Persistence in Practice

Sune Keller

Kongens Lyngby 2012
IMM-M.Sc.-2012-142

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2012-142

Summary

The goal of the thesis is to theoretically analyse and compare Node Copying and Rollback, two approaches to making a data structure partially persistent (allowing the access of any previous version), and to evaluate in practice which of them is more suited for use in two different usage scenarios: a sequential one, where operations are applied in long sequences generating versions with large data structures; and a randomized one, where operations are executed in a random order. A doubly linked list is used as the example data structure.

It is found that Node Copying performs significantly better than Rollback in terms of space complexity regardless of the usage scenario. It also performs better in accessing a desired version in the randomized scenario independently of the data set size.

In the sequential scenario, when the data set is large enough, the smaller time complexity related to navigating the data structure at a given version makes an optimized implementation of Rollback faster than Node Copying at reaching a given index inside a desired version.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

Lyngby, 30-November-2012

[illegible]

Sune Keller

Acknowledgements

I would like to thank my supervisers Inge Li Gørtz and Philip Bille of the Algorithms and Logic section of the department of Informatics and Mathematical Modelling of the Technical University of Denmark for their guidance and reflections throughout this project.

Contents

Chapter 1

Introduction

Persistence is the topic dealing with the general availability of previous versions of a data structure for various purposes. The antonym to “persistent” is in this context called “ephemeral”, which means that no previous version of a data structure is available. In the 1989 article by Driscoll et al. [DSST89], it is described how a data structure may be made persistent.

With partially persistent data structures, any previous version is accessible, but not modifiable, and as such changes to the data structure can only be made on the most recent version.

Partially persistent data structures can be compared with “rollback” databases as defined in [VL87]. “Rollback” databases allow reading from any previous version of the data structure, but only allow changing the most recent version. Driscoll et al. describe a method for making any bounded in-degree data structure partially persistent with $O(1)$ amortized space overhead per version and $O(1)$ amortized overhead for access of any version and operations on the most recent version. This method is called Node Copying.

My thesis will analyse and compare Node Copying and Rollback as two different approaches to obtaining partial persistence, using a doubly linked list as an example data structure.

1.1 Scope

In my thesis, only data structures with bounded in-degree will be considered, and the scope is limited to partially persistent data structures.

The thesis is mainly concerned with making a doubly linked list partially persistent.

When discussing time and space complexity, the RAM model will be used, and big-O notation is used when deriving and comparing such complexities.

CHAPTER 2

Method

In this chapter I will describe two approaches to partial persistence. I will analyse the asymptotic time and space complexities related to creating new versions to the data structure and to retrieving an element in a desired existing version.

2.1 Background

Tsotras and Kangelaris [TK95] define as the database state $s(t)$ the collection of objects that exist at time t in the real-world system modeled by the database. The ability to access $s(t)$ is here referred to as "temporal access", and is essentially the same as partial persistence in that it allows read-only access to all versions and read/write access only to the newest version. I will define the metric by which the approaches are measured as the cost of producing $s(t)$ and navigating to an element within it.

I will analyse the following two approaches to obtaining partial persistence:

Node copying which requires structural extensions of the underlying data structure in order to store modification records and other necessary information within the nodes of the data structure; and

Rollback which requires functional extensions of the underlying data structure in order to record the necessary information about the context in which an operation is carried out in order to be able to reproduce or revert it at a later time.

2.2 The Node Copying method

The purpose of the Node Copying method is to enable partial persistence in a node-based data structure with bounded in-degree p with an amortized constant factor overhead on space and time complexity and providing transparent navigation of the data structure.

Node Copying is a method described in [DSST89] by which a data structure may be made partially persistent through the systematic expansion of the node structure. An auxillary data structure maintaining entry points into the data structure, such as which node is the head of a linked list, is also introduced. These two modifications of the data structure enable queries to be made with $O(1)$ factor amortized overhead and $O(1)$ factor space per change, given that the original data structure has a bounded in-degree. In the following, I will describe in detail the general procedure as well as giving the concrete example of the linked list.

2.2.1 Node structure expansion

In the Node Copying method, the node structure is expanded by adding two arrays; one for “modifications” and one for “back pointers”.

Persisting fields. The modifications array records changes made to any of the fields of the node, including non-pointer fields, since the time when node was created — the original field values from the time of construction remain unchanged. A modification record consist of a version number, a field identifier and a value. When a fields is to be modified, a record is inserted in an empty slot of the array.

To find the value of a field at a given version v, the modifications array is looked through, and the most recent change before or at v is returned. If no modification is found, the original field value is returned.

In the Fat Node method described in [DSST89], there is no upper bound on the size of the modifications array. Therefore, the worst-case time to retrieve a field value is linearly proportional with the number of modifications.

Limited node size. To get constant factor amortized overhead for field access, the size of the modifications array is bounded to the in-degree bound p. The time to retrieve the field value is then at most $c \cdot p + k = O(1)$ where c is the time it takes to read each modification record and field identifiers and version numbers, and k is the time it takes to read the original field if neccessary.

If the modifications array is already occupied with previous changes when a new change is required, a copy n' of the original node n is made. The fields of n' are set to the most recent versions of the fields in n, except that the field which was to be changed is instead set to the required value. Note that the modifications array of n' is empty.

But if any nodes were – in the most recent version – pointing to n prior to the change, they will need to be updated to point to n'. To be able to know which nodes were pointing to n, a "back pointers" array is maintained. Its size must of course be equal to the in-degree bound p to be able to contain pointers to all nodes pointing to n. Whenever a pointer field in a node y is updated to point to another node x (either when y is constructed or when the field is modified as described above), the corresponding back pointer in x is updated to point to y.

Cascading effects. When n' is created, the back pointers of n are followed, and each of the nodes which pointed to n will have a modification added referring to the relevant field, and pointing to n'. But if the modifications array is already full in one of these nodes, a copy of *that* node will need to be made as described above. This cascading effect can repeat.

For example, when an element y is inserted at the head of a singly linked list, its NEXT pointer will point to the element x which used to be the head of the list. The "NEXT-back" pointer in x is then set to point to y. If the modifications array of x gets full and a copy x' needs to be made to allow a new modification, the "NEXT-back" pointer is followed to make a modification record in y that in the new version, it should point to x' instead of x. But if a modification was already made to y, e.g. its data field was updated, it will itself need to be copied according to the same scheme — i.e. there may be a cascading effect. In this example, where y was the head of the list, its copy y' should be indicated as the new head in the new version in the auxillary data structure.

Amortized constant factor overhead. To prove that – even with cascading effects – there is still a constant factor amortized overhead per field change, we shall employ the potential technique.

The intuition is that for every time we copy a node n, the copy n' will have an empty modifications array, and thus it will take p modifications to n' before a new copy must be made. A node of which a copy has already been made cannot be the target of yet another modification, and thus the available cheap modifications to n' account for the time spent on creating n'.

The following proof outline is based on the proof made in the original paper [DSST89], but a notable difference here is the absence of a copy pointer, and the specification that the modifications are stored in a fixed-size array.

We define a *live node* as a node which can be reached in the most recent version of the data structure, and a *full* live node as a live node with a full modifications array. Let the potential function $\Phi(v)$ be the number of full live nodes at version v. Then the potential at the initial version $\Phi(0)$ is zero, and since there cannot be a negative number of full live nodes, the potential is always greater than or equal to zero.

The *actual cost* of an update to a full live node which causes a total of k copies to be made (both due to the node itself being copied and due to any cascading effects) is $O(k) + O(1)$ since each copy costs $O(1)$ both in space and time, and modifying a field costs $O(1)$. For each of the copies created, the original node is no longer live (since nodes pointing to it are updated to point to the copy), and the modifications array of the copy is empty. Thus, each copy decreases the potential by 1, and the *total change in potential* $\Delta\Phi$ is $O(-k)$ plus $O(1)$ in case the last update made via a back pointer fills the modifications array of that node. Therefore, the amortized time and space cost for the entire update operation is $O(k) + O(1) + (O(-k) + O(1)) = O(1)$.

2.3 The Rollback method

Rollback is an approach to partial persistence based on the techniques described in [TK95], namely the combination of the naïve “copy” and “log” methods.

2.3.1 The naïve approaches

The “copy” approach makes a full copy of every version of the data structure and makes it available by direct indexing to achieve $O(1)$ access overhead factor. Creating each copy becomes more expensive in time and space the more elements are inserted. In the worst case, when only insertions and no deletions or modifications are made, the cost of creating n versions is $O\left(n^2\right)$.

The “log” approach conserves space by recording for each change made to the data structure just enough information necessary to undo or redo it, thus giving a space overhead factor per operation of $O(1)$. A “current” version $v_{current}$ of the data structure is maintained. Given $v_{current}$, the version v_x can be produced by undoing or redoing all the changes between $v_{current}$ and v_x depending on which is the oldest. As the number of versions n increases, accessing a specific version becomes potentially more costly. In the worst case, the overhead factor is $O(n)$ when $v_{current}$ is v_0 and v_x is v_n, or opposite.

2.3.2 The hybrid approach

An obvious hybrid of the two naïve approaches is to keep the records of each operation like in the “log” approach, and storing a full copy like in the “copy” approach only once every d versions. To access version v_x, the nearest version to v_x of which there exists a full copy, v_s, is retrieved and a mutable copy of the data structure at that version is made. Then, using the log of the versions from v_s to v_x, the data structure corresponding to v_x is produced and returned.

In the rest of this document, the term Rollback refers to the hybrid approach.

To reach the full copy nearest to version number v, the following expression is used:

$$\left\lfloor \frac{v + \frac{d}{2}}{d} \right\rfloor$$

With this indexing expression, full copies will be selected for all operations which are within $\pm\frac{d}{2}$ versions.

Naturally, d is a user parameter. If there are n versions, there will be $\left\lfloor \frac{n}{d} \right\rfloor$ full copies, and the maximum number of operations to reach a specific version is then $O(k + \frac{d}{2})$, where k is the time required to retrieve the full copy. It is now easy to see that the greater d is, the fewer full copies will be made, and thus the space cost is reduced accordingly. Likewise, the distance to the version in the middle between two full copies increases with d, inducing a higher cost for producing that version.

As an alternative to making a mutable copy of a full copy when a nearby version is requested, one could instead allow the full copies to be mutable and moved within $\pm\frac{d}{2}$ of their original location. This will increase the maximum number of operations between a full copy and the desired version to d, but will reduce greatly the time cost of making a working copy of the full copy. If the full copies are originially uniformly distributed, direct lookup is still possible, but it is then needed for each full copy to annotate which version it represents. If d is constant, this approach allows constant time access to any version of the data structure.

For large enough data sets, physical memory limits may restrict the ability to make new full copies. A way to defer this is to increase d by a factor c when near the limit and discard all but every c full copy that exists. This will free some memory for creating new full copies, but will increase the distance between them to $d' = c \cdot d$. Therefore, the claim no longer holds that the time to access any version of the data structure is constant. Also, the time spent previously on creating the full copies which are now discarded is no longer valuable.

The time required to make a changing operation to the latest version equals the time required to produce v_{latest}, plus the time required to carry it out using the underlying data structure, plus the time required to adding an operation record to the log. Carrying out the operation depends on the operation in question, but recording the operation in the log potentially requires extra information, e.g. a REMOVE operation needs to store information in the log allowing the reproduction of the removed element. Thus, the time complexity depends both on the operation in question and on the inverse operation, which may have different time complexity bounds.

It is obvious that with a doubly linked list, the overhead factor for making a change to the latest version is $O(1)$.

On the other hand, creating the full copy becomes more expensive the more elements are inserted in the preceeding versions. Thus it is intuitively not possible to show $O(1)$ factor amortized time overhead for a sequence of INSERT operations long enough to cause a full copy to be made.

2.3.3 Operations sequence optimization

For certain underlying data structures, it may prove feasible to pre-process the sequence of operations between a full copy v_s and the requested version v_x in order to reduce the actual work necessary to reach v_x.

Two different types of pre-processing to reduce work when dealing with a linked list are presented here: Eliminating superfluous operations and reordering operations. It is worth noting that both of these require knowledge of the underlying data structure, and thus require more programming time to implement.

2.3.3.1 Eliminating operations

The sequence of operations between v_s and v_x may contain some operations which will not need to be explicitly applied. The cases are, in the context of a linked list:

1. An INSERT operation followed by a REMOVE operation at the same effective index — both can be removed from the sequence since they cancel each other out.
2. An INSERT or a MODIFY operation followed by a MODIFY operation at the same effective index — the former can have its associated data value changed to that of the latter, and the latter can be removed from the sequence.
3. A MODIFY operation followed by zero or more MODIFY operations and a REMOVE operation at the same effective index — the former can be removed from the sequence.

Cases 1 and 2 may of course be combined in the sequence of one INSERT operation, one or more MODIFY operations and finally a REMOVE operataion.

Algorithm. Pseudo code for an algorithm handling those cases is found in Algorithm 2.1.

The general idea is to identify cases 1, 2 and 3 and make the necessary maintenance before removing the relevant operations.

Algorithm 2.1 An algorithm for eliminating superfluous operations

```
procedure EliminateSuperfluousOps(sequence of operations $S$)
    for each operation $o_i \in S$, from last to first do
        if $\text{TYPE}(o_i) \in \{\text{INSERT}, \text{MODIFY}\}$ then
            $c \leftarrow \text{INDEX}(o_i)$
            for each operation $o_j \in \{S | j > i\}$ do
                switch $\text{TYPE}(o_j)$ do
                    case INSERT
                        if $\text{INDEX}(o_j) \leq c$ then
                            $c \leftarrow c + 1$
                    case MODIFY
                        if $\text{INDEX}(o_j) = c$ then
                            $\text{DATA}(o_i) \leftarrow \text{DATA}(o_j)$
                            remove $o_j$ from $S$
                    case REMOVE
                        if $\text{INDEX}(o_j) < c$ then
                            $c \leftarrow c - 1$
                        else if $\text{INDEX}(o_j) = c$ then
                            switch $\text{TYPE}(o_i)$ do
                                case INSERT
                                    $c \leftarrow \text{INDEX}(o_i)$
                                    for each operation $o_k \in \{S | i < k < j\}$ do
                                        if $\text{INDEX}(o_k) > c$ then
                                            $\text{INDEX}(o_k) \leftarrow \text{INDEX}(o_k) - 1$
                                        else
                                            switch $\text{TYPE}(o_k)$ do
                                                case INSERT
                                                    if $\text{INDEX}(o_k) \leq c$ then
                                                        $c \leftarrow c + 1$
                                                case REMOVE
                                                    if $\text{INDEX}(o_k) < c$ then
                                                        $c \leftarrow c - 1$
                                    remove $o_j$ from $S$
                                    remove $o_i$ from $S$
                                case MODIFY
                                    remove $o_i$ from $S$
```

It iterates through the sequence backwards, i.e. from the last operation to the first. If an INSERT or a MODIFY operation o_i is found, it starts looking forwards for a matching MODIFY or REMOVE operation. This is done by iterating through the proceeding operations while keeping track of which index c of the inserted or modified element would have after each of the proceeding operations; if a proceeding operation o_j inserts or removes an element to the left of the originally inserted or modified element, then the index would be shifted to the right or to the left, respectively.

If o_j removes or modifies the element at index c, the type of o_i determines what happens next. Table 2.1 shows which action is taken depending on the types of o_i and o_j.

o_i \ o_j	MODIFY	REMOVE
INSERT	update o_i data, remove o_j	compensate between o_i and o_j, then remove both
MODIFY	update o_i data, remove o_j	remove o_i

Table 2.1: The table shows which action is taken depending on the types of o_i and o_j.

The compensation referred to in the case when o_i is an INSERT operation and o_j is a REMOVE operation works as follows:

- Like in the iteration of the operations proceeding o_i, the index c of the inserted element is maintained while the operations between o_i and o_j (both exclusive) are iterated first to last.
- Since no operation prior to o_j matches o_i, it is safe to assume that all operations between them have either greater or smaller indices than c. If an operation o_k between o_i and o_j works on a smaller index, it will compensate the maintained index c by -1 or +1. If o_k works on a greater index, o_k will itself have the index it works on reduced by 1, since its original index depended on the element of o_i being inserted before it.
- When all operations between o_i and o_j have been examined thusly, o_j and o_i are removed from the sequence and the backwards iteration is resumed.

When the backwards iteration is finished examining the first operation in the sequence, the algorithm terminates and the sequence now contains no operations matching cases 1, 2 or 3.

The space cost of the algorithm is in the order of the sequence size, i.e. $O(1)$, since it works directly on the sequence. In the worst case, when all n operations are INSERT or MODIFY operations, the time cost is $O\left(n^2\right)$, since with every INSERT operation, all the proceeding operations are examined when a matching REMOVE operation is not found.

2.3.3.2 Reordering operations

Applying each operation in the sequence separately is potentially a time expensive approach. Consider the case when each operation inserts an element at the end of a linked list; the total cost for n insertions on a list of length l would be $O\left((l+n)^2\right)$ if every operation application begins by iterating from the head of the list to the index of insertion.

It would indeed be more efficient to make the insertions in one straight iteration of the linked list, which would cost $O(l+n)$; l for reaching the end of the list, n for all the insertions. But this is only possible if the the operations are ordered by non-decreasing index of application. If they are not, they will have to first be reordered.

Reordering the sequence of operations by index of application requires some maintenance. Consider the case when an INSERT operation o_i, which has a small index of application, is moved to an earlier position in the sequence than it had originally. Any operations which formerly preceeded o_i, and which after the move will instead proceed it, should take into account the additional element being inserted; if their index of application is greater than that of o_i, it should be incremented by 1. If o_i is a REMOVE operation, the index of application should instead be decreased by 1 on those operations.

Algorithm. Pseudocode for an algorithm implementing the above approach is found in Algorithm 2.2.

In each iteration of the algorithm, the minimum operation is identified and moved to the left in the sequence such that eventually the sequence will be non-decreasingly ordered by index of application.

The space cost is linear in the number of operations being reordered. Consider the worst case, when the operations are ordered with strictly decreasing indices of application; the time cost is then $O\left(n^2\right)$, since the entire remaining sequence would have to be searched for the minimum index of application prior to the

Algorithm 2.2 Algorithm for reordering a sequence of operations

```
function ReorderOperations(sequence of operations $S$)
    $R \leftarrow$ empty set
    while $S \neq \emptyset$ do
        // Find left-most operation with minimum index of application
5:      $o_{min} \leftarrow$ op. with minimum index of application
        $index_{min} \leftarrow$ index in $S$ of $o_{min}$
        REMOVE $o_{min}$ from $S$
        APPEND $o_{min}$ to $R$

        // Compensate op.s which will now proceed rather than preceed $o_{min}$
10:     if $o_{min}$ is an INSERT op. then
            for each op. $o_i \in \{S | 0 \leq i < index_{min}\}$ do
                INDEX($o_i$) $\leftarrow$ INDEX($o_i$) + 1
        else if $o_{min}$ is a REMOVE op. then
            for each op. $o_i \in \{S | 0 \leq i < index_{min}\}$ do
15:             INDEX($o_i$) $\leftarrow$ INDEX($o_i$) - 1

    return $R$
```

removal of that element.

If the operations are already ordered by non-decreasing index of application, the algorithm only adds to the time cost, and so it may be worth checking that the sequence requires reordering prior to running the algorithm. That can be achieved in $O(n)$ time by testing whether each operation works on an index equal to or greater than the previous one.

The algorithm does not require superfluous operations to be eliminated with Algorithm 2.1 in order to yield a correct result, but I shall suffice with proving its correctness under the assumption that no operation works on the same effective element of the underlying linked list.

If the sequences are reordered as described, the total time cost of reaching version v_x is:

$$\begin{array}{rrl} \textsc{CheckIfUnordered} & +\textsc{ReorderOperations} & +\textsc{ApplyOperations} = \\ O(n) & +O(n^2) & +O(l+n) = \\ & & O(l+n^2) \end{array}$$

An example of the reordering of 10 operations by the algorithm is found in figure 2.1.

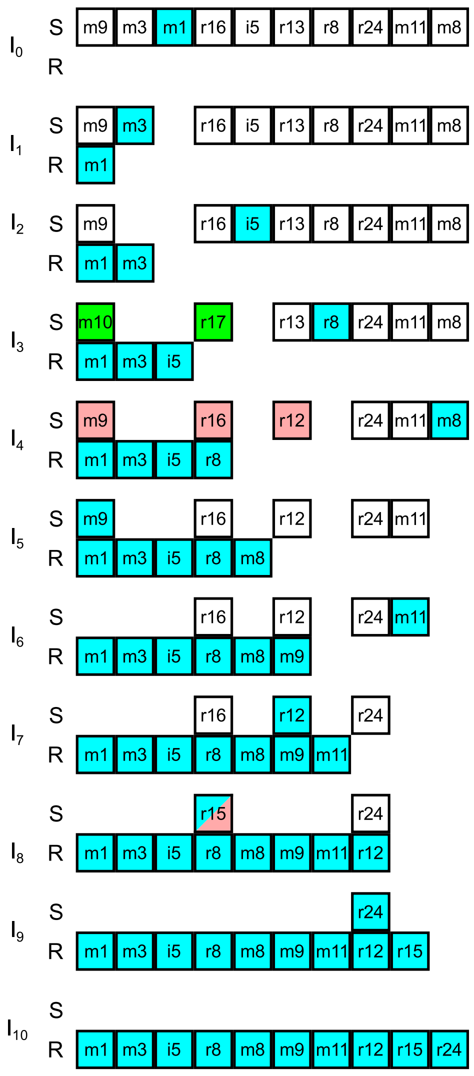

Figure 2.1: Example of the reorder algorithm applied to a sequence of 10 operations. In each iteration, the minimum operation in S is colored cyan. When an INSERT operation is removed from S, the operations to the left are colored green to indicate the incrementation of their label. Similarly, when a REMOVE operation is removed from S, the operations to the left are colored red to indicate the decrementation of their label.

Proof of reorder algorithm. The purpose of the algorithm is to reorder the operations in the given sequence S_0 such that they are ordered non-decreasingly by index of application, while yielding the same result as S_0, allowing the application of the entire sequence of operations in a single iteration through the underlying list.

Definition 2.1 Let S_0 be the original sequence given as input to the algorithm, and let O_0 be an empty sequence at the beginning of the first iteration. Let the operation $A||B$ signify the the unified sequence of A followed by B. The sequence $O_0||S_0$ corresponds to (yields the same result as) S_0. At the beginning of iteration i, let l_{min_i} be the minimum index of any operation in S_i, and let o_{min_i} be the left-most operation in S_i with such index. Let L_i be the operations to the left of o_{min_i} in S_i, and R_i those to the right, i.e. $S_i = L_i||o_{min_i}||R_i$.

Lemma 2.2 *At the beginning of iteration i, all operations in L_i have greater indices than o_{min_i}, and no operations in R_i have smaller indices than o_{min_i}.*

Since we intend to change the order of operations, consider the case of swapping two operations. Let operation o_b directly follow o_a in a sequence. If o_b works on a smaller index than o_a, letting o_b be applied before o_a will potentially shift the elements in the list on which the operations work, and o_a will subsequently work on the wrong element unless compensation is made.

Specifically, if o_b is an INSERT operation, o_a should be compensated by incrementing by 1 the index on which it works, and by decrementing it by 1 if o_b is a REMOVE operation. Let the compensated o_a be denoted o_{a1}. If o_b is a MODIFY operation, it does not shift the elements in the list, and as such nothing additional needs to be done after swapping the operations.

As a result, applying $o_a||o_b$ corresponds to applying $o_b||o_{a1}$.

Once o_b has been swapped with o_a and o_a compensated, o_b may again be swapped with any operation to its left in a similar fashion, if that operation also works on a greater index than o_b.

Lemma 2.3 *Any operation o_b may be swapped with the immediately preceeding operation o_a if o_a has a greater index of application. When o_a is replaced by o_{a_1}, which has been compensated as described above, the result will be the same.*

If o_{min_i} is swapped with each operation in L_i – which is possible by taking into account lemma 2.2 and lemma 2.3 – it eventually ends up to the left of the operations of L_i. Each of those operations would need compensation due to working on greater indices than o_{min_i}.

Let

$$t_i = \begin{cases} 1, & \text{if } \textsc{type}(o_{min_i}) = \textsc{insert}. \\ 0, & \text{if } \textsc{type}(o_{min_i}) = \textsc{modify}. \\ -1, & \text{if } \textsc{type}(o_{min_i}) = \textsc{remove}. \end{cases}$$

If we define Lc_i as the same as L_i, except adding t_i to the index of each operation, then applying $L_i||o_{min_i}$ corresponds to applying $o_{min_i}||Lc_i$.

Since the result of applying $o_{min_i}||Lc_i$ corresponds to applying $L_i||o_{min_i}$, no operations in R_i need adjustment.

Lemma 2.4 *If, at the end of iteration i, O_{i+1} is set to $O_i||o_{min_i}$ and S_{i+1} is set to $S_i \setminus \{o_{min_i}\}$, it follows that applying $O_{i+1}||S_{i+1}$ corresponds to applying $O_i||S_i$.*

Proof.

Since at the end of each operation, the operation with the minimum index is transferred from S_{i-1} to the back of O_i, the operations in O_i are ordered by non-decreasing index. At the end of the last iteration, where $i = |S_0|$ and lemma 2.4 has been applied $|S_0|$ times, S_i is empty and O_i contains as many operations as S_0, but in non-decreasing order of index of application, yet yielding the same result. $\square$

When compared to the worst-case cost of applying the operations individually — which is $O\left((l+n)^2\right)$ for n operations on an existing list of length l — it is worth noting that only in the case of an empty list and a long sequence of operations, the reordering algorithm is asymptotically as slow. In all other cases, it is *asymptotically* faster with its $O\left(l+n^2\right)$ time complexity.

It is shown in Section 3.2 that it is practically slower than applying operations indvidually for short enough lists.

2.4 Comparison of Node Copying and Rollback

As shown in the previous sections, the Node Copying and Rollback have different asymptotic time and space bounds.

The following general findings and expectations are derived from the preceeding analyses:

- Node Copying should have constant factor overhead per operation.
- Recalling that Node Copying spends $O(1)$ at retrieving the head node of a version, and Rollback spends $O(d)$, Node Copying is expected to be faster when d is large enough.
- Once the head node has been retrieved, Node Copying has a more time consuming way of navigating to a given index, because it must look through the modifications array of each node along the way, whereas Rollback just needs to follow the *next* pointer field. Therefore, the farther Node Copying has to navigate within a single version, the smaller becomes the advantage of fast retrieval of the head node of that version, when compared to the Rollback approach.
- Rollback should have constant factor overhead per access operation until a large enough number of versions are created. The factor will eventually increase when limit is reached, and as such it is not constant.

CHAPTER 3

Empirical Analysis

In this chapter I will analyse and discuss the experiments I have implemented and run to examine the performance of the different approaches in practice. The purpose of the empirical analysis is to make qualified recommendations as to which of the approaches to use in different usage scenarios.

3.1 Implementation

The implementations are programmed in C++ and conform to the C++11 standard. For compiling the program, the CMake 2.8.9 system has been used together with the GNU C++ compiler.

The source code of the experiment program should be available together with this document. It should also be obtainable from the following URL:

`https://github.com/sirlatrom/persistent-ds-cplusplus/archive/master.zip`

In the abstract class defined in “`AbstractDoublyLinkedList.h`” are declared the common functions that the different implementations should provide to the main experiment program, which is defined in “`main.cpp`”.

Compiling is done by changing into the “`build`” directory, running “`cmake ..`” and then “`make`”.

3.1.1 Execution environment

The empirical analysis is based on the output of executing different implementations on a machine with the following specifications:

CPU	Intel®Core™i5-2400 CPU @ 3.10GHz $\times$ 4
Memory	Hynix/Hyundai 2048 MB DDR3 RAM @ 1333 MHz $\times$ 2
Operating System	Ubuntu 12.10 64-bit

The machine has been put into single-user mode prior to execution, and no other non-operating system processes have run at the same time apart from the login shell.

3.1.2 Implemented operations

I have implemented Node Copying and Rollback for a doubly linked list. The Rollback implementation exists in two variants:

Blackbox Uses a simple, ephemeral doubly linked list for applying each operation between full copy and destination version separately.

Eliminate-Reorder Eliminates superfluous operations using Algorithm 2.1 and then reorders the remaining ones using Algorithm 2.2 before applying them in a single iteration through the underlying ephemeral doubly linked list.

Both variants default to a maximum of 4000 full copies and an initial distance of 65 operations between them. Whenever 4000 full copies have been made, every second one of them is deleted (i.e. the number of full copies then becomes 2000), and the distance is doubled. These figures have been chosen by bisecting within the two-dimensional between them to find the most efficient values for the execution environment. Both variants employ the technique described in Section 2.3.2 for moving the full copies within $\pm\frac{d}{2}$ of their original position.

All implementations support the following operations:

insert(i, d) Inserts an element with data d at index i.

modify(i, d) Modifies the data of the element at index i to d.

remove(i) Removes the element at index i.

These first three represent the usual operations available on a linked list, with bulk operation friendly parameters (conventional linked list implementations take a pointer to a node instead of an index). The difference is that they create a new version of the data structure.

head(v) Returns the head node of the list at version v.

size(v) Returns the size of the list at version v.

These next two are also usually available in a linked list implementation, but these variants take a version number v from which to return the information.

access(v, i) Returns the value of the element at index i in version v.

num_versions$()$ Returns the total number of versions.

These last two are implemented for the convenience of testing bulk usage of the data structure.

Let the operations INSERT(i, d), MODIFY(i, d), REMOVE(i) and ACCESS(v, i) be the ones benchmarked and henceforth be referred to as "the operations".

3.1.3 Implemented usage scenarios

In benchmarking the performance of the approaches in practice, we will look at how they perform under various usage scenarios. We cosider only scenarios concerning at least a sizeable total number of operations (i.e. $N \geq 1000$).

When either of the approaches are to be used in practice, one can imagine different usage scenarios:

Random The operations are executed in random order with no particular pattern, except that if an operation is illegal (such as removing when the list is empty), an INSERT operation is chosen. They may be weighted such that there is different probability for choosing different operations.

In my experiments, the probability is equal (25%) for INSERT, MODIFY, REMOVE and ACCESS operations. Thus, the number of operations of each type will be close to $\frac{1}{4}$ of the total number of operations.

Sequential The different types of operations are executed in sequences.

In my experiments, equally many INSERT, MODIFY, REMOVE and ACCESS operations are made, and in that order. Thus, the number of operations of each type will be exactly a quarter of the total number of operations. The index on which the operations operate on are still randomly chosen unless otherwise specified.

Other usage scenarios could be imagined, including specially tailored worst-case scenarios designed to stress the implementations to their fullest, and scenarios simulating how specific algorithms solving well-known problems, such as planar point location, would behave. However, due to time constraints, such scenarios have been left out.

3.1.4 Program executable arguments

The above scenarios have been implemented in the program `msc`, which accepts the following arguments:

`--count/-c {num}` Total number of operations to carry out (default: 1000).

`--randomize-operations` If passed, applies the operations as described in the Random scenario above – otherwise as in the Sequential secenario (default: off).

`--rollback-eliminate-reorder/-l` Will use the Eliminate-Reorder Rollback implementation (default).

`--rollback-blackbox/-r` Will use the Blackbox Rollback implementation.

`--node-copying/-p` Will use the Node Copying implementation.

`--max-snapshot-dist/-d {num}` Maximum number of operations between full copies (default: 65, applies only to the Rollback implementations).

`--max-num-snapshots/-m {num}` Maximum number of full copies before adaptive fallback is carried out (default: 4000, applies only to the Rollback implementations).

`--head-only/-h` If set, ACCESS operations will work on index 0, i.e. the head of the list, instead of a randomly chosen index.

`--store-results/-s` If set, will store results in an SQLite database file "sqlite.db".

Only the last one specified of the arguments `-p`, `-l` and `-r` is used.

3.2 Time measurements

I have run a series of experiments with various combinations of program arguments in order to determine the time-related performance of each implementation. The same arguments are used 10 times for each of the following total operation counts, which are exponentially spaced between 1000 and 2000000:

1000	2327	5415	12599	29317	68219	158740	369375	859506	2000000

Experiments simulating the Sequential scenario are not run for the last three counts, since with the Rollback implementations they would exceed the available memory of the execution environment, and with the Node Copying implementation they would take exceedingly long to complete.

If the `--head-only` argument is given, the index passed to the ACCESS operations is 0. Otherwise, the index is randomly selected from the range $[0..N[$ where N is the number of elements in the list at the version in question. Effectively, the Eliminate-Reorder implementation should perform worse than the Blackbox when this argument is given, since there is no reordering to be done when all operations work on index 0.

In the following graphs, all data points are averages over 10 runs with identical parameters, each representing the duration spent on the respective operations. For the Rollback variants, it includes fetching the relevant full copy, pre-processing the sequence of operations, and producing and navigating the resulting list. For Node Copying, it includes looking up which persistent node is the head at the version in question as well as navigating to the desired index. The Y error bars indicate ± 1.96 times the standard deviation, i.e. a 95% confidence interval. Data points for the Sequential scenario are marked by boxes, while those for the Random scenario are marked by diamonds. Please note that, unless otherwise stated, the count axis indicates the count of the operation type being discussed, and *not* the total operation count.

3.2.1 Access

For this operation, the time measured is that which it takes to get the head of the list of a random version and then iterating to a randomly chosen index within that list.

3.2.1.1 Random scenario

The following conclusions regarding the Random scenario are evident when examining the diamond-shaped data points plotted in Figure 3.1:

Node Copying is faster than either of the Rollback implementations regardless of `count`. This is most likely due to the constants being lower for Node Copying than for Rollback when accessing the head of a version, and the cost eventually increasing for the Rollback variants.

The Blackbox variant of the Rollback implementation shows to be faster than the Eliminate-Reorder variant. This is most likely due to the list never growing very large, owing to the REMOVE operations occurring with 25% probability mixed between other operations, shrinking the list. When the list is not very long, it is cheaper to iterate the list from the head to carry out the operations rather than running $O(n^2)$ time algorithms on the operation sequences.

The Rollback implementations both show an trend to increase toward the higher end of the count axis, which can be explained by the fact that the distance between full copies will double after operation no. $4000 \times 65 = 2.6 \times 10^5$ and again after operation no. $2.6 \times 10^5 + 130 \times 2000 = 5.2 \times 10^5$ changing operations.

When these events occur, some time is spent on discarding full copies, and afterwards it will take twice as long in expectation to reach a uniformly randomly chosen version.

3.2.1.2 Sequential scenario

The following conclusions regarding the Sequential scenario are evident when examining the box-shaped data points plotted in Figure 3.1:

Up to total operation counts of about 5415 corresponding to approx. $\frac{3}{4}5415 \approx 4061$ *versions, Node Copying is faster than Rollback.* But since longer sequences

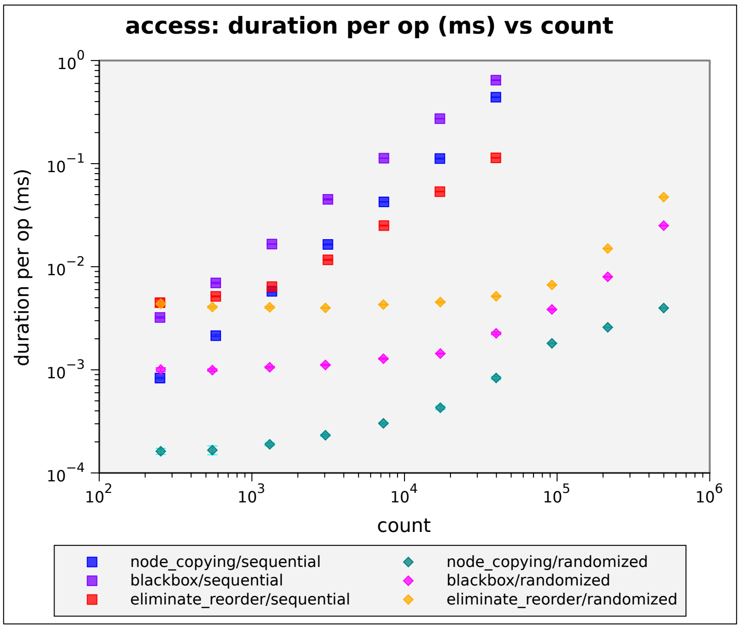

Figure 3.1: Results for ACCESS operations.

of INSERT operations are carried out with higher total operation counts, the average length of the list at a randomly chosen version increases.

Once the head of a given version has been obtained, Node Copying is slower at iterating through the list than Rollback. With large enough data sets, the list will be so long that the cost of iterating the list becomes greater, when compared to Rollback, than the benefit of being faster at getting the head node.

If the Blackbox variant of Rollback could allocate enough memory to complete a greater number of operations, it would likely surpass Node Copying in speed per ACCESS operation around the 10^5 mark.

The time spent on optimizing the operations sequence in the Eliminate-Reorder variant makes it perform worse at ACCESS *operations than the Blackbox variant with low operation counts.* Once enough elements are inserted, it pays off to reorder the operations for a single iteration through the list instead of n potentially full iterations for n operations.

This is the case already at a total operations count of 2327, which corresponds to a list length of at least 582 elements, where Eliminate-Reorder becomes faster than Blackbox. With slightly more than 5415 total operations, corresponding to a list length of at least 1354, it is also faster than Node Copying.

3.2.2 Insert

For this operation, the time measured is that which it takes to insert an element at a randomly chosen index of the list in its most recent version.

3.2.2.1 Random scenario

The following conclusions regarding the Random scenario are evident when examining the diamond-shaped data points plotted in Figure 3.2:

The Blackbox variant of Rollback dominates the other implementations with any operation count. Eliminate-Reorder starts out the slowest, but after 5415 total operations, corresponding to at least 1354 INSERT operations, it becomes faster than Node Copying. Eventually, Eliminate-Reorder and Blackbox are very nearly equally fast.

The simple explanation to this result is that Node Copying is slower than the Rollback implementations at iterating through the list to find the point of insertion. Once enough versions exist with long enough lists, this deficiency will cost Node Copying more than it gains from its fast retrieval of the head node.

3.2.2.2 Sequential scenario

The following conclusions regarding the Sequential scenario are evident when examining the box-shaped data points plotted in Figure 3.2. Note that prior to the `count` ACCESS operations, equally many INSERT, MODIFY and REMOVE operations have been executed, and thus a total of $1 + \frac{3}{4}\texttt{count}$ versions exist.

The Rollback variants are virtually equally fast.

This is because, in contrast to the Random scenario, the most recent version is already available when the INSERT operation is to be applied, and thus that version does not need to be retrieved before the operation can be applied.

Node Copying is slower than both Rollback variants, and increasingly more so as more INSERT *operations are carried out.* This result shows most clearly how,

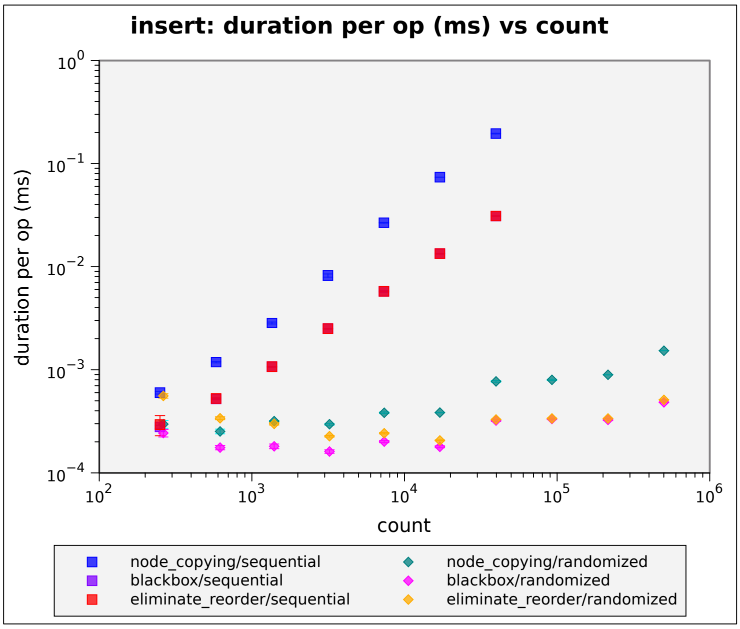

Figure 3.2: Results for INSERT operations.

as pointed out before, Node Copying is slower at iterating through the list to the point where the changing operation is supposed to take place.

3.2.3 Other operations

The MODIFY and REMOVE operations show virtually the same results as the INSERT operation, since the outcomes mostly depend on how long the list gets in the versions created — a figure which depends primarily on the number of INSERT operations.

The graphs are nevertheless included in Figure 3.3 for comparison.

3.2.3.1 Total duration

When looking at the total duration, i.e. the time from start to finish of the entire scenario, it turns out that Node Copying is the fastest — see Figure 3.4 — but this depends largely on the chosen probabilities/ratios of each operation.

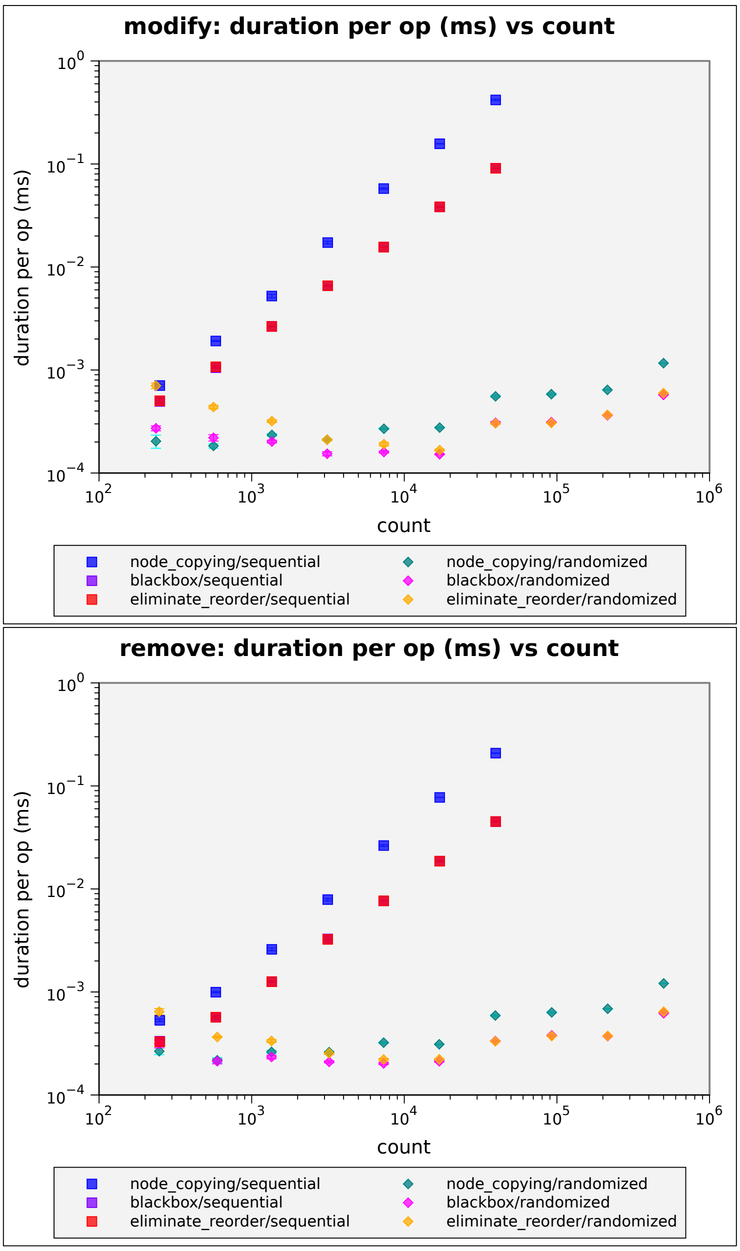

Figure 3.3: Results for MODIFY and REMOVE operations.

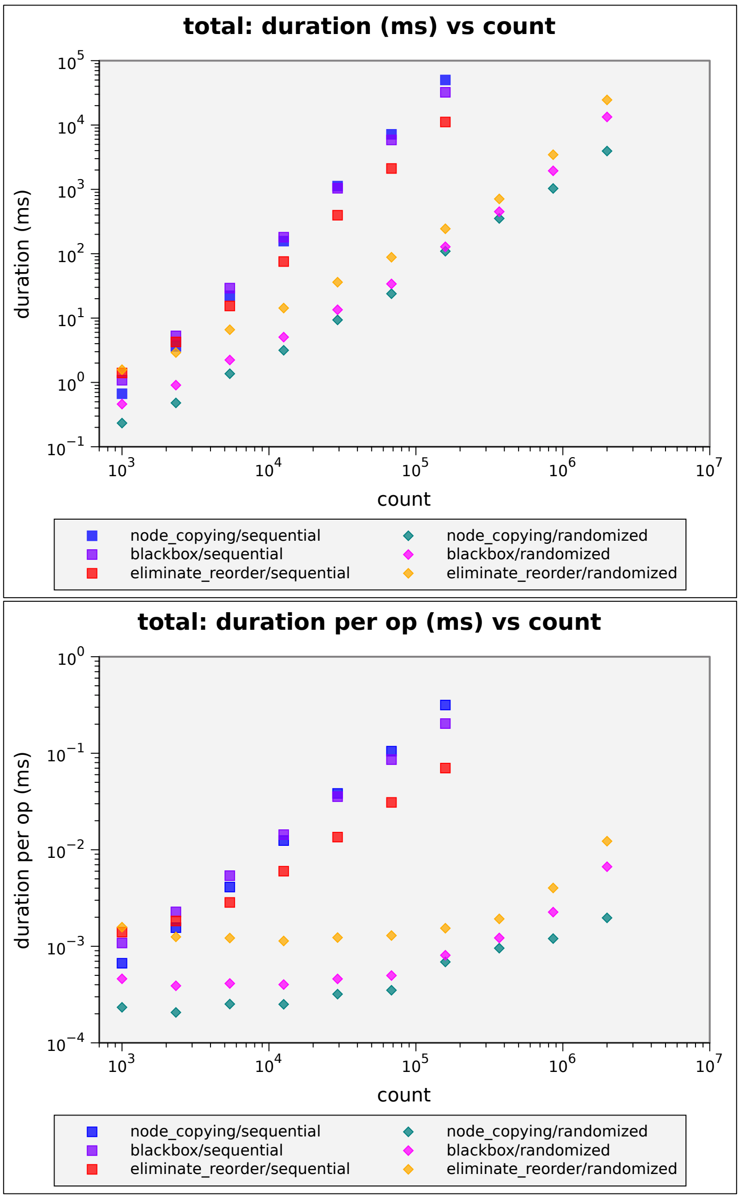

Figure 3.4: Total duration and duration per operation across all the operations of one experiment for each batch size.

3.3 Space estimates

In order to estimate the memory usage of the different implementations, preprocessor directives have been introduced which control whether time or measurements are made. If the `MEASURE_SPACE` symbol is defined, no lines of code which measure the time of operations are compiled or executed. Instead, code lines are introduced which estimate the memory usage.

The memory usage for an instance of the Rollback implementation with F full copies and N total operations is estimated according to the following formula:

$$\begin{aligned}
\text{total\_space} \quad = \quad & \textsc{size\_of}(\text{ephemeral node}) \times \sum_{i=1}^{F} \textsc{size}(\text{full copy}_i) \\
& + N \times \textsc{size\_of}(\text{operation record}) \\
& + F \times \textsc{size\_of}(\text{full copy record}) \\
& + \textsc{size\_of}(\text{auxillary DS})
\end{aligned}$$

The total memory reserved by the program when using Eliminate-Reorder for large data sets is measurably smaller when observed with OS utilites, than when using Blackbox. This is because fewer nodes are allocated which would be deleted again as part of getting form version $v_current$ to version v_x.

For Node Copying, the estimation is more accurate, given that every time a new persistent node is created, either due to an INSERT operation or due to a copy being made as described in Section 2.2.1, a counter is incremented by the size of a persistent node. The size of the auxillary data structure indicating which node is the head in each version is also estimated.

In Figure 3.5 it is clearly visible that, as expected, Rollback uses significantly more space than Node Copying in the Sequential scenario — and consistently more so in the Random scenario.

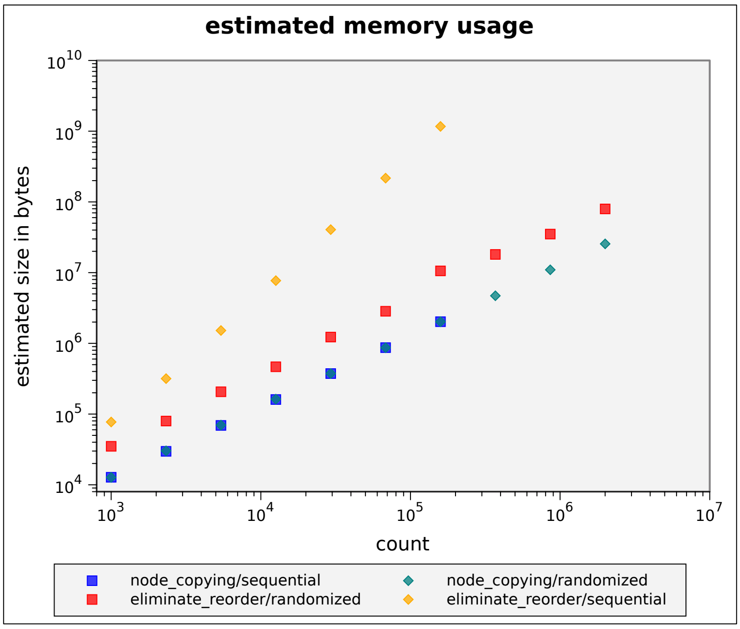

Figure 3.5: Estimated memory usage.

Chapter 4

Conclusion

I have analysed and compared two approaches to making a data structure partially persistent.

The two approaches have their strengths and weaknesses. As identified in the Method chapter and shown in the Empirical Analysis chapter, they yield different results under different usage scenarios.

If sufficient programming time is available for implementing an optimized Rollback approach, it is the recommended option when dealing with large numbers of operations in a manner similar to the Sequential scenario — provided that sufficient system memory is available. This is especially the case if many ACCESS operations are expected to be made which are followed by navigation far into the produced version.

On the other hand, a data structure which is not suitable for optimizations such as those which went into Eliminate-Reorder, could still be made partially persistent with reasonable effort using Node Copying, and the results would be better than when using Blackbox Rollback.

If the effective sizes of the data structure in the various versions do not become too large, it is recommended to apply the Node Copying approach, unless very few ACCESS operations are expected.

4.1 Future work

It could be investigated whether compression techinques could be applied when storing the full copies and/or the operations log in the Rollback implementations. If the benefit in terms of lower memory usage is great enough, it would allow working with larger data sets than with the implementations used in this thesis.

More advanced data structures, such as binary search trees, could be implemented to see whether the same general conclusions apply, or if they are specific to a doubly linked list. Notably, it would be interesting to see if eliminating superfluous operations or optimizing the order of the operations sequence or other such optimizations are possible and/or feasible for more advanced data structures.

Caching techniques or other optimizations could be employed to secure faster navigation of the underlying data structure with Node Copying.

More elaborate usage scenarios could be implemented and tested using the existing framework. E.g. version access patterns showing how well-known algorithms would perform, such as planar point location. Different probabilities in the Random and Sequential scenarios could also simulate different usage patterns.

It could be investigated whether approaches from different paradigms could effectively provide partial persistence and be compared to the existing implementations.

It might also be worthwhile to find efficient ways of converting between the two approaches such that if conditions change, the one which is most efficient can be used.

Bibliography

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86 – 124, 1989.

[TK95] Vassilis J. Tsotras and Nickolas Kangelaris. The snapshot index: An i/o-optimal access method for timeslice queries. *Information Systems*, 20(3):237 – 260, 1995.

[VL87] Vinit Verma and Huizhu Lu. A new approach to version management for databases. *Managing Requirements Knowledge, International Workshop on*, 0:645, 1987.