Deleting Keys of B-trees in Parallel

Heejin Park
hjpark@theory.snu.ac.kr

Kunsoo Park
kpark@theory.snu.ac.kr

Yookun Cho
cho@cse.snu.ac.kr

School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea

Abstract

The B-tree is a fundamental data structure that is used to access and update a large number of keys. In this paper we present a parallel algorithm on the EREW PRAM that deletes keys in a B-tree. Our algorithm runs in $O(t(\log k + \log n))$ time with $k$ processors, where $n$ is the number of keys in the B-tree, $t$ is the minimum degree of the B-tree, and $k$ is the number of unsorted keys to delete, and it improves upon the previous algorithm by a factor of $t$.

Keywords: B-trees, balanced search trees, parallel algorithms, dictionary operations

1. Introduction

B-trees are balanced search trees that are designed to minimize disk I/O operations and thus they are widely used in practice when the number of keys is so large that the keys need be stored in secondary storage devices. The height of a B-tree 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 keys in the B-tree and $t$ is the minimum degree of the B-tree. Other well-known balanced search trees are 2-3 trees [1] and red-black trees [2, 4]. Both of them have height $O(\log n)$ and support dictionary operations in $O(\log n)$ time.

There has been extensive research on accessing and updating balanced search trees concurrently in both synchronous and asynchronous environments. In the asynchronous case, emphasis is put on guaranteeing correctness of concurrent operations in the presence of asynchrony [7, 8, 9, 14, 15, 16, 19]. In the synchronous case, however, achieving fast algorithms is more emphasized [5, 10, 11, 12, 13, 17, 18]. In this paper we focus on fast algorithms for dictionary operations on B-trees in a synchronous environment.

For B-trees, Higham and Schenk [5] achieved $O(t(\log k + t \log n))$ time for search, $O(t(\log k + \log n))$ time for insertion, and $O(t^2(\log k + \log n))$ time for deletion with $k$ processors on the EREW PRAM, where $k$ is the number of keys to search, insert, and delete. In 2-3 trees [13] and red-black trees [12], each of search, insertion, and deletion can be done in $O(\log n + \log k)$ time with $k$ EREW PRAM processors. It is not surprising that the dictionary operations on B-trees are $t/\log t$ times slower than those on 2-3 trees and red-black trees on the EREW PRAM because the dictionary operations on B-trees are $t/\log t$ times slower than those on 2-3 trees and red-black trees in sequential computation. However, Higham and Schenk’s deletion algorithm is $t^2/\log t$ times slower than those on 2-3 trees and red-black trees.

We present a new parallel deletion algorithm on B-trees that runs in $O(t(\log k + \log n))$ time with $k$ processors on the EREW PRAM. Our deletion algorithm is $t/\log t$ times slower than those on 2-3 trees and red-black trees on the EREW PRAM. We first develop a deletion algorithm that runs in $O(t \log k \log n)$ time and then apply pipelining to achieve $O(t(\log k + \log n))$ time. Both our algorithm and Higham and Schenk’s deletion algorithm adopt pipelining to achieve better time complexities. However, every pipelining stage of our algorithm is performed in $O(t^2)$ time while a pipelining stage of Higham and Schenk’s algorithm is performed in $O(t^3)$ time.

This paper is organized as follows. In Section 2 we define B-trees and some intermediate trees that are used in our algorithm. In Section 3 we describe our parallel deletion algorithm.

2. Preliminaries

A B-tree of minimum degree $t \geq 2$ is a rooted tree having the following six properties. Properties 1-4 determine
the shape of a B-tree such as the number of keys in a node and the number of children of a node, and properties 5 and 6 determine the way keys are stored in a B-tree.

1. Every node has at most \(2t - 1\) keys.
2. Every node except the root has at least \(t - 1\) keys.
3. Every internal node has \(s + 1\) children if it has \(s\) keys.
4. Every leaf has the same depth.
5. The keys of a node are stored in increasing order.
6. The \(i\)th key of a node is larger than any key stored in the subtree rooted at the \(i\)th child of the node and smaller than any key stored in the subtree rooted at the \((i + 1)\)st child of the node.

Figure 1 (a) shows an example of a B-tree of minimum degree 3 with 22 keys.

We will define intermediate trees that are maintained in our deletion algorithm. A \(B'(i)\)-tree of minimum degree \(t \geq 2\) is a rooted tree having the following seven properties. Properties 1-4 determine the shape of a \(B'(i)\)-tree and properties 5-7 determine the way keys are stored in a \(B'(i)\)-tree. Properties 1-4 are the same as those of the B-tree and properties 5-7 are as follows.

5. Each key is either marked or unmarked.
6. The unmarked keys of a node are stored in increasing order.
7. The \(i\)th key of a node that is unmarked, is larger than any unmarked key stored in the subtree rooted at the \(i\)th child of the node and smaller than any unmarked key stored in the subtree rooted at the \((i + 1)\)st child of the node.

Figure 1 (b) shows an example of a \(B'(i)\)-tree of minimum degree 3 with 2 marked keys and 20 unmarked keys. It should be noted that this tree is not a B-tree because key 22 violates B-tree property 5 and key 7 violates B-tree property 6. However, since keys 7 and 22 are marked, they violate no \(B'(i)\)-tree properties. It is easy to see that if a tree \(T\) satisfies \(B'(i)\)-tree properties and no records are marked in \(T\), \(T\) is a B-tree. Since every leaf has the same depth in a \(B'(i)\)-tree, every simple path from a node to a descendant leaf contains the same number of nodes. A node is in level \(i\) if every simple path from the node to a descendant leaf contains \(i + 1\) nodes. For example, a leaf is in level 0 and the parent of the leaf is in level 1.

A \(B'(i)\)-tree, \(i \geq 0\), is a variant of the \(B'(i)\)-tree which allows a node in level \(i\) to have less than \(t - 1\) keys. A \(B'(i)\)-tree has seven properties. Properties 1 and 3-7 are the same as those of the \(B'(i)\)-tree and property 2 is as follows.

2. Every node in level \(i\) has at least \(\lceil (t - 1)/2 \rceil\) keys and every node in the other levels except the root has at least \(t - 1\) keys.

In a \(B'(i)\)-tree, an \textit{insufficient} node is a node having at most \(t - 2\) keys and a \textit{sufficient} node is a node having at least \(t - 1\) keys. In a \(B'(i)\)-tree, nodes in level \(i\) are either sufficient or insufficient and all nodes in the other levels are sufficient.

We describe how to represent a node in B-trees, \(B'(i)\)-trees, and \(B'(i)\)-trees. We represent a node \(v\) with \(l\) keys as a doubly-linked list of \(l\) records, \(r_1(v)\) through \(r_l(v)\) [5]. Let \(c_i(v)\) denote the \(i\)th child of node \(v\) and \(p(v)\) the parent of node \(v\). Each record \(r_i(v), 1 \leq i \leq l\), has three pointers such that the first pointer points to the first record of \(c_i(v)\), the second points to its previous record \(r_{i-1}(v)\) if \(1 < i \leq l\) and the last record of \(p(v)\) otherwise, and the third points to its next record \(r_{i+1}(v)\) if \(1 \leq i < l\) and the first record of \(c_{i+1}(v)\) otherwise.

A B-tree \(T\) (also \(B'(i)\)-tree and \(B'(i)\)-tree) can be reinterpreted as a binary tree if we consider records in each node in \(T\) as nodes and consider the pointers of each record as edges. We define \textit{inorder traversal} in \(T\) as the inorder traversal in the binary tree interpretation of \(T\). The \textit{inorder} of keys in \(T\) denotes the order of keys in \(T\) obtained by inorder traversal in \(T\). If we perform inorder traversal in a B-tree, we get a sorted list of all keys in the B-tree by B-tree properties 5 and 6. If we perform inorder traversal in a \(B'(i)\)-tree (also \(B'(i)\)-tree), the unmarked keys are stored in increasing order in the list obtained by inorder traversal by \(B'(i)\)-tree properties 6 and 7. Conversely, if unmarked keys are stored in increasing order in the list obtained by inorder traversal in \(T\), \(T\) satisfies \(B'(i)\)-tree properties 6 and 7.

Let key\((x)\) denote the key in record \(x\) and node\((x)\) the node that contains record \(x\). The \textit{successor}, denoted by succ\((x)\), of record \(x\) is the record which we visit right after \(x\) when we perform inorder traversal in \(T\). The successor of \(r_i(x)\) of internal node \(x\) is the first record of the leftmost leaf in the subtree rooted at \(c_{i+1}(x)\). An \textit{internal record} is a
record contained in an internal node and an internal key is a key in an internal record. A record \( x \) is in level \( i \) if \( \text{node}(x) \) is in level \( i \).

Our model of computation is the PRAM (parallel random-access machine), which is a shared-memory model of parallel computation that consists of a collection of identical processors and a shared memory [6]. Each processor is a RAM working synchronously and communicating via the shared memory. The EREW (exclusive-read exclusive-write) PRAM does not allow concurrent reads or concurrent writes to a memory location.

### 3. Deletion

We describe a parallel algorithm that deletes a set of \( k \) keys from a B-tree \( T \) in \( O(t(\log k + \log_{\text{st}} n)) \) time with \( k \) processors, where \( n \) is the number of keys in \( T \) and \( t \) is the minimum degree of \( T \). We first give an algorithm running in \( O(t(\log k \log_{\text{st}} n)) \) time with \( k \) processors and then modify it so that it runs in \( O(t(\log k + \log_{\text{st}} n)) \) time.

The \( O(t(\log k \log_{\text{st}} n)) \)-time algorithm consists of the following eight stages.

1. Search for \( k \) keys in \( T \) and mark the records holding the keys that are found.
2. Find the successors of the marked internal records.
3. Exchange the keys in marked internal records with the keys in unmarked records in leaves.
4. Find the successors of the successors of the marked records in level at least 2 if \( t = 2 \).
5. Remove a half or more of marked records in leaves so that \( T \) becomes a B'(0)-tree.
6. Convert B'(0)-tree \( T \) to a B'(1)-tree.
7. Find the successors of marked records in level 1.
8. Restore \( T \) to a B'-tree.

Stages 1 and 2 are performed only once and stages 3-8 are repeated \( O(\log k) \) times. After \( O(\log k) \) iterations of stages 3-8, no records remain marked in \( T \) and thus \( T \) becomes a B-tree. We now explain each stage in detail.

**stage 1.** Perform parallel search with \( k \) keys in B-tree \( T \) and mark the records holding the found keys: A key in a marked record will be called a marked key. After the parallel search, a processor is assigned to every marked record. Let \( p_x \) denote the processor assigned to record \( x \). We can perform this stage in \( O(\log k + t \log_{\text{st}} n) \) time with \( k \) processors using the parallel search algorithm for B-trees due to Higham and Schenk [5]. Since \( T \) was a B-tree before stage 1 and we just marked some keys in \( T \) in this stage, \( T \) is a B'-tree after stage 1.

**stage 2.** For each marked internal record \( x \) in B'-tree \( T \), \( p_x \) finds \( \text{succ}(x) \): Let us assume that \( x \) is the \( i \)th record of node \( v \). To find \( \text{succ}(x) \), \( p_x \) traverses down the subtree rooted at \( c_{i+1}(v) \) until it reaches the first record of the leftmost leaf in the subtree, which takes \( O(\log_{\text{st}} n) \) time. Processors assigned to marked internal records can find their successors concurrently without any read/write conflicts because they start traversing at distinct internal records. Hence, this stage takes \( O(\log_{\text{st}} n) \) time.

**stage 3.** Exchange some marked internal keys with unmarked keys in leaves in B'-tree \( T \): For each marked internal record \( x \), \( p_x \) finds the leftmost unmarked record \( x' \) in \( \text{node}(\text{succ}(x)) \) in \( O(t) \) time. If such record \( x' \) exists, \( p_x \) exchanges \( \text{key}(x) \) with \( \text{key}(x') \), unmarks \( x \), marks \( x' \), and is assigned to \( x' \). Otherwise (if all records in \( \text{node}(\text{succ}(x)) \) are marked), we do nothing on \( x \) and \( x \) remains marked. Processors assigned to marked internal records can perform this stage concurrently because no two successors of internal records share a node by Lemma 1. Thus, this stage can be performed in \( O(t) \) time. Note that after we exchange keys, every marked internal record has a marked successor in a leaf and thus the number of marked records in leaves is at least the number of marked internal records.

We show that \( T \) remains as B'-tree after we exchange keys. Consider the list obtained by inorder traversal in \( T \) before we exchange keys. Since \( x' \) is the leftmost unmarked record in \( \text{node}(\text{succ}(x)) \), \( \text{key}(x) \) and keys between \( \text{key}(x) \) and \( \text{key}(x') \) in the list were all marked and thus exchanging \( \text{key}(x) \) with \( \text{key}(x') \) does not change the order of unmarked keys in the list. Hence, \( T \) remains as B'-tree.

We now assume that for each marked internal record \( x \) at the beginning of stage 3, \( p_x \) knows \( \text{succ}(x) \). As we described before, stage 3 is repeated \( O(\log k) \) times. In the first iteration, this stage is performed after stage 2 and thus this assumption is satisfied. In most cases, \( \text{succ}(x) \) in stage 3 of the \( i \)th, \( i \geq 1 \), iteration is \( \text{succ}(x) \) in stage 3 of the \((i + 1)\)st iteration. For some \( x \), however, \( \text{succ}(x) \) in stage 3 of the \( i \)th iteration is removed in stage 5 of the \( i \)th iteration and is no more \( \text{succ}(x) \) in stage 3 of the \((i + 1)\)st iteration. For such \( x \), we need to find a record which will be \( \text{succ}(x) \) in stage 3 of the next iteration. We do it in stage 4.

**stage 4.** Find \( \text{succ}(\text{succ}(x)) \) in B'-tree \( T \) for each marked record \( x \) in level at least 2 if \( \text{node}(\text{succ}(x)) \) has only one record \( \text{succ}(x) \): If \( \text{node}(\text{succ}(x)) \) has only one record \( \text{succ}(x) \), \( \text{succ}(x) \) will be removed in stage 5 and \( \text{succ}(\text{succ}(x)) \) found in this stage will be the successor of \( x \). This stage is performed only if \( t = 2 \), because a node of a B'-tree may have only one record only if \( t = 2 \). Since the minimum degree of a B-tree is in general much larger than 2, this stage is not necessary in such cases. The details
of finding \( \text{succ}(\text{succ}(x)) \) in \( O(1) \) time are omitted. Since we do no write operations on \( T \) in this stage, \( T \) remains as \( B' \)-tree.

**Stage 5.** Remove a half or more of marked records in each leaf in \( B' \)-tree \( T \): In each leaf, some records are marked and the others are unmarked. This stage is composed of two steps. In a sequence of elements, an element is of rank \( i \) if the element is the \( i \)th leftmost element in the sequence. In step 1, we count the number of marked records in each leaf \( v \), denoted by \( b(v) \), and compute the ranks of marked records in \( v \). The processor assigned to the leftmost marked record performs step 1 in \( O(t) \) time. In step 2, we remove \( \lfloor b(v)/2 \rfloor \) marked records for each leaf \( v \). The processor assigned to the \( i \)th marked record for \( i \geq \lfloor b(v)/2 \rfloor + 2 \) terminates. The processor assigned to the \( i \)th marked record for \( i \leq \lfloor b(v)/2 \rfloor \) does nothing until stage 7 starts. The processor assigned to the \((\lfloor b(v)/2 \rfloor + 1)\)st marked record will be called the center processor of \( v \). The center processor of \( v \) removes the \( i \)th marked record for all \( i \geq \lfloor b(v)/2 \rfloor + 1 \) and checks if \( t-1 \) or more keys are left in \( v \). If so, the center processor terminates. Otherwise, the center processor remains alive and will be used in stage 6. Since removing marked records in leaves can be performed concurrently by the center processors of leaves, step 2 takes \( O(t) \) time and thus this stage takes \( O(t) \) time.

We show that \( T \) is a \( B'(0) \)-tree after stage 5. Stage 5 is performed after stage 3 if \( t \geq 2 \) and after stage 4 otherwise. In both cases, \( T \) was a \( B' \)-tree at the beginning of stage 5. Since we have removed only some records in leaves in this stage, \( T \) still satisfies \( B' \)-tree properties 1 and 3-7 after stage 5, which are the same as \( B'(0) \)-tree properties 1 and 3-7. We show that \( T \) satisfies \( B'(0) \)-tree property 2 by showing that each leaf has at least \((t-1)/2\) keys. Let \( l(v) \) denote the number of records in leaf \( v \) at the beginning of stage 5. Since we have removed \( \lfloor b(v)/2 \rfloor \) keys from \( v \) in this stage, \( l(v) - \lfloor b(v)/2 \rfloor \) keys are left in \( v \) after stage 5. Since \( l(v) - \lfloor b(v)/2 \rfloor \) is smallest when \( b(v) = l(v) \), at least \( \lfloor l(v)/2 \rfloor \) (i.e. \( l(v) - \lfloor l(v)/2 \rfloor \)) keys are left in \( v \). Since \( T \) was a \( B' \)-tree at the beginning of stage 5, \( l(v) \geq t-1 \) and thus \( v \) has at least \((t-1)/2\) keys after stage 5. Hence, \( T \) is a \( B'(0) \)-tree after stage 5.

We show that for each marked record \( x \) in level at least 2, \( p_x \) knows the successor of \( x \) after stage 5. At the beginning of stage 5, \( p_x \) knew \( \text{succ}(x) \). Since \( x \) is a marked record, all records in node \( \text{succ}(x) \) were marked at the beginning of stage 5. If \( \text{node}(\text{succ}(x)) \) had two or more records, \( \text{succ}(x) \) which was the first record in node \( \text{node}(\text{succ}(x)) \) is not removed. In this case, \( \text{succ}(x) \) will be the successor of \( x \) after stage 5. Otherwise (if \( t = 2 \) and \( \text{node}(\text{succ}(x)) \) had only one record \( \text{succ}(x) \) at the beginning of stage 5), \( \text{succ}(x) \) is removed in this stage. The record \( \text{succ}(\text{succ}(x)) \) found in stage 4 will become the successor of \( x \) because \( \text{succ}(\text{succ}(x)) \) is \( r_1(p(\text{succ}(x))) \) and thus \( \text{succ}(\text{succ}(x)) \) is not removed in this stage.

**Stage 6.** Convert \( B'(0) \)-tree \( T \) to a \( B'(1) \)-tree: We will show how to generally convert a \( B'(i) \)-tree \( T \) to a \( B'(i+1) \)-tree in \( O(t) \) time. To convert a \( B'(i) \)-tree to a \( B'(i+1) \)-tree, we apply **rebalancing steps** to all the subtrees rooted at the parents of insufficient nodes in level \( i \). We can apply the rebalancing steps concurrently to all the subtrees because the parents of the insufficient nodes are in level \( i+1 \) and thus all the subtrees are disjoint.

We now describe the rebalancing step. Let \( v \) denote the parent of an insufficient node. The rebalancing step is performed by the center processors of the insufficient children of \( v \). We first show how to rebalance two adjacent children of \( v \) where one or both of them are insufficient and then we describe the rebalancing step for the general case.

Rebalancing two adjacent children of \( v \) is as follows. Let \( c_j(v) \) and \( c_{j+1}(v) \) be the two adjacent children of \( v \) where one of them is insufficient and the other is sufficient or insufficient. Since each of \( c_j(v) \) and \( c_{j+1}(v) \) has at least \((t-1)/2\) keys and at most \( 2t-1 \) keys by \( B'(i) \)-tree properties 1 and 2, the total number of keys in \( c_j(v) \) and \( c_{j+1}(v) \) is at least \( t-2 \) and at most \( 4t-2 \). Rebalancing two adjacent children of \( v \) has two cases depending on the total number of keys in \( c_j(v) \) and \( c_{j+1}(v) \).

- If the total number of keys in \( c_j(v) \) and \( c_{j+1}(v) \) is \( 2t-3 \) or less, we merge \( c_j(v) \) and \( c_{j+1}(v) \) into \( c_{j+1}(v) \) and attach the children of \( c_j(v) \) to the merged \( c_{j+1}(v) \) such that the inorder of keys in the subtree rooted at \( v \) is preserved. Since the total number of keys in \( c_j(v) \) and \( c_{j+1}(v) \) is at least \( t-2 \) and at most \( 2t-3 \), the merged \( c_{j+1}(v) \) has at least \( t-1 \) keys and at most \( 2t-2 \) keys and thus the merged \( c_{j+1}(v) \) is sufficient. It should be noted that we decreased the number of keys in \( v \) by 1.

- Otherwise (if the total number of keys in \( c_j(v) \) and \( c_{j+1}(v) \) is \( 2t-2 \) or more), we make both of \( c_j(v) \) and \( c_{j+1}(v) \) have \( t-1 \) or more keys by moving keys in \( c_j(v) \), \( r_j(v) \), and \( c_{j+1}(v) \), and reattaching the children of \( c_j(v) \) and \( c_{j+1}(v) \) such that the inorder of keys in the subtree rooted at \( v \) is preserved. In this case, \( c_j(v) \) and \( c_{j+1}(v) \) are sufficient and the number of keys in \( v \) is not changed.

Rebalancing two adjacent children of \( v \) is similar to the rebalancing performed after we remove a key from a \( B \)-tree [3] and thus rebalancing two adjacent children of \( v \) can be performed in \( O(t) \) time by a processor.

We now describe the rebalancing step for the general case. We first count the number of children of \( v \) and compute the ranks of the children of \( v \) in \( O(t) \) time. The rebalancing step has three cases. Case 1 is when \( v \) has an even
number of children (i.e., an odd number of keys). Case 2.1 is when \( v \) has an odd number of children and an odd-rank child is sufficient. Case 2.2 is when \( v \) has an odd number of children and every odd-rank child is insufficient. We show how to rebalance the subtree rooted at \( v \) in each case.

Case 1. Node \( v \) has an even number of children: For every pair of children \( c_{2k-1}(v) \) and \( c_{2k}(v) \), \( k \geq 1 \), such that at least one of them is insufficient, we rebalance them concurrently by merging or moving keys as we described above. Now, every children of \( v \) is sufficient. We show that \( v \) has at least \( \lfloor (t-1)/2 \rfloor \) keys. Let \( l(v) \geq t-1 \) denote the number of keys in \( v \) before rebalancing. Since \( v \) had odd number of keys, \( l(v) = 2k+1 \) for some integer \( k \) and the number of children of \( v \) was \( 2k+2 \) before rebalancing. Since \( v \) had \( 2k+2 \) children, at most \( k+1 \) keys have been removed from \( v \) (if every pair of children has been merged) and thus at least \( k \) keys are left in \( v \). We have only to show \( k \geq \lfloor (t-1)/2 \rfloor \). Since \( 2k+1 = l(v) \geq t-1 \), \( k+1/2 \geq (t-1)/2 \) and thus \( k \geq \lfloor (t-1)/2 \rfloor \).

Case 2. Node \( v \) has an odd number of children:

Case 2.1. Node \( c_{2j-1}(v) \) for some \( j \geq 1 \) is sufficient: For every pair of children \( c_{2k-1}(v) \) and \( c_{2k}(v) \) for \( 1 \leq k \leq j-1 \) (resp. \( c_{2k}(v) \) and \( c_{2k+1}(v) \) for \( k \geq j \)), where at least one of them is insufficient, we rebalance them concurrently. It is easy to see that every children of \( v \) is sufficient. We show that \( v \) has at least \( \lfloor (t-1)/2 \rfloor \) keys. In this case, \( l(v) = 2k \) for some integer \( k \) and the number of children of \( v \) was \( 2k+1 \) before rebalancing. Since \( v \) had \( 2k+1 \) children, at most \( k \) keys have been removed from \( v \) and thus \( k \) keys are left in \( v \). Since \( 2k = l(v) \geq t-1 \), \( k \geq \lfloor (t-1)/2 \rfloor \).

Case 2.2. Every node \( c_{2j-1}(v) \), \( j \geq 1 \), is insufficient: In this case, \( O(t) \) number of children of \( v \) are insufficient. All processors except the processors assigned to \( c_{2j-1}(v) \), \( j \geq 1 \), terminate and the processors assigned to \( c_{2j-1}(v) \)’s cooperatively perform rebalancing. They build a B-tree of height 2 with the keys in \( v \) and \( c_k(v) \), \( k \geq 1 \), such that the root has \( \lfloor (t-1)/2 \rfloor \) more keys and the inorder of keys in \( T \) are preserved. Then, they attach the grandchildren of \( v \) to the leaves of the B-tree preserving the inorder of keys in the subtree rooted at \( v \). We show that the number of keys in \( v \) and \( c_k(v) \), \( k \geq 1 \), is large enough to build the B-tree of height 2. The root of the B-tree should have at least \( \lfloor (t-1)/2 \rfloor \) keys and each child of the root at least \( t-1 \) keys. Thus, the minimum number of keys to build the B-tree is \( \lfloor (t-1)/2 \rfloor + (t-1)(\lfloor (t-1)/2 \rfloor + 1) \). We count the number of keys in \( v \) and \( c_k(v) \)’s. Since \( v \) has at least \( t-1 \) keys and every child of \( v \) has at least \( \lfloor (t-1)/2 \rfloor \) keys, the number of keys in \( v \) and \( c_k(v) \)’s is at least \( t-1 + t\lfloor (t-1)/2 \rfloor \). Since \( t-1 + t\lfloor (t-1)/2 \rfloor = t-1 + t(\lfloor (t-1)/2 \rfloor + 1) + (t-1)(\lfloor (t-1)/2 \rfloor + 1) + \lfloor (t-1)/2 \rfloor \), the number of keys in \( v \) and \( c_k(v) \)’s is large enough to build the B-tree. Since there are \( O(t^2) \) keys in the B-tree and we have \( O(t) \) processors, we can perform rebalancing in \( O(t) \) time. The details of this rebalancing are omitted.

In every case, we showed that \( v \) has at least \( \lfloor (t-1)/2 \rfloor \) keys after rebalancing and that the children of \( v \) are sufficient. Since we do not change the number of keys in nodes in level \( j \) for \( j < i \) or \( j > i+1 \), \( T \) is a \( B'(i+1) \)-tree after rebalancing.

Let \( RB(i) \) denote the concurrent applications of rebalancing steps to all subtrees rooted at the parents of insufficient nodes in a \( B'(i) \)-tree. Facts 1 and 2 follow immediately from the description of rebalancing steps.

**Fact 1** \( RB(i) \), \( i \geq 0 \), accesses only the nodes in level \( i \) and \( i+1 \), and only the edges between nodes in level \( i-1 \), \( i \), \( i+1 \), and \( i+2 \).

**Fact 2** \( RB(i) \), \( i \geq 0 \), may move a record in level \( i \) to level \( i+1 \) and a record in level \( i+1 \) to level \( i \), but does not move records in level \( j \) for \( j \leq i-1 \) or \( j \geq i+2 \) to other levels.

**Lemma 2** follows immediately from Fact 2.

**Lemma 2** A record is in level \( i \) at least 2 in a \( B'(0) \)-tree if and only if the record is in level at least 2 in the \( B'(1) \)-tree which is obtained by applying \( RB(0) \) to the \( B'(0) \)-tree.

**stage 7.** Find the successor of every marked internal record: All marked records in level at least 2 after stage 6 were in level at least 2 after stage 5 by Lemma 2. Thus, \( succ(x) \) which \( p_x \) knows is the successor of \( x \) if marked record \( x \) is in level at least 2. For each marked record \( x \) in level 1, \( p_x \) finds \( succ(x) \) in \( B'(1) \)-tree \( T \). Since \( x \) is in level 1, \( p_x \) can find \( succ(x) \) in \( O(1) \) time. Now, \( p_x \) knows \( succ(x) \) for each marked internal record \( x \) in \( T \). Since \( T \) was a \( B'(1) \)-tree after stage 6 and we do no write operations on \( T \) in this stage, \( T \) remains as a \( B'(1) \)-tree.

**stage 8.** Restore \( B'(1) \)-tree \( T \) to \( B'(i) \)-tree: We apply \( RB(1) \), \( RB(2) \), \ldots, \( RB(h) \) until \( T \) becomes a \( B'(1) \)-tree where \( h \) is the height of \( T \). Since \( h \) is \( O(log_t n) \) and \( RB(i) \), \( i \geq 1 \), is performed in \( O(t) \) time, this stage takes \( O(t log_t n) \) time. Since the inorder of \( T \) is preserved in rebalancing, the successor of a marked internal record \( x \) remains unchanged. Hence, \( succ(x) \) which \( p_x \) knows is the successor of \( x \) after stage 8, which is necessary for stage 3 of the next iteration to start.
We consider the time complexity of this algorithm. Stage 1 takes \(O(\log k + t \log n)\) time and stage 2 takes \(O(\log n)\) time. Stages 3, 5, and 6 take \(O(t)\) time. Stages 4 and 7 take \(O(1)\) time and stage 8 takes \(O(t \log_k n)\) time. Since the number of marked records in leaves is at least the number of marked internal records and a half or more of marked records in leaves are removed in an iteration of stages 3-8, a quarter of marked records are removed in an iteration of stages 3-8, which implies stages 3-8 are repeated \(O(\log k)\) times. Overall, \(O(t \log k \log n)\) time is required.

We now pipeline stages 3-8 of the algorithm to get the time complexity of \(O(t \log k + \log n)\). We merge stages 3-7 and refer to them as \(PS\). The pipeline stages are \(PS\) and each \(RB(i), i \geq 1\), which are performed in \(O(t)\) time. Pipeline stage \(PS\) of the \((j+1)\)st, \(j \geq 1\), iteration is ready to start if \(PS\) of the \(j\)th iteration is finished because \(p_x\) knows \(\text{succ}(x)\) for each marked internal record \(x\) after stage 7 of the \(j\)th iteration, which is necessary for stage 3 of the \((j+1)\)st iteration to start. However, the pipeline stages cannot be performed back-to-back because \(RB(i)\) and \(RB(i+1)\) cannot be performed concurrently by Fact 1. We start \(PS\) of the \((k+1)\)st iteration at the beginning of \(RB(5)\) of the \(k\)th iteration. We first show that \(PS\) can be performed concurrently with \(RB(i), i \geq 5\).

\(RB(i), i \geq 5\), accesses nodes in level at least 5 and edges between the nodes whose levels are at least 4. Most operations in \(PS\) (finding the successors of successors in stage 4, removing keys in stage 5, \(RB(0)\) in stage 6, and finding successors in stage 7) do not access nodes in level at least 5 and edges between the nodes whose levels are at least 4. Only when we exchange keys in stage 3, we may access nodes in level at least 5. However, in this case we access the keys in the nodes while \(RB(i)\) accesses the pointers of the records in the nodes. Hence, \(PS\) and \(RB(i), i \geq 5\), can be performed concurrently without read/write conflicts. It follows from Fact 1 that \(RB(i)\) can be performed concurrently with \(RB(i+j), j \geq 4\). Hence, if we start \(PS\) of the \((k+1)\)st iteration at the beginning of \(RB(5)\) of the \(k\)th iteration, read/write conflicts are avoided.

Since the number of pipeline stages is \(O(\log n)\), \(PS\) is performed \(O(\log k)\) times, and each pipeline stages requires \(O(t)\) times, the time complexity of the pipelined algorithm is \(O(t \log k + \log n)\). Therefore, we get the following theorem.

**Theorem 1** Parallel deletion of \(k\) unsorted keys from a B-tree of minimum degree \(t \geq 2\) with \(n\) nodes can be done in \(O(t \log k + \log n)\) time with \(k\) processors on the EREW PRAM.

**References**


