**红黑树算法及优化思路**

**摘要**: 在《数据结构》的“搜索结构”一章中，我们学习了AVL树，这是一种搜索结构。相较于普通的二叉搜索树，AVL树通过独特的平衡化旋转操作确保了在极端情况下时间复杂度仍然保持在O(()。然而，这种平衡化旋转操作的频繁执行会消耗大量性能。因此，我们开始探讨另一种更高效的替代数据结构——红黑树。本文将对红黑树进行一系列研究和探讨，探索其作为一种数据结构的特性和优势。

**关键词:** 搜索结构 红黑树 算法优化

**Research and improvement of Red-Black tree**

**Abstract**: I In the "Search Structures" chapter of "Data Structures," we studied the AVL tree as a search structure. This data structure, in comparison to a regular binary search tree, ensures a time complexity of O() even in extreme scenarios due to its unique balancing rotation operations. However, the frequent execution of these balancing rotations consumes a significant amount of performance. Therefore, we contemplate an alternative, more efficient data structure to replace the AVL tree—the red-black tree. This article will undertake a series of studies on the red-black tree data structure.

**Key Words**: Search Structure; Red-Black tree; Algorithm optimization

—————————————

**1 引 言**

在本学期的数据结构课程中，笔者深入学习了关于搜索结构的章节。所学搜索结构主要涵盖使用顺序表或树形结构进行存储。在树形结构方面，重点学习了二叉搜索树及其优化版本AVL树。

AVL树相较于普通的二叉搜索树，以其维持左右子树高度差的绝对值不超过1（高度平衡）的特性而脱颖而出，确保了更为稳定的时间复杂度O()。然而，为了保持这种高度平衡特性，AVL树需要进行大量的平衡和旋转操作。在实际需要大量数据访问的情况下，这对硬件性能构成了巨大挑战。

因此，我们选择了性能更为优越的数据结构——红黑树进行研究。这一探究旨在了解其实际应用场景，并寻找可能存在优化空间的领域。

**2 红黑树基本介绍**

**2.1 红黑树的背景和相关概念**

红黑树(Red Black Tree)是一种自平衡二叉查找树，是在[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA?fromModule=lemma_inlink" \t "_blank)科学中用到的一种[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450?fromModule=lemma_inlink)，其是在1972年由[Rudolf Bayer](https://baike.baidu.com/item/Rudolf%20Bayer/3014716?fromModule=lemma_inlink" \t "_blank)发明的，在当时被称为平衡二叉B树(symmetric binary B-trees）。后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的“红黑树”[1] 。

红黑树是一种特化的AVL树，与AVL树相同其也是在进行插入和删除操作时进行一系列特殊操作来保证二叉查找树的“平衡”，目的自然也是为了其在查找使用时可以获得较高的查找性能。

相较于AVL树高度平衡的特性而言，红黑树所拥有的各种性质显然更加复杂，其包括：

1. 顾名思义，红黑树的结点都是红色或者黑色的。
2. 红黑树的根结点是黑色的。
3. 所有的叶子结点包括NULL结点都是黑色的。
4. 每个红色结点的两个子结点一定是黑色的，也就是说从每个叶子到根的所有路径上都不能有两个连续的红色结点。
5. 从任意一个结点到其每个叶子结点的所有路径都应该包含相同数目的黑色结点。[2]

由以上红黑树的性质也可推出红黑树的如下定理：

1. 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。
2. 红黑树的树高(height)不大于两倍的红黑树的黑深度(black\_depth)，即height <=2\* black\_depth。
3. 一棵拥有n个内部结点(不包括叶子结点) 的红黑树的树高height<=2\*。

由于第一条定理的存在，导致了红黑树本身是大致平衡的，插入、删除、查询等操作的最坏情况都是和红黑树的高度成比例,而在高度理论上限的范围内红黑树即便遇到最坏情况也是高效的。[2]

因其复杂的性质，导致红黑树在实现时的具体代码也更为复杂。但其在实践的极端情况下进行查找的时间复杂度也能保持在O()。(n指树的结点数量)，但其避免了AVL树在实际应用频繁存取时进行大量平衡化旋转所给性能带来的不必要损失。因此，笔者认为，在某种程度上，使用红黑树作为搜索结构是要优于AVL树的。

**2.2 红黑树的相关操作**

作为一种树形搜索结构，红黑树首先拥有最基本的插入、删除、查询结点等功能，下面笔者将从这几方面对红黑树的相关操作进行介绍：

###### 2.2.1 红黑树的查询结点操作

因为红黑树拥有最基本的二叉搜索树的性质，即父结点的键值(key)一定大于其左孩子(若其左孩子存在)而小于其右孩子(若其右孩子存在)。因此查询操作只需要从根结点开始根据键值来选择：

1. 若要查找的键值等于当前结点的键值，则返回当前键值对应的数据，查找成功。
2. 若要查找的键值小于当前结点的键值，则将用来查找的指向当前结点的指针指向当前结点的左孩子，而后继续迭代。如果当前结点没有左孩子，则说明当前红黑树中不含有此数据，查找失败。
3. 若要查找的键值大于当前结点的键值，则将用来查找的指向当前结点的指针指向当前结点的右孩子，而后继续迭代。如果当前结点没有右孩子，则说明当前红黑树中不含有此数据，查找失败。

###### 2.2.2 红黑树的插入结点操作

需要先根据红黑树查询结点的操作，对插入的结点的键值(key)进行操作。如果：

1. 找到了与该键值(key)相同的结点，则更新该结点的数据。
2. 未找到与该键值(key)相同的结点，则需要返回最后指针停留的位置(即所要插入的结点位置)，并申请新的空间插入新结点。

###### 2.2.3 红黑树的插入结点后的操作

为保持红黑树的性质并尽可能减少插入后的操作，所以在插入新结点的时候，默认插入的是红色结点。插入后，需要根据红黑树的具体情况进行如下循环判断与操作：

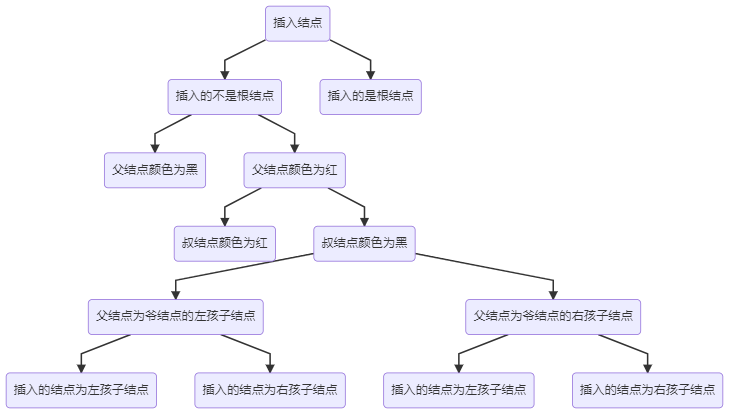
1. 若插入的红结点是红黑树的根结点，则为了满足红黑树的第二条性质，直接将本结点置为黑色，然后退出循环即可。
2. 若新插入的红结点的父结点是黑色结点，则不需要进行其他的特殊处理，直接退出循环即可。

若不满足以上两条，则说明新结点为红且不为根结点，其父结点也是红结点。那么将进行以下的条件判断：

1. 找到插入的新结点的叔结点(即其父结点的兄弟结点)，若存在且为红结点，则将新结点的叔结点、父结点以及爷结点(父结点的父结点)全部进行反色操作(黑色变红色，红色变黑色)，然后将指向新添加结点的指针指向其爷结点，并继续进行循环迭代。
2. 若其叔结点不存在或者存在且为黑结点，则将进行新的判断。
   1. 若父结点是爷结点的左孩子，则：
      1. 若当前结点是父结点的左孩子，则需要对当前结点的爷结点进行右旋操作，并将右旋后当前结点的父结点的颜色重新着色为黑色；将右旋后当前结点的爷结点的颜色重新着色为红色。
      2. 若当前结点是父结点的右孩子，则需要对当前结点的父结点进行左旋操作，并对左旋后的当前结点的爷结点进行右旋操作。再将右旋后的当前结点的爷结点的颜色重新着色为红色，将当前结点的颜色重新着色为黑色。
   2. 若父结点是爷结点的右孩子，则：
      1. 若当前结点是父结点的左孩子，则需要对当前结点的父结点进行右旋操作，并对右旋后的当前结点的爷结点进行左旋操作。再将左旋后的当前结点的爷结点的颜色重新着色为红色，将当前结点的颜色重新着色为黑色。
      2. 若当前结点是父结点的右孩子，则需要对当前结点的爷结点进行左旋操作，并将左旋后当前结点的父结点的颜色重新着色为黑色；将右旋后当前结点的爷结点的颜色重新着色为红色。

若当前结点的叔结点为黑结点或者不存在，则按实际情况执行上述判断与操作，并在结束操作直接跳出循环迭代。

插入结点的流程图如下：



###### 2.2.4 红黑树的删除结点操作

需要先根据红黑树查询结点的操作，对插入的结点的键值(key)进行操作。如果：

1. 未找到与该键值(key)相同的结点，则抛出异常提示，该红黑树中不存在该键值(key)的结点。
2. 若找到了该键值(key)的结点，则记录下该结点，并使用替代法进行循环迭代，循环的条件是当前结点至少拥有一个孩子结点。
   1. 若当前结点有右孩子，则使用一个临时变量替代结点记录下来当前结点的右孩子，并通过循环，判断替代结点的左孩子是否存在：
      1. 若存在，则令其指向其左孩子。
      2. 若不存在，则跳出循环。

定义一个结构体来存储结点的信息，其包含左孩子、右孩子、父结点的指针以及结点颜色。用其暂时存储当前结点和替代结点的信息。

然后将两个结点的颜色进行交换，而后进行一系列判断。如果当前结点不是根结点，则令当前结点的父结点将用以指向当前结点的指针改为指向替代结点；反之，则令替代结点成为根结点。

如果当前结点存在左孩子，则令左孩子的父结点指针指向替代结点。如果替代结点存在右孩子，则令替代结点的右孩子的父结点指针指向当前结点。

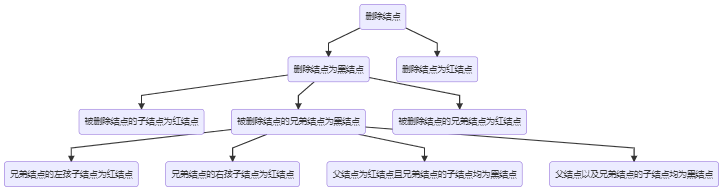
完成上述操作后，当前结点的左孩子一定是nullptr。然后令当前结点的右孩子指针指向替代结点的右孩子，并令当前结点的父结点指针指向替代结点的父结点。再令替代结点的左孩子、右孩子以及父结点指针分别指向当前结点的左孩子、右孩子以及父结点。再对当前结点的右孩子进行特殊处理，若其不指向替代结点，则令当前结点的父结点指针指向替代结点并令替代结点的父结点的左孩子指向当前结点。进行完上述所有的判断后继续进行循环迭代。

* 1. 若当前结点有左孩子但没有右孩子，则也需要进行一遍a)中操作，但所有涉及到左右孩子的操作全部相反，即a)中要求操作左孩子，则b)中需要对右孩子进行操作，而后继续进行循环迭代。

当不满足循环条件而跳出循环时，实际上被删除的目标结点已经变成了叶子结点，不再拥有子树，接着判断被删除的结点：

1. 若删除的是根结点，则直接删除该结点即可。
2. 若删除的是红色的叶结点，则需要将其父结点对应的孩子指针置为nullptr，然后删除即可。
3. 若删除的是黑色的叶结点，则在删除该结点后进行额外的操作。若其父结点是红色的，那么其兄弟结点一定是黑色的，此时需要对兄弟做旋转，再对父结点做旋转。父结点设为黑色，兄弟结点设为红色。具体相关操作时需要进行的特殊判断较多，这里不再赘述。

删除结点的流程图如下：



**3 红黑树的主要应用**

**3.1 几种树形数据结构的对比**

在涉及几种常见的树形结构时，二叉搜索树被认为是一种简单、易于实现、动态性能良好的可选结构。然而，在极端情况下，即按顺序插入有序数据时，二叉搜索树可能会退化成一个线性链表，导致查找时间复杂度从O( )降至O(n)。因此，在实际应用中，很少直接使用最基本的二叉搜索树作为数据存储结构。

相较之下，AVL树，即平衡二叉树，以其高度平衡的特性（左右子树高度差的绝对值不超过1）及左右子树均为AVL树的属性，在极端环境下依然能够保持O( )的查找效率。然而，由于在频繁存取数据时需要进行大量平衡化旋转操作，这使得在大规模动态存储操作的实际应用场景中，AVL树的性能可能受到影响。

B树和B+树被认为是适用于磁盘的数据结构，由于其树形宽而高度相对较低，在查找特定数据时通常只需访问较少的节点。因此，在数据量较大、外存占主导地位的情况下，B树由于较少的磁盘读取次数而具有更快的速度。然而，由于允许节点容纳多个关键字，导致节点定义不统一，从而降低了在节点中的查找效率。

红黑树是一种自平衡的二叉搜索树，引入了“颜色”概念，目的是简化红黑树的平衡条件。在红黑树中，只要求黑节点平衡，实际上以黑色高度作为AVL树“平衡因子”的替代。这种弱化带来的好处是减少了树形调整操作的次数，使得红黑树在大规模动态存储操作中的效率明显优于AVL树，尽管在查找效率上稍逊于高度平衡的AVL树。

一般情况下，红黑树的查找效率高于B树，但由于红黑树的树形较高，使得在一些特殊情况下，其动态存储的效率低于B树。这种权衡是数据查找和存取之间相互矛盾的结果，无法找到完美的解决方案，只能在不同情况下尽量选择更合适、更高效的数据结构来解决问题。

目前，基于红黑树的优越性质已广泛应用于实际生活场景，包括Linux的进程管理、内存管理、数据库数据管理、虚拟内存跟踪以及设备驱动等领域。甚至C++的一些STL，如map、multimap等，也采用了红黑树的变体。

**3.2 红黑树在Linux非实时任务调度中的应用**

自Linux内核2.4.10版本开始，红黑树这一数据结构在进程管理、内存管理、设备驱动以及虚拟内存的跟踪上得到广泛应用[4]。

在Linux内核版本超过2.6.24后，引入了新的调度程序CFS。在CFS中，所有非实时可运行进程都被挂载在一棵红黑树上，以虚拟运行时间为键值。通过这棵红黑树，系统可以更高效、更公平地调度所有任务进程。相比于先前的调度机制，CFS废弃了active/expired数组和动态计算优先级。它选择下一个任务时，采用基于时间计算键值的红黑树，根据所有任务占用CPU时间的状态来确定调度任务的优先级，并放弃了跟踪任务的睡眠时间以及区分其是否是交互任务。

在CFS的调度机制下，每个进程都被赋予一个虚拟运行时间。随着进程的运行，其虚拟运行时间增加。具有较高优先级的进程，相较于其他进程，其虚拟运行时间增长较慢。调度时总是优先选择虚拟运行时间较少的进程进行运行，以确保每个进程都有运行机会，实现所谓的“公平”。在一定时间内，优先级较高的进程实际运行时间也会长于优先级较低的进程。

在进程调度过程中，系统会根据键值的大小将准备就绪即将开始运行的任务进程全部插入红黑树中，同时红黑树内部完成一系列排序与平衡化操作。每次对任务进程进行调度时，调度器都会优先选择红黑树最左边的叶子节点进行处理，时间复杂度大约为O()[2] 。

**3.3 红黑树在Linux虚拟内存中的应用**

在32位的Linux内核中虚拟地址空间划分0~3G为用户空间，而3~4G为内核空间。因此对于每个任务进程而言，其可以使用大约4GB的虚拟空间。同时，为了更好表示进程所用的虚拟空间，Linux定义了虚拟存储区域(VMA)，每个VMA都是某个进程的一段连续虚拟空间，其中的单元依旧拥有相同的特征，通过对于线性链表的使用，所有的虚拟区域都按照地址进行排序。但是，如果发生了缺页中断的时候，搜索VMA到指定区域，则需要频繁地进行操作，因此，Linux选择了在动态存取方面具有很大优势的红黑树来提高效率。

其中使用mm\_struct用来描述一个进程的虚拟空间，定义如下[2] ：

struct mm\_strcut   
{   
 //指向若干VMA组成的链表   
 struct vm\_area\_struct\* mmap;   
 //指向红黑树   
 struct rb\_root mm\_rb;  
 //VMA的数量   
 int map\_count;  
 ...   
};

这里成员mm\_rb指向红黑树的根结点，然后该进程的所有虚拟空间块都会以起始的虚拟地址为键值(key)插入红黑树中，并且以后新申请的所有虚拟空间都会插入到这棵红黑树中，删除时，也是根据键值(key)删除树上对应的结点。而由于此处对红黑树的应用，才能够使得VMA的查找性能由O(n)提升到了O()，大大提升了查找数据的效率。

**3.4 红黑树在堆内存泄漏动态检测的应用**

###### 3.4.1 堆内存分配动态检测的基本原理

对于一般的程序，程序员在写代码的时候会频繁大量的使用new与delete运算符来进行堆内存的分配和释放，而实际上，这些操作都存在着较大的危险性，一旦程序员有所疏忽，常常会出现内存块的管理错误，从而容易导致内存的分配错误或者泄露[5]。

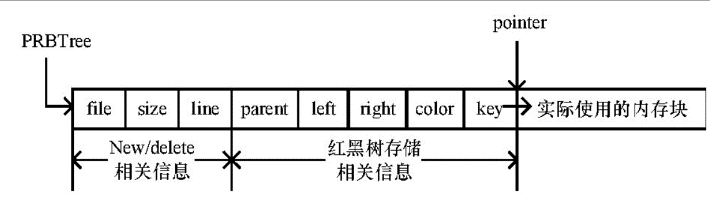
C++的编译器在对于new和delete运算符时调用的是其内部的new operate以及delete operate[6] 。先使用new operate分配足够的内存，然后再使用对应的数据类型的构造函数来对内存完成初始化。

但是，一般来说，new operate为自定义类型的对象分配内存的时候，实际分配到的内存要大于对象所需的实际内存。这部分多出来的内存空间，将用以记录该内存的相关信息(cookie技术)，实现策略随编译器的不同而不同。例如，g++则在所分配的内存的前4 Byte存储相关的信息，而后面的内存来存储相关的信息。在使用delete operate进行内存释放操作的时候，delete operate则根据前4 Byte的信息来对内存块完成正确的释放。

因此，如果想要动态检测有关的堆内存块的泄漏问题，就需要对new、delete、new[]、delete[]等四个运算符进行重载，并在重载时截获所需检测的内存操作信息。

###### 3.4.2 红黑树在对于内存检测的实现

使用红黑树来对每一个内存块的相关信息进行存储，这样可用于对每个内存块的分配与释放的跟踪记录。其中，键值(key)对应于内存块的地址，使用new运算符新分配一块堆内存后，其组织结构如下图所示。

 红黑树以及new/delete操作的相关信息作为附加信息附着于每个实际使用的内存块上(cookie)。PRBTree指针则指向每个用于保存附加信息的内存块，pointer指针指向实际使用的内存块。

###### 3.4.3 分析

在实际应用中，对分配的内存块，往往使用自管理的方式，因此使用哪种存储结构来进行内存块信息的存储，将对堆内存泄漏检测算法的实现的性能产生巨大的影响。

这里不妨假设分配的总内存块数目是N，而泄露的部分数目是M，显而易见，在正常情况下，N>>M，但由于各种可能存在的失误，会出现M偏大，这里假设M达到了量级。分情况对以下几种数据结构进行讨论：

1. 链表

链表的插入操作的时间复杂度为O(1)~O(N),而删除操作以及查询操作的时间复杂度亦然，这里插入和删除的时间复杂度均取O(N)，当对于泄漏的内存块M进行查询检测的时候，时间复杂度为O(M),因此总的时间消耗如下：

NO(N)+(N-M)O(N)+O(M)

1. 哈希表

哈希表的插入与删除操作的最优时间复杂度均为O(1)，但因为存在哈希冲突(哈希函数计算出的地址被别的元素占用)，具有较大的不确定性。需要找到较为理想的哈希函数，这里不妨设哈希桶的大小为T，总的时间消耗大约为：

NO(1)+(N-M)O(1)+O(T)

但值得注意的是哈希桶大小的选取非常重要，当T>>N时，会导致大多数项为空项，对泄漏的内存块M进行查询检测的时候会出现O(T)>>O(N)的情况；但当T<<N的时候，会发生大量的哈希冲突，插入与删除的时候会花费更多的时间。

1. 红黑树

由上文所述，红黑树在插入、删除、查询等操作的时间复杂度大约都是O(),与红黑树自身的高度在同一数量级，同时避免了删除结点的时间复杂度为O(N)的情况，因此使用红黑树的总耗时大约为：

NO()+(N-M)O()+O()

对比以上数据结构的总耗时可知，使用链表来作为存储结构的总耗时明显大于红黑树，而且因为N与T的取值会随数据规模的而改变，具有很强的不确定性，且要尽可能地避免哈希冲突。因此，在实际使用，红黑树的效率仍旧优于哈希表。

基于以上，在Windows和Linux平台的代码自动检测中，可以使用基于红黑树的检测算法来完成对于由new/delete所引起的堆内存泄漏的动态检测。

**4 对于传统红黑树算法的改进思路**

**4.1 传统算法所存在的问题**

作为一种高效的数据结构，红黑树在实现上不可避免地面临一些困难和算法复杂性的问题。传统的红黑树插入和删除操作的算法主要有两种，分别是Tarjan的算法（以下简称T算法）和Guibas以及Sedgewick合作设计的算法（以下简称GS算法）。

T算法的基本思想是先插入元素，然后对树进行调整，以确保插入元素后树仍然保持红黑树的性质。然而，这种方法存在一定的缺陷：需要增加父节点指针以便处理回溯问题，从而增加算法的空间复杂度，导致空间内存的浪费。当数据量较大时，这种浪费变得不可忽视。

GS算法对T算法进行了一系列改进，其主要思想与T算法相反，采用了先调整后插入的策略。在插入和删除操作时，先自上而下处理相关子树，进行了预先的“平衡”化旋转以及对节点的重新着色操作，然后再进行节点的插入或删除操作，以避免插入和删除后自下而上的回溯操作。然而，实际上，GS算法也存在一定的问题。如果插入的元素已经存在，或者需要删除的元素不存在，那么在查询对应节点时所做的大量预平衡处理将是无用功。假设在使用红黑树时经常出现这种情况，将对CPU性能占用产生灾难性影响，导致程序性能下降。

鉴于上述两种算法的弊端，笔者认为可以对算法进行一些小的改进，因此提出以下改进思路。

**4.2 针对传统算法所存在的问题的改进思路**

针对T算法，其底向上回溯问题导致必须使用大量空间存储父节点信息。在这方面，笔者建议优化使用栈这一数据结构。由于路径回溯遵循“先进后出”原则，这与DFS的非递归版本相契合。因此，考虑使用栈来优化回溯过程。

然而，仅仅依赖栈存储也存在问题，上向下和下向上的交替访问使得栈的先进后出性质难以完成操作，频繁调用栈还会降低程序性能。为此，笔者建议在遍历路径时使用数组存储节点，并使用变量作为下标指示当前访问节点，以简化代码操作。

至于GS算法，主要问题在于过多无用预处理导致性能下降。笔者建议首先使用时间复杂度为O( )的查询操作，对目标节点进行查询，若存在再进行预平衡处理以便后续插入或删除操作。

然而，就个人观点而言，笔者认为T算法的改进较为重要，尤其在几乎不存在插入重复或删除为空操作的情况下。GS的改进算法可能进行大量无用查询操作，反而降低性能。

**4.3 现代红黑树在网络信息分析时提出的改进方案**

当采集到海量数据量随时间非定性变化的网络信息时，从用户的需求角度出发，用户只关心排名靠前的数据流。而对于此类数量巨大并且非定性的数据，需要设计一个更高效率的排序算法。

此时，由于存在着大量的数据，对整个数据流量进行排序的方案从时间复杂度和空间复杂度的角度上并不现实，因此这里需要一种能可以高效完成排序任务，同时也能够顺序输出结构并存放于后台数据库的解决方案，其节点结构定义如下[7] ：

struct HASHRBTREENODE\_st   
{   
 //红黑树的结点属性   
 struct RBTREENODE Treenode；   
 //哈希表中链接指针   
 struct HASHRBTREENODE\_st\* pNext;   
};

哈希红黑树具有红黑树的节点属性，同时采用哈希表中的链接指针。可以按照哈希表的方式进行查找，同时将查找结果有序输出。通过将哈希表和红黑树这两种数据结构有机融合的方式，使得查找搜索的时间复杂度小于红黑树的O()，并且避免了哈希表中存在的无序特性及红黑树查找速率不足的缺点[2] 。

**5 结束语**

红黑树作为一种近似平衡的二叉查找树，在频繁增删节点时能够维持高效的运行效率。相对于AVL树，它不仅表现出良好的统计性能，还具备快速插入和删除的特点。本文首先介绍了红黑树的基本概念性质及相关操作，随后对红黑树与原始的二叉搜索树、AVL树、B树等进行了效率比较和优缺点的分析。同时，针对二叉树在Linux非实时任务调度、Linux虚拟内存的使用、堆内存泄漏动态检测等基本应用进行了详细列举，并据此查阅资料对各种应用方式进行了解释。

随后，对传统的两种红黑树算法以及现代红黑树在网络信息分析的应用中存在的问题进行了分析，并尝试提出了笔者的改进思路。通过这篇论文，笔者期望能成为深入研究更多数据结构和算法的契机，不断提升专业能力。同时，希望将所学知识有朝一日应用于实际，服务于人民，为个人的价值提升贡献一份力量。

**参考文献**

1. [马国富,张涵.红黑树关键算法研究[J].电脑知识与技术](https://baike.baidu.com/reference/2413209/900bMQRkAjbsiIWeVpHBR-fPc5Xb5qs7zTPmaHdIaQhzUuS8nbHXscaba9gGCV_0j9_rtfrhGpcuNbGCtouoNwD_wrtRDVRIc_sfMgkTboSlp7u1-63LdZSHXqyXAQtoKb1UQNfBYDgGZ1QtqSlPGn6-IYvjEvy-Ln28oq3tmEKw4Ndvh24PRbD7YS6pAIL_yadJ4jG65TNI42OeTkmOrffQWFW7gRf--V1dtnfe58bL5ML_65uvRn3EAYcQpwSsQ2YrNdY03PZlfPbsWTzVvvys1WZmWAs)——中国知网．2011-07-30

Ma Guofu, Zhang Han. Research on Key Algorithms of Red and Black Trees [J]. Computer Knowledge and Technology - China National Knowledge Infrastructure. 2011-07-30

1. [马博韬,孙鹏,朱小勇.红黑树算法研究综述[J].网络新媒体技术](https://baike.baidu.com/reference/2413209/eab4j6A2-NOxZDBzieJTkylz6xLO6JztDx1FzxLuXBn-EM8TAynOhZIT28Yiaknd3WuJQO9ybrf30avKT4p1TAE5Z9wXldwbzf_Maqi7X5aE40-4OSRWpdvQv23UddOtQDSJeDgg2aEmv-ZAJrlp9HSOfT4AVRfbdu8LNW1ThbnRQwMWCyPsbmtH2262PCyHdZxsPfGvK7UNXua95fGLK2O6We2Ogp-f7YF3KtJ9Up9ozoeckRlRAuhPeKkhvpZboRazO6CI2Ii14iJdBGsp103IXe4mW_UeAUw7)——中国知网．2018-07-04

Ma Botao, Sun Peng, Zhu Xiaoyong. Research Review of Red and Black Tree Algorithm [J]. Network New Media Technology - China National Knowledge Infrastructure. 2018-07-04

1. [冯涛,杨耀辉.红黑树的性能分析及其在实时数据库中的应用[J].科技视界](https://baike.baidu.com/reference/2413209/eab4j6A2-NOxZDBzieJTkylz6xLO6JztDx1FzxLuXBn-EM8TAynOhZIT28Yiaknd3WuJQO9ybrf30avKT4p1TAE5Z9wXldwbzf_Maqi7X5aE40-4OSRWpdvQv23UddOtQDSJeDgg2aEmv-ZAJrlp9HSOfT4AVRfbdu8LNW1ThbnRQwMWCyPsbmtH2262PCyHdZxsPfGvK7UNXua95fGLK2O6We2Ogp-f7YF3KtJ9Up9ozoeckRlRAuhPeKkhvpZboRazO6CI2Ii14iJdBGsp103IXe4mW_UeAUw7)——中国知网．2012-11

Feng Tao, Yang Yaohui. Performance analysis of red and black trees and their application in real-time databases [J]. Science and Technology Perspective - China National Knowledge Infrastructure. 2012-11

1. [程科.嵌入式Linux设备驱动程序的设计与研究[D]](https://baike.baidu.com/reference/2413209/eab4j6A2-NOxZDBzieJTkylz6xLO6JztDx1FzxLuXBn-EM8TAynOhZIT28Yiaknd3WuJQO9ybrf30avKT4p1TAE5Z9wXldwbzf_Maqi7X5aE40-4OSRWpdvQv23UddOtQDSJeDgg2aEmv-ZAJrlp9HSOfT4AVRfbdu8LNW1ThbnRQwMWCyPsbmtH2262PCyHdZxsPfGvK7UNXua95fGLK2O6We2Ogp-f7YF3KtJ9Up9ozoeckRlRAuhPeKkhvpZboRazO6CI2Ii14iJdBGsp103IXe4mW_UeAUw7)．电子科技大学2007

Cheng Ke. Design and Research of Embedded Linux Device Drivers [D]. University of Electronic Science and Technology of China, 2007

1. [葛瑶,李晓风,孔德光.基于红黑树的堆内存泄漏动态检测技术](https://baike.baidu.com/reference/2413209/eab4j6A2-NOxZDBzieJTkylz6xLO6JztDx1FzxLuXBn-EM8TAynOhZIT28Yiaknd3WuJQO9ybrf30avKT4p1TAE5Z9wXldwbzf_Maqi7X5aE40-4OSRWpdvQv23UddOtQDSJeDgg2aEmv-ZAJrlp9HSOfT4AVRfbdu8LNW1ThbnRQwMWCyPsbmtH2262PCyHdZxsPfGvK7UNXua95fGLK2O6We2Ogp-f7YF3KtJ9Up9ozoeckRlRAuhPeKkhvpZboRazO6CI2Ii14iJdBGsp103IXe4mW_UeAUw7)．计算机工程 2008

Ge Yao, Li Xiaofeng, Kong Deguang. Dynamic detection technology for heap memory leakage based on red and black trees. Computer Engineering 2008

1. ISO/IEC 14882.Programming Languages-C++[S].1998
2. 周彩兰,张亚芳,郭凤玲.哈希红黑树算法在网络信息分析中的应用[J].软件导刊 2007(13)

Zhou Cailan, Zhang Yafang, Guo Fengling. The Application of Hash Red Black Tree Algorithm in Network Information Analysis [J]. Software Guide 2007 (13)