On the performance of balance factors of the dichromatic balanced trees for massive data

Xiaodong WANG[1, a], Daxin ZHU[2, b,*]

[1] Fujian University of Technology, Fuzhou, 350118 China

[2] Quanzhou Normal University, Quanzhou, 362000 China

[a]email: wangxd13@163.com, [b]email:zhudx22@163.com, [*] Corresponding Author

Keywords: Balanced Trees; Recursive Algorithms; Closed-Form Solutions

Abstract. In this paper, we are interested in minimal number of red-nodes in a red-black tree. An $O(n^2 \log n)$ time dynamic programming algorithm is presented first for computing $s(n)$, the smallest number of red internal nodes in a red-black tree on n keys. We then improve the algorithm to a new $O(n)$ time algorithm. Then the algorithm is improved further to some $O(\log n)$ time recursive and nonrecursive algorithms. These improved algorithms finally led to a closed-form solution of $s(n)$.

Introduction

The rapid growth of the Internet and World Wide Web led to vast amounts of information available online. In addition, business and government organizations create large amounts of both structured and unstructured information which needs to be processed, analyzed, and linked. The storing, managing, accessing, and processing of this vast amount of data represent a fundamental need and an immense challenge in order to satisfy needs to search, analyze, mine, and visualize this data as information. Data-intensive computing is intended to address this need. Previous work in Data-intensive computing infrastructure demonstrates that certain deadline constrained applications demand predictable quality of service, often requiring a number of computing resources to be available over a well defined period, commencing at a specific time in the future; good requirements for advance reservation.

This paper describes the worst case balance factors of a data structure, a red-black tree, for storing information on computing resource availability and performing admission control of ordinary requests and reservations of computing resources. 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[2] 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, [4].

A red-black tree an be seen as a binary search tree with one extra bit of storage per node: its color, which can be either red or black. A red-black tree must satisfy the following red-black properties[5]:

(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. If we denote the black-height of a red-black tree T as $bh(T)$, then for any red-black tree T on n keys, we have $\frac{1}{2}\log n \le bh(T) \le 2\log n$.

A dynamic programming algorithm

Let T be a red-black tree on n keys. The smallest number of red internal nodes in a red-black tree on n keys can be denoted as $s(n)$. The values of $s(n)$ can be easily observed for the special case of $n = 2^k - 1$.

In the general cases, we denote the smallest 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. For any $1 \le i \le n, 1/2\log i \le j \le 2\log i$, we can denote,

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

Theorem 1

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

$$\begin{cases} a(i, j,0) = 1 + \alpha_1(i, j) \\ a(i, j,1) = \min\{\alpha_1(i, j), \alpha_2(i, j), \alpha_3(i, j), \alpha_4(i, j)\} \end{cases} \quad (2)$$

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

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

The improved algorithms

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

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

(2) In a minimal red-black tree, each level can have at most one red node, and there is at most one red node on the left spine of the 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 there is at most one red node on the left spine of the tree, the black-height of the minimal red-black tree on i keys must be $1+\log(i+1)$, the loop bodies of the Algorithm 1 for j can be restricted to $j = \log(i+1)$ to $1+\log(i+1)$, and thus the time complexity of the dynamic programming algorithm can be reduced immediately to $O(n^2)$ as follows.

Algorithm 2 $s(n)$

Input: Integer n, the number of keys in a red-black tree
Output: $s(n)$, the smallest 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 $j = \log(i+1)$ to $1+\log(i+1)$ do
7:         $\alpha_1 \leftarrow \min_{0\le t\le i/2}\{a(t,j-1,1)+a(i-t-1,j-1,1)\}$
8:         $\alpha_2 \leftarrow \min_{0\le t\le i/2}\{a(t,j,0)+a(i-t-1,j,0)\}$
9:         $\alpha_3 \leftarrow \min_{0\le t\le i/2}\{a(t,j-1,1)+a(i-t-1,j,0)\}$
10:        $\alpha_4 \leftarrow \min_{0\le t\le i/2}\{a(t,j,0)+a(i-t-1,j-1,1)\}$
11:        $a(i,j,0) \leftarrow \min\{1+\alpha_1, a(i,j,0)\}$
12:        $a(i,j,1) \leftarrow \min\{\alpha_1,\alpha_2,\alpha_3,\alpha_4, a(i,j,1)\}$
13:    end for
14: end for
15: return $\min_{\substack{0\le k\le 1\\ \log(n+1)\le j\le 1+\log(n+1)}}\{a(n,j,k)\}$
```

It is readily seen from observation (1) that every subtree in a minimal 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

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

and the size of its right subtree must be

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

Therefore, the minimal range $0 \le t \le i/2$ of the Algorithm 2 can be restricted to $t = left(i)$, and thus the time complexity of the dynamic programming algorithm can be reduced further to $O(n)$ as follows.

Algorithm 3 $s(n)$

Input: Integer n, the number of keys in a red-black tree
Output: $s(n)$, the smallest 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:     $t \leftarrow 2^{\lfloor \log i\rfloor -1} - 1 + \min\{2^{\lfloor \log i\rfloor -1}, i - 2^{\lfloor \log i\rfloor} + 1\}$
7:     for $j = \log(i+1)$ to $1+\log(i+1)$ do
8:         $\alpha_1 \leftarrow a(t,j-1,1)+a(i-t-1,j-1,1)$
9:         $\alpha_2 \leftarrow a(t,j,0)+a(i-t-1,j,0)$
10:        $\alpha_3 \leftarrow a(t,j-1,1)+a(i-t-1,j,0)$
11:        $\alpha_4 \leftarrow a(t,j,0)+a(i-t-1,j-1,1)$
12:        $a(i,j,0) \leftarrow \min\{1+\alpha_1, a(i,j,0)\}$
13:        $a(i,j,1) \leftarrow \min\{\alpha_1,\alpha_2,\alpha_3,\alpha_4, a(i,j,1)\}$
14:    end for
15: end for
16: return $\min_{\substack{0\le k\le 1\\ \log(n+1)\le j\le 1+\log(n+1)}}\{a(n,j,k)\}$
```

The time complexity of Algorithm 3 is reduced substantially to $O(n)$, but the space costs remain unchanged. The algorithm can be improved further in a different point of view. If we list the sequence of the values of $s(n)$ as a triangle $t(i,j), i = 0,1,\cdots, j = 1,2,\cdots,2^i$ as shown in Table 1, then we can observe some interesting structural properties of $s(n)$.

Table 1: A triangle of sequence $s(n)$

Row No.	$r(n)$															
0	0															
1	0	1														
2	0	1	1	2												
3	0	1	1	2	1	2	2	3								
4	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
5	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4

It is readily seen from Table 1 that the values in each row have some regular patterns as follows.

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

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

In the insight of these observations, a recursive algorithm for computing the values of $t(i, j)$ can be implemented as the following Algorithm 4.

```
Algorithm 4 $t(i, j)$
Input: Integer $i, j$, the row number and the collum number
Output: $t(i, j)$
1: if $i < 1$ then
2:     return $0$
3: else if $1 \le j \le 2^{i-1}$ then
4:     return $t(i-1, j)$
5: else
6:     return $t(i-1, j-2^{i-1}) + 1$
7: end if
```

It can be seen from Table 1 that for any positive integer n, if $t(i, j) = s(n)$, then $i = \lfloor \log(n+1) \rfloor$ and $j = n - 2^{\lfloor \log(n+1) \rfloor} + 2$. In a call of Algorithm 4, $t(\lfloor \log(n+1) \rfloor, n - 2^{\lfloor \log(n+1) \rfloor} + 2)$ will return the value of $s(n)$. It is obvious that the recursive depth of the Algorithm 4 is at most $\lfloor \log(n+1) \rfloor$. Therefore, Algorithm 4 requires only $O(\log n)$ time.

The closed-form solution of $s(n)$

From Table 1 and Algorithm 4 we can observed further that the values of $s(n)$ satisfy a simple recursive formula,

$$\begin{cases} s(2n+1) = s(n) \\ s(2n) = s(n-1) + 1 \end{cases} \qquad (3)$$

Let $h(n) = s(n-1)$, then $s(n) = h(n+1)$, and $h(n)$ satisfies,

$$\begin{cases} h(2n+1) = h(n) + 1 \\ h(2n) = h(n) \end{cases} \qquad (4)$$

Based on the discussions above, we finally can find a closed-form solution of $s(n)$.

Theorem 2

Let n be the number of keys in a red-black tree, and its binary expression be $n = \sum_{i=0}^{\log n} b_i 2^i$, then $s(n)$ can be computed by the following formula,

$$s(n) = a(n+1) - 1 \qquad (5)$$

where $a(n) = \sum_{i=0}^{\lfloor \log n \rfloor} b_i$ is the binary weight of n.

Proof.

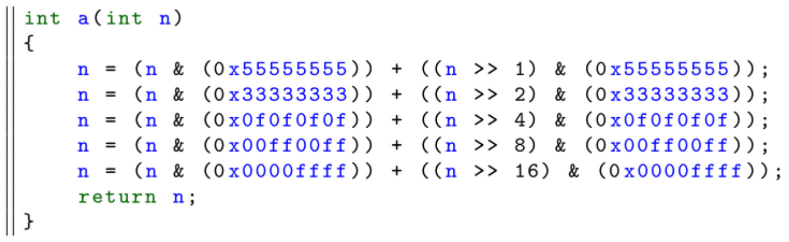
According to Algorithm 6, we can conclude that

$h(n) = \sum_{i=0}^{\lfloor \log n \rfloor} b_i - 1 = a(n) - 1$.

Therefore, $s(n) = h(n+1) = a(n+1) - 1$.

The proof is complete. ■

If the number of n can fit into a computer word, then $a(n)$ can be computed in $O(1)$ time. For example, the *C* programs for computing $a(n)$ can be found in [6] as follows.



```
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;
}
```

Therefore, by using formula (5) we can compute $s(n)$ in $O(1)$ time.

Acknowledgement

This work was supported in part by the Quanzhou Foundation of Science and Technology under Grant No.2013Z38, Fujian Provincial Key Laboratory of Data-Intensive Computing and Fujian University Laboratory of Intelligent Computing and Information Processing.

References

[1] Arne 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.