1. AVL树

前一章说到关于二叉树操作的时间复杂度时，我们提到AVL树可以改善二叉搜索树（BST）的性能。
在二叉搜索树中，搜索、插入和删除操作的时间复杂度取决于树的高度。
在最坏的情况下，如果树退化成链表（即树的高度为O(n)），这些操作的时间复杂度也将是$O(n)$。
理论上，我们可以维护一个完美平衡的树（即完全二叉树），这样树的高度将是$O(logn)$，从而确保搜索、插入和删除操作的时间复杂度为$O(logn)$。
完全二叉树的每一层都完全填满，除了可能的最后一层，这一层从左到右填充。
但维护一个完全平衡的树需要在每次插入或删除操作后进行大量的重建和节点移动，这在实践中是非常昂贵的。

AVL树是一种自平衡二叉搜索树，其中每个节点的两个子树的高度最多相差1。
这种平衡条件确保了树的高度保持在$O(logn)$，从而保证了操作的时间复杂度为$O(logn)$。
与完全平衡的树相比，AVL树需要的平衡努力要少得多，因为它们不需要在每次操作后都进行大量的重建和节点移动。
AVL树通过在插入和删除操作中执行旋转来保持平衡，这些旋转操作相对简单且高效。

平衡因子（Balance Factor）：
平衡因子是一个节点的左子树高度减去右子树高度的结果。
如果一个节点的平衡因子是-1、0或1，则该节点被认为是平衡的，即平衡节点（Balanced Node）。
如果一个节点的平衡因子是-1，即右子树比左子树高1，那么这个节点被认为是左偏的，即左偏节点（Left-Heavy Node）。
如果一个节点的平衡因子是+1，即左子树比右子树高1，那么这个节点被认为是右偏的，即右偏节点（Right-Heavy Node）。

1.1 旋转操作

如果一个节点的平衡因子不再平衡，那么就需要通过旋转操作来重新平衡这个节点。
一共有四种可能的旋转操作：

1. LL旋转（左-左旋转，Left-Left Rotation）。
2. RR旋转（右-右旋转，Right-Right Rotation）。
3. LR旋转（左-右旋转，Left-Right Rotation）。
4. RL旋转（右-左旋转，Right-Left Rotation）。

关于旋转的操作也可以查看算法文章里的讲解：AVL树讲解

1.1.1 LL旋转

当一个节点A的平衡因子为-2时，我们称该节点为左偏（left-heavy）。如果这个左偏的节点A的左子节点B的平衡因子为-1或0，那么节点A就处于LL不平衡状态（Left-Left Imbalance）。
为此我们使用LL旋转用来解决这种不平衡。

具体过程如下。

1. 初始状态：节点A是不平衡的，其平衡因子为-2（左子树比右子树高2层），并且其左子节点B的平衡因子为-1或0（也是左偏或平衡）。
   图中用黄色标记了节点A，用蓝色标记了节点B，用红色标记了节点C。
2. 右旋：首先，将节点A向右旋转，使得原本不受旋转影响的节点F保持在原位置，仍然是节点A的右子节点。
   在旋转过程中，节点B将取代节点A的位置，成为新的根节点。
3. 调整位置：节点A在旋转后将向下移动，成为节点B的右子节点。
   但是，节点B已经有一个右子节点E，这会导致冲突。
4. 避免冲突：为了避免冲突，节点E将在旋转过程中被附加到节点A的左子节点位置。
   这样，节点A现在有了左子节点E，而节点B成为了新的根节点，节点A是其右子节点。

1.1.2 RR旋转

当一个节点A的平衡因子为+2时，我们称该节点为右偏（right-heavy）。如果这个右偏的节点A的右子节点B的平衡因子为+1或0，那么节点A就处于RR不平衡状态（Right-Right Imbalance）。
为此我们使用RR旋转用来解决这种不平衡。

具体过程如下。

1. 初始状态：节点A的平衡因子为+2，表示它是一个右偏（right-heavy）节点，其右子节点B的平衡因子为+1或0，表示B也是右偏或平衡的。
   图中用黄色标记了节点A，用蓝色标记了节点B，用红色标记了节点C。
2. 左旋：首先，将节点A向左旋转，使得原本不受旋转影响的节点F保持在原位置，仍然是节点A的左子节点。
3. 调整位置：在旋转过程中，节点B将取代节点A的位置，成为新的根节点。
   节点A在旋转后将向下移动，成为节点B的左子节点。
4. 避免冲突：为了避免冲突，节点C（B的左子节点）将在旋转过程中被附加到节点A的右子节点位置。

1.1.3 LR旋转

LR不平衡：节点A的平衡因子为-2，意味着它的左子树比右子树高2层。同时，节点A的左子节点B的平衡因子为+1，意味着B的左子树比右子树低1层。
为此我们使用LR旋转用来解决这种不平衡。LR旋转是一种双旋转操作，用于解决LR不平衡。
首先对节点B进行左旋转，以减少B的右偏程度。这会使得B的右子节点成为新的子树的根，而B则成为这个新根的左子节点。
接着对节点A进行右旋转，以解决A的左偏问题。这会使得A的左子节点（现在已经是经过左旋转的B）成为新的子树的根，而A则成为这个新根的右子节点。

具体过程如下。

1. 初始状态：节点A的平衡因子为-2，节点B的平衡因子为+1。
   需要对A的左子树B进行左旋转，然后对A进行右旋转。
   图中用黄色标记了节点A，用蓝色标记了节点B，用红色标记了节点C。
2. 对B进行左旋转：左旋转A的左子树B，使得B与A断开连接。
   C节点（B的右子节点）取代B的位置，成为A的左子节点。

B成为C的左子节点。
3. 对A进行右旋转：由于B已经与A断开，并且C现在位于B原来的位置，我们需要对A进行右旋转。
   在右旋转过程中，A会与它的右子节点F发生冲突。
4. 避免冲突：为了避免冲突，将E（B的右子节点）附加到B作为右子节点，完成对B的旋转。
   然后将F（A的右子节点）附加到A作为左子节点，完成对A的旋转。

1.1.4 RL旋转

RL不平衡：节点A的平衡因子为+2，意味着它的右子树比左子树高2层。同时，节点A的右子节点B的平衡因子为-1，意味着B的右子树比左子树低1层。
为此我们使用RL旋转用来解决这种不平衡。RL旋转也是一种双旋转操作。
首先对节点B进行右旋转，以减少B的左偏程度。这会使得B的左子节点成为新的子树的根，而B则成为这个新根的右子节点。
接着对节点A进行左旋转，以解决A的右偏问题。这会使得A的右子节点（现在已经是经过右旋转的B）成为新的子树的根，而A则成为这个新根的左子节点。

具体过程如下。

1. 初始状态：节点A的平衡因子为+2，表示它是一个右偏（right-heavy）节点，其右子节点B的平衡因子为+1，表示B也是右偏的。
   图中用黄色标记了节点A和B，用红色标记了节点C、E和F。
2. 右旋转B：首先对节点B进行右旋转，使得B与A断开连接。
   C节点（B的左子节点）取代B的位置，成为A的右子节点。
   B成为C的右子节点。
3. 左旋转A：接着对节点A进行左旋转，以解决A的右偏问题。
   在左旋转过程中，A会与它的左子节点E发生冲突。
4. 调整节点位置：为了避免冲突，将F（C的左子节点）附加到B作为左子节点，完成对B的旋转。
   然后将E（A的左子节点）附加到A作为右子节点，完成对A的旋转。

对于后两种复杂的旋转其实还是建议参考算法课上的AVL旋转操作。AVL树讲解

1.2 插入操作

插入每一个节点后，如果出现不平衡节点，我们就使用旋转操作保持AVL树的平衡。
下图展示了插入25，20，5，34，50，30，10的全过程。

1.3 删除操作

删除操作同理，当AVL树不平衡时，通过旋转操作让AVL树平衡。
下面展示了删除34，30，50，5的全过程。

这两部分的操作不理解也可以去看这里算法的讲解。AVL树讲解

1.4 实现AVL树

AVL树同样还是一个二叉搜索树，所以我们可以让AVLTree类继承BST类。
代码如下。

```
public class AVLTree<E> extends BST<E> {
  /** Create an empty AVL tree using a natural comparator*/
  public AVLTree() { // super() is implicitly called
  }

  /** Create a BST with a specified comparator */
  public AVLTree(java.util.Comparator<E> c) {
    super(c);
  }
```

```java
/** Create an AVL tree from an array of objects */
public AVLTree(E[] objects) {
  super(objects);
}

@Override /** Override createNewNode to create an AVLTreeNode */
protected AVLTreeNode<E> createNewNode(E e) {
  return new AVLTreeNode<E>(e);
}

@Override /** Insert an element and rebalance if necessary */
public boolean insert(E e) {
  boolean successful = super.insert(e);
  if (!successful)
    return false; // e is already in the tree
  else {
    balancePath(e); // Balance from e to the root if necessary
  }

  return true; // e is inserted
}

/** Update the height of a specified node */
private void updateHeight(AVLTreeNode<E> node) {
  if (node.left == null && node.right == null) // node is a leaf
    node.height = 0;
  else if (node.left == null) // node has no left subtree
    node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
  else if (node.right == null) // node has no right subtree
    node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
  else
    node.height = 1 +
      Math.max(((AVLTreeNode<E>)(node.right)).height,
      ((AVLTreeNode<E>)(node.left)).height);
}

/** Balance the nodes in the path from the specified
 * node to the root if necessary
 */
private void balancePath(E e) {
  java.util.ArrayList<TreeNode<E>> path = path(e);
  for (int i = path.size() - 1; i >= 0; i--) {
    AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
    updateHeight(A);
    AVLTreeNode<E> parentOfA = (A == root) ? null :
      (AVLTreeNode<E>)(path.get(i - 1));

    switch (balanceFactor(A)) {
      case -2:
        if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {
          balanceLL(A, parentOfA); // Perform LL rotation
        }
        else {
          balanceLR(A, parentOfA); // Perform LR rotation
        }
        break;
      case +2:
        if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {
          balanceRR(A, parentOfA); // Perform RR rotation
        }
        else {
          balanceRL(A, parentOfA); // Perform RL rotation
        }
```

```
    }
  }
}

/** Return the balance factor of the node */
private int balanceFactor(AVLTreeNode<E> node) {
  if (node.right == null) // node has no right subtree
    return -node.height;
  else if (node.left == null) // node has no left subtree
    return +node.height;
  else
    return ((AVLTreeNode<E>)node.right).height -
      ((AVLTreeNode<E>)node.left).height;
}

/** Balance LL (see Figure 26.3) */
private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy

  if (A == root) {
    root = B;
  }
  else {
    if (parentOfA.left == A) {
      parentOfA.left = B;
    }
    else {
      parentOfA.right = B;
    }
  }

  A.left = B.right; // Make T2 the left subtree of A
  B.right = A; // Make A the left child of B
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
}

/** Balance LR (see Figure 26.5) */
private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.left; // A is left-heavy
  TreeNode<E> C = B.right; // B is right-heavy

  if (A == root) {
    root = C;
  }
  else {
    if (parentOfA.left == A) {
      parentOfA.left = C;
    }
    else {
      parentOfA.right = C;
    }
  }

  A.left = C.right; // Make T3 the left subtree of A
  B.right = C.left; // Make T2 the right subtree of B
  C.left = B;
  C.right = A;

  // Adjust heights
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
  updateHeight((AVLTreeNode<E>)C);
}
```

```
/** Balance RR (see Figure 26.4) */
private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy

  if (A == root) {
    root = B;
  }
  else {
    if (parentOfA.left == A) {
      parentOfA.left = B;
    }
    else {
      parentOfA.right = B;
    }
  }

  A.right = B.left; // Make T2 the right subtree of A
  B.left = A;
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
}

/** Balance RL (see Figure 26.6) */
private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.right; // A is right-heavy
  TreeNode<E> C = B.left; // B is left-heavy

  if (A == root) {
    root = C;
  }
  else {
    if (parentOfA.left == A) {
      parentOfA.left = C;
    }
    else {
      parentOfA.right = C;
    }
  }

  A.right = C.left; // Make T2 the right subtree of A
  B.left = C.right; // Make T3 the left subtree of B
  C.left = A;
  C.right = B;

  // Adjust heights
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
  updateHeight((AVLTreeNode<E>)C);
}

@Override /** Delete an element from the binary tree.
 * Return true if the element is deleted successfully
 * Return false if the element is not in the tree */
public boolean delete(E element) {
  if (root == null)
    return false; // Element is not in the tree

  // Locate the node to be deleted and also locate its parent node
  TreeNode<E> parent = null;
  TreeNode<E> current = root;
  while (current != null) {
    if (c.compare(element, current.element) < 0) {
      parent = current;
```

```
        current = current.left;
      }
      else if (c.compare(element, current.element) > 0) {
        parent = current;
        current = current.right;
      }
      else
        break; // Element is in the tree pointed by current
    }

    if (current == null)
      return false; // Element is not in the tree

    // Case 1: current has no left children (See Figure 23.6)
    if (current.left == null) {
      // Connect the parent with the right child of the current node
      if (parent == null) {
        root = current.right;
      }
      else {
        if (c.compare(element, parent.element) < 0)
          parent.left = current.right;
        else
          parent.right = current.right;

        // Balance the tree if necessary
        balancePath(parent.element);
      }
    }
    else {
      // Case 2: The current node has a left child
      // Locate the rightmost node in the left subtree of
      // the current node and also its parent
      TreeNode<E> parentOfRightMost = current;
      TreeNode<E> rightMost = current.left;

      while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right; // Keep going to the right
      }

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;

      // Eliminate rightmost node
      if (parentOfRightMost.right == rightMost)
        parentOfRightMost.right = rightMost.left;
      else
        // Special case: parentOfRightMost is current
        parentOfRightMost.left = rightMost.left;

      // Balance the tree if necessary
      balancePath(parentOfRightMost.element);
    }

    size--;
    return true; // Element inserted
  }
}
```

测试代码如下。

```
package chapter26;
```

```java
import chapter25.BST;

public class TestAVLTree {
  public static void main(String[] args) {
    // Create an AVL tree
    AVLTree<Integer> tree = new AVLTree<>(new Integer[]{25,
      20, 5});
    System.out.print("After inserting 25, 20, 5:");
    printTree(tree);

    tree.insert(34);
    tree.insert(50);
    System.out.print("\nAfter inserting 34, 50:");
    printTree(tree);

    tree.insert(30);
    System.out.print("\nAfter inserting 30");
    printTree(tree);

    tree.insert(10);
    System.out.print("\nAfter inserting 10");
    printTree(tree);

    tree.delete(34);
    tree.delete(30);
    tree.delete(50);
    System.out.print("\nAfter removing 34, 30, 50:");
    printTree(tree);

    tree.delete(5);
    System.out.print("\nAfter removing 5:");
    printTree(tree);

    System.out.print("\nTraverse the elements in the tree: ");
    for (int e: tree) {
      System.out.print(e + " ");
    }
  }

  public static void printTree(BST tree) {
    // Traverse tree
    System.out.print("\nInorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

1.3 时间复杂度分析

AVL树的时间复杂度分析表明，无论是搜索、插入还是删除操作，它们的时间复杂度都是$O(logn)$。这是因为：
AVL树的高度是$O(logn)$，这意味着从根节点到任何叶子节点的路径长度最多是$O(logn)$。
在balancePath()方法中，尽管需要遍历从新节点到根节点的路径，但每个节点上的操作（如更新高度、计算平衡因子和执行旋转）都是常数时间操作，不依赖于树的大小。
因此，AVL树能够保持高效的操作性能，即使在动态变化（插入和删除）的情况下。

2. 哈希（Hashing）

哈希在集合（Sets）和映射（Maps）中被使用。
如果不使用哈希，检查一个元素是否存在于集合中或从映射中检索一个值，可能需要遍历整个集合或映射。
哈希函数可以将每个元素或键映射到一个特定的索引，这样我们就可以跳转到元素或键应该在的位置。
哈希函数的输出通常称为哈希码（Hash Code）或哈希值（Hash Value）。
哈希的主要优势在于它提供了快速的访问速度。
平均情况下，我们可以在O(1)时间内搜索、插入和删除元素（不需要搜索和比较）。

2.1 哈希的实现

我们先说到数组，它允许通过索引（index）直接访问其元素。如果我们知道数组中某个元素的索引，那么我们可以在常数时间内$O(1)$访问或更新这个元素。
哈希表是一种使用数组来存储数据的实现方式，其中的数组被称为哈希表。哈希表通过哈希函数将键（key）映射到数组的索引，从而实现快速的数据访问。

哈希函数接受一个搜索键（Search Key）作为输入，并将其转换为一个整数值，这个整数值被称为哈希码（Hash Code）。
然后使用压缩（compressing）将输入数据映射到一个固定范围内的整数，这个整数通常用作哈希表（Hash Table）的索引。

下面的代码将得到一个哈希码。

```
Object o = new Object();
System.out.println(o.hashCode());
```

由于哈希码可能是一个非常大的数，而哈希表的大小通常是有限的（例如，我们通常不会有一个包含超过3亿个位置的哈希表），因此需要对哈希码进行压缩，以将其转换为哈希表的有效索引。

2.1 为基本数据类型生成哈希码

对于这些较小的整数类型（byte、short、char和int类型），可以直接转换为int类型来生成哈希码。
例子如下。

```
char a = 'A';
int charHash = (int) a;
```

对于float类型，可以使用Float.floatToIntBits(f)方法将其转换为原始的整数位表示形式，然后使用这个整数作为哈希码。
例子如下。

```
System.out.println(Float.floatToIntBits(1.23f))
```

对于long类型，由于其大小为64位，而int类型只能容纳32位，因此需要将long类型的高32位和低32位组合起来生成哈希码。
这可以通过使用XOR（异或）操作来完成，以确保结果适合int类型。
下面的代码演示了这一操作。

```
int hashCode = (int)(key ^ (key >>> 32));
```

例如，long value1 = 0xFFFFFFFFABCDEF00L;和long value2 = 0x12345678ABCDEF00L;直接转换为int类型都会得到ABCDEF00L，即直接丢失高32位的信息，只保留低32位的信息。
按照我们说的解决方案的结果如下。

double类型是64位的，可以使用Double.doubleToLongBits(value)方法将其转换为一个long类型值。这个方法返回一个表示double值的64位二进制补码表示形式。
所以现在使用使用与long类型相同的XOR操作来生成哈希码。

```
int hashCode = (int)(bits ^ (bits >>> 32))。
```

这确保了哈希码能够保留double值的高32位和低32位的信息。

对于String类型，生成哈希码的步骤如下：

1. 初始值：初始化一个hashCode变量为0。
2. 逐字符处理：对于字符串中的每个字符，执行以下操作：
   将当前的hashCode值乘以一个固定的质数（例如31、33、41等）。
   将当前字符的整数值加到结果上。
   这个过程累积了每个字符的效果，使得哈希码对字符及其顺序都敏感。

```
int hashCode = 0;
for (int i = 0; i < length; i++) {
    hashCode = 31 * hashCode + charAt(i);
}
```

2.2 压缩（Compressing）

哈希码可能是一个大整数，超出了哈希表索引的范围。
例如，如果哈希表的大小是11（索引从0到10），哈希码可能是一个大整数，如366712642
。我们需要将哈希码缩小以适应索引的范围。假设哈希表的索引在0和N-1之间，其中N
是哈希表的大小（容量），定义了可用于存储元素的槽数。最常见的缩放整数的方法是使用模运算（$modulusoperator$）：
h(hashCode) = hashCode \% N。N$是哈希表的大小，通常选择一个质数（例如11、13、31等）以确保索引均匀分布。
选择质数作为哈希表的大小有助于减少哈希冲突，因为质数在数学上具有某些理想的分布属性。

2.3 冲突处理（Handling Collisions）

当两个或多个键被哈希函数映射到哈希表的同一索引时，就会发生冲突。
例如哈希值是1和12，而哈希表的大小是11，所以12被压缩成了1，这时候就发生了冲突。

处理冲突的两种主要方法为开放寻址（Open Addressing）和分离链接（Separate Chaining）。
开放寻址（Open Addressing）是在发生冲突时，寻找哈希表中的另一个开放位置的过程。
开放寻址有许多变体：

1. 线性探测（Linear Probing）：从冲突的索引开始，按照固定步长（通常是1）顺序查找下一个开放位置。
2. 二次探测（Quadratic Probing）：按照二次函数（如i^2）的序列查找开放位置。
3. 双重哈希（Double Hashing）：使用第二个哈希函数来确定探测序列，通常提供比线性或二次探测更好的性能。

分离链接（Separate Chaining）是将所有具有相同哈希索引的条目放在同一个位置的链表中。

2.3.1 线性探测（Linear Probing）

如果在哈希表的hashTable[key % N]位置发生冲突，线性探测会检查下一个连续的位置hashTable[(key+1) % N]是否可用。
如果这个位置也不可用，继续检查hashTable[(key+2) % N]，依此类推，直到找到一个可用的单元格。

这里不直接+1而是还保留模运算的原因是确保索引值在哈希表的大小范围内。

下图展示了一个例子。
当需要从哈希表中删除一个条目时，首先搜索与键匹配的条目。
如果找到了匹配的条目，不是直接将其删除，而是在该位置放置一个特殊的标记，以表示该条目已被删除，但该位置仍然可以用于插入其他值。
这样做的原因是为了保持哈希表中元素的顺序，以便在后续的搜索、插入或删除操作中，可以通过线性探测策略找到正确的位置。

因此在哈希表中，每个单元格可以处于三种状态之一：被占用（occupied）、被标记（marked）、或为空（empty）。
被标记的单元格也是可用于插入新元素的。

线性探测导致了聚类问题：线性探测倾向于导致哈希表中连续的单元格被占用，这些连续占用的单元格形成的组被称为聚类或簇（cluster）。
当发生冲突时，新元素总是被放置在附近的空单元格中（例如k%N, (k+1)%N, (k+2)%N等）。
这种邻近放置的策略导致占用的单元格累积并形成聚类。
每个聚类实际上是一个探测序列，当检索、添加或删除条目时，都必须搜索这个序列。
这是线性探测的一个主要缺点，因为它增加了操作的复杂度和时间成本，特别是在聚类很长时。
例如我们对于前面的哈希表，再添加37的时候会进行额外的操作，因为原来的单元格已被占用。

2.3.2 二次探测（Quadratic Probing）

二次探测不是简单地线性地查找下一个位置，而是按照二次函数的序列来查找。这个序列通常形式为$(key + j^2)\%N$，其中j是一个非负整数（0, 1, 2, 3, ...），N是哈希表的大小。
例如，如果key是哈希码计算出的索引，那么探测的序列将是key，$(key + 1^2)\%N$，$(key + 2^2)\%N$，$(key + 3^2)\%N$，依此类推。

所以像之前的例子，26将会在第二步后直接添加。

2.3.3 双重哈希（Double Hashing）

双重哈希使用两个哈希函数来确定键在哈希表中的存储位置和冲突解决时的探测序列。
第一个哈希函数$h(key)$用于确定初始位置。
第二个哈希函数$h'(key)$用于确定探测序列的步长（step size），从而避免聚类（clustering）问题。
通常形式为$(h(key) + j * h\prime(key))\%N$, 例如，如果$key$是哈希码计算出的索引，那么探测的序列将是$h(key) + 0 * h\prime(key))\%N$，$h(key) + 1 * h\prime(key))\%N$，$h(key) + 2 * h\prime(key))\%N$，$h(key) + 3 * h\prime(key))\%N$，依此类推。
下面给一个完整的例子。

现在第一个哈希函数是$h(key) = key\%11$，第二个哈希函数是$h'(key) = 7 - key\%7$
对于$key = 12$，初始位置是1,步长是2，探测序列将是$1, 3, 5, 7, \ldots$
这里需要注意$h'(key)$需要保证步长永远不是0并且与与哈希表的大小N互质，以确保探测序列均匀分布，从而避免了线性探测中连续键值导致的聚类问题。
例如这里是$7 - key\%7$从而保证永远不是0，而7与哈希表的大小11互质。因此这里也可以是$h'(key) = 6 - key\%6$

2.3.4 分离链接（Separate Chaining）

它将所有具有相同哈希索引的条目放置在同一个位置，而不是寻找新的位置。

在分离链接方案中，哈希表的每个位置被称为一个桶（bucket）。
桶是用来存储具有相同哈希索引的多个条目的容器。
桶可以使用数组、ArrayList或LinkedList等数据结构来实现。
例如，如果使用LinkedList，那么每个桶就是一个链表，链表的头部存储在哈希表的对应位置。

2.4 负载因子（Load Factor）和重新哈希（Rehashing）

负载因子λ（通常表示为λ）是哈希表中元素数量与哈希表大小（即桶的数量）之间的比率。
所以负载因子的计算公式为：$\lambda = n/N$,其中n是哈希表中的元素数量，N是哈希表的总位置数。
对于开放寻址（Open Addressing）方案，负载因子λ的值在0到1之间。
如果哈希表为空，则$\lambda = 0$。
如果哈希表满了（即所有位置都被占用），则$\lambda = 1$。
对于分离链接（Separate Chaining）方案，负载因子λ可以是任何值，因为它不直接影响哈希表的探测过程。分离链接方案通过链表来处理冲突，因此即使哈希表的某些位置满了，也可以通过链表来存储更多的元素。

当负载因子λ增加时，冲突的概率也随之增加。
这是因为负载因子衡量了哈希表的填充程度，填充程度越高，冲突的可能性越大。
对于开放寻址（Open Addressing）方案，建议将负载因子维持在0.5以下。
对于分离链接（Separate Chaining）方案，负载因子可以维持在0.9以下。

当负载因子超过预设阈值时（例如，Java中的 HashMap 类通常设置为0.75），就需要增加哈希表的大小，并将所有条目重新加载到新的、更大的哈希表中，这个过程称为重新哈希（Rehashing）。
由于哈希表大小（N）已经改变，需要更新哈希函数以适应新的哈希表大小。
重新哈希是一个成本较高的操作，因为它涉及到重新计算哈希码和重新插入元素。所以为了减少重新哈希的频率，建议至少将哈希表大小加倍。

2.5 在Java中实现

HashMap的UML图如图所示。

MyMap类的代码如下。

```
public interface MyMap<K, V> {
  /** Remove all of the entries from this map */
  public void clear();

  /** Return true if the specified key is in the map */
  public boolean containsKey(K key);

  /** Return true if this map contains the specified value */
  public boolean containsValue(V value);

  /** Return a set of entries in the map */
  public java.util.Set<Entry<K, V>> entrySet();

  /** Return the first value that matches the specified key */
  public V get(K key);

  /** Return true if this map contains no entries */
  public boolean isEmpty();

  /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet();

  /** Add an entry (key, value) into the map */
  public V put(K key, V value);

  /** Remove the entries for the specified key */
  public void remove(K key);

  /** Return the number of mappings in this map */
  public int size();

  /** Return a set consisting of the values in this map */
  public java.util.Set<V> values();

  /** Define inner class for Entry */
  public static class Entry<K, V> {
    K key;
    V value;

    public Entry(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey() {
      return key;
    }

    public V getValue() {
      return value;
    }

    @Override
    public String toString() {
      return "[" + key + ", " + value + "]";
    }
  }
}
```

MyHashMap类的代码如下。

```
package org.example;

import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor used in the hash table
  private float loadFactorThreshold;

  // The number of entries in the map
  private int size = 0;

  // Hash table is an array with each cell that is a linked list
  LinkedList<MyMap.Entry<K,V>>[] table;

  /** Construct a map with the default capacity and load factor */
  public MyHashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a map with the specified initial capacity and
   * default load factor */
  public MyHashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a map with the specified initial capacity
   * and load factor */
  public MyHashMap(int initialCapacity, float loadFactorThreshold) {
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactorThreshold = loadFactorThreshold;
    table = new LinkedList[capacity];
  }

  @Override /** Remove all of the entries from this map */
  public void clear() {
    size = 0;
    removeEntries();
  }

  @Override /** Return true if the specified key is in the map */
  public boolean containsKey(K key) {
    if (get(key) != null)
      return true;
    else
      return false;
  }
```

```
@Override /** Return true if this map contains the value */
public boolean containsValue(V value) {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        if (entry.getValue().equals(value))
          return true;
    }
  }

  return false;
}

@Override /** Return a set of entries in the map */
public java.util.Set<MyMap.Entry<K,V>> entrySet() {
  java.util.Set<MyMap.Entry<K, V>> set =
    new java.util.HashSet<>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry);
    }
  }

  return set;
}

@Override /** Return the value that matches the specified key */
public V get(K key) {
  int bucketIndex = hash(key.hashCode());
  if (table[bucketIndex] != null) {
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key))
        return entry.getValue();
  }

  return null;
}

@Override /** Return true if this map contains no entries */
public boolean isEmpty() {
  return size == 0;
}

@Override /** Return a set consisting of the keys in this map */
public java.util.Set<K> keySet() {
  java.util.Set<K> set = new java.util.HashSet<K>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry.getKey());
    }
  }

  return set;
}
```

```java
@Override /** Add an entry (key, value) into the map */
public V put(K key, V value) {
  if (get(key) != null) { // The key is already in the map
    int bucketIndex = hash(key.hashCode());
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
        V oldValue = entry.getValue();
        // Replace old value with new value
        entry.value = value;
        // Return the old value for the key
        return oldValue;
      }
  }

  // Check load factor
  if (size >= capacity * loadFactorThreshold) {
    if (capacity == MAXIMUM_CAPACITY)
      throw new RuntimeException("Exceeding maximum capacity");

    rehash();
  }

  int bucketIndex = hash(key.hashCode());

  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] == null) {
    table[bucketIndex] = new LinkedList<Entry<K, V>>();
  }

  // Add a new entry (key, value) to hashTable[index]
  table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));

  size++; // Increase size

  return value;
}

@Override /** Remove the entries for the specified key */
public void remove(K key) {
  int bucketIndex = hash(key.hashCode());

  // Remove the first entry that matches the key from a bucket
  if (table[bucketIndex] != null) {
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
        bucket.remove(entry);
        size--; // Decrease size
        break; // Remove just one entry that matches the key
      }
  }
}

@Override /** Return the number of entries in this map */
public int size() {
  return size;
}

@Override /** Return a set consisting of the values in this map */
public java.util.Set<V> values() {
  java.util.Set<V> set = new java.util.HashSet<>();

  for (int i = 0; i < capacity; i++) {
```

```
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry.getValue());
    }
  }

  return set;
}

/** Hash function */
private int hash(int hashCode) {
  return supplementalHash(hashCode) & (capacity - 1);
}

/** Ensure the hashing is evenly distributed */
private static int supplementalHash(int h) {
  h ^= (h >>> 20) ^ (h >>> 12);
  return h ^ (h >>> 7) ^ (h >>> 4);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
    capacity <<= 1;
  }

  return capacity;
}

/** Remove all entries from each bucket */
private void removeEntries() {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      table[i].clear();
    }
  }
}

/** Rehash the map */
private void rehash() {
  java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
  capacity <<= 1; // Double capacity
  table = new LinkedList[capacity]; // Create a new hash table
  size = 0; // Reset size to 0

  for (Entry<K, V> entry: set) {
    put(entry.getKey(), entry.getValue()); // Store to new table
  }
}

@Override
public String toString() {
  StringBuilder builder = new StringBuilder("[");

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null && table[i].size() > 0)
      for (Entry<K, V> entry: table[i])
        builder.append(entry);
  }

  builder.append("]");
  return builder.toString();
```

```
  }
}
```

这里有一个补充哈希的操作，这是一种减少哈希冲突的技术，它通过将高位的哈希码移动到低位并与原始哈希码进行XOR（异或）操作，从而使得哈希码分布更加均匀。
这样也可以使得哈希码在哈希表中的分布更加均匀，减少聚类现象。
这个技术在代码中的方法是supplementalHash(int h)。

下图展示了HashSet的UML图。

MyHashSet 直接实现了 Collection接口，因此需要提供 iterator() 方法来支持元素的遍历。
而 MyHashMap 实现了自定义的 MyMap 接口，通过返回集合的方式来间接提供遍历能力。
其代码如下。

```
import java.util.*;

public class MyHashSet<E> implements Collection<E> {
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;

  // Specify a load factor threshold used in the hash table
  private float loadFactorThreshold;

  // The number of elements in the set
  private int size = 0;

  // Hash table is an array with each cell that is a linked list
  private LinkedList<E>[] table;

  /** Construct a set with the default capacity and load factor */
  public MyHashSet() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a set with the specified initial capacity and
   * default load factor */
  public MyHashSet(int initialCapacity) {
    this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
  }

  /** Construct a set with the specified initial capacity
   * and load factor */
  public MyHashSet(int initialCapacity, float loadFactorThreshold) {
    if (initialCapacity > MAXIMUM_CAPACITY)
      this.capacity = MAXIMUM_CAPACITY;
    else
      this.capacity = trimToPowerOf2(initialCapacity);

    this.loadFactorThreshold = loadFactorThreshold;
    table = new LinkedList[capacity];
  }

  @Override /** Remove all elements from this set */
  public void clear() {
```

```
    size = 0;
    removeElements();
  }

  @Override /** Return true if the element is in the set */
  public boolean contains(Object e) {
    int bucketIndex = hash(e.hashCode());
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      return bucket.contains(e);
    }

    return false;
  }

  @Override /** Add an element to the set */
  public boolean add(E e) {
    if (contains(e)) // Duplicate element not stored
      return false;

    if (size + 1 > capacity * loadFactorThreshold) {
      if (capacity == MAXIMUM_CAPACITY)
        throw new RuntimeException("Exceeding maximum capacity");

      rehash();
    }

    int bucketIndex = hash(e.hashCode());

    // Create a linked list for the bucket if it is not created
    if (table[bucketIndex] == null) {
      table[bucketIndex] = new LinkedList<E>();
    }

    // Add e to hashTable[index]
    table[bucketIndex].add(e);

    size++; // Increase size

    return true;
  }

  @Override /** Remove the element from the set */
  public boolean remove(Object e) {
    if (!contains(e))
      return false;

    int bucketIndex = hash(e.hashCode());

    // Create a linked list for the bucket if it is not created
    if (table[bucketIndex] != null) {
      LinkedList<E> bucket = table[bucketIndex];
      bucket.remove(e);
    }

    size--; // Decrease size

    return true;
  }

  @Override /** Return true if the set contains no elements */
  public boolean isEmpty() {
    return size == 0;
  }
```

```
@Override /** Return the number of elements in the set */
public int size() {
  return size;
}

@Override /** Return an iterator for the elements in this set */
public java.util.Iterator<E> iterator() {
  return new MyHashSetIterator(this);
}

/** Inner class for iterator */
private class MyHashSetIterator implements java.util.Iterator<E> {
  // Store the elements in a list
  private java.util.ArrayList<E> list;
  private int current = 0; // Point to the current element in list
  private MyHashSet<E> set;

  /** Create a list from the set */
  public MyHashSetIterator(MyHashSet<E> set) {
    this.set = set;
    list = setToList();
  }

  @Override /** Next element for traversing? */
  public boolean hasNext() {
    return current < list.size();
  }

  @Override /** Get current element and move cursor to the next */
  public E next() {
    return list.get(current++);
  }

  @Override /** Remove the element returned by the last next() */
  public void remove() {
    // Left as an exercise
    // You need to remove the element from the set
    // You also need to remove it from the list
  }
}

/** Hash function */
private int hash(int hashCode) {
  return hashCode & (capacity - 1);
}

/** Return a power of 2 for initialCapacity */
private int trimToPowerOf2(int initialCapacity) {
  int capacity = 1;
  while (capacity < initialCapacity) {
    capacity <<= 1;
  }

  return capacity;
}

/** Remove all e from each bucket */
private void removeElements() {
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      table[i].clear();
    }
  }
```

```
  }

  /** Rehash the set */
  private void rehash() {
    java.util.ArrayList<E> list = setToList(); // Copy to a list
    capacity <<= 1; // Double capacity
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Reset size

    for (E element: list) {
      add(element); // Add from the old table to the new table
    }
  }

  /** Copy elements in the hash set to an array list */
  private java.util.ArrayList<E> setToList() {
    java.util.ArrayList<E> list = new java.util.ArrayList<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        for (E e: table[i]) {
          list.add(e);
        }
      }
    }

    return list;
  }

  @Override
  public String toString() {
    java.util.ArrayList<E> list = setToList();
    StringBuilder builder = new StringBuilder("[");

    // Add the elements except the last one to the string builder
    for (int i = 0; i < list.size() - 1; i++) {
      builder.append(list.get(i) + ", ");
    }

    // Add the last element in the list to the string builder
    if (list.size() == 0)
      builder.append("]");
    else
      builder.append(list.get(list.size() - 1) + "]");

    return builder.toString();
  }

  @Override
  public boolean addAll(Collection<? extends E> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean containsAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public boolean removeAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
```

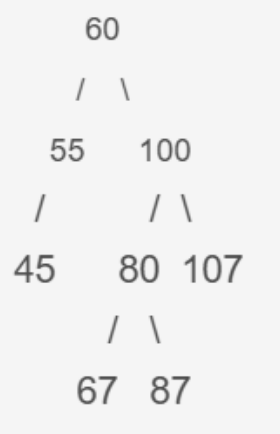
```
  }

  @Override
  public boolean retainAll(Collection<?> arg0) {
    // Left as an exercise
    return false;
  }

  @Override
  public Object[] toArray() {
    // Left as an exercise
    return null;
  }

  @Override
  public <T> T[] toArray(T[] arg0) {
    // Left as an exercise
    return null;
  }
}
```

3. 练习

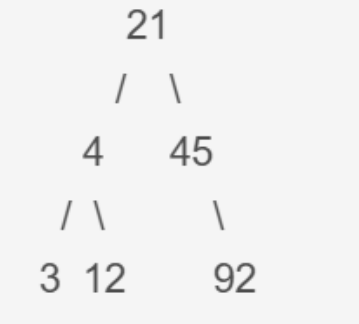
3.1 基础练习

1.这个AVL树添加80后的结果是什么？进行什么旋转实现的平衡?

答案：如图所示。RR旋转。



2.插入元素 3, 4, 45, 21, 92, 12 后的AVL树的前序遍历结果是什么?
答案：21，4，3，12，45，92。
具体的AVL树图如下。



3.2 进阶练习

3.2.1 开放寻址的哈希映射

创建一个名为 MyOpenAddressingMap 的新类，该类使用开放寻址和二次探测来实现 MyMap 接口。哈希函数应使用 f(key) = key % size，其中 size 是哈希表的大小，初始哈希表大小为4。当负载因子超过阈值（0.5）时，哈希表的大小翻倍。
示例代码如下。

```
import java.util.ArrayList;
import java.util.Arrays;

interface MyMap<K, V> {
    void put(K key, V value);
    V get(K key);
    boolean containsKey(K key);
    void remove(K key);
    int size();
    boolean isEmpty();
}

class MyOpenAddressingMap<K, V> implements MyMap<K, V> {
    private static final float LOAD_FACTOR_THRESHOLD = 0.5f;
    private ArrayList<Entry<K, V>> table;
    private int size;
    private int capacity;

    static class Entry<K, V> {
        K key;
        V value;
        boolean isDeleted;

        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
            this.isDeleted = false;
        }
    }

    public MyOpenAddressingMap(int initialCapacity) {
        this.capacity = initialCapacity;
        this.table = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) {
            table.add(null);
        }
        this.size = 0;
    }

    @Override
    public void put(K key, V value) {
        if (key == null) {
            throw new IllegalArgumentException("Key cannot be null");
        }

        int hashCode = key.hashCode();
        int index = hashCode % capacity;

        if (table.get(index) == null || table.get(index).isDeleted) {
            table.set(index, new Entry<>(key, value));
            size++;
            if ((float) size / capacity >= LOAD_FACTOR_THRESHOLD) {
                rehash();
            }
            return;
        }

        int j = 0;
```

```
        while (true) {
            index = (hashCode + j * j) % capacity;
            if (table.get(index) == null || table.get(index).isDeleted) {
                table.set(index, new Entry<>(key, value));
                size++;
                if ((float) size / capacity >= LOAD_FACTOR_THRESHOLD) {
                    rehash();
                }
                return;
            }
            j++;
        }
    }

    @Override
    public V get(K key) {
        if (key == null) {
            return null;
        }

        int hashCode = key.hashCode();
        int index = hashCode % capacity;

        if (table.get(index) != null && table.get(index).key.equals(key)) {
            return table.get(index).value;
        }

        int j = 0;
        while (true) {
            index = (hashCode + j * j) % capacity;
            if (table.get(index) == null) {
                return null;
            }
            if (table.get(index).key.equals(key)) {
                return table.get(index).value;
            }
            j++;
        }
    }

    @Override
    public boolean containsKey(K key) {
        return get(key) != null;
    }

    @Override
    public void remove(K key) {
        if (key == null) {
            return;
        }

        int hashCode = key.hashCode();
        int index = hashCode % capacity;

        if (table.get(index) != null && table.get(index).key.equals(key)) {
            table.get(index).isDeleted = true;
            size--;
            return;
        }

        int j = 0;
        while (true) {
            index = (hashCode + j * j) % capacity;
            if (table.get(index) == null) {
```

```
                return;
            }
            if (table.get(index).key.equals(key)) {
                table.get(index).isDeleted = true;
                size--;
                return;
            }
            j++;
        }
    }

    @Override
    public int size() {
        return size;
    }

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

    private void rehash() {
        ArrayList<Entry<K, V>> oldTable = table;
        capacity *= 2;
        table = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) {
            table.add(null);
        }
        size = 0;

        for (Entry<K, V> entry : oldTable) {
            if (entry != null && !entry.isDeleted) {
                put(entry.key, entry.value);
            }
        }
    }

    public static void main(String[] args) {
        MyOpenAddressingMap<Integer, String> map = new MyOpenAddressingMap<>(4);
        map.put(3, "3");
        map.put(4, "4");
        map.put(45, "45");
        map.put(21, "21");
        map.put(92, "92");
        map.put(12, "12");

        System.out.println("Map size: " + map.size());
        System.out.println("Contains key 3: " + map.containsKey(3));
        System.out.println("Value for key 45: " + map.get(45));
        map.remove(45);
        System.out.println("After removal, contains key 45: " + map.containsKey(45));
    }
}
```