

# An Efficient Implementation of Search Trees on $O(\lg N)$ Processors

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

---

This work was supported by the National Science Foundation Grant ECS-8110684, a Chevron U.S.A. Career Development Grant, a California MICRO Fellowship, the Air Force Office of Scientific Research Grant AFOSR-78-3596, and the Naval Electronic Systems Command Contract NESC-N00039-81-C-0569.



# An Efficient Implementation of Search Trees on $O(\lg N)$ Processors

*Michael J. Carey  
Clark D. Thompson*

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

## 1. Introduction

This paper proposes a scheme for using a linear array of  $O(\lg N)$  processors to maintain a balanced tree structure for  $N$  items. The scheme allows pipelined operation so that, while individual operations require  $O(\lg N)$  time,  $O(\lg N)$  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  $O(\lg N)$  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  $O(\lg N)$  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.

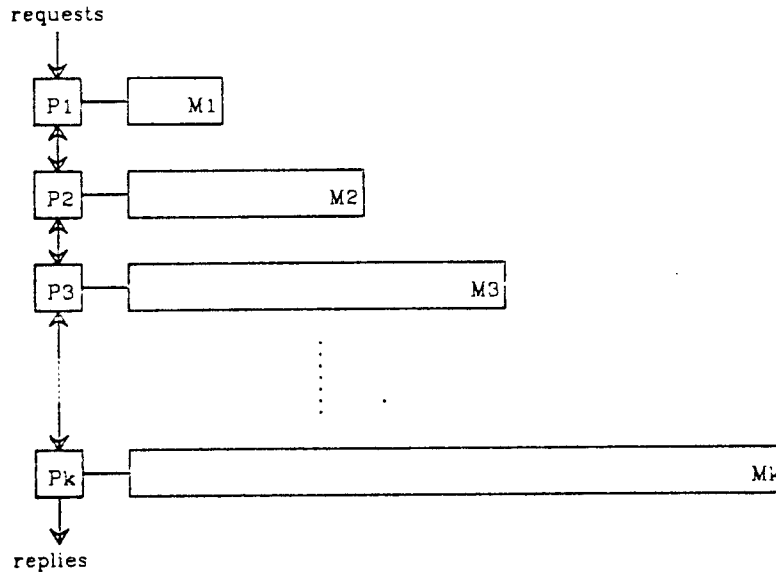


Figure 1: Parallel Architecture for Balanced Tree Maintenance.

### 3. Some Observations

Given the desiderata of parallel operation, minimal data movement, and balanced operation, the following comments are in order:

- (1) The data structures of choice are the 2-3 tree [Aho74] or the 2-3-4 tree [Gui78], since balancing operations are quite simple for these tree types. More generally, the B+ tree [Com79], a variant of B-trees [Bay72] where data resides only in leaf nodes, would be perfectly suitable here. The 2-3 tree is a special case of the B+ tree, and the 2-3-4 tree is an extension of the 2-3 tree where four-nodes are allowed in addition to two-nodes and three-nodes.
- (2) Top-down tree operations [Gui78, All80] are the only reasonable way to maintain balance in this scheme, as pipelining would be very difficult if node splits and merges were allowed to propagate upwards in the tree as a result of lower splits and merges. Thus, the 2-3-4 tree is the best choice, as this tree type lends itself to simple top-down insertion and deletion algorithms [Gui78].

- (3) The obvious way to map the problem to the architecture is to store one level of the tree with each processing element. The processor storing the leaf nodes will contain the data, and the other processors will simply store index nodes containing keys and pointers. Figure 2 illustrates a sample 2-3-4 tree stored in this way.

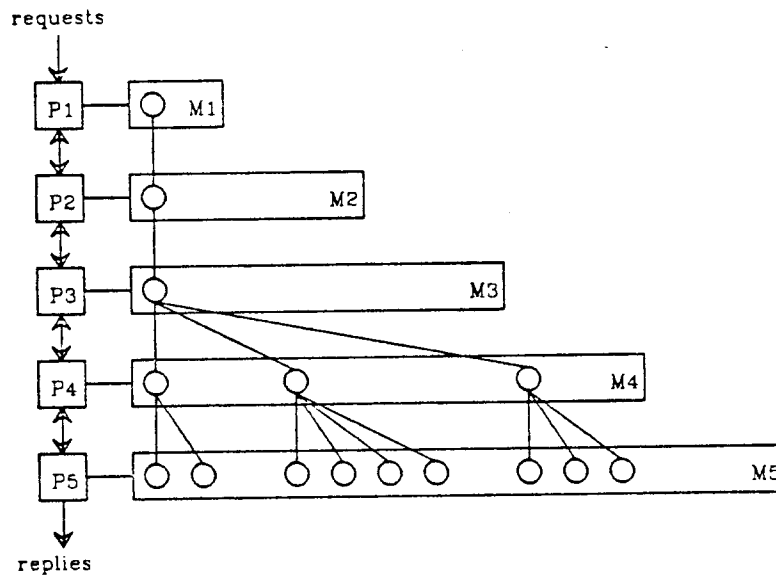


Figure 2: Example of 2-3-4 Tree Storage Layout.

- (4) When the root of the 2-3-4 tree is full, the height of the tree will have to grow by one level. Similarly, when the root of the tree and its sons are nearing the empty point, the height of the tree will have to shrink by one level. Hence, the tree should be started off on processor  $P_k$  and allowed to grow upwards towards  $P_1$  on root-splitting insertions and back down towards  $P_k$  on root-removing deletions. Processors above the level of the actual tree root will store unary nodes (nodes with a single tree pointer).

#### 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 keys 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, All80]. 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 pipelined version of the top-down node-splitting insert algorithm of Guibas and Sedgewick [Gui78, All80, McC82]. When the search encounters a node which is full, the transformation shown in Figure 3 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 the appendix.

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

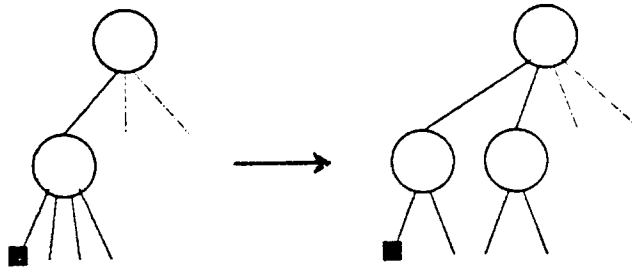


Figure 3: Insertion Transformation.

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( $m, np$ ) 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 Sedgwick top-down deletion algorithm [Gui78]. 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 [Gui78, 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 4 through 6 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 the appendix.

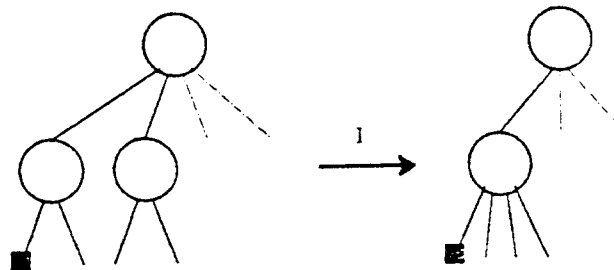


Figure 4: Deletion Transformation  $I$ .

Hence, when processor  $P_i$  receives a "DELETE(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 DELETE\_TRANSFORM( $m, p', p''$ ), where  $p''$  is the adjacent path pointer for  $p'$  and  $m$  is the splitting key for  $p'$  and  $p''$ , to  $P_{i+1}$ .  $P_{i+1}$  will apply a deletion



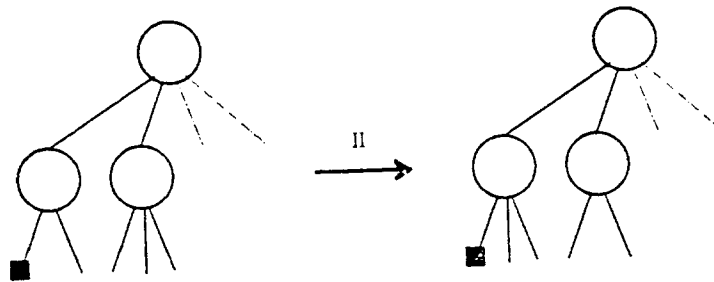


Figure 5: Deletion Transformation II.

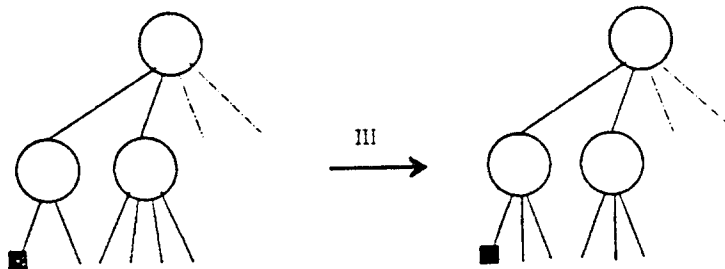


Figure 6: Deletion Transformation III.

transformation if one is applicable, either merging the nodes indicated by  $p'$  and  $p''$  or moving one of the offspring of the  $p''$  node into the  $p'$  node.  $P_{i+1}$  will then send  $\text{DELETE\_TRANSFORM\_REPLY}(m', np)$  to  $P_i$ . This reply will inform  $P_i$  of the new splitting key ( $m'$ ) resulting from a transformation if one occurred, and the pointer value  $np$  will indicate  $p'$  or  $p''$  if one of these nodes was deleted by the transformation.  $P_i$  will use this information to update 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  $P_{i+1}$  did some key and pointer rearranging in response to the  $\text{DELETE\_TRANSFORM}$  message.) Send  $\text{DELETE}(n, p')$  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  $\text{DELETE}(n, p')$  to  $P_k$ .  $P_k$  will attempt the deletion, sending back a *status* flag in a  $\text{DELETE\_REPLY}(status)$  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(*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(*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 [Com79], 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 7. The code for index processor  $P_{k-1}$  is sketched out in Figure 8, and the code for data processor  $P_k$  is sketched out in Figure 9.

This proposal allows SEARCH, INSERT, and DELETE operations to occur in  $O(\lg N)$  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. 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

```

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 ( $np \neq nil$ ) then
          Insert  $np$  and  $m$  into current index node;
        fi;
        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 ( $np \neq nil$ ) then
          Delete  $np$  from current index node;
        fi;
        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 7: Code for Processor  $P_i$ ,  $i = 1, 2, \dots, k-2$ .

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  $k = \lceil \lg N \rceil + 1$ . With a processor per level arrangement, this calls for  $O(\lg N)$  processors, as originally

```
while true do
  Receive reqMsg from  $P_{k-2}$ ;
  case MsgType(reqMsg) of
    SEARCH:
      begin
        Perform path selection;
        Send SEARCH( $n, p'$ ) to  $P_k$ ;
      end;
    INSERT:
      begin
        Perform path selection;
        Send INSERT( $n, p'$ ) to  $P_k$ ;
        Receive INSERT_REPLY( $np$ );
        if ( $np \neq nil$ ) then
          Insert  $np$  and  $n$  into the current index node;
        fi;
      end;
    DELETE:
      begin
        Perform path selection;
        Send DELETE( $n, p'$ ) to  $P_k$ ;
        Receive DELETE_REPLY( $status$ );
        if ( $status = no\ error$ ) then
          Delete  $p'$  from the current index node;
        fi;
      end;
    INSERT_TRANSFORM:
      begin
        Perform transformation if applicable;
        Send INSERT_TRANSFORM_REPLY( $m, np$ ) to  $P_{k-2}$ ;
      end;
    DELETE_TRANSFORM:
      begin
        Perform transformation if applicable;
        Send DELETE_TRANSFORM_REPLY( $m', np$ ) to  $P_{k-2}$ ;
      end;
  esac;
od;
```

Figure 8: Code for Processor  $P_{k-1}$ .

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

```
while true do
  Receive reqMsg from  $P_{k-1}$ ;
  case MsgType(reqMsg) of
    SEARCH:
      begin
        if (data item found) then
          Send out data item;
        else
          Send out error response;
        fi;
      end;
    INSERT:
      begin
        if (data item not found) then
          Insert data item;
          Send INSERT_REPLY( $np$ ) to  $P_{k-1}$ ;
          Send out acknowledgement;
        else
          Send INSERT_REPLY( $nil$ ) to  $P_{k-1}$ ;
          Send out error response;
        fi;
      end;
    DELETE:
      begin
        if (data item found) then
          Delete data item;
          Send DELETE_REPLY( $no\ error$ ) to  $P_{k-1}$ ;
          Send out acknowledgement;
        else
          Send DELETE_REPLY( $error$ ) to  $P_{k-1}$ ;
          Send out error response;
        fi;
      end;
  esac;
od;
```

Figure 9: Code for Processor  $P_k$ .

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  $O(\lg N)$  processors has been described. It requires  $O(\lg N)$  time for tree operations, but achieves  $O(1)$  throughput by allowing  $O(\lg N)$  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  $O(\lg N)$  processors might be applicable.

## References

- [Aho74] Aho, A., Hopcroft, J., and Ullman, J., "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Co., 1974.
- [All80] Allchin, J., Keller, A., and Wiederhold, G., "FLASH: A Language-Independent, Portable File Access System", Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1980.
- [Arm78] Armstrong, P., U.S. Patent 4131947, issued December 26, 1978.
- [Bay72] Bayer, R., and McCreight, E., "Organization and Maintenance of Large Ordered Indices", Acta Informatica 1(3), 1972.
- [Ben79] Bentley, J., and Kung, H., "Two Papers on a Tree-Structured Parallel Computer", Report CMU-CS-79-142, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.
- [Com79] Comer, D., "The Ubiquitous B-Tree", Computing Surveys 11(2), June 1979.
- [Gui78] Guibas, L., and Sedgewick, R., "A Dichromatic Framework for Balanced Trees", Proc. 19th Symposium on the Foundations of Computer Science, 1978.
- [Leh81] Lehman, P., and Yao, S., "Efficient Locking for Concurrent Operations on B-Trees", ACM Transactions on Database Systems 6(4), December 1981.
- [Lei79] Leiserson, C., "Systolic Priority Queues", Report CMU-CS-79-115, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.
- [McC82] McCord, R., Personal Communication.
- [Ott81] Ottman, T., Rosenberg, A., and Stockmeyer, L., "A Dictionary Machine (for VLSI)", Report RC 9060 (#39615), Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1981.

## Appendix: Transformation Correctness Proofs

This appendix contains proofs of the correctness of the tree transformations given in the body of the paper. It is shown that the both the insertion transformation and the deletion transformations always produce correct 2-3-4 tree structures. That is, given that they are applied to 2-3-4 trees which were built using the insertion and deletion algorithms described in the paper, they never yield incorrect substructures such as one-nodes or five-nodes.

In the remainder of the appendix, the term "root" refers to the actual root node, the first non-unary node on a path starting at processor  $P_1$ . The index of the processor containing the root will be taken to be  $r$ ,  $1 \leq r \leq k$ .

### Insertion Transformation

The top-down insertion algorithm described in the paper will suffice to handle all insertions, correctly maintaining the 2-3-4 tree structure, as long as the tree does not overflow the architecture (meaning that it reaches a height of  $k+1$ ). Since the insertion transformation involves splitting a four-node into two two-nodes, it will maintain the 2-3-4 tree structure correctly as long as the parent of a four-node it splits is not a four-node as well. Otherwise, a five-node would be created as a result of the split. Thus, to show correctness, it suffices to show the following:

*Claim:* Suppose the tree segment of Figure 10 exists at the beginning of an insert operation, with the insertion path as indicated in the figure. Suppose that node  $X$  is stored in processor  $P_i$ ,  $i \geq r$  and  $r > 1$ . (If  $P_r$  contains a four-node root node and  $r = 1$ , no further insertions are possible, as the architecture is considered to be full.) By the time processor  $P_i$  receives its INSERT( $n, p$ ) message,  $X$  will no longer be a four-node.

*Proof:* Noting that the insertion transformation has already been applied (if it was applicable) at the parent node of  $X$  when the INSERT( $n, p$ ) message arrives at processor  $P_i$ , we proceed by induction on  $i$ . For the basis, consider the insertion transformation when node  $X$  is stored in  $P_r$ , meaning that node  $X$  is the root node. The transformation applied at  $P_{r-1}$  will already have converted the four-node configuration in  $P_r$  into a legal "double-binary" configuration involving  $P_{r-1}$  and  $P_r$ , increasing the tree height by one in the process. Hence, by the time  $P_r$  receives its INSERT( $n, p$ ) message,  $X$  will no longer be a four-node.

For the induction step, assume that the claim holds for the first  $K$  levels of the tree,  $K \geq r$ , and consider a node  $X$  at level  $K+1$ . By the induction hypothesis,  $X$ 's parent must be of degree three or less, as the transformation applied at its own parent will guarantee this. Thus, even if  $X$  is a four-node, the troublesome configuration cannot arise. ■



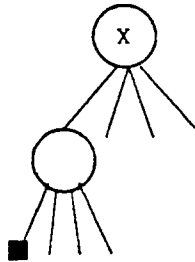


Figure 10: Unacceptable 2-3-4 Tree Configuration for Insertion.

### Deletion Transformations

The top-down deletion algorithm described in the paper will suffice to handle all deletions, correctly maintaining the 2-3-4 tree structure. The only deletion transformation that could possibly produce an incorrect 2-3-4 tree structure is transformation  $I$ . Were this transformation applied to a two-node whose offspring are two two-nodes, such as node  $X$  in Figure 11, a one-node would be produced. To prove that this cannot happen with the set of deletion transformations presented in the paper, it suffices to show the following:

*Claim:* Suppose the tree segment of Figure 11 exists at the beginning of a delete operation, with the deletion path as indicated in the figure. Suppose that node  $X$  is stored in processor  $P_i$ ,  $i \geq \tau$ . By the time processor  $P_i$  receives its  $\text{DELETE}(n, p)$  message,  $X$  will no longer be a two-node, except when  $i = \tau$ . In this latter case,  $X$  is the root node, and the tree height will be reduced by one level.

*Proof:* Noting that a deletion transformation has already been applied (if one was applicable) at the parent node of  $X$  when the  $\text{DELETE}(n, p)$  message arrives at processor  $P_i$ , we proceed by induction on  $i$ . For the basis, consider transformation  $I$  when node  $X$  is stored in  $P_\tau$ , meaning that node  $X$  is the root node. This transformation applied at  $P_\tau$  will convert  $X$  into a unary node, merging  $X$ 's two offspring nodes into a single, new, four-node root node at  $P_{\tau+1}$ , lessening the tree height by one in the process.

For the induction step, assume that the claim holds for the first  $K$  levels of the tree,  $K \geq \tau$ , and consider a node  $X$  at level  $i = K+1$ . Assume that the claim does not hold for this node. Suppose  $X$  has parent node  $XP$  at level  $K$ . There are only two possible cases to consider:

- (1)  $XP$  has two sons. By the induction hypothesis, the other son must have greater than two sons of its own. However, were this true, one of transformations  $II$  or  $III$  would have been applied when the algorithm reached  $P_{i-1}$ , and  $X$  would then have been transformed into a three-node before  $P_i$  received its  $DELETE(n,p)$  message. Thus, this case is impossible.
- (2)  $XP$  has more than two sons. In this case, one of transformations  $I$ ,  $II$ , or  $III$  would have been applied when the algorithm reached  $P_{i-1}$ , and  $X$  would then have been transformed into either a four-node or a three-node before  $P_i$  received its  $DELETE(n,p)$  message. Thus, this case is also impossible.

Since both of these cases have led to contradictions, the claim must be true. ■

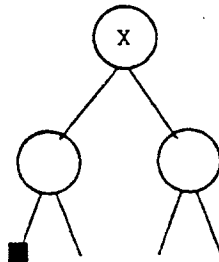


Figure 11: Unacceptable 2-3-4 Tree Configuration for Deletion.