Department of Electrical and Computer Engineering

UNIVERSITY OF THESSALY

MASTER'S THESIS

STUDY OF CONFLUENT PERSISTENCE FOR DATA STRUCTURES IN THE POINTER MACHINE MODEL.

Panayiotis Vlastaridis

Supervisors: Panayiotis Bozanis, Associate Professor
Panagiota Tsompanopoulou, Assistant Professor
Dimitrios Katsaros, Lecturer

VOLOS, GREECE

February 2014

Table of Contents

Prologue

This thesis is a study of a technique proposed by Sébastien Collette, John Iaconoy and Stefan Langermanz in their paper “Confluent Persistence Revisited” published in the Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms 2012. In this paper they propose a generic method for any data structure in the pointer machine model in order to perform efficient confluent persistence operations in terms of space and time costs.

More specifically, their method proves that confluent persistence can be achieved at a logarithmic amortized cost with the restrictions: i) the data structure is in the bounded in-degree model, widely used in previous work ii) merging of two different versions of the data structure that contain the same nodes is not allowed. Under these restrictions following their method outperforms previous methods by a O(n/log n)-factor improvement.

In order to help the reader to gain a better insight in their technique we show examples using Binary Search Trees.

Chapter 1 is an introduction to the basic concepts of data structures in the pointer machine model and Confluent persistence. Also previous work and applications of confluent persistence are discussed. Chapter 2 deals with the presentation of the various data structures used in order to augment any data structure in the pointer machine model to a confluent persistent one. In Chapter 3 we discuss how it is possible to navigate through the group of data structures presented in the previous chapter and run examples using Binary Search Trees. Finally, in Chapter 4 the complexity analysis of the performance is discussed.

GENERAL TERMS: Data Structures, Confluent Persistence, Pointer Machine Model.

ADDITIONAL TERMS AND PHRASES: Binary Search Trees, Merging data structures, Bounded in-degree model, linked data structure, ephemeral data structure, partial persistence, full persistence.

Chapter 1 Introduction - Basic Concepts.

Contents

1.1 Data Structures in the pointer machine model.

A data structure is a well-defined collection of data that supports a set of well-defined operations to be performed upon it. The operations performed in a data structure can be partitioned into two groups: query operations, that do not modify the structure, and updates that do modify the structure.

The data structures that we are going transform to confluent persistent data structures must belong to the pointer machine model. Data structures in the pointer machine model are described in Driscoll et al. [1] where they are referred as linked data structures. Data structures in this model have the following characteristics:

1. They are composed of a finite collection of nodes.
2. Each node contains a fixed number of fields.
3. The fields of a node are information fields or pointer fields.
4. Information fields can contain values of a specified type (integers, booleans, bit, etc.). The key information field is used to identify the node.
5. Pointers fields contain pointers to another node or a null value indicating no node. We are going to refer to pointers as edges as this is a convention used in the paper describing the technique we are going to study [2]. If a node x contains a pointer field to another node y, we call y the successor of x and x the predecessor of y.
6. All nodes in a structure are of the same type i.e. have exactly the same number of fields for each field type they contain.
7. Each structure has a unique node designated as the root which provides access to the linked data structure. While temporary pointers are allowed during the execution of an operation, no pointers into the structure are permitted to be maintained between operations other than to the roots.
8. Operations performed are a sequence of atomic operations all commencing from the entry point of the data structure i.e. its root. The atomic operations that we can perform are:
    a. Create a node.
    b. Delete a node.
    c. Follow an edge reading a pointer field.

d. Read value of an information field.
e. Change field value of a node (including edges). Changing one field is stipulated to cost O(1) time.

9. They must also support a merge operation where two roots are specified and at the end of the operation the second root is no longer considered to be a root; this implies that the connected components of the two roots have been merged. The merge operation is also a sequence of the above mentioned atomic operations

1.2. Confluent Persistence

Ordinary data structures are *ephemeral* as operations are performed only on their latest version, and in the case of an update, the current version is changed, and the past version is lost. Persistence is the concept, of allowing the execution of operations not just on the current version, but on past versions as well. There are three forms of persistence, partial persistence, full persistence and confluent persistence.

With *partial persistence* one can execute query operations on any previous version of the structure, but updates are only performed on the current (latest) version. In this model of persistence the versions can be represented as a linear list (timeline) since a new version is being derived only from the latest version in the list.

In *full persistence*, update and query operations may be performed on any version of the data structure that ever existed. In the case of an update on a past version, a new version is created, derived from that past version. In full persistence, versions no longer can be represented as a linear list, their topology can be represented by the version tree. In a version tree every node is a version, and an edge is drawn from every version to the version it was derived from.

Finally, confluent persistence is the strongest form of persistence, it allows the same operations as in full persistence but it also introduces merge or otherwise referred as meld operations. Merge operations take two versions and merges them into one to produce a single new version. These types of operations are also referred as confluent updates. In confluent persistence, since each version is derived from one or more previous versions, the graph, formed by the versions, derivation relationship is no longer a tree, it is now the version directed acyclic graph (DAG).

1.3. Applications of persistence

Below, we list some of the most important applications of persistence:

1) Algorithm Design. In the area of computational geometry various algorithms use persistence to find efficient solutions to various problems. More specifically in [3] planar point location, algorithms are developed utilizing persistent search trees. Planar point location involves preprocessing a polygonal subdivision of the plane defined by n line segments so that, given a point q, the polygon containing q, can be determined quickly. As an example planar point location helps given GPS coordinates to find the region on a map where you are located. Also in [4], ray shooting problem is solved in a more efficient way using persistence. Ray shooting is the problem of determining the first intersection between a ray (directed line segment) and a collection of polygonal or polyhedral obstacles. In order to process queries efficiently, the set of obstacle polyhedra is usually preprocessed into a data structure.

2) Problem solving. Again an example from computational geometry where persistence is used to solve optimally the segment intersection problem as shown in [5]. In more detail this involves a widely-used technique in computational geometry where by performing a sweep with a vertical line through a data set (set of points, set of segments, etc.), the set of items intersected by the line at every sweep step are stored in a data structure. If this data structure is made persistent, one can then retrieve all the information that has been swept through, by going back to previous versions.

3) Control Version Systems where one wants to keep records of all versions of some information. Typical examples include version control systems [6], [7] such as Subversion, Mercurial, Git and CVS. In these systems, all the versions of some files are stored, in order to be able to go back to any previous version of any file as well as merge two versions of a file or a directory of files.

4) Finally, in code debugging as shown in [8] where persistent data structures are used so one can check, at any point during the execution of code what the value of any variable was.

1.4. Previous work

Finding ways to efficiently perform partial and full persistence gained popularity in the 1980’s [7]. Persistence was firstly formally studied in [1] where a technique that converts data structures that belong in the pointer machine model and have bounded in-degree, was developed with a constant factor overhead (or otherwise referred as slowdown) in time and space for every operation. Overhead or slowdown is the extra time or space required to perform an atomic operation in a persistent data structure additional to the time and space required to perform the same atomic operation in the non-persistent implementation of the data structure. In the paper showing the above technique, Driscoll et al. [1] mentioned confluent persistence as an open problem. Confluent persistence was formally defined in 1994 [9].

Finally in 2003, Fiat and Kaplan [10] gave the first generic method to obtain confluently persistent structures. But their results show that the overhead may be often suboptimal since a merge of two versions can double the size of the data structure, and therefore in k merges, the persistent data structure will increase to a size equal to $2^{\Omega(k)}$. Fiat and Kaplan also prove a lower bound of confluent persistence, ($U \cdot e(D)$) bits to store all versions where U is the total number of atomic update operations performed and e(D) is the “effective" depth of the version DAG, defined as the logarithm of the maximum number of different paths from the root of the DAG to each of its versions. In terms of upper bounds, their method has a space complexity of $O(U \cdot e(D) \cdot \log U)$ bits and an overhead of $O(e(D) + \log U)$ time per operation on the confluent persistent data structure. Their results are considered as suboptimal since the effective depth can be linear in the number of updates performed fig.1.1. In worst-case scenarios their results show that there is not an improvement over simply copying the whole data structure to create each new version.

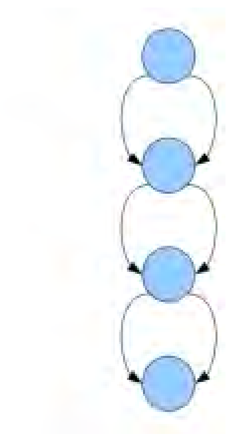


Figure 1.1
From [11] Version DAG where a string is concatenated with itself repeatedly and in such case e(D) is linear to the number of update operations (confluent updates).

1.5. Restrictions of the technique.

The technique we are going to study is an improvement in terms of complexity over the previous work presented above but with an additional restriction that confluent operations (merging) are allowed only between versions with no shared data nodes. Under this restriction O(log(n)) overhead for both time and space is possible following their technique. Also, their method does not take any measures to prevent the merging of data structures with common nodes and its the responsibility of the user to ensure that this never happens. The restriction of disjoint merges can be avoided by replacing one of the two common nodes with a duplicate before executing a merge operation but this may induce a space penalty equal to the size of the data structure being represented.

Another restriction which is also present in the previous work mentioned above is the constant bound for the incoming degree of the nodes for the data structure under confluent persistence representation. This restriction can also be bypassed as we can replace each node with high in-degree by a reversed balanced binary search tree (BST) of depth O(log maxdeg) where maxdeg is the maximum in-degree of a node. fig. 1.2. Incoming pointers are now pointing to the leaves of this reversed balanced BST and the root has the same number of outgoing pointers as the node to be replaced by this structure. This bypass of the second restriction induces only an O(log maxdeg) time penalty for the atomic operation “following a pointer”. There is no space penalty as the number of constant sized nodes added is linear to the number of incoming edges. Also other operations such as creating, deleting a node do not induce a time or space penalty as we can ensure that all outgoing and incoming edges are deleted before deleting the node itself.

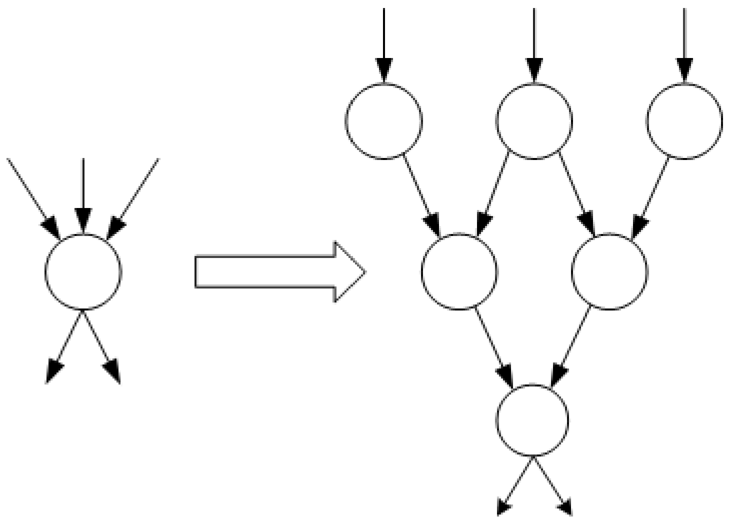


Figure 1.2. Transforming a node with maximum in-degree 3 (maxdeg) to a set of nodes with in-degree one using reversed balanced BSTs.

1.6 Binary search Trees

Throughout this thesis we are going to use Binary Search Trees (BSTs) to run examples of the technique involved in order to ease the reader understand the algorithms applied as well as the complexity analysis. We chose BSTs as they are easy to run and also their complexity analysis is simple as they are asymptotically linear O(n). The BST can be represented in the pointer machine model. A node in our BST implementation contains four fields:

- a key field which is used to identify the node with distinct values selected from a totally ordered set. Duplicates of key fields are not allowed.
- an information field where the data associated with the node is stored.
- left and right pointer fields to the left and right children (sons) of the node.

The nodes are arranged so the binary-search-tree property (symmetric order) is met, if x is a node, the key field in x is greater than all key fields in the items in its left subtree and smaller than all all key fields in the items in the right subtree of x. The BST has constant in-degree with value 1 and the tree root is the only entry node.

Operations of BSTs include:

- **Searching:** We first visit the root node where we examine if the Key field (k) we are searching matches, it is greater or smaller than the current node's key field (c). If k is greater than c we follow the right pointer visiting the right son node and in the opposite case we visit the left son node. Then we recursively apply the above comparison until we find a match or we end up in a leaf i.e. there is no match for our query.
- **Insert:** We follow the above steps as in searching until we reach a leaf, where we place our node and update it's parent left or right pointer field respectively.
- **Update information field:** After locating the node by searching, we change the value of its information field.
- **Delete:** There are three cases after performing a search operation to locate the node of interest: 1) Deleting a node with no children where we just simply remove it from the tree. 2) Deleting a node with one son where the node we want to remove its replaced by its son. 3) Deleting a node with two sons where the node we want to remove is replaced by the smallest node (i.e. with the smallest key field) in its right subtree.
- **Merge two binary search trees:** where we create a sorted list with the nodes of the second BST and then we use this list to insert its nodes to the first BST one after another.

In the case of confluent persistence for all the above operations we must determine also the version we are interested to perform the operation or in the case of a confluent merge the two versions we would like to merge.

Chapter 1 References.

[1] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. Journal of Computer and System Sciences, 38(1):86 – 124, 1989. Initially presented at STOC'86.

[2] Sebastien Collette, John Iacono, and Stefan Langerman. Confluent Persistence Revisited. In Symposium on Discrete Algorithms (SODA), pages 593-601, 2012.

[3] N. Sarnak and R. Tarjan. Planar point location using persistent search trees. Commun. ACM, 29(7):669–679, 1986.

[4] H. Kaplan, N. Rubin, and M. Sharir. Linear data structures for fast ray-shooting amidst convex polyhedra. Algorithmica, 55(2):283–310, 2009.

[5] A. Bouroujerdi and B. Moret. Persistence in Computational Geometry. In Proc. 7th Canadian

Conference in Computational Geometry, pages 241–246, 1995

[6] E.D. Demaine, S. Langerman, and E. Price. Confluently persistent tries for efficient version control. Algorithmica, 57(3):462–483, July 2010. Special issue of selected papers from 11th Scandinavian Workshop on Algorithm Theory, 2008.

[7] Joachim Gudmundsson (Ed.): Algorithm Theory - SWAT 2008, 11th Scandinavian Workshop on Algorithm Theory, Gothenburg, Sweden, July 2-4, 2008, Proceedings. pages 160-172, Springer 2008 Lecture Notes in Computer Science ISBN 978-3-540-69900-2

[8] Z. Liu. A persistent runtime system using persistent data structures. In SAC '96: Proceedings of the 1996 ACM symposium on Applied Computing, pages 429–436, New York, NY, USA, 1996.
ACM.

[9] J. Driscoll, D. Sleator, and R. Tarjan. Fully persistent lists with catenation. J. ACM, 41(5):943–959, 1994.

[10] A. Fiat and H. Kaplan. Making data structures confluently persistent. J. Algorithms, 48(1):16–58, 2003.

[11] MIT Computer Science and Artificial Intelligence Laboratory Lecture Notes on Advanced Data Structures Spring 2012. web-link: http://courses.csail.mit.edu/6.851/spring12/scribe/lec1.pdf

Chapter 2 The Data Structures involved.

Contents

2.1 Global Version Tree.

The first data structure we need to construct is a global version DAG (V), associated to the data structure (D) we would like to augment as a confluent persistent data structure. In our examples to follow we presume that the versions in the version DAG have out-degree at most 2. This can be easily circumvented in case we want to create more than 2 versions from a single version by adding a constant number of dummy version nodes. For every operation in a version of D a new version vertex is added to V as a child to the version we performed an operation. A new version added to V is declared by an integer ID, its value assigned by a version counter that we increment each time a new version is created. Given the above, we expect each version vertex in V, other that the root, to be either the child of the version we are performing an operation or the child of two versions in the case of a confluent merge between two versions of D.

Furthermore we maintain the following pointers in our global version DAG, V:

1. Bi-directional pointers between parent and child versions in order to determine a child version from which version it was derived.
2. A pointer from each version vertex v in DAG to the root node of D in version v in order to gain access to D via a version ID.

2.2 Global Version Link-Cut Tree.

The second data structure we need to maintain is a subgraph of V. This subgraph is a forest of disjoint trees, that includes all the version vertices of V but only a subset of its edges. This data structure maintains the properties of a link-cut tree as described in [1]. V and F are connected with bi-directional pointers between their corresponding versions so from a version v in V we can directly access the version v in F.

Every new version added to V is also added in F in a similar manner. The edges of V that are not present in F are excluded as follows. After a confluent merge of two versions in F we find their lowest common ancestor and we delete (or cut) the two edges connecting this lowest common ancestor to its descendants, an example is shown on Figure 2.1. In case where no lowest common ancestor is found, we do not cut any edge. Excluding these edges ensures that there are no cycles in subgraph F.

In a link-cut tree the lowest node (with smallest ID-key field) on any path between two nodes can be determined in O(logn) time. In our case the lowest node in the path between two version nodes we would like to merge is their lowest common ancestor. Also it is possible to check if two nodes belong to the same tree of the forest in O(logn) time [1]. Therefore the cost of maintaining F along with V has a complexity of O(logn) amortized time.

Finally if V is implemented as a sorted list figure 2.1 adding a new version if V and F can be achieved also in O(logn) amortized time.

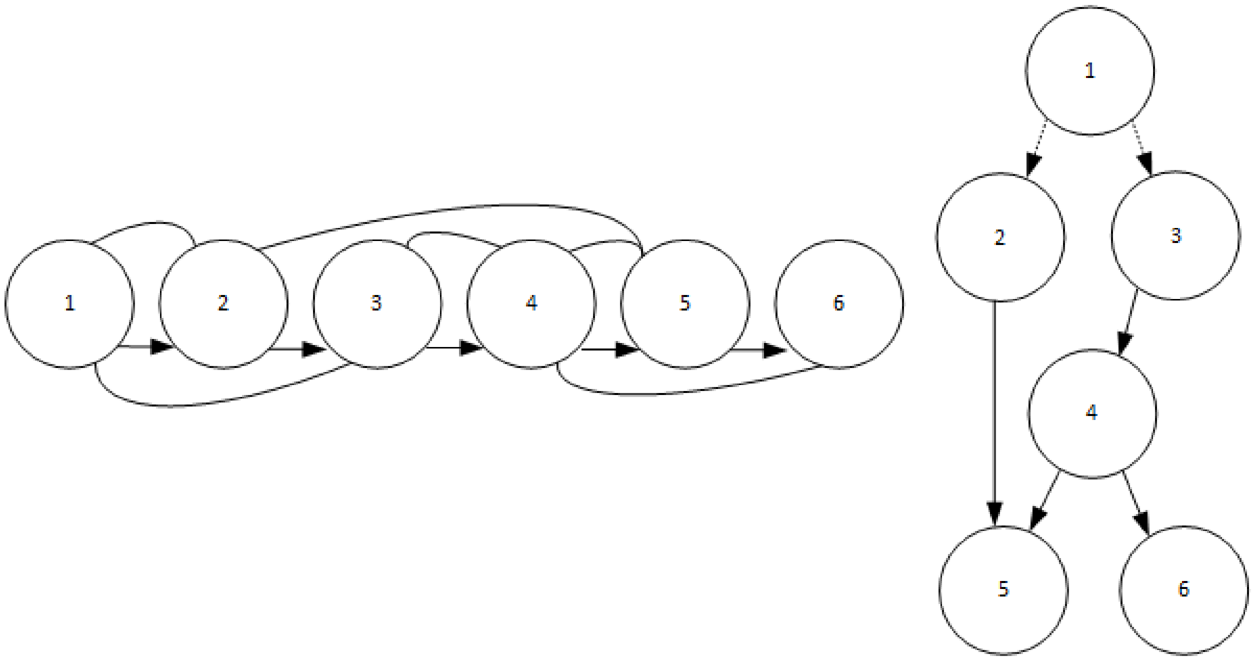


Figure 2.1

V represented as a sorted list, arcs represent relationship between versions, arrows belong to sorted list, and F where the merge between versions 2 and 4 produce version 5.

2.3 Compressed Local Version Tree.

Thirdly for each node d of D we maintain a Local Version Tree, LVT(d), data structure which is also a subgraph of V. This subgraph of V includes only the version vertices of V where node d is present in D and also their corresponding edges. This local version tree is in fact a tree as stated by the following lemma.

Lemma 1. The local version DAG of a node is a tree.

Proof. LVT is a connected graph since every new version added to LVT is a child of a version in LVT. If we assume that LVT(d) is not a tree then there exists a version vertex that is a child of two versions but this can be only the result of a confluent merge of two versions containing the same node which is forbidden by the constraint mentioned in 1.5 .

In order to reduce the space and time required to maintain LVTs for each node, a compressed local version tree cLVT is stored instead. A cLVT(d) of a node d contain only versions where one or more of the fields of d were changed, figure 2.2. A vertex present in cLVT is termed explicit and the nodes for which all fields remained unchanged and are represented by their parents are termed implicit. Representatives of a version vertex v in a cLVT are denoted by rep(v). All the fields (key, data and pointer fields) of a node d in a version v are stored in the corresponding explicit node v in cLVT.

2.4 Edge Overlays.

Similarly to cLVT we maintain a local version tree for every edge between two nodes in D. This edge local version tree is termed as a compressed overlay tree cOV(a,b) (where a and b represent the nodes that this edge is connecting in D). The cLVT is a connected subset of the intersection of cLVT(a) and cLVT(b), figure 2.3. In order to maintain cOVs we must also apply the following rules adding explicit nodes when required:

1. cOV(a,b) must always contain an explicit vertex for each version were the edge has been created or deleted.
2. Each explicit vertex of cOV(a,b) must appear in both cLVT(a) and cLVT(b) (but not necessarily reversely).

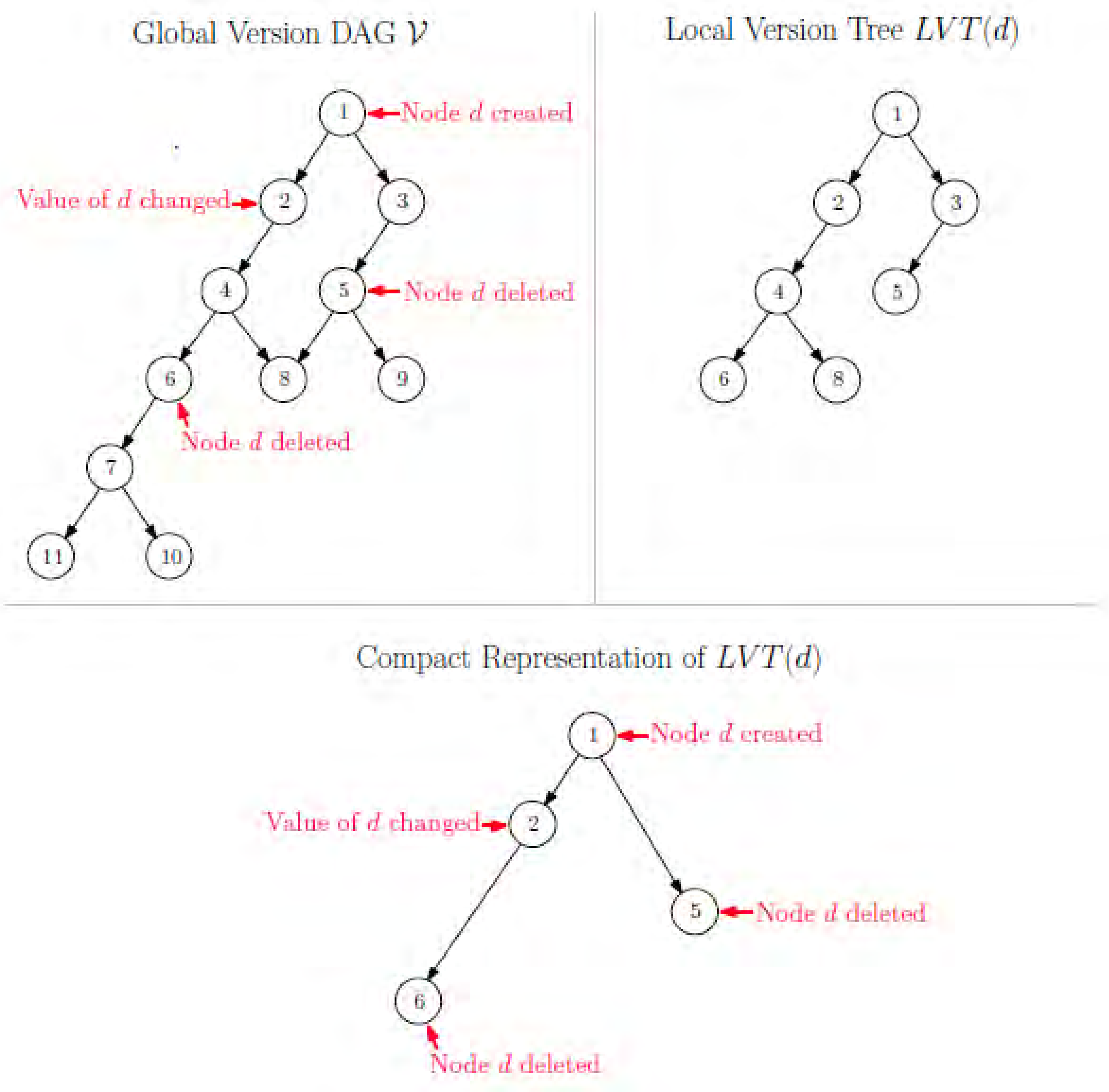


Figure 2.2

LVT and cLVT representations from [3]. Nodes 3, 4 and 8 of LVT(d) are implicit nodes in the cLVT(d) and are represented by nodes 1 and 2 respectively.

2.5 Node-edge navigation structure.

In order to navigate through the set of the above discussed data structures we need pointers interconnecting them, that they will allow us to visit target nodes in cLVTs in order to retrieve the values of their fields. The set of pointers we are going to apply is termed the node-edge navigation structure.

There are three types of pointers that we are going to apply:

1. **Version-edge pointers.** For each version v in a cLVT(a) in which an edge (a,b) existed, add pointers from v in cLVT(a) to its representative rep(a,b)(v) in cOV(a,b). For an explicit node in cLVT(a) that is not preset in cOV(a,b) we need to add a pointer to its representative rep(a,b)(v) in cOV which is its nearest ancestor present in both cLVT(a) and cOV(a,b)
2. **Edge-version pointers.** pointers from all versions v in cOV(a,b) to the same version v in cLVT(a) and cLVT(b).
3. **Inverse edge-version pointers.** a list of pointers, so that from a version node in cLVT(b) we can retrieve the list of its incoming edge-version pointers for edges that connect node b to their ancestors in D.

In order to reduce the space required to store all the above pointers the technique of fractional cascading [2] is applied. We mark certain vertices in cLVTs as senators, for which we apply the above pointers. Two senators in a root-to-leaf path in a cLVT are at distance λ+1, where λ is a small constant defined later. A senator node b, must appear explicitly in all its incoming cOV(a,b) as well as in the corresponding cLVT(a). If these newly inserted versions in cLVT(a) happen to become senators we will propagate senators to their neighbors and so on.

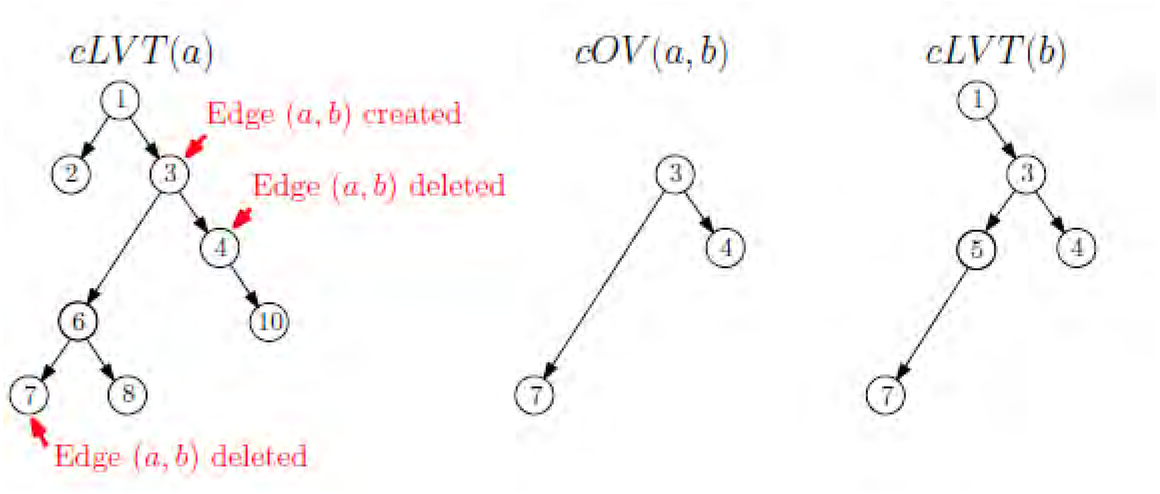


Figure 2.3

cOV representation from [3] for an edge between nodes a and b.

Chapter 2 References.

[1] D. Sleator and R. Tarjan. A data structure for dynamic trees. In Proceedings of the thirteenth annual ACM symposium on Theory of computing, STOC ’81, pages 114–122, 1981.

[2] B. Chazelle and L. Guibas. Fractional cascading: I. a data structuring technique. Algorithmica, 1:133–162, 1986.

[3] Sebastien Collette, John Iacono, and Stefan Langerman. Confluent Persistence Revisited. In Symposium on Discrete Algorithms (SODA), pages 593-601, 2012.

Chapter 3 Binary Search Tree Example - Navigation.

Contents

3.1 Synopsis of atomic operations.

Before we run examples using Binary Search Trees we would like to resume the rules governing the building of the set of data structures we described in the previous chapter.

Every atomic operation in our data structure will create a new version except the searching operation. A new version node is added in V and in F , as a child of the version(s) that we performing an operation on. For example in case we insert a node in version 5, a new version presumably number 6 will be the son of version 5 in V and F, similarly if we merge version 3 and 4 the new version derived from this merge assuming it receives from our version counter the id 5, version 5 will be the son of both 3 and 4.

Then we create an explicit version node in cLVT(d), where d is the root of the data structure D in the new version v. This is done in order to keep pointers from each version in V to an explicit node in the corresponding root cLVTs, as discussed in section 2.1. Also we add nodes for the new version v in cLVTs for all nodes in D that have their fields altered due to the atomic operation we are performing. For example if we insert node 6 as a left son of node 10 in version 2 and the new version created is 5 then we must include node version 5 in both cLVTs of node 6 (a new cLVT(6) is created) and 10 as node 10 has its left pointer field altered. Adding a new node in a cLVT requires to check if it will become a senator. This done by checking if it has a senatorial ancestor in the cLVT at distance greater than or equal $\lambda+1$, in which case it needs to become a senator. In case it becomes a senator we also add the required version edge pointers, edge version pointers and if necessary we update the list of the inversion edge-version pointers. This may require to add new nodes in cLVTs and cOV where we check again if new nodes in cLVTs need to become senators and so on.

Altering, creating or deleting pointer fields of a node in D requires also to add explicit nodes in the cOV representing this pointer field - edge. Explicit nodes for a version v of an edge in cOV require that are present also in cLVTs of the nodes this edge connects. This may insert new nodes or senators in cLVT as above.

In the case of a confluent merge, we need to find a lowest common ancestor and cut two edges in F. This merge operation then follows the above rule as it will include alterations in many pointer fields of various nodes.

3.2 Running an example with a Binary Search Tree.

We will run an example using Binary Search Tree (BST) as D, which we would like to augment to a confluent persistent BST. We will show insertion, delete, merge and update operations however, searching a node is going to be discussed later in 3.2. Firstly we add a node by giving a command in the form: Insert (Key field of node, data field of node, version). We would like to insert node 10 with key field: 10, data field: D10 and this insertion is going to take place at version 0 since no others exist. This operation will create version 1. In figure 3.1 we firstly represent D in version 1, D is not stored but we include it in our figures in order to show how changes in BST affect our set of data structures. Global version Tree V and Link-cut Version Tree F are identical at this point. We insert in V and F node 1 which represents version 1. We must always keep a pointer (root pointer denoted as RP) from V to the root of D for the version we just created. This is represented in the figure as root pointer RP to the compressed local version tree cLVT(10) and is pointing specifically to node 1 of cLVT10 shown in our figure as cLVT(10.1). Then we insert also node 1 in cLVT which is in grey as we choose to make a senators the roots of cLVTs. Senators will be represented as gray nodes in our diagrams. Next senator will be at distance λ+1 where λ we choose to be 2. Since there no edges to our senator we do not include any other navigation pointers. We finally store all fields of node 10 in D in cLVT(10.1) that is, its data field with value D10.

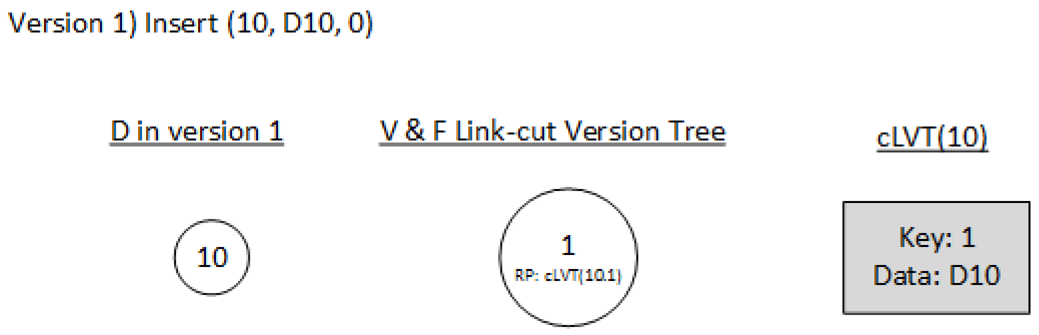


Figure 3.1 Insertion of Node 10 in a BST. cLVT(10.1) denote node with key 1 in cLVT(10)

Next we insert node 6 in version 1, with the operation Insert(6, D6, 1). This will produce version 2 so node 2 in V and F is inserted. The root pointer for node 2 in V is cLVT(10.2) as we must ensure for each new version the root pointer from the V points to that specific version in a cLVT. Hence root pointers must always point to explicit nodes in root cLVTs. Hence node 2 appears explicitly in cLVT for the above reason but also because some fields of node 10 have been altered. Node 10 has a new left pointer field pointing to node 6. Hence its left pointer field is also included in cLVT(10.2) as a left version-edge pointer VEPL. This left pointer field in D will point to node 6 but in our case points to edge overlay tree cOV(10.6).2. cOV(10.6).2 is created as an edge has been created between nodes 10 and 6 and it has an edge pointer to node 2 of the newly created cLVT(6). cLVT(6) is a senator and has a data field D6.

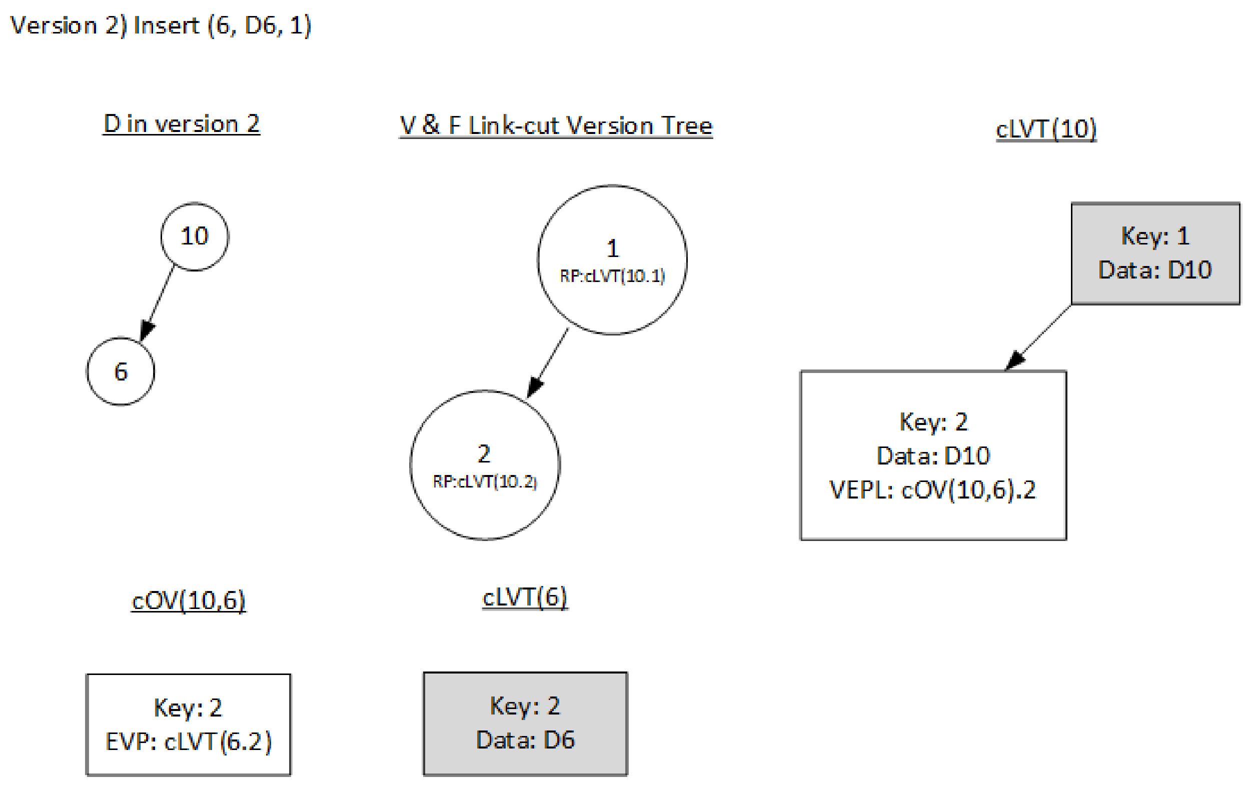


Figure 3.2 Insertion of Node 6 in version 1

Our next operation is delete node 10 in version 1 denoted as Delete(10, 1). New version 3 is inserted in V but D becomes empty so the RP in node 3 of V is null. In cLVT(10) node 3 is not inserted as in version 3 node 10 did not exist. Also cOV(10,6) and cLVT(6) are not affected by this operation.

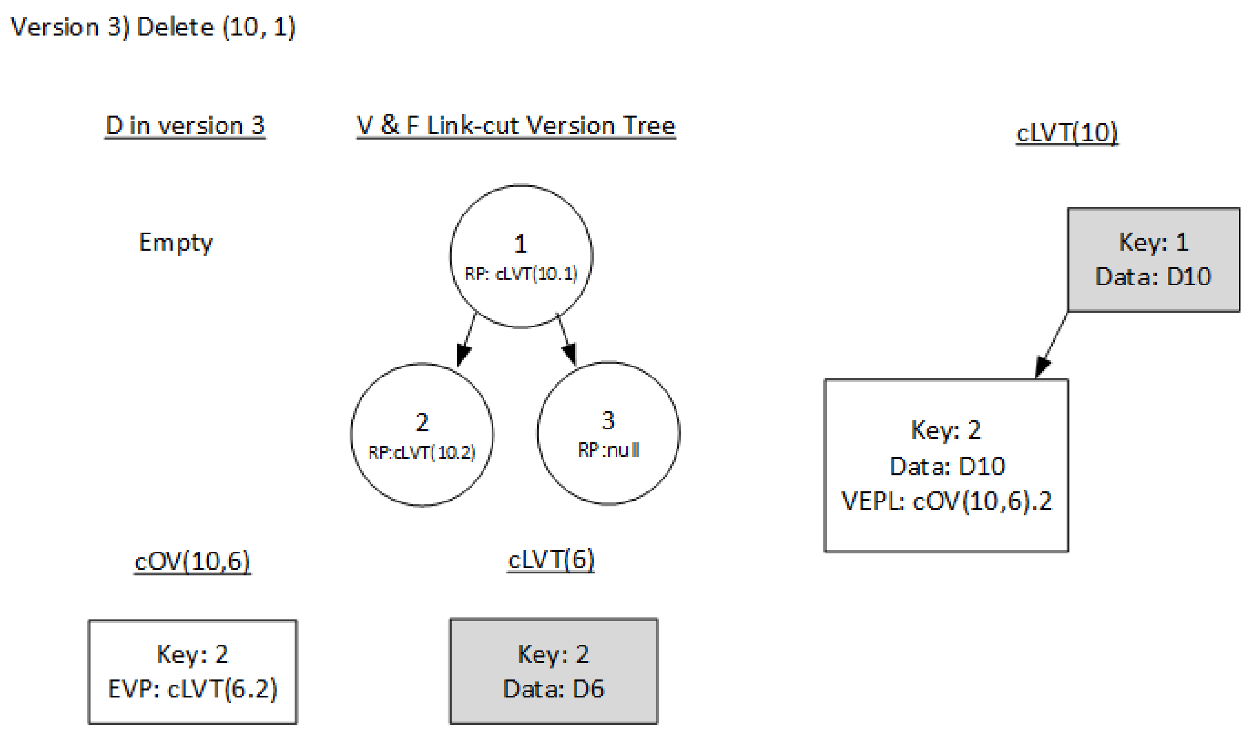


Figure 3.3 Deletion of node 10 in version 1.

Then similarly to above we add node 9 figure 3.4 so in our next operation we can perform a merge of two versions that do not share the same nodes, figure 3.5.

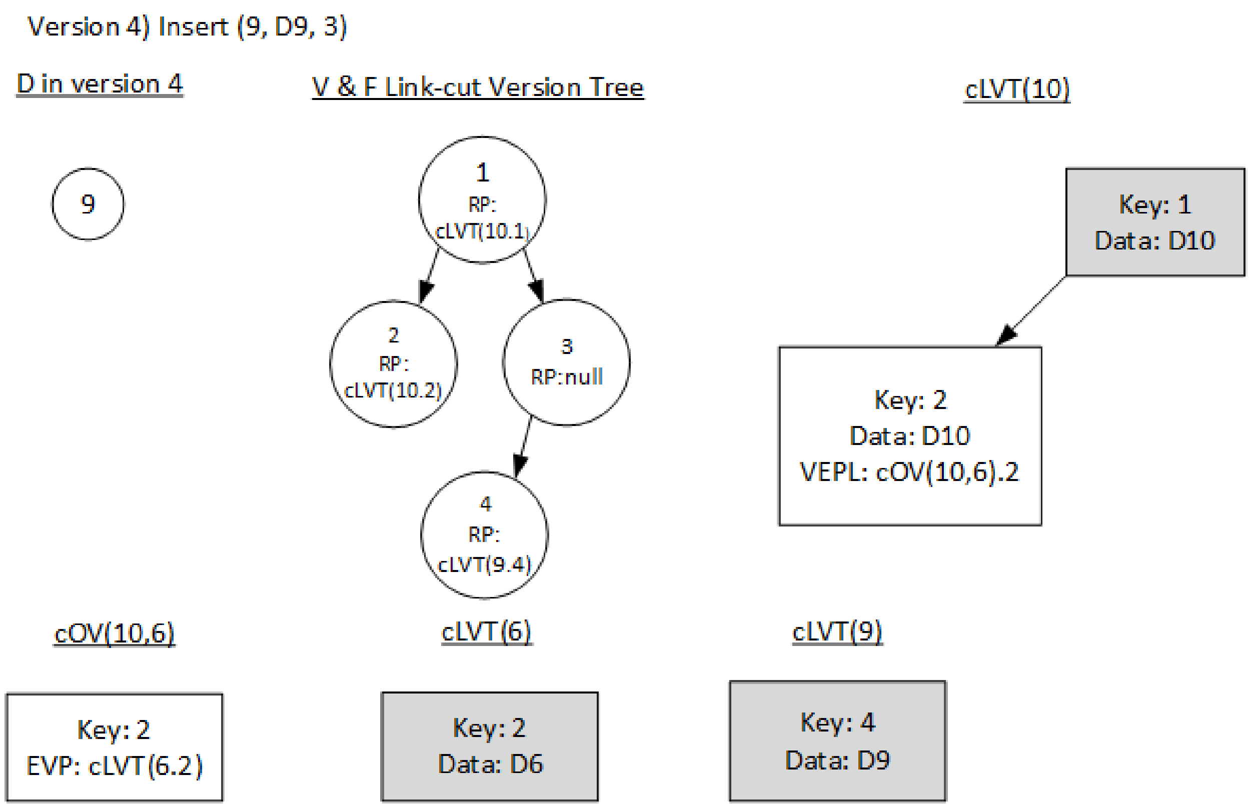


Figure 3.4 Insertion of node 9 in version 4.

The merge in the following figure 3.5 is between versions 2 and 4 (Merge(2,4)). We perform the binary search tree confluent merge as follows. We insert nodes of 2nd version, version 4 in a sorted list. Note that version 2 and 4 are not allowed to include the same nodes. Then we insert each node of the sorted list in the first version, version 2. Hence a merge is a sequence of insertions for our set of data structures. Version 4 includes only node 9 in our example which is inserted after node 6 in version 2. A new edge is created so node 5 is included in cOV(6,9) and thus it must appear explicitly in cLVT(6) and cLVT(9). That is why node 5 is inserted in cLVT(9) although its data fields have not been altered since version 4 and node cLVT(9.4) could have been its representative. Also note that in F we cut the links to the sons of the lowest common ancestor of version 2 and 4. This will not happen in V not separately included in the figure.

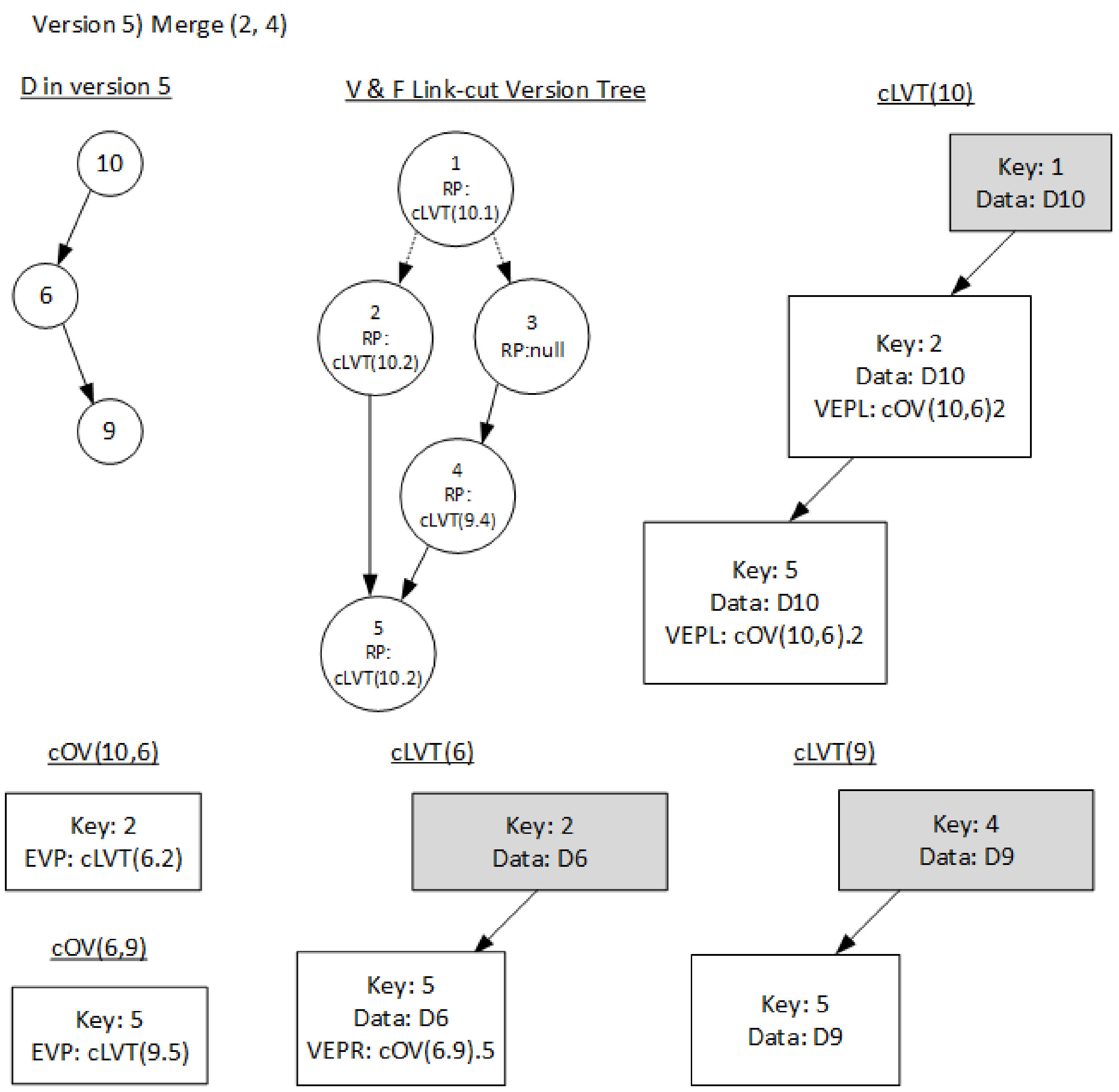


Figure 3.5 Merge between versions 2 and 4.

Last we update the value of the data field of node 9 in version 4 so it value changes from D9 in version 4 to D9.1 in version 6.

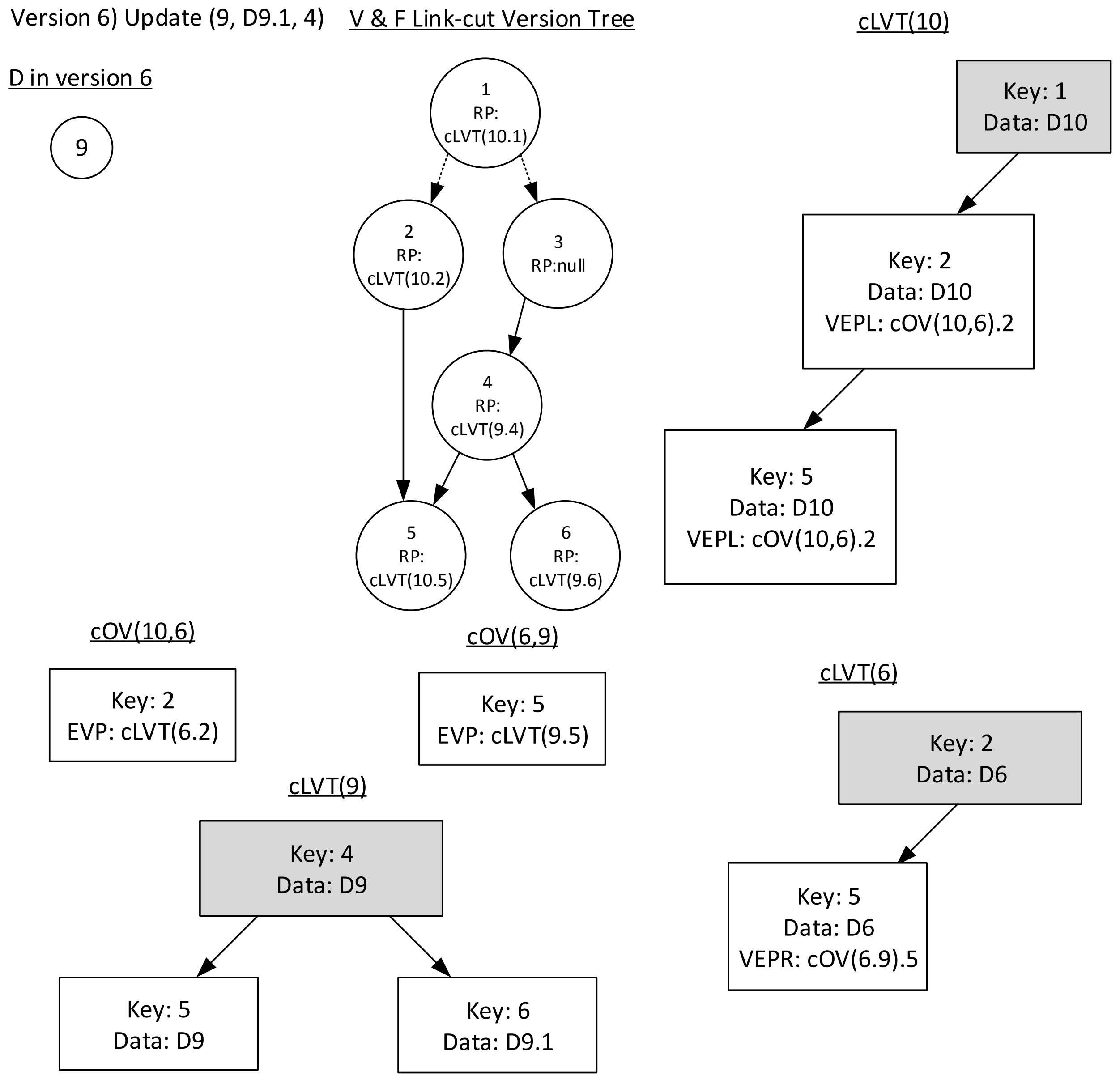


Figure 3.6 Update the data field of node 9 in version 4.

Next three operations are performed as above. The purpose of choosing the following set of operations it's because we wanted to end with a set of data structures that will help us explain the act of navigation (searching) which is described in the following section. In the figures below we only present modifications in cLVT(6), cLVT(9) and cOV(6,9) for saving space in our draws.

What is worth noting in figure 3.9 is the node cLVT(9.9) should not be included in cLVT(9) as in version 9 node 9 in D is not present, however as an edge is deleted, in cOV(6,9) node 9 must be explicitly included and this forces node 9 to be also explicitly included in cLVT(9).

Also note the introduction of two senators cLVT(6.9), cLVT(9.9) at distance λ+1 from their senator ancestors (in our examples we set λ to be equal to 2). In particular in cLVT(9.9) note that as a senator it has a list of inverse edge-version pointer, IEVP, that includes a pointer to its incoming edge in D. For reasons that we will become obvious later, version edge pointers VEP and EVP edge-version pointers are now included also as dashed arrows.

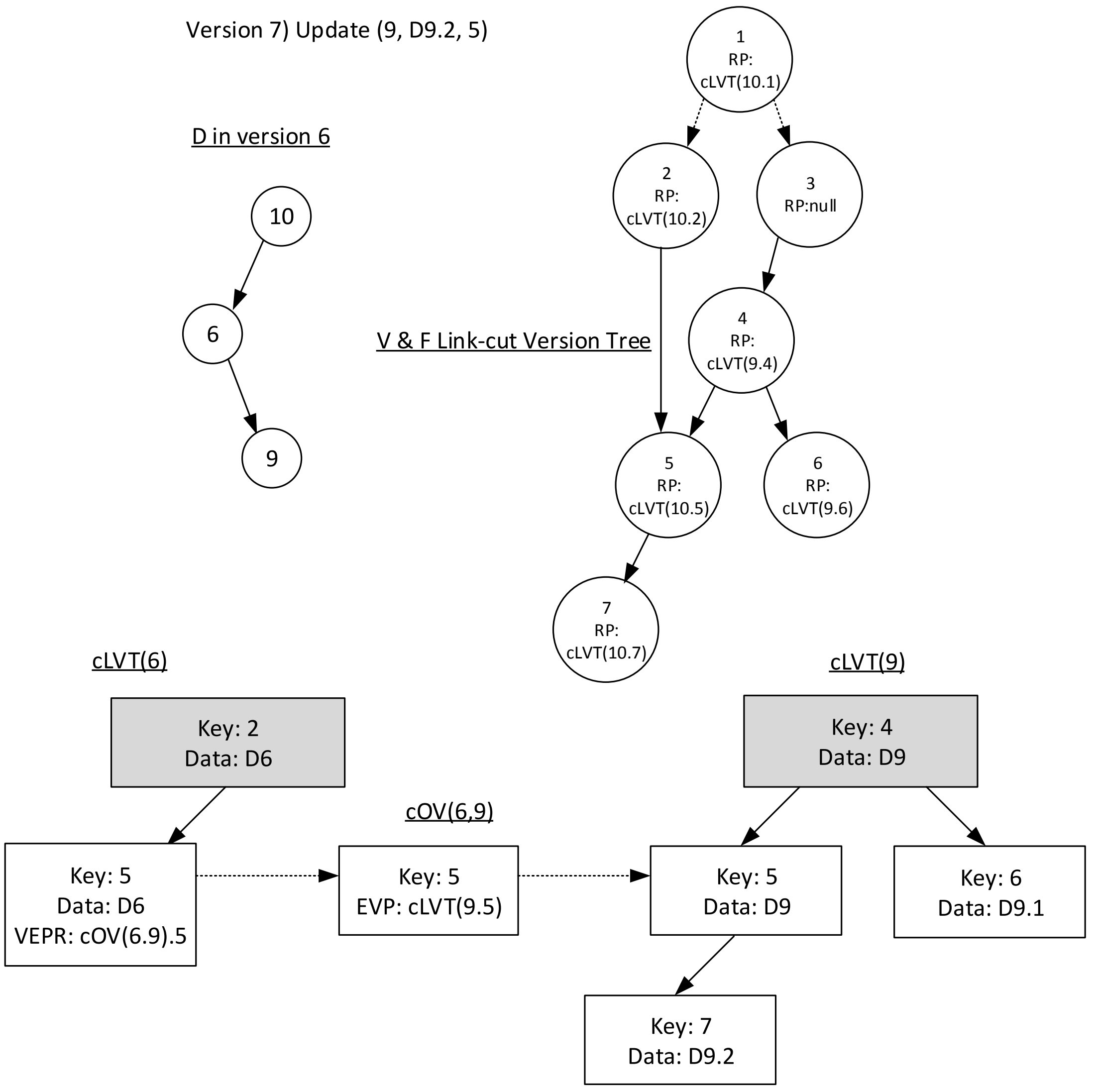


Figure 3.7 Update the data field of node 9 in version 5.

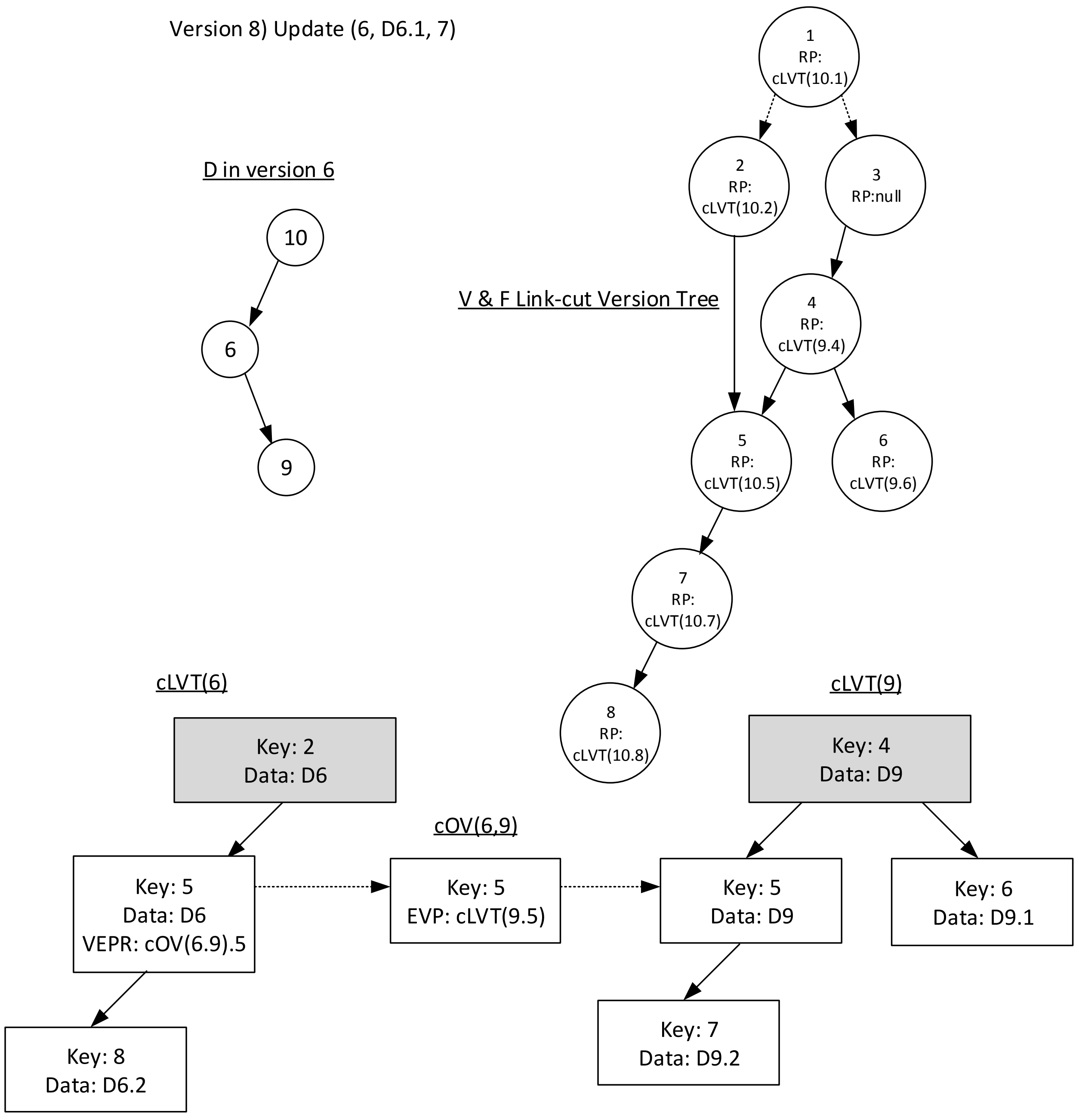


Figure 3.8 Update the data field of node 6 in version 7.

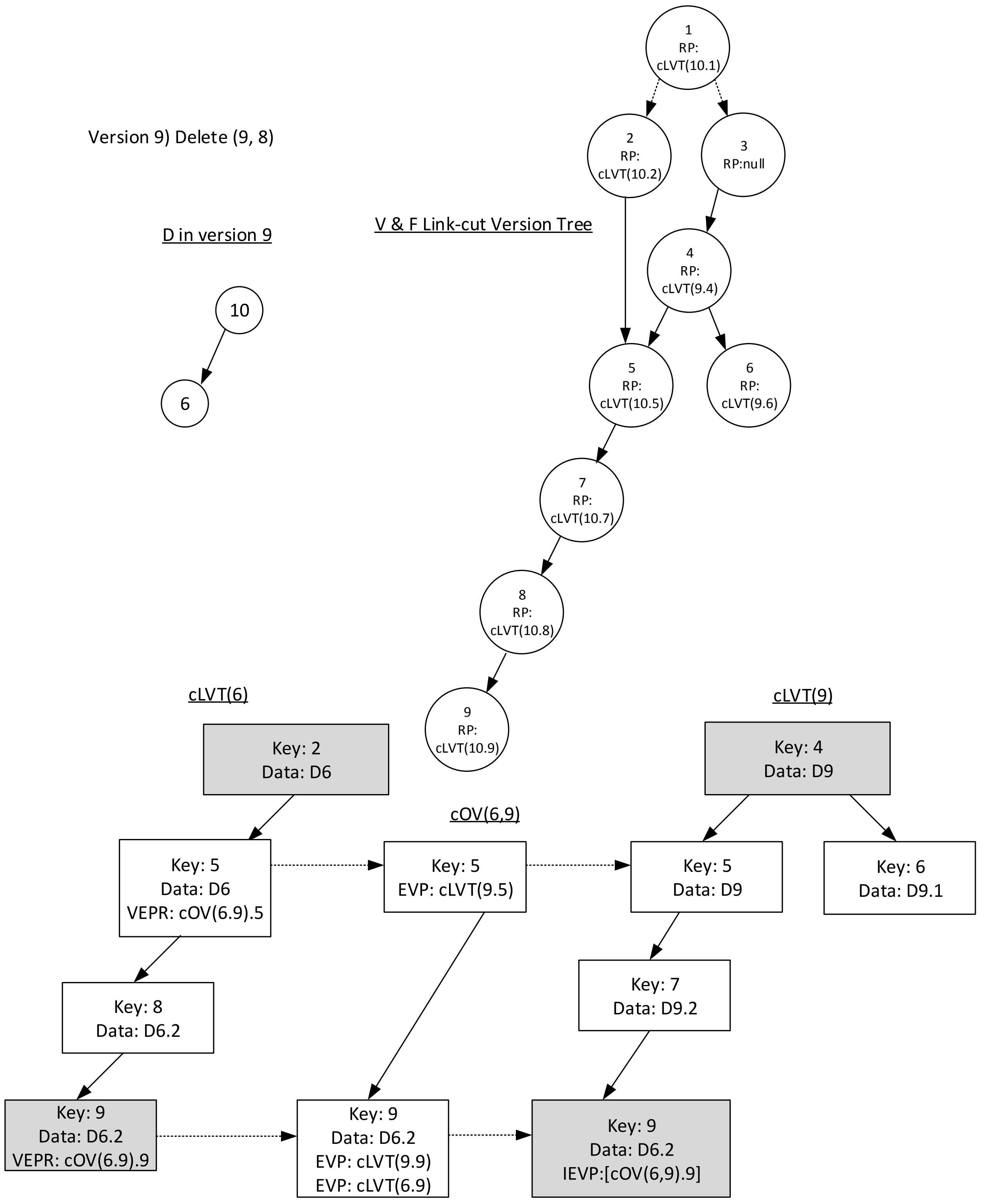


Figure 3.9 Delete node 9 in version 8.

3.3 Searching for a node.

So far we have shown examples of inserting, deleting, updating and merging without explaining how do we search for nodes in the set of the above data structures (V, F, cLVTs and cOV). In this section we describe the act of navigating through this set and we also explain why senators are necessary.

Firstly from the above examples it is obvious that we can easily determine the root d of D for a version v and all its fields. We search for v in V where there is root pointer RP to an explicit node in the root's cLVT(d) for this version. In figure 3.6 it is shown that all RP point to explicit nodes in the corresponding cLVTs.

From the explicit nodes in a root cLVT and by following the rules concerning D we follow edges. For example in binary trees we need to determine if we will follow the right or left pointer by comparing the key of the node we are searching to the key of node we are visiting. As an example in figure 3.5 what if we would like to determine the value of node 6 in version 2. We search for version 2 in V and from the there we follow the RP to node cLVT(10.2). We know that we are in a node with key field 10 and and we would like to find node 6 which is smaller hence we need to follow left pointer VEPL(cOV(10,6).2). We visit node cOV(10,6).2 and from there we follow EVP(6.2) where we end in node cLVT(6.2) where we can read all fields of node 6 in version 2. From the above we understand that following pointers as if all nodes were explicitly present in our set of data structures will be straightforward.

However the act of browsing through our set of data structures becomes more trivial when visiting nodes that are implicit i.e. they are not present in cLVTs and pointers lead to their explicit representatives. As an example we use a part of our set of data structures shown in the following figure 3.10 which we constructed in the series of examples above, figures 3.1-9. What if we wanted to find the value of node 9 in version 7. During our search we enter cLVT(6) in node cLVT(6.5) this is a representative node as we are searching for version 7 and we are in a version node with key field 5. Whenever we are redirected to a representative node we act as follows:

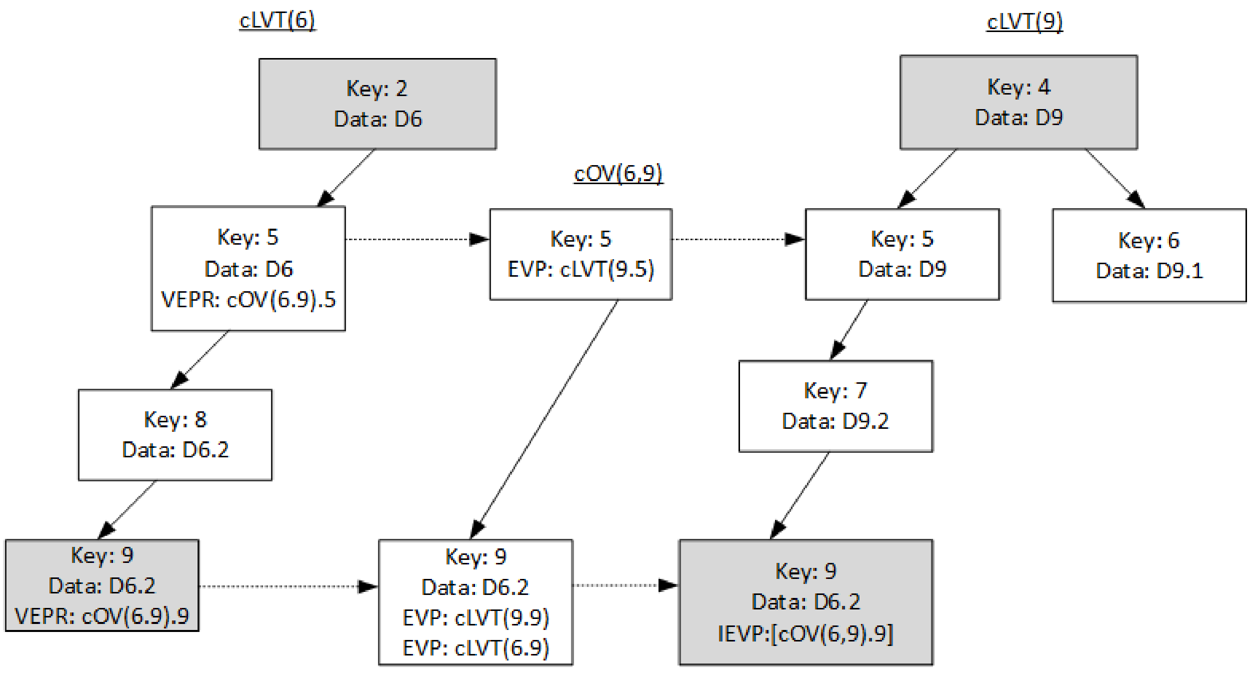


Figure 3.10 Navigation Example I

We locate the first vertex, on the path from the representative node e.g. cLVT(6.5), we have just arrived to the root of the cLVT, that explicitly exists in the edge we want to follow cOV(6,9). In our case it happens to be same node that we are at, cLVT(6.5), so there is no need to locate an ancestor. This may be a senator, or any version in which the edge (6,9) was modified, created or deleted. We also find the descendant vertex in the cLVT that is also explicitly present in cOV(6,9), the edge we would like to follow. Note that the distance between these nodes will always be at most λ+1, due to fractional cascading. In our example that is cLVT(6.9). In case there is no such descendent we will only use the first node (cLVT(6.5)) and this case will be examined later. We follow both VEPR of cLVT(6.5) and cLVT(6.9) to nodes to cOV(6,9).5 and cOV(6,9).9 and from there we follow EVP to nodes cLVT(9.5) and cLVT(9.9). In order to locate node 7 in cLVT(9) or its representative we browse the path bottom-up from cLVT(9.9) to cLVT(9.5). As we are looking for version 7 we stop as soon as the key of nodes we are visiting is smaller or equal to 7 and this is node we are searching for and all its fields can be read. In our example we have an explicit node for version 7 in cLVT(9).

In the other case where there is not a descendent in cLVT(6) figure 3.11, there are two possibilities. If cLVT(9.5) is a leaf in cLVT(9) then cLVT(9.5) is the representative of the node we are searching. Otherwise we need to determine if node 7 in cLVT(9) belongs to the left or the right subtree of cLVT(9.5). In all cases the node in cOV that redirected us in cLVT(9) must be a leaf otherwise there must have been a descendent node in cLVT(6), which it should have been detected earlier. Next we cut in F (temporarily) the one or two incoming edges connecting to the node with same key value as the the node we are at, in our case node 5 in F since we are at cLVT(9.5), figure 3.12.

Due to the constraint that we can not merge versions that share the same nodes and the fact that we have been redirected from a leaf in cOV all implicit and explicit nodes in the subtree of node cLVT(9.5) belong to the same component in F and are explicitly present in F (F includes explicitly all version nodes). Otherwise if this component in F is not connected this will indicate that we have merged versions that share the same nodes.

Moreover due to fractional cascading and the fact that we have been redirected from a leaf in cOV, we know that cLVT(9.5) can only have a constant number of explicit descendants (we will calculate this in the next chapter) which is a function of λ. Next we need to locate the node (in our case node 7) we are searching in F and its representative or explicit node in the cLVT.

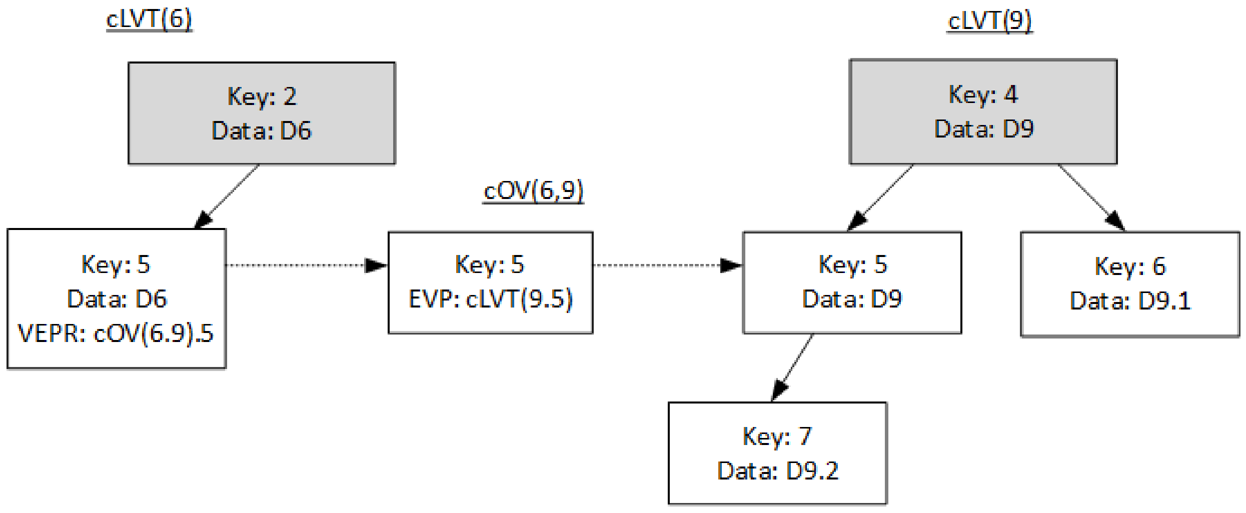


Figure 3.10 Navigation Example II

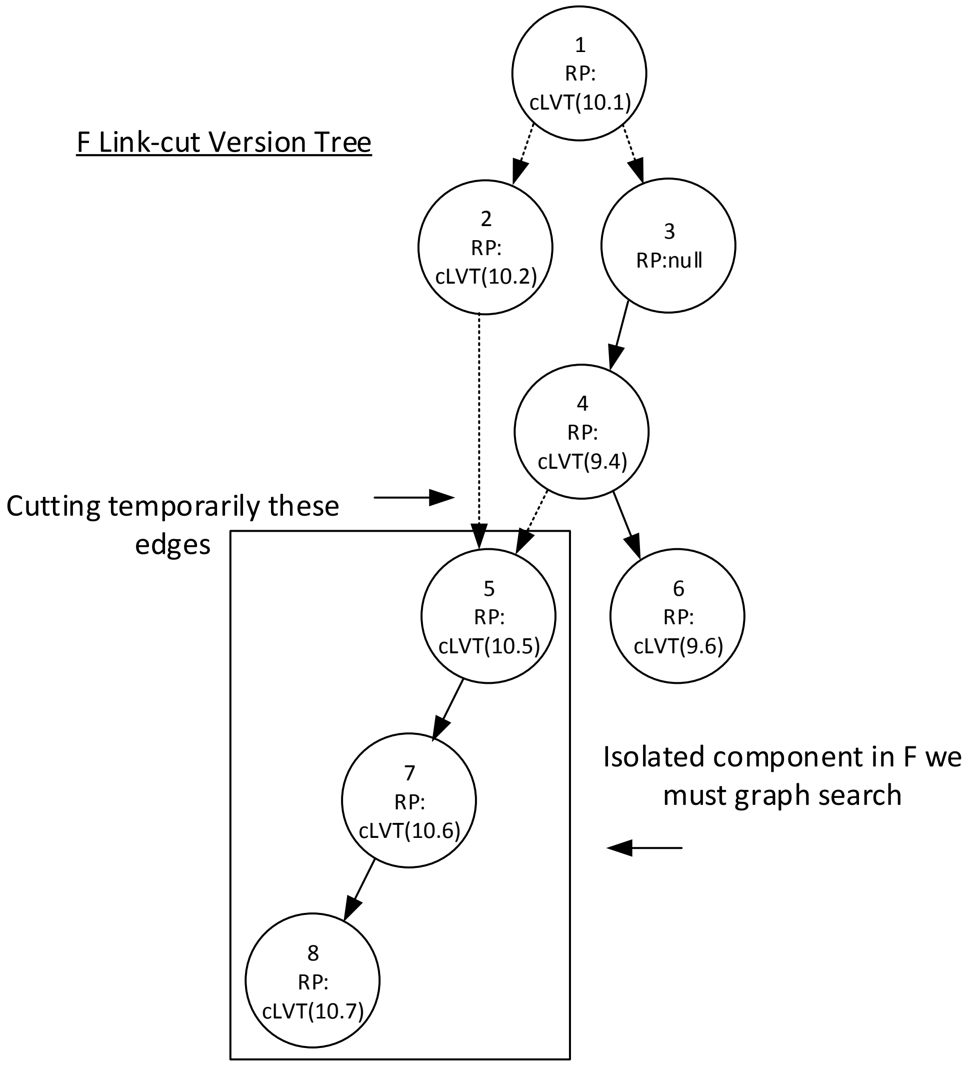
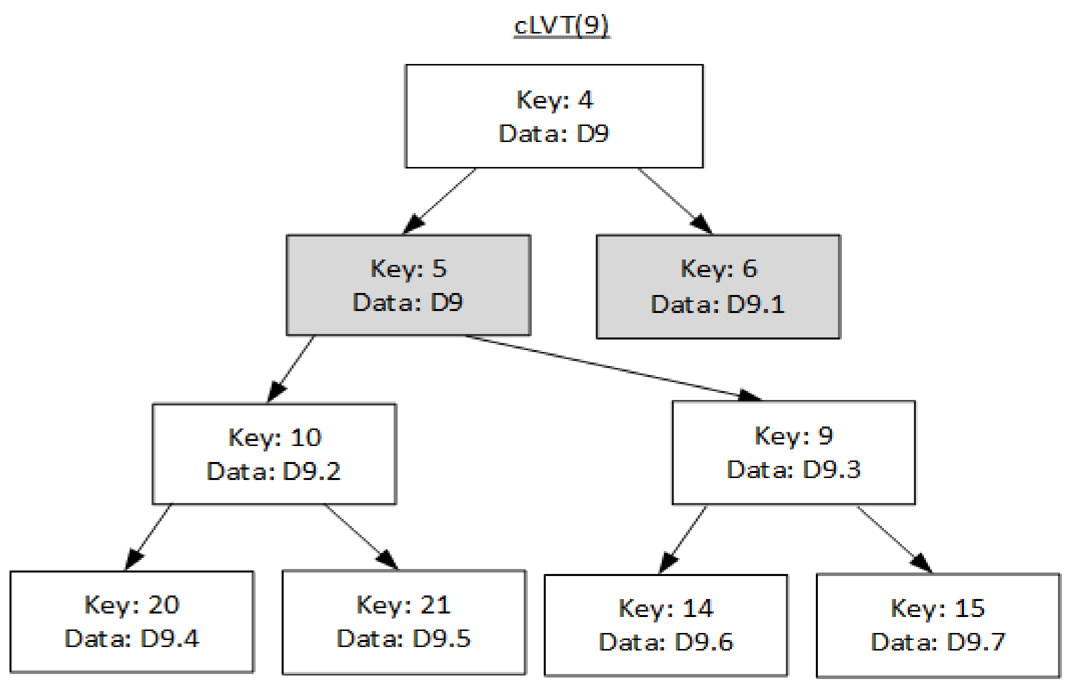


Figure 3.11 Graph Search in an isolated component of F.

But our example is simple to illustrate the complexity of such an act since in cLVT(9) and F, only a small number of nodes are present and node cLVT(9.5) has only one node as a subtree. In figure 3.12 we show a more complicated example to understand how this final act of searching can be performed in O(logn) time.



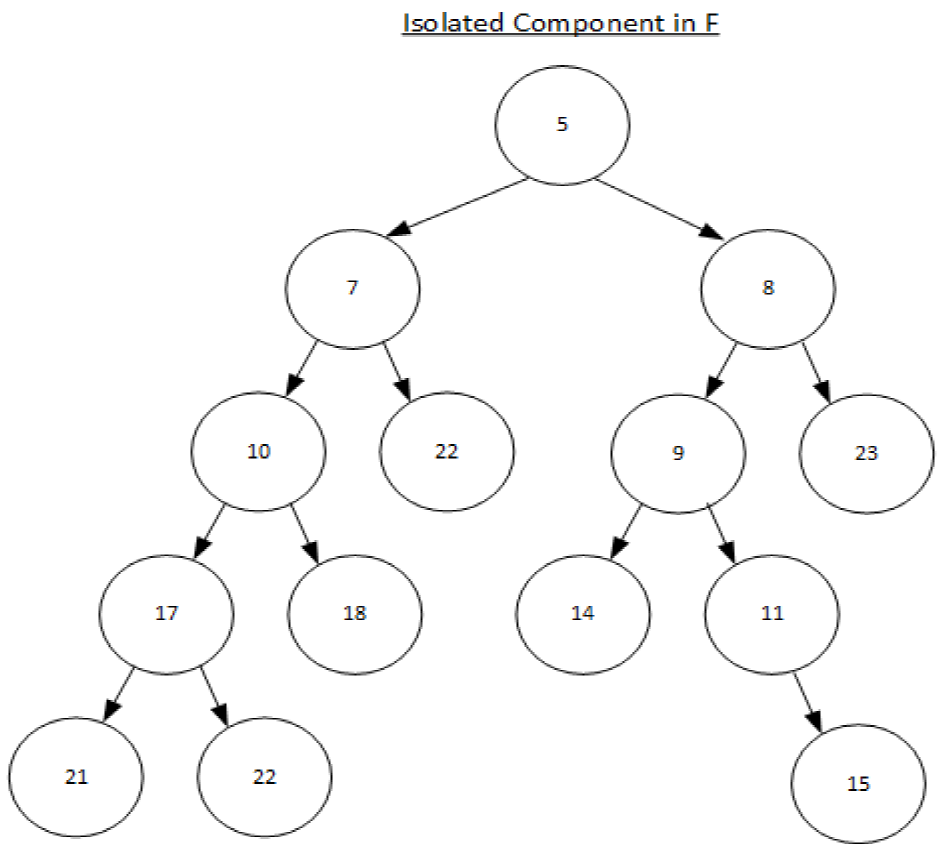


Figure 3.12 A more complicated navigation example

In Figure 3.12 suppose we are searching node 17 in this final act of navigation and the isolated component in F is rather large, hence a graph search is not indicated as it will be performed in more than logarithmic time, O(logn). Also suppose that node cLVT(9.5) is a senator and its subtree is fully occupied such as any possible insertion will cause another senator to be created. Hence Its subtree a balanced binary tree with size 7 due to λ value set to 2. There are 2 possible paths in F where node 17 could be located, path between nodes 10 and 20 and path between nodes 10 and 22 all other 4 paths shown in cLVT (5,10), (5,9), (9,14), (9,15) are excluded as 17 is not in their range. We binary search the nodes of these two paths as they appear in increasing order. In the first detection of node 17 we stop and we understand that node 10 in cLVT is the representative of node 17. From the above example we understand that with the help of the connected component in F and the bounded in size subtree of cLVT we need to perform a constant number of binary searches in certain paths in F.

Hence, due to the link-cut and min-heap properties of F searching a node in our set of data structures can be achieved in O(logn) time.

Chapter 4 Complexity Analysis.

Contents

4.1 Size of a senator subtree.

In the previous chapter during the operation of searching we saw that we may have to search for a node in the subtree of a senator in a cLVT. In the following Lemma we prove that the minimum size of a senatorial subtree is a function of λ.

Lemma 2. Let v ∈ cLVT(d) be a senatorial version. If one of its subtrees contains i > 0 senators, it has size at least 3i/2, provided λ ≥ 2.

Proof. A node is a senator if and only if it has no senatorial ancestor at distance less than λ+1. In our examples we fixed λ to be 2, hence the distance between senators is 3 as shown in figure 4.1. The smallest subtree with the largest number of senators versus non-senator ratio, is a complete balanced binary tree with i senators in the leaves (in our example i = 2), connected to a path from v of length $\lambda + 1 - \lceil \log_2 i + 1 \rceil$ (also in the figure below this equation equals to 2). The complete balanced tree has size larger or equal to 2i - 1 (in our case it has size of three). Hence the total number of nodes in the subtree is at least: $2i - 1 + \lambda + 1 - \lceil \log_2 i + 1 \rceil$, which is always larger or equal to 3i/2 i.e. $\lceil \log_2 i + 1 \rceil \leq \lambda + {}^{3i}/_{2}$, provided λ ≥ 2.

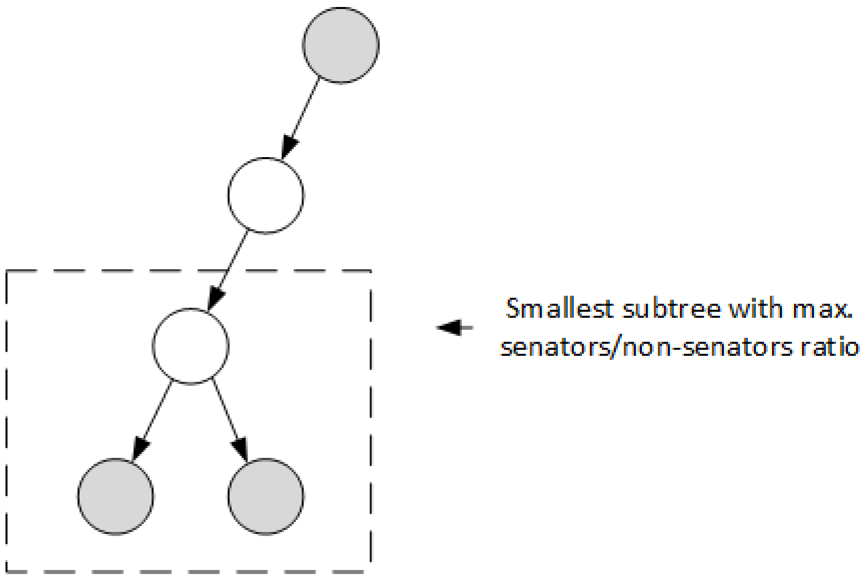


Figure 4.1. Lemma 2. Proof

4.2 The cost of adding a new explicit node in cLVT.

We will prove the below lemma using the potential method [1].

Lemma 3. Adding a new explicit version v in the local version tree cLVT(d) of a node $d \in D$ takes O(1) amortized time and space.

Proof. We set as potential function argument that whenever a new explicit version is added to a node d, we increase the potential **of cLVT(d)** by a constant γ. We want to prove that γ, a constant, upper bounds the amortized cost of node insertions in a cLVT.

We set the cost of adding a non-senatorial node to be one unit of potential as we only need to create a new vertex. Hence, after an insertion of a non senatorial the overall potential has increased by γ-1.

In calculating the cost of adding a senator we have to take into account that we may also need to add a new explicit version in every edge overlay cOV(d', d), and a new explicit version in each cLVT(d') where d' are parent nodes of d . If Δ is the in-degree of node d, this means a 2Δ extra insertions at most. Hence the increase in potential after a senatorial insertion is $\gamma-1-2\Delta$.

At the ith senator insertion the subtree has size at least 3i/2, proved in Lemma 2. Thus, the potential is at least $3i(\gamma-1)/2- 2i\Delta$ after the insertion. If we set $\gamma \geq (4\Delta+1)/3$, the overall increase in potential will be always greater or equal to zero. In the case senator creation is propagated to other cLVTs the above rule still holds in the other cLVTs.

4.3 Overall Performance.

Lemma 4. Given a version v, deriving a new version takes O(log n) amortized time and space (either through a new branch or a confluent merge).

Proof. We calculate the cost by summing up the costs of the following operations we perform each time a new version v is created:

- Firstly we add the new version v in V and in F in O(log n) time and O(1) space. Finding the place in V and F where the v must be placed can be done by binary search in case V is implemented as a sorted list.
- We must explicitly add v version in cLVT(d), where d is the root of the data structure at version v. From V we can easily access the previous version in cLVT(d) under which the new vertex is inserted at a constant distance. This new explicit version might be a senator, in which case we will have to propagate explicit versions in some neighbors of d. Lemma 3 declares that this can be done in O(1) amortized time and space.
- For a confluent merge operation, we need to find a lowest common ancestor and cut two edges in F. This due to the link-cut properties of F costs O(logn) amortized time.
- Finally perform the necessary additions in the corresponding cLVTs and cOVs which as shown in the Lemma 5 below can be done in O(log n) amortized time and space.

Lemma 5. Creating or deleting an edge, or modifying any field of a node d in the data structure at version v takes O(log n) amortized time and space.

Proof. Any operation in D will create a new version. According to the above Lemma 4 this is done in O(log n) time. After creating a new version in V, F and cLVT(d), where d is the root of D at version v, we will perform the necessary modifications in the corresponding cLVTs and cOVs that we find after browsing through our structure which costs O(logn) as explained in our previous chapter, section 3.2. According to Lemma 3, adding nodes to cOV and cLVTs when necessary costs O(1) amortized time, as they consist in modifying a constant number of nodes, and adding a constant number of explicit versions, even if creation of senators is propagated to neighbor nodes.

Theorem. A data structure D with maximum in-degree maxdeg can be made confluently persistent using O(Ulog(maxdeg)) space and time, where U is the number of operations performed to create all versions of the structure. Each subsequent modification of the structure takes O(log n) amortized time and O(1) space. Browsing the structure costs O(log (maxdeg) log (n)) amortized time per edge traversed. This holds provided no confluent merge operation uses twice the same node.

Proof. In Chapter 1 section 5 we showed how we can transform and maintain the structure so that the maximum in-degree is always bounded by Δ, for $\Delta \geq 2$. If we set $\lambda = 2$ and $\gamma = 3$, all lemmas hold. Any operation performed in D can be represented by a constant number of nodes and pointers in our set of data structures. Following a pointer between two nodes in our structure (d, d') only implies to look, in the cLVT(d), for vertices in a O(λ)-neighborhood, which is achieved in O(log n) time; and using a constant number of pointers, from cLVT(d) to cLVT(d') using cOV(d; d'). However our transformation for bounded in-degree in section 1.5 implies a O(log maxdeg) factor on following a pointer.

Corollary for data structures with bounded in-degree by a constant. A data structure D with constant in-degree can be made confluent persistent using Θ(U) space, where U is the number of operations performed to create the structure. Each subsequent modification of the structure or new version created takes O(log n) amortized time and O(1) space. Browsing the structure costs O(log n) amortized time per edge traversed. This holds provided no merge operation uses twice the same node.

Chapter 4 References.

[1] Data Structures, P. Bozanis, Tziola 2006 ISBN 960-418-084-3 pg. 16-17