On Determination of Balance Ratio for Some Tree Structures

Daxin Zhu, Tinran Wang, Xiaodong Wang

On determination of balance ratio for some tree structures

Daxin Zhu[1],Tinran Wang[3], and Xiaodong Wang[2⋆]

[1] Quanzhou Normal University, Quanzhou 362000, China
[2] Fujian University of Technology, Fuzhou 350108, China
[3] School of Mathematical Science, Peking University, Beijing, 100871, China

Abstract In this paper, we studies the problem to find the maximal number of red nodes of a kind of balanced binary search tree. We have presented a dynamic programming formula for computing $r(n)$, the maximal number of red nodes of a red-black tree with n keys. The first dynamic programming algorithm uses $O(n^2 \log n)$ time and uses $O(n \log n)$ space. The basic algorithm is then improved to a more efficient $O(n)$ time algorithm. The time complexity of the new algorithm is finally reduced to $O(n)$ and the space is reduced to only $O(\log n)$.

1 Introduction

This paper studies the worst case balance ratio of the red-black tree structure. A red-black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions. The data structure was originally presented by Rudolf Bayer in 1972 with its name 'symmetric binary B-tree' [2]. Guibas and Sedgewick named the data structure red-black tree in 1978, [4]. They introduced the red/black color convention and the properties of a red-black tree at length. A simpler-to-code variant of red-black trees was presented in [1,5]. This variant of red-black trees was called the variant AA-trees[8]. The left-leaning red-black tree[6] was introduced in 2008 by Sedgewick. It is a new version of red-black tree which eliminated a previously unspecified degree of freedom. Either 2-3 trees or 2-4 trees can also be made isometric to red-black trees for any sequence of operations [6].

2 The basic properties and algorithms

A red-black tree of n keys is denoted by T in this paper. In a red-black tree T of n keys, $r(n)$ and $s(n)$, are defined as the maximal and the minimal number of red internal nodes respectively. It is readily seen that in this case of $n = 2^k - 1$, the number of red nodes of T achieves its maximum, if the node from the bottom to the top are colored alternately red and black. the number of red nodes of T achieves its minimum, if the node from the bottom to the top are all colored black.

⋆ Corresponding author

In the special case of $n = 2^k - 1$, we can conclude,

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

The number of black nodes $b(n)$ can then be,

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

Therefore, in this case, the ratio of red nodes to black nodes is,

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

In the general cases, the maximal number of red nodes of a red-black tree with n keys can be denoted by $\gamma(n,0)$ if root is red, and by $\gamma(n,1)$ if root is black. We then have,

$$r(n) = \max\{\gamma(n,0), \gamma(n,1)\}.$$

It can be proved by induction further that $\gamma(n,0) \leq \frac{2n+1}{3}$ and $\gamma(n,1) \leq \frac{2n}{3}$. Therefore,

$$r(n) \leq \max\left\{ \frac{2n+1}{3}, \frac{2n}{3} \right\} = \frac{2n+1}{3}$$

Thus, for $n \geq 7$, we have

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

In the general cases, the maximal number of red nodes of a subtree of black-height j and size i can be denoted by $a(i,j,0)$ if root is red and by $a(i,j,1)$ if root is black. It follows from $\frac{1}{2}\log n \leq j \leq 2\log n$ that

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

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

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

Theorem 1 *$a(i,j,0)$ and $a(i,j,1)$ can be formulated for each $1 \leq i \leq n$, $\frac{1}{2}\log(i+1) \leq j \leq 2\log(i+1)$, as follows.*

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

Proof. Let i,j be two indices such that $1 \leq i \leq n$, $\frac{1}{2}\log(i+1) \leq j \leq 2\log(i+1)$. $T(i,j,0)$ is defined to be a red-black tree of i keys and black-height j, and its root red. $T(i,j,1)$ is defined similarly if its root black. The number of red nodes in $T(i,j,0)$ and $T(i,j,1)$ can be denoted respectively by $a(i,j,0)$ and $a(i,j,1)$.

(1) We consider $T(i,j,0)$ first. The two children of $T(i,j,0)$ must be black, since its root red. The two subtrees L and R must both have a black-height of j. The subtrees $T(t,j,1)$ and $T(i-t-1,j,1)$ which connected to a red node must be a red-black tree of black-height j and i keys. The number of red nodes is thus $1+a(t,j,1)+a(i-t-1,j,1)$. $T(i,j,0)$ have a maximal number of red nodes, and thus,

$$a(i,j,0) \geq \max_{0\leq t\leq i/2}\{1+a(t,j,1)+a(i-t-1,j,1)\} \tag{4}$$

On the other hand, If the number of red nodes of L and R are denoted by $r(L)$ and $r(R)$, and the sizes of L and R are t and $i-t-1$, then we can conclude that $r(L) \leq a(t,j,1)$ and $r(R) \leq a(i-t-1,j,1)$. Therefore we have,

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

It follows from (4) and (5) that,

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

(2) We now consider $T(i,j,1)$. The two subtrees L and R must both have a black-heights $j-1$.

If both L and R have a black root, then $T(t,j-1,1)$ and $T(i-t-1,j-1,1)$ must have black-height j and i keys. The number of red nodes must be $a(t,j-1,1)+a(i-t-1,j-1,1)$. It follows that

$$a(i,j,1) \geq \max_{0\leq t\leq i/2}\{a(t,j-1,1)+a(i-t-1,j-1,1)\} = \alpha_1(i,j-1) \tag{7}$$

The other three cases, can be discussed similarly.

$$a(i,j,1) \geq \max_{0\leq t\leq i/2}\{a(t,j-1,0)+a(i-t-1,j-1,0)\} = \alpha_2(i,j-1) \tag{8}$$

$$a(i,j,1) \geq \max_{0\leq t\leq i/2}\{a(t,j-1,1)+a(i-t-1,j-1,0)\} = \alpha_3(i,j-1) \tag{9}$$

$$a(i,j,1) \geq \max_{0\leq t\leq i/2}\{a(t,j-1,0)+a(i-t-1,j-1,1)\} = \alpha_4(i,j-1) \tag{10}$$

Therefore, we can conclude,

$$a(i,j,1) \geq \max\{\alpha_1(i,j-1),\alpha_2(i,j-1),\alpha_3(i,j-1),\alpha_4(i,j-1)\} \tag{11}$$

On the other hand, If the number of red nodes of L and R are denoted by $r(L)$ and $r(R)$, and the sizes of L and R are t and $i-t-1$, then we can conclude that $r(L) \leq a(t,j-1,1)$ and $r(R) \leq a(i-t-1,j-1,1)$. Therefore we have,

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

The other three cases can be discussed similarly that

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

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

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

It follows that

$$a(i,j,1) \leq \max\{\alpha_1(i,j-1),\alpha_2(i,j-1),\alpha_3(i,j-1),\alpha_4(i,j-1)\} \tag{16}$$

It follows from (11) and (16) that

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

It is clear that $\alpha_4(i,j)$ achieves its maximum at t_0 $\alpha_4(i,j) = a(t_0,j,0)+a(i-t_0-1,j,1), 0 \leq t_0 \leq i/2$, then $\alpha_3(i,j)$ achieves its maximum at $t_1 = i-t_0-1$, $\alpha_3(i,j) = a(t_1,j,1)+a(i-t_1-1,j,0), 0 \leq t_1 \leq i/2$. Thus, $\alpha_3(i,j) = \alpha_4(i,j)$, for each $1 \leq i \leq n, \frac{1}{2}\log(i+1) \leq j \leq 2\log(i+1)$, and finally we have,

$$a(i,j,1) = \max\{\alpha_1(i,j-1),\alpha_2(i,j-1),\alpha_3(i,j-1)\} \tag{18}$$

The proof is complete.

3 The improvement of time complexity

Some special pictures of the maximal red-black trees are listed in Fig. 2. It can be observed from these pictures of the maximal red-black trees that some properties of $r(n)$ are useful.

(1) The maximal red-black tree with $r(n)$ red nodes of n keys as shown above can be realized by a complete binary search tree.

(2) In the maximal red-black tree of n keys, the nodes along the left spine are colored red, black, $\cdots$, alternatively from the bottom to the top. In such a red-black tree, its black-height must be $\frac{1}{2}\log n$.

The dynamic programming formula in Theorem 1 we can be improved further from above observations. The second loop for j can be reduced to $j = \frac{1}{2}\log i$ to $1+\frac{1}{2}\log i$, since the black-height of i keys must be $1+\frac{1}{2}\log i$. The time complexity of the algorithm can thus be reduced immediately to $O(n^2)$.

It can be seen from observation (1) that the subtree of a T is a complete binary tree. If T has a size n, then its left subtree must have a size of

$$left(n) = 2^{\lfloor \log n \rfloor - 1} - 1 + \min\{2^{\lfloor \log n \rfloor - 1}, n - 2^{\lfloor \log n \rfloor} + 1\}$$

and its right subtree must have a size of

$$right(n) = n - left(n) - 1$$

It follows that the range $0 \le t \le i/2$ can be reduced to $t = left(i)$. The time complexity of the algorithm can thus be reduced further to $O(n)$.

Another efficient algorithm for $r(n)$ need only $O(\log n)$ space can be built as follows.

Theorem 2 *In a red-black tree T of n keys, let $r(n)$ be the maximal number of red nodes in T. The values of $d(1) = r(n)$ can be formulated as follows.*

$$d(m) = \begin{cases} h(m) & h(m) \le 1 \\ 1 + d(4m) + d(4m+1) + d(4m+2) + d(4m+3) & h(m) \mod 2 = 1 \\ d(2m) + d(2m+1) & h(m) \mod 2 = 0 \end{cases} \quad (19)$$

where

$$h(m) = \begin{cases} 1 + \lfloor \log n \rfloor - \lfloor \log m \rfloor & \frac{m}{2^{\lfloor \log n \rfloor - \lfloor \log m \rfloor}} \le n \\ \lfloor \log n \rfloor - \lfloor \log m \rfloor & \texttt{otherwise} \end{cases} \quad (20)$$

Proof. We can label the nodes of a maximal red-black tree like a heap. The root of the tree is labeled 1. The left child of a node i is labeled $2i$ and the right child is labeled $2i+1$. Let $d(i)$ denote the maximal number of red nodes of T and $h(i)$, denote the height at node i. Then we have $r(n) = d(1)$. It is easy to verify that if $\frac{i}{2^{\lfloor \log n \rfloor - \lfloor \log i \rfloor}} > n$, then 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 is obvious that if $h(i) \le 1$, then $d(i) = h(i)$.

Therefore, if $h(i)$ is even then the left subtree rooted at node $2i$ and the right subtree rooted at node $2i+1$ are both black. If $h(i)$ odd, the four nodes rooted at nodes $4i$, $4i+1$, $4i+2$ and $4i+3$ can be all maximal red-black trees. Therefore, if $h(i) > 1$, we have

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

The proof is complete.

A new recursive algorithm for $r(n)$ can be build as the following algorithm.

Algorithm 1 $t(i,j)$

Input: i, j, the row and the collum number
Output: $t(i,j)$

```
 1: if i < 2 then
 2:    return i
 3: else
 4:    if j = 1 then
 5:       return ⌈(2/3)(2^i − 1)⌉
 6:    else
 7:       if 2 ≤ j ≤ 2^(i−1) then
 8:          return t(i−1, j) + ⌈(2/3)(2^(i−1) − 1)⌉
 9:       else
10:          if j = 2^(i−1) + 1 then
11:             return 2^i − 1
12:          else
13:             return t(i−1, j−2^(i−1)) + ⌊(1/3)(2^(i+1) + 1)⌋
14:          end if
15:       end if
16:    end if
17: end if
```

It can be seen that the time complexity of the algorithm is $O(n)$, since each node is visited at most once in the algorithm. The space complexity of the algorithm is the stack space used in recursive calls. The depth of recursive is at most $\log n$, and the space complexity of the algorithm is thus $O(\log n)$.

In order to find $r(n)$, the algorithm can be reformulated in a non-recursive form as follows.

By further improvement of the formula, we can reduce algorithm to a very simple algorithm as follows.

It is clear that the time complexities of above two algorithms are both $O\log n$.

If the number of n can fit into a computer word, then $a(n)$ and $o(n)$ can be computed in $O(1)$ time.

Algorithm 2 $r(n)$

Input: n, the number of keys
Output: $r(n)$, the maximal number of red nodes

$r \leftarrow 1, j \leftarrow n$
for $i = 1$ to $\lfloor \log n \rfloor$ **do**
  if $j \bmod 2 = 1$ **then**
    $r \leftarrow r + \eta(i)$
  else
    $r \leftarrow r + \xi(i-1)$
  end if
  $j \leftarrow j/2$
end for
return r

Algorithm 3 $r(n)$

Input: n, the number of keys
Output: $r(n)$, the maximal number of red nodes

$r \leftarrow 1, x \leftarrow 0, y \leftarrow 1, j \leftarrow n$
for $i = 1$ to $\lfloor \log n \rfloor$ **do**
  if $j \bmod 2 = 1$ **then**
    $r \leftarrow r + y$
  else
    $r \leftarrow r + x$
  end if
  if $i \bmod 2 = 1$ **then**
    $x \leftarrow 2x + 1, y \leftarrow 2y + 1$
  else
    $x \leftarrow 2x, y \leftarrow 2y - 1$
  end if
  $j \leftarrow j/2$
end for
return r

```
int a(int n)
{
    n = (n & (0x55555555)) + ((n >> 1) & (0x55555555));
    n = (n & (0x33333333)) + ((n >> 2) & (0x33333333));
    n = (n & (0x0f0f0f0f)) + ((n >> 4) & (0x0f0f0f0f));
    n = (n & (0x00ff00ff)) + ((n >> 8) & (0x00ff00ff));
    n = (n & (0x0000ffff)) + ((n >> 16) & (0x0000ffff));
    return n;
}
```

```
int o(int n)
{
    n = ((n >> 1) & (0x55555555));
    n = (n & (0x33333333)) + ((n >> 2) & (0x33333333));
    n = (n & (0x0f0f0f0f)) + ((n >> 4) & (0x0f0f0f0f));
    n = (n & (0x00ff00ff)) + ((n >> 8) & (0x00ff00ff));
    n = (n & (0x0000ffff)) + ((n >> 16) & (0x0000ffff));
    return n;
}
```

4 Concluding remarks

We have presented a dynamic programming formula for computing $r(n)$, the maximal number of red nodes of a red-black tree with n keys. The time complexity of the new algorithm is finally reduced to $O(n)$ and the space is reduced to only $O(\log n)$.

Acknowledgement

This work was supported by Intelligent Computing and Information Processing of Fujian University Laboratory and Data-Intensive Computing of Fujian Provincial Key Laboratory.

References

1. A. Andersson, Balanced search treesmade simple, *Workshop on Algorithms and Data Structures*, 709 of *LNCS*, 1993, pp. 60-71.
2. R. Bayer, Symmetric binary B-trees, Data structure and maintenance algorithms, *Acta Informatica*, 1972, 1(4), pp. 290-306.
3. Michael T. Goodrich, and Roberto Tamassia, *Algorithm Design and Applications*, John Wiley & Sons, 2015.
4. Leo J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, *the 19th FOCS*, 1978, pp. 8-21.
5. P. Heejin, and P. Kunsoo, Parallel algorithms for redCblack trees, *Theoretical computer science*, 2001, 262(1-2), pp. 415-435.
6. R. Sedgewick, Left-leaning RedCBlack Trees, *http://www.cs.princeton.edu/rs/talks/LLRB/LLRB.pdf*
7. H. S. Warren, *Hackers Delight*, Addison-Wesley, second Ed., 2002.
8. M. A. Weiss, *Data Structures and Problem Solving Using C++*, Addison-Wesley, second Ed., 2000.