·经典评述·

红黑树算法研究综述

马博韬[1,2] 孙 鹏[1] 朱小勇[1]

([1] 中国科学院声学研究所 国家网络新媒体工程技术研究中心 北京 00190 [2] 中国科学院大学 北京 100049)

摘要: 针对内存数据管理中常用的红黑树算法开展研究,在介绍红黑树定义及特性的基础上,对比红黑树与二叉平衡树在插入删除及查找数据时的时间复杂度,对红黑树在各类节点颜色情况下插入删除操作进行了分类。同时,文章在 Linux 非实时任务调度、虚拟内存等应用场景下介绍红黑树的使用,最后基于红黑树存在的潜在不足,介绍了相关研究人员在特定场景下对其做出的改进优化。

关键词: 红黑树,AVL 树, 插入平衡,删除平衡

Overviews on Reseach of Red – Black Tree Algorithm

MA Botao[1,2], SUN Peng[1], ZHU Xiaoyong[1]

([1]National Network New Media Engineering Research Center, Chinese Academy of Sciences, Beijing, 100190, China,
[2]University of Chinese Academy of Science, Beijing, 100490, China)

Abstract: This paper studies the red – black tree algorithm which commonly used in memory data management. On the basis of introducing the definition and characteristics of red – black tree, it compares the time complexity of the red – black tree and binary balance tree when inserting and searching the data, and classifies the insertion and deletion operation of red – black tree under the circumstances of different nodes color. At the same time, the paper introduces the use of red – black tree in Linux non real – time task scheduling, virtual memory and other applications. Finally, based on the potential shortage of red – black tree, this paper introduces the improvement and optimization in the specific scene.

Keywords: Red – Black Tree, AVL Tree, insert – fixup, delete – fixup

1 引言

平衡二叉搜索树(Self – balancing binary search tree),又被称为 AVL 树,AVL 树是一棵空树或者由高度差绝对值不超过 1 的左右子树构成,并且要求左右两个子树都是平衡二叉树,可由 AVL 算法等方式实现。

AVL 算法是最先发明的自平衡二叉查找树算法,在 AVL 算法中任何节点的两个子树的高度差绝对值不超过 1,在查找、插入和删除在平均和最坏情况下均为 $O(\log_2 N)$。在频繁插入删除节点的场景下,AVL 算法由于要保持高度平衡的特性,需要进行频繁的平衡操作,因此导致效率的下降[1]。

作为对 AVL 树的一种改进,红黑树是一种自平衡二叉查找树,于 1972 年由 Rudolf Bayer 提出[2],最早命名为“对称二叉 B 树”,红黑树与 AVL 树的基本思想都是在进行插入和删除时通过保持树的平衡,以获得较高的查找性能。不同之处在于其左右子树高度差可能大于 1,因此对红黑树的平衡操作代价相对更低; 同

本文于 2018 – 06 – 05 收到。

时,红黑树具有较好的统计性能,查找复杂度为 O($\log_2$N) 。目前,基于拥有上述特性,红黑树已广泛应用于 Linux 的进程管理、内存管理,设备驱动及虚拟内存跟踪等一系列场景中。

本文主要就各种不同情形的红黑树结构插入及删除节点操作进行分类及讨论,以期能让使用者对其如何保持其特性的原理及实现方式有更加清晰的了解。在此基础上,本文就红黑树广泛使用的场景中抽取 3 个典型场景加以描述,在时间复杂度上分析相关研究人员选取红黑树数据结构的原因; 并且分析在特定环境下红黑树存在的不足以及可改进之处。

2　红黑树算法分析

2.1　红黑树定义

红黑树是一棵每个节点都带有颜色属性的二叉查找树,在每个节点的属性除了 1 个 key 和 3 个指针: parant、lchild、rchild 之外,还具有额外的 color 属性,要求每个节点为红色或黑色。在二叉查找树的一般要求之外,同时增加了如下的额外定义[3]:

性质 1: 节点是红色或黑色。

性质 2: 根节点是黑色。

性质 3: 每个叶节点(NIL 节点,空节点) 是黑色的。

性质 4: 每个红色节点的两个子节点都是黑色 (从每个叶子到根的所有路径上不能有两个连续的红色节点) 。

性质 5: 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

因此,红黑树也拥有以下的相关的定理[4]:

(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(h) 不大于两倍的红黑树的黑深度(bd) ,即 $h <= 2bd$。

(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 $h <= 2\log_2(n+1)$。

由于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,导致红黑树本身是大致平衡的。插入、删除、查找的最坏情况都和树的高度成比例,在高度理论上限的范围内红黑树遇到最坏情况时也是高效的。

2.2　红黑树的基础操作

由于红黑树每个节点均带有颜色属性,并且带有上述定义和相关定理,因此要求在进行插入删除操作时需要对原有节点进行旋转操作和节点的颜色属性改变,对红黑树的旋转操作以及对插入删除节点操作的情况加以分类。

2.2.1　红黑树的旋转操作

当对红黑树进行插入节点与删除节点操作时,由于插入节点或者删除节点之后,原红黑树节点之间颜色分布发生改变,即可能不满足红黑树的定义,为使改变后的树能在颜色上满足红黑树条件,需要进行旋转操作[5]。红黑树的旋转包括左旋及右旋:

如图 1 所示,A 为 X 节点左子树,B、C 分别为 Y 节点的左右子树。

对 X 节点进行左旋,则 X 成为 Y 节点的左节点,Y 的左子树 B 成为 X 节点的右子树。

对 Y 节点进行右旋,则 B 成为 Y 节点的左子树,Y 成为 X 节点的右子节点。

2.2.2　红黑树节点插入操作

红黑树插入节点步骤如下:

(1) 插入过程依据二叉查找树的查找方式,搜索到插入节点所在的某个叶子节点位置。

(2) 将新的插入节点插入在原树中一个已存在的空节点上,将插入节点颜色置为红色,同时增加两个空节点作为其子节点。

(3) 增加新节点后，其祖先节点的颜色根据红黑树的定义来确定，此时需要对树进行插入平衡操作，包括旋转或者改变节点颜色以保持红黑树的性质。

(4) 对于改变颜色的节点，需要向上继续对树进行修正，在最坏情况下，需要对从叶子节点到根节点上的所有路径处理，则插入的时间复杂度为 $O(\log_2 N)$。

红黑树插入节点的分类状态，如图 2 所示。

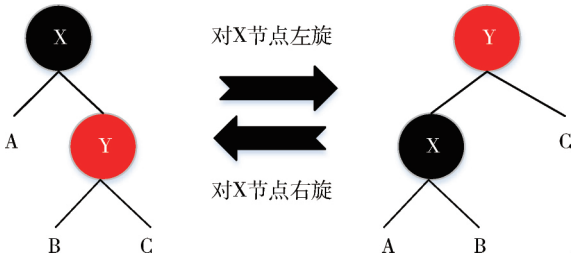

图 1 红黑树旋转操作图

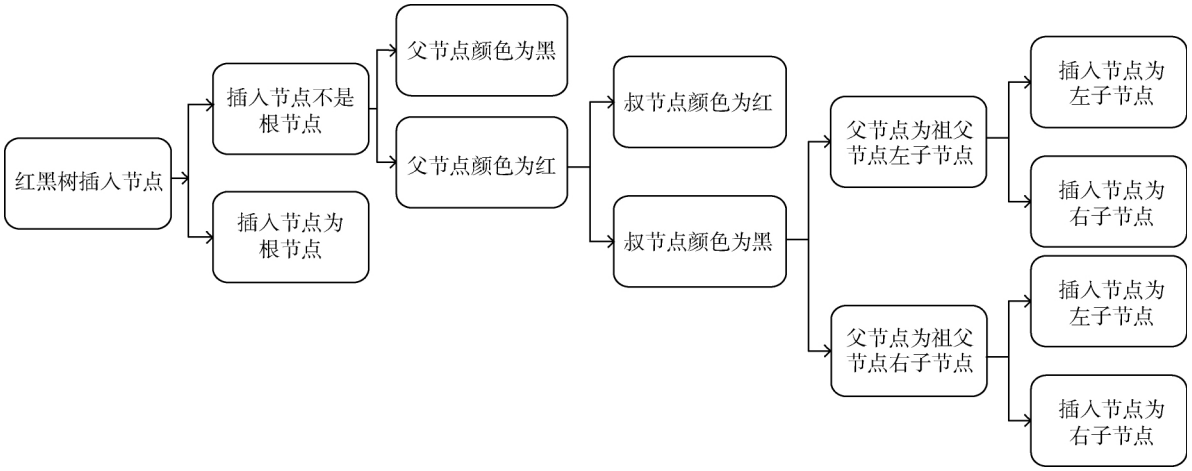

图 2 红黑树插入节点状态分类图

2.2.3 红黑树节点删除操作

红黑树的删除操作，需要对删除节点自身以及删除节点的父节点、子节点、兄弟节点颜色均加以分类。由于红黑树的定义中要求每个红色节点的两个子节点都是黑色，同时从每个叶子到根的所有路径上不能有两个连续的红色节点，因此若删除的节点为红色，则不需任何操作，红黑树的属性并未发生改变；但当删除节点为黑，则违背了红黑树定义，则需要 Delete - Fixup 进行修补，当节点删除后，需要其他节点在原位置替代该节点，当原删除节点为叶子节点时，则用空节点 NIL 进行替代。

因此，将删除节点操作分为以下几种情况考虑：

情况 1：删除节点颜色为红色。

情况 2：删除节点颜色为黑，同时存在子节点颜色为红。

情况 3：删除节点颜色为黑，被删除节点的兄弟节点为黑。

情况 4：删除节点颜色为黑，且被删除节点的兄弟节点为红。

对于每种情况下的分类如图 3 所示。

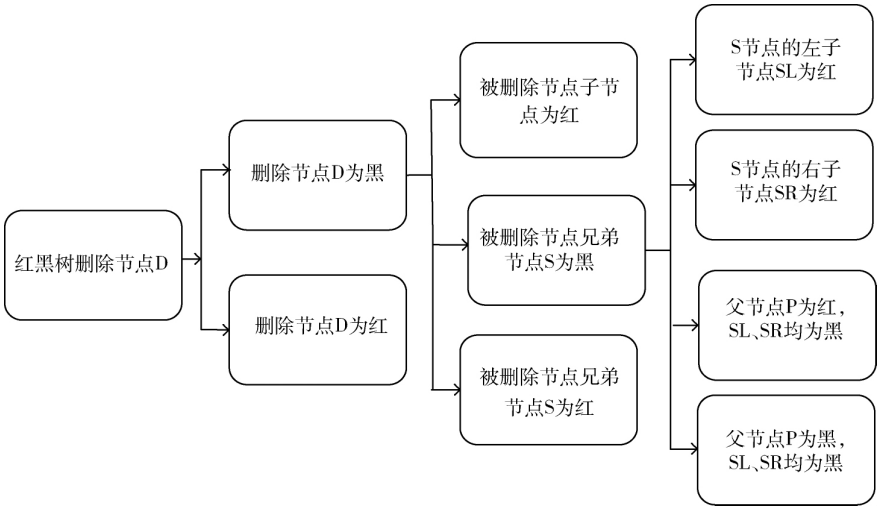

图 3 红黑树删除节点状态分类图

2.3　红黑树与 AVL 树比较

从效率角度出发,选用红黑树而不是 AVL 树的原因在于红黑树的高效率,为了保证绝对的平衡性,AVL 树需要进行更多的旋转及向根节点回溯以达到平衡。在插入节点时,平衡二叉树和红黑树都只需最多 2 次旋转操作,两者复杂度均为 O(1);当删除二叉平衡树节点时,为了保持树的平衡性,在最坏情况下需要维护从被删节点到根节点这条路径上所有节点的平衡性,因此需要旋转的复杂度为 $O(\log_2 N)$,而此时红黑树最多只需要 3 次旋转;在频繁插入删除情况下,仍能保证其统计性能[5]。在查找节点方面,由于红黑树是近似平衡的结构,因此查找效率上与 AVL 树差别并不大。

3　红黑树的场景应用分析

在如今大量应用场景中,红黑树已经取代原有的 AVL 树。例如可用红黑树提高连续属性数据流的分类挖掘效率[6]、提升实时数据寻址效率[7]、动态监测堆内存泄漏[8]、硬件加速[9];并且 Linux 内核从 2.4.10 开始,便将红黑树的数据结构应用于进程管理及内存管理,设备驱动及虚拟内存的跟踪上[10]。以下从三个场景介绍红黑树的应用。

3.1　红黑树在 Linux 非实时任务调度中的应用

Linux 的稳定内核版本在 2.6.24 之后,使用了新的调度程序 CFS,所有非实时可运行进程都以虚拟运行时间为 key 值挂在一棵红黑树上,以完成更公平高效地调度所有任务。CFS 弃用 active/expired 数组和动态计算优先级,不再跟踪任务的睡眠时间和区别是否交互任务,并且在调度中采用基于时间计算键值的红黑树来选取下一个任务,根据所有任务占用 CPU 时间的状态来确定调度任务优先级。

在 CFS 调度机制中,设定每个进程都有一个虚拟运行时间,进程运行一段时间后其虚拟运行时间会增加,优先级高的进程虚拟时间增加更慢,调度中总是选择虚拟运行时间少的进程进行调度运行,这样每个进程都有运行机会,在一定的时间内,优先级高的进程实际运行的累计时间较优先级低的进程更长。

在调度过程中,根据键值大小将准备就绪的任务插入红黑树叶子节点中,同时根据键值从小到大顺序自左向右排列。在每次调度时,调度器会选择红黑树最左边的叶子节点优先处理,操作时间复杂度为 $O(\log_2 N)$。

3.2　红黑树在 Linux 虚拟内存中的应用

32 位 Linux 内核虚拟地址空间划分 0 – 3G 为用户空间,3 – 4G 为内核空间,因此每个进程可以使用 4GB 的虚拟空间。同时,Linux 定义了虚拟存储区域(VMA)以便于更好表示进程所使用的虚拟空间,每个 VMA 是某个进程的一段连续虚拟空间,其中的单元具有相同的特征,所有的虚拟区域按照地址排序由指针链接为一个链表。当发生缺页中断时搜索 VMA 到指定区域时,则需要频繁操作,因此选用了红黑树以减少查找时间。

其中 mm_struct 用以描述一个进程的虚拟空间,定义如下:

```
struct mm_struct
{
    struct vm_area_struct *   mmap;          //指向若干 VMA 组成的链表
    struct rb_root mm_rb;                    //指向红黑树
    int map_count;                           //VMA 的个数
    …
}
```

成员 mm_rb 指向红黑树的根节点,该进程的所有虚拟空间块均以起始虚拟地址为 key 值挂在该红黑树上。该进程新申请的虚拟空间会插入到这棵树中,删除时则从树上摘除相应的节点。树中所有的 VMA 均由左指针指向相邻的低地址虚拟块,右指针指向相邻的高地址虚拟块。

由于红黑树的使用,能够使得 VMA 的查找性能由 O(N) 提升至 $O(\log_2 N)$,大大提升了查找效率。

3.3 红黑树在堆内存泄漏动态检测上的应用

堆内存泄漏是由于用户频繁创建、释放对象以及分配、释放内存块时在 C + + 语言中发生的场景。在检测是否存在堆内存泄漏时,可通过红黑树存储每个内存块的相关信息,用于追踪记录内存块的分配与释放,在应用中,通过关键字 key 对应于可用内存块的地址。使用 new 新分配一块堆内存,其新内存块组织结构如图 4 所示[8]。

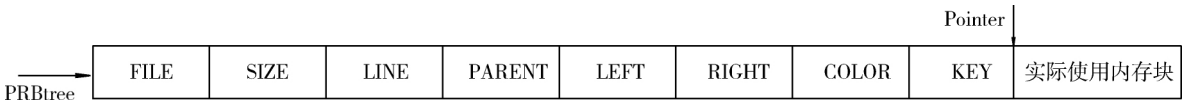

图 4 内存块组织结构

在应用中,对分配的内存块多为自管理方式,因此如何组织内存块的存储结构是影响堆内存泄漏动态监测算法性能的重要因素。设 N 为所有分配的堆内存块数目,M 为泄漏内存块的数目,在正常使用情况下,N 远大于 M。由于可能存在的失误,存在 M 数值偏大,假定 M 达到 $\log_2 N$ 量级。

讨论内存块为以下几种形式时的总时间消耗:

(1) 链表。

链表插入的最优时间复杂度为 O(1);删除内存块的最差情况时间复杂度为 O(N),平均时间为 O(N),搜索时间复杂度为 O(N),当搜索检测泄漏的内存块 M 时,时间复杂度为 O(M)。因此总的时间消耗为:

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

(2) 哈希表。

哈希表的插入删除时间复杂度最优均为 O(1),同时由于存在哈希冲突,存在不确定性。需要构建合适的哈希函数和哈希桶大小,设置哈希桶大小为 T。总时间平均消耗大致为:

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

(3) 红黑树。

由上文介绍可知,红黑树插入删除的时间复杂度均为 $O(\log_2(N))$,与红黑树高为同一量级,且查找的时间复杂度为 O(M),总耗时为:

$$NO(\log2(N)) + (N - M)O(\log2(N)) + O(M)$$

对比上述耗时可知,使用链表作为存储结构总耗时大于红黑树结构,同时,虽然 T、N 取值随程序规模大小改变,具有不确定性,但在实际使用中,需要兼顾哈希函数性质,T 取值较大,以避免出现较多哈希冲突,因此红黑树的时间效率优于哈希表。

因此在 Windows 和 Linux 平台代码的自动检测中,可使用基于红黑树的检测算法用以检测由 new/delete 引起的堆内存泄漏。

4 红黑树算法改进分析

尽管红黑树存在大量无需改动自身数据结构的使用场景,但并不意味着其不存在优化空间。在查询数据效率上,由于其时间复杂度为 $O(\log_2 N)$,在存在海量数据场景下,耗时较长,存在优化空间;同时,由于树状结构的制约,在并行计算时,需要设置大量锁以避免竞争。此时,相关研究人员提出了以下解决方案。

4.1 红黑树在网络信息分析中的改进

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,只需要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,由于存在大量数据,对整个数据流量进行排序的方案在开销上并不现实,因此需要一种既能高效排序、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[11]。因此在文献[11]中作者介绍哈希红黑树的使用方式,其节点结构定义如下:

```
typedef struct HASHRBTREENODE_st{
  RBTREENODE TreeNode;                  //红黑树的节点属性
  Struct HASHRBTREENODE_st*  pNext      //哈希表中链接指针
} HASHRBTREENODE;
```

根据不同的实际需求,以源 IP 等元素作为关键字进行数据流排序,成员包含有发送报文数 Pactets、发送数据信息字节数 Octets、会话次数 Sessions。采用如结构图 5 所示。

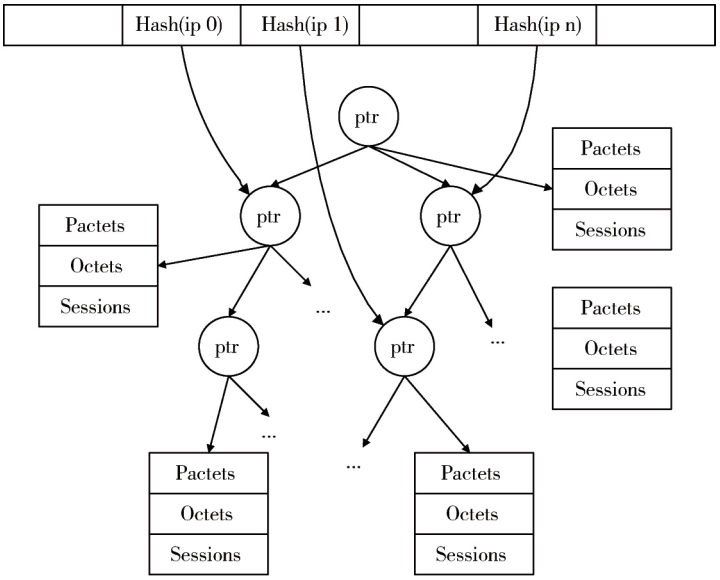

图 5　哈希红黑树存储结构

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。查找搜索的时间复杂度小于红黑树的 $O(\log_2 N)$,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。

4.2　在多线程处理时的缺陷

由于 AVL 树及红黑树在操作时存在平衡的过程,牵涉到众多节点,在更新时需要所有节点都参与操作。在多线程情况下,需要大量锁资源,尤其是在靠近根节点处存在大量资源竞争。

因此,William Pugh 在 1990 年提出了跳表的数据结构,这是一种可替代平衡树的数据结构。跳表通过依赖随机生成数以一定概率来保持数据的平衡,属于概率性的数据结构,其在实现上相对简单,在限定时间条件下能轻松实现[12]。跳表支持查找数据的时间复杂度为 $O((\log_{1/p} N)/p)$,式中 N 为跳表中元素的个数,p 为元素从第 i 层出现在第 i + 1 的概率。

在平均情况下,上述的概率 p 为常数,因此其插入、删除、查找数据时间复杂度为 $O(\log_2 N)$ 量级,其最坏情况下都为 $O(N)$,与红黑树和 AVL 树相比,效率相近,但其实现方式简单,维护更为容易,在更新的时候需要改动的地方很少,相对而言,跳表的操作更加局部性。

由于跳表的高效性及简单维护特性,因此在很多大数据系统中维护有序列表时选择了跳表结构,例如 LevelDB 在内存中暂存数据的结构 MemTable 是通过跳表实现的,同时,Redis 在 Sorted Set 数据结构时也采用了跳表结构[13]。

5　结束语

红黑树作为一种近似平衡的二叉查找树,可以在频繁增删节点时仍然保持高效的查找效率,相较于传统的 AVL 树,具有良好的统计性能以及具备快速插入删除的特点[14]。本文在对比红黑树与 AVL 树操作效率的基础上,介绍红黑树节点旋转的操作步骤,以及总结了红黑树进行插入删除操作的分类情况,同时列举部分红黑树的应用场景,以及在某些特定场景中红黑树存在的缺陷和基于此所作出的修改及选择权衡,希望能让红黑树使用者对其如何保持其特性的原理及实现方式有更加清晰的了解。

参　考　文　献

[1] R. Sedgewick and L. J. Guibas. A dichromatic framework for balanced trees [C] //19th Annual Symposium on Foundations of Computer Sciencep,1978: 8 - 21.

[2] Rudolf Bayer. Symmetric binary B – Trees: Data structure and maintenance algorithms [J]. Acta Informatica,1972,1(4): 290 – 306.
[3] Cormen, Thomas H, Leiserson, Charles E, Rivest, Ronald L, Stein, Clifford. Red – Black Trees. Introduction to Algorithms (second ed.) [M]. MIT Press,2001, 273 – 301.
[4] 唐自立. 红黑树的高度 [J]. 苏州大学学报,2006,22(3):33 – 36.
[5] Andersson A. Balanced Search Trees Made Simple [C] //The Workshop on Algorithms & Data Stractures. Springer – Verlag, 1993: 60 – 71.
[6] 陈煜,李玲娟. 基于红黑树的连续属性数据流快速决策树分类算法 [J]. 南京邮电大学学报(自然科学版),2017,37(02): 86 – 90.
[7] 安思成,吴克河,周欢,崔文超. 适用于广域测量系统的实时数据寻址红黑树算法 [J]. 华北电力大学学报,2016, 43(3): 95 – 110.
[8] 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 [J]. 计算机工程,2008,34(16):159 – 161.
[9] Carbon A, Lhuillier Y, Charles H P. Hardware Acceleration of Red – Black Tree Management and Application to Just – In – Time Compilation [J]. Journal of Signal Processing Systems, 2014, 77(1 – 2):95 – 115.
[10] 程科. 嵌入式 Linux 设备驱动程序的设计与研究 [D]. 电子科技大学,2007.
[11] 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):136 – 137.
[12] Bose P, Douïeb K, Morin P. Skip lift: A probabilistic alternative to red – black trees [J]. Journal of Discrete Algorithms, 2012, 14: 13 – 20.
[13] 程钢,马汉杰,王宇. 基于众核处理器的高并发视频转码与分发系统 [J]. 网络新媒体技术,2016,5(02):25 – 29.
[14] Okasaki, Chris. Red – black trees in a functional setting [J]. Journal of Functional Programming. 2000, 9 (4): 471 – 477.

作者简介

马博韬,男,(1992 –),博士研究生,研究方向: 嵌入式系统、容器虚拟化技术。
孙鹏,男,(1976 –),博士,研究员,研究方向: 信号与信息处理。
朱小勇,男,(1982 –),博士,副研究员,研究方向: 嵌入式系统,多媒体技术。