High-Concurrency Lock-Free Algorithms for Red-Black Trees

by

Jong Ho Kim

A thesis submitted to the faculty of graduate studies in partial fulfillment of the requirements for the degree of master of science at the department of computer science of the University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2.

©Copyright by Jong Ho Kim 2005

August 15, 2005

0-494-08880-X

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN:
Our file *Notre référence*
ISBN:

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

High-Concurrency Lock-Free Algorithms for Red-Black Trees

BY

Jong Ho Kim

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

Jong Ho Kim © 2005

Permission has been granted to the Library of the University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilms Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

Acknowledgements

I remember when I began my studies here it was a day in winter. Now it is a warm and sunny day out, and I am glad to finish my thesis in summer.

When I came to Winnipeg, I did not know anyone here. But I was lucky to meet my supervisors Dr. Helen Cameron and Dr. Peter Graham.

I would like to thank my supervisors for accepting me as their student, providing me with a topic for my thesis, and making it possible to write my thesis. Also, I would like to thank them for their valuable suggestions and ideas, and for carefully proofreading my thesis.

I would like to give special thanks to Dr. Cameron. To me, she has been a great example as a professor, supervisor/advisor, and as a person.

In the past several years here at the University of Manitoba, I learned a lot while also realizing that there is much more yet to learn. I will cherish each moment spent. And I would like to thank everyone for the wonderful educational experience!

Abstract

Red-black trees are a useful data structure because they have efficient dynamic set operations. Also, their insert and delete operations make constant structural changes. Hence, red-black trees have desirable properties that allow them to be implemented as a highly concurrent data structure. Yet not many high-concurrency red-black tree algorithms exist.

To implement a concurrent data structure correctly, a synchronization method is needed to ensure that concurrent operations on the structure do not interfere with one another. Conventionally, the synchronization method is a lock-based synchronization primitive. However, many problems associated with the lock-based methods have led researchers to consider lock-free synchronization methods.

In this thesis, we present a high-concurrency lock-free algorithm for red-black trees. We adapt an existing sequential red-black tree algorithm and an existing lock-free red-black tree insertion algorithm as a starting point. We use intention markers in our algorithm design. We then present a proof of correctness of the design. The proof contains some rules that become a part of the design. Finally, we present how we would approach implementing the algorithm design.

Contents

1 Introduction **9**

2 Background on Red-Black Trees **14**

2.1 Definition of Red-Black Trees and an Example 14

2.2 History of Red-Black Trees 16

2.3 Advantages of Red-Black Trees 17

2.4 Sequential Operations on Red-Black Trees 18

2.4.1 Some Implementation Considerations 18

2.4.2 Rotations 19

2.4.3 Sequential Insertion 19

2.4.4 Sequential Deletion 28

3 Related Work **38**

3.1 Problems with Traditional Lock-Based Methods 39

3.2 Universal Primitives 41

3.2.1 Compare-and-Swap 42

3.2.2 Load-Linked and Store-Conditional 44

3.2.3 Software Compare-and-Swap 45

3.2.4 Extensions of Compare-and-Swap 46
3.3 Existing Lock-Free Data Structures 47
3.4 Existing Lock-Free Red-Black Tree Algorithms 49
3.4.1 Software Transactional Memory (STM) 49
3.4.2 Ma's Insertion Algorithm 50
3.5 Conclusions Drawn from the Related Work 54
3.5.1 Reasons for Choosing Lock-Free Synchronization 54
3.5.2 Reasons for Choosing Compare-and-Swap 54
3.5.3 Reasons for Incorporating Ma's Insertion Algorithm . . 55

4 Design of the Concurrent Operations of Red-Black Trees and Proof of Correctness 56

4.1 Defining the Local Area 57
4.1.1 The Local Area for Insertions 57
4.1.2 The Local Area for Deletions 66
4.2 Intention Markers 76
4.2.1 Definition of Intention Markers 76
4.2.2 Necessity of Intention Markers 76
4.2.3 The Number of Markers Required 79
4.3 An Observation Concerning the Initial Local Areas for Insertions and Deletions 86
4.4 Increasing Concurrency by Releasing Unneeded Markers . . . 87
4.5 Sketch of the Algorithm Design 88
4.6 Proof of Correctness of the Algorithm Design 90
4.6.1 The Possible Case Transformations 90

4.6.2 Two Lemmas Required for a Proof of the Sufficiency of Four Markers 91
4.6.3 Preventing Double Markers and the Sufficiency of Four Markers 112
4.6.4 Summary of the Rules 186
4.7 Reasons for Not Using More Markers per Node 188

5 Some Implementation Details of the Algorithm 199
5.1 Implementing the Additional Rules 199
5.2 The Structure of a Node 203
5.3 Creating an Empty Tree 204
5.4 Overall Structure of the Algorithm 205
5.4.1 Using DCAS to Locate Insert and Delete Points 205
5.4.2 Establishing the Initial Local Area 209
5.4.3 Physical Placement and Removal of Nodes 211
5.4.4 Rebalancing the Tree 211

6 Conclusion 215
6.1 Future Work 216

A Patterns 218

List of Figures

2.1 A red-black tree. 16

2.2 A red-black tree, NIL omitted. 17

2.3 Rotations. 20

2.4 Pseudocode for left rotation. 21

2.5 Pseudocode for RB_Insert(T,x). 22

2.6 Pseudocode for RB_Insert_Fixup(T,x). 23

2.7 Insertion Case 1. 25

2.8 Insertion Case 2. The nodes pointed by → form a zig-zag shape. 26

2.9 Insertion Case 3. The nodes indicated by → form a straight line. 26

2.10 Possible violations of Properties 2 and 4 created by deleting a node. 30

2.11 Pseudocode for RB_Delete(T,z). 31

2.12 Pseudocode for RB_Delete_Fixup(T,x). 32

2.13 Deletion Case 1. 35

2.14 Deletion Case 2. (E means the node can be Either red or black.) 35

2.15 Deletion Case 3. 36

2.16 Deletion Case 4. 36

3.1 Compare-and-Swap. 43
3.2 Software CAS. 45
3.3 Double-Compare-and-Swap. A process atomically modifies two memory locations instead of one. 46
3.4 Without the local area protection, processes can interfere with each other. (This figure is based on Figures 4.1 and 4.2 in Ma [26].) 52

4.1 Nodes that a process must access during insertion Case 1. . . 58
4.2 Nodes that a process must access during insertion Case 2. . . 59
4.3 Nodes that a process must access during insertion Case 3. . . 60
4.4 The union of possible accesses for all insertion cases. 61
4.5 Reason for using 3-CAS in Ma [26]'s insertion algorithm, illustration. 65
4.6 Nodes that a process must access during deletion Case 1. . . 67
4.7 Nodes that a process must access during deletion Case 2. . . 68
4.8 Nodes that a process must access during deletion Case 3. . . 69
4.9 Nodes that a process must access during deletion Case 4. . . 70
4.10 The union of possible accesses for all deletion cases. 71
4.11 Deletion Case $1 \rightarrow 3 \rightarrow 4$ with four intention markers. The insertion process, in green, is in Case 3. (Picture 1 of 2.) . . . 74
4.11 Deletion Case $1 \rightarrow 3 \rightarrow 4$ with four intention markers. (Picture 2 of 2.) 75
4.12 Intention markers are the four nodes above the local area. . . 77

4.13 Deletion Case 2 and Deletion Case 2. The nodes marked with the $\leftarrow$ sign are nodes held by P_1 but are needed by P_2. The $\rightarrow$ sign point to nodes held by P_2 but are needed by P_1. 78
4.14 Insertion Case 1 and Deletion Case 2. Deadlock. 78
4.15 Deletion Case 2 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 4 with 3 markers. (Picture 1 of 2.) 82
4.15 Deletion Case 2 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 4 with 3 intention markers. (Picture 2 of 2.) 83
4.16 Deletion Case 2 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 4 with 4 markers. (Picture 1 of 2.) 84
4.16 Deletion Case 2 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 4 with 4 intention markers. (Picture 2 of 2.) 85
4.17 All possible case transformations (where '$\cdots$' means zero or more iterations of the case). 91
4.18 Pattern 1: A left rotation at x requires the marker on r to be moved to y. (Replica of Figure A.1.) 92
4.19 Pattern 2: A left rotation at x requires the marker on y to be moved to x. (Replica of Figure A.5.) 92
4.20 Pattern 3: A left rotation at x requires the marker on x to be moved to v, an ancestor of x. (Replica of Figure A.6.) 93
4.21 Pattern 4: A left rotation at x requires the marker on x to be moved to r, an ancestor of x. (Replica of Figure A.7.) 93
4.22 A situation where the markers of two processes come too close to one another. (Picture 1 of 2.) 116
4.22 A situation where the markers of two processes come too close to one another. (Picture 2 of 2.) 117

4.23 Part (a) is from Picture 9.2(c). P_2, in black here, is an insertion process. . . . 142
4.24 Four possible locations of P_4.x. P_4 is a deletion process. (Picture 1 of 2.) . . . 150
4.24 Four possible locations of P_4.x. (Picture 2 of 2.) . . . 151
4.25 P_4 itself does the Move-Up rule. (Picture 1 of 2.) . . . 160
4.25 P_4 itself does the Move-Up rule. (Picture 2 of 2.) . . . 161
4.26 Four possible locations of P_4.x. P_4 is an insertion process. (Picture 1 of 2.) . . . 163
4.26 Four possible locations of P_4.x. P_4 is an insertion process. (Picture 2 of 2.) . . . 164
4.27 Two processes involved in Problem 2. . . . 174
4.28 Possible locations of P_5. . . . 175
4.29 P_5 performs deletion Case 1 to Figure 4.28. . . . 175
4.30 Possible locations of P_6. . . . 177
4.31 P_6 performs deletion Case 1 to Figure 4.30. . . . 178
4.32 P_6 performs Cases 3 and 4 to Figure 4.30. . . . 179

5.1 Structure of a node. . . . 203
5.2 Initial tree created by CreateTree(). . . . 204
5.3 Insert algorithm structure chart. . . . 206
5.4 Delete algorithm structure chart. . . . 207

A.1 Pattern 1: A left rotation at x requires the marker on r to be moved to y. . . . 220
A.2 Left rotation at x. . . . 220

A.3 Left rotation at x. 220
A.4 Left rotation at x. 221
A.5 Pattern 2: A left rotation at x requires the marker on y to be moved to x. 221
A.6 Pattern 3: A left rotation at x requires the marker on x to be moved to v, an ancestor of x. 221
A.7 Pattern 4: A left rotation at x requires the marker on x to be moved to r, an ancestor of x. 222
A.8 Left rotation at x. 222

Chapter 1

Introduction

In multiprocessor, shared-memory computers, concurrent programming is useful since more work can potentially be done in a given time with many processes working concurrently instead of the processes being forced to work sequentially. Despite the usefulness of concurrency, only a few data structures have concurrent implementations. The few existing concurrent data structures include linked lists, queues, stacks, and B-trees.

A red-black tree is a useful data structure because it has logarithmic height with respect to its size, and its insert and delete operations take logarithmic time. In contrast, an ordinary binary search tree's operations may take linear time with respect to its size. Also, efficient red-black tree insertion and deletion algorithms, by Cormen et al. [4], for example, exist for its sequential operations.

A red-black tree can be used in any computer applications that require efficient searches in main memory.

Another type of tree, the AVL trees, also have operations that take log-

arithmic time. However, all operations in red-black trees make constant structural changes whereas a deletion operation in AVL trees may require logarithmic structural changes. Fewer structural changes mean less chance of conflicts between processes. Thus, the minimal structural changes allow red-black trees to be implemented as a highly concurrent data structure. Yet, not many concurrent algorithms exist for red-black trees.

To implement a concurrent data structure, a synchronization method is needed to ensure that processes work in a concurrent manner without interfering with one another. Traditionally, a lock-based synchronization method is used. However, problems associated with lock-based methods such as deadlock, priority inversion, and convoying have led researchers to consider lock-free synchronization methods. Lock-free methods do not incur these problems and, interestingly, some researchers [9, 27] have used them in the design and implementation of multiprocessor operating system kernels.

With lock-free synchronization methods, the overhead of obtaining and releasing locks are removed. Further, in parallel programming, the delays caused by blocking when a process attempts to acquire a lock held by another process can be highly disruptive to overall progress. This is because processes commonly interact frequently and a blocked process will quickly delay other processes as well. It is therefore better to periodically incur the need to recompute a partial result than to pay the high cost of blocking.

There exist many lock-free synchronization primitives that can be used to implement a concurrent data structure. However, only CAS (Compare-And-Swap) and LL/SC (Load-Linked and Store-Conditional) have been popular because they are more widely available through hardware support. Even

if hardware support is not available, these synchronization methods can be implemented in software.

General methods [1, 14, 16, 34, 40] that turn any sequential data structure into a wait-free concurrent data structure have been devised (see Section 3.2 for the description of wait-free). However, general methods are generally accepted as not being efficient. In contrast, some lock-free data structures implemented with data-structure-specific algorithms have shown better performance than those with the corresponding lock-based algorithms, as shown by Michael and Scott [29] and Greenwald and Cheriton [9].

Herlihy et al. [17] and Harris and Fraser [11] implemented lock-free red-black trees using their Software Transactional Memories (see Section 3.2.4 for a discussion of Software Transactional Memory). A transaction is similar to database transactions where each transaction either commits and takes effect or aborts and does not take effect. However, a transaction to do an insertion or a deletion may cover a large area of the tree, and may incur high re-execution cost in the event that a transaction aborts.

Ma [26] presents a lock-free red-black tree insertion algorithm, and uses the concept of local area, which bounds the current working area of a process to several neighbouring nodes. The local area is realized by adding a flag to a node, so when a process sets the flag of the node the process has the full control of the node. A local area consists of several neighbouring nodes that a process has full control of. Hence, the use of the local area concept may allow more concurrency and have lower re-execution cost. However, an efficient deletion algorithm is needed for Ma's insertion algorithm.

In our thesis, we present a concurrent, lock-free insertion and deletion

algorithms for red-black trees. In our algorithm, Cormen et al.'s sequential algorithm [4] and Ma's insertion algorithm are adapted with the addition of intention markers as a starting point. Intention markers are located immediately above each process's local area and help keep a safe distance between any insertion and deletion processes so that the processes can proceed without interfering with one another. The intention markers are also intended to indicate that a process intends to move its local area up where the markers are placed.

A process, during its operations, can move its local area to nodes where another process's intention markers are placed. However, a process may not move its markers to nodes in another process's local area.

With the intention markers added, we go through all the transformation cases that a process could go through in the proof of correctness, thereby making it a proof by enumeration. In the proof, we find that two markers could be moved to a node, and we determine that two or more markers on a node are a problem to be avoided. Hence, we add some rules that resolve this problem. These rules become part of our algorithm design.

We provide some implementation details that can be used as a guide to implement our algorithm. In the rules that were added in the proof, the processes involved will need to coordinate. We describe some details of this coordination using communication between processes via flags in a node.

The rest of this thesis is organized as follows.

In Chapter 2, we describe what a red-black tree is. We also discuss the history and advantages of red-black trees. We then explain Cormen et al.'s sequential red-black tree algorithm, which is used in our algorithm.

In Chapter 3, we discuss related work. We identify the advantages of lock-free methods over the lock-based methods. Then we discuss lock-free synchronization techniques and review some representative existing lock-free data structures, including red-black trees. We finally provide reasons for choosing a particular synchronization technique, CAS, and for incorporating Ma's red-black tree insertion algorithm.

In Chapter 4, we present our algorithm design. We first describe how we incorporate Ma's insertion algorithm with intention markers. In doing so, we explain in detail about a process's local area of effect and the use of intention markers. Then we present an observation to be used in a proof of correctness which is provided later, and then discuss a choice made to increase concurrency. After sketching the algorithm design, we present a proof of correctness of the algorithm. In the proof of correctness, we add some additional rules that become part of the final algorithm design. We finish by discussing reasons for not designing the algorithm in another way.

In Chapter 5, we present an approach to implementing our algorithm design. First, we discuss some implementation details of the rules added in the proof of correctness. Then we describe the structure of a node and an empty tree. Finally, after presenting an overall picture of the algorithm using structure charts, we explain the major components in the charts.

In the final chapter, Chapter 6, we provide a conclusion and future work for this thesis.

Chapter 2

Background on Red-Black Trees

In this chapter, we describe what a red-black tree is by giving a definition and an example. Then we very briefly discuss the history of red-black trees. We next identify the advantages of red-black trees and their challenges with respect to concurrency. Finally, we describe an existing sequential implementation of red-black trees which will serve as a reference in later chapters of the thesis.

2.1 Definition of Red-Black Trees and an Example

A red-black tree is a type of a binary search tree with the addition of a colour attribute and associated red-black properties that are used to maintain balance in the tree.

An ordinary binary search tree can have a worst-case height of n, given n internal nodes. In this case, the performance of an insert or a delete operation is no better than that of a linked list, which is $O(n)$ time. On the other hand, the red-black properties ensure that a red-black tree has height which is logarithmic in the number of internal nodes all the time.

A binary tree is either empty (consists of only an external node) or it consists of an internal root node with two binary trees as children. Thus, an external node has no children, and an internal node has exactly two children. A red-black tree is a binary search tree that satisfies five red-black properties. The red-black properties are as follows:

1. Every node is either red or black.
2. The root node is black.
3. External nodes are black.
4. A red node's children are both black.
5. All the paths from a node to its descendent leaves contain the same number of black nodes.

An example of a red-black tree is shown in Figure 2.1. Black nodes contain black-coloured keys, and red nodes contain red-coloured keys. Just as in binary search trees, red-black trees preserve the binary search tree property — the left subtree of a node `x` contains only nodes having keys less than `x`'s key, while the right subtree of the node `x` contains only nodes having larger keys than `x`'s key.

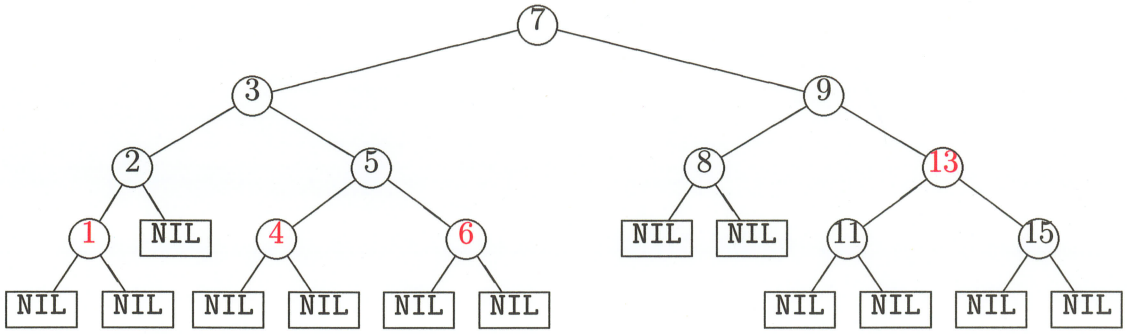


Figure 2.1: A red-black tree.

In addition to the binary search tree property, the tree in Figure 2.1 also has the red-black properties. First, each node is either red or black. Second, the root, whose key is 7, is black. Third, the external nodes, the `NIL` sentinels, are black. Fourth, the red nodes 1, 4, 6, and 13 have two black children each. Fifth, all of each node's paths to its descendent leaves have the same number of black nodes. For example, the path from the root down to `NIL` including 7, 9, 8, `NIL` has 4 black nodes, and so does the path 7, 3, 5, 4, `NIL`, etc.

We will omit the external nodes represented by `NIL` in Figure 2.1 from the pictures that follow in the thesis. However, we will assume that they are there. Our trees will now be displayed as in Figure 2.2. The letter '`R`' in the center of a node in Figure 2.2 means the colour of the node is red, and '`B`' means the colour of the node is black.

2.2 History of Red-Black Trees

Bayer [2] first described red-black trees. He called the class of trees he defined "symmetric binary B-trees" and showed that the height of such a tree with

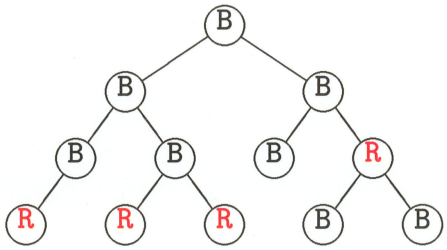


Figure 2.2: A red-black tree, NIL omitted.

n nodes is bounded by $O(\log_2 n)$. Olivié [32] independently discovered red-black trees, which he called "half-balanced binary search trees." Tarjan [38] shows that Bayer and Olivié describe the same class of trees. Later, Guibas and Sedgewick [10] introduced the convention of colouring the nodes of the tree red and black. Guibas and Sedgewick also provide an efficient insertion algorithm with $O(1)$ structural changes. Tarjan [38] describes a deletion algorithm with $O(1)$ structural changes.

2.3 Advantages of Red-Black Trees

Basic dynamic set operations such as insertions, deletions, and searches always take only $O(\log_2 n)$ time in a red-black tree with n nodes because of the logarithmic height of the tree. In contrast, the same operations take $O(n)$ worst-case time in an ordinary binary search tree with n nodes.

The AVL trees also offers $O(\log_2 n)$ operations. However, deletion in an AVL tree with n nodes may require $O(\log_2 n)$ structural changes whereas in red-black trees deletion only requires $O(1)$ structural changes. The minimal structural changes required in red-black trees may allow concurrency

more easily since the fewer structural changes needed, the less the chance of conflicts between the processes.

Because the logarithmic height of red-black trees guarantees $O(\log_2 n)$ execution time for the basic dynamic set operations and because of the $O(1)$ structural changes, red-black trees have desirable properties that should allow them to be implemented as a highly-concurrent data structure. Concurrency is still challenging because insertion and deletion processes must work without interfering with each other, and doing this efficiently can be difficult.

2.4 Sequential Operations on Red-Black Trees

In this section, we describe how a red-black tree is maintained using sequential operations. The operations include rotation, insertion, and deletion. The pseudocode for the operations in this section are taken from Cormen et al. [4].

2.4.1 Some Implementation Considerations

A node in a red-black tree has five fields. They are its colour, data or key, a link to the left child, a link to the right child, and a link to the parent. In this chapter, we will call them colour, key, left, right, and p, respectively.

Cormen et al. [4] use a sentinel node `nil[T]` to represent all the `NIL` external nodes of a red-black tree `T` as well as the root's parent. The colour of the sentinel node `nil[T]` is black; its other fields contain arbitrary values.

2.4.2 Rotations

Inserting or deleting a node may cause the resulting tree to violate red-black properties. The insert and delete operations perform some recolouring and/or restructuring of the nodes to restore the properties. To restructure a red-black tree, we use rotations. Rotations preserve the binary search tree property. There are two kinds of rotations: left and right rotations.

Figure 2.3 shows both left and right rotations. In the figure, `T` refers to the overall tree, and `x` is the root of the subtree to be rotated. In a left rotation, node `x`'s right child, `y`, becomes the new parent of `x`. `x`'s previous parent then becomes `y`'s parent, and y's previous left child becomes `x`'s new right child. Note that we must make sure that the right child of `x` exists before calling `Left_Rotate(T,x)`. Figure 2.4 contains the pseudocode for `Left_Rotate(T,x)`. `Right_Rotate` is symmetric to `Left_Rotate(T,x)`. All references to '`left`' and '`right`' are simply exchanged in the code.

2.4.3 Sequential Insertion

An insertion in a red-black tree starts out with the same processing as an ordinary binary search tree insertion. That is, we find the leaf node at which we should place the new node. Then we colour the new node red. By doing this, we may have introduced two violations of the red-black properties as follows:

- If the newly inserted node is the only node in the tree, then we know it is the root node and it should be black. Property 2, thus, is broken.
- If the new node's parent node is a red node, then Property 4 does not

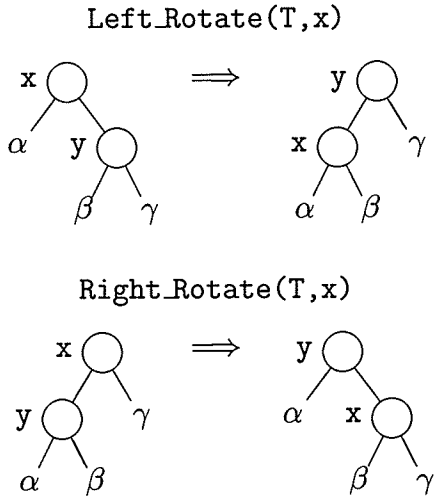


Figure 2.3: Rotations.

hold any more.

Therefore, the insertion routine `RB_Insert(T,x)` (see Figure 2.5) must call the routine `RB_Insert_Fixup(T,x)` (see Figure 2.6) to restore the red-black properties that have been broken. In these routines, `x` is a pointer to the new node, and we assume `key[x]` is not already in the tree.

The while loop of the `RB_Insert_Fixup(T,x)` can be broken into three cases. (There actually are six cases, but the other three cases are symmetric to the three we will discuss in which `p[x]` is a left child; see Lines 2 and 20 of Figure 2.6.) We now discuss how the three cases restore any violations of Property 2 or 4 and how they preserve the other red-black properties.

When a node is inserted into an empty tree, Property 2 is violated, but the other properties still hold. The while loop in `RB_Insert_Fixup(T,x)` is entered only while the parent of `x` is red. Since `x` is the new root node, the

```
Left_Rotate(T,x) {
1.    y = right[x];
2.    right[x] =  left[y];
3.    p[left[y]] =  x;
4.    p[y] = p[x];
5.    if (p[x] == nil[T])
6.      root[T] = y;
7.    else if (x == left[p[x]])
8.      left[p[x]] = y;
9.    else
10.     right[p[x]] = y;
11.   left[y] = x;
12.   p[x] = y;
13. }
```

Figure 2.4: Pseudocode for left rotation.

```
RB_Insert(T,x) {
      /* Search for the insert point, z. */
1.    z = nil[T];
2.    y = root[T];
3.    while(y != nil[T]) {
4.      z = y;
5.      if (key[x] < key[y])
6.        y = left[y];
7.      else
8.        y = right[y]
9.    } /* end while */
      /* Place new node x as child of z. */
10.   p[x] = z;
11.   if (z == nil[T])
12.     root[T] = x;
13.   else if (key[x] < key[z])
14.     left[z] = x;
15.   else
16.     right[z] = x;
17.   left[x] = nil[T];
18.   right[x] = nil[T];
19.   colour[x] = red;
20.   RB_Insert_Fixup(T,x);
21. }
```

Figure 2.5: Pseudocode for `RB_Insert(T,x)`.

```
RB_Insert_Fixup(T,x) {
1.    while (colour[p[x]] == red) {
2.      if (p[x] == left[p[p[x]]]) {
3.        y = right[p[p[x]]];
4.        if (colour[y] == red) { /* Case 1 */
5.          colour[p[x]] = black;
6.          colour[y] = black;
7.          colour[p[p[x]]] = red;
8.          x = p[p[x]];
9.        }
10.       else {
11.         if (x == right[p[x]]) { /* Case 2 */
12.           x = p[x];
13.           Left_Rotate(T,x);
14.         }
15.         colour[p[x]] = black; /* Case 3 */
16.         colour[p[p[x]]] = red;
17.         Right_Rotate(T,p[p[x]]);
18.       } /* end else at Line 10 */
19.     } /* end if at Line 2 */
20.     else { // p[x] = right[p[p[x]]];
          /* Symmetric to above. */
          /* Exchange right with left including Rotations. */
21.     }
22.   } /* end while at Line 1 */
23.   colour[root[T]] = black;
24. }
```


Figure 2.6: Pseudocode for `RB_Insert_Fixup(T,x)`.

parent of x is nil[T] and, is therefore, black. Consequently, we skip the while loop and execute Line 23. This changes x's colour to black, removing the only violation (of Property 2). Thus, we are assured that the first iteration of the loop is possible only when Property 4 is the only violated red-black property.

When we enter the while loop, p[x] is red. Since a red node cannot be the root, we are also assured that p[p[x]], the grandparent of x, exists, making the if statement at Line 2 safe.

Case 1 (Line 4 in Figure 2.6) is executed when x's uncle, named y in our pseudocode, is a red node. The processing done in Case 1 is illustrated in Figure 2.7.

Cases 2 and 3 are executed when y is black. In Case 2 (Line 11 in Figure 2.6), x is a right child of p[x], and x's grandparent, parent, and x form a "zig-zag," as shown in Figure 2.8. In Case 3 (Line 15 in Figure 2.6), x is a left child of p[x], and x's grandparent, parent, and x form a "straight" line as shown in Figure 2.9.

If Case 2 is executed, Case 3 is also executed; Case 3's processing immediately follows Case 2. However, Case 3 may be executed without Case 2 being previously entered.

Case 1 may be repeated while Cases 2 and 3, if executed, run once and terminate because all the red-black properties are restored.

In Case 1, node x's parent and y are recoloured to black and x's grandparent's colour is changed to red. Then we move x to point to the grandparent. At the start of the next iteration, therefore, we are assured that x points to a red node. We still have Properties 1 and 3 intact. Property 5 is also still

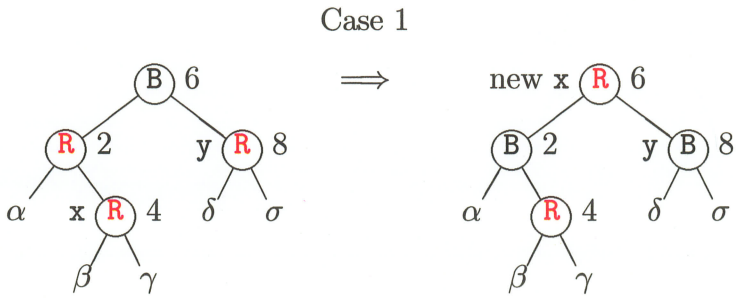


Figure 2.7: Insertion Case 1.

preserved since, as we can see from Figure 2.7, the number of black nodes from the root of the subtree, new `x`, to the leaves of the subtrees rooted at α, β, γ, δ, and σ still remain the same after Case 1. We have thus corrected the lone violation of Property 4 in the subtree.

After Case 1, if the new `x` is the root, we have created a violation of Property 2. However, since `p[x]` which is `nil[T]` is black, the loop terminates and the problem is corrected at Line 23. In contrast, if the new `x` is not the root, we may have created a violation of Property 4 between the new `x` and its parent. If indeed that is the case, we will repeat the loop. The next iteration may, again, be Case 1, 2, or 3. If the new `x` is not the root and the parent of the new `x` is black, then there is no violation of Properties 2 and 4, and we are done.

Case 2's purpose is to transform the tree into Case 3 by performing a left rotation. Before the rotation, we are making sure that `x`'s right child exists by moving `x` one level up since, when we move `x` up, the old `x` is the right child of the new `x`; see the left picture of Figure 2.8. Since the old `x`'s children were both black, after the rotation, the new `x`'s right child will also

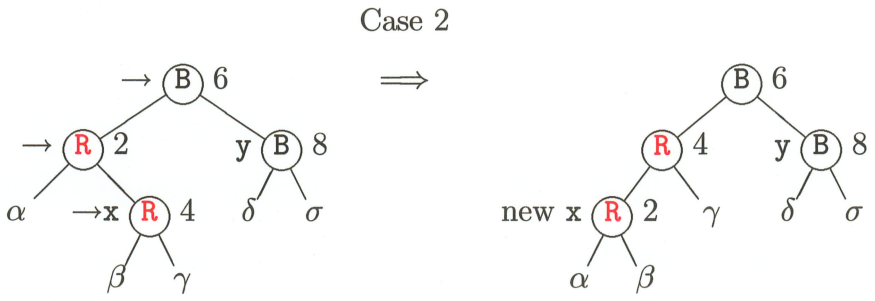


Figure 2.8: Insertion Case 2. The nodes pointed by → form a zig-zag shape.

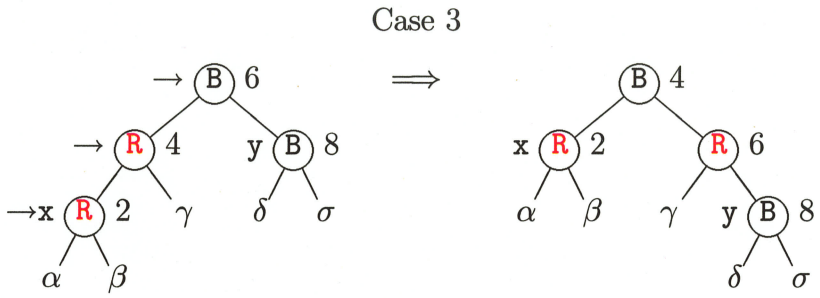


Figure 2.9: Insertion Case 3. The nodes indicated by → form a straight line.

be black, preserving Property 4 between x and its children. We also will still have Properties 1 and 3 intact. Property 2 still holds because we have not touched the root node. Also, from Figure 2.8, we can see that the number of black nodes from the root of the subtree to the leaves of any of the subtrees rooted at α, β, γ, δ, and σ is the same before and after the rotation, thereby preserving Property 5. Therefore, Property 4 between x and its parent is the only violation we have after Case 2. Then we fall through into Case 3; see Figure 2.6, Lines 14 and 15.

In Case 3, x is a left child of p[x] as shown in Figure 2.9. The parent of x's colour is changed to black. We change the grandparent's colour to red; then we perform a right rotation on the grandparent. Again, we have not done anything to affect Properties 1 and 3. A violation of Property 2 is not created because the black parent of x becomes the root of the subtree through the rotation.

Property 4 is corrected because p[x] is black now. Property 5 is preserved for all altered nodes because, as we can see in Figure 2.9, the number of black nodes from the root of the subtree to the leaves of subtrees rooted at α, β, γ, δ, and σ before and after the transformation is the same. Therefore, after Case 3 is executed, all the red-black properties hold, and the loop will terminate.

Thus, the while loop terminates because one of the following situations has occurred:

- We have reached the root after one or more iterations of Case 1.
- After Case-1 processing, the new x is not the root, but the new x's parent is black.
- We have performed Case 3.

In all three situations, we know that all red-black properties have been restored upon completion of `RB_Insert_Fixup(T,x)`.

We have thus shown how an insertion works in a red-black tree. We argue that an insertion takes $O(\log_2 n)$ time as follows. Since `RB_Insert(T,x)` runs in $O(h)$ time, where h is the height of the tree, and a red-black tree has height logarithmic with respect to its size n, `RB_Insert(T,x)` runs in

$O(\log_2 n)$ time. In `RB_Insert_Fixup(T,x)`, Case 1 may be repeated until `x` reaches the root node, so the while loop in `RB_Insert_Fixup(T,x)` can repeat at most $O(\log_2 n)$ times. A rotation takes constant time, and at most two rotations may be done since in Case 1, we do not perform any rotations, and in Case 2, we perform a single rotation and then immediately enter Case 3 in which we perform a single rotation and exit the while loop. Therefore, the total running time is $O(\log_2 n) + O(\log_2 n) + O(1)$, which is $O(\log_2 n)$.

2.4.4 Sequential Deletion

A deletion also starts out as if we are deleting a node in an ordinary binary search tree. That is, we find the node to be deleted; then we either splice out the node (if the node has no children or a single child) or we splice out the successor (in key order) of the node and copy the successor's value to the node (if the node has two children).

If we have deleted a red node, there is nothing to be done. The number of black nodes remains the same on the path where the red node was deleted. The red node could not have been the root since the root node is black. Because the children of the red node were black nodes, no two red nodes have become adjacent.

Deleting a black node, on the other hand, may create some violations of the red-black properties.

- If the root node was deleted and its red child became the new root as in Figure 2.10(a), then there is a violation of Property 2.

- A violation of Property 4 occurs when the deleted node's parent and its sole child were both red, as in Figure 2.10(b).
- Also, Property 5 may not hold because there is one fewer black node on the path now.

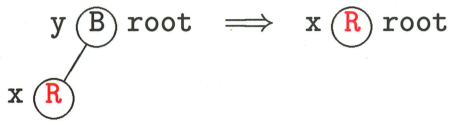
To restore these properties, `RB_Delete(T,z)` calls `RB_Delete_Fixup(T,x)`. (Figures 2.11 and 2.12 show the pseudocode of these functions.) Here, `z` is a pointer to a node with the key of the node to be deleted. The variable `y` in `RB_Delete(T,z)` is the node which is physically deleted; `y` may be the successor of `z` or `z` itself. The variable `x` is a pointer to the child of the deleted node `y` — that is, to the node that must replace `y`.

When `RB_Delete_Fixup(T,x)` is called, there may be a violation of Property 2, 4, or 5. However, as we can see in Figure 2.10, `x` must be red when we have a violation of Property 2 or 4 so the while loop condition at Line 1 (in Figure 2.12) is false. Therefore, we go straight to Line 24 and colour `x` black which restores Properties 2 or 4 and 5. We have thus compensated for a deletion of a black node `y`, by colouring `x` black. Therefore, the while loop in `RB_Delete_Fixup(T,x)` will be entered for the first time only when Property 5 is the only violated red-black property.

For deletion, the while loop may be divided into four different cases. (There actually are eight cases, but the other four are symmetric to the ones we will discuss. The two groups of four are determined by whether `x` is a left or a right child; see Lines 2 and 23 in Figure 2.12.) We will examine these four cases and discuss how they restore Property 5.

Figures 2.13 through 2.16 show the processing performed by each of the four cases. We note that `x`'s sibling, `w`, is not a `nil[T]` node because that

(a) The red child of the deleted root
becomes the root, violating Property 2.



(b) The red child of a deleted black node becomes
the red child of a red parent, violating Property 4.

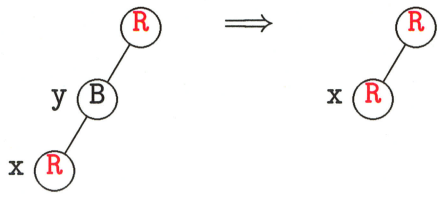


Figure 2.10: Possible violations of Properties 2 and 4 created by deleting a node.

```
RB_Delete(T,z) {
1.    if (left[z] == nil[T] || right[z] == nil[T])
2.      y = z;
3.    else
4.      y = TREE_SUCCESSOR(z); // Finds the key-order successor of z.
5.    if (left[y] != nil[T])
6.      x = left[y];
7.    else
8.      x = right[y];
9.    p[x] = p[y];
10.   if (p[y] == nil[T])
11.     root[T] = x;
12.   else {
13.     if (y == left[p[y]])
14.       left[p[y]] = x;
15.     else
16.       right[p[y]] = x;
17.     } // end else at Line 12
18.   if (y != z)
19.     key[z] = key[y];
20.   if (colour[y] == black)
21.     RB_Delete_Fixup(T,x);
22.   return y;
23. }
```

Figure 2.11: Pseudocode for `RB_Delete(T,z)`.

```
RB_Delete_Fixup(T,x) {
1.    while(x != root[T] && colour[x] == black) {
2.      if (x == left[p[x]]) {
3.        w = right[p[x]];
4.        if (colour[w] == red) { /* Case 1 */
5.          colour[w] = black;
6.          colour[p[x]] = red;
7.          Left_Rotate(T,p[x]);
8.          w = right[p[x]]; }
9.        if (colour[left[w]] == black && colour[right[w]] == black) {
10.         colour[w] = red; /* Case 2 */
11.         x = p[x]; }
12.       else {
13.         if (colour[right[w]] == black) { /* Case 3 */
14.           colour[left[w]] = black;
15.           colour[w] = red;
16.           Right_Rotate(T,w);
17.           w = right[p[x]]; }
18.         colour[w] = colour[p[x]]; /* Case 4 */
19.         colour[p[x]] = black;
20.         colour[right[w]] = black;
21.         Left_Rotate(T,p[x]);
22.         x = root[T]; } /* end else at Line 12 */ } /* end if at Line 2 */
23.     else { /* Symmetric Cases. */ } } /* end while at Line 1 */
24.   colour[x] = black;
25. }
```


Figure 2.12: Pseudocode for RB_Delete_Fixup(T,x).

would make the number of black nodes from x's parent to x and to w different. Node x is black since otherwise the loop would not have been entered.

Case 1 (Line 4 in Figure 2.12) differs from the other cases in that it is executed when w is red.

In Cases 2, 3, and 4, w is a black node. They distinguish themselves from one another by what colour w's children are. In Case 2 (Line 9 in Figure 2.12), w's children are both black. In Case 3 (Line 13 in Figure 2.12), w's left child is a red node. In Case 4 (Line 18 in Figure 2.12), w's right child is a red node.

Case 1's purpose is to convert Case 1 to Case 2, 3, or 4. In Case 1, we colour w black and also colour w's parent red and perform a left rotation on w's parent. This rotation results in x having a new w. As we can see in Figure 2.13, the previous left child of the previous w becomes the new right child of x's parent. Therefore, node x now has a new sibling that is black.

In Case 2, we colour w red and move x up one level. If Case 2 is executed right after Case 1, the loop terminates since the new x, the previous parent of x, is red. We colour the new x black in Line 24 and exit.

In Case 3, we colour w's left child black, we colour w red and perform a right rotation on w. Through the rotation, w's left child which was just coloured black becomes the new w. The right child of the new w is the previous w whose colour was changed to red. The result is that we have converted Case 3 to Case 4.

In Case 4, we change w's right child to black and push w's parent's colour down to w. We add a black node between x and the root of the subtree by colouring p[x] black and left rotating at p[x].

In looking at the four cases, there are two places where the while loop

can terminate: after Case 2 and Case 4.

Case 2's termination can be divided into two subcases. In one subcase, the new x is a red node; see Figure 2.14. Because the new x is red, we terminate from the loop. By colouring the new x black at Line 24 (in Figure 2.12), we have added a black node from the old x's parent to the leaves of the subtrees rooted at α and β, while maintaining the same number of black nodes to the leaves of the right subtrees rooted at γ, δ, σ, and ζ.

In the other subcase of termination from the loop after Case 2, the new x is the root. In this case, note that the root's colour, which is black, does not change. As we can see in Figure 2.14, the number of black nodes from the root to the leaves of the subtrees rooted at α and β does not change. The number of black nodes from the root to the leaves of the subtrees rooted at γ, δ, σ, and ζ has been decreased by one, however, since the previous x's sibling has changed its colour from black to red.

Case 4 terminates the loop because x is set to the root. We note in Figure 2.16 that a black node has been added to the path from the root of the subtree to the leaves of the subtrees rooted at α and β, while maintaining the same number to the leaves of the subtrees rooted at γ, δ, σ, and ζ.

Therefore, in all the cases where the loop terminates, we have compensated for the loss of a black node y in the paths from the root of the subtrees in Figures 2.13 to 2.16 to the leaves of the subtrees rooted at α and β by either adding a black node to the paths from the root of the subtrees in Figures 2.13 to 2.16 to the leaves of the subtrees rooted at α and β or by decreasing the number of black nodes by one from the paths from the root of the subtrees in the figures to the leaves of the subtrees rooted at γ, δ, σ,

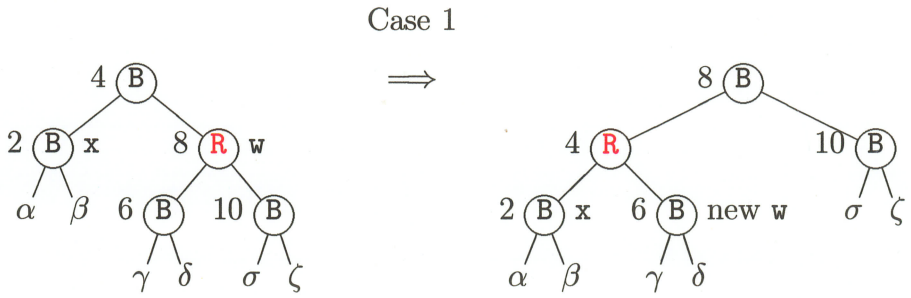


Figure 2.13: Deletion Case 1.

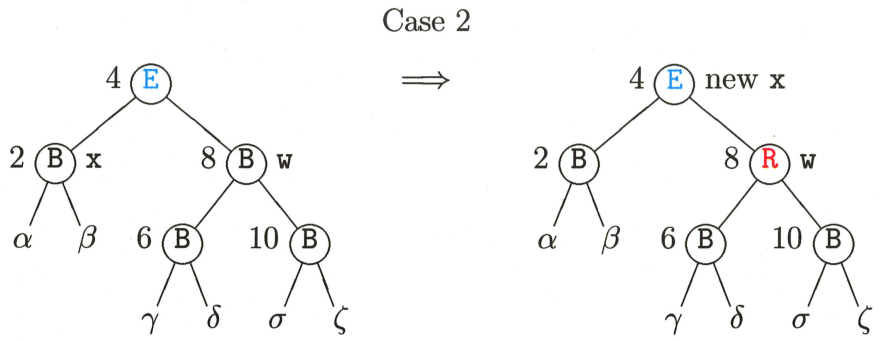


Figure 2.14: Deletion Case 2. (E means the node can be Either red or black.)

and ζ. These actions restore Property 5.

The processing in all four cases do not do anything to create new violations of Properties 1, 2, and/or 3. A possible violation of Property 4 by Case 2 is restored at Line 24 of `RB_Delete_Fixup(T,x)`. Also, we have shown how Property 5 is restored. Therefore, upon completion of `RB_Delete_Fixup(T,x)`, all the red-black properties hold.

We have now shown how a deletion works in a red-black tree. We argue that the running time of a deletion in a red-black tree with n nodes is

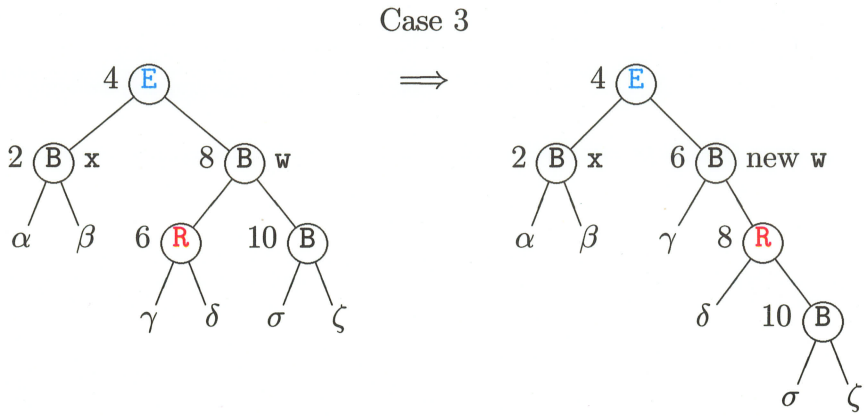


Figure 2.15: Deletion Case 3.

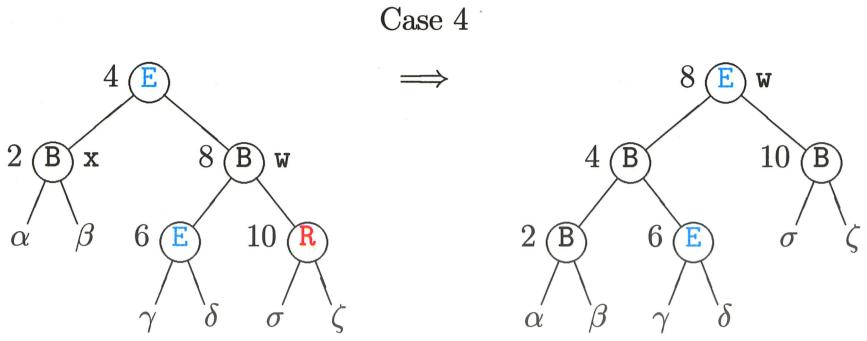


Figure 2.16: Deletion Case 4.

$O(\log_2 n)$. `RB_Delete(T,z)` runs in $O(h)$ time, where h is the height of the tree, and h is $O(\log_2 n)$ in a red-black tree with n nodes. In `RB_Delete_Fixup(T,x)`, we may repeat Case 2 until `x` reaches the root. Thus, we may walk up the tree, taking $O(\log_2 n)$ time. Cases 1, 3, and 4 take constant time, performing at most three rotations as follows: We do one rotation in Case 1, and Case 1 does not get repeated because other cases do not get converted to Case 1. We do not perform rotations in Case 2. In Case 3, we perform a rotation and immediately enter Case 4 in which we perform a rotation and exit the while loop. Thus, the total running time is $O(\log_2 n) + O(\log_2 n) + O(1)$, which is $O(\log_2 n)$.

Chapter 3

Related Work

In this chapter, we discuss the reasons for choosing lock-free methods and choosing a particular class of synchronization primitives for our concurrent red black tree algorithms. First, problems with lock-based methods are explained. Then we show that there exists an alternative solution, lock-free methods using universal primitives. The universal primitives are defined, and several of them are explained. The availability of the primitives in current commercial machine architectures as well as the primitives' software and hardware implementations in the literature are discussed. Then we discuss several data-structure-specific lock-free algorithms from the literature that exhibit better performance results than their lock-based counterparts. We also discuss existing lock-free red-black tree algorithms. Finally, our reasons for choosing lock-free methods and a particular synchronization primitive, and reasons for incorporating an existing red-black tree insertion algorithm are stated.

3.1 Problems with Traditional Lock-Based Methods

To implement a concurrent data structure correctly, a synchronization method is needed to ensure that concurrent operations on the structure do not interfere with one another.

Conventionally, a synchronization method is built on top of the well-known lock-based synchronization primitive Test-and-Set, which is a core hardware component of an implementation of a "semaphore" as introduced by Dijkstra [6]. Lock-based methods typically involve locking the entire data structure or a large part of it, thereby limiting potential concurrency. Moreover, there are several problems associated with lock-based methods. Some of the problems are as follows:

Possibility of deadlock: In certain situations, a group of concurrent processes may be unable to proceed because of the intertwining of requests for and acquisitions of locks between the processes. Deadlock was initially called "deadly embrace" by Dijkstra [6].

Priority inversion: A priority inversion [36] occurs when a higher priority process is blocked because a lower priority process holds a lock that the higher priority process needs.

Convoying: If a slow process holds a lock, many processes may line up to acquire the lock. The line of waiting processes forms a "convoy". An example of convoying, in the form of many threads waiting to obtain a lock in parallel systems

with multiprogramming, can be seen in a paper by Zahorjan et al. [41].

Problems such as these arise because of the blocking nature of locks and have led researchers to consider the use of alternative methods, lock-free methods. In contrast to lock-based methods, a process using lock-free methods can make changes to a shared memory location without obtaining a lock. Lock-free methods can also be non-blocking. "Non-blocking" here means that processes are not de-scheduled due to concurrent access by other processes and therefore no process can be indefinitely delayed by a slow process.

Massalin and Pu [27] and Greenwald and Cheriton [9] incorporate lock-free synchronization methods into the design and implementation of multiprocessor operating system kernels. Greenwald and Cheriton [9] describe their reasons for choosing lock-free methods as being:

- Lock-free methods allowed them to design an asynchronous RPC (Remote Procedure Call) handler without the possibility of deadlock.
- Lock-free methods allowed them to design a process scheduler without priority inversions, unwanted delays caused by cache misses and page faults, and without convoying problems.
- The lock-free synchronization primitives that they used — CAS (Compare-and-Swap) and DCAS (Double-Compare-and-Swap) — are simpler than lock-based solutions that implement critical sections. (The semantics of CAS and DCAS are described in Section 3.2.)

- Since CAS and DCAS are generally much faster to execute than lock-based critical sections, the possibility of a process failing and leaving the system in an inconsistent state is reduced.

In addition to removing problems caused by lock-based methods, Greenwald and Cheriton [9] further emphasize that:

> "there is a strong synergy between non-blocking synchronization and the design of the Cache Kernel [the name of their multiprocessor operating system kernel] for performance, modularity, and reliability."

Thus, the lock-free method is their "choice" for effective OS implementation rather than just being an "objective in itself." The advantages of lock-free techniques mentioned — performance, modularity, and reliability — can also be attained in red-black tree algorithms.

3.2 Universal Primitives

A synchronization primitive is said to be universal if it can be used to construct any wait-free or lock-free data structure. The wait-free property guarantees that *every* process will complete its operation in a finite number of steps, whereas the lock-free property only guarantees that *some* process will complete its operation in a finite number of steps. Herlihy [13, 15] proves that certain universal primitives do exist.

In showing the universality of a synchronization primitive, Herlihy [15] uses the concept of consensus number. Fischer et al. [8] define the consensus

problem as "reaching agreement among multiple remote processes". Herlihy [15] defines the consensus number of a synchronization primitive as "the maximum number of processes that can solve a consensus problem," using the synchronization primitive. Using the consensus number, Herlihy [15] specifies a necessary and sufficient condition for the universality of a synchronization primitive as follows:

> "An object is universal in a system of n processes if and only if it has a consensus number greater than or equal to n."

This condition implies that if a synchronization primitive's consensus number is infinity, then it can solve a consensus problem in a system for an unbounded number of processes. Therefore, a primitive with infinite consensus number is universal.

Herlihy [15] then shows that there are several synchronization primitives with infinite consensus number, namely, memory-to-memory move and swap, augmented queue, CAS, and fetch-and-cons. Elsewhere, Plotkin [33] also shows the universality of another primitive called the Sticky Byte. Any of these primitives can be used to construct wait-free data structures as well as lock-free data structures. However, due to space or complexity issues and the availability of hardware support, only CAS and its extension, DCAS and its variations, LL/SC (Load-Linked and Store-Conditional), have been popular.

3.2.1 Compare-and-Swap

CAS was introduced in the IBM 370 architecture [19]. The PowerPC [20], the SPARC [21], and the Motorola M68000 [31] architectures all support

```
CAS(vble, copy_of_vble, new) {
  /* Begin Atomic */
    if (vble == copy_of_vble) {
      vble = new;
      return SUCCESS;
    } else
      return FAIL;
  /* End Atomic */
}
```

Figure 3.1: Compare-and-Swap.

CAS. The Pentium architecture [5] supports CAS as the combination of the CMPXCHG (Compare and Exchange) instruction and a lock prefix, which is used to make the CMPXCHG instruction atomic.

Figure 3.1 shows the processing performed by CAS. Before calling CAS, a process makes a copy of the memory location `vble`. The memory location `vble` is modified by CAS to have the specified `new` value only if `vble` has the same value as the copy of `vble` made prior to the CAS. Thus, if another process changes the value of `vble` before the new value can be written, this can be detected and the original process can redo its computation of `new` using the new (up-to-date) value of `vble`.

Using CAS, however, what is known as the ABA problem can occur as follows:

ABA problem: A copy of a memory location `vble` is made by

a process P. Process Q changes `vble` to a different value. Then another process, R, changes `vble` to the same value as the copy of the `vble` that P made. P's CAS operation will then succeed even though there were two updates since the copy was made by P.

In certain applications the ABA problem may not matter while in others it may be significant.

3.2.2 Load-Linked and Store-Conditional

LL/SC is a variation of CAS. Lawrence Livermore National Laboratories first introduced LL/SC in the S-1 project [23]. The PowerPC [20] and the MIPS II [24, 30] architectures support LL/SC.

The semantics of LL/SC are as follows:

Load-Linked: Read a memory location. Many processes can concurrently read the memory location.

Store-Conditional: Store to the memory location read by LL only if the memory location has not been updated by another process since it was read. Only one process can therefore succeed in updating the memory location since the last update to the memory location.

LL/SC has the same semantics as CAS, but LL/SC achieves its purpose using a more convenient combination of two instructions. Either LL/SC or CAS can be used in place of the other, depending on availability as determined by the hardware support of a given architecture.

```
Software_CAS(vble, copy_of_vble, new) {
1.    acquire_lock();
2.    if (vble == copy_of_vble) {
3.      vble = new;
4.      return SUCCESS;
5.    } else {
6.      release_lock();
7.      return FAIL;
8.    }
9.  }
```

Figure 3.2: Software CAS.

3.2.3 Software Compare-and-Swap

The unavailability of hardware support for CAS can be a problem, but there is a software solution. Bershad [3] shows that a software CAS is feasible, using the widely-available Test-and-Set primitive. The software solution relies on an operating systems technique called roll-out.

The algorithm for a software implementation of CAS in Figure 3.2 is made non-blocking with the roll-out technique which can be described as follows:

1. During the execution of the software CAS, if process preemption occurs, the OS releases the lock immediately.

2. When the preempted process is rescheduled to use the processor again, the process resumes at:

```
DCAS(vble1, vble2, copy_of_vble1, copy_of_vble2, new1, new2) {
  /* Begin Atomic */
    if (vble1 == copy_of_vble1 and vble2 == copy_of_vble2) {
      vble1 = new1;
      vble2 = new2;
      return SUCCESS;
    } else
      return FAIL;
  /* End Atomic */
}
```

Figure 3.3: Double-Compare-and-Swap. A process atomically modifies two memory locations instead of one.

Line 1 if it was preempted without changing the data.

Line 4 if it was preempted after changing the data, between Lines 3 and 4.

Naturally, support for a software CAS depends on the availability of an OS provided roll-out mechanism.

3.2.4 Extensions of Compare-and-Swap

CAS can be easily and usefully extended to "double" CAS. Instead of modifying one memory location, two memory locations are atomically modified at the same time using DCAS; see Figure 3.3. The Motorola M68000 [31] architecture supports DCAS.

Not only can we extend CAS to DCAS, but we can also, of course, extend CAS to n-CAS, n being arbitrary, applying the same idea as above. Herlihy and Moss [18] present a mechanism for supporting n-CAS and call their architecture "Transactional Memory". Shavit and Touitou [37], Herlihy et al. [17], and Harris and Fraser [11] here each present a Software Transactional Memory (STM) — a software implementation of Transactional Memory. Israeli and Rappoport [22] here also provide algorithms for an implementation of n-CAS and LL/SCn (the extension of LL/SC), using CAS and LL/SC. Harris et al. [12] describe another set of algorithms for an implementation of n-CAS, using CAS.

It is worth noting that the ABA problem discussed in Section 3.2.1 can be removed using DCAS (or n-CAS, or LL/SCn with $n = 2$). A version number can be associated with each data structure or a part of the data structure, using an extra memory location, and then the version number can be tested and incremented using, for example, DCAS. Future updates will have different version numbers, so a change from A to B and then back to A may be detected.

3.3 Existing Lock-Free Data Structures

To find a general method (also called a universal method) to implement a lock-free data structure, Herlihy [14] devised the first general method to convert any sequential data structure into a wait-free concurrent data structure. Other general methods [1, 16, 34, 40] followed. General methods, however, are generally accepted as not being efficient when compared to lock-based

algorithms. For instance, LaMarca [25] shows that Herlihy's [14, 16] and Barnes' [1] general methods' performance are worse than a lock-based solution (spin-lock) in both his analysis and experimental results.

In contrast, some lock-free data structures implemented with data-structure-specific algorithms have shown better performance than the corresponding lock-based algorithms.

Michael and Scott [29] perform a series of tests on several lock-free data structures and their results show that the lock-free data structures that they considered in the tests perform better than their lock-based counterparts. The lock-free data structures considered in the testing include link-based queues by Michael and Scott [28] and Prakash et al. [35], a link-based stack by Treiber [39], an optimized version of a stack resulting from applying Herlihy's method [16], and an optimized version of a skew heap resulting from applying Herlihy's method [16]. In the testing, Michael and Scott [29] also considered an LL/SC implementation of counters, as well as quicksort and traveling salesman applications using some of the tested lock-free data structures. All the results showed better performance with the lock-free data structures than with their lock-based counterparts.

Another positive result for a lock-free data structure implemented with data-structure-specific algorithms is presented in Greenwald and Cheriton's [9] implementation of linked lists. They perform simulation-based testing on their linked list, which uses DCAS, and the results show that the lock-free linked list performs better than its lock-based counterpart.

3.4 Existing Lock-Free Red-Black Tree Algorithms

3.4.1 Software Transactional Memory (STM)

Herlihy et al. [17] and Harris and Fraser [11] implemented concurrent operations for red-black trees using their STMs.

The implementation details of Herlihy et al.'s STM and Harris and Fraser's STM are different. However, both Herlihy et al. and Harris and Fraser provide similar sets of application programming interfaces to do transactional operations. The interfaces naturally include functions to start, commit, and abort transactions.

In this section, we describe Harris and Fraser's implementation of red-black trees. Harris and Fraser's implementation of concurrent operations on red-black trees, as well as their STM, are available online at:
http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free.

The idea behind Harris and Fraser's implementation of red-black trees is that each process involved in updating the tree will optimistically perform all of the operations in one transaction (similar to transaction processing in databases). The nodes that are involved will store their old data, new data, and their version numbers in the STM. After a process completes its operations, the operations recorded in the STM will either be committed and take effect, or be aborted and redone. One example where a transaction must abort is the case when the version number of one of the recorded nodes has been changed by another process.

The STM approach makes implementing concurrent red-black trees easy.

However, at the time of committing a transaction, all the nodes in the tree that are recorded for updates during a transaction have to be acquired by the process doing the transaction using a series of CAS operations. This requirement is needed to effectively make a permanent update for the transaction. However, the nodes involved may comprise nodes located anywhere from a leaf to the root and cover a large area of the tree. Hence, the re-execution cost due to contention may be high since all the tentative updates to a transaction will be lost when the transaction is aborted.

Our algorithm will reduce the current working area of a process to just a few nearby nodes. In our algorithm, each process works in its local area, which consists of small number of nodes. Therefore, more processes may run concurrently in a red-black tree with less re-execution cost.

3.4.2 Ma's Insertion Algorithm

Ma [26] presents a lock-free insertion algorithm for concurrent red-black trees. Ma's algorithm is based on Cormen et al.'s sequential insertion algorithm [4] with the addition of the local area concept and lock-free techniques. In this section, we explain the concept of "local area" and the lock-free techniques used in Ma's algorithm.

Local Area

Ma introduces an extra flag field to each node, named `flag`. When a flag of a node is set by a process, the process gains control of the node.

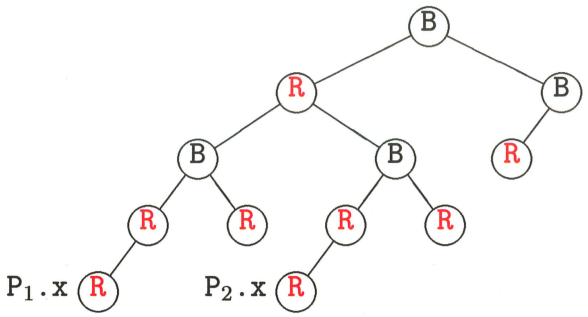
The local area is a set of nodes that a process has full control of. During insertions, the local area consists of `x`, `x`'s parent, `x`'s grandparent, and

x's uncle. Here, x refers to the new node at the beginning of the insertion. The determination of the local area for insertions is explained later in Section 4.1.1. A process obtains control of its local area by setting the flags of all of the nodes in its local area.

If a process does not have exclusive access to its local area and only has the control of one node, a problem illustrated in Figure 3.4 can occur and an inconsistent tree may result. In Figure 3.4(a), two processes, P_1 and P_2, are both in insertion Case 1. After their Case-1 processing, P_1 and P_2 move x two levels up as in Figure 3.4(b). P_1 may decide that it is in insertion Case 3 because P_1.x's uncle is black and P_1.x, its parent, and grandparent form a straight line. Similarly, P_2 may decide that it is in insertion Case 2 because P_2.x's uncle is black and P_2.x, its parent, and grandparent form a zig-zag. At this point, a problem can occur. If P_1 performs Case-3 processing before P_2 does its processing, P_2 will have a different parent; however, P_2 will perform its Case-2 processing without this information. Conversely, if P_2 performs Case-2 processing before P_1 does its processing, P_1 will have a different uncle; yet, P_1 will perform its Case-3 processing without knowing that the change has occurred.

With the local area concept, if several processes in a nearby neighbourhood have overlapping local areas, only one process which successfully obtains all the flags of its local area will be able to proceed to determine which case it is in and then perform the case processing. In our example, P_1 in Figure 3.4(b) may obtain the flags of its local area before P_2 does. P_1 will determine that it is in Case 3, perform Case 3, and exit. Then P_2 may obtain the flags of P_1's updated nodes, determine which case it is in, and do its

(a) Processes P_1 and P_2 are in insertion Case 1.



⇓

(b) Each process did Case-1 processing and moved its x two levels up.

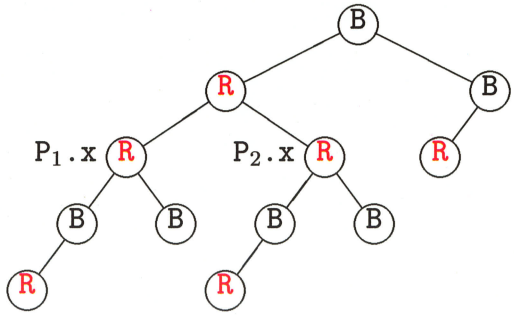


Figure 3.4: Without the local area protection, processes can interfere with each other. (This figure is based on Figures 4.1 and 4.2 in Ma [26].)

processing. Alternatively, P_2 may obtain the flags of its local area before P_1 does. P_2 will determine it is in Case 2, perform Cases 2 and 3, and exit. Then P_1 may obtain the flags of nodes in the updated local area, determine which case it is in, and do its processing. Therefore, the problems such as a process, say P_1, altering the state of nodes in the local area of another process, say P_2, and changing the case P_2 is in while P_2 is performing its case are removed.

Moving the Local Area

After a process performs Case 1, it moves its x up. In such a case, the process also moves its local area up. It first obtains the flags of nodes in its new local area, moves x up, and then releases the flags that it set for the nodes in its old local area. When the process is done rebalancing the tree, it only has to release the flags of the nodes in its "current" local area.

Lock-Free Primitives Used

Ma uses lock-free synchronization primitives, namely, CAS, DCAS and 3-CAS, to achieve concurrency. To set the flags of nodes to obtain the local area, Ma uses DCAS. One variable of the DCAS operation is used for checking if the node whose flag we are trying to set has been changed by another process since a copy was made by us. The other variable is the address of the flag to be set. 3-CAS is used in rotations. The use of 3-CAS for rotations in Ma's insertion algorithm is described in Section 4.1.1.

3.5 Conclusions Drawn from the Related Work

3.5.1 Reasons for Choosing Lock-Free Synchronization

Lock-free synchronization methods do not incur problems such as deadlock, priority inversion, unnecessary delay, and convoying that are associated with lock-based synchronization methods. For these reasons, some researchers have used them in the design and implementation of multiprocessor operating system kernels. Further, in parallel programming, the delays caused by blocking when a process attempts to acquire a lock held by another process can be highly disruptive to overall progress. This is because processes commonly interact frequently and a blocked process will quickly delay other processes as well. It is therefore better to periodically incur the need to recompute a partial result than to pay the high cost of blockage. Hence, we have chosen to use lock-free synchronization.

3.5.2 Reasons for Choosing Compare-and-Swap

Both CAS and LL/SC are universal and have equivalent semantics. CAS, DCAS, and LL/SC are supported by several commercial machine architectures. Also, software and hardware solutions for n-CAS and LL/SCn exist. Hence, either CAS or LL/SC can be used. To maintain consistency with the bulk of the available literature, we have chosen to use CAS, DCAS, and n-CAS for our concurrent red-black tree algorithms.

3.5.3 Reasons for Incorporating Ma's Insertion Algorithm

Many general methods have been introduced for concurrent lock-free data structures. However, they are generally accepted as being inefficient compared to their lock-based counterparts. In contrast, some data-structure-specific lock-free algorithms have shown better performance results than their lock-based counterparts. Therefore, we have chosen to design a data-structure-specific algorithm for the data structure of our choice — red-black trees.

In the STM approach, a process's transaction may cover a large area of the tree, from the root to leaf in the process's insertion or deletion path. Ma's insertion algorithm uses the concept of local area which bounds the current working area of a process to several neighbouring nodes. Hence, the use of the local area concept may allow more concurrency and have lower re-execution cost. Therefore, we have chosen to incorporate Ma's red-black tree insertion algorithm into our algorithm.

Chapter 4

Design of the Concurrent Operations of Red-Black Trees and Proof of Correctness

In this thesis, an existing efficient lock-free insertion algorithm, Ma's insertion algorithm [26] (see Section 3.4.2), is adapted with the addition of intention markers as a starting point. (The intention markers will be defined in this chapter in Section 4.2.) The search algorithm for red-black trees is not discussed in detail because it is very similar to the steps involved in locating the insertion point in the insertion algorithm or those for locating the key to be deleted in the deletion algorithm.

For our deletion algorithm, we incorporate the following techniques:

- Cormen et al.'s sequential deletion algorithm [4] (see Section 2.4.4) as a base,

- the local area concept of Ma's lock-free insertion algorithm (see Section 3.4.2), and
- intention markers.

The use of intention markers enables us to use DCAS alone for both insertions and deletions. This is an improvement over the algorithm for insertion presented by Ma because DCAS is widely available in hardware while 3-CAS (used by Ma's algorithm) is not; see Section 3.2.4.

This chapter is organized as follows. We first define the local area for insertion and deletion operations. Then we discuss intention markers in terms of their necessity and the number of markers required. We next discuss an observation regarding setting the initial local area, which is later used in the proof of correctness of our algorithm design. Then we discuss a choice made to increase concurrency. Finally, we show a proof of correctness of our algorithm design.

4.1 Defining the Local Area

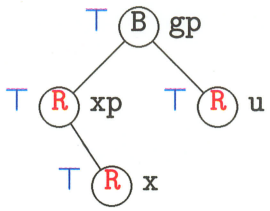
In this section, we explain our interpretation of how Ma [26] defined the local area for insertions, and then we apply the same principle to deletions.

4.1.1 The Local Area for Insertions

Nodes in the Local Area

Defining the local area involves deciding which nodes constitute a local area (of tree nodes where operations may result in changes) and therefore which

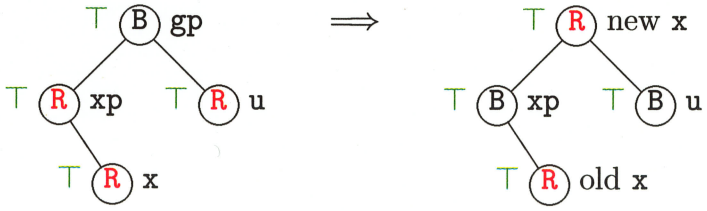
(a)⊤ indicates a node that the process must access to determine that it is in insertion Case 1.



Indications of Insertion Case 1:

1. `x->p` and `x`'s uncle is red.

(b)⊤ indicates a node that the process must access to perform insertion Case 1.

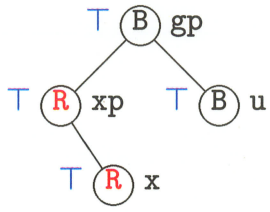


Operations performed in insertion Case 1:

1. `x->p->colour = black`.
2. `xUncle->colour = black`.
3. `x->p->p->colour = red`.
4. `x = x->p->p`.

Figure 4.1: Nodes that a process must access during insertion Case 1.

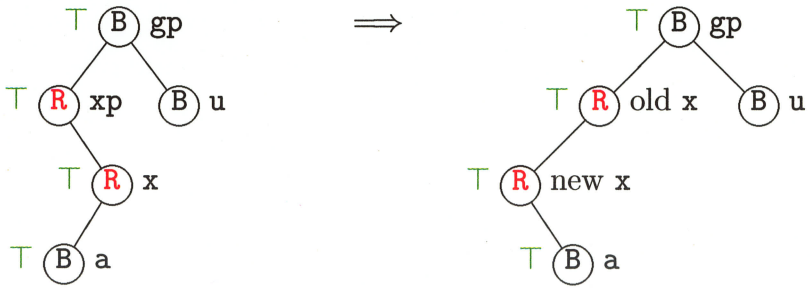
(a)⊤ indicates a node that a process must access to determine that it is in insertion Case 2.



Indication of insertion Case 2:

1. `x`'s uncle is black, and `x->p` is red.
2. `x`, `x->p`, and `x->p->p` make a zig-zag shape.

(b)⊤ indicates a node that the process must access to perform insertion Case 2.

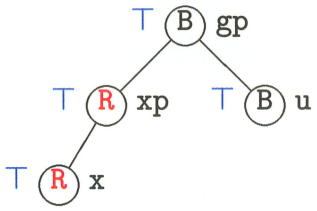


Operations performed in insertion Case 2:

1. `x = x->p`.
2. `Left_Rotate(x)`.

Figure 4.2: Nodes that a process must access during insertion Case 2.

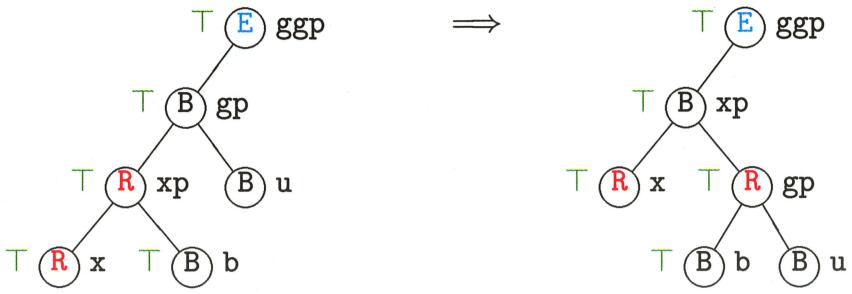
(a)⊤ indicates a node that the process must access to determine that it is in insertion Case 3.



Indications of insertion Case 3:

1. `x`'s uncle is black, and `x->p` is red.
2. `x`, `x->p`, and `x->p->p` form a straight line.

(b)⊤ indicates a node that the process must access to perform insertion Case 3.

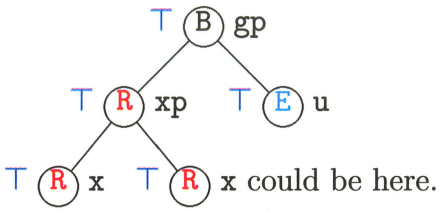


Operations performed in insertion Case 3:

1. `Right_Rotate(x->p->p)`.
2. `x->p->colour = black`.
3. `x->p->right->colour = red`.

Figure 4.3: Nodes that a process must access during insertion Case 3.

(a)⊤ indicates a node that a process must access to determine which case it is in.



(b)⊤ indicates a node that the process make when performing an insertion case.

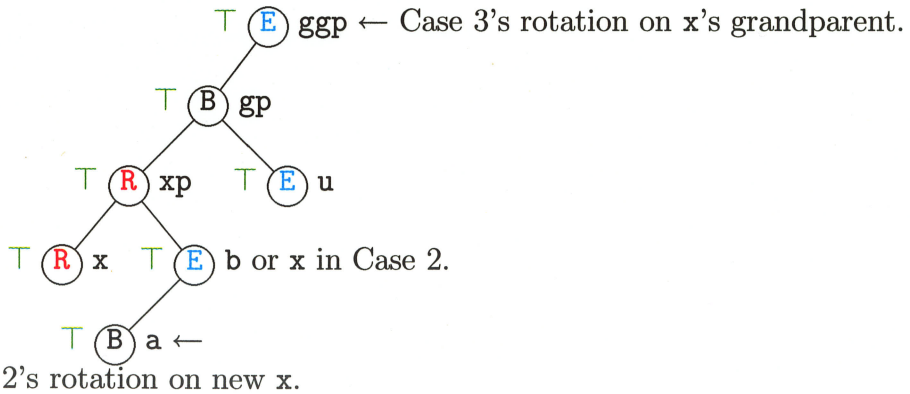


Figure 4.4: The union of possible accesses for all insertion cases.

collection of nodes' flags a process needs to set. Our objective is to include as small a number of nodes as possible to allow more potential concurrency.

The best way to show which nodes we should include is by drawing pictures. Two pictures are provided for each case. One picture shows the nodes that a process accesses to determine which case it is in. The other picture shows the nodes that a process must access while performing the case. These pairs of pictures appear in Figures 4.1 to 4.3. There are "mirror image" cases of the three cases shown depending on whether `x->p` is a left or a right child. Thus, for determining each case, `x`, `x->p`, and `x->p->p` must be accessed. The mirror image cases are symmetric to the three cases shown in Figures 4.1 to 4.3.

Figure 4.1 is for insertion Case 1. Part (a) shows the nodes that a process accesses to determine that it is in Case 1. The blue ⊤ marks indicate the nodes that the process accesses. The pseudocode below the picture shows the operations performed.

Part (b) of Figure 4.1 shows a similar picture for the Case-1 processing. The green ⊤ marks indicate the nodes that the process accesses. The left side of the picture is "before", and the right side is "after" the Case-1 processing. The Case-1 processing is described by the pseudocode below the picture.

Figures 4.2 and 4.3 depict the nodes that a process accesses for insertion Case 2 and Case 3, respectively.

Figure 4.4, which shows the union of access for all three insertion cases, is what we are especially interested in. As the example shown in Figure 3.4 (page 52) showed, we must include all the nodes that a process accesses to determine the case in the process's local area. Therefore, the nodes with

$\top$ marks in Figure 4.4(a) — x, x's parent, grandparent, and uncle — are included in the local area for the insertion algorithm; and only these nodes constitute a general insertion operation's local area.

Use of 3-CAS in Ma's Algorithm

In Figure 4.4(b), we can see that a process accesses three more nodes (at most two at a time) than what is specified by the local area. Two nodes are from Case 3 — x's great grandparent and x's sibling — due to Case 3's rotation on x's grandparent. The other node is from Case 2 — x's left child — due to Case 2's rotation on the new x which is the old xp.

Ma [26] uses 3-CAS to perform a rotation. In each rotation, four nodes are accessed. For example, consider the left rotation of insertion Case 2 performed on node xp in Figure 4.2(b). Nodes x, xp, gp, and a are accessed. However, node a is not in the local area of the process performing the rotation, and the process does not hold the flag of the node. In the 3-CAS operation, two shared memory variables are used to link node xp and node a, and the other variable is used to verify that node a has not been modified by another process.

At each rotation performed as part of an insertion, there is one node that a process does not hold the flag of — x's sibling in Case 3 and x's left child in Case 2. A process in Case 3 temporarily obtains the flag of x's great grandparent before doing a rotation, thus leaving the flag of one of the two nodes (that are outside of the local area but needed for the rotation) still to be obtained by the rotating process.

The existence of a node that a process does not hold the flag of, but which

is involved in a rotation, is the reason for requiring 3-CAS in Ma's algorithm. Temporarily acquiring the flag of x's sibling (in Case 3) or x's left child (in Case 2) cannot be done prior to a rotation. For insertion Case 3, consider the example shown in Figure 4.5(a) where P_1, in blue, and P_2, in green, are both in insertion Case 3. Suppose we allow the temporary acquiring of x's sibling in Case 3. P_1 is trying to obtain P_1.x's sibling, which P_2 holds, while P_2 is trying to obtain P_2.x's great grandparent, which is held by P_1. This situation would result in deadlock.

Similarly, the temporary acquiring of x's left child in Case 2 cannot be achieved. For an illustration of this problem, consider Figure 4.5(b). In the figure, P_1 is in insertion Case 2, and P_2 is in insertion Case 3. If we allow P_1.x to temporarily acquire its left child, then we have deadlock since P_1 is trying to obtain P_1.x's left child, which P_2 holds, and P_2 is trying to obtain P_2.x's great grandparent, which is held by P_1.

Using 3-CAS, as long as P_1.x's sibling in Figure 4.5(a) does not change (and this node is guaranteed to not change in this situation since P_2 is busy-waiting), P_1 can perform its rotation successfully. Also, as long as P_1.x's left child in Figure 4.5(b) is unchanged (again, the node will not be changed), P_1 can do its rotation successfully.

Using only DCAS

In our algorithm, we will be able to use only DCAS since the distance between processes provided by introducing intention markers into Ma's insertion algorithm will allow us to temporarily obtain the flags of x's sibling for Case 3 and x's left child for Case 2. As for obtaining x's great grandparent in Case 3,

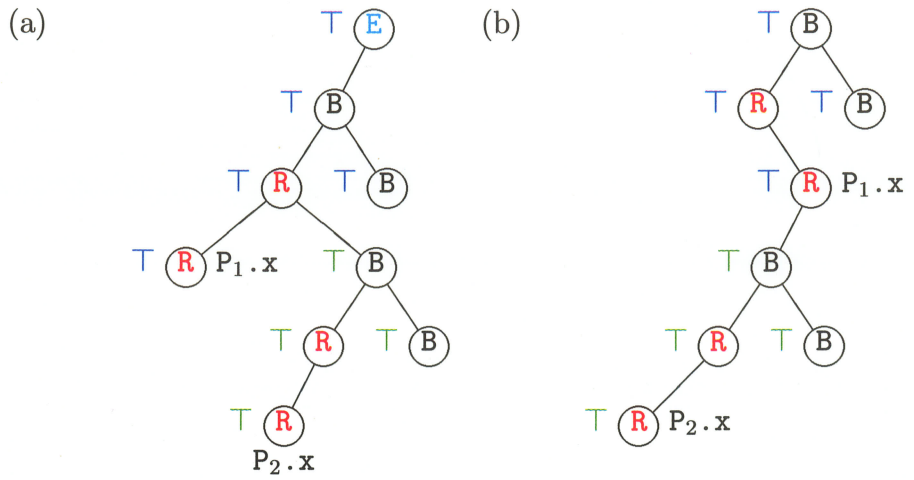


Figure 4.5: Reason for using 3-CAS in Ma [26]'s insertion algorithm, illustration.

we use the following technique.

As we will see in Section 4.4, a process that is doing a rotation will release its four intention markers before doing the rotation, to increase concurrency. When a process releases its markers, it first obtains the flags of the nodes on which markers are held, then releases the markers, and finally releases the flags of the nodes that contained the markers. The process releasing markers can be constrained to retain the flag of the lowest node that contained its markers. Retaining this flag effectively does the temporary acquiring of the flag of x's great grandparent for Case 3. Since the process already obtained the flag, retaining it is more efficient than releasing and re-obtaining it again.

A discussion of the safe distance between processes provided by the intention markers for temporary acquiring of x's sibling and x's left child is included as part of Section 4.1.2's explanation of why we use only DCAS.

Hence, rotations in insertions do not need to use 3-CAS anymore since we now have all the necessary flags required for a rotation. We only need to use DCAS to obtain the necessary flags for the rotations.

4.1.2 The Local Area for Deletions

Nodes in the Local Area

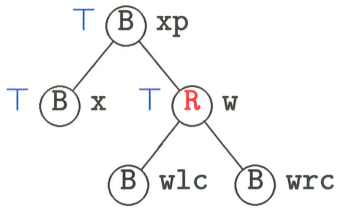
We now define the local area for a process doing a deletion in a similar manner. Figures 4.6 to 4.9 show the pictures corresponding to those for the insertion cases. For each of the four deletion cases, we show the nodes that a process must access to determine that it is in the corresponding case and the nodes that a process must access while performing the corresponding case. There are "mirror image" cases depending on whether `x` is a left or a right child. Thus, to determine which case a process is in, the process must access `x` and its parent. The mirror image cases are symmetric to the four cases shown in Figures 4.6 to 4.9.

Figure 4.10(a), which discusses the union of access for all four deletion cases, shows that we must include `x`, `x`'s parent, `x`'s sibling `w`, and `w`'s children for the same reasons as in the insertion cases. These nodes constitute a general deletion operation's local area. (We call `w`'s left and right child `wlc` and `wrc`, respectively.)

Using Only DCAS

As shown in Figure 4.10(b), a process accesses two more nodes than what is specified in part (a) of the figure. These two nodes are `x`'s grandparent (in

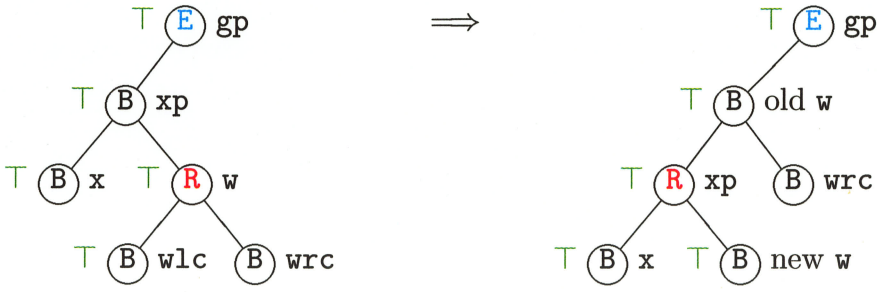
(a)⊤ indicates a node that the process must access to determine that it is in deletion Case 1.



Indications of deletion Case 1:

1. `w` is red, and `x` is black.

(b)⊤ indicates a node that the process must access to perform deletion Case 1.

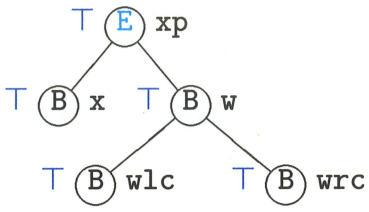


Operations performed in deletion Case 1:

1. `w->colour = black`.
2. `x->p->colour = red`.
3. `Left_Rotate(x->p)`.
4. `w = x->p->right`.

Figure 4.6: Nodes that a process must access during deletion Case 1.

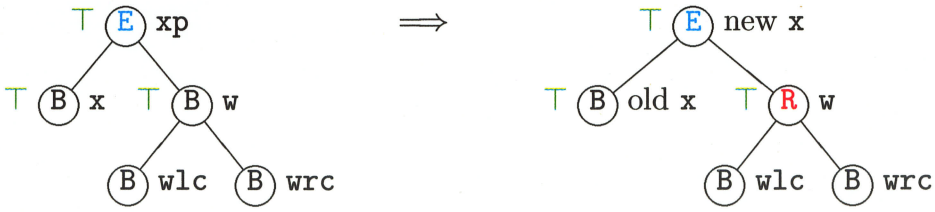
(a)⊤ indicates a node that the process must access to determine that it is in deletion Case 2.



Indications of deletion Case 2:

1. `w`'s children are both black.
2. `x` and `w` are black.

(b)⊤ indicates a node that the process must access to perform deletion Case 2.

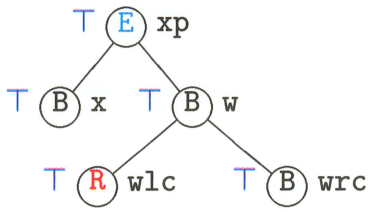


Operations performed in deletion Case 2:

1. `w->colour = red`.
2. `x = x->p`.

Figure 4.7: Nodes that a process must access during deletion Case 2.

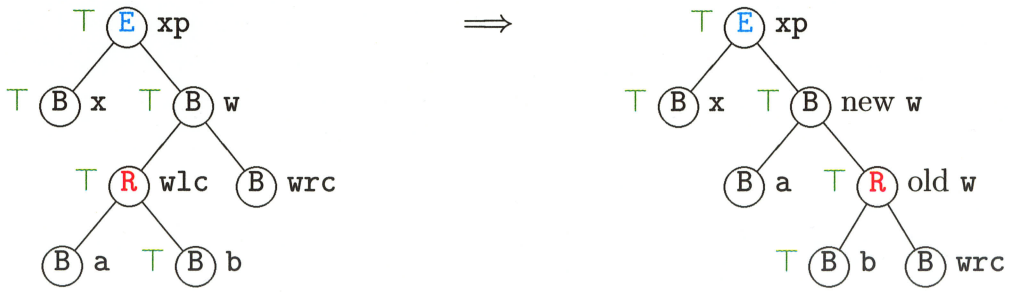
(a)⊤ indicates a node that the process must access to determine that it is in deletion Case 3.



Indications of deletion Case 3:

1. `w` and `x` are black.
2. `w`'s children are not both black.
3. `w`'s right child is black.

(b)⊤ indicates a node that the process must access to perform deletion Case 3.

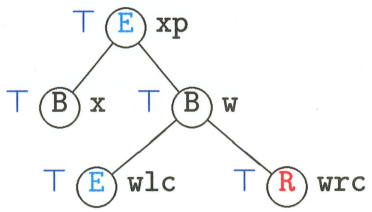


Operations performed in deletion Case 3:

1. `w->left->colour = black`.
2. `w->colour = red`.
3. `Right_Rotate(w)`.
4. `w = x->p->right`.
5. Do Case 4.

Figure 4.8: Nodes that a process must access during deletion Case 3.

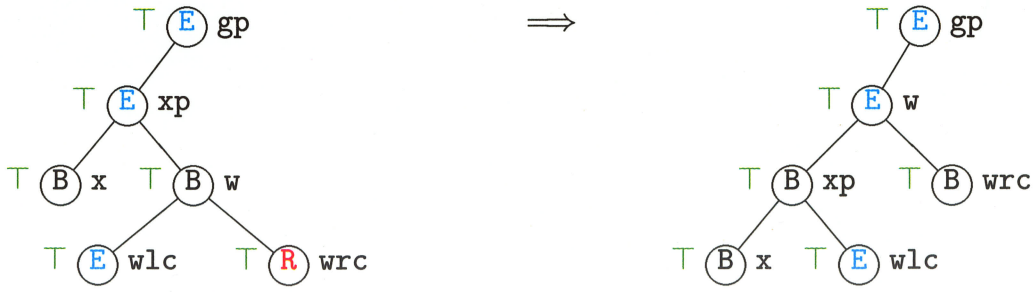
(a)⊤ indicates a node that the process must access to determine that it is in deletion Case 4.



Indications of deletion Case 4:

1. `w` and `x` are black.
2. `w`'s children are not both black.
3. `w`'s right child is red.

(b)⊤ indicates a node that the process must access to perform deletion Case 4.

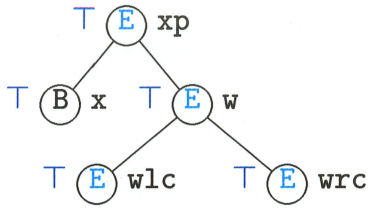


Operations performed in deletion Case 4:

1. `w->colour = x->p->colour`.
2. `x->p->colour = black`.
3. `w->right->colour = black`.
4. `Left_Rotate(x->p)`.
5. `x = root`.

Figure 4.9: Nodes that a process must access during deletion Case 4.

(a)⊤ indicates a node that a process accesses to determine which case it is in.



(b)⊤ indicates a node that a process accesses to perform a deletion case.

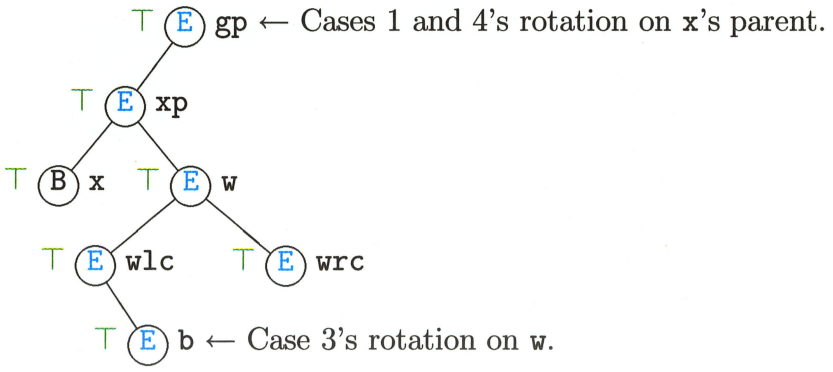


Figure 4.10: The union of possible accesses for all deletion cases.

Cases 1 and 4) and `wlc`'s right child (in Case 3). As in insertion, a process that is doing a rotation retains the flag of the lowest node that contain its markers when the process releases its flags. (In addition, in deletion Case 3, if the process did not release its markers by doing Case 1, then it must retain the flags of the two lowest nodes for the Insert-Jump-Ahead rule (page 145) which will be discussed later in Section 4.6.3. Further, if a flag is retained in Case 1, the flag will not be released immediately for the possible application of the Insert-Jump-Ahead rule in Case 4.) The retention of the flag ensures that the process doing a rotation has the flag of `x`'s grandparent. Moreover, introduction of intention markers makes a process able to temporarily obtain `wlc`'s right child.

As an example to show how the intention markers allow us to maintain a safe distance between processes for the temporary acquiring of `wlc`'s right child, we use a transformation of deletion Case $1 \rightarrow 3 \rightarrow 4$ shown in Figure 4.11. We select this case among all possible case transformations (listed later in Figure 4.17) because this transformation brings two processes the closest together (i.e., it is a worst-case scenario). Each of Cases 1, 3, and 4 pulls a process located lower in the same deletion path up one level — in this case, a total of three levels. (For an illustration of the "pull-up" concept, refer to Figure 4.11. In Figure 4.11(b), for example, the highest marker of $\mathtt{P}_2$ denoted by a green check mark is on $\mathtt{P}_1\mathtt{.w}$. After $\mathtt{P}_1$ performed Case-1 processing, part (c) shows that the highest marker of $\mathtt{P}_2$ is on $\mathtt{P}_1\mathtt{.xp}$; thus, it shows that $\mathtt{P}_2$ is "pulled up" one level to $\mathtt{P}_1\mathtt{.xp}$ by Case 1.)

In Figure 4.11(a), $\mathtt{P}_1$ is in deletion Case 1, and $\mathtt{P}_2$ is in insertion Case 3. The blue check marks indicate $\mathtt{P}_1$'s intention markers, and the green check

marks indicate P_2's intention markers. The blue $\top$ symbol next to a node indicates that P_1 has the flag of the corresponding node. And the green $\top$ symbol means the same for P_2. A process is allowed to place a flag on a node that has its marker field set. (See Section 4.2 for a full description of intention markers.) In part (b) of the figure, P_1 has released its markers and left the flag on the node where the lowest markers was placed. P_2 must release flags of nodes that contain its markers before doing its rotation, but P_1 holds the flags of nodes where two of P_2's markers are placed. P_2 must therefore wait until P_1 finishes.

We are especially interested in Figure 4.11(c) because P_1 in deletion Case 3 needs to temporarily acquire `wlc`'s right child (the node in Figure 4.11(c) pointed at by the $\longleftarrow$ sign). P_1 can obtain the flag of `wlc`'s right child because P_2 is still trying to obtain the flags of nodes that have P_2's markers. When P_2 fails to obtain flags of all four nodes that have its markers, it backs off and releases all the flags (of the nodes that have its markers) obtained so far and then restarts obtaining the flags of nodes that have its markers.

P_1 does Case 4 immediately following Case 3, but we omit the picture of Case 4. In Case 4, P_1 does a left rotation on `xp`. Case 4 does not involve temporarily obtaining flags not in the local area. Therefore, as in an insertion, we do not need 3-CAS to do rotations in the case of deletions.

Fortunately, insertions do not cause a process to get pulled up three levels from below and, therefore, temporarily acquiring `x`'s sibling in insertion Case 3 and `x`'s left child in insertion Case 2 are both safe. Therefore, we only need to use DCAS in our concurrent algorithms for red-black trees.

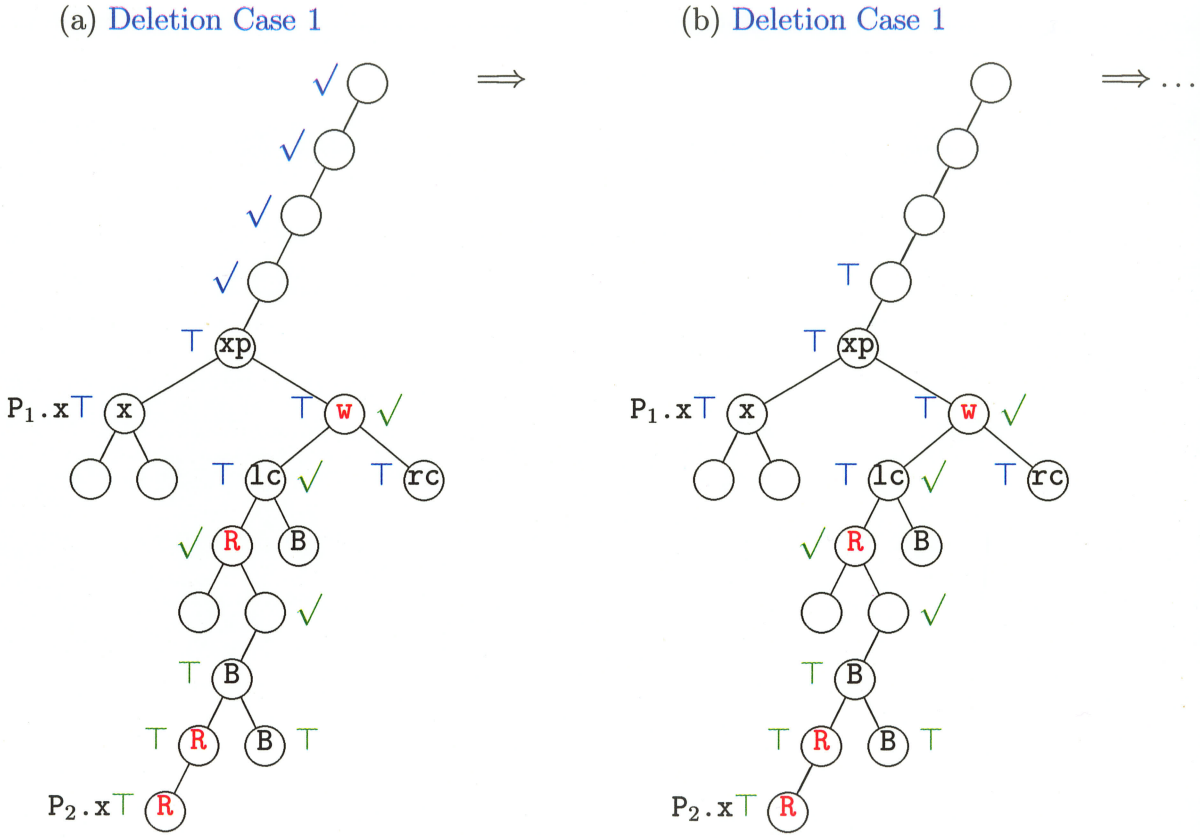


Figure 4.11: Deletion Case 1 → 3 → 4 with four intention markers. The insertion process, in green, is in Case 3. (Picture 1 of 2.)

(c) Deletion Case 3

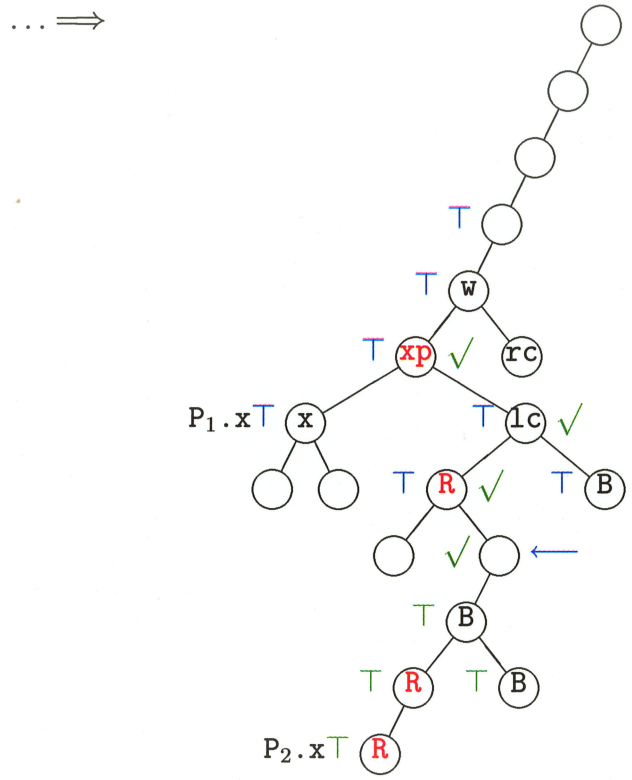


Figure 4.11: Deletion Case $1 \rightarrow 3 \rightarrow 4$ with four intention markers. (Picture 2 of 2.)

4.2 Intention Markers

4.2.1 Definition of Intention Markers

An intention marker is an integer field in a node (see Figure 5.1 in Section 5.2 for node structure definition) that is set by a process to indicate that the process intends to move its local area up to the node. By "sets a marker" we mean that the process places its process ID in the marker field of a node.

The main purpose of intention markers is to ensure a safe distance between any two processes so that a process is guaranteed to be able to proceed. Figure 4.12 shows a deletion and an insertion process each having four intention markers. The intention markers are represented as check marks. A $\top$ mark beside a node indicates that the flag of the corresponding node is held by the process.

A process, say $\mathtt{P}_1$, can set a node's flag if the flag field of the node is not already set. If the marker field of the node is set, $\mathtt{P}_1$ can still set the flag of the node as long as the flag field is not already set. In other words, a flag can be set on a node only if the node's flag field is clear.

$\mathtt{P}_1$ can set the marker field of a node if no other process has set the node's flag or the marker field. In other words, a marker can be set on a node only if the node's flag and marker fields are both clear.

Now we discuss the reason for introducing intention markers.

4.2.2 Necessity of Intention Markers

For insertions alone, we did not need an intention marker. However, for insertions to work with deletions, the use of local areas alone is not sufficient

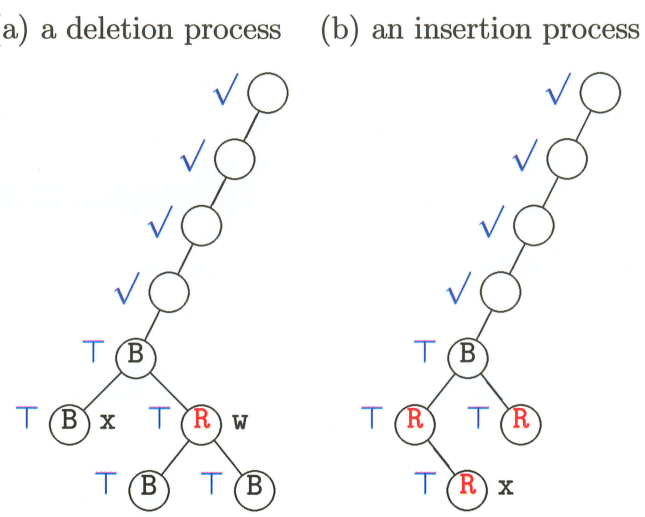


Figure 4.12: Intention markers are the four nodes above the local area.

to guarantee correct operations.

We now provide a couple of examples *without* intention markers to illustrate the need for intention markers. First, consider the example in Figure 4.13. In part (a) of the figure, processes P_1 and P_2 are both in deletion Case 2; and after their Case-2 processing (see part (b) of the figure), each process needs to move `x` one level up. Before moving up, a process needs to obtain the flags of its new local area. The node that should be the new P_1`.w` when P_1`.x` moves up, however, is currently held by P_2 as P_2`.x`'s parent, so P_1 will be waiting for P_2 to release the corresponding flag. The same is true for P_2 — the node that should be the new P_2`.w` when P_2`.x` moves up is currently P_1`.x`'s parent so P_2 will be waiting for P_1 to release P_1`.x`'s parent. Therefore, neither process can proceed, and they are deadlocked.

Figure 4.14 shows another example similar to that of Figure 4.13. In

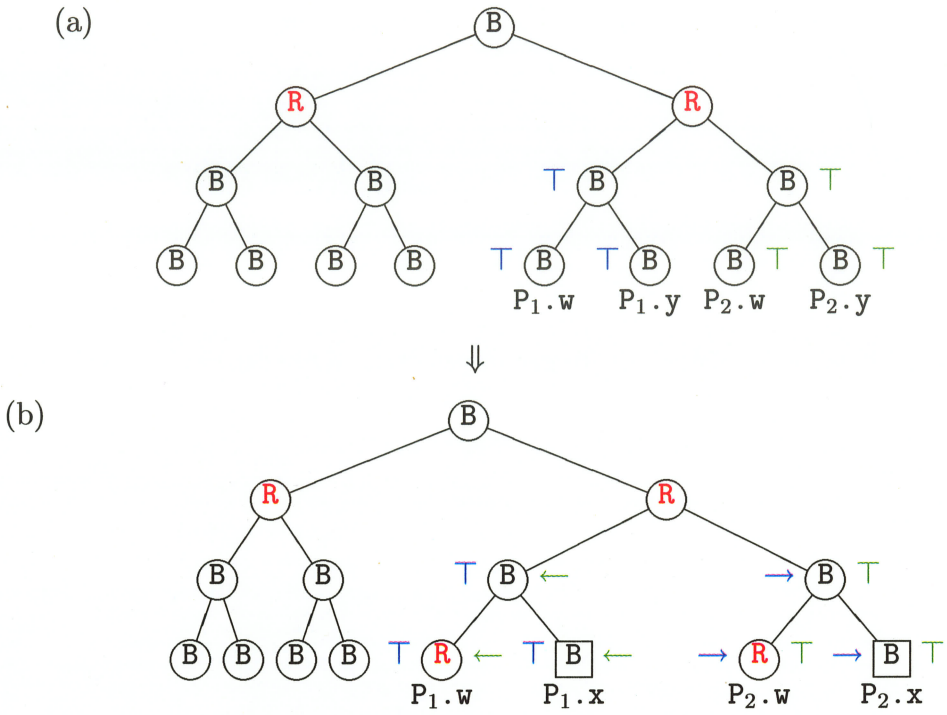


Figure 4.13: Deletion Case 2 and Deletion Case 2. The nodes marked with the ← sign are nodes held by P_1 but are needed by P_2. The → sign point to nodes held by P_2 but are needed by P_1.

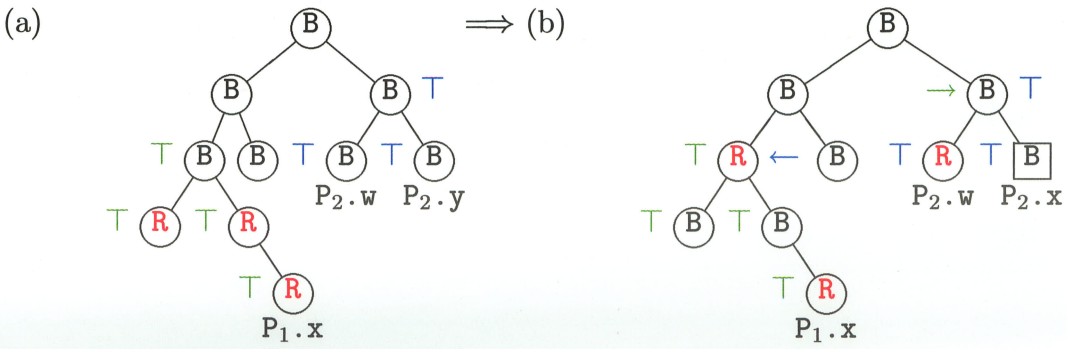


Figure 4.14: Insertion Case 1 and Deletion Case 2. Deadlock.

Figure 4.14(a), P_1 is in insertion Case 1, and P_2 is in deletion Case 2. After processing their respective cases (refer to Figure 4.14(b)), P_1 will try to obtain the flags on its new local area and move P_1.x two levels up. However, the node that will be the new uncle of the new P_1.x when P_1.x moves up is already held by P_2. Meanwhile, P_2 has to obtain the flags on its new local area, too. However, the node that will be the new P_2.w's left child when P_2.x moves one level up is already held by P_1. Hence, the processes are, again, deadlocked.

These examples show the necessity of two processes having a safe distance between them so that the above problems do not happen. We provide that safe distance through the use of intention markers. For instance, in Figure 4.13(a), if P_1 had an intention marker at P_1.y's grandparent, P_2 would not have been able to put its marker there and therefore would not have been able to be where it is at in the picture.

One intention marker, however, is not enough to provide a safe distance. For example, Figure 4.14's problem cannot be resolved with just one marker. In fact, as we will show, we need four markers.

4.2.3 The Number of Markers Required

Our objective is to have the fewest possible intention markers because fewer intention markers will allow for more concurrency. We began with one intention marker and went through examples of all possible case transformations (listed in Figure 4.17) by drawing pictures and placing processes in different cases at critical locations. We increased the number of intention markers if we found examples that showed that the chosen number of markers was

insufficient to ensure a safe distance.

We discovered that four markers are necessary and sufficient.

The Necessity of Four Markers

Figure 4.15 provides an example where neither two nor three markers will work. In the figure, the blue process, P_1, goes through deletion Cases 2, 1, 3, and 4 in sequence. The green process, P_2, is in insertion Case 1. Each process has three intention markers.

P_1 starts out in deletion Case 2 (refer to Figure 4.15(a)), and then it moves one level up. Now P_1 is in Case 1; see Figure 4.15(b). P_1 performs Case-1 processing and switches into Case 3. As a result, P_2 has been pulled one level closer to P_1; see Figure 4.15(c). P_1's performing Case 3 pulls P_2 one more level closer to P_1 and switches P_1 into Case-4 processing; see Figure 4.15(d). At this point, P_1 must obtain its new `wlc` which is, unfortunately, already held by P_2.

Meanwhile, throughout the steps of Figure 4.15, P_2 could not proceed any further because it needed to move two levels up. To move up, P_2 first needs to set two more intention markers at the top of the highest node of the four nodes that contain P_2's markers, but it could not set them because P_1 either had placed P_1's markers there or held the flags on the nodes.

Therefore, neither the deletion process nor the insertion process can move forward.

With four intention markers, we do not have the problem illustrated in Figure 4.15. Figure 4.16 shows the same situation as in Figure 4.15 but now with four markers. As we can see in Figure 4.16(d), now P_1 in deletion Case 4

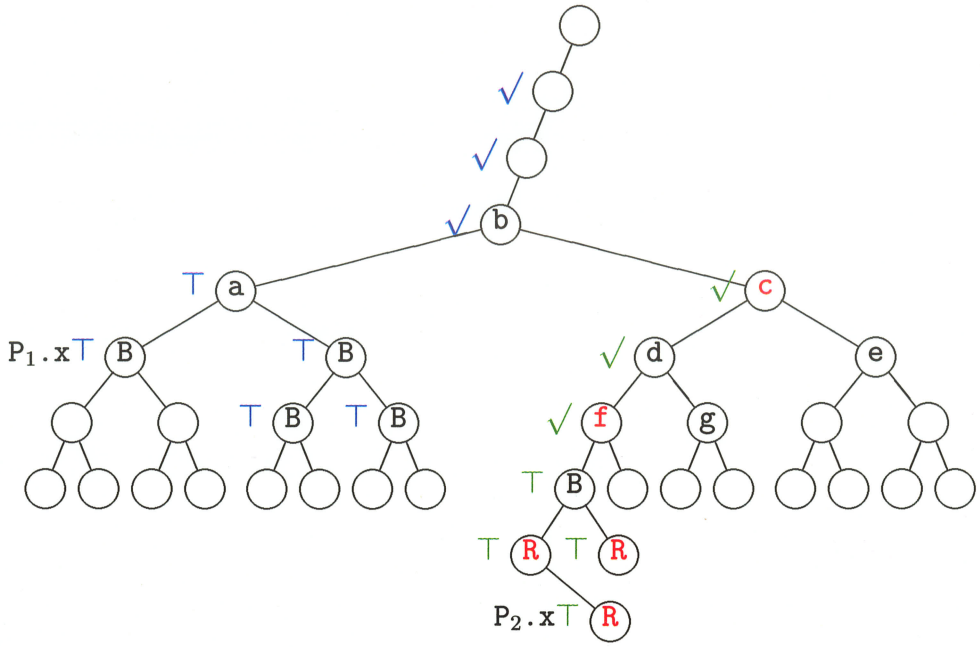
can obtain its new `wlc`'s flag and exit nicely because the processes have one extra distance between them than before.

The Sufficiency of Four Markers

As mentioned earlier in Section 4.1.2, from the list of all the possible transformations shown in Figure 4.17, Case 2 $\cdots \rightarrow 1 \rightarrow 3 \rightarrow 4$ or Case $1 \rightarrow 3 \rightarrow 4$ are the only transformations that bring two processes the closest to each other (because each of Cases 1, 3, and 4 pulls a process located below in the same deletion path one level up). Hence, these are the worst-case scenarios. We have shown in Figure 4.16 that four markers are sufficient for the particular case of Case $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$. (Figure 4.11 also serves as a good example that a safe distance is maintained between processes with four intention markers.) In the examples shown in these figures, we assume that a process doing a first non-repeating case or a repeating case has no markers on its `x` and `xp`. (See Claim 2 of Lemma 2 in Section 4.6.2. The definition of non-repeating and repeating cases is also described in Section 4.6.2.)

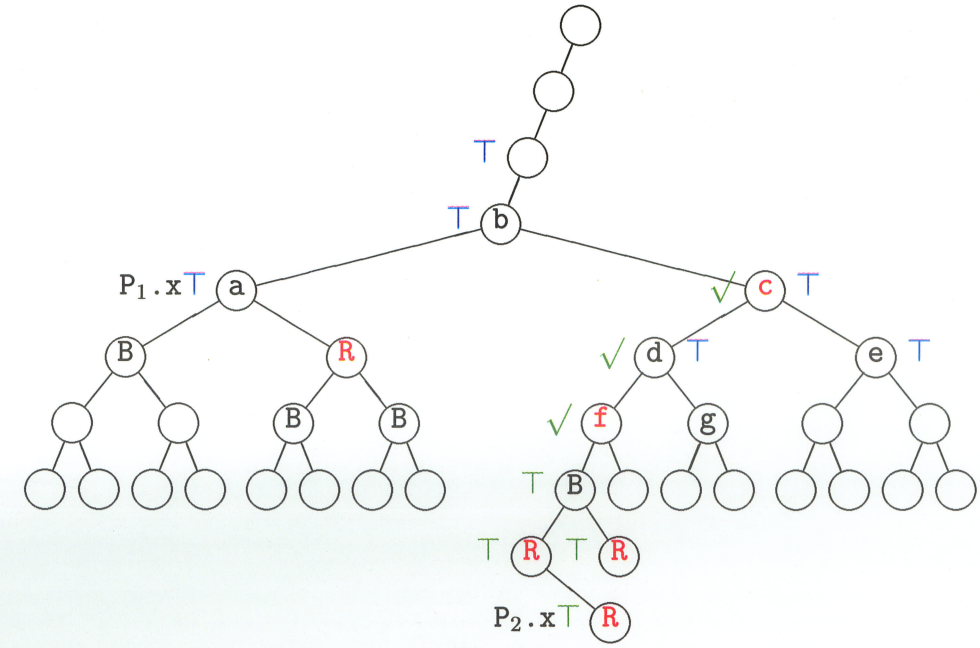
In Section 4.6, we present a proof of correctness of the algorithm. In the next two sections leading to Section 4.6, we describe an observation that `x` and `xp` of a process's initial local area have no markers and then discuss a choice we made to increase the potential concurrency. The fact that `x` and `xp` of a process's initial local area have no markers and that a process doing a first non-repeating case or a repeating case has no markers on its `x` and `xp` in its corresponding local area will be used in the proof of correctness of the algorithm. The proof goes through all the cases of transformations with four markers, adds some more rules and completes a proof of the sufficiency

(a) Deletion Case 2 — Insertion Case 1



⇓

(b) Deletion Case 1 — Insertion Case 1



82

⇓

⋮

Figure 4.15: Deletion Case 2 → 1 → 3 → 4 with 3 markers. (Picture 1 of 2.)

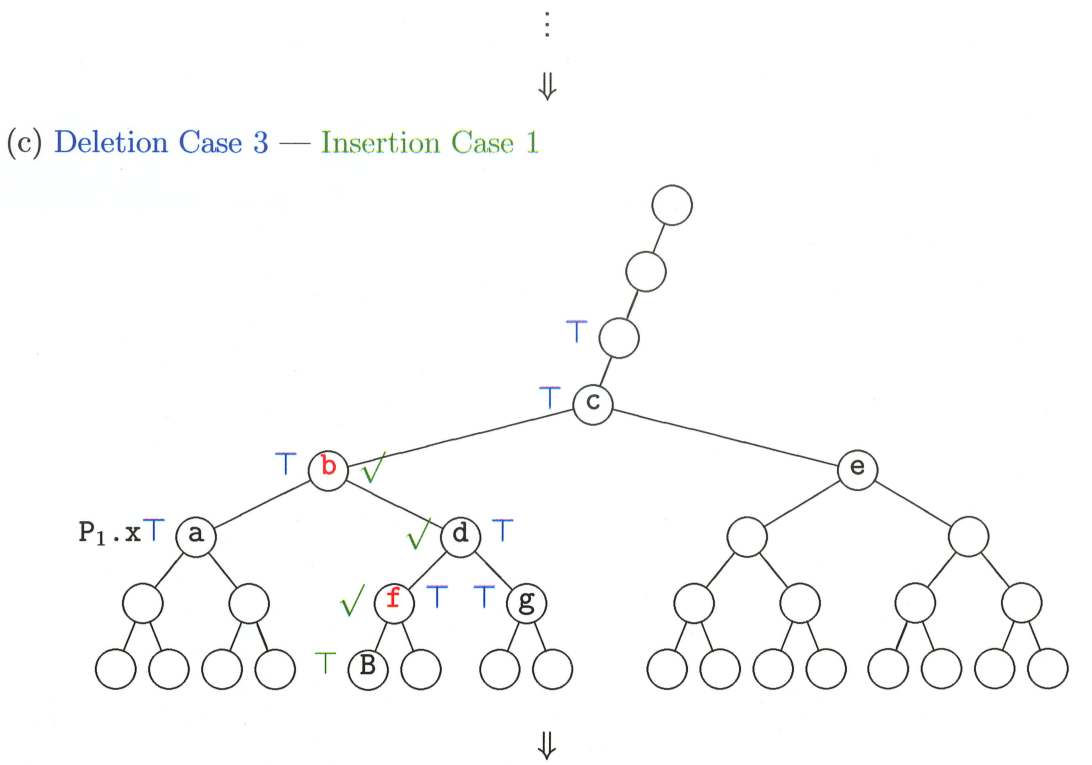
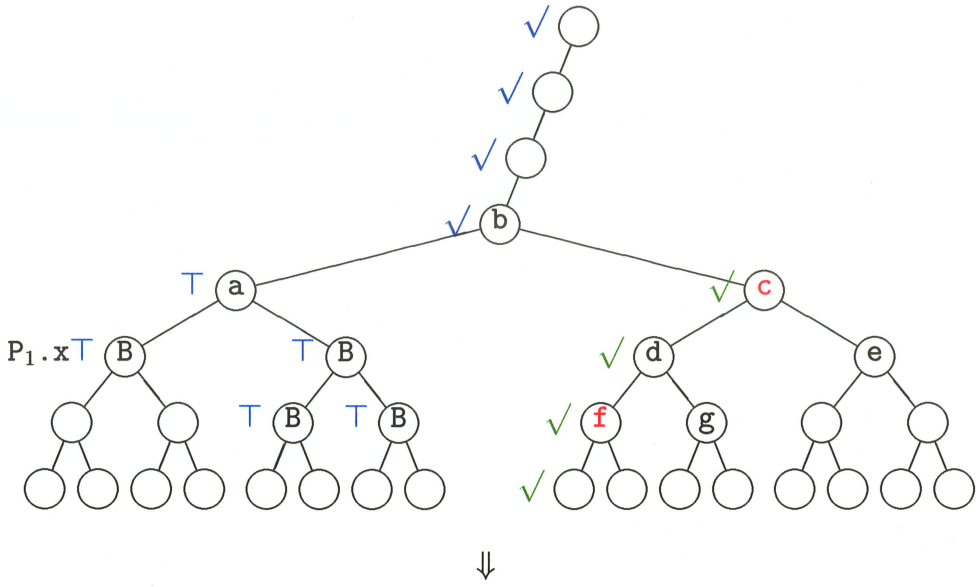


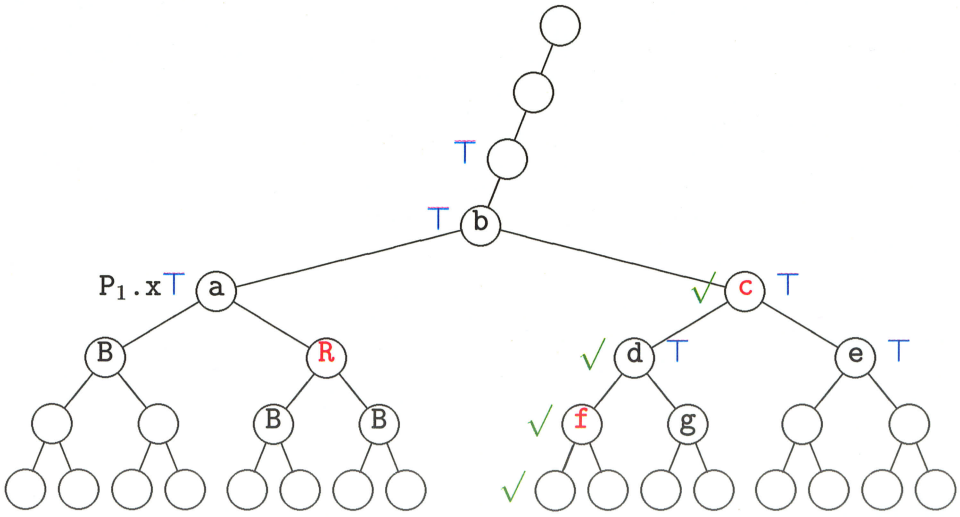
Figure 4.15: Deletion Case 2 → 1 → 3 → 4 with 3 intention markers. (Picture 2 of 2.)

(a) Deletion Case 2 — Insertion Case 1



⇓

(b) Deletion Case 1 — Insertion Case 1



⇓

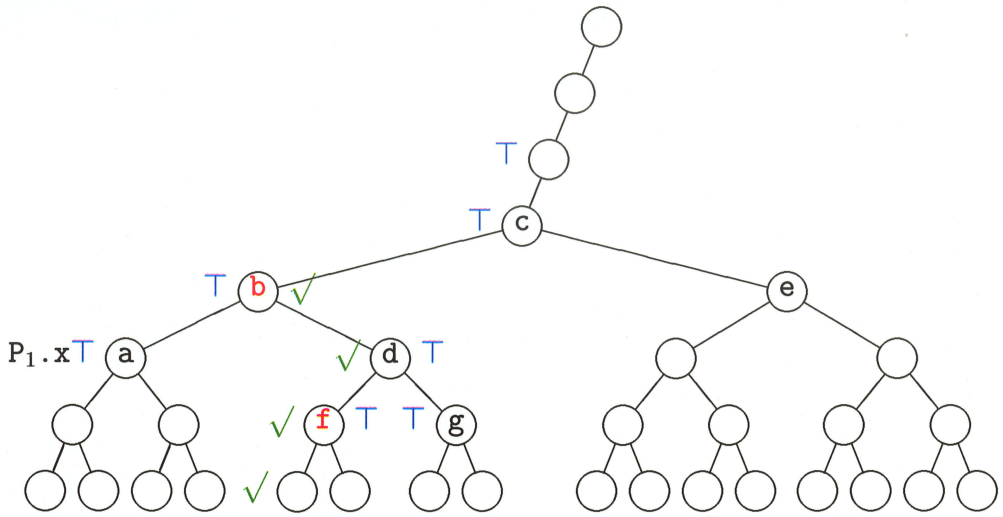
⋮

Figure 4.16: Deletion Case 2 → 1 → 3 → 4 with 4 markers. (Picture 1 of 2.)

⋮

⇓

(c) Deletion Case 3 — Insertion Case 1



⇓

(d) Deletion Case 4 — Insertion Case 1

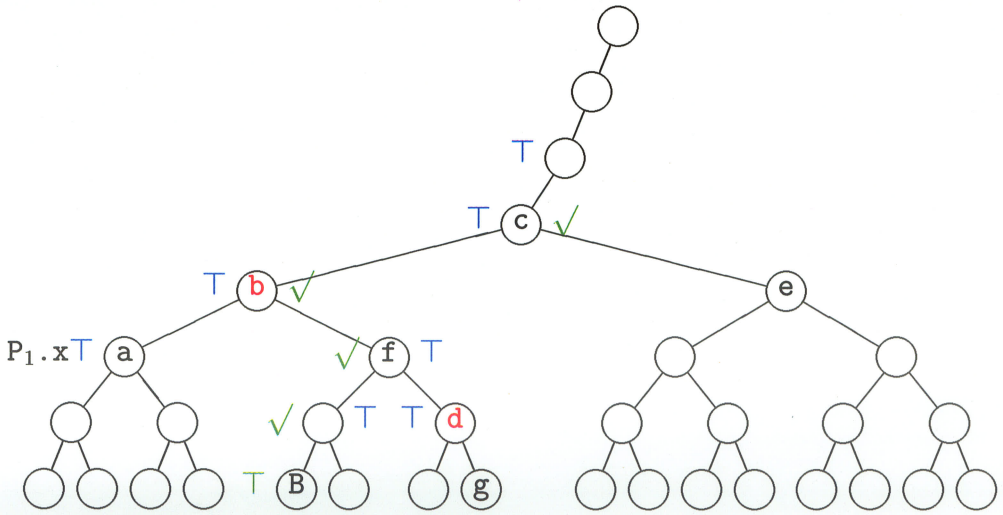


Figure 4.16: Deletion Case 2 → 1 → 3 → 4 with 4 intention markers. (Picture 2 of 2.)

of four markers which was just suggested.

4.3 An Observation Concerning the Initial Local Areas for Insertions and Deletions

We refer to the local area before any "fix up" has occurred as the initial local area. We now show that in an insertion process's initial local area, x (the new node), xp, and gp cannot have any markers and that in a deletion process's initial local area, x and xp cannot have any markers.

We now show that, in an insertion process P_1's initial local area, P_1.x, P_1.xp, and P_1.gp cannot have any markers. P_1.x is the new node, and so no marker can be on it. P_1.xp is either an internal node with two leaf children or an internal node with one non-leaf child (before x is inserted). If P_1.xp has one non-leaf child, that child must have two leaf children and will be P_1.x's sibling. There cannot be enough nodes in the path of P_1.x's sibling so that the highest marker of another process P_2 is on P_1.gp or P_1.xp and have P_2 to place the rest of its markers and its complete local area. Moreover, there cannot be enough nodes in the path of P_1.xp's sibling so that the highest marker of another process P_3 is on P_1.gp and have the rest of P_3's markers and its complete local area.

If P_1 is a deletion process, in its initial local area, P_1.x and P_1.xp cannot have any markers as the following shows. The node to be deleted is either an internal node with two leaf children or an internal node that has a single child. P_1.x is the child of the node to be deleted. When the node to be deleted is deleted, P_1.x replaces the node. Hence, P_1.x is either a leaf or an

internal node with two leaf children. There cannot be enough nodes in the path of P_1.x nor in the path of P_1.x's sibling, P_1.w, so that another process can put its highest marker on P_1.x or P_1.xp and have its complete local area.

4.4 Increasing Concurrency by Releasing Unneeded Markers

After any process performs a rotation, the process does not move up the tree any more. In fact, the process completes rebalancing the tree and exits.

In insertions, rotations occur in Cases 2 and 3. If a process performs Case-2 processing, the process immediately enters Case 3, performs Case-3 processing and then exits.

The same is also true for deletions; after a rotation is done, the process will soon exit. In deletions, rotations occur in Cases 1, 3, and 4. After a process performs Case-1 processing, the process enters Case 2, 3, or 4, and does not return to Case 1. If Case 2 immediately follows Case 1, the process has completed the rebalancing and exits. Case 3 and Case 4 are like insertion Case 2 and Case 3 because, after deletion Case 3, a process immediately enters Case 4 and then exits.

Therefore, immediately before a process P_1 performs a rotation, P_1 can remove its markers from the four nodes it had marked. This choice allows more concurrency because other processes can then place their markers on the nodes.

As mentioned in Sections 4.1.1 and 4.1.2, when a process releases its markers before a rotation, a process retains the flag of the lowest node (and

in deletion Case 3, the flags of the two lowest nodes) that have its markers. Therefore, when the rotating process releases its markers, at most three of the nodes become available to other processes.

Releasing markers before a rotation not only increases concurrency, but it also has another nice consequence. In Figure 4.16, we notice in part (d), if the blue process did not release its markers, the node labelled c would have had two markers on it. This problem is the result of a process being pulled up by the rotations of deletion Case 1 and Case 3 in parts (b) and (c) of the figure. This issue is resolved by our choice of releasing markers before a rotation. In our example, the deletion process's markers have been released in Case 1; see parts (a) and (b) of the figure.

4.5 Sketch of the Algorithm Design

We first summarize the algorithm design discussed so far, and then sketch how the rest of the design will be presented.

We adapt Ma [26]'s insertion algorithm as a starting point. Inside a local area, a process performs an operation from Cormen et al.'s sequential red-black tree algorithm [4] as described in Section 2.4.

The local area for insertions in our algorithm contains the same set of nodes as in Ma's algorithm. The local area for insertions was described in Section 4.1.1. Using the same concept, we defined the local area for deletions in Section 4.1.2. Refer to Section 3.4.2 on how Ma used a flag field to implement the local area; we use the same method, and we add the concept of intention markers.

We defined intention markers and described how an intention marker is stored and set by a process in Section 4.2.1. We explained why we need intention markers in Section 4.2.2. Then we discussed the number of markers required in Section 4.2.3. The necessity of four markers is shown, and the sufficiency of four markers is suggested. The sufficiency will be evident with the proof of the correctness of the algorithm which will be presented later.

Before a process can insert or delete a node, the process must obtain the flags of the nodes in its initial local area and must be able to set four markers above `xp` if it is deleting or above `gp` if it is inserting. In Section 4.3, we described what the initial local area is and presented an observation concerning the initial local area. That observation will be used in the proof of correctness of the algorithm.

If a process, say P_1, is in insertion Case 1 or deletion Case 2, after the case processing, P_1 must move up to its new local area. In such a case, P_1 sets one or two markers (depending on whether it is deleting or inserting, respectively) on the one or two ancestor nodes immediately above the node that contains P_1's current highest marker and also obtains the flags of the nodes that will be part of P_1's new local area. If P_1 is in any other case, P_1 will perform a rotation, and will then soon exit. P_1 can, therefore, release its unneeded markers before the rotation to increase the potential for concurrency. We described this choice in Section 4.4.

In Section 4.6, we will provide a proof of the correctness of the algorithm. In the proof, we go through all the possible case transformations enumerated in Figure 4.17 with the algorithm design discussed so far, then we come across a problem we refer to as the double marker problem. To resolve this

problem, we will add five new rules to the algorithm: the Spacing rule, the Move-Up rule, the Insert-Jump-Ahead rule, the Go-First rule, and the Exit rule. These rules will resolve the problem discovered and the proof will then show the correctness of the algorithm design.

4.6 Proof of Correctness of the Algorithm Design

In this section, we show the sufficiency of four markers for allowing processes to complete their work. In the pictures that follow in the rest of the thesis, we assume that the processes shown in the pictures hold the flags of their local areas as defined in Section 4.1 and we omit the $\top$ symbol.

As will be shown in this section, processes will need to cooperate. This cooperation involves interprocess communication. For instance, interprocess communication is needed for transferring flags where a process, $\mathtt{P}_1$, requests that another nearby process, $\mathtt{P}_4$, transfers the ownership of the flag of a node (required for $\mathtt{P}_1$ but held by $\mathtt{P}_4$) to $\mathtt{P}_1$. In Section 5.1, we will discuss some implementation details on the cooperation between processes, including how the ownership of a flag can be implemented.

4.6.1 The Possible Case Transformations

We define a case transformation to be the change from one case to the next case performed by a process as it fixes up the red-black tree properties. To simplify some arguments, we also include insertion Case 3 and deletion

Insertions	Deletions
Case 1 $\cdots$.	Case 2 $\cdots$.
Case 1 $\cdots \rightarrow$ a transformation below.	Case 2 $\cdots \rightarrow$ a transformation below.
Case 2 $\rightarrow$ 3.	Case 1 $\rightarrow$ 2.
Case 3.	Case 1 $\rightarrow$ 3 $\rightarrow$ 4.
	Case 1 $\rightarrow$ 4.
	Case 3 $\rightarrow$ 4.
	Case 4.

Figure 4.17: All possible case transformations (where '$\cdots$' means zero or more iterations of the case).

Case 4 individually as "case transformations", even though the process is not changing from one case to the next.

All possible case transformations are listed in Figure 4.17. Note that a process can repeat insertion Case 1 or deletion Case 2 at most $O(\log_2 n)$ times in a tree with n nodes.

4.6.2 Two Lemmas Required for a Proof of the Sufficiency of Four Markers

A process, say P_1, that performs a rotation can have in its local area markers of another process, say P_2, because P_1 is allowed to place a flag on a node that has another process's marker. Figures 4.18 through 4.21 show all the patterns of marker adjustment that P_1 might have to make to P_2's markers if P_1 does a rotation. These adjustments are done to maintain the proper

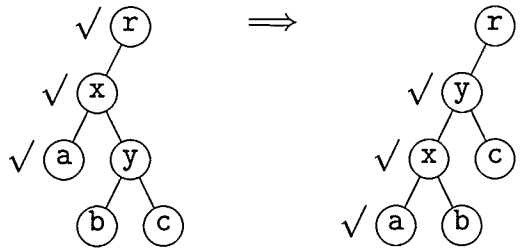


Figure 4.18: Pattern 1: A left rotation at x requires the marker on r to be moved to y. (Replica of Figure A.1.)

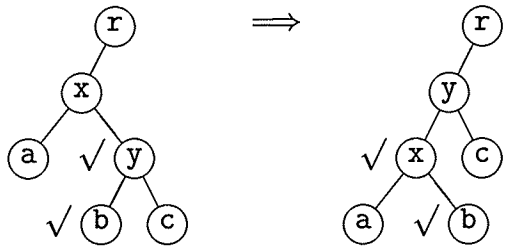


Figure 4.19: Pattern 2: A left rotation at x requires the marker on y to be moved to x. (Replica of Figure A.5.)

alignment of markers, ensuring that the markers of P_2 continue to be on the four immediate direct ancestor nodes of the grandparent of P_2's x in insertions and of the parent of P_2's x in deletions, after a rotation performed by P_1. The pictures in Figures 4.18 through 4.21 are the cases where P_2's markers are displaced. These were determined by looking at all the possible marker movements; see Appendix A.

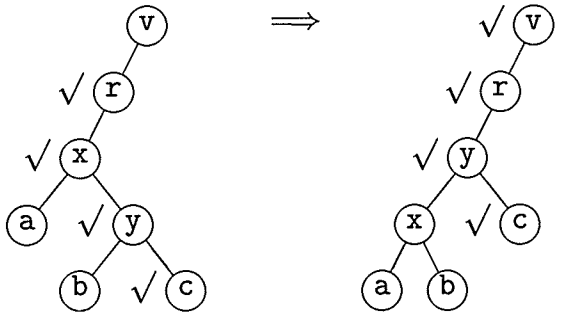


Figure 4.20: Pattern 3: A left rotation at x requires the marker on x to be moved to v, an ancestor of x. (Replica of Figure A.6.)

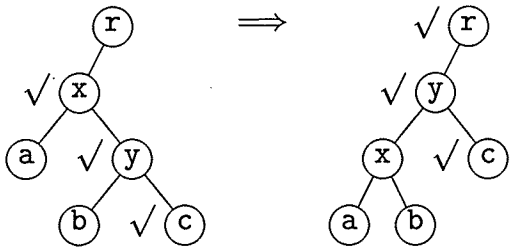


Figure 4.21: Pattern 4: A left rotation at x requires the marker on x to be moved to r, an ancestor of x. (Replica of Figure A.7.)

Lemma 1 *No markers are moved to or from nodes* `a`, `b`, *and* `c` *of Figures 4.18 through 4.21.*

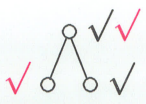
Lemma 1 is established by inspection of Figures 4.18 through 4.21.

A deletion process can repeat Case 2 many times, moving up the tree, before entering some other case. Case 2 is the *repeating case* for deletion processes. The case that a deletion process enters into (if any) immediately after its last Case 2 is referred to as its *first non-repeating case.* For an insertion process, Case 1 is the repeating case. The case (Case 2 or 3, if any) that the insertion process enters immediately after its last Case 1 is its first non-repeating case.

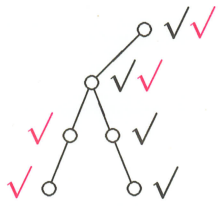
In the proof of the following lemma, we prove that the `x` and `xp` nodes of a process contain no markers when the process is in its repeating case or in its first non-repeating case. This fact is used in drawing the pictures in the lemma, and helps simplify the process of going through the transformation cases.

Lemma 2 *If we begin with only one marker per node, then*

(1) There are situations in which the rotations caused by insertions or deletions will put two markers on one node (as in the picture below) — we call this the double marker problem —, but



(2) Two consecutive double markers from the same two processes (as pictured below) are impossible.



Proof:* (By enumeration of the transformations listed in Figure 4.17.) When a process starts an insertion or a deletion, the process is only allowed to place a marker on a node if the node does not have a marker from another process and if the node's flag is not held by another process. We assume that each process is doing only one insertion or deletion at a time, or that each insertion or deletion task has a unique ID that is stored as the value of the marker field of a node.

We begin with a (correct) tree that currently has no more than one marker per node. Then we try to derive two or more markers on a node using rotations performed by some processes.

We name the process performing the rotation(s) $\mathtt{P}_1$. We also refer to other processes $\mathtt{P}_2$, $\mathtt{P}_3$, $\mathtt{P}_4$, and $\mathtt{P}_5$. We represent $\mathtt{P}_2$, $\mathtt{P}_3$, $\mathtt{P}_4$, and $\mathtt{P}_5$'s markers using black (√), magenta (√), cyan (√), and green (√) check marks, respectively.

Claim 1 *When a process first obtains flags and sets its markers prior to fixing up the tree,*

- *For a deletion process, the process's* $\boldsymbol{x}$ *and* $\boldsymbol{xp}$ *nodes have no markers.*

* This proof continues from page 95 to page 112.

- *For an insertion process, the process's* **x**, **xp**, *and* **gp** *nodes have no markers.*

Claim 1 follows from the observation discussed in Section 4.3.

Claim 2 *At the beginning of each repeating case and at the beginning of the first non-repeating case:*

- *In deletions, the* **x** *and* **xp** *nodes of a process will have no other process's marker on them.*
- *In insertions, the* **x**, **xp**, *and* **gp** *nodes of a process will have no other process's marker on them.*

Claim 2 can be established by noting that:

- For deletions:

  xp: By Claim 1, xp at the beginning of a deletion (before the fix-up) does not have a marker. Each time a deletion process performs Case 2, the process's x is moved up to xp. The new xp is then the node that previously contained the lowest marker of the process's previous local area, so no other process may have its marker there.

  x: By Claim 1, x at the beginning of a deletion does not have a marker. The new x after deletion Case 2 is the old xp, which did not have any marker as just described.

- For Insertions:

gp: By Claim 1, gp at the beginning of an insertion has no marker. Each time the process performs Case 1, the new x is the old gp and, therefore, the new gp is the node that previously contained the second lowest marker of the process's previous local area.

xp: By Claim 1, xp at the beginning of an insertion does not have a marker. Each time an insertion process performs Case 1, x of the process is moved up to the grandparent of x. The new xp is then the node that previously contained the lowest marker of the process's previous local area, so no other process may have its marker there.

x: By Claim 1, x at the beginning of an insertion has no marker. Each time the process performs insertion Case 1, the new x is the old gp which had no marker as just described.

Claim 3 *A marker from some other process can be moved up to* ***xp*** *by* P_1*'s rotations. However,* P_1 *soon exits after its rotation(s). Therefore, a process can have a marker from some other process on its* ***xp*** *only temporarily.*

Claims 1 to 3 apply to any process, not just P_1.

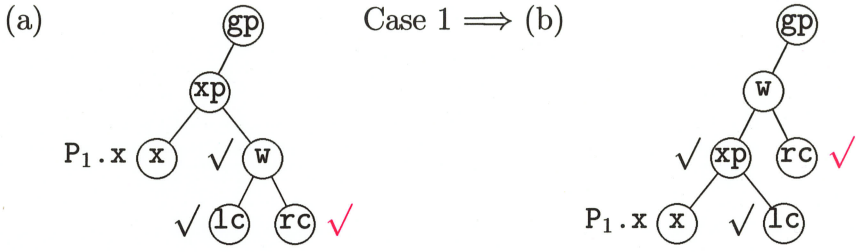
To complete the proof of Lemma 2, we will consider all the case transformations (from Figure 4.17) that create possible rotations in the following order: deletion Cases 1, $1 \rightarrow 2$, $1 \rightarrow 3 \rightarrow 4$, $1 \rightarrow 4$, $3 \rightarrow 4$, and 4, and then insertion Cases $2 \rightarrow 3$ and 3. We naturally omit deletion Case 2 and insertion Case 1 from Figure 4.17 because they perform no rotations and hence cannot contribute to the creation of double marker problem.

Deletion Case 1

In our deletion Case 1 pictures (see below), P_1 is doing a left rotation on `xp`. Because of Claim 2, no markers are on `xp` before the rotation of Case 1.

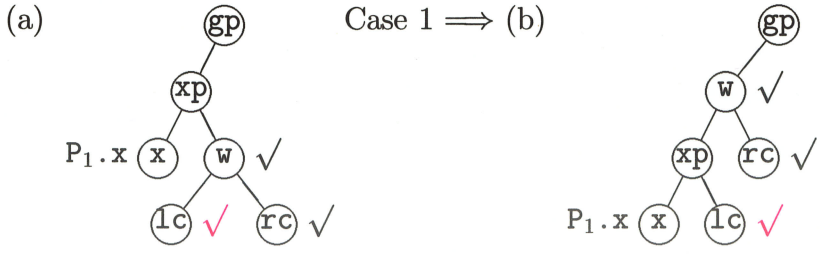
We do not have to consider the highest marker of P_2 being lower than P_1`.w` in Pictures 1(a) and 2(a) because, applying Lemma 1, we see that no markers are being moved to or from `x`, `lc`, and `rc` in the pictures for deletion Case 1.

Picture 1:



In the second picture, P_2 is located in `rc`'s branch.

Picture 2:



As can be seen, neither of these cases results in one node holding two markers.

Deletion Case 1 → 2

As explained in Section 2.4.4, if P_1 does Case 2 after Case 1 (as shown in Picture 1(b) or 2(b)), P_1 will exit immediately. Since Case 2 does no further rotations, we omit pictures for this transformation, which cannot cause one node to hold two markers.

Deletion Case 1 → 3 → 4

In deletion Case 3, P_1 performs a right rotation on `w`. In deletion Case 4, P_1 does a left rotation on `xp`.

We consider Cases 3 and 4 applied to Pictures 1(b) and 2(b); and go through all the cases derived from Pictures 1(b) and 2(b), doing Cases 3 and 4 on them.

Following a Case 1 rotation, another process's markers may be pulled up. The markers so pulled up may fall within P_1's new local area in which case the highest such marker will be on P_1's `xp`. If the markers pulled up are outside of P_1's new local area, they are not a concern, as will be seen in the rest of the proof of Lemma 2.

We focus on the position of P_2 (identified by the black check marks), especially in part (a) in the pictures below (which depict the situation after Case 1). We also place as many other concurrent processes as possible near P_1 so that we are constructing worst-case scenarios. We place the highest marker of P_2 on `xp`. Then we place the second highest marker of P_2 on `w`, because that is the only "path" P_2's markers can be on. Then the third highest marker can be on either `w`'s left child or right child. We first place

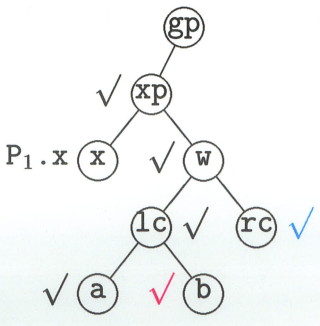
the marker on w's left child, node lc in the pictures. We then consider the cases of the lowest marker being on lc's left child and then on lc's right child. Then we place the third highest marker on w's right child, node rc in the pictures, and consider the cases of the lowest marker being on rc's left and then right child. In such a manner, we consider all the possible ways that P_2's markers can be placed. After all the resulting "paths" are considered, we then place the highest marker of P_2 one level lower, on w, and go through the same procedure again, and so on.

In Pictures 3 to 6, the highest marker of P_2 is on P_1.xp before Case 3 is done. Pictures 7 through 10 show the situations where P_2 is located one level lower than the corresponding locations in Pictures 3 to 6. Finally, in Pictures 11 and 12, we consider situations where P_2 is located one level lower than the corresponding locations in Pictures 7 to 10.

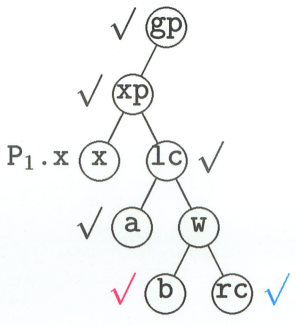
Applying Lemma 1, we see that nodes a, b, and rc in the pictures corresponding to Case 1 $\rightarrow$ 3 $\rightarrow$ 4 will have no markers moved to or from them during rotations. Therefore, we can safely omit the nodes located below a, b, and rc. We can also safely omit the cases where P_2's highest marker is located on or below these nodes.

Picture 3 (from Picture 1(b)):

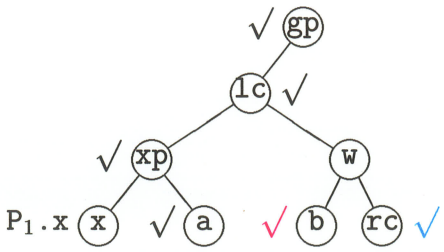
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



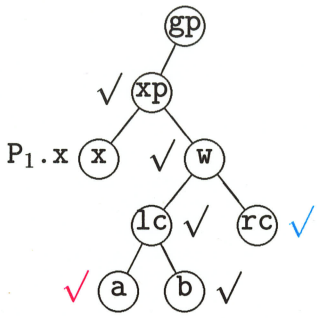
Case 4 $\Longrightarrow$ (c)



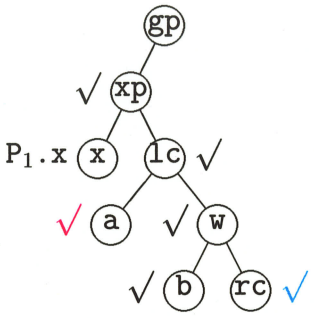
In Picture 4(a), the lowest marker of P_2 is on `b` instead of on `a` as in Picture 3(a).

Picture 4 (from Picture 1(b)):

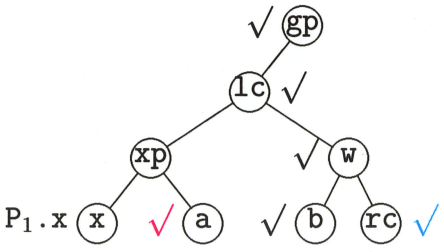
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



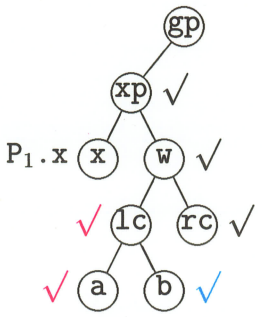
Case 4 $\Longrightarrow$ (c)



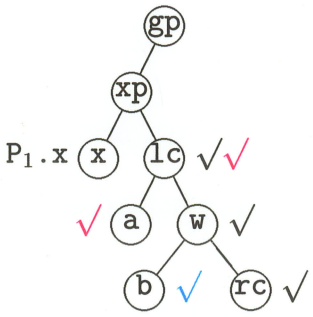
In Pictures 5(a) and 6(a), P_2's third highest marker is on `rc` instead of on `lc` as in Pictures 3(a) and 4(a).

Picture 5 (from Picture 1(b)):

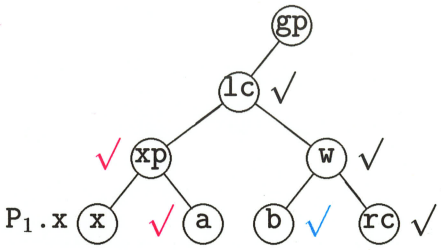
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)

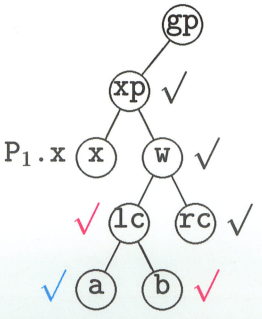


In Picture 5(b), a double marker is created on node `lc`, but this double marker is momentary and exists for a short period of time and only before P_1 exits since Case 4 results in the separation of the two markers as shown in Picture 5(c).

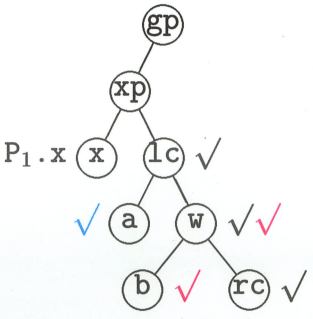
In Picture 6(a), P_3's second highest marker is on `b` instead of on `a` as in Picture 5(a).

Picture 6 (from Picture 1(b)):

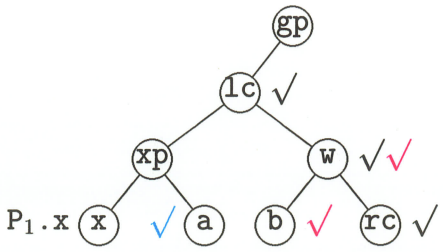
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)

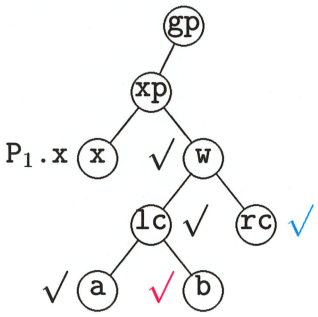


In Picture 6(c), a double marker is created on the node labelled w and will continue to exist after P_1 exits. This double marker problem will have to be explicitly addressed later.

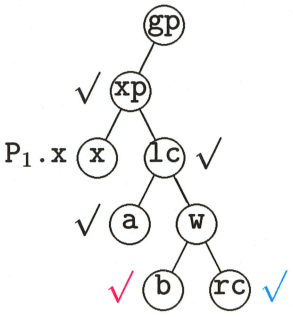
Pictures 7 through 10 show the situations where P_2 is located one level lower than the corresponding locations in Pictures 3 to 6.

Picture 7 (from Picture 1(b) or 2(b)):

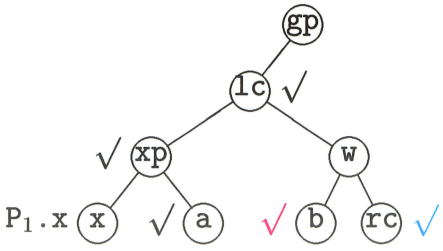
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



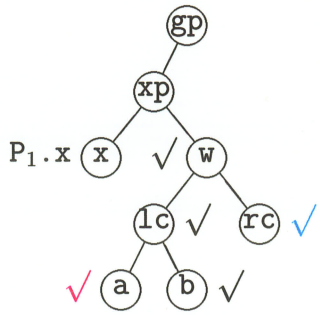
Case 4 $\Longrightarrow$ (c)



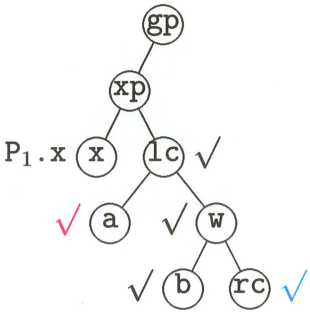
In Picture 8(a), the third highest marker of P_2 is on b instead of on a as in Picture 7(a).

Picture 8 (from Picture 1(b) or 2(b)):

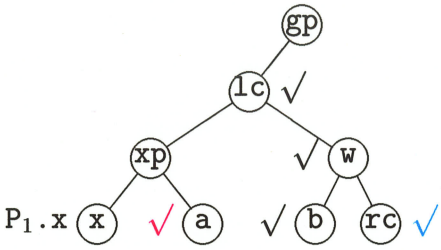
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



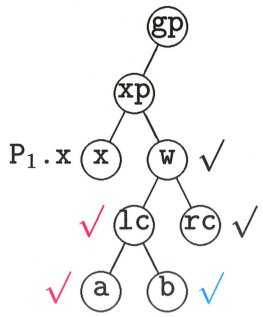
Case 4 $\Longrightarrow$ (c)



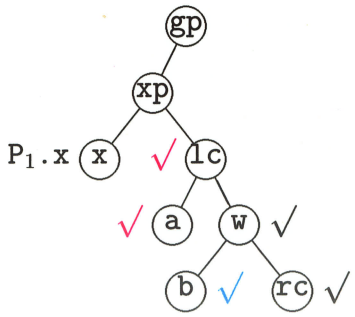
In Picture 9(a), the second highest marker of $\mathtt{P}_2$ is on `rc` instead of `lc` as in Pictures 7(a) and 8(a).

Picture 9 (from Picture 1(b) or 2(b)):

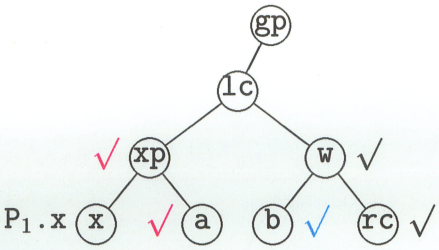
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



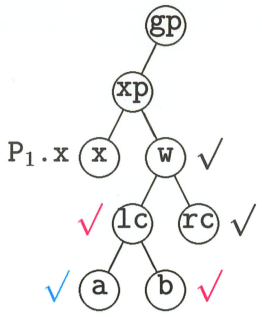
Case 4 $\Longrightarrow$ (c)



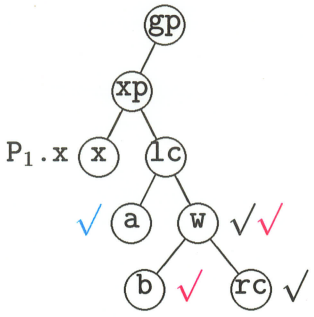
In Picture 10(a), the second highest marker of P_3 is on b instead of on a as in Picture 9(a).

Picture 10 (from Picture 1(b) or 2(b)):

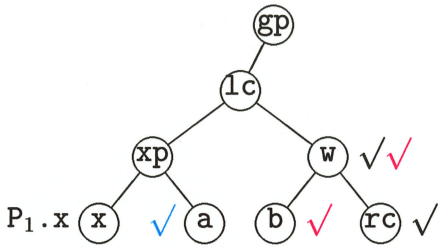
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)

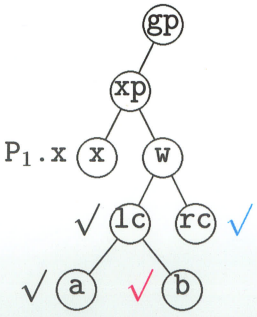


In Picture 10(c), a double marker is created on the node labelled w and, as in Picture 6, we will have to deal with it explicitly.

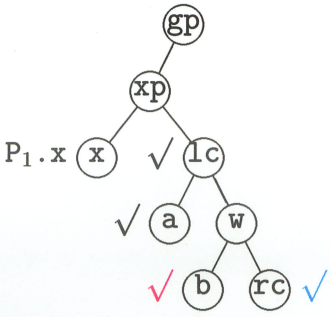
In Pictures 11 and 12, we consider situations where P_2 is located one level lower than the corresponding locations in Pictures 7 to 10.

Picture 11 (from Picture 1(b) or 2(b)):

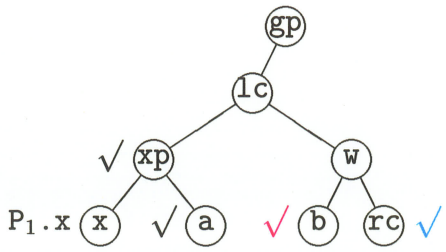
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



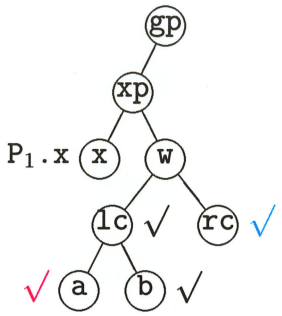
Case 4 $\Longrightarrow$ (c)



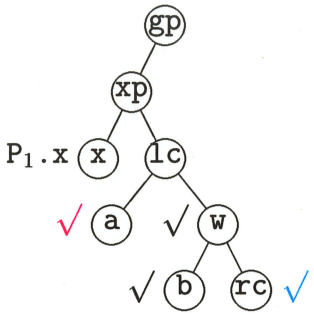
In Picture 12(a), the second highest marker of P_2 is on `b` instead of on `a` as in Picture 11(a).

Picture 12 (from Picture 1(b) or 2(b)):

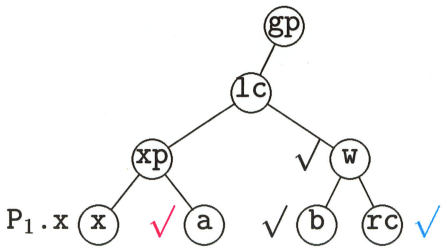
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)



In Pictures 11(a) and 12(a), if the cyan process, P_4, is on `P1.w`, the cases are analogous to Pictures 9(a) and 10(a). Also, in Pictures 9(a) and 10(a), if the black marker on `rc` is the highest of the black markers, then the cases are analogous to Pictures 11(a) and 12(a). Hence, we omit such cases.

After going through all the transformations of deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4, we found two cases where double markers occur — Pictures 6 and 10 —, but no two consecutive double markers are created.

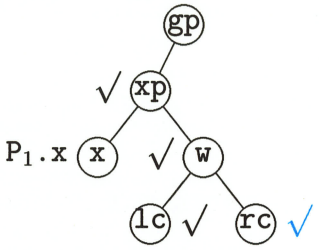
Deletion Case 1 → 4

In deletion Case 4, as mentioned before, P_1 performs a left rotation on xp.

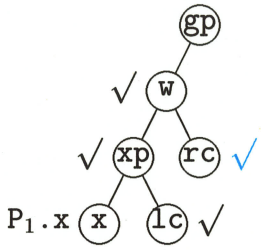
By Lemma 1, we know that no markers are moved to or from x, lc, and rc in the pictures of deletion Case 1 → 4. Hence, the pictures do not show the nodes located below x, lc, and rc. Also, we do not consider the cases where the highest marker of P_2 is located on or below these nodes.

Picture 13 (from Picture 1(b)):

(Case 1) ⟹ (a)



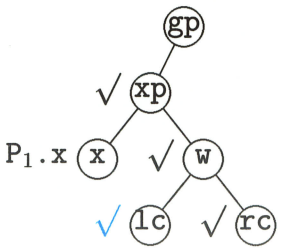
Case 4 ⟹ (b)



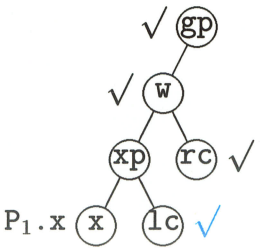
In Picture 14(a), the third highest marker of P_2 is on rc instead of on lc as in Picture 13(a).

Picture 14 (from Picture 1(b)):

(Case 1) ⟹ (a)



Case 4 ⟹ (b)



Imagine P_2 in Pictures 13(a) and 14(a) being a level lower. The resulting situation cannot create a double marker problem because the resulting pictures will be identical to Pictures 13(b) and 14(b) except that the highest marker of P_2 would be on xp in Picture 13(b) and on w in Picture 14(b).

Therefore, we omit these cases.

In deletion Case 1 $\rightarrow$ 4, none of the cases cause one node to hold two markers.

Deletion Case 3 $\rightarrow$ 4

Claim 4 *We can omit pictures for this case because deletion Case 3 $\rightarrow$ 4 is analogous to Pictures 7 through 12 of deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4.*

Claim 4 is established as follows. By Claim 2, a marker cannot be on P_1.xp as P_1 enters Case 3 in the transformation, since here Case 3 is the first non-repeating case. Pictures 7 through 12 cover P_1's Case 1 $\rightarrow$ 3 $\rightarrow$ 4 transformation where the highest node that a marker of a process beneath P_1 can be on is P_1.w as P_1 enters Case 3. That is, P_1.xp has no markers as P_1 enters Case 3 in these pictures. Hence, we omit pictures for deletion Case 3 $\rightarrow$ 4.

Deletion Case 4

Claim 5 *We can omit pictures for this case because deletion Case 4 is analogous to Pictures 1 and 2 of deletion Case 1.*

This claim is established as follows. Since we are only considering deletion Case 4 without Case 1 and/or Case 3 preceding it, by Claim 2, P_1.xp will not have a marker as P_1 enters Case 4, just as in Case 1. Also, as in Case 1, P_1 does a left rotation on xp. The difference between Case 4 and Case 1 is that P_1 immediately exits after Case 4. Therefore, we can omit pictures for deletion Case 4.

We have now considered all the deletion transformations. When we introduced the discussion of deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4 on page 99, we mentioned that, following a Case 1's rotation, the markers pulled up by P_1 that are outside P_1's new local area are not a concern. This is established by inspection of the previous pictures of this lemma as follows.

When P_1 performs deletion Case 1, the highest marker of a process can be pulled up outside the new local area of P_1 in two ways. They are shown in Picture 1(b) and Picture 2(b). In Picture 1(b), the highest marker of a process can be, at highest, placed on node `rc`, which is P_1's new grandparent's right child. In Pictures 3 to 14, where other deletion cases are performed on Picture 1(b), no markers are moved to gp's right child (not shown) from within the local area of P_1. Hence, double markers cannot be created on that node by P_1's rotations.

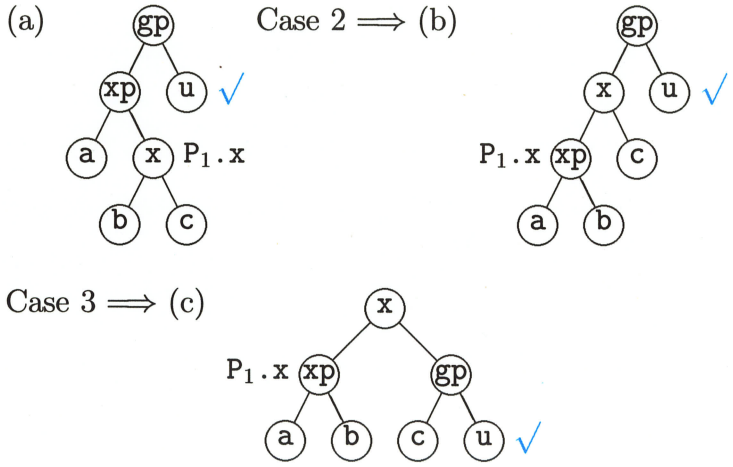
In Picture 2(b), the highest marker of a process can be, at highest, placed on node `w` which is P_1's new grandparent. Pictures 7 through 12 are derived from Picture 2(b). In Pictures 7 through 12, no markers are moved to node gp. Hence, double markers cannot be created on that node by P_1's rotations.

In deletion transformations, we found only two cases that produce double markers — Pictures 6 and 10 — and no transformations that produce two consecutive double markers. Now we move on to consider insertion Cases 2 $\rightarrow$ 3 and 3.

Insertion Case 2 → 3

In insertion Case 2, P_1 moves x to xp and performs a left rotation on the new x. Then, in Case 3, which follows Case 2, P_1 performs a right rotation on x's grandparent, gp. By Claim 2, we know that x, xp, and gp do not have any markers as P_1 enters Case 2. From Lemma 1, we see that no markers are moved to or from a, b, and c in Picture 15. Therefore, we omit the cases where the highest markers of other processes are on or lower than a, b and c.

Picture 15:



The only picture for insertion Case 2 → 3 shows that no double marker problem occurs as a result of the transformation.

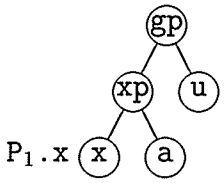
Insertion Case 3

In insertion Case 3, P_1 performs a right rotation on the grandparent gp of P_1.x. By Claim 2, we know that x, xp, and gp do not have any markers as P_1 enters Case 3. If P_1 does only Case 3 without Case 1, then no markers

are in the local area of P_1 because an insertion begins at the leaf. Thus, we only need to consider the situation where Case 3 is preceded by Case 1.

By Lemma 1, no markers are being moved to and from x, a, and u in Picture 16. Thus, we omit the cases where the highest markers of processes are located on or lower than x, a, and u in Picture 16. In Picture 16, there are no markers in the local area of P_1. Hence, we do not show the case processing in the picture.

Picture 16:



The only picture for insertion Case 3 has no markers shown, so no double marker problems can occur.

We have now gone through all the transformations for insertions, and there were no cases that produced a double marker situation.

Conclusion of the Proof of Lemma 2:

We have gone through all the deletion and insertion transformations and shown that there are only two situations where double markers are created on a node — Pictures 6 and 10. There are no situations that create two consecutive double markers from the same two processes.

Our results are generally applicable. That is, if we go through all the possible transformations of any process, the pictures drawn for P_1 in this

proof can be used to show the same results for any other chosen process. □

4.6.3 Preventing Double Markers and the Sufficiency of Four Markers

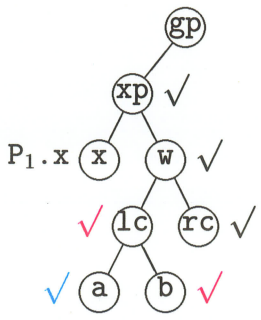
Here, we describe five necessary rules added to the basic algorithm so that we prevent double markers from happening in the first place. (The reason for choosing to prevent double markers is explained in Section 4.7.) We first present the Spacing rule. Then, in Theorem 1, which follows, we go through all the transformations as we did in Lemma 2 to show that the revised algorithm is correct. In the proof of the theorem, we discover the need for, and add, four more rules: the Move-Up rule, the Insert-Jump-Ahead rule, the Go-First rule, and the Exit rule.

In the following presentation of the Spacing rule, the naming convention of processes in the pictures is the same as that used in Lemma 2; refer to page 95.

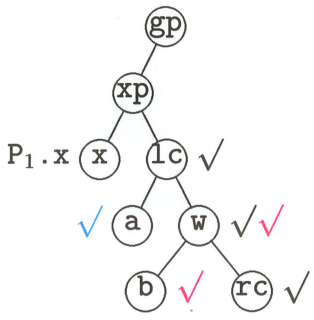
We focus on the two situations where the double markers occur in a node. The situations are shown in Pictures 6 and 10 of Lemma 2, and are reproduced here for the reader's convenience. In both pictures, P_1 is a deleting process and double markers result on node w.

Picture 6 (from Picture 1(b)):

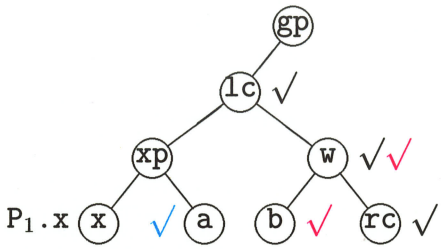
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)

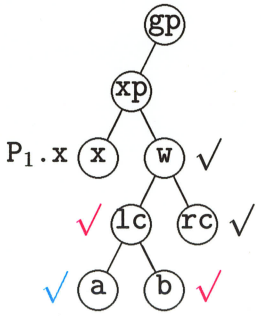


Case 4 $\Longrightarrow$ (c)

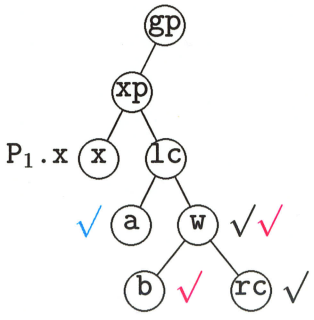


Picture 10 (from Picture 1(b) or 2(b)):

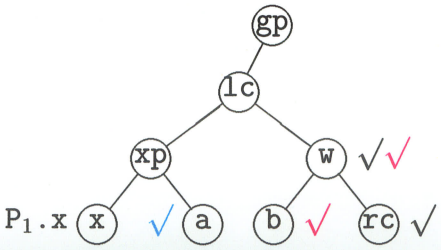
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



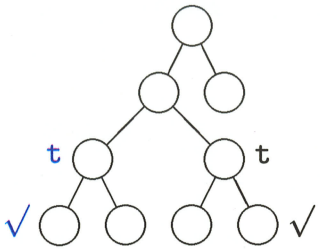
Case 4 $\Longrightarrow$ (c)



In Pictures 6(a) and 10(a), we observe that P_3's markers are on `lc` and `b` and that P_2's markers are on `w` and `rc`. After Case 3's right rotation on

w, in Pictures 6(b) and 10(b), P_3's marker that was on lc in part (a) of the pictures is moved to w and creates a double marker problem on w.

The purpose of the Spacing rule is to not allow P_3, in Pictures 6(a) and 10(a), to come up the tree to a point where P_3 can hold a marker on lc. That is, we require processes to leave a single space between themselves and another process when they are placing markers.



In the above picture, blue P_1 and black P_2 are both doing deletions. Both processes did Case 2 and are trying to move up. In each case, t denotes the node in which a new marker is to be placed.

The Spacing Rule: We allow a process to put a marker on node t only if the following three nodes have no flags or markers:

1. t,
2. the parent of t, and
3. the sibling of t.

To do this checking, the process obtains the flag of each of the above three nodes, using a sequence of CAS operations, in the order listed. After obtaining each flag, the process checks to see if any marker is on the node.

If there is a marker, the process releases the flags obtained so far (on the three nodes), and restarts the process of obtaining the flags (starting from t). Similarly, if a CAS fails, then the process releases the flags that it has obtained so far (on the three nodes) and restarts the process.

In the above picture, then either P_1 or P_2 will atomically succeed in placing a marker on its t, but not both. The process that successfully places a marker will proceed first. The other will back off and have to retry placing a marker on its t.

For insertions in Case 1, which move up by two nodes, we apply the Spacing rule twice; first on t and then on the parent of t. If we fail at the parent of t, we back off by releasing the marker on t and retry placing markers on both t and its parent.

To acquire the four markers above the initial local area, we apply the rule four times; once for each of the four markers we require.

The Spacing rule guarantees a single space between processes when processes are placing markers. Yet, the Spacing rule itself, alone, is not sufficient to produce a correct algorithm due to the possible effects of rotations. For instance, consider the situation shown in Figure 4.22.

In Figure 4.22(a), blue P_1, black P_2, magenta P_3, and cyan P_4 are deletion processes and have placed their markers according to the Spacing rule. In part (b) of the figure, P_1 has moved up (by repetitions of Case 2) and is about to perform a rotation. P_1 performs deletion Cases 1, 3, and 4, and the results are shown in parts (c), (d), and (e), respectively. In part (f), P_2 moves one level up. In part (g), P_4 moves up and is about to perform deletion Case 1. The result after P_4 performs deletion Case 1 is shown in part (h),

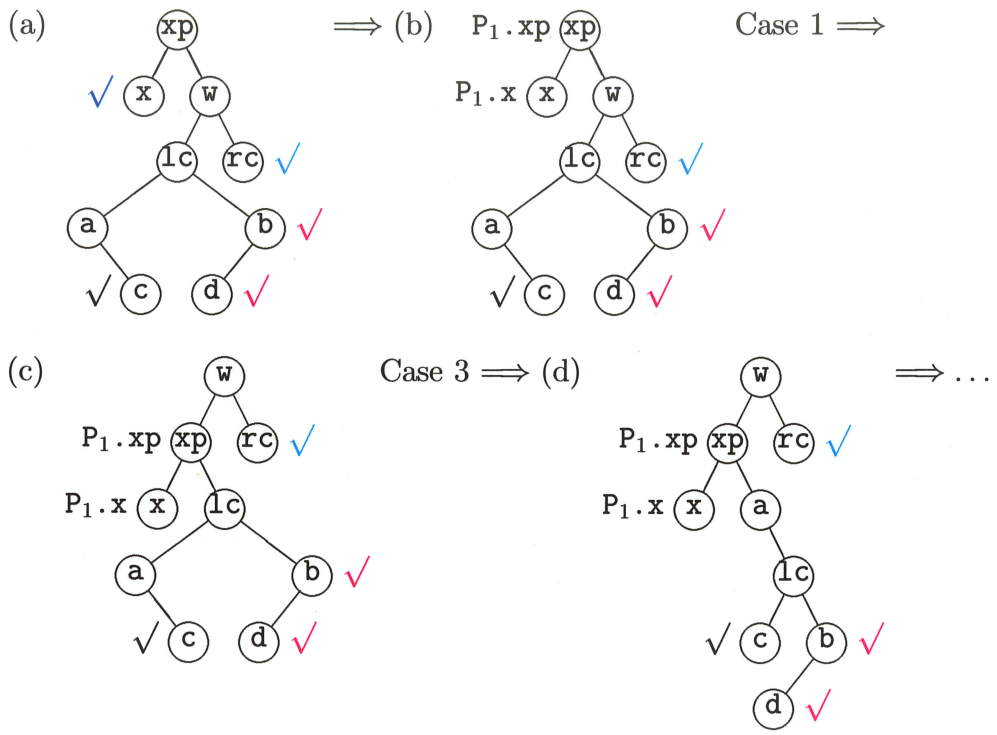


Figure 4.22: A situation where the markers of two processes come too close to one another. (Picture 1 of 2.)

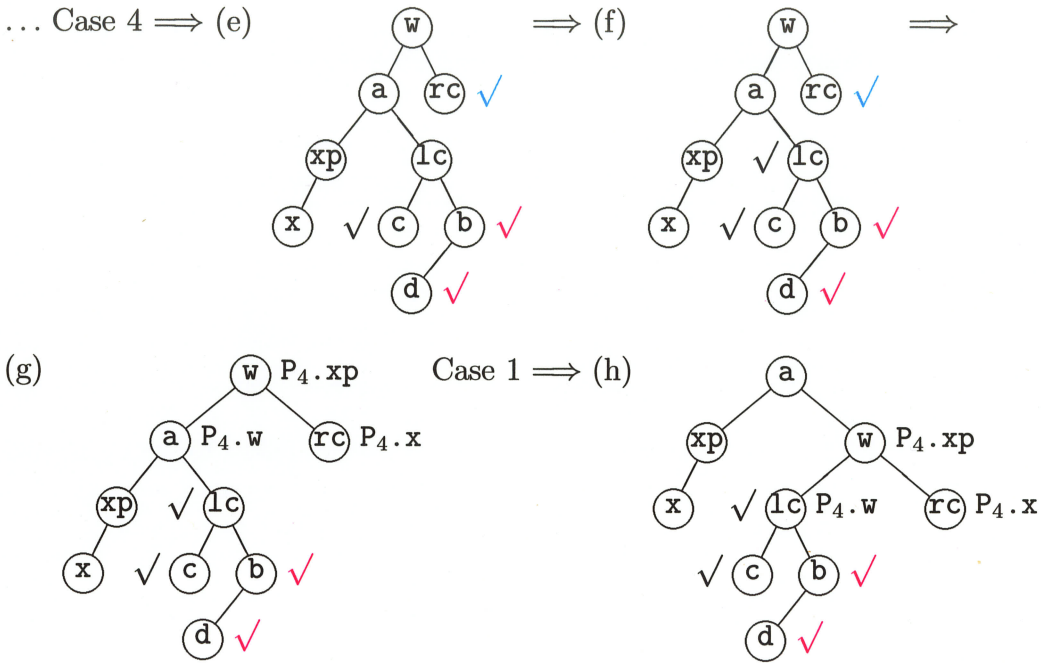


Figure 4.22: A situation where the markers of two processes come too close to one another. (Picture 2 of 2.)

and corresponds to Picture 10(a) where a double marker can be produced, in a symmetric way. In Figure 4.22(h), the processes are too close together, and this is why the double marker problem can occur.

To address this problem, we add some additional rules. The new rules involve processes working together to avoid situations that may lead to double markers. These new rules are described in the proof of the following theorem.

Theorem 1 *Using the Spacing rule and four additional rules, that will be described in this proof, our algorithm is correct.*

Proof:* (By enumeration of the transformations listed in Figure 4.17.) In this proof, we go through pictures of all the transformations, as we did in Lemma 2. After the rotations, in each picture, we show either that:

- we still have at least a single space between processes (as in the illustrations of the Spacing rule) or,
- in the cases where two processes are brought too close together by rotations of another process, that one of the two processes is allowed to proceed ahead of the other so that any other process performing a rotation cannot create double markers.

The naming convention used is, again, the same as that of Lemma 2; refer to page 95. As in Lemma 2, we consider a process $\mathtt{P}_1$ that is performing a rotation and represent the markers of two other processes, $\mathtt{P}_2$ and $\mathtt{P}_3$, using black ($\surd$) and magenta ($\surd$) check marks, respectively.

* This proof continues from page 118 to page 186.

We will go through all possible rotations that P_1 could be performing in the following order: deletion Cases 1, $1 \rightarrow 2$, $1 \rightarrow 3 \rightarrow 4$, $1 \rightarrow 4$, $3 \rightarrow 4$, and 4, and then we also consider insertion Cases $2 \rightarrow 3$ and 3. We assume the Spacing rule is applied and try to find situations where the space between markers of two processes is removed.

To this point, we have considered only the spacing of two processes within P_1's local area. We must also be concerned with the spacing between a marker in P_1's local area and another marker nearby, but outside of, P_1's local area. For example, refer to Figure 4.22(c) and 4.22(d) where the right rotation about node `lc` causes node `b` which, in Figure 4.22(c), is within P_1's local area to become too close to node `c` which was and continues to be outside of it.

The pictures in this theorem are similar to the pictures in Lemma 2. The differences are explained as we go through the pictures.

Deletion Case 1

In deletion Case 1, P_1 is doing a left rotation on `xp`. Because of Claim 2 in the proof of Lemma 2, no process has a marker on `xp` before the rotation of Case 1.

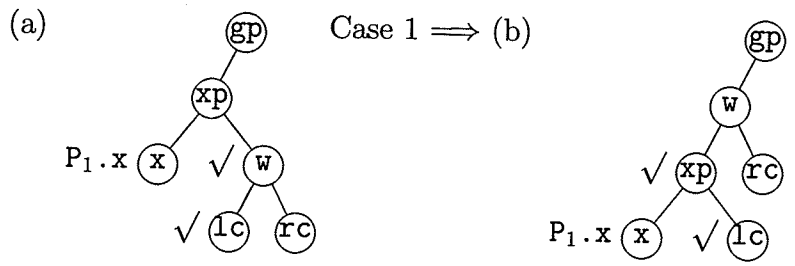
We do not have to consider the highest marker of P_2 being lower than P_1`.w` in the following Pictures 1(a) and 2(a) because, applying Lemma 1, we know that no markers are being moved to or from `x`, `lc`, and `rc`.

There cannot be situations as in Figure 4.22(d) in this transformation for the following reasons. In Picture 1 and Picture 2 corresponding to this transformation, after P_1 performs deletion Case 1, only `w` becomes a new

sibling of a node outside P_1's local area. (x and lc become new siblings, and xp and rc become new siblings; but, these nodes are in P_1's local area.) The new sibling of w is node gp's right child, and gp's right child cannot have a marker by the Spacing rule.

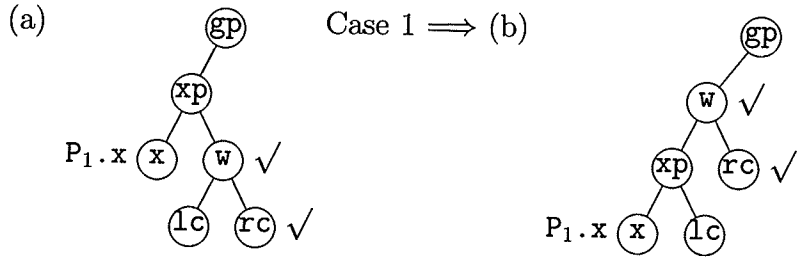
In Picture 1, no markers can be placed on rc, by the Spacing rule.

Picture 1:



In Picture 2, P_2 is located in rc's branch. By the Spacing rule, lc cannot have a marker.

Picture 2:



Claim 6 *If a marker is on P_1.xp after Case 1 as in Picture 1(b), then P_1's new gp and its right child have no markers.*

Claim 7 *If a marker is on rc after Case 1 as in Picture 2(b), then the new P_1.w has no markers, and P_1's new gp may have a marker.*

Neither of these cases results in two processes being brought too close to each other.

Deletion Case 1 → 2

As explained in Section 2.4.4, if P_1 does Case 2 after Case 1 (as in Picture 1(b) or 2(b)), P_1 exits immediately. Since Case 2 does no further rotations, we omit pictures for this transformation.

Deletion Case 1 → 3 → 4

In deletion Case 3, P_1 performs a right rotation on `w`. In deletion Case 4, P_1 does a left rotation on `xp`.

We consider Cases 3 and 4 for Pictures 1(b) and 2(b); and go through all the cases derived from Pictures 1(b) and 2(b), doing Cases 3 and 4 on them.

As in Lemma 2, we focus on the placement of P_2. We also try to put as many processes as possible in the pictures so that we are again constructing worst-case scenarios. First, P_2 is put as high as possible beneath P_1 so that P_2's highest marker is on P_1`.xp`, and we go through all the possible ways of placing P_2's other markers. Then we put the highest marker of P_2 down at P_1`.w` and apply the same procedure, and so on.

In Pictures 3 to 6, the highest marker of P_2 is on P_1`.xp` before Case 3 is done. Pictures 7 through 10 show cases where P_2 is located one level lower than the corresponding locations in Pictures 3 to 6. Finally, in Pictures 11 and 12, we go through cases where P_2 is located one level lower than the corresponding locations in Pictures 7 to 10.

By Lemma 1, we know that nodes `a`, `b`, and `rc` of the pictures in Case 1

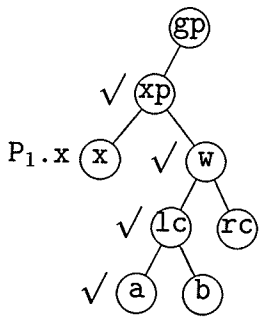
$\rightarrow 3 \rightarrow 4$ will have no markers moved to or from them during rotations. Therefore, we do not show the nodes below `a`, `b`, and `rc`. We also omit the cases where P_2's highest marker is on or below these nodes, except the special case of Picture 12.1 required to find situations such as that found in Figure 4.22(d).

For finding situations like that of Figure 4.22(d), we focus on pairs of nodes `a` and `x`, `b` and `rc`, and `lc` and gp's right child, in the following pictures in dealing with this transformation. After Case 3, nodes `a` and `w` become new siblings, and nodes `b` and `rc` become new siblings. (Nodes `x` and `lc` also become new siblings, but they are both in the local area of P_1.) We want to focus on nodes `a` and `x` rather than `a` and `w` because Case 4 immediately follows Case 3 and nodes `a` and `x` become new siblings. Nodes `b` and `rc` are still siblings after Case 4 is performed, and so we want to focus on these nodes as well. We also want to focus on nodes `lc` and gp's right child because they become new siblings after Case 4. (Nodes `xp` and `w` also become new siblings after Case 4, but they are both in the local area of P_1.) Finally, we note that gp's right child can have a marker only if Case 1 is done as in Picture 2.

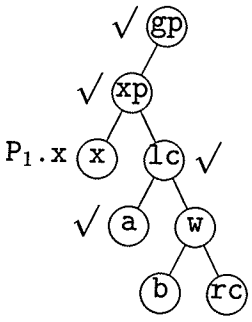
In Picture 3(a), by the Spacing rule, `b` and `rc` cannot have any markers.

Picture 3 (from Picture 1(b)):

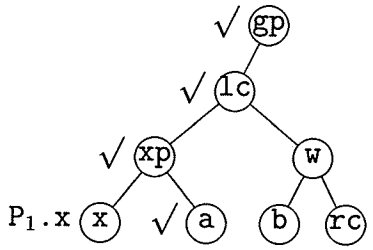
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



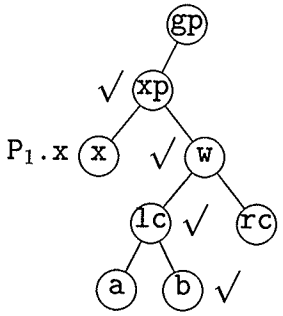
Case 4 $\Longrightarrow$ (c)



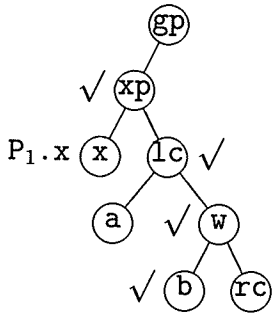
In Picture 4(a), the lowest marker of P_2 is on b instead of on a as in Picture 3(a). In Picture 4(a), a and rc cannot have any markers, by the Spacing rule.

Picture 4 (from Picture 1(b)):

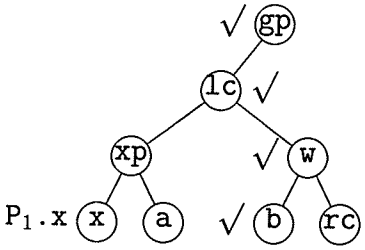
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)



If we apply the Spacing rule to Pictures 5(a) and 6(a) of Lemma 2 (on pages 102 and 102, respectively), we see that lc cannot have a marker. The result of applying the Spacing rule to Pictures 5 and 6 of Lemma 2 is that they both depict the same situation. Therefore, we will consider only one picture here, Picture 5.

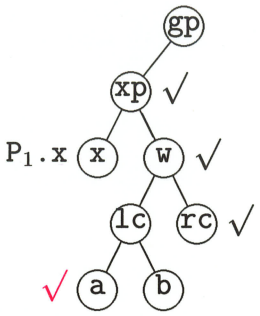
If we apply the Spacing rule to Picture 5 of Lemma 2, processes cannot

be on both a and b. Picture 5.1 shows what happens if a process is on a, and Picture 5.2 shows what happens if a process is on b, instead.

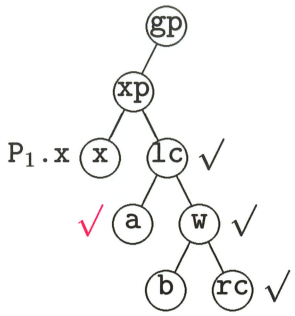
In Pictures 5.1(a) and 5.2(a), the third highest marker of P_2 is on rc instead of on lc as in Pictures 3(a) and 4(a). In Picture 5.1(a), P_3's marker is on a.

Picture 5.1 (from Picture 1(b)):

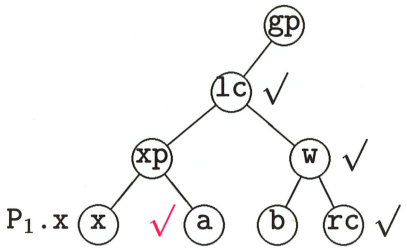
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



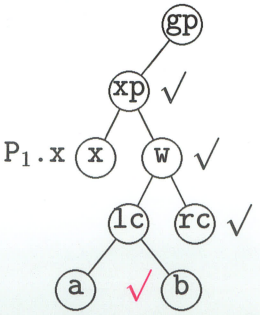
Case 4 $\Longrightarrow$ (c)



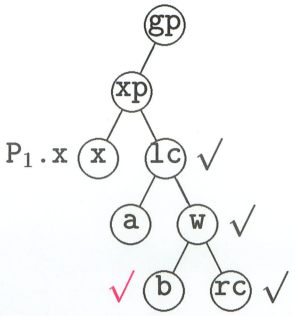
In Picture 5.2(a), P_3's marker is on b.

Picture 5.2 (from Picture 1(b)):

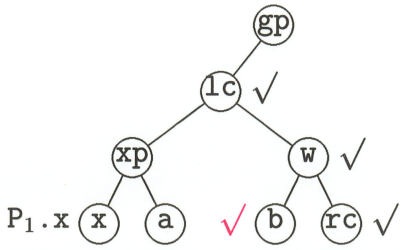
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)



In Picture 5.2(c), P_3 ends up being too close to P_2. This situation must be dealt with.

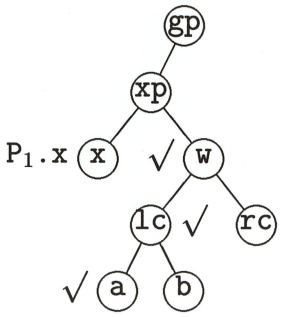
Picture 6: As described, exactly the same as Picture 5.

Pictures 7 through 10 show cases where P_2 is located one level lower than the corresponding locations in Pictures 3 to 6.

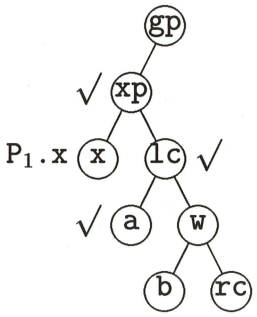
By the Spacing rule, in Picture 7, `b` and `rc` cannot have any markers.

Picture 7 (from Picture 1(b)):

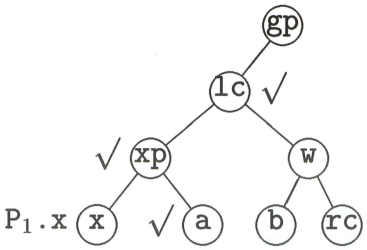
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



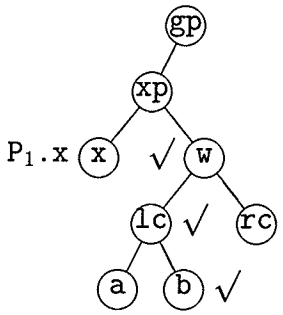
Case 4 $\Longrightarrow$ (c)



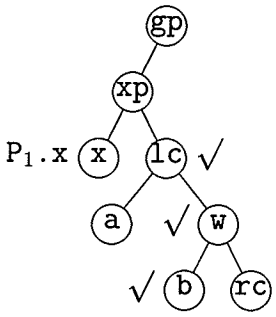
In Picture 8(a), the third highest marker of P_2 is on `b` instead of on `a` as in Picture 7(a). In Picture 8, `a` and `rc` cannot have any markers, by the Spacing rule.

Picture 8 (from Picture 1(b)):

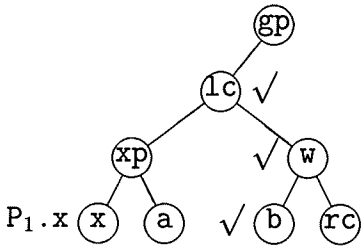
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)



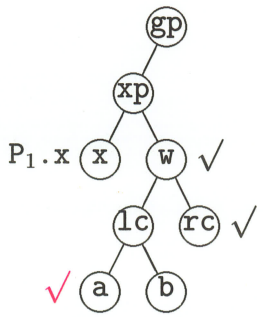
Because of the Spacing rule, Pictures 9 and 10 of Lemma 2 on page 104 now cannot have a marker on `lc`. Hence, Pictures 9 and 10 are the same. Therefore, we will, again, consider only one picture here, Picture 9.

If we apply the Spacing rule to Picture 9 of Lemma 2, processes cannot be on both `a` and `b`. Hence, we divide Picture 9 into Pictures 9.1 and 9.2 to see what happens when a process is on `a` or on `b`.

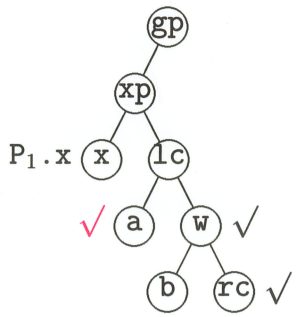
In Pictures 9.1(a) and 9.2(a), the second highest marker of P_2 is on `rc` instead of on `lc` as in Pictures 7(a) and 8(a). In Picture 9.1(a), P_3 is on `a`.

Picture 9.1 (from Picture 1(b)):

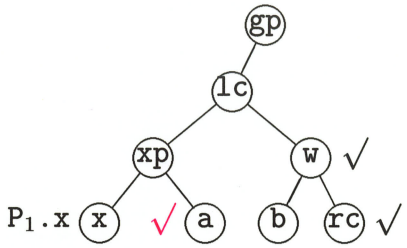
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



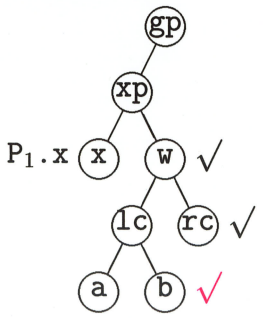
Case 4 $\Longrightarrow$ (c)



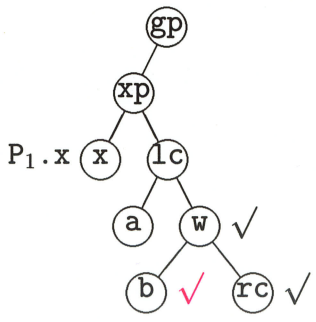
In Picture 9.2(a), P_3 is on b.

Picture 9.2 (from Picture 1(b)):

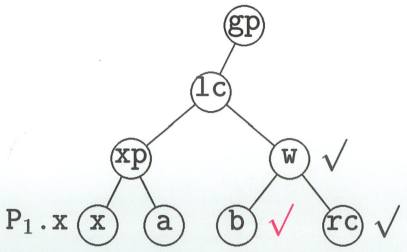
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)



In Picture 9.2(c), P_3 is too close to P_2. This situation will need to be dealt

with.

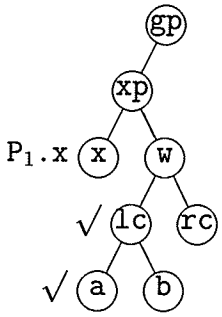
Picture 10: As described, exactly the same as Picture 9.

Finally, in Pictures 11 and 12, we go through cases where P_2 is located one level lower than the corresponding locations in Pictures 7 to 10.

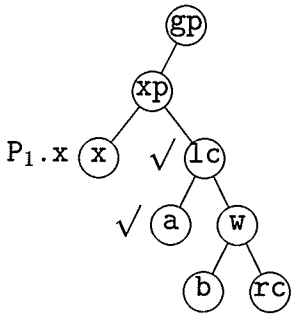
By the Spacing rule, in Picture 11(a), b and rc cannot have a marker.

Picture 11 (from Picture 1(b) or 2(b)):

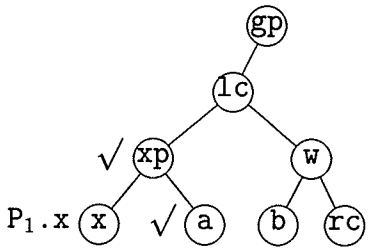
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



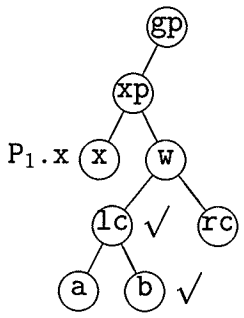
Case 4 $\Longrightarrow$ (c)



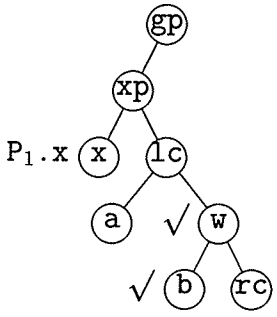
In Picture 12(a), P_2's second highest marker is on b instead of on a as in Picture 11(a). By the Spacing rule, in Picture 12, a and rc cannot have a marker.

Picture 12 (from Picture 1(b) or 2(b)):

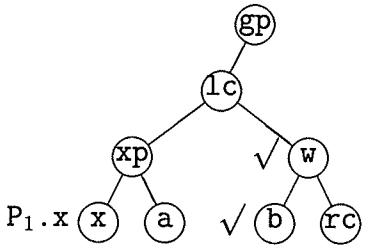
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)

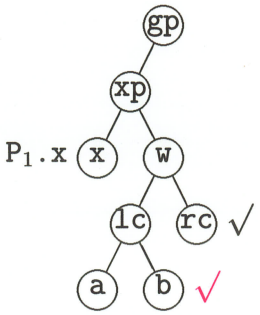


When we concluded the discussion of deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4 on page 106 in Lemma 2, we considered a variation of Pictures 11(a) and 12(a) of Lemma 2, where P_4 was on P_1.w, and showed that these were analogous to Pictures 9(a) and 10(a) of Lemma 2. Applying the Spacing rule to the variation of Pictures 11(a) and 12(a) of Lemma 2, where P_4 is on P_1.w, P_2 cannot be on lc, and the cases are the same as in Pictures 9.1(a) and 9.2(a). Hence, we omit these cases.

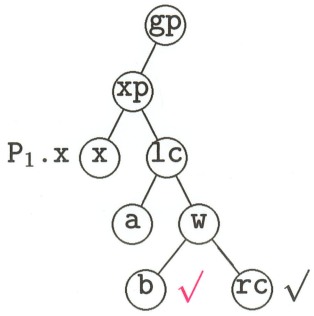
In Pictures 9.1(a) and 9.2(a), if P_2's highest marker is on rc, then we can simply remove the marker on w from Pictures 9.1 and 9.2 to illustrate this situation. We only draw the picture similar to Picture 9.2 (where the processes get too close). We call this Picture 12.1 so that Picture 13 corresponds to Picture 13 of Lemma 2.

Picture 12.1 (from Picture 1(b) or 2(b)):

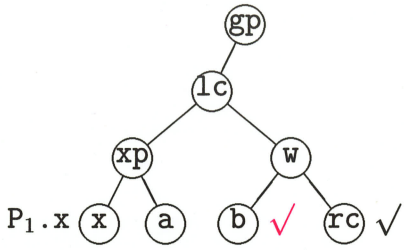
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



Case 4 $\Longrightarrow$ (c)



Picture 12.1 shows the same problem as in Figure 4.22(d). This situation, too, will need to be dealt with.

After going through the transformations of deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4, we have found three cases where two processes are brought too close to each other, shown in Pictures 5.2, 9.2, and 12.1.

Deletion Case 1 $\rightarrow$ 4

In deletion Case 4, we do a left rotation on `xp`.

By Lemma 1, we know that no markers are moved to and from `x`, `lc`, and `rc` in Pictures 13 and 14 illustrating deletion Case 1 $\rightarrow$ 4. Hence, we do not show the nodes located below `x`, `lc`, and `rc`. Also, we omit the cases where the highest marker of P_2 is located on or below these nodes.

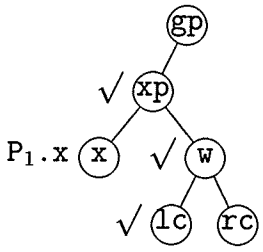
There cannot be situations like that in Figure 4.22(d) in this transformation. After Case 4, only node `w` becomes a new sibling of a node outside of

P_1's local area, gp's right child. (Nodes x and lc become new siblings, and nodes xp and rc become new siblings; but, they are all in the local area of P_1.) gp's right child can have a marker only if Case 1 is done as in Picture 2. If Case 1 is done as in Picture 2, then the highest marker of P_2 cannot be on w and, hence, even if gp's right child has a marker, there would be no problem.

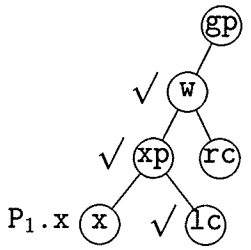
By the Spacing rule, in Picture 13, rc cannot have a marker.

Picture 13 (from Picture 1(b)):

(Case 1) $\Longrightarrow$ (a)



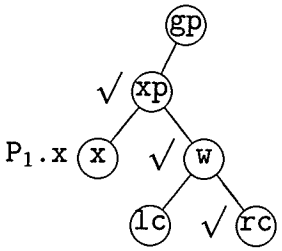
Case 4 $\Longrightarrow$ (b)



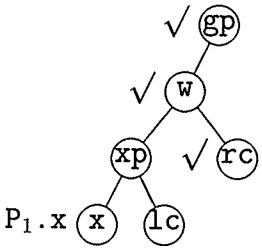
In Picture 14(a), P_2's third highest marker is on rc instead of on lc as in Picture 13(a). By the Spacing rule, in Picture 14, lc cannot have a marker.

Picture 14 (from Picture 1(b)):

(Case 1) $\Longrightarrow$ (a)



Case 4 $\Longrightarrow$ (b)



Imagine P_2 in Pictures 13(a) and 14(a) being another level lower. This situation cannot create any further problems because the resulting pictures will be identical to Pictures 13(b) and 14(b) except that the highest marker of P_2 would now be on xp in Picture 13(b) and on w in Picture 14(b). Therefore, we omit these cases.

In deletion Case 1 $\rightarrow$ 4, neither of the cases causes two processes to be brought too close to each other.

Deletion Case 3 $\rightarrow$ 4

We can omit pictures for deletion Case 3 $\rightarrow$ 4 because Claim 4 on page 108 (deletion Case 3 $\rightarrow$ 4 is analogous to Pictures 7 through 12) also applies here. Here, deletion Case 3 $\rightarrow$ 4 is analogous to Pictures 7 through 12.1.

Deletion Case 4

We can omit pictures for deletion Case 4 because Claim 5 on page 108 (deletion Case 4 is analogous to Pictures 1 and 2 of deletion Case 1) also holds.

We have now gone through all the deletion transformations. In all the deletion transformations, we found three cases that bring two processes too close to each other: in Pictures 5.2, 9.2, and 12.1. P_1 is doing deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4 or deletion Case 3 $\rightarrow$ 4 in the problem pictures. We now move on to consider insertion Cases 2 $\rightarrow$ 3 and 3.

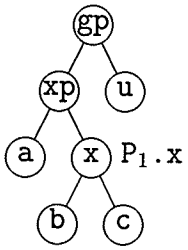
Insertion Case 2 $\rightarrow$ 3

In insertion Case 2, P_1 moves `x` to `xp` and performs a left rotation on the new `x`. Then, in Case 3, which follows Case 2, P_1 performs a right rotation on `x`'s grandparent, `gp`. By Claim 2, we know that `x`, `xp`, and `gp` do not have any markers as a process enters Case 2.

From Lemma 1, we see that no markers are moved to or from `a`, `b`, and

c in Picture 15. Therefore, we do not show the nodes located lower than a, b and c. By the Spacing rule, in Picture 15, no markers can be on a, b, c, or u. Since no markers are in the local area of P_1, we will omit the pictures showing the case processing.

Picture 15:



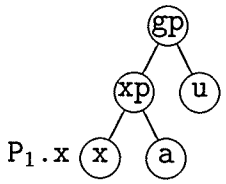
In the only picture of insertion Case 2 $\rightarrow$ 3, Picture 15, there are no markers, so no processes can be pulled too close to each other.

Insertion Case 3

In insertion Case 3, P_1 performs a right rotation on the grandparent of P_1.x, gp. By Claim 2, we know that x, xp, and gp do not have any markers as P_1 enters Case 3.

By Lemma 1, no markers are being moved to and from nodes x, a, and u in Picture 16. Thus, we omit the cases where the highest markers of processes are located on or lower than x, a, and u in Picture 16. (Moreover, by the Spacing rule, no markers can be on either a or u.) Since there are no markers in the local area of P_1, we omit the case processing in the picture.

Picture 16:



The only picture in insertion Case 3, Picture 16, has no markers and, therefore, no two processes can be brought too close to each other by the rotation.

We have now gone through all the pictures for insertions, and there were no cases that brought two processes too close to each other.

The problem pictures:

We identified three pictures from the deletion cases that brought two processes too close to each other: Pictures 5.2, 9.2, and 12.1. P_1 is either doing deletion Case $1 \rightarrow 3 \rightarrow 4$ or deletion Case $3 \rightarrow 4$ in the problem pictures.

The Four Additional Rules:

We present four rules that resolve the problems shown in the three pictures just listed. The rules involve processes directly working together.

The purpose of the first rule, the Move-Up rule, is for the process, P_1, doing the rotation to allow to move up one of the two processes that are brought too close to each other in the local area of P_1. This rule is intended to ensure that at least one of the two processes will be kept far enough away from any other process that will perform a rotation so that no double markers

can be created.

In all of the three problem pictures, P_1 performs deletion Case 4. Hence, after P_1 performs deletion Case 4, P_1 can determine if it should apply the Move-Up rule.

The Move-Up Rule: After P_1 performs deletion Case 4, if P_1 finds itself in the situation shown in Picture 5.2(c), 9.2(c), or 12.1(c), P_1 allows one of the two processes that are too close — P_2 or P_3 in the pictures — to move up to P_1's gp. P_1 allows the process which is located higher than the other one to move up. In Pictures 5.2 and 9.2, P_2 will be allowed to move up. In Picture 12.1, either P_2 or P_3 can be chosen, and we arbitrarily choose P_2 in our algorithm. Note that a process which is allowed to move up by P_1 is not constrained by the Spacing rule to maintain its distance from nodes marked by P_1. It is, still, constrained to maintain its distance from nodes marked by all other processes.

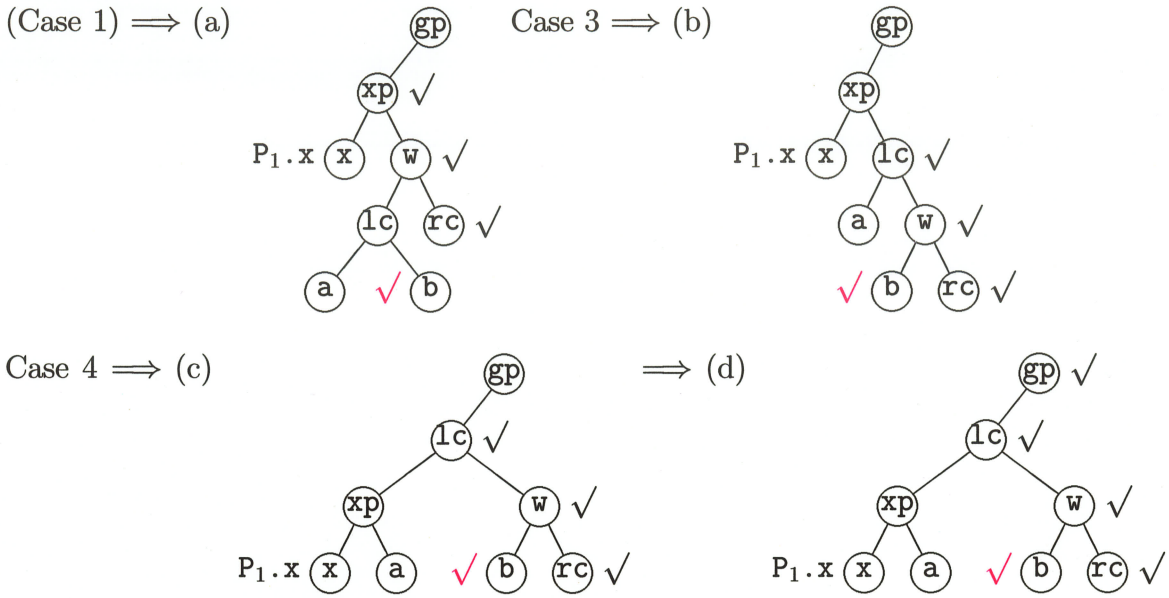
If the process allowed to move up is an insertion process, the insertion process may have to move up to P_1's ggp because an insertion process moves up two levels at a time; this situation results in the Insert-Jump-Ahead rule. We explain this situation with a picture just before describing the Insert-Jump-Ahead rule.

The process allowed to move up may do rotations and exit before reaching gp or ggp. These rotations, however, will not have any adverse affect on P_1 since P_1 has already done its fix-up processing and is simply waiting to exit.

In the following reproduction of Picture 5.2, we add part (d), which shows

the result of applying the Move-Up rule; P_2 moves up to node gp in part (d).

Expanded **Picture 5.2** (from Picture 1(b)):



The process performing rotations, P_1, must be able to determine that, after its rotations, two processes beneath P_1 are located too close to each other. In part (a) of Expanded Picture 5.2, b's flag is obtained temporarily to perform a right rotation on w for Case 3. Also, note that rc's flag is set because rc is in P_1's local area. In part (b), P_1 now must not release the flags of b and rc. Then, after the Case-4 processing shown in part (c), P_1 is able to detect that each of b and rc has a marker so that P_1 needs to apply the Move-Up rule. This algorithm also applies to the situations of Pictures 9.2 and 12.1, the other situations where two processes are brought too close to each other after deletion Case 4.

Claim 8 *If P_1 applies the Move-Up rule to the situations shown in the three problem pictures, at least one of P_2 and P_3 in the pictures will be kept far*

enough away from a process performing a rotation so that the rotation cannot create double marker problems.

Proof: In the three problem pictures, suppose P_1 applied the Move-Up rule. We call the process that is allowed to move up P_2. Since we will consider all the possible locations of P_2 in going through all the possible case transformations, it does not matter whether P_2 or P_3 is chosen.

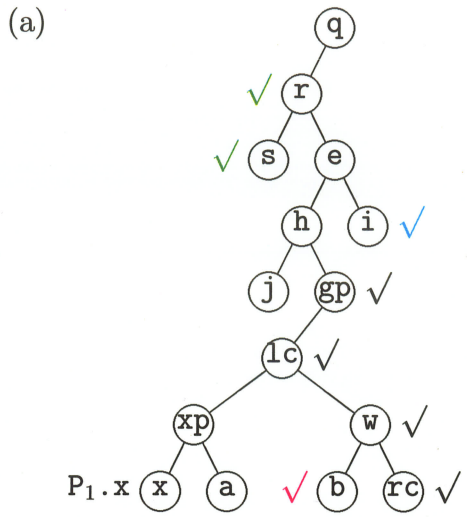
We assume here that P_2 has not done a rotation and exited on its way to P_1's gp or ggp. If P_2 did a rotation and has exited, P_2 would be gone from the picture, and the discussion would then be irrelevant.

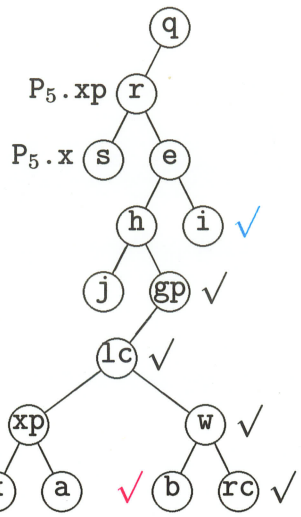
By performing the Move-Up rule, P_1 has not left a space between P_2 and P_3's markers. Instead, P_1 has ensured at least three levels of distance between the highest markers of P_2 and P_3.

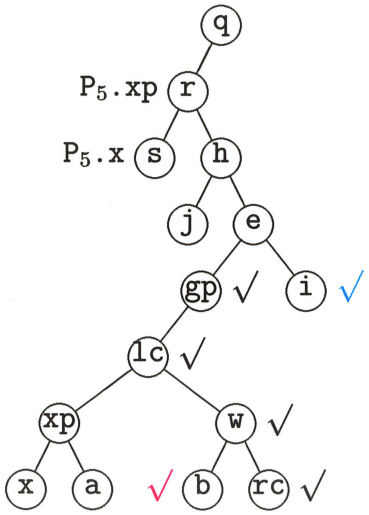
This situation is different from what we looked at before in the theorem. Before, we assumed a space between the markers of two processes. Now, one process's marker can be on a node adjacent to a node that contains another process's marker.

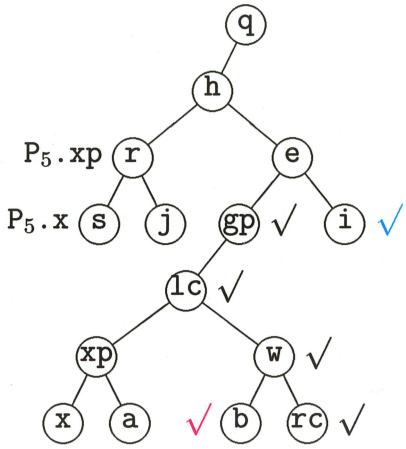
We consider another process P_4 located above but near P_2 and P_3, where P_4 is about to perform a rotation. We will consider all possible case transformations for P_4 to show that the double marker problem is not created. First, however, we show that a process P_4 can be located above P_2 and P_3, such that P_2 and P_3 are near enough that we need to check that the double marker problem cannot occur.

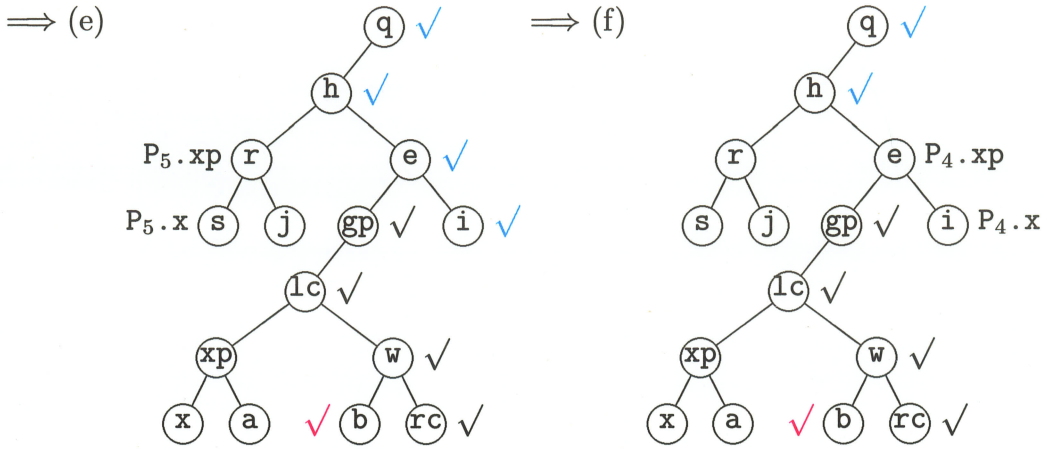
Below, we take the Expanded Picture 5.2 from page 136 and show more nodes above gp to demonstrate how P_4 could move into the necessary position above P_2 and P_3:

(a) 

$\Longrightarrow$ (b) 

Case 3 $\Longrightarrow$ (c) 

Case 4 $\Longrightarrow$ (d) 



In part (a) of the above picture, deletion process P_5, shown in green, is located in the subtree rooted at node `s`. (No processes can be on nodes `h` and `j` because the Spacing rule keeps processes from approaching so near to P_2's markers.) In part (b), P_5 moves up — this picture is analogous to Picture 12.1(a) (page 130) of the proof of this theorem. In parts (c) and (d) of the above picture, as in Picture 12.1, P_5 performs Cases 3 and 4. In part (e), P_5 applies the Move-Up rule, so P_4 is allowed to move up and P_5 exits. In part (f), P_4 is unconstrained by markers and moves further up, and P_2's highest marker is on `P4.w` (assuming P_4 is a deletion process). P_4 is now sufficiently close to P_2 to cause concern. This is just one example, though there are other ways that P_4 can get above but near P_2 and P_3.

Now we examine all the case transformations listed in Figure 4.17 that P_4 could go through, to ensure that no double markers occur, assuming that the Move-Up rule has been applied to put P_2 and P_3 close together. P_4 will serve the same purpose as P_1 in the previous pictures of this theorem. We can simply substitute P_4 into P_1's place as we go through the pictures.

We first assume that P_2 is a process that is doing a deletion. Later, we

will consider cases where P_2 is doing an insertion. We divide the cases this way because there were no markers present in the previous pictures for the insertion cases of this theorem, but we now find that P_3's marker can now be present in P_2's local area if P_2 is doing an insertion.

If P_4 is in deletion Case 1, then Picture 1 and Picture 2 on (page 120) of this theorem show that P_2 and P_3's markers cannot both be present in the local area of P_4 because P_3's highest marker is on the sibling of the node that has P_2's lowest marker, which cannot be in P_4's local area. Hence, there cannot be double marker situations after deletion Case 1.

If P_4 is in deletion Case $1 \rightarrow 2$, no more rotations are done after Case 1 and, therefore, this situation is the same as in the previous paragraph. Thus, no double markers are created.

If P_4 is in Case $1 \rightarrow 3 \rightarrow 4$, then Pictures 3 to 12.1 (pages 122–130) of this theorem show that P_3's highest marker, on the sibling of P_2's lowest marker, can be, at the highest, on `a` or `b` in part (a) of each picture, and by Lemma 1, no markers are moved to or from `a` or `b`. Hence, double markers are prevented in this case as well.

If P_4 is in deletion Case $1 \rightarrow 4$, then Pictures 13 and 14 (page 131) of this theorem show that P_3's highest marker can be, at the highest, on a child of `lc` or `rc`. Since `lc` and `rc`'s children are located outside P_4's local area, no markers of P_2 would be moved to the nodes that have P_3's markers, thereby preventing double marker situations.

The cases where P_4 is in deletion Case $3 \rightarrow 4$ or Case 4 have already been covered by previous cases; see Claims 4 and 5 on pages 108 and 108.

Finally, if P_4 is inserting, then Pictures 15 and 16 (pages 133 and 134) of

this theorem show that no markers are present in the local area of P_4. Thus, the double marker problem, again, cannot be created.

After the rotations done by P_4, the highest markers of P_2 and P_3 are at least three levels apart, just as they were before the rotations. This is shown below.

In Picture 1 on page 120 (of this theorem, substituting P_4 into P_1's place and assuming the Move-Up rule is applied), P_2's second highest marker is on node `lc` before and after the rotation. P_3 is in the subtree rooted at node `lc` before and after the rotation. Also, P_3's highest marker, being on the sibling of the node that contains the lowest marker of P_2, is located too far away from P_4 to be moved by P_4's rotation. Similar reasoning applies to Pictures 2, 5.1, 5.2, 6, 7, 8, 9.1, 9.2, 10, 11, 12, 12.1, 13, and 14.

If P_4 is doing an insertion, again, Pictures 15 and 16 show that no markers are present in P_4's local area, and no markers will be moved by P_4's rotations. Only Pictures 3 and 4 now remain to be considered.

In Picture 3(a), P_3's highest marker can be, at the highest, on node `b`, which is the sibling of node `a`. After the rotations, Picture 3(c) shows that `b` is still at the same level as node `a`. P_3 cannot move up to node `w` because of the Spacing rule. Therefore, the resulting situation did not bring the highest markers of P_2 and P_3 closer than three levels apart. (Moreover, if P_3's highest marker is on node `b`, now there is a space between P_2's and P_3's markers as shown in Picture 3(c). This case is covered by the pictures in this theorem.) Similar reasoning applies to Picture 4.

Not only are we assured of the three levels of distance between the highest markers of P_2 and P_3 after P_4's rotation, but we are also assured that

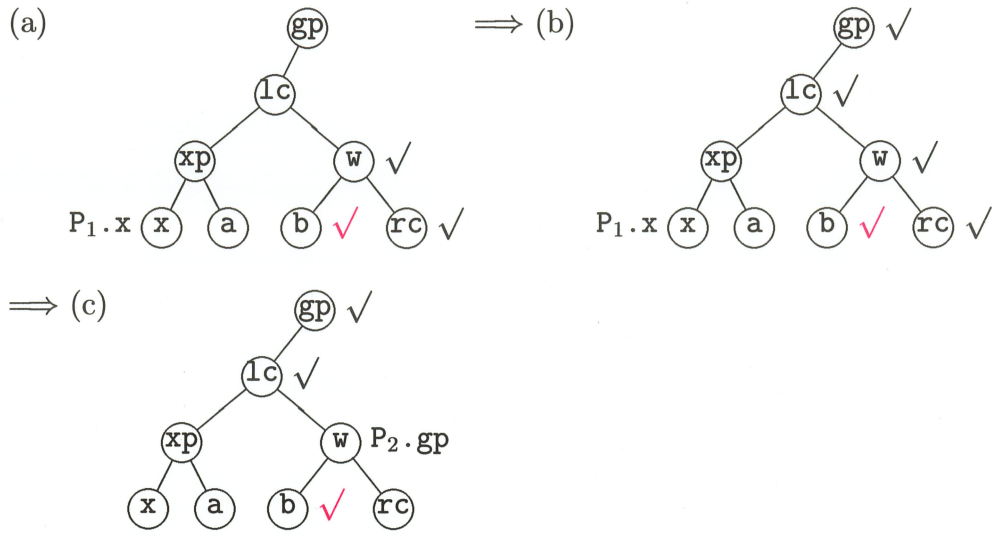


Figure 4.23: Part (a) is from Picture 9.2(c). P_2, in black here, is an insertion process.

the highest marker of P_3 will not be separated from its other markers by P_4's rotation because we have just seen that P_3's highest marker is too far away from P_4 and cannot be moved from the node it is currently located at. Because the other markers of P_3 are even further away from P_4, the rotation performed by P_4 cannot move them from their current locations.

We now assume that P_2 is a process doing an insertion. Then, after P_2 moves up, P_3's marker can be in the local area of P_2. To illustrate this, consider Figure 4.23.

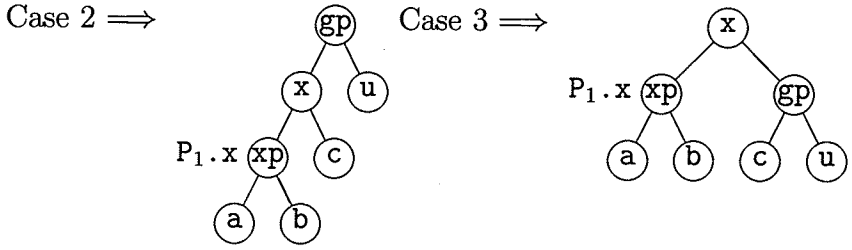
In Figure 4.23(a), P_1 has done deletion Case-4 processing, and P_2 is in insertion Case 1. P_1 detects that P_2 and P_3 are too close to each other and allows P_2 to move up. In part (b), P_2 moved up to gp, and P_1 exits. In part (c), P_2 moved up once more by doing insertion Case 1. The node labelled w

in the picture is now P_2.x's grandparent, which means P_3's highest marker is on P_2.x's uncle. If we think of P_2 as P_1 in Pictures 15 and 16 of this theorem, then a marker can now be added to node a, b, c, or u in Picture 15 and node a or u in Picture 16.

The above situation, however, does not present a problem since only one marker of one process, P_3's marker in our example, would be added in Pictures 15 and 16.

For Picture 15, P_3's marker would be on one of the nodes a, b, c, or u, depending on where the local area of P_3 is located. We know that, by Lemma 1, no markers are moved to or from a, b, and c. Furthermore, if a marker is on u, the marker stays at u after the rotations as shown in Picture 15 (page 110) of Lemma 2. Hence, no double markers can be created.

In Picture 15, after P_2 performs insertion Cases 2 and 3, a and b become siblings, and so do c and u as the following picture shows:
(from Picture 15)

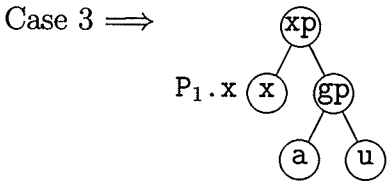


Since there would be at most one marker on these nodes, no two markers would be placed on siblings as in Picture 12.1, ensuring that insertions do not result in problems.

Likewise, in Picture 16, P_3 would be on one of a or u, but not both. We

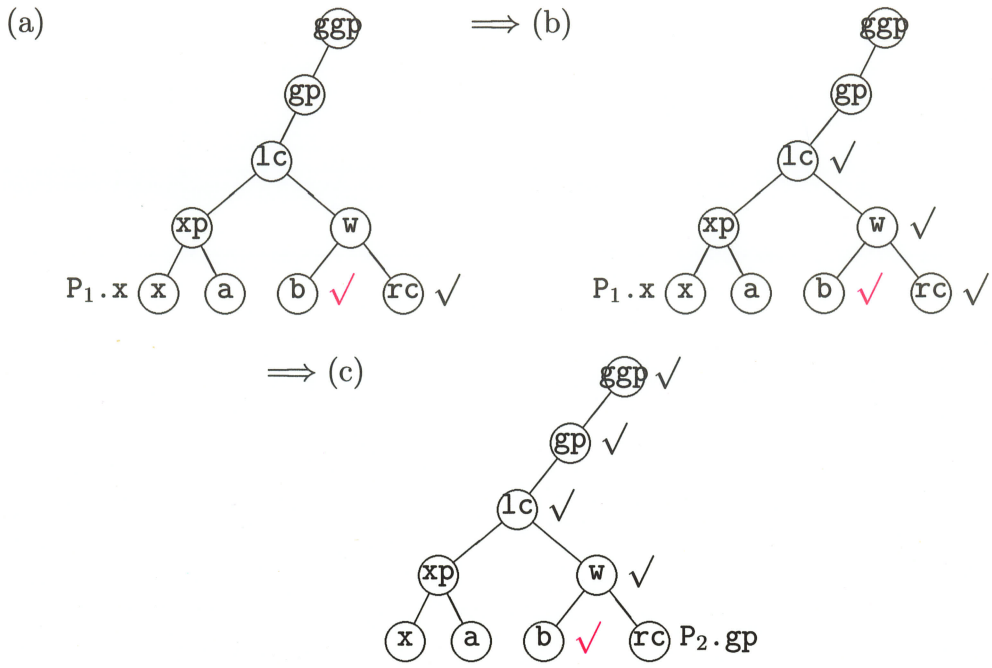
know, by Lemma 1, that no markers are being moved to and from x, a, and u. Hence, the rotation of P_4 does not move P_3's marker and does not create a double marker problem.

In Picture 16, after P_2 performs insertion Case 3, a and u become siblings as the following picture shows:
(from Picture 16)



Since there would be at most one marker on a and u, no two markers would be placed on the siblings, ensuring that insertions do not result in problems. □ End of Claim 8.

In Figure 4.23(a), P_2 is an insertion process, and P_1 has discovered that it must apply the Move-Up rule. If the highest marker of P_2 is on rc instead of on w, then P_2's highest marker will be moved up to lc first, since P_2 moves up two levels at a time in insertion Case 1. P_2 will not exit at that time if it has to perform Case 1 again. Allowing P_2 to advance only up to lc, however, provides only two levels of distance between the top markers of P_2 and P_3. As Picture 6 on page 102 of Lemma 2 shows, two levels of distance is not sufficient. Therefore, P_2 must be allowed to move up to ggp as shown in the following:



To allow P_2 to move up to ggp in the above picture, P_1 needs to have the flag of ggp. This requirement results in the Insert-Jump-Ahead rule which will be described next.

The Insert-Jump-Ahead Rule: If P_1 is in deletion Case 3, and did not perform Case 1 and, hence, did not release its four markers yet, then P_1 must retain the flags of the two lowest nodes that contain its markers when releasing markers before a rotation for Case 3. These two nodes are ggp and gp.

Note that the Insert-Jump-Ahead rule is applied without knowing whether the Move-Up rule will be applied but ensures that the Move-Up rule will always work, if it is applied.

The Move-Up rule is applied only when a process is in Case $1 \rightarrow 3 \rightarrow 4$

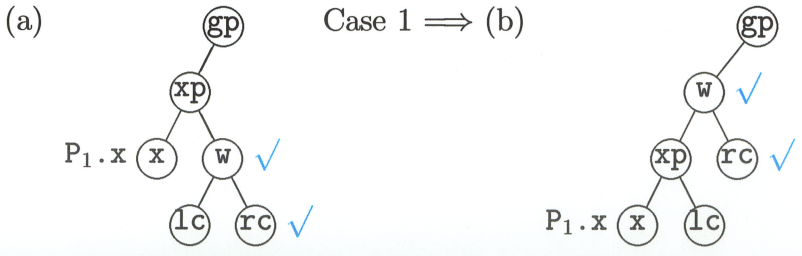
or in Case 3 → 4. The Insert-Jump-Ahead rule is applied only for Case 3 → 4. We now show that when a process is in Case 1 → 3 → 4, the Insert-Jump-Ahead rule is unnecessary.

If P_1 was in deletion Case 1, P_1 already has released the markers before Case 1's rotation and has retained the flag of the node that had its lowest marker, as explained in Section 4.1.2's explanation of why we use only DCAS (page 66). After Case 1, the node whose flag was retained becomes P_1.x's great grandparent, ggp. Therefore, if P_1 is in Case 4 preceded by Case 1 and needs to apply the Move-Up rule, P_1 already has the two necessary flags.

We now explain the need for the Go-First rule.

In Pictures 5.2(c) and 9.2(c) on pages 124 and 127, gp and its right child have no markers, since Pictures 5.2(a) and 9.2(a) are derived from Picture 1(b) on page 120 and Claim 6 on page 120 applies. Therefore, moving a process up to gp or ggp by the Move-Up rule is safe.

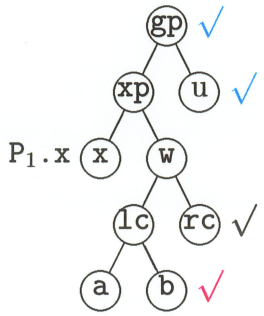
However, Picture 12.1(a) on page 130 may be derived from Picture 2(b) on page 120 and gp and its right child may have markers; see Claim 7 on page 120. For instance, suppose Picture 12.1(a) is derived from Picture 2(b) and w and rc of Picture 2(b) have markers as follows:



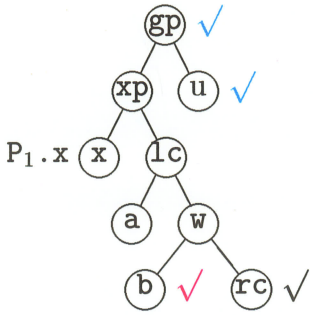
In this situation, gp and its right child in Picture 12.1 will also have markers because they correspond to w and rc in Picture 2(b). This situation is shown

as follows (modified from Picture 12.1):

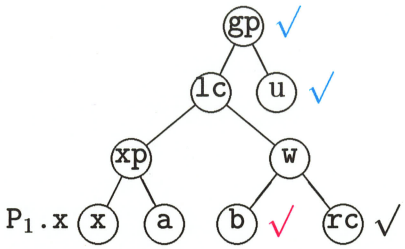
(Case 1) $\Longrightarrow$ (a)



Case 3 $\Longrightarrow$ (b)



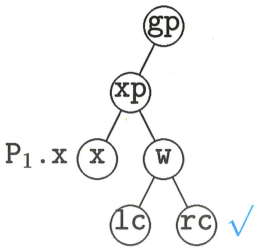
Case 4 $\Longrightarrow$ (c)



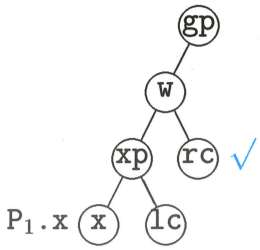
As we can see, in part (c) of the above picture, gp already has a marker from P_4. Therefore, P_1 must allow P_4 to proceed ahead before allowing P_2 to move up to gp or ggp.

Consider another case similar to the previous example. In Picture 2(b), suppose w did not have a marker and rc did, as follows:

(a)

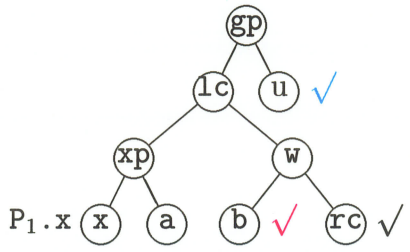


Case 1 $\Longrightarrow$ (b)



Then gp in Picture 12.1 will not have a marker, but its right child will have a marker as follows:

(modified from Picture 12.1(c))



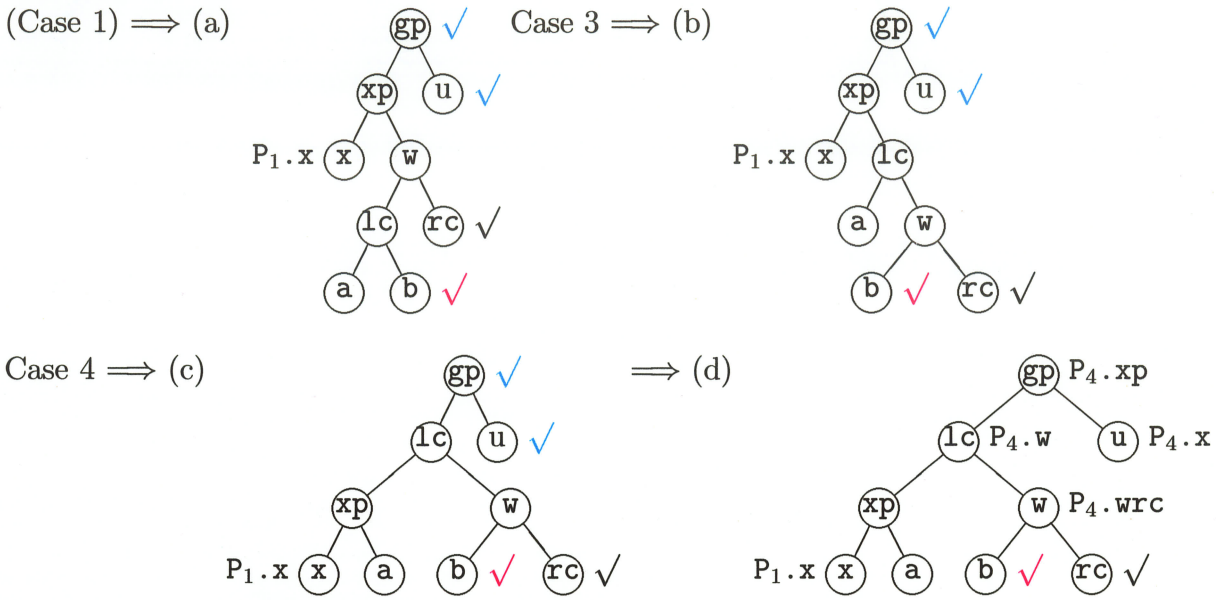
In this picture, P_1 still must allow P_4 to advance first since moving P_2 up to gp or ggp will bring P_2 too close to P_4.

The Go-First Rule: The Go-First rule requires the rotating process, P_1, to let any process located on gp or gp's right child to move ahead before applying the Move-Up rule. The Go-First rule requires that P_1 must not release the flag of rc (which is u in the above picture) in Picture 2(b) until P_1 exits. Like in the Move-Up rule, the process being allowed to go first is not constrained by the Spacing rule to maintain its distance from nodes marked by P_1.

Before doing the Move-Up-rule processing, P_1 checks to see if it should apply the Go-First rule. If P_1 determines that the Move-Up rule is not applicable, then the check for the Go-First rule is not necessary.

If the process P_4 on gp or u is moved up and performs a rotation, then some of the nodes in the local area of P_1 may become part of the local area of P_4 and may be included in P_4's rotation. We draw a picture to illustrate this situation as follows:

(modified from Picture 12.1)



In part (a) of the picture, `w` is marker-free as a result of Claim 7, because part (a) is derived from Picture 2(b). Also, `lc` is marker-free because of the Spacing rule. After the rotations of Cases 3 and 4, the result in part (c) shows that `w` and `lc` still have no markers. In part (d), P_4 moved up and the node labelled `lc` became P_4`.w`. P_4`.wrc`'s children each has a different marker (from P_2 and P_3, respectively). Since P_2 and P_3 have a space between them (provided by the node labelled `w` which is marker-free), we are assured that no double-marker situation is going to occur.

However, we must go through all the possible locations of P_4 to make sure that P_4's rotations will not create any problems. In the following claim, we go through all the transformation cases of P_4, find two problem situations caused by P_4's rotations, and resolve these situations.

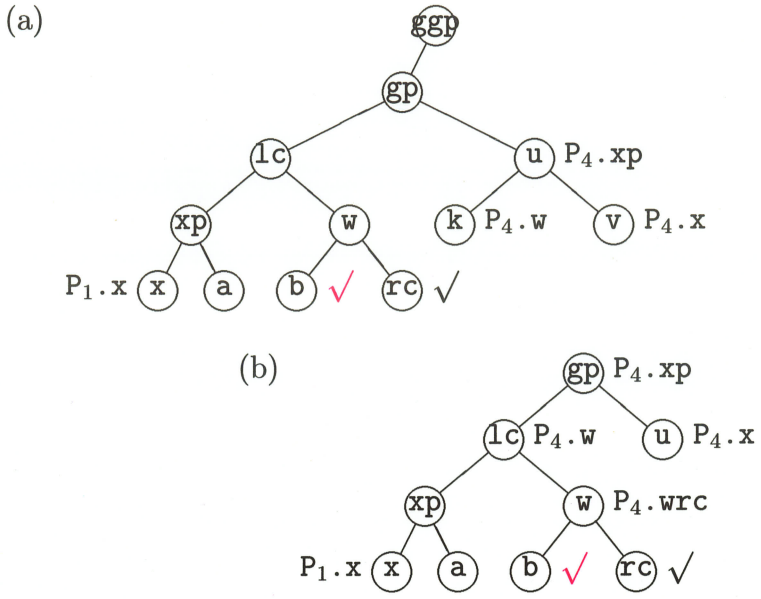


Figure 4.24: Four possible locations of P_4.x. P_4 is a deletion process. (Picture 1 of 2.)

Claim 9 *(a) P_4's rotations can cause two problem situations. (b) The two problem situations are resolved in the proof of this claim.*

Proof: In the pictures of this proof, the naming convention is the same as in Lemma 2; refer to page 95. We add another process P_6 which is coloured purple.

Proof of Part (a) of Claim 9

We consider four possible locations for P_4.x, shown in Figure 4.24. If P_4.x is located any other place, the situations would be similar to the ones shown in Figure 4.24 or P_4 would be far enough away from P_1 so that P_4's

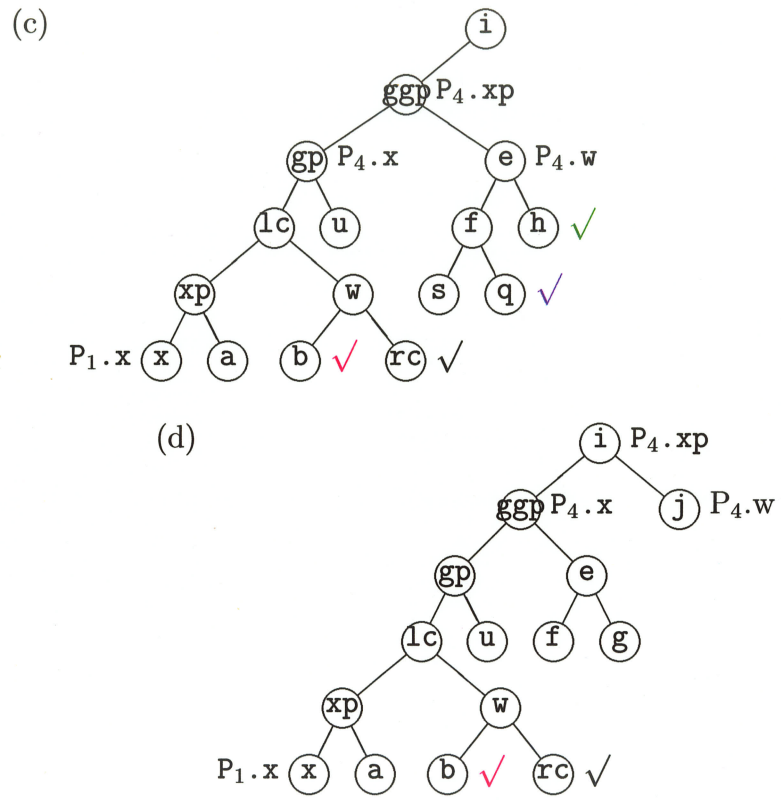


Figure 4.24: Four possible locations of $P_4.x$. (Picture 2 of 2.)

rotations would not affect P_1 in any adverse way, as will be discussed in more detail when we consider Figure 4.24(a) and Figure 4.24(d).

For each picture in Figure 4.24, we will go through all the possible transformation cases of P_4 to see if there are any problems. We particularly look for situations where a marker can be moved near to node gp or ggp in Figure 4.24 by rotations or by the application of the Move-Up rule by P_4. Because P_1's application of the Move-Up rule requires P_2 to be moved up to gp or ggp, if P_4's rotations or application of the Move-Up rule moves a marker onto, or on a node adjacent to, gp or ggp, then when P_2 moves up there, it will be too close to the marker of the process moved by P_4.

In Figure 4.24, P_4 is a deletion process. If P_4 were an insertion process, Pictures 15 and 16 (pages 133 and 134) of this theorem show that no markers would be inside P_4's local area. Hence, there would be no problems such as those discussed in the previous paragraph. However, we must consider insertion cases of P_4 because node gp or ggp of P_1 may be rotated away by P_4. In such situations, P_4 will transfer the flag of P_1's new gp or ggp to P_1 when P_4 exits. Also, the old gp or ggp's flag will be released by P_1. At the end of the proof of part (a), we will go through the insertion transformation cases for P_4 to identify these situations.

After going through the transformation cases of each of the four pictures in Figure 4.24, we will have found two problem situations. In the proof of part (b) of the claim, we will resolve these problems. Now we consider each picture of Figure 4.24.

As we go through the pictures of this theorem, we will substitute P_4 into P_1's place. Also, we will substitute P_5 and P_6 into P_2's and P_3's places, re-

spectively.

Figure 4.24(a)'s Transformation Cases of P_4

In Figure 4.24(a), P_4.x is on node gp's grandchild. By the Spacing rule, P_4.w, which is node k in Figure 4.24(a), cannot have a marker.

If P_4 is in deletion Case 1, then the situation is similar to Pictures 1 and 2 of this theorem. In Pictures 1 and 2 (page 120), P_5's highest marker will be on node lc or rc. So, after the rotation, no markers are moved too close to node gp or ggp. Node gp in Pictures 1 and 2 corresponds to node gp of Figure 4.24(a). Hence, after the rotation, no markers are moved too close to node gp or ggp in Figure 4.24(a).

If P_4 is in deletion Case 1 $\rightarrow$ 2, no more rotations are done after Case 1 and, therefore, this situation is the same as in the previous paragraph. Thus, no markers are moved too close to gp or ggp in Figure 4.24(a).

If P_4 is in deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4, then P_4.w can have a marker after Case 1. Hence, the situation is the same as in Pictures 7 to 12.1 (pages 125–130) of this theorem. Case 1 increases the distance between P_4.x and the node gp of Figure 4.24(a) by one additional level. Hence, node gp in Pictures 7 to 12.1 corresponds to node gp's right child in Figure 4.24(a). Pictures 7 to 12.1 show that no nodes will move up to gp in Pictures 7 to 12.1, ensuring that there will be a single space between a marker in P_4's local area and the node gp in Figure 4.24(a). However, if situations such as those shown in Pictures 9.2 and 12.1 (pages 127 and 130) happen to P_4, then P_4 must apply the Move-Up rule. Also, the following problem occurs.

Problem 1: If P_4 applies the Move-Up rule, P_4 will allow a process, P_5, located below P_4, to move up to node gp's child or node gp in Figure 4.24(a). Then, when P_1 applies the Move-Up rule after P_4 exits, P_5 could already be on P_1's gp so P_2 cannot be allowed to move up to there. If P_5's highest marker is on P_1's gp's child, P_2 can move up to P_1's gp or ggp, but would still then be too close to P_5.

If P_4 is in deletion Case $1 \rightarrow 4$, the situation corresponds to Pictures 13 and 14 (page 131). After Case 1, a marker can be brought up to node P_4.w. Then, in Pictures 13 and 14, a marker could be, at the highest, on node w. Since node w of Pictures 13(b) and 14(b) corresponds to node gp's grandchild in Figure 4.24(a) (due to Case 1 providing a space between node gp and P_4.x in Figure 4.24(a)), no marker is brought too close to gp or ggp in Figure 4.24(a).

If P_4 is in deletion Case $3 \rightarrow 4$, then the situation corresponds to Pictures 7 to 12.1 (pages 125–130) of this theorem; see Claim 4 on page 108. In Figure 4.24(a), it is worth noting that, in this transformation, the distance between gp and P_4.x is not increased by Case 1 as it is in Case $1 \rightarrow 3 \rightarrow 4$. Hence, the Move-Up rule by P_4 can move P_5 up to gp or ggp of Figure 4.24(a), whereas in Case $1 \rightarrow 3 \rightarrow 4$ P_5 could move at highest to node gp. Problem 1 happens in this transformation as well.

The case where P_4 is in deletion Case 4 is analogous to Case 1 transformation; see Claim 5 on page 108.

Problem 1 can occur if P_4.x is located two levels lower than in Figure 4.24(a) since P_5 can then be moved up to gp's child. Hence, if P_4.x is

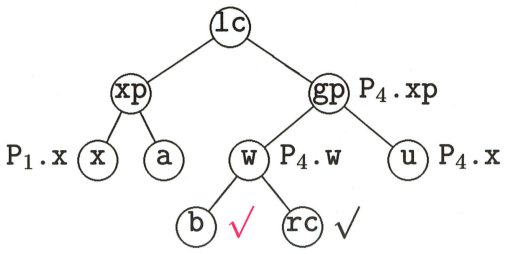
located up to two levels lower than in Figure 4.24(a), Problem 1 can occur. However, if P_4.x is located further below, P_4 would be far enough away from gp and cannot cause Problem 1.

Before moving on to Figure 4.24(b), we note that node u of Figure 4.24(a) can be rotated away from P_1 by P_4's Case 1 or Case 4 rotation. Therefore, if such a situation happens, P_1 needs to know that its gp has a new right child and P_4 must not release the flag of the new child of gp when it exits. The reason for retaining this flag will be evident when we describe the resolution to Problem 1.

Figure 4.24(b)'s Transformation Cases of P_4

In Figure 4.24(b), P_4.x is on gp's child.

If P_4 is in deletion Case 1, then the situation becomes as in the following picture:



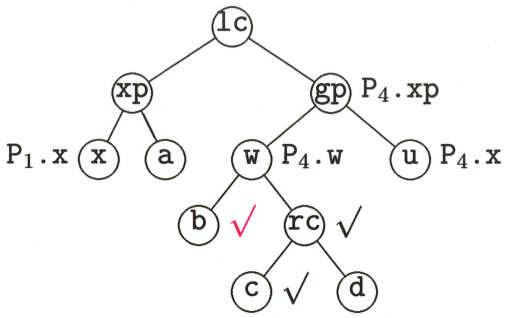
The picture shows that no markers are moved too close to P_1's gp and, hence, if P_4 exits after Case 1, there will be no problems. After P_4 exits, however, P_1 must know that node lc (instead of gp) or node ggp are now the nodes that P_2 should be allowed to move up to.

If P_4 is in deletion Case 1 $\rightarrow$ 2, then the situation is the same as in the previous paragraph since no more rotations are done after Case 1 and P_4 soon

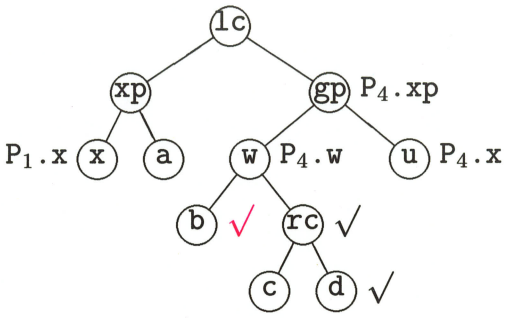
exits.

If P_4 is in deletion Case $1 \rightarrow 3 \rightarrow 4$, either P_4 applies the Move-Up rule on P_2 and P_3 so that P_1 does not have to apply the Move-Up rule or P_4's rotation moves markers so that the Move-Up rule is no longer applicable. In the picture for deletion Case 1 just shown, we add two more nodes below node `rc` to help us illustrate all the possible marker movements. Depending on the location of P_2, there can be two situations as follows:

(a)

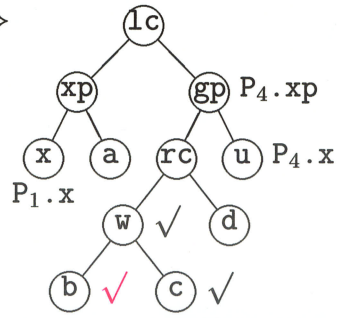


(b)

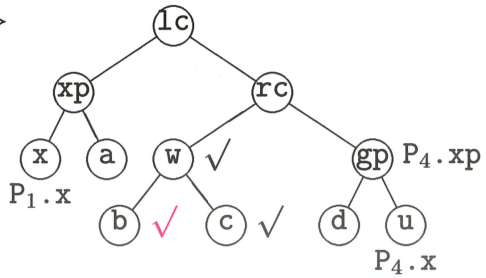


In part (a) of the above picture, P_2's second highest marker is on node `c`. In part (b), P_2's second highest marker is on node `d`. We now show what happens when Cases 3 and 4 are performed on each of these pictures. The following picture shows the result after Cases 3 and 4 are done to part (a):

Case 3 $\Longrightarrow$



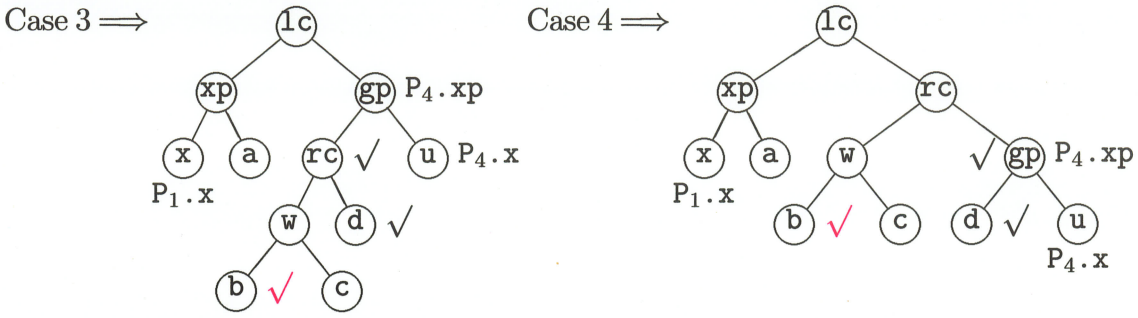
Case 4 $\Longrightarrow$



The resulting picture corresponds to Picture 9.2(c) (page 127) of this theorem. If such a situation happens, P_4 will apply the Move-Up rule and then exit. Then P_1 does not have to apply the Move-Up rule, because P_4 already

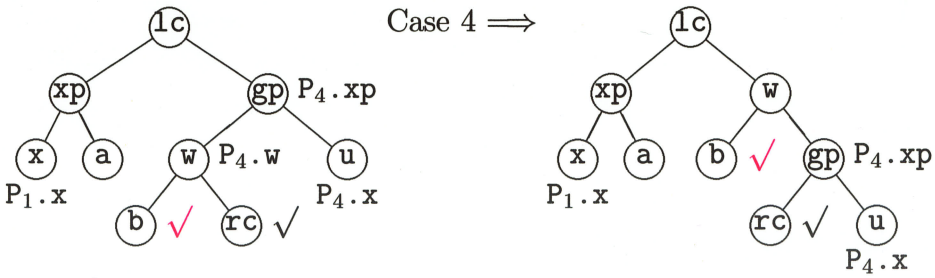
has moved P_2 up.

The following picture shows the result after Cases 3 and 4 are done to part (b):



The result of the above picture shows that P_2 and P_3 are now far apart so that the Move-Up rule is not applicable. Therefore, the two situations of deletion Case $1 \rightarrow 3 \rightarrow 4$ have no problems.

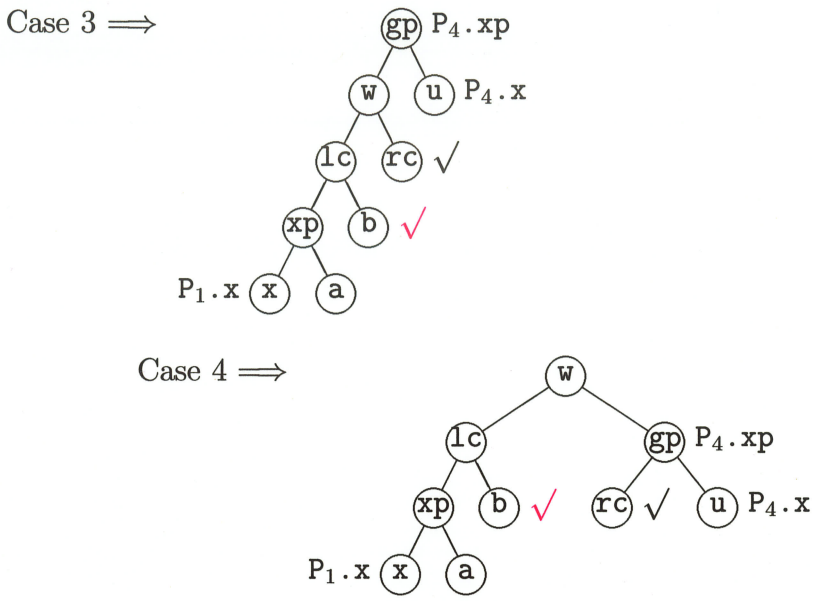
If P_4 is in Case $1 \rightarrow 4$, then node `rc` will be a new sibling of node `u` in the picture as follows:



As the result of the above picture shows, P_2's and P_3's markers now are sufficiently far from each other, so there will be no problems.

If P_4 is in deletion Case $3 \rightarrow 4$, then the situation corresponds to Pictures 7 to 12.1 (pages 125–130) of this theorem. Also, nodes `b` and `rc` in Figure 4.24(b) will each have new siblings as shown in the following picture:

(from Figure 4.24(b))



Hence, because P_2 and P_3 are now widely separated, P_1 does not need to apply the Move-Up rule.

If P_4 is in deletion Case 4, then the situation is analogous to deletion Case 1; see Claim 5.

Figure 4.24(c)'s Transformation Cases of P_4

In Figure 4.24(c), P_4.x is on node gp. P_4's w (node e) cannot have a marker by the Spacing rule.

If P_4 is in deletion Case 1, then the situation is similar to Figure 4.24(a)'s deletion Case 1, since P_4.w cannot have a marker. However, there can be one problem if a marker is on node f.

Problem 2: If node f, in Figure 4.24(c), has a marker then, after deletion Case 1, the node will be a new sibling of node gp. The marker on node f will then be too close to P_1's ggp.

If P_4 is in deletion Case 1 $\rightarrow$ 2, no more rotations are done after Case 1 and, therefore, this situation is the same as in deletion Case 1.

If P_4 is in deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4, P_4.w can have a marker after Case 1. Then the situation corresponds to Pictures 7 to 12.1 (pages 125–130) of this theorem. In Pictures 7 to 12.1, node xp corresponds to node ggp of Figure 4.24(c). In Pictures 7, 9.1, and 11, after the rotations, node xp's right child, node a, has a marker. Therefore, Problem 2 happens in these pictures. Pictures 8, 9.2, 12, and 12.1 remain to be considered (Picture 10 is the same as Picture 9 which is divided into Pictures 9.1 and 9.2).

In Pictures 8, 9.2, 12, and 12.1 of this theorem, node a does not have a marker and, thus, Problem 2 does not happen. However, in these pictures, either by proceeding itself, or through P_4's Move-Up rule, P_5 can be moved up to nodes lc and gp. Node lc corresponds to node ggp's parent in Figure 4.24(c); thus, P_5 would be too close to P_1's ggp. This situation is illustrated in Figure 4.25. Figure 4.25(a) is similar to Figure 4.24(c). Figure 4.25(b) shows the result after P_4 performs Case 1. Figure 4.25(c) shows the result after P_4 performs Case 3. And Figure 4.25(d) shows the result after P_4 performs Case 4 and applies the Move-Up rule to allow P_5, in green, to move up. In Figure 4.25(d), P_5's marker on node f is too close to node ggp.

The situation in Figure 4.25(d) is not a problem since P_2 is constrained

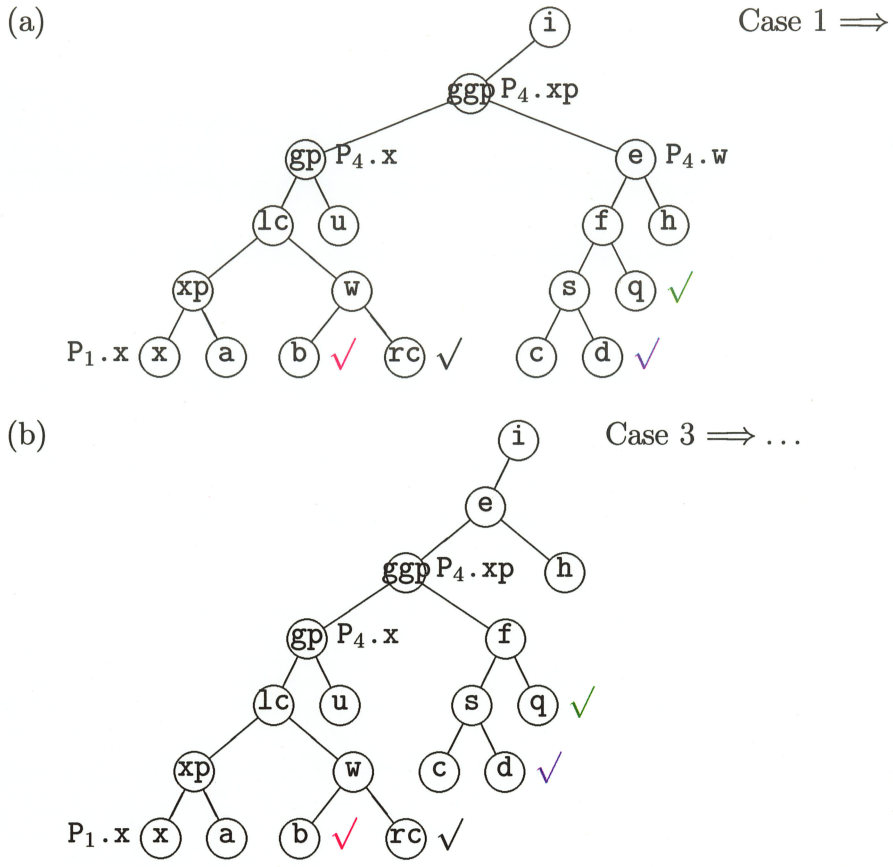


Figure 4.25: P_4 itself does the Move-Up rule. (Picture 1 of 2.)

by the Spacing rule to maintain its distance from processes other than P_1. Hence, P_2 will be able to move up to ggp only after P_5 is first moved away from node ggp's parent.

If P_4 is in deletion Case 1 $\rightarrow$ 4, then the situation corresponds to Pictures 13 and 14 (page 131) of this theorem. In Pictures 13(a) and 14(a), though, node xp would not have a marker. Picture 13(b) has Problem 2; node xp which corresponds to node ggp in Figure 4.24(c) has a marker on its child node.

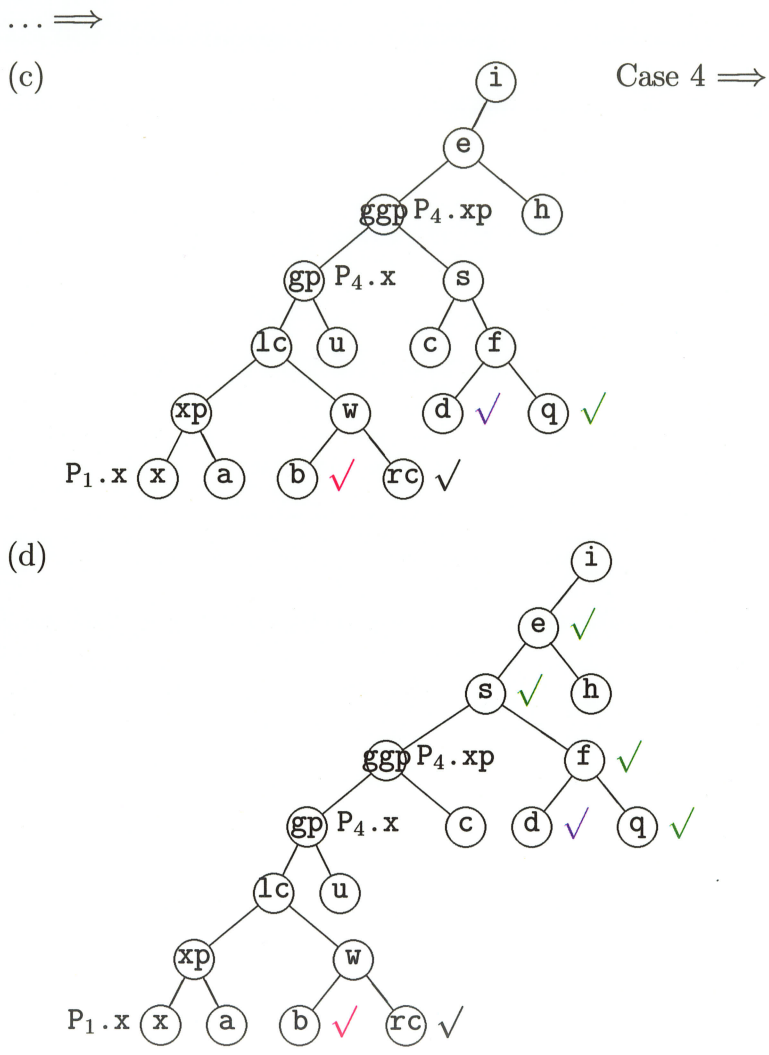


Figure 4.25: P_4 itself does the Move-Up rule. (Picture 2 of 2.)

If P_4 is in deletion Case 3 $\rightarrow$ 4, then the situation is analogous to deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4; see Claim 4.

If P_4 is in deletion Case 4, the situation is analogous to deletion Case 1; see Claim 5.

Figure 4.24(d)'s Transformation Cases of P_4

In Figure 4.24(d), P_4.x is on node **ggp**. Since P_4.x has no markers moved to it (as shown in Pictures 1 to 14 (pages 120–131) in this theorem which cover all the deletion cases), **ggp** will not have a marker moved to it. P_4's local area does not contain any nodes below **ggp** in the subtree rooted at **ggp**. Hence, no markers would be moved too close to **gp** or **ggp** from nodes below **ggp** in the subtree rooted at **ggp**. We have already shown in the discussion of Figure 4.24(c) that a marker moved close to **ggp** from above **ggp** as in Figure 4.25(d) is not a problem. Hence, if P_4.x is on the position or higher than in Figure 4.24(d), no more problems can be created. That is, if P_4.x moved up to its position in Figure 4.24(d) and P_4 is a deletion process, then P_4 has moved far enough up and cannot cause problems to P_1.

We have now gone through all the possible deletion transformation cases of P_4 and found two problems. Before discussing the solutions that resolve these problems, we go through the possible locations of P_4 assuming, this time, that it is an insertion process.

Since no markers are present in the local area of insertion cases as shown in Pictures 15 and 16 (pages 133 and 134) of this theorem, insertion cases will

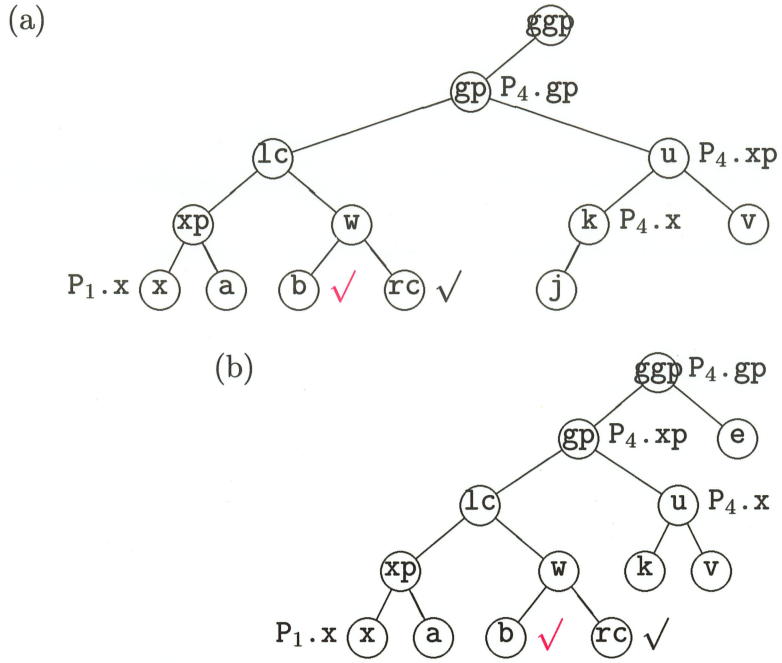


Figure 4.26: Four possible locations of $P_4.x$. P_4 is an insertion process. (Picture 1 of 2.)

not have the same problems as in deletion cases. (In Claim 8, we discussed that a marker can be added to P_2's local area. However, P_4 is not a process being moved up through the Move-Up rule, whereas P_2 is in Claim 8. Hence, the local area of an insertion process, P_4, will not have a marker in its local area.) However, we identify the cases where a flag on a node inside P_4's local area needs to be transferred to P_1 due to rotations.

We consider four possible locations of $P_4.x$ as shown in Figure 4.26. We will go through insertion Case 2 $\rightarrow$ 3 and Case 3 (the insertion cases that involve rotations) for each of the four pictures. If $P_4.x$ is located in any other place, P_4 would be far enough away from P_1 and cannot affect nodes

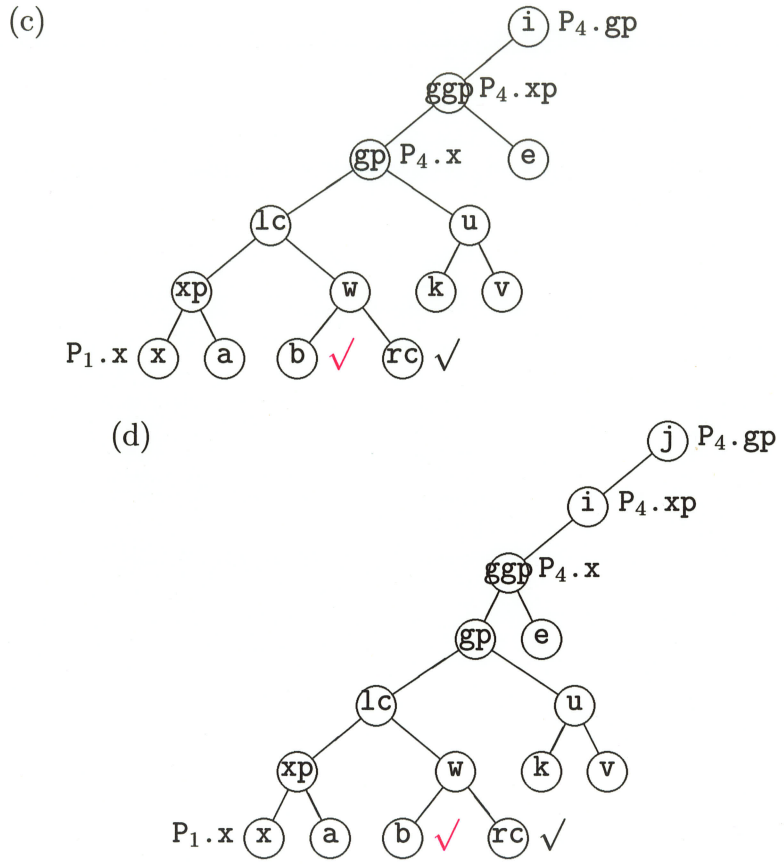


Figure 4.26: Four possible locations of $P_4.x$. P_4 is an insertion process. (Picture 2 of 2.)

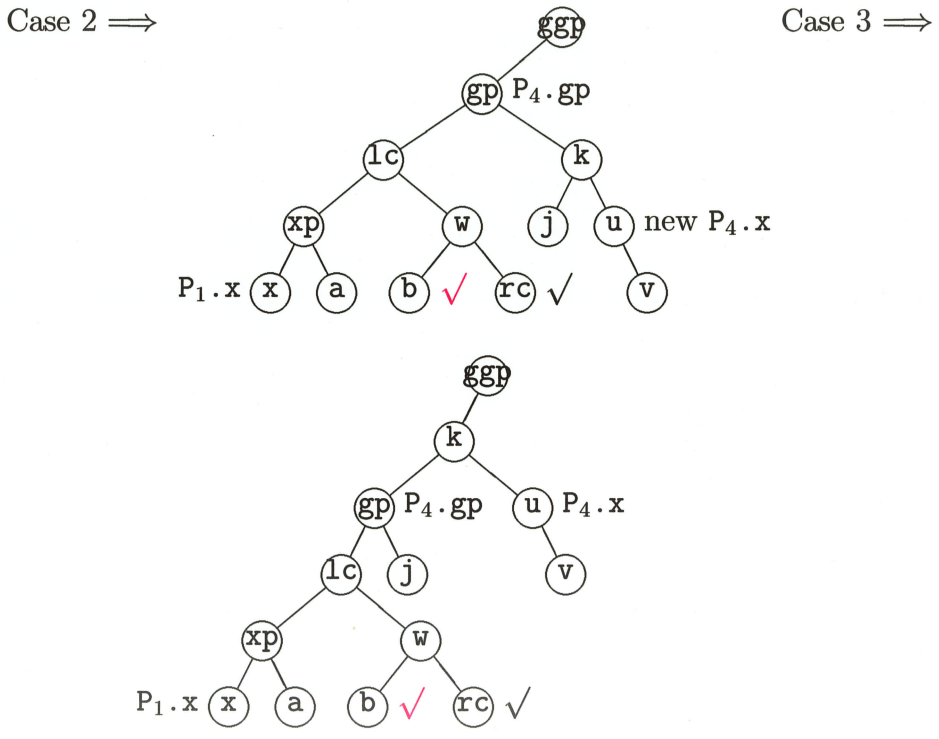
marked by P_1, as will be discussed when we consider Figure 4.26(a) and Figure 4.26(d).

Figure 4.26(a)'s Transformation Cases of P_4

In Figure 4.26(a), P_4.x is on node gp's grandchild.

If P_4 is in insertion Case 2 $\rightarrow$ 3, then the situation would be drawn as follows:

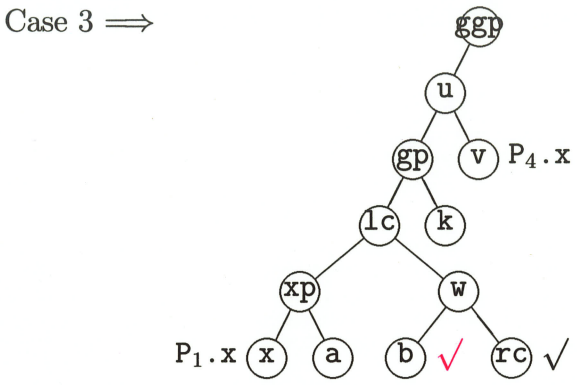
(from Figure 4.26(a))



The above picture shows that, after Case 2, node u gets rotated away from node gp and, after Case 3, node k becomes gp's parent. Also, k's left child, node j, becomes gp's new right child. Hence, P_4 must transfer the flags of nodes k and j to P_1 when it exits. P_1 must also know that it has a new ggp

and that gp has a new right child. This can be done by P_1 checking if gp's parent is still the old ggp, and checking if gp's right child is still node u, respectively.

If P_4 performs insertion Case 3, P_4.x must be on node v in Figure 4.26(a), and the result after Case 3 would look as follows:



The above picture shows that after Case 3, node u becomes the new ggp of P_1, and node k becomes gp's new right child. Again, P_1 must be aware of these facts.

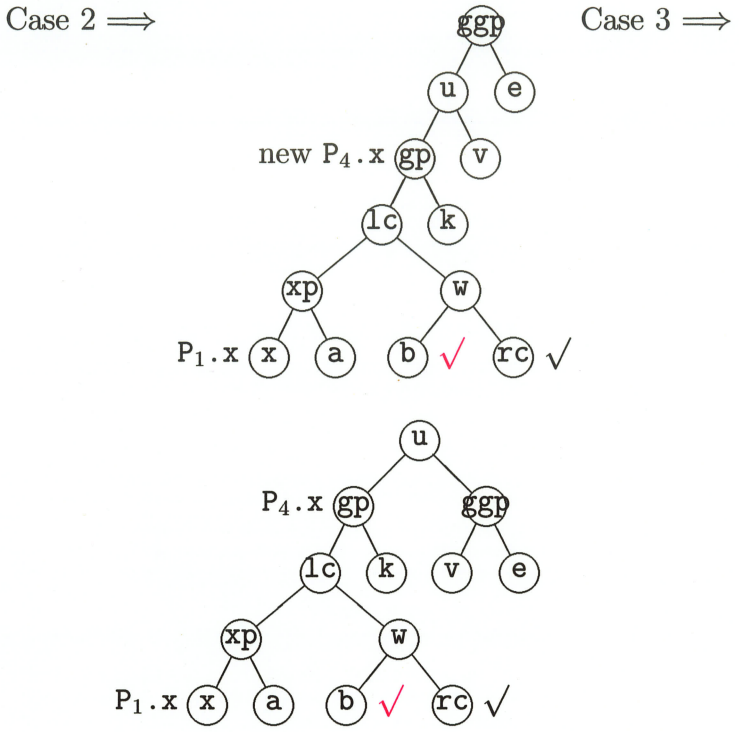
If P_4.x is located a level lower than its position in Figure 4.26(a), then Case 3's rotation would rotate node u away from node gp. If P_4.x is further below, then node u and the nodes above it would not be rotated and, thus, we do not consider such cases.

Figure 4.26(b)'s Transformation Cases of P_4

In Figure 4.26(b), P_4.x is on node gp's child.

If P_4 is in insertion Case 2 → 3, the situation can be drawn as follows:

(From Figure 4.26(b))



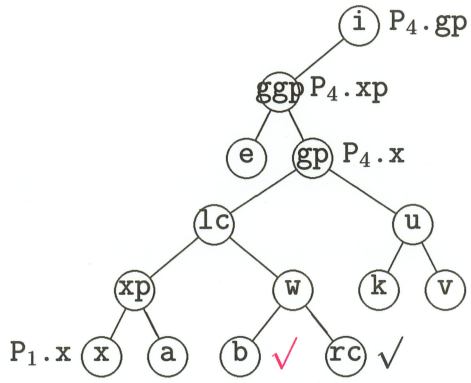
From the above picture, we can see that node u becomes the new ggp of P_1 after Case 2's rotation. Also, node gp has a new right child.

P_4 cannot be in Case 3 in Figure 4.26(b) since P_4.x cannot be on node lc. So we do not have to consider the case.

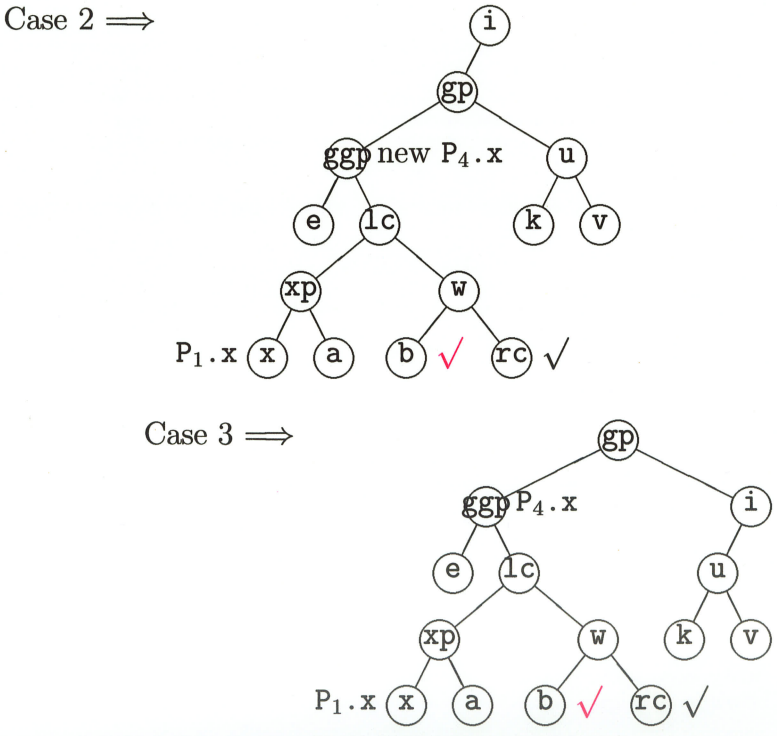
Figure 4.26(c)'s Transformation Cases of P_4

In Figure 4.26(c), P_4.x is on node gp.

If P_4 is in insertion Case 2 → 3, then node gp must be a right child, and the situation would be drawn as follows:



The following picture shows the result after P_4 performs insertion Cases 2 and 3 on the above picture:



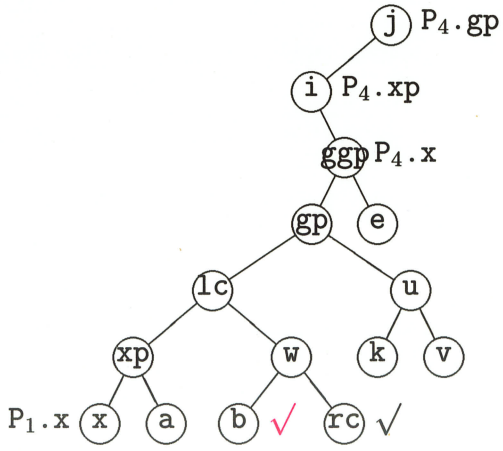
The result shows that node gp and ggp switch places, and node u gets rotated away from node gp. Again, P_1 needs to be aware of these changes.

If P_4 is in insertion Case 3, then the situation would start as in Figure 4.26(c). After the rotation of P_4.gp, node i's parent, which does not have any marker by the Spacing rule, would be the new parent of node ggp. Hence, the situation would not bring any marker too close to ggp from immediately above ggp. Also, node i, which cannot have a marker because it is P_4.gp (see Claim 2 on page 96), becomes the new right child of ggp. Hence, no markers would be too close to ggp from immediately below ggp.

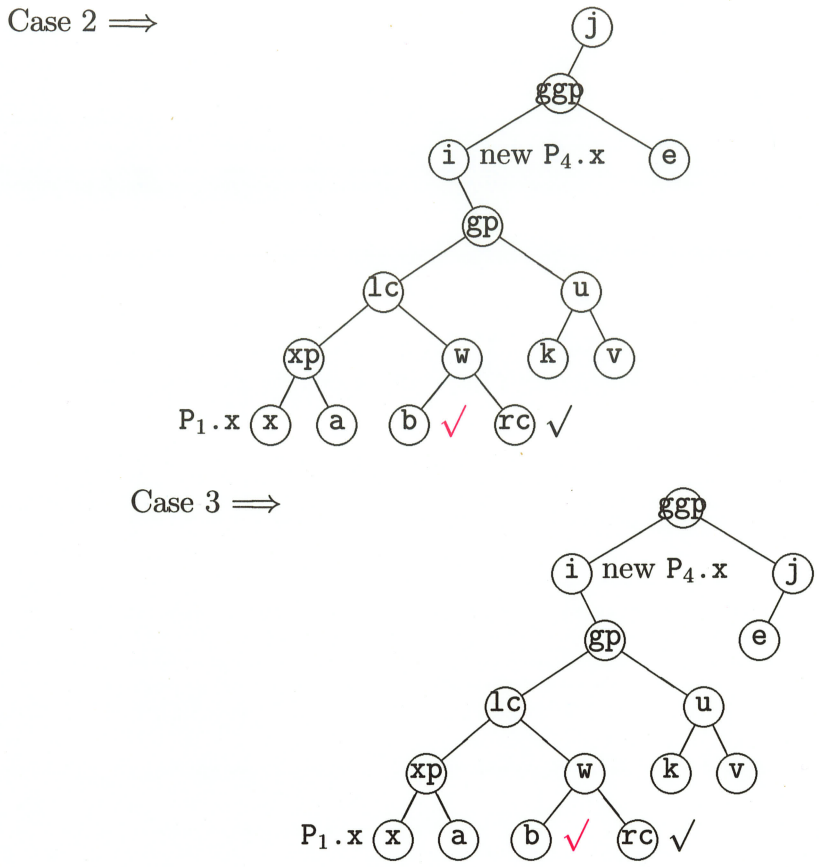
Figure 4.26(d)'s Transformation Cases of P_4

In Figure 4.26(d), P_4.x is on node ggp.

If P_4 is in insertion Case 2, then node ggp would be a right child as the following picture shows:



The following pictures show the results when P_4 performs insertion Cases 2 and 3 to the above picture:



The result after insertion Case 3 in the above picture shows that P_1 has a new ggp, node i. Hence, P_4 must transfer node i's flag to P_1 when P_4 exits.

If P_4 performs insertion Case 3 on Figure 4.26(d), a rotation is done on node j, and no nodes on or below ggp get rotated away from the nodes marked by P_1. Hence, no transferring of flags is needed.

If P_4 is located further above its location than in Figure 4.26(d), then P_4.x would be, at lowest, on node i. The rotations will be done on, at lowest, node j and cannot rotate any nodes on or below node ggp away from the nodes marked by P_1. In other words, if P_4.x moved up to P_1's ggp's parent and P_4 is an insertion process, then P_4 has moved far enough away

from P_1 and cannot cause any problems. Thus, we do not consider such cases.

We have now gone through P_4's insertion transformation cases and found the situations described in the previous paragraphs. These cases are not problems, and we just need to ensure that P_1 and P_4 are aware of these situations and keep track of the flags that they hold and release.

We have now considered all the transformation cases of P_4, and we have found two problem situations named Problem 1 and Problem 2. We now present solutions to these problems.

Proof of Part (b) of Claim 9

In Problem 1 (page 154), P_4's Move-Up rule can bring up a process P_5's marker on gp or its right child which is too close to where P_2's markers must be moved to.

Solution to Problem 1: To resolve Problem 1, P_1 applies the Go-First rule again with P_5 after P_4 exits. Hence, after P_4 exits, P_1 will check its gp and its right child again to see if any marker is on those nodes. If there are any markers, P_1 must apply the Go-First rule again.

We mentioned earlier that P_4 can rotate P_1's gp's child. We also mentioned that, in such a case, P_4 is required to transfer the flag on the node that became the new child of gp to P_1 when P_4 exits. This requirement is necessary because P_1 must check this node to apply the Go-First rule again.

The process P_5 has been moved up through P_4's Move-Up rule. Therefore, the corresponding insertion pictures (Pictures 15 and 16 on pages 133 and 134) of this theorem for P_5 now can have a marker present; see Claim 8 on page 136. In Figure 4.24(a) on page 150, a marker of another process (after P_5 is allowed to move up and P_4 exits), say P_6, can be on node k's left child (but no higher because the marker must have at least three levels of distance from node gp), for example, and that node can become node gp's right child if P_5 is an insertion process performing insertion Cases 2 and 3 where P_5.x was on node k before the rotations. If such a situation happens, P_1 must apply the solution to Problem 1 again with P_6 after P_5 exits. In all the other pictures of Figure 4.24, P_5 would be sufficiently far enough away from P_6 to cause any concerns. For example, a left rotation can be done on node gp if P_5.x is on node u, but the rotation cannot pull the marker on node k's child too close to node gp.

If P_5 in the previous paragraph was a deletion process, then the results of transformation cases that we went through in Figure 4.24 do not change because we can simply substitute P_5 into P_4's place in the pictures. In Figure 4.24(a), node k still would not have a marker because P_6's marker must have at least three levels of distance from node gp. In all the other pictures of Figure 4.24, P_5 would be further away from P_6 and cannot cause concerns. Therefore, by requiring P_1 to check for Problem 1 after the process P_5 involved in Problem 1 exits and, again, after P_6 (pulled up by P_5's rotations or P_5's Move-Up rule) exits and by applying the solution to Problem 1, the situation discussed in the previous paragraph and here is resolved and no further problems can be created.

In Problem 2 (page 159), P_4's rotation can bring a marker to node ggp's right child which is too close to where P_2's markers must be moved to.

Solution to Problem 2: To resolve Problem 2, before P_4 exits, P_4 must check to see if P_4.x is on P_1's gp and to see if P_4.x's sibling has a marker. If both conditions are true, P_4 allows the process on P_4.x's sibling to proceed first before P_4 exits. If P_4 is in the situation of Picture 9.1 (page 127) of this theorem, where P_4.x's sibling, node a in Picture 9.1, has a marker, but the process on a cannot proceed because of the marker on node w, P_4.xp's sibling. If this situation happens, we require P_4 to release all its flags except P_4.x, P_4.xp, and P_4.w. Then the process on P_4.xp's sibling will be able to proceed first. Figure 4.27 illustrates how the process on P_4.xp's sibling proceeds first. In Figure 4.27, the green process, P_5, moved two levels up, from its position where its highest marker was on P_4.xp's sibling, node e.

Now, we consider the possible locations of the two processes involved in Problem 2, to identify the cases where nodes such as P_1's gp or ggp get rotated and to ensure there are no more problems. Figure 4.27 shows the two processes (P_5 in green and P_6 in purple) involved in Problem 2. We first go through the possible locations of P_5 since it will proceed first according to Problem 2's solution.

We consider P_5.x's possible locations using Figure 4.28. If P_5.x is in any other place (below or above node e) than in Figure 4.28, then P_5's rotations would not move any of the nodes on or below ggp and cannot cause any

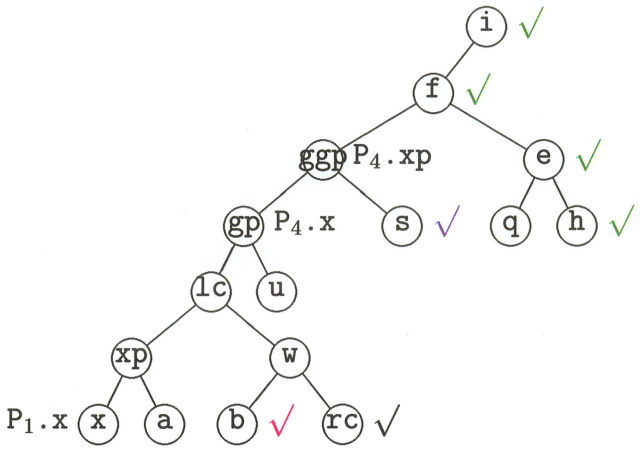


Figure 4.27: Two processes involved in Problem 2.

problems, as will be discussed when we go through the transformation cases of P_5.

Here, we assume that P_4 lets P_5 use the nodes inside P_4's local area.

If P_5 is in deletion Case 1, as Figure 4.29 shows, P_4.x has a new sibling. If P_5 exits at this point, then P_5 will transfer the flag of node f to P_4 before exiting so that P_4 can check for Problem 2. When P_4 does so, it will find that there is no marker on the node, and will be able to exit.

If P_5 is in deletion Case $1 \rightarrow 2$, the situation is the same as in the previous paragraph.

If P_5 is in deletion Case $1 \rightarrow 3 \rightarrow 4$, since P_5.w is the highest place P_6 can be after Case 1, the situation corresponds to Pictures 7 to 12.1 (pages 125–130) of this theorem. Node ggp in Figure 4.29 corresponds to node gp in Pictures 7 to 12.1 (substituting P_5 into P_1's place in the pictures of this theorem), and we notice that node gp's child in Pictures 7 to 12.1 can have

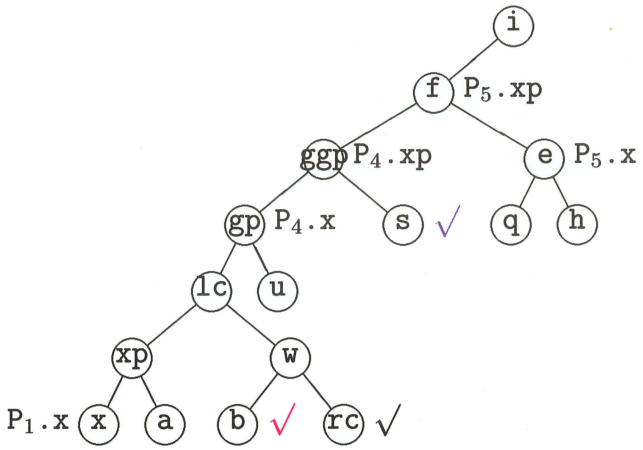


Figure 4.28: Possible locations of P_5.

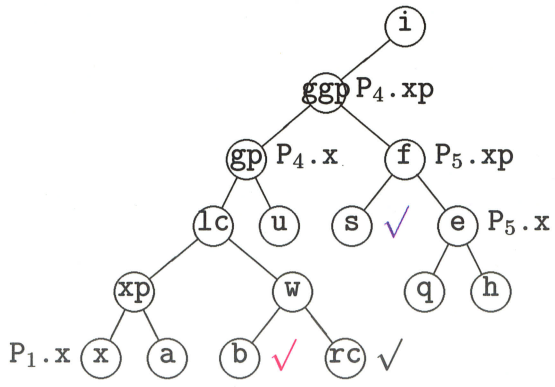


Figure 4.29: P_5 performs deletion Case 1 to Figure 4.28.

a marker after the rotations. Hence, a marker can be on node ggp's child in Figure 4.29. Therefore, P_4 then needs to apply the solution to Problem 2 again with the process on the ggp's child node after the rotations.

Similarly, in Pictures 9.2 and 12.1 (pages 127 and 130), P_5 itself may apply the Move-Up rule, which will bring a process below P_5 up to node ggp in Figure 4.29. If such a situation happens, again, P_4 then needs to apply the solution to Problem 2 with the process brought up by P_5.

If P_5 is in deletion Case 1 $\rightarrow$ 4, the situation corresponds to Pictures 13 and 14 (page 131), with the highest marker being a level lower than in the pictures since P_5.xp cannot have a marker after Case 1. In Pictures 13 and 14, the node gp corresponds to the node ggp in Figure 4.29. Picture 14 shows that ggp's child in Figure 4.29 can have a marker. Again, P_4 then needs to apply the solution to Problem 2 with the process rotated to ggp's child node by P_5.

If P_5 is in deletion Case 3 $\rightarrow$ 4 or in Case 4, the situations are analogous to previous cases; see Claims 4 and 5. Additionally, if P_5 performs Cases 3 and 4 on Figure 4.28, then as in Case 1, P_4.x will have a new sibling, the flag of which needs to be transferred to P_4 by P_5.

If P_5.x is in any other locations than its position in Figure 4.28, assuming it is a deletion process, P_5's local area would not include any of the nodes on or below nodes ggp so that P_5's rotations would not move the nodes.

If P_5 in Figure 4.28 is an insertion process, then the rotations of P_5 would not rotate ggp or any nodes below since ggp is outside P_5's local area. If P_5.x is located a level lower, then node ggp can be the uncle of P_5.x. However, rotations are done in x's parent and grandparent so that ggp and its children

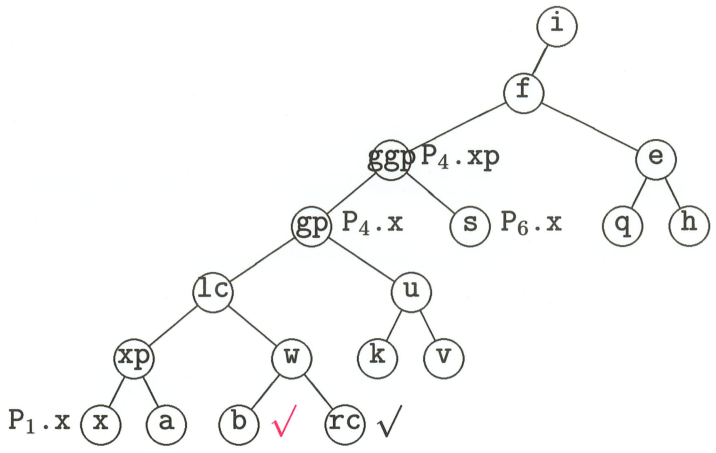


Figure 4.30: Possible locations of P_6.

will not be affected by the rotations. Any other locations are covered by the discussion of Figure 4.26.

We now move on to consider the possible locations of P_6 in Figure 4.27. We consider Figure 4.30 for the possible locations of P_6.x.

Here, we assume that P_1 or P_4 lets P_6 to use the nodes that it holds.

In Figure 4.30, nodes k or v could have a marker, but not on both due to the Spacing rule.

If P_6 did deletion Case 1, Figure 4.31 shows the result. In Figure 4.31, we see that P_1 has a new ggp, node f. Hence, P_6 needs to transfer the flag on node f to P_1. P_6 and P_4 can both exit without any problems because no markers are moved close to node gp or ggp by P_6's rotation.

If P_6 does deletion Case 2 after performing deletion Case 1, then the

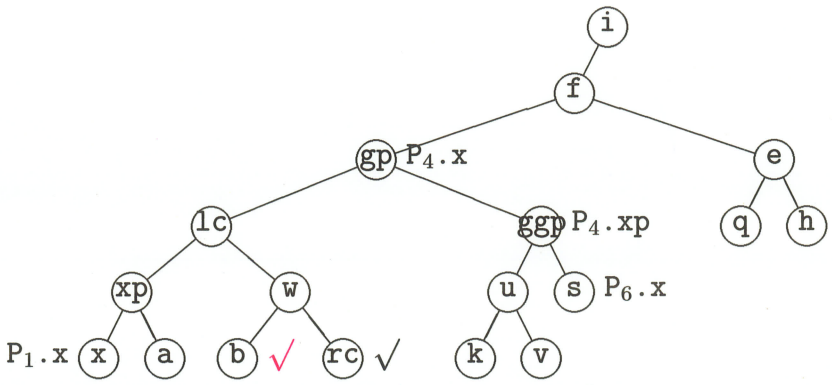


Figure 4.31: P_6 performs deletion Case 1 to Figure 4.30.

situation is the same as in the previous paragraph since no more rotations are done after Case 1.

If P_6 is in deletion Case 1 $\rightarrow$ 3 $\rightarrow$ 4, then the situation corresponds to Pictures 11 to 12.1 (pages 128–130) of this theorem, since after Case 1, P_6.w, which is node u in Figure 4.31, does not have a marker. We see that P_6 can apply the Move-Up rule in Picture 12.1 and bring a process below it up. If such a situation happens, the solution to Problem 1 applies, and P_1 must apply the Go-First rule with the process allowed to move up by P_6 after P_4 and P_6 exit. Also, node gp will have a new right child, node v, and P_6 must transfer the flag of that node to P_1 before exiting.

If P_6 is in deletion Case 1 $\rightarrow$ 4, then the situation corresponds to Pictures 13 and 14 (page 131) of this theorem. Since node u of Figure 4.31 is P_6.w after Case 1 and does not have a marker, the highest node, in Pictures 13 and 14, that can have a marker is lc or rc, leaving at least a single space between the node that has the marker (on lc or rc) and node gp, which corresponds to node gp in Figure 4.31. Therefore, after the transformation,

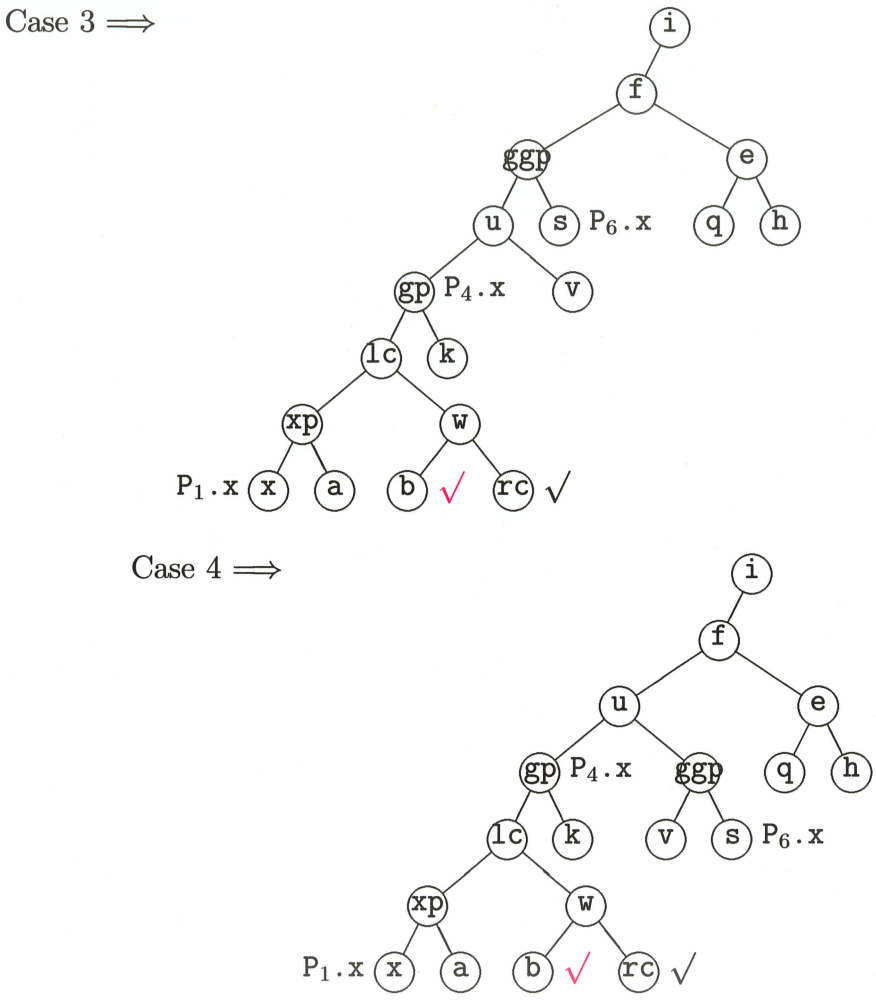


Figure 4.32: P_6 performs Cases 3 and 4 to Figure 4.30.

both P_4 and P_6 can exit safely.

If P_6 is in deletion Case 3 $\rightarrow$ 4, the situation is as shown in Figure 4.32. In Figure 4.32, node u becomes P_1's new ggp. Also, node k becomes node gp's right child. Again, P_6 needs to transfer the flag of node k to P_1 before P_6 exits. After P_4 and P_6 exit, if k has a marker, P_1 can apply the Go-First rule with the marker on k.

If P_6 is in deletion Case 4, the situation is analogous to deletion Case 1 because Claim 5 also holds here.

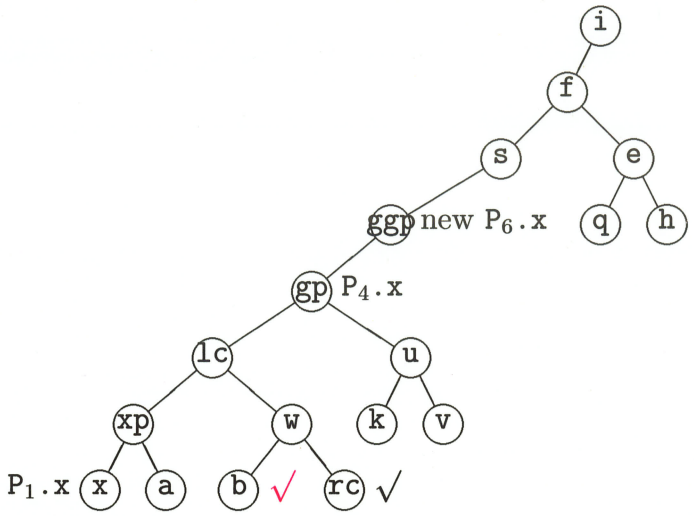
If P_6.x is above than its position in Figure 4.30 then, assuming it is a deletion process, P_6.x would be, at the lowest, on ggp, and no markers will be moved to ggp by P_6's rotations. Hence, there will be no problems. That is, if P_6.x moved up to node ggp and P_6 is a deletion process, then P_6 has moved far enough up so that no problems can occur by its rotations.

If P_6.x is located below than the position in Figure 4.30, then P_6 can apply the move up rule to allow a process beneath P_6 to move up too close to ggp. Then Problem 2 applies. Therefore, P_4 must check for Problem 2 again after P_6 exits. Also, if P_4.x has a new sibling because of P_6's rotations, P_6 must transfer the flag of that node to P_4 before exiting.

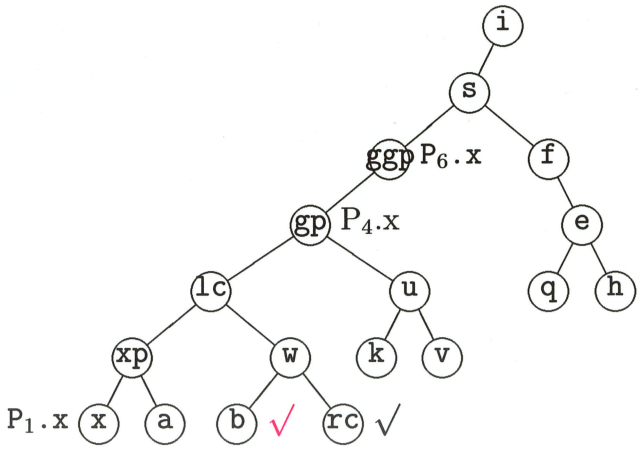
Now let us assume that P_6 is an insertion process. If P_6 is in insertion Case 2 $\rightarrow$ 3, then the following situation happens:

(from Figure 4.30)

Case 2 $\Longrightarrow$

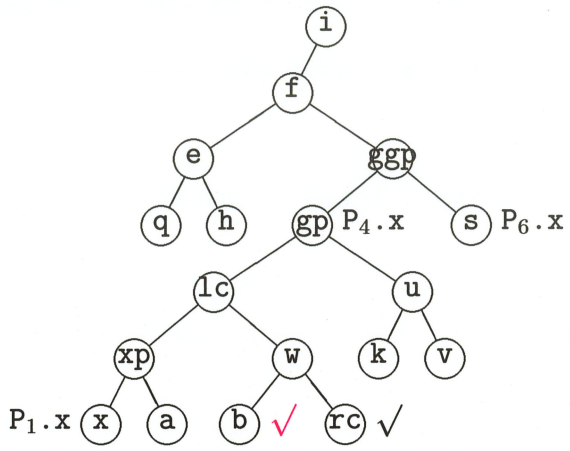


Case 3 $\Longrightarrow$

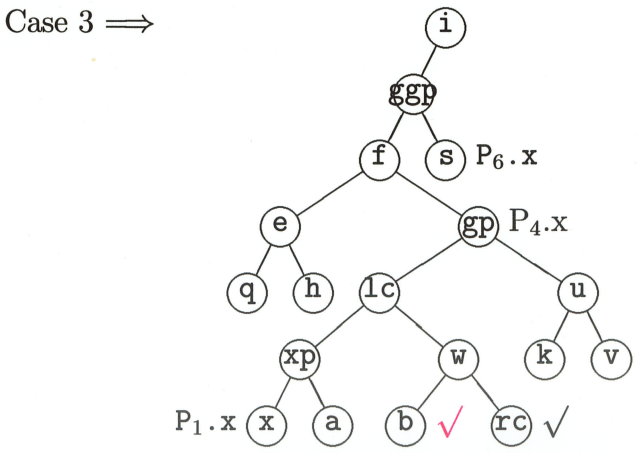


The result after the rotations of Cases 2 and 3 shows that no nodes below P_1's gp get rotated away from P_1's gp. Also, nodes gp and ggp are still in the same positions after the rotations. Hence, all nodes that P_1 holds are still in the same positions. Further, there were no markers moved too close to gp or ggp by P_6's rotations, and there are no problems.

In Figure 4.30, if P_6 is in insertion Case 3, instead of node ggp being node f's left child, node ggp must be node f's right child as the following picture shows:



The following picture shows what happens when P_6 performs insertion Case 3 on the above picture:



As the result in the above picture shows, node f becomes the new ggp for P_1. P_6, thus, needs to transfer the flag of node f to P_1. There were no markers moved too close to gp or ggp by P_6's rotation, and there are no problems.

If P_6.x is located one additional level higher than its position in Figure 4.30, and if P_6 performs insertion Case 2, then the situation just shown in the above picture can happen. That is, if ggp is a right child and its parent is a left child as in the above picture, since the rotation in Case 2 is on the new x, a left rotation on node f also happens. Thus, the situation is the same as in the previous paragraph. If P_6.x is located further above, P_6 would be far enough away from P_1, so it cannot create any problem situations. In other words, if P_6.x moved up to P_1's ggp's parent and P_6 is an insertion process, then P_6 has moved far enough away from P_1.

If P_6.x is located one level lower than its position in Figure 4.30, then a left rotation on ggp or a right rotation on node s can occur. However, these rotations would not rotate any nodes marked by P_1 away from P_1. Further, no markers would be moved too close to node gp or ggp by P_6's rotation, and so there are no problems. If P_6.x is located further below, a rotation can be done, at highest, on node s and cannot rotate any nodes marked by P_1. Moreover, no markers would be moved too close to node gp or ggp, and so there are no problems.

We mentioned, at the end of going through P_6's deletion cases, that if P_6.x is located lower than the position of Figure 4.30 on page 177, P_6 can apply the Move-Up rule to allow a process beneath P_6 to move up too close to ggp and that, in such a case, the solution to Problem 2 is reapplied after P_6 exits. If such a situation did occur and suppose an insertion process, say P_7, is allowed to move up by P_6, then the corresponding pictures of this theorem (Pictures 15 and 16) for P_7 can now have a marker present; see Claim 8. In Figure 4.30 then, another process's marker can be on node s's grandchild

(but no higher because the marker must have at least three levels of distance from node ggp), for instance. Then, if P_7.x is on node s's left child and performs insertion Cases 2 and 3, s's grandchild node that has a marker can be ggp's new right child. If such a situation happens, P_4 must apply the solution to Problem 2 again after P_7 exits.

If P_7 in the previous paragraph was a deletion process, the results that we went through for Figure 4.30 (now assuming P_7 is allowed to move up and P_6 exited, and so substituting P_7 into P_6's place) do not change since P_7 was located in the right subtree of node ggp in Figure 4.30 before moving up, and nothing is changed in the left subtree of node ggp (the marker that was too close to P_7 which caused P_6 to apply the Move-Up rule is in the right subtree of node ggp). Moreover, since we have already required P_4 to check for Problem 2 again after a deletion process P_6 exits, if P_7 is a deletion process, the situation is the same as with P_6. Hence, by requiring P_4 to check for Problem 2 after the process, P_6, involved in Problem 2 exits and, again, after P_7 (pulled up by P_6's rotations or P_6's Move-Up rule) exits regardless of the processes being insertion or deletion processes, the situation discussed in the previous paragraph and here is resolved and no further problems can be created.

We have now gone through all the possible transformation cases of P_5 and P_6 in the application of the solution of Problem 2, identified the situations that require P_1 to update the flags of the nodes that get rotated, and found no more problems. We therefore showed that the solutions resolve all the problems found in part (a) of this claim's proof.

Finally, we note that P_4, in Figure 4.24, or the process allowed to move up by P_4 in the solution of Problem 2 may itself be in a situation where it has to apply the Go-First rule after performing deletion Case 4. If such a situation happens, P_1 has to wait until P_4 is done. That is, P_1 indirectly has to wait until the process above P_4 has moved far enough up and away from P_4 or has exited after performing a rotation.

Even if the situation described in the previous paragraph propagates upwards, at the top of all the processes involved in such a situation, there has to be a process that has no other process above it. Since the highest-positioned process is not blocked by any other process, deadlock of lower-positioned processes waiting for higher-positioned processes and vice versa will not occur. □ End of Claim 9.

After the Go-First rule is performed, because P_4 may have changed the locations of markers in P_1's local area, P_1 must check again to see if the Move-Up rule is to be applied. The Move-Up rule may not have to be applied after doing the Go-First-rule processing.

Now we move on to consider the Exit rule. The Exit rule defines how P_1 detects that it has finished performing the Move-Up rule and, thus, can exit.

The Exit Rule: When applying the Move-Up rule, if P_1 detects that the process that is allowed to move up has placed its marker on gp or ggp, P_1 can exit. Also, P_1 can exit if it determines that the process allowed to move up has itself exited.

Conclusion of the Proof of Theorem 1:

Before the theorem, we focused on the two deletion situations that produced double markers — Pictures 6 and 10 of Lemma 2 — and produced rules that avoid creation of such double markers. We introduced the Spacing rule (page 114) to check three nodes — the node `t` in which the new marker is to be placed, the parent of `t`, and the sibling of `t` — before placing a marker so that, initially, we have a space between the markers of any two processes.

In the proof of the theorem, we went through all the pictures of Lemma 2 applying the Spacing rule. There were three cases where the space could be removed after rotations. We introduced four additional rules that resolve the three problem cases. Hence, processes either have a space between their markers or their highest markers are kept far enough apart to avoid creating double markers. □

4.6.4 Summary of the Rules

We added five rules. The first rule is the Spacing rule (page 114) and is introduced immediately after the proof of Lemma 2.

The Spacing rule's aim is to leave a space between markers of processes. The Spacing rule requires a process, when placing a new marker, to check three nodes — the node `t` where the new marker is to be placed, the parent of `t`, and the sibling of `t`. If all three nodes have no flags and markers, then the process places the new marker.

The second to the fifth rules are the Move-Up rule (page 135), the Insert-

Jump-Ahead rule (page 145), the Go-First rule (page 148), and the Exit rule (page 185), and are introduced in the proof of Theorem 1.

Using the Move-Up rule, a rotating process P_1 allows one of the two processes in its local area in the problem situations described to move up to gp or ggp, thereby preventing creation of double markers by any other process performing rotations. The process allowed to move up is not constrained by the Spacing rule to maintain its distance from nodes marked by P_1, but is still constrained to maintain its distance from nodes marked by any other processes.

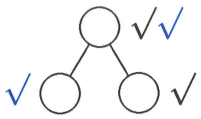
The Insert-Jump-Ahead rule requires P_1 in deletion Case 3 to retain flags of two lowest markers when it releases its markers before a rotation. This way, P_1 will have the flags of gp and ggp, allowing room for an insertion process to be moved up to ggp by the Move-Up rule.

The Go-First rule requires P_1 to let the process on gp or its right child to proceed before applying the Move-Up rule. As in the Move-Up rule, the process allowed to go first is not constrained by the Spacing rule to maintain its distance from nodes marked by P_1, but is still constrained to maintain its distance from nodes marked by any other processes.

The Exit rule defines when P_1 can exit. During the Move-Up rule, if the process P_2 that was allowed to move up has reached gp or ggp, P_1 can exit. If P_2 exits before reaching gp or ggp, then P_1 can also exit.

4.7 Reasons for Not Using More Markers per Node

In Lemma 2 of Section 4.6.2, we concluded that two markers on a node, as in the following picture, are possible: **Picture A:** A double marker; see Picture 10 of Lemma 2 on page 105.



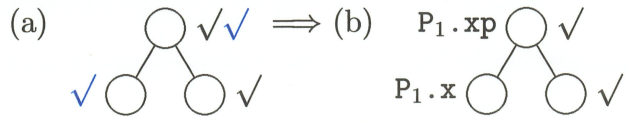
In Section 4.6.3, we added some rules so that our algorithm prevents the double marker situation from happening in the first place.

In this section, we describe why two markers on a node are a problem to be avoided. We assume that none of the five rules described in Section 4.6.3 apply here.

We follow a similar naming convention as used in Lemma 2; refer to page 95. We use processes $\mathtt{P}_1$, $\mathtt{P}_2$, $\mathtt{P}_3$, and $\mathtt{P}_4$. We represent the markers of $\mathtt{P}_1$, $\mathtt{P}_2$, $\mathtt{P}_3$, and $\mathtt{P}_4$ by blue (√), black (√), magenta (√), and cyan (√) check marks, respectively.

If we allow two markers per node as in Picture A above, since we must increase the number of marker fields in a node from one to two, checking the marker fields becomes more complex. For instance, a process must check two marker fields of a node to determine that no marker is present in the node before placing its marker. A process releasing its marker must also check both fields to determine which field contains its marker.

Also, we must increase the number of markers to set and release per process (as opposed to per node). To show the necessity of increasing the markers per process, consider **Picture B:** Reason for five markers per process.



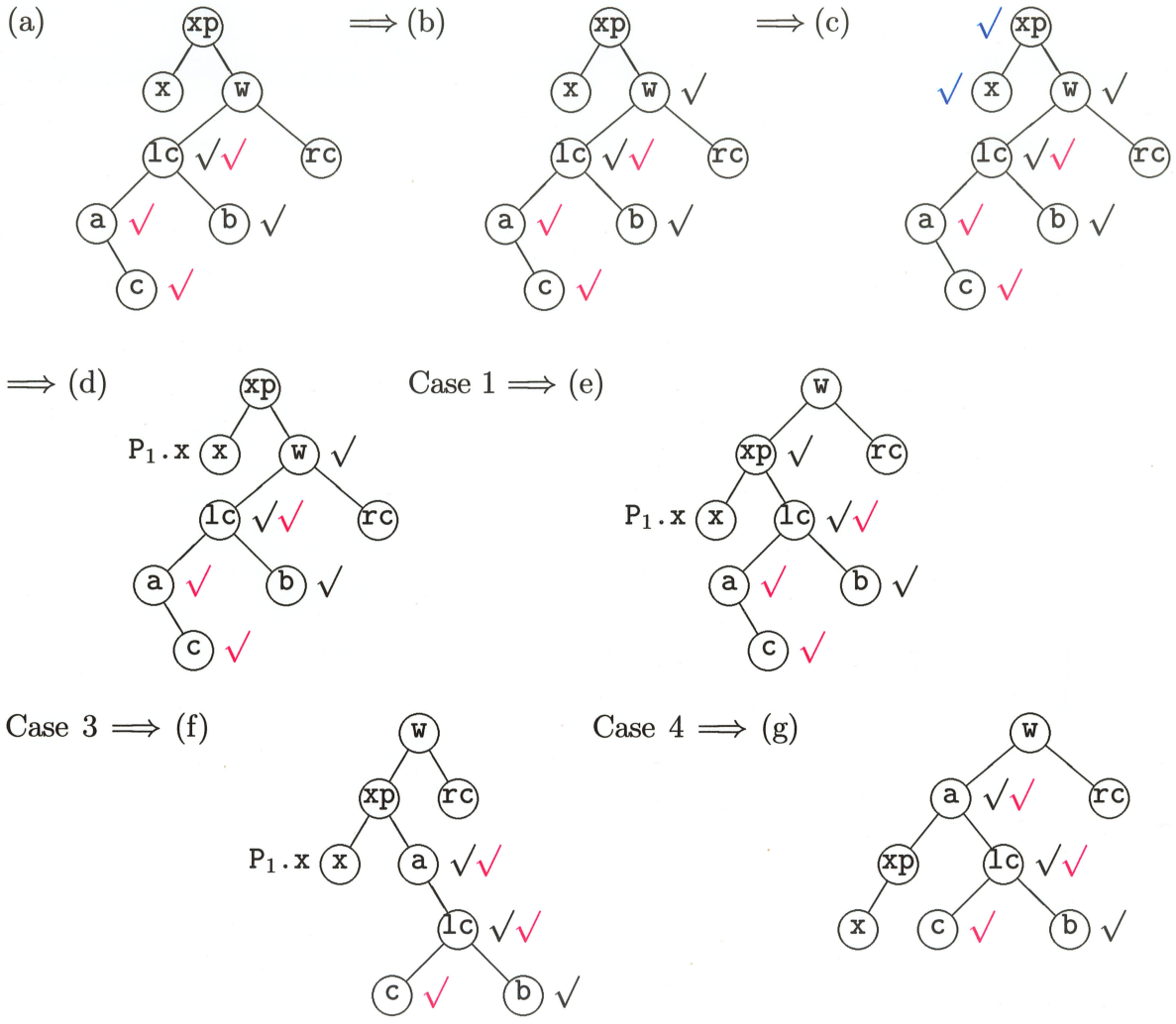
Part (a) of Picture B is the same as Picture A. In part (b), P_1 has moved up so that P_1 is in its first non-repeating case.

When we assumed only a single marker per node to begin with, `xp` of a process in its first non-repeating case could not have any marker; see Claim 2 of Lemma 2 on page 96. In Picture B, we assumed a double marker, and as Picture B(b) shows, the result is that a marker of another process can be on P_1`.xp` when P_1 is in its first non-repeating case. In Picture B(b), P_2 needs five markers to ensure a safe distance of four levels from P_1`.xp`.

The increased number of markers per process increases the distance between processes in general (that is, between processes not involved in the double marker problem) and, therefore, reduces concurrency. This is undesirable.

A bigger problem, however, is that two consecutive double markers by two different processes can occur:

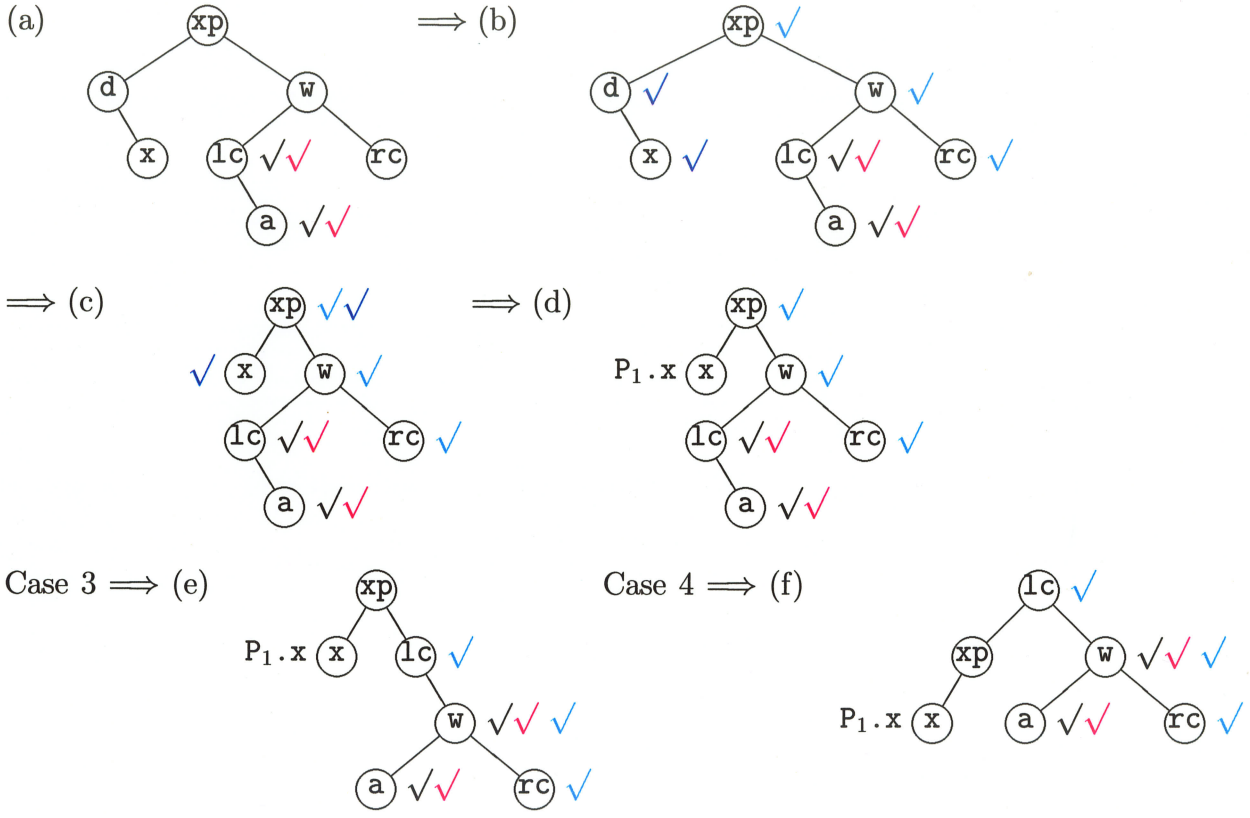
Picture C: Two Consecutive Double Markers.



Picture C(a) corresponds to Picture A; P_2 and P_3's topmost markers are on node `lc`. In part (b), P_2 moved one level up. In part (c), P_1 came up from the subtree rooted at node `x`. In part (d), P_1 moved further up and is about to perform a rotation. Parts (e), (f), and (g) show the results after P_1

performs deletion Cases 1, 3, and 4, respectively. We can see that, in parts (f) and (g), we have two consecutive double markers between P_2 and P_3.

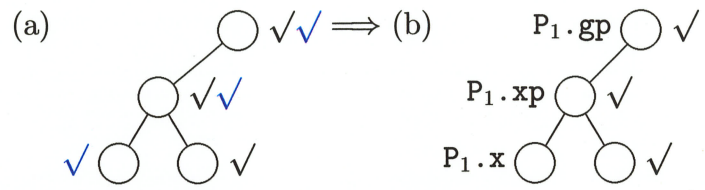
If two consecutive double markers occur, then three markers per node are needed as the following picture shows: **Picture D:** Three Markers in a Node.



Part (a) of Picture D corresponds to Picture C(g); the top two markers of P_2 and P_3 are on two consecutive nodes. In part (b), P_4 advanced before P_2 and P_3; P_4 came up from the subtree rooted at node `rc`. Also, P_1, located in the subtree rooted at node `d`, came up and placed its markers on nodes `x` and `d`. In part (c), the topmost marker of P_1 is joined with the topmost

marker of P_4 as in Picture A. (Node d has been rotated to become xp's parent and is just not shown in the picture.) In part (d), P_1 moved up and is about to perform deletion Case 3. Part (e) shows the result after Case 3 is performed. P_1 then performs deletion Case 4, and the result is shown in part (f). In parts (e) and (f) of the picture, we see that three markers are required on the node labelled w.

Not only must we increase marker fields per node from two to three, but we must also increase the number of intention markers per process from five to six. To see the reason for six markers per process, consider the following picture: **Picture E:** Reason for six markers per process.

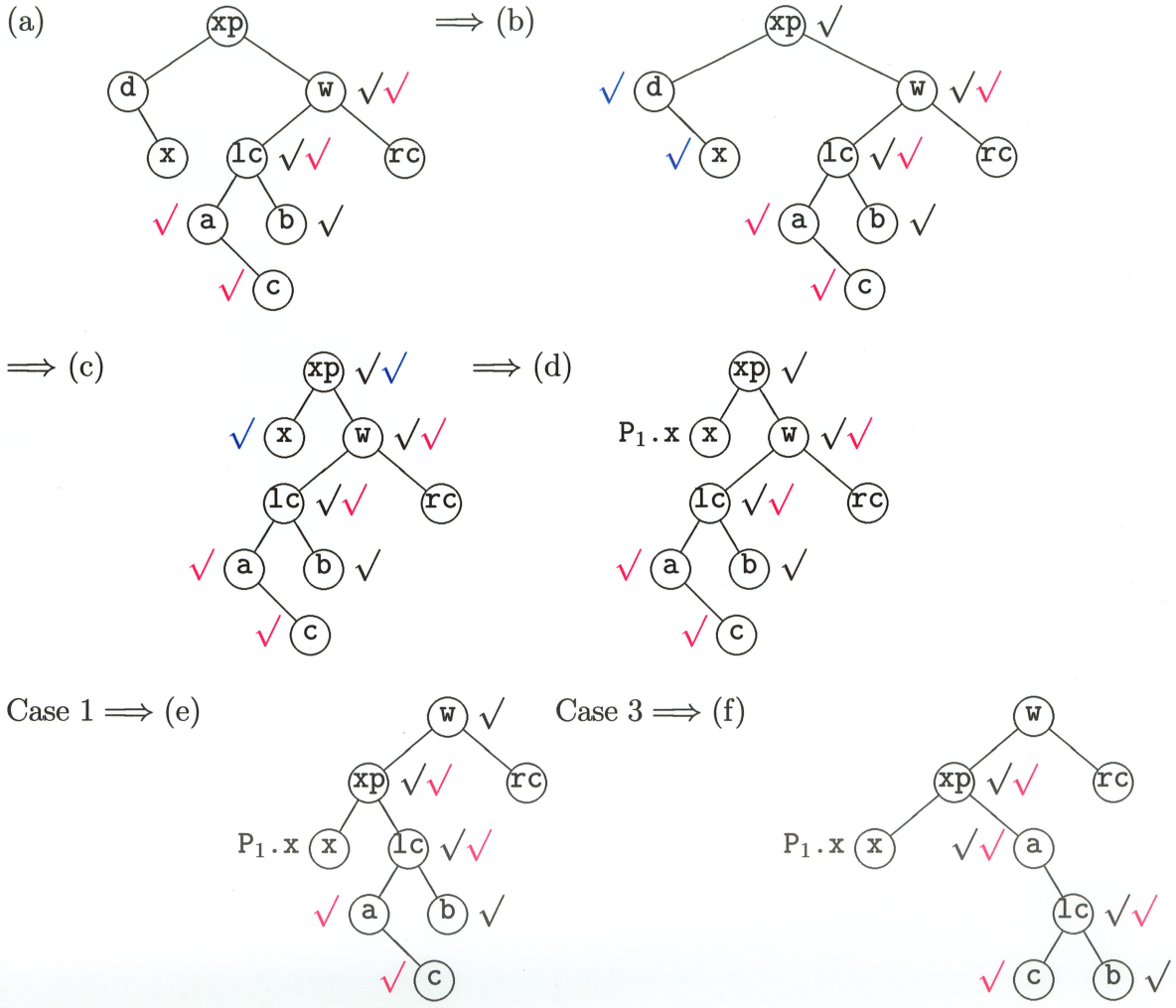


In part (a) of Picture E, P_1 and P_2's top two markers are "joined" to create the two consecutive double marker problem. In part (b), P_1 moved up and is in its first non-repeating case. Since the highest marker of P_2 is on P_1.gp and the second highest is on P_1.xp, P_2 needs four more markers in addition to the two highest markers, six markers in total, to ensure a safe distance of four levels from P_1.xp.

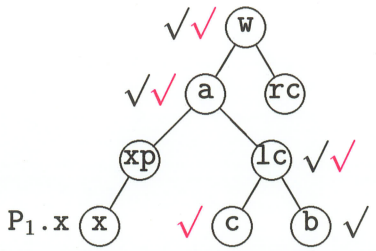
As we have seen, if the two consecutive double marker problem occurs, then we need to increase the number of marker fields per node from two to three; and, therefore, checking the marker fields when setting and releasing them becomes significantly more complex.

Moreover, by increasing the number of markers to set per process, the distance between processes is increased; and, therefore, concurrency is further reduced.

Yet a bigger problem is that three consecutive double markers can now occur as the following example shows: **Picture F:** Three Consecutive Double Markers.



Case 4 $\Longrightarrow$ (g)



Picture F(a) corresponds to Picture C(g); P_2 and P_3's two topmost markers are "joined" to create two consecutive double markers. In part (b), P_2 has moved one level up. Also, P_1, which is located in the subtree rooted at node `d`, came up and placed its markers on `x` and `d`. In part (c), P_1's topmost marker and P_2's topmost marker are "joined" to create a double marker problem as in Picture A. (As in Picture D, node `d` is rotated to become the parent of `xp` but is not shown.) In part (d), P_1 moved further up and is about to perform a rotation. Parts (e), (f), and (g) show the results after P_1 performed deletion Cases 1, 3, and 4, respectively. In parts (f) and (g), we can see that, between P_2 and P_3, we have three consecutive double markers.

We observe several similarities between the picture that shows a sequence of operations to produce two consecutive double markers — Picture C — and the picture that shows a sequence of operations that created three consecutive double markers — Picture F.

The similarities can be used to find a sequence of steps that create four consecutive double markers, five consecutive double markers, and so on. This increasing number of consecutive double markers will increase the number of markers that must be set and released per process and is the primary reason for avoiding more markers per node.

To illustrate this problem, we now describe the similarities between Picture C and Picture F. Then we use the similarities to draw a picture that shows steps that will create four consecutive double markers.

Both Pictures C(a) and F(a) show the case of the most recent highest number of consecutive double markers between $\mathtt{P}_2$ and $\mathtt{P}_3$; one (a double marker) for Picture C(a) and two (two consecutive double markers) for Picture F(a).

In Pictures C(b) and F(b) and Pictures C(c) and F(c), $\mathtt{P}_2$ moves one level up, and $\mathtt{P}_1$ also moves up and comes into the picture. $\mathtt{P}_1$ and $\mathtt{P}_2$ are joined to produce (the most recent highest number of consecutive double markers -1) consecutive double markers. In Picture C(c), this number (in parentheses in the previous sentence) is zero, $(1-1=0)$, and, therefore, $\mathtt{P}_1$ and $\mathtt{P}_2$'s markers are not joined. In Picture F(c), the number is $(2-1=1)$, and, therefore, $\mathtt{P}_1$ and $\mathtt{P}_2$ are "joined" to produce one consecutive double marker (henceforth referred to simply as a "double marker"). A double marker is shown to occur in Lemma 2 and, therefore, $\mathtt{P}_1$ and $\mathtt{P}_2$ can be joined in such a way.

In Pictures C(d) and F(d), $\mathtt{P}_1$ moves further up so that the lowest of the consecutive double markers between $\mathtt{P}_2$ and $\mathtt{P}_3$ is on $\mathtt{P}_1\mathtt{.wlc}$ (node `lc` in the pictures).

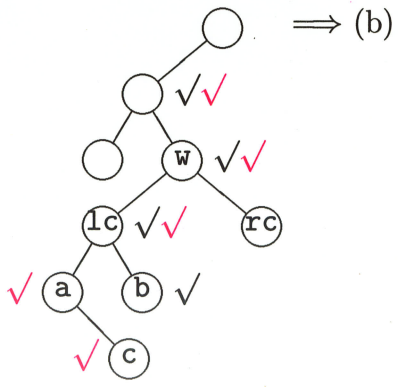
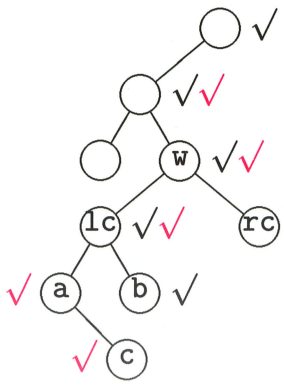
In Pictures C(d) and F(d), $\mathtt{P}_1\mathtt{.wlc}$'s left child, node `a` in the pictures, contains only $\mathtt{P}_3$'s marker. Similarly, the node that contains the highest marker of $\mathtt{P}_2$ has only $\mathtt{P}_2$'s marker. These facts do not change after performing Case 1; see Pictures C(e) and F(e).

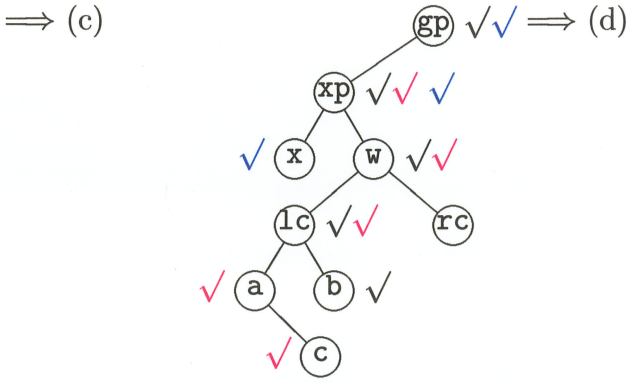
When Case 3 is performed, however, the single marker of $\mathtt{P}_3$ on node `a` is joined with the highest marker of $\mathtt{P}_2$ by marker adjustment. In Picture C(f),

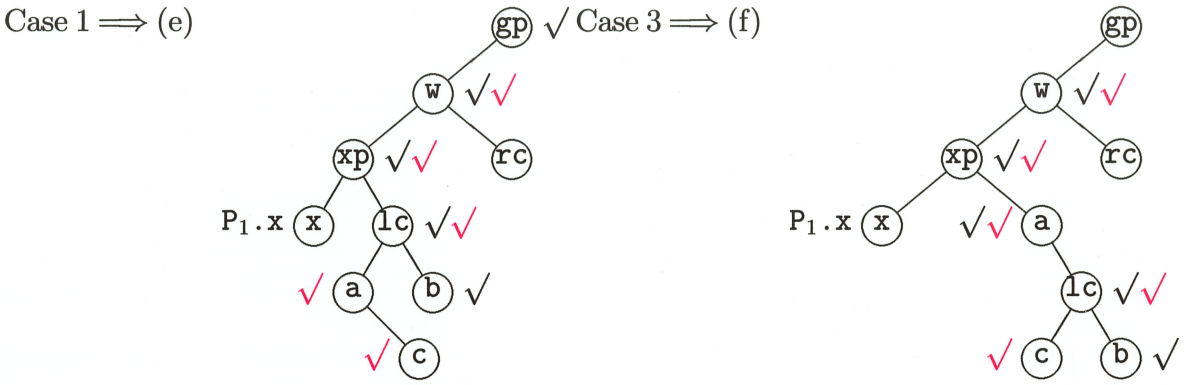
the highest marker of P_2 is moved down to node a, creating one additional double marker between P_2 and P_3. In Picture F(f), the two highest markers of P_2 are each moved one level down; the second highest marker of P_2 is moved down to node a, and the highest to node xp. In this case, node a now has two markers, and a's parent still has two markers, also creating one additional double marker between P_2 and P_3 in total. This entire process can be thought of as the highest marker of P_2 being moved down to a to create one additional double marker.

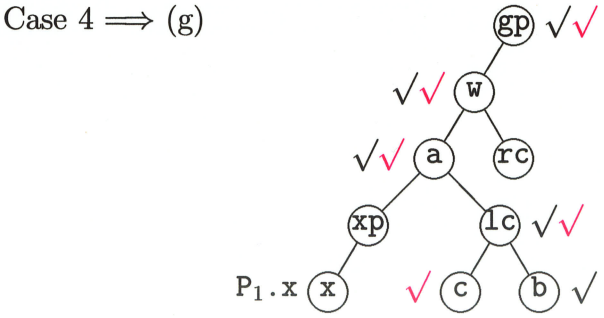
Finally, deletion Case 4 follows deletion Case 3, and Pictures C(g) and F(g) show that the new most recent highest number of consecutive double markers between P_2 and P_3 does not change after deletion Case 4.

We now use the similarities described above to construct a sequence of steps that lead to four consecutive double markers as follows: **Picture G:** Four Consecutive Double Markers.

(a)  (b) 







Picture G(a) has three consecutive double markers — the most recent highest number of consecutive double markers found in Picture F. In part (b), $\mathtt{P}_2$ moved one level up. In part (c), $\mathtt{P}_1$ is joined with $\mathtt{P}_2$ to create $(3-1=2)$ consecutive double markers. (Between part (b) and part (c), first, $\mathtt{P}_1$ and $\mathtt{P}_2$'s highest markers are joined to create a double marker as in Picture A. Then

two consecutive double markers between P_1 and P_2 are created. Detailed pictures showing these steps are omitted because two consecutive double markers between the same two processes have already been shown to occur in Picture C. Some nodes in parts (a) and (b) of Picture G have no labels because these nodes are involved in rotations during the steps just mentioned and change positions in the tree.) In part (d), P_1 moved further up and is about to perform a rotation.

In Picture G(d), notice that the lowest of the three consecutive double markers is on `lc`, as in Pictures C and F. Also, node `a` has a single marker, of P_3, and the parent of the node that contains the highest of the three consecutive double markers — node gp in the picture — has a single marker, of P_2.

After performing deletion Case 1, Picture G(e) shows that we still have a single marker, of P_3, on node `a` and a single marker, of P_2, on node gp. These markers join to add one more double marker after deletion Case 3 is performed; see part (f). Case 4 follows Case 3, and the result in part (g) shows that we have four consecutive double markers between P_2 and P_3.

The same method can be used to find a sequence of steps to create five consecutive double markers, then six consecutive double markers, and so on. Thus, if we do not prevent double markers, we cannot guarantee that we can keep processes far enough apart so that they do not interfere with one another.

Chapter 5

Some Implementation Details of the Algorithm

In this chapter, we give an overview of how we would approach implementing the algorithm. First, we describe a way to implement the additional rules discussed in Theorem 1 of Section 4.6.3 using communication between processes via flags in each node. Then we describe the structure of a node and the shape of the initial empty tree. We then present structure charts of the algorithm as a guide for implementing the algorithm. Finally, we explain the components of the structure charts by grouping the components according to their main tasks.

5.1 Implementing the Additional Rules

The Spacing rule is described in detail just before Theorem 1. The Insert-Jump-Ahead rule of having flags of two markers is straightforward and de-

scribed in Section 4.6.3. Hence, we focus on the other rules — the Move-Up rule, the Go-First rule, and the Exit rule.

These rules may be implemented in many different ways. The implementation would involve interprocess communication, such as communicating a process or task ID (if a process can have multiple tasks at the same time as opposed to multiple processes each having one task), coordinating the sharing flags of nodes, and signaling to allow another process to move up. Here, we introduce a method of interprocess communication and leave further details for future research; see Section 6.1.

To implement the Move-Up and the Go-First rules, we introduce five new fields in a node. We introduce a flag called `okToMoveUp` (with the intended meaning of 'it is okay to move up using repeating cases until the process allowed to move up has moved far enough away from P_1 (in the Go-First rule) or reached gp or ggp, or to proceed with non-repeating cases'). A second field we add is an integer field called `ownerID`, which contains the process ID of the local area's owner. A third field that we add is an integer field that holds the process ID of the process that is allowed to move up, and we call this field, `friendID`. A fourth field that we introduce is a flag called `goFirstDone` (with the intended meaning of 'the Go-First-rule processing is done'). The last field is also a flag and is called `moveUpDone` (with the intended meaning of 'the Move-Up rule processing is now done').

When P_1 (the process that did rotations and is waiting to exit) applies the Move-Up rule or the Go-First rule, the process that is allowed to move up by P_1 is not constrained by the Spacing rule to maintain its distance from the nodes marked by P_1. To allow for this variation in the Spacing rule, the

field ownerID has been added so that a process, say P_2, being allowed to move up by P_1 can now place P_2's marker on a node that has the process ID of P_1 in its ownerID field.

Earlier in Section 4.6, we mentioned that P_1 sometimes needs to request another process to transfer the ownership of a flag of a node held by the other process. For example, when P_1 applies the Go-First rule, the process, say P_4, that is allowed to move up needs to transfer the flag of a node to P_1 if P_4 performed rotations and is about to exit. The ownerID field can be used by P_4 to accomplish this transferring of the flag of a node. P_4 can simply place P_1's process ID on the ownerID field of the node that is needed by P_1 when the node currently has P_4's process ID in that field.

To allow a process to move up in either the Move-Up rule or the Go-First rule, P_1 sets the okToMoveUp field of the node that contains the highest marker of the process being allowed to move up.

To notify P_1 that the Move-Up rule or the Go-First rule processing has been completed, the process allowed to move up sets the moveUpDone or goFirstDone flag of P_1.x to true.

When P_1 must apply the Go-First rule, a process P_4 whose highest marker is on gp or its right child is allowed to move up. P_1 sets the flag okToMoveUp of the node that has the highest marker of P_4 to true, puts the process ID of P_4 on the friendID field of that same node, and sets the flag goFirstDone of node P_1.x to false. If P_4 finds out that P_4's process ID and the friendID match, then P_4 can proceed. When P_4 has moved far enough away from P_1 or is about to exit, P_4 sets goFirstDone of P_1.x to true.

When P_1 finds goFirstDone true (P_1 is checking or "polling" for this)

P_1 sets `goFirstDone` to false and determines if the Move-Up rule is to be applied. This check for the applicability of the Move-Up rule is done again because, as shown in the proof of Claim 9, P_4 could have moved nodes inside the local area of P_1 so that the Move-Up rule is no longer applicable. If the Move-Up rule is applied, P_1 checks to see if the Go-First rule is to be applied again before applying the Move-Up rule. This check is done because, as explained in the proof of Claim 9, after P_4 itself applies the Move-Up rule, another process could come up to place markers where P_4 was.

P_1 does similar processing for the Move-Up rule: P_1 sets `okToMoveUp` on the node that contains the highest marker of P_2 to true and sets the `friendID` to P_2's process ID (assuming that P_2 is the process being allowed to move up). P_1 also sets `moveUpDone` of P_1`.x` to false. If P_2 has moved up to gp or ggp or is about to exit, then P_2 sets `moveUpDone` on P_1`.x` to true.

When P_1 determines that `moveUpDone` is true or that the Move-Up rule is not applicable anymore, then P_1 sets `okToMoveUp` (of the node whose corresponding field was set to true previously) to false and exits.

Interprocess communication requires synchronization. Some interprocess communication mechanisms implicitly use blocking synchronization techniques (e.g., blocking message passing). We do not want to use such techniques. We therefore assume that all interprocess communication is accomplished using shared memory and lock-free primitives. Additionally, we observe that no deadlock is possible among processes using such techniques because the synchronization relationships follow the tree's structure which contains no cycles. Therefore, deadlock is impossible.

```
struct nodeStruct {
  int key;                    /* data */
  int colour;                 /* RED or BLACK */
  struct nodeStruct *p;       /* parent */
  struct nodeStruct *left;    /* left child */
  struct nodeStruct *right;   /* right child */
  boolean flag;               /* flag */
  int marker;                 /* intention marker (process ID) */
  boolean okToMoveUp;         /* rules */
  int ownerID;                /* rules */
  int friendID;               /* rules */
  boolean goFirstDone;        /* rules */
  boolean moveUpDone;         /* rules */
};
```

Figure 5.1: Structure of a node.

5.2 The Structure of a Node

Figure 5.1 shows the contents of a node. A node has a key, a colour (either red or black), a pointer to its parent, a pointer to each of its left and right children, a boolean flag field to allow a process to hold its local area, a marker field for intention markers, and the four fields used to implement the additional rules of Section 4.6.3 as described in Section 5.1.

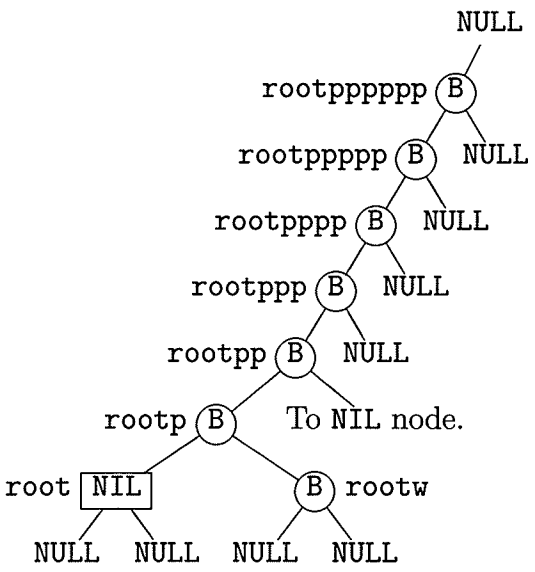


Figure 5.2: Initial tree created by CreateTree().

5.3 Creating an Empty Tree

Figure 5.2 shows a picture of the initial tree. As the picture shows, we must provide six "dummy" nodes above the root. The reason for the dummy nodes is that when an insertion process's `x` is at the root, the process has the flags of `x`'s parent, grandparent, and uncle plus four intention markers above the grandparent. (A deletion process will not reach the topmost node, only up to `rootppppp`.)

We name the dummy nodes `rootp`, `rootpp`, ..., `rootppppppp`, from bottom up. The topmost node, `rootppppppp`'s parent, is set to `NULL` and so are the right children of the dummy nodes except `rootpp`'s right child for the case when an insertion process's `x` is at the root; the `rootpp`'s right child is

the uncle and is set to `NIL`. `NIL` is the sentinel node; its colour is black, and its other fields contain arbitrary values. `NULL` is the null pointer in languages like C.

Lastly, we add `rootw`, the sibling of the root for the case when a deletion process's `x` is at the root or when an insertion process's `x` is at the root's child.

5.4 Overall Structure of the Algorithm

In this section, we present two structure charts that show function call trees for the insert and delete algorithms, organized from left to right. Figure 5.3 is for insertion, and Figure 5.4 is for deletion.

We used Ma [26]'s insertion algorithm with the addition of intention markers. From the functions taken from Ma's insertion algorithm, we changed some of the function names and divided some of the tasks in the functions into other functions, to be consistent with the names used in our deletion algorithm.

We now describe these functions. We group the functions by the four main tasks a process performs, each task is described in one of the next four sections. We ordered the sections so that they show how a process performs an insertion or a deletion in chronological order.

5.4.1 Using DCAS to Locate Insert and Delete Points

As in Ma [26], we use DCAS (see Section 3.2.4) to obtain the flags of nodes. An example of obtaining a flag is shown by the following code:

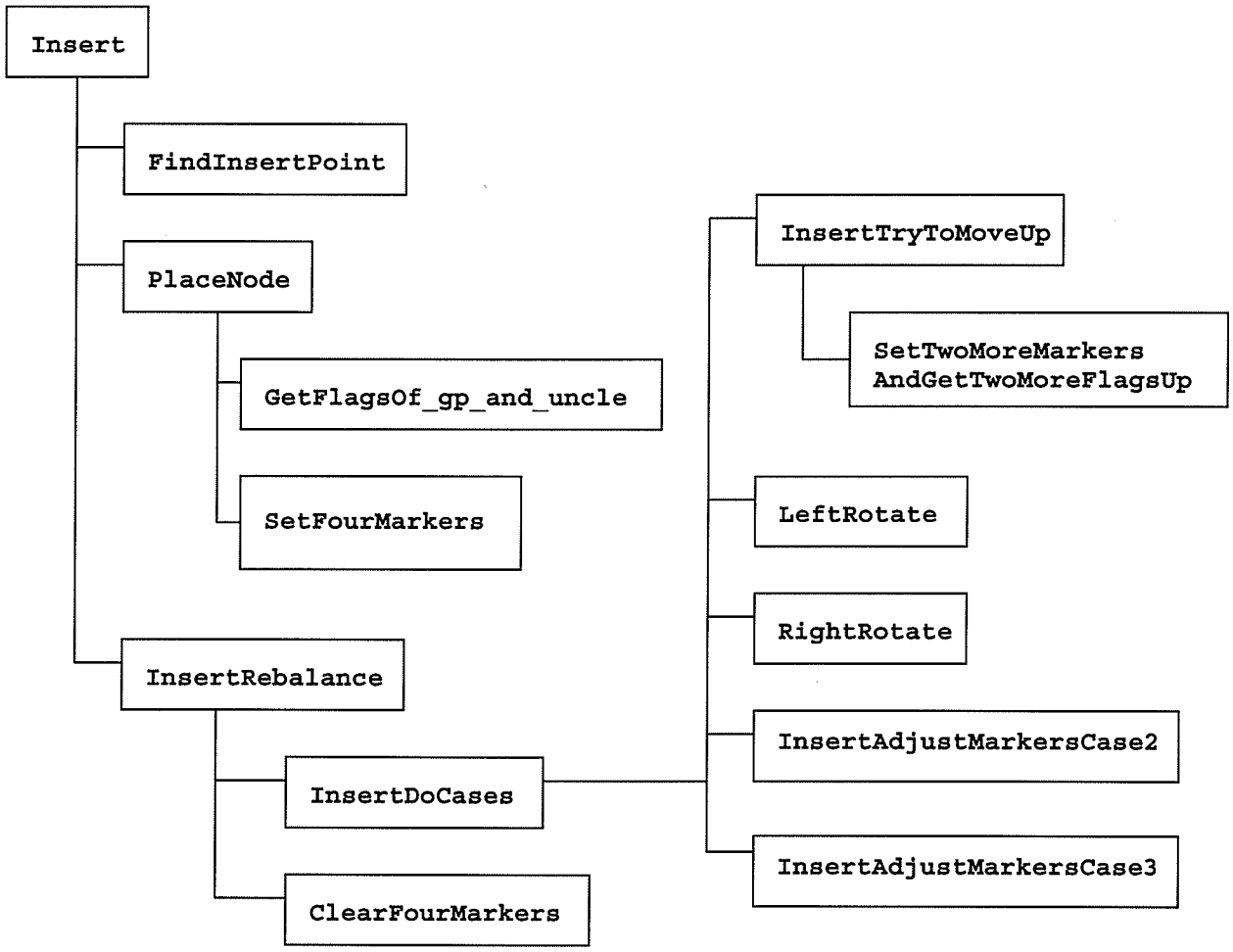


Figure 5.3: Insert algorithm structure chart.

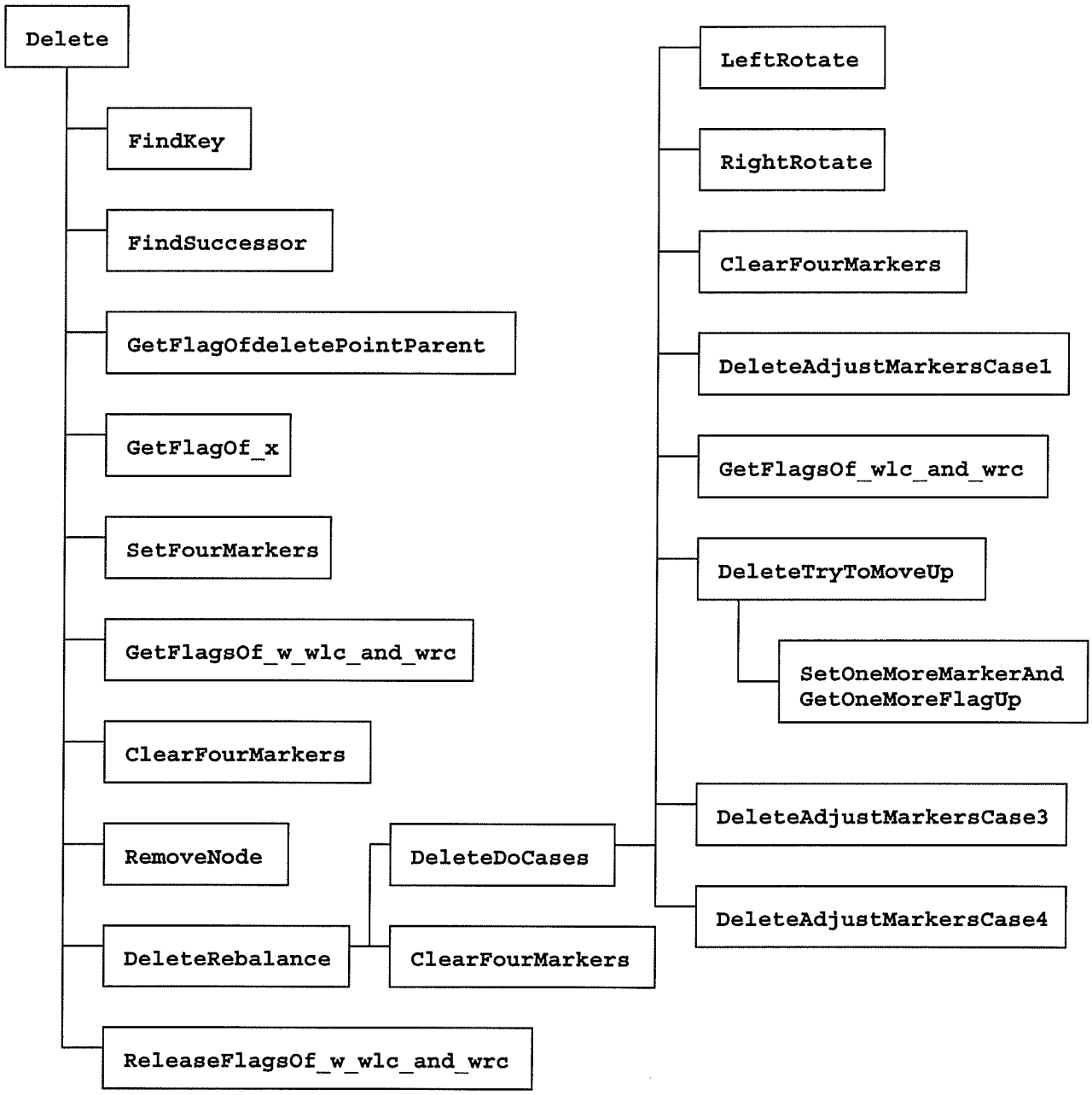


Figure 5.4: Delete algorithm structure chart.

```
1. copyOfNodeND = nodeND;  // Take a copy of a shared memory
                           // into a local variable.
2. result = DCAS(nodeND->flag, nodeND, FALSE, copyOfNodeND,
                 TRUE, nodeND);
```

In the above code, a copy of `nodeND` is first made in Line 1. Then, in Line 2, DCAS sets `nodeND`'s flag only if the flag is not already set and the node has not been changed since the copy was made at Line 1.

Finding the Insert Point

The method of using DCAS to obtain flags is used by an insertion process to locate the insert point, the node under which a new node is to be inserted. For illustration purposes, we assume that the tree is not empty and the root has two children.

Just like in an ordinary binary search tree insertion, a process finds an internal node with two leaf children under which the new node will be inserted.

An insertion process first obtains the flag of the root. Then we search to find the insert point. To travel down the tree, the process obtains the flag of the appropriate child of the root (depending on whether the new key is less or greater than the root's key), moves down to the child node, and then releases the root.

If the child of the root is not an internal node with two leaf children, then the process travels one more level down, obtaining the flag of the appropriate child and then releasing the flag of the root's child.

The insertion process repeatedly moves to the appropriate child in this

way until it is at the insert point. Along the way down, if a DCAS fails to obtain a flag, the process backs off by releasing the flags it holds and starts the insertion process again by competing for the flag of the root.

The function named `FindInsertPoint` (of Figure 5.3) does the search to locate the insert point.

Finding the Delete Point

A deleting process starts its search for the node that contains the given key at the root node and travels down the tree until it finds the node.

`FindKey` and `FindSuccessor` (of Figure 5.4) use the same technique used in `FindInsertPoint` to find the delete point. The node that contains the key to be deleted (the key holder) may not be the node physically deleted (the delete point). If the key holder has two children, the deletion process finds the inorder successor of the key holder, copies the key of the successor into the key holder, and then deletes the successor (the successor is the delete point in this case). As the names suggest, `FindKey` locates the key holder, and `FindSuccessor` locates the successor of the key holder.

5.4.2 Establishing the Initial Local Area

Before a process can physically place a new node into the tree or delete a node from the tree, the process must establish the initial local area (see Section 4.3) and set four intention markers. If a process fails to establish the local area or to set the intention markers, the process backs off by releasing all the flags it holds and the markers it has set so far and begins the insertion or deletion from the very beginning of calling `FindInsertPoint` or `FindKey`.

Establishing the Local Area of an Insertion Process

Upon successful completion of `FindInsertPoint`, we hold the flags of the new node, `x`, and the insert point, the parent of `x`. Then the next step in establishing the local area is to obtain the flags of `x`'s grandparent and uncle. `GetFlagsOf_gp_and_uncle` is responsible for obtaining these flags. Again, to obtain the flags, the process uses DCAS. If the DCAS fails, the function returns `FAIL`, and the process backs off and restarts the insertion.

If the insertion process gets the flags of `gp` and `uncle`, then the next step is to set four markers. To set a marker, a process uses the Spacing rule introduced in Section 4.6.3. The function `SetFourMarkers` uses DCAS to obtain the flags of the nodes where the markers are to be placed, places the markers of the process (the process ID), and releases the flags of the marked nodes. If a process fails to obtain flags using DCAS or if there is already a marker placed by another process, the function returns `FAIL`, and the process backs off and restarts.

Establishing the Local Area of a Deletion Process

After successfully finding the delete point, the deletion process has the flag of the node to be deleted. To establish the local area, the process needs to obtain the flags of the delete point's parent (which will be `xp`), `x`, which is the node that replaces the delete point, the sibling `w`, and the sibling's children `wlc` and `wrc`. The functions `GetFlagOfdeletePointParent`, `GetFlagOf_x`, and `GetFlagsOf_w_wlc_and_wrc` are responsible for obtaining these flags, using DCAS, respectively.

After obtaining the flag of the delete point's parent, the deletion process

calls `SetFourMarkers` before obtaining the flag of `w`. If a process fails to set four markers, the process does not have to set the flags of `w` and its children. However, if a process fails to set `w` or its children, the process must release the four markers set by calling `ClearFourMarkers`, as part of the backing off process.

5.4.3 Physical Placement and Removal of Nodes

After the local area is established and the intention markers are set, an insertion process physically places the new node, and a deletion process removes the delete point. The functions `PlaceNode` and `RemoveNode` are responsible for these tasks. Since a process holds the flags of the nodes in the local area, the process can safely do these operations of placing or removing the node as in the sequential insertion and deletion algorithm described in Section 2.4.

5.4.4 Rebalancing the Tree

After physically inserting a node or deleting a node, a process calls `InsertRebalance` or `DeleteRebalance` to fix up the red-black properties.

Insertion Rebalancing

`InsertRebalance`'s two main tasks are as follows:

- Repeatedly call `InsertDoCases`, which performs the case processings.
- Release the markers and flags held after the rebalancing is done.

The repetition of calls to `InsertDoCases` corresponds to the while loop of Figure 2.6, the sequential insertion algorithm.

Case 1 is the only case in which an insertion process moves up. This is where we call `InsertTryToMoveUp`. `InsertTryToMoveUp` first calls the function `SetTwoMoreMarkersAndGetTwoMoreFlagsUp`. This point is where we would need to apply the additional rules introduced in Theorem 1 of Section 4.6.3; also, see Section 5.1. After setting two more markers and flags, the process moves up and releases the old flags and markers, in `InsertTryToMoveUp`.

In Cases 2 and 3, `LeftRotate` and `RightRotate` are used for doing rotations. Before a rotation is done, a process releases four markers, using `ClearFourMarkers`, to increase the concurrency as described in Section 4.4. However, for Case 3's rotation at `x`'s grandparent, a process doing Case 2 retains the flag of the node that contains its lowest marker. If a process is in Case 3 without Case 2, the process does the operations of releasing markers and retaining the flag of the lowest marker in Case 3.

Also, before the rotations are done, a process checks to see if one of the patterns of Figures 4.18, 4.19, 4.20, and 4.21 applies. If the process finds that the markers of other processes in its local area match a pattern, it adjusts the markers as shown in the figures. The functions `InsertAdjustMarkersCase2` and `InsertAdjustMarkersCase3` are responsible for this detecting of the patterns and adjusting markers.

Deletion Rebalancing

`DeleteRebalance` carries out two main tasks. They are as follows:

- Repeatedly call `DeleteDoCases` that performs the four deletion cases.
- Release all the flags and markers held when the rebalancing is done.

The iteration of calls to `DeleteDoCases` corresponds to the while loop of Figure 2.12, the sequential deletion algorithm.

A process in Case 1, 3, or 4, which requires rotations, uses the functions `LeftRotate` and `RightRotate` to do rotations, and `ClearFourMarkers` for releasing markers, as in insertions.

After Case 1's rotation, the rotating process does not release the flag of `gp`'s right child. The process keeps the flag so that it can apply the Go-First rule of Section 4.6.3. Also, when releasing markers before a rotation to increase concurrency, the rotating process keeps the flags of the nodes that contained the two lowest markers for the Insert-Jump-Ahead rule.

After Case 4 is done, the rotating process checks to see if the Move-Up rule of Section 4.6.3 applies. For this checking, the process in Case 4 must have performed Case 3 prior to Case 4. Also, when the process was in Case 3, the process must not have released the flags of the old `rc` and `lc`'s right child after Case 3's rotation, as mentioned in Section 4.6.3.

Processes in Case 2 are also affected by the additional rules of Theorem 1. A process in Case 2 calls `DeleteTryToMoveUp` to try to move up the tree. In `DeleteTryToMoveUp`, the process first calls `SetOneMoreMarkerAndGetOneMoreFlagUp` to set one more marker and acquire one more flag to be added to its new local area. Then the process releases a set of flags in its old local area. If a process in Case 4 allows a process in Case 2 to move up for the Move-Up rule, the process in Case 2 should be able to move up and set the necessary marker needed for moving up, as mentioned in Section 5.1.

As in insertions, a process doing rotations must adjust the markers of other processes if it detects the patterns of Figures 4.18, 4.19, 4.20, and 4.21.

The functions `DeleteAdjustMarkersCase1`, `DeleteAdjustMarkersCase3`, and `DeleteAdjustMarkersCase4` are responsible for this task.

In Ma's insertion algorithm [26], `LeftRotate` and `RightRotate` require the use of 3-CAS. Our algorithm requires only DCAS; see Section 4.1.1 and 4.1.2. This is an improvement because DCAS is widely available in hardware while 3-CAS is not.

Chapter 6

Conclusion

In this thesis, we presented a high-concurrency lock-free algorithm for red-black trees. We first discussed red-black trees by explaining the history and advantages of red-black trees and an existing sequential algorithm. We then discussed related work including lock-free synchronization methods and lock-free concurrent data structures. Then we moved on to discuss lock-free red-black tree algorithms.

We based our algorithm on an existing sequential red-black tree algorithm and an existing efficient concurrent lock-free red-black tree insertion algorithm that we adapted. We added the idea of intention markers to ensure a safe distance between processes so that any insertion and deletion processes can proceed. We then presented a proof of correctness of our algorithm design. In the proof, we added five necessary rules that became part of the algorithm. Finally, we presented some implementation details of the algorithm.

6.1 Future Work

We conclude by discussing some future work. Our algorithm is not fully implemented, and this version is an early version which does not include interprocess communication mechanisms. Thus, some details of interprocess communication remain to take concrete form. Three of these details are discussed, and they are related to the rules added in the proof of correctness. Finally, we discuss an efficient search method.

We described in Section 5.1 that to notify P_1 that the Move-Up rule or the Go-First rule is done, the process allowed to move up sets the appropriate flag of P_1.x. It would be convenient if P_1 could send the address of P_1.x to the process allowed to move up. This could be easily done using a call back routine. The details of how this interprocess communication is implemented are left for future research.

If P_1 applies the Go-First rule, then the following processes must be able to use the nodes inside P_1 and/or P_4's local areas:

- A process, P_4, allowed to move up by the Go-First rule.
- A process allowed to move up by P_4 from P_1's ggp's right child in the solution of Problem 2.
- A process moving above P_1's ggp in the solution of Problem 2.

It would be convenient if P_1 and/or P_4 could let these other processes know that P_1 and/or P_4 are involved in one of the above situations and can share the flags in their local areas. The details of how this interprocess communication is implemented are left for future research.

The transferring of the ownership of a flag from the processes listed above to the process, P_1, performing the Go-First rule is sometimes necessary. For instance, the process, P_4, moved up by the Go-First rule, may need to transfer the ownership of a flag of a node to P_1. This transfer can be done by interprocess communication as follows. P_1 requests that P_4 transfers a particular node's flag from P_4 to P_1. P_4 can then simply grant the request. Some details of this transfer, using the `ownerID` field, are discussed in Section 5.1. The details of how this interprocess communication is implemented are left for future research.

A search algorithm is very similar to the steps involved in locating the insertion point in the insertion algorithm or those for locating the key to be deleted in the deletion algorithm. A more efficient search algorithm may be devised using fewer flags or using only the intention markers, and is left for future research.

Appendix A

Patterns

Figures A.1 to A.8 show all the pictures of marker movements during a rotation. In each of the figures, we perform a left rotation at `x`. The check marks represent markers of a process located beneath the process doing the rotation. Right rotations are symmetric.

We assume some minimum separation between the rotating process and the process below whose markers we see in the pictures.

In insertion Case 2, the rotation is done on `xp`. By Claim 2 of Lemma 2, `xp` does not have a marker before a rotation of Case 2 is done.

In insertion Case 3, a rotation is done on `gp`. By Claim 2 of the lemma, `gp` does not have a marker.

Hence, in insertions, a rotating process's rotation point is guaranteed to have no markers, assuring some distance between the rotating process and a process located beneath the rotating process. The highest node a marker beneath the rotating process can be located on before a rotation is performed is the rotation point's child.

In deletion Case 1, the rotating process's rotation point is `xp`. By Claim 2 of Lemma 2, we have no marker on `xp` before rotation. Hence, similar to the rotations during insertions, the highest node a marker beneath a rotating process can be located on before the rotation of deletion Case 1 is done is the rotation point's child.

In deletion Case 3, the rotation is done on `w`. Before Case 3, a marker beneath the rotating process could be pulled up at most to `xp` by Case 1. Hence, the highest a marker of a process beneath the rotating process can be on before the rotation of Case 3 is the node directly above the rotation point.

In Case 4, the rotation is done on `xp`. Before Case 4, a process beneath the rotating process can be pulled up at most to the rotating process's `gp`, two levels up from `w` by Cases 1 and 3. Therefore, as in Case 3, the highest a marker of a process below the rotating process can be on before the rotation of Case 4 is the node directly above the rotation point.

Therefore, in any rotations during insertions and deletions, the highest node a marker of a process beneath the rotating process can be on is the node directly above the rotation point.

Hence, in our pictures below, a marker can be at highest on `r` before the rotation.

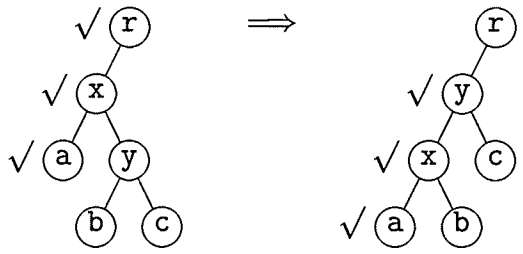


Figure A.1: Pattern 1: A left rotation at x requires the marker on r to be moved to y.

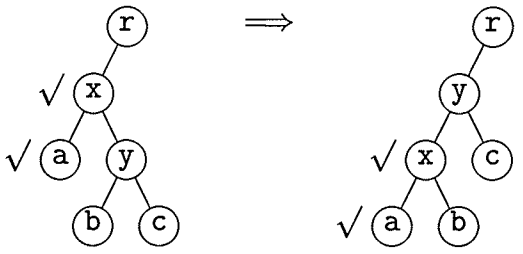


Figure A.2: Left rotation at x.

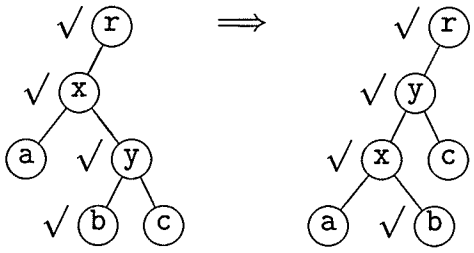


Figure A.3: Left rotation at x.

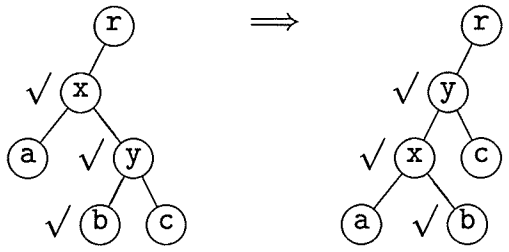


Figure A.4: Left rotation at x.

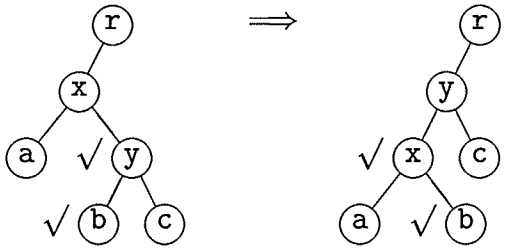


Figure A.5: Pattern 2: A left rotation at x requires the marker on y to be moved to x.

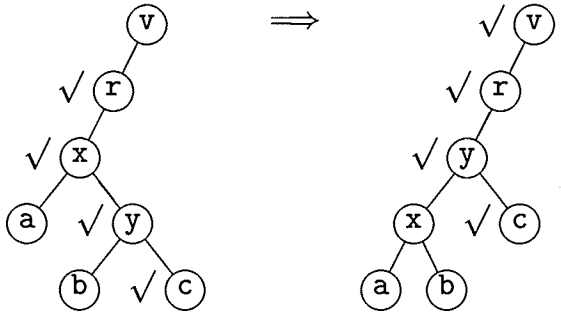


Figure A.6: Pattern 3: A left rotation at x requires the marker on x to be moved to v, an ancestor of x.

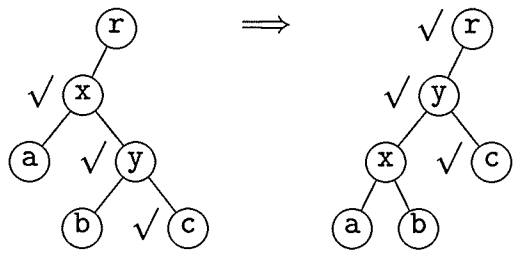


Figure A.7: Pattern 4: A left rotation at `x` requires the marker on `x` to be moved to `r`, an ancestor of `x`.

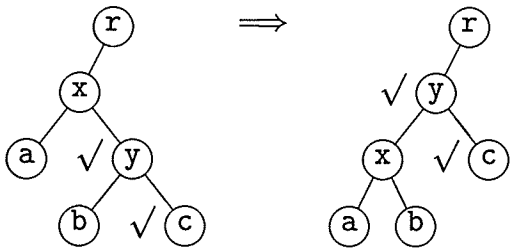


Figure A.8: Left rotation at `x`.

Bibliography

[1] Greg Barnes. A method for implementing lock-free data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, Velen, Germany, August 1993.

[2] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[3] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–273, Pittsburgh, Pennsylvania, USA, May 1993.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, USA, second edition, 2001.

[5] Intel Corporation. IA-32 intel architecture software developer's manual, volume 2: Instruction set reference, 2003.

[6] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[7] William B. Easton. Process synchronization without long-term interlock. In *Proceedings of the third ACM symposium on Operating Systems Principles*, pages 95–100, Palo Alto, California, USA, October 1971.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[9] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, pages 123–136, Seattle, Washington, USA, 1996.

[10] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, USA, 1978.

[11] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, Anaheim, California, USA, October 2003.

[12] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279, Toulouse, France, October 2002.

[13] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 276–290, Toronto, Ontario, Canada, August 1988.

[14] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, Washington, USA, March 1990.

[15] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[16] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[17] Maurice Herlihy, Mark Moir, Victor Luchangco, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of Distributed Computing*, pages 92–101, Boston, Massachusetts, USA, July 2003.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, California, USA, May 1993.

[19] IBM. System/370 principles of operation. Order Number GA22-7000. Reference Manual.

[20] IBM. PowerPC RISC microprocessor family: Programming environments manual for 64 and 32-bit microprocessors, version 2.0, 2003.

[21] SPARC International Inc. The SPARC architecture manual, version 9, 2000.

[22] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, Los Angeles, California, USA, August 1994.

[23] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[24] Gerry Kane. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

[25] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th annual ACM symposium on principles of distributed computing*, pages 130–140, Los Angeles, California, USA, August 1994.

[26] Jianwen Ma. Lock-free insertions of red-black trees. Master's thesis, University of Manitoba, October 2003.

[27] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.

[28] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, Pennsylvania, USA, May 1996.

[29] Maged M. Michael and Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 267–273, Geneva, Switzerland, 1997. IEEE.

[30] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The MIPS R4000 processor. *IEEE Micro*, 12(4):10–22, April 1992.

[31] Motorola. M68000 family programmer's reference manual, 1992.

[32] Henk J. Olivié. A new class of balanced search trees: Half-balanced binary search trees. *R.A.I.R.O. Theoretical Informatics*, 16(1):51–71, 1982.

[33] Serge A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–175, Edmonton, Alberta, Canada, 1989.

[34] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. Non-blocking algorithms for concurrent data structures. Technical Report 91-002, University of Florida, 1991.

[35] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.

[36] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[37] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Ontario, Canada, August 1995.

[38] Robert E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16(5):253–257, 1983.

[39] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

[40] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 212–222, San Diego, California, USA, July 1992.

[41] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Seminar on the Performance of Distributed and Parallel Systems*, pages 455–472, Kyoto, Japan, December 1988.