1-1-2001

Self-stabilizing binary search tree maintenance algorithm

Sylvain Ronan Brigant
University of Nevada, Las Vegas

SELF-STABILIZING BINARY SEARCH TREE MAINTENANCE ALGORITHM

by

Sylvain Ronan Brigant

Bachelor of Science
University of Nevada, Las Vegas
1999

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2001

UNLV

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

June 06, 20 01

The Thesis prepared by

Sylvain Ronan Brigant

Entitled

Self-Stabilizing Binary Search Tree Maintenance Algorithm

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Ajoy Kumar Datta
Examination Committee Chair

[illegible]
Dean of the Graduate College

John T. Minor
Examination Committee Member

[illegible]
Examination Committee Member

[illegible]
Graduate College Faculty Representative

PR/1017-53/1-00

ABSTRACT

Self-Stabilizing Binary Search Tree Maintenance Algorithm

by

Sylvain Ronan Brigant

Dr. Ajoy K. Datta, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Binary search tree is one of the most studied data structures. The main application of the binary search tree is in implementing efficient search operations. A binary search tree is a special binary tree which satisfies the property that for every processor p in the binary tree, the values of all the keys in the left subtree of p are smaller than that of p, and the values of all the keys in the right subtree of p are larger than that of p.

We present a self-stabilizing [Dij74] algorithm to maintain a binary search tree given a binary tree structure and a sequence of integers as input. This protocol uses neither the processors identifiers nor the size of the tree but assumes the existence of a distinguished processor (the root). The algorithm is self-stabilizing, meaning that starting from an arbitrary state, it is guaranteed to reach a legitimate state in a finite number of steps. The proposed algorithm assures that the set of integers eventually sent to the output environment is a permutation of the integers received from the input environment. The algorithm stabilizes in $O(hn)$ time units, where h and n represent the height and size, respectively, of the tree. The proposed algorithm is aimed at the hardwired binary tree structures where the topology of the trees cannot be adaptive to the change of the input values, but the input values are organized within a predefined environment.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. Ajoy K. Datta for his direction throughout the duration of this work. His confidence in me, dedication to his art, and belief in hard work are what motivated me to achieve this goal. Although I reached many brick walls while striving to always find a better result for the thesis, he never opted for the easy way out and was always available for help. I would also like to thank Dr. John Minor, Dr. Tom Nartker, and Dr. Henry Selvaraj for serving on my thesis committee. My special thanks to Dr. Franck Petit and Dr. Vincent Villain (Université de Picardie Jules Vernes, France) whose insight helped me improved the design of the algorithms.

I wish to thank all the faculty and staff of Computer Science Department, University of Nevada, Las Vegas for their sincere help and support during my graduate studies. On a more personal note, I would like to thank all my friends, especially Brad Botz and Melissa Leiper, for supporting me and not letting me give in to mediocrity. A very special thank you to Heather Howard for being so good to me and reviewing this thesis. Finally, I would like to dedicate this thesis to Philip Kernan Jr. who has always been there for me and has helped me reach all things important to me in life.

CHAPTER 1

INTRODUCTION

1.1 Self-Stabilization

The concept of self-stabilization was first introduced by Edsger W. Dijkstra in 1974 [Dij74]. It is now considered to be the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system guarantees that starting from an arbitrary state, the system converges to a legal configuration in a finite number of steps and remains in a legal state until another fault occurs. In a non-self-stabilizing system, the system designer needs to enumerate the faults, such as link/node failures, that the system will face, and then must add the corresponding recovery mechanisms. They are usually independent and may cause conflicts. Also, some obscure errors like memory corruption may be difficult to enumerate. It makes sense that, even if the error occurs rarely in the system, the networks should recover from those faults automatically [Var94]. In a large, distributed system, it is very hard to predict all the faults that may occur. Ideally, a system should continue its availability by correctly restoring the system state whenever the system exhibits incorrect behavior due to the occurrence of faults [AG93, Gou98]. The self-stabilizing technique provides a uniform mechanism to deal with not only arbitrary transient faults such as data, message, and location counter corruption [KP93], but also a variety of fault types like network congestion and software bugs [LAJ99]. The ability of the system to detect errors and correct itself without external intervention makes a self-stabilizing system more reliable, more powerful and more useful than a non-stabilizing system.

1.2 Binary Search Trees

Binary search tree is defined as a special binary tree which satisfies the property that for every processor p in the binary tree, the values of all the keys in the left subtree of p are smaller than that of p, and the values of all the keys in the right subtree of p are larger than that of p. When dealing with large amounts of information, the linear access time of most data structures is prohibitive. Binary search trees are a data structure for which the worst case running time of most operations is $O(log\ n)$. Binary search trees are very useful abstractions in computer science and find many important uses in fields such as compiler design, evaluation of arithmetic expressions, and the implementation of efficient search operations.

1.3 Our Contributions

Although many problems have been studied on the tree structures [Dol00] in the area of self-stabilization, there is not a stabilizing binary search tree algorithm to date. Our work takes an arbitrary binary tree as input. By arbitrary, we mean that the initial key values could be such that the tree is not a binary search tree. The presented stabilizing algorithm eventually produces a binary search tree where the sequence of output values of the processors is a permutation of the input sequence of integers. The stabilizing time of the algorithm is $O(hn)$ time units.

1.4 Outline of the thesis

The remainder of the thesis is organized as follows: Chapter 2 discusses the model of the system used in this work, along with some important definitions. Chapter 3 introduces the concept of wave schemes which are used throughout this thesis. Chapter 4 presents the search structure maintenance algorithm, its example, the specification of the problem, along with the correctness proof of this algorithm. Chapter 5 presents the binary search tree maintenance algorithm and the correctness proof of the BST Algorithm. Conclusions and some future research directions are discussed in Chapter 6.

CHAPTER 2

MODEL

2.1 Distributed System

A *distributed system* is an undirected connected graph, $S = (V, E)$, where V is a set of nodes ($|V| = n$) and E is the set of edges. Nodes represent *processors*, and edges represent *bidirectional communication links.* A communication link (p, q) exists iff p and q are neighbors. We consider networks which are *asynchronous* and *tree structured.* No processor has any identity except the one, called the **root**. Every processor p holds exactly one key value denoted as $K_p \in \mathbf{Z}$. In the traditional binary search tree, the key values are assumed to be unique. But, since this paper deals with the faulty environment, we assume that the key values may not be unique. We denote the set of *leaf* and *internal* processors by LP and IP, respectively.

The set of neighbors of every processor p is denoted as N_p. To simplify the presentation, we will consider one of the neighbors of processor p ($p \neq$ **root**), which is on the path from the **root** to p, as the *parent* of p. We will denote this special neighbor of p as P_p. We assume that $P_{\mathbf{root}} = \perp$, where $\perp$ indicates the null pointer. The rest of the neighbors of p will be assumed to compose the set of *children*, CH_p, of p, i.e., $CH_p = N_p \setminus \{P_p\}$. We will also denote the left child of p as L_p and the right child of p as R_p. Every processor p ($p \notin LP$) is itself the root of its subtree. We further define the subtree rooted at L_p (R_p respectively) as p's left subtree (right subtree).

We consider *semi-uniform* protocols. So, every processor with the same degree executes the same program, excluding the **root**. The program consists of a set of *shared variables* (henceforth, referred to as variables) and a finite set of actions. A processor can only write to its own variables and read its own variables and variables owned by the neighboring processors.

2.2 Program

Each action is of the following form: $<$ *label* $>::$ $<$ *guard* $>$ $\longrightarrow$ $<$ *statement* $>$. The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. When p executes a statement, we say that "p executes an action". An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step. This model is known as the *state* model. The *state* of a processor is defined by the values of its variables. The *state* of a system is the product of the states of all processors ($\in V$).

In the sequel, we refer to the state of a processor and system as a (*local*) *state* and *configuration*, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = \gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. The set of all possible computations of $\mathcal{P}$ in system S is denoted as $\mathcal{E}$.

A processor p is said to be *enabled* if there exists an action A such that the guard of A is true. Similarly, an action A is said to be enabled at p if the guard of A is true at p. We assume an *weakly fair and distributed daemon*. The *weak fairness* means that if a processor p is continuously enabled, then p will be chosen by the daemon in a finite amount of time. The *distributed* daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses at least one (possibly more) of these enabled processors to execute an action.

In order to compute the time complexity measure, we use the definition of *round* [DIM97]. This definition captures the execution rate of the slowest processor in any computation. Given a computation e, the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or the disable action) of every continuously enabled processor from the first configuration. Let e'' be the suffix of e, i.e., $e = e'e''$. The *second round* of e is the first round of e'', and so on.

2.3 Self-Stabilization

Let $\mathcal{X}$ be a set. $x \vdash P$ means that an element $x \in \mathcal{X}$ satisfies the predicate P defined on the set $\mathcal{X}$. A predicate is non-empty if there exists at least one element that satisfies the predicate. We define a special predicate **true** as follows: *for any* $x \in \mathcal{X}$, $x \vdash$ **true**.

We use the following term, *attractor* in the definition of self-stabilization.

Definition 2.3.1 (Attractor) *Let X and Y be two predicates of a protocol $\mathcal{P}$ defined on C of system $\mathcal{S}$. Y is an attractor for X if and only if the following condition is true:*

$\forall \alpha \vdash X : \forall e \in \mathcal{E}_\alpha : e = (\gamma_0, \gamma_1, ...) :: \exists i \geq 0, \forall j \geq i, \gamma_j \vdash Y$. *We denote this relation as* $X \triangleright Y$.

Definition 2.3.2 (Self-stabilization) *The protocol $\mathcal{P}$ is self-stabilizing for the specification $\mathcal{SP}_\mathcal{P}$ on $\mathcal{E}$ if and only if there exists a predicate $\mathcal{L}_\mathcal{P}$ (called the legitimacy predicate) defined on C such that the following conditions hold:*

1. $\forall \alpha \vdash \mathcal{L}_\mathcal{P} : \forall e \in \mathcal{E}_\alpha :: e \vdash \mathcal{SP}_\mathcal{P}$ *(correctness).*
2. *true* $\triangleright \mathcal{L}_\mathcal{P}$ *(closure and convergence).*

CHAPTER 3

WAVE SCHEMES

Our algorithm uses a special *propagation of information with feedback scheme*, called the *PFC (Propagation of Information with Feedback and Cleaning)* [BDPV99]. The PFC scheme [BDPV99] implements a state optimal and snap-stabilizing Propagation of Information with Feedback (PIF) scheme [Cha82, Seg83]. Moreover, this scheme is snap-stabilizing, i.e., it guarantees that the system always maintains the desirable behavior. A snap-stabilizing (also introduced in [BDPV99]) algorithm is also a self-stabilizing algorithm which stabilizes in 0 rounds, i.e., optimal in terms of the worst-case stabilization time. In this section, we give a quick overview of the PIF scheme and the PFC scheme. For more information on this scheme, refer to [BDPV99].

3.1 PIF Scheme

Let us quickly review the well-known *PIF scheme* [Cha82, Seg83] on tree structured networks. The PIF scheme is the repetition of a *PIF cycle* consisting of broadcast phase and feedback phase. The PIF cycle can be informally defined as follows: Starting from an initial configuration where no message has yet been broadcast, the **root** initiates the *broadcast* phase. The descendants of the **root** (except the leaf processors) participate in this phase by forwarding the broadcast message to their descendants. Once the broadcast phase reaches the leaf processors, since the leaf processors have no descendants, they notify their parent of the termination of the broadcast phase by initiating the *feedback* phase. When every processor, except the **root**, is done sending the feedback message to its parent, the **root** executes a special internal action indicating the *termination* or completion of the current PIF cycle.

3.2 PFC Scheme

Introduced in [BDPV99], the PFC adds a new phase called the *cleaning phase* to the PIF scheme. The cleaning phase is initiated by the leaf processors after they initiated the feedback phase (Figure 3.2(i)). As the feedback phase works its way back to the **root**, processors in the tree may participate in their cleaning phase (in parallel with the feedback phase), provided that they have executed their feedback phase and all their neighbors have also executed their feedback or cleaning phase (Figure 3.2(ii)). When the feedback phase reaches the descendants of the **root** (Figure 3.2(iii)), the **root** executes its cleaning phase. The **root** then waits until all of its descendants are in the cleaning phase (Figure 3.2(iv)) before initiating the next PIF cycle. To make sure that the cleaning phase does not meet the broadcast phase (i.e., the processors in the cleaning phase do not confuse the processors in the broadcast phase), a processor can clean its states only if all its neighbors are in the feedback or cleaning phase.

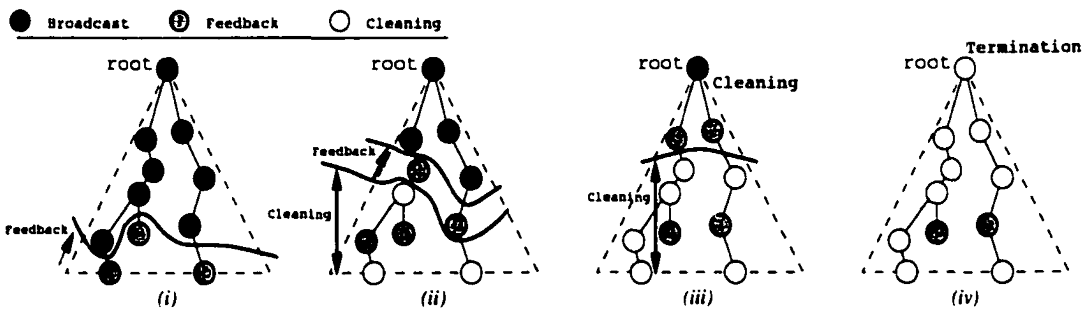

Figure 3.2.1: A $\mathcal{PFC}$ Cycle.

CHAPTER 4

SEARCH STRUCTURE MAINTENANCE ALGORITHM

We are now ready to propose a self-stabilizing search structure maintenance algorithm (Algorithm $\mathcal{SM}$). We present the overall idea about the algorithm. the data structure used by the algorithm. and finally. an informal explanation of the algorithm using an example.

4.1 Specification of the Binary Search Tree Problem

The tree produced by Algorithm $\mathcal{SM}$ must satisfy the following conditions:

Specification 4.1

[L] *For every processor p in the binary tree, the values of all the keys in the left subtree of p are smaller or equal to that of p.*

[R] *For every processor p in the binary tree, the values of all the keys in the right subtree of p are larger or equal to that of p.*

[V] *The output sequence of key values of the binary search tree is a permutation of the input sequence of key values of the binary tree.*

We also want the search structure maintenance algorithm to be self-stabilizing.

4.2 Search Structure Maintenance Algorithm

The goal of this algorithm is to create two distinct sets of key values at each processor, one in each of its left and right subtrees, such that all the key values in the left subtree are smaller or equal to that in its right subtree. In other words. Algorithm $\mathcal{SM}$ will satisfy Conditions [L] and [R] of Specification 4.1. We call this algorithm the search structure maintenance algorithm rather than the binary search tree maintenance algorithm because it does not satisfy Condition [V] of the specification of the BST problem. The algorithm starts from the root and follows the processors top-down in the tree, creating the above two sets at every processor along the way. The processors execute the following repeatedly:

Execute Range Evaluation Test:

- All key values within range: Do nothing.
- Some key values out of range: Execute Swap Cycle to swap the largest key in the left subtree with the smallest key value in the right subtree.

Range Evaluation Test. Every processor p keeps track of the range of key values in both its left and right subtrees using two variables, called *MinMax Values*. Min_p contains the smallest key value in p's subtree: similarly, Max_p contains the largest key value in p's subtree.

The first action executed by any given processor is to determine if all the MinMax values of its two subtrees are valid. We refer to this action as the *Range Evaluation Test*. The MinMax values in p's subtree are said to be within a valid range if $Min_{R_p} \geq K_p \geq Max_{L_p}$; that is, the smallest key value in p's right subtree is larger or equal to p's own key value which in turn is also larger or equal to the largest key value p's left subtree. If the above inequality does not hold, p is said to fail the *Range Evaluation Test*, and at that point, p may initiate a *Swap Cycle*.

Swap Cycle. The objective of the Swap cycle is to swap the key values of two processors, one in each subtree of an enabled processor p, such that the largest value in p's left subtree is moved to p's right subtree, and the smallest key value in p's right subtree is moved to its left subtree. Processor p must meet the following two conditions to initiate a swap cycle: (i) p has failed the range evaluation test and (ii) p is *temporarily stable*. If p meets both the above conditions, then p is said to be an *initiator*, denoted as `init`.

Definition 4.2.1 (Temporarily Stable) *A processor is called* temporarily stable *if it is in a Clean state and its parent is permanently stable. Note that since the* `root` *is the only processor without a parent, it will be* temporarily stable *if it is in Clean state.*

Definition 4.2.2 (Permanently Stable) *A processor is called* permanently stable *if it is temporarily stable and does not fail the range evaluation test in any future configuration.*

As mentioned before, the algorithm works normally top down starting at the `root`, since it is the only processor which can become temporarily stable regardless of the parent's status. When a temporarily stable processor p fails the range evaluation test, it becomes an `init` and can initiate a swap cycle. After the Swap cycle terminates, p executes the range evaluation test again and initiates another swap cycle if the test fails again. This cycle is repeated until the test's inequality holds for p. Next, p becomes permanently stable which allows its children to become temporarily stable, and, therefore, to become `init`, if necessary.

The swap cycle is implemented by using a slightly modified PFC scheme. The broadcast and feedback phases used to describe the PFC scheme are altered in the swap cycle and are called the *Search* and *Response* phases, respectively. The purpose of these two phases is not to get all processors of the tree involved as in the PFC scheme, but only to reach the two processors (one on each side of the `init`'s subtrees) and then carry the information back to the `init`. In initiating the Swap cycle, `init` first copies $Max_{L_{init}}$ (the key value of some processor p_j in its left subtree) and $Min_{R_{init}}$ (the key value of some processor p_k in its right subtree) into two temporary variables. These values are then sent down the tree in the Search phase until they reach p_j and p_k on two sides of `init`'s tree. The Search phase uses the MinMax variables to trace the path towards the two processors p_j and p_k, setting its status to alert only its child processor holding the desired key value in its MinMax variables. When processor p_j (p_k), holding value $Max_{L_{init}}$ ($Min_{R_{init}}$) as its key value, is found, it uses the information sent in the Search phase to replace its key value with p_k's (p_j's) key value and updates its MinMax values to reflect the changes. Next, p_j (p_k) initiates the Response phase. At each step of the Response phase, the enabled processors update their MinMax values based on the information received from their children and notify their parent. Upon receiving the Response phase, `init` updates its MinMax values and terminates the Swap cycle. In the meantime, both p_j and p_k initiate the clean phase of the algorithm as described in Chapter 3. All MinMax values in `init`'s subtree are now up-to-date.

The key values may not be unique in the tree. The Search phase may reach a processor q where both Max_{L_q} and Max_{R_q} (resp. Min_{L_q} and Min_{R_q}) of a processor are equal to $Max_{L_{init}}$ (resp. $Min_{R_{init}}$). Since we want to swap only one key value per subtree, the algorithm chooses the left path and ignores the right path of the `init`'s tree.

4.2.1 Data Structure

We have already discussed the variables (of every processor p) K_p, P_p, L_p, R_p, Max_p, and Min_p. Variable M_p records the status of p involved in a swap cycle: permanently stable (N), an initiator (I), clean (C), involved in a Search phase with its left child (SL), its right child (SR), or involved in a Response phase (R). Note that the special processor **root** does not have the states SL, SR, and R, and the leaf processors do not have I, SL, and SR. Variables F_p and T_p are used to hold (temporarily) $Min_{R_{init}}$ and $Max_{L_{init}}$, respectively, during the swap cycle. The self-stabilizing binary search tree maintenance algorithm (Algorithm $\mathcal{SM}$) is shown in Algorithm 4.2.1.

4.3 An Example of Algorithm $\mathcal{SM}$

We consider the case of an initiator which fails the range evaluation test and then executes a Swap cycle. This is explained using Figure 4.3.1. Processor b is the initiator of the swap cycle.

Configuration (i) - (ii). Configuration (i) shows our starting configuration. Processor a is permanently stable, meaning that the property $Max_{L_a} \leq K_a \leq Min_{R_a}$ holds. Since processor c is a leaf processor and its parent is permanently stable, c is permanently stable. Processor b executes Action SA_1, since it is temporarily stable (Predicate *Potential_Initiator*(p)), and it fails the range evaluation test (Predicate *Good_Range*(p)).

Algorithm 4.2.1 Algorithm $\mathcal{SM}$

```
Variables:
  $K_p$, $Min_p$, $Max_p$, $T_p$, $F_p \in \mathbf{Z}$
  $P_p \in N_p$ for $p \neq$ root, $P_p = \perp$ for $p =$ root
  $L_p$, $R_p \in N_p \cup \{\perp\}$; $M_p \in \{N, I, C, SL, SR, R\}$

Actions:
  $SA_1$ :: $Potential\_Initiator(p) \wedge \neg Good\_Range(p) \longrightarrow SInitiate_p$
  $SA_2$ :: $Potential\_Initiator(p) \wedge Good\_Range(p) \longrightarrow M_p := N$
  $SA_3$ :: $Sending\_Parent(p) \longrightarrow SForward_p$
  $SA_4$ :: $Ack\_Children(p) \longrightarrow SAck_p$
            {Error Correction}
  $SA_5$ :: $\neg Correct\_MinMax(p) \longrightarrow Update\_MinMax_p$
  $SA_6$ :: $Transient\_Status(p) \longrightarrow M_p := C$

Predicates:
  $Potential\_Initiator(p) \equiv Temp\_Stable(p) \wedge Correct\_MinMax(p) \wedge Clean\_Children(p)$
  $Good\_Range(p) \equiv Correct\_Subtrees(p) \wedge Correct\_Left(p) \wedge Correct\_Right(p)$
  $Sending\_Parent(p) \equiv (M_p = C) \wedge ((M_{P_p} = I) \vee ((M_{P_p} = SL) \wedge (L_{P_p} = p)) \vee$
      $((M_{P_p} = SR) \wedge (R_{P_p} = p)))$
  $Ack\_Children(p) \equiv ((M_p = I) \wedge (\forall d \in CH_p, M_d = R)) \vee$
      $((M_p = SL) \wedge (M_{L_p} = R)) \vee ((M_p = SR) \wedge (M_{R_p} = R))$
  $Correct\_MinMax(p) \equiv (Min_p = min(\{Min_d :: d \in CH_p\} \cup \{K_p\})) \vee$
      $(Max_p = max(\{Max_d :: d \in CH_p\} \cup \{K_p\}))$
  $Transient\_Status(p) \equiv ((M_p = R) \wedge (M_{P_p} \in \{R, C\})) \vee ((M_p \in \{I, N\}) \wedge (M_{P_p} \neq N)) \vee$
      $((M_p \in \{SL, SR\}) \wedge (M_{P_p} \notin \{I, SL, SR\})) \vee$
      $((M_p = N) \wedge \neg(Correct\_MinMax(p) \wedge Good\_Range(p)))$
  $Temp\_Stable(p) \equiv (M_p = C) \wedge ((P_p \neq \perp) \Rightarrow (M_{P_p} = N))$
  $Clean\_Children(p) \equiv (\forall d \in CH_p, M_d = C)$
  $Correct\_Subtrees(p) \equiv ((L_p \neq \perp) \wedge (R_p \neq \perp)) \Rightarrow (Min_{R_p} \geq Max_{L_p})$
  $Correct\_Left(p) \equiv (L_p \neq \perp) \Rightarrow (Correct\_Subtrees(p) \Rightarrow (Max_{L_p} \leq K_p))$
  $Correct\_Right(p) \equiv (R_p \neq \perp) \Rightarrow (Correct\_Subtrees(p) \Rightarrow (Min_{R_p} \geq K_p))$

Macros:
  $SInitiate_p \equiv$
    if $\neg Correct\_Subtrees(p)$
    then $(T_p, F_p) := (Max_{L_p}, Min_{R_p})$;
    else
      if $\neg Correct\_Left(p)$
      then $(T_p, F_p) := (Max_{L_p}, K_p)$; $K_p := Max_{L_p}$;
      else $(T_p, F_p) := (K_p, Min_{R_p})$; $K_p := Min_{R_p}$;
    $M_p := I$;
  $SForward_p \equiv$
    $(T_p, F_p) := (T_{P_p}, F_{P_p})$;
    if $(M_{P_p} = I) \wedge (R_{P_p} = p)$ then $(T_p, F_p) := (F_p, T_p)$;
    if $T_p = K_p$
    then $K_p := F_p$; $M_p := R$; $Update\_MinMax_p$;
    else
      if $(L_p \neq \perp) \wedge (T_p \in \{Min_{L_p}, Max_{L_p}\})$
      then $M_p := SL$;
      else
        if $(R_p \neq \perp) \wedge (T_p \in \{Min_{R_p}, Max_{R_p}\})$
        then $M_p := SR$;
        else $M_p := R$; $Update\_MinMax_p$;
  $SAck_p \equiv$
    if $M_p = I$ then $M_p := C$; else $M_p := R$;
    $Update\_MinMax_p$;
  $Update\_MinMax_p \equiv$
    $Min_p := min(\{Min_d :: d \in CH_p\} \cup \{K_p\})$;
    $Max_p := max(\{Max_d :: d \in CH_p\} \cup \{K_p\})$;
    if $M_p = N$ then $M_p := C$;
```

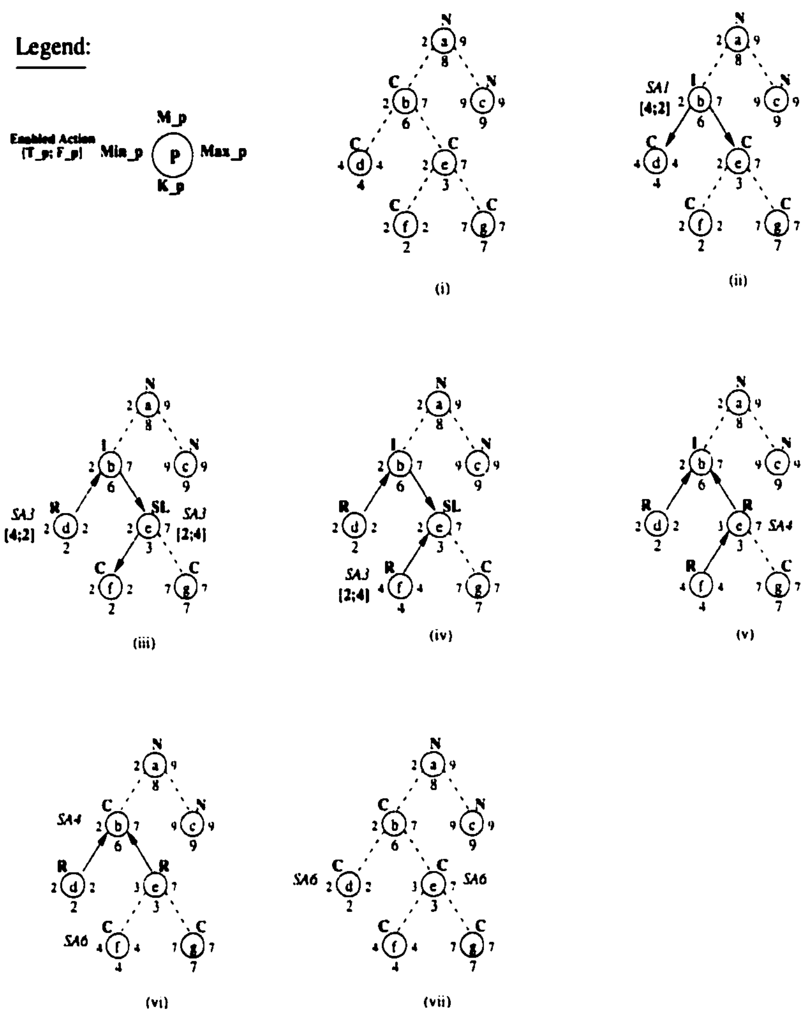

Figure 4.3.1: Execution of a Swap Cycle.

Processor b, the initiator **init**, initiates a swap cycle (Macro $SInitiate_p$) by copying its Max_{L_b} and Min_{R_b} into T_b and F_b, respectively, and setting its status to I.

Configuration (iii). Processors d and e receive the S-Broadcast. Because d holds the key value we are searching for in b's left subtree (Predicate $SForward$), it executes Action SA_3. Processor d first copies the values of T_b and F_b into its own variables, then copies F_d into its key variable K_d and initiates the Response phase after updating its MinMax values to reflect the changes (Macro $SForward_p$). On the other hand, since e is the right child of the initiator b, and it does not hold the key value we are searching for in the right subtree

(Predicate $SForward(p)$), it executes Action SA_3. Processor e first copies the values of T_b and F_b into its own variables and must also switch the two values in order for the the Search phase to find for the correct value down the right subtree. Next, e forwards the Search phase to its children using Macro $SForward_p$.

Configurations (iv). Processor f receives the Search phase; because g does not hold any of the MinMax values we are searching for, it ignored its parent's request. Since f holds the key value we are searching for in b's right subtree (Predicate $SForward$), it executes Action SA_3. Processor f first copies the values of T_e and F_e into its own variables; then it copies value F_f into its key variable K_f and initiates the Response phase after updating its MinMax values to reflect the changes (Macro $SForward_p$).

Configurations (v) - (vii). Upon receipt of the Response from both its children (Predicate $Ack\_Children(p)$), e executes Macro $SAck_p$ to join the Response phase and updates its MinMax values (Action SC_4). Processor b then executes Action SA_4 in Configuration (vi) to sets its status to *clean* (Macro $Ack\_Children(p)$). In the meantime, every processor in its subtree executes Action SA_6 to reset its status from *response* back to *clean*, and, eventually, the system reaches Configuration (vii) where b may initiate another Swap cycle.

4.4 Correctness of Algorithm $\mathcal{SM}$

We begin the Correctness section by giving a few definitions. We then show that all MinMax values and processor status' in the tree are eventually corrected. Once this has been established, the tree is in a *normal configuration*, and we show that Algorithm $\mathcal{SM}$ halts in a finite amount of time.

Lemma 4.1 *Starting from an arbitrary configuration, the MinMax values of the given binary tree are corrected in at most h rounds.*

Proof. Starting at the leaf processors, the MinMax values are corrected in 1 round (Action SA_5). Using induction on the height of the tree, all processors in the tree will have corrected MinMax values in at most h rounds. □

Lemma 4.2 *Assume all MinMax values are correct in the tree. Let a processor p be in a clean state, let P_p be permanently stable and assume the range evaluation test fails. Processor p initiates a Swap cycle in at most 3 rounds.*

Proof. Before p can initiate a Swap cycle, both L_p and R_p must also be in a clean state. We break our proof into three cases. Without loss of generality we only consider processor L_p and refer to that processor as processor l in the proof.

There are three cases:

Case 1 : Assume that $M_l = C$.

This case is trivial.

Case 2 : Assume that $M_l \in \{N, I, SL, SR\}$.

Using Action SA_6, M_l is reset to C in one round.

Case 3 : Assume that $M_l = R$.

Before M_l can be reset to C, its children must either be in a clean state or a response state. Without loss of generality, we only consider processor L_l:

i. Assume that $M_{L_l} = R$ *or* C. This case is trivial.

ii. Assume that $M_{L_l} \in \{N, I, SL, SR\}$. Using Action SA_6, M_{L_l} is reset to C in one round.

In the three cases, M_l is reset to C in at most 2 rounds. Therefore, M_p is set to I in at most 3 rounds and begins executing a Swap cycle. □

Property 4.4.1 *Each Swap cycle has a cost of at most $2h$ rounds.*

Proof. An initiator `init` initiates the Swap cycle by setting $C_{\texttt{init}} = I$. In at most h rounds, the Search phase either finds the given processor in either side of I's subtree or reaches the leaf processors of that subtree. The Response phase is then initiated, reaches **init** in at most h rounds, and is terminated once $C_{\texttt{init}}$ is reset to C. therefore the total cost of one Swap cycle to at most $2h$ rounds. □

Property 4.4.2 *Assume all MinMax values are correct in the tree. Let max_1 and min_1 be the correct MinMax values $Max_{L_{\texttt{init}}}$ and $Min_{L_{\texttt{init}}}$ respectively of a given initiator* **init** *before that initiator executes a Swap cycle. Let max_2 and min_2 be the correct MinMax values $Max_{L_{\texttt{init}}}$ and $Min_{L_{\texttt{init}}}$ respectively of initiator* **init** *after the termination of the Swap cycle.*

Then $\begin{cases} max_1 \geq max_2 \\ min_1 \leq min_2 \end{cases}$

Lemma 4.3 *Starting from an arbitrary configuration where all MinMax values are correct in the tree,* **root** *eventually becomes permanently stable.*

Proof. In oder to be permanently stable, **root** must meet the following requirements: $M_{\texttt{root}} = N$ and $Max_{L_{\texttt{root}}} \leq K_{\texttt{root}} \leq Min_{R_{\texttt{root}}}$. We break our proof into three cases:

Case 1 : Assume that $M_{\texttt{root}} = N$.

If $Max_{L_{\texttt{root}}} \leq K_{\texttt{root}} \leq Min_{R_{\texttt{root}}}$ then **root** is permanently stable. Otherwise, by Action SA_6, $M_{\texttt{root}}$ is reset to C.

Case 2 : Assume that $M_{\texttt{root}} = C$.

If $Max_{L_{\texttt{root}}} \leq K_{\texttt{root}} \leq Min_{R_{\texttt{root}}}$ then $M_{\texttt{root}}$ is reset to N and **root** is permanently stable. Otherwise, by Lemma 4.2, $M_{\texttt{root}}$ eventually is set to I and **root** begins executing a Swap cycle.

Case 3 : Assume that $M_{\texttt{root}} = I$.

Root has initiated a Swap cycle. By Property 4.4.1, the Swap cycle will terminated in at most $2h$ rounds and $M_{\texttt{root}}$ eventually is reset to C.

By Property 4.4.2, with each Swap cycle executed the values of the key variables in **root**'s left subtree consistently decrease while the key variables in **root**'s right subtree consistently increase. Since both sets of key variables are finite, eventually after the completion of a Swap cycle, $Max_{L_{\mathtt{root}}} \leq K_{\mathtt{root}} \leq Min_{R_{\mathtt{root}}}$ and $M_{\mathtt{root}}$ is reset to N making **root** permanently stable. □

Lemma 4.4 *Assume all MinMax values are correct in the tree. Starting from a configuration where* **root** *is permanently stable, all processors in the tree eventually become permanently stable.*

Proof. Starting with `root`'s children and, using induction on the height of the three, for every processor p whose parent is permanently stable, the proof follows from Lemma 4.3. We note that if C_p is equal to either SL, SR or R, those states are reset to C using Action SA_6. □

From Lemmas 4.1, 4.2, and 4.3 follows:

Theorem 4.4.1 *Starting from an arbitrary configuration, all processors in the given binary tree eventually become permanently stable and so form a binary search tree.*

4.5 Complexity of Algorithm $\mathcal{SM}$

We first establish the maximum number of swap cycles each initiator `init` can execute. We first establish the result for the **root**. Let us define a *correct Swap cycle* as a swap cycle where $T_{\mathtt{init}}$ and $F_{\mathtt{init}}$ is equal to $Max_{L_{\mathtt{init}}}$ and $Min_{R_{\mathtt{init}}}$, respectively. We show that the **root** starts executing such cycles in at most $3h$ rounds.

Lemma 4.5 *The* **root** *begins executing its first correct swap cycle in at most $3h$ rounds.*

Proof. By Lemma 4.1, all MinMax values in the tree are correct in at most h rounds. Starting from such a configuration, if $M_{\mathtt{root}}$ is arbitrarily set to I, the **root** begins executing a swap cycle without guaranteeing that $T_{\mathtt{root}}$ and $F_{\mathtt{root}}$ have been set to the newly corrected MinMax values. By Property 4.4.1, each swap cycle has a cost of at most $2h$ rounds, the **root** must begin executing its first correct swap cycle in at most $3h$ rounds. □

Lemma 4.6 *Every initiator* `init` *can execute at most $\frac{n}{2}$ correct swap cycles.*

Proof. It can be easily observed that the worst case of any initiator is to swap every key value from one side of its subtree to the other side of the subtree where the the two sides of the subtree have the same height. The total number of swap cycles in this type of subtree is equal to the minimum number of processors in either the left or the right subtree. Hence, the result follows. □

Lemma 4.7 *The* `root` *will be permanently stable in at most* $3h + 2h(\frac{n}{2})$ *rounds.*

Proof. Follows from Lemmas 4.5 and 4.6, and Property 4.4.1. □

Note that the cost of $3h$ rounds of the `root` to reach the first correct swap cycle will not be accrued by all other processors since once the `root` is permanently stable. its children are automatically temporarily stable and hence, can begin executing the correct swap cycles immediately.

Theorem 4.5.1 *The binary tree will be stabilized in* $O(hn)$ *rounds.*

Proof. Follows from Lemmas 4.6 and 4.7 by induction on the height of the tree. □

CHAPTER 5

BINARY SEARCH TREE MAINTENANCE ALGORITHM

Algorithm $\mathcal{SM}$ presented in Chapter 4 may still produce an incorrect binary search tree based on the input key values if some keys get corrupted. In this case, the algorithm will deliver an output sequence containing the corrupted input values. In other words. Algorithm $\mathcal{SM}$ does not satisfy Property [V] of Specification 4.1. We present a solution to this problem in this section. Algorithm $\mathcal{BST}$ presented in this section is an extension of Algorithm $\mathcal{SM}$ in that it satisfies Property [V] of Specification 4.1, and, hence, solves the BST problem.

First, we remove the silence property [DGS96] of Algorithm $\mathcal{SM}$ by running the algorithm repeatedly. Everytime we restart the search tree maintenance process, we reset the processors. Here, resetting means to initialize the key values to the input values. So, in case the keys were corrupted earlier, the next run of the algorithm will create the proper output which will be a permutation of the input sequence. So, we need to use a reset mechanism. Also, to be able to start the reset phase at the right time, we need to use a termination detection scheme. Both the termination detection and reset schemes are implemented using the PFC scheme. The Termination Detection Scheme (TDS) is initiated and terminated at the **root**. It returns *True* if all processors in the tree have finished swapping and the tree is a binary search tree. Otherwise, TDS returns *False*. When TDS returns *True*, the Reset Scheme (RS) is initiated. Starting from the **root**, RS copies each of the processor's key value to the output environment if the output value is not equal to the key value and copies the input environment's new value into the processors' key variable.

5.1 Termination Detection Scheme (TDS)

TDS runs in parallel with Algorithm $\mathcal{SM}$ and mutually exclusively with RS. TDS is presented in Algorithm 5.1.1. The **root** initiates a broadcast phase when RS has terminated (Predicate *Terminated*) and the **root** is permanently stable and has correct MinMax values (Predicate *Ready_Initiate*). All processors must be in the clean state C before taking part in the broadcast phase. At each step, the internal processors join TDS (Macro *TBroadcast*) and forward the broadcast to their children. The leaf processors are the first to decide (done in the feedback phase) if they are ready to terminate by executing Macro *Terminate*. They terminate if they are permanently stable and their MinMax values are correct. A non-leaf processor decides the termination in the feedback phase. It terminates only if all its children have terminated and they are also ready to terminate (Action TA_4). When the children of the **root** terminate, the **root** also terminates. At that point, we consider that Algorithm $\mathcal{SM}$ is no longer running, and we may reset the system (Macro *Terminate*). If during the feedback phase, a processor p is not permanently stable, has incorrect MinMax values, or receives a *Dont_Terminate* message from one of its children, then p executes Macro *Dont_Terminate*, meaning that it is not ready to be reset, and the value of *Dont_Terminate* eventually reaches the **root**. In this situation, the system is not reset and the **root** initiates a new TDS.

5.2 Reset Scheme (RS)

The RS runs in a mutually exclusive fashion to both the TDS and the Algorithm $\mathcal{SM}$. The RS is presented in Algorithm 5.2.1. The actions notation of the PFC scheme remains intact as shown [BDPV99]; only the needed predicates have been added as needed. Starting at the **root**, the wave initiates once the TDS has terminated (Predicate *Terminated*), meaning that both itself and Algorithm $\mathcal{SM}$ are inactive. All processors must be in the clean state C before taking part in the broadcast part of the scheme. Upon receiving the broadcast, each processor joins the RS (Macro *TBroadcast*) and updates the output environment's key value (OK_p) if that value is not equal to that of the processors and

Algorithm 5.1.1 Termination Detection Wave

```
Variables:
  $W_p \in \{T, R\}$; $S_p \in \{B, F, C\}$; $TD_p \in \{True, False\}$

Actions:
                    {Root Only}
  $TA_1$ :: $Ready\_TInitiate(p)$      $\longrightarrow$ $TInitiate_p$
  $TA_2$ :: $Root\_Ready\_TClean(p)$   $\longrightarrow$ $S_p := C$,
                                          if $Neighbors\_Terminated(p)$
                                          then $Terminate_p$,
                                          else $Dont\_Terminate_p$;
                    {Other Processors}
  $TA_3$ :: $Ready\_TBroadcast(p)$     $\longrightarrow$ $TBroadcast_p$;
  $TA_4$ :: $Ready\_TFeedback(p)$      $\longrightarrow$ $TFeedback_p$;
  $TA_5$ :: $Ready\_Clean(p)$          $\longrightarrow$ $S_p := C$;

Predicates:
  $Ready\_TInitiate(p) \equiv S_p = C \wedge (\forall q \in N_p :: S_q = C) \wedge \neg Terminated(p) \wedge Perm\_Stable(p)$
  $Root\_Ready\_TClean(p) \equiv S_p = B \wedge (\forall q \in N_p :: S_q = F) \wedge In\_TCycle(p)$
  $Neighbors\_Terminated(p) \equiv (\forall q \in N_p :: TD_q = True)$
  $Ready\_TBroadcast(p) \equiv S_p = C \wedge S_{P_p} = B \wedge (\forall d \in CH_p :: S_d = C) \wedge In\_TCycle(P_p)$
  $Ready\_TFeedback(p) \equiv In\_TCycle(P_p) \wedge S_{P_p} = B \wedge (\forall d \in CH_p :: S_d = F)$
  $Ready\_Clean(p) \equiv (S_p = F \wedge S_{P_p} \in \{F, C\}) \vee S_p = B \wedge S_{P_p} \in \{F, C\} \vee$
                     $S_p = F \wedge (\forall q \in N_p :: S_q \in \{F, C\})$
  $Terminated(p) \equiv TD_p = True$
  $Perm\_Stable(p) \equiv M_p = N \wedge Good\_Range(p)$
  $In\_TCycle(p) \equiv W_p = T$
  $Children\_Terminated(p) \equiv (\forall d \in CH_p :: TD_d = True)$

Macros:
  $TInitiate_p \equiv S_p := B, Join\_TCycle_p$;
  $Terminate_p \equiv TD_p := True$
  $Dont\_Terminate_p \equiv TD_p := False$
  $TBroadcast_p \equiv S_p := B, Join\_TCycle_p$
  $TFeedback_p \equiv$ if $In\_TCycle(p) \wedge S_p = B$
      then { $S_p := F$,
             if $Children\_Terminated(p) \wedge Perm\_Stable(p)$
             then $Terminate_p$;
             else $Dont\_Terminate_p$; }
      else if $S_p = C \wedge (L_p = R_p = \bot)$
           { $S_p := F, Join\_TCycle_p$,
             if $Perm\_Stable(p)$
             then $Terminate_p$;
             else $Dont\_Terminate_p$; }
  $Join\_TCycle_p \equiv W_p := T$
```

copies the input environment's value (IK_p) into its key variable (Macro *Reset_Values*). Upon reaching the leaf processors, the feedback scheme is initiated and is terminated once the **root** has received it. At each step of the feedback scheme, the MinMax values are updated to reflect the new values in the tree; therefore, once the **root** is reached all the MinMax values in the tree are up to date. Before terminating the wave, the **root** executes Macro *Dont_Terminate* enabling both the TS and Algorithm $\mathcal{SM}$.

Algorithm 5.2.1 Reset Wave

```
Variables:
  $OK_p,\ IK_p \in \mathbb{Z}$

Actions:
                  {Root Only}
  $RA_1$ :: $Ready\_RInitiate(p) \longrightarrow RInitiate_p;$
  $RA_2$ :: $Root\_Ready\_RClean(p) \longrightarrow Terminate\_Reset_p$
                  {Other Processors}
  $RA_3$ :: $Ready\_RBroadcast(p) \longrightarrow RBroadcast_p$
  $RA_4$ :: $Ready\_RFeedback(p) \longrightarrow RFeedback_p$
  $RA_5$ :: $Ready\_Clean_p \longrightarrow S_p := C;$

Predicates:
  $Ready\_RInitiate(p) \equiv S_p = C \wedge (\forall q \in N_p :: S_q = C) \wedge Terminated(p)$
  $Root\_Ready\_RClean(p) \equiv S_p = B \wedge (\forall q \in N_p :: S_q = F) \wedge In\_RCycle(p)$
  $Ready\_RBroadcast(p) \equiv S_p = C \wedge S_{P_p} = B \wedge (\forall d \in CH_p :: S_d = C) \wedge In\_RCycle(P_p)$
  $Ready\_RFeedback(p) \equiv S_{P_p} = B \wedge In\_RCycle(P_p) \wedge (\forall d \in CH_p :: S_d = F)$
  $In\_RCycle(p) \equiv W_p = R$

Macros:
  $RInitiate_p \equiv S_p := B, Reset\_Values_p, Join\_RCycle_p$
  $Terminate\_Reset_p \equiv S_p := C, Dont\_Terminate_p, Update\_MinMax_p;$
  $RBroadcast_p \equiv S_p := B, Reset\_Values_p, Join\_RCycle_p;$
  $RFeedback_p \equiv$ if $In\_RCycle(p) \wedge S_p = B$
      then $S_p := F, Update\_MinMax_p;$
      else if $S_p = C \wedge (L_p = R_p = \bot)$
      then $S_p := F, Reset\_Values_p, Update\_MinMax_p;$
  $Reset\_Values_p \equiv$ if $OK_p \neq K_p \wedge Is\_SN(p)$ then $OK_p := K_p;$
  $Join\_RCycle_p \equiv W_p := R$
```

5.3 Algorithm $\mathcal{BST}$

Minor changes must also be made to Algorithm $\mathcal{SM}$ in order to incorporate the Termination Detection Scheme and Reset Scheme. in Algorithm 5.3.1, only Action SA_1 is modified in order to have the algorithm execute only when the Termination Detection Scheme is active (Predicate *In_TCycle*). We name the resulting algorithm Algorithm $\mathcal{BST}$.

Algorithm 5.3.1 Algorithm $\mathcal{BST}$

Actions:

```
SA_1  ::  $Potential\_Initiator(p) \wedge \neg Good\_Range(p) \wedge In\_TCycle(p)$  $\longrightarrow$  $SInitiate_p$
```

5.4 Correctness of Algorithm $\mathcal{BST}$

We first show that starting from any arbitrary configuration where the **root** is executing the TDS, we are guaranteed to start executing the RS in a finite amount of time. We then show that starting from any arbitrary configuration where the **root** is executing the RS, we are guaranteed to start executing the TDS in a finite amount of time.

Lemma 5.1 *Assume the* **root** *is in a clean state. If the* **root** *starts the TDS, the scheme will return a value of True to the* **root** *only when Algorithm $\mathcal{BST}$ has terminated.*

Proof. By the properties of the PFC scheme, since the **root** is clean, both of its children become clean and the broadcast phase of the PFC begins executing. All internal processors forward the broadcast wave to their children until the leaf processors are reached. In initiating the feedback phase, each leaf processor evaluates whether or not it is stable and has correct MinMax values. The leaf processors then forward the feedback wave and their response to their parent. Upon receiving the feedback wave, each internal processor evaluates whether or not it is stable, has correct MinMax values, and has received a value *True* from both its children. If the given internal processor returns *True* to all three conditions, it itself forwards *True* to its parent. On the other hand, if the given processor returns *False* to one of more of the conditions, it itself forwards *False* to its parent. The feedback wave eventually reaches the **root** which also evaluates the above conditions. By Theorem 4.4.1, Algorithm $\mathcal{BST}$ eventually terminates and so all processors eventually become stable and have correct MinMax values. Therefore, the TDS eventually returns *True*. □

Lemma 5.2 *Assume the* **root** *is in a clean state. If the* **root** *executes the RS, the scheme will reach all leaf processors and reset the system in one cycle.*

Proof. By the properties of the PFC scheme, since the **root** is clean, both of its children become clean and the broadcast phase of the PFC begins executing. At each processor in the tree, the key value held by that processor is reset and the broadcast is forwarded to the leaf processors. Once the leaf processors receive the broadcast wave, each resets its key value and initiates the feedback phase which is guaranteed to reach the **root**. Once the feedback wave reaches to the **root**, the RS is terminated and the **root** becomes clean again.

□

Lemma 5.3 *Starting from a arbitrary state where it is enabled at the* **root**, *the TDS eventually returns True and the RS is then enabled.*

Proof. By the specification of PFC, $S_{\mathbf{root}}$ can only be either B or C. There are two cases:

Case 1 : Assume that $S_{\mathbf{root}} = C$.

If $TD_{\mathbf{root}} = True$, the TDS has just terminated and the RS begins executing.

If $TD_{\mathbf{root}} = False$, by the properties of the PFC scheme, the **root** now begins executing a new TDS and that cycle is guaranteed to reach all the leaf processors in the tree and return another TD value of $True$ or $False$ to the **root**. If that value is $True$, the Reset scheme begins executing. By Lemma 5.1, the TDS eventually returns $True$ and therefore, the RS eventually begins executing.

Case 2 : Assume that $S_{\mathbf{root}} = B$.

By the properties of the PFC scheme, the TDS is guaranteed to return a TD value of $True$ or $False$ to the **root**. If that value is $True$, the RS begins executing. By Lemma 5.1, the TDS eventually returns $True$ since Algorithm BST terminates in a finite amount of time. Therefore, RS eventually begins executing.

□

Lemma 5.4 *Starting from an arbitrary state where it is enabled at the* **root**, *the RS eventually terminates and the TDS is then enabled.*

Proof. By the specification of PFC, $S_{\mathtt{root}}$ can only be either B or C. There are two cases:

Case 1 : Assume that $S_{\mathtt{root}} = C$. If $TD_{\mathtt{root}} = False$, the Reset scheme has just terminated and the TDS begins executing.

If $TD_{\mathtt{root}} = True$, by Lemma 5.2, the **root** now begins executing a new RS and that cycle is guaranteed to reach all leaf processors and return to the **root** where the variable TD's value is changed to $False$ and the TDS begins executing.

Case 2 : $S_{\mathtt{root}} = B$.

By the properties of the PFC scheme, The RS is guaranteed to return to the **root** where the variable TD's value is changed to $False$ and the TDS begins executing.

$\square$

Property 5.4.1 *At any arbitrary state either the RS or the TDS is enabled at the* **root**.

Theorem 5.4.1 *The output sequence resulting from the improved binary search tree maintenance algorithm is a permutation of the input sequence of the given binary tree once the second TDS has terminated.*

Proof. We begin with an arbitrary configuration. By Property 5.4.1, the **root** must either have the RS or the TDS enabled. We break our proof into two cases.

Case 1 : The RS is enabled at the **root**.

By Lemma 5.4, we are guaranteed that the RS will eventually terminate and the TDS will then be enabled. However, we are not guaranteed that the RS fully executed and so, that the key values in the tree are not corrupted. The first execution of the TDS executes and by Lemma 5.1, returns $True$ only once a correct binary search tree has been produced by Algorithm $\mathcal{BST}$. Since the values in the tree are not guaranteed to be correct based on the input environment's values, we cannot yet state the output sequence is in fact a permutation of the input sequence.

The **root** now executes a new RS. By Lemma 5.2, the tree will now be reset and will receive a sequence of new key values from the input environment. The TDS then begins executing and by Lemma 5.1, returns *True* only once a correct binary search tree has been produced by Algorithm $\mathcal{BST}$. Since both the RS and TDS have now fully executed, we are assured that the output sequence is in fact a permutation of the input sequence.

Case 2 : The TDS is enabled at the **root**.

By Lemma 5.3, we are guaranteed that the TDS will eventually terminate and the RS will then be enabled. However, we are not guaranteed that the TDS fully executed and so, that the resulting tree is a binary search tree. The **root** now executes a RS. By Lemma 5.2, the tree will now be reset and will receive a sequence of new key values from the input environment. The TDS then begins executing and by Lemma 5.1, returns *True* only once a correct binary search tree has been produced by Algorithm $\mathcal{BST}$. Since both the RS and TDS have now fully executed, we are assured that the output sequence is in fact a permutation of the input sequence.

□

5.5 Complexity of Algorithm $\mathcal{BST}$

In this section, we give an informal explanation of the complexity results for BST followed by a proof of complexity. Because both the TDS and the RS are both PFC schemes, we can state that the cost of each of their cycle is $2h$. By Lemma 4.5.1, the given binary tree will stabilize to a binary search tree in at most $O(hn)$ rounds. Since Algorithm $\mathcal{BST}$ and the Termination Detection scheme execute concurrently, the only cost added by the Termination Detection scheme is the final cycle it executes once Algorithm $\mathcal{BST}$ terminates which returns *True*. We now can state the following:

Property 5.5.1 *The TDS will return True in at most $O(hn) + 2h$ rounds.*

We are now ready to give the proof of complexity for the improved binary search tree algorithms.

Theorem 5.5.1 *The given binary tree will stabilize in $O(hn)$ rounds.*

Proof. By Theorem 5.4.1, the binary tree will stabilized once the second TDS has terminated. Starting at any arbitrary configuration, `root` is either executing the TDS or the RS. We break our proof in two cases:

Case 1 : The TDS is enabled at the `root`.

The `root` executes as follows: First the TDS executes, followed by the RS, and finally the second TDS.

We thus have the following cost: $(O(hn) + 2h) + (2h) + (O(hn) + 2h) = O(hn)$

Case 2 : The RS is enabled at the `root`.

The `root` executes as follows: The first RS executes, the first TDS then executes, followed by the RS, and finally the second TDS.

We thus have the following cost: $(2h) + (O(hn) + 2h) + (2h) + (O(hn) + 2h) = O(hn)$

Therefore, the given binary tree will stabilize in $O(hn)$ rounds. □

CHAPTER 6

CONCLUSIONS

In this thesis, we presented a self-stabilizing binary search tree maintenance algorithm on binary tree structures. Our final algorithm (Algorithm $\mathcal{BST}$) is the culmination of three algorithms: a search structure maintenance algorithm, a termination detection algorithm, and a reset algorithm.

We first introduced the search structure maintenace algorithm, called Algorithm $\mathcal{SM}$, to transform given a binary tree containing non-unique key values into a binary search tree and showed that its stabilization time is $O(hn)$. Because Algorithm $\mathcal{SM}$ does not meet the validity specification which states that the set of integers eventually sent to the output environment is a permutation of the integers received from the input environment, we then presented a termination detection scheme and a reset scheme. Both algorithms utilize the PFC paradigm. We finally showed that the added algorithms did not increase the cost of Algorithm $\mathcal{BST}$, and so our final binary search tree maintance algorithm stabilized in $O(hn)$ time units.

The algorithm discussed in this thesis is the first self-stabilizing binary search tree maintenance algorithm on binary tree structures. So, this work hopefully will lead to similar research in other search structures. The worst time needed by the proposed algorithm to build a binary search tree from an arbitrary binary tree structure is $2hn$ rounds. This compares very well with the corresponding sequential algorithm: Given an input sequence of n integers, it would take $4hn$ steps (in the worst case) to build a binary search tree. We are currently working on further improvement of the time complexity (less than $2hn$ rounds) by increasing the degree of concurrency.

BIBLIOGRAPHY

[AG93] A Arora and MG Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[BDPV99] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*. IEEE Computer Society Press, 1999. to appear.

[Cha82] EJH Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8:391–401, 1982.

[DGS96] S Dolev, M Gouda, and M Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[DIM97] S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.

[Dol00] S Dolev. *Self-stabilizing*. MIT Press, 2000.

[Gou98] MG Gouda. *Elements of network protocol design*. John Wiley & Sons, Inc., 1998.

[KP93] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[LAJ99] C Labovitz, A Ahuja, and F Jahanian. Experimental study of internet stability and wide-area network failures. In *Proceedings of FTCS99*, 1999.

[Seg83] A Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[Var94] G Varghese. Self-stabilization by counter flushing. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

VITA

Graduate College
University of Nevada, Las Vegas

Sylvain Ronan Brigant

Local Address:
1405 Vegas Valley Drive apt. 156
Las Vegas, NV 89109

Home Address:
8 Rue Stang Ar C'hoat
Quimper, 29000, France

Degrees:
Bachelor of Science, Computer Science, 1999
University of Nevada, Las Vegas

Thesis Title: Self-Stabilizing Binary Search Tree Maintenance Algorithm

Thesis Examination Comittee:
Chairperson, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Tom Nartker, Ph.D.
Graduate Faculty Representative, Dr. Henry Selvaraj, Ph.D.