Concurrent Order Maintenance

Seth Gilbert*

December 17, 2003

Abstract

In this paper, we present a parallel data structure to dynamically maintain an ordered list of items. Our data structure supports two operations: Query and Insert. A Query operation indicates which of two items comes first in the ordering; an Insert operation adds a new item to the ordering. We present two versions of the algorithm. For each version, we determine both the amortized cost of an operation – useful for calculating the work of a parallel program – and the worst-case cost of an operation – necessary for understanding the critical path. The differing versions of the algorithm allow a trade-off between the cost of queries and the cost of insertions, and between the amortized work and the worst-case critical path.

1 Introduction

In this paper, we study the problem of maintaining an ordered list of items subject to concurrent accesses. We present a data structure that supports the following operations[1]:

- Insert(X, Y): Insert a new item, Y, immediately after item X.
- Query(X, Y): Determine whether X precedes Y.

In the sequential, non-concurrent, case, this problem has been well studied. Dietz [2] developed the first solution to this problem, and Dietz and Sleator [3] later published the fastest known algorithm, guaranteeing $O(1)$ amortized inserts and queries. Bender et al. [1] developed a simplified version that is the basis for the algorithm presented in this paper.

None of these prior algorithms, however, consider the issue of concurrency. Most of these solutions depend on amortization to reduce the cost of individual operations. Amortized data structures, though, are inherently difficult to use in a concurrent setting, since all concurrent operations may be delayed by an operation involved in worst case behavior. The non-amortized solutions (for example, [5], and also versions presented in [3] and [1]), on the other hand, are so complicated that it is hard to imagine an efficient concurrent version of these algorithms.

We present, then, the first concurrent solutions to the order-maintenance problem. The algorithms are described in a locking model where asynchronous processors take continuous steps. Since locks are held, progress depends on processors continuing to take steps, and for the sake of analysis (i.e., not correctness) it is assumed that the processors move in synchronized steps.

In this model, we provide both amortized and worst-case time bounds for each algorithm presented. The amortized bounds determine the total work required to perform a series of operations. The worst-case time bounds, on the other hand, can be used to determine the critical path of a parallel algorithm that uses the order-maintenance data structure.

Summary of Results Figure 1 summarizes the algorithms presented in this paper. Consider a parallel program with at most I Insert operations and Q Query operations. Further, assume that any path through the computation (i.e., any path through the DAG representing the parallel computation) has at most C_i Insert operations and C_q Query operations. We use C_i and C_q to bound the contribution of Insert and Query operations to the cost of the critical path.

*Joint work with Michael Bender and Jeremy Fineman.

[1] In many versions of the data structure, a delete operation is also supported. This does not add significant difficulty, but we omit it for the sake of simplicity.

	Amortized Query	**Amortized** Insert	**Max Critical Path Component**
Serial [3]	$O(1)$	$O(1)$	N/A
Main Alg.	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{p}{K} + \frac{K^\epsilon}{\epsilon}\right)$	$O\left(\frac{C_q}{\epsilon} + \frac{I}{K} + \frac{C_i K^\epsilon}{\epsilon}\right)$
Exp. Search	$O(\log\log K)$	$O\left(\frac{p}{K} + \sqrt{K} + \log\log K\right)$	$O\left(C_q \log\log K + \frac{I}{K} + C_i\sqrt{K}\log\log K\right)$
Conjectured	$O\left(\log\frac{1}{\epsilon}\right)$	$O\left(\frac{p}{K} + \sqrt{K} + \log\frac{1}{\epsilon} + K^\epsilon\right)$	$O\left(C_q \log\frac{1}{\epsilon} + \frac{I}{K} + \sqrt{K}C_i \log\frac{1}{\epsilon} + C_i K^\epsilon\right)$

Figure 1: Summary of results, assuming Q Query operations, I Insert operations, C_i critical path inserts, and C_q critical path queries. The constants $K,\ 2 \leq K < I$ and $\epsilon,\ 0 < \epsilon \leq 1/2$ trade-off between the cost of Query operations and the cost of Insert operations, and between the amortized and worst-case costs.

Figure 1 includes results for four algorithms. The first algorithm, *Serial*, refers to the serial solution, as described in [3] or [1]. This provides a benchmark to compare our new algorithms against. The next algorithm, *Main*, is the primary algorithm presented in this paper. The third, *Exponential Search*, provides an alternate variation, and the fourth, *Conjectured*, is yet another variation which we present but do not discuss in any detail.

Main Algorithm This algorithm is the primary result of this paper. The algorithm is parameterized by two constants, $K,\ 2 \leq K < I$, and $\epsilon,\ 0 < \epsilon \leq 1/2$. The constant K allows the data structure to trade-off worst-case time versus amortized time. When $K = 1$, good amortized results are achieved, however the critical path could be quite long, including all I Insert operations. (This solution, where $K = 1$, can be achieved by trivially adaptating Dietz and Sleator to accept concurrent operations.) If, instead, K is larger, then the worst-case cost decreases, thereby reducing the critical path. This, though, increases the amortized cost of each operation, resulting in more total work. The constant ϵ trades off between the cost of queries and the cost of inserts. For example, if $\epsilon = 1/4$ and $C_i = I/p$ (i.e., the inserts are evenly spread among p possible processors), then choosing $K = O(p^{4/5})$ minimizes the critical path. This leads to a critical path of $O(4C_q + 4I/p^{4/5})$, which implies $O(4)$ cost Query operations, and approximately $4p^{1/5}$ cost Insert operations for both amortized and critical path analysis (when the total cost is divided by C_i, the number of insertions on the critical path).

Exponential Search Our second algorithm is a variant of the main algorithm that relies on an exponential search tree. This introduces a $\log\log K$ to the various results; the parameter ϵ is no longer used, as in choosing to use an exponential search tree, the trade-off between queries and insertions has been fixed. The resulting time bounds are generally worse than the main algorithm; however the algorithm demonstrates a different variant of the approach presented in this paper. Further, it leads naturally to the third, conjectured, algorithm which may be of more interest.

Conjectured Algorithm We briefly discuss a third algorithm, that combines some of the properties of the prior two algorithms. In particular, this algorithm reduces the cost of queries to $\log 1/\epsilon$, from $1/\epsilon$ in the main algorithm. However this is accomplished at the cost of slower Insert operations.

Overview We first briefly review the Dietz and Sleator algorithm in Section 2. We state the basic modeling assumptions in Section 3. In Section 4 we present the overall structure of our algorithm and claim that it correctly solves the order-maintenance problem. We present the details of the main algorithm in Section 5, and show that it guarantees the performance claims. In Section 6, we discuss the Exponential Search Tree algorithm, and in Section 7 we discuss some associated ideas, including the modified algorithm and use in the non-determinator.

2 Serial Order Maintenance

In this section, we describe at a high level the serial order-maintenance algorithm of Dietz and Sleator [3], as simplified by Bender et al. [1]. Later, this algorithm is used as a black-box component of our new algorithm. As a result, many of the details are omitted. The tool of indirection, though, is presented as part of this algorithm; this tool plays a key role in our later constructions.

We first present the basic algorithm, which guarantees $O(1)$ cost Query operations and $O(\log n)$ cost Insert operations, where n is the maximum number of items in the data structure. We then explain how Dietz and Sleator use indirection to achieve constant time Insert operations.

The Basic Algorithm Each item in the data structure is associated with a unique tag; the ordering of the tags is consistent with the total order maintained by the data structure. As a result, Query operations are quite simple: it suffices to compare the tags of the two items, and indicate which is larger. We use the notation $tag(X)$ to refer to the tag associated with item X.

The primary difficulty, then, is to assign tags to the items, and to maintain the ordering of the tags as new items are inserted. If an Insert operation attempt to insert an item, Y, after some item, X, and there is no item that already has $tag(X) + 1$, then we simply assign Y a new tag that is larger than $tag(X)$ and smaller than X's successor. It turns out that in the worst case it does not matter which of these tags is assigned to Y; we can simply assign Y the tag $tag(X) + 1$.

On the other hand, if some item already has been assigned $tag(X) + 1$, then the tags are reorganized to make room for item Y. The key question (and heart of the algorithm) is to determine how many items to reorganize. We do not go into the details, instead simply claiming that when the range of reorganization is chosen appropriately, an Insert operation only has to relabel $\log n$ items, in an amortized analysis.

Indirection Using the basic algorithm, inserting n items into the data structure has a cost of $O(n \log n)$. The goal, then, is to reduce this cost to $O(n)$.

Instead of storing $O(n)$ items in the data structure, we will instead only store $O(n/\log n)$ items in the data structure. Each of these items is itself a list of $\log n$ items. That is, each item in the main data structure is itself a pointer to a second level data structure that maintains a list of at most $\log n$ items.

In the main data structure, there are now only $O(n/\log n)$ insertions, each of which has a cost of $\log n$. Therefore the total cost of insertions into the main data structure is $O(n)$.

The other cost to consider is the insertion into the second level data structure. The second level data structure only maintains $\log n$ items. It turns out that it is straightforward to maintain $\log n$ items. Assume we are inserting item Y after item X. Consider the task of assigning a tag to each inserted item. In this case, choose the tag that is exactly half way between that of X and X's successor. Each insertion at worst divides the smallest available range of tags in half. Before any inserts, the tags range from 0 to n^2; that is, the smallest range of tags has size n. This can be divided in half $\log n$ times before running out of tags. Therefore, before any reorganization of the tags is necessary, the list is storing the maximum allowable number of items.

Therefore the algorithm precedes as follows. In order to Insert item Y after item X, first examine X's second level list. If the list is not yet full, then simply add Y to X's second level list. Otherwise, if the list is full, then we must split X's second level list and add a new second level list to the main data structure. First, create a new second level list. Next, copy half the items (with the largest tags) from the first second level list to the new second level list. Finally, insert the new second level list in the main data structure after the first second level list.

[2] In fact, some polynomial of n is used.

3 System Model

In this section we describe the concurrency model that we use for the rest of the paper, and then discuss other assumptions we are making.

Concurrency First, we assume that the amount of concurrency is bounded; that is, there is some upper bound on the number of operations that will concurrently access the data structure. We refer to this bound as p, the maximum number of processors executing the parallel program that is accessing the data structure. (In fact, it suffices that the concurrency is bounded, not the number of processors that might be executing the program.)

Adaptability Throughout this paper, we assume that a bound on the number of Query and Insert operations is known in advance, and that the data structure is designed with this in mind. The Dietz and Sleator order-maintenance algorithms, however, can adapt to increasing (and decreasing) numbers of elements. When the data structure contains too many elements, the size is doubled and all the data is copied to the new data structure; this can introduce at most $O(1)$ amortized cost. We omit this complication, though a hint is given in Section 7 as to why this is not a problem in the concurrent case.)

Bounded Insertions We also assume that at any given time, for every item X, only a single Insert operation attempts to insert a new item after X. This restriction limits the concurrency in any one area of the data structure. This restriction only effect the performance of the algorithms discussed, however. Correctness is always maintained.

Locks Next, we assume that processors can lock and unlock certain pieces of the data structure. There are two different locking mechanisms used. First, we assume a subcomponent, the **DietzSleatorList** that contains a lock and unlock primitive. Second, we assume that the algorithm has access to a *compare-and-swap* (CAS) operation. The CAS function takes three parameters:

1. var
2. $old\text{-}value$
3. $new\text{-}value$

The function atomically checks whether the current value of the var is equal to $old\text{-}value$, and if so, sets var to $new\text{-}value$. In the successful case it returns true; otherwise, it returns false

We assume that both locking mechanisms are fair. That is, the locks are scheduled in a queuing fashion; if one operation attempts to acquire the lock prior to another operation even beginning, then it will get the lock first.

DietzSleatorList We assume that our algorithm has access to a **DietzSleatorList** component that implements the original Dietz and Sleator algorithm (or the Bender et al. simplification). Everything assumed here is readily generated using the existing serial order-maintenance algorithms.

We assume that this component supports Query and Insert operations in amortized $O(1)$ time, and worst-case $O(u)$ time, where u is the size with which the data structure is initialized. We assume that the data structure can handle concurrent query operations, but no more than one Insert operation at a time. (We argue that it is not difficult to support concurrent Query operations while an Insert is ongoing, as long as the reorganization is performed correctly.) We therefore assume the following basic function:

- NewDS(n): produces a new Dietz and Sleator data structure designed to hold at most n items.

We further assume that this list supports a set of functions that operate on items, adding them to the list and performing queries. Each item inserted into the **DietzSleatorList** contains two fields: (1) parent: contains a pointer to the current list containing the item, and (2) list: a pointer to another (arbitrary) **DietzSleatorList**. Notice, then, that the element X is always stored in $X.parent.list$. The list data structure supports the following operations:

- Insert(X, Y): Inserts Y after X, and sets $Y.parent$ to $X.parent$, a pointer to the list itself. This is used to add extra children to the parent of X. The function returns a pointer to the inserted Y.

- Query(X, Y): Returns true if $X < Y$. This is used to determine the order of X and Y within the parent of X.
- full?: Returns true if the list data structure is full, i.e. the node can contain no more children.
- size: Returns the maximum number of elements that the list is designed to hold.
- split$(\textit{new-list}, \textit{new-parent})$: Removes the largest half of the items from the list, and inserts them into *new-list*. This is used to split some node, X, in the following way: the application creates a new node, Y, and the split function is called with $Y.list$ and Y as the parameters. The children of X are then shared between X and Y (with the order maintained correctly). All the items in the list have their *parent* pointer updated: items that have moved to Y have *parent* set to Y. This operation is assumed to take $O(u)$ time, where u is the size with which the data structure is initialized.
- lock: locks the data structure.
- unlock: unlocks the data structure

4 The Concurrent Algorithm Framework

In this section, we present the overall framework of our algorithm. The framework supports multiple variations, and we later present various algorithms based on this structure. We first describe the algorithm (Section 4.1), and then argue that it correctly solves the order-maintenance problem (Section 4.2).

4.1 Algorithm Description

At a high level, our algorithm consists of a B-tree with a fixed depth (i.e., the root node never splits). Each node in the B-tree maintains an ordered list of children, using a **DietzSleatorList**. Each leaf of the tree represents a single item inserted into our data structure.

Overall, the tree maintains the invariant that a left-to-right traversal of the tree visits the leaves in the appropriate order specified by the pattern of (concurrent) insertions. As new leaves are inserted, nodes in the tree split to ensure that a node only holds an appropriate number of children. When a node splits, as in a B-tree, it may cause a series of splits higher up in the tree.

As long as the tree is relatively shallow, operations should be relatively efficient. A deep tree, on the other hand, would lead to expensive operations, as an operation might be required to traverse the entire depth of the the tree.

The variations in the algorithm are based on differing the fan-out of the tree at different levels. All the algorithms presented use a large root node. The algorithms different at the lower levels of the tree.

A schematic of the data structure is presented in Figure 5. Some nodes are omitted (as in the real data structure, all leaves have the same depth). Notice that each node contains a **DietzSleatorList** that maintains its children in order; each child has a pointer to its parent. We now explain the algorithm in more detail.

NewNode(Item X, $\textit{level}$)
1 $X.parent.list \leftarrow$ NewDSList$(f(level))$ $\triangleright$ The function f varies depending on the algorithm.
2 $X.parent.list$.Insert(**null**, X)
3 **if** *level* > 0
4 **then** $X.parent \leftarrow$ NewNode$(X.parent,\ level - 1)$
5 **return** X

Figure 2: List Creation Routine

```
Query(Item X, Item Y)
 1  if (X.parent = Y.parent)
 2     then return X.parent.list.Query(X, Y)
 3  if X.split-lock = init-from-Z                 ▷ If a split is in progress, find the real parent.
 4     then if X.parent = Z
 5             then X ← Z
 6                  split ← X
 7  if Y.split-lock = init-from-Z
 8     then if Y.parent = Z
 9             then Y ← Z
10                  split ← Y
11  if (X.parent ≠ Y.parent)                      ▷ Now check the real parents.
12     then return Query(X.parent, Y.parent)
13     else  if split = Y
14              then return true
15     else  if split = X
16              then return false
17     else  return X.parent.list.Query(X, Y)
18
```

Figure 3: Query Routine

Creating a New Data Structure We first discuss the creation of a new data structure. The pseudocode is presented in Figure 2. The data structure is created by inserting the first element (which may well be a dummy element).

The creation routines depends on two parameters that are specified in the algorithm variations: a function, f, and the maximum depth of the tree, $level$. The function f determines the size of the **DietzSleatorList** at a given level of the tree. It must always return a number no smaller than two; it makes no sense to have an ordered list of one item.

During the creation, a single item is inserted, creating a single root-to-leaf path in the tree. The data structure is created recursively, first initializing the leaf node, and then recursively (in Line 4) creating a new parent node with a smaller depth. When the depth reaches zero, the root node is created.

Query Operations We next discuss the Query operation. A simple Query is depicted in Figure 6. The pseudocode is presented in Figure 3. The Query operation is passed two parameters, X and Y. If both X and Y have the same parent, then their order is determined by the **DietzSleatorList** maintained by the parent (line 2).

First, however, it is necessary to find the real parents of X and Y. If a split is in progress, a node may point to a sibling, rather than a parent. When a node is split, a new node is created to take on some of the original node's children. Eventually, this node is inserted in to the tree. Until then, it simply maintains a pointer to the sibling whose children it is acquiring. Lines 3–10 checks if this is the case, and updates X and Y accordingly, remembering in the variable *split* which of them was splitting. (If they were both splitting, they cannot have the same parent.)

Finally, the operation checks if the parents are equal. If the parents are equal and one of them was a new node involved in a split operation, that node must be on the right. In this case, we have found the answer to the query. Otherwise, the operation simply queries the **DietzSleatorList**. If the parents are not equal, the Query operation is called recursively on their parents.

Insert Operations The most complicated part of the algorithm is the Insert operation. The pseudocode is presented in Figure 4. Figure 7 depicts an insert during an ongoing split operation.

The operation begins by acquiring a lock on the parent of X (line 1). This lock will be held until the insertion is complete.

Next, we check whether the parent of X is full. In the simple (and common) case when the parent is not full Y is inserted after X in the parent node of X: in Lines 27–28, a lock is taken on the parent and Y is inserted into the list of chilren.

```
Insert(Item X, Item Y, bool leaf?)
 1  X.parent.list.lock
 2  if (X.parent.list.full?)                   ▷ Note: If X.parent is the root, it is never full.
 3     then
 4          oldParent ← X.parent               ▷ Begin splitting X.parent.list.
 5          size ← X.parent.list.size
 6          Item Node.list ← NewDSList(size)   ▷ Create new node to split into.
 7          Node.parent ← X.parent
 8          while CAS(X.parent.split-lock, 0, split-to-Y) = false
 9              do Spin                        ▷ Prevent X.parent from splitting again
10          while CAS(Node.split-lock, 0, init-from-X) = false
11              do Spin                        ▷ Prevent Node from splitting again.
12          Node.list.lock
13          X.parent.list.split(Node.list, Node)  ▷ Do the split.
14          X.parent.list.Insert(X, Y)         ▷ And insert the original item after X.
15          if leaf? = false
16            then X.split-lock ← 0            ▷ Release lower level split-lock
17                 Y.split-lock ← 0
18          Node.list.unlock
19          oldParent.list.unlock              ▷ Unlock original X.parent.list
20          Insert(X.parent, Node, false)      ▷ Insert new node after X.parent in X.parent.parent.list.
21          if oldParent.split-lock = split-to-Node
22            then oldParent.split-lock ← 0
23          if Node.split-lock = split-from-oldParent
24            then Node.split-lock ← 0
25     else
26          ▷ Otherwise just do the insert.
27          X.parent.list.Insert(X, Y)
28          X.parent.list.unlock
29  return Y
```

Figure 4: Insertion Routine

If the parent of X is already full, however, the parent node must be split. First, a new node is created and initialized with an appropriate sized **DietzSleator** list (line 6).

Before any children can be added to this node, its parent must be set (line 7), and the *split-lock* must be taken on both $X.parent$ and the new node.

Finally, starting on line 12, we can actually do the split. Notice that acquiring this lock on $Node$ will always succeed, since no other process yet has a pointer to $Node$.

Once the split is accomplished, Y can actually be inserted into the list (line 14). At this point, we have actually accomplished our initial goal. We can now release many of the locks, to allow other operations to proceed. In particular, any *split-lock*s that are held by X and Y are released; by holding the *split-lock*'s in $X.parent$ and $Y.parent$, we no longer need to hold the *split-lock*s in X and Y. At this point, we can also unlock both the new node, and the originaly parent of X. (Notice that X's parent may have changed during the split, so we need to maintain the original parent.)

Finally, we do the recursive insertion that inserts the new node into the parent. After this is done, we make sure the *split-lock* has been released. (Depending on the parents of X and Y after the split, the recursive $actInsert$ may or may not have done this.

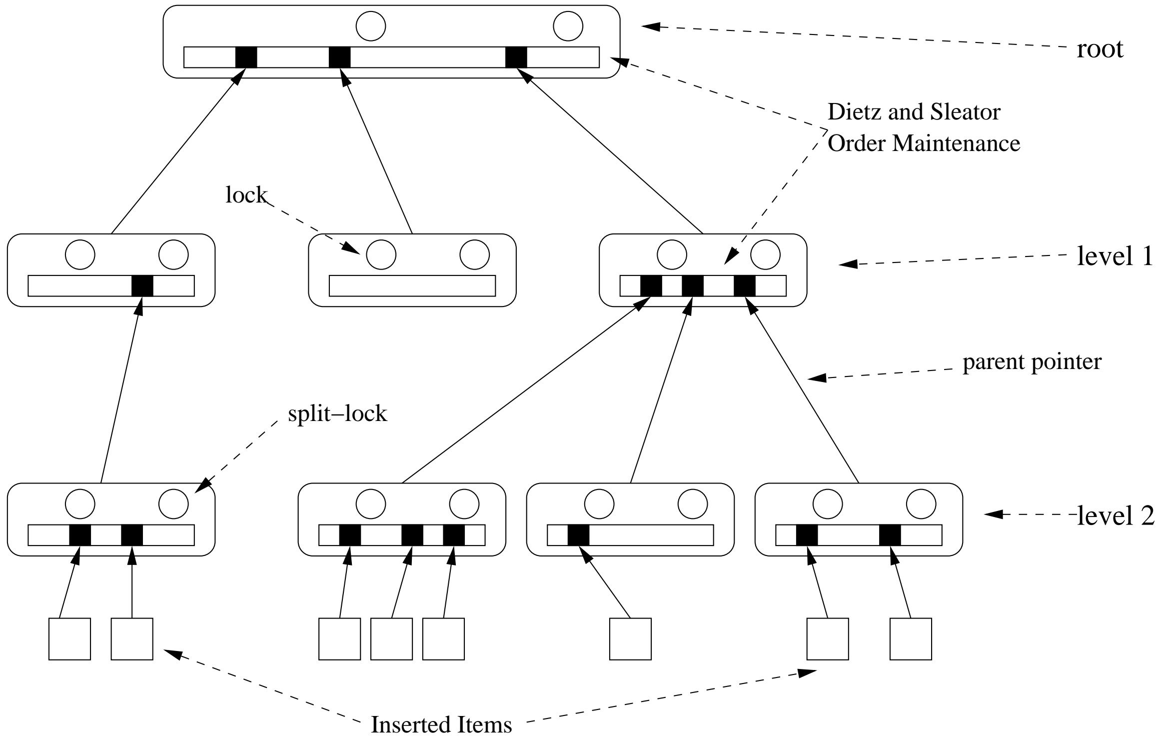

Figure 5: Depiction of a B-tree used for order maintenance. The large ovals are nodes in the tree. The rectangles within the nodes are DietzSleatorLists that maintain the ordering of the children. The circles are various locks. The rectangles at the bottem represent leaf elements inserted into the data structure.

4.2 Correctness

In this section, we first show that the algorithm is deadlock free, and then claim that it correctly solves the order-maintenance problem.

Theorem 4.1 *The algorithm is deadlock free.*

Proof. We only need consider the Insert operation, as no other operations take locks. (All line references are therefore in Figure 4.) There are two places where locks are taken: line 1 and lines 8 and 10. (Notice that the lock on $Node.list$ can never cause a deadlock, since when the lock is taken, $Node.list$ is still empty, and therefore has no children. As a result, there can be no ongoing Insert operations, and therefore no other process can have locked this list.)

Notice that during the execution of an Insert operation, there is a recursive call to Insert on line 20, which may recursively lock certain lists. However all locks are acquired in a well-defined order, starting at the leaves and moving toward the root of the tree. Therefore, there can be no deadlock: whichever process first takes the lock for the least common ancestor of the contending sets of locks will complete. (Some locks are released in a hand-over-hand manner as the tree is ascended; this helps efficiency, but is not required to avoid deadlock.) □

Theorem 4.2 *The algorithm correctly implements an order-maintenance data structure.*

Proof (sketch). The Insert operations maintain the invariant that a left-to-right, depth-first traversal of the tree reflects the desired order of the elements. A regular insert at a leaf clearly maintains the order, by the correctness of the data structure maintaining the order of children. A split too maintains this order, by creating a new node and inserting it directly after the old node whose children it takes. (Formally, this holds by induction on the depth of the recursion.)

If no split is in progress, it is clear that the Query operation correctly returns this order. If a split is in progress, then the Query operations finds the real parent, as if the split were not going on. □

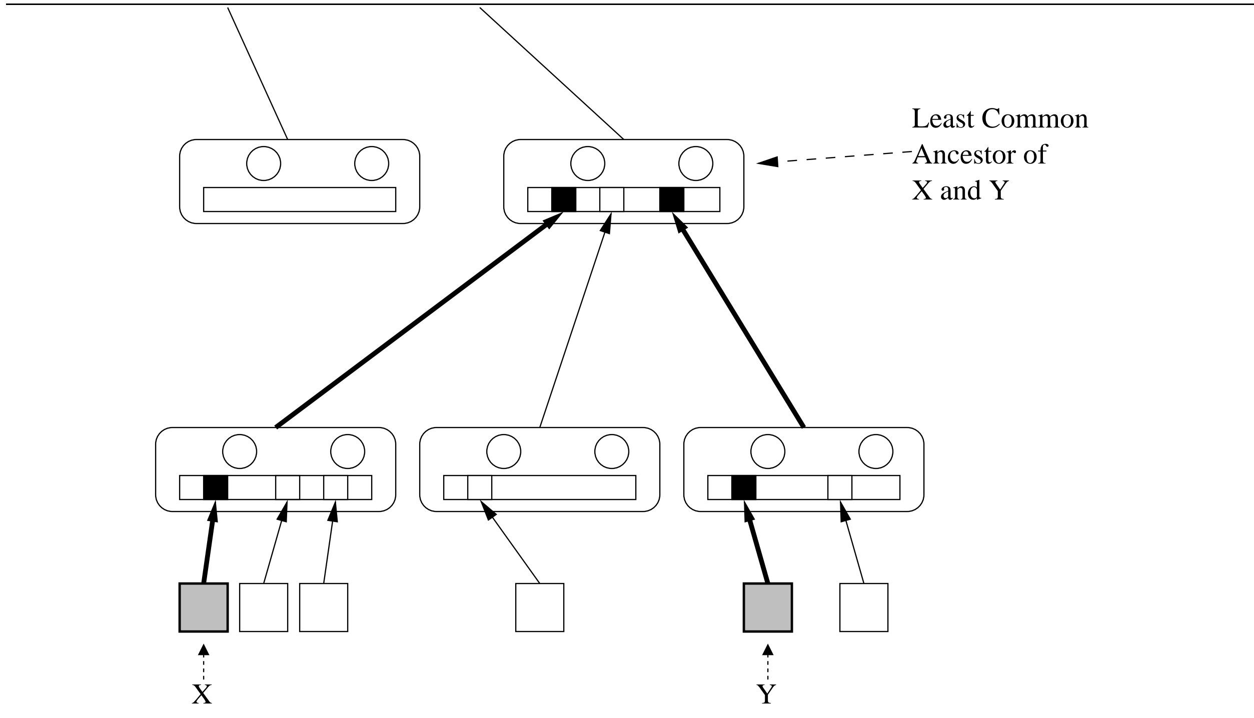

Figure 6: Depiction of a simple Query operation. Items X and Y are being compared. The correct answer will be returned by their least common ancestor.

5 Main Algorithm

In this section we describe the first algorithm. We first present the algorithm, and then claim that it guarantees the desired time bounds.

Let ϵ and K be the two constants that parameterize the algorithm. We assume that $K^\epsilon \geq 2$; this will be the fan-out of the tree, which must be at least 2. Every node in the tree contains a Dietz and Sleator list. That is, every time the NewList function is called in Figure 2, instead the NewDietzSleator function is used. The root of the B-tree is specified to have I/K elements. That is:

$$f(0) = \frac{I}{K} .$$

All other nodes in the tree are specified to have size K^ϵ. That is:

$$\forall i > 0, \ f(i) = K^\epsilon .$$

Lemma 5.1 Query *operations take worst-case time $O(1/\epsilon)$.*

Proof. A simple calculation shows that the tree has exactly depth $1/\epsilon + 1$: each subtree of the root contains at most K items, and the fan-out of the tree is K^ϵ. Since there is no locking, and there can be at most one Dietz and Sleator Query per level of the tree, this immediately implies that Query operations take time $O(1/\epsilon)$, as desired. □

Lemma 5.2 *Assume there are I* Insert *operations, with at most concurrency p. Then each operations takes at most*

$$O\left(\frac{p}{K} + \frac{K^\epsilon}{\epsilon}\right)$$

amortized time.

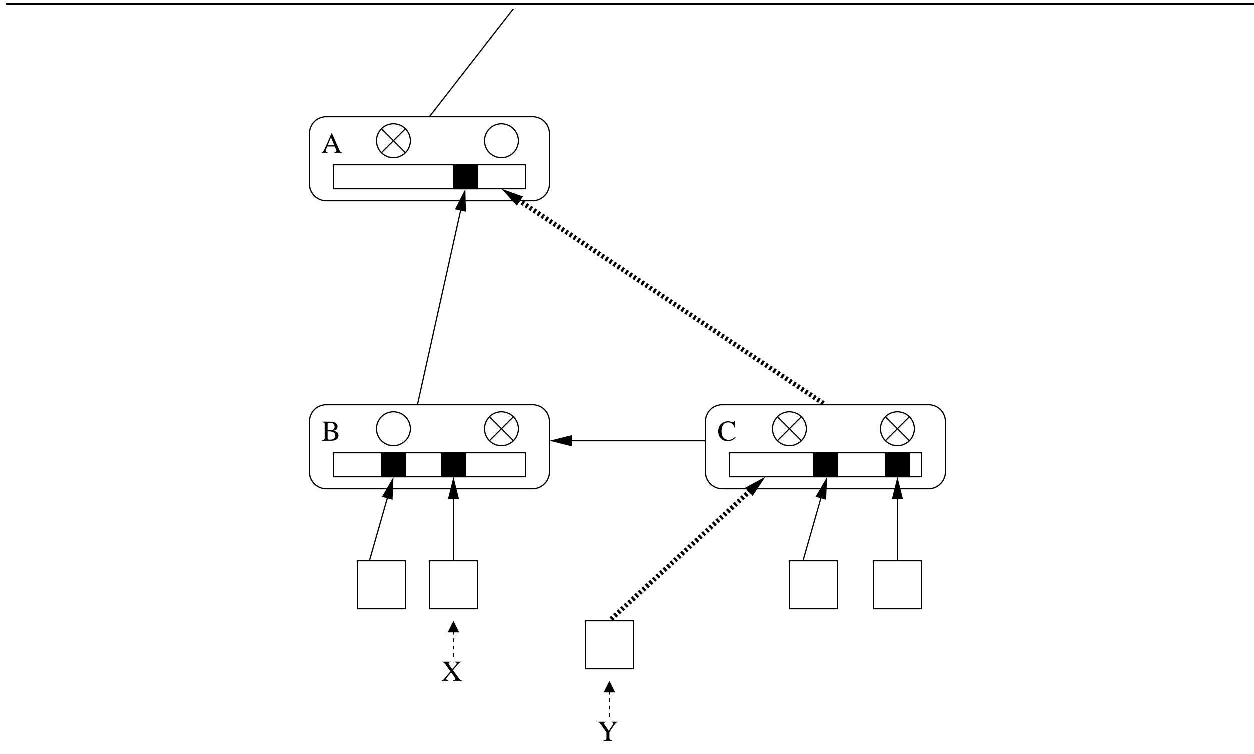

Figure 7: Depiction of an Insert operation in progress and a concurrent split operation. Node B is splitting. New node C has been generated, and will be inserted into A. Note the "X"'s in the circles, marking the locks as taken. The fuzzy-line arrow indicate an insertion in progress. Note how node C is currently pointing to node B while the split is in progress. Node Y was the leaf inserted that caused this split.

Proof. We first determine the total work necessary to accomplish the I Insert operations. Notice that each Insert operation might cause a low-level insertion into a Dietz and Sleator order-maintenance structure at each level of the tree; that is, each Insert operation might cause $1/\epsilon$ Dietz and Sleator insertions.

Each of these $1/\epsilon$ insertions can cause work to occur in one of the following ways:

1. Waiting for $X.parent.list$ lock, when no one has acquired $X.parent.split\text{-}lock$.
2. Splitting the node:
   (a) Acquiring $X.parent.split\text{-}lock$. This causes work by forcing all other processes to Spin until the lock is released. We do not count the work caused by a *split-lock* higher in the tree being held. If a higher level *split-lock* (for example, $X.parent.parent.split\text{-}lock$) is blocking progress of a node, we charge the work against the insertion holding that lock, not this one.
   (b) Copying the data to the new node (i.e., doing the split operation).
3. Inserting Y after X in the Dietz and Sleator data structure.

There are two things to notice. First, we do not include here the work required to do the recursive insert; that work is accounted for in the appropriate other insertion. Second, we do not count as work the time waiting to acquire the *split-lock*; instead we will count as work the number of other processes forced to wait as a result of this insertion holding the *split-lock*. It turns out that this is easier to bound: if no insertion at a higher level in the tree is blocking the current operation, we can determine how long it will take to complete and therefore how much work to charge against this insertion.

We consider insertions into the root node seperately (i.e., $level = 0$). The root node never splits; therefore it only has costs (1) and (3). Throughout an execution, there are at most I/K insertions into the root node. While the root

node is locked, other Insert operations may have to wait. In particular, for each processor, the work induced by waiting for the lock (i.e., cost (1)) is at worst the time it takes for *all* other insertion operations into the root to complete. There are at most I/K insertions into the root node, and the Dietz and Sleator order-maintenance structure guarantees that all these other operations take at most time $O(I/K)$ when occurring sequentially. Therefore, all p processors will have to spend at most pI/K time waiting for locks. As we have already pointed out, the cost of actually performing the insertions is $O(I/K)$, leading to an overall amortized cost of $O(p/K)$ time per operation.

We now consider the children of the root, i.e., $level = 1$. At any given time, there can be at most K^ϵ insertions into a level one node. By the properties of the Dietz and Selator list, the cost of inserting these elements will be $O(K^\epsilon)$. In particular, each operation only has to wait for all concurrent operations to complete before obtaining the lock, and (if no process has acquired $X.parent.split\text{-}lock$) this will take at most time $O(K^\epsilon)$. Therefore each operation takes amortized time $O(K^\epsilon)$ (for cost (1)).

We now calculate the cost split operations at a level one node. First, notice that there are at most $O(I/K)$ split operations. Any given split operation only effects the $O(K)$ nodes in the two sub-trees effected: the sub-tree being split and the new sub-tree created. We need to bound how long a process may hold the *split-lock*s in order to determine how much waiting may be induced. Notice that the main component of the split operation is the recursive insertion. Over the entire execution, however, all the recursive insertions to split one node can take at most $O(I/K)$ time, since that is the bound on all insertions into the root node. Therefore, cost (2.a) is equal to $O(I/K) * O(K) = O(I)$, leading to amortized constant work per operation. Similarly, the cost of the copying the data (cost 2.b), $O(K^\epsilon)$, is amortized over the K^ϵ insertions that must precede a split, leading again to constant time amortized work. Overall, then, each insertion requires amortized $O(K^\epsilon)$ overall cost.

Finally, we consider the cost of splitting a node at an arbitrary level that is greater than one. Consider some level ℓ. For all the costs except (2.a), the costs are as in the previous case, i.e., $O(K^\epsilon)$. We consider the costs of acquiring the *split-lock*. At some node at level level ℓ, there are at most

$$\frac{I}{\frac{K}{K^{\epsilon\ell}}} = \frac{K^{\epsilon\ell} I}{K}$$

splits that occur throughout the entire I insertions. Each of these causes a recursive insertion, and in the worst-case each of these takes at most $O(K^\epsilon)$ time to complete. At any given time, there are at most

$$\frac{K}{K^{\epsilon\ell}}$$

descendents that are effected by the acquisition of the lock. Therefore, the total cost is $O(IK^\epsilon)$, leading to an amortized cost yet again of $O(K^\epsilon)$

Overall, then, each of the I/ϵ insertions not into the root has amortized cost $O(K^\epsilon)$, and the insertion into the root has amortized cost $O(p/K)$. Therefore, in total, each of the I Insert operations (which consists of $1/\epsilon$ insertions) has amortized cost $O(p/K + K^\epsilon)$, as desired. □

Lemma 5.3 *Assume some C_i Insert operations occur non-concurrently in an execution with I Insert operations. Then, in the worst-case, these operations take at most:*

$$O\left(\frac{I}{K} + \frac{C_i K^\epsilon}{\epsilon}\right) .$$

Proof (sketch). This proof follows essentially the same lines as the last proof, so we only sketch the details. The key issue to note is that at every level except the root, the bounds guaranteed in Lemma 5.2 are in fact worst-case bounds. The costs of acquiring locks require only waiting for every concurrent operation to complete, which can take at most $O(K^\epsilon)$. The costs of completing an insertion into the Dietz and Sleator list similarly can take at most time $O(K^\epsilon)$. To bound the time required waiting for *split-lock*s requires noticing that all C_i operations are non-concurrent, i.e., only one of them can be waiting for a split-lock at a time. Therefore, each can be blocked for the time it takes one insertion to occur at every level. For all the levels except the root, this is $O(K^\epsilon)$, leading to a cost per operation of $O(K^\epsilon/\epsilon)$, plus the cost of the root.

All C_i insertions at the root node cost in total no more than $O(I/K)$, as this is the maximum cost of all root node insertions (including waiting for locks). Similarly, this bounds the amount of time that these insertions might have to wait for a *split-lock* because of other insertions occurring at the root node.

Therefore, in total, these operations cost at most $O(I/K + C_i K^\epsilon/\epsilon)$. □

These three lemmas together guarantee the desired time bounds.

6 Exponential Search Tree

Our second algorithm is based on the exponential search tree. Again, we choose the size of the root to be I/K. Each subtree is then of size K, and decreases exponentially at every level. That is,

$$f(\ell) = \max\left(K^{\left(\frac{1}{2}^{\ell}\right)}, 2\right) .$$

Therefore, the first level (i.e., the children of the root) are of size $\sqrt{K}$, the second level is of size $K^{1/4}$, etc., until finally the lists are of size two, at which point the tree terminates. First, we determine the cost of Query operations:

Lemma 6.1 *A* Query *operation takes at most* $\log\log K$ *time.*

Proof. A simple calculation shows that the tree is at most $\log\log K + 1$ levels deep. Each subtree of the root contains K items. Since $K^{1/2^{\ell}} = 2$ when $1/2^{\ell} = O(1/logK)$, which implies that $\ell = O(\log\log K)$. This is a standard property of exponential search trees, and immediate implies that queries take $O(\log\log K)$. ☐

Lemma 6.2 *A* Insert *operation takes:*

$$O\left(\frac{p}{K} + \sqrt{K} + \log\log K\right)$$

amortized time, if there is at most p concurrency. C_i *non-concurrent operations take:*

$$O\left(\frac{I}{K} + C_i\sqrt{K}\log\log K\right)$$

Proof (sketch). This proof is, again, quite similar to that of Lemma 5.2. First we notice that, as before, the cost of inserts at the root is $O(I/K)$. This also bounds the amount of time that nodes block while waiting for *split-lock*s at the first level. At each level of the tree, there are at most $I/K^{1/2^{\ell}}$ Insert operations, each of which has at cost $K^{1/2^{\ell}}$ waiting for locks at level ℓ (i.e., not including waiting for *split-lock*s. This leads to an amortized cost of $O(1)$ per level, and hence a total of $O(\log\log K)$ per operation.

Finally, we must examine the cost of waiting for *split-lock*s. At level one, it is as in the previous case, with $O(K)$ insertions potentially waiting for $O(I/K)$ time. Level two, however, is the worst case. Here, as at other levels, there are $O(I/K^{1/2^{\ell}})$ insertions each of which blocks $K^{1/2^{\ell}}$. Each insertion into the first level, however, takes $\sqrt{K}$ time, leading to that term. The lower levels simply generate smaller terms in the geometric sequence, and hence are absorbed into the constant.

All the other costs are as in Lemma 5.2 and are omitted here. The worst-case bound comes as before from noticing that most bounds are in fact worst-case bounds, except the insertions into the root node. ☐

7 Conjectured Algorithm and Applications

In this section, we present one final algorithm, without presenting any formal analysis, and then briefly present an application of this work to the parallel non-determinator.

7.1 Modified Exponential Search Tree

While the second algorithm presented takes advantage of the exponential search tree, it does not allow the application to choose a trade-off between the cost of Query and Insert operations. Therefore, we propose that the tree begin as an exponential tree, until some level K^{ϵ} is reached, and then continue one more level at K^{ϵ} to finish the tree. Again, $f(0) = I/K$. As in the exponential search tree, we assign:

$$f(\ell) = \max\left(K^{\left(\frac{1}{2}^{\ell}\right)}, K^{\epsilon}\right) .$$

Notice that in this case, the lowest levels of the tree have size K^{ϵ}. It turns out that these trees have depth $\log 1/\epsilon$, thereby giving good query times. The resulting algorithm has insertion times somewhere between Algorithms 1 and 2.

7.2 Parallel Non-Determinator

It has been shown that the key to designing a good parallel non-determinator lies in designing a parallel order-maintenance structure [4]. In this case, when running a program with total work T_1 and critical path T_∞, there are no more than $Q = T_1$ Query operations and $I = pT_\infty$ Insert operations. Further, there are at most $C_q = T_\infty$ Query operations along the critical path, and $C_i = T_\infty$ Insert operations. Therefore, choosing $K = p$ and $\epsilon = 1/2$, the resulting non-determinator has total work:

$$O\left(T_1 + p\sqrt{p}\,T_\infty\right)$$

and a critical path of length:

$$O\left(\sqrt{p}\,T_\infty\right)$$

resulting in a total running time of:

$$O\left(\frac{T_1}{p} + \sqrt{p}\,T_\infty\right)$$

on an optimal scheduler.

8 Conclusions and Future Work

In this paper, we have presented the first algorithm for concurrent order-maintenance. We have also shown that there is a connection between B-trees and the order-maintenance problem.[3]

While we present one algorithm for concurrent B-trees that is conducive to the problem at hand, this connection suggests that alternate B-tree constructions may yield interesting results. Unfortunately, to our knowledge, none of the concurrent B-tree algorithms provide the bounds on work and critical path that are necessary for this data structure.

It therefore remains an open problem to determine good B-tree constructions that can be used to support order-maintenance. Similarly, it seems likely that non-blocking B-tree constructions can be used to develop lock-free and obstruction-free versions of this algorithm. Again, however, existing constructions do not provide the necessary analysis of work and critical path.

References

[1] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th European Symposium on Algorithms*, pages 152–164, 2002.

[2] P. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Symposium on Theory of Computing*, 1982.

[3] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Symposium on Theory of Computing*, pages 365–371, 1987.

[4] Jeremy Fineman. On-the-fly detection of determinacy races in fork-join programs, dec 2003.

[5] D. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *SIGMOD*, 1986.

[3]Notice that this is in some ways the inverse of the relationship between order-maintenance and trees that is used in [4] to develop a linear time non-determinator.