Leaf-Line Tree:

A Data Structure for Efficiently Searching Dynamic Tree-Like Partial Orders

by

Marius Cătălin Iordan

Brent Heeringa, Advisor

A thesis submitted in partial fulfillment of the requirements for the Degree of Bachelor of Arts with Honors in Computer Science

Williams College
Williamstown, Massachusetts

May 22, 2009

Contents

1 Introduction **11**
1.1 Goal . . . 12
1.2 Contributions . . . 12
1.3 Thesis Overview . . . 13

2 Background **14**
2.1 Partial Orders and Posets . . . 14
2.2 Searching in Partially Ordered Sets . . . 15
2.3 The Static Problem for Tree-Like Posets . . . 17
2.3.1 Strategy Functions (Rankings) . . . 17
2.3.2 The Node Model . . . 18
2.3.3 The Edge Model . . . 19
2.4 The Dynamic Problem for Tree-Like Posets . . . 19

3 Central Elements in Posets **21**
3.1 The Central Element . . . 21
3.2 Central Elements and the Structure of Posets . . . 22

4 The Leaf-Line Tree **25**
4.1 Preliminaries and Definitions . . . 25
4.1.1 Optimal Search Strategy . . . 26
4.2 Query Model . . . 26
4.3 Construction . . . 27
4.3.1 Example . . . 29
4.3.2 Analysis . . . 30
4.4 Search . . . 34
4.4.1 Test Membership . . . 34
4.4.2 Find Location . . . 35
4.5 Neighbors . . . 35
4.6 Insertion . . . 37
4.6.1 Node Insertion . . . 38
4.6.2 Edge Insertion . . . 41
4.6.3 Analysis . . . 44
4.7 Deletion . . . 63
4.7.1 Deletion Procedure . . . 63
4.7.2 Restructuring Procedure . . . 64
4.7.3 Analysis . . . 67

5 Conclusions **77**
5.1 Contributions . . . 77
5.2 Future Work . . . 77
5.2.1 Asymptotical Optimality . . . 77
5.2.2 The Node Query Model . . . 78
5.2.3 Other Types of Partial Orders . . . 78

List of Figures

2.1 Examples of Hasse diagrams for a total order (a) and two posets (b)-(c). 15
2.2 Example poset graph G (i) with optimal node (ii) and edge rankings (iii). Edges are implicitly directed left-to-right, but this detail is not required for constructing either a node, or an edge strategy function following the algorithm described in [16]. 17
2.3 Optimal node (i) and edge decision trees (ii) for example graph G shown in Fig. 2.2 (i). 18
2.4 Poset P (left), a decision tree T for P (center), and the incorrect decision tree T' obtained by applying a standard balanced binary search tree rotation to T (right). . 20

3.1 Tree-like partial order categories: SOURCE (a), SINK (b), and CENTER (c). 22
3.2 Poset P which contains a complete bipartite subgraph: before (left) and after (right) inserting virtual central element V. 23

4.1 In addition to *closer to* x or *closer to* y, we allow an edge (x, y) to answer HERE when a node u falls between x and y in the partial order. Thus, given the partial order in (i) and an element u to be inserted, we can properly distinguish the partial orders given in (ii)-(iv). 27
4.2 Examples of a line contraction where we build a balanced binary search tree from a path (i) and of a leaf contraction where we build a linear search tree from a star node (ii). 28
4.3 Poset Q (i) undergoing repeated contraction steps and iterations of the construction algorithm (ii)-(v). 31
4.4 Leaf-Line tree associated with poset Q shown in Fig. 4.3 (i). The red nodes indicate impossible answers. 32
4.5 Node Insertion, Case 1.1 – B is the final node in the contraction and exactly 3 nodes survive after ROUND(B)−1 iterations: $B, A,$ and M : (i) poset structure after ROUND(B)−1 iterations before insertion, (ii) poset structure after ROUND(B)−1 iterations after insertion, (iii) poset structure after ROUND(M) iterations after insertion, (iv) data structure before insertion, (v) data structure after insertion. 47
4.6 Node Insertion, Case 1.2 – B is the final node in the contraction and either 2 or more than 3 nodes survive after ROUND(B)−1 iterations : (i) poset structure changes for ROUND(A) < ROUND(B), (ii) poset structure changes for ROUND(A) = ROUND(B), (iii) data structure changes. 48
4.7 Node Insertion, Case 2 – B is not the final node and ROUND(A) < ROUND(B) : (i) poset structure changes for TYPE(B) = LEAF and PARENT(B) = E, (ii) poset structure changes for TYPE(B) = LINE and PARENT(B) = (E, F), (iii) data structure changes. 49
4.8 Node Insertion, Case 3.1 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LEAF : (i) poset changes, (ii) data structure changes. 50
4.9 Node Insertion, Case 3.2.1 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B. 51

4.10 Node Insertion, Case 3.2.2 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B. 52
4.11 Node Insertion, Case 3.2.3 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B. 53
4.12 Node Insertion, Case 3.2.4 (a) – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B. 54
4.13 Node Insertion, Case 3.2.4 (b) – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B. 55
4.14 Node Insertion, Case 3.2.5 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B. 56
4.15 Edge Insertion, Case 1.1 Example – B is the final node, $\gamma > \alpha$ or $\gamma = \alpha$ and $\delta > \beta$ with ROUND(B) = k : (i) $\alpha < \gamma$ and $\alpha = \beta$, (ii) $\alpha < \gamma$ and $\alpha > \beta$, (iii) $\alpha = \gamma$, $\delta > \beta$, and $\alpha > \beta$. 57
4.16 Edge Insertion, Case 1.2 Example – B is the final node, $\alpha > \gamma$ or $\alpha = \gamma$ and $\beta \geq \delta$ with ROUND(B) = k : (i) $\alpha > \gamma$ and $\gamma = \delta$, (ii) $\alpha > \gamma$ and $\gamma > \delta$, (iii) $\alpha = \gamma$, $\beta > \delta$, and $\gamma > \delta$, (iv) $\alpha = \gamma = \beta = \delta$. 59
4.17 Edge Insertion, Case 2.1 Example – B is not the final node, TYPE(B) = LEAF, PARENT(B) = E, and A does not steal C_{line} : (i) if ROUND(A) $\leq$ ROUND(B) we perform a Node Insertion, and (ii) if ROUND(A) > ROUND(B) we insert B into edge (A, E). . 60
4.18 Edge Insertion, Case 3.1 Example – B is not the final node, TYPE(B) = LINE, PARENT(B) = (E, F), and A does not steal C_{left} or C_{right} : (i) if $\gamma = \delta$ we perform a Node Insertion, otherwise (ii) we insert A into edge (B, M). 61
4.19 Edge Insertion, Case 3.2 Example – B is not the final node, TYPE(B) = LINE, PARENT(B) = (E, F), and A steals C_{right}, but not C_{left} (WLOG) : (i) if ROUND(A) < ROUND(B) we insert A into edge (B, F), (ii) if ROUND(A) > ROUND(B) we insert B into edge (A, E), and (iii) if ROUND(A) = ROUND(B) we insert A into $BST(E, F)$. . 62
4.20 Deletion, Case 1.1 – ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LINE with PARENT(N) = (G, H) : (i) TYPE(D) = LEAF, D is not the final node, and N is not in C_{line} direction, (ii) TYPE(D) = LEAF, D is not the final node, and N is in the C_{line} direction, (iii) TYPE(D) = LINE and N is not in the C_{left} or C_{right} directions, (iv) TYPE(D) = LINE and N is in the C_{left} direction (WLOG), (v) D is the final node in the contraction process. 69
4.21 Deletion, Case 1.2 – ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LEAF with PARENT(N) = D : (i) TYPE(D) = LEAF and D is not the final node, (ii) TYPE(D) = LINE, (iii) D is the final node in the contraction process. 70
4.22 Deletion, Case 2 – ROUND(N) > ROUND(D) : (i) TYPE(D) = LEAF and PARENT(D) = N, (ii) TYPE(D) = LINE and PARENT(D) = (F, N). 71
4.23 Restructuring, Case 2 – TYPE(N) = LEAF, PARENT(N) = E, and $\alpha = \beta$ OR TYPE(N) = LINE, PARENT(N) = (E, F), and $\alpha = k - 1$: (i) assigning ROUND(N) = k for TYPE(N) = LEAF, (ii) assigning ROUND(N) = k for TYPE(N) = LINE. 72
4.24 Restructuring, Case 3.1 – TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration k 73
4.25 Restructuring, Case 3.2 – TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration $k - 1$ 74
4.26 Restructuring, Case 3.3 – TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration $i < k - 1$ 75
4.27 Restructuring, Case 4 – TYPE(N) = LINE, PARENT(N) = (E, F), $\alpha < k - 1$, L and R as defined below.: (i) $L \neq F$ and $R \neq E$, (ii) $L \neq F$ and $R = E$ (WLOG), (iii) $L = F$ and $R = E$ 76

Abstract

Identifying an optimal search strategy for a given set and maintaining its efficiency under dynamic operations (i.e. insertion and deletion) is a fundamental problem in data stuctures. The classical variant of this problem considers underlying sets which are total orders. In this case, the problem can be solved optimally by balanced binary search trees such as AVL trees and Red-Black trees. We consider the related problem where the structure of the underlying set is relaxed to a tree-like partial order. Recent results have shown that an optimal search strategy can be built in linear time for such partial orders in the static case. However, due to structural properties inherent to partial orders, such solutions cannot easily be extended to the dynamic case of the problem. Previous to the work described in this thesis, the dynamic setting of the problem has not yet been investigated.

We present the Leaf-Line Tree, a data structure that supports efficient *insert*, *delete*, *test membership*, and *neighbor* operations on a set S of n elements drawn from a partial order whose underlying Hasse diagram is a tree. The data structure requires $O(n)$ time and space to be built initially and each of the operations above requires at most a factor of $O(\log\ w)$ more steps than the optimal number in the worst case. Here w is the width of the poset (w is bounded above by n).

Acknowledgments

First and foremost, thanks is due to my advisor, Brent Heeringa, for his mentorship and constant encouragement, not only while writing my thesis, but for the two years before that as well. Brent was invaluable in cultivating my love for theoretical computer science. For showing me how to take the first steps into this field and instilling in me his own dedication to great research and clarity of explanation, I would like to express my deepest gratitude.

I would also like to thank all members of the Computer Science department for being great people, as well as terrific mentors and attentive teachers. I want to especially mention professors Morgan McGuire, my second reader, Stephen Freund, and Jeannie Albrecht for providing me with great experiences both within and outside the classroom, and for their unwavering support throughout my Williams career.

I am very grateful to everyone outside the department, as well, for their patience and encouragement as I devoted a great amount of time and effort to my thesis. Most importantly, I want to thank Elise Piazza for her kindness and patience and for always being there for me throughout my time here. Finally, I would like to thank my family for their encouragement and support, for believing in me, and for always being with me in spirit, if not in person.

All of you mentioned above have shaped my life immensely and for that you have my deepest gratitude and thanks.

Chapter 1

Introduction

Maintaining a dynamic set S of n elements drawn from a universe $\mathcal{U}$ of $m \gg n$ elements is a fundamental problem in data structures. When S is totally ordered, the operations

- insert,
- delete,
- test membership, and
- neighbor

are all supported in $O(\log\ n)$ time with classical balanced search trees like AVL trees and Red-Black trees [9].

In this thesis, we address the problem when S is partially ordered. One way to view a totally ordered set S is as a line where element x appears to the left of element y if and only if $x < y$. Similarly, a partial order can be described by a directed acyclic graph, called its Hasse diagram. Here two nodes are comparable if there is a directed path from one node to the other. However, nodes may also be incomparable if no such path exists.

Given a node x from a totally ordered universe, one can test whether x is in S by binary searching the line representing S. This search strategy defines an optimal binary search tree for S. However, finding an optimal binary search tree for a general partial order is NP-hard [2]. Thus, our results focus on partial orders whose underlying universe forms a directed tree. We call these tree-like partial orders. Searching in tree-like partial orders has been studied extensively [1, 7, 8, 10, 11, 12, 14, 15, 16, 17, 18, 20]. Recent results have shown that, in contrast to directed acyclic graphs, one can find an optimal binary search strategy for trees in linear time [15].

Just as balanced binary search trees maintain asymptotic optimality for a dynamic set that is totally ordered, we are interested in maintaining asymptotic optimality for a dynamic set when S is partially ordered. Presently, the only viable solution to the problem of searching in dynamical partial orders is to recompute the optimal search strategy after each change to the underlying set. This operation is very expensive: it takes linear time in the size of the set. Our work provides the first *efficient* solution to the dynamic setting of the problem.

A reasonable place to start our investigation is by attempting to find decision tree restructuring strategies similar to the rotations which allow asymptotically optimal balanced binary tree restructuring (i.e. similar to AVL trees and Red-Black trees). However, as we show in Section 2.4, we cannot straightforwardly apply the same transformations to the optimal partial order search trees from [15] as we would to the total order structures. This leads to invalid search trees because partial order elements may be incomparable. Furthermore, it is unclear how to redefine the rotation rules in order to achieve both (1) a similar performance guarantee and (2) valid dynamic balanced search trees for partial orders. It is also unclear how one could adapt other search structures proposed in the literature which claim weaker than optimal bounds on efficiency (see for example [11]) to maintain their guarantees under dynamic operations.

1.1 Goal

Our goal is succinctly stated as follows:

> *Build a data structure that supports efficient* insert*,* delete*,* test membership*, and* neighbor *operations for a dynamic set of elements drawn from a tree-like partial order.*

Motivating our work are practical problems in filesystem synchronization and software testing described by Ben-Asher *et al.* [1]. Further motivating this goal are applications related to rankings in sports, college admissions, and conference submissions. Other areas quoted in [4] where advances in solving this problem would have marked impact are comparing the evolutionary fitness of different strains of bacteria, understanding input-output relations among a set of metabolic reactions, or understanding causal influences among a set of interacting genes and proteins.

In the first scenario, two copies of a filesystem are maintained on separate machines. One copy is the source and the other copy is a mirror. Changes to the source are propagated to the mirror. Following some communication failure, the two filesystems may lose synchronization. To find the differences, one might test directory checksums at the root of each filesystem, then follow differing checksums in subdirectories to isolate the problems. However, since some subdirectories extend much deeper than others, it might be better to start searching at the root of a subdirectory rather than the root of the entire filesystem. Furthermore, if network communication is costly, then minimizing the number of checksum tests becomes the primary goal when performing the search. This minimization problem naturally maps to the problem of finding optimal search strategies in tree-like partial orders. If the filesystem is quite large, then the time required to find the optimal static search strategy may be somewhat prohibitive – maintaining a dynamic search strategy with respect to changes in the filesystem may be more beneficial.

1.2 Contributions

Our primary contribution is the Leaf-Line tree, a data structure that supports efficient *insert*, *delete*, *test membership*, and *neighbor* operations on a set S of n elements drawn from a partial order whose

underlying Hasse diagram is a tree. We assume pointer-based Hasse diagrams and constant-time comparisons between elements in the partial order [4]. The Leaf-Line tree is space efficient – it requires $O(n)$ space where n is the size of the dynamic set – and can be built in $O(n)$ time given a Hasse diagram of the poset. Each operation takes time $OPT \cdot O(\log\ w)$ where w is the *width* of the partial order. The *width* represents a natural obstacle when searching in a partial order.

Leaf-Line Tree is essentially a decision tree which can be updated in $OPT \cdot O(\log\ w)$ steps, guaranteeing better than linear performance *whenever possible*. The reader should note that the optimal search strategy itself might be linear in the number of elements of the partial order considered. In that case, a linear time penalty cannot be avoided when performing any operation on the search strategy (such as search, insert, or delete) as $O(n)$ is the information theoretic lower bound on the time needed to fully disambiguate all elements of the partial order [4].

1.3 Thesis Overview

We present our work as follows:

Chapter 2 is an overview of related work on searching in general and tree-like partial orders.

Chapter 3 describes how any subset P drawn from a partially ordered universe $\mathcal{U}$ always forms a directed tree provided that P contains a special *central* element.

Chapter 4 describes the Leaf-Line Tree data structure and proves it maintains an efficient search strategy for dynamic tree-like partially ordered sets.

Chapter 5 provides an overview of our accomplishments and directions for future work.

Chapter 2

Background

2.1 Partial Orders and Posets

A partial order is a binary relation '$\leq$' over a set $\mathcal{U}$ (called the *universe*) which is reflexive, antisymmetric, and transitive, i.e., for all a, b, and c in $\mathcal{U}$, we have that:

- $a \leq a$ (reflexivity)
- if $a \leq b$ and $b \leq a$, then $a = b$ (antisymmetry)
- if $a \leq b$ and $b \leq c$, then $a \leq c$ (transitivity)

A partially ordered set (poset) is a subset of n items drawn from a universe $\mathcal{U}$ of $m \gg n$ elements, together with a partial order defined on $\mathcal{U}$ that describes, for certain pairs of elements in the set, the requirement that one of the elements must precede the other. A partially ordered set formalizes the intuitive concept of an ordering, sequencing, or arrangement of the elements of a set. However, a poset differs from a total order in that some pairs of elements in the poset are incomparable.

Definition 2.1. Two elements x, y in a poset P are *comparable* if $x \leq y$ or $y \leq x$. If x and y are not comparable, we say they are *incomparable*.

Thus, while the Hasse diagram of a total ordering is a line (all elements to the left of a position are smaller and all elements to the right are greater), the Hasse diagram for a poset is a directed acyclic graph where the nodes signify the elements of the underlying set S and the edges denote the binary relation between two adjacent elements (for a more formal discussion of partial orders see [19]).

Equivalently, if the directed acyclic graph G is the Hasse diagram of P, then two nodes A, B in G are comparable if a directed path exists between A and B or vice versa. This statement follows from the the transitivity of partial orders. We give examples of this type of structure in Fig. 2.1.

The *width* of a partial order is a very useful measure of its complexity. We provide the following definition adapted from [4].

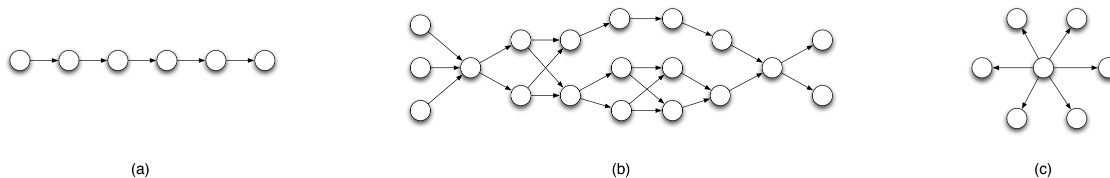

Figure 2.1: Examples of Hasse diagrams for a total order (a) and two posets (b)-(c).

Definition 2.2. The *width* of a partially ordered set P is the maximum size of a subset of mutually incomparable elements from P.

A totally ordered set has width 1 since all elements are comparable (Fig. 2.1 (a)). The poset in Fig. 2.1 (b) has 19 elements and width 3. When the Hasse diagram of a poset with n elements forms a star, the width reaches its maximum value of $n-1$. In this case, the poset graph consists of a root node with directed edges from it to every other node (or from every other node to it), as shown in Fig. 2.1 (c). The root is comparable to every other node, but the leaves are all incomparable to one another, hence the stated width.

When analyzing the efficiency and complexity of algorithms that solve some problem about a poset P, we will use the following notions of complexity adapted from [4].

Definition 2.3. For a poset algorithm we define *query complexity* as the number of disambiguation questions (or comparisons of the form $u \leq x$) asked during the run of the algorithm and *total complexity* as the total number of operations performed by the most efficient implementation of the algorithm.

2.2 Searching in Partially Ordered Sets

Classical problems involving search limit their underlying space to linear orderings of elements. For a totally ordered set with n elements the optimal search strategy is binary search (see [10]). Thus, one can find an element in $O(\log\ n)$ time and one can build the binary search tree in $O(n)$ time. For partial orders, the best known search strategies correspond to binary decision trees capable of unequivocally separating all elements of a poset P . Each leaf in a decision tree is an element of P and each internal node is a question of the form $u \leq x$, where $x \in S$ and u is the element under consideration. If the answer to the question is YES then u truly is less than or equal to x and the search continues down the right branch of the decision tree. If the answer is NO then either $u > x$ or u and x are *incomparable* and the search continues down the left branch of the decision tree. In this way, each element u follows a single path in the decision tree from the root down to a leaf.

The height of a decision tree is the length of the longest root-to-leaf path. Said differently, it is the maximum number of questions, in the worst-case, that one needs to ask about an element in order to determine its place in P. A search tree is optimal for P if there is no other decision tree with smaller height.

Given a partial order $\leq$ over a set $\mathcal{U}$ and a poset $P \subseteq \mathcal{U}$ of size n and width w, we identify three natural problems: the *sorting problem*, the *static problem*, and the *dynamic problem*.

(1) Sorting Problem: *Given P and $\leq$, build the Hasse diagram for P*

This problem was first described by Faigle and Turán in [5] as "sorting a poset" and the authors give two solutions in the form of algorithms for sorting posets. Each of these algorithms has query complexity $O(wn \log \frac{n}{w})$, but their total complexity in not analyzed and it is unclear whether they can be implemented in polynomial time.

More recently, Daskalakis *et al.* give an algorithm that requires only $O(wn + n \ \log \ n)$ queries in the worst case for sorting a poset (which is within a constant factor of the information theoretical lower bound) [4]. Daskalakis *et al.* also propose a method of building the Hasse diagram of a poset P based on a variant of MergeSort which matches the query complexity of those put forward by Faigle and Turán, but who additionally gives a provable upper bound of $O(w^2 n \log \frac{n}{w})$ on the total complexity of the problem (showing it is indeed solvable in polynomial time). The exact total complexity of the problem remains unknown.

In this thesis, we will not address the *sorting problem*, but rather attempt to (a) extend the applicability of existing results for the *static problem* to a larger class of partial orders and (b) provide a new data structure that solves the *dynamic problem* for a large class of partial orders.

(2) Static Problem: *Given the Hasse diagram for P, build an optimal search tree for P*

A recent result by Carmo *et al.* shows that for general partial orders, constructing such a decision tree is NP-hard [2], and thus there is no known polynomial-time algorithm for building an optimal search strategy.

Due to this limitation, one direction of research has focused on algorithms yielding search strategies which are not optimal, but provably close to optimal. The most recent and tightly bounded approximations to the problem are provided again by Carmo *et al.*, who proposed polynomial-time algorithms for constructing a $(1 + o(1))$-approximation for typical partial orders under the *random graph* model and a 6.34-approximation for partial orders under the *uniform* model. In the *random graph* model, a partial order is built on a set of n nodes by independently selecting partial order relations between pairs of elements from the set according to a fixed probability $0 \leq p \leq 1$. Under such considerations, the authors prove that *almost surely* the height of the binary search tree they produce is at most $lg\ n + O(\sqrt{\log n} \log \log n)$. In the *uniform* model, a partial order is chosen with uniform probability from the set of all possible partial orders on a set of n elements. This assumption yields the construction of a ternary search tree of height *almost surely* at most $6.34 \log_3 n$.

The bounds found by Carmo *et al.* in both the random graph and the uniform models above are probabilistic and, furthermore, they provide only results pertaining to query complexity and no visible estimation of the running time of their algorithm, other than stating polynomial-time efficiency. In real-world examples, the cost of producing such a tree may outweigh the benefits of obtaining a solution within a constant factor of the optimal, and furthermore, since we have no knowledge of the underlying distribution of the partial order, this probabilistic approach leaves open the question of providing an efficient search strategy for good worst-case performance on all possible posets, instead of *almost all* posets.

(3) Dynamic Problem: *Given an optimal search tree for P, maintain its optimality under dynamical operations*

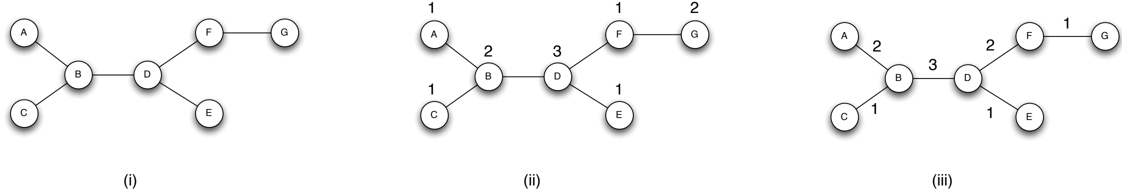

Figure 2.2: Example poset graph G (i) with optimal node (ii) and edge rankings (iii). Edges are implicitly directed left-to-right, but this detail is not required for constructing either a node, or an edge strategy function following the algorithm described in [16].

Intuitively, there should exist data structures for posets analogous to those which allow near-optimal total order dynamic decision tree restructuring (red-black trees, AVL trees, skip lists, etc. [3], [6]).

However, the dynamic setting of the problem of searching in partial orders has not been investigated in the literature thus far. Most publications on posets do not make any reference to this question, while others simply mention it as “an interesting open problem” [4]. Therefore, no results analogous to the data structures mentioned above yet exist for this problem, other than the naïve reconstruction of the static case poset decision tree after every update operation.

2.3 The Static Problem for Tree-Like Posets

Since searching in partially ordered sets is an NP-complete problem, a large amount of research has been directed towards the problem of constructing efficient optimal solutions in posets with special structural characteristics.

The most common simplification investigated in the literature is the restriction of the problem from posets with general directed acyclic graph representations to posets whose Hasse diagram is a tree (Fig. 2.2 (i)). In this case, an optimal search strategy remains a minimum height decision tree separating every pair of elements in the poset (a tree in which the worst-case number of questions is the lowest) (see [16] and Fig. 2.3).

As with general partially ordered sets, research efforts have focused exclusively on the *static problem*. The key intermediary step to finding an optimal solution in this case is defining a strategy function on the poset graph.

2.3.1 Strategy Functions (Rankings)

The prevalent approach to constructing an efficient search strategy for a tree-like partial order involves defining a strategy function (or ranking) on either the nodes or the edges of the partial order tree. The ranking can subsequently be used to generate a valid optimal or near-optimal decision tree for the specified poset.

We will use the following definition adapted from [16].

Definition 2.4. Let $G = (C_1, C_2)$ be the graph representation of a poset P, where C_1 is the set of vertices and C_2 is the set of edges. Choose $i \in \{1, 2\}$. Then, a function $f : C_i \rightarrow \mathbb{Z}_+$ is a

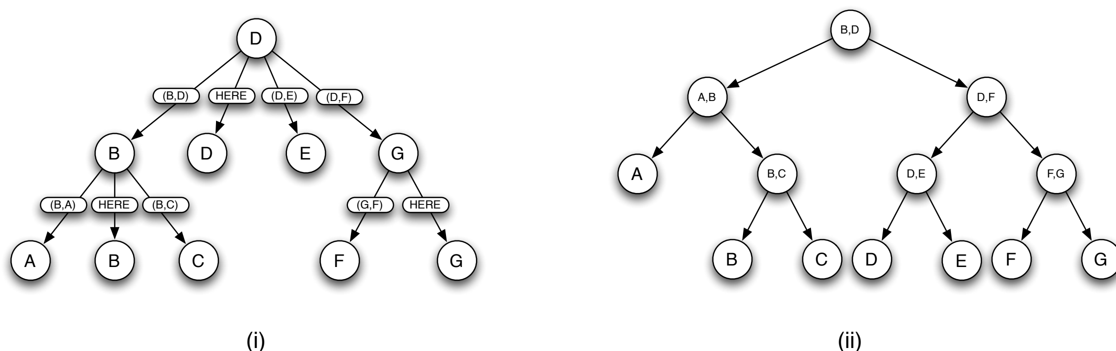

Figure 2.3: Optimal node (i) and edge decision trees (ii) for example graph G shown in Fig. 2.2 (i).

strategy function if for any distinct elements $x, y \in C_i$, if $f(x) = f(y)$, then there exists $z \in C_i$ on the simple path from x to y such that $f(z) > f(x)$.

This definition implies that if two distinct vertices (edges) are assigned the same value by a strategy function, then there must be another vertex (edge) on the simple path between the first two which is assigned a greater value by the same strategy function. Thus, for a node strategy function (Fig. 2.2 (ii)), if vertices B and G are assigned the same value ($f(B) = f(G) = 2$), then there exists a higher value vertex D with $f(D) = 3$ on the path from B to G. Similarly, for an edge strategy function (Fig. 2.2 (iii)), if edges (F, G) and (D, E) are assigned the same value ($f(F, G) = f(D, E) = 1$), then there exists a higher value edge (D, F) with $f(D, F) = 2$ on the path from (F, G) to (D, E).

This notion of a strategy function (ranking) is very important due to a result stated by Mozes *et al.* in [15] (proven in a modified form in [16] by Onak and Parys), which is summarized in the following Lemma.

Lemma 2.5. For every tree T, the worst-case number of queries in an optimal searching strategy in T equals the lowest possible maximum of a strategy function on T.

In the next two subsections we give a short overview of the edge and node query models. Depending on which model one chooses, the questions asked in the decision tree for the partial order will be either nodes or edges from the graph representation of the poset (see Fig. 2.3).

2.3.2 The Node Model

In the node model, the goal is to find an element in the poset and we are allowed to ask questions only about vertices (see Fig. 2.2 (ii) and Fig. 2.3 (i)). For each vertex v in the poset graph, the possible answers are:

- v is the target
- the outgoing edge from v which starts the simple path from v to the target

A polynomial-time algorithm for finding an optimal node ranking (strategy function) in a tree was first given in [7] by Iyer *et al.*, whose solution ran in $O(n \log n)$-time. This result was then improved by Schaffer in [18] who gave a linear time solution to the problem.

Finally, Onak and Parys [16] gave an alternative linear time solution to the one given by Schaffer and further showed how this ranking can be used to construct a node model decision tree for the partial order in linear time (Fig. 2.3 (i)).

2.3.3 The Edge Model

In the edge model, the goal is to find an element in the poset and we are allowed to ask questions only about edges (see Fig. 2.2 (iii) and Fig. 2.3 (ii)). For each edge $e = (x, y)$ in the poset graph, the possible answers are:

- vertex x is closer to the target (or may be the target)
- vertex y is closer to the target (or may be the target)

The first $O(n^2)$-time algorithm for computing a strategy function for the edge model was given by Zhou and Nishizeki, who subsequently improved their own bound to $O(n \log n)$-time in [20]. The first linear time construction of a ranking was given by Lam and Yue in [12], and it was proven in [1] that constructing an optimal search decision tree for the edge model (Fig. 2.3 (ii)) supports a polynomial-time solution, which was first identified as $O(n^4 \log^3 n)$ and subsequently refined to $O(n^3)$ in [16] , and finally to its lower theoretical bound of $O(n)$ in [15] .

For the remainder of this thesis, when we refer to the problem of searching in partial orders, we are implicitly assuming the edge query model.

2.4 The Dynamic Problem for Tree-Like Posets

All previous work on searching in tree-like partial orders focuses on the static version of the problem where the partial order does not support insertions or deletions. Therefore, no data structures yet exist that avoid the overhead of rebuilding the decision tree after every update for the dynamic setting.

Just as binary search trees for a fixed total order are the starting point for balanced binary search trees, one might hope to maintain a dynamic partial order by directly updating the optimal search trees for a fixed partial order.

However, applying natural rotations to the decision tree for a poset P does not always yield another valid decision tree in the context of P. Consider the poset P with decision tree T shown in Fig. 2.4. Notice that node (A, B) contains an imbalance in the height of its children: (A, C) has height 3, while B has height 1. Applying an ordinary AVL or Red-Black tree rotation operation to T yields tree T' of strictly smaller height. Suppose we want to search for node B using our new decision tree T'. From Section 2.3.3 we expect that by asking the first question (A, C) we descend down the branch corresponding to the endpoint closest to B in P, which is A. Unfortunately, B cannot be found in the subtree rooted at (A, D) and the search fails.

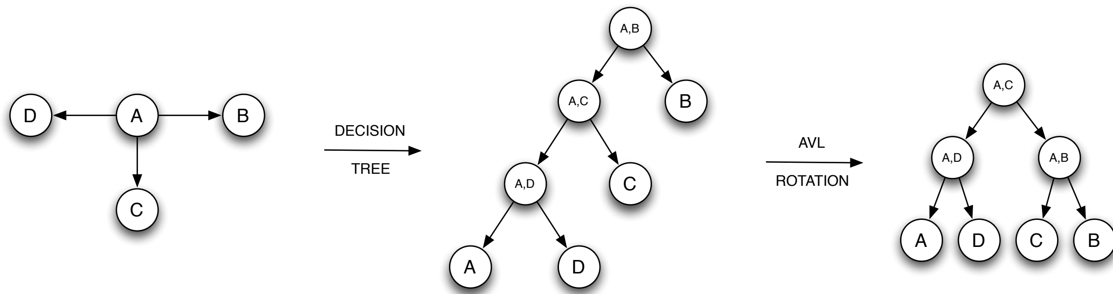

Figure 2.4: Poset P (left), a decision tree T for P (center), and the incorrect decision tree T' obtained by applying a standard balanced binary search tree rotation to T (right).

Thus, applying standard rotation operations to partial order decision trees in order to decrease their height in the event of an imbalance no longer makes sense given the relationships between elements in the poset. Furthermore, it is unclear how to extend or redefine the notion of *optimality maintaining rotations* to partially ordered set decision trees, so directly applying these ideas to our problem seems difficult. Because of this, the Leaf-Line Tree data structure we propose in Chapter 4 offers a substantially different approach to the problem than those given by previous work in the static case.

Chapter 3

Central Elements in Posets

In this chapter, we show that any subset P of n elements drawn from a partially ordered universe $\mathcal{U}$ of $m \gg n$ elements always forms a directed tree provided that subset contains a *central* element.

We first introduce the notion of a *central* element of a partial order and then show how always keeping at least one such node in the collection (either as a real node if it already exists, or as a *virtual* node if it doesn't) ensures that P is a tree-like poset.

3.1 The Central Element

We begin with a definition.

Definition 3.1. A node $x \in P$ is a **central element** if every other element in P is either reachable from x, or can reach x, but not both.

The existence of a central element in every poset was originally shown in [13]. Onak and Parys [16] also describe how all tree-like partial orders P contain at least one central element E and can be classified into three distinct categories based on the properties of this element (Fig. 3.1):

SOURCE – E is the smallest element in P and acts as a source (all nodes are reachable from E).

SINK – E is the largest element in P and acts as a sink (E is reachable from every node).

CENTER – E is a central element, but is neither a source, nor a sink for nodes in P.

Note that for partial orders of type CENTER, the central element need not be unique. In Fig. 3.1 (c), both E and F are considered central elements given the definition above.

Lemma 3.2. Let P be a poset of type CENTER with $g > 2$ central elements where E is the smallest central element and F is the largest central element. If $C \in P$ such that $E < C < F$, then C is a central element. Furthermore, C has degree exactly 2.

Proof. If $A, B \in P$ are central elements, then A and B are comparable by Definition 3.1. Thus, all central elements lie on the same path in P.

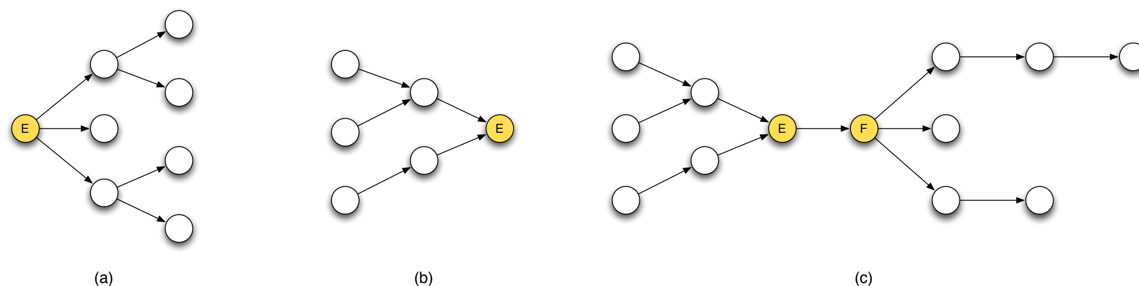

Figure 3.1: Tree-like partial order categories: SOURCE (a), SINK (b), and CENTER (c).

Suppose $C > E$ for some $C \in P$. It follows that C and F are comparable since F is a central element. If $C < F$, then C must lie on the path between E and F (since P is a tree). Now suppose $C < F$ for some $C \in P$. Similarly, it follows that C and E are comparable and if $C > E$, then C must lie on the path between E and F.

Let $\{E, C_1, \ldots, C_k, F\}$ be the path between E and F containing all elements $E \leq C_i \leq F$. Choose C_i. Then there exists a path from all elements $H < E$ to C_i through E. There is also a path from C_i to all elements $H > F$ through F. Since all C_i are comparable, it follows that all C_i are central elements.

Finally, suppose that some C_i had degree greater than 2. Then there exists $H \in P$ such that either $H > C_i$ and H is not comparable to F or $H < C_i$ and H is not comparable to E. Since E and F are central elements, this is a contradiction. □

3.2 Central Elements and the Structure of Posets

Tree-like partially ordered sets always contain a central element, while some arbitrary subset S of a tree-like universe $\mathcal{U}$ may not. We provide the following Lemma summarizing our claim.

Lemma 3.3. Let P be a subset of a tree-like universe $\mathcal{U}$. P is a tree-like partially ordered set if and only if P contains at least one central element.

Proof. Suppose P is of type SOURCE (Fig. 3.1, (a)). By eliminating the sole central element E, we obtain a forest of disconnected trees, each rooted at the children of node E.

Similarly, if P is of type SINK (Fig. 3.1, (b)), then by eliminating the sole central element E, we obtain a forest of disconnected trees, each rooted at the neighbors of node E.

Suppose $P \subseteq \mathcal{U}$ is of type CENTER (Fig. 3.1, (c)) and contains no central elements. Let E be the smallest central element and F be the largest central element in $\mathcal{U}$. Then by Lemma 3.2 for all elements $C \in P$, either $C < E$ or $C > F$. If all $C < E$ or all $C > F$, then we have an analogous problem to when P is of type SINK or SOURCE, respectively. If P contains at most one element C such that either $C < E$ or $C > F$, then C would be a sink or a source for P which is a contradiction. Finally, if at least 2 element $C < E$, and at least 2 elements $C > F$, then P must contain a complete bipartite subgraph as shown in Fig. 3.2.

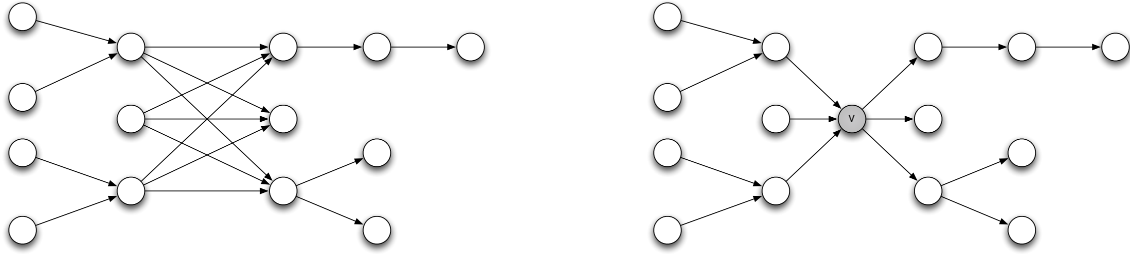

Figure 3.2: Poset P which contains a complete bipartite subgraph: before (left) and after (right) inserting virtual central element V.

Thus, if P contains no central elements of U, then P is not a tree-like partially ordered set and this concludes our proof.

□

Corollary 3.4. Let P be a subset of a tree-like universe $\mathcal{U}$. If P is not tree-like, then we can obtain a tree-like partially ordered set from P by adding exactly one *virtual* central element to P.

Proof. If P is not tree-like, then by the previous lemma the Hasse diagram for P is either a forest of disconnected trees or contains a complete bipartite subgraph.

Suppose that the Hasse diagram for P is a forest of disconnected trees. If $\mathcal{U}$ is of type Source (Fig. 3.1, (a)) then P doesn't contain the central element. We subsequently add the central element E to P as a virtual node. Now, all formerly disjoint trees in P have a common ancestor and are now connected to the root node E.

If $\mathcal{U}$ is of type Sink (Fig. 3.1, (b)), then the proof is analogous to the Source case detailed above (except all elements now have a common sink node, rather than a common ancestor).

If $\mathcal{U}$ is of type Center (Fig. 3.1, (c)). We subsequently add a virtual central element to P. Now, all formerly disjoint trees in P have either a common ancestor or a common sink node and are thus connected.

Finally, suppose that the Hasse diagram for P contains a complete bipartite subgraph. Then $\mathcal{U}$ must be of type Center by previous lemma. Furthermore, P does not contain any of the central elements of $\mathcal{U}$. We subsequently add a virtual central element to P. This new element is concurrently larger than all nodes on the left side of the bipartite subgraph and smaller than all nodes on the right. The new poset we obtain is tree like and no longer contains a bipartite subgraph (Fig. 3.2, right). □

The best known algorithms for searching in forests of tree-like partial orders [16] must incur the cost of disambiguating between each connected component of the forest. This operation takes linear time in the number of components. However, by including exactly one virtual central element, we obtain a single tree-like poset from the forest. We can take advantage of the newly created structure to generate an asymptotically optimal decision tree for the original poset.

Moreover, no efficient algorithms exist to search a poset which contains a complete bipartite subgraph. By including a virtual central element, we construct a new tree-like poset which is a

superset of the initial poset, but contains exactly one additional element. Again, we can use efficient search algorithms for tree-like posets to perform an asymptotically optimal search of a previously intractable structure.

For implementation purposes, we can identify the central elements of a poset of size n (or the lack thereof) in $O(n)$ time. For Source and Sink posets, the central element is the smallest (respectively the largest) element. For Center posets, the central elements are a collection of consecutive elements such that the smallest has non-zero in-degree and the largest has non-zero out-degree, with all elements in-between having in- and out-degree exactly 1. When managing a dynamical tree-like poset, if we try to delete the last central element from our working set, we then immediately replace it with a *virtual* node. This node is simply a placeholder and a true central element would take its place as soon as one in reinserted into the poset. The sole purpose of the virtual node is to allow us to approximate the current set of elements with a tree-like partially ordered set.

Most importantly, in order to extend a subset of a tree-like universe to a tree-like poset we never require knowledge about the actual shape and structure of underlying universe. The only information we need is whether the universe is tree-like. This immediately implies that the universe contains at least one central element and so our extension method is applicable to any of its non-tree-like subsets [13].

Chapter 4

The Leaf-Line Tree

In this chapter we present the Leaf-Line tree, a data structure that supports efficient *insert*, *delete*, *test membership*, and *neighbor* operations on a set S of n elements drawn from a partial order whose underlying Hasse diagram is a tree. We build the Leaf-Line tree by defining a recursive contraction process on the Hasse diagram and incrementally creating specialized search structures for the portions of the graph we contract. These structures are subsequently joined together to form the final search tree for the set S.

4.1 Preliminaries and Definitions

For the remainder of this chapter we will assume that $P \subseteq \mathcal{U}$ is a tree-like partially ordered set of size n, maximum node degree d, and width w. We assume that P is tree-like since we only need at most one virtual central element to be added to P in order to ensure this fact by Corollary 3.4.

A poset P is given by its pointer-based Hasse diagram. Although the poset tree contains directed edges, we will ignore this fact throughout this chapter. For our purposes we only need to remark that in a tree there exists a unique path from a node to any other node, disregarding edge orientation. We can further assume for simplicity that P is represented as a rooted tree of type Source and that the width w is equal to the number of leaves. In this case, a leaf is a node of degree 1 . We will use the notation P interchangeably for both the poset and its undirected Hasse diagram.

Before we describe how to build the data structure from the Hasse diagram, we need to prove a few preliminary results. Consider the following definition.

Definition 4.1. An element $x \in P$ is a **star node** if it has degree at least 3.

We will subsequently prove a very important relationship between the width of a tree-like poset and the number of star nodes in its Hasse diagram.

Theorem 4.2. Let S be the total number of star nodes in P. Then $S \leq w - 1$.

Proof. All non-empty trees have at least one leaf. Since P is a rooted tree, each node except for the root has in-degree exactly 1. Thus each star node has out-degree at least 2. For each node x,

each outgoing edge from x contains at least one leaf. Thus, for every star node in P, the number of leaves grows by at least 1. Since the number of leaves is equal to w we have that $S \leq w - 1$. □

4.1.1 Optimal Search Strategy

In order to prove the efficiency of our search algorithms, we must first establish lower bounds on the query complexity of the optimal search strategy. We begin by providing the following definition.

Definition 4.3. A **chain** $C \subseteq P$ is a subset of mutually comparable elements. The **longest chain** (denoted henceforth by $\mathcal{C}$) is the chain in P with the largest number of elements.

We are now ready to establish a lower bound on the number of queries required by the optimal search strategy in the worst case.

Lemma 4.4. Let OPT be the query complexity of the optimal search strategy. Then $OPT \geq \max\{d; \log n\}$.

Proof. Let x be a node of highest degree d in P. Then, to find x in the poset we require at least d queries, one for each edge adjacent to x by [11]. This implies $OPT \geq d$. Also, since querying any edge reduces the problem space left to search by at most a half, we infer that $OPT \geq \log n$. We conclude that $OPT \geq \max\{d; \log n\}$. □

Corollary 4.5. $OPT \geq \max\{\log C; \log w\}$.

Proof. This follows immediately from the fact that n is an upper bound on both w and $\mathcal{C}$. □

4.2 Query Model

Recall that a search tree for P is a binary decision tree that, for all $u \in \mathcal{U}$, determines if u is in P, and if not, determines where u would appear in P. Thus, each leaf in the decision tree is an element of P and each internal node of the decision tree is a question of the form $u \leq x$ where $x \in P$ and $u \in \mathcal{U}$ is the element under consideration. If the answer to the question is YES then u truly is less than or equal to x and the search continues down the right branch of the decision tree. If the answer is NO then either $u > x$ or u and x are incomparable and the search continues down the left branch of the decision tree. In this way, each element u follows a single path in the decision tree from the root down to a leaf.

Because our trees are not oriented, we have no notion of $\leq$ so we consider questions on edges: given an edge (x, y) and an element u we ask: *is u is closer to x or to y?* With rooted, directed trees, asking a question of the form $u \leq y$ is identical to asking the incoming edge (x, y) of y whether u is closer to x or to y in the tree. However, because we are dealing with dynamic sets, u may actually fall between x and y, which presents a problem for current models.

For example, say we are told to insert some element u into the partial order given in Fig. 4.1 (i). After querying node a we learn that $u \leq a$. After querying node b we learn that $u > b$ or incomparable to b. Similarly, after querying node c we learn that $u > c$ or incomparable to c. Thus,

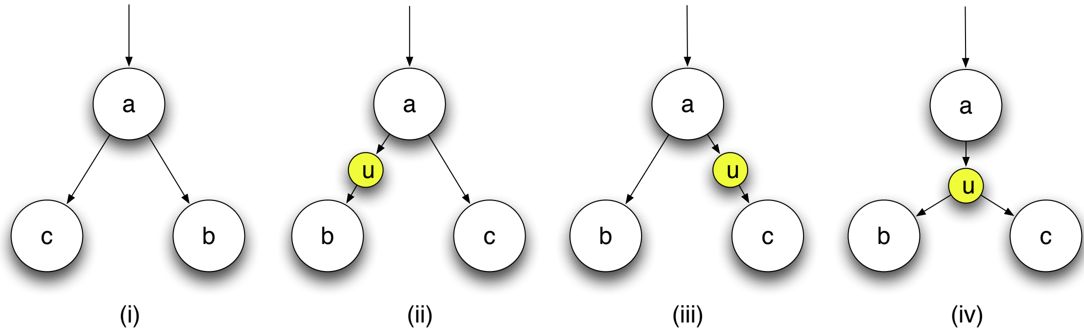

Figure 4.1: In addition to *closer to* x or *closer to* y, we allow an edge (x, y) to answer HERE when a node u falls between x and y in the partial order. Thus, given the partial order in (i) and an element u to be inserted, we can properly distinguish the partial orders given in (ii)-(iv).

the partial orders given in Fig. 4.1 (ii)-(iv) are all consistent with the answers. Because of this, our model must be able to distinguish between *greater than* and *incomparable*. We allow an edge (x, y) to additionally answer HERE when an element u falls strictly between a and b in the partial order. This information is sufficient to implement insertion and deletion. For example, if u is actually the neighbor of both c and b (as shown in Fig. 4.1 (iv)) then questions (a, b) and (b, c) both answer HERE. Note that the ability to answer HERE is equivalent to asking $u \leq a$ and then $b > u$, so any edge query can be simulated by two comparisons. In fact, since we ultimately are performing two comparison operations anyway, we allow edges queries between any two nodes in the tree. We now give a formal denition of edge queries for dynamic sets.

Definition 4.6. Let u be an element in $\mathcal{U}$ and x and y be two nodes in P. Then an **edge query** on (x, y) with respect to u has one of the following three answers:

- HERE – on or off the path between x and y
- x – in the subtree rooted at x not including this path (may be equal to x)
- y – in the subtree rooted at y not including this path (may be equal to y)

We assume edge queries are constant-time comparisons, as data structures exist that allow us this abstraction [4]. Also, we emphasize that any ternary search tree with edge queries has an equivalent binary search tree with element comparisons such that the height of the binary search tree is at most twice that of the ternary search tree.

4.3 Construction

We build the Leaf-Line tree inductively on P, from the bottom-up, by repeating the following two steps which constitute one iteration:

1. contract every chain of contiguous degree-2 nodes into a balanced binary search tree (Fig. 4.2 (i)); and then

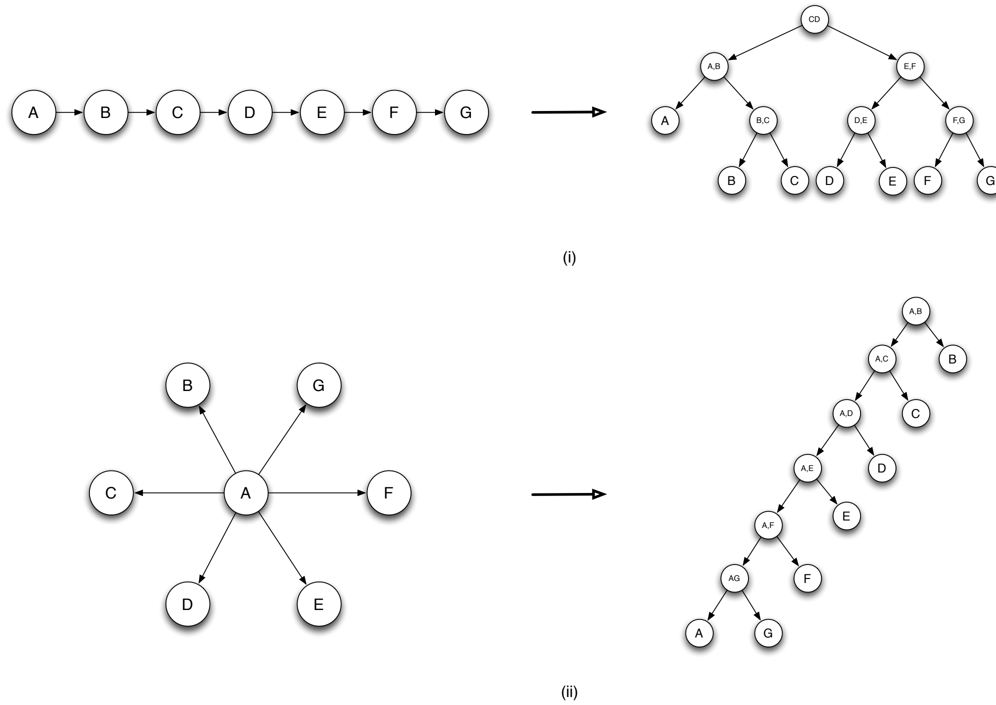

Figure 4.2: Examples of a line contraction where we build a balanced binary search tree from a path (i) and of a leaf contraction where we build a linear search tree from a star node (ii).

2. contract the leaves of every star node into a linear search tree (Fig. 4.2 (ii)).

Let $L_0 = P$ be the original partial order tree. If the algorithm takes k iterations total, then the final result is a single node which we label L_k . In general, let L_i be the tree after executing i iterations of the construction. We also refer to L_i as level i of the construction.

Each contraction step in the algorithm builds a component search structure of the Leaf-Line tree. In particular, each path contraction yields an edge question which determines whether we should search the BST it represents and each star contraction yields a node representing a linear search tree on the edges contracted into it (including itself). In this way, given any level L_i of the construction, we can view an edge (x, y) as a search tree over all the nodes falling strictly between x and y. If (x, y) is an actual edge in L_0 then the BST for it is empty. Similarly, we can view a node x as linear search tree of edge questions that correspond to the edges contracted into it on a previous step. The final question in this sequence yields the node x. If no edges were ever contracted, then the search tree for x is just the node itself. We now give details on how each contraction step forms its corresponding search tree.

Base Cases

Let every edge in L_0 be an empty balanced binary search tree. Let every node in L_0 be a search tree composed of just the node itself.

Line Contraction

Consider the path contraction step of iteration i: If $x_2, \ldots, x_{t-1}$ is a chain of contiguous degree-2 nodes in L_i bounded on each side by non-degree-2 nodes x_1 and x_t respectively, we contract this chain into a balanced binary search tree (BST) over the nodes $x_2, \ldots, x_{t-1}$. We assume the BST performs edge queries to identify the nodes instead of the classical node queries. Recall that each node represents a search tree. The result of the path contraction is an edge labeled (x_1, x_t). This edge represents a question. If u is an element of $\mathcal{U}$ then an edge query on (x_1, x_t) with respect to u will continue the search either on the search trees given by x_1 or x_t or, in the case of a HERE answer, on the balanced binary search tree over $x_2, \ldots, x_{t-1}$.

Leaf Contraction

Consider the star contraction step of iteration i: If $y_1, \ldots, y_t$ are all degree-1 nodes in L_i adjacent to a node z in L_i, we contract them into the linear search tree (LST) associated with z. The LST consists of edge questions of the form (z, y_j). If u is an element of $\mathcal{U}$ then an edge query on (z, y_j) with respect to u will continue the search either (1) on the current LST with question (z, y_{j+1}) if u is closer to z, (2) on the LST associated with y_j if u is closer to y_j, or (3) on the BST given by the edge (z, y_j).

If nodes were already contracted into the LST from a previous iteration, we now add the new ones, as well. Each question (z, y_j) in the LST is asked sequentially and so the height of the LST is equal to the number of nodes contracted into z.

Definitions

With respect to the construction algorithm above, we define three properties which classify all nodes in a poset once a Leaf-Line tree has been built.

Definition 4.7. The **round** of a node x is the iteration i where x was contracted. We say ROUND(x) $= i$.

Definition 4.8. The **type** of a node represents the step where the node was contracted. If node x is *line contracted*, we say TYPE(x) = LINE, otherwise we say TYPE(x) = LEAF.

Definition 4.9. We define the PARENT of a node x as follows:

- if TYPE(x) = LINE and x was contracted into the BST associated with some edge (y, z) then PARENT(x) = (y, z).
- if TYPE(x) = LEAF and x was contracted into the LST associated with some node t then PARENT(x) = t.

4.3.1 Example

In this section we give an example of the Leaf-Line tree construction. We show how a poset Q undergoes repeated iterations of the construction algorithm (Fig. 4.3), as well as the final resulting decision tree for Q (Fig. 4.4).

Suppose Q has the tree structure illustrated in Fig. 4.3 (i). We associate an empty BST with every edge in $L_0 = Q$ and an LST comprised of only the node itself with every node in L_0. The first path contraction creates BSTs for the chains $\{G, H, I, J, K\}$, $\{P\}$, $\{S, T\}$, $\{W\}$, and associates them with the edges (F, L), (F, R), (R, V), (R, X), respectively. We obtain the tree in Fig. 4.3 (ii).

The first iteration ends with a star contraction step that adds collections of leaves $\{A, B\}$, $\{D, E\}$, $\{M, N\}$, $\{V, X\}$ to the LSTs of elements C, F, L, R, respectively. We thus obtain the tree in Fig. 4.3 (iii).

At this point, the tree contains no chains of degree-2 nodes, and the next path contraction has no effect. Finally, the second star contraction reduces the tree to a single node (thus ending the construction process) by contracting the final leaves C, L, R into the LST of node F as shown in Fig. 4.3 (v). The resulting Leaf-Line decision tree is shown in Fig. 4.4.

Notice that in the Leaf-Line tree some answers to certain edge queries are marked as *impossible* (in Fig. 4.4 these are denoted by empty nodes with red borders). This happens because the answer HERE to an edge query (x, y) implies that the node u we seek is not equal to either x or y (recall Definition 4.6). However, if there is at least one node on the path between x and y, we need to ask the edge questions adjacent to nodes x and y on that path in order to determine whether u should be placed between two elements. Such a question cannot answer x or y since the HERE answer eliminated this possibility. Thus these choices are impossible in the decision tree.

4.3.2 Analysis

To prove the efficiency of our algorithm we need the following technical result.

Lemma 4.10. The construction algorithm runs for at most $\log w$ iterations.

Proof. Suppose that the algorithm runs for k iterations creating trees $L_0, \ldots, L_k$ and let S_i be the number of star nodes in tree L_i. After path contraction, the new tree L'_i will contain only the stars and leaves of L_i. We say L'_i is a full tree. After a subsequent star contraction, all leaves are pruned away into LSTs and the tree L_{i+1} contains exactly the star nodes of tree L_i. Since L'_i is a full tree, the number of leaves is at least half of the size of L'_i. Thus, after path contracting q nodes and star contracting at least half of the remaining ones, the size of tree L_{i+1} is:

$$|L_{i+1}| = \frac{|L_i| - q}{2} \leq \frac{|L_i|}{2}$$

By Lemma 4.2, $S_0 \leq w$ and so $|L_1| = S_0 \leq w$. Since after each iteration we have less than half the number of elements as before, the construction process must end after at most $\log w$ iterations. □

Given the construction algorithm and the previous lemma, we claim the following efficiency guarantee:

Theorem 4.11. The Leaf-Line tree can be built in $O(n)$ time from the Hasse diagram of a poset of size n.

Proof. For each path contraction we need to identify all degree-2 nodes in the current tree and construct BSTs from them, which we then associate with specific edges. This is a linear time

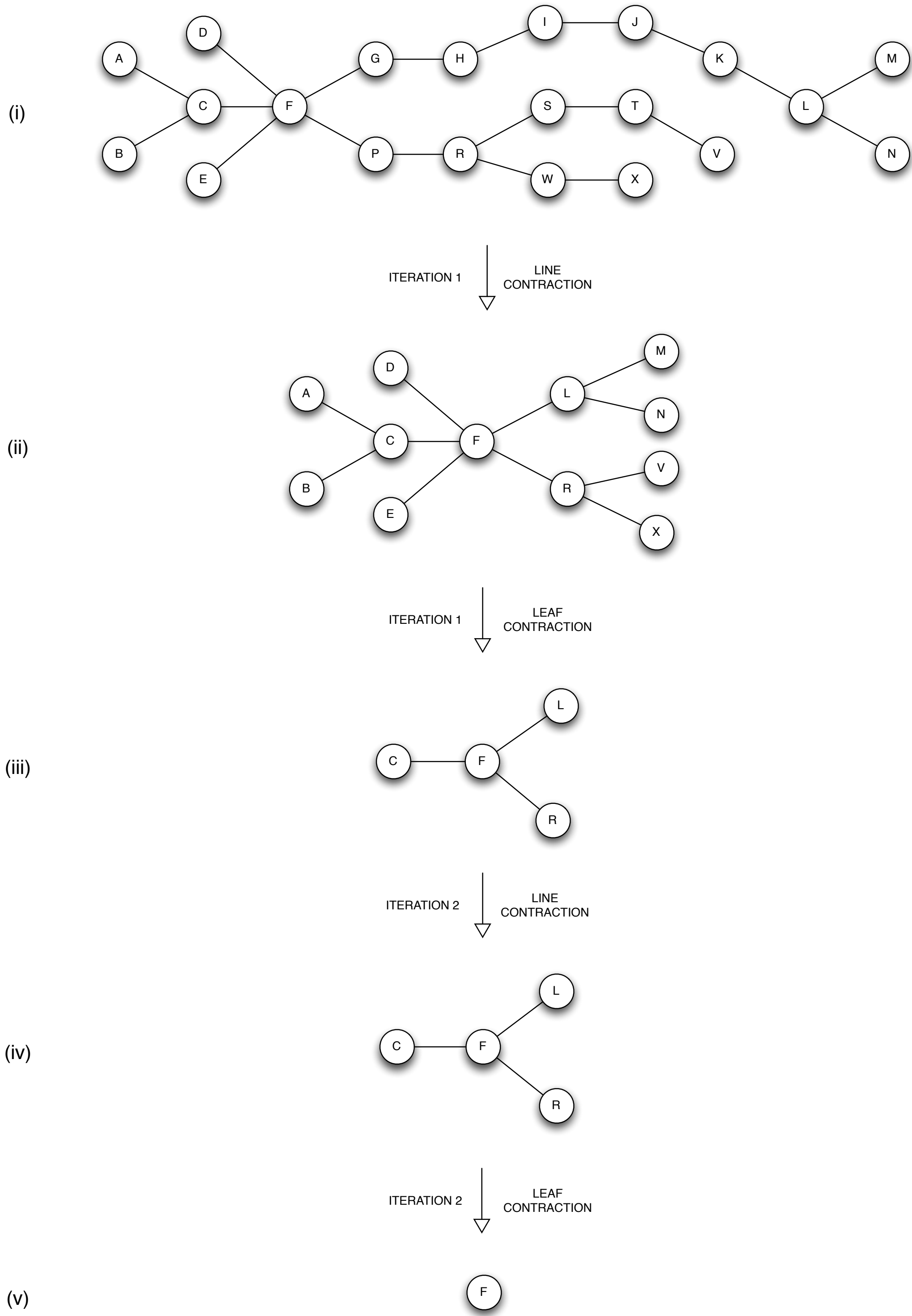

Figure 4.3: Poset Q (i) undergoing repeated contraction steps and iterations of the construction algorithm (ii)-(v).

Figure 4.4: Leaf-Line tree associated with poset Q shown in Fig. 4.3 (i). The red nodes indicate impossible answers.

operation in the size of the tree at iteration i. For each star contraction we add each leaf in the tree to some LST corresponding to its non-degree-1 neighbor. This operation is also linear in the size of the tree.

By the proof of Lemma 4.10 we know the size of the tree is halved after each iteration. Starting with n nodes in $L_0 = P$, the total number of operations performed is:

$$O(n) + O(\frac{n}{2}) + O(\frac{n}{4}) + \cdots + O(1) = O(2n) = O(n).$$

□

Lemma 4.12. On any root-to-leaf path in the Leaf-Line tree there is at most one BST and one LST for each iteration i of the construction algorithm.

Proof. On a root-to-leaf path, the Leaf-Line tree contains LST and BST data structures in decreasing order of the iteration i since the data structure is built incrementally from the bottom up. Suppose we are currently in $LST(A)$. The immediately accessible search structures from this point (aside from ourselves) are:

- $LST(B)$ for all queries $(A, B) \in LST(A)$
- $BST(A, C)$ for all queries $(A, C) \in LST(A)$

If query $(A, B) \in LST(A)$, then TYPE(B) = LEAF and so ROUND$(B) <$ ROUND(A) by construction. If D is a node in $BST(A, C)$, then ROUND$(D) \leq$ ROUND$(C) <$ ROUND(A) since D was line contracted before C was leaf contracted into $LST(A)$.

Now suppose we are currently in $BST(A, B)$. All nodes C contracted into this BST have equal ROUND j by construction. The next accessible search structures are:

- $BST(D, E)$ for all queries $(D, E) \in BST(A, B)$
- $LST(C)$ for each leaf C of $BST(A, B)$ (this LST may consist of only the node C)

If F is a node in $BST(D, E)$, then ROUND$(F) < j$ since F was line contracted before all nodes in $BST(A, B)$ (otherwise, F would be in $BST(A, B)$). If C is a node in $BST(A, B)$ then ROUND(C) $= j$.

Finally, consider a root-to-leaf path. Suppose at some point we are in $LST(A)$ and the next search structure we enter is $LST(B)$. It follows from above arguments that ROUND(A) is strictly smaller than ROUND(B). Suppose at some point we are in $BST(C, D)$ and the next structure on the path is $BST(E, F)$. Then for all nodes G line contracted into $BST(C, D)$ and all nodes H line contracted into $BST(E, F)$, we have ROUND(G) strictly smaller than ROUND(H) and this concludes our proof. □

Theorem 4.13. Let OPT be the query complexity of the optimal search strategy. The height of the Leaf-Line tree is at most $O(\log w) \cdot OPT$.

Proof. From the previous Lemma we know that on any root-to-leaf path in the Leaf-Line tree we search at most one BST and one LST for each iteration i of the construction algorithm. In each LST we perform at most d queries where d is the maximum node degree in P. In each BST we ask at most $\log \mathcal{C}$ questions, where $\mathcal{C}$ is the size of the longest chain in P.

Since there at most $\log w$ iterations by Lemma 4.10, it follows by Lemma 4.4 and Corollary 4.5 that the height h of the Leaf-Line tree is bounded above by:

$$h \leq (d + \log \mathcal{C}) \log w \leq (OPT + OPT) \log w \leq OPT \cdot 2 \log w = O(\log w) \cdot OPT.$$

□

4.4 Search

Given a Leaf-Line tree for a poset we identify two types of search based on the granularity of the expected response:

Test membership: we seek a true or false answer indicating whether an element $u \in \mathcal{U}$ is in P.

Find location: we want to identify the precise location where an element from the universe must be inserted in the Hasse diagram of our poset P.

The *Test membership* search is a fast test that establishes whether an element $u \in \mathcal{U}$ is in P. However, this type of search does not provide us with the location of u in the Hasse diagram for P. We require this information when inserting u in P. *Find Location* is a slower, yet more specific search, that provides enough information for the insertion algorithm to run.

4.4.1 Test Membership

We present the following algorithm which gives a true or false answer to whether an element $u \in \mathcal{U}$ is in P. Note that the only terminal nodes in the Leaf-Line tree are either leaves of the poset P or empty BSTs (corresponding to a final HERE answer).

Algorithm 4.1 The **Test Membership** Algorithm.

```
Follow root-to-leaf path in Leaf-Line tree given by successively querying for element u
if search terminates at a poset leaf then
    u is in P
else if search terminates at an empty BST then
    u is not in P
end if
```

The following result is self-evident.

Theorem 4.14. Algorithm 4.1 has query complexity equal to the height of the Leaf-Line tree.

4.4.2 Find Location

If a node $u \in \mathcal{U}$ is not in P, we need an algorithm to find the position where u should be inserted in P. Because we are dealing with dynamic posets, recall from Section 4.2 that a node $u \in \mathcal{U}$ may actually fall between nodes a and b in P. Furthermore, if node a has several neighbors then u may be juxtaposed between a and only *some* of those nodes, as shown in Fig. 4.1. If we receive a HERE answer, we must query all edges (a, b_i), where b_i is a neighbor of a, in order to disambiguate between all possibilities. Every HERE answer indicates that node u falls between a and b_i. If we receive multiple HERE answers, the search stops and the insertion location of u is between a and all b_is that answered HERE. If we receive only one HERE answer, we continue the search into the BST associated with edge (a, b_i). Note that in our query model, it is enough to query each *direction* incident to a (i.e. each edge in $LST(a)$) in order to identify to which neighbors of a is node u adjacent. The algorithm for finding the insertion location for node u in P is given below.

Algorithm 4.2 The **Find Location** Algorithm.

```
Follow Algorithm 4.1
if query (x, y) answers HERE then
    query all edges in LST(x) and LST(y)
    query any other edge adjacent to x or y (i.e. in BST(Parent(x)) and BST(Parent(y)) if
            TYPE(x) = LINE or TYPE(y) = LINE)
    if we obtain at least one more HERE answer then
        search terminates: u is inserted within all edges that answered HERE
    else
        search BST(x, y)
    end if
end if
```

Theorem 4.15. Algorithm 4.2 has query complexity within a constant factor of the height of the Leaf-Line tree.

Proof. When we get a HERE answer, we stop searching in the current structure. After determining the number of HERE responses, the search either terminates or continues in a subsequent BST on the same root-to-leaf path. Searching for a neighbor edge to a given edge in a BST is $O(\log s)$ in its size by [9]. By the proof of Lemma 4.12, this implies Algorithm 4.2 searches at most two more LSTs and two more BSTs per iteration. By Lemma 4.4, Algorithm 4.2 performs at most:

$$(3d + 3\log \mathcal{C}) \log w \leq (3OPT + 3OPT) \log w \leq OPT \cdot 6 \log w = O(\log w) \cdot OPT.$$

queries, which is within a constant factor of the height of the tree by Theorem 4.13. □

4.5 Neighbors

Given an element $p \in P$, a natural question is finding the neighbors of p in the poset. Since our data structure does not take into account directed edges, we will focus on the analogous problem

of identifying the *neighbors* of node p in P. Note that once a neighbor x has been found, we can classify it as a *predecessor* or *successor* of p in constant time by querying the edge (x, p).

We present the following algorithm for finding *all* the neighbors of element p.

Algorithm 4.3 An Algorithm for **Finding All Neighbors** of an Element $p \in P$.

```
Follow root-to-leaf path in Leaf-Line tree given by successively querying for element p
When p has been found:
for all edges (p, x) in LST(p) do
    Find closest element z to p in direction (p, x)
    Report z as a neighbor of p
end for
if TYPE(p) = LEAF then
    Let PARENT(p) = a_node (if it exists)
    Find closest element t to p in direction (p, a)
    Report t as a neighbor of p
else if TYPE(p) = LINE then
    Let PARENT(p) = (a, b)_edge
    Find adjacent elements i < p and j > p in BST(a, b) (if they exist)
    Find closest elements t, u to p in directions (p, i) and (p, j)
    Report t and u as neighbors of p
end if
```

We can find the closest element to p in a certain direction (p, z) by first choosing HERE on this edge question and subsequently always choosing the endpoint closest to p of all asked queries. When this choice would lead to an *impossible* node, we chose HERE, and continue as before. Finally, when the endpoint closest to p is not a valid choice on query (p, x) *and* the HERE BST is empty, we know that x is the nearest neighbor of p in that direction. For example, in Fig. 4.4, to find the nearest neighbor of node F on the direction (F, L), we first chose HERE on this edge, then we select endpoints I and G for the following two queries as these nodes are closest to F and yield a valid choice. Finally, for question (F, G), neither the answer F, nor the answer HERE are valid, and we conclude that G is the neighbor we seek.

Since balanced binary search trees support efficient *neighbor* operations [9], we can find each of the adjacent elements $i < p$ and $j > p$ in $BST(a, b)$ using at most $\log \mathcal{C}$ queries.

Theorem 4.16. Let H be the height of the Leaf-Line tree. Algorithm 4.3 can find each neighbor in $O(H)$ time and has query complexity $H \cdot O(d)$.

Proof. To find a neighbor in one of the directions $(p, x) \in LST(p)$ we need at most H queries. If TYPE(p) = LEAF, finding the parent of p is a constant time operation (no queries needed since we store this information for each node) and identifying the neighbor takes H queries in the worst case. If TYPE(p) = LINE, finding the adjacent elements to p requires fewer than $2 \log \mathcal{C}$ queries and we can discover the neighbors using at most $2H$ queries. Since $\log \mathcal{C} \leq OPT$ by Corollary 4.5 and $H = OPT \cdot O(\log w)$ by Theorem 4.13, we conclude that we can find each neighbor in $O(H)$ time.

Finally, each node $p \in P$ has at most d neighbors, and so Algorithm 4.3 has query complexity $O(H) \cdot O(d) = H \cdot O(d)$. □

4.6 Insertion

In this section we describe an efficient method of updating the Leaf-Line tree when inserting a new element $u \in \mathcal{U}$ into poset P. Depending on the location of u in P, we distinguish between two types of insertion:

Node insertion: u is inserted as a leaf in the poset tree

Edge insertion: u is inserted between two or more elements in the poset tree

To identify the position of u in P we use the Find Location search algorithm (Algorithm 4.2). For *node insertion* the new element u has only one neighbor in P. For *edge insertion* the algorithm returns a collection of nodes comprised of a node $B \in P$ together with a subset of nodes from $LST(B)$. These are all the HERE answers obtained when querying edges in $LST(B)$. The insertion procedure also depends on the new neighbor(s) of u in P.

Definition 4.17. The **insertion point** of a node u is:

- the only neighbor of u in the case of a *node insertion*
- the node whose LST we query to find the other neighbors of u in the case of an *edge insertion*

Nodes in the Leaf-Line tree are completely characterized by the following three properties defined in Section 4.3: ROUND, TYPE, and Parent. In designing the insertion algorithms our goal is for the updated tree to be essentially the same tree (modulo differences in the balanced binary search trees) as we would have built using the induction algorithm in Section 4.3 had we started with the poset $P \cup \{u\}$. In the algorithms provided in this section we describe the updates that need to occur both to the Leaf-Line tree and to the properties of nodes such that we maintain this invariant.

Note We use the following conventions for all our figures in this chapter:

- the insertion point B is denoted by a doubly-stroked node border
- unreachable nodes in the Leaf-Line tree have a dark red border
- nodes already contracted in the $LST(K)$ are denoted by smaller unmarked nodes radially neighboring the larger marked node K
- node superscripts denote ROUND numbers
- continuous lines denote an edge that exists at the current iteration step
- dotted lines signal the existence of a path between their endpoints in the Leaf-Line tree, not necessarily comprised of nodes, edges, or data structures visible at this iteration step
- other colors (i.e. blue, green, gray) are used to clarify the pre-insertion and post-insertion positions of various subsets of the Leaf-Line tree

4.6.1 Node Insertion

Let A be the node to be inserted in P at insertion point B. We distinguish three cases which are detailed below. We prove in Section 4.6.3 that our algorithms are exhaustive, as they provide valid updates for all node insertion possibilities which may occur in practice.

Case 1: *B is the Final Node in the Contraction Process*

The insertion point B is the highest node in the Leaf-Line tree and has no PARENT. We have the following possibilities:

Case 1.1: There exists node M such that only B, A, and M survive after ROUND$(B)-1$ iterations

After ROUND$(B)-1$ iterations we perform a subsequent line contraction and B becomes the only node in $BST(M, A)$. We arbitrarily chose M to become the final node and path contract A into $LST(M)$. We describe the formal changes to the poset and data structure in Fig. 4.5 and Algorithm 4.5.

Case 1.2: There exists no such node M

If ROUND$(A) <$ ROUND(B) A is contracted into $LST(B)$ at some point during the contraction process. If ROUND$(A) =$ ROUND(B), then after enough iterations the poset consists of only nodes A and B and now undergoes an additional contraction. In this case, we can arbitrarily choose to increase ROUND(B) by one unit and contract A into $LST(B)$. We describe the formal changes to the poset and data structure in Fig. 4.6 and Algorithm 4.6.

Case 2: *B is not the Final Node and* ROUND*$(A) <$* ROUND*(B)*

Since ROUND$(A) <$ ROUND(B), node A would be leaf contracted into B before the iteration which contracts B into its PARENT. We thus insert A into $LST(B)$, regardless of the TYPE of B (Fig. 4.7 and Algorithm 4.7).

Case 3: *B is not the Final Node and* ROUND*$(A) =$* ROUND*(B)*

When ROUND$(A) =$ ROUND(B), we have the following possibilities:

Case 3.1: TYPE$(B) =$ LEAF

We know that PARENT$(B) = E$ for some node E of ROUND higher than ROUND(B). Since ROUND$(A) =$ ROUND(B), nodes A and B are now contracted at the same iteration. At that step, A is the farthest such node from E in this direction, so B is line contracted and A is leaf contracted and replaces B as a LEAF node in $LST(E)$. If $BST(E, B)$ consists of nodes line contracted at iteration ROUND(B), as well, then B will become their peer. Otherwise, if $BST(E, B)$ consists of nodes line contracted at a previous iteration, we create a new $BST(A, E)$ which contains a single node: B (Fig. 4.8 and Algorithm 4.8).

Case 3.2: TYPE(B) = LINE

Let PARENT(B) = (E, F). After ROUND(B) iterations, node B will have degree 3 instead of 2 since A survives. Similarly to Case 2 (Fig. 4.7 and Algorithm 4.7), A is leaf contracted into $LST(B)$. However, B is no longer line contracted at this step, but survives one extra iteration itself. This implies that ROUND(B) increases by one which triggers other changes depending on the relationship between B and its parent edge. In all scenarios, since B was formerly line contracted into $BST(E, F)$, we must split this structure into $BST(E, B)$ and $BST(B, F)$ comprised of all nodes in the initial BST that fall between E and B, respectively between B and F. We associate these new BSTs with edges (E, B) and (B, F), respectively.

Let ROUND(B) refer to the non-incremented value and let PARENT(B) = (E, F). WLOG, let max(ROUND(E), ROUND(F)) = ROUND(F). We then identify the following subcases:

- *3.2.1: B becomes the sole node in a new $BST(E, F)$*

  When ROUND(B)+1 < ROUND(E) ≤ ROUND(F), the node B will still be line contracted between E and F, but one iteration after it was contracted in the initial poset. Furthermore, B becomes the only node between E and F to be line contracted at iteration ROUND(B)+1.

  We encounter a similar situation if ROUND(B)+1 = ROUND(E) < ROUND(F) and TYPE(E) = LEAF. Prior to the node insertion, all nodes between E and F were contracted in the first ROUND(E)−1 iterations. Now, B is the only node to be line contracted in the same ROUND as E, which is subsequently leaf contracted.

  In both scenarios, a new balanced binary search tree is created whose only queries are the edges (E, B) and (B, F). We describe the formal changes to the poset and data structure in Fig. 4.9 and Algorithm 4.9.

- *3.2.2: B is line contracted together with E*

  Suppose ROUND(B)+1 = ROUND(E) < ROUND(F) and TYPE(E) = LINE. Adding node A that is contracted at ROUND $k - 1$ will cause B to still be in the poset when iteration k begins. Since the only neighbors of B that have survived to this point are E and F we know B will be line contracted together with E into some $BST(F, H)$ = PARENT(E) (Fig. 4.10 and Algorithm 4.10).

- *3.2.3: B is leaf contracted into $LST(F)$*

  Suppose ROUND(E) < ROUND(B)+1 < ROUND(F). Since PARENT(B) = (E, F) it follows that ROUND(B) ≤ ROUND(E). Then ROUND(E) < ROUND(B)+1 implies ROUND(B) = ROUND()E before inserting. Furthermore, TYPE(E) = LEAF, otherwise (E, F) could not be PARENT(B).

  After ROUND(E)−1 post-insertion iterations, B will have degree 3 instead of 2 since A survives. In the subsequent iteration, E and A are leaf contracted into $LST(B)$ and finally one iteration later, B is contracted into $LST(F)$, since ROUND(B)+1 < ROUND(F).

- *3.2.4: B is contracted during the same iteration as E and F*

  Suppose ROUND(E) = ROUND(B)+1 = ROUND(F). Because of node A, B survives one extra iteration and its only neighbors are E and F, both previously contracted at this step.

  (a) If TYPE(E) = TYPE(F) = LINE, then E and F were contracted into the same $BST(G, H)$ (poset P is a tree). Since B survives, but its only neighbors are E and F, it follows that all 3 nodes are now line contracted together into $BST(G, H)$ (Fig. 4.12 and Algorithm 4.12).

  (b) If TYPE(E) = LINE and TYPE(F) = LEAF, then E was line contracted into some $BST(G, F)$, where ROUND(G) > ROUND(F). After insertion, B survives one extra iteration and its only neighbors are E and F. Thus, B is subsequently line contracted into $BST(G, F)$ together with E (Fig. 4.13 and Algorithm 4.13).

- *3.2.5: Recursive step: B becomes the new node to be inserted at insertion point F*

  If ROUND(E) < ROUND(B)+1 = ROUND(F), then similarly to Case 3.2.3, E and A are leaf contracted into $LST(B)$ on iteration ROUND(B) (non-incremented value) (Fig. 4.14, Algorithm 4.14). But on the subsequent iteration, node F has one extra neighbor: node B. We note that reasoning about the contraction of F is analogous to applying the Node Insertion procedure: we are seeking to *insert* node B at insertion point F knowing ROUND(B) (incremented value) = ROUND(F).

We summarize the cases of the Node Insertion procedure as follows:

Case 1 B is the final node in the contraction process.

- *1.1* Exactly 3 nodes survive after ROUND(B)−1 iterations — **Fig. 4.5, Algorithm 4.5**
- *1.2* Otherwise — **Fig. 4.6, Algorithm 4.6**

Case 2 B is not the final node and ROUND(A) < ROUND(B) — **Fig. 4.7, Algorithm 4.7**

Case 3 B is not the final node and ROUND(A) = ROUND(B)

- *3.1* TYPE(B) = LEAF — **Fig. 4.8, Algorithm 4.8**
- *3.2* TYPE(B) = LINE and PARENT(B) = $(E, F)_{edge}$
  - *3.2.1* ROUND(B)+1 < ROUND(E), ROUND(F) or ROUND(B)+1 = ROUND(E) < ROUND(F) and TYPE(E) = LEAF — **Fig. 4.9, Algorithm 4.9**
  - *3.2.2* ROUND(B)+1 = ROUND(E) < ROUND(F) and TYPE(E) = LINE — **Fig. 4.10, Algorithm 4.10**
  - *3.2.3* ROUND(E) < ROUND(B)+1 < ROUND(F) — **Fig. 4.11, Algorithm 4.11**
  - *3.2.4* ROUND(B)+1 = ROUND(E) < ROUND(F)
    - *(a)* TYPE(E) = TYPE(F) = LINE — **Fig. 4.12, Algorithm 4.12**
    - *(b)* TYPE(E) = LINE and TYPE(F) = LEAF — **Fig. 4.13, Algorithm 4.13**
  - *3.2.5* ROUND(E) < ROUND(B)+1 = ROUND(F) — **Fig. 4.14, Algorithm 4.14**

4.6.2 Edge Insertion

Let A be the node inserted in P at insertion point B and consider the set $S = \{C_1, \ldots, C_n, C_{left}, C_{right}\}$ where (B, C_i) is an edge in $LST(B)$. If $\text{TYPE}(B) = \text{LINE}$ then C_{left}, C_{right} are the two corresponding directions on the line. If $\text{TYPE}(B) = \text{LEAF}$, then $C_{left} = C_{right} = C_{line}$.

The Find Location algorithm will return a collection WLOG $S_{found} = \{C_1, \ldots, C_k\}$, where all edges (or directions) $(B, C_j), 1 \le j \le k$ answered HERE when queried about node A. Furthermore, let $S_{stolen} = S_{found} \setminus \{C_{left}, C_{right}\}$ and $S_{not\ stolen} = S \setminus (S_{found} \cup \{C_{left}, C_{right}\}$.

We first describe an algorithm to insert a node A (implicitly $LST(A)$ whose final leaf is A) *into an edge* (E, F) when A is adjacent to endpoint E in the poset P (Algorithm 4.4). This algorithm searches down into the Leaf-Line tree for the BST created at iteration $\text{ROUND}(A)$ adjacent to node E. If found, A will be inserted into this BST. However, if this BST is empty or does not exist, then DS in Algorithm 4.4 will be the LST corresponding to the node M closest to E within the edge (E, F). In this case, a new $BST(E, M)$ will be created and populated with the element A.

Algorithm 4.4 Insert Node A into edge (E, F) when A is adjacent to E in P.

while Current data structure was built at iteration $\text{ROUND}(A)$ or higher **do**
  Run *Test Membership* search algorithm (Algorithm 4.1) for A starting in $BST(E, F)$
end while
Let $DS = \text{PARENT}(\textit{Data Structure Reached}) = LST(M)$ or $BST(E, M)$ for some node $M \in P$
Updated Properties:
$\text{TYPE}(A) \leftarrow \text{LINE}$
$\text{PARENT}(A) \leftarrow (E, M)_{edge}$
Data Structure Updates:
insert A into $BST(E, M)$

We provide the following preliminary definitions before describing our *edge insertion* algorithm.

Definition 4.18. Let α, β, γ, and δ be defined as follows:

- $\alpha = \max_{C_i \in S_{notstolen}} \{\text{ROUND}(C_i)\}$ is the maximum ROUND number of all the nodes *not stolen* by A from B. If no such node exists, then $\alpha = 0$.

- Choose $F \in C_i \in S_{not\ stolen}$ such that $\text{ROUND}(F) = \alpha$. Then $\beta = \max_{C_i \in S_{not\ stolen} \setminus \{F\}} \{\text{ROUND}(C_i)\}$ is the second maximum ROUND number of all the nodes *not stolen* by A from B (counting duplicates). If $S_{not\ stolen}$ has fewer than 2 elements then $\beta = 0$.

- $\gamma = \max_{C_i \in S_{stolen}} \{\text{ROUND}(C_i)\}$ is the maximum ROUND number of all the nodes *stolen* by A from B. If no such node exists, then $\gamma = 0$.

- Choose $F \in C_i \in S_{stolen}$ such that $\text{ROUND}(F) = \gamma$. Then $\delta = \max_{C_i \in S_{stolen} \setminus \{F\}} \{\text{ROUND}(C_i)\}$ is the second maximum ROUND number of all the nodes *stolen* by A from B (counting duplicates). If S_{stolen} has fewer than 2 elements then $\delta = 0$.

For *edge insertion*, we distinguish the following three cases:

Case 1: *B is the Final Node in the Contraction Process*

The insertion point B is the highest node in the Leaf-Line tree and has no PARENT. We let k denote the ROUND of B before insertion.

Case 1.1: A has more ROUND $k-1$ neighbors than B (notwithstanding each other) We argue that before insertion B could only have either one or at least three ROUND $k-1$ neighbors. If it had none, then the contraction process would have finished in less than k iterations, which is a contradiction. If B had two such neighbors, then only B and these two nodes would have survived after $k-1$ iterations. But a tree with three nodes necessarily forms a path and we cannot leaf contract both the neighbors of B.

Now, if $\gamma > \alpha$, then A has at least one such neighbor and B has none. It follows that A will replace B as the final node in the contraction process. Furthermore, if $\alpha = \beta$, i.e. B has two neighbors of equal maximum ROUND, then B will survive exactly $\alpha+1$ iterations. We can thus use the Node Insertion algorithm to reason about how B will be contracted given that A is the new final node (Fig. 4.15 (i)). However, if $\alpha > \beta$, then B has only one neighbor of maximum ROUND. B will survive exactly $\beta + 1$ iterations since it already has two other neighboring directions that survive to higher ROUNDS: node A and the neighbor corresponding to α. In this case, we must use Algorithm 4.4 to discover where B is contracted into the edge (A, M), where $M \in S_{not\ stolen}$ and ROUND$(M) = \alpha$ (Fig. 4.15 (iii)).

If $\gamma = \alpha$, but $\delta > \beta$, we must have that $\delta = \alpha$ by the argument above (B cannot have exactly two neighbors of ROUND $k-1$). Thus, A has more such neighbors than B and A becomes the new final node in the contraction process. Since α cannot equal β, we must insert B into the edge corresponding to the neighbor of ROUND $k-1$ using Algorithm 4.4 (Fig. 4.15 (ii)).

We present a formal description of this case in Algorithm 4.15.

Case 1.2: B has more ROUND $k-1$ neighbors than A (notwithstanding each other) This case is symmetrical to Case 1.1. If either $\alpha > \gamma$, or $\alpha = \gamma$ and $\beta \geq \delta$, then B remains the final node in the contraction process. We then argue analogously to above that either A is inserted into the edge (B, M), where $M \in S_{stolen}$ and ROUND$(M) = \gamma$, or A is Node Inserted at insertion point B (Fig. 4.16). We present a formal description of this case in Algorithm 4.16.

Case 2: *B is not the Final Node and* TYPE*(B)* = LEAF

Let PARENT$(B) = E$. Recall that we denote direction (B, E) by C_{line}. We distinguish the following two cases:

Case 2.1: A does not steal C_{line} Since the C_{line} direction survives for at least k iterations, A is not contracted until the iteration immediately after its second-to-last neighbor disappears. We infer that ROUND$(A) = \delta + 1$. To compute the new ROUND(B), we must take into account not only nodes in $S_{not\ stolen}$, but also A itself, also a neighbor of B. After recomputing α and β in accordance with this criterion, we use a similar argument to infer that ROUND$(B) = \beta + 1$. If ROUND$(A) \leq$ ROUND(B), we can use the Node Insertion procedure to determine the

remaining changes (Fig. 4.17 (i)). Otherwise, we know ROUND(A) cannot exceed $\delta+1 \leq \gamma+1 \leq$ ROUND(B). A must now replace B in $LST(E)$. Also, if the new round of B is less currently than prior to inserting A, then B will be contracted within the new edge (A, E): we use Algorithm 4.4 (Fig. 4.17 (ii)). We present a formal description of this case in Algorithm 4.17.

Case 2.2: A steals C_{line} We note that this case is analogous to the previous one. If we replace A with B and vice versa, we reduce this case to the problem of inserting node B at insertion point A when B does not steal C_{line}.

Case 3: *B is not the Final Node and* TYPE*(B)* = LINE

Let PARENT(B) = (E, F). Recall that we denote direction (B, F) by C_{left} and direction (B, E) by C_{right}. We distinguish the following three cases:

Case 3.1: A does not steal either C_{left} or C_{right} For node B, the C_{left} and C_{right} directions survive for at least k iterations. Also, because $\alpha = k - 1$ or $\gamma = k - 1$ (or both) we know ROUND(B) $\geq k$. Then A is not contracted until the iteration immediately after its second-to-last neighbor disappears. We infer that ROUND(A) $= \delta + 1$. If $\gamma = \delta$, then the subtree of B in the direction of A will consists of only the node A after $\delta + 1$ iterations. The situation is thus analogous to Node Inserting A at insertion point B (Fig. 4.18 (i)). However, if $\gamma > \delta$, then we must use Algorithm 4.4 to discover where A is contracted into the edge (B, M), where $M \in S_{stolen}$ and ROUND(M) $= \gamma$ (Fig. 4.18 (ii)). We present a formal description of this case in Algorithm 4.18.

Case 3.2: A steals C_{right}, but not C_{left} (WLOG) Since both A and B have two directions which survive for at least k iterations (C_{left} and C_{right}), the time they are contracted is determined by the largest ROUND nodes in S_{stolen}, respectively $S_{not\ stolen}$. Thus, ROUND(A) $= \gamma + 1$ and ROUND(B) $= \alpha + 1$. Furthermore, at least one of A and B must be contracted at iteration k since there exists a node of ROUND $k - 1$ in $S_{stolen} \cup S_{not\ stolen}$ (otherwise ROUND(B) could not be k). If ROUND(A) $<$ ROUND(B), then ROUND(B) $= k$, and we use Algorithm 4.4 to discover where A is contracted into the edge (B, F) (Fig. 4.19 (i)). Similarly, if ROUND(A) $>$ ROUND(B), then ROUND(A) $= k$, and we use Algorithm 4.4 to discover where B is contracted into the edge (A, E) (Fig. 4.19 (ii)). Finally, if ROUND(A) = ROUND(B) $= k$, then A and B are line contracted simultaneously into $BST(E, F)$ (Fig. 4.19 (iii)). We present a formal description of this case in Algorithm 4.19.

Case 3.3: A steals both C_{left} and C_{right} We note that this case is analogous to Case 3.1. If we replace A with B and vice versa, we reduce this case to the problem of inserting node B at insertion point A when B does not steal either C_{left} or C_{right}.

We summarize the cases of the Edge Insertion procedure as follows:

Case 1 B is the Final Node in the Contraction Process

1.1 $\gamma > \alpha$ or $\gamma = \alpha$ and $\delta > \beta$ **Fig. 4.15, Algorithm 4.15**

1.2 $\alpha > \gamma$ or $\alpha = \gamma$ and $\beta \geq \delta$ **Fig. 4.16, Algorithm 4.16**

Case 2 B is not the Final Node and TYPE(B) = LEAF

2.1 A does not steal C_{line} **Fig. 4.17, Algorithm 4.17**

2.2 A steals C_{line} **Analogous to Case 2.1**

Case 3 B is not the Final Node and TYPE(B) = LINE

3.1 A does not steal either C_{left} or C_{right} **Fig. 4.18, Algorithm 4.18**

3.2 A steals C_{left}, but not C_{right} (WLOG) **Fig. 4.19, Algorithm 4.19**

3.3 A steals both C_{left} and C_{right} **Analogous to Case 3.1**

4.6.3 Analysis

We begin by providing the following correctness and efficiency guarantees.

Lemma 4.19. ROUND(A) $\leq$ ROUND(B) when invoking the Node Insertion procedure.

Proof. The Node Insertion procedure is invoked when we add a new leaf u to poset P, recursively from within Case 3.2.5, or externally by the Edge Insertion procedure. When u is a new leaf, ROUND(u) = 1 and the condition is satisfied. A precondition for Case 3.2.5 is ROUND(B)+1 = ROUND(F) and we increment ROUND(B) by exactly one unit before the recursive call. Finally, in the Edge Insertion procedure we invoke Node Insertion for all cases except Case 3.2:

- Case 1.1 We insert node B at insertion point A. By definition, α is smaller than the initial ROUND of B (which is k). We increment this value by one before invoking Node Insertion and thus ROUND(A) $= k \geq \alpha + 1 \geq$ ROUND(B).
- Case 1.2 We insert node A at insertion point B. By definition, $\gamma < k =$ ROUND(B), and thus ROUND(A) $= \delta + 1 = \gamma + 1 \leq$ ROUND(B).
- Case 2.1 The condition is ensured by the `if` test.
- Case 2.2 The proof is analogous to Case 2.1.
- Case 3.1 The proof is analogous to Case 1.2.
- Case 3.3 The proof is analogous to Case 3.1.

□

Lemma 4.20. If PARENT(B) $= (E, F)_{edge}$ and ROUND(E) = ROUND(F), then at least one of E, F has TYPE LINE. Furthermore, ROUND(B) is strictly smaller than at least one of ROUND(E), ROUND(F).

Proof. Suppose that ROUND(E) = ROUND(F) and TYPE(E) = TYPE(F) = LEAF. Then both E and F were contracted in the same leaf contraction step into some $LST(M)$, where ROUND(M) > ROUND(E). This implies that $BST(E, F)$ cannot exist in the Leaf-Line tree, which is a contradiction.

Since PARENT$(B) = (E, F)_{edge}$ we know that ROUND$(B) \leq$ ROUND(E), ROUND(F) by construction. Suppose that ROUND$(B) =$ ROUND$(E) =$ ROUND(F). We know from the previous argument that either TYPE$(E) =$ TYPE$(F) =$ LINE, or TYPE$(E) =$ LEAF and TYPE$(F) =$ LINE. In the former case, B, E, and F would have been line-contracted at the same time, which yields a contradiction. In the latter case, F and B would have been line-contracted at the same time, which also yields a contradiction. □

Lemma 4.21. Node Insertion has query complexity proportional to the height of the Leaf-Line tree.

Proof. To find the position of A in the poset we run the Find Location algorithm. By Theorem 4.15, the query complexity of this operation is $O(H)$, where H is the height of the Leaf-Line tree.

Updating the properties of a node is $O(1)$. Inserting into, removing from, and splitting a BST takes $O(\log s)$ queries, where s is the size of the tree. Since $s \leq n$, these operations are $O(\log n)$ in the number of queries in the worst case. Inserting into an LST is $O(1)$ and removing from an LST is linear in its size which is bounded above by d, the maximum node degree in P. Thus, unless we are invoking the algorithm recursively (Case 3.2.5), we are performing a constant number of operations, each of which has query complexity either $O(d)$ or $O(\log n)$. By Corollary 4.4 and Theorem 4.13, this operation is $O(d + \log n) < O(H)$.

Finally, suppose we are invoking the algorithm recursively. At each recursive call site, the ROUND of the element to be inserted is higher than the previous invocation by exactly one unit. Thus, by Lemma 4.10, we can have at most $\log w$ nested recursive calls and so the maximum complexity of our algorithm is:

$$\log w \cdot O(d + \log n) \leq \log w \cdot O(OPT) \leq OPT \cdot O(\log w) = O(H).$$

□

Lemma 4.22. Algorithm 4.4 has query complexity proportional to the height of the Leaf-Line tree.

Proof. By Theorem 4.14, the Test Membership algorithm has query complexity $OPT \cdot O(\log w)$. Furthermore, updating node properties is $O(1)$ and inserting a node into a BST is $O(\log n)$ by [9]. By Lemma 4.4, $OPT \geq \log n$ and the result follows. □

Lemma 4.23. Edge Insertion has query complexity proportional to the height of the Leaf-Line tree.

Proof. To find the position of A in the poset we run the Find Location algorithm. By Theorem 4.15, the query complexity of this operation is $O(H)$, where H is the height of the Leaf-Line tree.

Analogously to the proof of Lemma 4.21, updating node properties is $O(1)$, inserting into or removing from a BST is $O(\log n)$, inserting into an LST is $O(1)$ and removing from an LST is $O(d)$. During any run of the Edge Insertion procedure, we perform a constant number of BST operations and property updates and at most $O(d)$ LST removals and insertions before invoking a separate procedure. However, we can force the LST removals to occur concurrently with querying the corresponding LST edges in the Find Location algorithm. Thus each query that answers HERE will

remove the appropriate LST from the current data structure, which is a constant time operation at that point. If only one HERE answer is obtained and the search should continue, we reinsert the only LST removed, which is also $O(1)$. Thus, by Lemma 4.4, we require only $O(H)$ queries notwithstanding external procedure calls.

Computing α, β, γ, and δ is $O(d)$ since they depend only on the nodes initially adjacent to the insertion point. Also, in Cases 2.2 and 3.3, replacing A with B and vice versa is $O(OPT)$ since we need only replace B with A in $LST(\text{PARENT}(B))$ and insert at most $O(d)$ stolen nodes into $LST(A)$ (these nodes were already removed from $LST(B)$ by the Find Location algorithm as argued above). This is an $O(d) + O(\log n) \leq O(OPT) \leq O(H)$ operation by Lemma 4.4.

The only external procedures invoked by Edge Insertion are inserting into an edge (Algorithm 4.4) and Node Insertion. These are never called more than once and our proof is concluded by Lemmas 4.21 and 4.22. □

We are now ready to state our main result concerning the insertion procedure.

Theorem 4.24. When inserting an element u into P, the insertion procedure ensures the same Leaf-Line tree that would be generated if we started with poset $P \cup \{u\}$ and ran the construction algorithm. Furthermore, the insertion procedure has query complexity proportional to the height of the Leaf-Line tree.

Proof. By Lemmas 4.19 and 4.20, the Node Insertion procedure handles all valid inputs. The Edge Insertion procedure is also exhaustive by the definition of its cases. By construction, our insertion procedures affect all the necessary modifications (both to the properties of nodes and to the Leaf-Line tree) to ensure that we obtain the same data structure after a dynamical operation as we would if we re-ran the contraction algorithm with the updated poset as input. Finally, by Lemmas 4.21 and 4.23, we require at most $OPT \cdot O(\log w)$ queries to efficiently update the Leaf-Line tree using our insertion algorithms. □

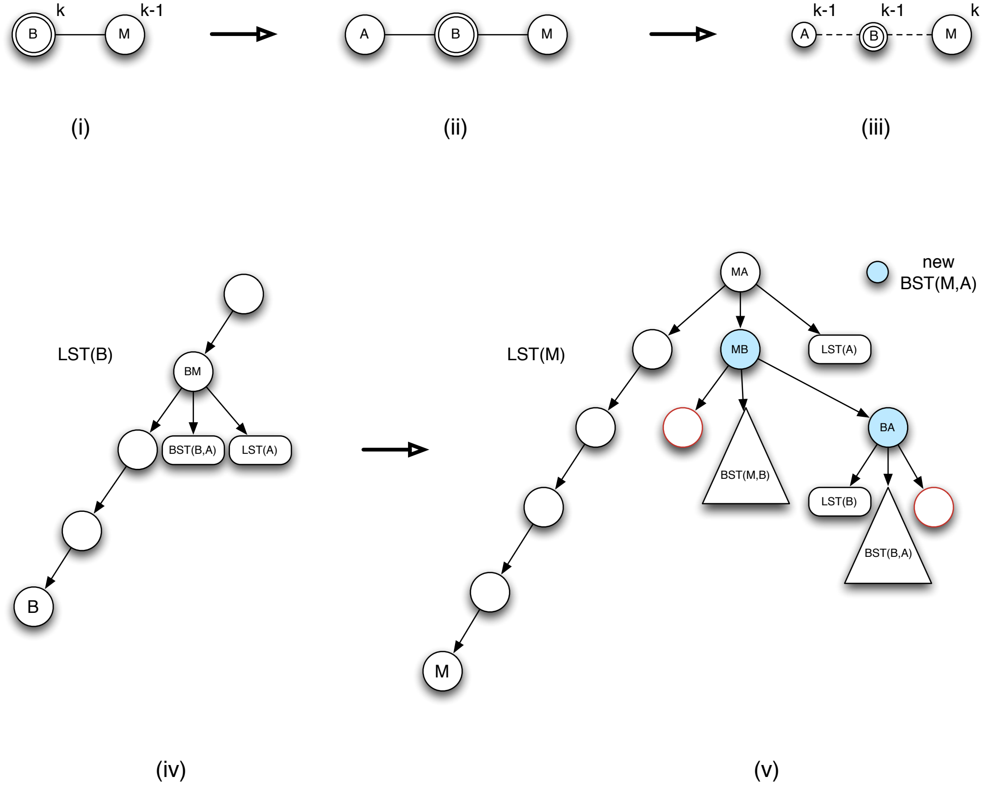

Figure 4.5: Node Insertion, Case 1.1 – B is the final node in the contraction and exactly 3 nodes survive after $\text{ROUND}(B){-}1$ iterations: B, A, and M : (i) poset structure after $\text{ROUND}(B){-}1$ iterations before insertion, (ii) poset structure after $\text{ROUND}(B){-}1$ iterations after insertion, (iii) poset structure after $\text{ROUND}(M)$ iterations after insertion, (iv) data structure before insertion, (v) data structure after insertion.

Algorithm 4.5 Node Insertion Case 1.1: B is the final node in the contraction and exactly 3 nodes survive after $\text{ROUND}(B){-}1$ iterations: B, A, and M.

Updated Properties:

$\text{TYPE}(A) \leftarrow \text{LEAF}$
$\text{PARENT}(A) \leftarrow M$
$\text{TYPE}(B) \leftarrow \text{LINE}$
$\text{ROUND}(B) \leftarrow \text{ROUND}(B){-}1$
$\text{PARENT}(B) \leftarrow (A, M)$
$\text{ROUND}(B) \leftarrow \text{ROUND}(A){+}1$
$\text{PARENT}(M) \leftarrow \text{NONE}$ (M is now the final node)

Data Structure Updates:

remove M from $LST(B)$
create new $BST(A, M)$
insert B into $BST(A, M)$
insert A into $LST(M)$

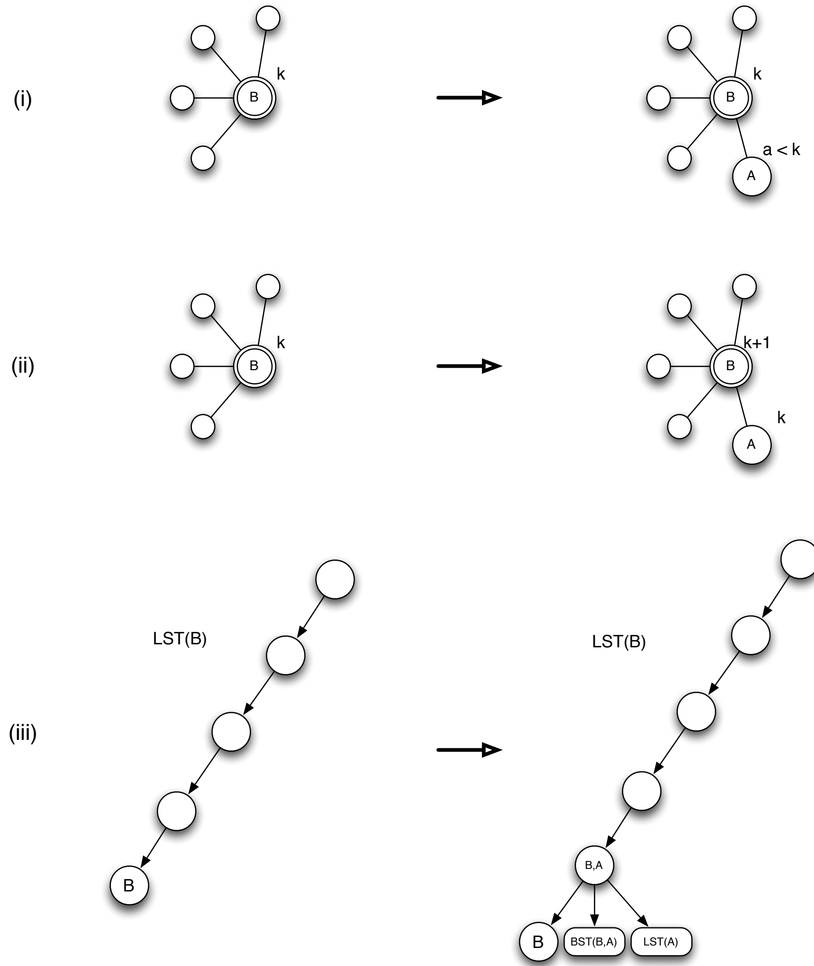

Figure 4.6: Node Insertion, Case 1.2 – B is the final node in the contraction and either 2 or more than 3 nodes survive after $\textsc{round}(B){-}1$ iterations : (i) poset structure changes for $\textsc{round}(A) < \textsc{round}(B)$, (ii) poset structure changes for $\textsc{round}(\mathrm{A}) = \textsc{round}(B)$, (iii) data structure changes.

Algorithm 4.6 Node Insertion Case 1.2: B is the final node and either fewer or more than 3 nodes survive after $\textsc{round}(B){-}1$ iterations.

Updated Properties:

$\textsc{type}(A) \leftarrow \textsc{leaf}$
$\textsc{parent}(A) \leftarrow B$

if $\textsc{round}(A) = \textsc{round}(B)$ **then**
  $\textsc{round}(B) \leftarrow \textsc{round}(B) + 1$
end if

Data Structure Updates:

insert A into $LST(B)$

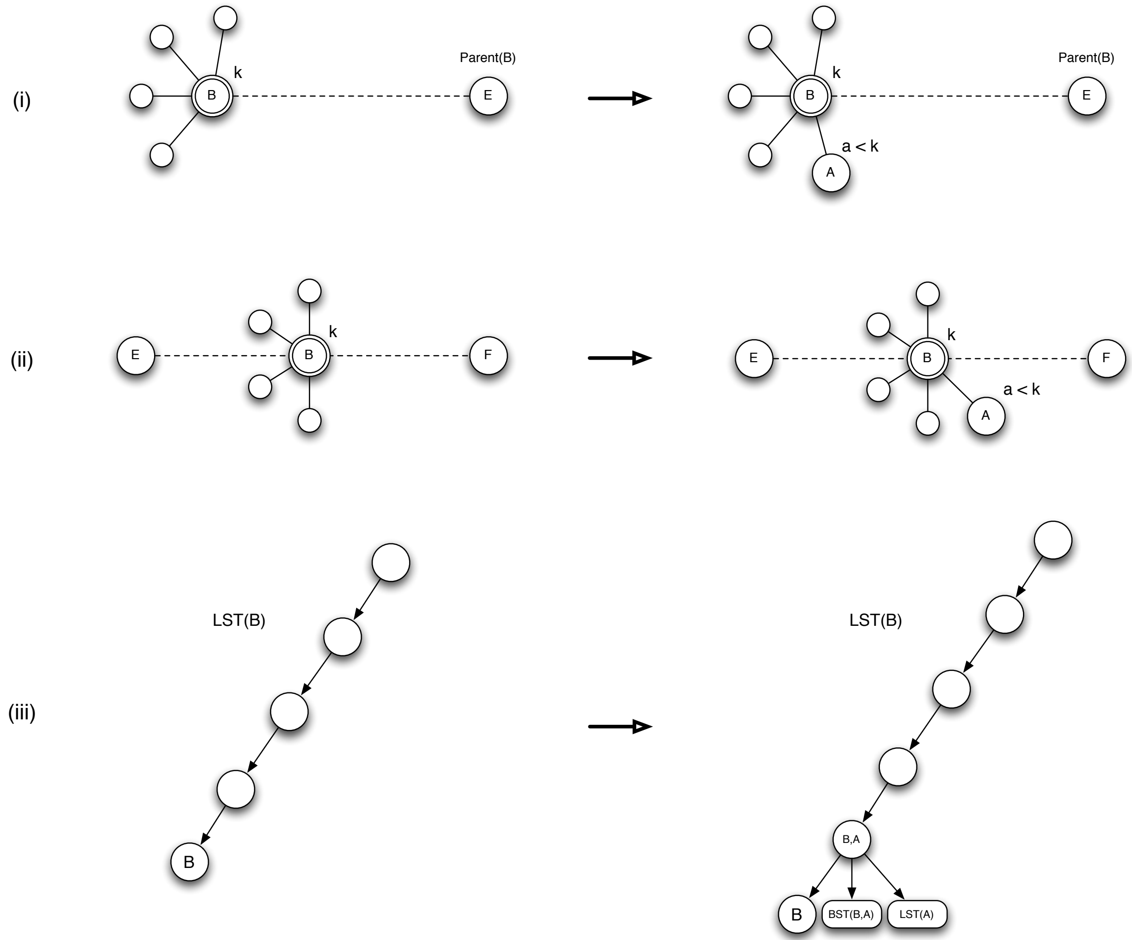

Figure 4.7: Node Insertion, Case 2 – B is not the final node and $\textsc{round}(A) < \textsc{round}(B)$: (i) poset structure changes for $\textsc{type}(B) = \textsc{leaf}$ and $\textsc{parent}(B) = E$, (ii) poset structure changes for $\textsc{type}(B) = \textsc{line}$ and $\textsc{parent}(B) = (E, F)$, (iii) data structure changes.

Algorithm 4.7 Node Insertion Case 2: B is not the final node and $\textsc{round}(A) < \textsc{round}(B)$.

Updated Properties:

$\textsc{type}(A) \leftarrow \textsc{leaf}$
$\textsc{parent}(A) \leftarrow B$

Data Structure Updates:

insert A into $LST(B)$

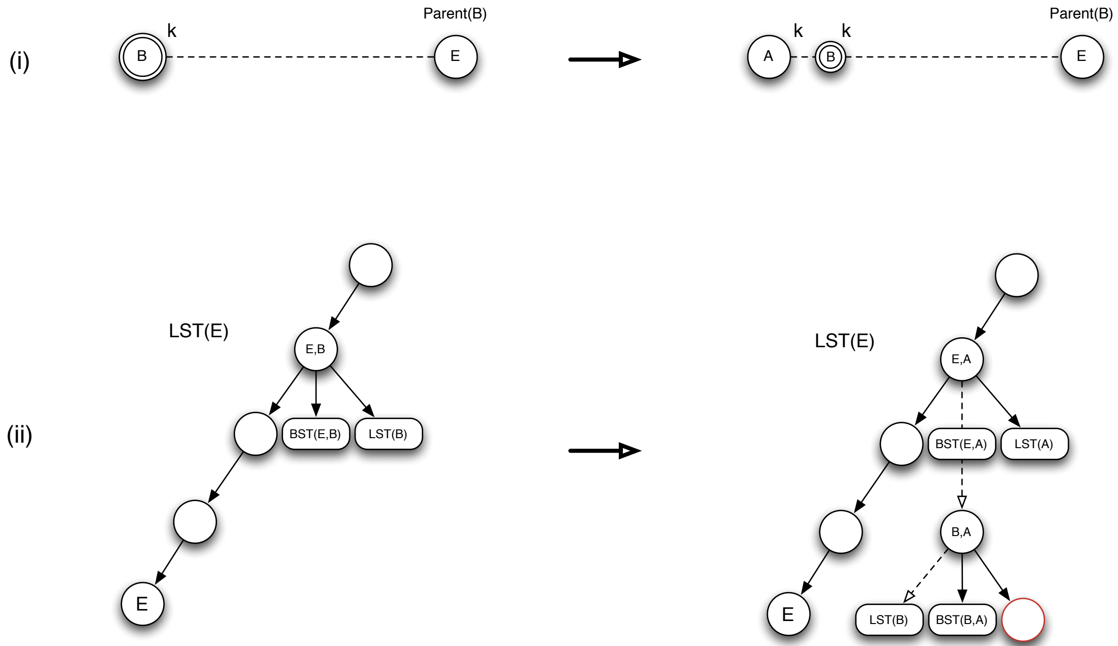

Figure 4.8: Node Insertion, Case 3.1 – B is not the final node, $\textsc{round}(A) = \textsc{round}(B)$ and $\textsc{type}(B) = \textsc{leaf}$: (i) poset changes, (ii) data structure changes.

Algorithm 4.8 Node Insertion Case 3.1: $\textsc{round}(A) = \textsc{round}(B)$ and $\textsc{type}(B) = \textsc{leaf}$.

Let $\textsc{parent}(B) = E$

Updated Properties:

$\textsc{type}(A) \leftarrow \textsc{leaf}$
$\textsc{parent}(A) \leftarrow E$
$\textsc{type}(B) \leftarrow \textsc{line}$
$\textsc{parent}(B) \leftarrow (A, E)$

Data Structure Updates:

replace B with A in $LST(E)$
insert B into $BST(A, E)$

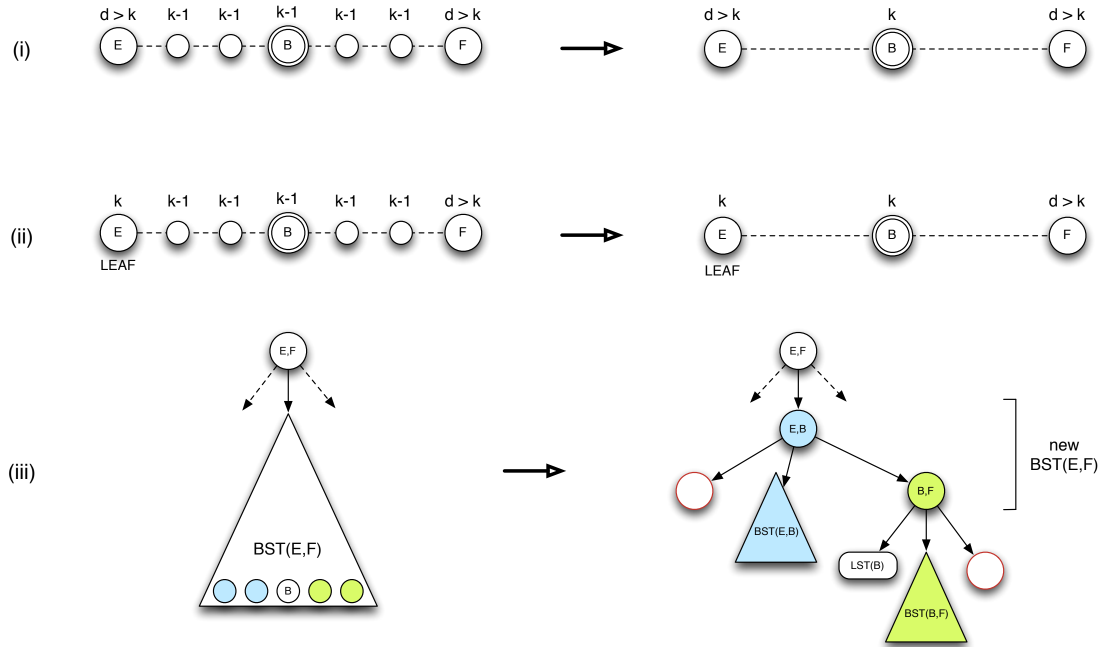

Figure 4.9: Node Insertion, Case 3.2.1 – B is not the final node, $\textsc{round}(A) = \textsc{round}(B)$ and $\textsc{type}(B) = \textsc{line}$: (i) poset changes for node B, (ii) data structure changes for node B.

Algorithm 4.9 Node Insertion Case 3.2.1: $\textsc{round}(B){+}1 < \textsc{round}(E)$, $\textsc{round}(F)$, or $\textsc{round}(B){+}1 = \textsc{round}(E) < \textsc{round}(F)$ and $\textsc{type}(E) = \textsc{leaf}$, where $\textsc{parent}(\mathrm{B}) = (\mathrm{E,F})$.

For node A, proceed as in Fig. 4.7 (ii) and (iii), Algorithm 4.7

Updated Properties:

$\textsc{round}(B) \leftarrow \textsc{round}(B){+}1$

Data Structure Updates:

split $BST(E,F)$ into $BST(E,B)$ and $BST(B,F)$
create new $BST(E,F)$
insert B into $BST(E,F)$

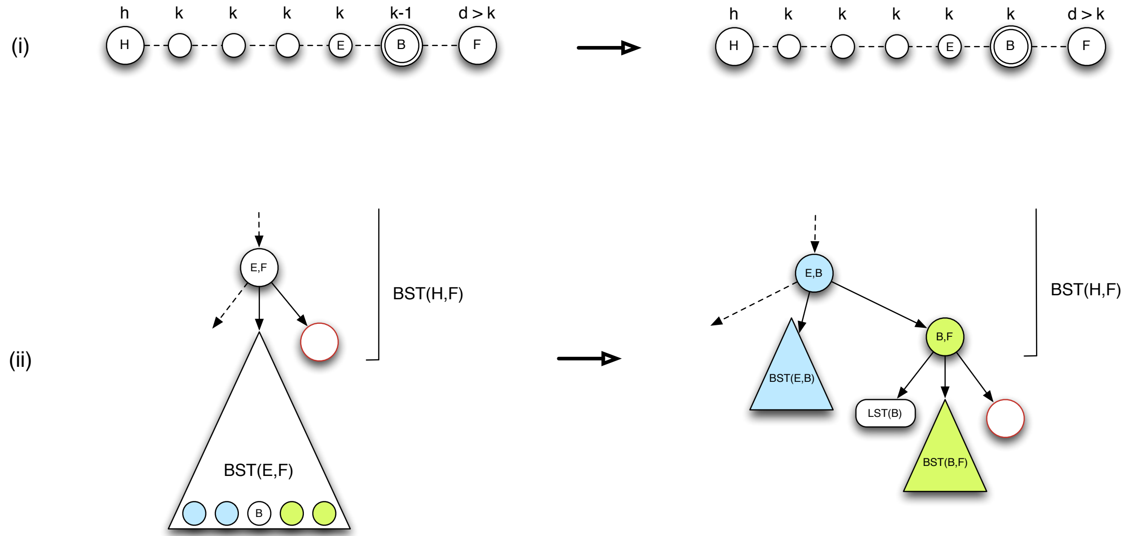

Figure 4.10: Node Insertion, Case 3.2.2 – B is not the final node, $\text{ROUND}(A) = \text{ROUND}(B)$ and $\text{TYPE}(B) = \text{LINE}$: (i) poset changes for node B, (ii) data structure changes for node B.

Algorithm 4.10 Node Insertion Case 3.2.2: $\text{ROUND}(B){+}1 = \text{ROUND}(E) < \text{ROUND}(F)$ and $\text{TYPE}(E) = \text{LINE}$, where $\text{PARENT}(B) = (E, F)$.

For node A, proceed as in Fig. 4.7 (ii) and (iii), Algorithm 4.7
Let $\text{PARENT}(E) = (F, H)$

Updated Properties:

$\text{ROUND}(B) \leftarrow \text{ROUND}(B){+}1$
$\text{PARENT}(B) \leftarrow (F, H)$

Data Structure Updates:

split $BST(E, F)$ into $BST(E, B)$ and $BST(B, F)$
insert B into $BST(F, H)$

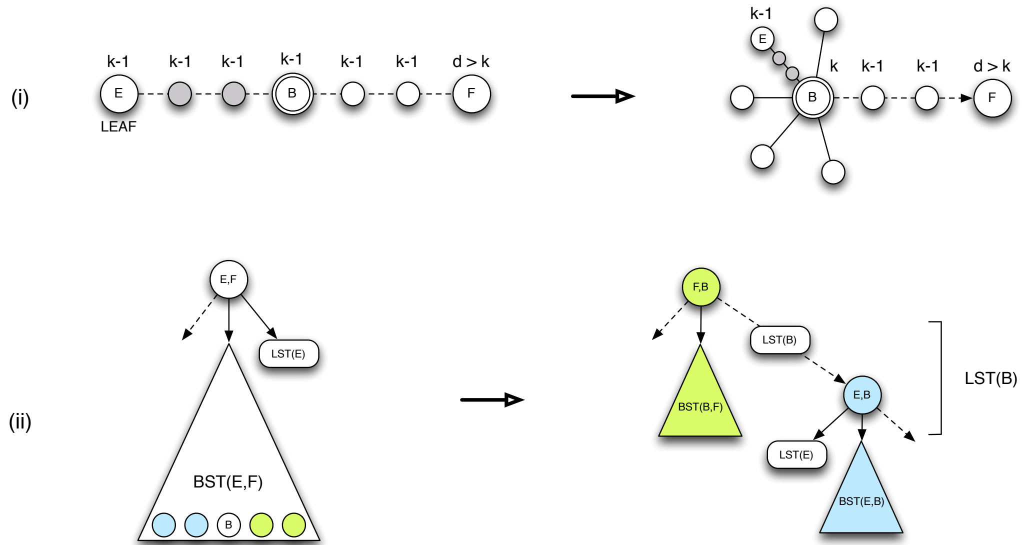

Figure 4.11: Node Insertion, Case 3.2.3 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B.

Algorithm 4.11 Node Insertion Case 3.2.3: ROUND(E) < ROUND(B)+1 < ROUND(F), where PARENT(B) = (E, F)

For node A, proceed as in Fig. 4.7 (ii) and (iii), Algorithm 4.7

Updated Properties:

ROUND(B) ← ROUND(B)+1
TYPE(B) ← LEAF
PARENT(B) ← F

Data Structure Updates:

split $BST(E, F)$ into $BST(E, B)$ and $BST(B, F)$
insert E into $LST(B)$
insert B into $LST(F)$

.

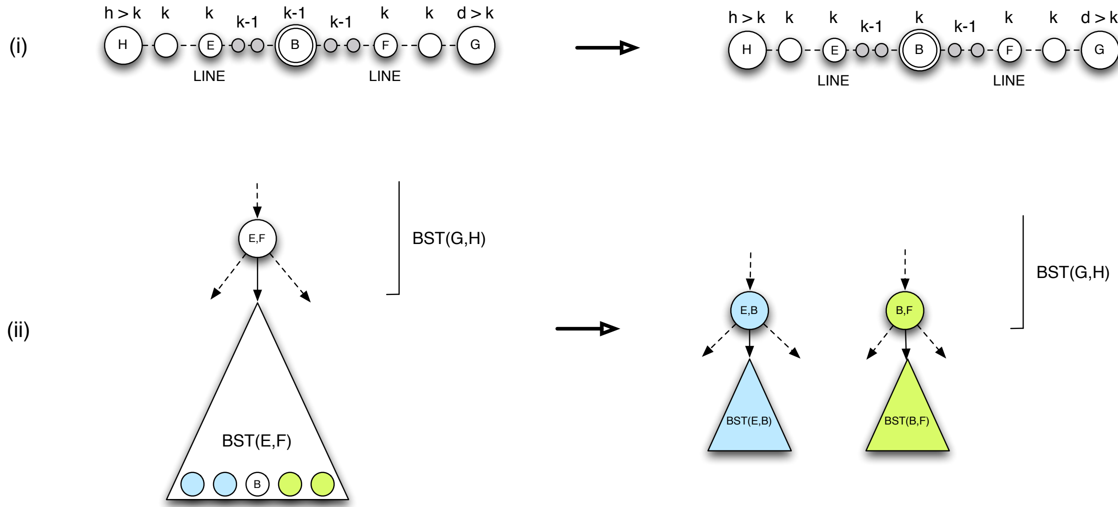

Figure 4.12: Node Insertion, Case 3.2.4 (a) – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B.

Algorithm 4.12 Node Insertion Case 3.2.4 (a): ROUND(E) = ROUND(B)+1 = ROUND(F) and TYPE(E) = TYPE(F) = LINE, where PARENT(B) = (E, F).

For node A, proceed as in Fig. 4.7 (ii) and (iii), Algorithm 4.7
Let PARENT(E) = PARENT(F) = (G, H)

Updated Properties:

ROUND(B) ← ROUND(B)+1
TYPE(B) ← LINE
PARENT(B) ← (G, H)

Data Structure Updates:

split $BST(E, F)$ into $BST(E, B)$ and $BST(B, F)$
insert B into $BST(G, H)$

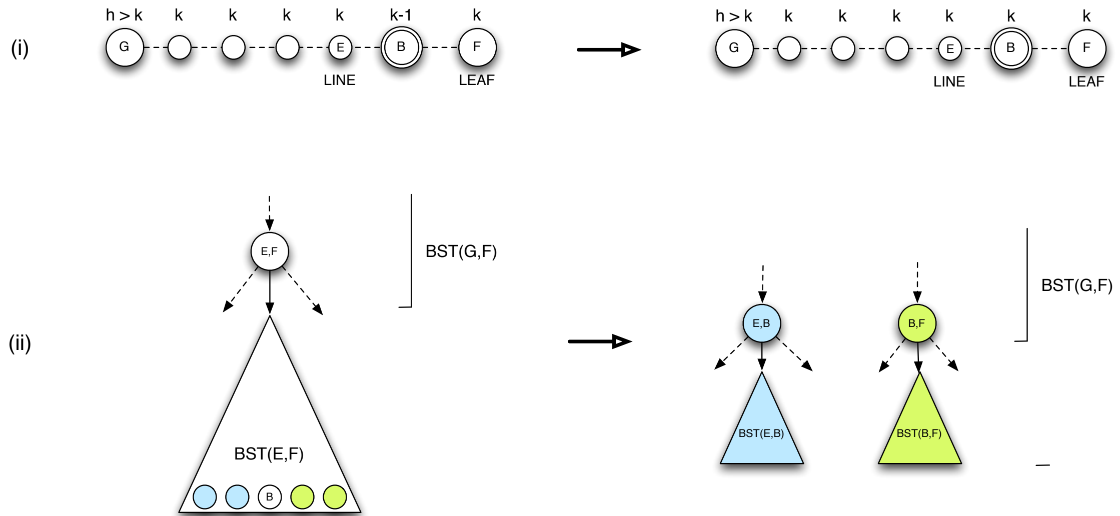

Figure 4.13: Node Insertion, Case 3.2.4 (b) – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B.

Algorithm 4.13 Node Insertion Case 3.2.4 (b): ROUND(E) = ROUND(B)+1 = ROUND(F), TYPE(E) = LINE, and TYPE(F) = LEAF, where PARENT(B) = (E, F).

For node A, proceed as in Fig. 4.7 (ii) and (iii), Algorithm 4.7
Let PARENT(E) = (F, G)

Updated Properties:

ROUND(B) ← ROUND(B)+1
PARENT(B) ← (F, G)

Data Structure Updates:

split $BST(E, F)$ into $BST(E, B)$ and $BST(B, F)$
insert B into $BST(F, G)$

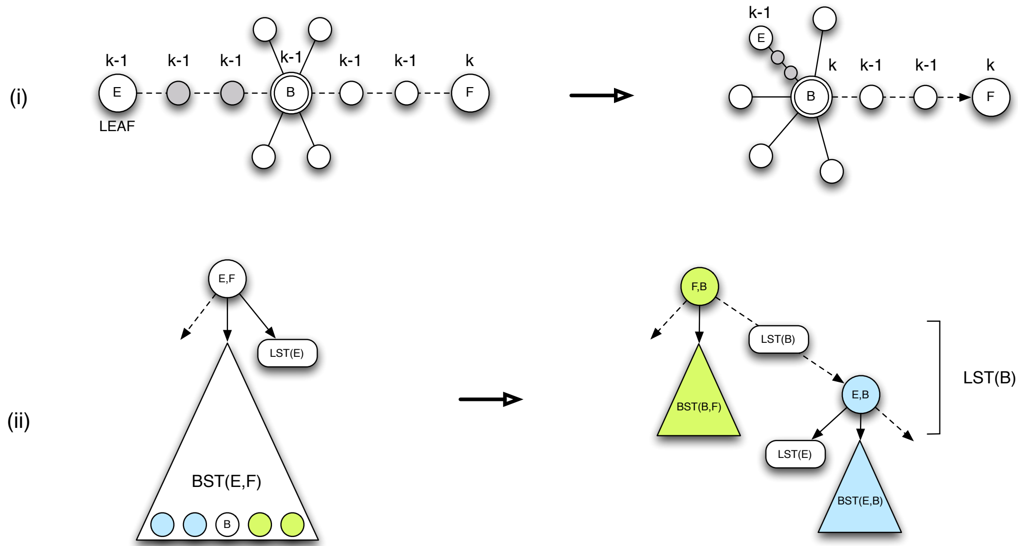

Figure 4.14: Node Insertion, Case 3.2.5 – B is not the final node, ROUND(A) = ROUND(B) and TYPE(B) = LINE : (i) poset changes for node B, (ii) data structure changes for node B.

Algorithm 4.14 Node Insertion Case 3.2.5: ROUND(E) $<$ ROUND(B)+1 = ROUND(F), where PARENT(B) = (E, F).

For node A, proceed as in Fig. 4.7 (ii) and (iii), Algorithm 4.7

Updated Properties:

ROUND(B) $\leftarrow$ ROUND(B)+1
TYPE(B) $\leftarrow$ LEAF

Data Structure Updates:

split $BST(E, B)$ into $BST(E, B)$ and $BST(B, F)$
insert E into $LST(B)$

RECURSIVE STEP: Insert B at **insertion point** F

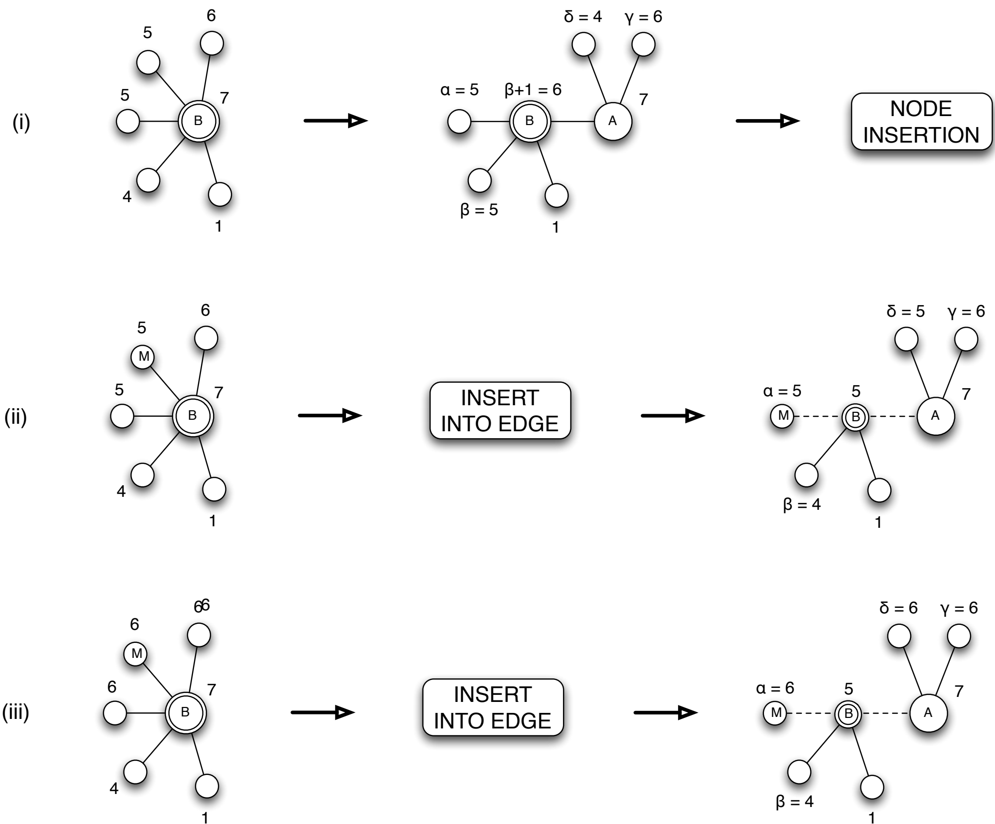

Figure 4.15: Edge Insertion, Case 1.1 Example – B is the final node, $\gamma > \alpha$ or $\gamma = \alpha$ and $\delta > \beta$ with $\textsc{round}(B) = k$: (i) $\alpha < \gamma$ and $\alpha = \beta$, (ii) $\alpha < \gamma$ and $\alpha > \beta$, (iii) $\alpha = \gamma$, $\delta > \beta$, and $\alpha > \beta$.

Algorithm 4.15 Edge Insertion Case 1.1: B is the final node, $\gamma > \alpha$ or $\gamma = \alpha$ and $\delta > \beta$ with ROUND$(B) = k$.

Updated Properties:

TYPE$(A) \leftarrow$ LEAF
ROUND$(A) \leftarrow k$
PARENT$(A) \leftarrow$ NONE (A is now the final node)
ROUND$(B) \leftarrow \beta + 1$

Data Structure Updates:

remove $C_i \in S_{stolen}$ from $LST(B)$
insert $C_i \in S_{stolen}$ into $LST(A)$

if $\alpha = \beta$ **then**
  Node Insertion: node B at insertion point A (final node)
else
  Let M be the unique node of ROUND α in $S_{not\ stolen}$
  remove M from $LST(B)$
  insert M into $LST(A)$
  insert B **into edge** (A, M) (Algorithm 4.4)
end if

Algorithm 4.16 Edge Insertion Case 1.2: B is the final node, $\alpha > \gamma$ or $\alpha = \gamma$ and $\beta \geq \delta$ with ROUND$(B) = k$.

Updated Properties:

ROUND$(A) \leftarrow \delta + 1$

Data Structure Updates:

remove $C_i \in S_{stolen}$ from $LST(B)$
insert $C_i \in S_{stolen}$ into $LST(A)$

if $\gamma = \delta$ **then**
  Node Insertion: node A at insertion point B (final node)
else
  Let M be the unique node of ROUND γ in S_{stolen}
  remove M from $LST(A)$
  insert M into $LST(B)$
  insert A **into edge** (B, M) (Algorithm 4.4)
end if

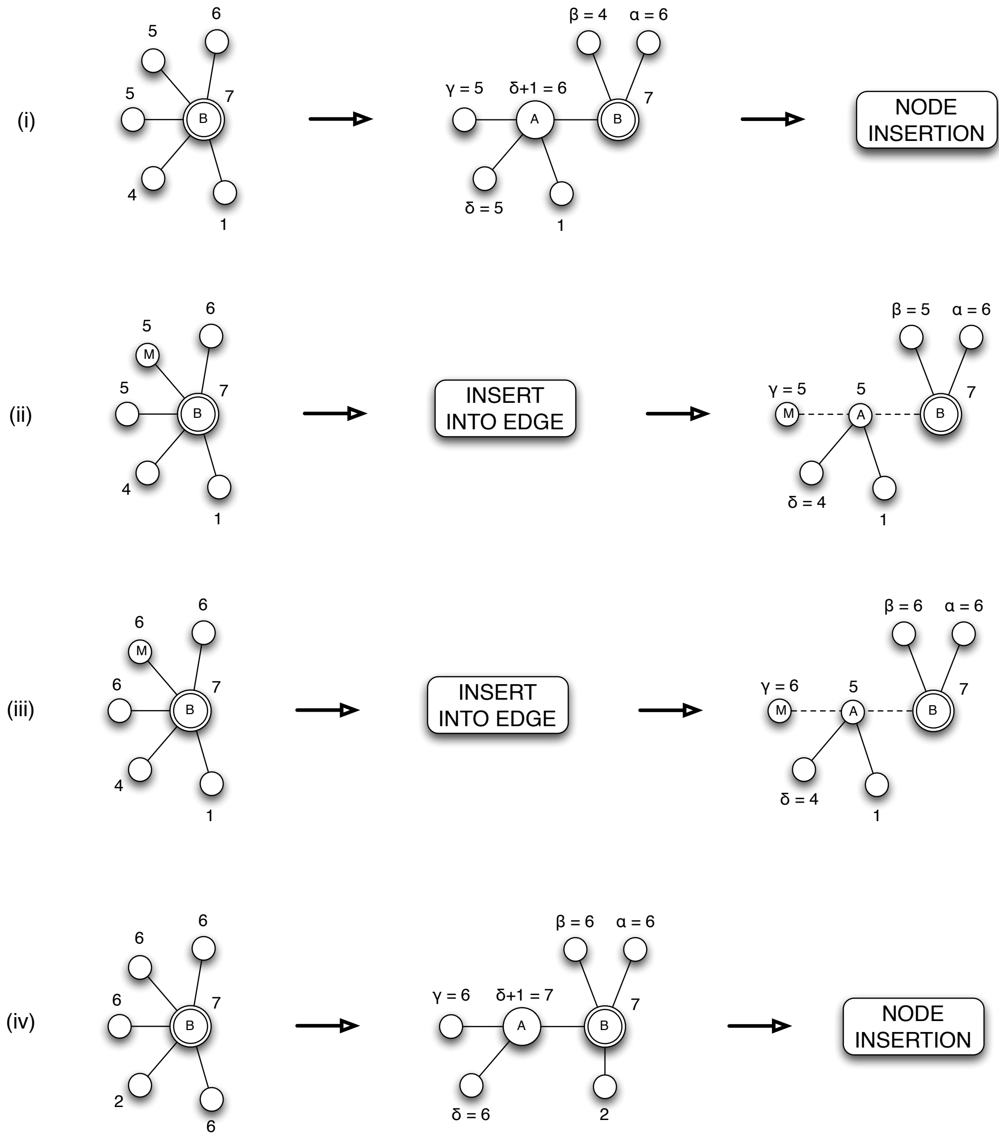

Figure 4.16: Edge Insertion, Case 1.2 Example – B is the final node, $\alpha > \gamma$ or $\alpha = \gamma$ and $\beta \geq \delta$ with $\text{ROUND}(B) = k$: (i) $\alpha > \gamma$ and $\gamma = \delta$, (ii) $\alpha > \gamma$ and $\gamma > \delta$, (iii) $\alpha = \gamma$, $\beta > \delta$, and $\gamma > \delta$, (iv) $\alpha = \gamma = \beta = \delta$.

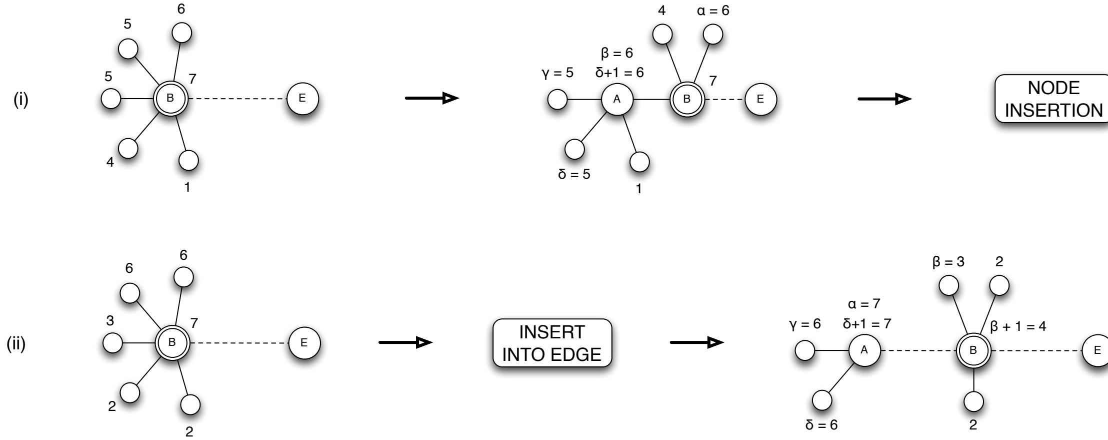

Figure 4.17: Edge Insertion, Case 2.1 Example – B is not the final node, TYPE(B) = LEAF, PARENT(B) = E, and A does not steal C_{line} : (i) if ROUND(A) $\leq$ ROUND(B) we perform a Node Insertion, and (ii) if ROUND(A) > ROUND(B) we insert B into edge (A, E).

Algorithm 4.17 Edge Insertion Case 2.1: B is not the final node, TYPE(B) = LEAF, PARENT(B) = E, and A does not steal C_{line}.

Updated Properties:

ROUND(A) $\leftarrow \delta + 1$
Recompute α and β by artificially considering A in $S_{not\ stolen}$
ROUND(B) $\leftarrow \beta + 1$

Data Structure Updates:

remove $C_i \in S_{stolen}$ from $LST(B)$
insert $C_i \in S_{stolen}$ into $LST(A)$
if ROUND(A) $\leq$ ROUND(B) **then**
  Node Insertion: node A at insertion point B
else
  remove B from $LST(E)$
  insert A into $LST(E)$
  insert B **into edge** (A, E) (Algorithm 4.4)
end if

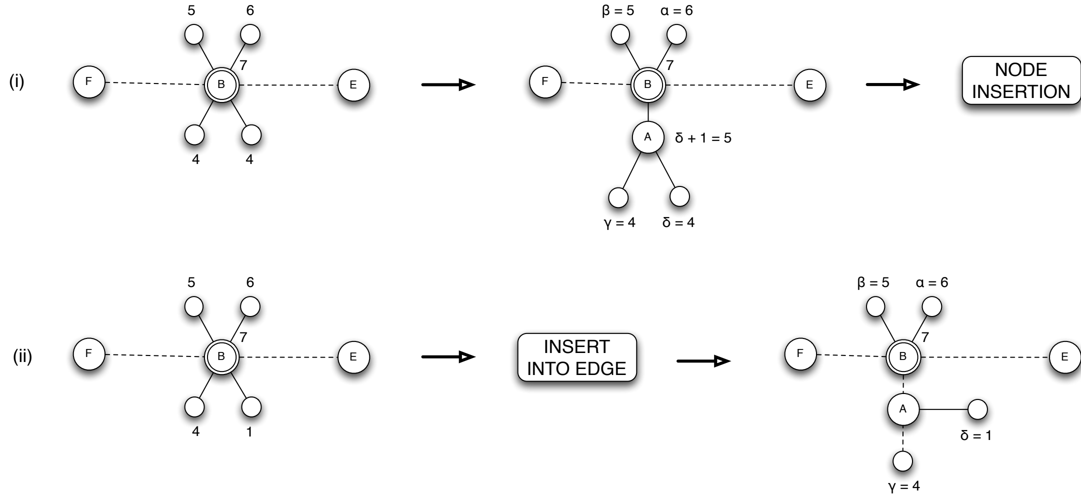

Figure 4.18: Edge Insertion, Case 3.1 Example – B is not the final node, TYPE(B) = LINE, PARENT$(B) = (E, F)$, and A does not steal C_{left} or C_{right} : (i) if $\gamma = \delta$ we perform a Node Insertion, otherwise (ii) we insert A into edge (B, M).

Algorithm 4.18 Edge Insertion Case 3.1: B is not the final node, TYPE(B) = LINE, PARENT(B) = (E, F), and A does not steal C_{left} or C_{right}.

Updated Properties:

ROUND$(A) \leftarrow \delta + 1$

Data Structure Updates:

remove $C_i \in S_{stolen}$ from $LST(B)$
insert $C_i \in S_{stolen}$ into $LST(A)$

if $\gamma = \delta$ **then**
  Node Insertion: node A at insertion point B
else
  Let M be the unique node of ROUND γ in S_{stolen}
  remove M from $LST(A)$
  insert M into $LST(B)$
  insert A **into edge** (B, M) (Algorithm 4.4)
end if

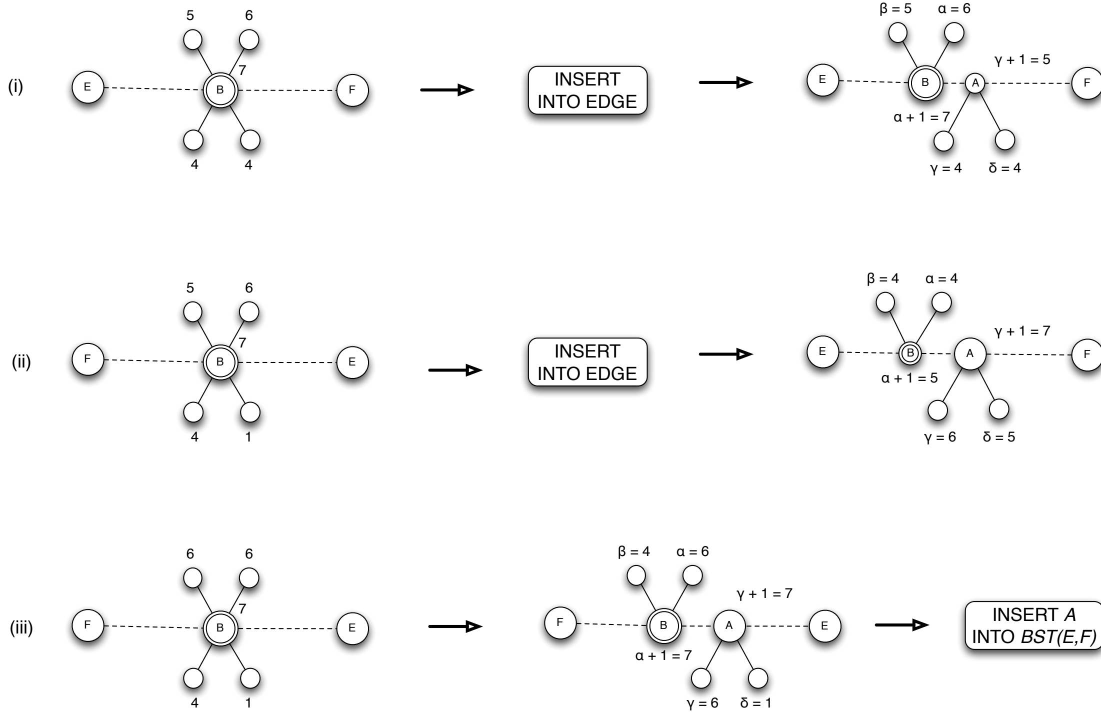

Figure 4.19: Edge Insertion, Case 3.2 Example – B is not the final node, TYPE(B) = LINE, PARENT(B) = (E, F), and A steals C_{right}, but not C_{left} (WLOG) : (i) if ROUND(A) < ROUND(B) we insert A into edge (B, F), (ii) if ROUND(A) > ROUND(B) we insert B into edge (A, E), and (iii) if ROUND(A) = ROUND(B) we insert A into $BST(E, F)$.

Algorithm 4.19 Edge Insertion Case 3.2: B is not the final node, TYPE(B) = LINE, PARENT(B) = (E, F), and A steals C_{right}, but not C_{left}.

Updated Properties:

ROUND(A) $\leftarrow \delta + 1$
ROUND(B) $\leftarrow \alpha + 1$

Data Structure Updates:

remove $C_i \in S_{stolen}$ from $LST(B)$
insert $C_i \in S_{stolen}$ into $LST(A)$
if ROUND(A) < ROUND(B) **then**
  insert A **into edge** (B, F) (Algorithm 4.4)
else if ROUND(A) > ROUND(B) **then**
  remove B from $BST(E, F)$
  insert A into $BST(E, F)$
  insert B **into edge** (A, E) (Algorithm 4.4)
else
  insert A into $BST(E, F)$
end if

4.7 Deletion

In this section we describe an efficient method of updating the Leaf-Line tree when deleting an element u from poset P. Although we usually ignore edge directions, we will require the following notion.

Definition 4.25. The **highest direction** H away from a node u corresponds to the incoming edge into u in the directed poset tree. If u is the Source, then no such direction exists.

Since we have restricted ourselves to posets of type SOURCE, every node will have exactly one incoming edge, except for the Source which has none. We can state analogous definitions for the *lowest direction* in SINK posets (where each node has exactly one outgoing edge, except for the Sink) or for the *central direction* in CENTRAL posets (which can be thought of as a SINK poset and a SOURCE poset connected by the central element [16]).

To identify the position of u in P we use a modified version of the Test Membership search algorithm (Algorithm 4.2). Thus, if node u is found, we augment the algorithm to return a pointer to u instead of the boolean value TRUE. The deletion procedure depends on the nodes in $LST(u)$, as well as the incident line directions C_{line}, C_{left}, and C_{right} defined in the previous section (some may not exist for a particular node u).

Similarly to the insertion, our goal is to update the Leaf-Line tree and the properties of the nodes such that the resulting tree is essentially the same (modulo differences in the balanced binary search trees) as we would have built using the induction algorithm in Section 4.3 had we started with the poset $P \setminus \{u\}$. In the algorithms provided in this section we describe the updates that need to occur both to the Leaf-Line tree and to the properties of nodes such that we maintain this invariant.

We describe a procedure for deleting element u from P that subsequently uses a recursive restructuring technique that updates the Leaf-Line tree after u has been deleted. Finally, we prove that our method is efficient (Section 4.7.3).

We assume the same figure conventions for deletion as we did in the previous section for insertion, with the modification that the node to be deleted D is now doubly-bordered, instead of the insertion point B.

4.7.1 Deletion Procedure

Suppose we want to delete node D from P. If D is the Source element, then we don't delete D, but instead record that we keep it as a *virtual* element in our poset (if it is later reinserted we will update this information accordingly). Consider the set $S_{neighbors} = \{C_1, \ldots, C_n\}$ where (D, C_i) is an edge in $LST(D)$. Let $S = S_{neighbors} \cup \{C_{left}, C_{right}\}$. If TYPE$(D)$ = LINE then C_{left}, C_{right} are the two corresponding directions on the line. If TYPE(D) = LEAF, then $C_{left} = C_{right} = C_{line}$. Finally, let N be the nearest neighbor of D in the highest direction H and let $S_{remaining} = S_{neighbors} \setminus \{C_k\}$, where (D, C_k) is the direction away from D corresponding to N. Due to the properties of Hasse diagrams, when we delete D, all former neighbors of D are now neighbors of N. We distinguish the following two cases for deletion:

Case 1: *N replaces D*

Suppose ROUND(N) $\leq$ ROUND(D). Then N is neither the PARENT of D if TYPE(D) = LEAF, nor one of the endpoints of the PARENT edge of D if TYPE(D) = LINE. We have the following cases:

Case 1.1: TYPE(N) = LINE Since ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LINE, N must be contained in the closest BST to D in the highest direction H. After deleting D, N survives as many iterations as D formerly did since all of D's former neighbors are now neighbors of N (N has the same number of directions leading away from it as D and each direction survives the same amount as it previously did). Thus, N replaces D and the contraction process would have built a structurally identical Leaf-Line tree, only without having N in the BST it occupied before deletion (Fig. 4.20). We present a formal description of this case in Algorithm 4.20.

Case 1.2: TYPE(N) = LEAF Since ROUND(N) $\leq$ ROUND(D), it follows that $N = C_i \in S_{neighbors}$. After deleting D, all of D's former neighboring directions are now neighboring N, except the one containing N itself. Thus, N's survival time may change since it now occupies the same position as D, only with one less adjacent direction. We apply the Restructuring algorithm for node N (Fig. 4.21 and Algorithm 4.21).

Case 2: *N absorbs D*

The closest neighbor to D in the highest direction H is either the PARENT of D if TYPE(D) = LEAF, or one of the endpoints of the PARENT edge of D if TYPE(D) = LINE. After deletion, all of the nodes previously leaf contracted into $LST(D)$ will now contract into $LST(N)$. Although N gains incident directions, their survival time is strictly lower than that of D (by construction). Since N essentially loses a direction who formerly contracted at ROUND(N)−1, we must use the Restructuring procedure to reason about the new contraction ROUND of N (Fig. 4.22 and Algorithm 4.22).

We summarize the cases of the Deletion procedure as follows:

Case 1 ROUND(N) $\leq$ ROUND(D)

- *1.1* TYPE(N) = LINE — **Fig. 4.20, Algorithm 4.20**
- *1.2* TYPE(N) = LEAF — **Fig. 4.21, Algorithm 4.21**

Case 2 ROUND(N) > ROUND(D) — **Fig. 4.22, Algorithm 4.22**

4.7.2 Restructuring Procedure

To complete the deletion procedure, we need to reason about the changes to the Leaf-Line tree that must occur so that we maintain our construction invariant. We want the final tree to be the same, regardless of whether we build it after the deletion or we modify the existing one.

Suppose that a node N which initially was contracted at ROUND k has lost a neighbor due to the deletion of some node D. Then, N may now be contracted in an earlier iteration. If $LST(N)$ contains edges $\{(N, C_1), \ldots, (N, C_k)\}$, consider the set $T_{neighbors} = \{C_1, \ldots, C_k\}$. We provide the following preliminary definition before describing our restructuring algorithm.

Definition 4.26. Let α and β be defined as follows:

- $\alpha = \max_{C_i \in T_{neighbors}} \{\text{ROUND}(C_i)\}$ is the maximum ROUND number of all the nodes in $T_{neighbors}$. If no such node exists, then $\alpha = 0$.
- Choose $F \in C_i \in T_{neighbors}$ such that ROUND$(F) = \alpha$. Then $\beta = \max_{C_i \in T_{neighbors} \setminus \{F\}}$ $\{\text{ROUND}(C_i)\}$ is the second largest ROUND number of all the nodes in $T_{neighbors}$ (counting duplicates). If $T_{neighbors}$ has fewer than 2 elements then $\beta = 0$.

Suppose that ROUND$(N) = k$ before deletion. For our restructuring procedure, we identify the following three cases:

Case 1: *N is the Final Node in the Contraction Process*

By Case 1.1 of the Edge Insertion procedure, in the former contraction process N had either exactly one, or at least three ROUND $k-1$ neighbors. Of these, at most one was lost. To reason about the new ROUND of N, we only need to take into account when the directions adjacent to N get contracted. This is analogous to Case 1 of the Node Insertion procedure where N as the final node in the contraction process and the 'inserted node' is F such that $(N, F) \in LST(N)$ and ROUND(F) $= \alpha$.

Case 2: *N is not the Final Node and* ROUND*(N) remains k*

If TYPE(N) = LEAF then $\alpha = \beta = k-1$ before a neighbor was lost. If $\alpha = \beta$ after the loss, as well, then they must both equal $k-1$. This implies that N is leaf contracted after exactly k iterations since N also has a PARENT that survives for more than k iterations (Fig. 4.23 (i)).

If TYPE(N) = LINE, then $\alpha = k-1$ before a neighbor was lost. If α remains $k-1$ after the loss, then N is line contracted after exactly k iterations since N also has a PARENT edge whose endpoints survive at least until the k-th iteration (Fig. 4.23 (ii)).

We provide a formal description of this case in (Algorithm 4.23).

Case 3: *N is not the Final Node,* TYPE*(N) =* LEAF*, and* ROUND*(N) becomes $k-1$*

Suppose that PARENT$(N) = E$. If TYPE(N) = LEAF then $\alpha = \beta = k-1$ before a neighbor was lost. Furthermore, the LEAF nodes corresponding to α and β must have two more additional directions that survive exactly $k-2$ iterations. This implies that if $\alpha > \beta$ after the neighbor was lost, then $\alpha = k-1$ and $\beta = k-2$.

After $k-2$ iterations, N only has two neighboring directions: the ones corresponding to α and E. Hence, N is line contracted at iteration $k-1$ and ROUND$(N) = k-1$. Let Q denote the node in $LST(N)$ with ROUND$(Q) = k-1$. Q will replace N as a LEAF of E and we create a new $BST(Q, E)$ which will contain N. Moreover, if $BST(Q, N)$ was created at iteration $k-1$, we include all of its nodes in the new $BST(Q, N)$. We have the following three possibilities:

Case 3.1: $BST(N, E)$ was created at iteration k This implies there exists a node M closest to N such that ROUND$(M) = k$ and TYPE(M) = LINE. First, N is line contracted into $BST(Q, M)$

at iteration $k-1$, together with any nodes on the path from Q to M that were previously contracted at this iteration (i.e. all the components of $BST(Q,N)$ and/or $BST(N,M)$ if they were previously created at iteration $k-1$). Subsequently, Q is leaf contracted into $LST(M)$ at iteration $k-1$ and finally, M is leaf contracted into $LST(E)$ at iteration k (replacing N). Thus, direction (N,E) survived k iterations before N lost a neighbor, but is now fully contracted after only $k-1$ iterations. This could affect the contraction time of E, and so we must run the Restructuring procedure for E next, considering that E has 'lost' a neighbor of ROUND k (Fig. 4.24 and Algorithm 4.24).

Case 3.2: $BST(N,E)$ was created at iteration $k-1$ This implies that E is the closest node to N of ROUND k or higher in direction (N,E). Similarly to above, N is line contracted into $BST(Q,E)$ at iteration $k-1$, together with all nodes in $BST(N,E)$ and possibly all the nodes in $BST(Q,N)$, but only if they were previously line contracted at iteration $k-1$. Subsequently, Q is leaf contracted into $LST(E)$ at iteration $k-1$ (replacing N). Thus, direction (N,E) survived k iterations before N lost a neighbor, but is now fully contracted after only $k-1$ iterations. This could affect the contraction time of E, and so we must run the Restructuring procedure for E next, considering that E has 'lost' a neighbor of ROUND k (Fig. 4.25 and Algorithm 4.25).

Case 3.3: $BST(N,E)$ was created at iteration $i < k-1$ Again, this implies that E is the closest node to N of ROUND k or higher in direction (N,E). Similarly to above, N is line contracted into $BST(Q,E)$ at iteration $k-1$, together with the nodes in $BST(Q,N)$, but only if they were previously line contracted at iteration $k-1$. Subsequently, Q is leaf contracted into $LST(E)$ at iteration $k-1$ (replacing N). Thus, direction (N,E) survived k iterations before N lost a neighbor, but is now fully contracted after only $k-1$ iterations. This could affect the contraction time of E, and so we must run the Restructuring procedure for E next, considering that E has 'lost' a neighbor of ROUND k (Fig. 4.26 and Algorithm 4.26).

Case 4: *N is not the Final Node,* TYPE*(N) =* LINE*, and* ROUND*(N) becomes $k-1$*

Suppose that PARENT(N) $= (E,F)$. If TYPE(N) = LINE then $\alpha = k-1$ before a neighbor was lost. Furthermore, the LEAF node corresponding to α must have two more additional directions that survive exactly $k-2$ iterations. This implies that if $\alpha < k-1$ after the neighbor was lost, then $\alpha = \beta = k-2$. N is line contracted after $k-1$ iterations between the two closest nodes L and R to N of ROUND k or higher in directions (F,N) and (N,E), respectively. Since nodes on the path between E and F have ROUND at most k, it follows that the only valid choices for L and R are nodes in $BST(E,F)$ (peers of N) and the two endpoints E and F. In all cases, all nodes between L and R previously contracted at iteration $k-1$ will now be line contracted into a new $BST(L,R)$ in the same step as N (Fig. 4.27 and Algorithm 4.27).

We summarize the cases of the Restructuring procedure as follows:

Case 1 N is the Final Node in the Contraction Process

Analogous to Case 1 of Node Insertion

Case 2 TYPE(N) = LEAF and $\alpha = \beta$ or TYPE(N) = LINE and $\alpha = k-1$

Fig. 4.23, Algorithm 4.23

Case 3 TYPE(N) = LEAF and $\alpha > \beta$

3.1 $BST(N, E)$ was created at iteration k **Fig. 4.24, Algorithm 4.24**

3.2 $BST(N, E)$ was created at iteration $k-1$ **Fig. 4.25, Algorithm 4.25**

3.3 $BST(N, E)$ was created at iteration $i < k-1$ **Fig. 4.26, Algorithm 4.26**

Case 4 TYPE(N) = LINE and $\alpha < k-1$ **Fig. 4.27, Algorithm 4.27**

4.7.3 Analysis

We first provide the following correctness and efficiency guarantees.

Lemma 4.27. The Restructuring procedure has query complexity proportional to the height of the Leaf-Line tree.

Proof. The claim is true for Case 1 by Lemma 4.21 and is trivially true for Case 2.

Updating the properties of a node is $O(1)$. Inserting into, removing from, and merging two BSTs takes $O(\log s)$ queries, where s is the size of the tree. Since $s \leq n$, these operations are $O(\log n)$ in the number of queries in the worst case. Inserting into an LST is $O(1)$ and removing from an LST is linear in its size which is bounded above by d, the maximum node degree in P. Note that finding the closest node in a given direction is $O(H)$ by the proof of Theorem 4.16. Also, the third to last operation in Algorithms 4.24, 4.25, and 4.26 is equivalent to either a void operation, copying a BST, or merging two separate BSTs (we include either all or none of the nodes of each of the two trees). Finally, computing α and β is $O(d)$ since they depend only on the nodes initially adjacent to N.

Thus, unless we are invoking the algorithm recursively (Case 3), we are performing a constant number of operations, each of which has query complexity either $O(d)$ or $O(\log n)$. By Corollary 4.4 and Theorem 4.13, this operation is $O(d + \log n) < O(H)$.

Suppose we are invoking the algorithm recursively. At each recursive call site, the ROUND of the element to be inserted is higher than the previous invocation by at least one unit since TYPE(N) = LEAF and we are invoking the procedure for PARENT(N). Thus, by Lemma 4.10, we can have at most $\log w$ nested recursive calls and so the maximum complexity of our algorithm is:

$$\log w \cdot O(d + \log n) \leq \log w \cdot O(OPT) \leq OPT \cdot O(\log w) = O(H).$$

□

Lemma 4.28. Deletion has query complexity proportional to the height of the Leaf-Line tree.

Proof. To find the position of D in the poset we run a slightly modified version of the Test Membership algorithm. By Theorem 4.14, the query complexity of this operation is $O(H)$, where H is the height of the Leaf-Line tree.

Analogously to the proof of the previous Lemma, updating node properties is $O(1)$, inserting into or removing from a BST is $O(\log n)$, inserting into an LST is $O(1)$ and removing from an LST is $O(d)$. During any run of the Deletion procedure, we perform a constant number of BST operations and property updates, $O(1)$ LST removals, and $O(d)$ LST insertions before invoking a separate procedure. Thus, by Lemma 4.4, we require only $O(d) + O(\log n) < O(H)$ queries notwithstanding external procedure calls.

The only external procedure invoked by Deletion is the restructuring protocol. This is never called more than once and our proof is concluded by Lemma 4.27. □

We are now ready to state our main result concerning the deletion procedure.

Theorem 4.29. When deleting an element u from P, the deletion procedure ensures the same Leaf-Line tree that would be generated if we started with poset $P \setminus \{u\}$ and ran the construction algorithm. Furthermore, the deletion procedure has query complexity proportional to the height of the Leaf-Line tree.

Proof. The deletion procedure handles all cases by the exhaustive definition of its cases. By construction, our deletion procedure affects all the necessary modifications (both to the properties of nodes and to the Leaf-Line tree) to ensure that we obtain the same data structure after a dynamical operation as we would if we re-ran the contraction algorithm with the updated poset as input. Finally, by Lemma 4.28 we require at most $OPT \cdot O(\log w)$ queries to efficiently update the Leaf-Line tree using our deletion algorithms. □

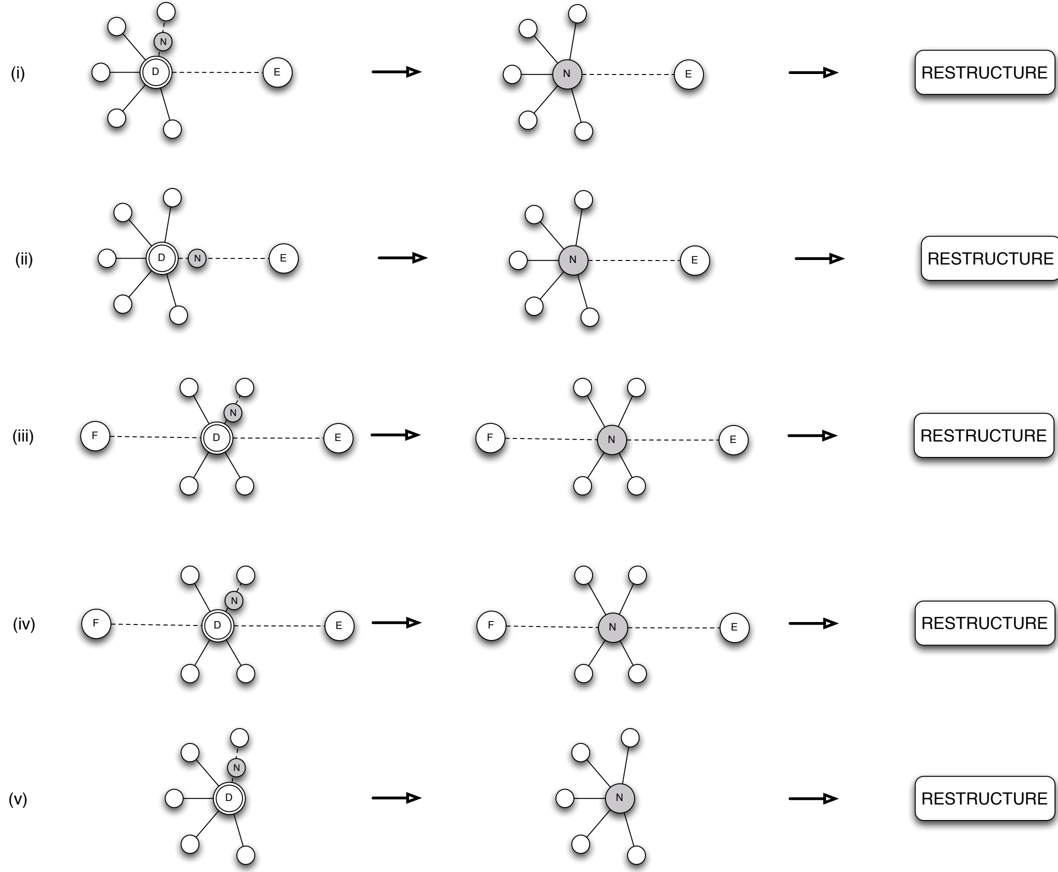

Figure 4.20: Deletion, Case 1.1 – ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LINE with PARENT(N) = (G, H) : (i) TYPE(D) = LEAF, D is not the final node, and N is not in C_{line} direction, (ii) TYPE(D) = LEAF, D is not the final node, and N is in the C_{line} direction, (iii) TYPE(D) = LINE and N is not in the C_{left} or C_{right} directions, (iv) TYPE(D) = LINE and N is in the C_{left} direction (WLOG), (v) D is the final node in the contraction process.

Algorithm 4.20 Deletion Case 1.1: ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LINE with PARENT(N) = (G, H)

Updated Properties:

TYPE(N) $\leftarrow$ TYPE(D)
ROUND(N) $\leftarrow$ ROUND(D)
PARENT(N) $\leftarrow$ PARENT(D)

Data Structure Updates:

remove N from $BST(G, H)$
insert $C_i \in S_{remaining}$ into $LST(N)$
remove D from parent structure PS (if D is not final node)
insert N in parent structure PS (if D is not final node)

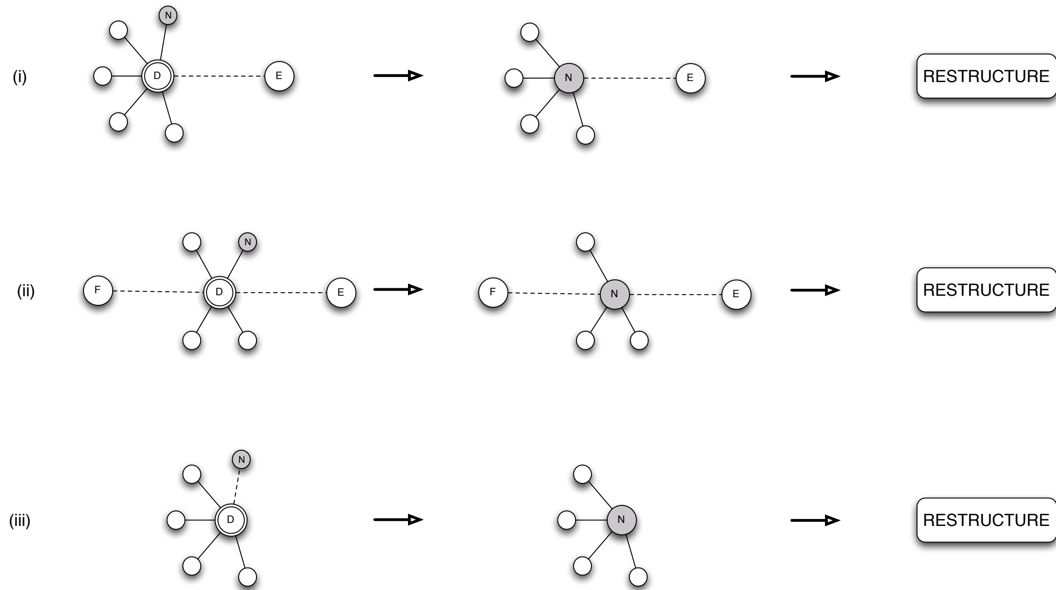

Figure 4.21: Deletion, Case 1.2 – ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LEAF with PARENT(N) = D : (i) TYPE(D) = LEAF and D is not the final node, (ii) TYPE(D) = LINE, (iii) D is the final node in the contraction process.

Algorithm 4.21 Deletion Case 1.2: ROUND(N) $\leq$ ROUND(D) and TYPE(N) = LEAF with PARENT(N) = D

Updated Properties:

TYPE(N) $\leftarrow$ TYPE(D)

Data Structure Updates:

remove N from $LST(D)$
insert $C_i \in S_{remaining}$ into $LST(N)$
remove D from parent structure PS (if D is not final node)
insert N in parent structure PS (if D is not final node)
Apply **Restructuring Procedure** to node N

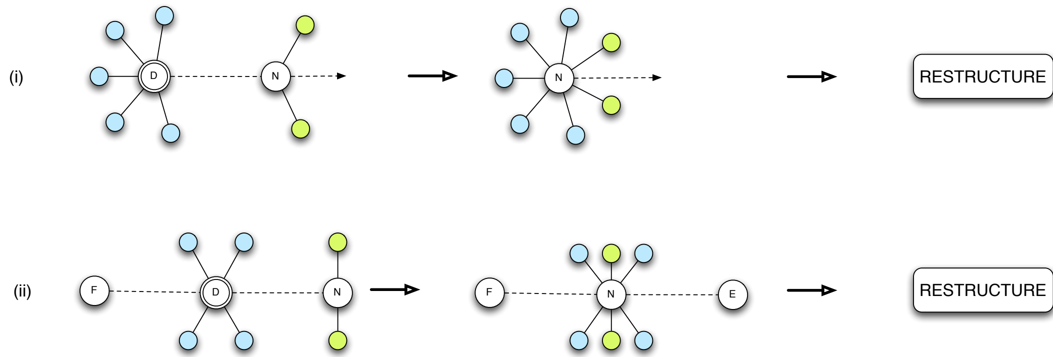

Figure 4.22: Deletion, Case 2 – ROUND(N) $>$ ROUND(D) : (i) TYPE(D) = LEAF and PARENT(D) = N, (ii) TYPE(D) = LINE and PARENT(D) = (F, N).

Algorithm 4.22 Deletion Case 2: ROUND(N) $>$ ROUND(D)

Data Structure Updates:

insert $C_i \in S_{neighbor}$ into $LST(N)$
remove D from parent structure PS (if D is not final node)
Apply **Restructuring Procedure** to node N

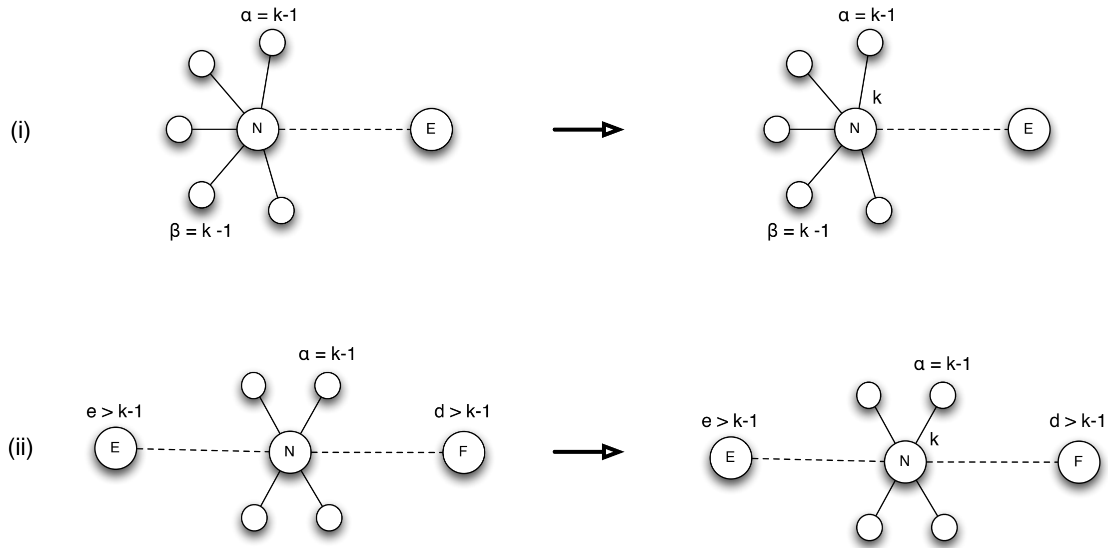

Figure 4.23: Restructuring, Case 2 – TYPE(N) = LEAF, PARENT(N) = E, and $\alpha = \beta$ OR TYPE(N) = LINE, PARENT(N) = (E, F), and $\alpha = k - 1$: (i) assigning ROUND(N) = k for TYPE(N) = LEAF, (ii) assigning ROUND(N) = k for TYPE(N) = LINE.

Algorithm 4.23 Restructuring Case 2: TYPE(N) = LEAF and $\alpha = \beta$ OR TYPE(N) = LINE and $\alpha = k - 1$

Updated Properties:

ROUND(N) $\leftarrow k$

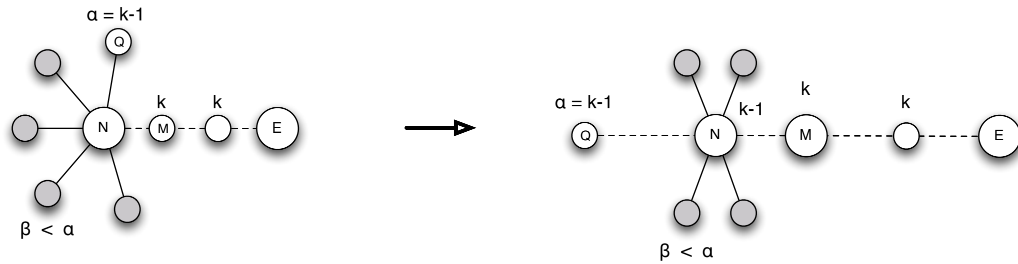

Figure 4.24: Restructuring, Case 3.1 – TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration k .

Algorithm 4.24 Restructuring Case 3.1: TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration k.

Let M be the closest node to N in $BST(N, E)$
Let Q be the node corresponding to α (edge (Q, N) in $LST(N)$)

Updated Properties:

TYPE(M) $\leftarrow$ LEAF
PARENT(Q) $\leftarrow M$
TYPE(N) $\leftarrow$ LINE
ROUND(N) $\leftarrow k - 1$
PARENT(N) $\leftarrow (Q, M)$

Data Structure Updates:

remove M from $BST(N, E)$
insert M into $LST(E)$
remove Q from $LST(N)$
insert Q into $LST(M)$
remove N from $LST(E)$
create new $BST(Q, M)$
insert all nodes of ROUND $k - 1$ from $BST(Q, N)$ and $BST(N, M)$ into $BST(Q, M)$
insert N into $BST(Q, M)$
Apply **Restructuring Procedure** to node E

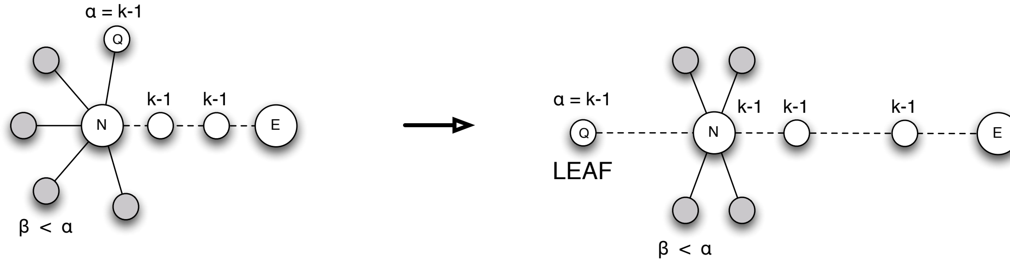

Figure 4.25: Restructuring, Case 3.2 – TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration $k - 1$.

Algorithm 4.25 Restructuring Case 3.2: TYPE(N) = LEAF, PARENT(N) = E, $\alpha > \beta$, and $BST(N, E)$ was created at iteration $k - 1$.

Let Q be the node corresponding to α (edge (Q, N) in $LST(N)$)

Updated Properties:

PARENT(Q) $\leftarrow E$
TYPE(N) $\leftarrow$ LINE
ROUND(N) $\leftarrow k - 1$
PARENT(N) $\leftarrow (Q, E)$

Data Structure Updates:

remove Q from $LST(N)$
insert Q into $LST(E)$
remove N from $LST(E)$
create new $BST(Q, E)$
insert all nodes of ROUND $k - 1$ from $BST(Q, N)$ and $BST(N, E)$ into $BST(Q, E)$
insert N into $BST(Q, E)$
Apply **Restructuring Procedure** to node E

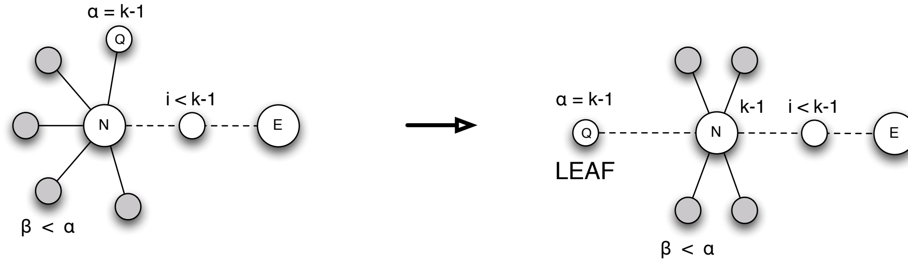

Figure 4.26: Restructuring, Case 3.3 – TYPE(N) = LEAF, PARENT$(N) = E$, $\alpha > \beta$, and $BST(N, E)$ was created at iteration $i < k - 1$.

Algorithm 4.26 Restructuring Case 3.3: TYPE(N) = LEAF, PARENT$(N) = E$, $\alpha > \beta$, and $BST(N, E)$ was created at iteration $i < k - 1$.

Let Q be the node corresponding to α (edge (Q, N) in $LST(N)$)

Updated Properties:

PARENT$(Q) \leftarrow E$
TYPE$(N) \leftarrow$ LINE
ROUND$(N) \leftarrow k - 1$
PARENT$(N) \leftarrow (Q, E)$

Data Structure Updates:

remove Q from $LST(N)$
insert Q into $LST(E)$
remove N from $LST(E)$
create new $BST(Q, E)$
insert N into $BST(Q, E)$
Apply **Restructuring Procedure** to node E

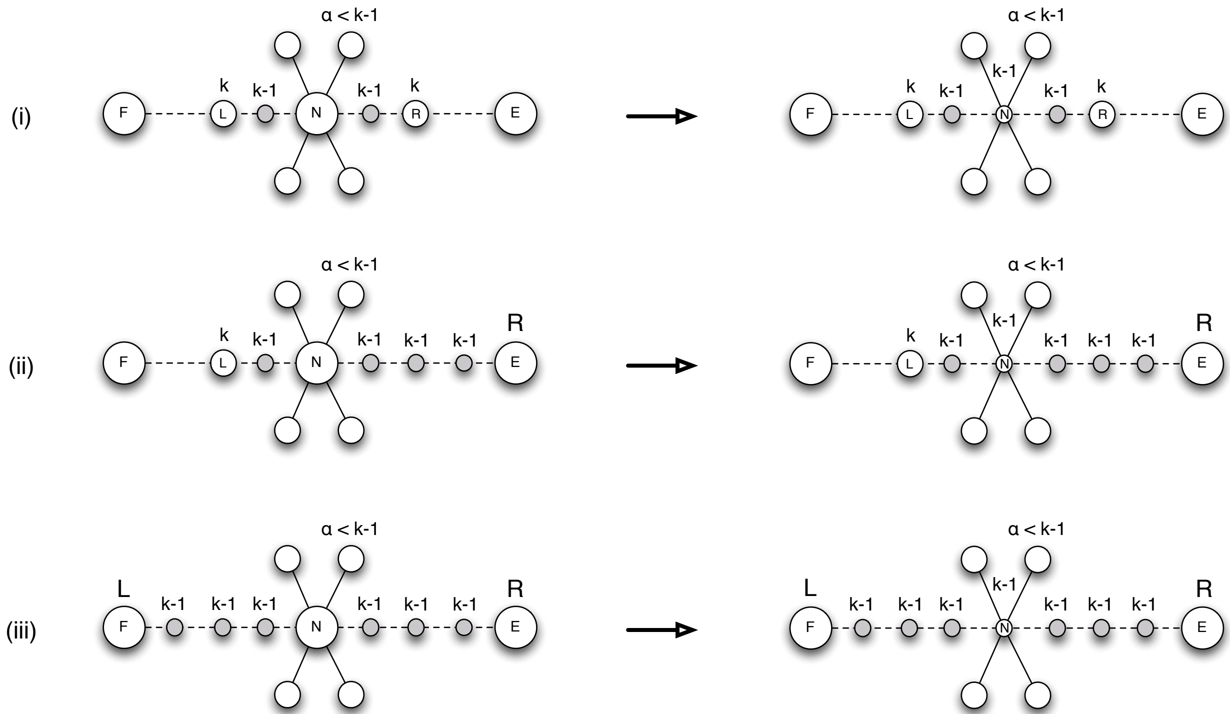

Figure 4.27: Restructuring, Case 4 – TYPE(N) = LINE, PARENT(N) = (E, F), $\alpha < k-1$, L and R as defined below.: (i) $L \neq F$ and $R \neq E$, (ii) $L \neq F$ and $R = E$ (WLOG), (iii) $L = F$ and $R = E$.

Algorithm 4.27 Restructuring Case 4: TYPE(N) = LINE, PARENT(N) = (E, F), $\alpha < k-1$.

Let L be the closest node of ROUND at least k in direction (F, N) (L may be F)
Let R be the closest node of ROUND at least k in direction (N, E) (R may be E)

Updated Properties:

ROUND(N) $\leftarrow k-1$
PARENT(N) $\leftarrow (L, R)$

Data Structure Updates:

remove N from $BST(E, F)$
create new $BST(L, R)$
insert all nodes of ROUND $k-1$ from $BST(L, N)$ and $BST(N, R)$ into $BST(L, R)$
insert N into $BST(L, R)$

Chapter 5

Conclusions

5.1 Contributions

We have developed the Leaf-Line tree, a data structure that supports efficient *insert*, *delete*, *test membership*, and *neighbor* operations on a set S of n elements drawn from a partial order whose underlying Hasse diagram is a tree. This is a fundamental problem in data structures with a wide variety of practical applications such as filesystem synchronization and rankings in sports, college admissions, and conference submissions. Our data structure provides an efficient solution to this problem.

The Leaf-Line tree is space efficient – it requires $O(n)$ space where n is the size of the dynamic set – and can be built in $O(n)$ time given a Hasse diagram of the poset. The height of the Leaf-Line tree is within a factor of $O(\log w)$ of the height of the optimal search strategy for static tree-like posets, where w is the *width* of the partial order. All above operations – *insert*, *delete*, *test membership*, and *neighbor* – have query complexity within a constant factor of the height of the tree. Although the width may be as large as n in some cases (the *width* represents a natural obstacle when searching in a partial order), we guarantee close to optimal behavior and better than linear performance whenever possible.

5.2 Future Work

5.2.1 Asymptotical Optimality

Although the Leaf-Line tree provides an efficient way of searching in tree-like partial orders, our performance is still a factor of $O(\log w)$ away from optimal. We believe this is a acceptable price to pay for being able to maintain our bounds on the height of the tree under dynamic operations. However, one of our goals for future research is closing this gap between our performance and the optimal.

Furthermore, we are presently unaware of what the true total complexity of this problem is. Any significant progress in this direction could show that an asymptotically optimal data structure for

searching in dynamic partially ordered sets has yet to be found. Nevertheless, it could also provide validation for our method if research shows that our performance bounds are within a constant factor of the best polynomial-time solution possible.

5.2.2 The Node Query Model

For the static problem of searching in partial orders, linear time and space solutions exist for both the Node and Edge Query models. When solving the dynamic setting of this problem, we built the Leaf-Line tree assuming the Edge Query model. We subsequently seek an analogous data structure for searching dynamic partial orders under the Node Query model. Any new solution in this case would first require an extension to the current Node Query model. The extension should allow for a fast way to find the insertion point of a new element in the poset (similar to the extension to the Edge Query model which allowed us to implement the *Find Location* search procedure).

5.2.3 Other Types of Partial Orders

The current version of the Leaf-Line tree only provides efficient *insert*, *delete*, *test membership*, and *neighbor* operations for tree-like partially ordered sets. We wish to define a generalization of our construction algorithm that will allow us to maintain similar data structures for much larger classes of partial orders. The new algorithm must identify meaningful information about nodes and structure in more general directed acyclic graphs, as opposed to trees.

Finally, if we successfully extend the contraction algorithm to provide efficient decompositions for a larger class of partial orders, then a new natural problem arises:

> *Build a data structure that supports efficient* insert*,* delete*,* test membership*, and* neighbor *operations for a dynamic set of elements drawn from a partial order which supports both* **node and edge** *insertions and deletions.*

Motivating finding a solution for this new problem are applications to programming languages research such as compiler design optimization and maintaining efficient structural decompositions of dynamic control-flow graphs.

Bibliography

[1] Ben-Asher, Y., Farchi, E., and Newman, I. (1999) Optimal search in trees. *SIAM Journal on Computing*, **28**, 2090–2102.

[2] Carmo, R., Donadelli, J., Kohayakawa, Y., and Laber, E. (2004) Searching in random partially ordered sets. *Theoretical Computer Science*, **321**, 41–57.

[3] Cormen, T. E., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001) *Introduction to Algorithms, 2nd Ed.*. MIT Press, McGraw-Hill Book Company.

[4] Daskalakis, C., Karp, R. M., Mossel, E., Riesenfeld, S., and Verbin, E. (2007) Sorting and selection in posets. *arXiv:0707.1532*.

[5] Faigle, U. and Turán, G. (1988) Sorting and recognition problems for ordered sets. *SIAM Journal of Computation*, **17**, 100–113.

[6] Goodrich, M. T. and Tamassia, R. (2000) *Data Structures and Algorithms in Java, 2nd Ed*. John Wiley and Sons.

[7] Iyer, A. V., Ratliff, H. D., and Viyajan, G. (1988) Optimal node-ranking of trees. *Information Processing Letters*, **28**, 225–229.

[8] Iyer, A. V., Ratliff, H. D., and Viyajan, G. (1991) On an edge-ranking problem of trees and graphs. *Discrete Applied Mathematics*, **30**, 43–52.

[9] Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Searching and Sorting, 2nd Ed*. Addison-Wesley Professional.

[10] Laber, E. and Molinaro, M. (2008) An approximation algorithm for binary searching in trees. *Automata, Languages and Programming*, **5125**, 459–471.

[11] Laber, E. and Nogueira, L. T. (2001) Fast searching in trees. *Electronic Notes in Discrete Mathematics*, **7**, 1–4.

[12] Lam, T. W. and Yue, F. L. (1998) Optimal edge ranking of trees in linear time. *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 436–445.

[13] Linial, N. and Saks, M. (1985) Every poset has a central element. *Journal of Combinatorial Theory*, **A 40**, 195–210.

[14] Linial, N. and Saks, M. (1985) Searching ordered structures. *Journal of Algorithms*, **6**, 86–103.

[15] Mozes, S., Onak, K., and Weimann, O. (2008) Finding an optimal tree searching strategy in linear time. *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1096–1105.

[16] Onak, K. and Parys, P. (2006) Generalization of binary search: Searching in trees and forest-like partial orders. *FOCS*, pp. 379–388.

[17] Prisco, R. D. and Santis, A. D. (1993) On binary search trees. *Information Processing Letters*, **45**, 249–253.

[18] Schaffer, A. A. (1989) Optimal node ranking of trees in linear time. *Information Processing Letters*, **33**, 91–99.

[19] Schroder, B. S. W. (2003) *Ordered Sets: An Introduction*. Boston: Birkhauser.

[20] Zhou, X. and Nishizeki, T. (1995) Finding optimal edge-rankings of trees. *In Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 122–131.