A Simple Dynamic Programming Algorithm for Counting Red Nodes in Red-Black Trees

Daxin Zhu, Xiaodong Wang, and Jun Tian

***Abstract*—In this paper, we are interested in the number of red nodes in red-black trees. We first present an $O(n^2\log n)$ time dynamic programming solution for computing $r(n)$, the largest number of red internal nodes in a red-black tree on n keys. 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.**

***Index Terms*—Red-black trees, red internal nodes, dynamic programming, linear time solution**.

I. Introduction

A red-black tree is a special type of binary tree, used in computer science to organize pieces of comparable data, such as strings or numbers. The original data structure was invented in 1972 by Rudolf Bayer [1] with its name 'symmetric binary B-tree'. In a paper entitled 'A Dichromatic Framework for Balanced Trees', Guibas and Sedgewick named it red-black tree in 1978 [2]. In their paper they studied the properties of red-black trees at length and introduced the red/black color convention. Andersson [3] gives a simpler-to-code variant of red-black trees. Weiss [4] calls these 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.

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either red or black. It satisfies the following red-black properties [6]:

1) A node is either red or black.
2) The root is black.
3) All leaves (NIL) are black.
4) Every red node must have two black child nodes.
5) Every path from a given node to any of its descendant leaves contains the same number of black nodes.

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.

The property 2) is sometimes omitted in practice. Since the root can always be changed from red to black, but not necessarily vice-versa, this property has little effect on analysis. A binary search tree that satisfies red-black properties 1), 3), 4), and 5) is sometimes called a relaxed red-black tree. In this paper we will discuss the relaxed red-black tree and call a relaxed red-black tree a red-black tree.

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.

The organization of the paper is as follows. In the following 3 sections we describe our presented algorithm for computing the largest number of red internal nodes in a red-black tree on n keys. In Section II we present a dynamic programming algorithm for the problem. We then improve the algorithm to a new $O(n)$ time algorithm in Section III. Based on the structure of the solution we finally come to a linear time recursive algorithm using only $O(\log n)$ space. Some concluding remarks are in Section IV.

II. A dynamic Programming Algorithm

A. Some Special Cases

$$
\begin{aligned}
r(n) = r(2^k-1) &= \sum_{i=0}^{\lfloor (k-1)/2\rfloor} 2^{k-2i-1} = 2^{k-1}\sum_{i=0}^{\lfloor (k-1)/2\rfloor}\frac{1}{4^i}\\
&= \frac{2^{k-1}}{3}\left(4-\frac{1}{4^{\lfloor (k-1)/2\rfloor}}\right) = \frac{2^{k+1}-2^{k-1-2\lfloor (k-1)/2\rfloor}}{3}\\
&= \frac{2^{k+1}-2^{(k-1)mod 2}}{3} = \frac{2^{k+1}-2+kmod2}{3}\\
&= \frac{2(2^k-1)+kmod2}{3} = \frac{2n+\log(n+1)mod2}{3}
\end{aligned}
$$

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 $n=2^k-1$. It is obvious that in this case,

Manuscript received June 16, 2014; revised November 1, 2014. This work was supported in part by the Natural Science Foundation of Fujian (Grant No.2013J01247), and Fujian Provincial Key Laboratory of Data-Intensive Computing and Fujian University Laboratory of Intelligent Computing and Information Processing.

Daxin Zhu is with Quanzhou Normal University, Quanzhou, China (e-mail: dex@qztc.edu.cn).

Xiaodong Wang and Jun Tian are with Fujian University of Technology, Fuzhou, China (e-mail:wangxd135@139.com, tianjunfjmu@126.com).

when the node colors are alternately red and black from the bottom level to the top level of *T*, the number of red internal nodes of *T* must be maximal. When all of its internal nodes are black, the number of red internal nodes of *T* must be minimal. Therefore, in the case of $n = 2^k - 1$, we have, Then, the number of black nodes *b*(*n*) must be,

$$\begin{aligned} b(n) &= n - r(n) \\ &= n - \frac{2n + \log(n+1) mod 2}{3} \\ &= \frac{n - \log(n+1) mod 2}{3} \end{aligned}$$

Therefore, the ratio of red internal nodes to black internal nodes is,

$$r(n)/b(n) = \frac{2n + \log(n+1) mod 2}{n - \log(n+1) mod 2}$$

If $k mod\, 2 = 0$, then $r(n)/b(n) = 2$, otherwise $k mod 2 = 1$, $r(n)/b(n) = \frac{2n+1}{n-1}$.

It follows that for any k, if $n = 2^k - 1$, then $r(n)/b(n) \le \frac{2n+1}{n-1}$.

Notice that, $\frac{d}{dx}\left(\frac{2x+1}{x-1}\right) = -\frac{3}{(x-1)^2} < 0$, and $\lim_{n\to\infty} \frac{2n+1}{n-1} = 2$, the values of $\frac{2n+1}{n-1}$ decrease monotonically to 2. In the special case of $k = 3$, $r(n)/b(n) = \frac{2n+1}{n-1}$ gets its maximal value of $5/2 = 2.5$.

Therefore, we have,

$$0 \le r(n)/b(n) \le \frac{2n+1}{n-1} \le 2.5$$

This formula can also be generalized to general n.

In the general cases, denote the largest number of red internal nodes in a red-black tree on n keys be $\gamma(n,0)$ if root red and $\gamma(n,1)$ if root black respectively. Then, $r(n) = \max\{\gamma(n,0), \gamma(n,1)\}$. We can prove by induction that $\gamma(n,0) \le \frac{2n+1}{3}$ and $\gamma(n,1) \le \frac{2n}{3}$. It follows that $r(n) \le \max\left\{\frac{2n+1}{3}, \frac{2n}{3}\right\} = \frac{2n+1}{3}$

Therefore, for $n \ge 7$, we have

$$0 \le \frac{r(n)}{n - r(n)} \le \frac{\frac{2n+1}{3}}{n - \frac{2n+1}{3}} = \frac{2n+1}{n-1} \le 2.5$$

B. The Dynamic Programming Formula

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,

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

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.

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

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.

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,

$$\begin{aligned} a(i, j, 0) \ge \max_{0\le t\le i/2}\{&1 + a(t, j-1,1) \\ &+ a(i-t-1, j-1,1)\} \end{aligned} \tag{4}$$

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,

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

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

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

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, 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,

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

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. By the recursive formula (2) and (3) , the dynamic programming algorithm for computing the largest number of red internal nodes in a red-black tree on n keys can be implemented as the following Algorithm 1.

```
Algorithm 1 : $r(n)$
Input: Integer $n$ , the number of keys in a red-black tree
Output: $r(n)$ , the largest number of red nodes in a red-black tree on $n$ keys
1: for all $i, j, k$ , $0 \le i \le n, 0 \le j \le 2\log n$ , and $0 \le k \le 1$ do
2:     $a(i, j, k) \leftarrow 0$
3: end for
4: $a(1,1,0) \leftarrow a(2,1,1) \leftarrow a(3,2,0) \leftarrow 1; a(3,1,1) \leftarrow 2$
   {boundary condition}
5: for $i = 4$ to $n$ do
6:     for $j = \frac{1}{2}\log i$ to $2\log i$ do
7:         $\alpha_1 \leftarrow \max_{0 \le t \le i/2} \{a(t, j-1, 1) + a(i-t-1, j-1, 1)\}$
8:         $\alpha_2 \leftarrow \max_{0 \le t \le i/2} \{a(t, j, 0) + a(i-t-1, j, 0)\}$
9:         $\alpha_3 \leftarrow \max_{0 \le t \le i/2} \{a(t, j-1, 1) + a(i-t-1, j, 0)\}$
10:        $\alpha_4(i, j) \leftarrow \max_{0 \le t \le i/2} \{a(t, j, 0) + a(i-t-1, j-1, 1)\}$
11:        $a(i, j, 0) \leftarrow \max\{1 + \alpha_1, a(i, j, 0)\}$
12:        $a(i, j, 1) \leftarrow \max\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, a(i, j, 1)\}$
13:    end for
14: end for
15: return $\max_{0 \le k \le 1, \frac{1}{2}\log n \le j \le 2\log n} \{a(n, j, k)\}$
```

It is obvious that the Algorithm 1 requires $O(n^2 \log n)$ time and $O(n \log n)$ space.

The algorithm for computing $s(n)$, the smallest number of red nodes in a red-black tree on n keys can be built similarly.

III. The Improved Dynamic Programming Solutions

We have computed $r(n)$ and the corresponding red-black trees using Algorithm 1. From some examples of the computed red-black trees with largest number of red nodes, 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 $\frac{1}{2}\log n$.

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 $1+\frac{1}{2}\log i$, 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 $O(n^2)$.

It is readily seen from observation (1) that every subtree in a maximal red-black tree must be a complete binary search tree. If the size of a complete binary search tree *T* is n, then the size of its left subtree must be $\text{left}(n)=2^{\lfloor\log n\rfloor-1}-1+\min\{2^{\lfloor\log n\rfloor-1}, n-2^{\lfloor\log n\rfloor}+1\}$ and the size of its right subtree must be $\text{right}(n)=n-\text{left}(n)-1$.

Therefore, the maximal range $0\le t\le i/2$ of the Algorithm 2 can be restricted to $t=\text{left}(i)$, and thus the time complexity of the dynamic programming algorithm can be reduced further to $O(n)$. The time complexity is reduced substantially to $O(n)$, but the space costs remain unchanged. In the insight of above observations (1) and (2), we can build another efficient algorithm to compute $r(n)$ using only $O(\log n)$ space.

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 $r(n)=d(1)$ can be computed by the following recursive formula.

$$d(m)=\begin{cases} h(m) & h(m)\le 1 \\ 1+d(4m)+d(4m+1) & \\ +d(4m+2)+d(4m+3) & h(m)\bmod 2=1 \\ d(2m)+d(2m+1) & h(m)\bmod 2=0 \end{cases} \tag{18}$$

where

$$h(m)=\begin{cases} 1+\lfloor\log n\rfloor-\lfloor\log m\rfloor & \frac{m}{2^{\lfloor\log n\rfloor-\lfloor\log m\rfloor}}\le n \\ \lfloor\log n\rfloor-\lfloor\log m\rfloor & \text{"otherwise"} \end{cases} \tag{19}$$

Proof. In a maximal red-black tree, we can label the nodes as a pre-order sequence like a heap. The root is labeled 1. For each node i in the tree, its left child is labeled $2i$ and its right child is labeled $2i+1$. If we denote $d(i)$, the largest number of red nodes and $h(i)$, the height of the subtree rooted at node i, then it is obvious that $r(n)=d(1)$. It is not difficult to verify that in the case of $\frac{i}{2^{\lfloor\log n\rfloor-\lfloor\log i\rfloor}}>n$, we have $h(i)=\lfloor\log n\rfloor-\lfloor\log i\rfloor$, otherwise, $h(i)=1+\lfloor\log n\rfloor-\lfloor\log i\rfloor$.

It can be verified directly that if $h(i)\le 1$, then $d(i)=h(i)$.

It follows from observation (2) that if $h(i)$ is even then node i is red and its left and right subtrees rooted at nodes $2i$ and $2i+1$ are both maximal red-black trees of black root. In the case of $h(i)$ odd, the node i is black and its four grand children rooted at nodes $4i$, $4i+1$, $4i+2$ and $4i+3$ are all maximal red-black trees. Therefore, we can conclude that in the case of $h(i)>1$,

$$d(i)=\begin{cases} 1+d(4i)+d(4i+1)+ & h(i) \quad odd \\ d(4i+2)+d(4i+3) & \\ d(2i)+d(2i+1) & h(i) \quad even \end{cases}$$

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 efficiently. Since the algorithm visit each node at most once, the time cost of the algorithm is thus $O(n)$. The space used by the algorithm is only the stack space requirement of recursive calls. The recursive depth is at most $\log n$, and therefore the space cost of the algorithm is $O(\log n)$.

IV. Concluding Remarks

We have suggested a dynamic programming solution for computing $r(n)$, the largest number of red internal nodes in a red-black tree on n keys. The dynamic programming algorithm requires $O(n^2\log n)$ time and $O(n\log n)$ space. We then improve the algorithm to a new $O(n)$ time algorithm. Based on the structure of the solution we finally come to a linear time recursive algorithm using only $O(\log n)$ space. The smallest number of red internal nodes in a red-black tree on n keys can be computed analogously.

References

[1] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, 1972, pp. 290-306.

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

[3] A. Andersson, "Balanced search treesmade simple," in *Proc. the third Workshop on Algorithms and Data Structures*, vol. 709, 1993, pp. 60-71.

[4] M. A. Weiss, *Data Structures and Problem Solving Using C++*, Addison-Wesley, 2000.

[5] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. the 19th Annual Symposium on Foundations of Computer Science*, 1978, pp. 8-21.

[6] R. Sedgewick. Left-Leaning Red CBlack Trees. [Online]. Available: http://www.cs.princeton.edu/ rs/talks/LLRB/LLRB.pdf

Daxin Zhu received his M.Sc. degree in computer science from Huaqiao University of China in 2003. He is now an associate professor in Quanzhou Normal University of China. His current research interests include design and analysis of algorithms, network architecture and data intensive computing.