ELSEVIER

Theoretical Computer Science 262 (2001) 415–435

Theoretical Computer Science

www.elsevier.com/locate/tcs

Parallel algorithms for red–black trees ☆

Heejin Park, Kunsoo Park *

Department of Computer Engineering, Seoul National University, Seoul 151-742, South Korea

Received November 1996; revised March 2000; accepted June 2000
Communicated by A. Apostolico

Abstract

We present parallel algorithms for the following four operations on red–black trees: construction, search, insertion, and deletion. Our parallel algorithm for constructing a red–black tree from a sorted list of n items runs in $O(1)$ time with n processors on the CRCW PRAM and runs in $O(\log\log n)$ time with $n/\log\log n$ processors on the EREW PRAM. Our construction algorithm does not require the assumptions that previous construction algorithms used. Each of our parallel algorithms for search, insertion, and deletion in red–black trees runs in $O(\log n + \log k)$ time with k processors on the EREW PRAM, where k is the number of unsorted items to search for, insert, or delete and n is the number of nodes in a red–black tree.

Keywords: Red–black trees; Balanced search trees; Parallel algorithms; Dictionary operations

1. Introduction

The red–black tree is a balanced binary search tree whose height is $O(\log n)$ and dictionary operations such as search, insertion, and deletion are performed in $O(\log n)$ time in sequential computation, where n is the number of nodes in the red–black tree. Bayer [2] invented "symmetric binary B-trees" and Guibas and Sedgewick [7] called them "red–black trees".

Another well-known balanced search tree is the 2–3 tree [1], which also has height $O(\log n)$ and supports dictionary operations in $O(\log n)$ time. However, there are some important differences between the 2–3 tree and the red–black tree. An internal node of the 2–3 tree has 2 or 3 children, while an internal node of the red–black tree has

☆ This work was supported by the Brain Korea 21 Project.
* Corresponding author. Fax: 00-82-2-886-7589.
E-mail address: kpark@theory.snu.ac.kr (K. Park).

0304-3975/01/$ - see front matter
PII: S0304-3975(00)00287-5

2 children. In the 2–3 tree, items are stored in the leaves, while items are stored in internal nodes in the red–black tree. The leaves of the 2–3 tree have the same depth, but the leaves of the red–black tree may not.

In this paper we consider the following four operations on red–black trees: construction of a red–black tree from a sorted list of n items, search for k unsorted items in a red–black tree, insertion of k unsorted items into a red–black tree, and deletion of k unsorted items from a red–black tree. Let n be the number of nodes in the red–black tree in each of search, insertion, and deletion.

Some parallel algorithms have been developed for the above four operations on balanced search trees. Moitra and Iyengar [9, 10] gave a parallel algorithm for constructing a balanced binary search tree of minimum height, which runs in O(1) time with n processors on the EREW PRAM. Wang and Chen [15] gave two parallel algorithms for constructing the 2–3 tree. One runs in O(1) time with n processors on the CREW PRAM, and the other runs in O(log log n) time with $n/\log\log n$ processors on the EREW PRAM. However, those algorithms in [9, 10, 15] rely on assumptions that are not easy to satisfy. Moitra and Iyengar assumed that the depth of each node is known in advance and Wang and Chen assumed that $\lfloor \log_2 i \rfloor$ and $\lfloor \log_3 i \rfloor$ are computed in O(1) time by a single processor. Paul et al. [11] gave parallel algorithms for search, insertion, and deletion on the 2–3 tree. Each of these algorithms runs in O($\log n + \log k$) time with k processors on the EREW PRAM.

We present parallel algorithms for the four operations on the red–black tree. Our construction algorithm runs in O(1) time with n processors on the CRCW PRAM and in O(log log n) time with $n/\log\log n$ processors on the EREW PRAM. We do not assume that the depth of each node is known in advance or that $\lfloor \log_2 i \rfloor$ is computed in O(1) time by a single processor ($\lfloor \log_3 i \rfloor$ is not relevant to the construction of the red–black tree).

Our parallel algorithms for search, insertion, and deletion on the red–black tree run in O($\log n + \log k$) time with k processors on the EREW PRAM. The time and processor complexities of our algorithms are the same as those of Paul et al. [11] for the 2–3 tree. However, designing parallel algorithms for insertion and deletion on the red–black tree is harder than that on the 2–3 tree. In the 2–3 tree, all items are stored in the leaves and the leaves have the same depth, i.e., all insertions or deletions of items occur only in the bottom level of the 2–3 tree. Hence, after some items are inserted or deleted, rebalancing can be done level by level from bottom to top of the 2–3 tree. In the red–black tree, however, there are two major difficulties:

1. The leaves of the red–black tree may not have the same depth. To rebalance from bottom to top after some items are inserted (resp. deleted), we introduce *itree*(i) (resp. *dtree*(i)) which is a generalization of the red–black tree for parallel insertion (resp. deletion).
2. In the red–black tree, the items are stored in internal nodes, i.e., deletions of items can occur in any place of the red–black tree, which makes parallel deletion particularly harder. To overcome this, we find the predecessors of the items to be deleted

in O(1) time in each of O(log k) iterations of parallel deletion, and swap the items to be deleted with the items in the predecessors.

This paper is organized as follows. In Section 2, we give some notations and definitions for red–black trees. We describe our parallel algorithms for construction, search, insertion, and deletion in Sections 3, 4, 5, and 6, respectively. We conclude in Section 7.

2. Preliminaries

Our model of computation is the parallel random-access machine (PRAM), which is a shared-memory model of parallel computation that consists of a collection of identical processors and a shared memory [8]. Each processor is a RAM working synchronously and communicating via the shared memory. The exclusive-read–exclusive-write (EREW) PRAM does not allow concurrent reads or concurrent writes to a memory location. The concurrent-read–exclusive-write (CREW) PRAM allows only concurrent reads to a memory location. The concurrent-read–concurrent-write (CRCW) PRAM allows both concurrent reads and concurrent writes to a memory location, and it has several variants depending on how concurrent writes are handled. We use the weakest version (called *common* in [6]), in which the concurrent writes are allowed only when all processors are attempting to write the same value.

We first describe the input for each of the four operations. For construction, let $a_1,\ldots,a_n$ be the given sorted list of items and let $p_1,\ldots,p_n$ be the processors. For search, insertion, and deletion, we first sort the given k items by Cole's parallel merge-sort in O(log k) time with k processors on the EREW PRAM [3], and let $a_1,\ldots,a_k$ be the sorted items from smallest to largest and $p_1,\ldots,p_k$ the processors. For simplicity, we assume that no two items are the same.

We give some notations and definitions. Let $root(T)$ denote the root node of a red–black tree T and $item(x)$ denote the item stored in node x. Let $p(x)$ denote the parent of node x and $p^{n+1}(x)$ the parent of $p^n(x)$, $n\geqslant 1$. Let $rchild(x)$ denote the right child of node x and $lchild(x)$ the left child of x. The *successor* of node x is the node with the smallest item larger than $item(x)$. The *predecessor* of node x is the node with the largest item smaller than $item(x)$. Each node x has a space for its item, a bit for its color (red or black), and three pointers to $p(x)$, $lchild(x)$, and $rchild(x)$. If a node does not have a parent or a child, `nil` is stored in the corresponding pointer. We will regard `nil` as a pointer to an *external node* (*leaf*) and the nodes holding items as *internal nodes* (Fig. 2).

A red–black tree is a binary search tree satisfying the following *red–black properties* [5]:

1. Every node is either red or black.
2. Every external node (`nil`) is black.
3. If a node is red, then both its children are black.

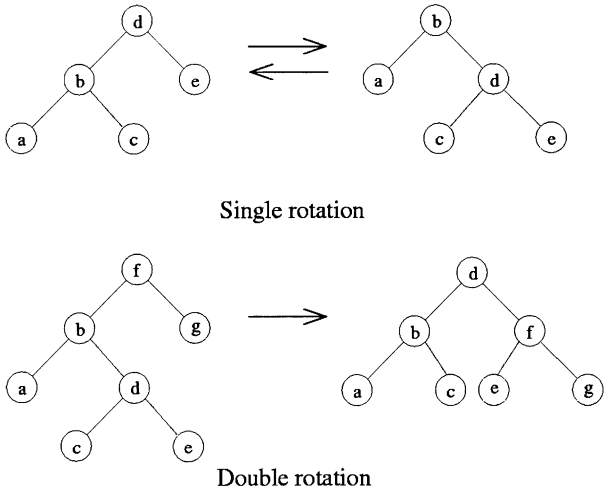


Fig. 1. Rotations. A symmetric variant for double rotation is not shown.

4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

The red–black properties can be rewritten using nonnegative *ranks* instead of red and black colors [14]:

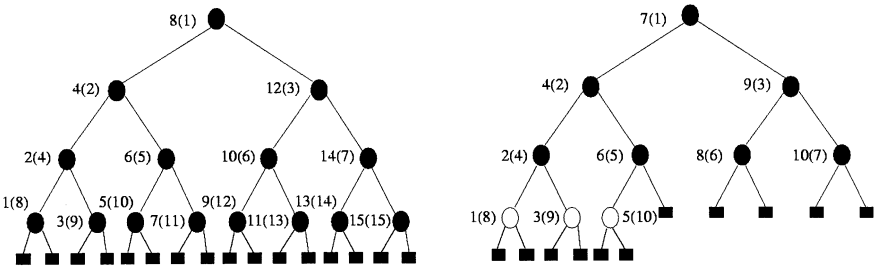
(a) If x is any node with a parent, $rank(x) \leqslant rank(p(x)) \leqslant rank(x) + 1$.
(b) If x is any node with a grandparent, $rank(x) < rank(p^2(x))$.
(c) If x is an external node, $rank(x) = 0$ and $rank(p(x)) = 1$ if x has a parent.

The above conditions (a)–(c) are called *balance conditions*. The rank of node x corresponds to the number of black nodes in any simple path from x to a descendant leaf. Hence, $rank(p(x)) = rank(x) + 1$ if x is black and $rank(p(x)) = rank(x)$ otherwise. Note that $rank(x)$ need not be stored in x. We will use balance conditions rather than red–black properties because balance conditions are more convenient in describing and proving our algorithms.

The atomic operations used for rebalancing red–black trees are *promotion*, *demotion*, *single rotation*, and *double rotation* [14]. A promotion (resp. demotion) of a node is to increase (resp. decrease) the rank of the node by 1. A single rotation and a double rotation are shown in Fig. 1. Each of these atomic operations takes O(1) time.

3. Construction

In this section we give a parallel algorithm for constructing a red–black tree from a sorted list of n items, which runs in O(1) time with n processors on the CRCW



(a) 15 nodes (b) 10 nodes

Fig. 2. Internal nodes are represented by circles and external nodes by rectangles. Dark circles (resp. rectangles) are black internal (resp. external) nodes, and white circles are red internal nodes. The number in the parentheses by a node is the index of the processor that creates the node. The number outside the parentheses is the index of the item stored in the node.

PRAM and in O(log log n) time with $n/\log\log n$ processors on the EREW PRAM. We first describe the algorithm on the CRCW PRAM, and then on the EREW PRAM.

Our parallel construction algorithm on the CRCW PRAM is divided into three parts: (1) build an empty binary tree; (2) store the n items into internal nodes; (3) color each node red or black. The only difficult part in this construction is part (2). In our algorithm, processors are assigned to internal nodes in breadth-first order (Fig. 2), and each processor finds the item to be stored in its node. In Moitra and Iyengar's algorithm [9, 10], the ith processor is assigned to the ith item and each processor computes the destination node of its item:

1. Build an empty binary tree. Processor p_i, $1 \leqslant i \leqslant n$, creates an internal node x_i and two edges between x_i and its children x_{2i} and x_{2i+1}. Note that if $2i > n$ (resp. $2i + 1 > n$), its left (resp. right) child is an external node. Let $D(i)$ be the depth of x_i. Then $D(i) = \lfloor \log i \rfloor$.
2. Processor p_i, $1 \leqslant i \leqslant n$, finds the item to be stored in internal node x_i and stores it in x_i as follows. We first consider the case of a complete binary tree and then the general case:
   - If there are $n = 2^q - 1$ items for some q, we build a complete binary tree. For example, the red–black tree constructed from 15 items is shown in Fig. 2(a). We exploit the fact that the items to be stored in internal nodes of a same depth form an arithmetic progression. See Fig. 2(a). Let $CI(i)$ be the index of the item to be stored in x_i and $J(i)$ the number of nodes whose depths are $D(i)$ and whose items are smaller than the item to be stored in x_i. According to [9], $J(i) = i - 2^{D(i)}$ and $CI(i) = (2J(i) + 1)2^{\lfloor \log n \rfloor}/2^{D(i)}$. For example, if $i = 6$ in Fig. 2(a), then $D(i) = 2$, $J(i) = 2$, and $CI(i) = 10$. Processor p_i stores $a_{CI(i)}$ in x_i.
   - If there are $n \neq 2^q - 1$ items for any q, we build a binary tree with minimum height such that the lengths of the longest path and the shortest path from the root to external nodes differ by 1, and that the deepest internal nodes are located

at the leftmost side of the tree as in Fig. 2(b). We first define $CI(i)$ and $CL(i)$ in the complete binary search tree T' whose depth is the same as the empty binary tree for n. Let x'_i be the ith internal node when we visit T' in breadth-first order. Let $CI(i)$ be the index of the item to be stored in x'_i, and $CL(i)$ the number of the deepest internal nodes in T' whose items are smaller than the item stored in x'_i. Let $L(n)$ be the number of the deepest internal nodes in the empty binary tree for n, and $I(i)$ the index of the item to be stored in x_i. According to [10], $CL(i) = \lfloor CI(i)/2 \rfloor$, $L(n) = n + 1 - 2^{\lfloor \log n \rfloor}$, and $I(i) = \min(CI(i), CI(i) - CL(i) + L(n))$. For example, if $i = 6$ in Fig. 2(b), then $CI(i) = 10, CL(i) = 5, L(n) = 3$, and $I(i) = 8$. Processor p_i stores $a_{I(i)}$ in x_i.

3. Color the internal nodes. If $n = 2^q - 1$, every processor colors its node black. Otherwise, processors whose nodes are the deepest internal nodes (i.e., $2^{\lfloor \log i \rfloor} = 2^{\lfloor \log n \rfloor}$) color their nodes red and other processors color their nodes black.

It is easy to see that all steps above take O(1) time except for computation of $2^{\lfloor \log i \rfloor}$ and $2^{\lfloor \log n \rfloor}$. For these values, we initialize an auxiliary array P such that $P[i] = 2^{\lfloor \log i \rfloor}$ for $1 \leqslant i \leqslant n$. In the appendix, we will show how to compute array P in O(1) time with n CRCW processors. Hence, we can construct a red–black tree from a sorted list of n items in O(1) time with n processors on the CRCW PRAM.

Constructing the red–black tree on the EREW PRAM is similar to that on the CRCW PRAM except that we initialize an additional array N and that each processor constructs O(log log n) internal nodes:

- Array N is initialized such that $N[i] = 2^{\lfloor \log n \rfloor}$ for $1 \leqslant i \leqslant n$. This array is needed to avoid read conflicts on the EREW PRAM. In the appendix we will show how to compute arrays P and N in O(log log n) time using $n/\log\log n$ EREW processors.
- Each processor p_i, $1 \leqslant i \leqslant \lfloor n/\lfloor \log\log n \rfloor \rfloor$, constructs O(log log n) internal nodes. Since processor p_i can build a node n_j, store an appropriate item in it, and color it in O(1) time using $P[j]$ and $N[j]$, a processor can construct O(log log n) nodes in O(log log n) time.

Hence, we construct a red–black tree from a sorted list of n items in O(log log n) time with $n/\log\log n$ processors.

Theorem 1. *Parallel construction of a red–black tree from a sorted list of n items can be done in* O(1) *time with n processors on the CRCW PRAM and in* O(log log n) *time with* $n/\log\log n$ *processors on the EREW PRAM.*

4. Search

Given an item a and a red–black tree T with n nodes, the sequential search algorithm [5, 14] finds out and returns a node of T containing a in O(log n) time. If such a node is not in T, it returns an external node where item a should be inserted. In parallel

search, given k items and a red–black tree T, we return the nodes each of which is the result of sequential search of an item.

Parallel search is easy on the CREW PRAM. Each processor p_i, $1 \leqslant i \leqslant k$, searches for item a_i sequentially. Since no write operations are involved, this algorithm takes O(log n) time on the CREW PRAM. On the EREW PRAM, however, it may cause read conflicts.

On the EREW PRAM, we describe a parallel search algorithm that is similar to the Paul et al. [11] for 2–3 trees. It requires O(log $n+\log k$) time with k processors. Recall that $a_1, \ldots, a_k$ are the sorted items from smallest to largest. Block $b(i,j)$, $1 \leqslant i \leqslant j \leqslant k$, denotes the sorted list of elements $a_i, a_{i+1}, \ldots, a_j$. For block $b(i,j)$, only processor p_i is active among processors $p_i, p_{i+1}, \ldots, p_j$ and it maintains $b(i,j)$. Block $b(i,j)$ is *smaller* (resp. *larger*) than $b(k,l)$ if $j<k$ (resp. $l<i$). Block $b(i,j)$ is a *subblock* of $b(k,l)$ if $k \leqslant i \leqslant j \leqslant l$. We say that block $b(i,j)$ *hits* an internal node x if $a_i \leqslant item(x) \leqslant a_j$. The main part of parallel search proceeds in stages, which are repeated until we find all search results of the items. Initially (stage 0), block $b(1,k)$ is at $root(T)$. In a stage, each block either traverses down the tree (i.e., moves from a node to its child) or it is halved. The invariant of a stage is that the search results of all items in each block are in the subtree rooted at the node where the block is. The following stage is performed simultaneously by each active processor. We denote by x the node where a block $b(i,j)$ is.

- If x is an internal node, there are three cases:
  - $item(x) < a_i$: Processor p_i sets x as $rchild(x)$ (i.e., $b(i,j)$ goes down to $rchild(x)$).
  - $item(x) > a_j$: p_i sets x as $lchild(x)$.
  - $a_i \leqslant item(x) \leqslant a_j$ (i.e., $b(i,j)$ hits x): If $i=j$, p_i returns x. Otherwise, p_i splits $b(i,j)$ into $b(i, \lfloor (i+j)/2 \rfloor)$ and $b(\lfloor (i+j)/2 \rfloor + 1, j)$. Processor p_i keeps $b(i, \lfloor (i+j)/2 \rfloor)$ and activates $p_{\lfloor (i+j)/2 \rfloor + 1}$ to maintain $b(\lfloor (i+j)/2 \rfloor + 1, j)$.
- If x is an external node, there are two cases:
  - p_i has one item in its block $b(i,j)$: p_i returns x.
  - p_i has two or more items in $b(i,j)$: p_i divide $b(i,j)$ into $b(i, \lfloor (i+j)/2 \rfloor)$ and $b(\lfloor (i+j)/2 \rfloor + 1, k)$. Processor p_i keeps $b(i, \lfloor (i+j)/2 \rfloor)$ and activates $p_{\lfloor (i+j)/2 \rfloor + 1}$ to maintain $b(\lfloor (i+j)/2 \rfloor + 1, j)$.

We will show that the above algorithm can be performed on the EREW PRAM by showing that at most 4 processors access the same item at the same time. We say that $b(i, \lfloor (i+j)/2 \rfloor)$ and $b(\lfloor (i+j)/2 \rfloor + 1, j)$ are *generated* at x in stage $l \geqslant 1$ if $b(i,j)$ splits at x in stage l. A block at x in stage l was either generated at x in stage $l-1$ or it arrived at x in stage $l-1$. Since at most one block can hit x at the same time, at most two blocks were generated at x in stage $l-1$. The following lemma shows that at most two blocks arrive at x in a stage, which means that at most 4 processors access the same item at the same time.

Lemma 1. *At most two blocks arrive at a node x in a stage. If two blocks arrive at x in stage l, one of the blocks is smaller and the other is larger than all the blocks*

that arrived at x in stages 1 *to* $l-1$. *If one block arrives at node x in stage l, the block is either smaller or larger than all the blocks that arrived at x in stages* 1 *to* $l-1$.

Proof. We prove this lemma by induction on the depths of nodes in the tree. Since $b(1,k)$ is at $root(T)$ in stage 0 and no blocks arrive at $root(T)$ in other stages, $root(T)$ satisfies the lemma.

Assume that all nodes of depth m satisfy the lemma. Let x be a node of depth $m+1$. A block that arrives at x in stage l either arrived at $p(x)$ or was generated at $p(x)$ in stage $l-1$. We show that x satisfies the lemma by the following two cases:

1. *No blocks were generated at* $p(x)$ *in stage* $l-1$: All blocks that arrive at x in stage l arrived at $p(x)$ in stage $l-1$. Since at most two blocks could arrive at $p(x)$ in stage $l-1$, at most two blocks can arrive at x in stage l. Consider the case that two blocks, b_1 and b_2, arrived at $p(x)$ in stage $l-1$. (The case of one block is similar.) By inductive hypothesis, one of the blocks (say, b_1) is smaller and the other (b_2) is larger than the blocks that arrived at $p(x)$ in stages 1 to $l-2$. Since the blocks that arrived at x in stages 1 to $l-1$ are subblocks of the blocks that arrived at $p(x)$ in stages 1 to $l-2$, b_1 is smaller and b_2 is larger than the blocks that arrived at x in stages 1 to $l-1$.
2. *A block* $b(i,j)$ *hit* $p(x)$ *and two blocks* $b(i,\lfloor(i+j)/2\rfloor)$ *and* $b(\lfloor(i+j)/2\rfloor+1,j)$ *were generated in stage* $l-1$: Consider the case that x is $lchild(p(x))$. (The other case is similar.) Block $b(\lfloor(i+j)/2\rfloor+1,j)$ cannot go down to x because $item(x)\leqslant a_j$. Blocks smaller than $b(\lfloor(i+j)/2\rfloor+1,j)$ may go down to x in stage l. If $b(i,\lfloor(i+j)/2\rfloor)$ does not hit $p(x)$, $b(i,\lfloor(i+j)/2\rfloor)$ arrives at x in stage l. Block $b(i,\lfloor(i+j)/2\rfloor)$ is larger than the blocks that arrived at x in stages 1 to $l-1$ because $b(\lfloor(i+j)/2\rfloor+1,j)$ and any block larger than $b(\lfloor(i+j)/2\rfloor+1,j)$ cannot go down to x. By inductive hypothesis, there is at most one block (say, b) such that b arrived at $p(x)$ in stage $l-1$ and b is smaller than the blocks that arrived at $p(x)$ in stages 1 to $l-2$ (and thus smaller than $b(\lfloor(i+j)/2\rfloor+1,j)$). Block b arrives at x in stage l and it is smaller than the blocks that arrived at x in stages 1 to $l-1$, which are subblocks of the blocks that arrived at $p(x)$ in stages 1 to $l-2$. □

Now, we compute the running time of the above algorithm. It is easy to see that a stage takes O(1) time. In one stage, a block either traverses down the tree or it is halved. Since the depth of a tree is $O(\log n)$ and there are k items, the total time is $O(\log n+\log k)$. Therefore, we get the following theorem.

Theorem 2. *Parallel search for k unsorted items in a red–black tree with n nodes can be done in* $O(\log n+\log k)$ *time with k processors on the EREW PRAM.*

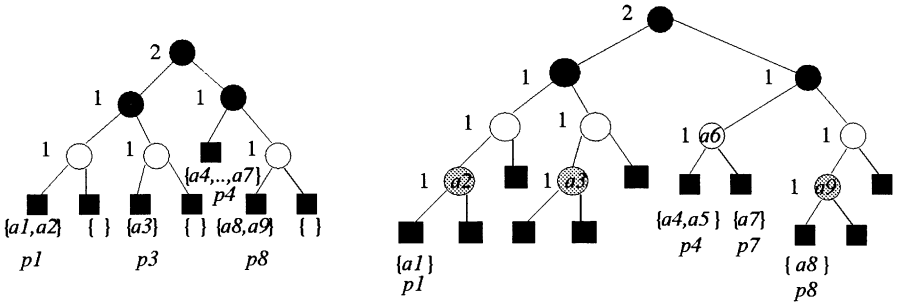


Fig. 3. The number by an internal node is its rank. Shaded nodes are illegal nodes: (a) after stage 2, (b) after stage 3.

5. Insertion

We describe an EREW PRAM algorithm that inserts a set of k items into a red–black tree consisting of n nodes in $O(\log n + \log k)$ time with k processors. A node is *legal* if it satisfies all the balance conditions (Section 2), and *illegal* otherwise. We say that a number of nodes are *incomparable* if a node is not an ancestor or a descendant of any other node. An *itree*(i) is a generalization of the red–black tree such that some nodes of rank i may be illegal. The conditions for *itree*(i) are as follows:

(1) All the nodes except some nodes of rank i are legal.
(2) Some nodes of rank i may be illegal. An illegal node x satisfies $rank(x) = rank(p(x)) = rank(p^2(x)) = i$ and balance condition (c). The color of an illegal node is red.
(3) The illegal nodes are incomparable.

We first give an algorithm running in $O(\log n \log k)$ time and then modify it so that it runs in $O(\log n + \log k)$ time. The $O(\log n \log k)$-time algorithm consists of four stages: (1) search for k items in T; (2) make a block for each external node that some items reached; (3) insert the center item of each block into T; (4) restore T to a red–black tree. Stages 1 and 2 are performed only once but stages 3 and 4 are repeated $O(\log k)$ times.

Stage 1: Perform parallel search with k items in the red–black tree T. Using the parallel search algorithm in the previous section with k sorted items $a_1, \ldots, a_k$, we find k internal or external nodes in $O(\log n + \log k)$ time. Let e_i be the node where a_i reaches. Note that multiple items may reach the same external node, e.g., a_4, a_5, a_6, and a_7 in Fig. 3(a).

Stage 2: For each external node, we make a block consisting of the items that reached the node (Fig. 3(a)). Since the items that reached the same external node are consecutive items, we can represent those items by a block. For example, we represent

a_4, a_5, a_6 and a_7 as a block $b(4,7)$. To make a block, we find the smallest and the largest items for each external node as follows:

1. Compute array FB (resp. LB) such that $FB[i]$ (resp. $LB[i]$), $1 \leqslant i \leqslant k$, is 1 if and only if a_i is the smallest (resp. largest) item in the external node a_i reached. First, p_i, $1 \leqslant i \leqslant k$, initialize $FB[i]$ as 0. Then, p_1 writes 1 in $FB[1]$ if e_1 is an external node and p_i, $2 \leqslant i \leqslant k$, writes 1 in $FB[i]$ if e_i is an external node and $e_{i-1} \neq e_i$. Array LB is computed similarly.
2. Compute array M such that $M[i]$, $1 \leqslant i \leqslant k$, holds the index of the largest item of the ith block. Initially, $M[i]$, $1 \leqslant i \leqslant k$, is 0. Let $\mathtt{psl}(i) = LB[1] + \cdots + LB[i]$. If $LB[i] = 1$, p_i writes i in $M[\mathtt{psl}(i)]$. In Fig. 3(a), $M[3] = 7$ since a_7 is the largest item of the third block.
3. If $FB[i] = 1$, p_i makes a block $b(i, M[\mathtt{psf}(i)])$, where $\mathtt{psf}(i) = FB[1] + \cdots + FB[i]$. In Fig. 3(a), p_4 makes $b(4,7)$ since $\mathtt{psf}[4] = 3$ and $M[3] = 7$. Computing $\mathtt{psl}(i)$ and $\mathtt{psf}(i)$ requires O(log k) time by prefix-sum operations.

We will repeat stages 3 and 4 until no blocks are left at the external nodes. From now on, only processors whose items are the smallest in their blocks are active.

Stage 3: Insert the center item of each block into T. Let $b(i, l_i)$ denote the block at external node e_i and let $c_i = \lceil (i + l_i)/2 \rceil$. Processor p_i creates an internal node x of rank 1 (i.e., a red node), stores a_{c_i} into x, and makes x a child of $p(e_i)$ instead of e_i. If $i < c_i$, p_i moves $b(i, c_i - 1)$ to $lchild(x)$ and activates p_{c_i} to maintain a_{c_i}. If $c_i < l_i$, p_i moves $b(c_i + 1, l_i)$ to $rchild(x)$ and activates p_{c_i+1} to maintain $b(c_i + 1, l_i)$. See Fig. 3(b).

We refer to the operations performed in stage 3 as an *attachment*. It is easy to see that an attachment is done in O(1) time. A tree after an attachment is depicted in Fig. 3(b), where all illegal nodes are of rank 1. Lemma 2 shows that it is generally true.

Lemma 2. *After an attachment, T is a red–black tree or an itree*(1).

Proof. If no illegal nodes exist after an attachment, T is a red–black tree. Otherwise, T is an *itree*(1) as follows. We first show that T satisfies condition (1) for *itree*(1). Every node x which was in T before an attachment remains legal after an attachment because $rank(x)$, $rank(p(x))$, and $rank(p^2(x))$ are not changed by an attachment. Hence, illegal nodes, if any, are some of the internal nodes of rank 1 created by an attachment.

Let Y denote the set of all illegal nodes and y be an element of Y (i.e., $rank(y) = 1$). It is easy to see that T satisfies condition (3) for *itree*(1) and balance condition (c). Since $p(y)$ was the parent of an external node before an attachment, $rank(p(y)) = 1$. Since $rank(p(y)) = 1$ and $p(y)$ is legal, $rank(p^2(y)) = 1$ or 2. Since y is illegal, $rank(p^2(y)) = 1$, i.e., y satisfies condition (2) for *itree*(1). □

Stage 4: Restore T to a red–black tree. We show how to convert an *itree*(i) to an *itree*($i+1$) in O(1) time, which implies that an *itree*(1) can be restored to a red–black

tree in O(log n) time. To convert an *itree*(i) to an *itree*($i+1$), we apply *rebalancing steps* to all subtrees rooted at the grandparents of illegal nodes. We first show that the grandparents of illegal nodes are black nodes of the same rank.

Lemma 3. *In an itree*(i), $i \geqslant 1$, *the grandparents of illegal nodes are black nodes of rank i.*

Proof. Let x be an illegal node of rank i. The rank of $p^2(x)$ is i by definition of *itree*(i) and the rank of $p^3(x)$ is $i+1$ because $p(x)$ is legal. Hence, $p^2(x)$ is a black node of rank i. □

By Lemma 3, the grandparents of illegal nodes are incomparable. Hence, we can apply the rebalancing steps concurrently to all subtrees rooted at the grandparents of illegal nodes in an *itree*(i).

We now describe a rebalancing step in parallel insertion. Let x be a grandparent of some illegal nodes. The rebalancing step has several cases depending on the number of illegal nodes in the subtree rooted at x, and it is performed by the leftmost processor (the processor whose item is the smallest in the subtree).

Rebalancing step (Fig. 4)

Case 1: One node is illegal:

1.1. Two children of x are of rank i: Promote x.

1.2. One child of x is of rank i and the other is of rank $i-1$: Perform a single or double rotation.

Case 2: Two nodes are illegal:

2.1. Two illegal nodes have the same parent: If the sibling of the parent is of rank i, promote x; otherwise (the sibling of the parent is of rank $i-1$), perform a rotation and promote x.

2.2. Two nodes have different parents: Promote x.

Case 3: Three or four nodes are illegal: Promote x.

Let $RBI(i)$ denote a concurrent application of rebalancing steps to all subtrees rooted at the grandparents of illegal nodes in an *itree*(i). Lemma 4 shows that $RBI(i)$ converts an *itree*(i) to an *itree*($i+1$) or a red–black tree. In stage 4, we perform $RBI(1)$, $RBI(2)$, ... until T becomes a red–black tree.

Lemma 4. *If we perform* $RBI(i)$ *in* T *which is an itree*(i), T *becomes a red–black tree or an itree*($i+1$).

Proof. If no illegal nodes exist in T after $RBI(i)$, T is a red–black tree. Otherwise, T is an *itree*($i+1$) as follows. Since all illegal nodes are of rank $i+1$ and incomparable after $RBI(i)$, T satisfies conditions (1) and (3) for *itree*($i+1$). As in the proof of Lemma 2, one can show that T satisfies condition (2) for *itree*($i+1$). □

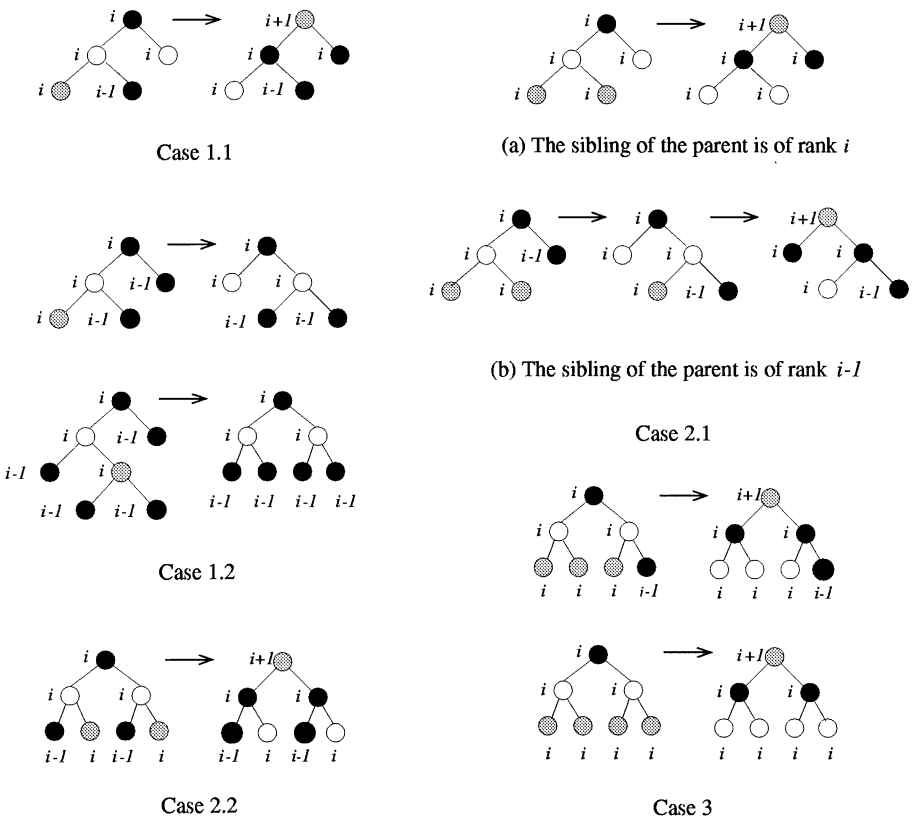


Fig. 4. The rebalancing step in parallel insertion. Shaded nodes in the subtree to the left of an arrow are illegal nodes and those to the right may be illegal according to the ranks of their grandparents. Symmetric variants are not shown.

We now consider the time complexity of parallel insertion. In each iteration of stages 3 and 4, blocks with one item are removed and blocks with two or more items are halved. Thus, no blocks are left after $O(\log k)$ iterations of stages 3–4. Since stage 3 takes $O(1)$ time and stage 4 takes $O(\log n)$ time, a simple implementation of stages 3 and 4 requires $O(\log n \log k)$ time.

We can apply pipelining to stages 3 and 4 to get the time complexity of $O(\log n + \log k)$. Stage 3 and each $RBI(i)$ are performed in $O(1)$ time. $RBI(i)$ may access and update the nodes of rank i, and the edges between nodes of ranks $i-1$ and i, of ranks i and i, and of ranks i and $i+1$. Thus, back-to-back executions of $RBI(i)$'s may cause read/write conflicts. To avoid read/write conflicts, we start stage 3 of the $(j+1)$th iteration at the beginning of $RBI(2)$ of the jth iteration so that $RBI(1)$ of the $(j+1)$st iteration and $RBI(3)$ of the jth iteration are performed concurrently. Since the number of pipeline stages is $O(\log n)$ and stage 3 is performed $O(\log k)$ times, the time complexity of the pipelined algorithm is $O(\log n + \log k)$. Therefore, we get the following theorem.

Theorem 3. *Parallel insertion of k unsorted items into a red–black tree with n nodes can be done in $O(\log n + \log k)$ time with k processors on the EREW PRAM.*

6. Deletion

We describe an EREW PRAM algorithm that deletes a set of k items from a red–black tree consisting of n nodes in $O(\log n + \log k)$ time with k processors. Let $succ(x)$ denote the successor of node x and $pred(x)$ the predecessor of x. A node is *deletable* if at least one of its children is an external node and *undeletable* otherwise. A deletable node is of rank 1 by its definition. A $dtree(i)$ is a generalization of the red–black tree such that some nodes of rank i may be illegal. The conditions for $dtree(i)$ are as follows:

(1) All the nodes except some nodes of rank i are legal.
(2) Some nodes of rank i may be illegal. An illegal node x satisfies $rank(x) + 2 = rank(p(x))$. If x is an external node, x is of rank 0. The color of an illegal node is undefined.
(3) The illegal nodes are incomparable.

As in parallel insertion, we first give an algorithm running in $O(\log n \log k)$ time with k processors and then modify it so that it runs in $O(\log n + \log k)$ time. The $O(\log n \log k)$-time algorithm consists of four stages: (1) search for k items in T and mark the items to be deleted; (2) exchange the marked items in undeletable nodes with the unmarked items in their predecessors; (3) remove all deletable nodes with marked items from T; (4) restore T to a red–black tree. Stage 1 is performed only once but the remaining stages are repeated $O(\log k)$ times.

Stage 1: Perform parallel search with k items in the red–black tree T, and mark the items. A node with a marked item will be called a *marked node*.

Stage 2: For each marked undeletable node x, we find $pred(x)$ and exchange $item(x)$ with $item(pred(x))$ if $pred(x)$ is unmarked. If $pred(x)$ is marked, we do nothing on x in this stage. Since x is undeletable, $pred(x)$ is in the left subtree of x and $pred(x)$ is deletable. Since processors traverse down the tree concurrently starting from distinct nodes in finding their predecessors, no read/write conflicts occur. Note that after stage 2, the number of marked deletable nodes is at least the number of marked undeletable nodes because every marked undeletable node has a marked deletable predecessor.

Stage 3: Remove all marked deletable nodes from T. Since deletable nodes are of rank 1, they are in the subtrees rooted at black nodes of rank 1. In each subtree rooted at a black node of rank 1 that contains marked deletable nodes, the leftmost processor removes the marked deletable nodes and creates a new subtree consisting of the remaining nodes (Fig. 5). Since black nodes of rank 1 are incomparable, we can remove all marked deletable nodes concurrently.

We show that T becomes a $dtree(0)$ after we remove all marked deletable nodes from T. Fig. 5 shows all kinds of new subtrees created. If one or two internal nodes

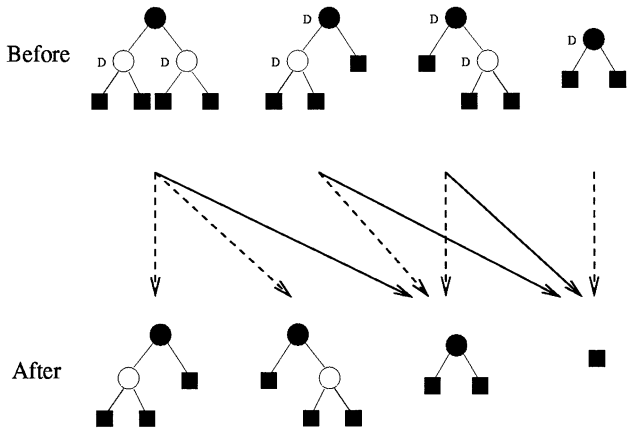


Fig. 5. Removing marked deletable nodes.Deletable nodes are indicated by 'D'. If a subtree contains one marked deletable node, the resulting subtree is pointed to by a dashed arrow. If a subtree contains two marked deletable nodes, the resulting subtree is pointed to by a solid arrow.

are left in a subtree after we remove marked deletable nodes, no illegal nodes are generated. Otherwise (no internal nodes are left), an empty subtree (an external node) is created and the external node is illegal because it is of rank 0 and its parent (which was the parent of a black node of rank 1) is of rank 2. Hence, T satisfies condition (2) for $dtree(0)$. Since all illegal nodes are of rank 0 and incomparable, conditions (1) and (3) for $dtree(0)$ are satisfied.

Stage 4: Restore T to a red–black tree. We show how to convert a $dtree(i)$ to a $dtree(i+1)$ in O(1) time, which implies that a $dtree(0)$ can be restored to a red–black tree in $O(\log n)$ time. To convert a $dtree(i)$ to a $dtree(i+1)$, we apply *rebalancing steps* concurrently first to all subtrees rooted at red nodes of rank $i+2$ and then to all subtrees rooted at black nodes of rank $i+2$. Since nodes of the same color and the same rank are incomparable, we can apply rebalancing steps concurrently.

We describe a rebalancing step in parallel deletion. Let x be a parent of some illegal nodes. The rebalancing step has several cases depending on the number of illegal children of x, and it is performed by the leftmost processor.

Rebalancing step (Fig. 6)

Case 1: One node is illegal:

1.1. The legal child s of x is of rank $i+1$:
  - 1.1a. Both children of s are of rank i: Demote x.
  - 1.1b. The child of s farthest from the illegal node is of rank $i+1$. Perform a single rotation.
  - 1.1c. The child of s farthest from the illegal node is of rank i and the other child of s is of rank $i+1$: Perform a double rotation.

1.2. The legal child of x is of rank $i+2$: Perform a single rotation and proceed as in case 1.1.

Case 2: Two nodes are illegal: Demote x.

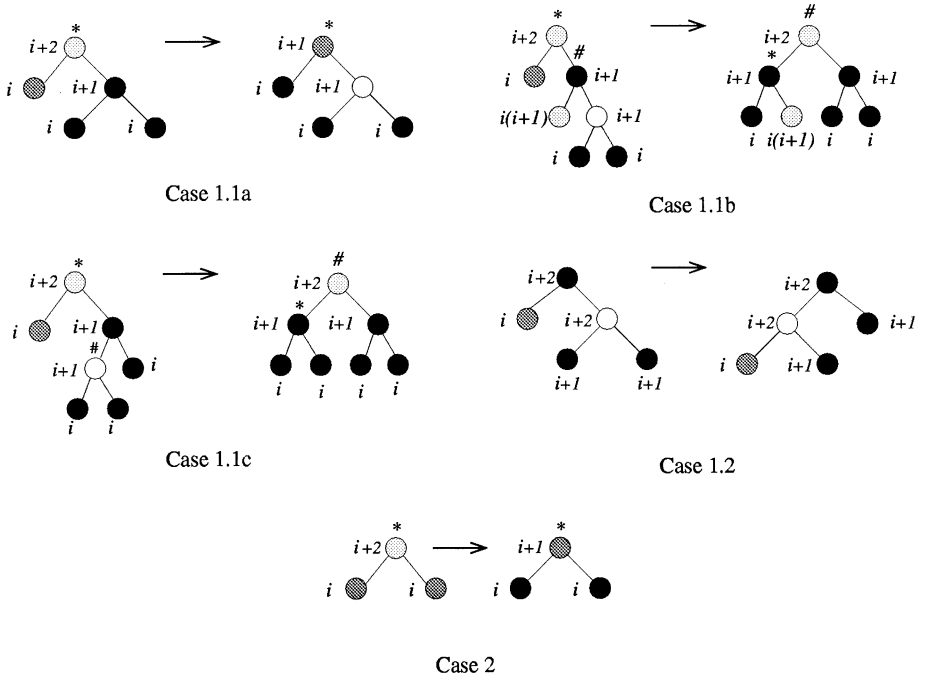


Fig. 6. The rebalancing step in parallel deletion. Heavily shaded nodes to the left of an arrow are illegal nodes and those to the right may be illegal according to the ranks of their parents. Lightly shaded nodes are red or black. Nodes whose ranks are changed from $i+1$ to $i+2$ (resp. from $i+2$ to $i+1$) have $\sharp$'s (resp. $*$'s). Symmetric variants are not shown.

Let $RBD(i)$ denote two concurrent applications of rebalancing steps such that the first is to all subtrees rooted at red nodes of rank $i+2$ that have illegal children and the second is to all subtrees rooted at black nodes of rank $i+2$ that have illegal children. Lemma 5 shows that $RBD(i)$ converts a $dtree(i)$ to a $dtree(i+1)$ or a red–black tree. In stage 4, we perform $RBD(0)$, $RBD(1)$, ... until T becomes a red–black tree.

Lemma 5. *If we perform $RBD(i)$ in T which is a $dtree(i)$, T becomes a red–black tree or a $dtree(i+1)$.*

Proof. In a rebalancing step, cases 1.1b, 1.1c, and 1.2 do not generate any new illegal nodes. The demoted nodes in cases 1.1a and 2 become illegal if they were black before the rebalancing step. Thus, the first concurrent application of rebalancing steps of $RBD(i)$ generates no new illegal nodes and the second one may generate illegal nodes of rank $i+1$. If no illegal nodes are generated, T becomes a red–black tree. Otherwise, one can easily show that T is a $dtree(i+1)$ from the fact that all illegal nodes are of rank $i+1$ and incomparable, and their parents are of rank $i+3$. □

We now consider the time complexity of the above algorithm. In stage 1, the parallel search takes $O(\log n + \log k)$ time. In stage 2, finding predecessors of marked undeletable nodes takes $O(\log n)$ time. It is easy to see that stage 3 takes $O(1)$ time and stage 4 takes $O(\log n)$ time. Since at least one half of marked items are removed in an iteration of stages 2–4, stages 2–4 are repeated $O(\log k)$ times. Overall, $O(\log n \log k)$ time is required.

The above algorithm is hard to improve by applying pipelining because finding predecessors in stage 2 is not easily divided into pipeline stages which should be performed concurrently without read/write conflicts. Hence, we modify the above algorithm so that finding predecessors takes $O(1)$ time (except the first one in the first iteration) and we apply pipelining to the modified algorithm to get the time complexity of $O(\log n + \log k)$.

Before describing the modified algorithm, we list some facts and lemmas on $RBD(i)$. Facts 1 and 2 follow immediately from the description of the rebalancing steps (Fig. 6).

Fact 1. *$RBD(i)$ accesses the nodes of rank i, $i+1$, and $i+2$, and edges between nodes of rank i, $i+1$, $i+2$, and $i+3$.*

Fact 2. *$RBD(i)$ may change a node of rank $i+1$ to a node of rank $i+2$ and a node of rank $i+2$ to a node of rank $i+1$.*

Lemma 6. *If $rank(x) \leqslant 2$ for a node x in a $dtree(0)$, $rank(x) \leqslant 3$ in the $dtree(2)$ which is obtained by applying $RBD(0)$ and $RBD(1)$ to the $dtree(0)$.*

Proof. It follows from Fact 2. □

Lemma 7. *A node x is deletable in a $dtree(2)$ if and only if x is deletable in the red–black tree obtained by applying $RBD(2)$, $RBD(3), \ldots$ to the $dtree(2)$.*

Proof. Since $RBD(i)$ for $i \geqslant 2$ does not access the subtrees rooted at the nodes of rank 1 by Fact 1, a deletable node in $dtree(2)$ is deletable in the red–black tree. By Fact 2, a node whose rank was at least 2 in $dtree(2)$ is still of rank at least 2 after $RBD(i)$ for $i \geqslant 2$ is applied. Hence, an undeletable node in $dtree(2)$ is undeletable in the red–black tree. □

We introduce procedure A that plays an important role in finding predecessors in $O(1)$ time. Let $pred(x)^{i+1}$ denote the predecessor of $pred^i(x)$, $i \geqslant 1$ and $U(x)$ denote $pred^i(x)$ such that all $pred^k(x)$, $k < i$, are marked deletable nodes and $pred^i(x)$ is not a marked deletable node. The input of procedure A is a marked deletable $pred(x)$ where x is a marked undeletable node. Since x is undeletable, the left subtree of x is not empty, and thus it contains $pred(x)$ and $rchild(pred(x))$ is an external node. Procedure A returns $U(x)$ if it is in the left subtree of x and a `nil` pointer otherwise. We describe procedure A in the following. Note that procedure A accesses nodes of rank at most 2 (Fig.7).

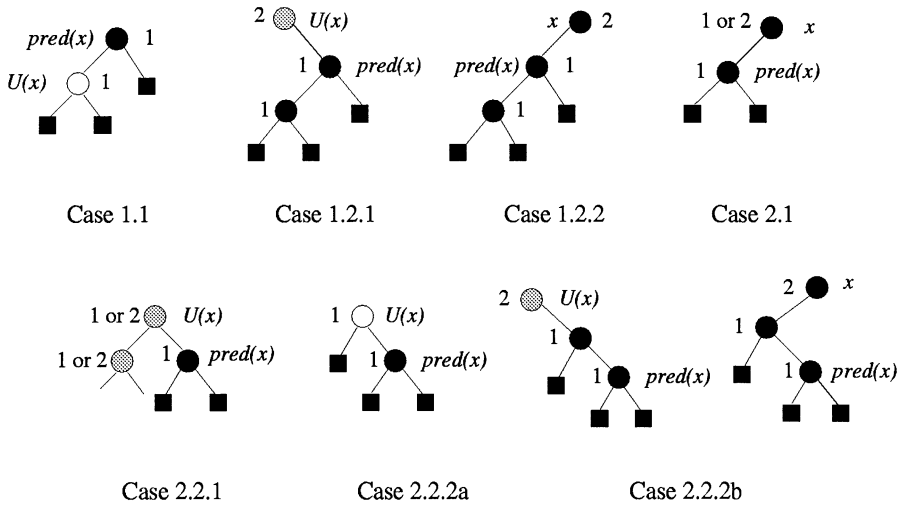


Fig. 7. Dark circles are marked internal nodes and white circles are unmarked internal nodes. Shaded circles are marked or unmarked internal nodes.

Procedure A.

Case 1: $lchild(pred(x))$ is an internal node: Since $pred(x)$ is of rank 1, so is $lchild(pred(x))$. Hence, $lchild(pred(x))$ is deletable and $lchild(pred(x))$ is $pred^2(x)$.

1.1. $lchild(pred(x))$ is unmarked: Return $lchild(pred(x))$ as $U(x)$.

1.2. $lchild(pred(x))$ is marked:

1.2.1. $pred(x)$ is $rchild(p(pred(x)))$: $p(pred(x))$ is $pred^3(x)$. Since $p(pred(x))$ is of rank 2, it is undeletable. Return $p(pred(x))$ as $U(x)$.

1.2.2. $pred(x)$ is $lchild(p(pred(x)))$: $p(pred(x))$ is x, which means that $U(x)$ is not in the left subtree of x. Return a `nil` pointer.

Case 2: $lchild(pred(x))$ is an external node:

2.1. $pred(x)$ is $lchild(p(pred(x)))$: $p(pred(x))$ is x. Return a `nil` pointer.

2.2. $pred(x)$ is $rchild(p(pred(x)))$: $p(pred(x))$ is $pred^2(x)$.

2.2.1. The sibling of $p(pred(x))$ is an internal node: $p(pred(x))$ is undeletable. Return $p(pred(x))$ as $U(x)$.

2.2.2. The sibling of $p(pred(x))$ is an external node: $p(pred(x))$ is deletable.

2.2.2a. $p(pred(x))$ is unmarked: Return $p(pred(x))$ as $U(x)$.

2.2.2b. $p(pred(x))$ is marked: If $p(pred(x))$ is $rchild(p^2(pred(x)))$, $p^2(pred(x))$ is $pred^3(x)$. Since $p^2(pred(x))$ is undeletable, return $p^2(pred(x))$ as $U(x)$; otherwise, return a `nil` pointer because $p^2(pred(x))$ is x.

Lemma 8. *If* $U(x)$ *is not in the left subtree of* x, $rank(x) \leqslant 2$.

Proof. It follows from Cases 1.2.2, 2.1, and 2.2.2b. □

Now, we describe the modified algorithm. The first stage is performed only once but the remaining stages are repeated $O(\log k)$ times.

Stage 1: Perform parallel search with k items in T, and mark the items. For each marked undeletable node x, find $pred(x)$ in $O(\log n)$ time.

Stage 2: For each marked undeletable node x, we exchange $item(x)$ with $item(pred(x))$ if $pred(x)$ is unmarked. If $pred(x)$ is marked, we perform procedure A with $pred(x)$. (Note that x is marked undeletable and $pred(x)$ is marked deletable.) Then, either we get $U(x)$ or $rank(x) \leqslant 2$ by Lemma 8. We need to perform procedure A before stage 3 because once a marked $pred(x)$ is removed in stage 3 we cannot find $U(x)$ from x in $O(1)$ time. In a subtree rooted at a node of rank 2, there may be several procedure A's with distinct $pred(x)$'s. Since the number of procedure A's in the subtree is constant, we perform the procedure A's one by one in $O(1)$ time.

Stage 3: Remove marked deletable nodes from T and perform $RBD(0)$ and $RBD(1)$ in T.

Stage 4: For each marked undeletable node x, find $pred(x)$ in $O(1)$ time as follows. Since x was marked undeletable in stage 2, either we have found $U(x)$ or $rank(x) \leqslant 2$ in stage 2. If we found $U(x)$, $pred(x)$ is $U(x)$ because all marked deletable nodes are removed in stage 3. Otherwise, we find $pred(x)$ in the left subtree of x, which requires $O(1)$ time because $rank(x) \leqslant 3$ by Lemma 6.

Stage 5: Restore T into a red-black tree by performing $RBD(2)$, $RBD(3), \ldots$ in T. Since the undeletable nodes in the red-black tree are the undeletable nodes in stage 4 by Lemma 7, all the marked undeletable nodes in the red-black tree know their predecessors, which is necessary for stage 2 of the next iteration to start.

We now consider the time complexity of the modified algorithm. In stage 1, the parallel search and finding predecessors can be performed in $O(\log n + \log k)$ time. It is easy to see that stages 2–4 take $O(1)$ time and stage 5 takes $O(\log n)$ time. Since stages 2–5 are repeated $O(\log k)$ times, the time complexity of this modified algorithm is still $O(\log n \log k)$.

We pipeline stages 2–5 of the modified algorithm to get the time complexity of $O(\log n + \log k)$. We merge stages 2–4 and refer to them as SM. The pipeline stages are SM and each $RBD(i)$, $i \geqslant 2$, which are performed in $O(1)$ time. The pipeline stages cannot be performed back-to-back because $RBD(i)$ and $RBD(i+1)$ cannot be performed concurrently by Fact 1. We start SM of the $(k+1)$th iteration at the beginning of $RBD(5)$ of the kth iteration. We first show that SM can be performed concurrently with $RBD(i)$, $i \geqslant 5$. $RBD(i)$, $i \geqslant 5$, accesses nodes of rank at least 5 and edges between the nodes of rank at least 5. Most operations in SM (procedure A, $RBD(0)$, $RBD(1)$, and finding predecessors) do not access nodes of rank at least 5 and edges between the nodes of rank at least 5. Only when we exchange items in stage 2, we may access nodes of rank at least 5. But, in this case we access the items in the nodes while $RBD(i)$ accesses the ranks of the nodes. Hence, SM and $RBD(i)$, $i \geqslant 5$, can be performed concurrently without read/write conflicts. It follows from Fact 1 that $RBD(i)$

can be performed concurrently with $RBD(i+j)$, $j \geqslant 4$. Hence, if we start SM of the $(k+1)$st iteration at the beginning of $RBD(5)$ of the kth iteration, read/write conflicts are avoided. Since the number of pipeline stages is $O(\log n)$ and SM is performed $O(\log k)$ times, the time complexity of the pipelined algorithm is $O(\log n + \log k)$. Therefore, we get the following theorem.

Theorem 4. *Parallel deletion of k unsorted items from a red–black tree with n nodes can be done in* $O(\log n + \log k)$ *time with k processors on the EREW PRAM.*

7. Conclusion

We have presented parallel algorithms for four operations on red–black trees which are construction, search, insertion, and deletion. It takes $O(1)$ time by n processors to construct a red–black tree on the CRCW PRAM and $O(\log\log n)$ time by $n/\log\log n$ processors on the EREW PRAM. Each of parallel search, insertion, and deletion takes $O(\log n + \log k)$ time by k processors on the EREW PRAM.

Appendix A. Initialization of arrays P and N

We initialize arrays P and N such that $P[i] = 2^{\lfloor \log i \rfloor}$ and $N[i] = 2^{\lfloor \log n \rfloor}$ for all $1 \leqslant i \leqslant n$. Since $2^{\lfloor \log i \rfloor}$ is the largest power of two smaller than i, it can be obtained from i by converting all 1 bits to 0 bits except the leftmost 1 bit in the bit representation of i. We show how to compute array P in $O(1)$ time with n processors on the CRCW PRAM and how to compute arrays P and N in $O(\log\log n)$ time with $n/\log\log n$ processors on the EREW PRAM.

A.1. CRCW PRAM

Integer $\lfloor \log i \rfloor$ and an entry $P[i] = 2^{\lfloor \log i \rfloor}$ for some $1 \leqslant i \leqslant n$ can be computed in $O(1)$ time with $\log n$ processors as follows. Fich et al. [6] developed a method that finds the location of the leftmost 1 in the bit representation of an integer i in $O(1)$ time with $\log n$ processors on the CRCW PRAM. Finding the location of the leftmost 1 bit in the bit representation of i means computing integer $\lfloor \log i \rfloor$. Once the location of the leftmost 1 bit of i is found, we can compute $P[i]$ from i by converting all 1 bits to 0 bits except the leftmost 1 bit in $O(1)$ time with $\log n$ processors.

If $n \log n$ processors are available, all n entries of P are computed in $O(1)$ time by allocating $\log n$ processors to each entry. To reduce the number of processors from $n \log n$ to n, we first compute $O(n/\log n)$ entries and then compute other entries from the computed entries. The following steps show how to compute array P in $O(1)$ time with n processors.

Step 1: Compute $\lfloor \log n \rfloor$.

Step 2: Compute $P[i]$ for all i's that are multiples of $\lfloor \log n \rfloor$. Processors from $p_{(j-1)\lfloor \log n \rfloor + 1}$ to $p_{j\lfloor \log n \rfloor}$ are allocated to the computation of $P[j\lfloor \log n \rfloor]$, where $1 \leqslant j \leqslant \lfloor n/\lfloor \log n \rfloor \rfloor$.

Step 3: Compute $P[i]$ for all i's that are not multiples of $\lfloor \log n \rfloor$ between $\lfloor \log n \rfloor$ and $\lfloor n/\lfloor \log n \rfloor \rfloor \cdot \lfloor \log n \rfloor$. Let $m_j = j\lfloor \log n \rfloor$ for $1 \leqslant j \leqslant \lfloor n/\lfloor \log n \rfloor \rfloor$. Since $\lfloor \log m_{j+1} \rfloor \leqslant \lfloor \log 2m_j \rfloor = 1 + \lfloor \log m_j \rfloor$, $2^{\lfloor \log m_{j+1} \rfloor}$ is $2^{\lfloor \log m_j \rfloor}$ or $2^{\lfloor \log m_j \rfloor + 1}$ $(= 2P[m_j])$. Processor p_i, $m_j < i < m_{j+1}$, sets $P[i]$ as $P[m_j]$ if $i < 2P[m_j]$; as $2P[m_j]$ otherwise.

Step 4: Compute $P[i]$ for $1 \leqslant i < \lfloor \log n \rfloor$ or $\lfloor n/\lfloor \log n \rfloor \rfloor \lfloor \log n \rfloor < i \leqslant n$. Compute the $O(\log n)$ entries of P using $\log^2 n$ processors.

A.2. EREW PRAM

We first show how to compute $\lfloor \log i \rfloor$ and $2^{\lfloor \log i \rfloor}$ for an integer $1 \leqslant i \leqslant n$ in $O(\log \log n)$ time with a single processor. Since $\lfloor \log i \rfloor$ is a $(\lfloor \log \log n \rfloor + 1)$-bit integer, let $b_{\lfloor \log \log n \rfloor} \cdots b_0$ be the bit representation of $\lfloor \log i \rfloor$ (i.e., $\lfloor \log i \rfloor = \sum_{j=0}^{\lfloor \log \log n \rfloor} b_j 2^j$). Since

$$2^{\lfloor \log i \rfloor} = 2^{\sum_{j=0}^{\lfloor \log \log n \rfloor} b_j 2^j} = \prod_{j=0}^{\lfloor \log \log n \rfloor} 2^{b_j 2^j},$$

we will compute $2^{\lfloor \log i \rfloor}$ by determining b_j's from $b_{\lfloor \log \log n \rfloor}$ to b_0. We first initialize an array Q in step 1 and then compute $2^{\lfloor \log i \rfloor}$ in $\lfloor \log \log n \rfloor + 1$ iterations of steps 2 and 3. At the end of every iteration, we maintain $x = \prod_{j=k+1}^{\lfloor \log \log n \rfloor} 2^{b_j 2^j}$.

Step 1: Initialize array Q such that $Q[j] = 2^{2^j}$ for all $0 \leqslant j \leqslant \lfloor \log \log n \rfloor$ by repeated squaring. Set k as $\lfloor \log \log n \rfloor$ and x as 1.

Step 2: If $x \cdot 2^{2^k}$ $(= x \cdot Q[k]) \leqslant i$, set x as $x \cdot 2^{2^k}$; otherwise, leave x unchanged.

Step 3: Set k as $k - 1$. If $k = -1$, return x $(= 2^{\lfloor \log i \rfloor})$; otherwise, go to step 2.

It is easy to see that step 1 takes $O(\log \log n)$ time and that steps 2 and 3 take $O(1)$ time. Since steps 2 and 3 are repeated $\lfloor \log \log n \rfloor + 1$ times, the total time is $O(\log \log n)$.

Now, we compute array P on the EREW PRAM. We first compute $O(n/\log \log n)$ entries and then compute other entries.

Step 1: Each processor p_i, $1 \leqslant i \leqslant \lfloor n/\lfloor \log \log n \rfloor \rfloor$, computes $\lfloor \log \log n \rfloor$ by repeated squaring.

Step 2: Compute $P[i]$ for all i's that are multiples of $\lfloor \log \log n \rfloor$. Each processor p_j computes $P[j\lfloor \log \log n \rfloor]$, where $1 \leqslant j \leqslant \lfloor n/\lfloor \log \log n \rfloor \rfloor$.

Step 3: Compute $P[i]$ for all i's that are not multiples of $\lfloor \log \log n \rfloor$ between $\lfloor \log \log n \rfloor$ and $\lfloor n/\lfloor \log \log n \rfloor \rfloor \lfloor \log \log n \rfloor$. Let $m_j = j\lfloor \log \log n \rfloor$ for $1 \leqslant j \leqslant \lfloor n/\lfloor \log \log n \rfloor \rfloor$. Processor p_j computes $P[i]$ for all $m_j < i < m_{j+1}$. Since $\lfloor \log m_{j+1} \rfloor \leqslant \lfloor \log 2m_j \rfloor = 1 + \lfloor \log m_j \rfloor$, $2^{\lfloor \log m_{j+1} \rfloor}$ is $2^{\lfloor \log m_j \rfloor}$ or $2^{\lfloor \log m_j \rfloor + 1}$ $(= 2P[m_j])$. Processor p_j sets $P[i]$ as $P[m_j]$ if $i < 2P[m_j]$; as $2P[m_j]$ otherwise.

Step 4: Compute $P[i]$ for $1 \leqslant i < \lfloor \log \log n \rfloor$ or $\lfloor n/\lfloor \log \log n \rfloor \rfloor \lfloor \log \log n \rfloor < i \leqslant n$. Compute the $O(\log \log n)$ entries of P using $\log \log n$ processors.

It is easy to see that steps 1 and 3 take $O(\log\log n)$ time with $n/\log\log n$ processors. Steps 2 and 4 take $O(\log\log n)$ time because $2^{\lfloor \log i \rfloor}$ for some $1 \leqslant i \leqslant n$ is computed in $O(\log\log n)$ time. Hence, array P is computed in $O(\log\log n)$ time with $n/\log\log n$ processors on the EREW PRAM. The computation of array N is similar to (in fact, simpler than) that of array P.

Acknowledgements

We would like to thank the referees for their valuable comments which helped improve the presentation of this paper.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, USA, 1974.
[2] R. Bayer, Data structure and maintenance algorithms, Acta Inform. 1 (1972) 290–306.
[3] R. Cole, Parallel merge sort, SIAM J. Comput. 17 (1988) 770–785.
[4] S.A. Cook, C. Dwork, R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, SIAM J. Comput. 15 (1986) 87–97.
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 1990.
[6] F.E. Fich, P. Ragde, A. Wigderson, Relations between concurrent-write models of parallel computation, SIAM J. Comput. 17 (1988) 606–627.
[7] L.J. Guibas, R. Sedgewick, A dichromatic framework for balanced trees, Proc. FOCS'78, 1978, pp. 8–21.
[8] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, Reading, MA, USA, 1992.
[9] A. Moitra, S.S. Iyengar, A maximally parallel balancing algorithm for obtaining complete balanced binary trees, IEEE Trans. Comput. 34 (6) (1985) 563–565.
[10] A. Moitra, S.S. Iyengar, Derivation of a parallel algorithm for balancing binary trees, IEEE Trans. Software Eng. 12 (3) (1986) 442–449.
[11] W. Paul, U. Vishkin, H. Wagener, Parallel dictionaries on 2–3 trees, in: Proc. ICALP'83, Lecture Notes in Comput. Sci., vol. 154, Springer, Berlin, 1983, pp. 597–609.
[12] R. Sedgewick, Algorithms in $C++$, Addison-Wesley, Reading, MA, USA, 1992.
[13] R.E. Tarjan, Updating a balanced search tree in O(1) rotations, Inf. Process. Lett. 16 (1983) 253–257.
[14] R.E. Tarjan, Data Structures and Network Algorithms, SIAM, Philadelphia, PA, USA, 1983.
[15] B.F. Wang, G.H. Chen, Cost-optimal parallel algorithms for constructing 2-3 trees, J. Parallel Distributed Comput. 11 (1991) 257–261.