Institut für Informatik

# Document Version Management Using an Adapted Segment Tree

*Dieter Sosna*

**Report Nr. 9(97)**

**ISSN 1430-3701**

**This page remains intentionally blank**

# Document Version Management Using an Adapted Segment Tree

Dieter Sosna

**Abstract**

We describe a data structure and algorithms based on segment trees. They are used to manage different versions of a document and to reconstruct the version which was valid at a given time in the past or to get the most recent version. Difficulties arise because it is not known when a version will be replaced by a newer one. Thus unbounded time intervals are to be handled. The data structure also supports the retrieval of the history of a document.

**Keywords:** data structure, segment tree, version management.
**CR-Classification:** E.1, F.2.2, H.2.m

## Introduction

A digital library, prototyped in the MeDoc-project ([BBK97], [Rah97]), contains digitally stored documents, which are especially prepared for the requirements of such a library.
Generally a document consists of several parts. Digital storage allows for easy and cheap replacement of parts of a document by newer versions. This replacement leads to a new version of the whole document. We want to manage these versions, in particular we want to obtain the last version of a

document or to reconstruct the version which was valid at a given time in the past, because a user of the library should always obtain the version of a document he has ordered or paid for - even if a newer version exists.

## Stages of a version

Let us consider a certain version of a part. It has an *active time*. The active time starts with the creation of this version of that part. At the time of creation the version may be a *draft version*. After completion the version becomes a *final version*. The final version may not be altered, it is released to the users of the library. Kim [Kim90] called it a released version. Altering a final version requires a new draft version which then becomes a final version. At this moment the former final version is called an *old version* and its active time is over, because we want to have exactly one final version of each part at a time. Draft, final and old versions are called *stages* of a version.

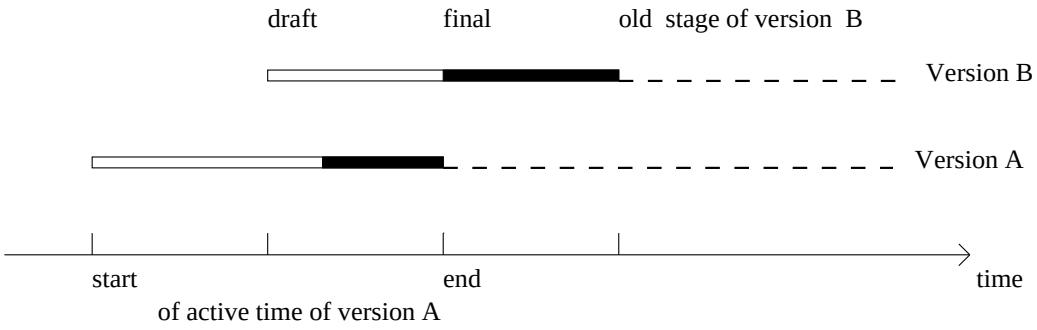


Fig.1: The stages of versions

## The adapted segment tree of versions (stv)

The active time of a version may be considered an interval of a time axis. Thus, managing versions of a document means managing a collection of time intervals. Each version of each part contributes one interval to the collection. We can write the query for the valid version of the document at a given time

as follows:
Find all intervals of the collection covering the given time and having the stage property “final“ at that time.
The segment tree is a well-known data structure representing such a collection of line segments or intervals [Sam89]. Considering such a collection, all creation times and all times at which a version becomes an old version form a set of points $\{t_k\}$ on the time axis. Thus, these points are sorted. If the collection contains $m$ intervals, the corresponding point set consists of at most $2m$ different points. Most existing applications of segment trees presume that all points $t_k$ are known before constructing the tree. This assumption is not fulfilled here. Creating a new draft version or changing a draft to a final version adds one point to the set. Its time value is greater than that of all previous $t_k$. Adaptations of the data structures and the insertion procedure are necessary to assure completeness of the tree.
Further difficulties arise because it is not known when a released version will be replaced by a newer one. When creating a version, its active time is assumed to be unbounded. Therefore, handling of unbounded intervals must be provided for.

## Data structures

The segment tree is a complete binary tree. Its anchor carries a time stamp, which stores the start time of the oldest segment.
Each non-leaf node has the following components:
- a time as the key value, which is used for navigation
- two pointers to the right and the left child,
- a sequential list of covering intervals.

The leaf nodes consists of the sequential list only.
All nodes have a further pointer which provides an in-order traversal of the tree ( it might be replaced by an in-order traversal procedure with loss of performance).
This tree is called adapted segment tree of versions (stv).

Figure 2 (next page) shows an adapted segment tree of versions and the corresponding time intervals.

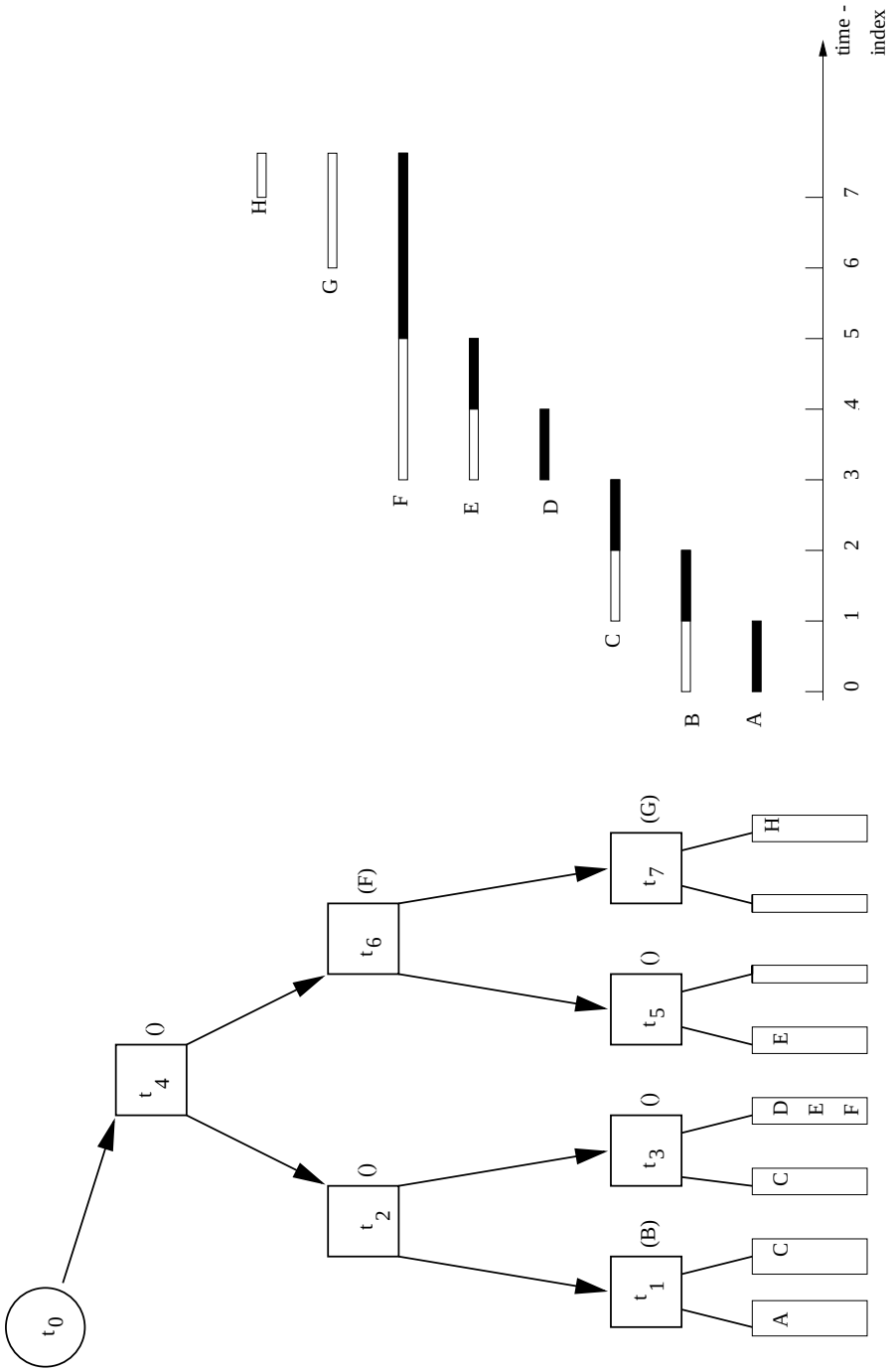


Fig.2: Segment tree without in-order-traversal pointer of intervals A ...H, which are given in the right part

The time value for *infinity* ($\infty$) is defined as a time greater than any time possible in reality. The key of each new non-leaf node will be initialized with this value. During the insertion procedure it is replaced by value for a real time.

## Algorithms

The insertion procedure of a binary tree applied to our data leads to a degenerated tree, because the sequence $\{t_j\}$ is monotonously growing. In what follows we will describe an insertion algorithm that prevents degeneration of the tree and preserves completeness.
We assume that we have a tree of height $h(h > 0)$ and that all nodes are in use, i. e. all non-leaf nodes have a key less than infinity. The tree has $2^{(h-1)}$ leaf nodes and $2^{(h-1)} - 1$ non-leaf nodes (if $h > 1$). It represents the times $t_0, t_1, ..., t_k$, $k = 2^{h-1} - 1$.

*Tree expansion:* To insert data at time $t_{k+1}$ a new non-leaf node has to be created and its key replaced by time $t_{k+1}$ (this is the last used node now). This node becomes the new root node, the whole old tree becomes its left child, the right child is formed by a complete binary tree of height $h$ with new non-leaf nodes and empty lists. The in-order traversal pointers are constructed, too. The result is a stv of height $h + 1$. Data insertion now follows the rules used in case of no expansion as described below.

*Insertion* of a new version or changing a draft to final version at time $t_{k+1}$: If necessary, a tree expansion is carried out, otherwise the next non-leaf node is found by in-order traversal from the last used node. The key of the found node is set to $t_{k+1}$ and it becomes the last used node.
Data insertion follows these four steps:

1. Start at the root node and, given the new time $t_{k+1}$ navigate to the corresponding leaf node.

2. Find all versions, which are active at time $t_{k+1} - \varepsilon$ (with $t_{k+1} - \varepsilon > t_k$) and do not become an old version at $t_{k+1}$, by means of the find-procedure, described below, and copy them to the leaf node of step 1

by appending them to the sequential list.

3. Append versions created at time $t_{k+1}$ and change stage properties, if necessary.

4. (Recreation of the segment tree property) Find the greatest natural number $j$ with the property: $k+1$ divisible by $2^j$. Start at the leaf node of step 1 and repeat the following $j$ times: Go to the parent node of the node and move (not copy) all versions belonging to both child nodes to this node without changing the order of succession.

*Find:* This algorithm returns all versions which are active at a given time $t \geq t_0$: Start at the root node and navigate to the leaf node. Return the lists of all nodes which are passed on this path in the sequence they are passed.

Figure 3 (next page) illustrates insertion with tree expansion.

## Notes:

The find-procedure returns only active versions. They are sorted by their creation time.
The segment tree of versions may be used to manage the versions of one part only or the versions of one or even more documents. In the later cases the lists returned by the find-procedure must be processed sequentially to obtain the final version of a desired part. If these lists are too long they could be replaced by other structures. In most applications, the creation time of a final version will be further in the past than the creation time of other existing draft versions. In this case, the final version of a part is returned before draft versions are found. If a stv manages the versions of one part only, to find the final version only the first list entries of at most $h$ lists have to be processed, where $h$ grows logarithmically with the number of versions.
The history of a part of a document is found by in-order traversal of the whole tree until a node is reached whose key equals infinity and processing the lists of the thereby passed nodes.

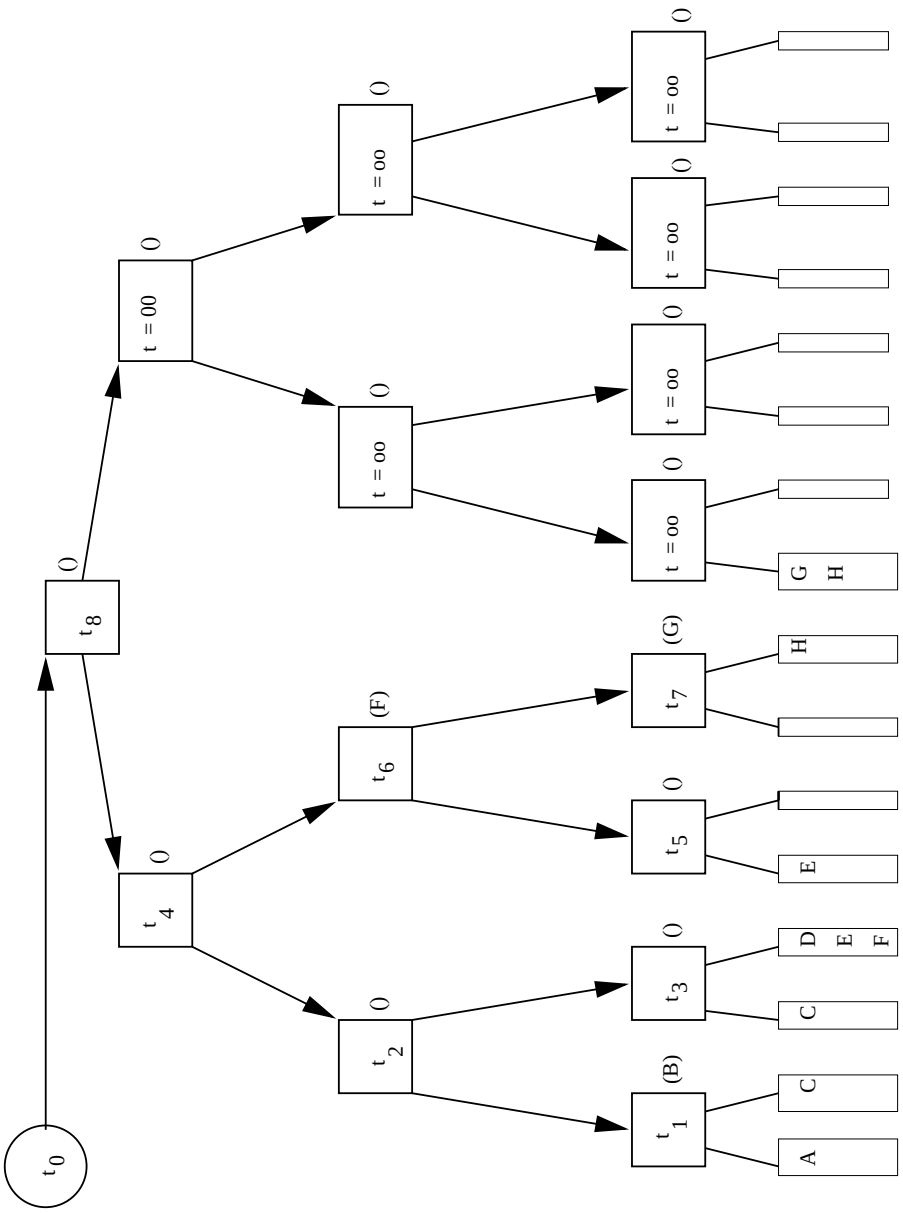


Fig.3: Expansion of a segment tree; at $t_8$ the active time of version F finished and version G becomes a final version. Hint.: oo means $\infty$.

## Conclusion

The adapted segment tree in connection with the proposed algorithms provides an effective data structure to manage versions of digital documents. The algorithms of handling the data are sketched only and should be described in detail when using the tree in an application. The absence of an algorithm for deletion of data allows of creating the history of a document or of regenerating the set of the versions of documents existing at a given time in the past.

# References


[BBK97] Michael Breu and Anne Brüggemann-Klein. Das MeDoc-Projekt: Ein Überblick. 1997. To be published in the journal "Informatik/Informatique".

[Kim90] Won Kim. *Introduction to Object-Oriented Databases.* The MIT Press, Cambridge, 1990.

[Rah97] Erhard Rahm. Überblick zum MeDoc-Projek. 1997. Informationen aus Anlaß der Leipziger Buchmesse,
URL: http://www.informatik.uni-leipzig.de/medoc/ueberblick.ps.

[Sam89] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989. Reprint with corrections, 1994.



Author's address:
Dr. Dieter Sosna
email: dieter@informatik.uni-leipzig.de
Institut für Informatik
Universität Leipzig

Augustus-Platz 10/11
04109 Leipzig
Germany