# An efficient implementation of the optimal paging algorithm

A thesis submitted for the degree of
**Bachelor of Computer Science**

by

**Edgardo Deza**

Supervisor:
Prof. Dr. Ulrich Meyer

Advisers:
Dipl.-Inf. Andrei Negoescu
Gabriel Moruz, PhD

Reviewers:
Prof. Dr. Ulrich Meyer, Prof. Dr. Georg Schnitger

Professorship for Algorithm Engineering
Institute for Computer Science
Goethe-Universität, Frankfurt am Main

GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

## Declaration of authorship

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Frankfurt am Main, November 15, 2012

(Edgardo Deza)

## Acknowledgement

I thank Ulrich Meyer for giving me the opportunity to write my thesis on this interesting topic. His enthusiasm for algorithm engineering has kindled my interest in online paging.

I am grateful to Andrei Negoescu for being my adviser. Without his help and encouragement this thesis would not have been possible.

I thank Gabriel Moruz for his support. He always had an open office door whenever I had a question, and his expertise proved valuable.

I wish to thank Georg Schnitger. He immediately accepted my request for being the second reviewer of my thesis.

I am indebted to Diana Firnges. She has often advised me on my study plan.

## Abstract

In this Bachelor thesis we designed a new data structure denoted as LTrees which keeps track of the costs that are incurred by the optimal offline paging algorithm OPT [1]. We analyze LTrees and show that for a page-set size $m$ it answers the question whether a requested page is in OPT's cache in $\mathcal{O}(\alpha(2m))$ amortized time, which is virtually constant since the inverse Ackermann function $\alpha$ is smaller than 5 for any practical input size. This presents an improvement over other approaches that require $\mathcal{O}(\log(m))$ [2] and $\mathcal{O}(\log(k))$ [3] time, where $k$ is the cache size. The space required by LTrees is $\theta(m)$.

Using real-world traces we have examined how LTrees behaves in comparison to a timestamp based implementation [12] which we refer to as TStamp. The experiments demonstrate that the runtime for LTrees is robust towards different cache sizes, whereas the runtime for TStamp can dramatically increase.

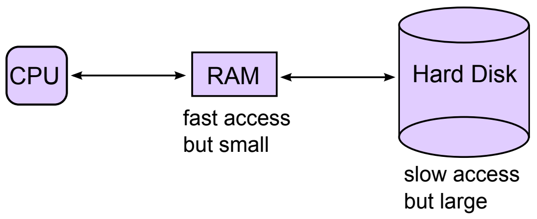
Our data structure can be used as a subroutine for the page replacement algorithm RLRU [2]. Another application is the calculation of the empirical competitive ratio [7].

# Contents

# 1 Introduction

## 1.1 Memory hierarchy and paging

We consider a two-level memory hierarchy that consists of a fast but small memory and a slow but large memory. An example for this can be found in a personal computer: on the one hand the RAM is fast compared to the hard disk, on the other hand the size of the RAM is much smaller than that of the hard disk [4]. The reason why the hard disk is slow is because it is also a mechanical device. It consists of a head that is moved by an actuator arm and a rotating disk similar to a record player. To find the right data the head has to be moved to the right track, and the disk rotates until the data is below the head. In contrast, the RAM is a purely electrical device.



The data on the hard disk is divided into blocks called *pages*. When the CPU needs to access data the corresponding pages have to be loaded from the hard disk to the RAM such that the CPU has fast access to the data. This requires no further action as long as the RAM has still some space available. However, if the RAM is full, i.e. there is no room for new pages from the hard disk, we have to create space for the new page by evicting a page from the RAM.

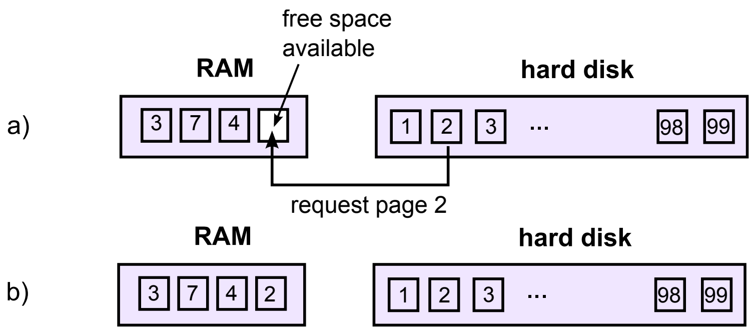


Figure 1: a) The RAM has space for 4 pages, and the hard disk consists of 99 pages. Page 2 is requested and loaded from the hard disk to the RAM. b) Page 2 is now in RAM where it can be accessed by the CPU.

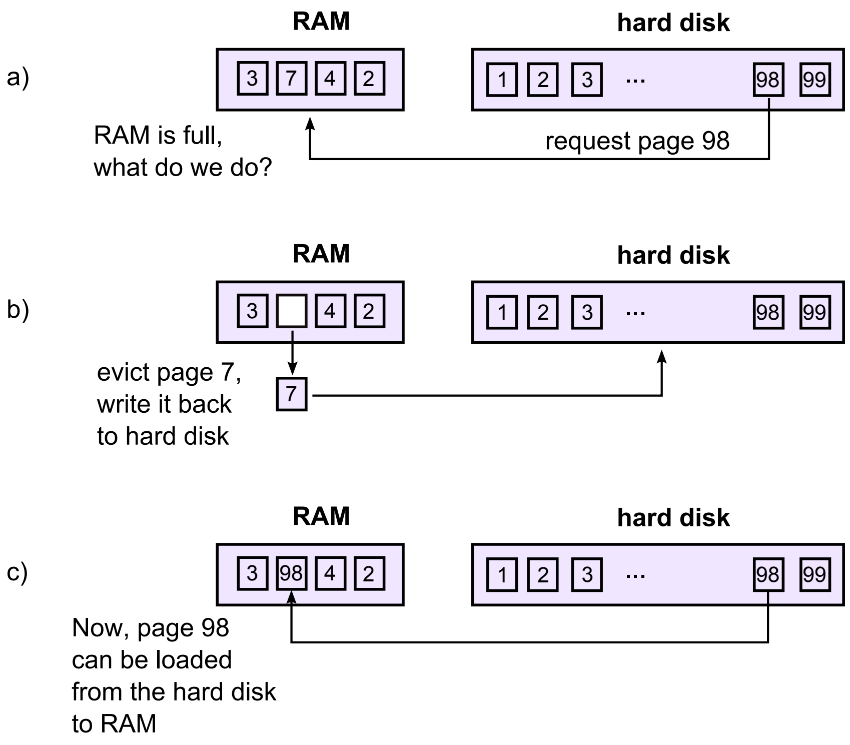


Figure 2: a) We request page 98 and try to load it from the hard disk to RAM. However, the RAM is full. b) We pick an arbitrary page and evict it, e.g. page 7. c) Now, page 98 can be loaded from the hard disk to RAM.

The question is, which page shall we evict? Does it even matter? Indeed, it does. Consider these two cases:

- Case 1: Suppose we have just evicted page 7 such that it is on the hard disk, and our next request is page 7. Then we have to access the hard disk to load page 7 to the RAM again.
- Case 2: Instead of page 7 we evict page 4. If we request page 7 now, nothing is to be done. We do not need to load page 7 from the hard disk to RAM since page 7 is still in RAM.

In case 2 we made a better decision since no access to the hard disk was necessary, and recall that accessing the hard disk is slow. In general the fast memory is called *cache* and the slow one simply *memory*. Thus, from now on we will refer to the RAM as cache and the hard disk as memory. When we request a page $p$ there are two possibilities:

1. $p$ is in cache and nothing is to be done.

2. $p$ is not in cache which we will refer to as *cache miss*. If the cache is full, a page has to be evicted such that $p$ can be loaded from memory to cache (page replacement).

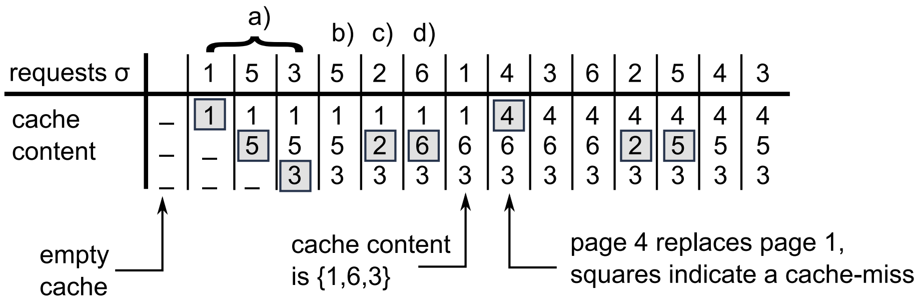
This is referred to as *paging* [5]. Thus, paging can be understood as memory management where we have to decide which pages to keep in cache. The goal is to minimize the number of cache misses. In the next section we will describe some paging algorithms.

## 1.2 Paging algorithms

We have mentioned that we are trying to minimize the number of cache misses, and that this number depends on which page we evict. We will describe some paging algorithms that use different page replacement policies. An excellent demonstration of the different paging algorithms can be found in [6]. In the following, $k$ is the cache size, i.e. the number of pages that fit in the cache, $M$ is the page-set, i.e. the set of all pages, and $\sigma$ is the request sequence.

### 1.2.1 Longest Forward Distance (LFD)

Suppose we have a cache-size of $k = 3$, and the page-set is $M = \{1, 2, 3, 4, 5, 6\}$. We request the pages $\sigma = (1, 5, 3, 5, 2, 6, 1, 4, 3, 6, 2, 5, 4, 3)$. The algorithm *Longest Forward Distance (LFD)* evicts the page whose next request lies furthest in the future. Let us have a look at the cache content in figure 3.



| requests σ | | 1 | 5 | 3 | 5 | 2 | 6 | 1 | 4 | 3 | 6 | 2 | 5 | 4 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cache content | _ | $\boxed{1}$ | 1 | 1 | 1 | 1 | 1 | 1 | $\boxed{4}$ | 4 | 4 | 4 | 4 | 4 | 4 |
| | _ | _ | $\boxed{5}$ | 5 | 5 | $\boxed{2}$ | $\boxed{6}$ | 6 | 6 | 6 | 6 | $\boxed{2}$ | $\boxed{5}$ | 5 | 5 |
| | _ | _ | _ | $\boxed{3}$ | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

Figure 3: Example for the algorithm LFD processing a request sequence $\sigma$. The cache size is $k = 3$. Numbers enclosed in a square indicate a cache miss. In total there are 8 cache misses.

a) In the beginning the cache is empty and is filled by the first three requests 1,5 and 3. These pages were not in the cache and result in three cache misses.
b) The next request is 5, and since 5 is already in the cache nothing is to do.

c) Page 2 is requested and is not in cache (cache miss). Since the cache is full we will have to evict a page, but which one? According to the LFD policy we evict the page whose next request lies furthest in the future, and that is page 5 (see figure 4). We evict page 5 and replace it by page 2.

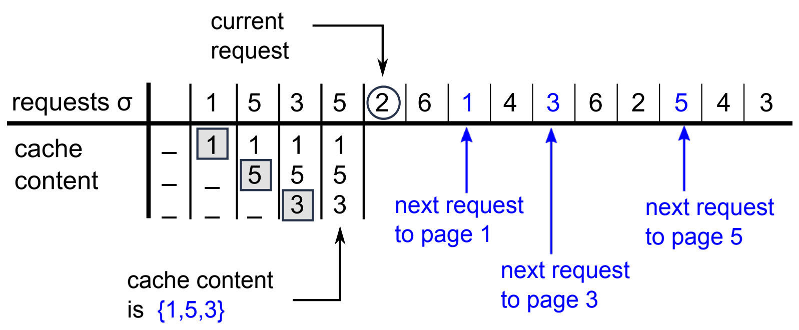


Figure 4: The cache content is {1,5,3}. Out of these three pages the next request for page 5 lies furthest in the future. Hence, LFD evicts page 5.

d) Page 6 is requested and is not in cache (cache miss). Again we evict the page whose next request lies furthest in the future, and that is page 2 (see figure 5).

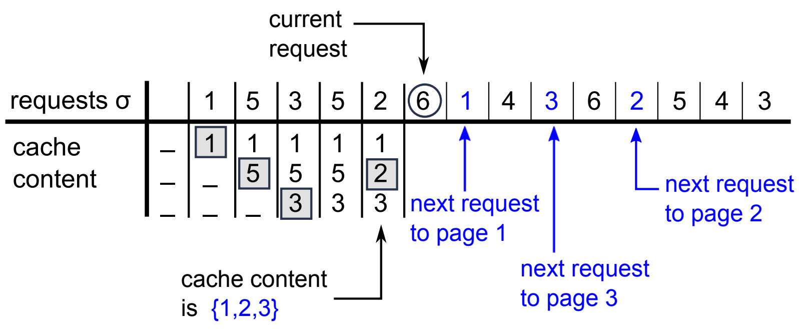


Figure 5: The cache content is {1,2,3}. Out of these three pages the next request for page 2 lies furthest in the future. Hence, LFD evicts page 2.

In figure 3 we can see how LFD processes the whole request sequence $\sigma$. There are in total 8 cache misses or 8 page faults. Note that it is common to use the term *page fault* instead of cache miss.

Belady devised LFD and proved in [1] that it is the *optimal* paging algorithm, i.e. there is no other algorithm that yields less page faults when processing a request sequence. However, it is not practical since for a current request it replaces the page whose next request lies furthest in the future. Knowledge of the future is usually not available to us.

LFD is called an *offline* paging algorithm which means that at any time the algorithm can look at the whole request sequence. In contrast, *online* paging algorithms only see the current and past requests. Their decision on which page to replace depends only on these requests.

### 1.2.2 Least Recently Used (LRU)

*Least Recently Used (LRU)* is an online paging algorithm, and as such decides which page to evict by only looking at the pages that have been requested so far. It replaces the page that has not been requested for the longest period of time. An example will illustrate this policy. Suppose we have a cache size of $k = 3$ and the page-set is $M = \{1, 2, 3, 4, 5, 6\}$. The request sequence is the same as before, i.e. $\sigma = (1, 5, 3, 5, 2, 6, 1, 4, 3, 6, 2, 5, 4, 3)$. Let us examine the cache content.

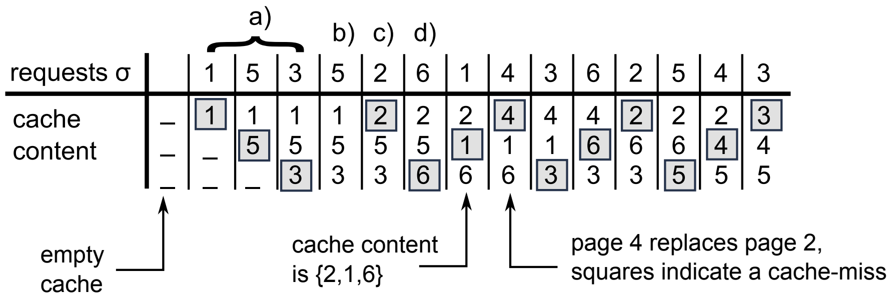


Figure 6: Example for the algorithm LRU processing a request sequence $\sigma$. The cache size is $k = 3$. Numbers enclosed in a square indicate a cache miss.

a) The first three requests 1,5 and 3 will just fill the empty cache. These pages have not been in the cache and evoke three cache misses.
b) The next request is 5, and since 5 is already in the cache nothing is to do.
c) Page 2 is requested and not in cache (cache miss). Since the cache is full we have to replace a page. According to the LRU policy we evict the page that has not been requested for the longest period of time, and that is page 1 (see figure 7).
d) Page 6 is requested and not in cache (cache miss). Since the cache is full we have to replace a page. According to the LRU policy we evict the page that has not been requested for the longest period of time, and that is page 3 (see figure 8).
In figure 6 we can see how LRU processes the whole request sequence. There are in total 13 page faults.

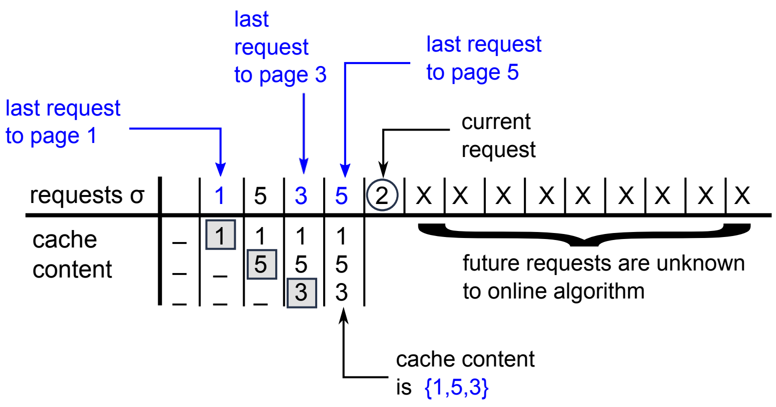


Figure 7: The cache content is {1,5,3}. Out of these three pages 1 has not been requested for the longest period of time. Hence, LRU evicts page 1. Notice that the future requests are unknown to LRU.

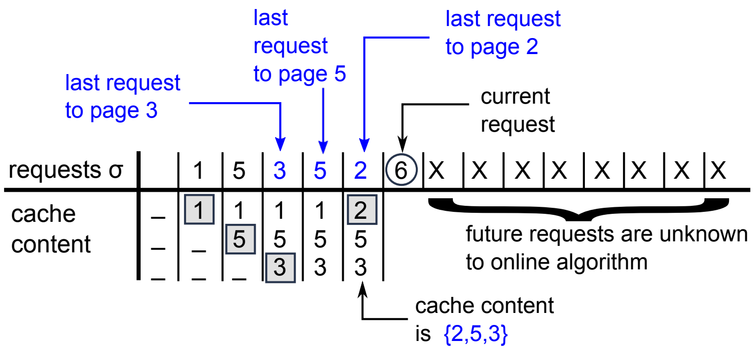


Figure 8: The cache content is {2,5,3}. Out of these three pages 3 has not been requested for the longest period of time. Hence, LRU evicts page 3. Notice that the future requests are unknown to LRU.

## 1.3 Competitive ratio

There are many more paging algorithms [8]. The question is, how good are they? One popular way to gauge the performance of a paging algorithm is to use the *competitive ratio* $c$ which is defined as follows [5]: an algorithm A is $c$-*competitive* if

$$\mathrm{A}(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + b$$

holds for an arbitrary request sequence $\sigma$, where $\mathrm{A}(\sigma)$ is the number of cache misses performed by A, $c$ is the competitive ratio, $\mathrm{OPT}(\sigma)$ is the number of cache misses caused by the optimal paging algorithm OPT and $b$ is some constant. We know from section 1.2.1 that LFD is the optimal paging algorithm such that we will refer to LFD as OPT.

Intuitively, the competitive ratio compares the performance of A and OPT. For instance, it is known that LRU is $k$-competitive [9], where $k$ is the cache size:

$$\mathrm{LRU}(\sigma) \leq k \cdot \mathrm{OPT}(\sigma) + b$$

For $k = 3$ we have

$$\mathrm{LRU}(\sigma) \leq 3 \cdot \mathrm{OPT}(\sigma) + b$$

which guarantees that in the worst case LRU can only have up to three times as many page faults as OPT (plus some constant $b$). However, this worst-case guarantee has the drawback of not being meaningful in practice. For example, LRU and the algorithm FIFO are both known to be $k$-competitive [9], however LRU performs better in practice than FIFO [10].

A more experimental approach is to consider a real-world trace, i.e. a long sequence $\sigma$ of page requests that has been obtained by tracing memory accesses. For this trace we count the number of cache misses by A and OPT respectively and calculate $A(\sigma)/OPT(\sigma)$ which serves as an empirical competitive ratio [7]. We typically determine the ratio for different values of $k$ and plot $A(\sigma)/OPT(\sigma)$ against $k$, see figure 9. In order to determine the ratio we obviously need the algorithm OPT for which we will describe an efficient implementation in the next sections.

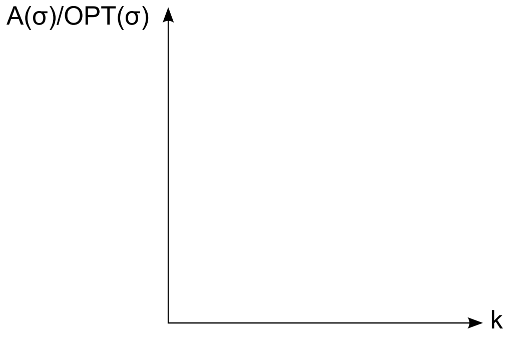


Figure 9: In experiments the empirical ratio $A(\sigma)/OPT(\sigma)$ is plotted against different cache sizes $k$.

# 2 Optimal offline paging algorithm

## 2.1 Layer partition

We have mentioned that the Longest Forward Distance algorithm (LFD) yields the minimum number of cache misses when processing a request sequence. As such we refer to it as the optimal offline paging algorithm OPT meaning that no other algorithm performs better. It works by evicting the page in cache whose next request lies furthest in the future. Obviously, knowledge about the cache content is required. However, Koutsoupias and Papadimitriou have found something remarkable in [11]: when requesting a page it is possible to determine whether OPT will perform a cache miss without knowing the precise cache content! Instead a *layer partition* is used. Later, a *compressed* version of the layer partition was introduced by Moruz and Negoescu in [12] which we will describe here.

The compressed layer partition $(\mathcal{L}_0|\mathcal{L}_1|...|\mathcal{L}_k)$ consists of $k+1$ layers (disjoint sets) $\mathcal{L}_0, \mathcal{L}_1, ..., \mathcal{L}_k$, where $k$ is the number of pages that fit in the cache. Each page is in exactly one layer. Example: For $k = 3$ we have $k + 1 = 4$ layers, and the partition may be given by $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (1, 3|\emptyset|4, 2, 5|6)$.

Let $p_1, p_2, ..., p_k$ be the first $k$ pairwise distinct pages in the request sequence $\sigma$, and $M$ be the set of all pages. The initial partition is then defined as

$$\begin{aligned}&\mathcal{L}_0 = M \backslash \{p_1, ..., p_k\}\\ &\mathcal{L}_1, ..., \mathcal{L}_{k-1} \text{ are empty sets}\\ &\mathcal{L}_k = \{p_1, ..., p_k\}\end{aligned}$$

**Example 0**: Let $M = \{1, 2, 3, 4, 5, 6\}$, $k = 3$ and $\sigma = (2, 5, 5, 2, 1, 3, 4, 6, 2, 6, 4)$. The first $k = 3$ pairwise distinct requested pages in $\sigma$ are $\{2, 5, 1\}$, such that the initial partition is given by $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (3, 4, 6|\emptyset|\emptyset|2, 5, 1)$.

Upon a request to page $p$ the layer partition is modified according to the following *update rules*, where $\mathcal{L}'_i$ is the updated layer and $\mathcal{L}_i$ the old layer:

**Case 1:** if $p \in \mathcal{L}_0$, then
$\mathcal{L}'_0 = \mathcal{L}_0 \backslash \{p\}$
$\mathcal{L}'_{k-1} = \mathcal{L}_{k-1} \cup \mathcal{L}_k$
$\mathcal{L}'_k = \{p\}$
and $\mathcal{L}'_j = \mathcal{L}_j$ for $j \neq 0, k-1, k$

**Case 2:** if $p \in \mathcal{L}_k$, then
do nothing

**Case 3:** if $p \in \mathcal{L}_i, 0 < i < k$, then
$\mathcal{L}'_{i-1} = \mathcal{L}_{i-1} \cup \mathcal{L}_i \backslash \{p\}$
$\mathcal{L}'_k = \mathcal{L}_k \cup \{p\}$
$\mathcal{L}'_j = \mathcal{L}_{j+1}$ for $j = i, i+1, ..., k-2$
$\mathcal{L}'_{k-1} = \emptyset$
and $\mathcal{L}'_j = \mathcal{L}_j$ for $j = 0, ..., i-2$

A more visual representation of the updated layer partition $\mathcal{L}^p$ is the following:

$$\boxed{\mathcal{L}^p = \begin{cases} (\mathcal{L}_0\backslash\{p\}|\mathcal{L}_1|\dots|\mathcal{L}_{k-2}|\mathcal{L}_{k-1}\cup\mathcal{L}_k|\{p\}), & \text{if } p\in\mathcal{L}_0 \\ (\mathcal{L}_0|\mathcal{L}_1|\dots|\mathcal{L}_{k-1}|\mathcal{L}_k), & \text{if } p\in\mathcal{L}_k \\ (\mathcal{L}_0|\dots|\mathcal{L}_{i-2}|\mathcal{L}_{i-1}\cup\mathcal{L}_i\backslash\{p\}|\mathcal{L}_{i+1}|\dots|\mathcal{L}_{k-1}|\emptyset|\mathcal{L}_k\cup\{p\}), & \text{if } p\in\mathcal{L}_i, \\ & 0<i<k \end{cases}}$$

- **Case 1**: If $p \in \mathcal{L}_0$, then $\mathcal{L}_k$ is merged into $\mathcal{L}_{k-1}$. Afterwards $p$ is removed from $\mathcal{L}_0$, and the new layer $\mathcal{L}'_k$ contains only $p$.
- **Case 2**: If $p \in \mathcal{L}_k$, then nothing is to do.
- **Case 3**: If $p \in \mathcal{L}_i$ with $0 < i < k$, i.e. $p$ is neither in $\mathcal{L}_0$ nor in $\mathcal{L}_k$, then $p$ is removed from $\mathcal{L}_i$ and inserted in $\mathcal{L}_k$, and $\mathcal{L}_i\backslash\{p\}$ is merged into $\mathcal{L}_{i-1}$. Moreover, the content of the layers $\mathcal{L}_j$ with $j = i+1, i+2, \dots, k-1$ all move one position to the left. As a consequence the new layer $\mathcal{L}'_{k-1}$ becomes empty.

The following examples illustrate the update rules.

**Example 1:** We consider case 1 with $p \in \mathcal{L}_0$. The layer partition may be given by

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 9,10 | 7,8 | 5,6 | 3,4 | 1,2 |

For the layer partition above we request page $p = 9$. Since $p \in \mathcal{L}_0$, we (1.i) merge $\mathcal{L}_k$ into $\mathcal{L}_{k-1}$:

(1.i)

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 9,10 | 7,8 | 5,6 | 1,2,3,4 | $\emptyset$ |

(1.ii) We then remove $p$ from $\mathcal{L}_0$ and finally insert it into the last layer:

(1.ii)

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 10 | 7,8 | 5,6 | 1,2,3,4 | 9 |

In case 1 not much happens. Intuitively, page $p$ moves to $\mathcal{L}_k$ and pushes the content of $\mathcal{L}_k$ one position to the left, to $\mathcal{L}_{k-1}$.

**Example 2:** We consider case 2 with $p \in \mathcal{L}_k$. The layer partition is given by

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 9,10 | 7,8 | 5,6 | 3,4 | 1,2 |

We request page 2 and then page 1. This requires no further action since both pages are in $\mathcal{L}_k$.

**Example 3:** We consider case 3 with $p \in \mathcal{L}_i, 0 < i < k$. The layer partition is given by

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 9,10 | 7,8 | 5,6 | 3,4 | 1,2 |

We request page $p = 7$. Since $p \in \mathcal{L}_i$ with $0 < i < k$, i.e. $p$ is neither from $\mathcal{L}_0$ nor $\mathcal{L}_k$, we do the following:
(3.i) $p = 7$ is removed from $\mathcal{L}_i = \mathcal{L}_1$ and inserted in $\mathcal{L}_k$:

(3.i)

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 9,10 | 8 | 5,6 | 3,4 | 1,2,7 |

(3.ii) $\mathcal{L}_i \backslash \{p\} = \{8\}$ is merged into $\mathcal{L}_{i-1}$:

(3.ii)

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 8,9,10 | $\emptyset$ | 5,6 | 3,4 | 1,2,7 |

(3.iii) The contents of layers $\mathcal{L}_2$ and $\mathcal{L}_3$ move one position to the left. The new layer $\mathcal{L}_{k-1} = \mathcal{L}_3$ is empty now:

(3.iii)

| $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3$ | $\mathcal{L}_4 = \mathcal{L}_k$ |
|---|---|---|---|---|
| 8,9,10 | 5,6 | 3,4 | $\emptyset$ | 1,2,7 |

The formal description of case 3 may look complicated but is actually easy to understand. Intuitively, if we request a page $p \in \mathcal{L}_i$ with $0 < i < k$, then all the layers from $\mathcal{L}_i$ to $\mathcal{L}_{k-1}$ jump one position to the left, and as a consequence $\mathcal{L}_{k-1}$ becomes empty. Then page $p$ is moved to $\mathcal{L}_k$.

**Example 4**: We also want to consider how the layer partition processes a whole request sequence $\sigma$ from beginning to end. Let $M = \{1, 2, 3, 4, 5, 6\}$ be the page-set, $k = 3$ the cache size and $\sigma = (4, 2, 4, 5, 6, 4, 1, 2, 1, 6, 2, 4)$ the request sequence.

The first $k = 3$ pairwise distinct requested pages are $\{4, 2, 5\}$, from which we can deduce the initial partition $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (1, 3, 6|\emptyset|\emptyset|4, 2, 5)$, see also example 0. Table 1 shows the requests after these first 3 distinct pages.

| request | $\mathcal{L}_0$ | $\mathcal{L}_1$ | $\mathcal{L}_2$ | $\mathcal{L}_3 = \mathcal{L}_k$ | OPT |
|---|---|---|---|---|---|
| 4,2,5 | 1,3,6 | $\emptyset$ | $\emptyset$ | 4,2,5 | 3 cache misses |
| $6 \in \mathcal{L}_0$ | 1,3 | $\emptyset$ | 4,2,5 | 6 | 1 cache miss |
| $4 \in \mathcal{L}_2$ | 1,3 | 2,5 | $\emptyset$ | 6,4 | |
| $1 \in \mathcal{L}_0$ | 3 | 2,5 | 6,4 | 1 | 1 cache miss |
| $2 \in \mathcal{L}_1$ | 3,5 | 6,4 | $\emptyset$ | 1,2 | |
| $1 \in \mathcal{L}_3$ | 3,5 | 6,4 | $\emptyset$ | 1,2 | |
| $6 \in \mathcal{L}_1$ | 3,5,4 | $\emptyset$ | $\emptyset$ | 1,2,6 | |
| $2 \in \mathcal{L}_3$ | 3,5,4 | $\emptyset$ | $\emptyset$ | 1,2,6 | |
| $4 \in \mathcal{L}_0$ | 3,5 | $\emptyset$ | 1,2,6 | 4 | 1 cache miss |

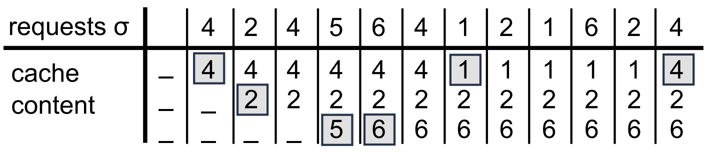
Table 1: Illustration of example 4. The layer partition processes a request sequence $\sigma$. The second line shows the initial partition after the first $k = 3$ pairwise distinct pages in $\sigma$. The last column describes OPT's behaviour according to property (Pr1). There are in total 6 cache misses.

Now, the layer partition has an interesting property [7]:

**Property (Pr1):** After the first $k$ pairwise distinct requested pages OPT will perform a cache miss upon a request to page $p$, if and only if $p \in \mathcal{L}_0$ in the layer partition.

Let us examine what this means for the request sequence in example 4. In table 1 we see that there are 3 requests with $p \in \mathcal{L}_0$ (pages 6,1,4). OPT will perform a cache miss for these requests according to (Pr1) such that we have 3 page faults.

Moreover, we know that OPT's cache is empty in the beginning, so the first $k = 3$ pairwise distinct requested pages could not have been in the cache and lead to 3 additional cache misses. In total OPT should have performed 3+3=6 page faults. We should check this using the LFD policy from section 1.2.1.



| requests σ | | 4 | 2 | 4 | 5 | 6 | 4 | 1 | 2 | 1 | 6 | 2 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cache | – | $\boxed{4}$ | 4 | 4 | 4 | 4 | 4 | $\boxed{1}$ | 1 | 1 | 1 | 1 | $\boxed{4}$ |
| content | – | – | $\boxed{2}$ | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | – | – | – | – | $\boxed{5}$ | $\boxed{6}$ | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

Figure 10: The algorithm LFD processing the request sequence $\sigma$ from example 4. Squares indicate a cache miss. There are in total 6 page faults.

As you can see in figure 10 there are indeed 6 page faults. Compare figure 10 with table 1 and notice how OPT will perform a cache miss, if and only if the requested page is from $\mathcal{L}_0$ in the layer partition (after the first $k$ pairwise distinct requested pages in the beginning).

This is quite a remarkable result since it allows us to determine whether OPT will perform a page fault without using the LFD policy! Instead, all we have to do is use the layer partition. Notice that when we use the LFD policy we form OPT's cache content. In contrast, when we use the layer partition we do not know the precise content of OPT's cache (see appendix). The question is, how do we implement the layer partition efficiently? We will answer this in the next sections.

## 2.2 Union-Find data structure

We have seen that the update rules for the layer partition include the union of disjoint sets, namely the layers. Galler and Fischer [13] developed an efficient *union-find* data structure to maintain elements that are partitioned in disjoint sets. Each set consists of a tree of elements with a parent pointer, and is represented by the root whose parent pointer points to itself.
Example: Suppose we have the set $S = \{1, 2, ..., 8\}$ that is partitioned into $P_1 = \{1, 2\}$, $P_2 = \{3, 4, 5, 6, 7\}$, $P_3 = \{8\}$. This partition may be represented by the trees in figure 11.

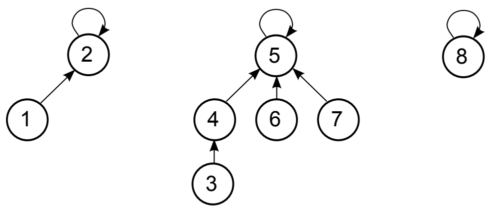


Figure 11: Example for the Galler-Fischer tree data structure. The trees represent the sets $P_1 = \{1, 2\}$, $P_2 = \{3, 4, 5, 6, 7\}$ and $P_3 = \{8\}$.

A root is identified by checking if the parent pointer points to itself. This is the case for elements 2, 5 and 8. The root denotes the set, e.g. 5 represents $P_2$. This allows us to determine whether two elements are in the same set simply by comparing their roots. For instance, 3 and 7 belong to the same set since their root is 5.

The data structure supports three operations: *union*, *find* and *makeset*. The *find* operation determines in which set an element is by following the parent pointers till the root. The *union* operation on two sets works by linking the root of one tree to the root of the other tree, see figure 12. A *makeset* operation creates a node with a parent pointer pointing to itself.

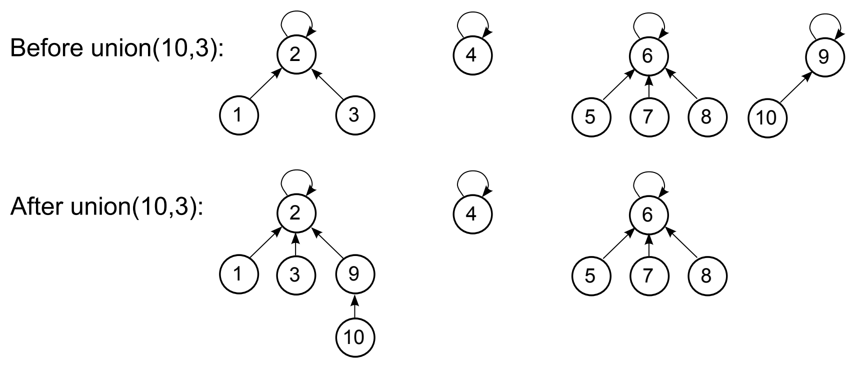


Figure 12: Example for a union operation in the Galler-Fischer tree data structure. Union(10,3) is performed by first finding the root of 10 and 3 respectively via find(10) and find(3). The root of 10 is then linked to the root of 3.

*Union by rank.* Under certain circumstances a find operation can be expensive, e.g. if we have elements $1, 2, ..., n$ and apply the consecutive operations union(1,2), union(2,3), ..., union($n$-1,$n$), then a tree of height $n - 1$ is generated. A find operation will then require $\theta(n)$ time in the worst case.

A technique known as *union by rank* [16] prevents this by considering the *rank* which is an upper bound for the height of a tree [29, 15]. The tree with smaller rank is then linked to the tree with greater rank, see figure 13. This keeps the trees flat and guarantees that a union or find operation will run in $\mathcal{O}(\log(n))$ time.

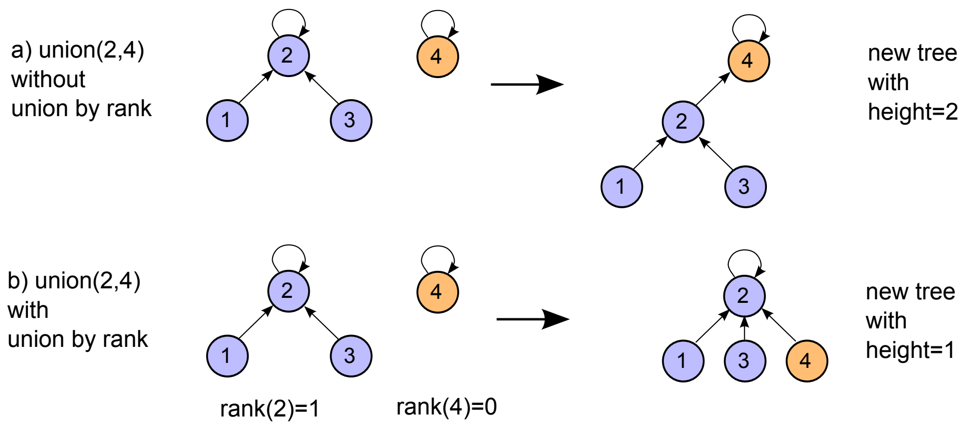


Figure 13: Two trees are unified via union(2,4). In a) the union is performed without union by rank which generates a tree of height 2. In contrast, b) is performed with union by rank, i.e. the tree with smaller rank is linked to the tree with greater rank. This yields a tree of height 1.

*Path compression.* The find operation involves following the parent pointers until we reach the root $r$. However, we can improve the runtime if we traverse the path to $r$ for a second time and link each node we pass directly to $r$, see figure 14. This *path compression* will make subsequent find operations for these attached nodes and their children faster.

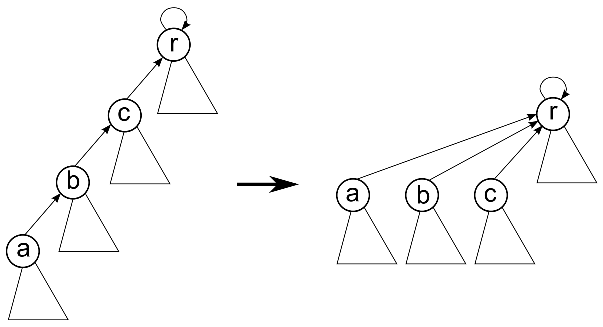


Figure 14: find(a) follows the parent pointers until node $r$ passing the nodes $b$ and $c$. In a second traversal $a$, $b$ and $c$ are directly linked to $r$. Subsequent find operations for $a$, $b$, $c$ and their children (nodes in subtrees) run faster.

Using union by rank together with path compression Tarjan showed the following in [15, 16, 14]:

**Tarjan's result**: Let $n$ be the number of elements, each one in a different set such that we have $n$ singleton sets. Performing $\mu$ union- or find operations on these sets (with $\mu \geq n$) requires $\mathcal{O}(\mu \cdot \alpha(n))$ time, where $\alpha(n)$ is a very slowly growing function known as inverse Ackermann function [17].

Note that the *amortized runtime* per operation is $\mathcal{O}(\alpha(n))$, which is effectively constant since $\alpha(n)$ is smaller than 5 for all practical values of $n$, see [18]. To emphasize the meaning of this let us review the runtime improvements for the union-find data structure by Galler and Fischer for $n$ elements and $\mu \geq n$ operations:

- No union by rank and no path compression:
  runtime $= \mu \cdot \mathcal{O}(n)$, since the worst-case requires $\theta(n)$ time
- Union by rank, but no path compression:
  runtime $= \mu \cdot \mathcal{O}(\log(n))$
- Union by rank with path compression:
  runtime $= \mu \cdot \mathcal{O}(\alpha(n))$, approximately $\mu \cdot \mathcal{O}(1)$

A nice java-applet that demonstrates this union-find data structure can be found in [20]. Note that during path compression the rank, which is an upper bound for the height, does not change [21].

## 2.3 Linked trees implementation of the layer partition (LTrees)

The update rules for the layer partition include the *union* of disjoint sets and the *deletion* of an element from a set. It is therefore obvious to adopt the union-find tree data structure by Galler and Fischer. However, we have to modify it since it does not support the deletion of elements, and deleting an element efficiently is not a trivial task. Here, we had the idea [23] of using *lazy deletion*, i.e. we do not immediately delete a node but instead mark it. Marked nodes are then deleted in batches.

We implement the layer partition as follows: each layer is represented by a tree, and the roots of adjacent layers are linked to each other, see figure 15.

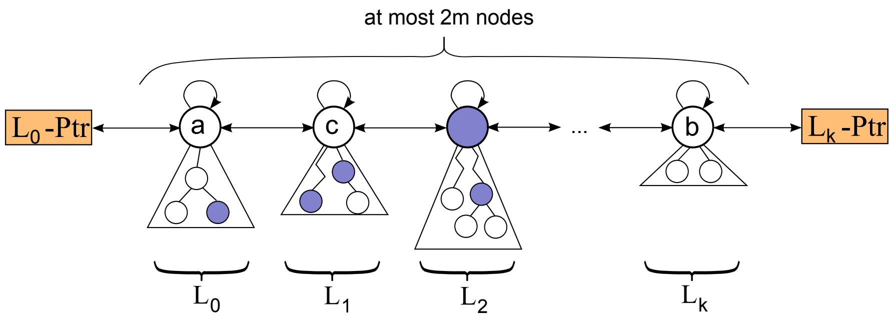


Figure 15: The data structure for the layer partition $(\mathcal{L}_0|\mathcal{L}_1|...|\mathcal{L}_k)$. Each layer is represented by a tree. $\mathcal{L}_0$-Ptr and $\mathcal{L}_k$-Ptr are special pointers used to decide whether a page is in $\mathcal{L}_0$ or $\mathcal{L}_k$. The trees with root nodes $a$ and $b$ represent layers $\mathcal{L}_0$ and $\mathcal{L}_k$ respectively. Filled nodes are marked as deleted. There are at most $k+1$ trees with $k$ being the cache size. The total number of nodes is restricted to $2m$, where $m$ is the page-set size.

Recall that we need to determine whether a page $p$ is in $\mathcal{L}_0$, $\mathcal{L}_k$ or in $\mathcal{L}_i$ with $0 < i < k$ in order to apply the appropriate update rule. This can be done with the help of the pointers $\mathcal{L}_0$-Ptr, $\mathcal{L}_k$-Ptr and the find($p$) operation which returns the root of $p$:

- If $a$=find($p$), i.e. $a$ is the root of $p$, and $\mathcal{L}_0$-Ptr points to $a$, then $p$ is in $\mathcal{L}_0$ (see root node $a$ in figure 15).
- If $b$=find($p$), i.e. $b$ is the root of $p$, and $\mathcal{L}_k$-Ptr points to $b$, then $p$ is in $\mathcal{L}_k$ (see root node $b$ in figure 15).
- If $c$=find($p$), i.e. $c$ is the root of $p$, and neither $\mathcal{L}_0$-Ptr nor $\mathcal{L}_k$-Ptr point to $c$, then $p$ is in $\mathcal{L}_i$ with $0 < i < k$ (see e.g. root node $c$ in figure 15).

Empty layers are not represented by a tree. Instead every root node has a variable *count* that stores the number of consecutive empty sets that are adjacent to the left, see figure 16.

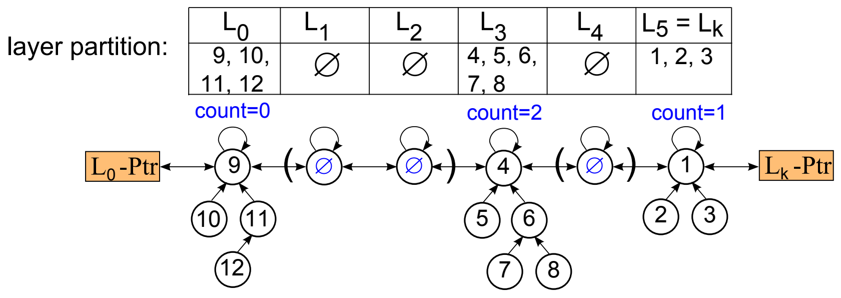


Figure 16: The layer partition is represented by the linked trees. The count variable describes the number of empty sets adjacent to the left. The nodes with empty sets do not exist and are only drawn for illustration purposes.

The update rules upon request to page $p$ are implemented as follows:

- Case (i): If $p \in \mathcal{L}_0$, then $\mathcal{L}_k$ is merged into $\mathcal{L}_{k-1}$. Afterwards, $p$ is removed from $\mathcal{L}_0$ and inserted as new layer $\mathcal{L}'_k$ such that $\mathcal{L}'_k = \{p\}$.

  This is done by marking the corresponding node for $p$ in the tree of $\mathcal{L}_0$ as deleted and inserting a new node for $p$ as new layer $\mathcal{L}'_k = \{p\}$. The union of $\mathcal{L}_k$ and $\mathcal{L}_{k-1}$ is performed by linking the root node in $\mathcal{L}_k$ to the root node in $\mathcal{L}_{k-1}$, (or vice versa link the root node in $\mathcal{L}_{k-1}$ to $\mathcal{L}_k$ depending on the ranks) see figure 17. If $\mathcal{L}_{k-1}$ is empty, such that $\mathcal{L}_k$ is merged with an empty layer, then the count variable for layer $\mathcal{L}_k$ is decremented since the number of empty sets to the left decreases.

- Case (ii): If $p \in \mathcal{L}_k$, then nothing is to do.

- Case (iii): If $p \in \mathcal{L}_i$ with $0 < i < k$, then $p$ is removed from $\mathcal{L}_i$ and added to $\mathcal{L}_k$. Afterwards $\mathcal{L}_i \backslash \{p\}$ is merged into $\mathcal{L}_{i-1}$, and an empty set is inserted as new layer $\mathcal{L}'_{k-1}$ such that $\mathcal{L}'_{k-1} = \emptyset$.

  This is done by marking the corresponding node for $p$ in the tree of $\mathcal{L}_i$ as deleted and linking a new node for $p$ to the tree of $\mathcal{L}_k$. The trees of $\mathcal{L}_i$ and $\mathcal{L}_{i-1}$ are then merged (alternatively, if $\mathcal{L}_{i-1}$ is empty, then the count variable for $\mathcal{L}_i$ is decremented). Inserting an empty set as $\mathcal{L}'_{k-1}$ is performed by increasing the count variable of the root node in $\mathcal{L}_k$, see figure 18.

Note that in (i) and (ii) we can check efficiently whether $p$ is in $\mathcal{L}_0$ or $\mathcal{L}_k$ with the help of the pointers $\mathcal{L}_0$-Ptr and $\mathcal{L}_k$-Ptr. In contrast, in (iii) we cannot determine efficiently the index $i$ for a page $p \in \mathcal{L}_i$ [22].

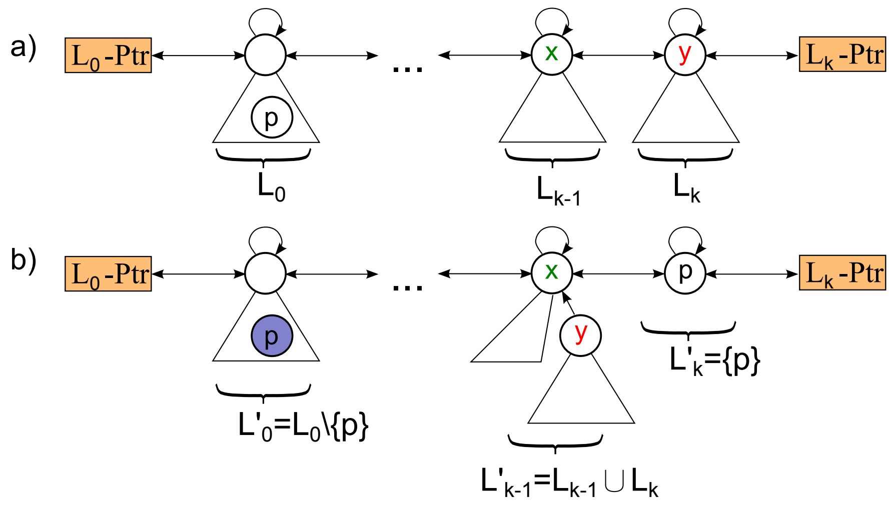


Figure 17: Implementation of case 1 of the update rules for the layer partition. a) shows the data structure before and b) after the request to page $p \in \mathcal{L}_0$. The filled node represents a node that is marked as deleted. Node $y$ is linked to node $x$. (or vice versa $x$ is linked to $y$ depending on the ranks).

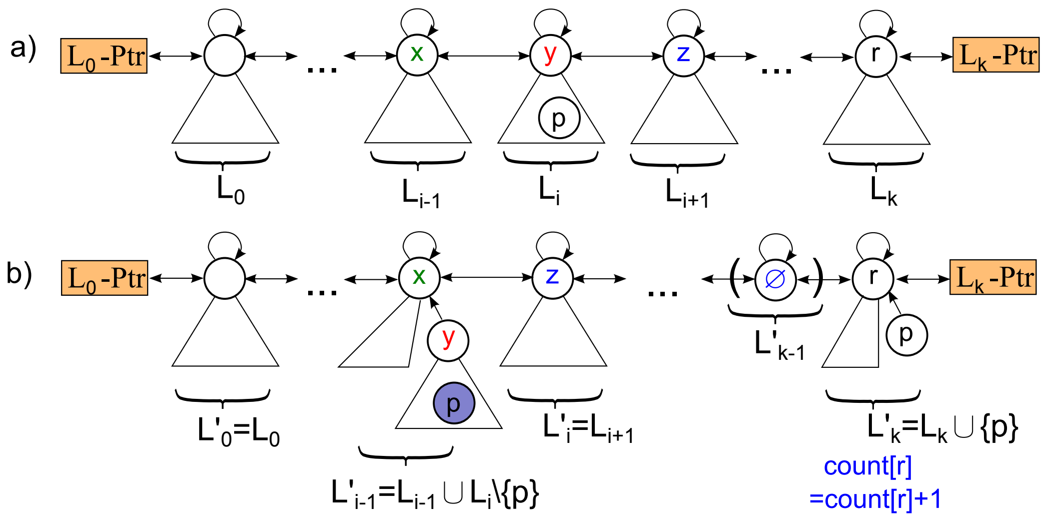


Figure 18: Implementation of case 3 of the update rules for the layer partition. a) shows the data structure before and b) after the request to page $p \in \mathcal{L}_i$ with $0 < i < k$. The filled node represents a node that is marked as deleted. An empty set is inserted by increasing the count variable of node $r$. Node $y$ is linked to node $x$ (or vice versa $x$ is linked to $y$ depending on the ranks).

### 2.3.1 Cleanup procedure for LTrees

*Lazy deletion.* In the cases $p \in \mathcal{L}_0$ and $p \in \mathcal{L}_i$ with $0 < i < k$ the number of nodes increases by one since we mark the old node for $p$ and insert a new node for $p$. To prevent an arbitrary growth of the data structure we will restrict the number of nodes to $2m$, where $m$ is the is the page-set size. If the total number of nodes reaches the value $2m$, i.e. there are $m$ regular and $m$ marked nodes, we perform a cleanup procedure. The cleanup procedure removes all marked nodes and consists of the following steps (see figure 19):

*Step 1*: Apply a find operation with path compression to all $2m$ nodes. This step creates trees with a height of at most 1.

*Step 2*: Half of the nodes are marked as deleted. We remove them as follows:
For all nodes $u$:
Case 1: If $u$ is a root, do nothing
Case 2: If $u$ is a child:
  if $u$ is marked as deleted, then remove the node
  if $u$ is not marked as deleted:
    if parent (root) of $u$ is not marked as deleted, then do nothing
    if parent (root) of $u$ is marked as deleted, then replace the root by $u$

*Step 3*: After step 2 there may still be some marked root nodes without children. These will be removed by simply going through all root nodes.

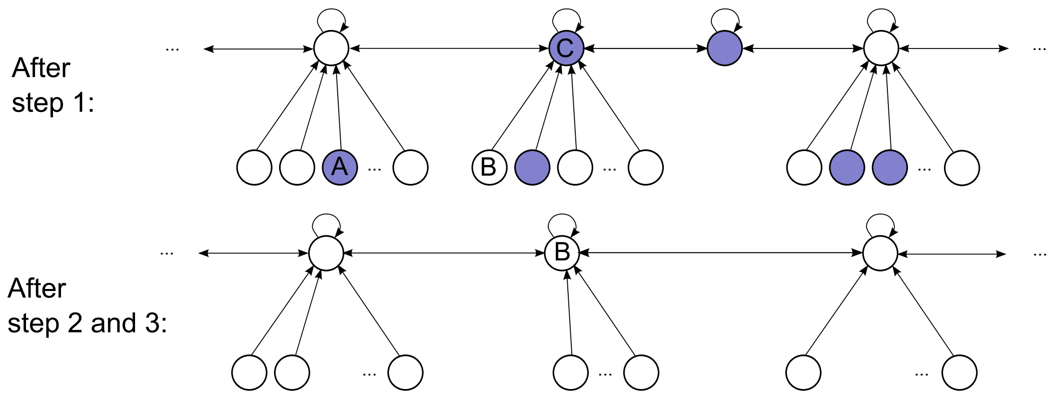


Figure 19: Cleanup procedure: After step 1 all trees have height of at most 1. Filled nodes represent nodes that are marked as deleted. In step 2 and 3 marked nodes are removed. Node A is removed. Node B replaces node C.

We will refer to this data structure as *LTrees* which stands for **L**inked **T**rees. Recall that LTrees implements the layer partition and can therefore tell us whether OPT will perform a cache miss upon a page request.

### 2.3.2 Runtime analysis for LTrees

When a page $p$ is requested we pass it to our linked trees data structure which applies the appropriate update rule and tells us whether OPT will perform a cache miss. The question is, how much time LTrees needs to process the request. There are two phases, see figure 20.

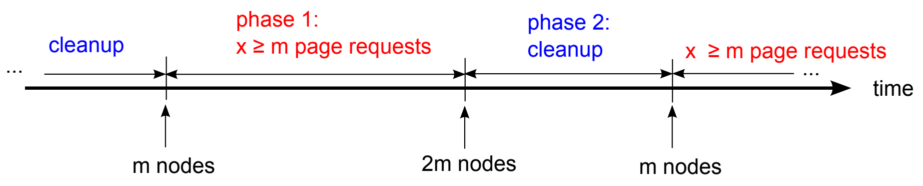


Figure 20: There are two phases for LTrees. In phase 1 we start with $m$ nodes and request pages until we have $2m$ nodes. Then in phase 2 the cleanup procedure takes place removing $m$ marked nodes.

**Runtime for phase 1:**
- At the beginning of phase 1 we have exactly $m$ regular nodes in LTrees, namely one node for each page, where $m$ is the page-set size.

- As we request pages, LTrees grows in the cases where $p \in \mathcal{L}_0$ and $p \in \mathcal{L}_i$ with $0 < i < k$ until it consists of $2m$ nodes. Since we start with $m$ nodes we need at least $m$ requests to reach the size of $2m$ nodes.

- We could need more than $m$ requests because a request does not necessarily add a node to LTrees (case $p \in \mathcal{L}_k$). Thus, in phase 1 we perform $x \geq m$ page requests until the data structure consists of $2m$ nodes.

- Each request requires at least a find operation (to determine in which layer a requested page $p$ is) and possibly a union operation (cases $p \in \mathcal{L}_0$ and $p \in \mathcal{L}_i$ with $0 < i < k$) yielding at most $2x$ union or find operations. Since the union and find operation require $\mathcal{O}(\alpha(2m))$ time according to Tarjan's result, the total runtime $t_1$ for phase 1 is

$$t_1 = 2x \cdot \mathcal{O}(\alpha(2m)) = x \cdot \mathcal{O}(\alpha(2m)).$$

- However, note that Tarjan's result assumes that we start with $m$ singleton sets (situation A). Instead at the beginning of phase 1 we have $m$ nodes, and some of them form trees of height 1 (situation B), see again figure 19. Thus not all trees represent singleton sets as opposed to situation A. The question is, are we allowed to use Tarjan's result?

We can use Tarjan's result if we pretend having started at situation A and getting to situation B with the following *transformation*, see figure 21. In situation A we have $m$ singleton sets. We first perform at most $m-1$ union operations to get trees of height 1 ($m-1$ is the maximum number of union operations we can perform on $m$ elements until all elements are in the same tree). To finally arrive at situation B we link $k+1$ root nodes to each other and set the pointers $\mathcal{L}_0$-Ptr and $\mathcal{L}_k$-Ptr which requires $k \cdot \mathcal{O}(1)$ time. The root nodes can be found by applying $m$ find operations. In summary, the transformation and the $x$ page requests imply

1. $(m-1)$ union operations
2. linking roots requiring $k \cdot \mathcal{O}(1)$ time
3. $m$ find operations
4. $2x$ union or find operations

Therefore, we have $[(m-1)+m+2x]$ union or find operations each taking $\mathcal{O}(\alpha(2m))$ time.

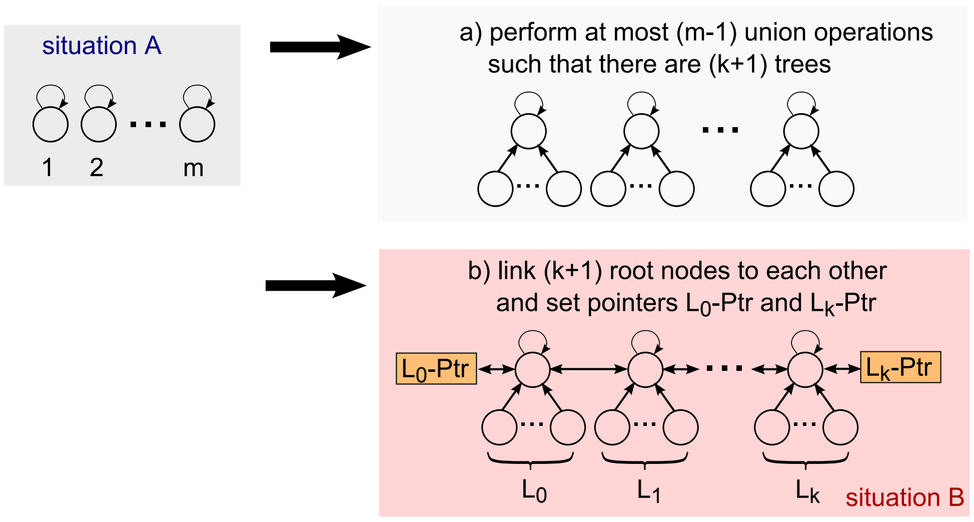

situation A
1
2
m
a) perform at most (m-1) union operations such that there are (k+1) trees
b) link (k+1) root nodes to each other and set pointers L0-Ptr and Lk-Ptr
L0-Ptr
Lk-Ptr
L0
L1
Lk
situation B

Figure 21: The transformation starting from situation A and ending in situation B. Situation B corresponds to LTrees right after a cleanup procedure, see also figure 19.

Let us see what happens if we include the transformation before phase 1. The runtime $t_1'$ including the transformation is calculated as follows (note that in phase 1 we process $x \geq m$ requests):

$$\begin{aligned}
t_1' =&[(m-1)+m+2x]\cdot\mathcal{O}(\alpha(2m))+k\cdot\mathcal{O}(1)\\
\leq&[2m+2x]\cdot\mathcal{O}(\alpha(2m))+k\cdot\mathcal{O}(1)\\
\leq&[2m+2x]\cdot\mathcal{O}(\alpha(2m))+k\cdot\mathcal{O}(\alpha(2m))\\
=&[2m+k+2x]\cdot\mathcal{O}(\alpha(2m))\\
\leq&[2m+m+2x]\cdot\mathcal{O}(\alpha(2m)),\ \text{since } m\geq k\\
\leq&[3m+3x]\cdot\mathcal{O}(\alpha(2m))\\
\leq&[3x+3x]\cdot\mathcal{O}(\alpha(2m)),\ \text{since } x\geq m\\
=&x\cdot\mathcal{O}(\alpha(2m))
\end{aligned}$$

Thus, the runtime for phase 1 remains $x\cdot\mathcal{O}(\alpha(2m))$ even if we include the transformation.

**Runtime for phase 2:**
- Now, we will examine the runtime of phase 2 where the cleanup procedure takes place. At this point LTrees consists of $2m$ nodes. For all these nodes we perform a find operation with path compression yielding a runtime of $2m\cdot\mathcal{O}(\alpha(2m))$, see step 1 of the cleanup procedure.

- Afterwards, nodes that are marked as deleted are removed by going through all nodes which takes $2m\cdot\mathcal{O}(1)$ time (step 2 of the cleanup procedure). Note that removing a marked node can be achieved in $\mathcal{O}(1)$ time, see again figure 19.

- In a last step marked root nodes without children are removed. Since there are at most $k+1$ root nodes, this requires $k\cdot\mathcal{O}(1)$ time (step 3 of the cleanup procedure). The total runtime $t_2$ for phase 2 is:

$$\begin{aligned}
t_2 =&2m\cdot\mathcal{O}(\alpha(2m))+2m\cdot\mathcal{O}(1)+k\cdot\mathcal{O}(1)\\
=&2m\cdot\mathcal{O}(\alpha(2m))+[2m+k]\cdot\mathcal{O}(1)\\
\leq&2m\cdot\mathcal{O}(\alpha(2m))+[2m+m]\cdot\mathcal{O}(1),\ \text{since } m\geq k\\
=&2m\cdot\mathcal{O}(\alpha(2m))+3m\cdot\mathcal{O}(1)\\
\leq&3m\cdot\mathcal{O}(\alpha(2m))+3m\cdot\mathcal{O}(1)\\
\leq&3m\cdot\mathcal{O}(\alpha(2m))+3m\cdot\mathcal{O}(\alpha(2m))\\
=&m\cdot\mathcal{O}(\alpha(2m))
\end{aligned}$$

**Amortized runtime per request**:
Considering phase 1 and 2 together we get:

$$\begin{aligned}
t_1 + t_2 =& x \cdot \mathcal{O}(\alpha(2m)) + m \cdot \mathcal{O}(\alpha(2m)) \\
=& [x + m] \cdot \mathcal{O}(\alpha(2m)) \\
\leq& [x + x] \cdot \mathcal{O}(\alpha(2m)), \text{ since } x \geq m \\
=& 2x \cdot \mathcal{O}(\alpha(2m)) \\
=& x \cdot \mathcal{O}(\alpha(2m))
\end{aligned}$$

Since in phase 1 and 2 we have in total $x$ page requests, the *amortized* runtime $t_{am}$ per page request is:

$$t_{am} = \frac{t_1 + t_2}{x} = \frac{x \cdot \mathcal{O}(\alpha(2m))}{x} = \mathcal{O}(\alpha(2m))$$

Recall that the inverse Ackermann function $\alpha$ is an extremely slow growing function with $\alpha(n) < 5$ for all practical values of $n$, which makes $t_{am}$ effectively constant. Moreover, $t_{am}$ is independent of the cache size $k$ making the amortized runtime per request robust towards changes in $k$.

Note that amortized means that an expensive operation, namely the cleanup procedure, is executed, but this does not happen very often. We can conclude the following proposition.

**Proposition 1**: LTrees processes a page request in $\mathcal{O}(\alpha(2m))$ amortized time, where $m$ is the page-set size.

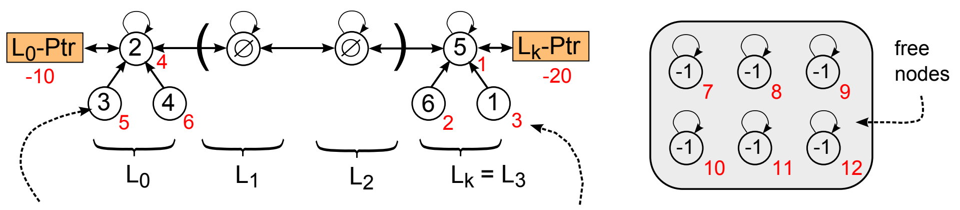
### 2.3.3 Space requirements for LTrees

LTrees requires $\theta(m)$ space since it consists of linked trees with a total number of $2m$ nodes. This becomes more evident in the next section where we describe how exactly the data structure is implemented in Java.

## 2.4 Java implementation of LTrees

To implement LTrees in a programming language such as Java we will use eight arrays, two variables and a queue.

**Example 1**: Let us consider an example where the cache size is $k = 3$, the page-set $M = \{1, 2, 3, 4, 5, 6\}$ and the page-set size $m = 6$. The layer partition shall be given by $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (2, 3, 4|\emptyset|\emptyset|5, 6, 1)$, see figure 22. Each node has a *page* value, a *parent* pointer, a *left* and *right* pointer, a *count* value, a *marked* value and a *rank* value. Each of these attributes is



The numbers within a node correspond to a page.

The red numbers next to a node serve as an identification number (node id).

address array

| | | | | | | |
|---|---|---|---|---|---|---|
| page value | 1 | 2 | 3 | 4 | 5 | 6 |
| node id | 3 | 4 | 5 | 6 | 1 | 2 |

The address array has size m=6.
If we are searching for a node with a certain page value, we can use the address array.
E.g. which node has page value 2?
Since address[2]=4 we know that page 2 is in node 4.

The following arrays of size 2m describe all 2m=12 nodes:

page value array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page value | 5 | 6 | 1 | 2 | 3 | 4 | -1 | -1 | -1 | -1 | -1 | -1 |

value[5]=3 means that node 5 has page value 3.

parent array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| parent node | 1 | 1 | 1 | 4 | 4 | 4 | -1 | -1 | -1 | -1 | -1 | -1 |

parent[3]=1 means that node 3 has node 1 as parent .

left array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| left node | 4 | -1 | -1 | -10 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

left[1]=4 means that node 1 has node 4 as left neighbor.
left[4]=-10 means that $L_0$-Ptr is adjacent to the left of node 4.
left[2]=-1 means that node 2 has no left neighbor.

right array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| right node | -20 | -1 | -1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

right[1]=-20 means that $L_k$-Ptr is adjacent to the right of node 1.

count array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count value | 2 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |

count[1]=2 means that two empty sets are adjacent to the left of node 1.
count[4]=0 means no empty sets are to the left of node 4.

marked array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| marked | 0 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |

marked[6]=0 means that node 6 is not marked. An array value of 1 would indicate a marked node.

rank array

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| rank value | 1 | 0 | 0 | 1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |

rank[4]=1 means that node 4 has a rank value of 1 (the rank is essentially the height).

There are in total 2m=12 nodes.
6 of these nodes are currently used while the other 6 are still free.

free nodes have an array value of -1

The pointers $L_0$-Ptr and $L_k$-Ptr are implemented by two variables L0-Ptr and Lk-Ptr:
L0-Ptr=4 means that $L_0$-Ptr points to node 4.
Lk-Ptr=1 means that Lk-Ptr points to node 1.

A queue Q maintains the id of free nodes:
Q = [ 7, 8, 9, 10, 11, 12 ]

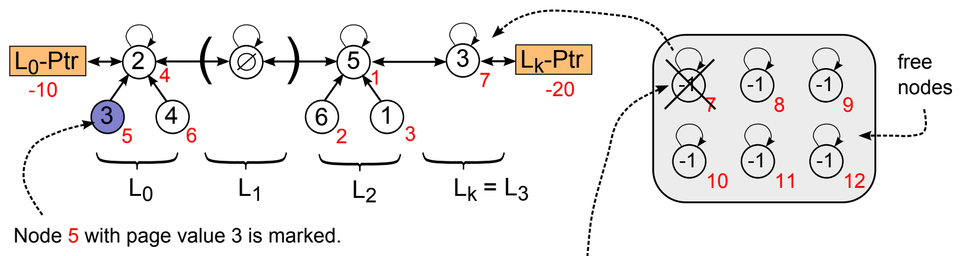
Figure 22: Implementation of LTrees in Java using arrays, two variables and a queue.

represented by an array of size $2m$ since there are in total $2m$ nodes. Two variables are used for the pointers $\mathcal{L}_0$-Ptr and $\mathcal{L}_k$-Ptr. A queue maintains which of the $2m$ nodes are still free.

Let us request page 3 to see how the arrays and queue are updated. The layer partition becomes $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (2,4|\emptyset|5,6,1|3)$, see figure 23.

page 3 is requested:

A free node is used to insert a new node with page value 3.



| address array | page value | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | node id | 3 | 4 | 7 | 6 | 1 | 2 |

page value 3 is now in the newly inserted node 7: address[3]=7

The queue tells us that node 7 was free:
Q = [ ~~7~~, 8, 9, 10, 11, 12 ]
A dequeue yields:
Q = [ 8, 9, 10, 11, 12 ]

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page value | 5 | 6 | 1 | 2 | 3 | 4 | 3 | -1 | -1 | -1 | -1 | -1 |
| parent | 1 | 1 | 1 | 4 | 4 | 4 | 7 | -1 | -1 | -1 | -1 | -1 |
| left | 4 | -1 | -1 | -10 | -1 | -1 | 1 | -1 | -1 | -1 | -1 | -1 |
| right | 7 | -1 | -1 | 1 | -1 | -1 | -20 | -1 | -1 | -1 | -1 | -1 |
| count | 1 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | -1 | -1 | -1 |
| marked | 0 | 0 | 0 | 0 | 1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 |
| rank | 1 | 0 | 0 | 1 | 0 | 0 | 0 | -1 | -1 | -1 | -1 | -1 |

The variable Lk-Ptr is updated. Lk-Ptr now points to node 7.

L0-Ptr = 4
Lk-Ptr = 7

We mark node 5 by setting marked[5]=1.

We set right[1]=7 and count[1] = count[1]-1 = 1.

Column 7 describes the newly inserted node 7.

Here, we can see why we need the address array. E.g. how did we know that upon request to page 3 we had to mark node 5?
Well, address[3] returned node id 5 in the previous address array.

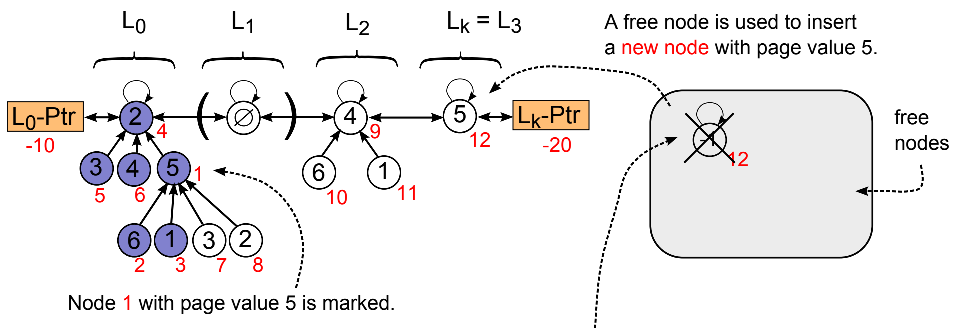
We also recognize that a column in the table represents a node, e.g. column 7 describes node 7.

Figure 23: Implementation of LTrees in Java. The arrays, pointer variables and queue are updated upon request to page 3.

**Example 2**: Let us have a further look at the mechanics of LTrees, in particular at the cleanup procedure. Suppose the layer partition is given by $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (3,2|\emptyset|4,6,1|5)$ which we get after the request sequence

$\sigma = (5, 6, 1, 3, 2, 4, 6, 1, 5)$. In figure 24 we see how LTrees looks like after the last request (page 5). Note that the queue becomes empty indicating that no free nodes are available.

page 5 is requested:



Node 1 with page value 5 is marked.

address array

| | | | | | | |
|---|---|---|---|---|---|---|
| page value | 1 | 2 | 3 | 4 | 5 | 6 |
| node id | 11 | 8 | 7 | 9 | 12 | 10 |

The queue tells us that node 12 was free:
Q = [ $\cancel{12}$ ]
A dequeue yields an empty queue:
Q = [ ]

page value 5 is now in the newly inserted node 12:
address[5]=12

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page value | 5 | 6 | 1 | 2 | 3 | 4 | 3 | 2 | 4 | 6 | 1 | 5 |
| parent | 4 | 1 | 1 | 4 | 4 | 4 | 1 | 1 | 9 | 9 | 9 | 12 |
| left | -1 | -1 | -1 | -10 | -1 | -1 | -1 | -1 | 4 | -1 | -1 | 9 |
| right | -1 | -1 | -1 | 9 | -1 | -1 | -1 | -1 | 12 | -1 | -1 | -20 |
| count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| marked | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| rank | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

The variable Lk-Ptr is updated:

L0-Ptr = 4
Lk-Ptr = 12

We mark node 1 by setting marked[1]=1.

Column 12 describes the newly inserted node 12.

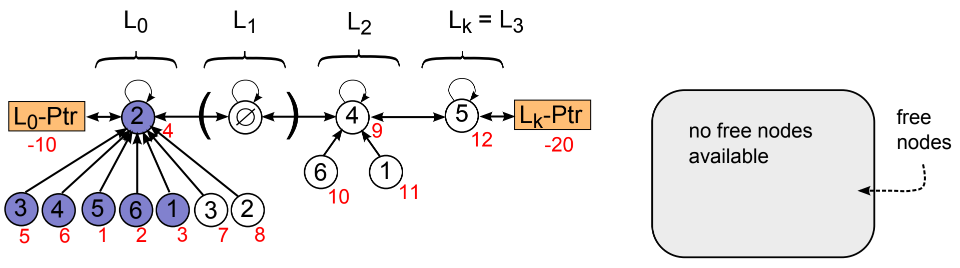
There are no free nodes which becomes apparent if we look at the empty queue.
6 nodes are unmarked and the other 6 nodes are marked. The problem is whenever we request pages from $L_0$ and $L_i$ with 0<i<k, we need a free node from the queue.
In order to have free nodes again we will have to call the cleanup procedure which removes the 6 marked nodes and puts them into the queue again.

Figure 24: Implementation of LTrees in Java. The arrays, pointer variables and queue are updated upon request to page 5. Note that the queue becomes empty.

**Example 3**: We have just seen that the queue is empty which means that all $2m = 12$ nodes are used in the linked trees data structure. The problem is whenever we request pages from $\mathcal{L}_0$ and $\mathcal{L}_i$ with $0 < i < k$, we need a free node from the queue. In order to have free nodes again we will have to call the cleanup procedure which removes the $m = 6$ marked nodes and puts them into the queue again. In step 1 of the cleanup procedure we apply a find operation *with path compression* to all $2m = 12$ nodes which creates trees with height of at most 1, see figure 25.

Cleanup procedure:

Step 1: Apply a find operation with path compression to all 2m=12 nodes.
This creates trees with height of at most 1.



address array

| page value | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| node id | 11 | 8 | 7 | 9 | 12 | 10 |

The queue is empty:
Q = [ ]

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page value | 5 | 6 | 1 | 2 | 3 | 4 | 3 | 2 | 4 | 6 | 1 | 5 |
| parent | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 9 | 9 | 9 | 12 |
| left | -1 | -1 | -1 | -10 | -1 | -1 | -1 | -1 | 4 | -1 | -1 | 9 |
| right | -1 | -1 | -1 | 9 | -1 | -1 | -1 | -1 | 12 | -1 | -1 | -20 |
| count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| marked | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| rank | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

The variables L0-Ptr and Lk-Ptr:

L0-Ptr = 4
Lk-Ptr = 12

The rank value of nodes 1 and 4 are updated due to path compression:
rank[1] = 0
rank[4] = 1

The parent array shows that nodes 2,3,7,8 are now directly linked to node 4 due to path compression.

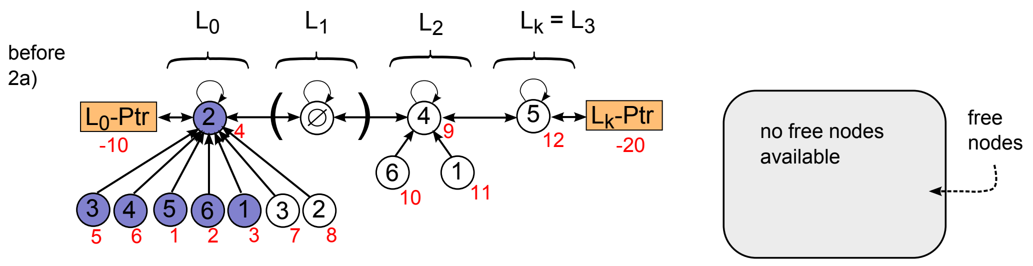
Figure 25: Step 1 of the cleanup procedure. A find operation with path compression is applied to all $2m$ nodes. This creates trees with height of at most 1. The parent and rank array are updated. Note that the rank is only updated during the cleanup procedure, but usually it is not changed as mentioned in [21].

Let us examine step 2 of the cleanup procedure where we remove marked nodes. In 2a) this is done for non-root nodes. The queue is filled with the id of these nodes, see figure 26.

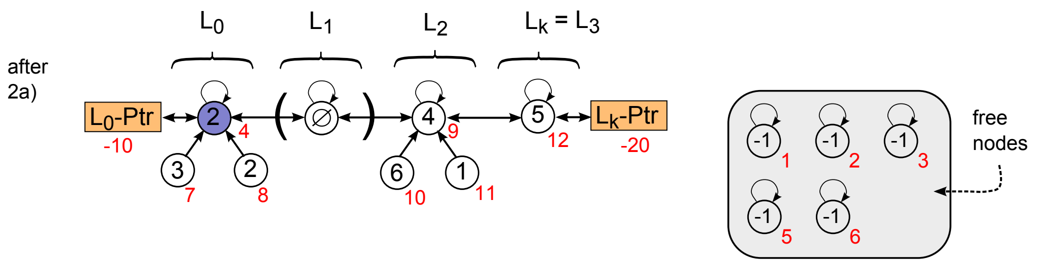
In 2b) marked root nodes are replaced by one of their children, see figure 27. After this step all previously marked nodes have been removed and free nodes are available again in the queue. Requesting pages is possible again.

Cleanup procedure:

Step 2: Remove marked nodes.
For all nodes u:
    Case 1: If u is a root node, do nothing.
    Case 2: If u is a child:
        2a) if u is marked as deleted, then remove the node
        (and put it in the queue).



We remove nodes 5,6,1,2,3 and put them in the queue according to case 2a):



address array

| page value | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| node id | 11 | 8 | 7 | 9 | 12 | 10 |

The queue is filled with node id's:
Q = [ 1,2,3,5,6 ]

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page value | -1 | -1 | -1 | 2 | -1 | -1 | 3 | 2 | 4 | 6 | 1 | 5 |
| parent | -1 | -1 | -1 | 4 | -1 | -1 | 4 | 4 | 9 | 9 | 9 | 12 |
| left | -1 | -1 | -1 | -10 | -1 | -1 | -1 | -1 | 4 | -1 | -1 | 9 |
| right | -1 | -1 | -1 | 9 | -1 | -1 | -1 | -1 | 12 | -1 | -1 | -20 |
| count | -1 | -1 | -1 | 0 | -1 | -1 | 0 | 0 | 1 | 0 | 0 | 0 |
| marked | -1 | -1 | -1 | 1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 |
| rank | -1 | -1 | -1 | 1 | -1 | -1 | 0 | 0 | 1 | 0 | 0 | 0 |

The variables L0-Ptr and Lk-Ptr:

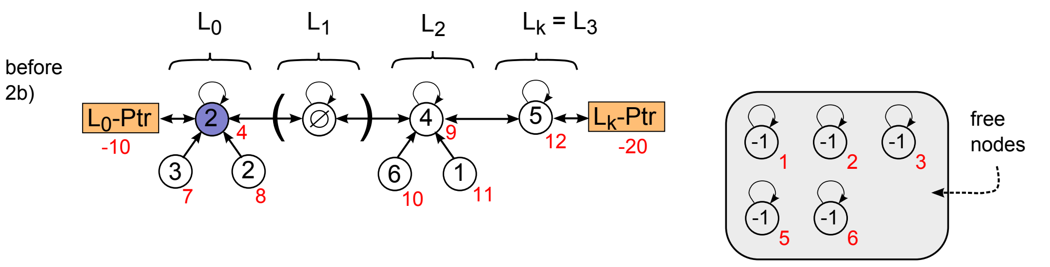
L0-Ptr = 4
Lk-Ptr = 12

Nodes 1,2,3,5,6 are removed from the linked trees and put into the queue. See columns 1,2,3,5,6 for which the array values have been set to -1.

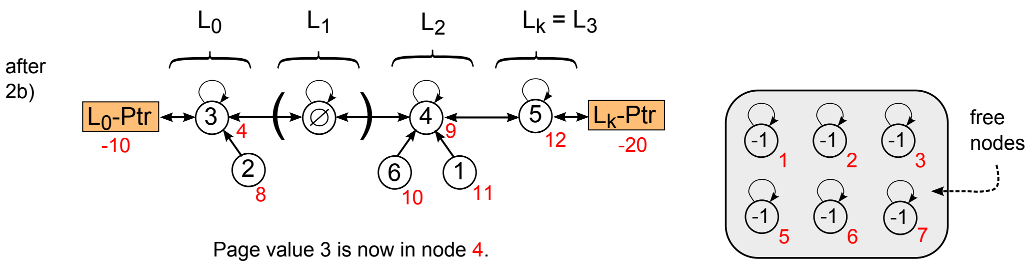
Figure 26: Step 2, case 2a) of the cleanup procedure. Marked nodes (non-root nodes) are removed and put into the queue.

Cleanup procedure:

```
Step 2: Remove marked nodes.
For all nodes u:
    Case 1: If u is a root node, do nothing.
    Case 2: If u is a child:
            2a) if u is marked as deleted, then remove the node
                (and put it in the queue).
            2b) if u is not marked as deleted:
                    if parent (root) of u is not marked as deleted, then do nothing.
                    if parent (root) of u is marked as deleted, then replace root by u.
```



Instead of replacing node 4 by its child node 7, it suffices to replace the page value:
- The page value of node 4 is replaced by the page value of node 7. Afterwards node 4 is unmarked.
- Then node 7 is removed and put into the queue.



Page value 3 is now in node 4.
address[3]=4

address array

| page value | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| node id | 11 | 8 | 4 | 9 | 12 | 10 |

Node id 7 is put into the queue:
Q = [ 1,2,3,5,6,7 ]

| node id | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page value | -1 | -1 | -1 | 3 | -1 | -1 | -1 | 2 | 4 | 6 | 1 | 5 |
| parent | -1 | -1 | -1 | 4 | -1 | -1 | -1 | 4 | 9 | 9 | 9 | 12 |
| left | -1 | -1 | -1 | -10 | -1 | -1 | -1 | -1 | 4 | -1 | -1 | 9 |
| right | -1 | -1 | -1 | 9 | -1 | -1 | -1 | -1 | 12 | -1 | -1 | -20 |
| count | -1 | -1 | -1 | 0 | -1 | -1 | -1 | 0 | 1 | 0 | 0 | 0 |
| marked | -1 | -1 | -1 | 0 | -1 | -1 | -1 | 0 | 0 | 0 | 0 | 0 |
| rank | -1 | -1 | -1 | 1 | -1 | -1 | -1 | 0 | 1 | 0 | 0 | 0 |

The variables L0-Ptr and Lk-Ptr:

L0-Ptr = 4
Lk-Ptr = 12

Node 7 is removed from the linked trees and put into the queue. See column 7 for which the array values have been set to -1.

Figure 27: Step 2, case 2b) of the cleanup procedure. Marked root nodes are replaced by their children.

## 2.5 Timestamp implementation of the layer partition

Another possibility to implement the layer partition and therefore OPT is to use a timestamp based data structure which we will refer to as *TStamp*. It has been devised by Moruz et al. [12]. Each layer $\mathcal{L}_i$ is represented by a tuple $(t, v)$ where $t$ is the time of the layer creation. $t$ remains unchanged until $\mathcal{L}_i$ is merged with $\mathcal{L}_0$. The variable $v$ is defined as $v := 1 + e$, where $e$ is the number of empty layers adjacent to the left (this is similar to the count variable in LTrees). The layer $\mathcal{L}_0$ is not stored by TStamp.

Moreover, TStamp stores information about when each page has last been requested.

**Example 1**: Suppose we have the request sequence $\sigma = (5, 6, 1, 3, 2, 4, 5)$, a cache size of $k = 3$ and the page-set $M = \{1, 2, 3, 4, 5, 6\}$. The pages in $\sigma$ are requested at time $t$, see table 2.

| pages in $\sigma$ | 5 | 6 | 1 | 3 | 2 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| time $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Table 2: The pages in $\sigma$ are requested at time $t$.

**Request first three pages**: The layer partition after the first $k = 3$ pairwise distinct requested pages is given in figure 28. Notice that we have added a line that states for each layer when it was created (except for the empty layers and $\mathcal{L}_0$ which are ignored), e.g. $\mathcal{L}_3$ was created when page 5 was requested at time $t = 1$.

By layer creation we mean case $p \in \mathcal{L}_0$ of the update rules for the layer partition, since $\mathcal{L}_k$ will only contain page $p$ such that $\mathcal{L}_k = \{p\}$ can be considered as newly created layer. The other case is the first request in $\sigma$.

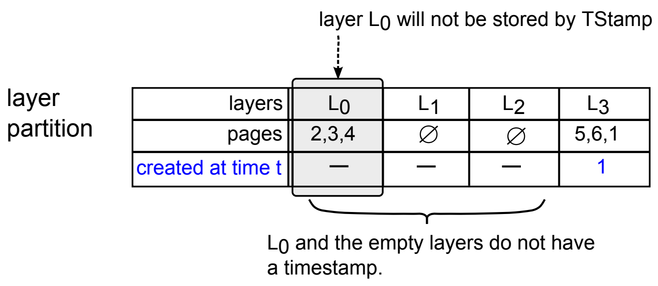
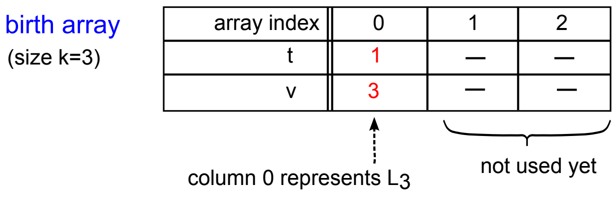

layer L0 will not be stored by TStamp
layer partition
layers
L0
L1
L2
L3
pages
2,3,4
∅
∅
5,6,1
created at time t
—
—
—
1
L0 and the empty layers do not have a timestamp.

Figure 28: The layer partition after the first $k = 3$ pairwise distinct requested pages. The third row shows the time of the layer creation.

How does TStamp store the layer partition? TStamp encodes it in an array *birth* with tuples $(t, v)$, see figure 29. The name birth indicates that $t$ describes the time when a layer was created. Its size is $k$ because at most $k$ layers, namely $\mathcal{L}_1$ to $\mathcal{L}_k$, are stored ($\mathcal{L}_0$ is not stored).

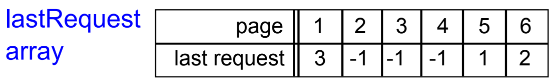

birth array
(size k=3)
column 0 represents L3
not used yet

| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 1 | — | — |
| v | 3 | — | — |

Figure 29: TStamp encodes the layer partition in an array called birth. Its entries are tuples $(t, v)$.

Layer $\mathcal{L}_3$ is represented by column 0 which contains the tuple $(t, v) = (1, 3)$. This means that $\mathcal{L}_3$ has been created at time $t = 1$, and its value for $v := e+1$ equals 3 since $e = 2$ empty layers are adjacent to the left.

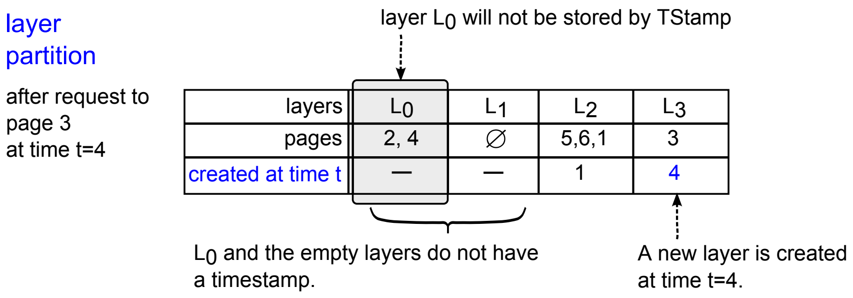
Moreover, TStamp keeps an array *lastRequest* that assigns to each page the time of its last request, see figure 30.


lastRequest
array

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 3 | -1 | -1 | -1 | 1 | 2 |

Figure 30: The array lastRequest after the first 3 requests. For each page the time of its last request is stored. The value -1 indicates that a page has not been requested yet.

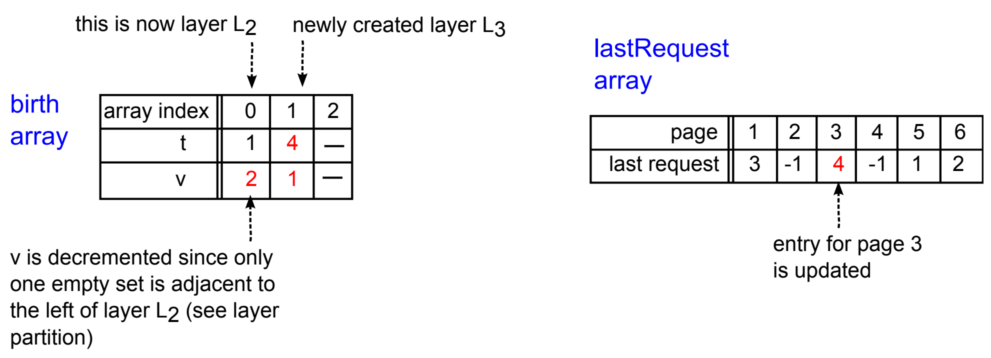
**Request(3)**: Let us process the next request which is page $3 \in \mathcal{L}_0$ at time $t = 4$. The updated layer partition is shown in figure 31.


layer
partition
after request to
page 3
at time t=4
layer L0 will not be stored by TStamp
L0 and the empty layers do not have
a timestamp.
A new layer is created
at time t=4.

| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | 2, 4 | ∅ | 5,6,1 | 3 |
| created at time t | — | — | 1 | 4 |

Figure 31: The layer partition after request to page 3 at time $t = 4$.

A new layer is created at time $t = 4$ such that we have to update the birth array, see figure 32. The content of the birth array tells us that there are


this is now layer $L_2$
newly created layer $L_3$
birth array
lastRequest array
v is decremented since only one empty set is adjacent to the left of layer $L_2$ (see layer partition)
entry for page 3 is updated

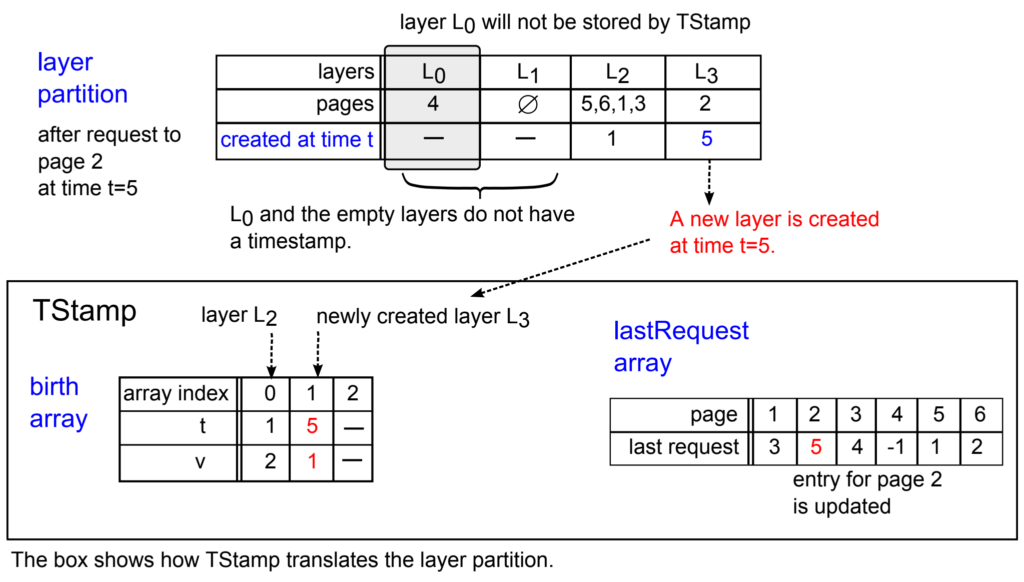
| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 1 | 4 | — |
| v | 2 | 1 | — |

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 3 | -1 | 4 | -1 | 1 | 2 |

Figure 32: TStamp uses the two arrays birth and lastRequest. birth is updated because a new layer is created at time $t = 4$.

two non-empty layers. The first one with $(t, v) = (1, 2)$ has been created at time $t = 1$ and has $e = v - 1 = 1$ empty set adjacent to the left. The second one with $(t, v) = (4, 1)$ has been created at time $t = 4$ and has $e = v - 1 = 0$ empty layers adjacent to the left. We recognize that this is indeed the case for the layer partition in figure 31.

**Request(2)**: We process the next request which is page $2 \in \mathcal{L}_0$ at time $t = 5$. The updated layer partition is shown in figure 33.


layer $L_0$ will not be stored by TStamp
layer partition
after request to page 2 at time t=5
$L_0$ and the empty layers do not have a timestamp.
A new layer is created at time t=5.
TStamp
layer $L_2$
newly created layer $L_3$
birth array
lastRequest array
entry for page 2 is updated
The box shows how TStamp translates the layer partition.

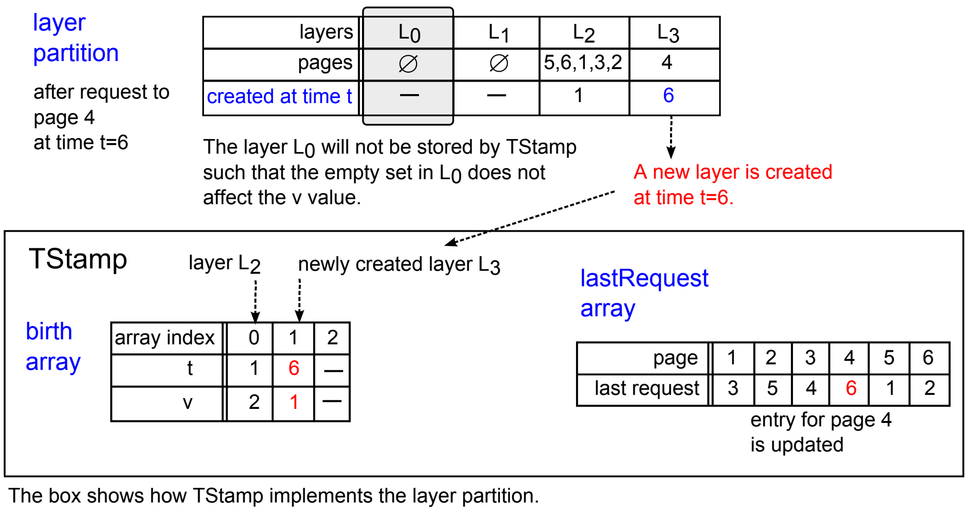
| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | 4 | ∅ | 5,6,1,3 | 2 |
| created at time t | — | — | 1 | 5 |

| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 1 | 5 | — |
| v | 2 | 1 | — |

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 3 | 5 | 4 | -1 | 1 | 2 |

Figure 33: The layer partition, birth array and lastRequest array upon request to page 2 at time $t = 5$. We recognize that the birth array encodes the structure of the layer partition.

**Request(4):** We request page $4 \in \mathcal{L}_0$ at time $t = 6$, see figure 34.


layer partition
after request to page 4 at time t=6
The layer L0 will not be stored by TStamp such that the empty set in L0 does not affect the v value.
A new layer is created at time t=6.
TStamp
layer L2
newly created layer L3
birth array
lastRequest array
entry for page 4 is updated
The box shows how TStamp implements the layer partition.

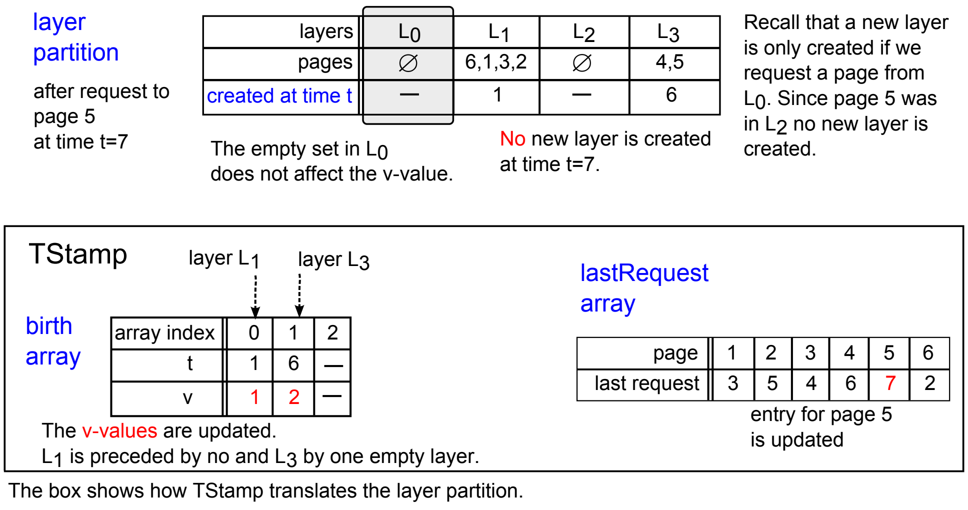
| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | $\varnothing$ | $\varnothing$ | 5,6,1,3,2 | 4 |
| created at time t | — | — | 1 | 6 |

| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 1 | 6 | — |
| v | 2 | 1 | — |

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 3 | 5 | 4 | 6 | 1 | 2 |

Figure 34: The layer partition and TStamp at time $t = 6$.

**Request(5):** We request page $5 \in \mathcal{L}_2$ at time $t = 7$, see figure 35.


layer partition
after request to page 5 at time t=7
The empty set in L0 does not affect the v-value.
No new layer is created at time t=7.
Recall that a new layer is only created if we request a page from L0. Since page 5 was in L2 no new layer is created.
TStamp
layer L1
layer L3
birth array
lastRequest array
The v-values are updated.
L1 is preceded by no and L3 by one empty layer.
entry for page 5 is updated
The box shows how TStamp translates the layer partition.

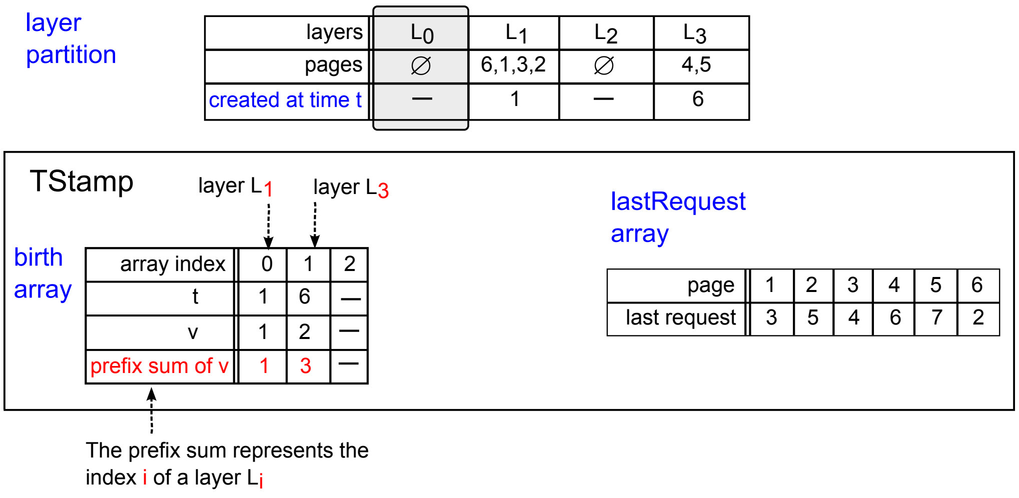
| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | $\varnothing$ | 6,1,3,2 | $\varnothing$ | 4,5 |
| created at time t | — | 1 | — | 6 |

| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 1 | 6 | — |
| v | 1 | 2 | — |

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 3 | 5 | 4 | 6 | 7 | 2 |

Figure 35: The layer partition and TStamp at time $t = 7$.

Until now we have not mentioned how we can fully reconstruct the layer partition from the birth and lastRequest array, in particular how the index $i$ is determined for a page $p \in \mathcal{L}_i$. For this purpose we will introduce a *prefix sum* of $v$, see figure 36. We recognize that the prefix sum represents the index $i$ for a layer $\mathcal{L}_i$! As an example we will determine in which layer page 3



| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | ∅ | 6,1,3,2 | ∅ | 4,5 |
| created at time t | — | 1 | — | 6 |

| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 1 | 6 | — |
| v | 1 | 2 | — |
| prefix sum of v | 1 | 3 | — |

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 3 | 5 | 4 | 6 | 7 | 2 |

Figure 36: The birth array is extended by a prefix sum of $v$. The prefix sum corresponds to the index $i$ of a layer $\mathcal{L}_i$.

is. In figure 36 we will walk along the birth array from right to left and ask some questions:

We are at array index 1 of the birth array: can page 3 be in $\mathcal{L}_3$? No, it can't since page 3 has last been requested at time $t = 4$ but $\mathcal{L}_3$ has been created at time $t = 6$. A page should have been requested at time $t \geq 6$ to be in $\mathcal{L}_3$ which is not the case for page 3.

We notice that the $t$ values in the birth array decrease if we go from right to left, i.e. for two layers $\mathcal{L}_i$ and $\mathcal{L}_j$ with $i < j$ created at time $\mathcal{L}_i.t$ and $\mathcal{L}_j.t$ respectively we have $\mathcal{L}_i.t < \mathcal{L}_j.t$. For example, we can see that $\mathcal{L}_1.t < \mathcal{L}_3.t$ with $\mathcal{L}_1.t = 1$ and $\mathcal{L}_3.t = 6$.

We move one position to the left in the birth array, i.e. to array index 0, and ask: can page 3 be in $\mathcal{L}_1$? Yes, it can since page 3 has last been requested at time $t = 4$ and layer $\mathcal{L}_1$ has been created at time $t = 1$. The idea is that for a page $p$ to be in layer $\mathcal{L}_i$ the following must hold[1]:

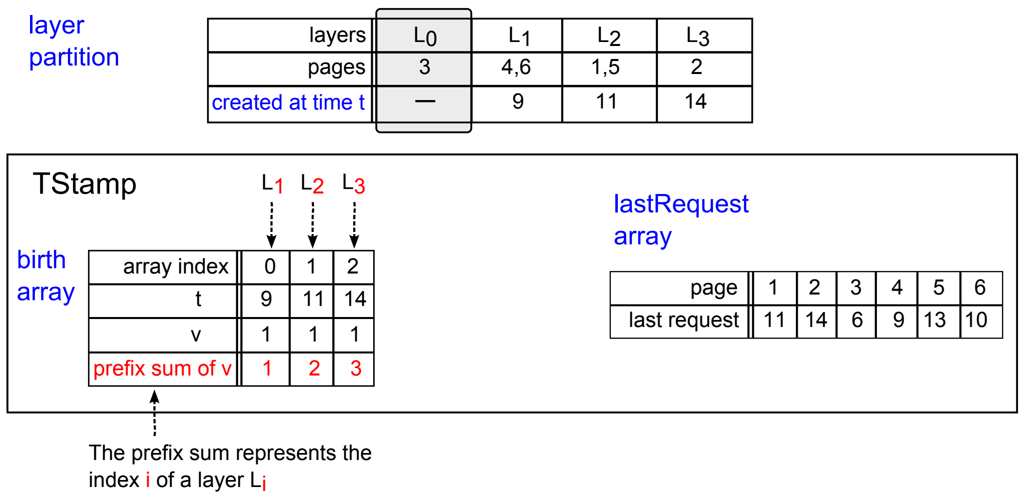
$$\text{lastRequest}[p] \geq \mathcal{L}_i.t$$

To determine $\mathcal{L}_i$ we move along the birth array from right to left, and as soon as the condition above is fulfilled we have found $\mathcal{L}_i$. Let us illustrate this for page 6. Once again we move from right to left in the birth array:

- Is $6 \in \mathcal{L}_3$? No, since $\text{lastRequest}[6] < \mathcal{L}_3.t$.
- Is $6 \in \mathcal{L}_1$? Yes, since $\text{lastRequest}[6] \geq \mathcal{L}_1.t$.

[1]The concept is similar to the *table of contents* in a book. We can easily determine in which chapter we are by looking at the page numbers of the chapter headings.

**Example 2:** Suppose the cache size is $k = 3$, the page-set $M = \{1, 2, 3, 4, 5, 6\}$ and the layer partition is given by $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (3|4, 6|1, 5|2)$, see figure 37.



| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | 3 | 4,6 | 1,5 | 2 |
| created at time t | — | 9 | 11 | 14 |

| array index | 0 | 1 | 2 |
|---|---|---|---|
| t | 9 | 11 | 14 |
| v | 1 | 1 | 1 |
| prefix sum of v | 1 | 2 | 3 |

| page | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| last request | 11 | 14 | 6 | 9 | 13 | 10 |

Figure 37: TStamp implementation of the layer partition in example 2.

We will determine in which layer page 3 is by looking only at birth and lastRequest. The prefix sum tells us the layer index. Once again we move from right to left in the birth array:

- Is $3 \in \mathcal{L}_3$? No, since lastRequest[3]$< \mathcal{L}_3.t$.
- Is $3 \in \mathcal{L}_2$? No, since lastRequest[3]$< \mathcal{L}_2.t$.
- Is $3 \in \mathcal{L}_1$? No, since lastRequest[3]$< \mathcal{L}_1.t$.

We cannot move further to the left in birth which means that $3 \in \mathcal{L}_0$.

Let us determine in which layer page 2 is. Moving from right to left in the birth array we ask:

- Is $2 \in \mathcal{L}_3$? Yes, since lastRequest[2]$\geq \mathcal{L}_3.t$.

Note that instead of moving from right to left in the birth array we can also use binary search.

### Update rules upon a page request

Recall that upon a request to page $p$ the layer partition is modified according to three update rules (see again section 2.1), and in the previous examples we have seen how the cases $p \in \mathcal{L}_0$ and $p \in \mathcal{L}_i$ with $0 < i < k$ are handled by TStamp. We want to formalize how TStamp translates the update rules:

- Case (i): If $p \in \mathcal{L}_0$, then $\mathcal{L}_k$ is merged into $\mathcal{L}_{k-1}$. Afterwards, $p$ is removed from $\mathcal{L}_0$ and inserted as new layer $\mathcal{L}'_k$ such that $\mathcal{L}'_k = \{p\}$.

Merging $\mathcal{L}_k$ with $\mathcal{L}_{k-1}$ is done as follows: if $\mathcal{L}_{k-1}$ is empty, then the $v$ value for $\mathcal{L}_k$ is decremented since the number of empty layers to the left of $\mathcal{L}_k$ decreases by one. Then we append a new column in the birth array with a tuple $(t, v)$, where $t$ is the current timestamp and $v = 1$. This column represents the new layer $\mathcal{L}'_k$ (see figure 32).

Otherwise, if $\mathcal{L}_{k-1}$ is not empty, then we simply overwrite the $t$ value for $\mathcal{L}_k$ by the current timestamp[2] (see figure 33).

Finally, we set lastRequest[$p$] to the current timestamp such that the new last layer $\mathcal{L}'_k$ contains only $p$.

- Case (ii) If $p \in \mathcal{L}_k$, then the layer partition remains the same. Therefore, TStamp does not need to be updated.

- Case (iii): If $p \in \mathcal{L}_i$ with $0 < i < k$, then $p$ is removed from $\mathcal{L}_i$ and added to $\mathcal{L}_k$. Afterwards $\mathcal{L}_i \backslash \{p\}$ is merged into $\mathcal{L}_{i-1}$. The layers $\mathcal{L}_j$ with $j = i+1, i+2, \ldots, k-1$ all move one position to the left. An empty set is inserted as new layer $\mathcal{L}'_{k-1}$ such that $\mathcal{L}'_{k-1} = \emptyset$.

  In order to move $p$ from $\mathcal{L}_i$ to $\mathcal{L}_k$ we set lastRequest[$p$] to the current timestamp.

  Merging $\mathcal{L}_i \backslash \{p\}$ with $\mathcal{L}_{i-1}$ and shifting layers $\mathcal{L}_j$ with $j = i+1, i+2, \ldots, k-1$ one position to the left is done as follows: if $\mathcal{L}_{i-1}$ is empty, then the $v$ value for $\mathcal{L}_i$ is decremented (see figure 35).

  Otherwise, if $\mathcal{L}_{i-1}$ is not empty, then we delete the column for $\mathcal{L}_i$ in the birth array and move all the following columns one position to the left (see figure 38).

  Finally, to insert an empty set as new layer $\mathcal{L}'_{k-1}$ we simply increment the $v$ value for $\mathcal{L}_k$.

### 2.5.1 Runtime analysis for TStamp

Cases (i) and (ii) are handled by TStamp in $\mathcal{O}(1)$ time. In case (iii) however, processing a request to page $p \in \mathcal{L}_i$ with $0 < i < k$ can be slow for TStamp as shown in figure 38: an update of the layer partition involves merging two layers. TStamp implements this union of two layers by deleting an entry in the birth array. In the worst case this is the first entry such that all $k-1$ following entries have to be shifted one position to the left which requires $\mathcal{O}(k)$ time.

[2] If you think of the *table of contents analogy*, then this corresponds to putting the last chapter heading to the very end of a book.

**Remark (R1)**: The worst-case time is $\mathcal{O}(k)$, but we should consider the following: although the birth array has size $k$ deleting the first column can still be fast. The reason is that the birth array is not necessarily filled with $k$ columns. For example, in figure 36 you can see that the last column of the birth array is not occupied since out of the three layers $\mathcal{L}_1, \mathcal{L}_2$ and $\mathcal{L}_3$ only the two non-empty layers $\mathcal{L}_1$ and $\mathcal{L}_3$ are stored as columns.

Or consider this example: if $k = 2000$, then there may only be 10 non-empty layers such that only 10 columns are occupied in the birth array. In that case deleting the first column implies shifting only 9 columns instead of $k - 1 = 1999$ columns.

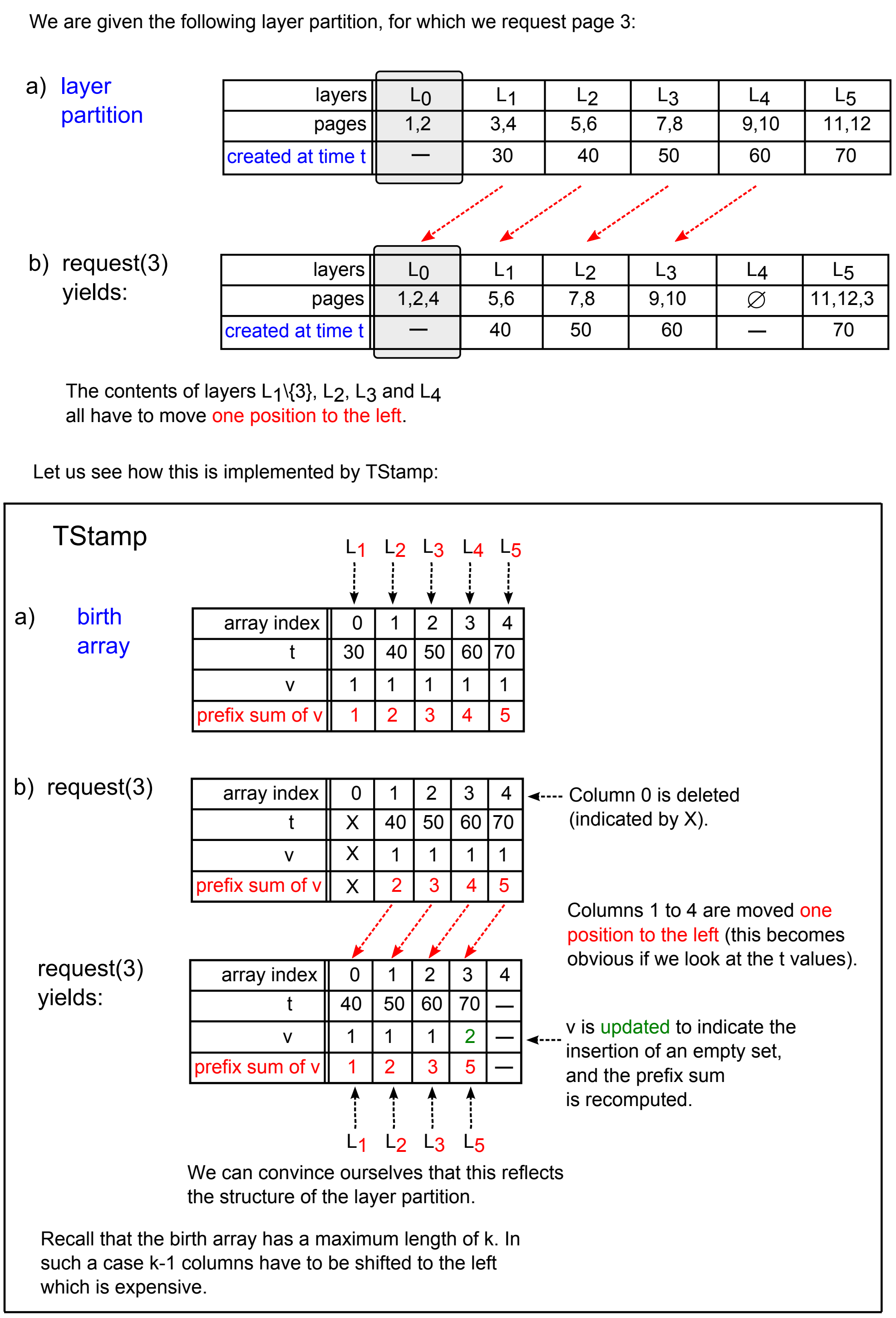
If $l$ is the number of non-empty layers, then $l - 1$ columns have to be shifted to the left, and processing a page request then requires $\mathcal{O}(l)$ time. Moruz et al. [12] examined how TStamp processes some real-world traces. They showed empirically that $l$ is a small number, i.e. there are significantly less than $k$ non-empty layers.

Moreover, a page that is requested from $\mathcal{L}_k$ is processed by TStamp in $\mathcal{O}(1)$ time, and for the traces examined by Moruz et al. 99% of the requested pages are from $\mathcal{L}_k$.

**Proposition 2**: If $p \in \mathcal{L}_0$ or $p \in \mathcal{L}_k$, then TStamp processes a request to page $p$ in $\mathcal{O}(1)$ time. If $p \in \mathcal{L}_i$ with $0 < i < k$, then TStamp requires $\mathcal{O}(l)$ time with $l$ being the number of non-empty layers. The worst-case time is $\mathcal{O}(k)$.

### 2.5.2 Space requirements for TStamp

TStamp stores the arrays birth and lastRequest. The birth array has size $k$ and the lastRequest array size $m$. Since $m \geq k$ the total space required by TStamp is $\theta(m)$.



We are given the following layer partition, for which we request page 3:

a) layer partition

| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ | $L_4$ | $L_5$ |
|---|---|---|---|---|---|---|
| pages | 1,2 | 3,4 | 5,6 | 7,8 | 9,10 | 11,12 |
| created at time t | — | 30 | 40 | 50 | 60 | 70 |

b) request(3) yields:

| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ | $L_4$ | $L_5$ |
|---|---|---|---|---|---|---|
| pages | 1,2,4 | 5,6 | 7,8 | 9,10 | ∅ | 11,12,3 |
| created at time t | — | 40 | 50 | 60 | — | 70 |

The contents of layers $L_1\setminus\{3\}$, $L_2$, $L_3$ and $L_4$ all have to move one position to the left.

Let us see how this is implemented by TStamp:

TStamp

a) birth array

| array index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| t | 30 | 40 | 50 | 60 | 70 |
| v | 1 | 1 | 1 | 1 | 1 |
| prefix sum of v | 1 | 2 | 3 | 4 | 5 |

b) request(3)

| array index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| t | X | 40 | 50 | 60 | 70 |
| v | X | 1 | 1 | 1 | 1 |
| prefix sum of v | X | 2 | 3 | 4 | 5 |

Column 0 is deleted (indicated by X).

Columns 1 to 4 are moved one position to the left (this becomes obvious if we look at the t values).

request(3) yields:

| array index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| t | 40 | 50 | 60 | 70 | — |
| v | 1 | 1 | 1 | 2 | — |
| prefix sum of v | 1 | 2 | 3 | 5 | — |

v is updated to indicate the insertion of an empty set, and the prefix sum is recomputed.

We can convince ourselves that this reflects the structure of the layer partition.

Recall that the birth array has a maximum length of k. In such a case k-1 columns have to be shifted to the left which is expensive.

Figure 38: Page $3 \in \mathcal{L}_1$ is requested. TStamp merges $\mathcal{L}_1\setminus\{3\}$ with $\mathcal{L}_0$ by deleting the first column in the birth array and shifting all the following columns one position to the left.

## 2.6 Comparison of LTrees and TStamp

We want to compare some features of LTrees and TStamp, where $k$ is the cache size and $m$ the page-set size, see table 3. It is not clear which of the data structures has the better runtime when processing a request sequence $\sigma$. For this reason we we will compare their runtimes experimentally for five real-world traces in the next section.

| | LTrees | TStamp |
|---|---|---|
| process request to page $p$ | $\mathcal{O}(1)$ for $p \in \mathcal{L}_k$, $\mathcal{O}(\alpha(2m))$ amortized for $p \in \mathcal{L}_0$ and $p \in \mathcal{L}_i$ with $0 < i < k$ | $\mathcal{O}(1)$ for $p \in \mathcal{L}_k$, $\mathcal{O}(1)$ for $p \in \mathcal{L}_0$, $\mathcal{O}(l)$ for $p \in \mathcal{L}_i$ with $0 < i < k$ and $l$ non-empty layers, $\mathcal{O}(k)$ worst-case |
| space | $\theta(m)$ | $\theta(m)$ |

Table 3: Comparison of the data structures LTrees and TStamp. $k$ is the cache size and $m$ the page-set size.

### 2.6.1 Borrowing a technique from TStamp

Before we present the experimental results we will modify LTrees a little by borrowing a technique from TStamp. Upon a request to page $p$ the case $p \in \mathcal{L}_k$ can be recognized by TStamp very quickly in $\mathcal{O}(1)$ time by comparing lastRequest[$p$] with the time $\mathcal{L}_k.t$ that states when $\mathcal{L}_k$ was created. If lastRequest[$p$]$\geq \mathcal{L}_k.t$, then $p$ is in layer $\mathcal{L}_k$.

For LTrees we will adopt the lastRequest array and introduce a variable $t_0$ that stores the time when $\mathcal{L}_k$ was created (which is also the time when we have last requested a page from $\mathcal{L}_0$). Similar to TStamp we check if lastRequest[$p$]$\geq t_0$ to recognize the case $p \in \mathcal{L}_k$.

Although LTrees processes the case $p \in \mathcal{L}_k$ in $\mathcal{O}(1)$ anyway since the tree for $\mathcal{L}_k$ has always height of at most 1, we found that this technique improves the runtime of LTrees in the experiments.

# 3 Experiments

## 3.1 Setup

In this section the runtimes of LTrees and TStamp are compared for the traces in table 4. The traces, which can be found in [24], represent long request sequences and have been obtained from real-world applications.

| trace name (request sequence $\sigma$) | number of requests (length of $\sigma$) | unique pages (page-set size $m$) |
|---:|---:|---:|
| OLTP | 914,145 | 186,880 |
| S1 | 3,995,316 | 1,309,698 |
| P6 | 12,672,123 | 773,770 |
| P1 | 32,055,473 | 2,311,485 |
| SPC1 like | 41,351,279 | 6,050,363 |

Table 4: Traces with their features.

Two setups have been used:

- Samsung notebook PC, Intel Atom processor @1.66 GHz, 1 GB RAM, Windows 7, Java SE7
- Lenovo desktop PC, Intel Core i3-2100 CPU @3.1 GHz, 4 GB RAM, Windows 7, Java SE6

The runtime for processing a trace was measured three times, and the median was chosen to compare LTrees with TStamp. Both data structures have been implemented in Java.

For LTrees we had to increase the Java heap space, i.e. the memory available to Java, since the traces have a large page-set size $m$, and LTrees uses several arrays of size $2m$. In the Eclipse IDE the Java heap space can be changed via the argument -Xmx[space]. For the Samsung notebook PC we chose -Xmx550M and for the Lenovo desktop PC -Xmx1024M.

Regarding the queue in LTrees the following was observed for the trace P6 and a cache size $k = 4096$ on the Samsung notebook PC: we have first used the queue that is offered by the Java API for which there are several implementations. We started with the LinkedList implementation which lead to a runtime of 29 seconds. After some research [25] we used the faster ArrayDeque implementation which processed the trace in 24 seconds. Note that the queue from the Java API is designed to store objects. However, the pages in the tracefile are represented by integers. So we switched to a queue for primitive data types [26] which improved the runtime to 20 seconds.

## 3.2 Results

The results are presented in a table and a corresponding plot. The table shows the runtimes when processing the trace file for different cache sizes $k$. Notice that $k$ is not determined by the trace file but can be set by us. For $k$ we have chosen powers of 2 as described in [24], i.e. $k \in \{1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288\}$. An exception is trace OLTP where we chose $k \in \{1000, 2000, 5000, 10000, 15000\}$. The purpose of the table is to demonstrate the behaviour of the data structure towards increasing cache sizes.

To compare LTrees with TStamp the ratios of their runtimes are displayed in a plot for different cache sizes. Since $k$ increases in powers of two the $x$-axis is shown in a logarithmic scale. A red line at height 1 of the $y$-axis marks points of equal runtimes making it easier to recognize where each data structure performs better. Points below this line indicate a better performance for TStamp, whereas points above this line mean a better performance for LTrees.

Table 14 lists for each trace the number of cache misses performed by OPT. These values are obtained by counting the number of requested pages from layer $\mathcal{L}_0$ (see again property (Pr1) in section 2.1) and are equal for both LTrees and TStamp since they both implement the layer partition. The values also include the first $k$ pairwise distinct requested pages which cause a cache miss since these pages have not been in the initially empty cache.

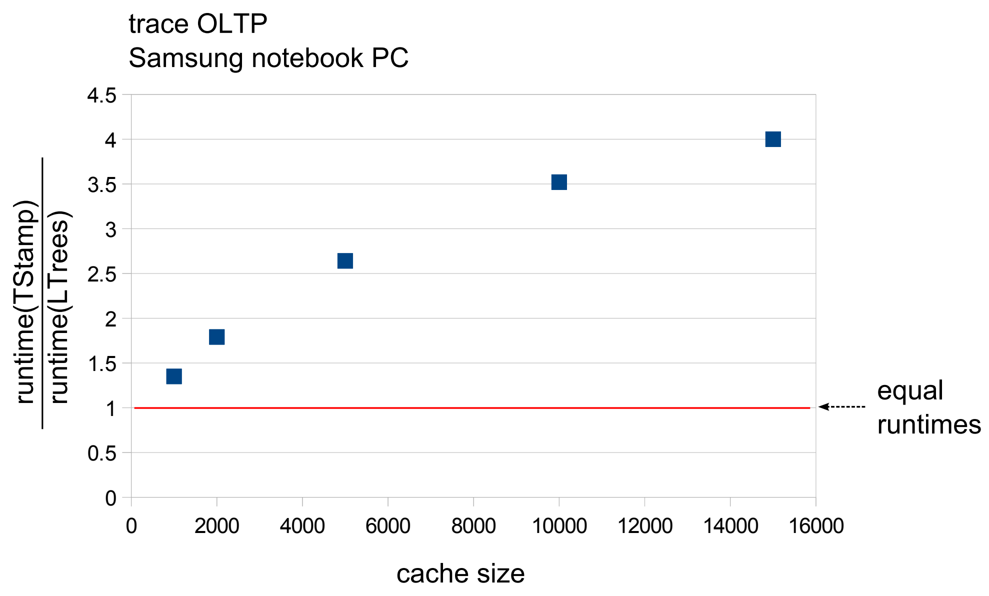


Figure 39: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace OLTP processed on Samsung notebook PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1000 | 1.560 | 2.106 | 1.35 |
| 2000 | 1.576 | 2.823 | 1.79 |
| 5000 | 1.607 | 4.244 | 2.64 |
| 10000 | 1.592 | 5.601 | 3.52 |
| 15000 | 1.544 | 6.178 | 4.00 |

Table 5: Results for the trace OLTP using the Samsung notebook PC. The trace was not processed on the Lenovo desktop PC because the runtimes were too small and thus inaccurate.

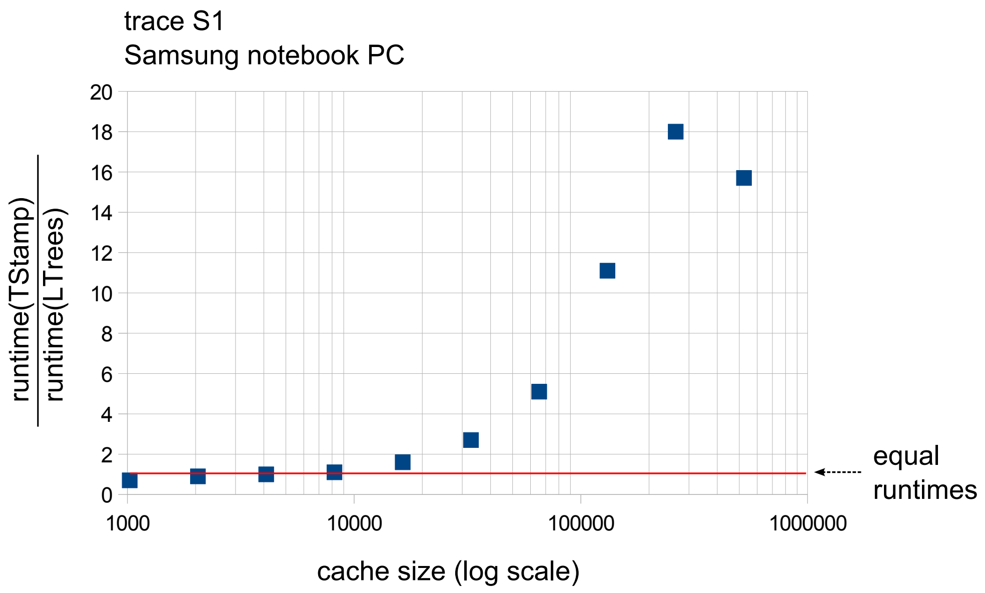


Figure 40: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace S1 processed on Samsung notebook PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 7 | 5 | 0.7 |
| 2048 | 7 | 6 | 0.9 |
| 4096 | 7 | 7 | 1.0 |
| 8192 | 7 | 8 | 1.1 |
| 16384 | 7 | 11 | 1.6 |
| 32768 | 7 | 19 | 2.7 |
| 65536 | 7 | 36 | 5.1 |
| 131072 | 7 | 78 | 11.1 |
| 262144 | 7 | 126 | 18.0 |
| 524288 | 7 | 110 | 15.7 |

Table 6: Results for the trace S1 using the Samsung notebook PC.

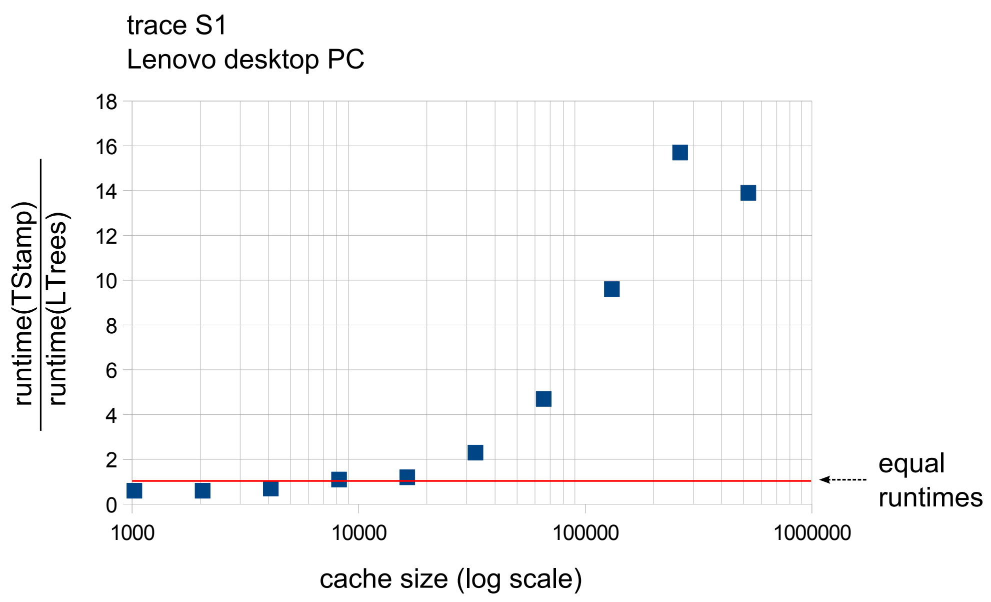


Figure 41: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace S1 processed on Lenovo desktop PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 1.092 | 0.640 | 0.59 |
| 2048 | 1.076 | 0.687 | 0.64 |
| 4096 | 1.092 | 0.764 | 0.70 |
| 8192 | 1.123 | 1.187 | 1.06 |
| 16384 | 1.139 | 1.405 | 1.23 |
| 32768 | 1.154 | 2.606 | 2.26 |
| 65536 | 1.124 | 5.227 | 4.65 |
| 131072 | 1.140 | 10.905 | 9.57 |
| 262144 | 1.124 | 17.661 | 15.71 |
| 524288 | 1.117 | 15.538 | 13.91 |

Table 7: Results for the trace S1 using the Lenovo desktop PC.

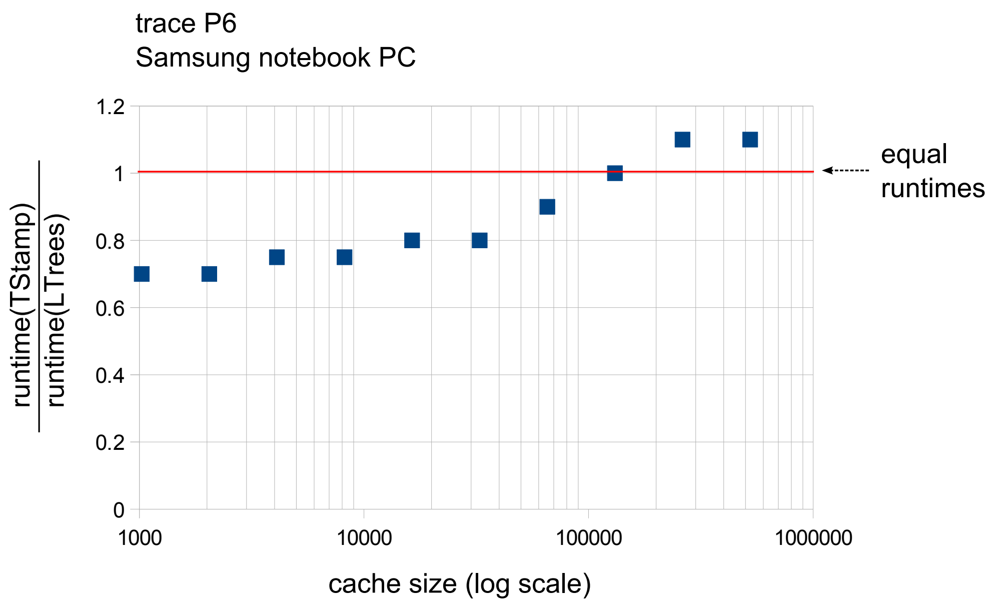


Figure 42: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace P6 processed on Samsung notebook PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 20 | 14 | 0.7 |
| 2048 | 20 | 14 | 0.7 |
| 4096 | 20 | 15 | 0.75 |
| 8192 | 20 | 15 | 0.75 |
| 16384 | 19 | 16 | 0.8 |
| 32768 | 19 | 16 | 0.8 |
| 65536 | 18 | 17 | 0.9 |
| 131072 | 17 | 17 | 1.0 |
| 262144 | 17 | 18 | 1.1 |
| 524288 | 16 | 17 | 1.1 |

Table 8: Results for the trace P6 using the Samsung notebook PC.

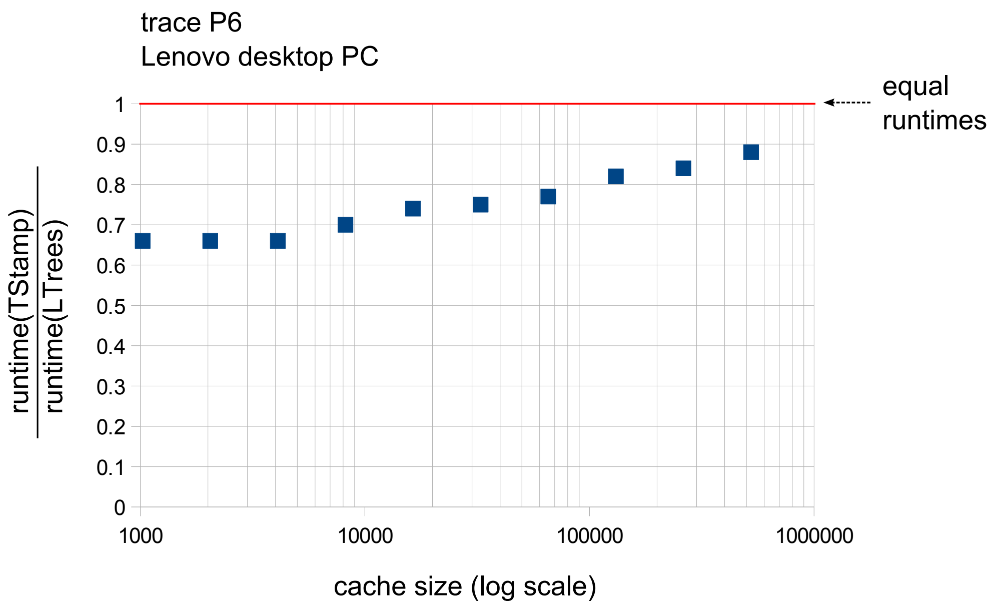


Figure 43: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace P6 processed on Lenovo desktop PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 2.122 | 1.405 | 0.66 |
| 2048 | 2.138 | 1.420 | 0.66 |
| 4096 | 2.232 | 1.468 | 0.66 |
| 8192 | 2.169 | 1.514 | 0.70 |
| 16384 | 2.185 | 1.623 | 0.74 |
| 32768 | 2.201 | 1.655 | 0.75 |
| 65536 | 2.138 | 1.654 | 0.77 |
| 131072 | 2.076 | 1.702 | 0.82 |
| 262144 | 2.029 | 1.701 | 0.84 |
| 524288 | 1.920 | 1.686 | 0.88 |

Table 9: Results for the trace P6 using the Lenovo desktop PC.

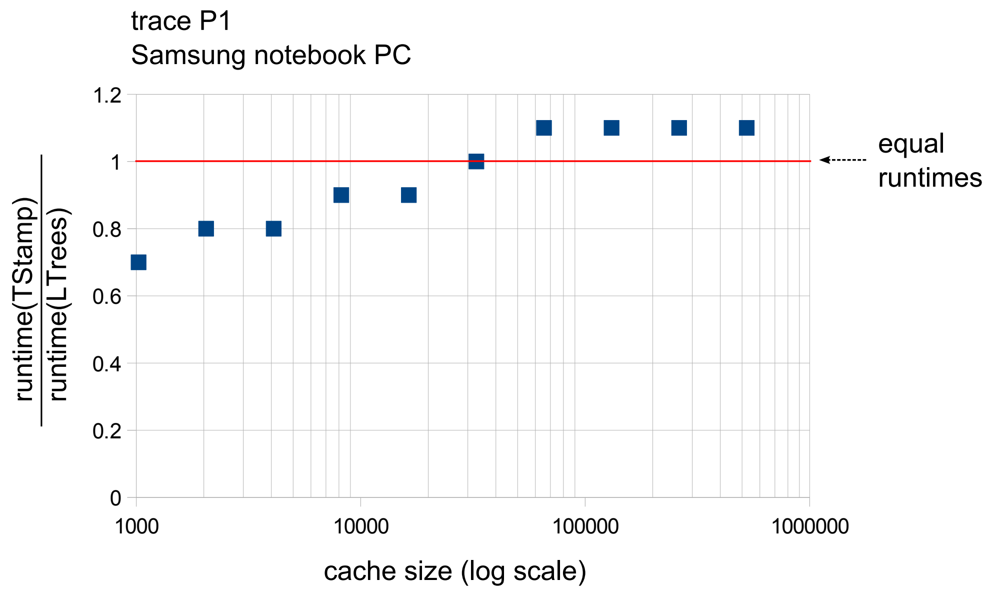


Figure 44: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace P1 processed on Samsung notebook PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 52 | 36 | 0.7 |
| 2048 | 50 | 38 | 0.8 |
| 4096 | 51 | 39 | 0.8 |
| 8192 | 48 | 41 | 0.9 |
| 16384 | 48 | 43 | 0.9 |
| 32768 | 47 | 46 | 1.0 |
| 65536 | 46 | 50 | 1.1 |
| 131072 | 46 | 51 | 1.1 |
| 262144 | 45 | 49 | 1.1 |
| 524288 | 42 | 45 | 1.1 |

Table 10: Results for the trace P1 using the Samsung notebook PC.

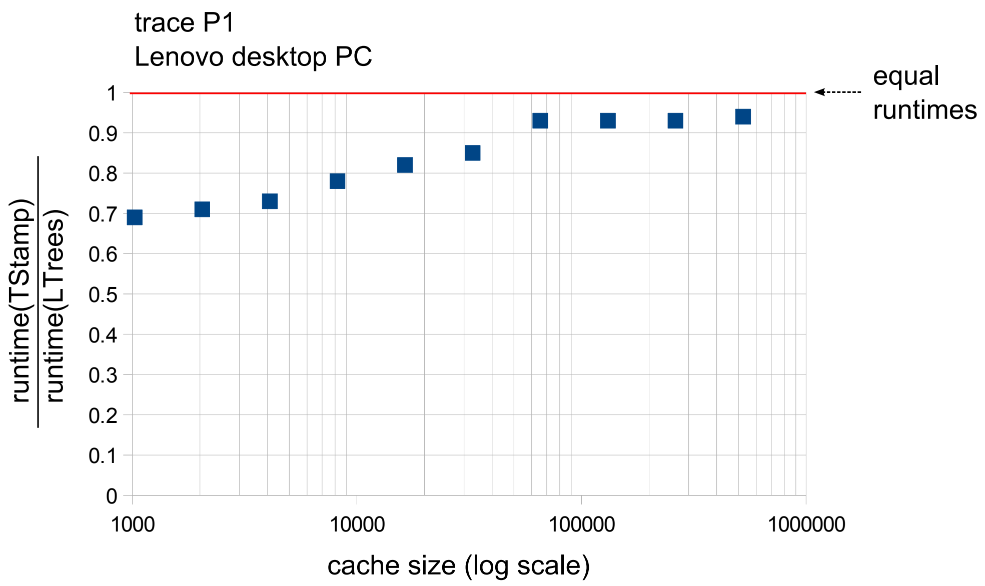


Figure 45: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace P1 processed on Lenovo desktop PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 5.336 | 3.667 | 0.69 |
| 2048 | 5.352 | 3.792 | 0.71 |
| 4096 | 5.243 | 3.808 | 0.73 |
| 8192 | 5.307 | 4.135 | 0.78 |
| 16384 | 5.273 | 4.306 | 0.82 |
| 32768 | 5.367 | 4.556 | 0.85 |
| 65536 | 5.258 | 4.868 | 0.93 |
| 131072 | 5.149 | 4.769 | 0.93 |
| 262144 | 5.055 | 4.697 | 0.93 |
| 524288 | 4.603 | 4.338 | 0.94 |

Table 11: Results for the trace P1 using the Lenovo desktop PC.

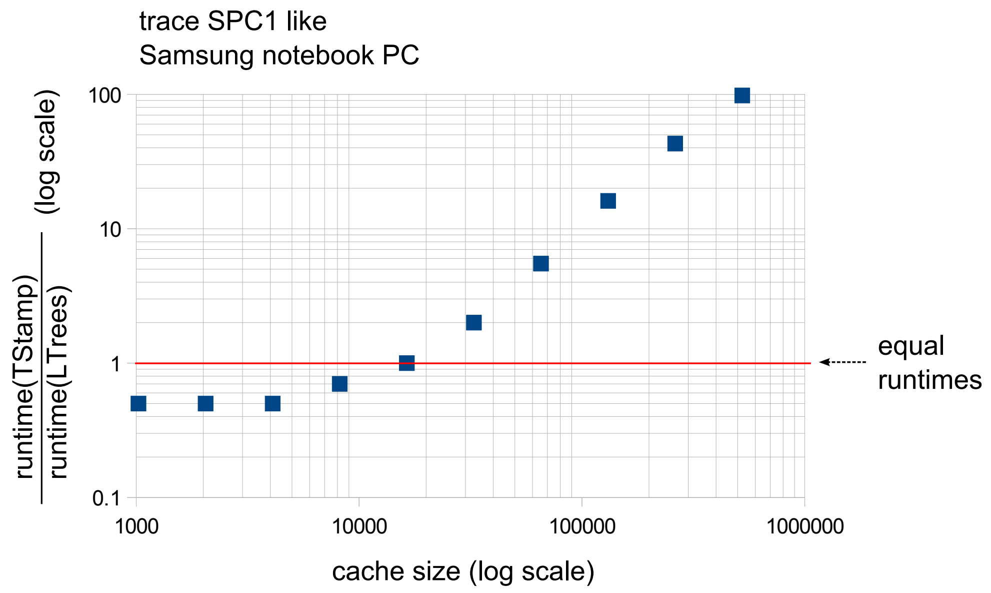


Figure 46: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace SPC1 like processed on Samsung notebook PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 139 | 63 | 0.5 |
| 2048 | 141 | 66 | 0.5 |
| 4096 | 138 | 75 | 0.5 |
| 8192 | 137 | 92 | 0.7 |
| 16384 | 138 | 137 | 1.0 |
| 32768 | 137 | 270 | 2.0 |
| 65536 | 139 | 765 | 5.5 |
| 131072 | 144 | 2313 | 16.1 |
| 262144 | 143 | 6145 | 43.0 |
| 524288 | 145 | 14212 | 98.0 |

Table 12: Results for the trace SPC1 like using the Samsung notebook PC.

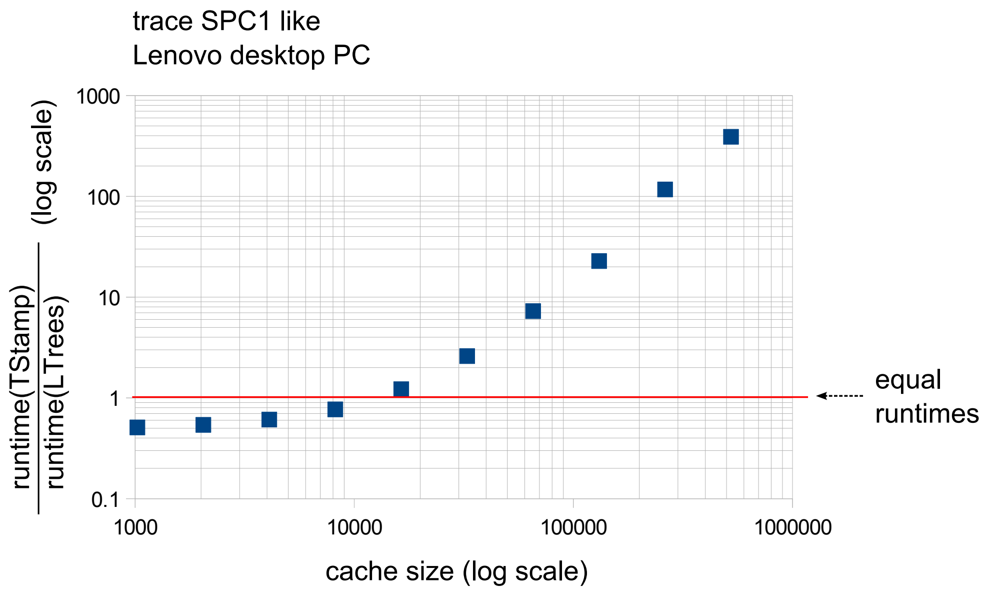


Figure 47: The ratio runtime(TStamp)/runtime(LTrees) is plotted against the cache size. The red line corresponds to points of equal runtimes.

trace SPC1 like processed on Lenovo desktop PC

| cache size | runtime LTrees (in seconds) | runtime TStamp (in seconds) | runtime(TStamp)/ runtime(LTrees) |
|---:|---:|---:|---:|
| 1024 | 13.978 | 7.084 | 0.51 |
| 2048 | 14.071 | 7.552 | 0.54 |
| 4096 | 14.056 | 8.550 | 0.61 |
| 8192 | 14.447 | 11.108 | 0.77 |
| 16384 | 14.383 | 17.597 | 1.22 |
| 32768 | 14.712 | 38.282 | 2.60 |
| 65536 | 14.977 | 108.904 | 7.27 |
| 131072 | 15.289 | 348.368 | 22.8 |
| 262144 | 15.445 | 1811.227 | 117.2 |
| 524288 | 15.929 | 6365.187 | 390.4 |

Table 13: Results for the trace SPC1 like using the Lenovo desktop PC.

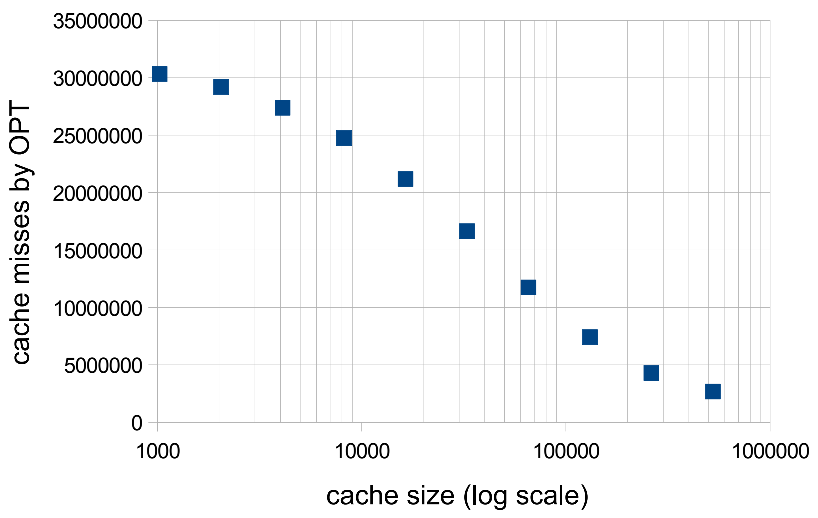


Figure 48: Cache misses caused by OPT when processing trace P1 for different cache sizes.

| cache size | trace S1 | trace P6 | trace P1 | trace SPC1 like |
|---:|---:|---:|---:|---:|
| 1024 | 3,913,529 | 12,323,422 | 30,319,804 | 40,932,071 |
| 2048 | 3,878,293 | 12,141,333 | 29,187,708 | 40,744,270 |
| 4096 | 3,828,086 | 11,799,023 | 27,381,737 | 40,480,354 |
| 8192 | 3,759,145 | 11,158,014 | 24,752,758 | 40,095,501 |
| 16384 | 3,658,723 | 9,940,268 | 21,182,902 | 39,512,083 |
| 32768 | 3,512,624 | 7,762,875 | 16,637,048 | 38,639,773 |
| 65536 | 3,270,542 | 4,196,139 | 11,741,790 | 37,353,435 |
| 131072 | 2,873,783 | 1,520,374 | 7,410,604 | 35,430,956 |
| 262144 | 2,297,465 | 827,165 | 4,293,906 | 32,562,546 |
| 524288 | 1,637,573 | 773,770 | 2,680,440 | 27,990,208 |

Table 14: Number of cache misses caused by OPT when processing a trace at different cache sizes. The values decrease with increasing cache size.

## 3.3 Discussion

**Trace OLTP:** This is the smallest of the five examined traces. LTrees outperforms TStamp on all cache sizes. Starting at $k = 1000$ LTrees is faster by a factor of 1.3, and this factor increases to 4.0 at $k = 15000$.

**Trace S1:** As expected the runtimes for LTrees are not influenced by the cache size $k$ since the amortized runtime per page request is $\mathcal{O}(\alpha(2m))$ and thus independent of $k$. In contrast, we observe that the times for TStamp strongly depend on the cache size. A larger $k$ also means a greater processing time because TStamp processes a page request in $O(k)$ time.

For smaller cache sizes (1024 to 4096) TStamp is faster by a factor of 2, but for greater cache sizes (131072 to 524288) LTrees is faster by a factor of 10.

**Trace P6:** Again the times for LTrees remain almost constant with increasing cache size (they even decrease a little). Surprisingly, TStamp is not affected by an increasing cache size. We can explain this by recalling that TStamp can still be fast for large $k$ if the number of non-empty layers is small, see again remark (R1) in section 2.5.1. This seems to be the case for this trace. On all cache sizes TStamp outperforms LTrees. For smaller $k$ TStamp is faster by a factor of 1.5. However, as $k$ increases this factor decreases to 1.1.

**Trace P1:** Here, the results are similar to P6. This suggests that P1 and P6 do not generate many non-empty layers in the layer partition which is an advantage for TStamp.

**Trace SPC1 like:** The runtimes for LTrees increase by a negligible amount. By comparison the times for TStamp increase dramatically. For $k = 1024$ TStamp is faster by a factor of 2. However, for $k \geq 131072$ the situation is reversed and LTrees is 15 times faster (for $k = 524288$ LTrees is even more than 100 times faster).

**Number of cache misses:** In table 14 we observe that the number of cache misses caused by OPT decrease with increasing cache size $k$. The explanation is that a greater $k$ means a higher probability of finding a requested page in cache, which in turn implies a lower probability for a cache miss.

# 4 Conclusion and future work

## 4.1 Conclusion

For the traces SPC1 like and S1 TStamp is 2 times faster on smaller cache sizes, whereas for greater cache sizes LTrees is more than 10 times faster. So, which data structure shall we choose to simulate the optimal offline paging algorithm OPT? A look at the absolute runtimes may give the answer, e.g. let us examine the times for the trace SPC1 like in table 13.

We measured the runtime for 10 different cache sizes ranging from $k = 1024$ to $k = 524288$. Due to its robustness LTrees required about 16 seconds for each $k$ which means that the overall runtime is $10 \cdot 16$ seconds = 160 seconds. TStamp on the other hand already requires more time with 350 seconds at $k = 131072$, let alone for all 10 cache sizes together. Thus, if we wish to simulate OPT for several cache sizes we should pick LTrees.

Moreover, TStamp is only faster for smaller cache sizes where the runtime is small anyway, e.g. at $k = 1024$ the difference between runtime(LTrees)=14 seconds and runtime(TStamp)=7 seconds is 7 seconds. Compare this to $k = 262144$ where runtime(LTrees)=16 seconds and runtime(TStamp)=1800 seconds = 30 minutes. Here, we have to wait for half an hour till TStamp has processed the trace.

For the trace P6 the results in figure 43 and table 9 are in favor of TStamp. On all cache sizes TStamp outperforms LTrees. Trace OLTP demonstrates a reverse situation. Here, LTrees outperforms TStamp on all cache sizes.

In summary, the robustness towards increasing cache sizes is a strong argument for selecting LTrees. It means that when we conduct experiments we won't get surprised by a sudden increase of the runtime.

## 4.2 Future work

For a future project or thesis it would be interesting to implement LTrees in C++ instead of Java. This rules out a possible influence of the Java garbage collector. Another aspect is the implementation of the nodes. For each node attribute we introduced an array, such as the parent array and the rank array. However, we could have also used a *Node class* that contains these attributes and whose objects represent the nodes. All nodes are then stored in an array of type *Node*. We decided not to follow this approach in Java for two reasons. First, the objects require more space. Second, in Java there is no guarantee that the objects are allocated contiguously [27]. In C++ the situation is different [28], and it would be interesting to know if such an approach improves the runtime.

One could also examine how other path compression schemes and using union by size [13] instead of union by rank change the runtime. A survey of path compression schemes is given in [29, 30].

LTrees allows us to determine the empirical competitive ratio [7]. This could be used to decide adaptively which page replacement algorithm we should employ.

Furthermore, LTrees can be utilized as a subroutine for the paging algorithm RLRU [2] to improve the runtime from $\mathcal{O}(\log(m))$ to $\mathcal{O}(\alpha(2m))$ amortized time, where $m$ is the number of unique pages requested.

Moruz and Negoescu introduced the so-called attack rate $r$ which measures how "hard" a request sequence is [7]. Due to its robustness towards different cache sizes it is possible for LTrees to determine $r$ in reasonable time.

Finally, we want to mention the union-find data structure by Alstrup et al. [31] which supports deletions in constant time, and which has been simplified by Ben-Amram and Yoffe [32, 33]. Using this data structure to implement the layer partition would make for a nice project.

# Appendix

## A Interpretation of the layer partition

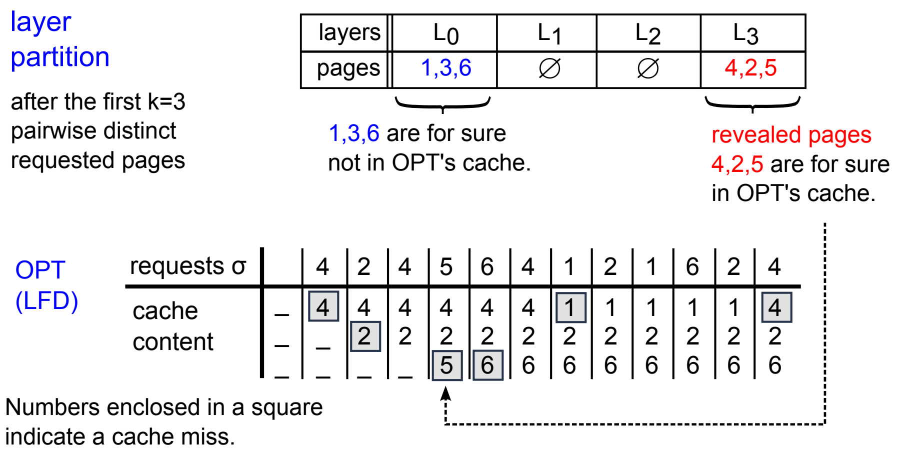
We want to further describe the relationship between the layer partition $(\mathcal{L}_0|\mathcal{L}_1|...|\mathcal{L}_k)$ and the optimal offline paging algorithm OPT as done in [12, 7]. When a page $p$ is requested it can be assigned to one of the following three categories:

1. $p \in \mathcal{L}_k$. Pages requested from $\mathcal{L}_k$ have the property of being in OPT's cache. For this reason they are called *revealed*.

2. $p \in \mathcal{L}_0$. Pages requested from $\mathcal{L}_0$ are not in OPT's cache and cause a page fault.

3. $p \in \mathcal{L}_i$ with $0 < i < k$. These pages might be in OPT's cache, but this depends on the future requests. They are denoted *unrevealed* since we cannot be sure that they are in OPT's cache.

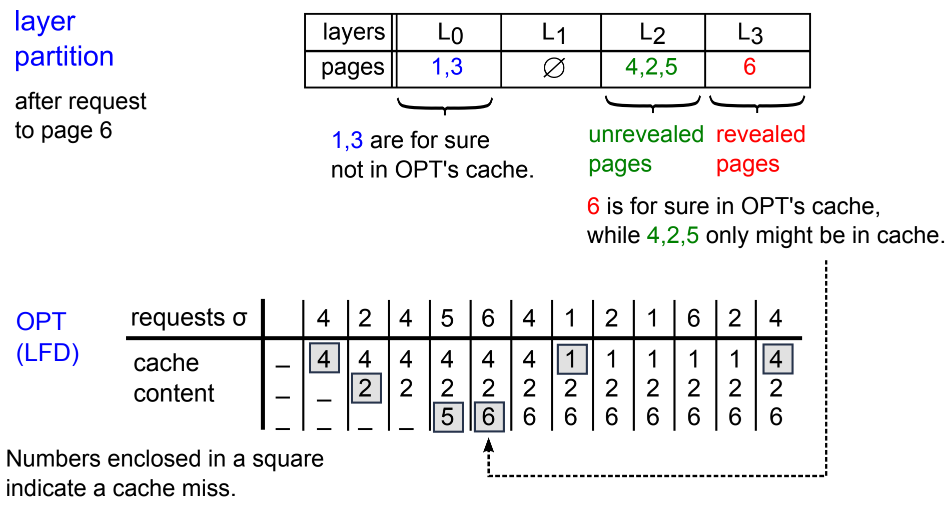
Let us revisit an example from section 2.1 with $M = \{1, 2, 3, 4, 5, 6\}$ being the page-set, $k = 3$ the cache size and $\sigma = (4, 2, 4, 5, 6, 4, 1, 2, 1, 6, 2, 4)$ the request sequence. The first $k = 3$ pairwise distinct requested pages are $\{4, 2, 5\}$, from which we can deduce the initial partition $(\mathcal{L}_0|\mathcal{L}_1|\mathcal{L}_2|\mathcal{L}_3) = (1, 3, 6|\emptyset|\emptyset|4, 2, 5)$. Now, according to our interpretation the pages in $\mathcal{L}_k = \mathcal{L}_3$ are in the cache of OPT. This is indeed the case since the first $k$ distinct pages will fill the initially empty cache (see figure 49).



layer partition
after the first k=3 pairwise distinct requested pages
layers | L0 | L1 | L2 | L3
pages | 1,3,6 | ∅ | ∅ | 4,2,5
1,3,6 are for sure not in OPT's cache.
revealed pages 4,2,5 are for sure in OPT's cache.
OPT (LFD)
requests σ: 4 2 4 5 6 4 1 2 1 6 2 4
cache content
Numbers enclosed in a square indicate a cache miss.

Figure 49: Comparison of the layer partition with OPT's cache.

Page 6 is requested next. Since 6 is in $\mathcal{L}_0$ this causes a cache miss and we will have to evict one of the pages 4,2,5 from the cache. The layer partition translates this by shifting 4,2,5 from $\mathcal{L}_k$ to $\mathcal{L}_{k-1}$ such that pages 4,2,5 become unrevealed. For these pages we cannot be sure anymore that they are in OPT's cache since one of them gets replaced (see figure 50).


layer partition
after request to page 6
1,3 are for sure not in OPT's cache.
unrevealed pages
revealed pages
6 is for sure in OPT's cache, while 4,2,5 only might be in cache.
OPT (LFD)
Numbers enclosed in a square indicate a cache miss.

| layers | $L_0$ | $L_1$ | $L_2$ | $L_3$ |
|---|---|---|---|---|
| pages | 1,3 | ∅ | 4,2,5 | 6 |

| requests σ | | 4 | 2 | 4 | 5 | 6 | 4 | 1 | 2 | 1 | 6 | 2 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cache content | – | $\boxed{4}$ | 4 | 4 | 4 | 4 | 4 | $\boxed{1}$ | 1 | 1 | 1 | 1 | $\boxed{4}$ |
| | – | – | $\boxed{2}$ | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | – | – | – | – | $\boxed{5}$ | $\boxed{6}$ | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

Figure 50: Comparison of the layer partition with OPT's cache.

## References

[1] Belady, L. A. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(1966):78-101.

[2] J. Boyar, L. M. Favrholdt, and K. S. Larsen. The relative worst-order ratio applied to paging. *Journal of Computer and System Sciences*, 73(5):818-843, 2007.

[3] Gerth Stølting Brodal, Gabriel Moruz, Andrei Negoescu: OnlineMin: A Fast Strongly Competitive Randomized Paging Algorithm. *WAOA 2011*: 164-175

[4] There are actually more levels in the memory hierarchy: CPU, L1 cache, L2 cache, RAM, hard disk. Instead of RAM and hard disk one can also examine the relationship between two other levels such as L2 cache and RAM, see *Memory Hierarchy Design - Part 1. Basics of Memory Hierarchies* by John L. Hennessy, Stanford University, and David A. Patterson, University of California, Berkeley - September 25, 2012. `http://www.edn.com/design/systems-design/4397051/Memory-Hierarchy-Design-part-1`

[5] Susanne Albers. Lecture notes on Competitive Online Algorithms. *BRICS, Aarhus university*, 1996. `http://www2.informatik.hu-berlin.de/~albers/papers/brics.pdf`

[6] `http://www.youtube.com/user/johncctang`.

[7] Gabriel Moruz, Andrei Negoescu. Outperforming LRU via competitive analysis on parametrized inputs for paging. *SODA 2012*: 1669-1680

[8] Susanne Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2): 3-26 (2003)

[9] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28:202-208, 1985. `http://www.cs.cmu.edu/~sleator/papers/amortized-efficiency.pdf`

[10] M. Chrobak, J. Noga. LRU is better than FIFO. *Algorithmica*, 23:180-185, 1999.

[11] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30:300-317, 2000.

[12] G.Moruz, A. Negoescu, C. Neumann, and V.Weichert. Engineering efficient paging algorithms. In *Proc. 11th International Symposium on Experimental Algorithms*, 2012.

[13] B. A. Galler and M. J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301-303, 1964.

[14] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215-225, 1975.

[15] M. C. Golumbic and I. B.-A. Hartman. *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications (Operations Research/Computer Science Interfaces Series)*. Springer-Verlag New York, Inc., 2005.

[16] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983.

[17] We use the one-parameter version $\alpha(n)$ of the inverse Ackermann function, but in his original work Tarjan uses the two parameter version $\alpha(m, n)$ [16, 14]. The relationship between them is described by Jeff Erickson in his lecture notes [18]. Cormen et al. also use the one-parameter version in their book [19]. Tarjan himself describes his result with the one-parameter version in [15]. Moreover, Tarjan does not necessarily perform $n$ makeset operations in the beginning to create $n$ elements. He considers $n$ makeset, at most $n-1$ union and $m \geq n$ find operations in an arbitrary order.

[18] Jeff Erickson. Algorithms Course Materials, Chapter 16: Data Structures for Disjoint Sets, Department of Computer Science, University of Illinois at Urbana-Champaign, 2011. `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/`

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms (3. ed.). MIT Press 2009.

[20] `http://www.cs.unm.edu/~rlpm/499/uf.html`

[21] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. `http://www.brpreiss.com/books/opus5/html/page410.html`

[22] We can determine the index $i$ for a page $p \in \mathcal{L}_i$ with $0 < i < k$ as follows: we first apply a find($p$) operation which returns the root node

$r$ in $\mathcal{O}(\alpha(2m))$ time. From $r$ we follow the left pointers and count the number of layers we traverse (including the empty layers) until we reach the root node of $\mathcal{L}_0$, which requires $\mathcal{O}(k)$ time. This gives us the value for $i$ in $\mathcal{O}(\alpha(2m) + k)$ time. If the number of non-empty layers is $l < k$, then the time reduces to $\mathcal{O}(\alpha(2m) + l)$.

[23] My advisor Andrei Negoescu had the idea of removing elements in a tree by using lazy deletion. He also devised the steps of the cleanup procedure.

[24] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, pp. 115-130, March 31-April 2, 2003. `http://www.almaden.ibm.com/cs/people/dmodha/`

[25] `http://stackoverflow.com/questions/2313062/a-fast-queue-in-java`

[26] `http://harvestsoft.net/files/queue.pdf`

[27] `http://stackoverflow.com/questions/9632288/can-i-allocate-objects-contiguously-in-java`

[28] `http://stackoverflow.com/questions/8297993/how-is-an-array-of-objects-stored-in-memory`

[29] R. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, Volume 31 Issue 2, 1984.

[30] Md. Mostofa Ali Patwary, Jean Blair and Fredrik Manne. Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure. *Proceedings of 9th International Symposium on Experimental Algorithms (SEA'10)*, Springer LNCS 6049, pp. 411-423, 2010.

[31] Stephen Alstrup, Inge Li Gørtz, Theis Rauhe, Mikkel Thorup, Uri Zwick. Union-Find with Constant Time Deletions. *ICALP 2005*: 78-89

[32] Amir M. Ben-Amram, Simon Yoffe: A simple and efficient Union-Find-Delete algorithm. *Theor. Comput. Sci.*, 412(4-5): 487-492, 2011.

[33] Amir M. Ben-Amram, Simon Yoffe: Corrigendum to "A simple and efficient Union-Find-Delete algorithm" [Theoret. Comput. Sci. 412(4-5) 487-492]. *Theor. Comput. Sci.*, 423: 75, 2012.