A Pipelined Architecture for Search Tree Maintenance

Michael J. Carey
Clark D. Thompson

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

A scheme for maintaining a balanced search tree on \( O(lgN) \) parallel processors is described. \( O(lgN) \) search, insert, and delete operations are allowed to run concurrently, with each operation executing in \( O(lgN) \) timesteps. The scheme is based on pipelined versions of top-down 2-3-4 tree manipulation algorithms.

1. Introduction

This paper proposes a scheme for using a linear array of \( O(lgN) \) processors to maintain a balanced tree structure for \( N \) items. The scheme allows pipelined operation so that, while individual operations require \( O(lgN) \) time, \( O(lgN) \) operations may be at varying stages of execution at any point in time. Also, the scheme avoids excessive data movement between processors.

Similar work on VLSI "dictionary machines" has been reported in recent years by Bentley and Kung [Ben79], Leiserson [Lei79], and Ottman, Rosenberg, and Stockmeyer [Ott81]. The salient feature of the scheme presented here is that \( O(lgN) \) processors are required. The earlier schemes were based on the use of \( O(N) \) processors organized in tree-like configurations.

2. Architecture

The architecture used here is a linear array of \( O(lgN) \) identical processing elements, each with their own private memory attached. Processor \( P_1 \) has memory capable of storing a single tree node, and each processor \( P_i, 1 < i < k \), has twice the amount of memory of its predecessor \( P_{i-1} \). The last processor, \( P_k \), must have memory sufficient to hold all of the data which is to be stored in the machine. Also, to
provide for the processing of range queries, the memory of processor \( P_k \) must be dual-port for external accessibility.

The processors operate independently, in an MIMD manner. The communication paths between processors are bidirectional. The resulting machine architecture, similar to the heapsort machine architecture of Armstrong [Arm78], is shown in Figure 1. In the figure, the architecture is shown storing a 2-3-4 tree using the scheme described in the next section.

![Figure 1: Parallel Architecture for Balanced Tree Maintenance.](image)

### 4. Operations for Tree Maintenance

In this section, the 2-3-4 tree manipulation operations will be described. A 2-3-4 tree is a balanced search tree where two, three, or four pointers appear in each internal (index) node, and all data items appear in external (leaf) nodes. The tree manipulations described here are basically just parallel versions of normal 2-3 or B+ tree searching [Bay72, Aho74, Com79] and top-down 2-3-4 tree insertion and deletion [Gui78, Ail80]. Thus, in the interest of brevity and clarity, the description will be somewhat informal, with actual pointer and key manipulations omitted.

#### 4.1. Searching

The SEARCH operation for the parallel 2-3-4 tree scheme is a simple, pipelined version of normal B+ tree searching [Com79]. Hence, when processor \( P_i \) receives a "SEARCH(key \( n \), using pointer \( p \))" message, it should do the following:

Case \#1 - \( P_i \) contains an index node (\( i < k \)):

Follow the pointer \( p \) to the appropriate index node in local memory. Use the key value \( n \) to select the appropriate pointer (\( p' \)) to follow from here. Send SEARCH\((n, p')\) to \( P_{i+1} \).

Case \#2 - \( P_i \) contains a data node (\( i = k \)):

Given key \( n \) and pointer \( p \), see if pointer \( p \) points at a data node containing key \( n \). If so, send the data to the outside world. If not, send out a message indicating that the desired data was not found.
4.2. Insertion

The INSERT operation for the parallel 2-3-4 tree scheme is a pipeline version of the top-down node-splitting insert algorithm of Guibas and Sedgewick [Guib78, All80, McC82]. When the search encounters a node which is full, the transformation shown in Figure 2 is applied, ensuring that future node splits will not cause upwardly propagating splits. Note that the insertion transformation results in an increase in the actual tree height when applied at the root node. The figure depicts the transformation in terms of 2-3-4 trees, with optional pointers drawn in dotted lines and the search path pointer indicated via a small black square. Though the figure shows the insertion path as being the leftmost path, the transformation applies in the obvious way regardless of the path. The correctness of this transformation is proven in [Car82].

![Figure 2: Insertion Transformation.](image)

Hence, when processor $P_i$ receives an "INSERT(key $n$, using pointer $p$)" message, it should do the following:

**Case #1** - $P_i$ contains an internal index node ($i < k - 1$):

- Follow the pointer $p$ to the appropriate index node in local memory. Use the key value $n$ to select the appropriate pointer ($p'$) to follow from here. Send INSERT_TRANSFORM($p'$) to $P_{i+1}$. $P_{i+1}$ will apply the insertion transformation if it is applicable, splitting the next node on the search path for key $n$, and send INSERT_TRANSFORM_REPLY($n, np$) to $P_i$. This reply will inform $P_i$ of the new splitting key ($m$) and new offspring pointer ($np$) resulting from a node split if one occurred, and $P_i$ will insert this information into its current index node.

- Now, once again use $n$ to select the appropriate path ($p'$) to follow from here. (The path may be different if $P_{i+1}$ performed a node split in response to the INSERT_TRANSFORM message.) Send INSERT($n, p'$) to $P_{i+1}$.

**Case #2** - $P_i$ contains the last index node ($i = k - 1$):

- Follow the pointer $p$ to the appropriate index node in local memory. Use the key value $n$ to select the appropriate pointer ($p'$) to follow from here. Send INSERT($n, p'$) to $P_k$. $P_k$ will attempt the insertion, sending back either a pointer to the newly inserted node or a nil pointer in an INSERT_REPLY($np$) message. A nil pointer indicates that the key was a duplicate and no insert took place. If no error occurred, insert the key and new pointer information into the current index node.

**Case #3** - $P_i$ contains a data node ($i = k$):

- If the indicated data item is not already present, install it in a new data node, send $P_{k-1}$ a pointer $np$ to the new node in an INSERT_REPLY($np$) message, and send the outside world an acknowledgement. If the indicated data item is already present, send $P_{k-1}$ an INSERT_REPLY(nil) message, and send the outside world an error response.

4.3. Deletion

The delete operation is a modified version of the Guibas and Sedgewick top-down deletion algorithm [Guib78]. The modification is based on the observation that old keys may be used to guide searches in $B^+$ trees [Com79], since all predecessors of a deleted key are predecessors of its successor key, and all successors of its successor key are also successors of the deleted key. Thus, it is not necessary to delete the instances of a data item's key from the index portion of a 2-3-4 tree when deleting the item.

The basic idea of top-down deletion is that, when a node with the minimum allowable number of keys is encountered, a transformation that adds keys must be performed to ensure that deletions cannot propagate upwards [Guib78, All80, McC82]. No paper in the literature has described these transformations in terms of standard 2-3-4 trees or $B^+$ trees in a particularly comprehensible manner, so they will be described here in some detail. There are three such transformations, depicted in Figures 3 through 5 for 2-3-4 trees. Note that deletion transformation $I$ can result in a decrease in the tree height when applied at the root node. While the figures depict the deletion path as being the leftmost path, the transformations apply in the obvious way regardless of the path. The correctness of these transformations is proven in [Car82].

![Figure 3: Deletion Transformation I.](image)

Hence, when processor $P_i$ receives a "DELETE(key $n$, using pointer $p$)" message, it should do the following:
Use the key value \( n \) to select the appropriate pointer \( (p') \) to follow from here. Send DELETE\( (n,p') \) to \( P_k \). \( P_k \) will attempt the deletion, sending back a status flag in a DELETE\_REPLY\( (\text{status}) \) message, indicating either success or failure. A failure indication would mean that the key was a duplicate and no deletion took place. If no error occurred, delete the appropriate key and pointer information from the current index node.

Case \#3 - \( P_i \) contains a data node \( (i = k) \):

If the indicated data item is already present, delete its data node, send \( P_{k-1} \) a DELETE\_REPLY\( (\text{no error}) \) message, and send the outside world an acknowledgement. If the indicated data item is not already present, send \( P_{k-1} \) a DELETE\_REPLY\( (\text{error}) \) message, and send the outside world an error response.

4.4. Range Queries

In order to handle range queries, two additional operations may be provided: CEILING\_SEARCH\( (n,p) \) and FLOOR\_SEARCH\( (n,p) \). The handling of these requests is identical to that of SEARCH\( (n,p) \) for all but processor \( P_k \). For this last processor, these operations cause a pointer to the indicated data item to be returned in place of the data item itself. (The appropriate data item is the one with the smallest key \( m \) such that \( m \geq n \) for the CEILING\_SEARCH operation and the one with the largest key \( m \) such that \( m \leq n \) for the FLOOR\_SEARCH operation.) If \( P_k \) keeps sequential data nodes linked together as a sequence set \([\text{Com}79]\), then sequentially executing these two search operations will be sufficient to provide external access to all of the data items in the range delimited by the two search queries.

5. Some Implementation and Performance Issues

There are several possible schemes for implementing tree transformations. Our approach, where transformation messages and replies carry all of the relevant keys and pointers, minimizes the amount of information that a parent node needs to store about each of its offspring nodes. With this scheme, a processor receiving a transformation request message decides for itself which transformation to apply, sending a reply notifying the requestor (its parent) of any new key and pointer information. The overall structure of the code executed by index processors \( P_1 \) through \( P_{k-2} \) is sketched out in Figure 6. The code for index processor \( P_{k-1} \) is similar, differing only in that \( P_{k-1} \) does not send transformation messages to processor \( P_k \). The structure of the code for data processor \( P_k \) is also similar, except that the operations performed by \( P_k \) in response to request messages from \( P_{k-1} \) are the actual data lookups, insertions, and deletions, and \( P_k \) sends response messages to the outside world. Details of the code for the last two processors may be found in \([\text{Car}82]\).

This proposal allows SEARCH, INSERT, and DELETE operations to occur in \( O(lgN) \) time. More specifically, since the mode of operation of the pipeline is based on a request/reply paradigm, half of the processors in the array can be processing requests at any given point in time.
while true do
  Receive reqMsg from $P_{i-1}$;
  case MsgType(reqMsg) of
    SEARCH:
      begin
        Perform path selection;
        Send SEARCH($n.p'$) to $P_{i+1}$;
      end;
    INSERT:
      begin
        Perform path selection;
        Send INSERT_TRANSFORM(p') to $P_{i+1}$;
        Receive INSERT_TRANSFORM_REPLY(m, np) from $P_{i+1}$;
        if $(np \neq \text{nil})$ then
          Insert np and m into current index node;
        end;
        Send INSERT($n.p'$) to $P_{i+1}$;
      end;
    DELETE:
      begin
        Perform path selection;
        Send DELETE_TRANSFORM(m, p', p'') to $P_{i+1}$;
        Receive DELETE_TRANSFORM_REPLY(m', np) from $P_{i+1}$;
        Replace old splitting key with $m'$;
        if $(np \neq \text{nil})$ then
          Delete np from current index node;
        end;
        Send DELETE($n.p'$) to $P_{i+1}$;
      end;
    INSERT_TRANSFORM:
      begin
        Perform insertion transformation if applicable;
        Send INSERT_TRANSFORM_REPLY(m, np) to $P_{i-1}$;
      end;
    DELETE_TRANSFORM:
      begin
        Perform a deletion transformation if applicable;
        Send DELETE_TRANSFORM_REPLY(m', np) to $P_{i-1}$;
      end;
  esac;
od;

Figure 6: Code for Processor $P_i$, $i = 1, 2, \ldots, k-2$.

The reason for this factor of two is that, until a processor $P_i$ receives its reply from $P_{i+1}$, the keys and/or pointers in $P_i$ may be incorrect. Thus, the attainable level of concurrency in a $k$ processor configuration is $k/2$.

The number of levels required for storing $N$ elements is easily determined as follows: In the worst case, every node is almost empty, containing only two pointer fields. In this case, the 2-3-4 tree is purely binary. Thus, the worst case number of tree levels required (including the data level) is $k = \lceil \log N \rceil + 1$. With a processor per level arrangement, this calls for $O(\log N)$ processors, as originally stated.

Instead of storing between two and four keys per index node, the number of keys per node could be allowed to range between $d$ and $2d$. This is simply a generalization of the 2-3-4 tree scheme, a variant of B+ trees [Com79] where $2d$ (instead of $2d-1$) is the allowed maximum number of keys. The corresponding insertion and deletion transformations are fairly obvious. The advantage to choosing $d > 2$ is that fewer tree levels are needed for a fixed data set size $N$, reducing the number of processors required and reducing the response time per query. On the other hand, small values of $d$ such as $d = 2$ offer the best concurrency and throughput rates, as more processors are available to process portions of queries. Thus, a tradeoff exists, and the best choice for $d$ is application-dependent.

6. Summary and Future Research

A 2-3-4 tree maintenance scheme using $O(\log N)$ processors has been described. It requires $O(\log N)$ time for tree operations, but achieves $O(1)$ throughput by allowing $O(\log N)$ concurrency on all tree operations. The extension of this scheme from 2-3-4 trees to the more general B+ tree structure is trivial. This scheme could be a useful component for index maintenance in a machine architecture specialized for information storage and retrieval.

Several avenues seem appropriate for future research along these lines. First, it would be useful to determine the worst-case memory requirements for processors in the array based on the possible tree structures allowed by the top-down algorithms. Second, it would be interesting to see if the level of attainable concurrency can be improved from $k/2$ to $k$. The work of Lehman and Yao on B-link trees [Leh81] might be applicable here. Finally, it would be interesting to investigate other classes of search problems for which a linear, pipelined array of $O(\log N)$ processors might be applicable.

References


Addendum

At the Purdue WACO conference, a question arose regarding the ability of the architecture to store all possible 2-3-4 trees of \( N \) data items. This addendum will show that, with the memories configured as described in Section 2, the architecture can indeed store all such trees. Let \( M_j \) be the size (in nodes) of the memory of processor \( P_j \). For 2-3-4 trees, if the last processor \( (P_k) \) is to store \( N \) data nodes, the memory requirements are:

\[ M_k \geq N, \quad M_{k-1} \geq \lceil N/2 \rceil, \quad M_{k-2} \geq \lfloor N/2 \rfloor, \ldots \]

Since \( \lfloor N/2^j \rfloor = \lfloor N/2^{j+1} \rfloor \) (both are equal to the binary value of \( N \) right-shifted by \( j+1 \) bits), these memory requirements can be restated in more general terms as:

\[ M_j \geq \lceil N/2^{k-j} \rceil, \quad 1 \leq j \leq k \]

In Section 5 we found that \( k = \lceil \log M \rceil + 1 \). Substituting this value for \( k \) yields:

\[ M_j \geq \lceil (N/2^{\log M}) 2^{-j} \rceil, \quad 1 \leq j \leq k \]

Since \( N/2^{\log M} \leq 1 \), the memory requirement is seen to be:

\[ M_j \geq 2^{j-1}, \quad 1 \leq j \leq k \]

Thus, it is clear that the memory configuration of Section 2, where \( M_j = 2^{j-1} \), is sufficient. Hence, the architecture can indeed store all possible 2-3-4 trees of \( N \) data items.