Contents

3.3 Path Copying

We wish to determine the appropriate technique for implementing Immutable Data Structure in a concurrent execution environment. In a serial execution environment ease of implementation and performance are important considerations. However, in a concurrent execution environment ensuring the consistency of the data structure is the primary consideration. This section reviews the techniques for implementing Immutable Data Structures described in the literature.

The main contribution of this section is an examination of techniques for implementing Immutable Data Structures. This section focuses on the applicability of each technique in a concurrent execution environment.

3.3.1 Implementing Immutable Data Structures

Techniques for implementing Immutable Data Structures are described in the seminal work of Driscoll [DSST86]. This section examines how a complete immutable binary tree, with values on leaves, can be implemented using these techniques.

Full copying

The easiest technique for making a data structure immutable is to copy the entire data structure including the application values when any change is made. This technique is called naïve copying. A similar technique called full copying causes the structure to be copied while leaving the application values in place.

Figure 3.1 illustrates how the full copy technique is used to maintain a complete immutable binary tree.

The children of an immutable node should be copied before the node itself. Typically, a full copy of an immutable tree is implemented using a post-order traversal of the nodes. The performance overheads of full copying are significant so the technique has received little attention.

Path copying

The copying of the data structure can be restricted to the copying of those nodes that are modified by the operation. A path is a set of connected nodes linking the root with one or more leaves. A path copy operation creates a new path within

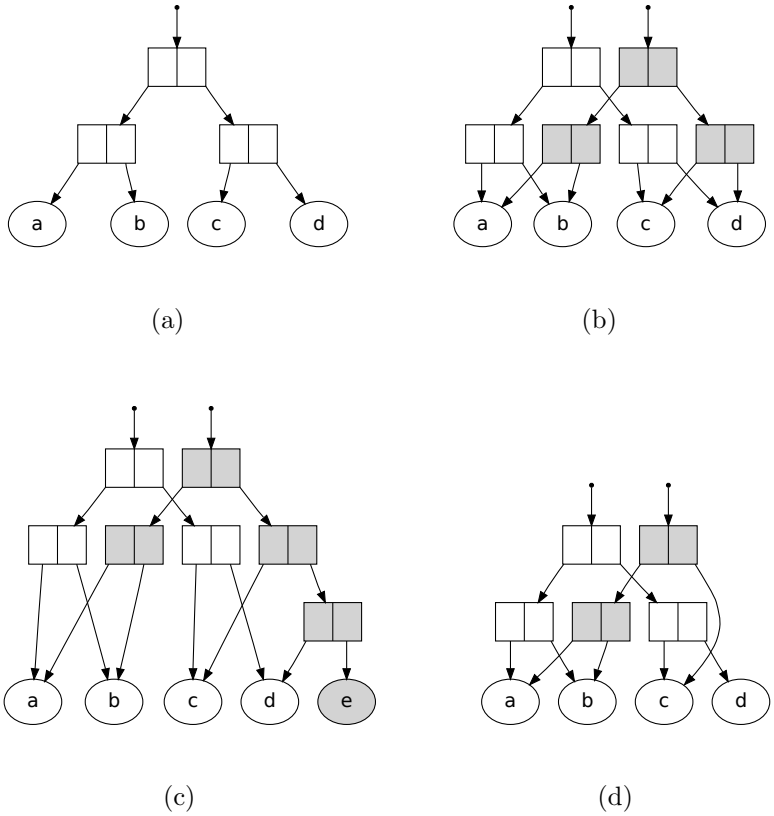


Figure 3.1: **Full copying technique.**
(a) Original complete binary tree.
(b) A full copy duplicates the data structure. The nodes created during the operation are shaded. Each node of the data structure is copied to create a new structure.
(c) The leaf e is inserted into the data structure by copying all of the nodes and adding a new node.
(d) The leaf d is removed from the data structure by copying all of the nodes except the parent of the leaf being removed.

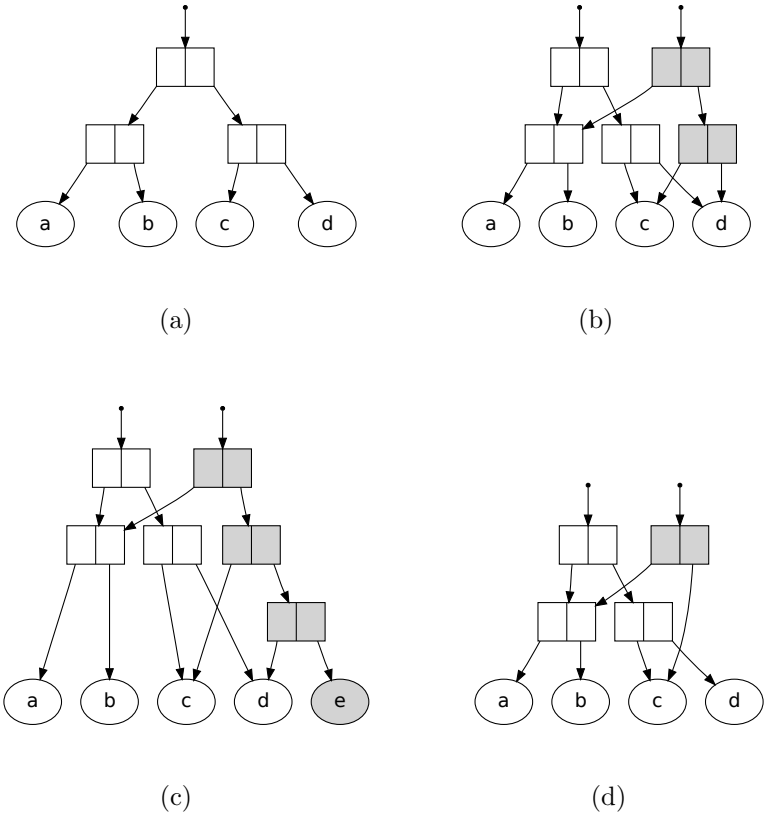


Figure 3.2: **Path copying technique.**
(a) Original complete binary tree.
(b) A leaf to root path copy. The nodes created during the operation are shaded. Each node on the path is copied to make a new data structure that has nodes in common with the original data structure.
(c) The leaf e is inserted into the data structure by copying the path to a leaf and adding a new node.
(d) The leaf d is removed from the data structure by copying the path to the parent of a leaf.

the data structure. A path copy operation may create a new path by copying an existing path from leaf to root or by selectively copying nodes in some other way.

Figure 3.2 illustrates how root to leaf path copying technique is used to maintain a complete immutable binary tree.

In the context of functional programming languages an Immutable Data Structure maintained using the path copying technique is called a purely functional data structure.

Section 3.3.3 discusses path copying in more detail.

Fat node

The fat node technique is based on the idea of recording changes to the data structure within the nodes themselves. The nodes modified by an operation are regarded as alternatives. Fat nodes can be implemented either by a list of alternative values for a node or by a list of alternative values for each pointer within a node.

Figure 3.3 illustrates how the fat node technique is used to maintain a complete immutable binary tree.

The fat node technique can be implemented by augmenting a node with an additional reference. If this reference is null then the node is the most recent and its children are interrogated to determine the path. If the reference is not null then the node or pointer has been superseded by a new value so the reference should be followed instead.

An access function determines the correct alternative based on a version number. Driscoll describes how alternatives can be associated with a range of version numbers [DSST86].

The fat node technique requires less copying than the path copy technique, because a path copy copies all of the nodes that the fat node copies together with additional nodes to the root.

The fat node technique is fairly easy to implement. However, long lists of alternative nodes can build up and these lists can degrade performance. To improve performance the lists can be split using the path copying technique but this comes at the cost of additional implementation complexity.

The fat node technique has received attention as the basis of maintaining persistent data structures in computational geometry.

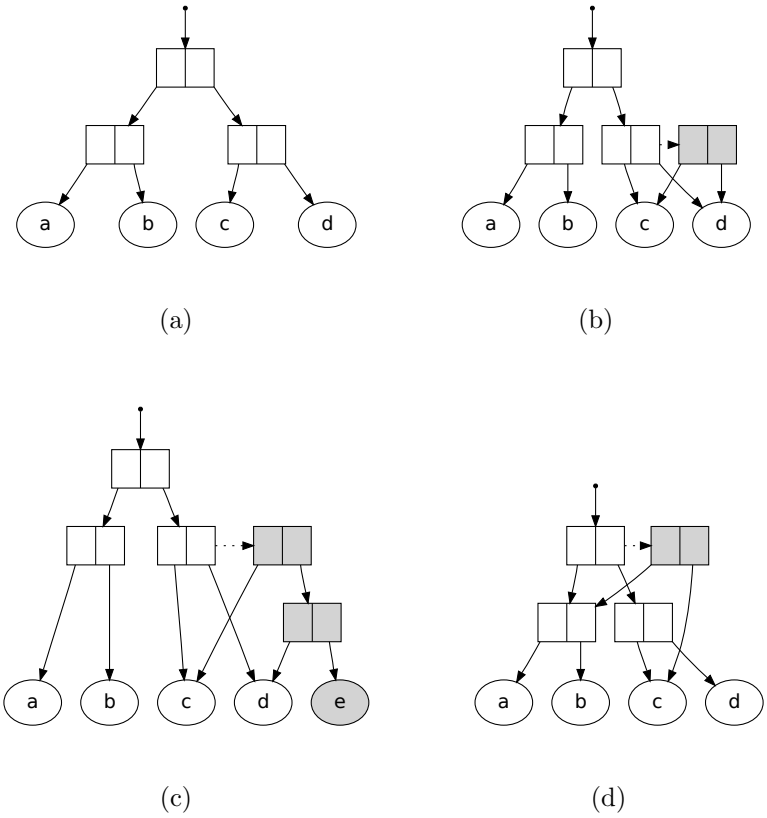


Figure 3.3: **Fat node technique.**
(a) Original complete binary tree.
(b) A node is added to the fat node. New nodes are created as needed. The nodes joined by the horizontal dotted line are regarded as part of a single fat node. The fat node is a list of past values for the node. The nodes created during the operation are shaded.
(c) The leaf e is inserted into the data structure by the creation of an alternative value for the parent of the leaf c.
(d) The leaf d is removed from the data structure by the creation of an alternative value for its grandparent.

Node copying

The node copying technique is similar to the fat node technique except that a finite number of alternatives are maintained within a node instead of in a list. These can either be alternative values for pointers or alternative values for the node. When the node becomes full it may be split. Cole describes the node copying technique in relation to persistent data structures [Col86].

Figure 3.4 illustrates how the node copying technique is used to maintain a complete immutable binary tree.

A node contains alternative values for the references to its children which are initialised to null and used if set. Each node in the data structure contains a set of alternative values. Typically, the alternative pointers are set to null when the node is created and subsequently modified when a path is modified. When a node becomes full it is split into multiple nodes by creating new paths to nodes in a similar manner to the path copying technique.

The path copying technique requires that a new path containing copies of all the nodes from leaf is created during each operation, whereas the node copying technique copies only part of the path, so the node copying technique often performs better than the path copying technique. Typically, two alternative values are stored in a single cache line so the overhead of checking whether the alternative value is in use is low.

The node copying technique is more difficult to implement than the fat node technique because node splitting requires that the path copying technique must also be implemented.

The node copying technique has also received attention as the basis of maintaining persistent data structures in computational geometry.

3.3.2 Supporting Concurrent Access

It is useful to distinguish between an immutable memory location and one that is singly-assigned. An immutable memory location is always observed to contain a constant value and it is unreachable until it is assigned. A singly-assigned memory location can be observed to contain either no value or a value that is constant once set.

In the context of concurrent execution an immutable memory location can be safely written because it is unreachable in its uninitialised state, whereas a shared

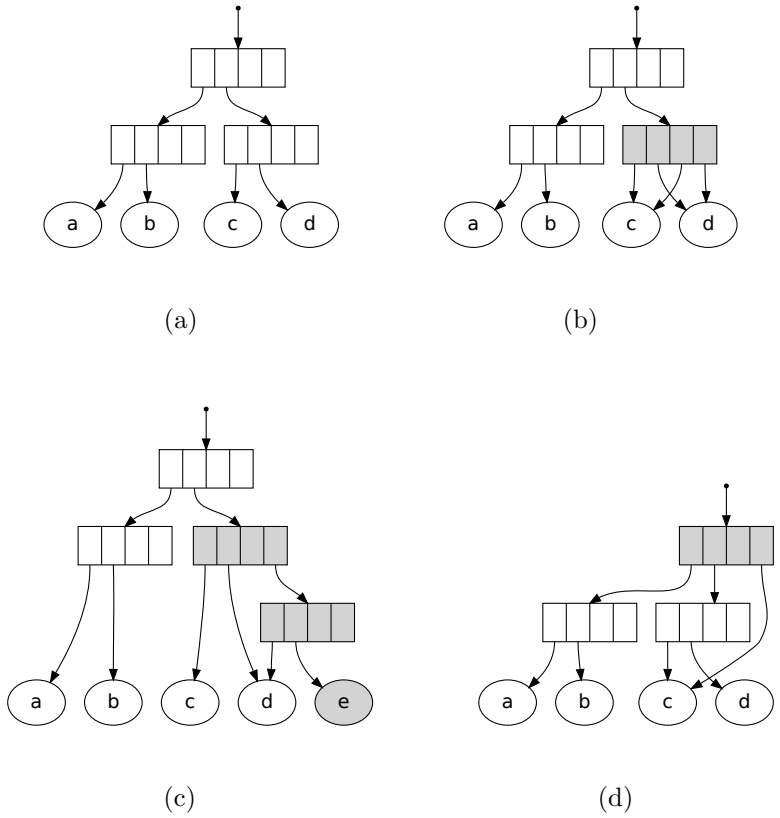


Figure 3.4: **Node copying technique.**
(a) Original complete binary tree.
(b) A node is copied. The nodes affected by the operation are shaded. A node contains a left and right pointer and an alternative left and right pointer. In order to find a leaf a test is done to determine whether the alternative pointers are null and if not the new path is taken.
(c) The leaf e is inserted into the data structure by using the alternative right pointer of the parent of the leaf c.
(d) The leaf d is removed from the data structure by using the alternative right pointer of its grandparent.

singly-assigned memory location must always be written by an atomic instruction to prevent race conditions.

An immutable memory location can be safely cached by multiple processors without requiring any mechanism to ensure cache coherence, whereas a singly-assigned memory location may not because the location can be accessed and cached in an uninitialised state and is, in effect, mutable. A cache coherency mechanism is required to ensure that all processors observe the correct value of a singly assigned memory location.

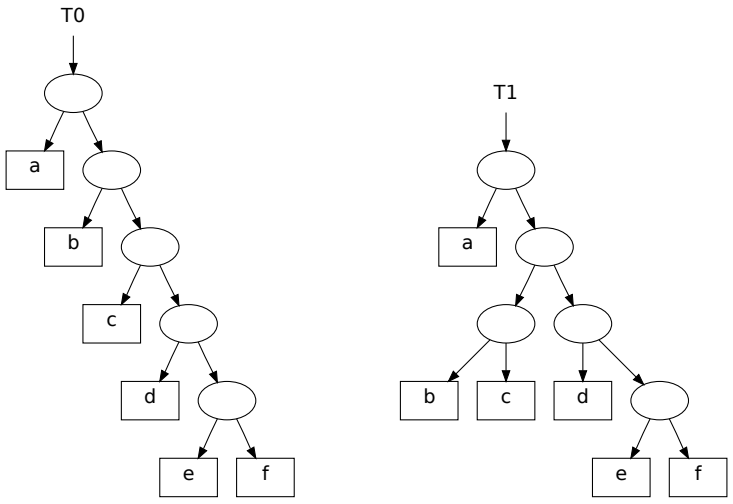
The fat node and node copying techniques both require that a singly-assigned memory location is checked for a null value within the data structure. The fat node technique checks whether the alternative value is null and the node copying technique checks whether a reference to an alternative node is null. In a concurrent execution environment the fat node and node copying techniques require that these shared singly-assigned locations must be modified by an atomic hardware instruction. Coherent caches and memory barriers are also required to support these singly-assigned values. These restrictions mean that the fat node and node copying techniques are not suitable for implementing Immutable Data Structures in a concurrent execution environment.

The full copy and path copy techniques do not rely on singly-assigned memory locations. In the context of concurrent execution the path copying and full copying techniques share the advantage, over the other techniques, that modifications to the data structure can be made to appear atomic because a new version of the data structure only becomes visible to other processors when the root node is written. They also share the advantage that all values are immutable so they can be cached in processor-unique caches without requiring that these caches implement a coherency protocol.

3.3.3 Path copying transformations

Okasaki relates trees to number systems [Oka98]. We can use this relationship to show that the path copying technique can be used to transform the topology of an immutable binary tree arbitrarily.

Let S be the set of expressions {{a.b.c.d.e.f},{a.(b.c).d.e.f},...} in which the letters a to f are in the same order. The set S corresponds to the set of possible topologies of the binary tree in which the leaves are in the same order from left to right. If the binary operator '.' is regarded as being associative then all of the



(a) (b)

Figure 3.5: **Bracket operations.**
(a) A degenerate tree T0 corresponding to the expression {a.b.c.d.e.f} in which the binary operator '.' acts on the values a to f, which are represented by the leaves in order from left right.
(b) A tree T1 corresponding to an expression in which the brackets have been re-arranged to cause some operations to take precedence over others. The operation acting on b and c takes precedence and the expression can be written as {a.(b.c).d.e.f)}.

expressions in the set S are equivalent. This topological equivalence is the basis of tree balancing.

For any expression in S there is a transformation, essentially removing brackets, that maps it onto the expression {a.b.c.d.e.f}. Similarly, for any expression in S there is a transformation, essentially adding brackets, that maps the expression {a.b.c.d.e.f} onto it.

Figure 3.5 illustrates bracket operations acting on a degenerate tree.

For any tree topology there is a transformation that maps it onto an equivalent degenerate tree. Similarly, for any tree topology there is a transformation that maps the degenerate tree onto it. These transformations can be broken down into a finite sequence of steps, each of which corresponds to adding or removing brackets from the expression.

By using a similar informal argument we can show that a value can be added

to or removed from an expression and that this corresponds to the insertion and deletion of leaves in a degenerate tree.

Implementing topological changes using path copying

The operations that add or remove brackets from an expression correspond to topological transformations of the tree. These transformations can be implemented using the path copying technique.

Figure 3.6 illustrates bracket operations implemented by path copying.

Given a degenerate tree it is possible to transform it into an equivalent tree with any topology without altering the original by using the path copying technique. Similarly, it is possible to transform a tree with any topology into an equivalent degenerate tree without altering the original.

It is interesting to note that these trees really are equivalent. The trees T0 and T1 in figure 3.5 are equivalent in the vague sense that an in-order traversal returns the same values in the same order, whereas the trees T2 and T3 in figure 3.6 are equivalent because the same leaves are shared by both trees, they are in the same order and they exist at the same moment in time.

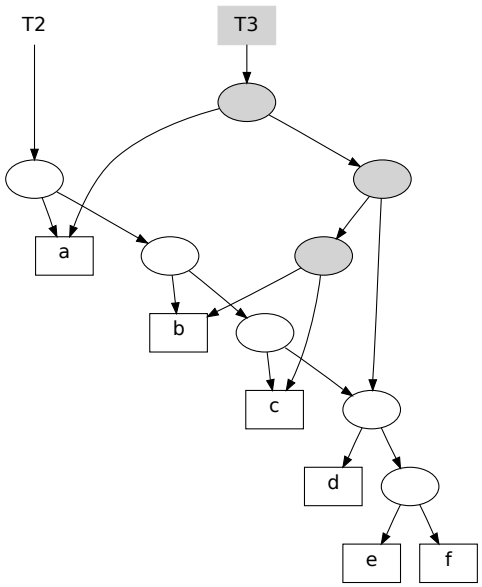
Implementing structural changes using path copying

The operations that add or remove values from an expression correspond to structural transformations of the tree.

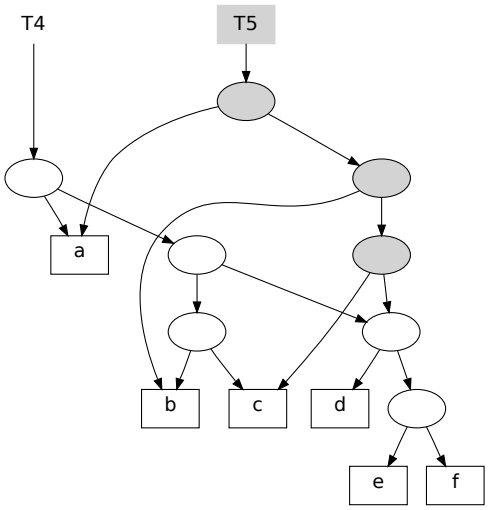
A value can be inserted into an expression at any point and correspondingly a leaf can be inserted into the degenerate tree at any point to create a new degenerate tree. A value can be removed from an expression at any point and correspondingly a leaf can be removed from a degenerate tree at any point to create a new degenerate tree.

Figure 3.7 illustrates how these transformations can be implemented using the path copying technique.

By using the path copying technique it is possible to make structural changes to a tree, by inserting or removing an arbitrary number of leaves, without altering the original.

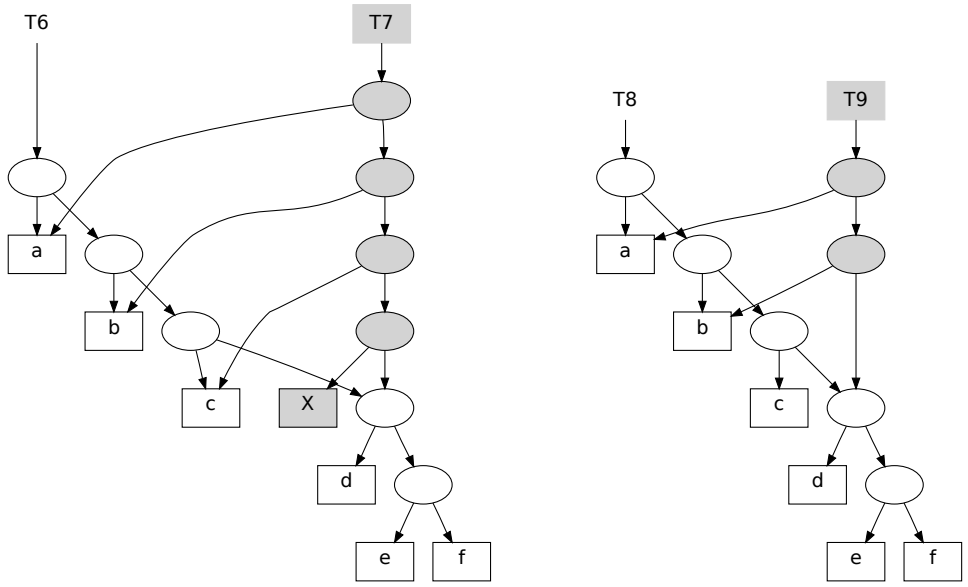


(a)



(b)

Figure 3.6: **Immutable add bracket and remove bracket operations.**
(a) A tree corresponding to an expression in which the precedence of one operation has been elevated. Degenerate tree T2 representing the expression {a.b.c.d.e.f} and the tree T3 representing the expression {a.(b.c).d.e.f}.
(b) A tree corresponding to an expression in which the precedence of one operation has been reduced. Tree T4 representing the expression {a.(b.c).d.e.f} and the degenerate tree T5 representing the expression {a.b.c.d.e.f}.



(a) (b)

Figure 3.7: **Immutable insert and delete operations.**
(a) Insertion of an item into the degenerate binary tree. Degenerate tree T6 representing the expression {a.b.c.d.e.f} and the degenerate tree T7 representing the expression {a.b.c.X.d.e.f}.
(b) Removal of an item from the degenerate binary tree. Degenerate tree T8 representing the expression {a.b.c.d.e.f} and the degenerate tree T9 representing the expression {a.b.d.e.f}.

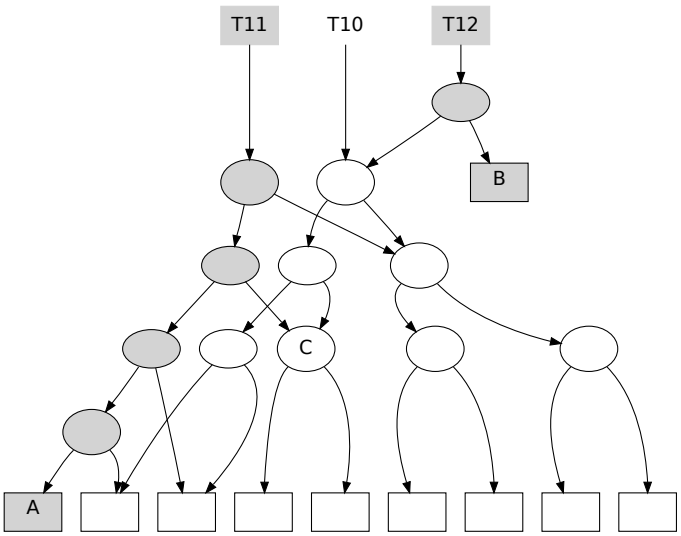


Figure 3.8: **The leaf to root path copying technique** is distinct from other path copying techniques because it preserves the position of nodes relative to the root.
A new leaf A is added to tree T10 by a root to leaf path copy to create tree T11. The path copy preserves the position of nodes that are not copied relative to the root. For example, the node C is the right child of the root's left child in both versions.
A new leaf B is added to tree T10 by another path copying technique to create tree T12. The relative positions of the existing nodes are not preserved.

3.3.4 Previous work

A leaf to root path copy operation preserves the position of existing nodes relative to the root, but not all path copy operations preserve the relative position of existing nodes.

The literature does not make a distinction between a leaf to root path copy or the creation of a new version through the construction of a new path in some other way. Driscoll describes path copying without detailed consideration of how it is achieved [DSST86]. Okasaki describes many copying optimisations which are not leaf to root copies [Oka98].

Figure 3.8 illustrates the distinction between leaf to root path copy and other path copying techniques.

Bibliography

[Col86] Richard Cole. Searching and storing similar lists. *J. Algorithms*, 7(2):202–220, 1986.

[DSST86] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998.