Concurrent, Lock-Free Insertion in Red-Black Trees

BY

JIANWEN MA

A Thesis

Submitted to the Faculty of Graduate Studies

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

October 22, 2003

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

Concurrent, Lock-Free Insertion in Red-Black Trees

BY

JIANWEN MA

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

MASTER OF SCIENCE

Abstract

In shared memory parallel computer systems, concurrent processes operate on shared data structures. The consistency of the data structure is commonly maintained using mutual exclusion, which is implemented by locking. Locking is an easy way of ensuring the consistency of the shared data structures, but it has several drawbacks. The major drawback is that it increases the latency of all the processes as the shared data structure is accessed in a serial manner. It can also result in undesirable execution orderings (e.g. "convoying" where a number of high priority processes are queued waiting for a low priority process that happens to have the lock) and even in deadlock where the process currently accessing the shared data structure may have to be pre-empted or terminated.

A different approach that avoids mutual exclusion implements lock-free data structures using optimistic synchronization primitives such as `Compare & Swap (CAS)` and `Double Compare & Swap (DCAS)`. Rather than preventing concurrency to ensure consistency, optimistic techniques allow the concurrency to take place and then ensure consistent results afterwards. Using such techniques, processes can concurrently read and write these data structures and still maintain their consistency. This is a particularly efficient approach when the probability of concurrent access is relatively low and is very attractive for parallel programming where long delays due to locking are unacceptable. Lock-free data structures reduce the latency of concurrent processes while avoiding deadlocks and other negative, scheduling-related anomalies.

The use of lock-free techniques has generally been limited to relatively simple data structures (e.g. lists and queues) that have limited interconnection between nodes and where updates to the structures are highly localised. This has left out a large number of data structures of importance to the computing community. Perhaps foremost among these are the tree structures. Tree-based data structures are heavily used in many areas of computer science and are found at the heart of many different algorithms. Various types of balanced trees are of particular importance due to the performance guarantees they can offer for such basic

operations as searching for a node in a tree, inserting a new node in a tree and deleting old nodes from a tree. The chief reason why lock-free implementations of such structures have not been done is difficulty of designing such implementations. Lists and queues can be relatively easily updated by, typically, changing a single pointer within the structure. Making updates to balanced trees is considerably more complex, typically requiring multiple operations to be performed at different points in the tree (i.e. non-local updates).

This thesis presents a lock-free insertion algorithm for a type of balanced tree known as red-black trees. The algorithm presented in this thesis adds "guard" variables to each node in the red-black tree. These variables are carefully set and cleared using CAS-type optimistic synchronization primitives to prevent concurrently executing processes from interfering with one another. In essence, each process guards the explicit region of the tree where it may make changes. This approach also ensures that concurrent processes are prevented from "passing" one another in the tree which could cause incorrect (i.e. unbalanced) trees to result. A lock-free insertion algorithm is developed and arguments are also made as to its correctness.

Acknowledgements

I am indebted to my supervisors Dr. Peter Graham and Dr. Helen Cameron for their guidance and support during my thesis research.

I would also like to thank my other committee members for their valuable suggestions while proofreading my thesis document.

I must also express my sincere gratitude to my parents, brother and husband whose constant encouragement and support motivated me throughout my Master's Program.

Finally, to all my friends, who have made my stay in Winnipeg a memorable one, I thank you for your wonderful company.

Table of Contents

1 Introduction 10
1.1 Tree-Based Data Structures 10
1.2 Parallel Computing 10
1.3 Synchronization in Parallel Programming 11
1.4 Thesis Organization 13
2 Related Work 14
2.1 Synchronization Primitives 14
2.2 The ABA Problem 16
2.3 Existing Work on Lock-Free Structures 18
2.3.1 Herlihy's Design 18
2.3.2 Prakash's Design 19
2.3.3 Valois' Design 20
2.3.4 Greenwald and Cheriton's Design 22
2.3.5 Farook's Design 24
3 Problem Description 27
3.1 Red-Black Tree Definition 27
3.2 Operations on Red-Black Trees 27
3.2.1 Red-Black Tree Node Structure 27
3.2.2 Rotations 28
3.2.3 Tree Traversal 31
3.2.4 Insertions 32
3.2.5 Deletions 40
3.3 Potential Problems with Concurrent Insertions 41
3.3.1 Case Recognition 41
3.3.2 Case Analysis 43
4 Performing Concurrent Insertions 73
4.1 Concurrent Insertions 73
4.1.1 Simple Locking 73
4.1.2 A Naïve Lock-Free Method 74
4.1.3 A Lock-Free Method with Operation Combining 74
4.2 Concurrent Rotations and Colour Updates 75
4.2.1 Naïve Application of CASn 75
4.2.2 Careful Use of CASn 78
4.2.3 Operation Combining 79
4.3 Selected Solution Strategy 79
4.3.1 Concurrent Insertions 79
4.3.2 Concurrent Rotations and Colour-updates 79
5 The Algorithm and Its Correctness 81
5.1 The Implementation of Red-Black Tree Nodes 81
5.2 General Algorithm Description 81
5.3 Global Variable Declarations 84
5.4 Creating an Empty Red-Black Tree 84
5.5 Concurrent Insertion 85
5.5.1 Tree Traversal 87

5.5.2 Placing a New Node in the Tree 90
5.5.3 Rebalancing the Red-Black Tree After an Insertion 93
5.5.4 Concurrent Local Colour Update and/or Rotation(s) 98
5.5.5 Concurrent Left Rotation 100
6 Conclusions and Future Work 104
6.1 Conclusions 104
6.2 Future Work 105
7 Bibliography 106

List of Figures

Figure 2-1–CAS synchronization primitive. 15
Figure 2-2–DCAS synchronization primitive. 15
Figure 2-3–Generic CASn synchronization primitive. 16
Figure 2-4–Avoiding the ABA problem using DCAS. 17
Figure 2-5–Structure of Valois linked list. 20
Figure 2-6–Valois' deletion algorithm. 21
Figure 2-7–Deleting extra auxiliary nodes. 22
Figure 3-1–Left rotation. 28
Figure 3-2–Right rotation. 29
Figure 3-3–Rotation when the links pointing to red nodes are both to the right. 30
Figure 3-4–Rotation when the links pointing to red nodes are right then left. 30
Figure 3-5–Rotation when the links pointing to red nodes is both to the left. 31
Figure 3-6–Rotation when the links pointing to red nodes are left then right. 31
Figure 3-7–Red-black tree traversal 32
Figure 3-8–Algorithm RB-Insert: red-black tree insertion. 33
Figure 3-9–Algorithm InsertFixup: rebalance tree after insertion. 35
Figure 3-10–Initial red-black tree (insertion example 1). 35
Figure 3-11–State after adding node x (insertion example 1) 36
Figure 3-12–The tree after initial re-colouring (insertion example 1). 36
Figure 3-13–The tree after second re-colouring (insertion example 1). 37
Figure 3-14–The tree after all re-colourings (insertion example 1) 37
Figure 3-15–Initial red-black tree (insertion example 2). 38
Figure 3-16–State after adding node x (insertion example 2) 38
Figure 3-17–The tree after re-colouring (insertion example 2). 39
Figure 3-18–The tree after the left rotate (insertion example 2) 39
Figure 3-19–The tree after the right rotate, etc. (insertion example 2). 40
Figure 3-20–The "zigzag" shape required for Case2. 42
Figure 3-21–The "straight" shape required for Case3. 42
Figure 3-22–P1 and p2 both in Case 1: first placement of p1, first access pattern. 45
Figure 3-23–P1 and p2 both in Case 1: first placement of p1, second access pattern. 46
Figure 3-24–P1 and p2 both in Case 1: first placement of p1, third access pattern. 46
Figure 3-25–P1 and p2 both in Case 1: first placement of p1, fourth access pattern. 47
Figure 3-26–P1 and p2 both in Case 1: second placement of p1 47
Figure 3-27–P1 and p2 both in Case 1: third placement of p1. 48
Figure 3-28–P1 and p2 both in Case 1: fourth placement of p1. 48
Figure 3-29–P1 in Case 1, p2 in Case 2: first placement of p1, first access pattern. 49
Figure 3-30–P1 in Case 1, p2 in Case 2: first placement of p1, second access pattern. 50
Figure 3-31–P1 in Case 1, p2 in Case 2: second placement of p1, first access pattern. 50
Figure 3-32–P1 in Case 1, p2 in Case 2: second placement of p1, second access pattern. 51
Figure 3-33–P1 in Case 1, p2 in Case 2: third placement of p1, first access pattern. 51
Figure 3-34–P1 in Case 1, p2 in Case 2: third placement of p1, second access pattern. 52
Figure 3-35–P1 in Case 1, p2 in Case 2: fourth placement of p1, first access pattern. 52

Figure 3-36–P1 in Case 1, p2 in Case 2: fourth placement of p1, second access pattern. 53
Figure 3-37–P1 in Case 1, p2 in Case 3: first placement of p1, first access pattern. 54
Figure 3-38–P1 in Case 1, p2 in Case 3: first placement of p1, second access pattern.... 54
Figure 3-39–P1 in Case 1, p2 in Case 3: second placement of p1..... 55
Figure 3-40–P1 in Case 1, p2 in Case 3: third placement of p1. 55
Figure 3-41–P1 in Case 1, p2 in Case 3: fourth placement of p1. 56
Figure 3-42–P1 in Case 2, p2 in Case 1: first placement of p1, first access pattern. 57
Figure 3-43–P1 in Case 2, p2 in Case 1: first placement of p1, second access pattern.... 57
Figure 3-44–P1 in Case 2, p2 in Case 1: first placement of p1, third access pattern. 58
Figure 3-45–P1 in Case 2, p2 in Case 1: first placement of p1, fourth access pattern. 58
Figure 3-46–P1 in Case 2 and p2 in Case 1: second placement of p1..... 59
Figure 3-47–P1 in Case 2, p2 in Case 2: first placement of p1, first access pattern. 60
Figure 3-48–P1 in Case 2, p2 in Case 2: first placement of p1, second access pattern.... 60
Figure 3-49–P1 in Case 2 and p2 in Case 2: second placement of p1..... 61
Figure 3-50–P1 in Case 2, p2 in Case 3: first placement of p1, first access pattern. 62
Figure 3-51–P1 in Case 2 and p2 in Case 3: first placement of p1, second access pattern. 62
Figure 3-52–P1 in Case 2 and p2 in Case 3: second placement of p1..... 63
Figure 3-53–P1 in Case 3, p2 in Case 1: first placement of p1, first access pattern. 64
Figure 3-54–P1 in Case 3 and p2 in Case 1: first placement of p1, second access pattern. 64
Figure 3-55–P1 in Case 3 and p2 in Case 1: first placement of p1, third access pattern.. 65
Figure 3-56–P1 in Case 3 and p2 in Case 1: first placement of p1, fourth access pattern. 65
Figure 3-57–P1 in Case 3 and p2 in Case 1: second placement of p1..... 66
Figure 3-58–P1 in Case 3, p2 in Case 2: first placement of p1, first access pattern. 67
Figure 3-59–P1 in Case 3, p2 in Case 2: first placement of p1, second access pattern.... 68
Figure 3-60–P1 in Case 3 and p2 in Case2: second placement of p1..... 68
Figure 3-61–P1 in Case 3 and p2 in Case 3: first placement of p1, first access pattern... 69
Figure 3-62–P1 in Case 3, p2 in Case 3: first placement of p1, second access pattern.... 70
Figure 3-63–P1 in Case 3 and p2 in Case 3: second placement of p1..... 70
Figure 3-64–P1 in Case 2 and p2 is searching down the tree. 71
Figure 3-65–P1 in Case 3 and p2 is searching down the tree. 72
Figure 4-1 –Original tree before concurrent insertion. 75
Figure 4-2–P1 inserts node A and starts backing up the tree. 76
Figure 4-3–P2 inserts node B and starts backing up the tree..... 77
Figure 4-4–Process p2 proceeds past p1 going up the tree..... 77
Figure 4-5–Sample tree of possible method 2. 78
Figure 5-1–Structure of a red-black tree node. 81
Figure 5-2–The function call tree. 82
Figure 5-3–Global Tree Node Declarations. 84
Figure 5-4–The algorithm CreateTree() 85
Figure 5-5–Dummy Nodes in an Empty Red-Black Tree 85
Figure 5-6–The algorithm Insert(key). 87

Figure 5-7–The algorithm – Traverse(KEY). 89
Figure 5-8–The algorithm – Placenode(newNode,insertPoint). 92
Figure 5-9–The algorithm Insert-Rebalance(x). 96
Figure 5-10 –The algorithm Update-Rotation(&x,&caseFlag). 100
Figure 5-11–The algorithm–Left-Rotation(z). 102
Figure 5-12–z is root and the left child of y is not empty. 102
Figure 5-13–z is a non-root node and the left child of y is not empty. 103

1 Introduction

This thesis addresses the development of a lock-free parallel insertion algorithm for a type of balanced binary search tree (BST) known as red-black trees.

1.1 Tree-Based Data Structures

Trees are one of the most common data structures and are used in many areas of computer science to represent hierarchical structures. For example, a binary tree is often used to represent arithmetic expressions in a compiler or interpreter. Trees also have many important applications in searching and indexing. Trees are generally useful for solving a wide variety of algorithmic problems.

Balanced trees support efficient searching by ensuring that the length of any root to leaf path is always sub-linear. Perhaps the best-known kind of tree satisfying this condition is an AVL tree. AVL trees are binary search trees that are kept very tightly balanced (the heights of the subtrees of a node differ by at most 1). They overcome the O(n) worst case performance of ordinary binary search trees, and can be shown to have O(log n) performance for all significant operations including searching for a node with a given key, inserting a new node, and deleting a node. Red-black trees are another type of balanced BST. They originate from R. Bayer's symmetric binary B-trees [1] and Henk J. Olivie's Half-balanced binary search trees [4]. They are somewhat *less* balanced than AVL trees. They are, however, still logarithmic in height.

1.2 Parallel Computing

A number of applications that scientists and engineers are currently developing require computational capabilities that are beyond the scope of current sequential computers. Examples of such problems include:

- Long-term weather prediction,
- Protein folding,
- Ocean circulation models,
- Quantum chromo-dynamics, and

- Models of human vision.

Existing sequential computers are too slow, and/or cannot meet the I/O requirements of such demanding applications. Computer designers cannot just keep making individual processors faster (as has been done), since certain physical limitations are now being reached beyond which it will not be possible to drive signals through circuits any faster. In response to these challenges, parallel machines that use multiple processors to allow more than one computation to be done at once have been developed. Such machines are now becoming increasingly prevalent (from dual processor desktop machines to common eight processor "departmental servers", as well as large-scale parallel machines having many more processors).

1.3 Synchronization in Parallel Programming

In shared memory parallel computer systems, data (stored in memory) are shared between several processors and therefore access to it must be controlled in some way. If synchronization is not provided and several processes try to access and modify a datum at the same time, the state of the datum can become inconsistent. Synchronization is used to avoid such consistency problems. We seek to ensure that concurrent operations occur in a valid order that preserves the correctness of all concurrently executing processes. Synchronization is also referred to as "concurrency control" and is commonly enforced either pessimistically (lock-based) or optimistically (lock-free). Efficient synchronization is important for achieving good performance in parallel programming.

The most common use of lock-based synchronization is ensuring mutually exclusive access to critical sections (regions of code that access shared data). Mutual exclusion refers to permitting just one processor have access to a particular memory location (or data item/structure) at a time. Locks provide a means for implementing mutual exclusion and avoiding data races[1]. A lock must

[1] Data races occur when unsynchronized threads or parallel processes access shared variables and at least one access is a write operation. Races result in non-deterministic results based on which process "wins the race" to access the shared data.

be acquired before accessing shared data and released once access is complete. Locks ensure that only one processor may access (and possibly modify) a shared data structure at any given time.

Locks correctly protect the critical sections of an application, but require the execution of many lock acquire and release operations when locks are used to guard frequently-accessed concurrent data structures such as queues, priority queues, lists, trees, etc. The chief advantage of using locks is that they are simple and easy to use. This is especially true for large and/or complex data structures (e.g. trees) where a single lock may guard the entire structure. Another good thing about locks is that they are well studied and understood. Unfortunately, the delay of a process that is in a critical section (and hence holding a lock) due to a page fault, multitasking pre-emption, etc. creates a bottleneck that can cause performance problems. Such a process may indirectly block the execution of other processes leading to undesirable effects.

An alternative to mutual exclusion, the use of lock-free data structures, has been proposed. Lock-free structures use no locks and consequently the time normally spent on acquiring and releasing locks is avoided. Further, undesirable scheduling effects (as just described) do not occur. Lock-free techniques are based on the use of non-blocking, low-level atomic synchronization primitives such as `compare & swap` (`CAS`) and `load-linked & store-conditional` (`LL/SC`). The primitive that has been most commonly used for developing lock-free algorithms is `CAS` (and variants of it) although lock-free algorithms may also be implemented with the `load-linked & store-conditional` instructions. Other advantages of using lock-free methods include freedom from deadlock, congestion, and starvation. In general, lock-free concurrency control provides less overhead, potentially greater concurrency, and the avoidance of certain undesirable scheduling anomalies.

Although a number of lock-free algorithms for shared data structures have been proposed, most of the work on efficient lock-free structures has focused on linked lists and queues. In particular, there are no complete lock-free concurrent algorithms for operations on balanced trees. This thesis explores the use of lock-

free concurrency control to support a concurrent implementation of insertion into red-black trees. Specifically, it will present an algorithm for doing concurrent insertions that ensures correctness through the use of `Compare and Swap` (`CAS`) type primitives. The basic idea behind the algorithm is to add flag variables to each node in the tree and use $\texttt{CAS}_n$ (`CAS` with `n` arguments) to guard regions of the tree where updates may be made by atomically setting and resetting those flags and modifying pointers internal to the data structure. By carefully using `CAS` operations in this way all concurrent processes (traversing the tree and inserting new nodes into it) will execute successfully without interfering with one another. The algorithm presented is lock-free, so the undesirable properties associated with the use of locks in parallel algorithms are avoided. The algorithm may, however, require processes to wait for one another[1]. This is necessary so that one process may not "pass" another along some path in the tree and consequently cause that process to function incorrectly. The likelihood of indefinite postponement in such situations, however, is unlikely in all but the smallest trees.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 reviews work related to this thesis. Chapter 3 begins by describing serial operations on red-black trees and then talks about the need to synchronize operations from multiple processes concurrently operating on a red-black tree. Chapter 4 discusses some possible strategies for synchronizing operations on red-black trees without the use of locks. Chapter 5 presents the design of concurrent insertion and traversal algorithms and includes pseudo-code and correctness arguments for them. Finally, Chapter 6 concludes the thesis and provides some directions for future research.

[1] So the algorithm is not, strictly speaking, wait-free.

2 Related Work

This chapter reviews work related to that presented in this thesis. It begins by discussing how synchronization is provided generally and then focuses on lock-free techniques.

2.1 Synchronization Primitives

Synchronization primitives are used to safely implement operations on concurrently-accessed data structures. By convention, a concurrent data structure/object implementation is said to be *lock-free* if it guarantees that some process will always complete its operation in a finite number of steps, and it is said to be *wait-free* if it guarantees that each process will complete its operation in a finite number of steps.

Synchronization primitives can be divided into two classes. As proved by Herlihy [6], primitives in the first class cannot be used to design non-blocking or wait-free implementation of such simple data structures such as trees, queues, and lists. These include the commonly available `read`, `write`, `test & set (TAS)`, and `fetch & op` primitives. The second class can be used to develop lock-free algorithms. These primitives include `Compare & Swap (CAS)`, `Double Compare & Swap (DCAS)`, and the `Load-Linked & Store-Conditional (LL/SC)` primitives. It is primitives in Herlihy's second category that are of interest in this thesis.

`CAS` is a three-operand *atomic* instruction of the form:

```
CAS(PRIVCOPY, NEWVAL, SHVBLE)
```

where `PRIVCOPY`, `NEWVAL`, and `SHVBLE` are single word variables. The algorithm for `CAS` is shown in Figure 2-1. `SHVBLE` is a shared variable and `PRIVCOPY` is a private copy of that variable made earlier by a process. `NEWVAL` is a value that the process is attempting to put into `SHVBLE`. The process is allowed to do so only if no other process has modified `SHVBLE` since the private copy (`PRIVCOPY`) of it was made. If `SHVBLE` has not changed since it was last

read, the new value (NEWVAL) is put into SHVBLE and TRUE is returned, otherwise CAS returns FALSE to indicate that it has failed.

```
CAS(PRIVCPY,NEWVAL,SHVBL)
Begin atomic
if (SHVBL == PRIVCPY) {
     SHVBL = NEWVAL;
     return(TRUE); }
else
     return(FALSE);
End atomic
```

Figure 2-1–CAS synchronization primitive.

A generic DCAS operation, shown in Figure 2-2, is of the form:

DCAS(PRIVCPY1,NEWVAL1,SHVBL1,PRIVCPY2,NEWVAL2,SHVBL2)

DCAS atomically updates locations SHVBL1 and SHVBL2 to values NEWVAL1 and NEWVAL2 respectively if and only if SHVBL1 still holds the value PRIVCPY1 and SHVBL2 still holds the value PRIVCPY2 when the operation is invoked. In this case it returns TRUE to indicate success. Otherwise it returns a FALSE value indicating that the operation failed.

```
DCAS(PRIVCPY1,NEWVAL1,SHVBL1,
     PRIVCPY2,NEWVAL2,SHVBL2)
Begin atomic
if ((SHVBL1==PRIVCOPY1) &&
     (SHVBL2 == PRIVCOPY2)) {
          SHVBL1 = NEWVAL1;
          SHVBL2 = NEWVAL2;
          return(TRUE);}
else
          return(FALSE);
End atomic
```

Figure 2-2–DCAS synchronization primitive.

Due to the complexity of concurrent insertion in trees, the CAS and DCAS primitives alone are insufficient to implement lock-free insertions. Thus, a family of CAS-like primitives is assumed (i.e. new primitives CAS3, CAS4, etc. that are like CAS and DCAS but which operate on sets of three, four, etc. operands). Such a generic version of CAS (CASn) would then be of the following form:

```
CASn(PRIVCPY1,NEWVAL1,SHVBL1,
```

```
PRIVCPY2,NEWVAL2,SHVBL2,
...
PRIVCPY_n,NEWVAL_n,SHVBL_n);
```

Code for CASn is shown in Figure 2-3. CASn is an n-word version of CAS that operates on n adjacent words in memory simultaneously. CASn atomically updates locations SHVBL1, SHVBL2, ... SHVBLn to the values NEWVAL1, NEWVAL2, ... NEWVALn (respectively) if SHVBL1 holds the value PRIVCPY1, SHVBL2 holds PRIVCPY2, etc. up to and including SHVBLn, which must hold PRIVCPYn when the operation is invoked. If the updates are successful, CASn returns TRUE. Otherwise it returns FALSE to indicate that the operation failed.

```
CASn(PRIVCPY1,NEWVAL1,SHVBL1,
     PRIVCPY2, NEWVAL2,SHVBL2,
     ...,
     PRIVCPYn,NEWVALn,SHVBLn)
Begin atomic
if ((SHVBL1==PRIVCPY1)&&
    (SHVBL2==PRIVCPY2)&&
         ...
    (SHVBLn==PRIVCPYn)){
        SHVBL1=NEWVAL1;
        SHVBL2=NEWVAL2;
               ...
        SHVBLn=NEWVALn;
        return(TRUE);}
else
         return(FALSE);
End atomic
```

Figure 2-3–Generic CASn synchronization primitive.

2.2 The ABA Problem

The ABA problem occurs when using CAS. It is possible that SHVBL will be the same as PRIVCPY (refer to Figure 2-1) even though SHVBL has been modified one or more times since the process made a copy of it. In such a situation, a CAS operation performed by the process will succeed even though the value of SHVBL has changed. This may be a problem when used in certain situations.

There are several ways to avoid this problem. One commonly used approach makes use of the double-word version of the Compare & Swap operation,

DCAS (shown in Figure 2-2) where SHVBL1 contains the value of the variable in question and SHVBL2 contains a version number. Whenever the variable is modified, the corresponding version number is incremented by one. If a process wants to update a variable, it must check the version number to determine that it has not been incremented. If the version number has been incremented, then some other process has updated the value of the variable (possibly to the same value) and the DCAS operation fails.

Figure 2-4 shows how the DCAS primitive can be used to avoid the ABA problem. In the figure, a process that wants to update node to contain a new data value (newVal) calls the function Node_Update. The node has two fields: the data and a version number. The local variables data and counter are used to preserve copies of the contents of node. When Node_Update executes the DCAS operation, it checks to see that the copies of the data and counter fields are still the same as the fields in node. If this is true, then no other process has changed node and the DCAS operation will update node and return TRUE. Otherwise, if either field of node (data or count) has been changed by another process, the DCAS operation fails and returns FALSE. Since the DCAS always increments the version number, the ABA problem is avoided. When an update fails, the caller of Node_Update must decide what action to take and must eventually call Node_Update again to retry the update with a new value.

```
Node_Update(node,newVal)
boolean  r;
datatype data;
int      counter;

data = node→data;
counter = node→counter;
r=DCAS(data,newVal,node→data,
       counter,counter+1,node→counter);
return r;
```

Figure 2-4–Avoiding the ABA problem using DCAS.

2.3 Existing Work on Lock-Free Structures

Previous work has explored lock-free and non-blocking implementations of several data structures. According to Herlihy, a "concurrent object[2] implementation" is said to be lock-free if *some* process must complete an operation after the system as a whole takes a finite number of steps, and it is said to be wait-free if *each* process must complete an operation after a finite number of steps. Non-blocking algorithms for concurrent data structures guarantee that a data structure is always accessible, and hence these algorithms are robust and fast. The term "lock-free" is commonly used interchangeably with "non-blocking".

Most of the previous work on lock-free structures has focused on linked list and queue structures. The main operations performed on linked lists and queues are inserting and deleting elements and searching for specific data items. Some of the algorithms proposed allow more than one process to concurrently modify the data structure while others allow only one process to modify the data structure at a time, in which case concurrency is limited to reading structure elements. The following is a brief description of some of the important lock-free algorithm designs for lists and queues.

2.3.1 Herlihy's Design

Herlihy [7] proposed a completely general methodology for transforming sequential code for any data structure into a concurrent non-blocking or wait-free implementation by using novel synchronization and memory management algorithms. These algorithms apply atomic `Read`, `Write`, `Load-Linked`, and `Store-Conditional` operations to a shared memory. In his transformation, the basic technique for performing operations on the data structure in question is by changing a pointer within the structure. The process first uses the `Load-Linked` instruction to obtain a copy of the relevant pointer before it tries to perform an operation. The process then creates a private copy of the data structure (or part thereof) referenced by the pointer that it may manipulate freely. Finally the process uses the `Store-Conditional` instruction to verify that the

[2] In this context an "object" is meant to be any shared data structure. It is not intended to imply any special significance with respect to object-oriented programming.

original pointer to the data structure has not been changed when it wants to replace the old pointer with the pointer to the updated private copy. If the pointer has changed, the replacement fails and the process starts the entire operation again. In general, the process often needs to copy the entire data structure so this method can be very expensive. In addition, only one process can modify the data structure at a given time. Therefore, concurrent updates are not allowed in this method. It would be desirable to create more specific techniques that address these limitations and this has been done for certain simple data structures.

2.3.2 Prakash's Design

Prakash [7] proposed a method of designing non-blocking algorithms for any concurrent data structure using queues as an example. In his design, the `Compare & Swap` synchronization primitive is used to maintain the consistency of the data structure. Many processes are allowed to concurrently access the data structure, but modifications to the data structure are made in a serial fashion. Thus, only one modification operation (by a single process) may be done on the data structure at a time. Further, because Prakash's technique makes the entire state information necessary to complete any operation globally available, when a slow process blocks a fast process, the fast process can help the slow one to finish its operation (using the global state information) so it can then proceed with its own operation.

Compared with Herlihy's method, the design Prakash presented is an improvement since it has much lower memory requirements. In Herlihy's method, the amount of memory required to ensure that a process can always make its needed copies is very large when there are many concurrent processes. Unlike Herlihy's method, with Prakash's design, the required changes are not made to the copies of the data structure but to the data structure itself, thus Prakash's method avoids the extra memory requirement. But to make n processes concurrent, the algorithms have a system latency of O(n). Moreover, in this paper, the author does not take into account the ABA problem (discussed in Section 2.2).

2.3.3 Valois' Design

Valois [11] used the single-word CAS (Compare & Swap) synchronization primitive to implement operations on a linked list. In his design, access to the list was accomplished via a "cursor". At any given time, the cursor was said to be visiting a ceratin item/node in the list. Every node in the linked list had an extra node, called the auxiliary node. The auxiliary nodes were added to the data structure to overcome problems associated with concurrent insertion and deletion operations. An auxiliary node is one that contains only a next field. The structure of an empty linked list included two dummy nodes as the first and last nodes separated by an auxiliary node. A cursor was implemented as a data structure containing three pointers: the first pointer was target which pointed to the list node at the position the cursor was visiting. The second pointer was pre-aux which pointed to the auxiliary node immediately preceding the target, and the third pointer was pre-node which pointed to the last visited normal node. The pre-node pointer was used only by the routine Valois_delete (described below). The structure of a linked list (and an example cursor) is shown in Figure 2-5.

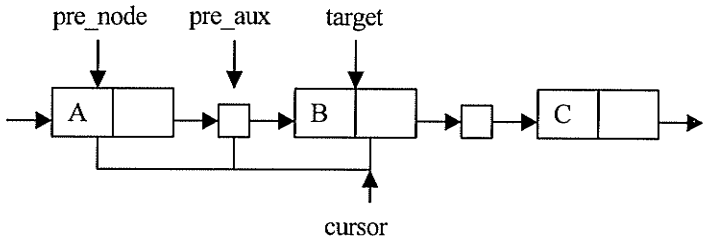

Figure 2-5–Structure of Valois linked list.

In Valois' algorithm for concurrently inserting new nodes, c is a cursor so c.target points to the node at the position the cursor is visiting and c.pre_aux points to the auxiliary node immediately preceding c.target. A new node inserted into the list includes the node as well as a new auxiliary node (the new auxiliary node follows the new node in the list) and insertion can only occur between an existing auxiliary node and a normal node in the list. A process implements the insertion operation using the CAS instruction. The CAS operation first checks if c.pre_aux still points to c.target. If the

pointer has not changed, then c.pre_aux is set to point to the new node (whose new auxiliary node already points to c.target) and TRUE is returned to the process. If the pointers have changed, a FALSE flag is returned to indicate that the cursor has become invalid (because the structure of the list has changed since the pointers in the cursor were last read). The process must then restart the operation from the beginning.

```
Valois_delete(c:cursor)returns boolean
d = c→target;
n = c→target→next;
r = CAS(c→pre_aux→next, n, d);
if (!r)
   return (FALSE);
d→back_link=c→pre_node;
p = c→pre_node;
while (p→back_link != NULL) {
   q = saferead(p→back_link);
   release(p);
   p = q; }
s = saferead(p→next);
while (n→next is not a normal node) {
   q = saferead(n→next);
   release(n);
   n = q; }
repeat
   r = CAS(p→next, s, n)
   if(!r) {
        release(s);
        s = saferead(p→next); }
until (r or (p→back_link != NULL) or
       (n→next is not a normal cell));
release(p);
release(s);
release(n);
return (TRUE);
```

Figure 2-6–Valois' deletion algorithm.

In Valois' algorithm [11] for concurrently deleting a node from the linked list (shown in Figure 2-6), a cursor c is passed as a parameter. If the target node in the cursor structure is deleted successfully, a TRUE flag will be returned to the process. If the list structure has changed since the pointers were last read, a FALSE flag will be returned to the process to indicate that the operation has failed. The process then has to re-read the pointers in the cursor and retry the

deletion operation. The SAFEREAD operation (used in Figure 2-6) is called to atomically read a pointer and increment the reference count in the item/node being pointed at.

Successive deletion operations may lead to a chain of adjacent auxiliary nodes. To remove the extra auxiliary nodes, each normal node of the linked list is provided with another link called back_link (shown in Figure 2-6). This back-link field is initially set to NULL. When a node is deleted from the list, back_link is set to point at pre_node. Using these back_links, the list can be traversed back to a non-auxiliary node that has not been deleted as illustrated in Figure 2-7. The entire chain of auxiliary nodes can then be removed from the list and replaced with a single auxiliary node. This has the effect of decreasing the overall length of the linked list and thereby speeding up traversals.

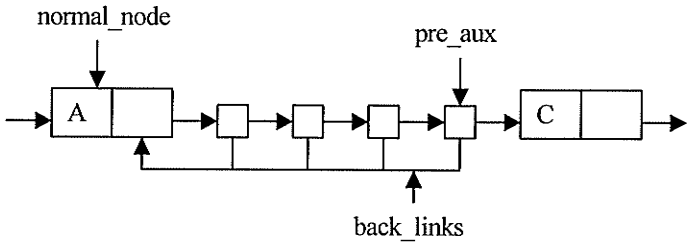

Figure 2-7–Deleting extra auxiliary nodes.

The main advantage of Valois' design is that it allows processes to concurrently traverse the linked list and insert or delete nodes. The other advantage is that it can also easily avoid the ABA problem by simply maintaining a count field (representing a version number) in each of the nodes. The major disadvantage of the design is the additional overhead (in both time and space) that arises because each node has an auxiliary node associated with it. The time overhead is due to the increase in latency of processes performing their operations because of the need to traverse the auxiliary nodes and the space overhead is because each node has an auxiliary node associated with it.

2.3.4 Greenwald and Cheriton's Design

In Greenwald and Cheriton's design [1], the DCAS synchronization primitive is used to implement concurrent operations on a linked list or other simple data

structures. A version number on the *entire* data structure is used to solve the ABA problem. Incrementing the version number means that a process has performed an operation on the structure (e.g. the linked list has been changed). Using this approach, Greenwald and Cheriton implemented a multiprocessor operating system kernel and run-time library providing high performance, reliability and modularity using `DCAS`, thereby showing the effectiveness of even a very simple lock-free design technique for OS implementation.

In their scheme, during the deletion of a node, a process searches down a linked list from the head node to the tail node to find the desired element (or detect the end of the list). If the node is found, the process makes sure that the version number has not changed since it was last read, and that the node has not been deleted. Then the node is deleted from the list and the version number of the entire list is incremented using `DCAS`. If the node is not found, the process checks if the version number has changed. If the version number has changed, the process tries the deletion operation again (since the node to be deleted may have just been inserted by another concurrent process); otherwise, it returns `NULL` to indicate that some other process has already deleted the node.

During the insertion of a node, a process traverses the linked list from the head node until it finds the node after which the new node is to be inserted. It also checks the version number to verify that the linked list has not been modified since the process last read from the list. If the version number has not changed then the insertion operation is done (and the version number incremented) using `DCAS`; otherwise, a `FALSE` value is returned to the calling process indicating that the linked list has changed and the process must restart the insertion operation from the beginning.

The main advantage of Greenwald and Cheriton's design is simplicity (and that it avoids the ABA problem by using version numbers). The major disadvantage of this method is that concurrency is limited to just reading, not writing/updating because each time the version number is incremented, other concurrent processes have to re-read their pointers, even though they may have been accessing completely different parts of the list.

2.3.5 Farook's Design

Compared with Greenwald and Cheriton's and Valois' algorithms, Farook's design [13] for concurrent linked lists provides improved performance. The essence of his algorithm is to adopt count fields and redundant pointers to overcome the problems that could occur during concurrent updates to the same region of the list.

The structure of the linked list in Farook's design includes two dummy nodes, `head` and `tail`. The head node points to the first node in the linked list while the tail node points to the last node in the linked list. Each node of the linked list consists of four fields: a data field, two pointer fields, and a counter field. One of the pointers is used to maintain an updated linked list so that the pointers do not point to any node that may have been deleted from the linked list or to ones that have been marked for deletion. The other pointer is used to allow concurrent processes to traverse the linked list even though some of the nodes in the list may have been marked for deletion. Each process that is manipulating a particular node (`target`) maintains three pointers, `prev`, `target`, and `next`. `Prev` points to the node preceding the `target` node and `next` points to the node immediately following the `target` node.

In the algorithm for concurrently deleting nodes from the linked list, the `DCAS` synchronization primitive is used. Deletion is accomplished using three routines: `CURSOR`, `TRY-DELETE` and `DELETE` (which calls `CURSOR` followed by `TRY_DELETE`). The `DELETE` algorithm locates the required target node that has to be deleted (by using `CURSOR`) and, on finding it, calls `TRY_DELETE`.

When the `DELETE` function calls the `CURSOR` function to determine if the required key exists in the list, there are three possible results: `CURSOR-AGAIN`, `FALSE`, and `TRUE`. A return of `FALSE` indicates that the required key was not found in the list. A return of `TRUE` indicates that the key exists and `TRY-DELETE` should be invoked to delete the node. A return of `CURSOR-AGAIN` indicates that the list was disturbed by another process during the traversal and forces the restart of the deletion process searching from the beginning of the list.

The TRY_DELETE routine tries to delete the target node. It uses DCAS to check if the pointers prev and target have changed. If the pointers have not changed, the pointer ("next node") field of prev is set to point to next, and the pointer field of the target node is set to NULL, indicating that the node has been deleted. TRUE is returned and the deletion is complete. If any of the pointers has changed, the DCAS operation returns FALSE and there are then three cases to consider. If the next node has changed, the process reads the new next node to make the pointer field of the target node point to the new next node and returns a status flag of "DELETE-AGAIN" to the DELETE routine indicating that the three pointers prev, target, and next have been updated and the TRY_DELETE function has to be invoked again to delete the node. If the prev pointer has changed, a FALSE value is returned by DCAS. A "CURSOR-AGAIN" status is then returned to the DELETE function, which restarts the entire operation from the beginning. If the target node itself has changed, a flag of "FALSE" is returned.

The algorithm for concurrently inserting nodes in the linked list also consists of multiple parts, namely: TRY-INSERT and INSERT. The INSERT function first searches for the node (target) before which the new node has to be inserted. A FALSE value is returned if the desired node is not found. Otherwise, a TRUE flag is returned along with three-pointers (prev, target, and next) and the INSERT function invokes TRY-INSERT to insert the new node before the target node. The INSERT function uses the CAS operation to check if the prev pointer still points to target. If it does, then prev's pointer field is set to point to the new node (which already points at target). If prev has changed, the process updates its prev pointer to point at the target node and a RETRY flag is returned to the INSERT algorithm, which calls TRY-INSERT again.

Farook's technique achieves better performance than both Valois' and Greenwald and Cheriton's algorithms. The reasons for this performance gain are that the traversal time is reduced since only n nodes are required in a linked list to represent n nodes and because Farook's algorithms allow concurrent read and write operations by processes. Another advantage of Farook's technique is that it

requires less time to delete nodes and maintain consistency of the linked list than does Valois' algorithm (because Valois' algorithm requires at least 2n nodes in a linked list to represent n nodes while Farook's algorithm only requires n+2 nodes).

3 Problem Description

3.1 Red-Black Tree Definition

A red-black tree is a form of binary search tree in which each node is assigned a colour: either red or black. We call a binary search tree a red-black tree if it satisfies the following five properties:

P1. Every node is coloured either red or black;

P2. Every leaf is a `NIL` node and is coloured black;

P3. If a node is red, then both its children are black;

P4. Every simple path from a node to any of its descendant leaves contains the same number of black nodes.

P5. The root is black.

The number of black nodes on a path from a node to a leaf is known as the black-height of the node, and the number of black nodes on a path from the root to a leaf is called the black-height of the tree. The properties listed above guarantee that any path from the root to a leaf is no more than twice as long as any other and, thus, that the tree is approximately balanced.

3.2 Operations on Red-Black Trees

The operations on red-black trees include traversal, insertion and deletion. All operations on the tree must maintain the four red-black properties by using special rotation operations. Before introducing these operations, the structure of each node in a red-black tree is described.

3.2.1 Red-Black Tree Node Structure

Each node of a red-black tree contains the fields: `colour`, `key`, `left`, `right`, and `P` (the parent pointer). The `colour` field requires just one bit. If this bit is on, we say that the node is red. If it is off, we say the node is black. If a child or the parent of a node does not exist, the corresponding pointer field of the node is defined to have the value `NIL`. These non-existent child nodes are the external nodes (normally, the leaves) of the binary search tree. Other normal (key-bearing)

nodes are the internal nodes of the tree. In implementations of the red-black tree algorithms, a sentinel is commonly used to represent NIL[5]. This sentinel simplifies certain boundary conditions in the code. In a red-black tree, the sentinel nil[T] has the same fields as an ordinary node in the tree, its colour field is black, and its other fields (P, left, right, and key) can be set to arbitrary values (though the pointer fields normally point to itself). In the red-black tree, all NIL pointers are replaced by pointers to the sentinel node: nil[T]. To save space, a single sentinel nil[T] is normally used to represent all the NIL.

Note that in the following diagrams, the black sentinel nodes have been omitted to minimize the size and complexity of the diagrams.

3.2.2 Rotations

Insertions and deletions on red-black trees may destroy the red-black properties. To restore the properties, the colours of some of the nodes in the tree may need to be changed and also the pointer structure may need to change. A rotation is used to change the pointer structure when a structure change is required.

```
LEFT-ROTATE(T, x)

y = x→right;
x→right = y→left;
if (y→left != NIL)
   y→left→parent = x;
if (y != NIL)
   y→parent = x→parent;
if (x→parent == NIL)
   T→root = y;
else if (x == x→parent→left)
   x→parent→left = y;
else
   x→parent→right = y;
y→left = x;
if(x != NIL)
   x→parent = y;
```

Figure 3-1–Left rotation.

A rotation is a local operation in a search tree that preserves the search tree order of the nodes. There are two possible rotations: left rotation and right

rotation, for which pseudo code is shown in Figure 3-1 and Figure 3-2, respectively. When a left rotation on a node x is performed, we necessarily assume that its right child y is non-NIL, and similarly, when a right rotation is done on a node, we assume that its left child is non-NIL.

There are four cases (described below) where rotations are necessary when doing insertions on red-black trees. In all cases, illustrated in the figures below, the colour of the current node (which resides under a red node) has been set to red thereby violating property P3 and a rotation is required to restore Property P3.

```
RIGHT-ROTATE (T, x)
y = x→left;
x→left = y→right;
if (y→right != NIL)
   y→right→parent = x;
if(y != NIL)
   y→parent = x→parent;
if (x→parent == NIL)
   T→root = y;
else if (x == x→parent→left)
   x→parent→left = y;
else
   x→parent→right = y;
y→right = x;
if (x != NIL)
   x→parent = y;
```

Figure 3-2–Right rotation.

In the first case (shown in Figure 3-3), a red node is the right child of another red node, which is the right child of another node. If the uncle is black, this requires a left rotation.

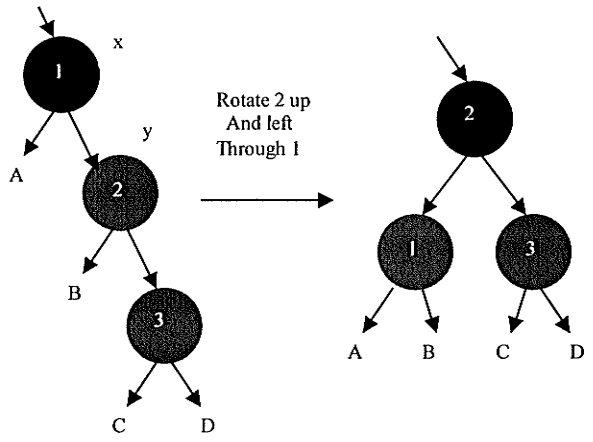

Figure 3-3–Rotation when the links pointing to red nodes are both to the right.

In the second case (Figure 3-4) the links pointing to the red nodes are first a right link, then beneath that a left link. To restore property P3, if the uncle is black, we must first perform a right rotation, and then perform a left rotation (i.e. a double rotation).

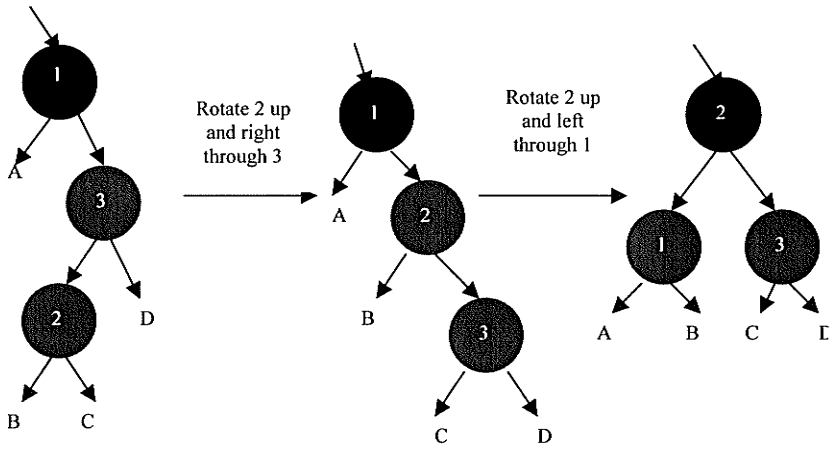

Figure 3-4–Rotation when the links pointing to red nodes are right then left.

In the third case (Figure 3-5), one red node is a left child of the other, which is itself a left child of some other node. If the uncle is black, this situation requires a right rotation. This case is symmetric with the first case.

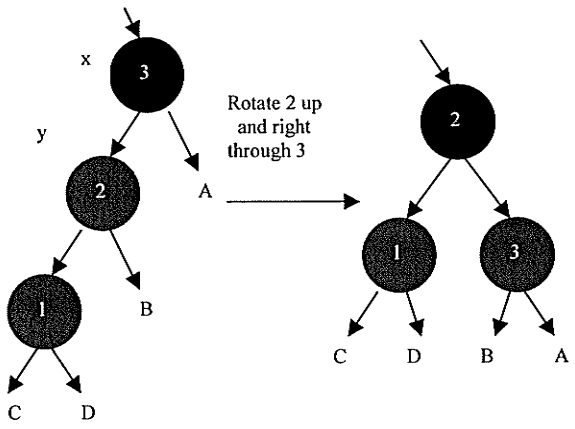

Figure 3-5–Rotation when the links pointing to red nodes is both to the left.

In the fourth and final case (Figure 3-6), the links pointing to the red nodes are first a left link, then beneath that a right link. To restore property P3, if the uncle is black, we must first perform a left rotation, and then perform a right rotation (i.e. another double rotation). This case is symmetric with the second case.

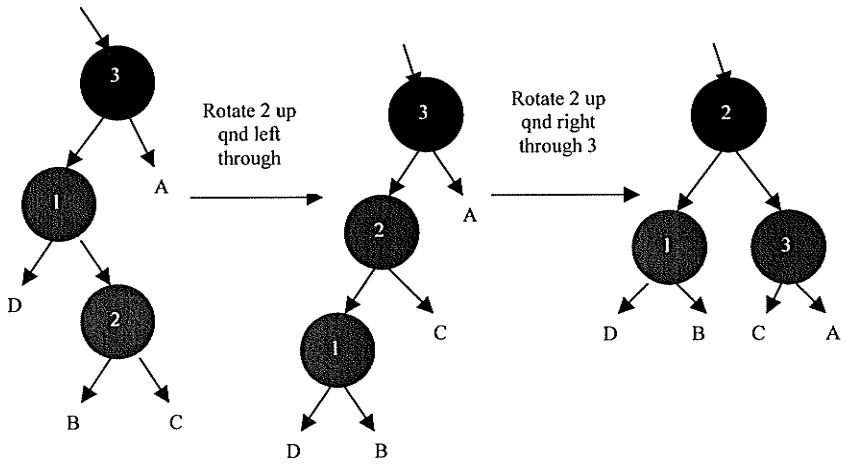

Figure 3-6–Rotation when the links pointing to red nodes are left then right.

3.2.3 Tree Traversal

A traversal of a red-black tree (see Figure 3-7) begins at the root of the tree. If the current node is `NIL`, a `STATUS_KEY_NOT_FOUND` flag is returned to indicate that a node containing the desired key value is not in the tree. Otherwise, the given key is compared with the current node's key. If the given key equals the

current node's key, a STATUS_FOUND flag is returned to indicate that the node containing the given key has been found. If the given key is less than the current node's key, the current node is set to the current node's left child and the process repeats. If the given key is greater than the current node's key, the current node is set to the current node's right child and the process repeats.

```
Traverse(T, key);
current = root[T];
while (current != NIL){
   if(current→key == key)
      return STATUS_FOUND;
   else if(current→key < key)
           current = current→left;
        else
           current = current→right;
}
return STATUS_KEY_NOT_FOUND;
```

Figure 3-7–Red-black tree traversal.

3.2.4 Insertions

As with ordinary search trees, every insertion takes place at a leaf. The first step is to find the position in the tree where the new key will be inserted and then add a newly created node with the new key at that point. The search for the insertion point starts at the root of the tree. For each node visited, the new key is compared with the current node's key and if the new key is less than the current node's key, the current node is set to be the current node's left child. If the new key is greater than the current node's key, the current node is set to be the current node's right child (i.e. the traversal follows the tree branches left and right as required.). If the current node is NIL, a leaf has been reached and a node containing the new key is inserted at that position.

After its insertion, the new node, x, is coloured red. To guarantee that the red-black tree properties are preserved, it may be necessary to "fix up" the resulting tree by re-colouring some of the nodes and then possibly performing one of the rotations described previously. This processing is accomplished by the routine InsertFixup (refer to Figure 3-9). The overall insertion process, including any necessary call to InsertFixup, is shown in Figure 3-8.

When a new red node is inserted into a red-black tree, the only rule that might be immediately violated is property P3 ("If a node is red, both its children are black"). This violation will occur only if the new node's parent is also red. Because the new node replaces a (black) `NIL` node, and it is red with `NIL` children, properties P1, P2, and P4 continue to hold. The `while` loop in the `InsertFixup` algorithm (Figure 3-9) has the job of moving the one violation of property P3 up the tree without violating property P4 ("Every simple path from a node to any descendant leaf contains the same number of black nodes"). There are three main cases to consider (there are six, but three are symmetric to the others, depending on whether the new node is a left child or a right child of its parent). The three base cases are described below:

```
RB-Insert(T,x)
current = root;                                          [1]
parent = NIL;                                            [2]
while (current != NIL) {                                 [3]
   if (current→key == key)                               [4]
      return STATUS_DUPLICATE_KEY;                       [5]
   parent = current;                                     [6]
   if (current→key < key)                                [7]
      current = current→left;                            [8]
   else                                                  [9]
      current = current→right;                           [10]
}                                                        [11]
x→parent = parent;                                       [12]
x→left = NIL;                                            [13]
x→right = NIL;                                           [14]
x→colour = RED;                                          [15]
x→key = key;                                             [16]
if (parent!= NIL)                                        [17]
   if(parent→key == key)                                 [18]
       parent→left = x;                                  [19]
   else                                                  [20]
      parent→right = x;                                  [21]
else                                                     [22]
   root = x;                                             [23]
InsertFixup(x);                                          [24]
return STATUS_OK;                                        [25]
```

Figure 3-8–Algorithm RB-Insert: red-black tree insertion.

Case 1 If the new node's parent and uncle are both red, then the colour of the parent and the uncle are changed to black, and the colour of the grandparent is set to red, thereby maintaining property P4. The only problem that might arise is that the grandparent might have a red parent (i.e. two red nodes in a row which again violates property P3). Thus the while loop repeats with the grandparent of the node.

Case 2 If the new node's parent is red and its uncle is black and it is a right child of its parent and its parent is a left child, then a left rotation about the parent (which preserves property P4) is performed, and then everything proceeds as in Case 3.

Case 3 If the new node's parent is red and its uncle is black and assuming the new node is a left child of its parent and its parent is a left child, then the colour of its parent is changed to black, the colour of its grandparent is changed to red, and the tree is rotated right about the node's parent which also preserves property P4. Because property P3 is satisfied, there are no longer two red nodes in a row, so the while loop can then terminate.

```
InsertFixup(T,x)
x→colour = RED;                                          [1]
while (x→parent→colour == RED)) {                        [2]
   if (x→parent == x→parent→parent→left) {               [3]
       y = x→parent→parent→right;                        [4]
       if (y→colour == RED) {                            [5]
          x→parent→colour = BLACK;                       [6]
          y→colour = BLACK;                              [7]
          x→parent→parent→colour = RED;                  [8]
          x = x→parent→parent;                           [9]
       } else if (x == x→parent→right) {                 [10]
          x = x→parent;                                  [11]
          LEFT-ROTATE(T,x);                              [12]
          x→parent→colour = BLACK;                       [13]
          x→parent→parent→colour = RED;                  [14]
          RIGHT-ROTATE(T,x→parent→parent);               [15]
       }                                                 [16]
   } else {/* same as then clause at line [3] */        [17]
          /* with "right" and "left" exchanged */       [18]
```

```
    }                                                    [19]
  } // end while at line [2]                            [20]
  T→root→colour = BLACK;                                 [21]
```

Figure 3-9–Algorithm InsertFixup: rebalance tree after insertion.

The algorithm also guarantees that the root of the tree is black. In all three cases, the new node's grandparent is black, since its parent is red, and property P3 can only be violated between the new node and its parent.

To clarify the insertion algorithm, we now consider two example red-black tree insertions. For the first (simple) example, the original red-black tree is shown in Figure 3-10 (the digit in each node in the tree represents the key of the node). In the example, a new node whose key is 1 will be inserted.

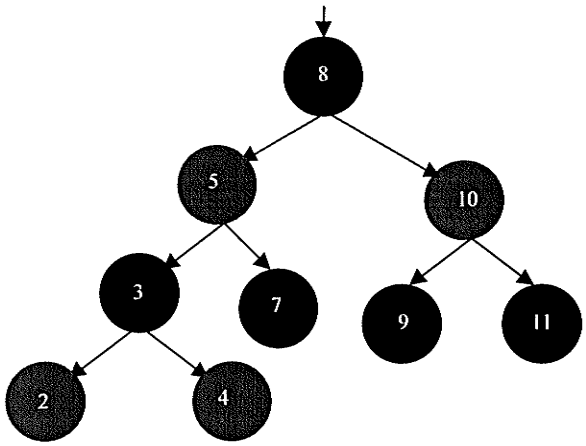

Figure 3-10–Initial red-black tree (insertion example 1).

The new red node, `x`, (containing the key 1) is inserted as the left child of the leaf node 2. Because node 2 is red and its parent and uncle are red, this situation is case 1 (described previously).[3]

The state of the tree after adding the new red node (containing the key 1) is shown in Figure 3-11. The steps necessary to ensure that the red-black properties hold after the insertion of the new node are now presented.

[3] Only the case where x is the left child of its parent is discussed. The case for a right child is symmetric.

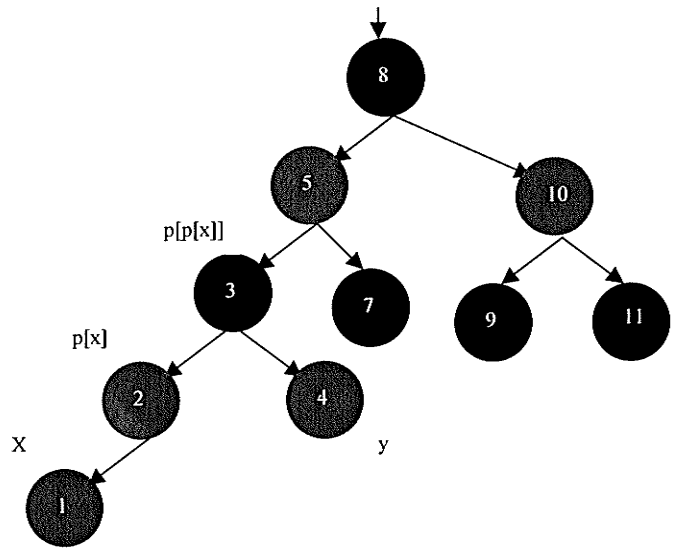

Figure 3-11–State after adding node x (insertion example 1).

Step 1: Because the parent and uncle of x are red, its parent and uncle are re-coloured black and its grandparent is re-coloured red, and `x` is set to point to its grandparent (node 3 in the example). After these re-colourings, the tree is as shown in Figure 3-12.

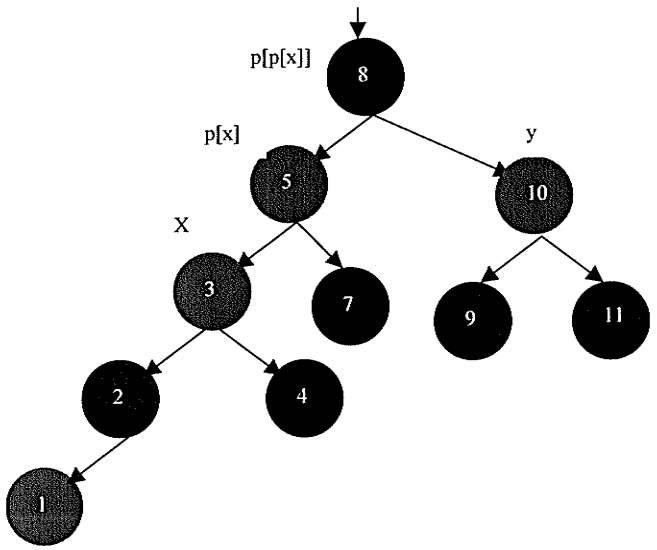

Figure 3-12–The tree after initial re-colouring (insertion example 1).

Step 2: Because `x`, its parent and uncle are red, the re-colourings must be repeated with `x`'s grandparent again becoming the new `x` as shown in Figure 3-13. This repetition is implemented using the `while` loop in Figure 3-9.

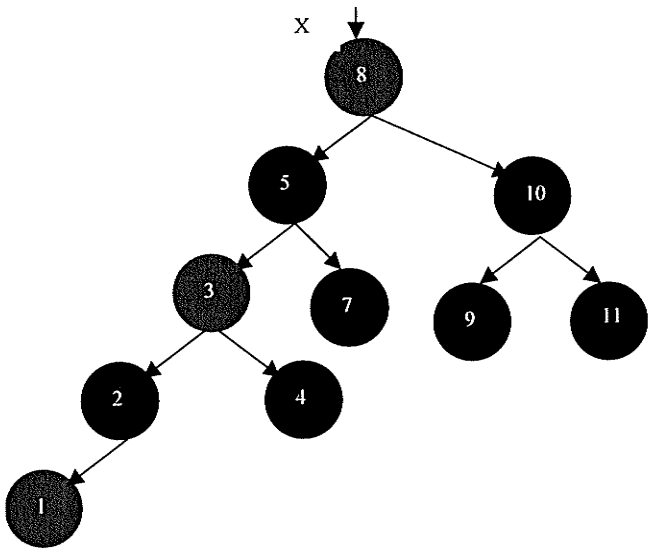

Figure 3-13–The tree after second re-colouring (insertion example 1).

Step 3: x now points to the root so the while loop terminates and the root is coloured black. The resulting tree is shown in Figure 3-14. Note that all the red-black properties once again hold (in particular, the number of black nodes along any path from the root to a leaf is the same).

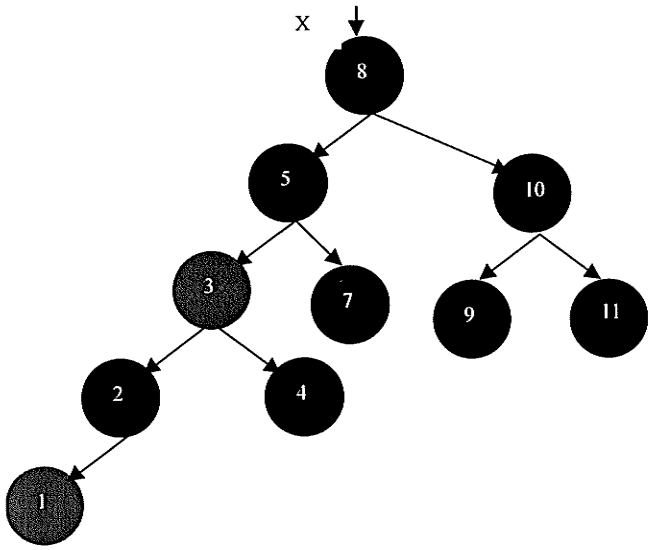

Figure 3-14–The tree after all re-colourings (insertion example 1).

In the second (more complicated) red-black tree insertion example, a new node (whose key is 5) is inserted into the tree shown in Figure 3-15.

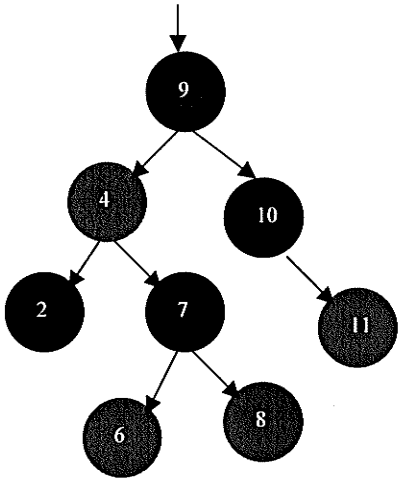

Figure 3-15–Initial red-black tree (insertion example 2).

The new red node, `x`, is inserted as the left child of node 6. Because both node 6 and `x` are red and so is the uncle 8, we again have Case 1. The state of the tree after the new node (with key 5) is added is shown in Figure 3-16.

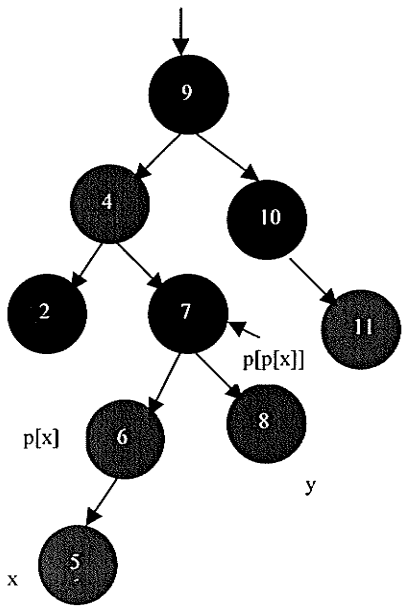

Figure 3-16–State after adding node x (insertion example 2).

Step 1: Because the parent and uncle of x are red, its parent and uncle are re-coloured black and its grandparent is re-coloured red, and `x` is set to point to its grandparent (node 7 in the example). The resulting tree is shown in Figure 3-17.

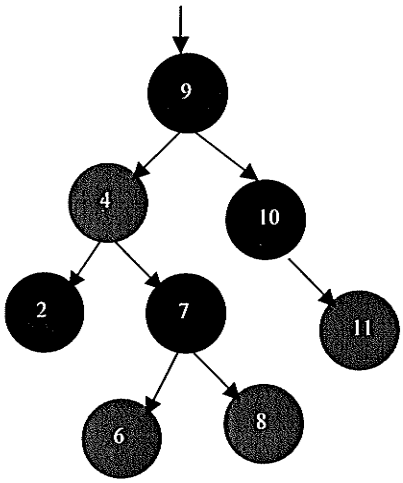

Figure 3-15–Initial red-black tree (insertion example 2).

The new red node, x, is inserted as the left child of node 6. Because both node 6 and x are red and so is the uncle 8, we again have Case 1. The state of the tree after the new node (with key 5) is added is shown in Figure 3-16.

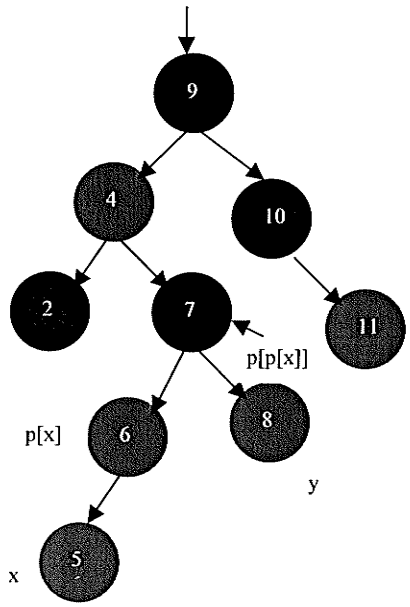

Figure 3-16–State after adding node x (insertion example 2).

Step 1: Because the parent and uncle of x are red, its parent and uncle are re-coloured black and its grandparent is re-coloured red, and x is set to point to its grandparent (node 7 in the example). The resulting tree is shown in Figure 3-17.

Because the uncle (node 10) of x is black and because x is the right child of its parent, this is Case 2 in algorithm InsertFixup (Figure 3-9).

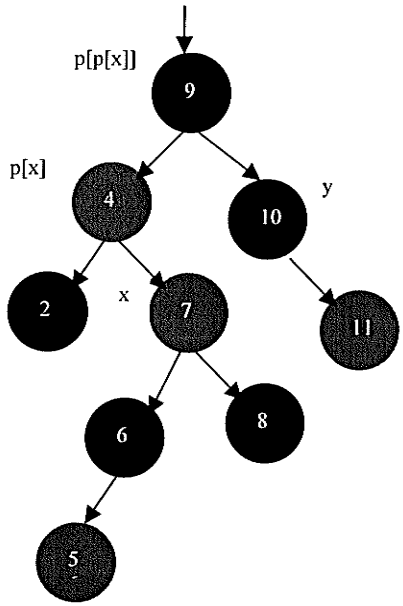

Figure 3-17–The tree after re-colouring (insertion example 2).

Step 2: A left rotation must be done on the parent of x, which results in the tree shown in Figure 3-18. This is Case 3 in algorithm InsertFixup (Figure 3-9).

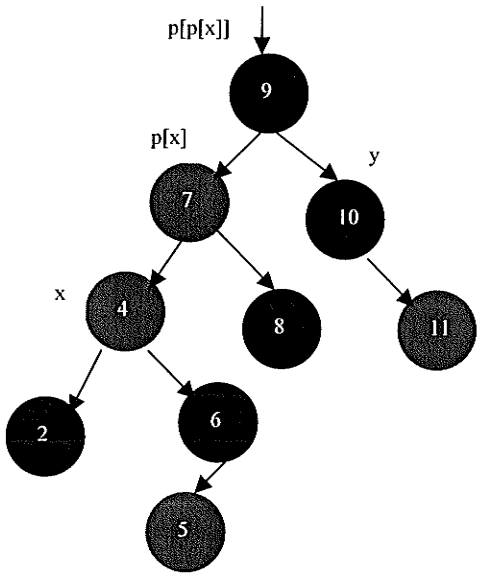

Figure 3-18–The tree after the left rotate (insertion example 2).

Step 3: The parent of `x` is coloured black, its grandparent is coloured red, and a right rotation must be done on the grandparent of `x`. Since the parent of `x` is black, the `while` loop now terminates. Finally, the root is coloured black and the resulting red-black tree is shown in Figure 3-19.

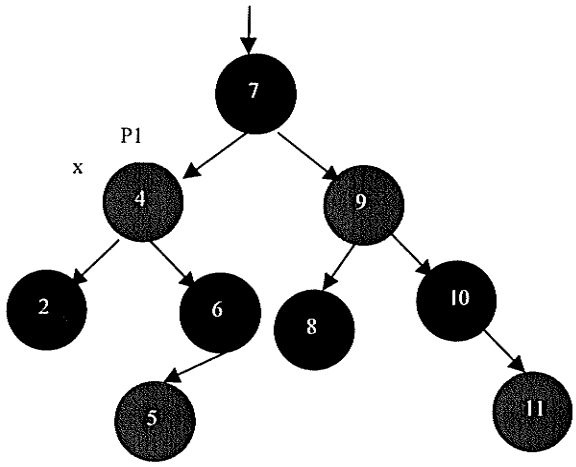

Figure 3-19–The tree after the right rotate, etc. (insertion example 2).

3.2.5 Deletions

Red-black deletion begins with an ordinary deletion using the "lefty-method" as is done in any ordered search tree. Using the lefty-method, if the node to be deleted has two children, then the node that will replace the deleted node will be the *left*-most node in the right subtree of the node to be deleted. Of course, the consequences of such a deletion on the red-black properties also has to be considered. If the node deleted is black, after the node is deleted, the number of black nodes on the path that previously included the deleted node will have decreased by one; hence, any ancestor node of x violates property 4. To restore the red-black tree properties, the tree must be rebalanced somehow. In the serial algorithm described by Cormen, Leiserson and Rivest [5], the function `RB-DELETE-FIXUP(T,x)` is used to re-colour the tree after the deletion of a black node. The algorithm considers four cases and operates in a fashion similar to insertions. In each case, a process operates in a local area of the tree on those nodes that need to be re-coloured and/or rotated. Once colour-updates and/or rotations are done in that area of the tree, the process moves up the tree until all the necessary re-

colouring is done or the root of the tree is reached. Since the focus of this thesis is on concurrent *insertions* into red-black trees, the deletion of nodes in red-black tree will not be discussed further.

3.3 Potential Problems with Concurrent Insertions

This thesis is concerned with supporting lock-free *concurrent* insertions into red-black trees. Therefore the thesis now focuses on identifying the problems that may occur when concurrent insertions are attempted in red-black trees. These problems will be used to determine the characteristics of a suitable lock-free insertion algorithm. For simplicity, only two concurrent processes are considered.

3.3.1 Case Recognition

There are a number of specific cases that can be identified that must be dealt with to allow concurrent insertions. Also, there are certain constraints on what is possible which help to identify the specific cases. These constraints are now considered and an attempt is made to identify the key cases that must be handled.

3.3.1.1 Shape Constraints on the Subtree in Each Case

There are constraints on the *shape* of the part of a red-black tree (i.e. the subtree, the neighbourhood includes x, p[x], p[p[x]] and uncle) that may be involved in some of the three insertion cases (as described in Section 3.2.4).

Case 1: There are no constraints on subtree shape when the insertion algorithm is executing code to handle Case 1 (i.e. moving up the tree after an insertion).

Case 2: According to Lines 3 and 10 in the serial algorithm (Figure 3-9), the shape of the subtree in Case 2 (i.e. the first of two rotations) must be a "zigzag" as illustrated in Figure 3-20.

Case 3: According to Lines 3 and 13 in the serial algorithm, the shape of the subtree in Case 3 (i.e. the last rotation) must be "straight" as illustrated in Figure 3-21.

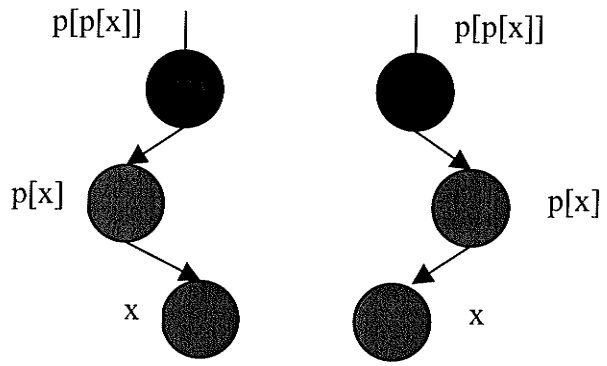

Figure 3-20–The "zigzag" shape required for Case2.

3.3.1.2 Color Constraints on the Subtree in Each Case

There are also constraints on the possible colours of the nodes in a subtree that may be involved in each of the three insertion cases.

Case 1: Due to Lines 1, 2 and 5 in the serial algorithm, the node x must be red, its parent must be red and its uncle must also be red.

Case 2 and Case 3: Due to Lines 1, 2 and 10 in the serial algorithm, the node x must be red, its parent must be red and its uncle must be black.

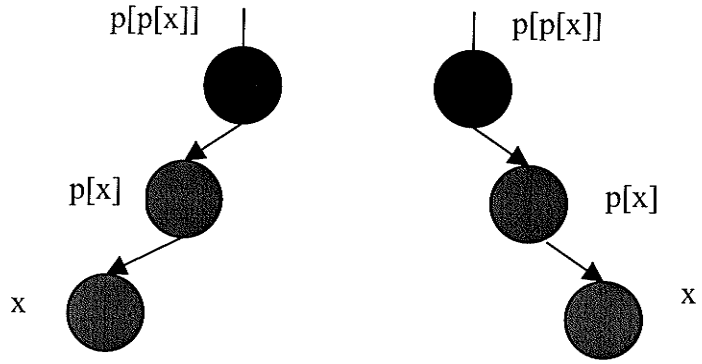

Figure 3-21–The "straight" shape required for Case3.

3.3.1.3 Access Constraints in Each Case

Finally, there are also constraints on the nodes in a subtree that may be accessed during each of the three insertion cases based on the processing that they will perform. These constraints, as will be seen later, will be the most important ones considered in the concurrent insertion algorithm presented in this thesis.

Case 1: According to the serial algorithm, a process in Case 1 may access the tree nodes: {x, p[x], p[p[x]], sibling[p[x]][4]}.

Case 2: According to the serial algorithm, a process in Case 2 (which performs the first single rotation), operates on the tree nodes: {x, p[x], p[p[x]], left[x]} for left-rotation operations and on the nodes: {x, p[x], p[p[x]], right[x]} for right-rotation operations.

Case 3: According to the serial algorithm, a process in Case 3 operates on the nodes: {p[x], p[p[x]], p[p[p[x]]], left[p[x]]} for left-rotation operations and on the nodes: {p[x], p[p[x]], p[p[p[x]]], right[p[x]]} for right-rotation operations.

3.3.2 Case Analysis

Being aware of the constraints on the various operations, it is now possible to consider the individual cases that may arise when two processes attempt concurrent insertions into a red-black tree.

In the following discussion, it is assumed that a second process, p2, is beneath a first process, p1, in the tree. This assumption is made without loss of generality since, of course, the case where p1 is beneath p2 is symmetric. In the following description, the phrase "a process is at a node" simply means that "a process' x variable is pointing to that node".

The diagrams presented below can be referred to later to confirm the correctness of the presented concurrent red-black tree insertion algorithm. Note that the diagrams illustrate worst-case scenarios not all of which may be possible in all situations due to specific patterns of concurrency.

3.3.2.1 Symbol Description

In the following figures and text, '√' indicates that the second process p2 cannot be at the node so marked due to shape constraints on p2. An 'x' indicates that if p2 is at the node so marked, it is "out of range" (i.e. will not access any nodes in common with p1) and therefore cannot affect the nodes accessed by p1. An '*' indicates that p2 cannot be located at the node so marked due to colour

[4] From this point forward 'sibling[x]' will be abbreviated as 's[x]'.

constraints on p2. Finally, a 'c' indicates that there will be a conflict between process p1 and p2 if p2's x is at the node marked 'c'. Further, in the following figures, if p2 is at a descendant node we have not drawn, there is no conflict between p1 and p2 because the local area of p1 does not overlap the local area of p2.

Finally, in what follows, 'U' stands for 'up', 'D' stands for 'down', 'R' stands for 'right', and 'L' stands for 'left'.

3.3.2.2 The Possible Cases

Each of the two processes may be in any of the three possible insertion cases, so there are eleven possible cases to consider. (If p1 is in Case 1, then p2 may be in Case 1, Case 2, or Case 3. Similarly, there are three options for p2 when p1 is in Case 2 and when it is in Case 3, and when p1 is in Case 2 and Case 3, p2 is searching the tree). We now consider the nine cases individually.

3.3.2.3 Process P1 is in Case1

Given that the first process (p1) is in Case 1 (i.e. it is moving back up the tree re-colouring nodes as necessary but is not doing any sort of rotation) what happens when the second process (p2) is in each of the three possible cases must be considered.

When Process P2 is also in Case 1

Since p1 is in Case 1, there are no shape constraints on the subtree where p1 operates. Due to the colour constraints described earlier, p1's x must be red, its parent must be red and its uncle must also be red. Since p1 is in Case 1, it will operate on the node set {x, p[x], p[p[x]], s[p[x]]} (as determined by its access constraints). If process p2 is located at any of the positions marked 'c' in the following figures, there will be a conflict between p1 and p2 because at least one node is in the node sets of both p1 and p2. Another way to view this situation is that when a process is in Case 1, it will exhibit one of four possible "access patterns" [5], UR-UR-DR, UL-UR-DR, UR-UL-DL or UL-UL-DL corresponding to its access constraints and the placement of its x in the tree. There

[5] An "access pattern" simply describes the sequence in which nodes will be accessed, relative to node x. This is illustrated in Figure 3-22 where the pattern is UR-UR-DR.

are four different placements of p1's x that must be considered. The first possible placement of p1's x is on the left "edge" of the tree. Figure 3-22, Figure 3-23, Figure 3-24 and Figure 3-25 illustrate this placement and all four possible access patterns of p2 (which is also in Case 1). Figure 3-26, Figure 3-27 and Figure 3-28 show the other possible placements when p1 is in Case 1 (p1's x is an internal left child, p1's x is on the left edge of the tree, and p2's x is an internal right child, respectively). Only a single figure is provided for each of these placements since the possible access patterns for p2 are, of course, unchanged.

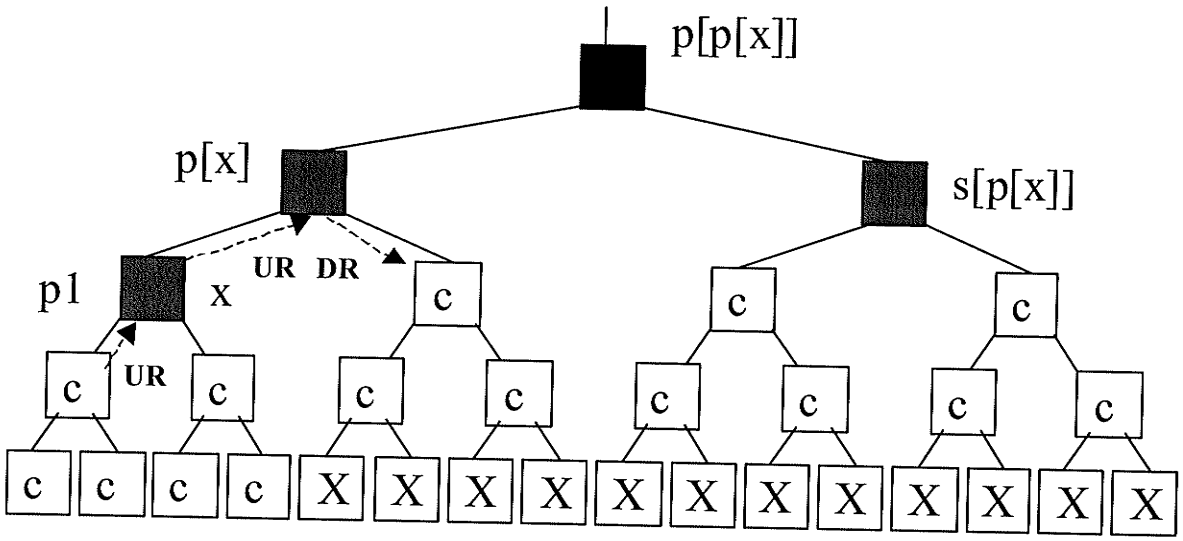

Figure 3-22–P1 and p2 both in Case 1: first placement of p1, first access pattern.

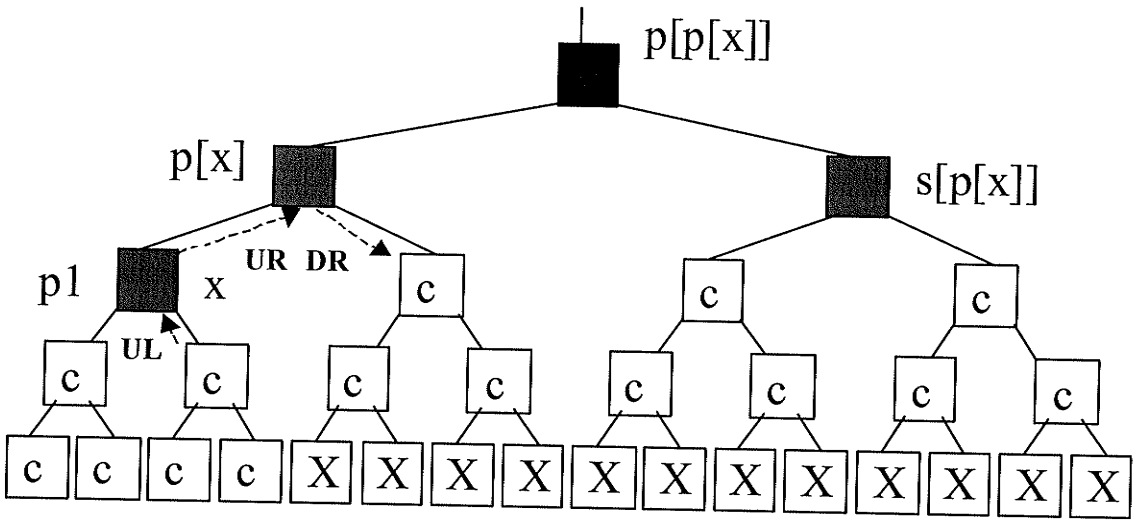

Figure 3-23–P1 and p2 both in Case 1: first placement of p1, second access pattern.

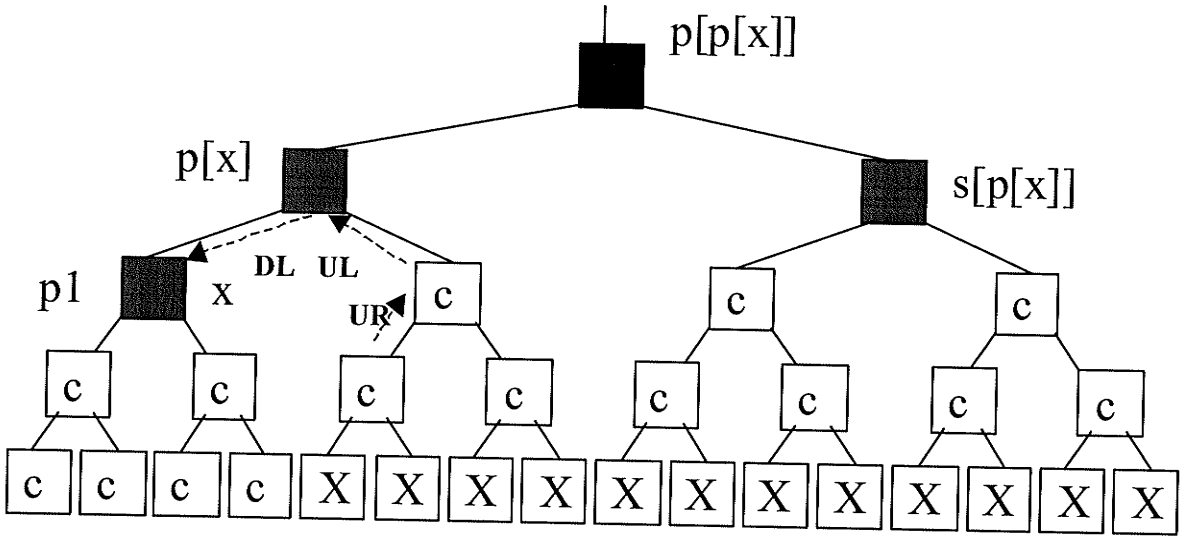

Figure 3-24–P1 and p2 both in Case 1: first placement of p1, third access pattern.

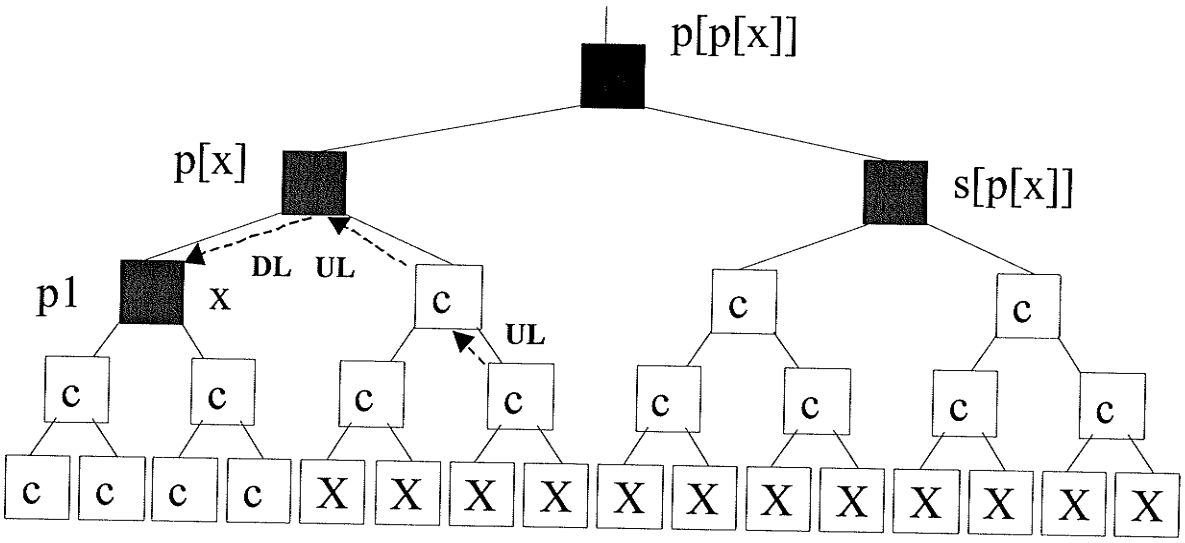

Figure 3-25–P1 and p2 both in Case 1: first placement of p1, fourth access pattern.

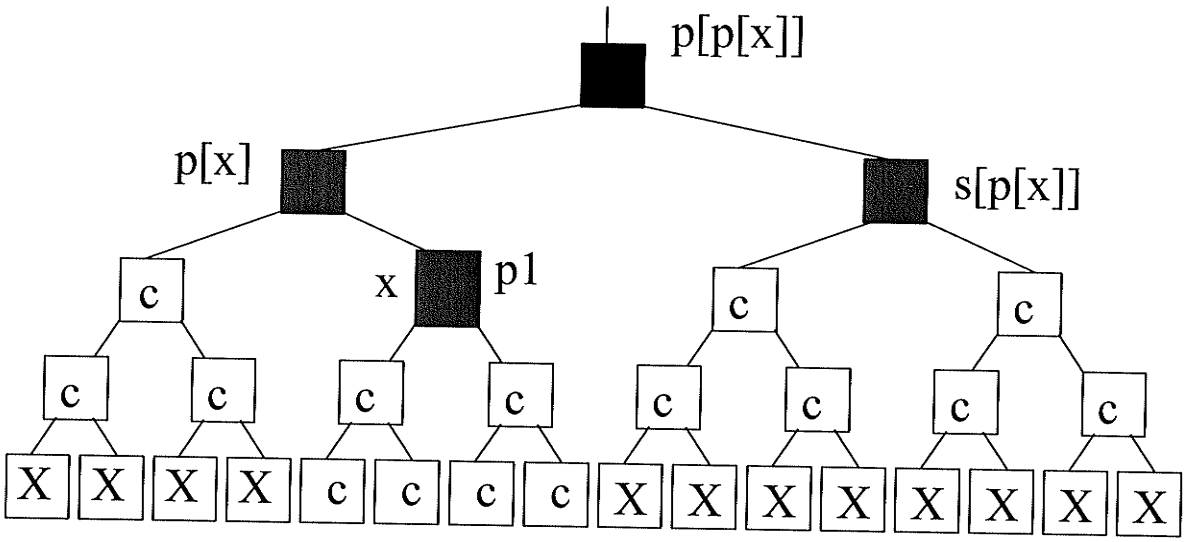

Figure 3-26–P1 and p2 both in Case 1: second placement of p1.

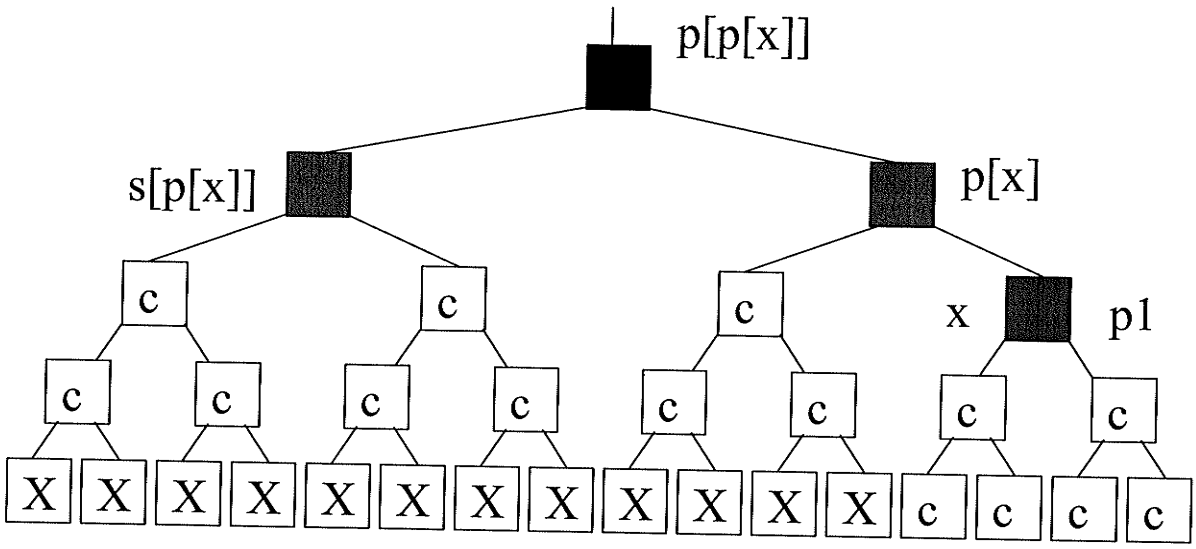

Figure 3-27–P1 and p2 both in Case 1: third placement of p1.

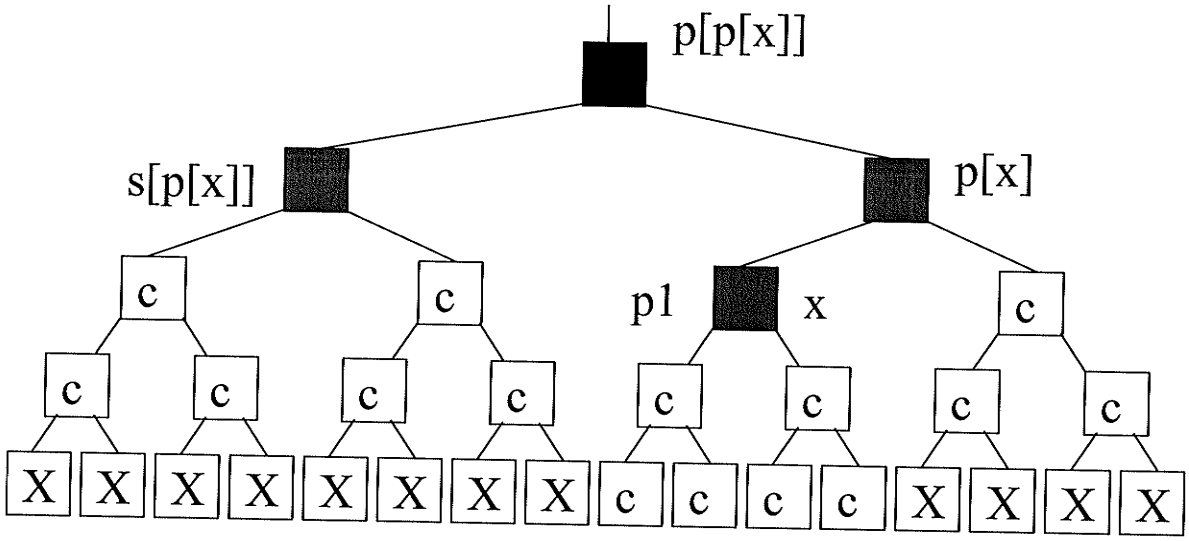

Figure 3-28–P1 and p2 both in Case 1: fourth placement of p1.

When Process P1 is in Case 1 and P2 is in Case 2

As before, since process `p1` is in Case 1, there is no shape constraint on the subtree where `p1` locates, and `p1`'s `x` is red, its parent is red and its uncle is also red. We know `p2` will operate only on the node set containing `x`, `p[x]`, `p[p[x]]`, `p[p[p[x]]]`, and `left[x]` or `right[x]`. If process `p2` (which is now in Case 2) is located on the positions marked 'c' in the following

figures, there will be a conflict between `p1` and `p2`, since their node sets will intersect. There are two access patterns for a process (like `p2`) that is in Case 2: DL-UL-UR-DR and DR-UR-UL-DL. Figure 3-29 and Figure 3-30 show these access patterns for the first possible placement of `p1`. Figure 3-31, Figure 3-32, Figure 3-33, Figure 3-34, Figure 3-35, and Figure 3-36 show the other possible placements.

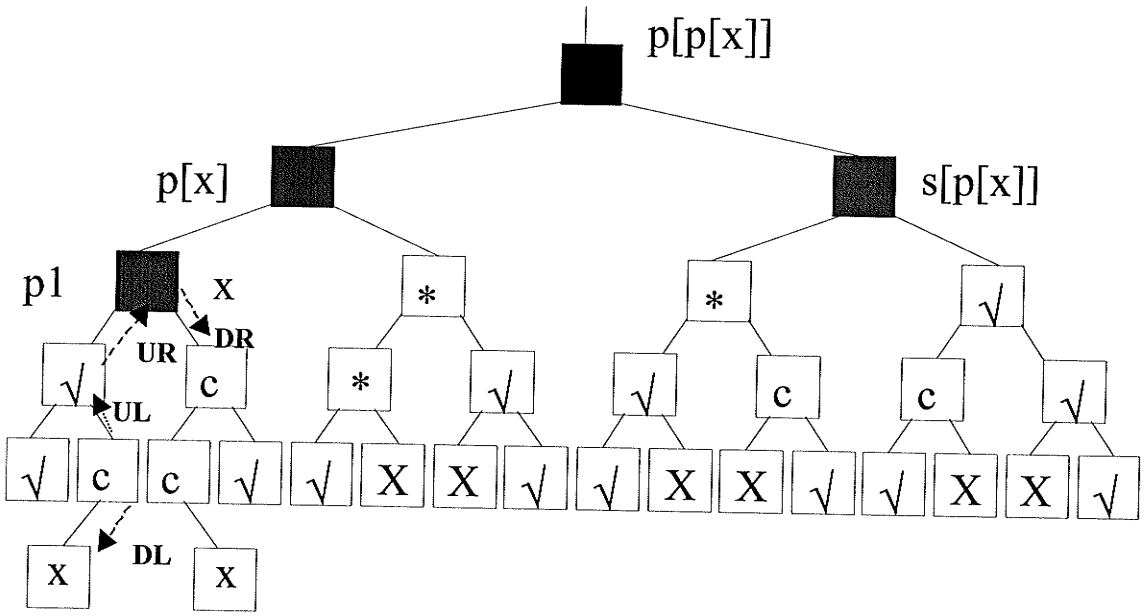

Figure 3-29–P1 in Case 1, p2 in Case 2: first placement of p1, first access pattern.

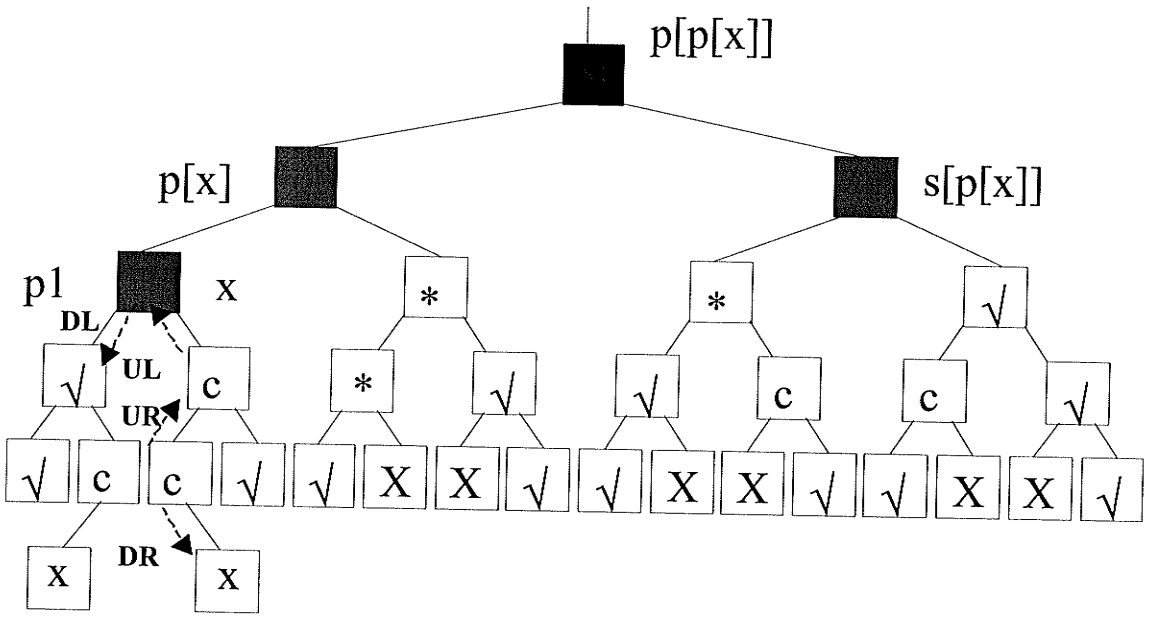

Figure 3-30–P1 in Case 1, p2 in Case 2: first placement of p1, second access pattern.

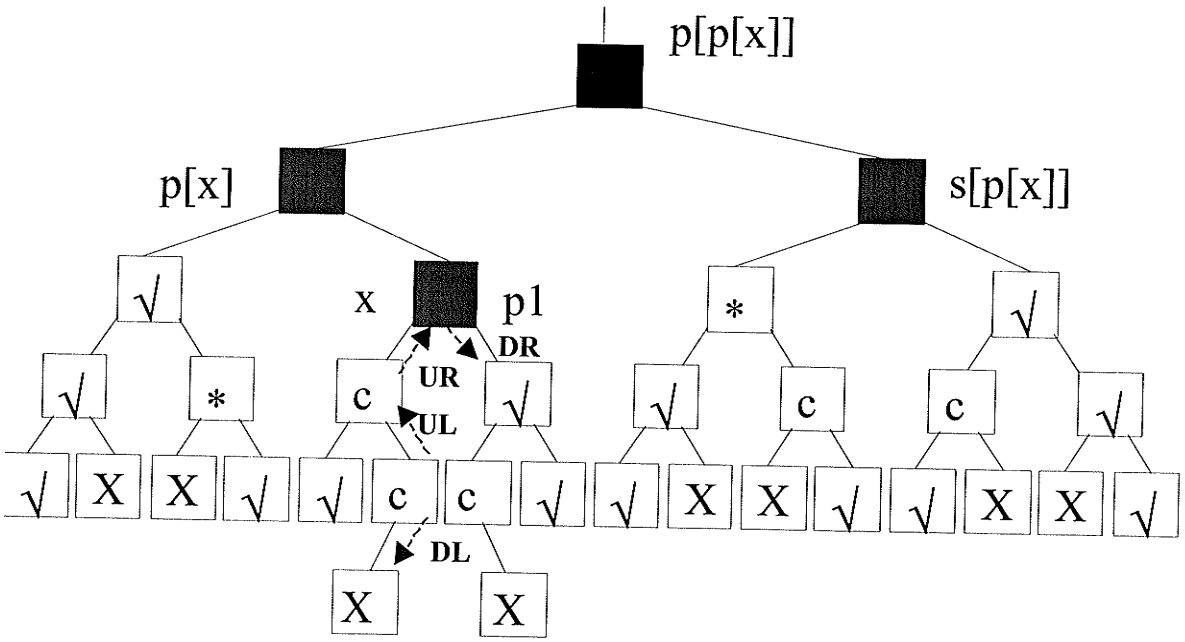

Figure 3-31–P1 in Case 1, p2 in Case 2: second placement of p1, first access pattern.

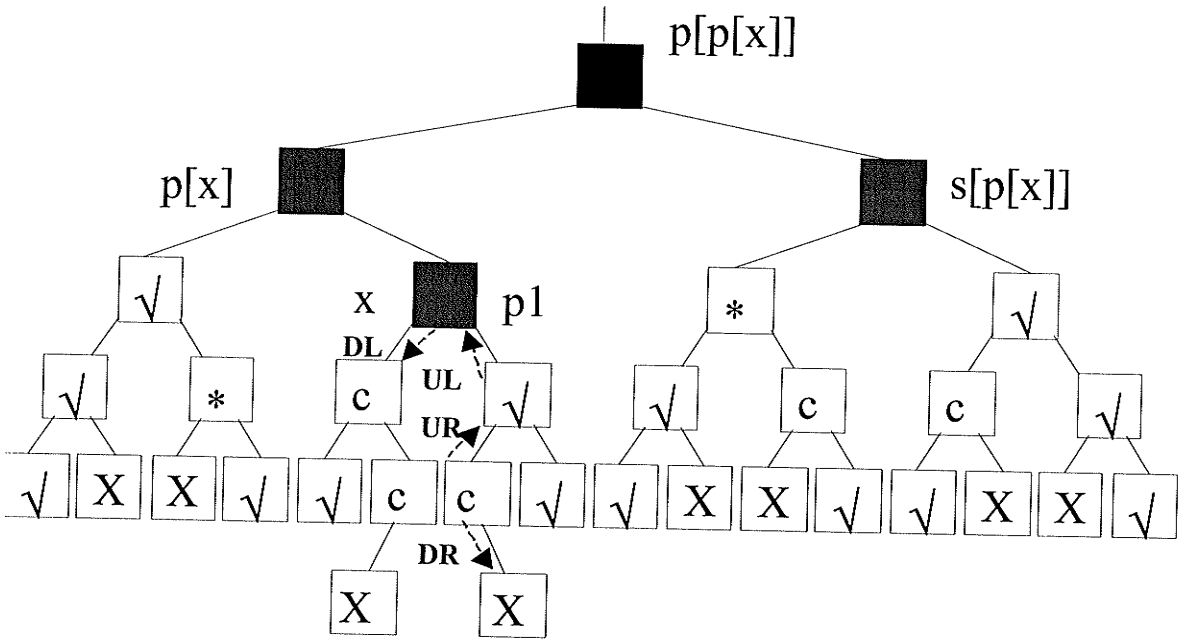

Figure 3-32–P1 in Case 1, p2 in Case 2: second placement of p1, second access pattern.

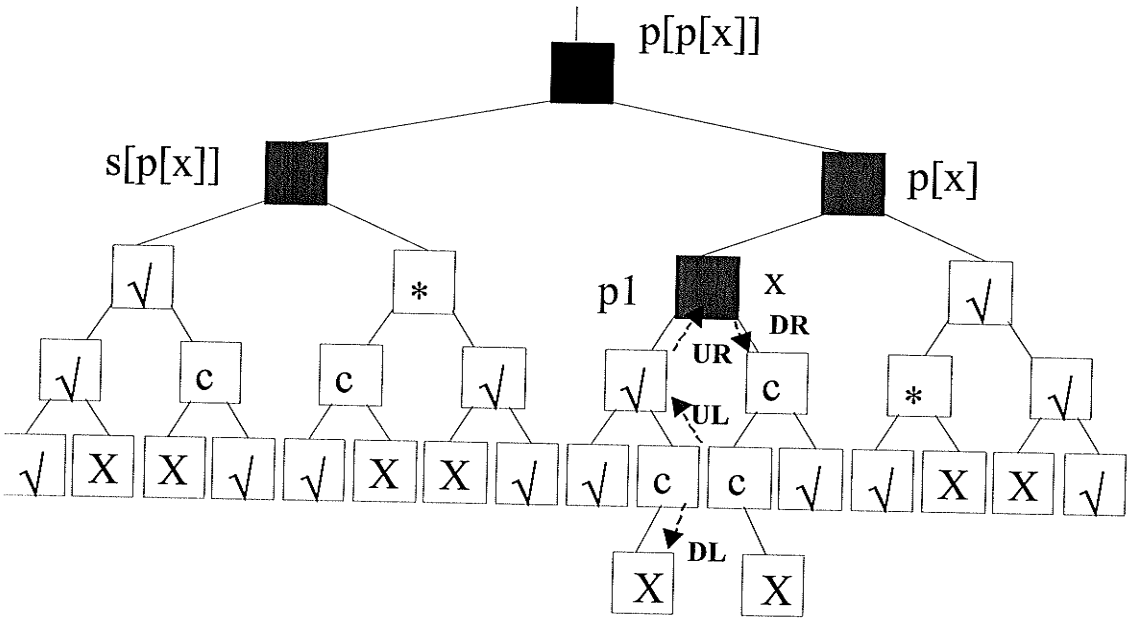

Figure 3-33–P1 in Case 1, p2 in Case 2: third placement of p1, first access pattern.

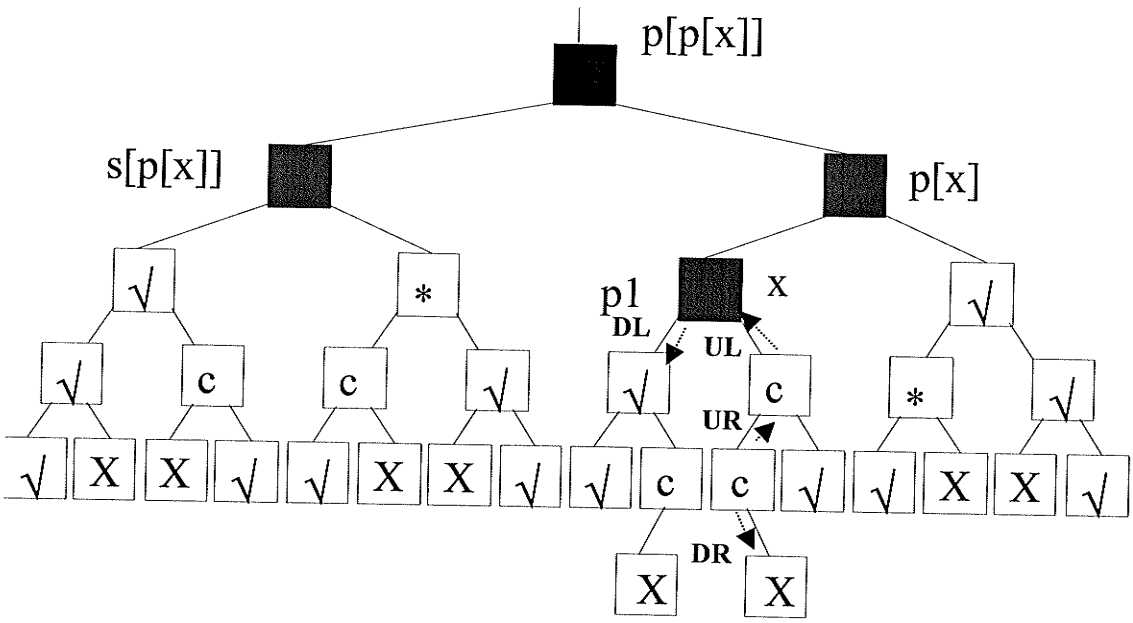

Figure 3-34–P1 in Case 1, p2 in Case 2: third placement of p1, second access pattern.

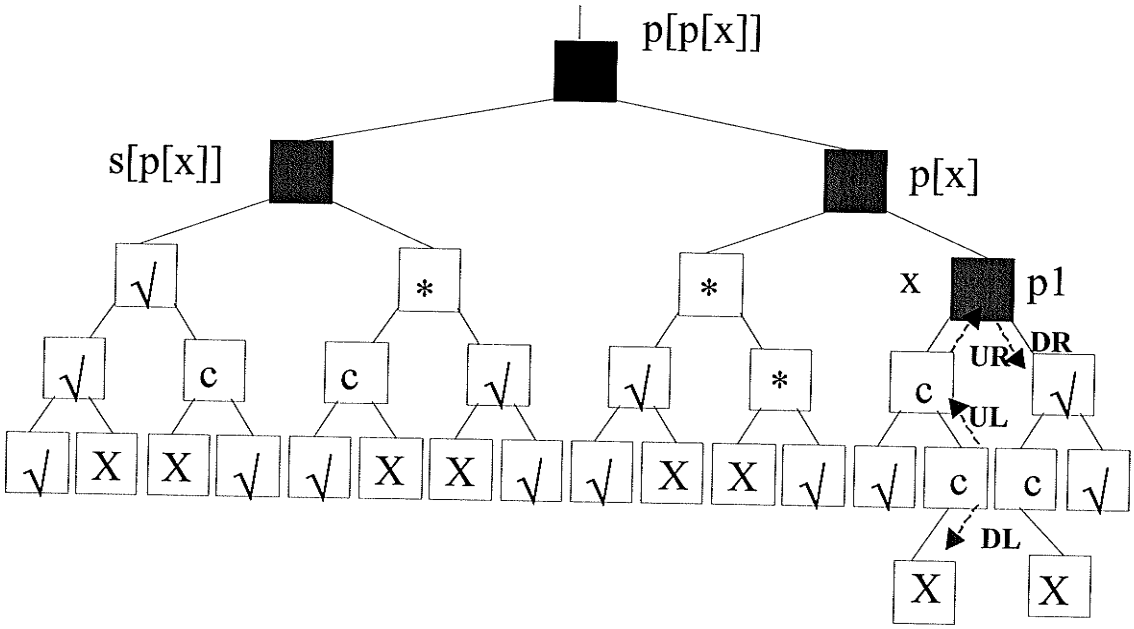

Figure 3-35–P1 in Case 1, p2 in Case 2: fourth placement of p1, first access pattern.

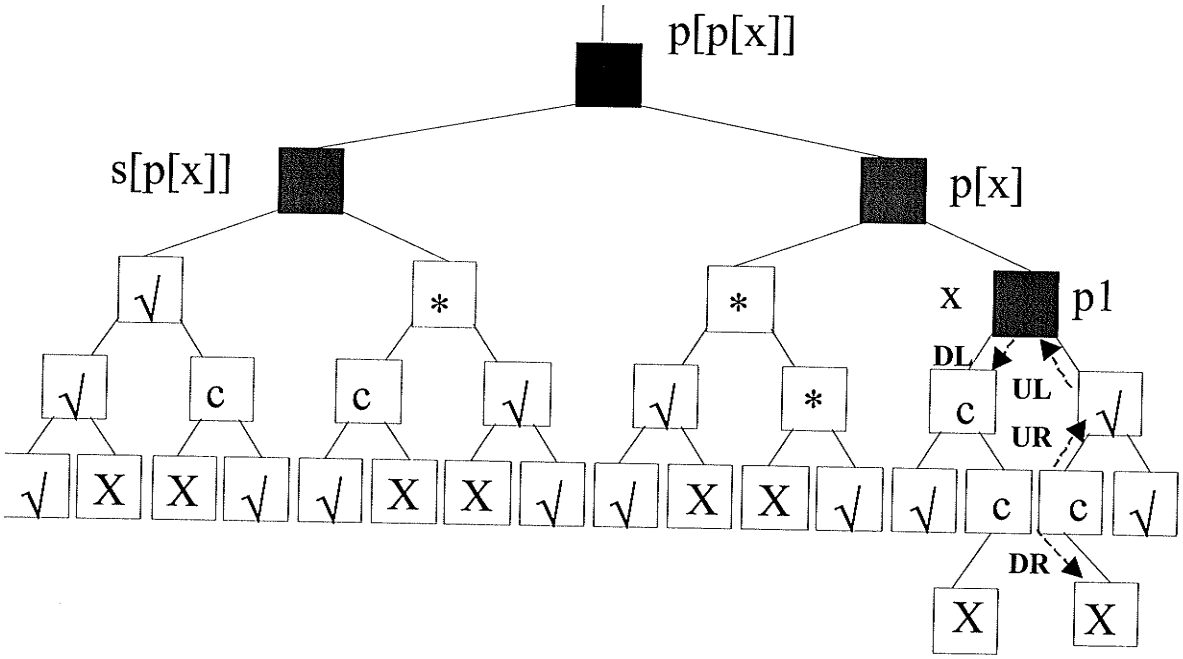

Figure 3-36–P1 in Case 1, p2 in Case 2: fourth placement of p1, second access pattern.

When Process P1 is in Case 1 and P2 is in Case 3

As before, since process `p1` is in Case 1 there is no shape constraint on the subtree where `p1` operates, `p1`'s `x` is red, its parent is red and its uncle is also red and `p1` will operate on the node set {`x`, `p[x]`, `p[p[x]]`, `s[p[x]]`}. If process `p2` (which is now in Case 3) locates on the positions marked 'c' in the following figures, there will be a conflict between `p1` and `p2` since their node sets will intersect. When a process (like `p2`) is in Case 3, there are two possible access patterns relative to `x`'s parent: DR-UR-UR and DL-UL-UL. Figure 3-37 shows these access patterns for the first possible placement of `p1`. Figure 3-38, Figure 3-39, Figure 3-40 and Figure 3-41 show the other possible placements.

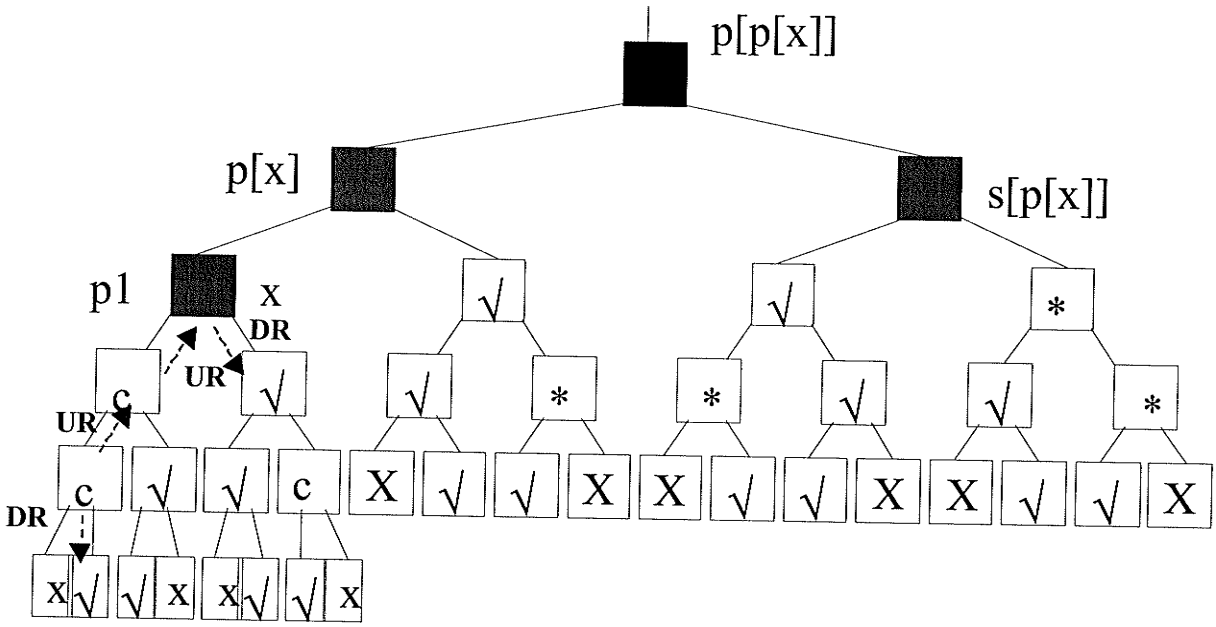

Figure 3-37–P1 in Case 1, p2 in Case 3: first placement of p1, first access pattern.

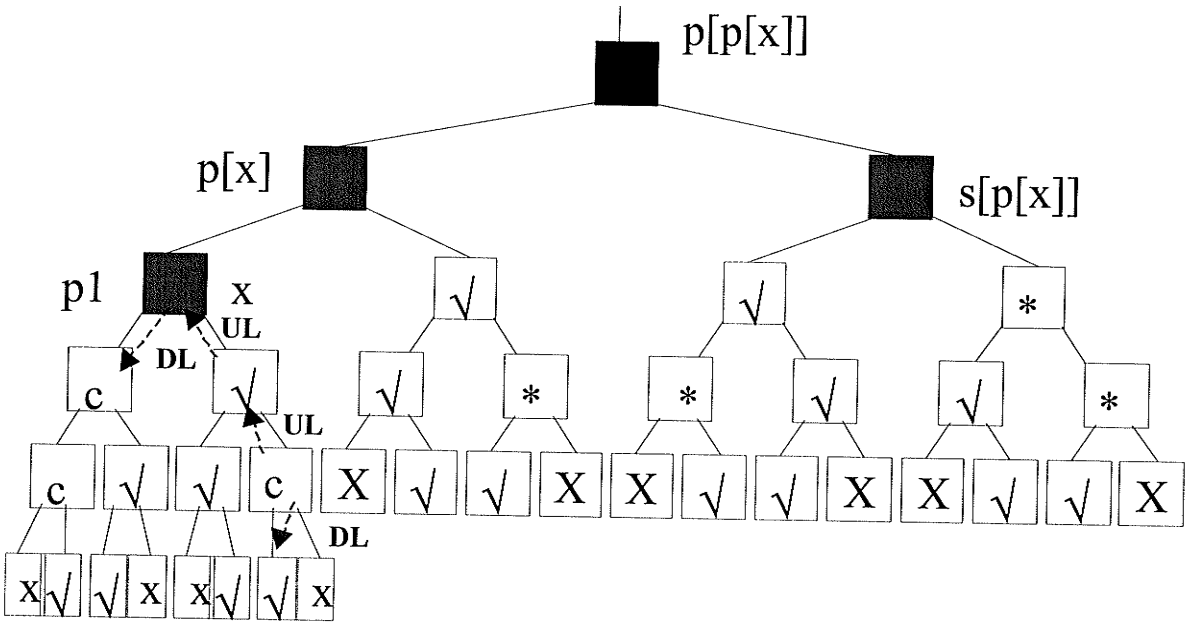

Figure 3-38–P1 in Case 1, p2 in Case 3: first placement of p1, second access pattern.

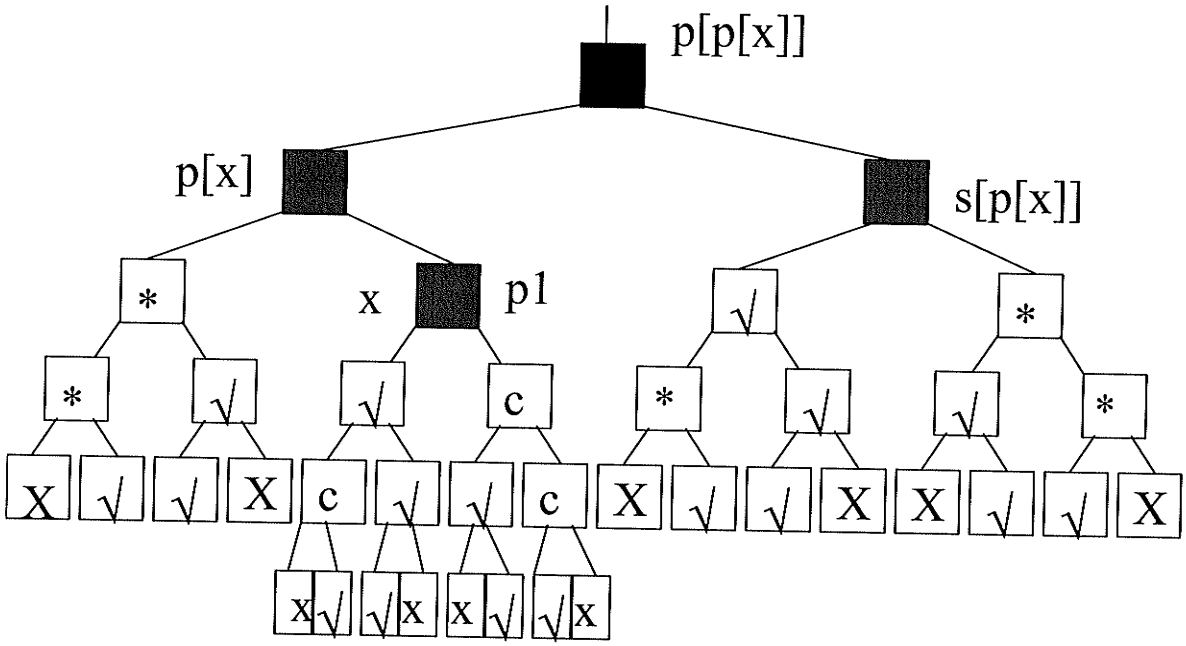

Figure 3-39–P1 in Case 1, p2 in Case 3: second placement of p1.

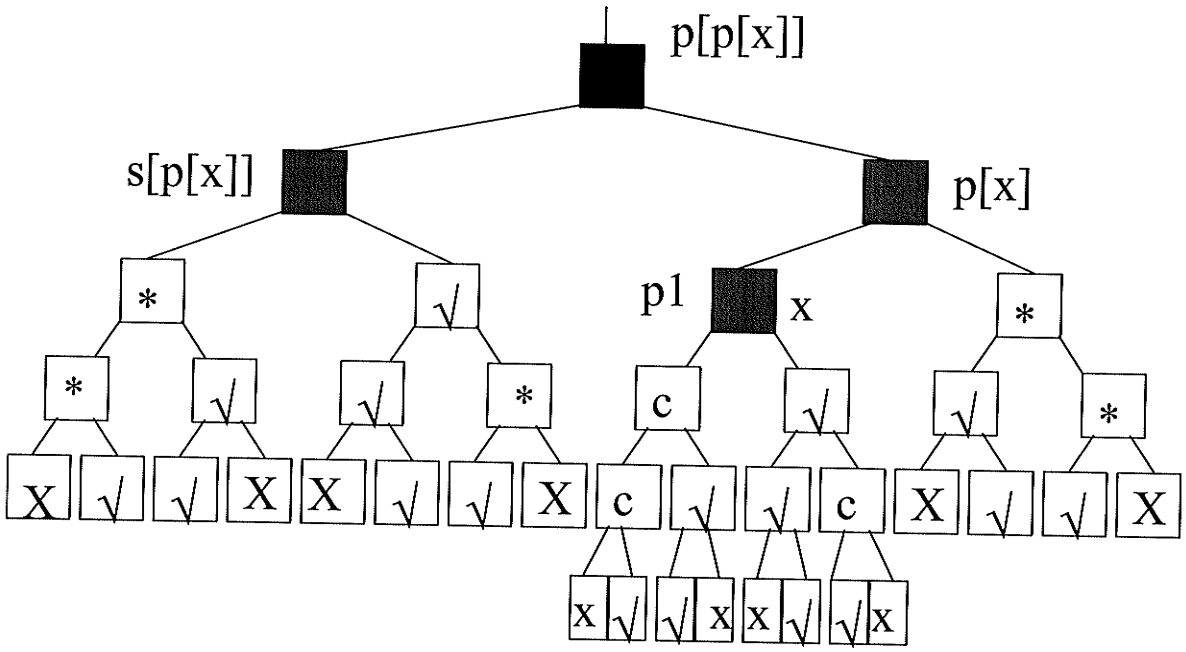

Figure 3-40–P1 in Case 1, p2 in Case 3: third placement of p1.

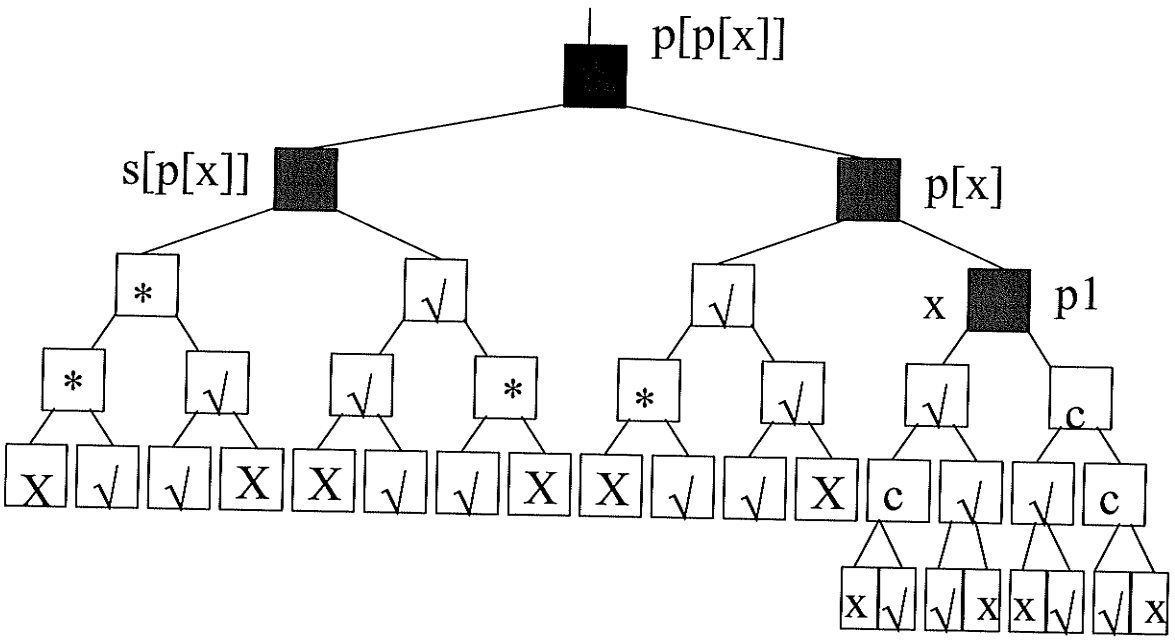

Figure 3-41–P1 in Case 1, p2 in Case 3: fourth placement of p1.

3.3.2.4 Process P1 is in Case 2

When Process P2 is in Case 1

As shown in Figure 3-42, Figure 3-43, Figure 3-44, and Figure 3-45, `p2` is in Case1. The shape of the subtree where `P1` operates is "zigzag" (as shown in Figure 3-20). Due to the colour constraints mentioned previously, `p1`'s `x` is red, its parent is red, but its uncle is black. Due to access constraints, `p1` will operate on the node set {`x, p[x], p[p[x]], p[p[p[x]]]`, and either `left[x]`, or `right[x]`}. If process `p2` locates on the positions marked 'c' in Figure 3-42, Figure 3-43, Figure 3-44 and Figure 3-45, there is a conflict between `p1` and `p2`, because at least one node is in the node sets of both `p1` and `p2`. When `p2` is in Case 1, there are four possible access patterns for `p2`, UR-UR-DR, UL-UR-DR, UR-UL-DL and UL-UL-DL. Figure 3-46 shows the symmetric situation, and illustrates the other possible access patterns for `p2` as stated above.

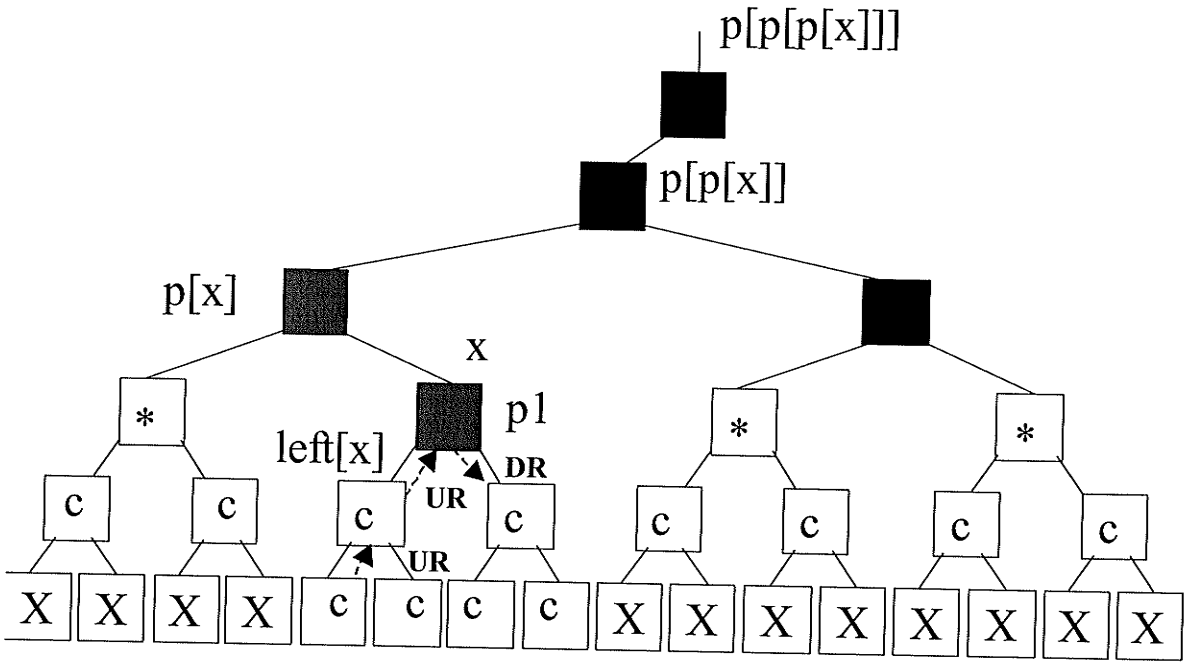

Figure 3-42–P1 in Case 2, p2 in Case 1: first placement of p1, first access pattern.

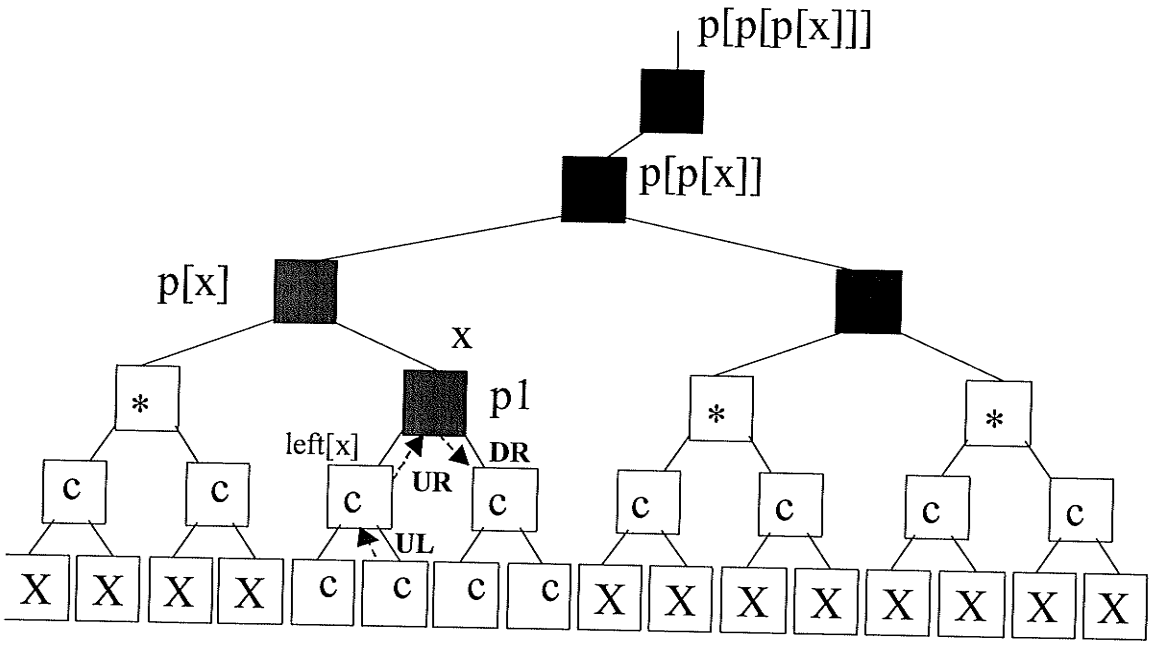

Figure 3-43–P1 in Case 2, p2 in Case 1: first placement of p1, second access pattern.

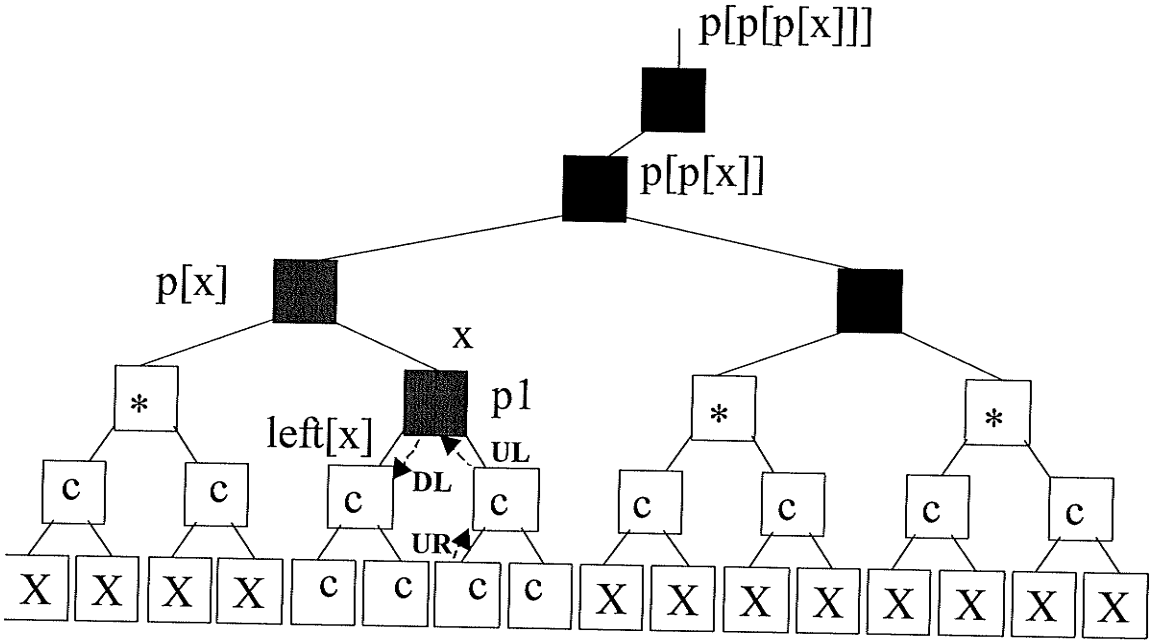

Figure 3-44–P1 in Case 2, p2 in Case 1: first placement of p1, third access pattern.

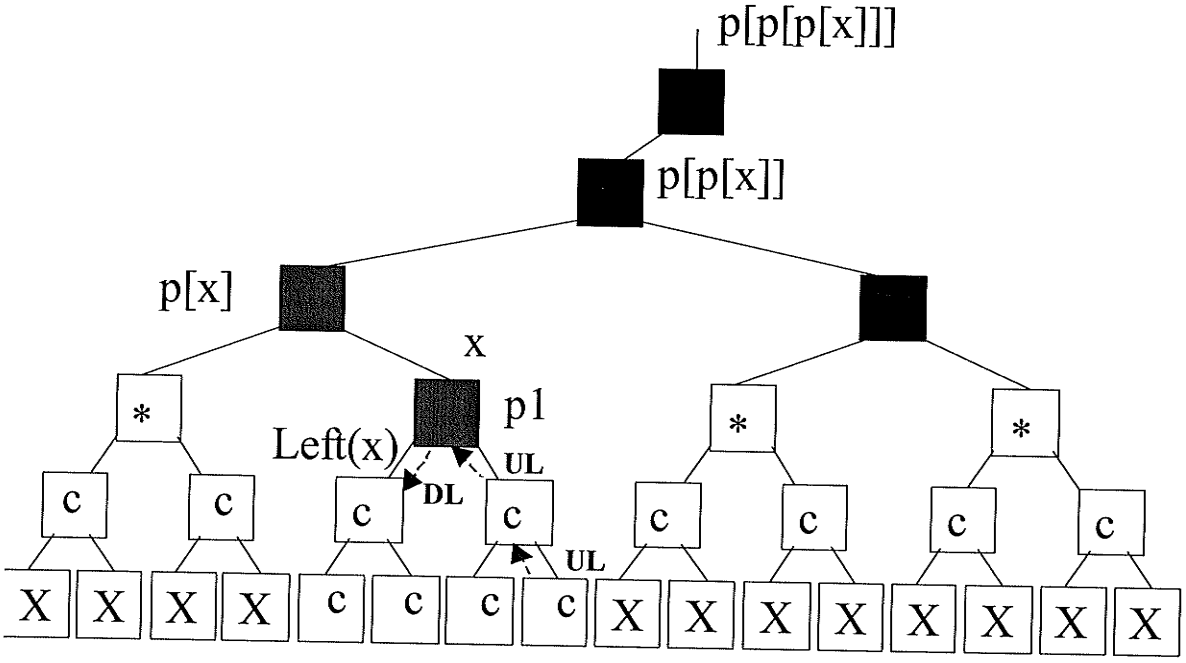

Figure 3-45–P1 in Case 2, p2 in Case 1: first placement of p1, fourth access pattern.

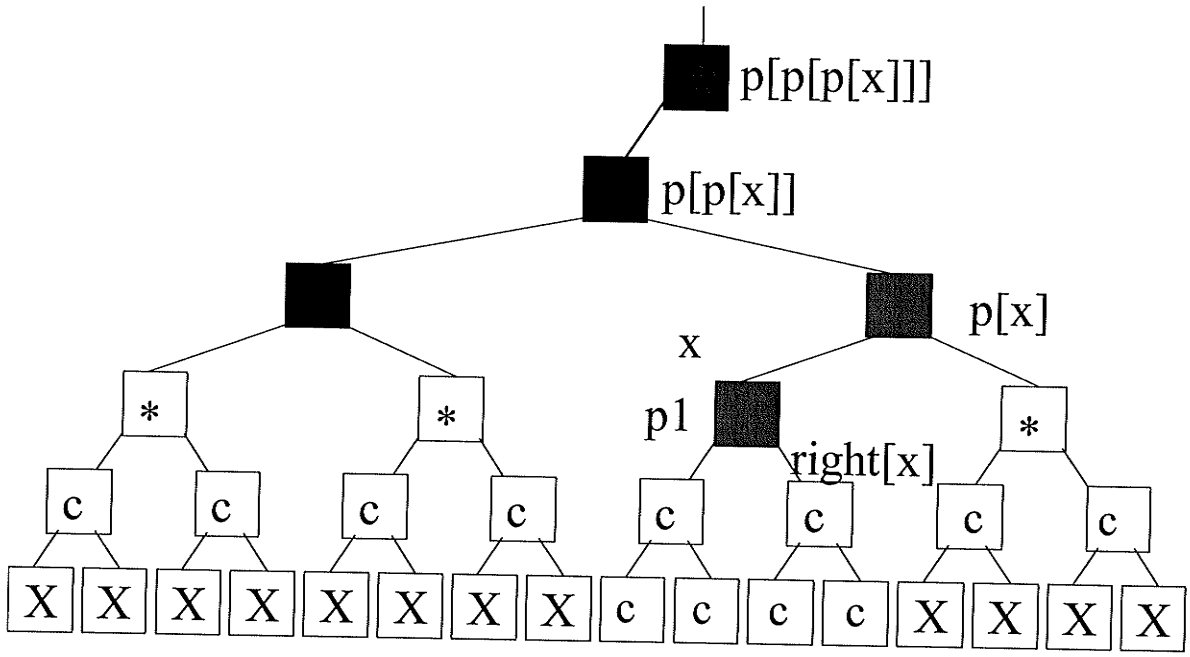

Figure 3-46–P1 in Case 2 and p2 in Case 1: second placement of p1.

When Process P2 is in Case 2 (Process P1 is still in Case 2)

As shown in Figure 3-47 and Figure 3-48, process `p1` is in Case 2. The shape of the subtree where `p1` operates is "zigzag" (shown in Figure 3-20). Due to the colour constraints mentioned above, `p1`'s `x` is red, its parent is red, but its uncle is black. Again, due to access constraints, `p1` will operate on the node set {`x`, `p[x]`, `p[p[x]]`, `p[p[p[x]]]`, and either `left[x]` or `right[x]`}. If process `p2` locates on the positions marked 'c' in Figure 3-47 and Figure 3-48, there is a conflict between `p1` and `p2`, because at least one node is in the node sets of both `p1` and `p2`. When `p2` is in Case 2, there are two possible access patterns for `p2`, DL-UL-UR-DR and DR-UR-UL-DL. Figure 3-49 shows the symmetric scenarios, and illustrates the other possible access patterns.

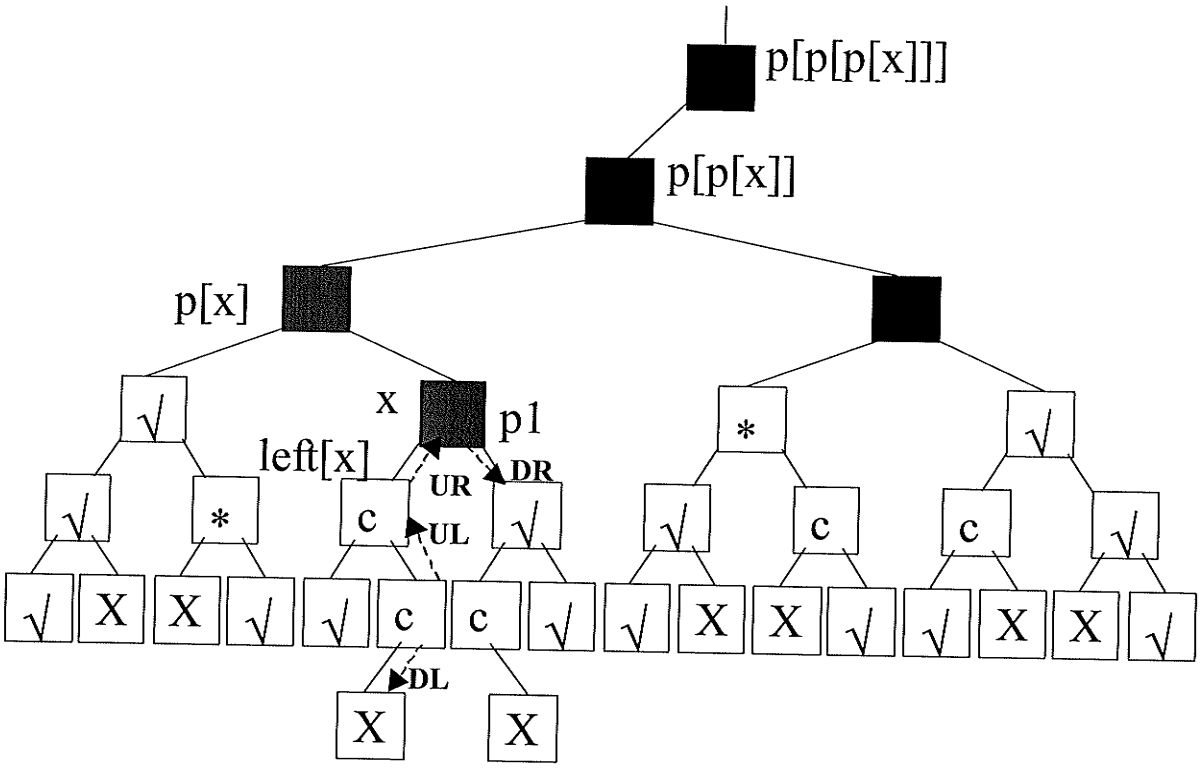

Figure 3-47–P1 in Case 2, p2 in Case 2: first placement of p1, first access pattern.

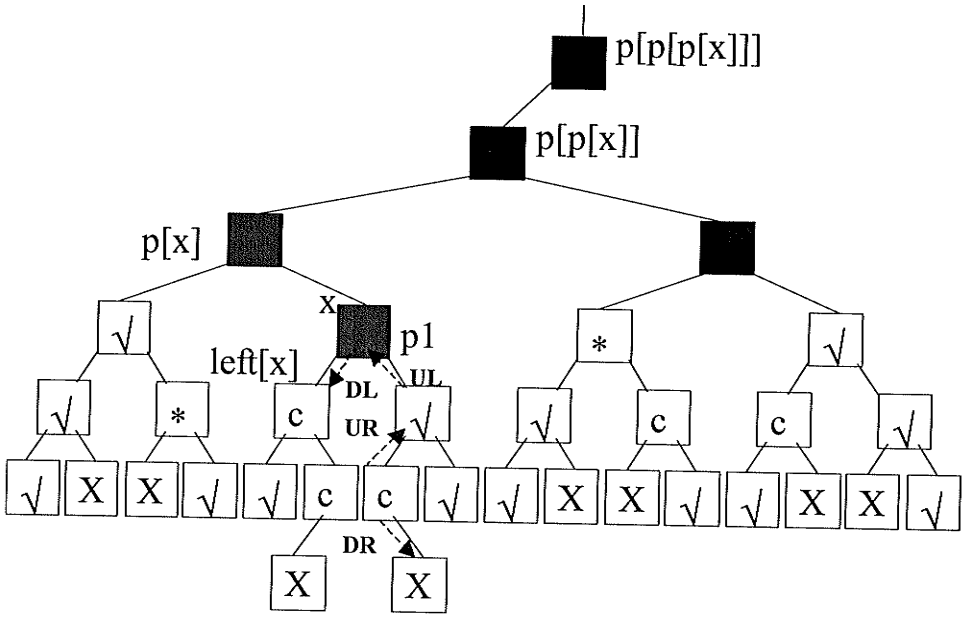

Figure 3-48–P1 in Case 2, p2 in Case 2: first placement of p1, second access pattern.

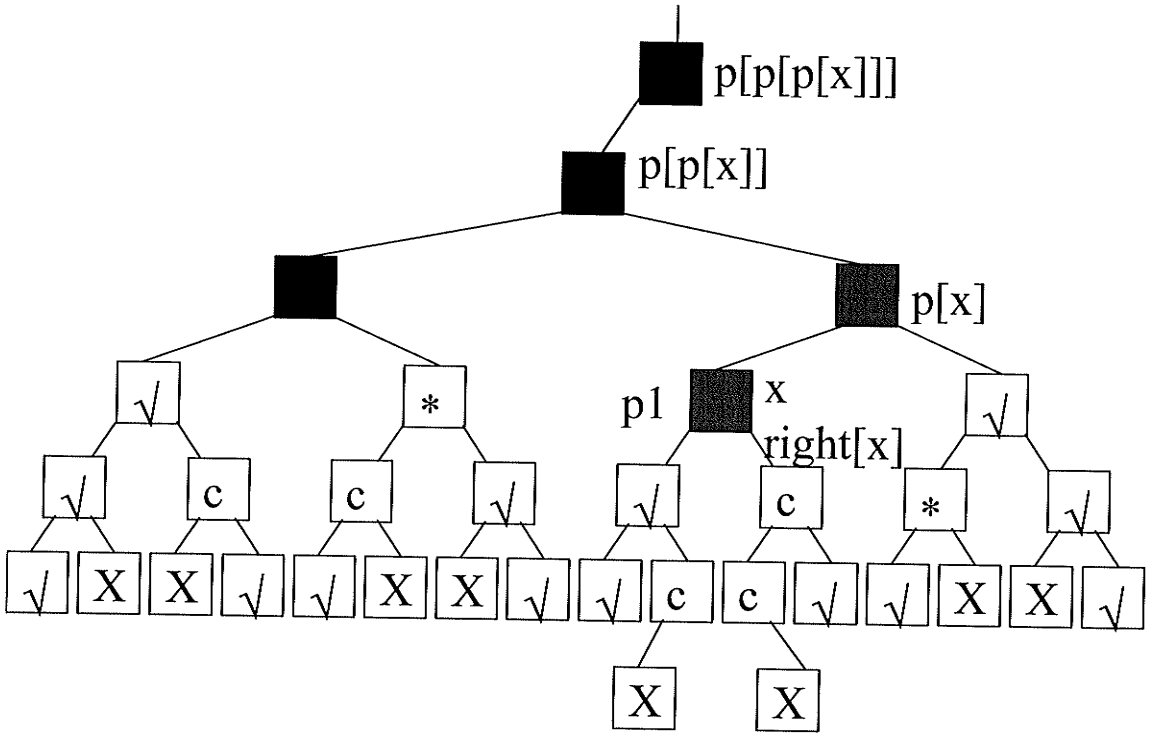

Figure 3-49–P1 in Case 2 and p2 in Case 2: second placement of p1.

When Process P2 is in Case 3 (Process P1 is still in Case 2)

As shown in Figure 3-50 and Figure 3-51, Process `p1` is in Case 2. So the shape of the subtree where `p1` operates is "zigzag" (as shown in Figure 3-20). Due to the colour constraints mentioned above, `p1`'s `x` is red, its parent is red, but its uncle is black. Again, due to access constraints, `p1` will operate on the node set {`x, p[x], p[p[x]], p[p[p[x]]]`, and either `left[x]`, or `right[x]`}. If Process `p2` locates on the positions marked 'c' in Figure 3-50 and Figure 3-51, there is a conflict between `p1` and `p2`, because at least one node is in the node sets of both `p1` and `p2`. When `p2` is in Case 3, there are two possible access patterns for `p2`, UR-UR-DR and UL-UL-DL. Figure 3-52 shows the symmetric situations, and illustrates the other possible access patterns.

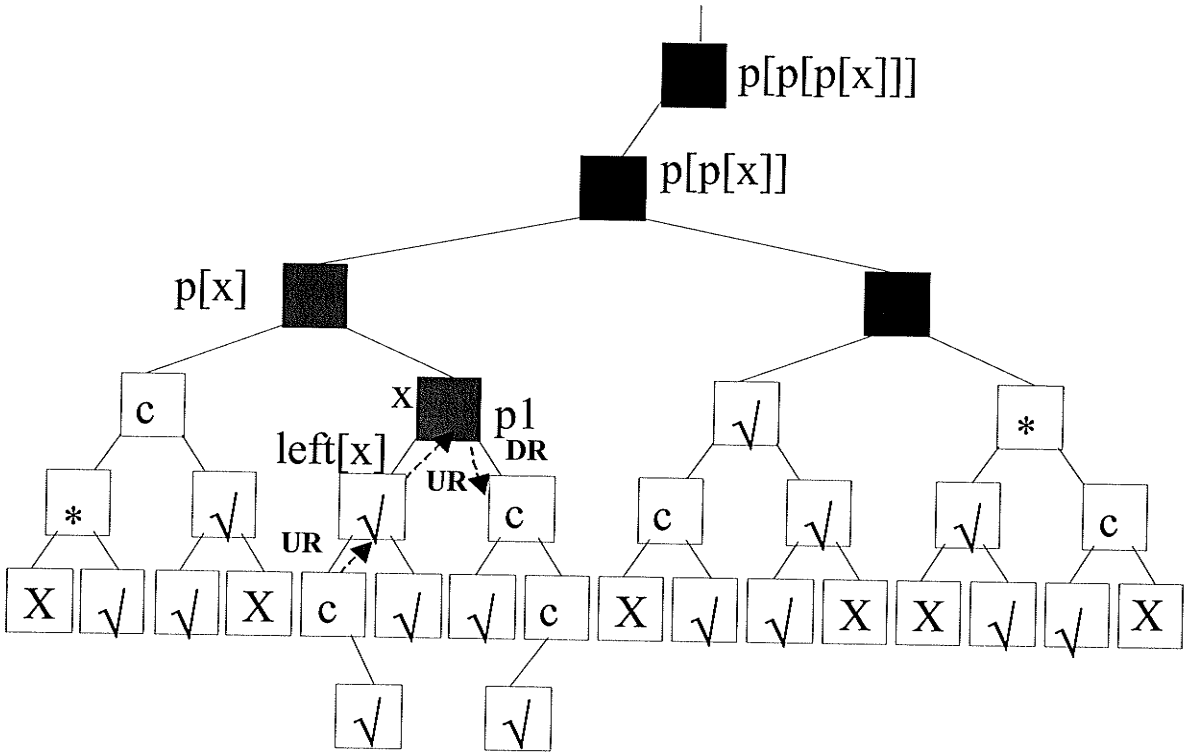

Figure 3-50–P1 in Case 2, p2 in Case 3: first placement of p1, first access pattern.

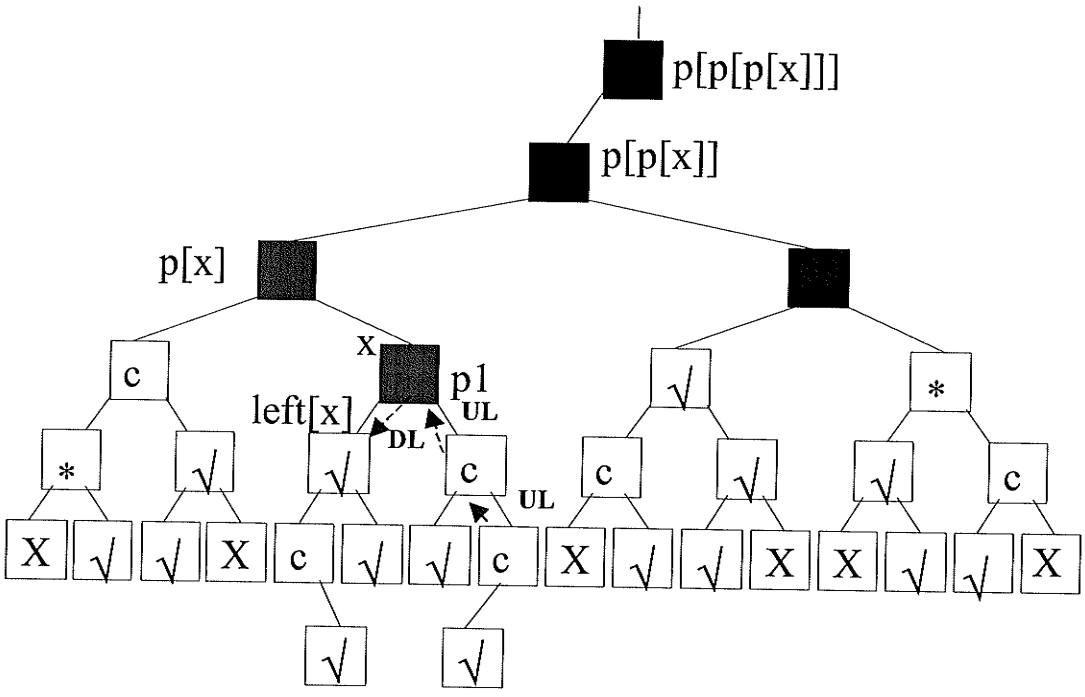

Figure 3-51–P1 in Case 2 and p2 in Case 3: first placement of p1, second access pattern.

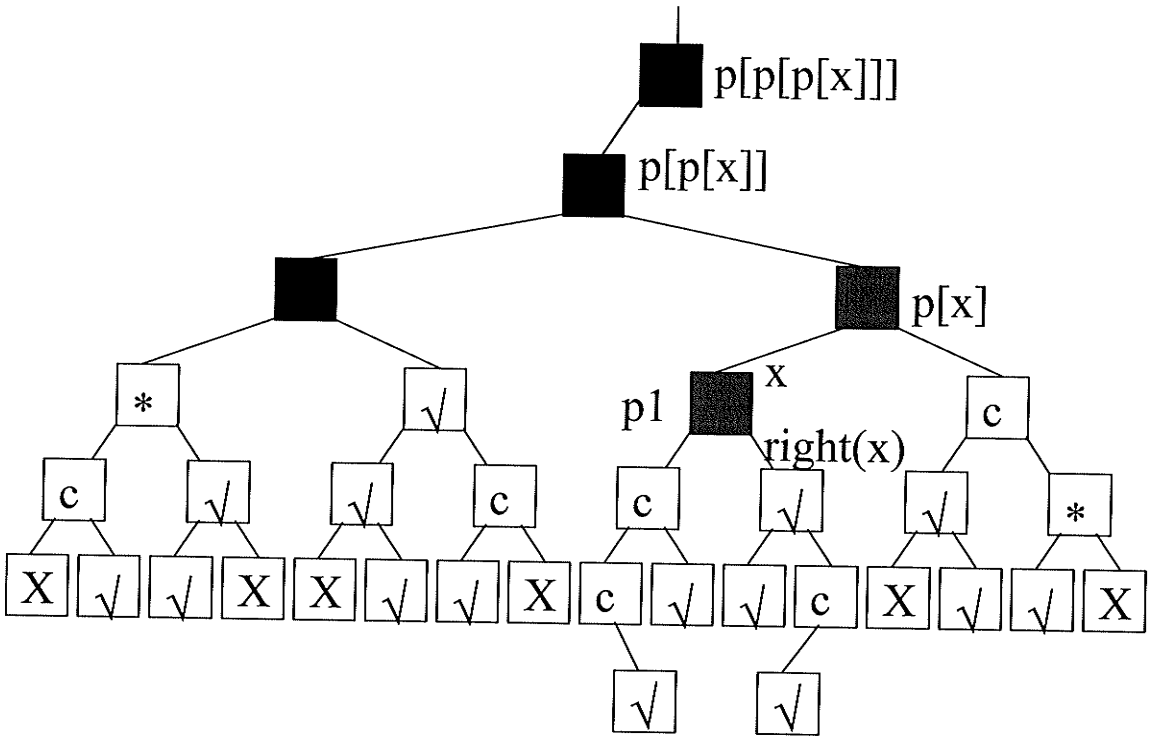

Figure 3-52–P1 in Case 2 and p2 in Case 3: second placement of p1.

3.3.2.5 Process P1 is in Case 3

When Process P2 is in Case 1

As shown in Figure 3-53, Figure 3-54, Figure 3-55 and Figure 3-56, process `p1` is in Case 3. The shape of the subtree where `p1` operates is "straight" (as shown in Figure 3-21). Due to the colour constraints mentioned earlier, `p1`'s `x` is red, its parent is red, but its uncle is black. Again, due to access constraints, `p1` will operate on the node set {`x`, `p[x]`, `p[p[x]]`, `p[p[p[x]]]`, and `s[x]`}. If process `p2` locates on the positions marked 'c' in Figure 3-53, Figure 3-54, Figure 3-55 and Figure 3-56, there is a conflict between `p1` and `p2`, because at least one node is in the node sets of both `p1` and `p2`. When `p2` is in Case 1, there are 4 possible access patterns for `p2`, UR-UR-DR, UL-UR-DR, UR-UL-DL and UL-UL-DL. Figure 3-57 shows the symmetric situations, and illustrates the other access patterns.

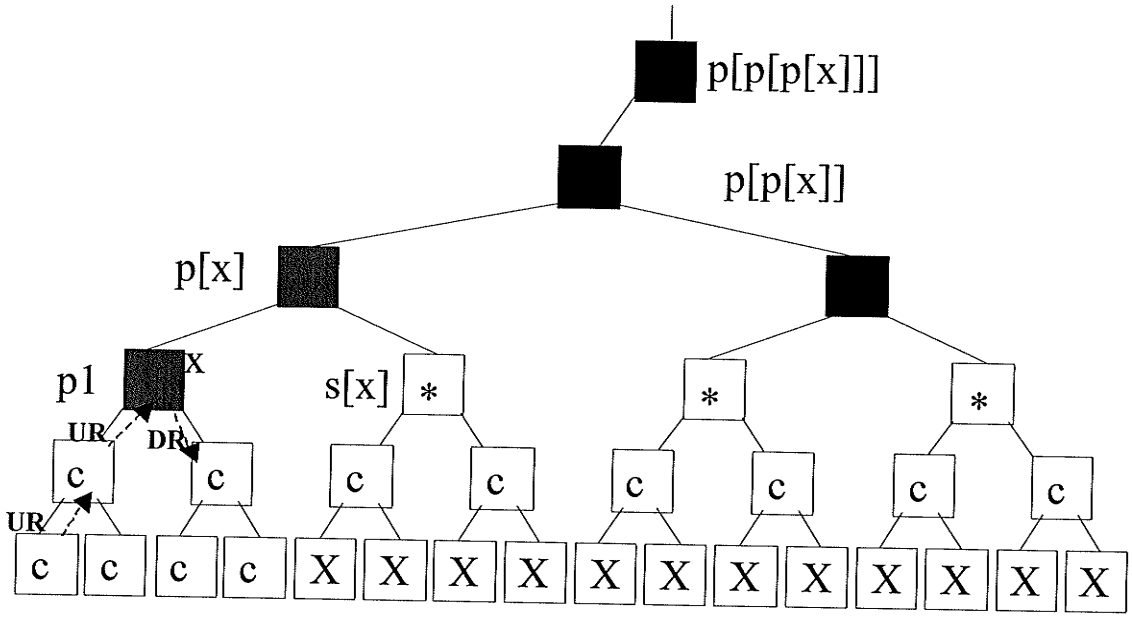

Figure 3-53–P1 in Case 3, p2 in Case 1: first placement of p1, first access pattern.

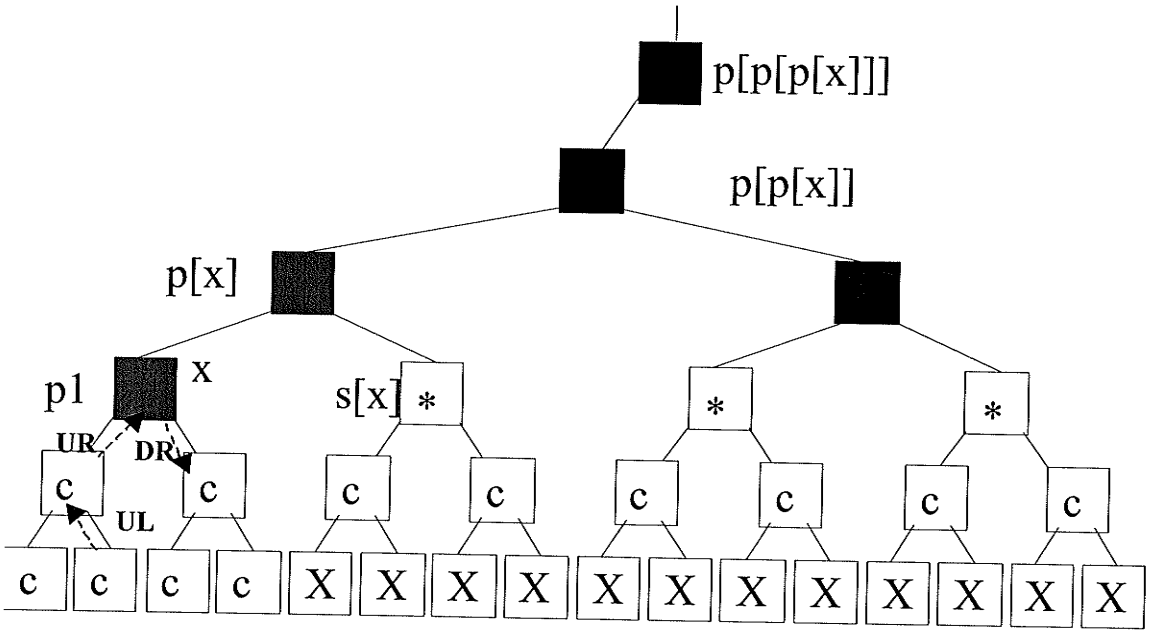

Figure 3-54–P1 in Case 3 and p2 in Case 1: first placement of p1, second access pattern.

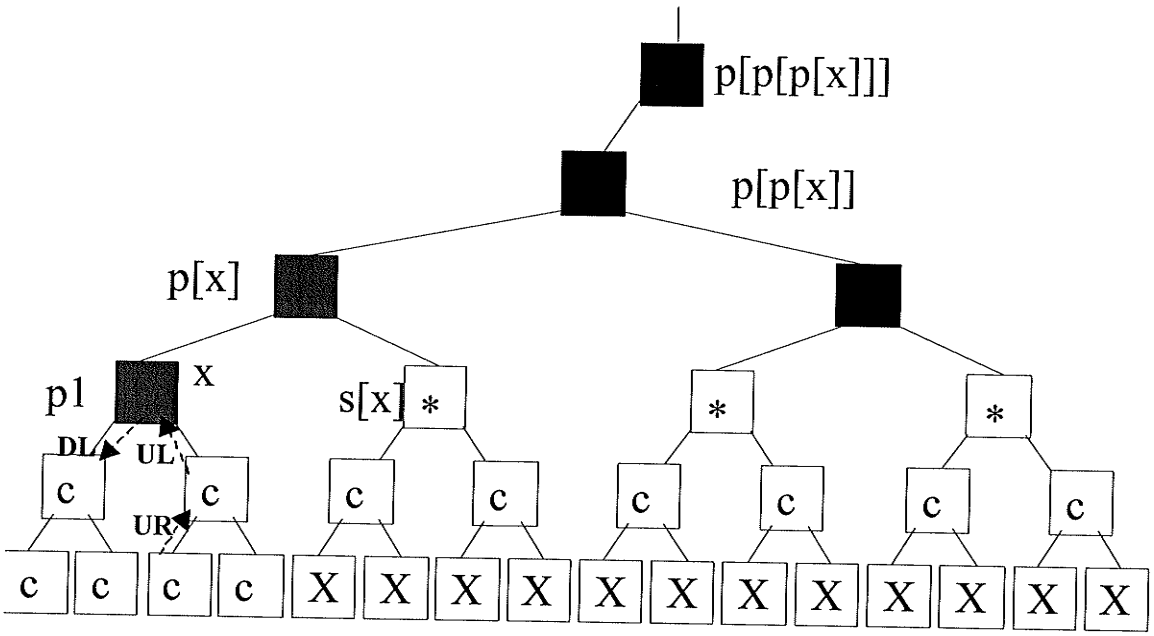

Figure 3-55–P1 in Case 3 and p2 in Case 1: first placement of p1, third access pattern.

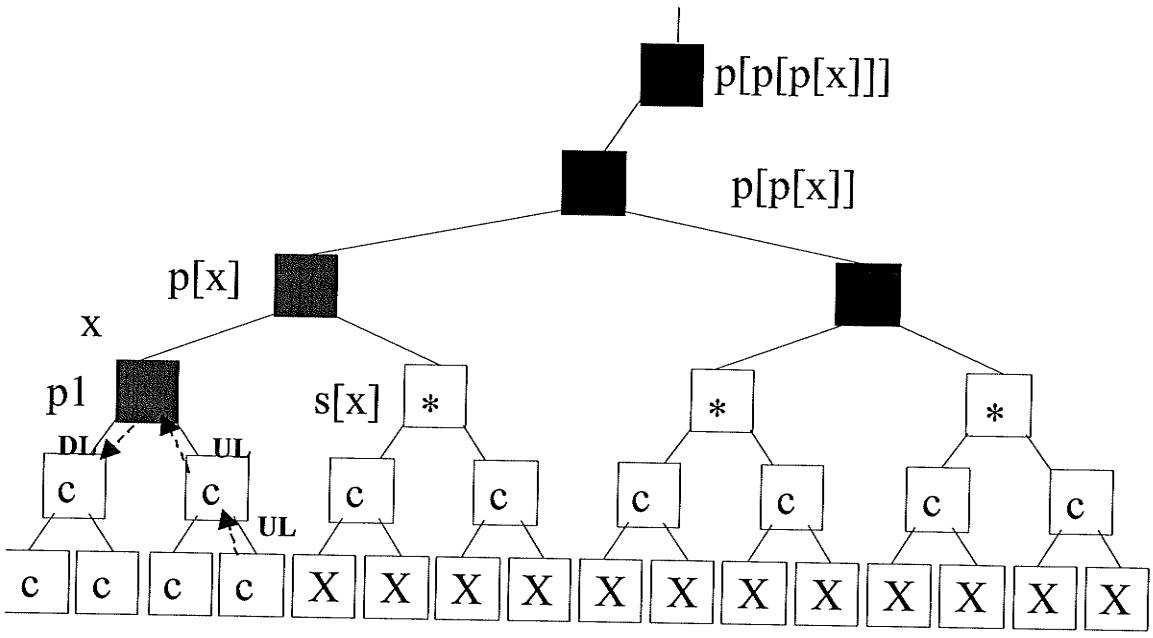

Figure 3-56–P1 in Case 3 and p2 in Case 1: first placement of p1, fourth access pattern.

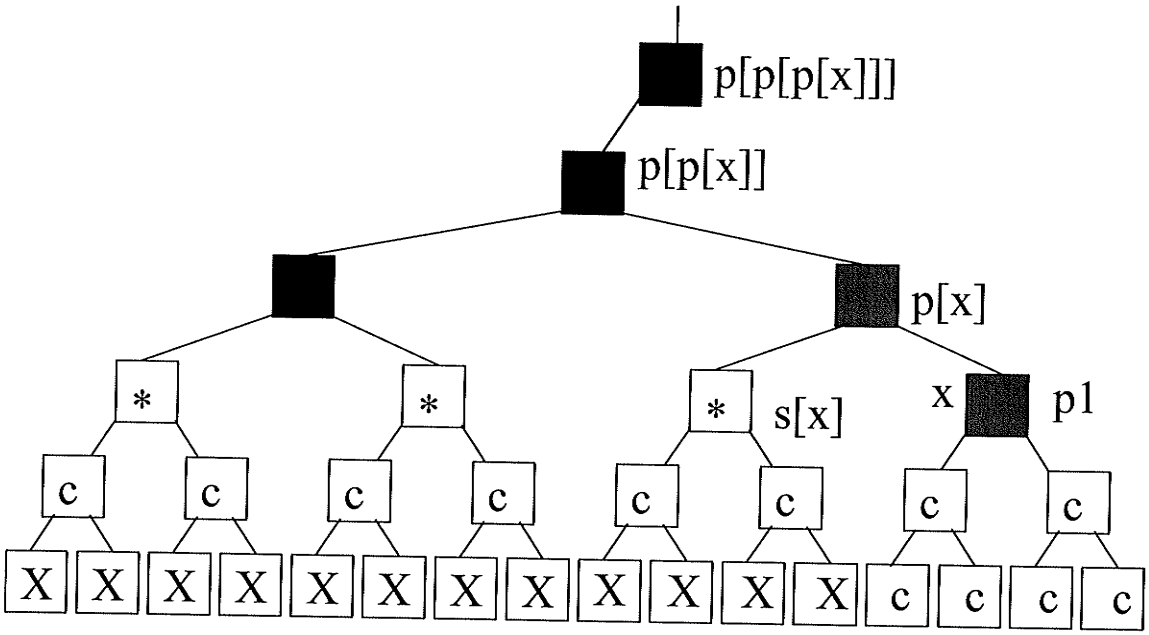

Figure 3-57–P1 in Case 3 and p2 in Case 1: second placement of p1.

When Process P2 is in Case 2 (Process P1 is still in Case 3)

As shown in Figure 3-58 and Figure 3-59 , process `p1` is in Case 3. The shape of the subtree where `p1` operates is "straight" (as shown in Figure 3-21). Due to the colour constraints mentioned above, `p1`'s `x` is red, its parent is red, but its uncle is black. Again, due to access constraints, `p1` will operate on the node set {`x`, `p[x]` , `p[p[x]]` , `p[p[p[x]]]` , and `s[x]`}. If process `p2` locates on the positions marked 'c' in Figure 3-58 and Figure 3-59, there is a conflict between `p1` and `p2`, because at least one node is in the node sets of both `p1` and `p2`. When `p2` is in Case 2, there are two possible access patterns for `p2`, DL-UL-UR-DR and DR-UR-UL-DL. Figure 3-60 shows the symmetric scenarios, and illustrates the other access patterns.

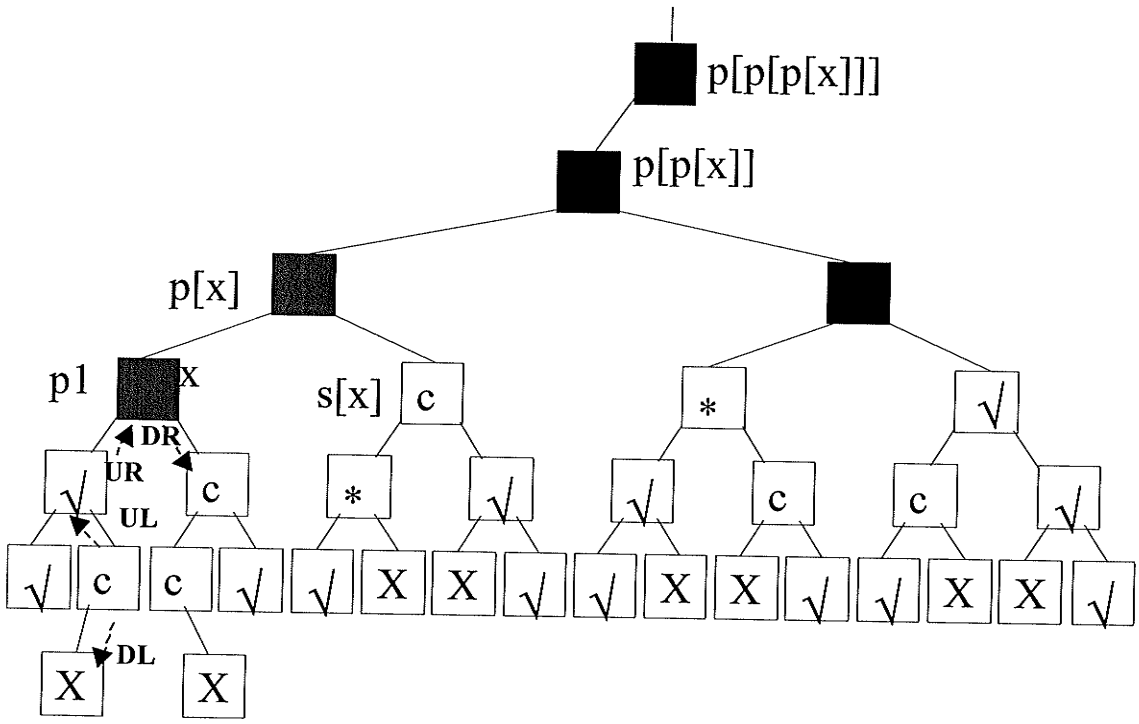

Figure 3-58–P1 in Case 3, p2 in Case 2: first placement of p1, first access pattern.

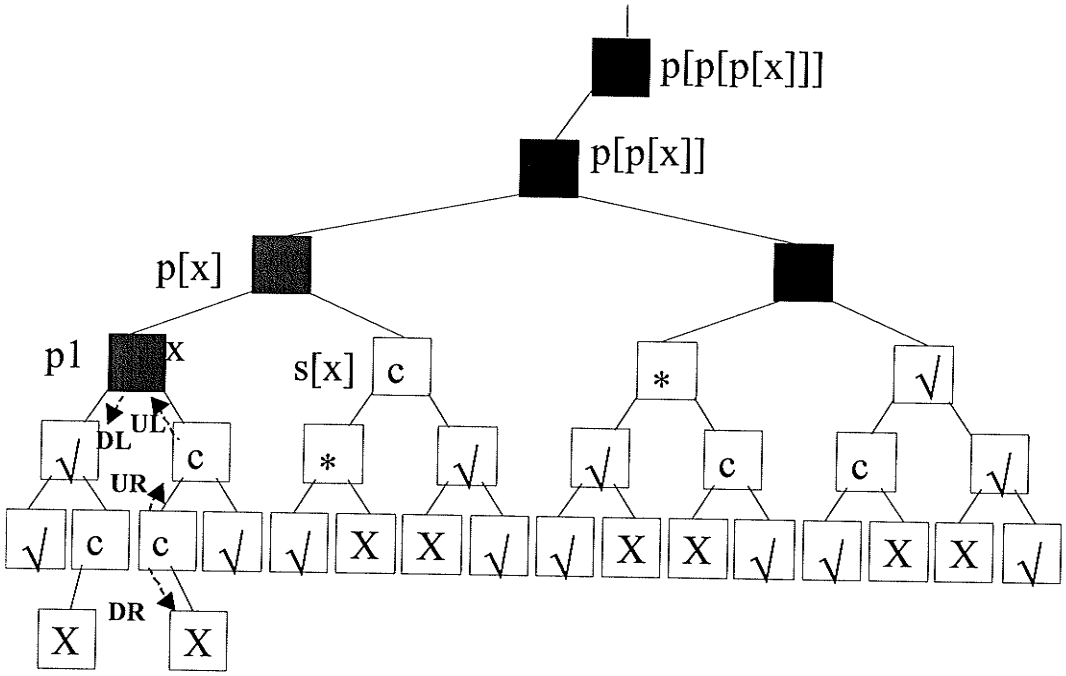

Figure 3-59–P1 in Case 3, p2 in Case 2: first placement of p1, second access pattern.

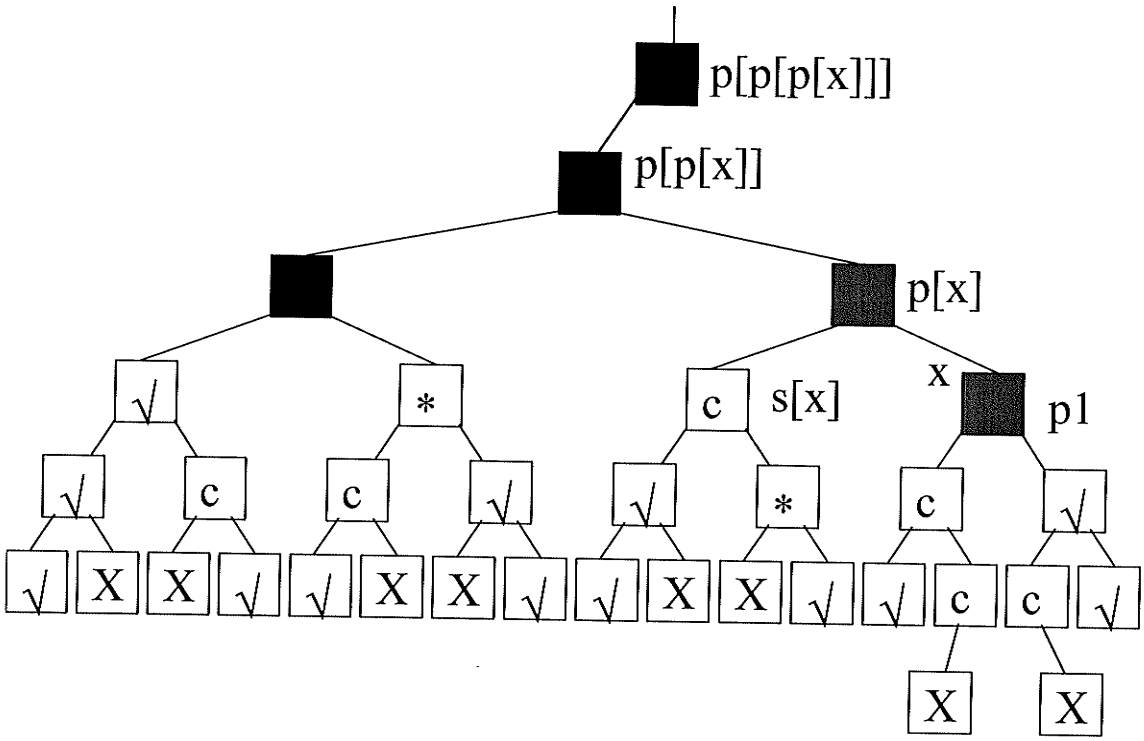

Figure 3-60–P1 in Case 3 and p2 in Case2: second placement of p1.

When Process P2 is in Case 3 (Process P1 is still in Case 3)

As shown in Figure 3-61 and Figure 3-62, process `p1` is in Case 3 and the shape of the subtree where `p1` operates is "straight". Due to the colour constraints mentioned above, `p1`'s `x` is red, its parent is red, and its uncle is black. Again due to access constraints, `p1` will operate on the node set {`x, p[x], p[p[x]], p[p[p[x]]],` and `s[x]`}. If process `p2` locates on the positions marked 'c' in Figure 3-61 and Figure 3-62, there is a conflict between `p1` and `p2`, because at least one node is in the node sets of both `p1` and `p2`. When `p2` is in Case 3, there are two possible access patterns for `p2`, DR-UR-UR-DR and DL-UL-UL-DL. Figure 3-63 shows symmetric situations, and illustrates the other access patterns.

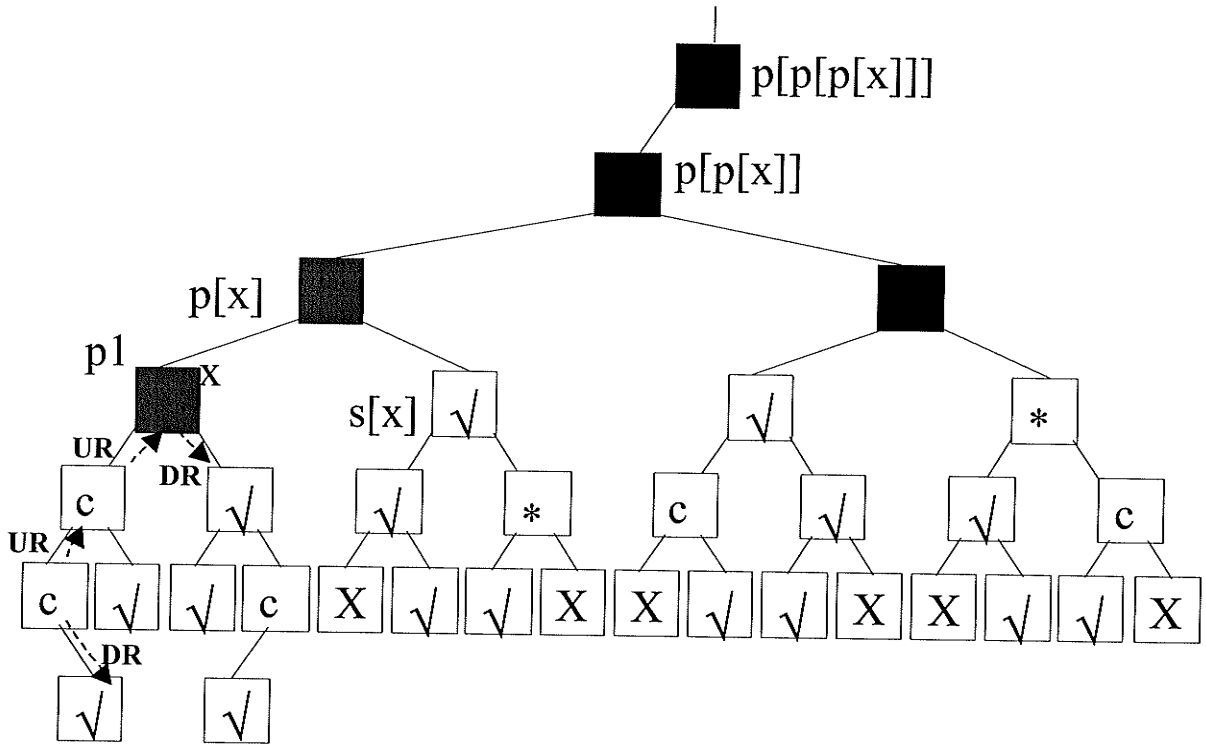

Figure 3-61–P1 in Case 3 and p2 in Case 3: first placement of p1, first access pattern.

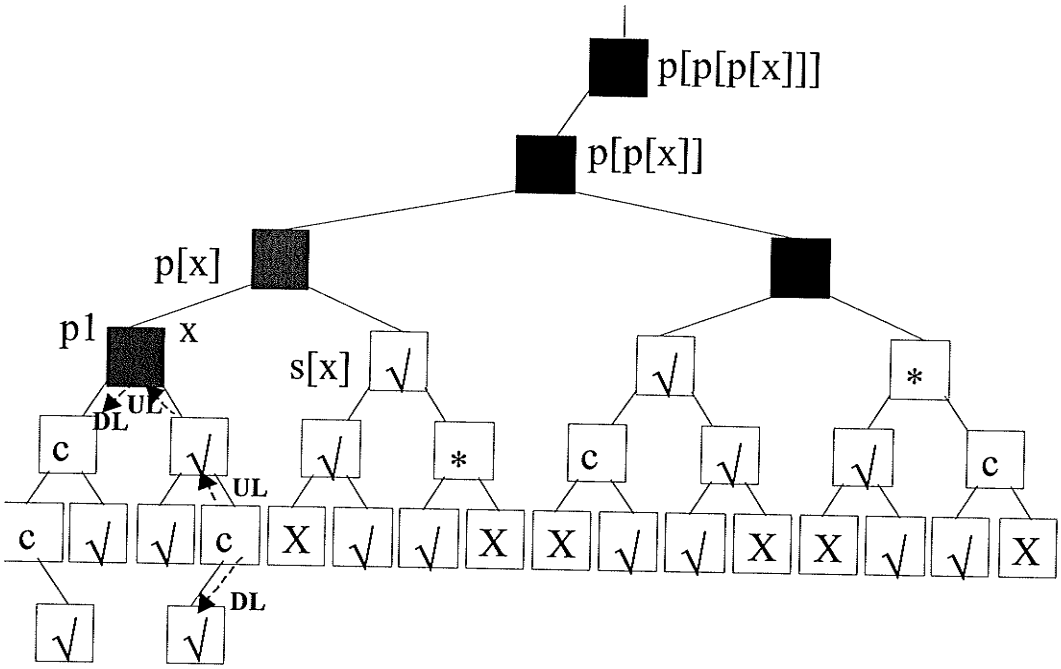

Figure 3-62–P1 in Case 3, p2 in Case 3: first placement of p1, second access pattern.

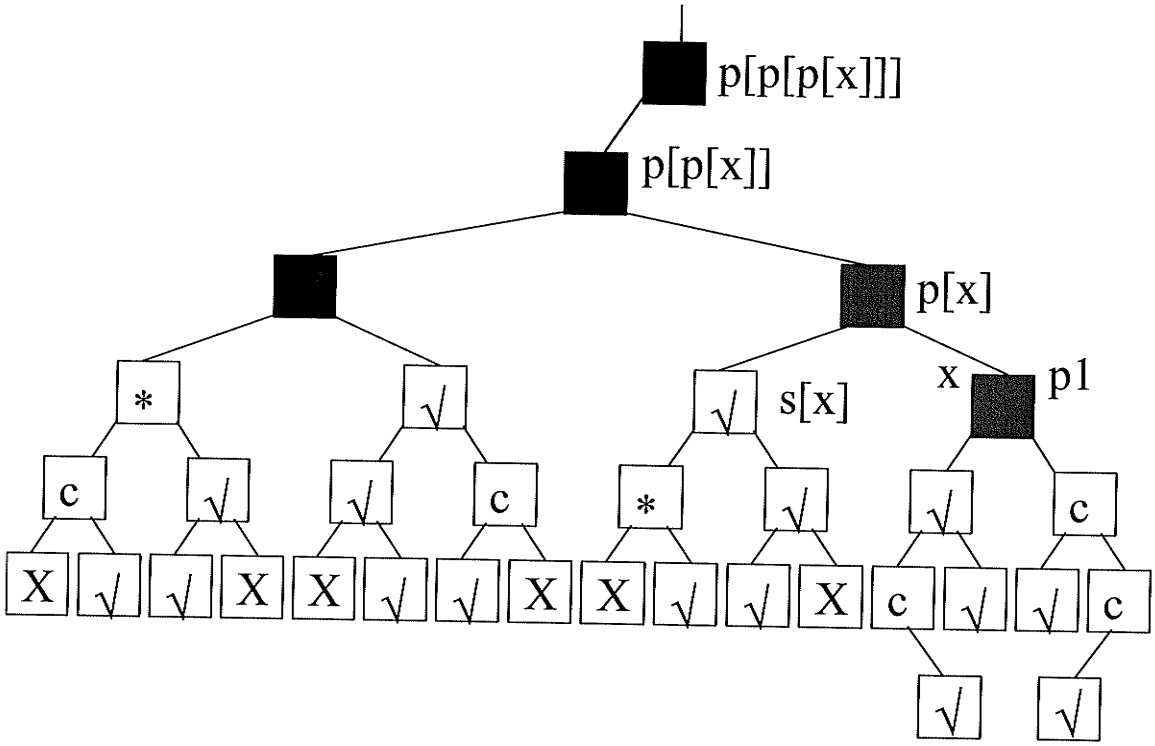

Figure 3-63–P1 in Case 3 and p2 in Case 3: second placement of p1.

When Process P1 is in Case 1 and Process P2 is searching down the Tree

When process p1 in Case 1, it only does re-colouring in the local area of the tree, and it does not change the structure of the local area. Thus, when another process, p2, is searching down the tree to find its insertion point, there is no conflict between p1 and p2.

When Process P1 is in Case 2 and Process P2 is searching down the Tree

As shown in Figure 3-64, process p1 is in Case 2 and another process, p2, is searching down the tree to find its insertion point. The shape of the subtree where p1 operates is "zigzag" (as shown in Figure 3-20). Due to the colour constraints mentioned above, p1's x is red, its parent is red, and its uncle is black. Due to access constraints, p1 will operate on the node set {x, p[x], p[p[x]], p[p[p[x]]], and left[x]}. If process p2 locates on the positions marked 'c' in Figure 3-64, there is a conflict between p1 and p2, because the node where p2 is located is in the node set of p1. There is only one possible access pattern for p1: DR-UR-UL-DL.

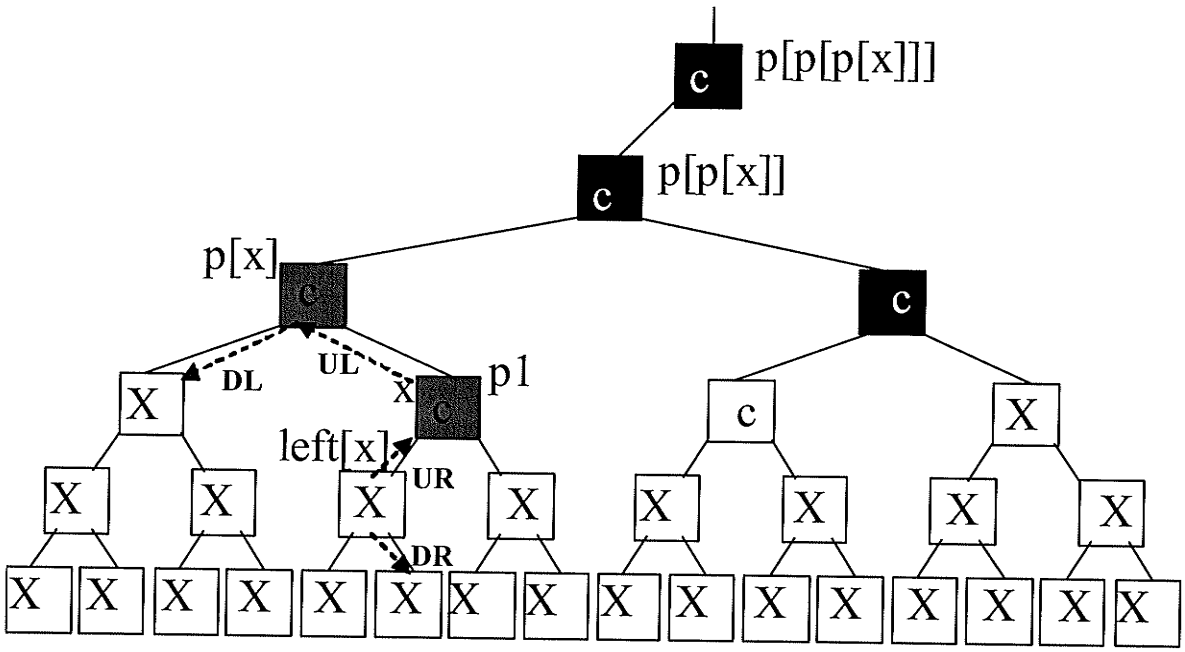

Figure 3-64–P1 in Case 2 and p2 is searching down the tree.

When Process P1 is in Case 3 and Process P2 is searching down the Tree

As shown in Figure 3-65, Process p1 is in Case 3 and another process, p2, is searching down the tree to find its insertion point. The shape of the subtree where p1 operates is "straight" (as shown in Figure 3-21). Due to the colour constraints mentioned above, p1's x is red, its parent is red, and its uncle is black. Again due to access constraints, p1 will operate on the node set {x, p[x], p[p[x]], p[p[p[x]]], and s[x]}. If process p2 locates on any of the positions marked 'c' in Figure 3-65, there is a conflict between p1 and p2, because the node where p2 is located is in the node set of p1. There is only one possible access pattern for p1: DR-UR-UR-DR.

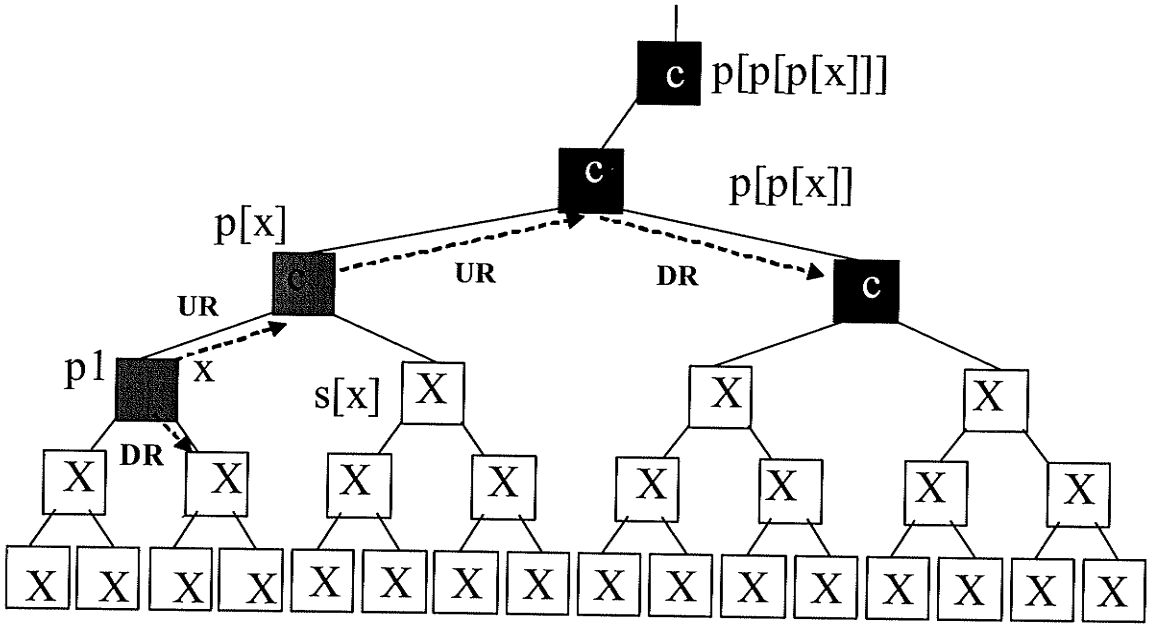

Figure 3-65–P1 in Case 3 and p2 is searching down the tree.

4 Performing Concurrent Insertions

Any correct concurrent algorithm for inserting into a red-black tree must provide the same results as a correct nonconcurrent algorithm.[6] Incorrect concurrency therefore occurs when concurrent insertions transform the red-black tree into a state different from what it would be if those same insertions had been performed sequentially in some order. Any potential solution to controlling concurrent insertions in red-black trees must ensure equivalence to a sequence of serial operations. There are many possible ways in which this sequencing may be done.

4.1 Concurrent Insertions

Initially, general ideas for how concurrent insert operations might be controlled are discussed. The detailed sub-operations that must be done to accomplish an insertion (traversal, re-colouring, and rotations) are discussed later. In general, concurrent insertions could potentially be correctly implemented using any one of the following *candidate* approaches. The approach proposed in this thesis and discussed in detail in the following chapter is presented in Section 4.3.

4.1.1 Simple Locking

A lock on the entire tree could be provided, which each process has to acquire before performing an insertion operation. This approach is simple and guarantees equivalence to a sequence of serial operations (occurring in the order in which the processes acquire the lock on the red-black tree). Such an approach is, however, highly undesirable. It provides no concurrency of updates to the red-black tree since only one inserting process may hold the lock at a time. Further, this technique has all the problems associated with lock-based concurrency control described earlier in this thesis. A technique that does not require locking and which provides concurrency control at a much finer level of granularity than the

[6] This forms the basis for a correctness criterion for concurrent implementations and is a re-statement of the concept of "serializability" from database theory.

entire tree is clearly needed. Information from Section 3.3 will be used to develop such a technique.

4.1.2 A Naïve Lock-Free Method

The general approach that will be used to overcome the problems associated with concurrent insertions is to use lock-free primitives (based on CASn) to allow only one of the concurrent processes to successfully complete its operation. Other processes trying to concurrently insert nodes whose colours or pointers have changed since they were last read (due to the operation of other concurrent processes) must fail. In a naïve implementation, these processes would be required to re-read their pointers and retry their operations. Thus, a process that fails to insert its node will have to call the tree traversal algorithm (TRAVERSE) again to find a new location for the node to be inserted at and then begin the insertion all over again. Given appropriate CAS primitives, this approach would certainly work and provides finer granularity concurrency control (since insertions occurring in different parts of the tree would not interfere with one another) and therefore improved concurrency. Unfortunately, the cost of redoing the entire traversal and insertion is high. This cost would have a negative effect on overall performance if the insertion rate on the tree is high and particularly if the insertions tend to be localized within the tree.

4.1.3 A Lock-Free Method with Operation Combining

A potentially better method to handle lock-free insertion might be to add Prakash's [7] concept of "operation combining". The basic idea of this technique would be that processes "announce" their intention to perform an insertion in a globally shared data structure before beginning the operation. If the inserting process can complete its operation it does so and removes the description of its operation from the shared structure. If it cannot complete the operation, it simply records the "state" of the operation (up to the point where it would block) and exits. In this approach a later process, before performing its own operation, must perform any incomplete operations left by other "incomplete" processes. By using this approach the high cost of re-executing failed insertion operations can be avoided.

The price paid, however, is significantly increased complexity in the implementation. It would be nice if this complexity could be avoided without incurring the extra cost of re-executing failed insertions.

4.2 Concurrent Rotations and Colour Updates

To determine a better lock-free red-black insertion technique, the specific problems associated with the sub-operations of concurrent insertion (i.e. concurrent rotations, colour updates, and tree traversals) must be considered. Insight gained in this process will be used to develop the algorithm that is presented at the end of this chapter. In general, the problems with concurrent sub-operations can potentially be solved by using one of the following approaches.

4.2.1 Naïve Application of CASn

Using CAS-type primitives to atomically update the necessary colour fields and/or pointers in a tree node, it is possible to ensure that only one of a number of concurrent processes operating on that node is allowed to successfully complete its individual operation. Other processes trying to concurrently update node colours or perform rotation operations on nodes whose colours or pointers have changed since they were last read will fail. Unfortunately, this sort of naïve application of CASn is insufficient to guarantee overall correctness. The problem is illustrated by considering the re-colouring of a tree after two processes, `p1` and `p2`, have inserted new nodes. The original tree is shown in Figure 4-1.

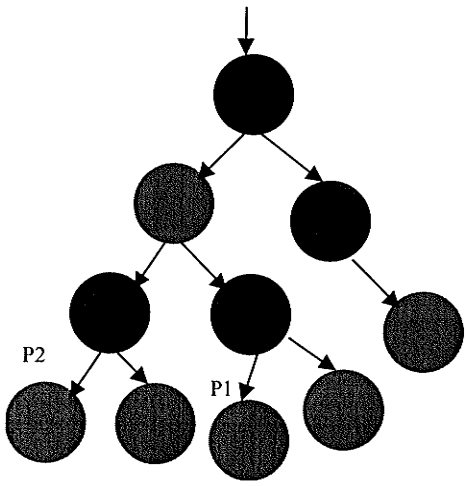

Figure 4-1 –Original tree before concurrent insertion.

Suppose that a process, p1, inserts a new node A and begins "backing up" the tree updating colours as shown in Figure 4-2. Another process, p2, then inserts another new node B and also starts backing up updating colour fields until it finds its rotation point. When p2 proceeds past p1 going up the tree and does its higher-level colour-updates and rotation operations before p1, a dangerous situation occurs. The local area where p2 must do its operations (including the node p2, its parent, grandparent and uncle) partly overlaps the local area where p1 must do its operations. Both processes are in Case 3 of the serial algorithm (shown in Figure 3-9). The node x is the parent of node y, and the parent of the node x is the grandparent of the node y, and the uncle of y is the sibling of x as shown in Figure 4-3. The colours of x, y and the parent of x (which is also the grandparent of y) are red. If p2 is faster than p1 and passes p1 going up the tree and does its higher-level colour changes and rotations, p2 may change the colour of some nodes that are in the local area of p1. Worse, p2 may change the structure of the local area of p1. When p2 completes its re-colouring, p1 may be in a different local area than before p2 passed it. When p1 completes its re-colouring, the tree will not be balanced as shown in Figure 4-4.

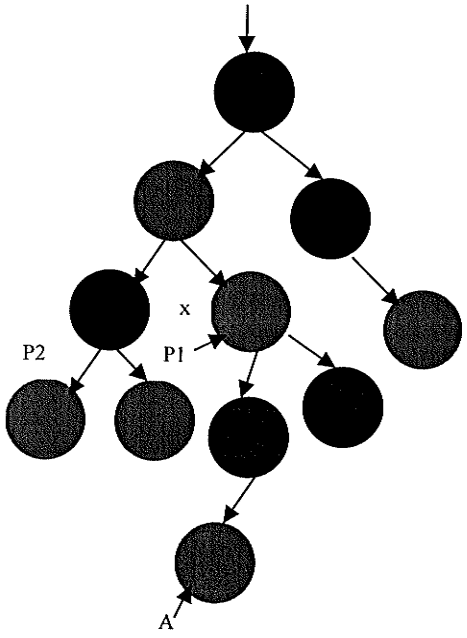

Figure 4-2–P1 inserts node A and starts backing up the tree.

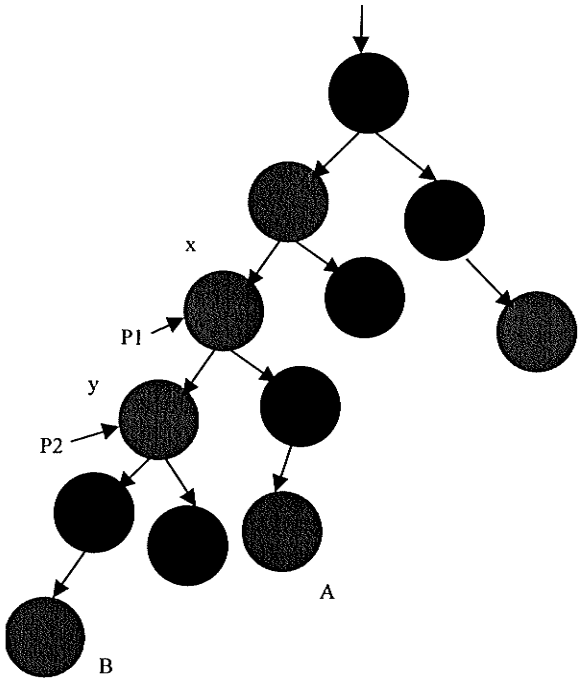

Figure 4-3–P2 inserts node B and starts backing up the tree.

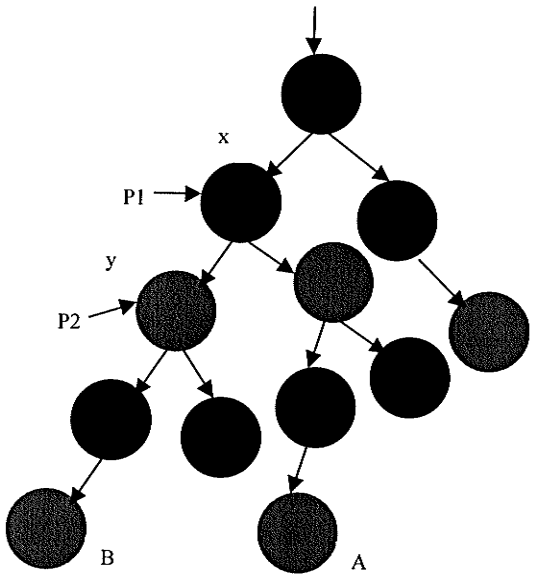

Figure 4-4–Process p2 proceeds past p1 going up the tree.

Concurrent processes clearly must be prevented from "passing" each other to ensure that the resulting tree will be equivalent to one produced by serial operations on the tree. Furthermore, this is a general problem not restricted to

insertions alone. The same general approach can likely be used to handle concurrency in deletion operations though it will not be discussed in this thesis.

4.2.2 Careful Use of CASn

The passing problem can be overcome using the following approach. When some process p2 tries to proceed past another process p1 going up the tree, p2 must first complete the work of p1 *at that node*. Afterwards, p2 can do its own work at the node. In this way, p2 and p1 could essentially alternate their steps up the tree, first p1 then p2. For this approach to work, each process must somehow be aware of where in the tree other processes are operating. This awareness can be created by having each process set a flag in the tree node it is visiting. Again, CAS type operations can be used to effect the necessary local changes in the tree and, now, can also be used to test and set the flag as required. Unfortunately, there are some other problems with this method. Figure 4-5 shows a scenario where two processes are re-colouring tree nodes after their respective insertions. When p2 tries to proceed past p1, it first wants to finish the work of p1. If, however, a third process, p3, attempts to proceed past p2 going up the tree it is required to finish the work of p2. This problem extends to many processes so the algorithm quickly becomes complex and impractical. It is better to simply prevent passing.

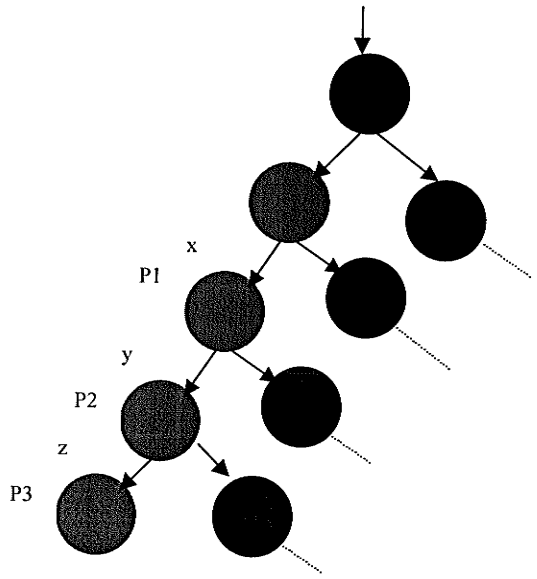

Figure 4-5–Sample tree of possible method 2.

4.2.3 Operation Combining

Another possible approach to overcoming the passing problem is by *combining* operations. Rather than attempting to complete another process' work, the work of the two processes might be combined. Given that there may be many processes operating at the same location in the tree, however, this approach would suffer from the same complexity as the one presented in Section 4.2.2.

4.3 Selected Solution Strategy

4.3.1 Concurrent Insertions

The main drawback of the naïve use of CAS to handle concurrent insertions is that some processes will have to restart their entire operations again. The approach that uses operation completion also has disadvantages. The latency of each of the processes increases because of the time required to perform the operations of the other processes. Apart from this, the approach may also result in wasted parallelism since processes can only read concurrently while updates are done in a serial manner. One of the design goals for this thesis is to maximize the concurrency of operations on the tree. Moreover, it is very difficult to implement operation completion. The use of CAS is therefore adopted to avoid the problems associated with concurrent insertions but this the use of CAS is done carefully and at a finer level of granularity to improve concurrency.

4.3.2 Concurrent Rotations and Colour-updates

As discussed previously, it is necessary to ensure that concurrent processes do not "pass" one another while backing up the tree re-colouring and, possibly, doing rotation operations. Considering this and the design goal of maximising the concurrency of operations on the tree, we adopt the following approach, which uses a flag in each node (as suggested in Section 4.2.2) to prevent passing.

The general idea behind the approach is to set a flag field for each node of the tree that indicates whether or not some process is potentially manipulating the node. The flag field for each node is initialized to FALSE (which means that no process is accessing the node). Setting the flag field at a given node "protects" a local area around the node that, for insertion, includes the node itself, its parent,

uncle and grandparent. This protection is accomplished by having processes check the flags at *all* nodes their operation may affect. When a process arrives at some node, it checks if the flag of any node it may modify is `TRUE`. If so, the process must wait until the flags of all local such nodes become `FALSE`, meaning that no other process is operating in that local area. This processing prevents processes from passing one another!

Once all necessary flags are found to be `FALSE`, the process will set the flags of the nodes it will operate on to `TRUE`. Only then can it safely release the flags of the previous nodes it held, which will allow a waiting process to advance. Once all this is done, the process advances to the new node to perform its necessary operation(s). This processing continues until the process arrives at the root or exits having completed all of its processing. Using this approach, some processes may be delayed but the duration of the delay should be small since processes are executing in parallel and should complete operations in a local area of the tree quite quickly.

5 The Algorithm and Its Correctness

5.1 The Implementation of Red-Black Tree Nodes

Each node in the red-black tree consists of six fields: a data field, a colour field, a flag field, and three pointer fields. The structure of a node is shown in Figure 5-1. The three pointer fields point to the left child of the node, to the right child of the node, and to the parent of the node. The flag field is used to indicate that some process is currently manipulating the node.

If a child or the parent of a node does not exist, the corresponding pointer field of the node is defined to have the value `NIL`. It is also assumed that the `NIL` node points to itself in all cases. In the implementation of the red-black tree concurrent insertion algorithm, a sentinel node "nil[T]" is used to represent `NIL`. This sentinel simplifies the boundary conditions in the code by allowing us to treat `NILs` as if they were nodes of the red-black tree. The sentinel nil[T] has the same fields as an ordinary node in the tree, its colour field is `black`, its data and flag fields can be set to arbitrary values, and all pointers to `NIL` are replaced by pointers to itself. Note that in the following diagrams, the black sentinel nodes have been omitted to keep the diagrams simple.

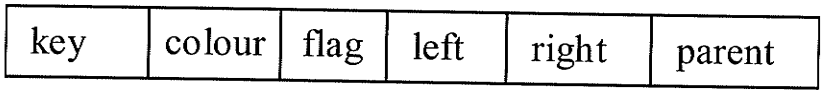

Figure 5-1–Structure of a red-black tree node.

5.2 General Algorithm Description

An initial function `CreateTree()` (Figure 5-4) described in Section 5.4 is used in advance to create an empty red-black tree prior to any insertion operations.

Figure 5-2 shows a function call tree of the routines used to implement the concurrent red-black insertion algorithm. The function `Insert(key)` (Figure 5-6) implements the red-black tree insertion and is called for each node to be added. It first calls the function `Traverse(key)` (Figure 5-7) to search for the

insertion point, and then calls the function `PlaceNode(newNode, insertPoint)` (Figure 5-8) to add a new node at the appropriate insertion point and then finally calls the function `Insert-Rebalance(x)` (Figure 5-9) to adjust the resulting tree so that it again satisfies the red-black properties.

The function `Insert-Rebalance(x)` in turn calls the function `Update-Rotation(&x,&caseFlag)` (Figure 5-10) to perform any necessary colour updates and rotation(s). `Update-Rotation(&x,&caseFlag)` implements the concurrent colour update of the red-black nodes in CASE 1 and CASE 3 and calls the function `Left-Rotation(z)` (Figure 5-11) or `Right-Rotation(x)` as needed. The functions `Left-Rotation(z)` and `Right-Rotation(x)` implement the left and right rotation, respectively.

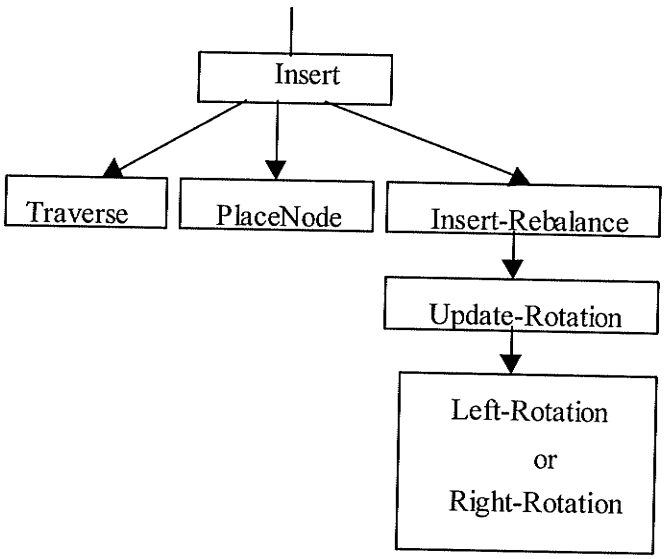

Figure 5-2–The function call tree.

Each of the routines just described acquires flags on all the nodes that it may modify to guard against conflicts with other concurrent processes. These flags are also held between the different routines as required, to prevent incorrect executions caused by another process changing the tree between the routines.

The CASn primitives (including DCAS) are used for two purposes: to acquire flags when needed and to change the structure of the tree by changing pointer values in tree nodes. Ideally, a single "high-degree" CASn operation (one where the value of 'n' exceeds two) would be used to make all the necessary changes (both acquiring flags and changing pointers) at the same time. This high degree is

undesirable, however, since existing machines typically only provide support for CAS and DCAS in hardware. Thus, sequences of "low-degree" CAS and DCAS operations are used instead wherever possible. These sequences must be generated carefully to ensure that their effect is the same as a single, higher-degree CASn operation.

Another complicating factor in the code presented is that pointer dereferences used in CASn operations (e.g. `CAS(oldval,newval,ptr→variable)`) are potentially dangerous and must therefore, generally, be avoided. This complication arises because the dereferencing will occur before the CASn is executed (since the compiler will generate the sequence of dereferencing operations prior to the CAS or DCAS instruction). As a result, if another process changes the structure of the tree between the dereferencing and the CAS, the pointers used may be incorrect. Thus, it is necessary to check that pointers have not been changed in each CAS or DCAS operation unless the current process holds the flag on the node containing the pointers used in the dereferencing (in which case, they cannot be changed by another process). In the algorithm, we explicitly add code to make a copy of a pointer to a node before the CASn. The pointer dereferencing added by the compiler due to references in the CASn is added between our explicit copying of the pointer and the CASn. Thus, our explicit checking in the CASn that the pointer still points to the correct node also checks that the dereferencing performed by the compiler refers to the correct node.

An example of this complication is as shown in Figure 5-7, the pointer variable `savert` is used to save a copy of the root (line 7). When a process calls DCAS operation (lines 8 to 9), the pointer dereferencing is added between `savert` and the DCAS. Thus, the process must check that the pointer variable `savert` has not been changed between the dereferencing and the DCAS by other processes to guarantee its correctness.

Whenever a CASn fails, the code is designed so that the minimum possible amount of code is re-executed before the CASn is re-tried. This design maximizes for efficiency.

5.3 Global Variable Declarations

The global variables defined in Figure 5-3 point to specific tree nodes of interest. These nodes include the root node of the tree, the single NIL node, the parent of the root and its sibling. These variables are referred to in the concurrent insertion code and are initialised in the function CreateTree() (shown in Figure 5-4).

```
// global variables pointing to specific tree nodes
node *root;           // ptr to root node              [1]
node *NIL;            // ptr to NIL node               [2]
node *rtParent;       // ptr to root's parent          [3]
node *rtSibling;      // ptr to root's sibling         [4]
```

Figure 5-3–Global Tree Node Declarations.

5.4 Creating an Empty Red-Black Tree

The function CreateTree() shown in Figure 5-4 is used to create and initialize an empty red-black tree with dummy root nodes. It must be called before any operation is performed on the tree. An empty red-black tree consists of a single NIL node (the root) and dummy parent and sibling nodes. The dummy nodes above the root are used to guarantee that the tree has a parent and sibling for the real root in all cases even when there is little or nothing in the tree. These extra nodes simplify the code by eliminating the need to check for boundary conditions when inserting at or near the root.

The process first creates the dummy parent, sibling and NIL nodes (lines 6 to 8) and then initializes the pointer, flag and colour fields of these nodes (lines 9 to 22). Each dummy node flag field is set to FALSE (line 15, 16, 21) to prevent any potential deadlock.

```
CreateTree();
// This routine is non-parallel. It is called          [1]
// to create an empty tree (with dummy parent,         [2]
// and sibling nodes before any operations are done.   [3]
// This includes the creation of the NIL node.         [4]
                                                       [5]
rtParent = new node();                                 [6]
```

```
rtSibling = new node();                    [7]
NIL = new node();                          [8]
root=NIL;                                  [9]
rtParent→parent=NIL;                       [10]
rtSibling→parent=rtParent;                 [11]
rtSibling→right=NIL;                       [12]
rtSibling→left=NIL;                        [13]
rtParent→left=root;                        [14]
rtParent→right=rtSibling;                  [15]
rtParent→flag=FALSE;                       [16]
rtSibling→flag=FALSE;                      [17]
rtParent→colour=black;                     [18]
rtSibling→colour=black;                    [19]
NIL→left=NIL;                              [20]
NIL→right=NIL;                             [21]
NIL→parent=rtParent;                       [22]
NIL→flag=FALSE;                            [23]
NIL→colour=black;                          [24]
```

Figure 5-4–The algorithm CreateTree()

The structure resulting from the creation of an empty red-black tree is shown in Figure 5-5.

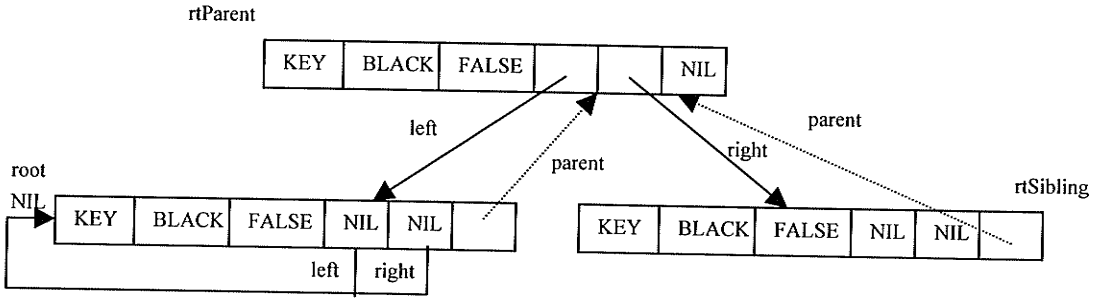

Figure 5-5–Dummy Nodes in an Empty Red-Black Tree

5.5 Concurrent Insertion

The algorithm for concurrent insertion into a red-black tree is shown in Figure 5-6. The function `Insert(key)` calls the function `Traverse (key)`, the function `PlaceNode(newNode,insertPoint)` and the function `Insert-Rebalance(x)` in that order. These functions implement an ordinary binary

search tree insertion of a new node and then a re-balancing of the resulting tree after the insertion of the new node.

An inserting process first allocates and initializes a new node (lines 6 to 12). It then calls the function `Traverse(key)` (shown in Figure 5-7) to find the correct insertion point (line 17). It then calls the function `PlaceNode` (shown in Figure 5-8) to place the new node in the tree at the insertion point (line 19). The process then calls the function `Insert-Rebalance(x)` (shown in Figure 5-9) to complete any necessary re-colouring and restructuring of the tree after the insertion (line 24). If the insertion on line 19 fails, the process will return to line 17 to find a new insertion point and try again until the insertion succeeds or the return value of the variable `result` is `FirstInsert` (in which case the new node is being inserted into an empty tree). This looping is implemented using the `repeat...until` statement on lines 15 to 26. If the new node is being inserted into an empty tree, the process does not need to call the function `Insert-Rebalance(x)` to re-color the tree, because the color of the newly inserted node is set to `black` (shown in Figure 5-8), the tree is balanced already.

A process may fail to insert a new node due to another insertion being performed concurrently by another process. Specifically, when a process finds the insertion point for a given `key` value and tries to insert the new node, a second process may try to insert another new node at that same point or perform a rotation at the insertion point. If the second process is successful, the first process must fail and then search again from the root to find the new insertion point for the given `key` value since the key values in the tree may have been changed. This procedure must be repeated until the process inserts the new node successfully.

```
Insert(key);
node *newNode;                                              [1]
node *insertPoint;                                          [2]
enum result {Success,Failure,FirstInsert};                  [3]
                                                            [4]
// Create and initialize the new node                       [5]
newNode = new node();                                       [6]
newNode→parent = NIL;                                       [7]
newNode→left = NIL;                                         [8]
```

```
newNode→right = NIL;                                                    [9]
newNode→colour = RED;                                                   [10]
newNode→key = key;                                                      [11]
newNode→flag = TRUE;                                                    [12]
                                                                        [13]
// insert the new node                                                  [14]
repeat {                                                                [15]
   // Traverse tree to find insertion point                             [16]
   insertPoint=Traverse(key);                                           [17]
   // add new node to tree                                              [18]
   result=PlaceNode(newNode,insertPoint);                               [19]
   if (result==Success){                                                [20]
      // node was added successfully so make                            [21]
      // tree red-black again by doing the                              [22]
      // necessary colour updates and rotations                         [23]
      Insert-Rebalance(newNode);                                        [24]
   }                                                                    [25]
} until ((result==Success)||(result==FirstInsert));                     [26]
```

Figure 5-6–The algorithm Insert(key).

5.5.1 Tree Traversal

The function `Traverse(key)` shown in Figure 5-7 is used to traverse a red-black tree to find the insertion point for a new node with the given `key` value. As each node is being visited during the traversal down the tree, the node's flag is held by the traversing process. This flag will ensure that no other concurrent process can interfere with the searching process. The synchronization primitive `DCAS` is used to safely acquire the flag on the next node in the traversal. This acquiring is done using DCAS to set the next node's flag to TRUE if and only if the pointer to the next node has not been changed (at lines 8-9, 22-23, and 36-37). The pointer variable `savert` is explained in section 5.2.

In `Traverse(key)`, the process first uses the `repeat...until` loop and `DCAS` operation to set the flag at the root where the traversal begins (lines 6 to 10). The process then uses the `while` loop (at lines 13 to 48) to find the insertion point. It begins searching for the insertion point from the root of the red-black tree. If the root node points to `NIL` (the tree is empty), the process does not enter the `while` loop and simply returns `NIL` (since the variable `insertPoint` is initially set to `NIL` (line 2)) to its caller, `Insert(key)` (line 49). If the variable x, which initially points to the root (line 12), is not `NIL` (the tree is not empty and

the process is not at a leaf node of the tree), and the process finds the given `key` value, it first releases the flag at `x` and then exits on an error since we cannot insert a node that already exists in the tree (lines 14 to 17).

```
Traverse(key);
node    *x;                  // traversal pointer          [1]
node    *insertPoint=NIL; // initially no insert point[2]
node    *savert, *saverc, *savelc;                        [3]
bool    success;                                          [4]
                                                          [5]
repeat {     // get flag on root to start                 [6]
   savert=root; // save a copy of the root                [7]
   success=DCAS(FALSE,TRUE,root→flag,                     [8]
                savert,savert,root);                      [9]
} until (success);                                        [10]
                                                          [11]
x = root;  // search for insertion point from root        [12]
while (x != NIL) {                                        [13]
   if (key == x→key) {                                    [14]
      x→flag = FALSE; // release flag                     [15]
      exit(Error_KeyExists);                              [16]
   }                                                      [17]
   insertPoint = x;                                       [18]
   if (key < x→key) { // follow left child pointer        [19]
      if (x→left!=NIL) { // get flag on left child        [20]
         success=CAS(FALSE,TRUE,x→left);                  [21]
         if (!success) {                                  [22]
            x→flag = FALSE; // release flag               [23]
            return Failure;                               [24]
         } // end if at line 22                          [25]
         x=x→left;                                        [26]
         insertPoint→flag = FALSE;                        [27]
      } else {                                            [28]
         x=x→left;                                        [29]
      } // end if at line 20                             [30]
   } else {  // follow right child pointer                [31]
      if (x→right!= NIL){ // get flag on right child      [32]
         success=CAS(FALSE,TRUE,x→right);                 [33]
         if (!success) {                                  [34]
            x→flag = FALSE; // release flag               [35]
            return Failure;                               [36]
         } // end if at line 34                          [37]
         x=x→right;                                       [38]
         insertPoint→flag = FALSE;                        [39]
      } else {                                            [40]
         x=x→right;                                       [41]
      } // end if at line 32                             [42]
   } // end if at line 19                                [43]
} // end while at line 13                                [44]
```

```
return insertPoint;                                                    [45]
```

Figure 5-7–The algorithm – Traverse(KEY).

At this point, the process has not found the given `key` value, so it sets the variable `insertPoint` to point to `x` (since it may insert under this node) and then checks if the given `key` is smaller than the key in `x` (lines 19 to 30). If the given `key` is smaller than the key in `x` and the left child of `x` is not `NIL` (line 20), the process uses `CAS` to attempt to acquire the flag of the left child of `x`. If the `CAS` operation fails, it is because another process already holds the flag of the left child of x, so the process must release the flag it holds on `x` and return `Failure` to its calling function `Insert(key)` (lines 22 to 25) so that it can traverse again to find a new insert point. If the `CAS` operation succeeds, the process can safely move down to the left child of x (line 26) and then release the flag at `insertPoint` (line 27). If the left child of x is `NIL` (line 20), the insertion point has been found. The process does not need to set the flag of the left child of x but instead moves directly down to the left child (line 29) which has the effect of setting `insertPoint` correctly. When `x` becomes `NIL`, the process will exit from the `while` loop and return the insert point (`insertPoint`) to the calling function (line 45) which will point to the correct insertion point and the tree traversal is done. If the given `key` is bigger than the key in the node `x` (lines 31 to 43), the processing is symmetric to what was just described – the right child is followed instead of the left child.

On lines 21 and 33, the process uses `CAS` to acquire the flag at the left or right child of `x` as it descends the tree. As described earlier, this acquisition is necessary to prevent interference with other processes doing rotates, etc. If it is impossible to acquire a flag at some point (the `CAS` operation fails), the process must release the flag it holds at `x` before it returns to allow any processes waiting for the flag to continue execution.

Note that on successful completion, the flag at the insertion point continues to be held by the process for later use by the `PlaceNode` routine

5.5.2 Placing a New Node in the Tree

The function `PlaceNode(newNode,insertPoint)` shown in Figure 5-8 is used to place a new node in the tree at the identified insertion point. `CAS` and `DCAS` instructions are used to acquire tree flags and change ("swing") pointers as needed. First, an `if` statement is used to check if the tree is empty (line 5) which requires special-case processing. If the tree is empty, the new node becomes the root of the tree (lines 5 to 12) and the insertion is done (more detail on this processing is provided below). If the tree is not empty, the process acquires the flags of the grandparent and uncle of the new node (lines 16 to 50) before inserting the new node into the tree (lines 55 to 63) (again, more detail is provided below).

When the insert point is `NIL` (i.e. the tree is empty) (lines 5 to 12), the new node is inserted as the root, `rtParent` is then set to be the parent of the new root and the new root becomes its left child (lines 7 to 9). Because the flag at the root (`NIL`) of an empty tree was acquired in the function `Traverse(key)`, this is safe. Afterwards, the process releases the flags at `NIL` and `newNode` (which were set in `Insert(key))` and returns a status of `FirstInsert` to `Insert(key)`. The new node's colour is also set to black (line 6).

```
PlaceNode(newNode,insertPoint);
// flags on newNode and insertPoint are held          [1]
bool ok;       // got needed flags?                    [2]
node *uncle,*savep;                                    [3]
                                                       [4]
if (insertPoint == NIL) { // tree is empty             [5]
   newNode→colour = black;                             [6]
   newNode→parent=rtParent;                            [7]
   rtParent→left=newNode;                              [8]
   root=newNode;                                       [9]
   NIL→flag=FALSE; // release NIL node                 [10]
   newNode→flag=FALSE;                                 [11]
   return FirstInsert;                                 [12]
} else { // The tree is not empty so ...               [13]
   newNode→parent = insertPoint;                       [14]
   // set the flags of the grandparent & uncle         [15]
   if (insertPoint==insertPoint→parent→left) {         [16]
```

```
      savep=insertPoint→parent; // save parent ptr      [17]
      uncle=savep→right;                                [18]
      ok=CAS(FALSE,TRUE,savep→flag);                    [19]
      if (ok) {                                         [20]
         ok=CAS(FALSE,TRUE,uncle→flag);                 [21]
         if (ok) {                                      [22]
            ok=DCAS(savep,savep,insertPoint→parent,     [23]
                    uncle,uncle,savep→right);           [24]
            if (!ok) { // back off                      [25]
               savep→flag=FALSE;                        [26]
               uncle→flag=FALSE;                        [27]
            }                                           [28]
         } else {                                       [29]
            savep→flag=FALSE;                           [30]
         } // end if at line 22                        [31]
      }                                                 [32]
   } else { // uncle is left child not right            [33]
      savep=insertPoint→parent; // save parent ptr      [34]
      uncle=savep→left;                                 [35]
      ok=CAS(FALSE,TRUE,savep→flag);                    [36]
      if (ok) {                                         [37]
         ok=CAS(FALSE,TRUE,uncle→flag);                 [38]
         if (ok) {                                      [39]
            ok=DCAS(savep,savep,insertPoint→parent,     [40]
                    uncle,uncle,savep→left);            [41]
            if (!ok) { // back off                      [42]
               savep→flag=FALSE;                        [43]
               uncle→flag=FALSE;                        [44]
            }                                           [45]
         } else {                                       [46]
            savep→flag=FALSE;                           [47]
         } // end if at line 39                        [48]
      }                                                 [49]
   } // end if at line 16                              [50]
   if (!ok) {                                           [51]
      newNode→parent=FALSE;                             [52]
      insertPoint→flag=FALSE; // release flag           [53]
      return Failure;           // avoids deadlock      [54]
   }                                                    [55]
   if (newNode→key < insertPoint→key){                  [56]
      /*Insert as left child */                         [57]
      insertPoint→left = newNode;                       [58]
      return Success;                                   [59]
   } else{ /*Insert as right child */                   [60]
      insertPoint→right = newNode;                      [61]
      return Success;                                   [62]
   }                                                    [63]
} // end else at line 13                               [64]
```

Figure 5-8–The algorithm – Placenode(newNode,insertPoint).

If the tree is not empty (line 13), before the process inserts a new node, it must first set the flags at the grandparent and uncle of the new node (the flag at its parent (`insertPoint`) will have already been acquired in `Traverse(key)`). If `insertPoint` is the left child of its parent, then the uncle of the new node is set to be the right child (lines 17 to 18). The process first uses `CAS` to set the flag of the grandparent of the new node (line 19). If the `CAS` operation succeeds, the process uses `CAS` again to set the flag of the uncle of the new node (line 38). If the process fails to set the flag of the uncle of the new node, it must release the flag of the grandparent (line 47) before aborting. If the flags acquired using the CAS operations are both acquired successfully, the process then uses `DCAS` to verify that the grandparent and uncle pointers are unchanged (lines 40 to 41) from what they were when the flags were acquired. If either has changed, the process must release the acquired flags (lines 26 to 27) since they may be held on the wrong nodes. Otherwise, the flags at the grandparent and uncle of the new node have truly been acquired successfully. If any of the `CAS` or `DCAS` operations fails (the value of ok will be set to `FALSE`) and the process will set the parent of the new node to NIL and release the flag at `insertPoint` and return `Failure` to `Insert(key)` (lines 51 to 54) causing it to restart the insertion process.

If the parent of the new node (`insertPoint`) is the right child of its grandparent, then its uncle is set to be the left child, and the processing done is symmetric to that just described with the left and right child pointers exchanged (lines 34 to 49).

Note that if another process is rotating about the grandparent of the insert point then it may be possible that although the test at line 16 succeeds, the insert point is no longer a left child. This is detected at line 21 when the process then fails to obtain a flag on the uncle, which, due to the rotation is now the insert point. This results in the `CAS` failing and the insertion being restarted. Similarly, if the test at line 16 fails, a rotation may cause the insert point to no longer be a right child and this will cause a failure of the `CAS` at line 38 and restarting of the insertion process.

Finally, at lines 57 and 60, the new node is physically inserted as the left (or right) child of the insertion point. This physical insertion is safe because we hold the flags on both `insertPoint` and its parent so that no other process can interfere with the insertion.

Note the flags at the new node, its parent, grandparent and uncle continue to be held by the process so they can be used by the `Insert-rebalance` routine.

5.5.3 Rebalancing the Red-Black Tree After an Insertion

The correct insertion of nodes into red-black trees consists of two parts: the physical insertion of the new node as a leaf node in the tree (which has just been described) and the re-balancing of the tree after this insertion. The re-balancing is necessary since an insertion without re-balancing may result in a tree that violates one or more of the red-black properties. To concurrently re-balance a tree after an insertion, `CAS and DCAS` operations are used to acquire the flags of nodes needed as the process works its way up the tree, re-colouring nodes and/or performing rotate operations. The acquisition of these flags guards against inconsistencies due to processes passing one another.[7]

```
Insert-Rebalance(x):
// we hold flags on x, p(x), p(p(x)) & uncle(x)          [1]
node  *oldx;          // points to previous x             [2]
node  *uncle, *olduncle;                                 [3]
node  *savep, *savegp;                                   [4]
node  *brother;                                          [5]
node  *nephew;                                           [6]
bool  ok;                                                [7]
bool  updateSucceeds; // Update-Rotation succeeded?      [8]
enum  caseFlag {NOOP,DID_CASE1,DID_CASE3};               [9]
                                                         [10]
caseFlag = NOOP; // initially not doing any case         [11]
// define uncle for first iteration                      [12]
if (x→parent==x→parent→parent→left) {                    [13]
   uncle=x→parent→parent→right;                          [14]
} else { // uncle is left child not right                [15]
   uncle=x→parent→parent→left;                           [16]
}                                                        [17]
                                                         [18]
while ((x != root) && (x→parent→colour == RED)) {        [19]
   // do colour-update and/or rotation as required       [20]
   repeat {                                              [21]
```

[7] The problems associated with not guarding nodes during re-balancing were discussed in Chapter 3.

```
   updateSucceeds = Update-Rotation(x,caseFlag);                    [22]
} until (updateSucceeds)                                            [23]
                                                                    [24]
// CASE 1:move to grandparent after colour update                   [25]
if (caseFlag == DID_CASE1) {                                        [26]
   oldx = x;          // save pointer to the old x                 [27]
   olduncle = uncle; // save ptr to old uncle                      [28]
   x =x→parent→parent; // up to grandparent                        [29]
   repeat { // find new uncle of x and get flags                   [30]
      if (x→parent == x→parent→parent→left){                       [31]
         savep=x→parent;                                           [32]
         savegp=savep→parent;                                      [33]
         uncle = savegp→right;                                     [34]
         ok=CAS(FALSE,TRUE,savep→flag);                            [35]
         if (ok) {                                                 [36]
            ok=CAS(FALSE,TRUE,savegp→flag);                        [37]
            if (ok) {                                              [38]
               ok=CAS(FALSE,TRUE,uncle→flag);                      [39]
               if (ok) {                                           [40]
                  ok=DCAS(savep,savep,x→parent,                    [41]
                        savegp,savegp,savep→parent);               [42]
                  if (ok) {                                        [43]
                   ok=CAS(uncle,uncle,savegp→right);               [44]
                  }                                                [45]
                  if (!ok) {                                       [46]
                     savep→flag=FALSE;                             [47]
                     savegp→flag=FALSE;                            [48]
                     uncle→flag=FALSE;                             [49]
                  }                                                [50]
               } else {                                            [51]
                  savep→flag=FALSE;                                [52]
                  savegp→flag=FALSE;                               [53]
               } // end if at line 40                             [54]
            } else {                                               [55]
               savep→flag=FALSE;                                   [56]
            } // end if at line 38                                [57]
         }                                                         [58]
      } else {                                                     [59]
         savep=x→parent;                                           [60]
         savegp=savep→parent;                                      [61]
         uncle = savegp→left;                                      [62]
         ok=CAS(FALSE,TRUE,savep→flag);                            [63]
         if (ok) {                                                 [64]
            ok=CAS(FALSE,TRUE,savegp→flag);                        [65]
            if (ok) {                                              [66]
               ok=CAS(FALSE,TRUE,uncle→flag);                      [67]
               if (ok) {                                           [68]
                  ok=DCAS(savep,savep,x→parent,                    [69]
```

```
                        savegp,savegp,savep->parent);
                    if (ok) {
                      ok=CAS(uncle,uncle,savegp->left);
                    }
                    if (!ok) {
                       savep->flag=FALSE;
                       savegp->flag=FALSE;
                       uncle->flag=FALSE;
                    }
                 } else {
                    savep->flag=FALSE;
                    savegp->flag=FALSE;
                 } // end if at line 68
              } else {
                 savep->flag=FALSE;
              } // end if at line 66
           }
        }
     } until (ok); // repeat at line 30

     // Release old flags for CASE 1 of algorithm
     oldx->parent->flag = FALSE;
     olduncle->flag = FALSE;
     oldx->flag = FALSE;
  }
  // in case 3 loop will exit: parent will be black
} // end while at line 19

switch (caseFlag) {
case NOOP: // release pre-allocated flags
     x->parent->parent->flag=FALSE;
     x->parent->flag=FALSE;
     uncle->flag=FALSE;
     x->flag=FALSE;
     break;
case DID_CASE1:// release last set of old flags
     x->parent->parent->flag = FALSE;
     x->parent->flag = FALSE;
     uncle->flag = FALSE;
     x->flag = FALSE;
     break;
case DID_CASE3: // release flags on ROTATED x, etc.
     if (x == x->parent->left) {
        brother = x->parent->right;
        nephew = x->parent->right->right;
     } else {
        brother = x->parent->left;
        nephew = x->parent->left->left;
```

[70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [111] [112] [113] [114] [115] [116] [117]

```
        }                                                    [118]
        x→parent→flag = FALSE;                               [119]
        brother→flag = FALSE;                                [120]
        nephew→flag = FALSE;                                 [121]
        x→flag = FALSE;                                      [122]
        break;                                               [123]
} // end switch at line 98                                  [124]
root→colour = black;                                         [125]
```

Figure 5-9–The algorithm Insert-Rebalance(x).

In the function `Insert-Rebalance(x)` (Figure 5-9), the code first defines the uncle of x for the first iteration (lines 13 to 17), and then uses a `while` loop to backtrack up the tree from the insertion point towards the root. This is the fundamental processing performed by `Insert-Rebalance(x)`. During this processing, a process first checks that the current node is not the root of the tree and that the colour of its parent is `RED` (line 19). If these conditions are true, the process uses a `repeat` loop to repeatedly call `Update-Rotation` (Figure 5-10) (which either updates node colours in CASE 1 or performs rotation(s) in CASE 2 and/or CASE 3) until it succeeds (lines 21 to 23). Because the flags on x, its parent, grandparent and uncle have already been acquired, the process does not need to explicitly acquire these flags before beginning. If the return value (`caseFlag`) from `Update-Rotation` is `DID_CASE1` (indicating CASE 1 processing was performed), then the process saves the value of `x` in `oldx` (line 27), and `uncle` in `olduncle` (line 28), and then continues backtracking up the tree to `x`'s grandparent (line 29). It uses the `repeat` statement (lines 30 to 88) to acquire the flags around the new local area of the tree where the process is located (this includes the parent of `x`, its grandparent and uncle). To improve the performance of the algorithm, the process calls four `CASs` and one `DCAS` instructions to replace one `CAS6` instruction in the same parameter order as `CAS6` (the parent of x, its grandparent and uncle). To do this safely, the process first saves pointers to the parent of `x`, its grandparent and uncle into the pointer variables `savep`, `savegp`, and `uncle` (lines 32 to 34). It then uses a `CAS` operation to acquire the flag of the parent of `x` (line 35). If the `CAS` operation succeeds, the process uses another `CAS` operation to acquire the flag of the

grandparent of x (line 38). If the process fails to set the flag at the grandparent of x, it releases the flag of the parent (line 56). If the flags of the parent and grandparent of x are both set successfully, the process continues to use another CAS operation to acquire the flag of the uncle of x. If this CAS operation fails, the process releases the flags on the parent and grandparent of x (lines 52 to 53). If all three flags are acquired successfully, the process uses a DCAS operation to verify that the parent and grandparent of x are unchanged (lines 41 to 42). If either the parent or grandparent has changed, the process releases all the acquired flags (lines 47 to 49). If the parent of x and its grandparent were unchanged, the process then checks to see if the uncle of x was unchanged (line 44). If the uncle of x has changed, the process again must release all the flags (lines 47 to 49). Otherwise, the flags at the parent of x, its grandparent and uncle have all been successfully acquired. (Note that it is safe to check the pointer to the uncle separately because the preceding DCAS has already verified that the pointer to the grandparent, which is used in the check, is unchanged).

If the parent of x is the right child of its grandparent (instead of the left), then its uncle is the left child of its grandparent, and the processing done (lines 60 to 86) is symmetric to that which was just described (lines 32 to 58).

If any of CAS and DCAS instructions fails due to other processes doing rotates in the same area of the tree, the `repeat` statement will continue to re-execute these instructions until they succeed. Since the process was in CASE 1 of the serial algorithm, once the process succeeds in allocating the new flags at the grandparent, it releases the flags in its previous local region (`oldx`, its parent and `olduncle`) (lines 91 to 93).

CASE 1 processing continues until the `while` loop condition becomes FALSE which indicates that the process has finished its re-colouring. The process is then either at the root of the tree or the colour of the parent of x is no longer red (indicating that backtracking up the tree is now complete, possibly following any necessary rotation(s) which would have finished with CASE 3 processing). At this point, the process needs to release any flags that it still holds. It first checks the value of `caseFlag` to determine which case was performed and, hence, which

set of flags it must release. Releasing the flags is done by the `switch` statement at lines 98 to 124. If `caseFlag` is `NOOP`[8], the process never executed the body of the `while` loop so it simply releases the flags on `x`, its parent, grandparent and uncle (lines 100 to 103) which were held when `Insert-Rebalance` was invoked. If `caseFlag` is `DID_CASE1`, the process finished with a CASE 1, so it releases the flags of `x`, its parent, grandparent and uncle (lines 106 to 109). If `caseFlag` is `DID_CASE3` (the process returns successfully from CASE 3 and possibly, a previous CASE 2), the process must release the flags of `x`, its parent, brother and nephew (lines 119 to 122).

5.5.4 Concurrent Local Colour Update and/or Rotation(s)

The routine `Update-Rotation(&x, &caseFlag)` shown in Figure 5-10 performs the CASE 1, 2 and 3 processing from the serial red-black tree insertion algorithm described in Cormen, Leiserson and Rivest [5]. This processing is accomplished with the help of separate routines for performing the rotations. Again, `CASn` primitives are used to ensure consistency of the tree and correctness of the concurrent implementation.

In CASE 1 of the serial algorithm (lines 9 to 15), the process updates the colours of the nodes as needed (line 11 to 13). It then sets the parameter `caseFlag` to `DID_CASE1` (line 14) and returns `TRUE` (line 15) to `Insert-Rebalance(x)` to indicate that it has successfully done the re-colouring.

In CASE 2 (lines 17 to 25), the process saves `x` to the variable `oldx` (line 18), `x` is made to point to its parent (line 19), and then the function `Left-Rotation(x)` is called to do the needed left-rotation at `x` (line 20).[9] If the left-rotation succeeds, the process enters CASE 3 of the serial algorithm. Otherwise, the process resets `x` to point to `oldx` and returns `FALSE` to its calling function `Insert-Rebalance(x)` (lines 21 to 24) which will re-invoke `Update-`

[8] The variable `caseFlag` stores the return value from `Update-Rotation()`. `caseFlag` is initialized to `NOOP` to detect the case when insertion does not cause any red-black property violations so no processing will be done in `Insert-Rebalance()`.

[9] The symmetric case (where a right rotation would be needed) is obvious and therefore not shown but would be provided in the else clause at line 46 in the algorithm.

`Rotation(&x,&caseFlag)` to attempt the left-rotation again. Because the flags at x, its parent, grandparent and uncle are all still held, other processes cannot change the structure of the nodes between the calls. Other processes can, however, change the left child of x since no flag is held for it. Thus, the rotation can, in fact, fail and need to be redone (as just described).

Update-Rotation(&x,&caseFlag):

```
// we hold flags on x, p(x), p(p(x)) & uncle(x)                    [1]
node  *xUncle;                                                     [2]
bool  OK;                                                          [3]
node  *oldx, *ggp;    // ggp is great-grandparent                  [4]
                                                                   [5]
if (x→parent == x→parent→parent→left) {                            [6]
   // the parent is a left child                                   [7]
   xUncle = x→parent→parent→right;                                 [8]
   if (xUncle→colour == RED) {                                     [9]
      // CASE 1 in [CLR] - re-colouring                           [10]
      x→parent→colour = BLACK;                                    [11]
      xUncle→colour = BLACK;                                      [12]
      x→parent→parent→colour = RED;                               [13]
      caseFlag = DID_CASE1;                                       [14]
      return TRUE;                                                [15]
   } else { // rotation(s) will be needed                         [16]
      if (x == x→parent→right) { // CASE 2 in [CLR]               [17]
         oldx=x; // save old x in case rotate fails               [18]
         x = x→parent;                                           [19]
         OK = Left-Rotate(x);                                    [20]
         if (!OK) {                                              [21]
            x=oldx;  // undo change to x                         [22]
            return FALSE;                                        [23]
         }                                                       [24]
      }                                                          [25]
      // In CASE 3, if the right-rotation fails,                 [26]
      // CASE 3 fails but the algorithm still works              [27]
      // because the process will return FALSE to                [28]
      // Insert-Rebalance, and Insert-Rebalance will             [29]
      // call Update-Rotation again to complete CASE3            [30]
      repeat { // get great grandparent's flag                   [31]
         ggp=x→parent→parent→parent;                             [32]
         OK=DCAS(FALSE,TRUE,ggp→flag,                            [33]
                 ggp,ggp,x→parent→parent→parent);                [34]
      } until (OK);                                              [35]
      OK = Right-Rotate(x→parent→parent);                        [36]
      if(!OK) {                                                  [37]
         ggp→flag = FALSE;                                       [38]
         return FALSE;                                           [39]
      } else {                                                   [40]
         x→parent→colour = BLACK;                                [41]
```

```
            x→parent→right→colour =RED; // former G.P.   [42]
            caseFlag = DID_CASE3;                        [43]
            ggp→flag = FALSE;                            [44]
            return TRUE;                                 [45]
        }                                                [46]
    }                                                    [47]
} else  // symmetric to then clause at line 7           [48]
```

Figure 5-10 –The algorithm Update-Rotation(&x,&caseFlag).

On lines 31 to 46, the process is doing CASE 3 processing. The process first uses the DCAS in the `repeat` statement to guarantee that the flag at the great-grandparent of `x` is acquired successfully (lines 31 to 35). The process then calls the function `Right-Rotate` to do a right-rotation at the grandparent of `x` (line 36). If the right-rotation succeeds, the process updates the colours of the parent of x and its sibling, sets the variable `caseFlag` to `DID_CASE3`, releases the flag at the great grandparent of x, and then returns `TRUE` to `Insert-Rebalance(x)` (lines 41 to 45). If the right-rotate at the grandparent of `x` fails, the process releases the flag at the great-grandparent of x then returns `FALSE` to `Insert-Rebalance(x)` and, again, `Insert-Rebalance(x)` will re-call `Update-Rotation`. Because the structure of the local area where the process is operating (including x, its parent, grandparent and uncle) cannot change because flags are held, the process will enter CASE 3 immediately when `Update-Rotation()` is re-invoked and will redo the right-rotation.

5.5.5 Concurrent Left Rotation

The algorithm that implements a concurrent left-rotation operation is shown in Figure 5-11. The code for a right-rotation is not shown (it is, of course, symmetric to the left-rotation code). It is in the rotation code that the only high-degree CASn operation is required. A three-argument `Compare & Swap (CAS3)` operation is needed because insufficient flags are held on entry to each rotate routine to guarantee correctness. Specifically, the rotate routines must concurrently change two pointers to effect the rotate at the same time that the validity of one of the pointers is ensured.

In `Left-Rotate`, the process first checks if the node `z` (about which the rotate will occur) is the root of the tree. If `z` is the root of the tree (as judged by

the fact that the node `rtParent` is z's parent) (line 6), then special processing is required.

```
Left-Rotate(z):
// z is the root of the rotation subtree. The locks                [1]
// held at this point are: z, z→parent, & z→right                  [2]
bool  OK;                                                          [3]
node  *zrl, *zr;                                                   [4]
                                                                   [5]
if (z→parent == rtParent) {                                        [6]
   // rotating at the root                                         [7]
   zrl=z→right→left;                                               [8]
   zr=z→right;                                                     [9]
   OK=CAS3(z→right,zrl,z→right,                                    [10]
           z→right,z,zrl→parent,                                   [11]
           zrl,zrl,z→right→left);                                  [12]
   if (OK) {                                                       [13]
      // update other links                                        [14]
       root = zr;                                                  [15]
       rtParent→left=root;                                         [16]
       root→parent=rtParent;                                       [17]
       z→parent=root;                                              [18]
       root→left=z;                                                [19]
   }                                                               [20]
} else {                                                           [21]
   // rotating under the root (parent, etc. exist)                 [22]
   if (z == z→parent→left) {                                       [23]
     // z is left child                                           [24]
      zrl=z→right→left;                                            [25]
      zr=z→right;                                                  [26]
      OK=CAS3(z→right,zrl,z→right,                                 [27]
              z→right,z,zrl→parent,                                [28]
              zrl,zrl,z→right→left);                               [29]
      if (OK) {                                                    [30]
         // update other links                                     [31]
          z→parent→left=zr;                                        [32]
          z→right→parent=z→parent;                                 [33]
          z→parent=zr;                                             [34]
          z→right→left=z;                                          [35]
      }                                                            [36]
   } else {                                                        [37]
     // z is right child                                          [38]
       zrl=z→right→left;                                           [39]
       zr=z→right;                                                 [40]
       OK=CAS3(z→right,zrl,z→right,                                [41]
               z→right,z,zrl→parent,                               [42]
               zrl,zrl,z→right→left);                              [43]
```

```
            if (OK) {                                          [44]
               // update other links                           [45]
               z→parent→right=zr;                             [46]
               z→right→parent=z→parent;                       [47]
               z→parent=zr;                                   [48]
               z→right→left=z;                                [49]
            }                                                 [50]
      }                                                       [51]
}                                                             [52]
return OK;                                                    [53]
```

Figure 5-11–The algorithm–Left-Rotation(z).

If the relevant grandchild of z exists, the Left-Rotate code uses a CAS3 operation to update the right child of z to point to the relevant grandchild and the parent of the relevant grandchild to point to z, assuming that the relevant grandchild has not changed (lines 19 to 21). If the CAS3 operation succeeds (OK is TRUE), the process then updates the other links (lines 22 to 29) needed to complete the rotation (OK will already be set to TRUE). If the CAS3 fails (OK is FALSE) due to another process modifying the tree, the process will return FALSE to Update-Rotation(), which will re-invoke Left-Rotate() again until it succeeds. A tree showing this scenario is illustrated in Figure 5-12.

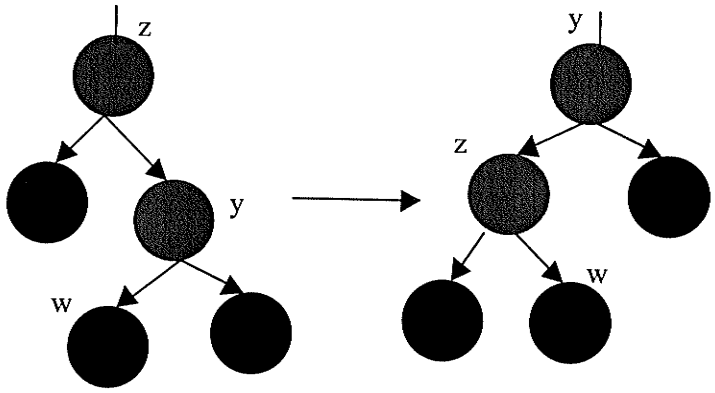

Figure 5-12–z is root and the left child of y is not empty.

If z is not the root of the tree, there are two situations to consider: z is the left child of its parent (line 33) or the right child of its parent (line 57). If z is the left child of its parent, the process first uses a CAS3 operation to update the right child of z to point to the relevant grandchild and the parent of the relevant

grandchild to point to z, assuming that the relevant grandchild is unchanged (lines 46 to 48). If the CAS3 operation succeeds (OK is TRUE), the process updates the other links needed to complete the rotation (lines 51 to 54). Otherwise, FALSE will, again, be returned to Update-Rotation() so it can re-invoke Left-Rotate(). A tree showing the scenario where the relevant grandchild exists is shown in Figure 5-13.

The case where z is the right child of its parent is symmetric and the code implementing the rotation in this case is at lines 55 to 78.

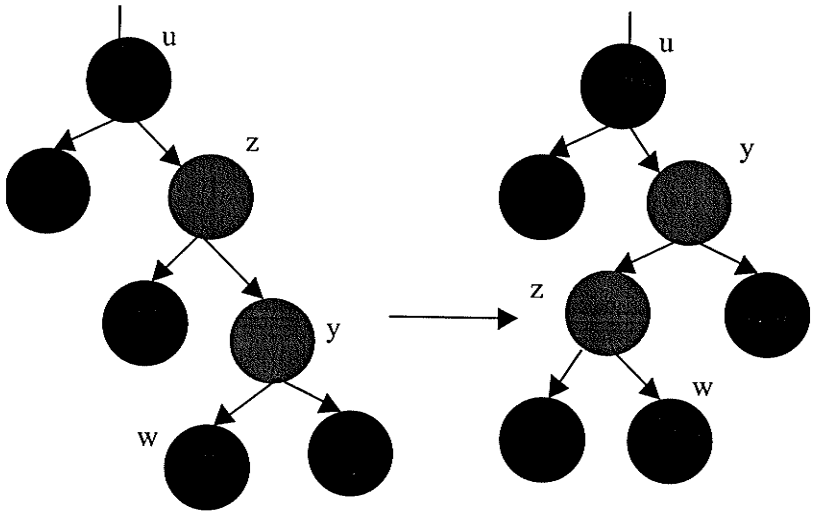

Figure 5-13–z is a non-root node and the left child of y is not empty.

6 Conclusions and Future Work

6.1 Conclusions

The red-black tree is a complicated data structure. The serial algorithms for the red-black tree operations are certainly more complex than the algorithms simpler data structures such as queues and lists. Accordingly, the lock-free concurrent insertion algorithm for red-black trees presented in this thesis is significantly more complex than corresponding lock-free concurrent algorithms for lists and queues. The heart of this complexity is the need to change multiple fields in the data structure atomically.

In this thesis, we present algorithms for operations on red-black trees that have concurrent processes manipulating them without the use of mutual exclusion. The algorithms designed include traversal of the red-black tree, insertion of nodes into the tree and the related re-balancing of the tree after insertion. The Compare & Swap (CAS), Double Compare & Swap (DCAS), and Compare & Swap 3 (CAS3) synchronization primitives are used to implement the algorithms. The algorithms presented in this thesis are, to the best of my knowledge, the first to address efficient lock-free operations on complex data structures generally and are certainly the first for Red-Black trees. Contributions of the thesis include:

A. A lock-free concurrent insertion algorithm for red-black trees that supports a high degree of concurrency by allowing concurrent read and writes operations by multiple processes accessing the tree.

B. A compatible lock-free concurrent traversal algorithm for red-black trees.

C. A general technique, setting a flag field in data structure nodes to control nearby concurrency, that should be applicable to other complex data structures and which helps to simplify the complexity of lock-free concurrent algorithms.

D. An in-depth case analysis (provided in Section 3.3.2) of concurrent operations in Red-Black trees that should be useful to people doing additional research on concurrent algorithms for red-black trees.

6.2 Future Work

In this thesis, shape, colour, and access constraints were considered in the development of the concurrent insertion algorithm. It may be possible to relax the access constraints in certain scenarios based on conditions imposed by other constraints in future research work. This work could lead to more efficient algorithms.

The chief disappointment with the algorithms presented in this thesis is the need for the use of `CAS3` operations to implement the rotations. Conventional hardware provides only `CAS` and `DCAS` operations so the requirement for `CAS3` may limit the practical applicability of the algorithms developed.[10] This suggests another area for possible future work – a redesign of the presented algorithms in an attempt to eliminate the need for `CAS3` primitives. This redesign might be done by somehow acquiring a flag on the relevant grandchild of the rotation point prior to invoking any rotation algorithm.

Another aspect of possible future work would be to consider the implementation of a concurrent deletion algorithm for red-black trees. Compared to the concurrent insertion algorithm, the concurrent deletion algorithm may, however, prove to be more complicated. In addition to having to deal with interactions with existing insertion and traversal algorithms, which will add complexity, the scope of activity during a deletion may be larger than during an insertion (i.e. more nodes may have to be operated on).

Finally, an implementation of the algorithms on real hardware and a performance analysis using a suite of real applications to accurately assess the benefits, liabilities and performance of the proposed technique would be well worth doing in the future.

[10] Though it does not, of course, take away from the theoretical significance of the results.

7 Bibliography

[1] R. Bayer. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms". Acta Informatica, 1(4): 290-306, November 1972.

[2] W. B. Easton. "Process Synchronization Without Long-Term Interlock". In Proceedings of the Third Symposium on Operating Systems Principles, pages 95-100,1972

[3] L. Lamport. "Concurrent Reading and Writing". Communications of the ACM, 20(11):806-811,1977.

[4] J. Olivie. "A New Class of Balanced Search Trees: Half-balanced Search Trees". R.A.I.R.O. Informatique Theoretique, 16: 51-71, 1982.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. "Introduction to Algorithms". Cambridge, Mass.: MIT Press/McGraw-Hill, 1991.

[6] M. Herlihy. "Wait-Free Synchronization". ACM Transactions on Programming Languages and Systems, 11(1):124-149, January 1991.

[7] S. Prakash, Y. H. Lee, and T. Johnson. "Non-Blocking Algorithms for Concurrent Data Structures". Technical Report TR91-002, University of Florida, 1991.

[8] M. Herlihy. "A Methodology for Implementing Highly Concurrent Data Objects". ACM Transactions on Programming Languages and Systems, 15(5): 745-770, 1993.

[9] G. Barnes, G. "A Method for Implementing Lock-Free Shared Data Structures". In 5th International Symposium on Parallel Algorithms and Architectures, pages 261-270, 1993.

[10] J. D. Valois. "Implementing Lock-Free Queues". In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, pages 64-69, October 1994.

[11] J. D. Valois. "Lock-Free Linked Lists Using Compare-and-Swap". In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 214-222, August 1995.

[12] M. Greenwald and D. Cheriton. "The Synergy Between Non-Blocking Synchronization and Operating System Structure". In Proceedings of the Second Symposium on Operating System Design and Implementation, USENIX, pages 123-136, October 1996.

[13] M. Farook. "Fast Lock-Free Link Lists in Distributed Shared Memory". MSc Thesis, University of Manitoba. 1998.