

数据结构

数据结构是以某种形式将数据组织在一起的集合，它不仅存储数据，还支持访问和处理数据的操作。算法是为求解一个问题需要遵循的、被清楚指定的简单指令的集合。下面是自己整理的常用数据结构与算法相关内容，如有错误，欢迎指出。

为了便于描述，文中涉及到的代码部分都是用 Java 语言编写的，其实 Java 本身对常见的几种数据结构，线性表、栈、队列等都提供了较好的实现，就是我们经常用到的 Java 集合框架，有需要的可以阅读这篇文章。[Java - 集合框架完全解析](#)

一、线性表

1. 数组实现

2. 链表

二、栈与队列

三、树与二叉树

1. 树

2. 二叉树基本概念

3. 二叉查找树

4. 平衡二叉树

5. 红黑树

四、图

五、总结

一、线性表

线性表是最常用且最简单的一种数据结构，它是 n 个数据元素的有限序列。

实现线性表的方式一般有两种，一种是使用数组存储线性表的元素，即用一组连续的存储单元依次存储线性表的数据元素。另一种是使用链表存储线性表的元素，即用一组任意的存储单元存储线性表的数据元素（存储单元可以是连续的，也可以是不连续的）。

数组实现

数组是一种大小固定的数据结构，对线性表的所有操作都可以通过数组来实现。虽然数组一旦创建之后，它的大小就无法改变了，但是当数组不能再存储线性表中的新元素时，我们可以创建一个新的大的数组来替换当前数组。这样就可以使用数组实现动态的数据结构。

代码 1 创建一个更大的数组来替换当前数组

```
int[] oldArray = new int[10];

int[] newArray = new int[20];

for (int i = 0; i < oldArray.length; i++) {
    newArray[i] = oldArray[i];
}
```

```

// 也可以使用 System.arraycopy 方法来实现数组间的复制
// System.arraycopy(oldArray, 0, newArray, 0, oldArray.length);

oldArray = newArray;
代码 2 在数组位置 index 上添加元素 e
//oldArray 表示当前存储元素的数组
//size 表示当前元素个数
public void add(int index, int e) {

    if (index > size || index < 0) {
        System.out.println("位置不合法...");
    }

    //如果数组已经满了 就扩容
    if (size >= oldArray.length) {
        // 扩容函数可参考代码 1
    }

    for (int i = size - 1; i >= index; i--) {
        oldArray[i + 1] = oldArray[i];
    }

    //将数组 elementData 从位置 index 的所有元素往后移一位
    // System.arraycopy(oldArray, index, oldArray, index + 1,size - index);

    oldArray[index] = e;

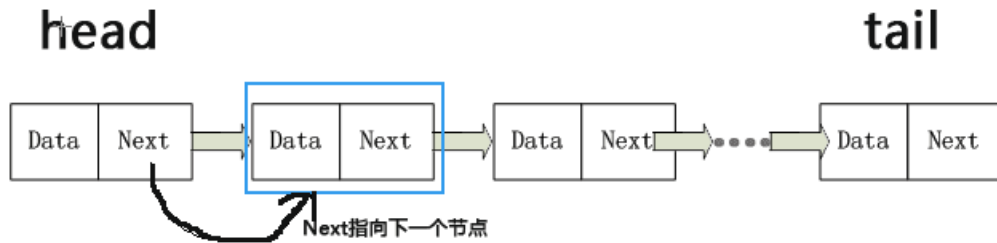
    size++;
}

```

上面简单写出了数组实现线性表的两个典型函数，具体我们可以参考 Java 里面的 ArrayList 集合类的源码。数组实现的线性表优点在于可以通过下标来访问或者修改元素，比较高效，主要缺点在于插入和删除的花费开销较大，比如当在第一个位置前插入一个元素，那么首先要把所有的元素往后移动一个位置。为了提高在任意位置添加或者删除元素的效率，可以采用链式结构来实现线性表。

链表

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列节点组成，这些节点不必在内存中相连。每个节点由数据部分 Data 和链部分 Next，Next 指向下一个节点，这样当添加或者删除时，只需要改变相关节点的 Next 的指向，效率很高。



单链表的结构

下面主要用代码来展示链表的一些基本操作，需要注意的是，这里主要是以单链表为例，暂时不考虑双链表和循环链表。

代码 3 链表的节点

```
class Node<E> {
    E item;
    Node<E> next;

    //构造函数
    Node(E element) {
        this.item = element;
        this.next = null;
    }
}
```

代码 4 定义好节点后，使用前一般是对头节点和尾节点进行初始化

//头节点和尾节点都为空 链表为空

```
Node<E> head = null;
```

```
Node<E> tail = null;
```

代码 5 空链表创建一个新节点

//创建一个新的节点 并让 head 指向此节点

```
head = new Node("nodedata1");
```

//让尾节点也指向此节点

```
tail = head;
```

代码 6 链表追加一个节点

//创建新节点 同时和最后一个节点连接起来

```
tail.next = new Node("node1data2");
```

//尾节点指向新的节点

```
tail = tail.next;
```

代码 7 顺序遍历链表

```
Node<String> current = head;
```

```
while (current != null) {
```

```
    System.out.println(current.item);
```

```
    current = current.next;
```

```
}
```

代码 8 倒序遍历链表

```
static void printListRev(Node<String> head) {  
//倒序遍历链表主要用了递归的思想  
    if (head != null) {  
        printListRev(head.next);  
        System.out.println(head.item);  
    }  
}
```

代码 单链表反转

//单链表反转 主要是逐一改变两个节点间的链接关系来完成

```
static Node<String> revList(Node<String> head) {  
  
    if (head == null) {  
        return null;  
    }  
  
    Node<String> nodeResult = null;  
  
    Node<String> nodePre = null;  
    Node<String> current = head;  
  
    while (current != null) {  
  
        Node<String> nodeNext = current.next;  
  
        if (nodeNext == null) {  
            nodeResult = current;  
        }  
  
        current.next = nodePre;  
        nodePre = current;  
        current = nodeNext;  
    }  
  
    return nodeResult;  
}
```

上面的几段代码主要展示了链表的几个基本操作，还有很多像获取指定元素，移除元素等操作大家可以自己完成，写这些代码的时候一定要理清节点之间关系，这样才不容易出错。

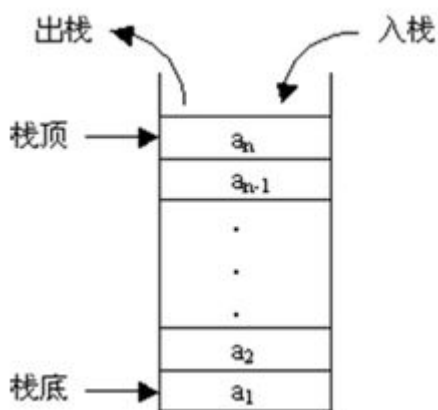
链表的实现还有其它的方式，常见的有循环单链表，双向链表，循环双向链表。循环单链表 主要是链表的最后一个节点指向第一个节点，整体构成一个链环。双向链表 主要是节点中包含两个指针部分，一个指向前驱元，一个指向后继元，JDK 中 `LinkedList` 集合类的实现就是双向链表。循环双向链表 是最后一个节点指向第一个节点。

二、栈与队列

栈和队列也是比较常见的数据结构，它们是比较特殊的线性表，因为对于栈来说，访问、插入和删除元素只能在栈顶进行，对于队列来说，元素只能从队列尾插入，从队列头访问和删除。

栈

栈是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫作栈顶，对栈的基本操作有 push(进栈)和 pop(出栈)，前者相当于插入，后者相当于删除最后一个元素。栈有时又叫作 LIFO(Last In First Out)表，即后进先出。



栈的模型

下面我们看一道经典题目，加深对栈的理解。

? 题目

有六个元素6,5,4,3,2,1 的顺序进栈,问下列哪一个不是合法的出栈序列? ()

A. 5 4 3 6 1 2 B. 4 5 3 2 1 6 C. 3 4 6 5 2 1 D. 2 3 4 1 5 6

关于栈的一道经典题目

上图中的答案是 C，其中的原理可以好好想一想。

因为栈也是一个表，所以任何实现表的方法都能实现栈。我们打开 JDK 中的类 Stack 的源码，可以看到它就是继承类 Vector 的。当然，Stack 是 Java2 前的容器类，现在我们可以使用 LinkedList 来进行栈的所有操作。

队列

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端 (front) 进行删除操作，而在表的后端 (rear) 进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操

作的端称为队尾，进行删除操作的端称为队头。



队列示意图

我们可以使用链表来实现队列，下面代码简单展示了利用 `LinkedList` 来实现队列类。

代码 9 简单实现队列类

```
public class MyQueue<E> {  
  
    private LinkedList<E> list = new LinkedList<>();  
  
    // 入队  
    public void enqueue(E e) {  
        list.addLast(e);  
    }  
  
    // 出队  
    public E dequeue() {  
        return list.removeFirst();  
    }  
}
```

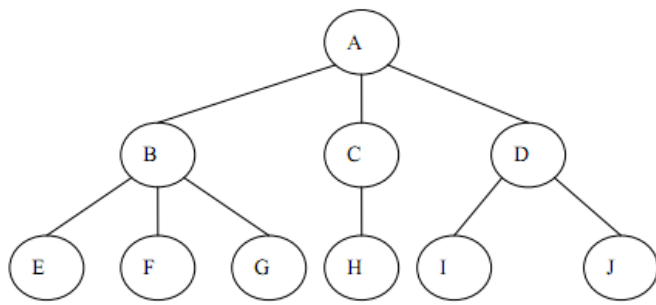
普通的队列是一种先进先出的数据结构，而优先队列中，元素都被赋予优先级。当访问元素的时候，具有最高优先级的元素最先被删除。优先队列在生活中的应用还是比较多的，比如医院的急症室为病人赋予优先级，具有最高优先级的病人最先得治疗。在 `Java` 集合框架中，类 `PriorityQueue` 就是优先队列的实现类，具体大家可以去阅读源码。

三、树与二叉树

树型结构是一类非常重要的非线性数据结构，其中以树和二叉树最为常用。在介绍二叉树之前，我们先简单了解一下树的相关内容。

树

树是由 n ($n \geq 1$) 个有限节点组成一个具有层次关系的集合。它具有以下特点：每个节点有零个或多个子节点；没有父节点的节点称为根节点；每一个非根节点有且只有一个父节点；除了根节点外，每个子节点可以分为多个不相交的子树。



树的结构

二叉树基本概念

定义

二叉树是每个节点最多有两棵子树的树结构。通常子树被称作“左子树”和“右子树”。二叉树常被用于实现二叉查找树和二叉堆。

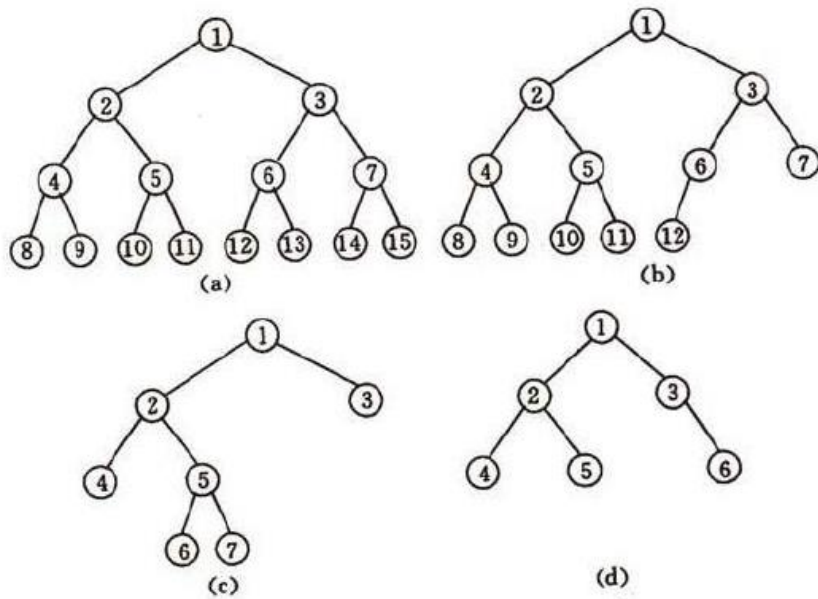
相关性质

二叉树的每个结点至多只有 2 棵子树(不存在度大于 2 的结点)，二叉树的子树有左右之分，次序不能颠倒。

二叉树的第 i 层至多有 2^{i-1} 个结点；深度为 k 的二叉树至多有 $2^k - 1$ 个结点。

一棵深度为 k ，且有 $2^k - 1$ 个结点的二叉树称之为 满二叉树 ；

深度为 k ，有 n 个结点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中，序号为 1 至 n 的节点对应时，称之为 完全二叉树 。



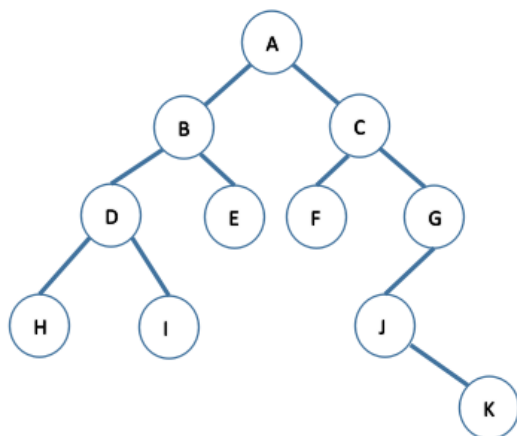
特殊形态的二叉树

(a) 满二叉树; (b) 完全二叉树; (c)和(d)非完全二叉树

三种遍历方法

在二叉树的一些应用中,常常要求在树中查找具有某种特征的节点,或者对树中全部节点进行某种处理,这就涉及到二叉树的遍历。二叉树主要是由3个基本单元组成,根节点、左子树和右子树。如果限定先左后右,那么根据这三个部分遍历的顺序不同,可以分为先序遍历、中序遍历和后续遍历三种。

(1) 先序遍历 若二叉树为空,则空操作,否则先访问根节点,再先序遍历左子树,最后先序遍历右子树。(2) 中序遍历 若二叉树为空,则空操作,否则先中序遍历左子树,再访问根节点,最后中序遍历右子树。(3) 后序遍历 若二叉树为空,则空操作,否则先后序遍历左子树访问根节点,再后序遍历右子树,最后访问根节点。



先序遍历	ABDHI ECFGJK
中序遍历	HDIB EAF CJKG
后序遍历	HID EBF KJGCA

给定二叉树写出三种遍历结果

树和二叉树的区别

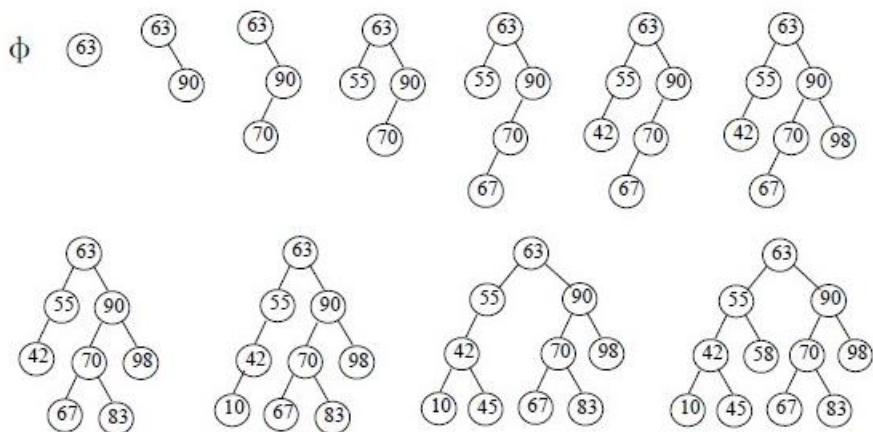
(1) 二叉树每个节点最多有 2 个子节点，树则无限制。(2) 二叉树中节点的子树分为左子树和右子树，即使某节点只有一棵子树，也要指明该子树是左子树还是右子树，即二叉树是有序的。(3) 树决不能为空，它至少有一个节点，而一棵二叉树可以是空的。

上面我们主要对二叉树的相关概念进行了介绍，下面我们将从二叉查找树开始，介绍二叉树的几种常见类型，同时将之前的理论部分用代码实现出来。

二叉查找树

定义

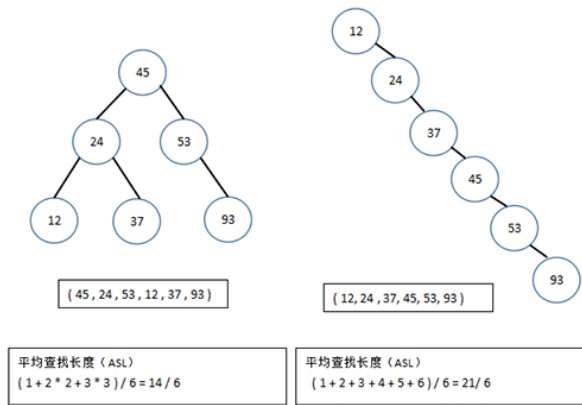
二叉查找树就是二叉排序树，也叫二叉搜索树。二叉查找树或者是一棵空树，或者是具有下列性质的二叉树：(1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；(2) 若右子树不空，则右子树上所有结点的值均大于它的根结点的值；(3) 左、右子树也分别为二叉排序树；(4) 没有键值相等的结点。



典型的二叉查找树的构建过程

性能分析

对于二叉查找树来说，当给定值相同但顺序不同时，所构建的二叉查找树形态是不同的，下面看一个例子。



不同形态平衡二叉树的 ASL 不同

可以看到，含有 n 个节点的二叉查找树的平均查找长度和树的形态有关。最坏情况下，当先后插入的关键字有序时，构成的二叉查找树蜕变为单支树，树的深度为 n ，其平均查找长度 $(n+1)/2$ (和顺序查找相同)，最好的情况是二叉查找树的形态和折半查找的判定树相同，其平均查找长度和 $\log_2(n)$ 成正比。平均情况下，二叉查找树的平均查找长度和 $\log n$ 是等数量级的，所以为了获得更好的性能，通常在二叉查找树的构建过程需要进行“平衡化处理”，之后我们将介绍平衡二叉树和红黑树，这些均可以使查找树的高度为 $O(\log(n))$ 。

代码 10 二叉树的节点

```
class TreeNode<E> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E e) {
        element = e;
    }
}
```

二叉查找树的三种遍历都可以直接用递归的方法来实现：

代码 12 先序遍历

```
protected void preorder(TreeNode<E> root) {
    if (root == null)
        return;

    System.out.println(root.element + " ");

    preorder(root.left);

    preorder(root.right);
}
```

代码 13 中序遍历

```
protected void inorder(TreeNode<E> root) {  
  
    if (root == null)  
        return;  
  
    inorder(root.left);  
  
    System.out.println(root.element + " ");  
  
    inorder(root.right);  
}
```

代码 14 后序遍历

```
protected void postorder(TreeNode<E> root) {  
  
    if (root == null)  
        return;  
  
    postorder(root.left);  
  
    postorder(root.right);  
  
    System.out.println(root.element + " ");  
}
```

代码 15 二叉查找树的简单实现

```
/**  
 * @author JackalTsc  
 */  
public class MyBinSearchTree<E extends Comparable<E>> {  
  
    // 根  
    private TreeNode<E> root;  
  
    // 默认构造函数  
    public MyBinSearchTree() {  
    }  
  
    // 二叉查找树的搜索  
    public boolean search(E e) {  
  
        TreeNode<E> current = root;  
  
        while (current != null) {
```

```

        if (e.compareTo(current.element) < 0) {
            current = current.left;
        } else if (e.compareTo(current.element) > 0) {
            current = current.right;
        } else {
            return true;
        }
    }

    return false;
}

// 二叉查找树的插入
public boolean insert(E e) {

    // 如果之前是空二叉树 插入的元素就作为根节点
    if (root == null) {
        root = createNewNode(e);
    } else {
        // 否则就从根节点开始遍历 直到找到合适的父节点
        TreeNode<E> parent = null;
        TreeNode<E> current = root;
        while (current != null) {
            if (e.compareTo(current.element) < 0) {
                parent = current;
                current = current.left;
            } else if (e.compareTo(current.element) > 0) {
                parent = current;
                current = current.right;
            } else {
                return false;
            }
        }
        // 插入
        if (e.compareTo(parent.element) < 0) {
            parent.left = createNewNode(e);
        } else {
            parent.right = createNewNode(e);
        }
    }
    return true;
}

// 创建新的节点

```

```

        protected TreeNode<E> createNewNode(E e) {
            return new TreeNode(e);
        }
    }
}

```

// 二叉树的节点

```

class TreeNode<E extends Comparable<E>> {

    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E e) {
        element = e;
    }
}

```

上面的代码 15 主要展示了一个自己实现的简单的二叉查找树，其中包括了几个常见的操作，当然更多的操作还是需要大家自己去完成。因为在二叉查找树中删除节点的操作比较复杂，所以下面我详细介绍一下这里。

二叉查找树中删除节点分析

要在二叉查找树中删除一个元素，首先需要定位包含该元素的节点，以及它的父节点。假设 `current` 指向二叉查找树中包含该元素的节点，而 `parent` 指向 `current` 节点的父节点，`current` 节点可能是 `parent` 节点的左孩子，也可能是右孩子。这里需要考虑两种情况：

`current` 节点没有左孩子，那么只需要将 `parent` 节点和 `current` 节点的右孩子相连。

`current` 节点有一个左孩子，假设 `rightMost` 指向包含 `current` 节点的左子树中最大元素的节点，而 `parentOfRightMost` 指向 `rightMost` 节点的父节点。那么先使用 `rightMost` 节点中的元素值替换 `current` 节点中的元素值，将 `parentOfRightMost` 节点和 `rightMost` 节点的左孩子相连，然后删除 `rightMost` 节点。

// 二叉搜索树删除节点

```

public boolean delete(E e) {

    TreeNode<E> parent = null;
    TreeNode<E> current = root;

    // 找到要删除的节点的位置
    while (current != null) {
        if (e.compareTo(current.element) < 0) {
            parent = current;
            current = current.left;
        } else if (e.compareTo(current.element) > 0) {
            parent = current;

```

```

        current = current.right;
    } else {
        break;
    }
}

// 没找到要删除的节点
if (current == null) {
    return false;
}

// 考虑第一种情况
if (current.left == null) {
    if (parent == null) {
        root = current.right;
    } else {
        if (e.compareTo(parent.element) < 0) {
            parent.left = current.right;
        } else {
            parent.right = current.right;
        }
    }
} else { // 考虑第二种情况
    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;
    // 找到左子树中最大的元素节点
    while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right;
    }

    // 替换
    current.element = rightMost.element;

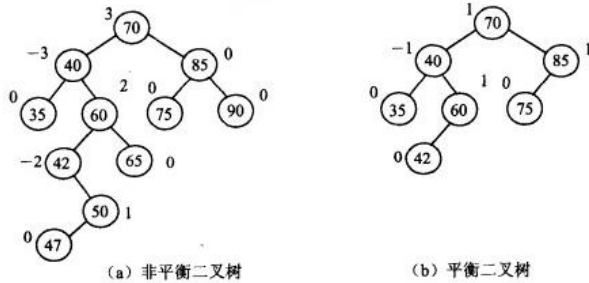
    // parentOfRightMost 和 rightMost 左孩子相连
    if (parentOfRightMost.right == rightMost) {
        parentOfRightMost.right = rightMost.left;
    } else {
        parentOfRightMost.left = rightMost.left;
    }
}

return true;
}

```

平衡二叉树

平衡二叉树又称 AVL 树，它或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过 1。



平衡二叉树

AVL 树是最先发明的自平衡二叉查找树算法。在 AVL 中任何节点的两个儿子子树的高度最大差别为 1，所以它也被称为高度平衡树， n 个结点的 AVL 树最大深度约 $1.44\log_2 n$ 。查找、插入和删除在平均和最坏情况下都是 $O(\log n)$ 。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

红黑树

红黑树是平衡二叉树的一种，它保证在最坏情况下基本动态集合操作的事件复杂度为 $O(\log n)$ 。红黑树和平衡二叉树区别如下：(1) 红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。(2) 平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。点击查看更多