An Optimal Algorithm for Computing the Largest Number of Red Nodes

Daxin Zhu[1, a], Xiaodong Wang[2, b,*]

[1]Quanzhou Normal University, 362000 Quanzhou, Fujian, China.

[2]Fujian University of Technology, Fuzhou, 350108 Fujian, China.

[a]email: dex@qztc.edu.cn, [b]wangxd@139.com, [*]Corresponding author

Keywords: Red-black trees; dynamic programming; data structure; time complexity

Abstract. In this paper, we investigate the problem to compute the largest number of red nodes in red-black trees in red-black trees. We first present a dynamic programming solution for computing $r(n)$, the largest number of red internal nodes in a red-black tree on n keys in $O(n^2 \log n)$ time. Then the algorithm is improved to a new $O(n)$ time algorithm. Based on the structure of the solution we finally present a linear time recursive algorithm using only $O(\log n)$ space.

Introduction

Red-black tree was invented in 1972 by Rudolf Bayer[2]. Guibas and Sedgewick named it red-black tree in 1978[4]. In their paper they studied the properties of red-black trees at length and introduced the red/black color convention. Andersson [1] gives a simpler-to-code variant of red-black trees. Weiss [7] calls this variant AA-trees. An AA-tree is similar to a red-black tree except that left children may never be red. In 2008, Sedgewick introduced a simpler version of the red-black tree called the left-leaning red-black tree[5] by eliminating a previously unspecified degree of freedom in the implementation. Red-black trees can be made isometric to either 2-3 trees or 2-4 trees,[5] for any sequence of operations.

The number of black nodes on any simple path from, but not including, a node x down to a leaf is called the black-height of the node, denoted $bh(x)$. By the property (5), the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. The black-height of a red-black tree is defined to be the black-height of its root.

We are interested in the number of red nodes in red-black trees in this paper. We will investigate the problem that in a red-black tree on n keys, what is the largest possible ratio of red internal nodes to black internal nodes, and what is the smallest possible ratio.

Dynamic Programming Algorithm

Let T be a red-black tree on n keys. The largest and the smallest number of red internal nodes in a red-black tree on n keys can be denoted as $r(n)$ and $s(n)$ respectively. The values of $r(n)$ and $s(n)$ can be easily observed for the special case of $n = 2^k - 1$. In the general cases, we denote the largest number of red internal nodes in a subtree of size i and black-height j to be $a(i, j, 0)$ when its root red and $a(i, j, 1)$ when its root black respectively. Since in a red-black tree on n keys we have $\frac{1}{2}\log n \le j \le 2\log n$, we have,

$$\gamma(n,k) = \max_{\frac{1}{2}\log n \le j \le 2\log n} a(n, j, k) \tag{1}$$

Furthermore, for any $1 \le i \le n, \frac{1}{2}\log i \le j \le 2\log i$, we can denote,

© 2015. The authors - Published by Atlantis Press

$$\begin{cases} \alpha_1(i,j) = \max\limits_{0\le t\le i/2}\{a(t,j-1,1)+a(i-t-1,j-1,1)\} \\ \alpha_2(i,j) = \max\limits_{0\le t\le i/2}\{a(t,j,0)+a(i-t-1,j,0)\} \\ \alpha_3(i,j) = \max\limits_{0\le t\le i/2}\{a(t,j-1,1)+a(i-t-1,j,0)\} \\ \alpha_4(i,j) = \max\limits_{0\le t\le i/2}\{a(t,j,0)+a(i-t-1,j-1,1)\} \end{cases} \quad (2)$$

Theorem 1

For each $1 \le i \le n, \frac{1}{2}\log i \le j \le 2\log i$, the values of $a(i,j,0)$ and $a(i,j,1)$ can be computed by the following dynamic programming formula.

$$\begin{cases} a(i,j,0) = 1+\alpha_1(i,j) \\ a(i,j,1) = \max\{\alpha_1(i,j),\alpha_2(i,j),\alpha_3(i,j),\alpha_4(i,j)\} \end{cases} \quad (3)$$

Proof.

For each $1 \le i \le n, \frac{1}{2}\log i \le j \le 2\log i$, let $T(i,j,0)$ be a red-black tree on i keys and black-height j with the largest number of red internal nodes, when its root red. $T(i,j,1)$ can be defined similarly when its root black. The red internal nodes of $T(i,j,0)$ and $T(i,j,1)$ must be $a(i,j,0)$ and $a(i,j,1)$ respectively.

(1) We first look at $T(i,j,0)$. Since its root is red, its two sons must be black, and thus the black-height of the corresponding subtrees L and R must be both $j-1$. For each $0 \le t \le i/2$, subtrees $T(t,j-1,1)$ and $T(i-t-1,j-1,1)$ connected to a red node will be a red-black tree on i keys and black-height j. Its number of red internal nodes must be $1+a(t,j-1,1)+a(i-t-1,j-1,1)$. In such trees, $T(i,j,0)$ achieves the maximal number of red internal nodes. Therefore, we have,

$$a(i,j,0) \ge \max_{0\le t\le i/2}\{1+a(t,j-1,1)+a(i-t-1,j-1,1)\} \quad (4)$$

On the other hand, we can assume the sizes of subtrees L and R are t and $i-t-1$, $0 \le t \le i/2$, WLOG. If we denote the number of red internal nodes in L and R to be $r(L)$ and $r(R)$, then we have that $r(L) \le a(t,j-1,1)$ and $r(R) \le a(i-t-1,j-1,1)$. Thus we have,

$$a(i,j,0) \le 1+\max_{0\le t\le i/2}\{a(t,j-1,1)+a(i-t-1,j-1,1)\} \quad (5)$$

Combining (4) and (5), we obtain,

$$a(i,j,0) = 1+\max_{0\le t\le i/2}\{a(t,j-1,1)+a(i-t-1,j-1,1)\} \quad (6)$$

(2) We now look at $T(i,j,1)$. Since its root is black, there can be 4 cases of its two sons such as red and red, black and black, black and red or red and black. If the subtree L or R has a red root, then the black-height of the corresponding subtree must be j, otherwise, if its root is black, then the black-height of the subtree must be $j-1$.

In the first case, both of the subtrees L and R have a black root. For each $0 \le t \le i/2$, subtrees $T(t,j-1,1)$ and $T(i-t-1,j-1,1)$ connected to a black node will be a red-black tree on i keys and black-height j. Its number of red internal nodes must be $a(t,j-1,1)+a(i-t-1,j-1,1)$. In such trees, $T(i,j,1)$ achieves the maximal number of red internal nodes. Therefore, we have,

$$a(i,j,1) \ge \max_{0 \le t \le i/2} \{a(t, j-1,1) + a(i-t-1, j-1,1)\} = \alpha_1(i,j) \quad (7)$$

For the other three cases, we can conclude similarly that

$$a(i,j,1) \ge \max_{0 \le t \le i/2} \{a(t, j,0) + a(i-t-1, j,0)\} = \alpha_2(i,j) \quad (8)$$

$$a(i,j,1) \ge \max_{0 \le t \le i/2} \{a(t, j-1,1) + a(i-t-1, j,0)\} = \alpha_3(i,j) \quad (9)$$

$$a(i,j,1) \ge \max_{0 \le t \le i/2} \{a(t, j,0) + a(i-t-1, j-1,1)\} = \alpha_4(i,j) \quad (10)$$

Therefore, we have,

$$a(i,j,1) \ge \max\{\alpha_1(i,j), \alpha_2(i,j), \alpha_3(i,j), \alpha_4(i,j)\} \quad (11)$$

On the other hand, we can assume the sizes of subtrees L and R are t and $i-t-1$, $0 \le t \le i/2$, WLOG. In the first case, if we denote the number of red internal nodes in L and R to be $r(L)$ and $r(R)$, then we have that $r(L) \le a(t, j-1,1)$ and $r(R) \le a(i-t-1, j-1,1)$, and thus we have,

$$a(i,j,1) \le \max_{0 \le t \le i/2} \{a(t, j-1,1) + a(i-t-1, j-1,1)\} = \alpha_1(i,j) \quad (12)$$

For the other three cases, we can conclude similarly that

$$a(i,j,1) \le \max_{0 \le t \le i/2} \{a(t, j,0) + a(i-t-1, j,0)\} = \alpha_2(i,j) \quad (13)$$

$$a(i,j,1) \le \max_{0 \le t \le i/2} \{a(t, j-1,1) + a(i-t-1, j,0)\} = \alpha_3(i,j) \quad (14)$$

$$a(i,j,1) \le \max_{0 \le t \le i/2} \{a(t, j,0) + a(i-t-1, j-1,1)\} = \alpha_4(i,j) \quad (15)$$

Therefore, we have,

$$a(i,j,1) \le \max\{\alpha_1(i,j), \alpha_2(i,j), \alpha_3(i,j), \alpha_4(i,j)\} \quad (16)$$

Combining (11) and (16) , we obtain,

$$a(i,j,1) = \max\{\alpha_1(i,j), \alpha_2(i,j), \alpha_3(i,j), \alpha_4(i,j)\} \quad (17)$$

The proof is complete. ■

According to Theorem 1, our algorithm for computing $a(i,j,k)$ is a standard 2-dimensional dynamic programming algorithm.

Improvement of the Algorithm

We have computed $r(n)$ and the corresponding red-black trees using Algorithm 1. Some pictures of the computed red-black trees with largest number of red nodes are listed. From these pictures of the red-black trees with largest number of red nodes in various size, we can observe some properties of $r(n)$ and the corresponding red-black trees as follows.

(1) The red-black tree on n keys with $r(n)$ red nodes can be realized in a complete binary search tree, called a maximal red-black tree.

(2) In a maximal red-black tree, the colors of the nodes on the left spine are alternatively red, black, $\cdots$, from the bottom to the top, and thus the black-height of the red-black tree must be $\frac{1}{2}\log n$.

(3) In a maximal red-black tree of k levels, if all of the nodes of the last two levels ($k, k-1$) and all of the black nodes of the last third level ($k-2$) are removed, the remaining tree is also a maximal red-black tree.

From these observations, we can improve the dynamic programming formula of Theorem 1 further. The first improvement can be made by the observation (2). Since the black-height of the maximal red-black tree on i keys must be $\frac{1}{2}\log i$, the loop bodies of the Algorithm 1 for j can be restricted to $j=\frac{1}{2}\log i$ to $1+\frac{1}{2}\log i$, and thus the time complexity of the dynamic programming algorithm can be reduced immediately to $O(n^2)$. By the observation (3), the time complexity of the algorithm can be reduced substantially as follows.

Theorem 2

Let n be the number of keys in a red-black tree, and $r(n)$ be the largest number of red nodes in a red-black tree on n keys. The values of $r(n)$ can be computed by the following recursive formula.

$$r(n)=\begin{cases} n-\lfloor \log n \rfloor & n<8 \\ r(p)+q & n\geq 8 \end{cases} \tag{19}$$

where

$$\begin{cases} p=2^{\lfloor \log n \rfloor -2}+\lceil (n-2^{\lfloor \log n \rfloor}+1)/4 \rceil -1 \\ q=n-2^{\lfloor \log n \rfloor -1}-2\lceil (n-2^{\lfloor \log n \rfloor}+1)/4 \rceil +1 \end{cases} \tag{20}$$

Proof.

Let T be a maximal red-black tree of size n. It is obvious that T has $k=1+\lfloor \log n \rfloor$ levels.

(1) The formula can be verified directly for the case of $n<8$.

(2) In the case of $n\geq 8$, we have $k>3$. The number of nodes in the last level of T must be $s=n-2^{\lfloor \log n \rfloor}+1$. These nodes are all red nodes of T. It is readily seen that every 4 red nodes in the last level correspond to 2 black nodes in level $k-1$ of T. Thus the number of black nodes in level $k-1$ must be $b=2\lceil (n-2^{\lfloor \log n \rfloor}+1)/4 \rceil$. It follows that the number of red nodes in level $k-1$ of T is $2^{\lfloor \log n \rfloor -1}-b$. Therefore, the number of red nodes in the last two levels of T is $s+2^{\lfloor \log n \rfloor -1}-b$, which is exactly $q=n-2^{\lfloor \log n \rfloor -1}-2\lceil (n-2^{\lfloor \log n \rfloor}+1)/4 \rceil +1$.

Let T' be the subtree of T by removing all of the nodes of the last two levels ($k,k-1$) and all of the black nodes in level $k-2$ from T. Since every 2 black nodes in level $k-1$ correspond to 1 red node in level $k-2$ of T, the number of red nodes in level $k-2$ is obviously $b/2$, and thus the size of T' must be $2^{\lfloor \log n \rfloor -2}+b/2$, which is exactly $p=2^{\lfloor \log n \rfloor -2}+\lceil (n-2^{\lfloor \log n \rfloor}+1)/4 \rceil -1$. It follows from observation (3) that $r(n)=r(p)+q$.

The proof is complete.

According to Theorem 2, a new recursive algorithm for computing the largest number of red internal nodes in a red-black tree on n keys can be implemented.

For the same problem, we can build another efficient algorithm in a different point of view. Let us look at the sequence of the values of $r(n)$ listed in the increasing order of n.

If we list the sequence as a triangle $t(i,j), i=0,1,\cdots, j=1,2,\cdots,2^i$, then we can observe some interesting structural properties of $r(n)$.

It is readily seen that the values in each row have some regular patterns as follows.

(1) For the fist elements $t(i,1)$ in each row $i=0,1,\cdots$, we have,

$t(2j+1,1)=2t(2j,1)+1, t(2j,1)=2t(2j-1,1), j=1,2,\cdots$.

(2) For the elements $t(i,2^{i-1}+1)$ in each row $i=1,2,\cdots$, we have, $t(i,2^{i-1}+1)=2^i-1$.

(3) For the elements $t(i, j), 2 \le j \le 2^{i-1}$ in each row $i = 2,3,\cdots$, we have, $t(i, j) = t(i-1, j) + c$ where c is a constant.

(4) For the elements $t(i, j), 2^{i-1} + 2 \le j \le 2^i$ in each row $i = 2,3,\cdots$, we have,

$t(i, j) = t(i-1, j-2^{i-1}) + d$ where d is a constant.

In the insight of these observations, we can build another efficient algorithm to compute $r(n)$ as follows.

Theorem 3

Let n be the number of keys in a red-black tree, and $r(n)$ be the largest number of red nodes in a red-black tree on n keys. If the values of $r(n)$ are listed as a triangle $t(i, j), i = 0,1,\cdots, j = 1,2,\cdots,2^i$ as shown in Table 2, then the values of $t(i, j)$ can be computed by the following recursive formula.

$$t(i, j) = \begin{cases} i & i < 2 \\ \xi(i) & 2 \le i, j = 1 \\ t(i-1, j) + \xi(i-1) & 2 \le i, 2 \le j \le 2^{i-1} \\ 2^i - 1 & 2 \le i, j = 2^{i-1} + 1 \\ t(i-1, j-2^{i-1}) + \eta(i) & 2 \le i, 2^{i-1} + 2 \le j \le 2^i \end{cases} \tag{21}$$

where

$$\begin{cases} \xi(i) = \dfrac{2}{3}\left(2^i - \dfrac{3 + (-1)^i}{4}\right) = \left\lceil \dfrac{2}{3}(2^i - 1) \right\rceil \\ \eta(i) = \dfrac{1}{3}(2^{i+1} + (-1)^i) = \left\lfloor \dfrac{1}{3}(2^{i+1} + 1) \right\rfloor \end{cases} \tag{22}$$

According to Theorem 3, a recursive algorithm for computing the values of $t(i, j)$ can be implemented as the following Algorithm.

Algorithm 3 $t(i, j)$

Input: Integer i, j, the row number and the collum number

Output: $t(i, j)$

if $i < 2$ **then**
    return i
else
    if $j = 1$ **then**
        return $\lceil \frac{2}{3}(2^i - 1) \rceil$
    else
        if $2 \le j \le 2^{i-1}$ **then**
            return $t(i-1, j) + \lceil \frac{2}{3}(2^{i-1} - 1) \rceil$
        else
            if $j = 2^{i-1} + 1$ **then**
                return $2^i - 1$
            else
                return $t(i-1, j-2^{i-1}) + \lfloor \frac{1}{3}(2^{i+1} + 1) \rfloor$
            end if
        end if
    end if
end if

It can be seen that for any positive integer n, if $t(i, j) = r(n)$, then $i = \lfloor \log(n+1) \rfloor$ and $j = n - 2^{\lfloor \log(n+1) \rfloor} + 2$. In a call of Algorithm, $t(\lfloor \log(n+1) \rfloor, n - 2^{\lfloor \log(n+1) \rfloor} + 2)$ will return the value of $r(n)$. It is obvious that the recursive depth of the Algorithm is at most $\lfloor \log(n+1) \rfloor$. Therefore, our new algorithm requires only $O(\log n)$ time.

Acknowledgement

This work was supported in part by the Natural Science Foundation of Fujian (Grant No.2013J01247), Fujian Provincial Key Laboratory of Data-Intensive Computing and Fujian University Laboratory of Intelligent Computing and Information Processing.

References

[1] Andersson, Balanced search treesmade simple, In Proceedings of the Third Workshop on Algorithms and Data Structures, vol. 709 of Lecture Notes in Computer Science, 1993, pp. 60-71.

[2] R. Bayer, Symmetric binary B-trees: Data structure and maintenance algorithms, Acta Informatica, 1(4), 1972, pp. 290-306.

[3] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., Introduction to algorithms, 3rd ed., MIT Press, Cambridge, MA, 2009.

[4] Leo J. Guibas and Robert Sedgewick, A dichromatic framework for balanced trees, In Proceedings of the 19th Annual Symposium on Foundations of Computer Science, 1978, pp. 8-21.

[5] Robert Sedgewick, Left-leaning Red?CBlack Trees, http://www.cs.princeton.edu/ rs/talks/ LLRB/LLRB.pdf

[6] Henry S. Warren, Hacker's Delight, Addison-Wesley, second edition, 2002.

[7] Mark Allen Weiss, Data Structures and Problem Solving Using C++, Addison-Wesley, second edition, 2000.