Applied Mathematics & Information Sciences Letters
An International Journal

http://dx.doi.org/10.12785/amisl/030303

On the Maximal Number of Monochrome Nodes in the Dichromatic Balanced Trees

Daxin Zhu [1], *Xiaodong Wang*[2,*] *and Jun Tian*[3]

[1] Faculty of Mathematics & Computer Science, Quanzhou Normal University, 362000 Quanzhou, China
[2] Fujian University of Technology, 350108 Fuzhou , China
[3] Fujian Medical University, 350004 Fuzhou , China

Received: 31 Mar. 2015, Revised: 25 Jun. 2015, Accepted: 26 Jun. 2015
Published online: 1 Sep. 2015

Abstract: In this paper, we investigate the number of nodes of same color in dichromatic balanced trees. We first present a dynamic programming algorithm for computing the maximum number of red internal nodes in a dichromatic balanced trees on n keys in $O(n^2 \log n)$ time. Then the time complexity is improved to $O(n)$. Finally, we can present an $O(n)$ time algorithm using only $O(\log n)$ space based on the special structure of the solution.

Keywords: dichromatic balanced trees, colored internal nodes, dynamic programming, time complexity

1 Introduction

The red-black tree is a data structure which was invented originally in 1972 by Rudolf Bayer[2] with the name 'symmetric binary B-tree'. It is a special kind of binary tree, which can be used in computer science for organizing the comparable data, such as numbers or strings. In his paper 'A Dichromatic Framework for Balanced Trees', Sedgewick and Guibas named the data structure red-black tree in 1978, [4]. In their paper, the red/black color convention and the properties of their trees at length were introduced. A simpler-to-code variant of the data structure was presented by Andersson [1]. This variant of red-black tree was called AA-trees[7]. An AA-tree is very similar to the red-black tree presented previously except that in an AA-tree, its left children may not be red. Sedgewick introduced a more simpler version of the red-black like tree he called it the left-leaning red-black tree[5] in 2008. In the new tree, an unspecified degree of freedom are eliminated in the implementation. In this paper, we can see that for any sequence of operations, the red-black tree can be isometric to either a 2-3 tree or a 2-4 tree,[5].

A red-black tree can be defined as follows. It is a kind of binary search tree with colors. In a node of the red-black tree, one extra bit of storage ie reserved for its color, the color can be either black or red. A red-black tree satisfies the following properties called the red-black properties[5]:

(1) The color of a node can be either red or black.

(2) The color of root is black.

(3) The color for all leaves are black.

(4) The color of the child nodes for every red node must be black.

(5) The number of black nodes on every path of the tree from a given node to its descendant leaves are the same.

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

* Corresponding author e-mail: wangxd@fjut.edu.cn

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 2, we present a dynamic programming algorithm for the problem. We then improve the algorithm to a new $O(n)$ time algorithm in section 3. Based on the structure of the solution we finally come to an $O(n)$ time recursive algorithm using only $O(\log n)$ space. Some concluding remarks are in section 4.

2 An $O(n^2 \log n)$ time algorithm

2.1 Some formulas for easy cases

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, 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,

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

Then, the number of black nodes $b(n)$ must 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, the ratio of red internal nodes to black internal nodes is,

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

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

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

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,

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

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 $\gamma(n,0) \leq \frac{2n+1}{3}$ and $\gamma(n,1) \leq \frac{2n}{3}$. It follows that

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

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

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

2.2 $O(n^2 \log n)$ time dynamic programming solution

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 $\frac{1}{2}\log n \leq j \leq 2\log n$, we have,

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

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

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

Theorem 1 *For each $1 \leq i \leq n$, $\frac{1}{2}\log i \leq j \leq 2\log i$, 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.

(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,

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

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,

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

(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,

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

For the other three cases, we can conclude similarly that

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

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

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

Therefore, we have,

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

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,

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

For the other three cases, we can conclude similarly that

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

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

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

Therefore, we have,

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

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

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

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.

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.

3 An improved $O(n)$ time recursive algorithm

We have computed $r(n)$ and the corresponding red-black trees using Algorithm 1. We can list some pictures of the computed red-black trees with largest number of red

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

for all i, j, k , $0 \le i \le n, 0 \le j \le 2\log n$, and $0 \le k \le 1$ **do**
  $a(i, j, k) \leftarrow 0$
end for
$a(1,1,0) \leftarrow a(2,1,1) \leftarrow a(3,2,0) \leftarrow 1; a(3,1,1) \leftarrow 2$ {boundary condition}
for $i = 4$ to n **do**
  for $j = \frac{1}{2}\log i$ to $2\log i$ **do**
    $\alpha_1 \leftarrow \max_{0 \le t \le i/2} \{a(t, j-1, 1) + a(i-t-1, j-1, 1)\}$
    $\alpha_2 \leftarrow \max_{0 \le t \le i/2} \{a(t, j, 0) + a(i-t-1, j, 0)\}$
    $\alpha_3 \leftarrow \max_{0 \le t \le i/2} \{a(t, j-1, 1) + a(i-t-1, j, 0)\}$
    $\alpha_4 \leftarrow \max_{0 \le t \le i/2} \{a(t, j, 0) + a(i-t-1, j-1, 1)\}$
    $a(i, j, 0) \leftarrow \max\{1 + \alpha_1, a(i, j, 0)\}$
    $a(i, j, 1) \leftarrow \max\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, a(i, j, 1)\}$
  end for
end for
return $\max_{0 \le k \le 1} \{a(n, j, k)\}$
$\frac{1}{2}\log n \le j \le 2\log n$

Algorithm 2 $r(n)$

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

for all i, j, k , $0 \le i \le n, 0 \le j \le 2\log n$, and $0 \le k \le 1$ **do**
  $a(i, j, k) \leftarrow 0$
end for
$a(1,1,0) \leftarrow a(2,1,1) \leftarrow a(3,2,0) \leftarrow 1; a(3,1,1) \leftarrow 2$ {boundary condition}
for $i = 4$ to n **do**
  for $j = \frac{1}{2}\log i$ to $1 + \frac{1}{2}\log i$ **do**
    $\alpha_1 \leftarrow \max_{0 \le t \le i/2} \{a(t, j-1, 1) + a(i-t-1, j-1, 1)\}$
    $\alpha_2 \leftarrow \max_{0 \le t \le i/2} \{a(t, j, 0) + a(i-t-1, j, 0)\}$
    $\alpha_3 \leftarrow \max_{0 \le t \le i/2} \{a(t, j-1, 1) + a(i-t-1, j, 0)\}$
    $\alpha_4 \leftarrow \max_{0 \le t \le i/2} \{a(t, j, 0) + a(i-t-1, j-1, 1)\}$
    $a(i, j, 0) \leftarrow \max\{1 + \alpha_1, a(i, j, 0)\}$
    $a(i, j, 1) \leftarrow \max\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, a(i, j, 1)\}$
  end for
end for
return $\max_{0 \le k \le 1} \{a(n, j, k)\}$
$\frac{1}{2}\log n \le j \le 1 + \frac{1}{2}\log n$

nodes. 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 $\frac{1}{2}\log n$.

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 $O(n^2)$ as follows.

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

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

and the size of its right subtree must be

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

Therefore, the maximal 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 $r(n)$

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

for all i, j, k , $0 \le i \le n, 0 \le j \le 2\log n$, and $0 \le k \le 1$ **do**
  $a(i, j, k) \leftarrow 0$
end for
$a(1,1,0) \leftarrow a(2,1,1) \leftarrow a(3,2,0) \leftarrow 1; a(3,1,1) \leftarrow 2$ {boundary condition}
for $i = 4$ to n **do**
  $t \leftarrow 2^{\lfloor \log i \rfloor - 1} - 1 + \min\{2^{\lfloor \log i \rfloor - 1}, i - 2^{\lfloor \log i \rfloor} + 1\}$
  for $j = \frac{1}{2}\log i$ to $1 + \frac{1}{2}\log i$ **do**
    $\alpha_1 \leftarrow a(t, j-1, 1) + a(i-t-1, j-1, 1)$
    $\alpha_2 \leftarrow a(t, j, 0) + a(i-t-1, j, 0)$
    $\alpha_3 \leftarrow a(t, j-1, 1) + a(i-t-1, j, 0)$
    $\alpha_4 \leftarrow a(t, j, 0) + a(i-t-1, j-1, 1)$
    $a(i, j, 0) \leftarrow \max\{1 + \alpha_1, a(i, j, 0)\}$
    $a(i, j, 1) \leftarrow \max\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, a(i, j, 1)\}$
  end for
end for
return $\max_{0 \le k \le 1} \{a(n, j, k)\}$
$\frac{1}{2}\log n \le j \le 1 + \frac{1}{2}\log n$

The time complexity of Algorithm 3 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 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*

$$d(m) = \begin{cases} h(m) & h(m) \leq 1 \\ 1+d(4m)+d(4m+1)+d(4m+2)+d(4m+3) & h(m) \mod 2 = 1 \\ d(2m)+d(2m+1) & h(m) \mod 2 = 0 \end{cases} \quad (18)$$

where

$$h(m) = \begin{cases} 1+\lfloor \log n \rfloor - \lfloor \log m \rfloor & \frac{m}{2^{\lfloor \log n \rfloor - \lfloor \log m \rfloor}} \leq n \\ \lfloor \log n \rfloor - \lfloor \log m \rfloor & \texttt{otherwise} \end{cases} \quad (19)$$

tree on n keys. The values of $r(n) = d(1)$ can be computed by the following recursive formula.

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 $h(i) \leq 1$, 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 $h(i) > 1$,

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

The proof is complete. ■

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 as the following Algorithm 4.

Algorithm 4 $d(m)$

Input: Integer n, the number of keys in a red-black tree
Output: $d(m)$, the largest number of red nodes in a subtree rooted on node m

1: **if** $h(m) \leq 1$ **then**
2: **return** $h(m)$
3: **else if** $h(m) \mod 2 = 1$ **then**
4: **return** $1+d(4m)+d(4m+1)+d(4m+2)+d(4m+3)$
5: **else**
6: **return** $d(2m)+d(2m+1)$
7: **end if**

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 $O(\log n)$.

4 Concluding remarks

We have suggested a new dynamic programming solution for computing the maximum number of red internal nodes in a dichromatic balanced trees on n keys. The new 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 an $O(n)$ time recursive algorithm using only $O(\log n)$ space.

The smallest number of red internal nodes in a red-black tree on n keys will have similar properties. We will study the problem further.

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 RedCBlack 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.

[8] Shyu S.J., Tsai C.Y., Finding the longest common subsequence for multiple biological sequences by ant colony optimization, *Comput Oper Res* 36(1), 2009, pp. 73-91.

[9] Tang C.Y., Lu C.L., Constrained multiple sequence alignment tool development and its application to RNase family alignment, *J Bioinform Comput Biol* 1, 2003, pp. 267-287.

[10] Tsai Y.T., The constrained longest common subsequence problem, *Inform Process Lett* 88(4), 2003, pp. 173-176.

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.

Xiaodong Wang is currently a professor in Computer Science Department of Quanzhou Normal University and Fuzhou University,China. Has experience in computer science and applied mathematics. The areas of interest are design and analysis of algorithms,exponential-time algorithms for NP-hard problems,strategy game programming.