0.75;

    public int put(int key, int value) {
        int index = key % array.length;

        // 在链表中查找 key 所在的结点
```

```
        // 如果找到了，更新
        // 所有结点都不是 key，插入一个新的结点
        for (Node cur = array[index]; cur != null; cur = cur.next) {
            if (key == cur.key) {
                int oldValue = cur.value;
                cur.value = value;
                return oldValue;
            }
        }
        Node node = new Node(key, value);
        node.next = array[index];
        array[index] = node;
        size++;

        if (loadFactor() >= LOAD_FACTOR) {
            resize();
        }

        return -1;
    }

    private void resize() {
        Node[] newArray = new Node[array.length * 2];
        for (int i = 0; i < array.length; i++) {
            Node next;
            for (Node cur = array[i]; cur != null; cur = next) {
                next = cur.next;
                int index = cur.key % newArray.length;
                cur.next = newArray[index];
                newArray[index] = cur;
            }
        }
        array = newArray;
    }

    private double loadFactor() {
        return size * 1.0 / array.length;
    }

    public HashBucket() {
        array = new Node[8];
        size = 0;
    }

    public int get(int key) {
        int index = key % array.length;

        Node head = array[index];
        for (Node cur = head; cur != null; cur = cur.next) {
            if (key == cur.key) {
                return cur.value;
            }
        }
```

```
        return -1;
    }
}
```

6.11 性能分析

虽然哈希表一直在和冲突做斗争，但在实际使用过程中，我们认为哈希表的冲突率是不高的，冲突个数是可控的，也就是每个桶中的链表的长度是一个常数，所以，通常意义下，我们认为**哈希表的插入/删除/查找时间复杂度是 O(1) 。**

6.12 和 java 类集的关系

1. HashMap 和 HashSet 即 java 中利用哈希表实现的 Map 和 Set
2. java 中使用的是哈希桶方式解决冲突的
3. java 会在冲突链表长度大于一定阈值后，将链表转变为搜索树（红黑树）
4. java 中计算哈希值实际上是调用的类的 hashCode 方法，进行 key 的相等性比较是调用 key 的 equals 方法。所以如果要用自定义类作为 HashMap 的 key 或者 HashSet 的值，**必须覆写 hashCode 和 equals 方法**，而且要做到 equals 相等的对象，hashCode 一定是一致的。

内容重点总结

- 熟练使用 Map 和 Set
- 掌握搜索树背后原理和基本实现
- 掌握哈希表背后原理和基本实现

课后作业

- 博客总结数据结构中的搜索及 java 中的 Map 和 Set
- 完成课堂上的面试题练习
- 完成搜索树的实现
- 完成哈希表的实现