Incremental Index Structures

Master's Thesis

Paul Theodor Pyl

Freie Universität Berlin – Bioinformatik

24.05.2009

Erstprüfer:

Prof. Dr. Knut Reinert

Freie Universität Berlin

AG Algorithmische Bioinformatik

Zweitprüfer:

Dr. Gunnar W. Klau

CWI Life Sciences and NISB

Algorithmic computational biology

Contents

1. Introduction

1.1 Motivation

In the past years, the rapid development of DNA sequencing technologies has continuously made the process of producing sequence data from the genomes of various organisms exponentially faster and cheaper. As a result, the amount of data available for analysis has increased tremendously as well. In particular, next-generation sequencing technologies (NGS) like the Illumina Genome Analyzer continue to contribute to this trend.

To illustrate this development, consider the time it took to sequence the first human genome: the Human Genome Project (HGP)(Venter *et al.* , 2001), generating costs of over $3 billion, took 13 years (1990–2003) to complete. Nowadays, equivalent amounts of sequence data can be generated in less than a month's time, and this service can be offered commercially for less than $\$50,000$ (Illumina Inc., 2009).

This has a palpable benefit for the public: extrapolating from the development of the last twenty years, we expect the realization of truly personalized medicine through the availability of individual whole-genome analysis to large portions of the population.

Obviously, the methods for analyzing this data have to keep pace in terms of affordability and speed.

NGS generate the genomic sequence data in the form of relatively short subsequences (about 30 to 300 bases, compared to the 3 bio. bas). Those subsequences are called *reads*. To recover the complete base sequence of a new genome from those reads, we can use a reference genome (for example the resulting sequence of the HGP). This process relies heavily on the close relationship between members of the same species, caused by the high similarity of their genomic sequence.

If, for a sufficiently high number of reads, we can find their corresponding subsequences in the reference genome, we can then reconstruct the new genome by assembling those correctly positioned reads. Note that, since there will be differences between the sequences, not all reads can be matched to the reference without error.

The general concept of read-mapping is illustrated in **Figure 1.1**.

Many, if not all, tools for genome analysis rely on the quick identification of *substrings* (or subsequences) within large data-sets. The development of the algorithms and data structures to make meaningful analyses of the increasing number of genomic sequences available is handled in the field of bioinformatics. For the task of finding substrings within genomic sequences, so-called *substring indices* have proven useful and are now a widely used means of providing a fast method for finding substrings within large datasets.

A substring index is a data structure that stores the positions of all substrings of a given sequence. It allows the efficient retrieval of the positions of all occurrences of any substring

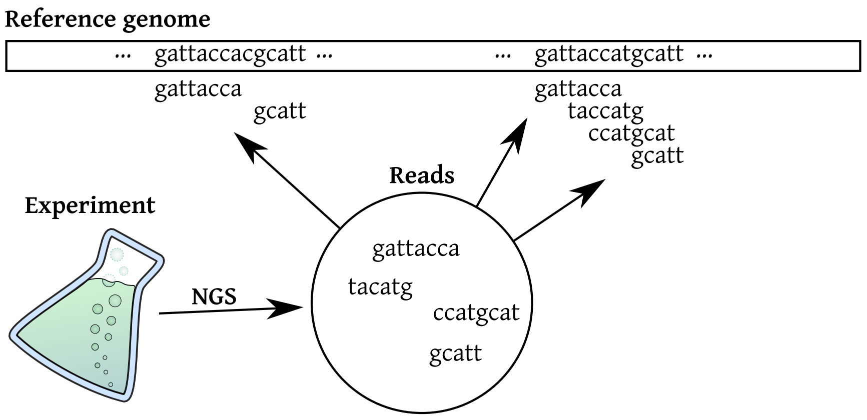


Figure 1.1: The general concept of read-mapping.
Note how the reads are determined experimentally and then matched to the reference genome.

within the sequence. The construction of such a substring index, however, is a time-costly process. The initial investment in terms of computation time has to be amortized by the time saved through the fast searching capabilities provided by the index.

There are many applications for substring indices in bioinformatics, in particular the aforementioned process of read-mapping: An index is created for the reference genome, and is used to locate the subsequences corresponding to individual reads. To compensate for inexact matches between reads and the reference, we cannot search directly for the reads, since the substring index only finds exact matches. Instead, we can search for substrings of the reads (some of which will match exactly). Thus, we can identify regions in the reference where the read matches with few errors. **Figure 1.2** illustrates the usage of substring indices in read-mapping.

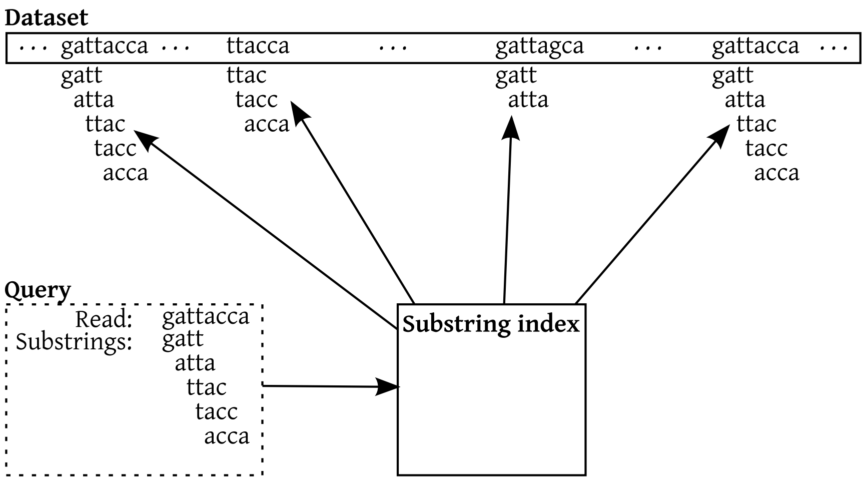


Figure 1.2: Example of the application of substring indices to read-mapping.
Note how the substring index returns the position within the text for each substring. In this way we can find the read exactly or with a certain amount of errors, depending on the number of matching substrings in an area.

In this way we can use known genomes as templates to assemble the reads of a new genome if the two organisms are related closely enough. For example, we can map the reads of the genome of a newly discovered plant to the *annotated genomes* of other plants. An annotated genome consists of a genomic sequence and the information about the positions of known genes. We can use this information to identify the possible positions of related genes on the newly assembled genome.

Genomic sequence data is still subject to change, whenever new scientific research leads to a revision, for example of the human genome. We then have to deal with the problem that the substring index build over the last genome revision is now no longer usable with the new revision. When a new version of the indexed sequence is released we have to recompute the index. The initial investment of computation time for creating the index of the previous revision is lost.

Most of the substring indices used today are static: they cannot adapt to changes in the sequence and must be recomputed. Some work has been done in the area that resulted in the development of *dynamic substring indices*, for example by Salson *et al.* (2009b; 2009a). Those dynamic index structures adapt to changes in the text.

In this thesis we will explore means to avoid the re-computation of a substring index when the text is modified. We will work with an *extended suffix array* as the index structure, and develop an algorithm for adapting the index to the changed text. We will avoid the time-consuming index creation and instead use modifications to the index that make it usable with the revised sequence.

In the situation where a new revision of the human genome is published, the problem of distributing the sequence data of the new revision arises. Full-genome sequence data can be several gigabytes in size and therefore cannot easily be obtained via email or by downloading it from a server, even if compression is used.

For a new revision of a genome the number of differences to the last revision will likely be small in relation to the sequence length. It would be desirable to have a means of encoding only those changes (see **Section 1.4.2** and Christley *et al.* (2009) for more information). This would make it easy to get the relevant data about the new revision without having to download the whole sequence. The sequence data of the new revision could then be obtained by applying the relevant changes to the previous revision.

Unfortunately the modification of large sequences that are stored continuously in memory is a time costly task. In a worst-case scenario, the deletion or insertion of a single position in the sequence may require us to move almost the whole sequence in memory (**Figure 1.3**).

Position	0	1	2	3	4	5	6	7
Value	g	a	t	t	a	c	c	a

$\rightarrow$

Position	0	1	2	3	4	5	6	7
Value	g	t	t	a	c	c	a	–

Figure 1.3: Deleting a value from a continuous string.
A deletion of the value at position 1 from the sequence “gattacca” results in moving all values at positions ≥ 2 by one position to the right.

To avoid moving that many positions in memory we will develop a data structure that stores information about changes made to a sequence without actually applying those modifications. We will use the data structure to allow for random and sequential access operations to work as if the changes were already applied to the sequence. This data structure will allow us to interact with the string as if it were modified, without actually having to perform the modifications in memory.

We call this process of storing changes rather than applying them directly *journaling*. We call our data structure storing the modifications to a sequence T the journal J of T. **Figure 1.4** illustrates the general usage of the journal for the deletion of a single value from the text "gattacca". The tables show the return values of accessing the sequence and the journal in each column. The corresponding positions are given in the first row. We can see that the sequence is unchanged while the journal behaves as if the modification was already applied.

Position	0	1	2	3	4	5	6	7
Sequence	g	a	t	t	a	c	c	a
Journal	g	a	t	t	a	c	c	a

$\longrightarrow$

Position	0	1	2	3	4	5	6	7
Sequence	g	a	t	t	a	c	c	a
Journal	g	t	t	a	c	c	a	–

Figure 1.4: Example for journaling a deletion.
The deletion of the value at position 1 from the sequence "gattacca" is stored in the journal. Note that the sequence is not modified, but the journal returns the correctly modified data. The journal only stores the information for the gray column.

In this thesis we will use the SeqAn C++ library (Döring *et al.* , 2008) to create a journaling data structure that stores modifications to a text and behaves in the way described above. We will also develop an algorithm for synchronizing an extended suffix array (see **Section 1.2.0.8**) to a modified text when the changes have been stored in a journal.

In the next sections the problems mentioned above will be defined in more detail and we will have a look at the scenario in which those problems normally arise. Once we have understood the problems, we can formulate the specific goals of this thesis. We will also have a look at related work by other groups.

In **Section 2** we will define and explain the underlying concepts of this thesis. This will include the layout of our algorithm for synchronizing an index to a changed text and the design of a data structure that stores sequences and uses journaling. We will continue by providing information about the implementation of the algorithm and the journaling data structure in **Section 3**.

Section 4 contains the results of benchmarks comparing our journaling data structure to conventional means of storing sequences. We also compare our synchronization algorithm to a state-of-the-art index creation algorithm.

A discussion of the benchmark results will be given in **Section 5**. Possible improvements and further research will be discussed in **Section 5.3** where we will provide examples for the applicability of our results.

1.2 Problem Definition and Notation

Before we define the problems that need to be solved, we will firstly consider the scenario in which those problems arise and define some notations used throughout this thesis.

Since we will deal with problems concerning the modification of texts and their indices, this section will give a brief summary of the scenario to which the contents of the thesis will be applied. In our scenario we use a substring index to perform fast searches on a large text.

1.2.0.1 The Text

In this thesis a text (also referred to as a "sequence", "string" or "dataset") is simply a collection of values that are in some defined order. We formally define a text in the following way:

Definition 1 (A text T over an alphabet Σ)**.**
Given a sorted alphabet Σ a sequence $T \in \Sigma^*$ of values from this alphabet is called a text. For a text of length $|T| = n$ the value at position k (with $0 \leq k < n$) in the text is defined as $T[k]$.

Since we will deal with the bioinformatical application of indices our text will normally consist of genomic sequence data which is comprised of elements from the alphabet $\Sigma := \{A, C, G, T, N\}$ where N is the wildcard character which represents any of the other characters. To allow for fast searching of substrings in the text an index structure is build and we will now consider the general concept of such an index and start by defining classes of substrings.

1.2.0.2 Suffixes, Prefixes and Infixes

To better understand the structure of substring indices we have to define basic classes of substrings first. Those basic classes are *suffixes*, *prefixes* and *infixes*.

The substring T' ranging from positions k to l (inclusively) in the text T is written as $T[k \ldots l]$ where $0 \leq k \leq l < n = |T|$ holds.

Definition 2 (Prefix)**.**
A prefix P_l of text T is defined as the substring $T[0 \ldots l]$.

Definition 3 (Infix)**.**
An infix $I_{k,l}$ of text T is defined as the substring $T[k \ldots l]$.

Definition 4 (Suffix)**.**
A suffix S_k of text T is defined as the substring $T[k \ldots n-1]$.

For a text T of length $|T| = n$ and its suffixes, prefixes and infixes the following equalities hold:

$$T = P_{n-1} = I_{0,n-1} = S_0$$

The different kinds of substrings and their relation to each other are shown in **Figure 1.5**.

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Sequence T	g	a	t	t	a	c	a	g	c	a	a	c	g	c	t
Suffix S_4					a	c	a	g	c	a	a	c	g	c	t
Infix $I_{4,9}$					a	c	a	g	c	a					
Prefix P_9	g	a	t	t	a	c	a	g	c	a					

Figure 1.5: Example of a suffix, prefix and infix of the text T.
Note how the infix $I_{k,l}$ is also a prefix of the suffix S_k and a suffix of the prefix P_l where $k = 4$ and $l = 9$. The common part of the three substrings is marked gray.

1.2.0.3 The Index

An index over a text is a data structure that allows for efficient searching of substrings in the text. It contains the positions of all substrings that occurr in the text.

Ideally the time consumption for searching with the index should be much smaller than that for searching via any other means (e.g. brute-force etc.). To achieve this kind of fast searching, a certain amount of time has to be invested in the construction of the index that will then be amortized by the reduced search time compared to other methods. Consequently there are application scenarios where an index is not the most efficient approach.

If only very few searches have to be performed it is possible that searching via some smart pattern matching algorithm will yield the desired results faster than an index. In fact the index construction might take longer than the time needed to find the desired substrings via the pattern matching algorithm.

Figure 1.6 illustrates the basic concept of finding a pattern in a text via a substring index.

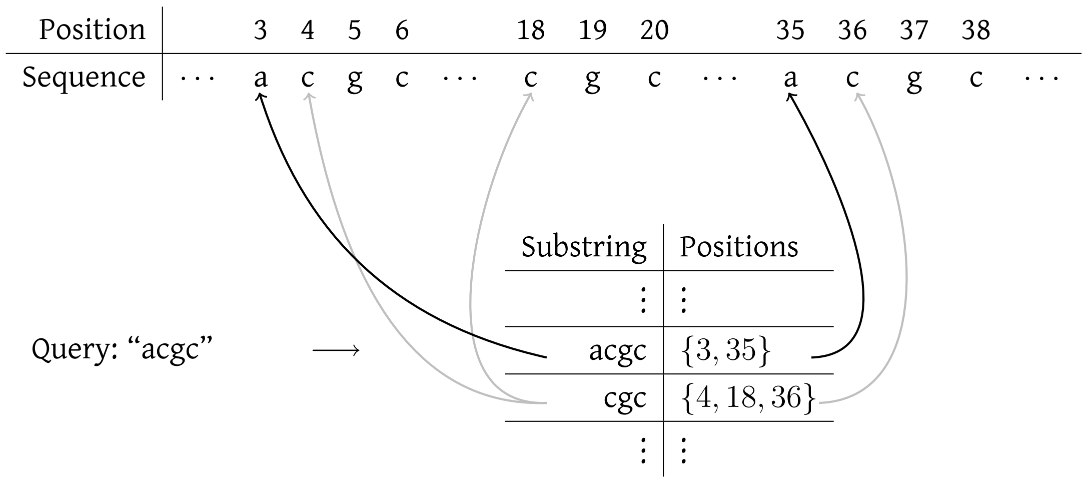


Figure 1.6: Using indices for substring searches.
The index is generated in a preprocessing step and queries for a given pattern are directed to the index which contains the data about all the occurrences of substrings in the text.

We will deal with an *extended suffix array* (ESA) as our substring index. The ESA consists of a suffix array, a lcp-table and the inverse suffix array. We will now proceed by defining those components of the index.

1.2.0.4 The Suffix Array

A suffix array is an index over a text. It was first introduced in 1993 by Gene Myers and Udi Manber in their paper **Suffix arrays: a new method for on-line string searches** (Manber & Myers, 1993).

The suffix array is based on the concept, that every infix of T must also be a prefix of a suffix of T. When we search for a pattern p in the text we normally expect this pattern to occur in the form of one or more infixes of T. The substring $T' = T[k \dots l]$ that ranges from positions k to l in the text (with $0 \leq k \leq l < n$) is also contained in the suffix $S_k = T[k \dots n-1]$ of T as a prefix of this suffix. This prefix of S_k is ranging from the positions k to l in the text which translate to the positions 0 to $l-k$ in the suffix. It is the infix $I_{k,l}$ of T and the prefix P_{k-l} of S_k, see **Figure 1.5** for an illustration.

Considering the fact that each infix of T is the prefix of a suffix of T we can now see that a lexicographical ordering of all suffixes of T implies a lexicographical ordering of all infixes of T. The information about the ordering of all infixes of T is contained in the suffix array.

The suffix array contains this lexicographical ordering and constitutes a sorted array on which a binary search for a given substring (the pattern p) can be performed. This search will return the suffixes from the text that have the pattern p as their prefix. It is described in more detail in the next section.

Ideally the result of this search will be an interval in the suffix array containing all the suffixes of which p is a prefix and therefore all the positions at which p occurs in the text T. The general concept of a suffix array as an index structure is illustrated in **Figure 1.7**

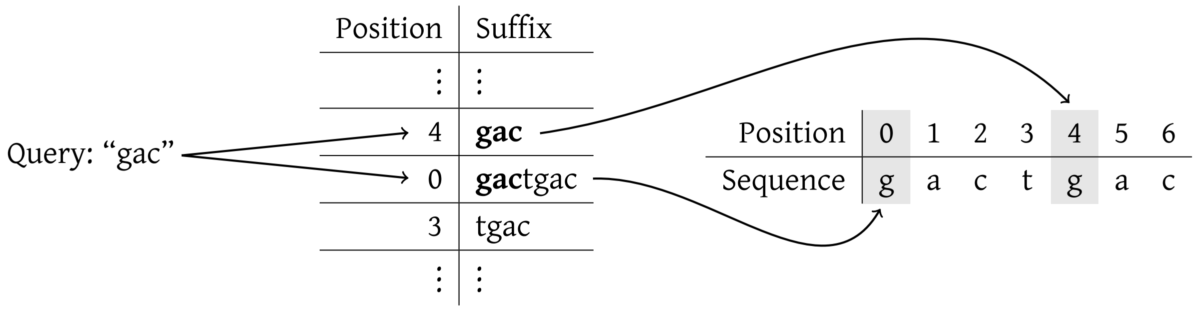


Figure 1.7: Concept of the suffix array as an index structure. The Suffix array contains the lexicographical ordering of all suffixes in the text, the suffixes that have the pattern as a prefix are adjacent in the suffix array and their starting positions in the text correspond to the occurrences of the pattern. For the pattern "gac" the relevant suffixes are marked **bold** in the SA and the corresponding positions in the text are in the gray columns.

Note that a special letter $\$ \notin \Sigma$ must be defined that is by definition lexicographically smaller than all other values from Σ. This is done to ensure the comparability of suffixes of different lengths. For $\$$ the following holds:

$$\exists \$ \notin \Sigma \, \forall q \in \Sigma : \$ < q$$

In this way we can guarantee, that a suffix S_i that is a prefix of another suffix S_j is lexicographically smaller than S_j. Consider the suffixes S_3 and S_1 in the example given in **Table 1.1** and **Table 1.2** where the suffixes and suffix array of the text "acbc" are given.

T	acbc
S_0	acbc$
S_1	cbc$
S_2	bc$
S_3	c$

Table 1.1: Suffixes of the text "acbc"
Note that the terminal $ is not part of the original text but is added when generating the suffixes.

Position	Suffix
0	acbc$
2	bc$
3	c$
1	cbc$

Table 1.2: Suffix array of the text "acbc".
Note how the suffix "c$" is lexicographically smaller than "cbc$" because of the terminal $ character.

When we search the suffix array for a substring (e.g. "c") we know that all suffixes that begin with this substring are adjacent to each other in the index. The beginning and end of the interval of suffixes that share the substring we look for as a common prefix can be determined via binary search on the suffix array.

1.2.0.5 Searching in the Suffix Array

Since searching the suffix array for a pattern is the crucial task when we want to find occurrences of this pattern in the text we will now have a closer look at the way this search is performed.

As we already discussed in the section above the key algorithm is a binary search on the sorted array of suffixes. A binary search makes use of the knowledge that the array is sorted to improve the running time.

Because we will be comparing a pattern of length m with suffixes of different lengths that can be smaller or larger than m we will define some relations for this task first:

Definition 5 (Prefix of length m).
For a given sequence $p \in \Sigma^*$ and $m \in \mathbb{N}$ we define the following function:

$$\text{prefix}_m = \begin{cases} p & \text{if } |p| \leq m, \\ p[0 \ldots m-1] & \text{if } |p| > m. \end{cases}$$

Definition 6 (Comparing strings of different lengths).
Let $p, q \in \Sigma^*$ and $m \in \mathbb{N}$ then we define the relations $<_m$, $=_m$ and $>_m$ in the following way:

$$\begin{aligned} p <_m q &\Leftrightarrow \text{prefix}_m(p) < \text{prefix}_m(q) \\ p =_m q &\Leftrightarrow \text{prefix}_m(p) = \text{prefix}_m(q) \\ p >_m q &\Leftrightarrow \text{prefix}_m(p) > \text{prefix}_m(q) \end{aligned}$$

Tho composite operations $\leq_m$ and $\geq_m$ can be modeled as $<_m \wedge =_m$ and $>_m \wedge =_m$.

The binary search is performed by taking three suffixes from the suffix array as the high, low and pivot element respectively. The *pivot* element is the one we will use to determine the sub-interval in which to continue the binary search.

By comparing the pattern to the pivot element we can decide in which half of the interval between high and low element the pattern might be found and we can adjust the elements accordingly and repeat the process until the pattern is found.

In this way we can find all occurrences of a pattern p of length $|p| = m$ in a text T of length $|T| = n$ in $\mathcal{O}(m \cdot \log n)$ time. This is because the text contains n suffixes and performing a binary search on their lexicographical ordering will result in checking $\log n$ suffixes. For each suffix which we have to compare to p we perform at most m character comparisons in the worst case, hence the running time of $\mathcal{O}(m \cdot \log n)$ is needed.

Since all suffixes containing p as a prefix are adjacent in the suffix array we would like to find the first and the last of those suffixes thereby defining an interval in the suffix array that contains all occurrences of p. We call this interval $S[l \dots h-1]$ with the lower boundary l and the upper boundary h. Note that we define h as the index of the element directly behind the interval in the suffix array. Finding those boundaries is done via binary searches that differ in their criterion for choosing the sub-interval for the recursion and is shown in **Algorithm 1**. Here we define the algorithms for finding the lower boundary l and the upper boundary h of the pattern-containing interval $S[l \dots h-1]$ in the suffix array S of text T ($|S| = |T| = n$).

Algorithm 1 We use binary search to compute the upper and lower boundary of the pattern-containing interval in the suffix array. Pattern p is contained in $S[l_1 \dots h_1 - 1]$. Note that l_1 points to the first position of the interval or to the correct insertion position if p is not contained in S. h_1 points directly behind the interval or to the correct insertion position if p is not contained in S. If $l_1 = h_1$ we know that the interval has length 0 and p is not part of T.

```
1: $l_1 \leftarrow 0$
2: $l_2 \leftarrow n$
3: while $l_1 \neq l_2$ do
4:     $piv \leftarrow \lfloor \frac{l_1+l_2}{2} \rfloor$
5:     if $T[S[piv] \dots n-1] <_{|p|} p$ then
6:         $l_1 \leftarrow piv + 1$
7:     else
8:         $l_2 \leftarrow piv$
9:     end if
10: end while
11: return $l_1$
```

```
1: $h_1 \leftarrow 0$
2: $h_2 \leftarrow n$
3: while $h_1 \neq h_2$ do
4:     $piv \leftarrow \lfloor \frac{h_1+h_2}{2} \rfloor$
5:     if $T[S[piv] \dots n-1] \leq_{|p|} p$ then
6:         $h_1 \leftarrow piv + 1$
7:     else
8:         $h_2 \leftarrow piv$
9:     end if
10: end while
11: return $h_1$
```

We choose three elements from the array, a low, high and a pivot element. The low and high element constitute an interval on the array of which the pivot element is a part, we chose this as the middle element of that interval. We then compare our pattern with the pivot element, and depending on whether it is smaller or bigger we repeat the process in the interval from low to pivot element or from pivot element to high element, respectively.

In a sorted array, if our pattern is larger than the pivot element, we know that it cannot be found anywhere before the pivot element in the array, since all suffixes that lie before the pivot element are smaller or equal to it.

When searching for a pattern p in a suffix array S with the low element $S_{pos_{low}}$ (the $S[pos_{low}]$-th suffix), the high element $S_{pos_{high}}$ (the $S[pos_{high}]$-th suffix) and the pivot element $S_{pos_{pivot}}$ (the $S[pos_{pivot}]$-th suffix) the following holds:

$$0 \leq pos_{low} \leq pos_{pivot} \leq pos_{high} < |S| = n$$
$$\forall_{0 \leq k < pos_{pivot}} : S_{pos_{pivot}} \geq S_k$$
$$p >_{|p|} S_{pos_{pivot}} \Leftrightarrow \forall_{0 \leq k < pos_{pivot}} : p >_{|p|} S_k$$

After comparing the pattern to the pivot element we know in which half of the interval between high and low element the pattern might be found. We then make the old pivot element the respective new high or new low element and choose a new pivot element from the new interval.

When the high and low element are the same we have found the respective border of the interval in S that contains the pattern. If the upper and lower boundary of the interval are the same we know that p is not contained in the text.

1.2.0.6 The LCP-Table

The lcp-table is an auxiliary table to the suffix array and provides information about the **l**ongest **c**ommon **p**refix of two adjacent suffixes, an example is illustrated in **Table 1.3**.

Position	Suffix	lcp-Value
0	acbc$	0
2	bc$	0
3	**c**$	1
1	**c**bc$	0

Table 1.3: suffix array and lcp-table of the text "acbc", the common prefix of suffixes S_3 and S_1 is marked as **bold** text.

The entries of the lcp-table contain for each suffix the length of its longest common prefix with its successor in the suffix array. This allows for faster searching within the suffix array. Generally the lcp-value of two strings is simply the length of their longest common prefix.

When we have the information about the longest common prefix of the adjacent suffixes in the suffix array we can use this information to reduce the time consumption of the search for a pattern p.

Definition 7 (lcpValue(s, s')).
Given a sorted alphabet Σ and the two sequences $s, s' \in \Sigma^*$ we define the function

$$\text{lcpValue}(s, s') \mapsto \mathbb{N}$$

to return the length of the longest common prefix of s and s'.

To understand how this performance improvement is achieved we have to regard the special properties of the lcp-table. When performing the binary search in the suffix array it will occur after some iteration steps, that the low and high element have a longest common prefix value that is grater than 0. This means that they have some characters at their beginning in common. This also implies that in the interval between the low and high element in the suffix array there can be no suffix that does not share those first q characters otherwise the suffix array would not be in lexicographical order.

#	Position	Suffix	lcp-Value
768	12	ccagactata...	2
769	324	**cct**aca...	5
770	1022	**cct**acct...	4
771	1312	**cct**attata...	3
772	57	**cct**cttg...	3
773	777	cctgcata...	2
774	212	ccgactaca...	7

Table 1.4: An interval in a suffix array showing the relation between suffix array and lcp-table. Note that this only represents a part of a larger suffix array. The field # denotes the index of the suffix within the suffix array while the field "Position" gives the position of the suffix in the text. The common prefix of all suffixes between positions 769 and 773 in the suffix array are marked as **bold** text.

Consider the example given in **Table 1.4**. Here we can see that all suffixes in the interval from position 769 to position 773 in the suffix array have a common prefix of length 3 which is also the lcp-value of the 324-th and 777-th suffix.

It is immediately clear that for two arbitrarily chosen suffixes S_k and S_l from the suffix array there can exist no suffixes in the interval defined by them that do not have the first lcpValue(S_k, S_l) characters with both suffixes in common.

If there were such a suffix the suffix array would not be a lexicographical ordering. Consider the example from **Table 1.4** and imagine there would be an additional suffix inserted in the interval that had only its first character in common with the others, for example the suffix $cgt\$$.

Now the suffix array would not be in lexicographical order since the suffix $cgt\$$ cannot be smaller than one suffix starting with cct and at the same time larger than another suffix that is also starting with cct. It simply cannot be inserted between suffixes that have a longer common prefix than the inserted suffix has with them.

This knowledge about the common prefix of suffixes within an interval in the suffix array can be used to reduce the time consumption of searching for a pattern p of length $|p| = m$ from the former $\mathcal{O}(m \cdot \log n)$ to $\mathcal{O}(m + \log n)$.

This is achieved by estimating the lcp-value of the pattern and the pivot element which allows us to skip the first lcpValue(p, pivot) values because they are not important for determining the lexicographical order of the two. This method of improving the running time for searching the suffix array by using lcp-values is described in the paper by *Gene Myers* and *Udi Manber* (Manber & Myers, 1993).

Another noteworthy result of this property of the lcp-table is, that the lcp-value of two arbitrarily chosen suffixes p and q is equal to the minimum of the interval in the lcp-table between those two suffixes. In our example from **Table 1.4** we can see that the minimum of the interval in the lcp-table between the 324-th and the 777-th suffix is 3 which is also their lcp-value (calculated as $min(\{5, 4, 3, 3\}) = 3$).

The lcp-value of the last suffix is not considered for this minimum since it describes that suffixes relation to its successor in the suffix array, which is not part of the interval.

1.2.0.7 The Inverse Suffix Array

The inverse suffix array is used to efficiently find for each suffix its position within the suffix array. It is needed since the suffix array only provides information about the lexicographical order of the suffixes of T but is does not allow us to find the exact position of a specific suffix within this order directly. We would have to use binary search with the suffix sequence as pattern to find the position of a suffix in the suffix array (or simply iterate the suffix array and compare the positions).

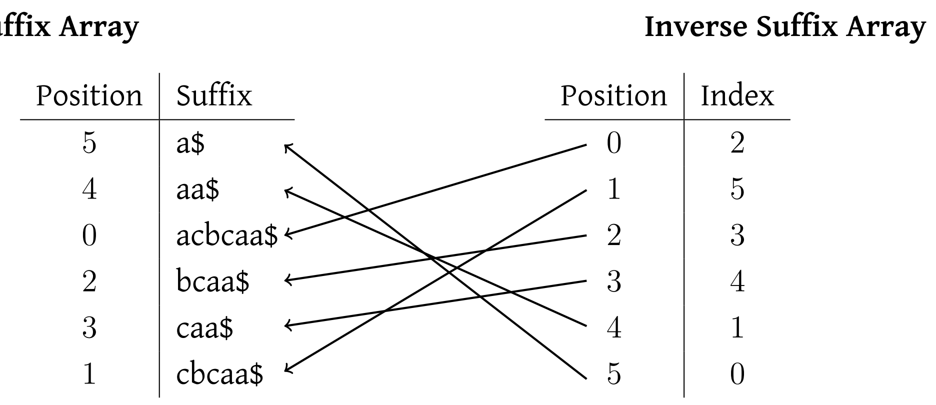
For a suffix array $suftab$ and an inverse suffix array $sufinv$ over a text T of length n the following relation holds:

$$\forall i \in 0 \ldots n-1 : \text{sufinv}[\text{suftab}[i]] = i$$

This means that the inverse suffix array contains at its i-th position the position of the i-th suffix within the lexicographic ordering of the suffixes of T which is its position within the suffix array.

An illustration to clarify the relation between suffix array and inverse suffix array is given in **Figure 1.8**.

The inverse suffix array is not necessary for finding patterns in the text, but it greatly improves the running time of the synchronization algorithm as will be shown in **Section 3**.



Suffix Array

Position	Suffix
5	a\$
4	aa\$
0	acbcaa\$
2	bcaa\$
3	caa\$
1	cbcaa\$

Inverse Suffix Array

Position	Index
0	2
1	5
2	3
3	4
4	1
5	0

Figure 1.8: Suffix array and inverse suffix array of the text T = "acbcaa". The first number is the starting position of the corresponding suffix, the inverse suffix array stores for each starting position the position within the suffix array.

1.2.0.8 The Extended Suffix Array

An extended suffix array is an index structure consisting of the suffix array of a text and its lcp-table. Within this thesis the inverse suffix array will also be considered to be part of the extended suffix array. It is a suffix array that is extended by some auxiliary tables (lcp-table and inverse suffix array in our case) to improve the performance in searching and synchronizing.

1.2.1 Summary of the Scenario

As already mentioned **Section 1.1**, the situation which is of interest for this thesis is that of a text, that is subject to minor changes over time, for example a genomic sequence that is updated according to new scientific data.

We have an extended suffix array containing a suffix array, an inverse suffix array and a lcp-table as an index structure over such a text.

It is desirable to conceive a way of adapting the index to the minor changes made to the text, rather than recomputing the whole extended suffix array. Such an index structure will be called incremental, because as the text changes incrementally so does the index, adapting to the modifications of the text. Ultimately we will design an algorithm to synchronize an extended suffix array with a changed text, thereby making our extended suffix array an incremental index structure.

The following sections will discuss the formalization of problems that have to be overcome in order to achieve the goals of this thesis (which will further be specified in **Section 1.3**).

1.2.2 Efficient Modification of Strings

For the implementation of an incremental index structure a certain amount of infrastructure is needed to ensure feasibility of the approach. The problems concerning performance become more clear, when we consider the necessary operations that will be performed on the text and the index (the suffix array and the lcp-table).

Obviously the modifications of the text will constitute insertions, deletions and replacements of substrings. Since our goal is to provide a means to adapt the index to minor changes in the text, these changes will translate directly to modifications in the index that are of the same type of operation.

As we have already hinted at in **Section 1.1** and **Figure 1.3** some problems can occur when we want to perform such operations on large sequences. Consider a text T of length n that is stored in a data structure that keeps the text continuous in memory. When performing an insertion or deletion on this text, the computational effort of the operation is defined by the number of positions that have to be moved.

For example when inserting a substring P of length m into T at position k with $k < n$ then the positions k to $n-1$ of T have to be moved by m to make sufficient space to accommodate the inserted substring. For a data structure that is saved continuously (e.g. in an array) this means that potentially a lot of values have to be moved in memory.

When counting copy operations of positions of T the computational effort of an insertion at position k can be described as $\mathcal{O}(m+n-k)$ since the last $n-k$ positions in T have to be moved and the m positions of the substring have to be inserted. An example is provided in **Figure 1.9**.

$$\begin{aligned} T &= \text{gattaca} \\ |T| &= n = 7 \\ P &= \text{xx} \\ |P| &= m = 2 \\ k &= 3 \end{aligned}$$

```
RAM ( before insertion )
Address: 00  01  02  03  04  05  06  07  08  09  0A  0B
Content:  g   a   t   t   a   c   a

RAM ( after insertion  )
Address: 00  01  02  03  04  05  06  07  08  09  0A  0B
Content:  g   a   x   x   t   t   a   c   a
```

Figure 1.9: Insertion into a continuously stored string.
The above example demonstrates the insertion of substring P = “xx” into text T = “gattaca” at position 2 when an array is used to store the string. By looking at the representation in RAM we can see, that in addition to the two insertions of values also five copy operations are needed to yield the modified sequence “gaxxttaca”.

When we assume that the positions of insertions are uniformly distributed along the text it is apparent that on average the computational effort will be $\mathcal{O}(m+n/2)$. A similar argument can be made for deletions, where positions greater than that of the deletion have to be moved to make the text continuous again (**Figure 1.3**). Replacing a substring with another substring of equal length m can be accomplished in $\mathcal{O}(m)$, because every position that has to be deleted can simply be overwritten with the corresponding new value.

For small sequences this is not really a concern, but for bioinformatical applications with large genomic sequences and their indices, the moving of substrings becomes a performance critical issue. A first problem that has to be solved can now be formalized as follows:

Problem 1 A way of storing sequence information has to be conceived, that will allow for insertion and deletion of substrings in very little time (less than $\mathcal{O}(m + n/2)$ for operations of length m).

1.2.3 Synchronizing the Index

Another important issue is the translation of modifications to the text into modifications to the index (suffix array and lcp-table).

Ideally we would like to keep as much of the original index as possible and only change very few positions. In this way we will generate an index that again is synchronous with the text. The approach we use in this thesis relies on determining for each suffix in the suffix array, whether its position within the suffix array is influenced by the changes made to the text.

For a text with exactly one modification, for example the insertion of a single character at position k, some rules for determining the influenced suffixes apply.

Obviously all the suffixes of positions greater than k will not be influenced by this insertion, since they do not contain the inserted character and therefore their lexicographical ordering with respect to each other will remain unchanged. The only thing that has to be done for them is shifting their positions by the length of the insertion (1 in our example), since the old suffix at position $k + 3$ is now at position $k + 4$ in the text and so on.

Then there are the $m = 1$ indices that have to be inserted which start at the inserted positions. Furthermore, for all the suffixes that start before the beginning of the insertion it has to be determined, if their lexicographical relation to their neighbors in the suffix array is influenced by the change in the text. Here the information stored in the lcp-table can be used. **Table 1.5** shows the listing of the suffix array (SA) and lcp-table (LCP) before and after the insertion given in the example from **Figure 1.9**.

In this table we can see, that the insertion of the two characters “xx” into the string “gattaca” leaves most of the initial suffix array intact, since the modified positions do not influence the suffixes lexicographical ordering. This can be determined by looking at the lcp-values (longest common prefix) which give for each suffix the number of characters that it has in common with its successor in the suffix array.

For example the suffix $SA[1]$ with lcp-value $LCP[1] = 1$ has one character in common with its successor in the SA. The position in the text, that determines the lexicographical ordering is $T[SA[1]+LCP[1]]$ since the first $LCP[1]$ positions of $SA[1]$ and $SA[2]$ are identical. Subsequently the value $T[SA[1] + LCP[1]]$ must then define $SA[1]$ to be lexicographically smaller than $SA[2]$.

When we know which suffixes are influenced by the changes, we can remove them from the suffix array and reinsert them at the correct positions to make the suffix array lexicographically ordered again. Additionally the suffixes that start at positions inserted in the

	SA_{old}			SA_{new}		
Index	Position	Suffix	LCP_{old}	Position	Suffix	LCP_{new}
0	6	**a**	1	8	**a**	1
1	4	**a**ca	1	6	**a**ca	1
2	1	attaca	0	1	axxttaca	0
3	6	ca	0	7	ca	0
4	0	gattaca	0	0	gaxxttaca	0
5	3	**t**aca	1	5	**t**aca	1
6	2	ttaca	0	4	ttaca	0
7				3	**x**ttaca	1
8				2	xxttaca	0

Table 1.5: The suffix array and lcp-table both before and after the insertion of the substring "xx" into the text "gattaca", to clarify the meaning of the lcp-values for each suffix the common prefix with its successor in the suffix array is marked **bold**.

text have to be added to the suffix array and those starting at positions deleted from the text have to be removed. Also the lcp-table must be modified according to the changes made to the suffix array.

Our second problem can now be defined as follows:

Problem 2 An algorithm has to be found, that can determine the operations needed to synchronize the index with the modified text. The number of operations should be as small as possible.

It is important to note that the suffix array of a large text is itself a large sequence (of positions in the text) and the problems with modifying large sequences apply to the suffix array as well. The efficient modification of large sequences described in **Section 1.2.2** is important for modifying the text and for synchronizing the index.

1.2.4 Summary of the Problems to be Solved

We now have defined the two main problems that we want to tackle in this thesis. We have shown that modifications to large sequences can be time costly and we will have to define a method to store a sequence and its modifications in a more efficient way.

As was described in **Section 1.1** and **Section 1.2.0.3**, building an index is also a time costly task and the initial investment of computation time for building the index is justified by the decrease of search time through the index. We have shown that an index synchronization algorithm that is faster than rebuilding the whole index is needed to prevent a waste of the computation time invested in the index construction.

The two main problems to tackle in his thesis have been defined as follows:

Problem 1 A way of storing sequence information has to be conceived, that will allow for insertion and deletion of substrings in very little time (less than $\mathcal{O}(m + n/2)$ for operations of length m).

Problem 2 An algorithm has to be found, that can determine the operations needed to synchronize the index with the modified text. The number of operations should be as small as possible.

We will now formulate the goals of this thesis that we want to achieve for solving the problems we defined.

1.3 Goals

After having defined the problems at hand, the goals of this thesis can now be formalized. Since the results of this thesis shall become part of the SeqAn C++ Library, the necessary implementations will utilize the functionality provided by SeqAn (Döring *et al.* , 2008).

SeqAn is a C++ programming library that provides data structures and algorithms for solving bioinformatical problems. It is a versatile tool with many functionalities, but we will concentrate on handling sequences and indices with SeqAn in the framework of this thesis. We will integrate the results of our research with the SeqAn C++ programming library by providing an implementation of the string class that allows for fast modifications of large instances.

To make use of the suffix array creation and search algorithms provided by SeqAn, we will have to implement the interface of our string class to be compatible with the SeqAn indices and finders (data structures that encapsulate text as well as index and provide searching functionality). After having done that we can use our class with the SeqAn implementations of an extended suffix array. We will create an algorithm to perform the synchronization of the index when the text has been modified.

To solve the problems defined in **Section 1.2.4**, we will take the following approaches:

1. Implement a string class within SeqAn that allows for efficient modifications of large instances, thus providing a solution to Problem 1.

2. Implement the necessary interface for the string class to make use of SeqAn finders and indices thereby allowing us to use the extended suffix array implementation of SeqAn.

3. Implement an algorithm to synchronize the suffix array and lcp-table of an extended suffix array with a modified text

In addition we will also use performance tests to determine the feasibility of our approach.

1.4 Similar Aproaches

A short overview of work significant to the research done here will be provided in this section. We will consider ways of reducing the space consumption of indices and text which would help to prevent the problems described in **Section 1.2.2**. We will also have a look at work that was done on dynamically updated index structures. This concerns papers with goals similar to ours as we defined them in the above section.

1.4.1 Lazy Suffix Trees

One interesting concept is that of lazy data structures. Those are data structures that are not build up completely on creation, rather they are calculated piecewise whenever a request to a part that has not yet been created is made. In this way the memory consumption is reduced since only the requested parts are created. Also when using a lazy index structure the initial creation is no longer the most time costly step since only the required parts of the index will be created "on demand".

We will now consider the example of a *lazy suffix tree*. A suffix tree over a text T is a data structure that basically stores the same information as the suffix array (see **Section 1.2.0.4**). It does so by saving all suffixes from T in the form of a tree. The edges have strings associated with them and each path from the root node to a leaf corresponds to a suffix in T. **Figure 1.10** gives an example of a suffix tree over the text "banana".

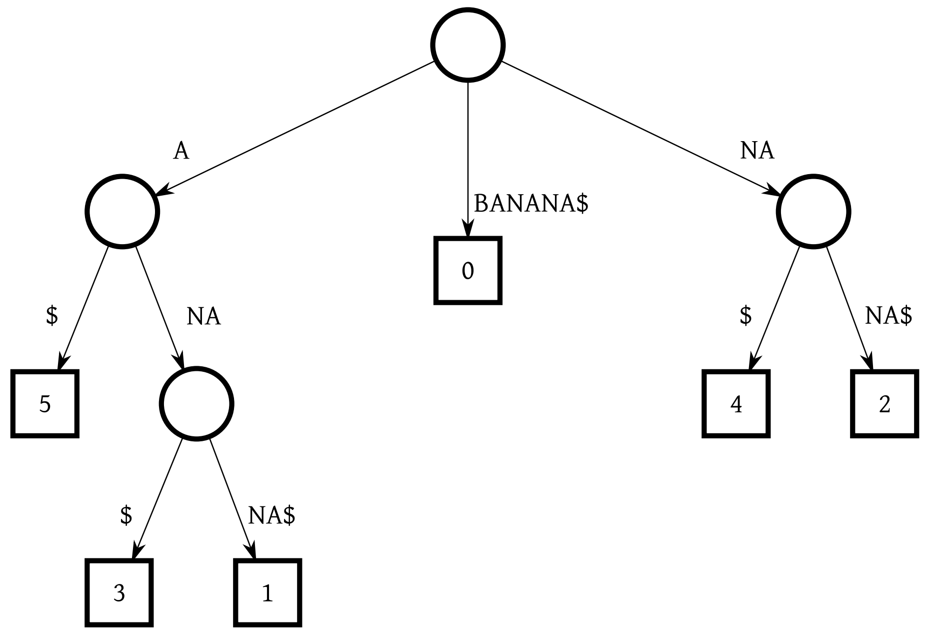


Figure 1.10: Suffix tree of the text "banana".
Boxes correspond to leaves for which the path from the root node represents a suffix of the text. Note how the special character $ is used to make sure each suffix has a unique path from the root to a leaf assigned to it. Without the $ as terminal charcter in the suffixes, those that are prefixes of other suffixes would end at internal nodes.

Each internal node of the suffix tree has at least two children and the leaves of all suffixes that have the path from the root node to this internal node as their common prefix are stored in the subtree rooted in that internal node.

When searching for a pattern in the suffix tree we have to follow the edges that correspond to the pattern. If we end up at a leaf, the pattern is a suffix of T and we can return the position stored in the leaf. If the traversal of the tree ends at an internal node the pattern is a prefix of all suffixes that are stored in the subtree rooted in the internal node. The search for the pattern “a” in our example suffix-tree will end at the leftmost internal node, therefore the pattern “a” occurs in the text at positions 5, 3 and 1 because those are the suffixes stored as leafs in the subtree rooted in the internal node. When the pattern does not occur in the text the search will end in the root node or in the middle of an edge and we can abort the search.

A lazy implementation of a suffix-tree will only calculate the subtrees up to the depth that was requested. When we, for example, would only request the positions of the pattern “na” from the lazy suffix tree we have to construct the rightmost subtree up to the first internal node so that the edge represents the substring “na”. For each possible first character we have constructed the subtrees and stored the information about all suffixes ending in the complete not yet constructed subtrees below them, the resulting suffix tree is shown in **Figure 1.11**.

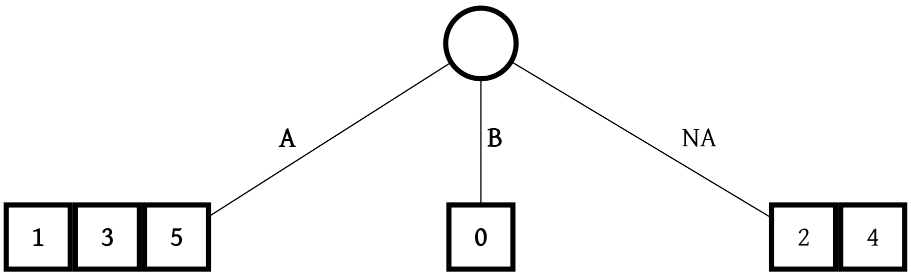


Figure 1.11: Lazy suffix tree of the text “banana”.
The status after the positions of pattern “na” have been requested is shown.

When searching for the pattern “a”, we will get the positions 5, 3 and 1 without actually having to construct the subtree that contains the corresponding suffixes.

In **Figure 1.12** the lazy suffix tree after the next query is shown. The pattern requested was “ana”. Note how the leftmost subtree is now expanded to the relevant depth for the pattern.

Basically a lazy suffix tree, as published by Gierich *et al.* in 1999 (Giegerich *et al.* , 1999), will only compute itself up to the required depth of a search performed on it. It will start out empty and whenever data is requested that lies within a not yet computed subtree of the suffix tree the relevant subtree will be constructed on demand. This approach makes the requesting of data potentially more time costly, since the lazy suffix tree might need to calculate a subtree first, before returning the requested data.

Since the suffix tree does not need to be computed directly, the initial time consumption is very small. If many requests concern the same areas of the suffix tree only the initial request

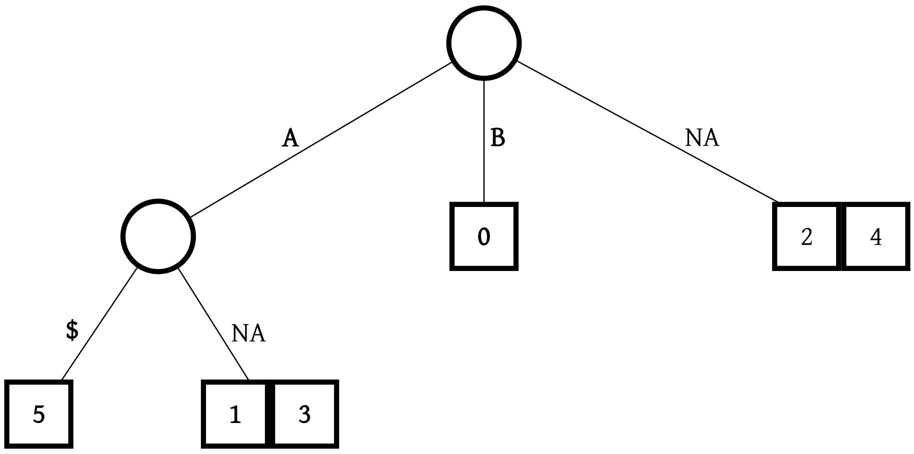


Figure 1.12: Lazy suffix tree of the text "banana".
The Status after the positions of patterns "na" and "ana" have been requested is shown.

will be slow but when this area is then computed the lazy suffix tree is as fast as a normal suffix tree.

Ideally the lazy suffix tree will never have to compute all subtrees and will therefore remain relatively small when compared to the complete suffix tree, that has been computed in its entirety beforehand. So the memory consumption can be significantly smaller, depending on the usage scenario.

1.4.2 Encoding Genomes by Their Differences to a Common Reference

In the paper **Human genomes as email attachments** published by Christley *et al.* (2009) ways of compressing genomic data by using a reference genome for decoding are discussed. The idea behind this is to define a human genome not by its sequence, but by its differences to a reference genome. In this way the amount of data that has to be stored is greatly reduced, the authors managed to compress all the relevant information about the differences between two human genomes to fit into 4Mb of memory. The encoded genome can be obtained by changing a predefined reference genome according to the stored differences. Of course the problem of modifying large sequences remains to be solved.

Also note that the way of encoding the differences described by Christley *et al.* does not allow for transparent access to the sequence in the same way journaling would do. The changes would have to be applied first, which is a time and probably also space costly thing to do. This approach solves the problem of transporting human genomic data via the Internet but it does not provide an easily applicable way of achieving the goals we defined in this thesis. The general idea in its application to the techniques that we want to use is discussed in **Section 5**.

1.4.3 Adapting the Burrows-Wheeler Transform to Changes in the Text

A data structure quite similar to the suffix array is the Burrows-Wheeler transform. It was first described by Burrows & Wheeler (1994). Basically the BWT is a reordering of the text and was originally designed for use in data compression. The BWT itself does not compress the text, but the reordering is done in such a way that it then can be compressed easier and more effectively. Of course, the recovery of the original text from the BWT is also possible. It does share a common drawback with the suffix array, because it is also a static data structure. When the text is modified the BWT also has to be completely recomputed.

Recently an algorithm for dynamically updating a Burrows-Wheeler transform (BWT) when the underlying text is changed was presented by Salson *et al.* in their paper **A four-stage algorithm for updating a Burrows-Wheeler transform** published in 2009b. The authors also identified the strong relationship between suffix array and BWT and proposed adapting their approach to the dynamic updating of a suffix array, which they later published in (Salson *et al.* , 2009a) (**Section 1.4.4**). Note that the work done in (Salson *et al.* , 2009b) was used as the basis for Salson *et al.* (2009a), but the papers apeared in reverse order in print.

To show how the BWT is relevant to our work we will now discuss this data structure: The BWT is constructed by considering cyclic shifts of a text.

Definition 8 (Cyclic shifts of a text).
We consider a text T of length n of which a substring from positions k to l is denoted as $T[k \dots l]$ (with $0 \leq k \leq l < n$).

A cyclic shift $T^{[i]}$ of order i is defined as follows:

$$T^{[i]} = T[i \dots n-1]\$T[0 \dots i-1]$$

Basically a cyclic shift of order i of the text T is constructed by shifting the text by i positions to the left and appending the prefix that would be shifted to negative positions at the end. All possible cyclic shifts of the text “banana$” are shown in **Table 1.6**.

Order	Cyclic shift						
0	b	a	n	a	n	a	$
1	a	n	a	n	a	$	b
2	n	a	n	a	$	b	a
3	a	n	a	$	b	a	n
4	n	a	$	b	a	n	a
5	a	$	b	a	n	a	n
6	$	b	a	n	a	n	a

Table 1.6: Cyclic shifts of a text.
All possible cyclic shifts of the text “banana$” are shown.

Note that for the examples used in this sections we assume that the text is terminated with a so called *sentinel letter* ($) that guarantees the lexicographic uniqueness of suffixes. It is used to avoid problems with suffixes that are prefixes of other suffixes. We defined the use of this sentinel letter in a different way in **Section 1.2.0.4** where we appended it only to the suffixes but explicitly stated that it is not part of the text. In which way this letter is handled is not important for its function and so both definitions serve the same purpose.

Definition 9 (Burrows-Wheeler transform).
The BWT of a text T is defined as the last column of the matrix M that contains the lexicographical ordering of all cyclic shifts of T.

For the example text “banana$” the BWT can be found in the gray marked column from **Table 1.7**. Also note that the array of the orders of the cyclic shifts which is shown in the first column corresponds directly to the suffix array for the text (**Table 1.7**). It contains the starting positions of all suffixes of “banana$” in lexicographical order.

Order	Cyclic shift						
6	$	b	a	n	a	n	a
5	a	$	b	a	n	a	n
3	a	n	a	$	b	a	n
1	a	n	a	n	a	$	b
0	b	a	n	a	n	a	$
4	n	a	$	b	a	n	a
2	n	a	n	a	$	b	a

Table 1.7: Lexicographically ordered cyclic shifts.
The lexicographical ordering of all cyclic shifts of the text “banana$” is shown. The gray rightmost column corresponds to the BWT of the text.

Note that the first column of M can be easily calculated from the last one, because M is lexicographically ordered and we know that each column of M contains all letters from the text. It is obvious that the first column will contain the letters from the alphabet of T in their lexicographical order and the only information needed to construct it is the count of each of those letters in T.

The approach for dynamically updating the BWT used four stages (Salson *et al.* , 2009b) to determine which cyclic shifts have been influenced by the operation. It then performs a series of modifications to rows in the BWT and also inserts new rows and reorders those whose lexicographical ordering was changed by the operation. For this approach the first column of M is used, but, since it is trivially computable from the BWT and easily encodable in little space, this is not a major drawback. The detailed description of both the BWT and the dynamic updating algorithm can be found in (Salson *et al.* , 2009b), and we will not go into more detail here.

1.4.4 Dynamic Extended Suffix Arrays

Salson *et al.* described an approach to dynamically adapt an extended suffix array in their paper **Dynamic extended suffix arrays** published in 2009a. Their approach solves the same problem we are adressing with our synchronization algorithm, which is that of preventing a re-computation of the suffix array when the text is changed. It is based on the work done by the same group in (Salson *et al.*, 2009b) and extends the approach used to dynamically update the BWT to be applicable to extended suffix arrays.

We explained in the section above that the BWT and the suffix array are closely related. Since we can calculate the suffix array from the BWT, we can also translate changes made to the BWT during the 4-stage update algorithm to changes made to the suffix array. In this way the correctly modified suffix array can be obtained. The authors also made use of the lcp-table and the inverse suffix array as auxiliary tables of which the inverse suffix array is essential to the updating algorithm.

Their approach is more memory intensive than the one we present here, since they have to store the BWT in addition to the suffix array, inverse suffix array and lcp-table. To reduce space consumption, the authors applied a technique called sampling to the suffix array and inverse suffix array. They only stored a sample (a subset) of those arrays and computed the values that are not stored directly when they were requested. The authors' approach gives an interesting perspective on tackling the problem, but since we do not use the BWT in our algorithm we cannot directly profit from their work.

2. Concepts

In this section the different concepts used to solve the problems and to achieve the goals of this thesis will be presented, including the design of the custom string class in SeqAn as well as a formal description of the algorithm used to synchronize index and text. Firstly we will discuss the different aspects of the custom string class that need to be considered. It will provide a means of journaling changes to the string without actually applying them while still providing transparent access to the string as though it were modified.

2.1 Journaling

The approach to efficiently implement insertions and deletions used in this thesis relies on the concept of journaling the changes made to the string without actually applying them in memory. To achieve this behavior, the changes must be stored in a so-called journal. Whenever the user interacts with the journaled string the changes stored in the journal are taken into consideration for this interaction. The results of data access operations need to be computed accordingly, so that we can access the string through the journal as though it were modified.

Consider an example where we have an insertion of sequence P of length $|P| = m \geq 1$ into the text T of length $n > m$ at position k with $0 \leq k < n$. When we do not apply this insertion directly we have to store the position where P would have been inserted as well as the sequence of P itself in the journal. When the user now requests the value at position $k+1$ the journal must return $P[1]$ instead of $T[k+1]$ because the requested position $k+1$ lies within the insertion that has not yet been performed. To achieve this behavior an additional layer of abstraction has to be introduced that is inserted between the user and the data structure.

Before describing this concept in more detail some definitions are needed, that will help to clarify the following explanations.

Definition 10 (The String).
The string is the underlying data structure to which modifications have to be made. The user should not interact with the string directly, but rather use the layer of abstraction provided by the journal.

Definition 11 (The Journal).
The journal is the data structure with which the user will interact, it is used for performing insertions and deletions on the string as well as for requesting values from the string. It will handle the efficient implementation of modifications by journaling them.

By combining the string with a journal that stores information about the performed operations we can greatly reduce the time consumption of insertions and deletions. We will now proceed by discussing the basics of the concept of journaling.

2.1.1 Basics

Journaling in the context of this thesis describes the concept of tracking changes to the underlying data structure without actually applying the tracked modifications. For this purpose a journaling data structure (the journal) must be conceived. This data structure has to be capable of storing changes made to the underlying data structure (the string). The journal must allow for transparent access to the string so that the user does not have to, and in fact cannot, know, whether his changes have been applied directly or have been journaled. The journal will provide transparent access to the underlying string.

Consider the following example: Given the sequence $S =$ abcdefghij, its journal J as well as $k = 2$ and $l = 4$ as the start and end positions for a deletion, the following behavior is expected:

Initial State

Position	0	1	2	3	4	5	6	7	8	9
Sequence	a	b	c	d	e	f	g	h	i	j
Journal	a	b	c	d	e	f	g	h	i	j

Deleted positions 2, 3 and 4

Position	0	1	2	3	4	5	6	7	8	9
Sequence	a	b	c	d	e	f	g	h	i	j
Journal	a	b	f	g	h	i	j	–	–	–

Table 2.1: Deleting from a sequence by using the journal.
Note how the sequence remains unchanged while the journal correctly applies the deletion when we use its data access function. Percieved throgh J the sequence is now "abfghij".

We have a journal J that keeps track of changes to the sequence S which stores characters and we want to perform a deletion on S that removes the characters from position k to l (inclusively with $k < l < |S| = n$). This operation is performed on the journal and is stored in J. It will be taken into consideration when data is requested from the journal.

The string S still has length n, but a request for the character $S[k+1]$ performed through J will yield the character $S[l+1]$. For a requested position ($k+1$ in our example) the journal calculates the actual position ($l+1$ in this case) of the character in the string and returns the correct value. The computation of the actual position in the string must be performed by J when the data is requested.

For a journal to behave in the way described above, it has to fulfill the following requirements:

1. Performing the operations insert and delete on the journal should take very little time

2. The journal has to keep track of all operations that have been performed on it without modifying the underlying string
3. When data for a position k is requested from the journal, it should behave as if the stored operations would have been applied to the underlying string
4. A method to apply the journaled operations to the underlying string (making the changes persistent) must also be provided. After persisiting the changes, the journal is empty again.

Table 2.1 and **Figure 2.1** provide examples for the journaling of a delete operation.

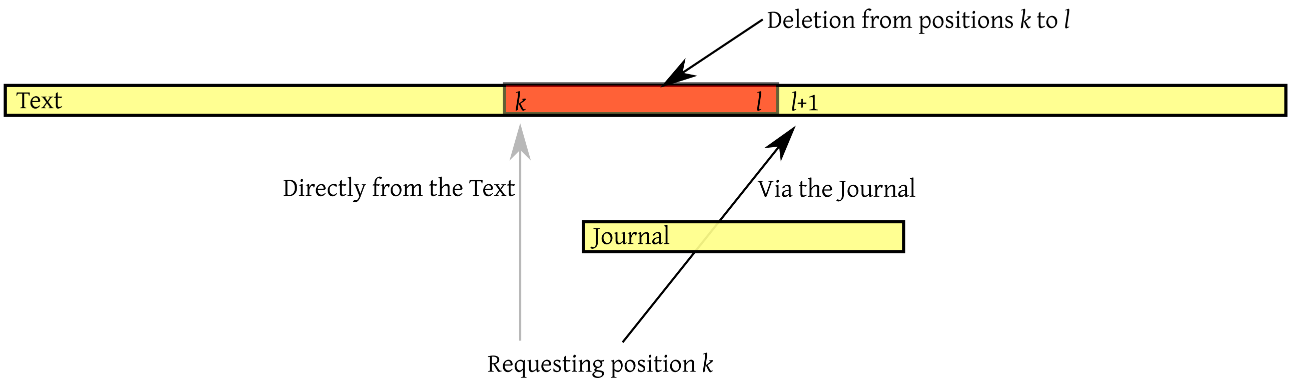


Figure 2.1: Journaling a delete operation.
The gray arrow shows the result of directly accessing position k in the text, the black arrow shows the result of querying the journal for position k.

A first step on the way to designing a string class with journaling capabilities is to conceive a means of storing the information about already applied operations. This will allow for easy computation of the correct return values when data is requested. The string class developed for this thesis uses a journal tree to store information about applied operations and to compute return values accordingly. We will now discuss the concept of the journal tree in more detail.

2.1.2 The Journal Tree

Before we start discussing the journal tree we will define physical positions and virtual positions since we use these concepts in the following explanations.

Definition 12 (Physical Position)**.**
The physical position of a value v in a text T is defined as its position relative to the beginning of T. For $v = T[k]$ the physical position is k since v is the $(k+1)$-th value in the text.

Definition 13 (Virtual Position)**.**
The virtual position of a value v is defined with respect to a journaled text T and an associated journal J. The virtual position of the value v is the position that the value would have in the string if all the operations stored in J would be applied to T.

In the example shown in **Figure 2.1** the query for the virtual position k from the journal yields the physical position $l + 1$ of the value in the text.

Consider an operation at a given position k of length m. It affects the virtual positions of all the following suffixes. A deletion influences all suffixes S_d with $k + m \leq d < n$ and an insertion affects all suffixes S_i with $k \leq i < n$. Each deletion reduces those virtual positions by its length (m) and each insertion increases them. We can now define the journal J as a function that performs a transformation from the virtual positions to the physical positions of values in T:

$$J(k) : \mathbb{N} \mapsto \mathbb{N}$$

In this thesis the journal was implemented utilizing a journal tree. A journal tree is basically a binary search tree on the positions in the modified string (the virtual positions) where the nodes represent insertions, deletions and parts of the original string. To get a character from the modified string the journal tree has to be traversed until the node that contains the requested virtual position is found. From the data in that node, the physical position and therefore the correct value can be computed and returned.

The diagram in **Figure 2.2** illustrates the basic concept of the journal tree with the example from **Table 2.1**.

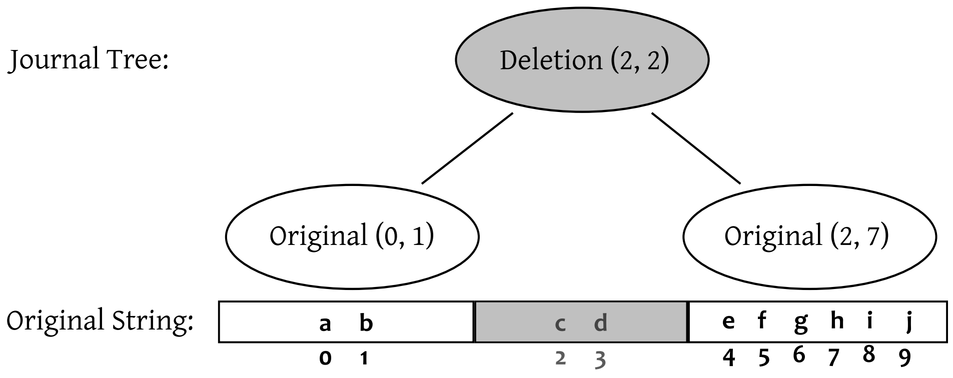


Figure 2.2: Example of a journal tree after storing a deletion.
Note that after the deletion of positions 2, 3 and 4 the virtual positions of the values "fghij" are reduced by 3, while the values still appear at the same physical position in the unmodified underlying string (see **Definitions 12** and **13**). The nodes are labeled with two numbers representing the virtual starting position and the length of the block of data represented by the corresponding node.

2.1.3 The Journal Tree – Basics

The concept of journaling by means of a tree is to use a binary search tree for accessing the underlying string. The nodes of this tree correspond to blocks of data which can be parts of the underlying string, deletions or insertions. Each operation is stored in the tree as a node representing a block of the operation type. The transformation $J(k) : \mathbb{N} \mapsto \mathbb{N}$ will be stored implicitly in the nodes and structure of the journal tree.

In the most basic setup the journal tree is stored independently from the underlying string, interacting with the string by means of pointers to blocks of data. It will consist of nodes that contain information about the type of operation and block of data they represent. A node of the journal tree stores the following information:

- the type of block that is represented by this node (a part of the underlying string, an insertion or a deletion)
- a reference to the beginning of the represented block, this can point into the underlying string or to an inserted string somewhere else in memory (this corresponds to physical position of the first value in the block)
- the virtual position and the length of the block (see **Definition 13**)

If the journal tree is stored separately from the underlying string the strings that are inserted can also be stored separately. For each insertion the journal tree will contain a node representing this operation and that node will point to the blocks of data in the inserted string. In this way a repeat in the underlying string could easily be modeled by repeatedly inserting a substring of the underlying string.

In **Figure 2.3** we show the journal tree for the example given in **Figure 1.9** to demonstrate a possible setup. Here the text T = gattaca with $|T| = n = 7$ is modified by inserting the substring $T' =$ xx with $|T'| = m = 2$ at position $k = 3$.

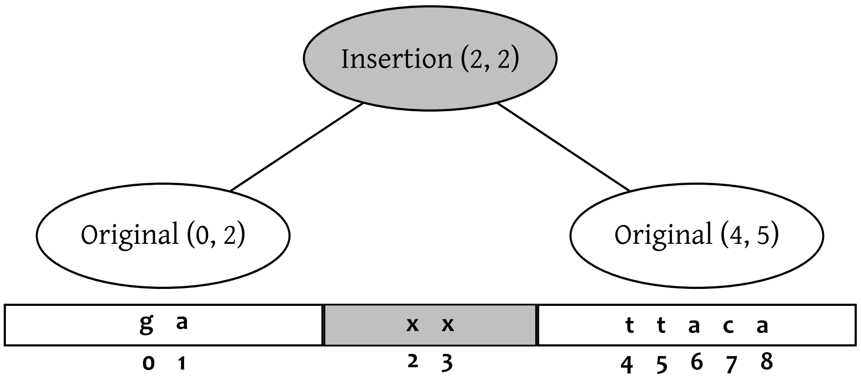


Figure 2.3: Journal tree for the example from **Figure 1.9**.

The resulting changes in the computer's RAM are shown in **Figure 2.4**, comapre this to the status shown in **Figure 1.9**.

The advantage of this way of handling operations is that they can be performed in very little time no matter how big the string is, since the only thing to do is modifying the journal tree by adding nodes and copying the inserted values into the computers RAM (if they are not accessible by the journal already).

```
RAM ( before insertion ):
 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
  g   a   t   t   a   c   a

RAM ( after insertion  ):
 00  01  02  03  04  05  06  07  08  09  0A  0B  0C  0D  0E  0F
  g   a   t   t   a   c   a                   x   x
```

Figure 2.4: Journaling and the computers memory.
Here it can be seen that the journaled insertion of the string "xx" now happens without changing the original string in RAM. The inserted string is stored at a different address and the journal tree is modified to include the changes. Compare this to the example from **Figure 1.9**.

2.1.4 The Journal Tree – Considerations

The time consumption for inserting a sequence of length m into a text of size n is now reduced from $\mathcal{O}(m+n/2)$ to $\mathcal{O}(m+\log q)$ (for a journal tree with q nodes) because all we have to do is store the m inserted positions somewhere in memory and modify the journal tree. Before we can modify the journal tree we have to find the correct node that must be modified to store the insertion. This finding is done by traversing the journal tree (which is a binary search tree) and costs $\mathcal{O}(\log q)$ time in the average case. If the string that should be inserted is already present in the computer's memory, the time consumption is in fact $\mathcal{O}(\log q)$ because only the journal tree modification remains to be done and the node then can store a pointer to the already present insertion data.

This advantage of fast modification of the string provided by the journal tree comes with some drawbacks. The main disadvantage of the journal tree and journaling in general is the slower data access, because for every requested virtual position the correct physical position has to be calculated first. For a string stored in a continuous array the cost of retrieving a value at position k is $\mathcal{O}(1)$ since its memory address can be easily computed from the position and the memory address of the array start.

For our journaled string this is not feasible anymore because the values are not stored continuously in the computers memory. Instead of this simple computation we now have to traverse the journal tree until we find the node representing the block that contains the requested virtual position. When the correct block is found we have to compute the correct physical position of the requested value with regard to the memory address of the block. In our implementation we will store the physical position within the node so that we do not have to recompute it for each data access. This means that for a journal tree consisting of q nodes the cost of retrieving a single value is increased from $\mathcal{O}(1)$ to $\mathcal{O}(\log q)$ in the average case. The $\mathcal{O}(\log q)$ is a result of the traversal of the binary search tree that is our journal tree. Subsequently a journal that has stored many operations becomes slower for data access.

This effect can be countered by regularly applying the journaled operations to the underlying string, thereby reducing the size of the tree to one node. The concept of applying the changes (*flattening* the journaled string) will be explained in **Section 2.2.4.2**.

All of this has to be considered when using journal trees, since the performance gained from efficient insertions can easily be lost again due to the costly data access. There are several strategies that can be applied to reduce the impact of this drawbacks. For example balancing the journal tree can guarantee $\mathcal{O}(\log q)$ running time for data access. If we use an unbalanced journal tree it is possible to have a time consumption of $\mathcal{O}(q)$ for finding the correct node both for data access and for storing a new operation. The journal tree can degenerate in such a way that is has a height of q when storing q nodes, in this case the worst-case running time for finding a node in this tree is $\mathcal{O}(q)$. **Figure 2.5** shows an example of this effect.

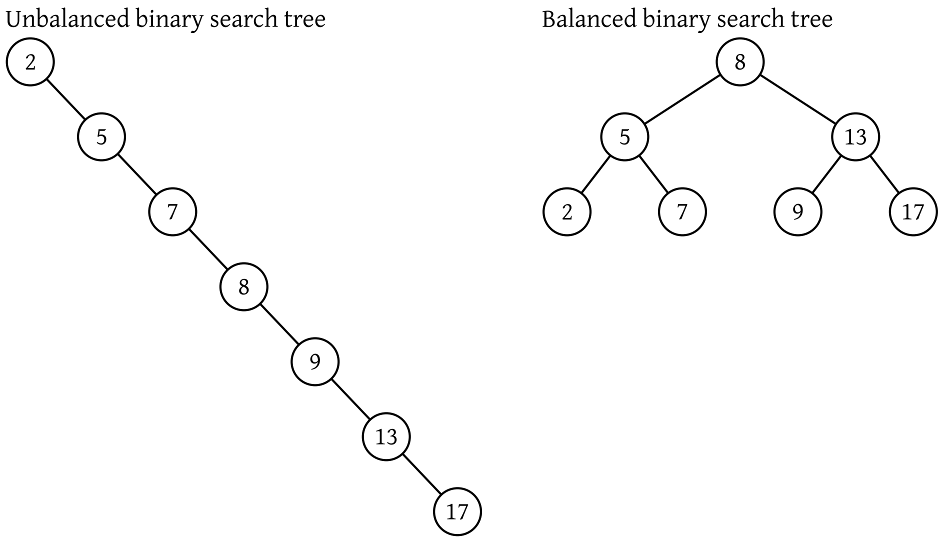


Figure 2.5: Balanced and unbalanced binary search trees.
The same binary search tree in both unbalanced and balanced form. Note the difference in tree height.

Basically, a mechanism for maintenance of the journal tree has to be implemented, that reduces the performance loss due to slow data access.

2.2 Synchronizing the Index

We will now describe a general approach for synchronizing the suffix array and lcp-table with a changed text. For the explanations that follow, we will not make a distinction between deletions and insertions, and deal with modified positions in general. The important information about a position in the text is not whether it has been changed because of a deletion or an insertion, it is more important that the value at the position has changed.

In the situation in which we will perform this synchronization we have a text T and a journal J associated with this text. We also have an extended suffix array consisting of a suffix array, lcp-table and inverse suffix-array as the index structure. We make use of the information contained in the journal to modify the index in such a way that it is synchronous with the text.

2.2.1 Finding Influenced Suffixes

The main task when synchronizing the index to a changed text is to determine which suffixes have been influenced by the changes. A suffix is considered influenced when a modification in the text has occurred that changes the position of the suffix in the lexicographical ordering of all suffixes from the text (i.e. the suffix array).

The lcp-table does not require special attention since the lcp-values are computed from the suffixes in the suffix array. Any changes made to the suffix array will imply changes that have to be made to the lcp-table because the successors of suffixes have changed.

It is important to note that for two suffixes, S_k and S_l starting at positions $SA[i] = k$ and $SA[i+1] = l$ within the text, their lexicographic relation is determined by the values at the positions $k + \text{lcpValue}(S_k, S_l)$ and $l + \text{lcpValue}(S_k, S_l)$ in the text. **Figure 2.6** illustrates this behavior.

$$
\begin{aligned}
S &= \text{accgcct} \\
\text{lcpValue}(S_k, S_l) &= 2
\end{aligned}
$$

	Position	Suffix
	⋮	⋮
S_k	1	c c g c c t $
S_l	4	c c t $
	⋮	⋮

Figure 2.6: Order defining positions and the lcp-value.
The lexicographical relation of $S_k =$ ccgcct$ and $S_l =$ cct$ (for text $T =$ accgcct, $k = 1$ and $l = 4$) is determined by the comparison of the values $S_k[2] = g$ and $S_l[2] = t$ which can be found in the gray column. The longest common prefix describes the part of the suffixes that is equal, so the lcp-value of 2 means that the suffixes have their first two values in common.

lcpValue(S_k, S_l) is a function returning the length of the longest common prefix of the two suffixes, it is defined in **Definition 7**. The lcp-value for each suffix and its successor in the suffix array is stored in the lcp-table. For our suffixes S_k and S_l which are stored at positions i and $i+1$ in the suffix array the lcp-value is stored in $LCP[i]$. Since there is no successor to the last suffix in the suffix array, the last entry of the lcp-table is set to zero.

From the observation illustrated in **Figure 2.6** we can conclude that the suffix S_k (with $SA[i] = k$) can only be influenced by a modification if the position of that modification lies within the interval $T[SA[i] \ldots SA[i] + \max(LCP[i], LCP[i-1])]$ in the text. We use the maximum of the lcp-values of S_k and its predecessor and successor in the lcp-table since this is also the maximum of all lcpValue(S_k, S_u) with $u \neq kand 0 \leq u < |SA|$. We call this interval the *sensitive interval* of the suffix.

Definition 14 (The Sensitive Interval of a Suffix)**.**
Given a text $T \in \Sigma^*$ over a sorted alphabet Σ, the associated suffix array SA and the lcp-table LCP we define the sensitive interval of a suffix S_k (with $SA[i] = k$) as:

$$T[SA[i] \ldots SA[i] + \max(LCP[i], LCP[i-1])]$$

It contains the positions within the text that define the lexicographical relation of S_k and S_l (with $SA[i+1] = l$) respectively S_j (with $SA[i-1] = j$). Subsequently, changes made to the text within the sensitive interval of S_k can influence its position within the suffix array.

The sensitive intervals of the first and last suffix in the suffix array are only dependent on the respective successor and predecessor since the first suffix has no predecessor and the last has no successor.

Modifications to the text at positions greater than $SA[i] + LCP[i]$ do not influence the lexicographical relation of S_k to its successor. Simply put, “aba” is lexicographically smaller than “acaba” because of the “b” at position 1. The “a” at position 2 does not change this. To influence this relation we would have to modify the first or second position of “aba” because “abx” is still lexicographically smaller than “acaba” since their relation is determined at their respective second position. The sensitive interval of “aba” with respect to its successor “acaba” has length 2 since $\text{lcpValue}(\text{aba}, \text{acaba}) = 1$.

Finding influenced suffixes can now be done by checking for all suffixes whether somewhere within their sensitive intervals a change has occurred. If a suffix is found to be influenced, its position within the suffix array must be recomputed and the suffix must be repositioned accordingly. In this way the suffix array is restored to being in lexicographical order. For this approach to work it must be guaranteed that the suffix array and lcp-table were synchronous with the text before the modifications occurred. If they were not synchronous the assumption about the sensitive intervals of the suffixes does not hold and we cannot easily determine which suffixes are influenced and which are not.

2.2.2 Deleted, Inserted and Shifted Suffixes

Now we can find the suffixes that are influenced and recompute their positions in the suffix array and the associated values in the lcp-table. Other suffixes have to be added to the suffix array because they start in a block that was inserted into the text. Finally, some suffixes will have to be removed from the suffix array because their starting-positions lie within a block of the string that was deleted.

All suffixes that start at a position that is larger than the position of at least one operation performed on the string must be shifted so that their physical positions match their virtual positions. Consider the example of a deletion of length m that has occurred at position k. Now all suffixes starting at positions $k \dots k+m-1$ have to be deleted from the suffix array. Additionally, all the suffixes starting at positions $k+m \dots n-1$ (where n is the length of the underlying string before the deletion) have to be shifted by $-m$, because the old suffix at position $k+m+1$ is now at position $k+1$ in the modified text. Even though those suffixes are not influenced and retain their lexicographical ordering with respect to each other, their starting positions have been reduced by the length m of the deletion.

When we use a journal tree to keep track of the changes made to the text, we can determine the shifts by comparing the physical positions of the suffixes (which are the entries in the

	SA_{old}		SA_{new}		
Index	Position	Suffix	Position	Suffix	Comment
0	0	coffee	0	coe	not shifted
1	5	e	2	e	shifted by -3 because of deletion
2	4	ee	1	oe	shifted by -3 because of deletion
3	3	fee			
4	2	ffee			
5	1	offee			

Table 2.2: Shifting suffixes after a deletion.
The suffix array and lcp-table of the text "coffee" both before and after the deletion described in the source code from **Listing 2.1**. Note how the suffixes starting after the deleted block have to be shifted because their positions in the modified text have changed. Also note that, even though the suffixes have changed due to the deletion, none of the changes occurred in their respective sensitive intervals, thereby preserving their lexicographical relation to each other.

suffix array) to their virtual position and modify the entries in the suffix array to contain those virtual positions.

Listing 2.1 illustrates the erasure of a substring when using C++ and SeqAn (Döring *et al.* , 2008). **Table 2.2** shows the necessary modifications to the associated suffix array.

```
String<char> the_string = "coffee";
std::cout << the_string << std::endl;     //prints 'coffee'
erase(the_string, 2, 5);                  //erase positions 2,3 and 4; the interval [2..5)
std::cout << the_string << std::endl;     //prints 'coe'
```

Listing 2.1: Deleting from a string in SeqAn. The complete interface for working with sequences can be found in the online documentation available from the SeqAn homepage.

The difference between the virtual and physical position of a suffix (the shift) is determined by the sum of the lengths of all operations that occurred in the string at positions smaller than the position of the suffix. For this calculation the length of a deletion must be a negative number representing the number of deleted suffixes. To calculate the length of the shift correctly is not a trivial task, since all the successive operations influence each other. It will be described in more detail later on.

2.2.3 The Algorithm

After we have now discussed the general tasks that have to be performed for index synchronization, we will now formalize the algorithm used in this thesis. Before we explain the various stages of the algorithm, we will describe the initial situation. The following elements must be present for the algorithm to work properly:

- A text T that is in some initial state.
- A journal J that kept track of all the changes to T that occurred since the initialization. It must provide access to the data concerning the journaled operations. This will be a journal tree of the kind that was described in **Section 2.1**.
- An Index containing a suffix array and lcp-table that are synchronous with the initial state of the string.

This situation where an index was synchronous with the initial text and must now be synchronized to the modified text is an example of a typical application scenario for the algorithm described in this thesis. We will now present the general layout of the synchronization algorithm which will consist of the following steps:

1. determine for each suffix in the suffix array whether it must be deleted or shifted and modify the suffix array accordingly (deleting and shifting suffixes as is necessary)
2. determine for the shifted suffixes if they are influenced by any of the operations stored in the journal tree and remove them from the suffix array
3. determine the set of all suffixes that have to be added to the suffix array because they start within insertion blocks
4. combine the sets of influenced and inserted suffixes and sort this set lexicographically
5. merge the remaining part of the suffix array and the set of inserted and influenced suffixes to form the synchronized lexicographically sorted new suffix array

In all these steps it is important to make sure that the lcp-table is modified to encompass the changes made to the suffix array. The following sections will now explain the different stages of the algorithm in more detail.

2.2.3.1 Deleting and Shifting Suffixes

The first step in the synchronization is the calculation of the correct shifts for the suffixes. The suffixes starting in deleted blocks must also be removed from the suffix array.

The length of the shifts that the suffixes must undergo is computed from the data stored in the nodes of the journal tree, this is done in one inorder traversal of the journal tree. It is important to compute the shifts successively from the nodes traversed inorder because the shifts are cumulative for operations occurring successively in the text. For example the suffix at position k, where k is greater than the positions of two insertions of length 4 and one deletion of length 5, must be shifted by $4 + 4 - 5 = 3$ because its virtual position is reduced by three compared to its physical position. The algorithm used to compute the shifts is described in pseudo-code in **Algorithm 2**.

Algorithm 2 The algorithm for computing the shifts of suffixes

```
1: function $calculate\_shifts(J)$
2:     $N \leftarrow J.nodes\_inorder()$ ▷ The set N contains all the Nodes of Journal Tree J inorder
3:     $Sh \leftarrow \{(0,0)\}$ ▷ Set Sh will contain Pairs that describe the shifts.
4:     for all $n \in N$ do
5:         if $!n.isOriginal()$ then ▷ If the node contains an insertion or deletion block
6:             $shift \leftarrow Sh.back().second$ ▷ get last shift length
7:             $shift+ = n.length$ ▷ add length of current block
8:             $Sh \leftarrow \{(n.pos, shift)\} \cup Sh$ ▷ add new pair to Sh
9:         end if
10:    end for
11:    $Sh \leftarrow \{(length(J), 0)\} \cup Sh$ ▷ A dummy pair provides an upper boundary for binary search
12:    return $Sh$
13: end function
```

The first step is to get the nodes of the journal tree in their inorder sequence. We then initialize the set Sh with an initial pair $(0, 0)$. The pairs stored in Sh contain two values, the first value is a position in the text and the second value is the associated shift length. The meaning of such a pair (a, b) is that all positions greater or equal to a must be shifted by b. Now for each node it is checked whether it is an insertion or deletion, which means that it is not a part of the original string.

When a node representing an insertion or deletion is found, all positions greater than the starting position of the node's block must be shifted by its length plus the shift length accumulated from all operations at smaller positions. As an example we will consider a text of length n where there are two insertions of length 2 at positions k and l with $k < l$. After running **Algorithm 2** the set Sh would contain the pairs $Sh = (0, 0), (k, 2), (l, 4), (n, 0)$. All positions before k will not be shifted, all positions between k and l are influenced only by the first insertion and therefore are shifted by 2 and all positions greater or equal to l are influenced by both insertions and have to be shifted by 4.

The pairs in Sh are sorted according to their first values so that we can perform a binary search on them to find the shift for a given position. **Figure 2.7** provides a visual example.

When we now want to find the shift for a given suffix $SA[i] = k$, a binary search can be performed on the set Sh generated by **Algorithm 2**. This is done by comparing the suffixes position k with the first values of the pairs in Sh and returning the second value (the shift-by) of the pair Sh_j that has the greatest first value while $k > Sh_j.first$ still holds. The function `getshift` that computes the shift given a position and a set of pairs is therefore a simple abstraction of a binary search algorithm (see **Algorithm 3**).

The deleted suffixes can be found in a similar manner. We can find all the virtual positions of suffixes that have to be deleted by simply traversing the journal tree inorder and, for each

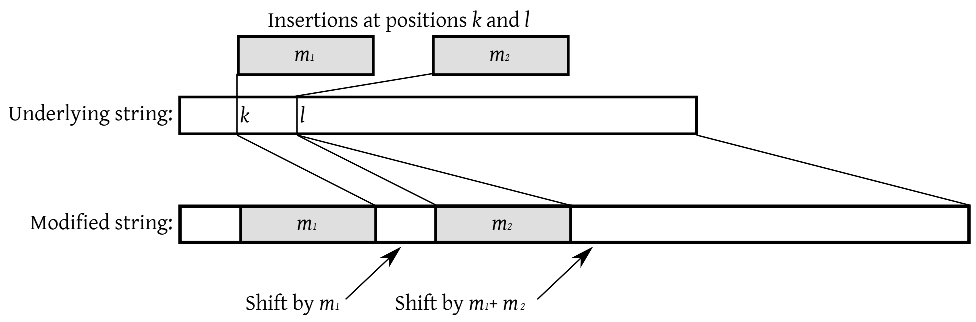


Figure 2.7: Two successive insertions and their influence on the shifts for suffixes.

Algorithm 3 The algorithm for finding the shift of a given suffix. Note that the elements of pairs from S can be accessed via the two functions $first$ and $second$.

```
1:  function getshift($pos, Sh$)
2:      $low \leftarrow begin(Sh)$                    ▷ get iterator to first pair from Sh
3:      $up \leftarrow end(Sh) - 1$                   ▷ get iterator to last pair from Sh
4:      while $true$ do
5:          $mid \leftarrow Sh[floor(up - low/2)]$
6:          if $pos < mid \rightarrow first$ then      ▷ Position lies in lower half of interval
7:              $up \leftarrow mid$
8:          else if $pos > (mid + 1) \rightarrow first$ then   ▷ Position lies in upper half of interval
9:              $low \leftarrow mid$
10:         else                                  ▷ Position is in the block described by mid
11:             return $mid \rightarrow second$
12:         end if
13:     end while
14: end function
```

node that represents a deletion, saving the position and length of its block. The pseudo-code for finding deleted suffixes is presented in **Algorithm 4**.

The algorithm to determine if a suffix has to be deleted can again be implemented via binary search. We simply compare the suffix position with the elements of the set D and check whether a deletion block exists that contains our suffixes position.

It is apparent that the order in which suffixes are marked for deletion or shifting is important. Shifting will ultimately result in suffixes that had positions greater than the position of the deletion block being assigned positions within the deletion block. If we shift first and delete afterwards we would delete suffixes that originally were not part of a deletion block. Therefore, for each suffix in the suffix array, we first have to check whether it has to be deleted and, only if that is not the case, shift is to the correct position. The complete setup for the first part of the synchronization-algorithm is described in pseudo-code in **Algorithm 5**

This algorithm takes a journal J to a text and an index I over the text as input. The suffix

Algorithm 4 The algorithm for computing the suffixes to be deleted

```
function $calculate\_deletions(J)$
    $N \leftarrow J.nodes\_inorder()$ ▷ The set N contains all the Nodes of Journal Tree J inorder
    $D \leftarrow \{\}$ ▷ Set D will contain Pairs that describe blocks of deletions.
    for all $n \in N$ do
        if $n.isDeletion()$ then ▷ If the node contains a deletion block
            $D \leftarrow \{(n.pos, abs(n.length))\} \cup D$ ▷ use abs(n.length) for deletions
        end if
    end for
    return $D$
end function
```

Algorithm 5 The algorithm for synchronizing text and index - first steps

```
procedure $delete\_and\_shift(J, I)$ ▷ Input is journal $J$ and index $I$
    $SA \leftarrow I.suffix\_array()$ ▷ SA holds the suffix array to be synchronized
    $LCP \leftarrow I.lcp\_table()$ ▷ LCP holds the corresponding lcp-table
    $D \leftarrow calculate\_deletions(J)$
    $Sh \leftarrow calculate\_shifts(J)$ ▷ store shifts in $S$ and deletions in $D$
    for $i \leftarrow 1, length(SA)$ do
        if $is\_deleted(SA[i], D)$ then ▷ $is\_deleted$ uses binary search similar to $get\_shift$
            delete $SA[i]$
            if $i > 1$ then
                $LCP[i-1] \leftarrow min(LCP[i], LCP[i-1])$ ▷ get new lcp-value
            end if
            delete $LCP[i]$ ▷ the suffix and corresponding lcp-value have been deleted
        else
            $SA[i] \leftarrow SA[i] + get\_shift(SA[i], Sh)$ ▷ shift the suffix to its correct position
        end if
    end for
end procedure
```

array SA and lcp-table LCP are requested from the index, those are synchronous with the underlying text T but not with the journal J which has some operations stored in a journal tree. Now the two functions described in **Algorithm 2** and **Algorithm 4** are used to compute the sets of pairs describing the necessary shifts and deletions.

In the central loop the suffix array is iterated and for each $SA[i] \in SA$ $(0 \leq i < |SA| = n)$ it is checked, if this suffix starts at a position that has to be deleted. This check is done via the function is_deleted which is not explicitly described here, since it is is similar to get_shift. If the suffix $SA[i]$ must be deleted it is removed from SA, otherwise it is shifted to its correct position. Also when a deletion occurs in the suffix array at position $i > 1$ the lcp-table has to be adjusted at the position $i - 1$ since the successor of the $i - 1$-th suffix has changed from $SA[i]$ to $SA[i + 1]$.

It is important to understand the properties of the lcp-table correctly because they can be used to determine the lcp-value at position $i - 1$ without having to recompute it from the two suffixes $SA[i - 1]$ and $SA[i + 1]$. We can calculate it easily before deleting $SA[i]$. In this algorithm the fact is used, that the lcp-value for two arbitrarily chosen suffixes in the suffix array is always the minimum over all the lcp-values in the interval between those suffixes. So in our case $LCP[i - 1]$ has to be set to the lcp-value of suffixes $SA[i - 1]$ and $SA[i + 1]$ when $SA[i]$ is deleted, which is the minimum of the interval $LCP[i - 1 \ldots i]$ and can be computed with the expression used in line 14 of the pseudo-code given in **Algorithm 5**. An illustration of this property of the lcp-table is given in **Table 2.3**.

Index	#	SA	LCP
0	0	**aaa**aaab	5
1	1	**aaa**aab	4
2	2	**aaa**ab	3
3	3	**aaa**b	2
4	4	aab	1
5	5	ab	0
6	6	b	0

Table 2.3: The suffix array and lcp-table of the string "aaaaaab"
The common prefix of suffix 0 and 3 is marked in **bold** letters. It is of length 3 which is the minimum of the lcp-interval $LCP[0 \ldots 2] = \{5, 4, 3\}$. This example also illustrates that a suffix can never have more leading characters in common with another suffix occuring behind it in the suffix array, than it has with its direct successor in the suffix array.

After the completion of this loop the suffix array contains only suffixes that have not been deleted and are pointing to their correct positions in the text (the positions that will result from applying the changes stored in J to the text T).

The approach presented in **Algorithm 5** is obviously not optimal since we iterate over all suffixes and check whether they have to be deleted. If the inverse suffix array is available we can easily delete all relevant suffixes by iterating over the set of pairs describing deletions. For each pair we can use the inverse suffix array to find the corresponding positions in the suffix array and delete them from the suffix array.

In fact, if the inverse suffix array is available we can perform both deletions and shifts "on the fly" while traversing the journal tree inorder. This is because for each node we can use the inverse suffix array to find the positions of the relevant suffixes in the suffix array and perform the necessary deletions and shifts (see **Section 1.2.0.4** and **Section 1.2.0.7** for the relation between the two arrays).

2.2.3.2 Finding Influenced Suffixes

After all the deleted suffixes have been removed from the suffix array the next step is to find out which of the remaining suffixes are influenced by the the operations to be performed. Those are suffixes that have to be repositioned in the suffix array because their lexicographical relation to their neighbors in the suffix array is not correct.

In **Section 2.2.1** we described that, to determine whether a suffix $SA[i]$ is influenced or not it must be determined if any of the first $LCP[i] + 1$ positions of this suffix have been altered. Therefore we introduced the concept of the sensitive interval of $SA[i]$ in **Section 2.2.1**. This interval is the prefix of $SA[i]$ that, when modified, changes the suffixes lexicographical relation to its neighbors in the suffix array.

If the inverse suffix array is available we can use it to speed up the process of finding influenced suffixes. The inverse suffix array $sufinv$ contains for each suffix its position in the suffix array, such that $SA[i] = k \Leftrightarrow sufinv[k] = i$ holds. With this information it is simple to find all the suffixes influenced by an operation at position p. All suffixes that could possibly be influenced by this operation have positions smaller than p in the inverse suffix array, for example $SA[sufinv[p-1]]$.We only have to check the suffixes starting at position $p-1$ until one is found that is not influenced, if that is the case the search can be stopped. We check the suffixes in descending order, meaning we start at position $p-1$ and continue with position $p-2$ and so on, until we reach the first suffix at position $p-p=0$ or a non-influenced suffix is found.

The condition that the search can be stopped, if one non-influenced suffix is found, is based on **Theorem 1** from the paper **Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications** by Kasai *et al.* (2001). Here the authors prove that, for two suffixes S_i and S_{i-1} and their respective adjacent suffixes in the lcp-table S_j and S_k, if lcpValue(S_{i-1}, S_j) is h then lcpValue(S_i, S_k) is at least $h-1$. Conclusively any suffix S_u that is not influenced by an operation at position p has an lcp-value that is at least $h-x$ where h is the lcp-value of suffix S_v and its adjacent suffix in the lcp-table (with $0 \leq v < u < |SA|$ and $u - v = x$). By using this theorem we have shown that the sensitive intervals of all suffixes that start at positions smaller than k can not have a larger upper boundary than the sensitive interval of S_k, they can, however, have a smaller one.

The following paragraph gives a more detailed example of the way the influenced suffixes are determined when we use the inverse suffix array.

Example
Assume that for the operation at position p we have found the suffixes starting at positions $p-1$ and $p-2$ to be influenced. In the next step we find that the suffix starting at $SA[sufinv[p-3]] = p-3$ has a sensitive interval of length 2. Now it is immediately clear that the suffix starting at $p-3$ is not influenced by the operation at p since the operation does not lie within its sensitive interval $T[p-3 \ldots p-1]$. The position in the text that defines the lexicographical relation of the suffix to its neighbors in the suffix array is $p-3+LCP[sufinv[p-3]] = p-3+1 = p-2$. Hence any change in the suffix occurring behind the position $p-2$ in the text does not influence its lexicographical ordering.

Additionally we know that for each suffix S_k starting at a position smaller than $p-3$ in the text ($0 \leq k < p-3$) the $p-3$-th suffix of the text is again a suffix of S_k. Because the position $p-2$ has been found to be the one that determines the lexicographical relationship for S_{p-3}, and it is contained in all the suffixes starting at positions smaller than $p-3$, this position will also be lexicographically defining for those suffixes. The suffixes could have another lexicographically defining position that is smaller than $p-2$ but none that is bigger. Considering that S_{p-3} is a suffix of each S_k we know that no S_k can be influenced by the operation at position p because its lcp-value would have to be greater or equal to $p-k-1$. This, however, would imply that the infix $I_{k,p-1}$ is contained in the text at least twice. This creates a contradiction, because then the infix $I_{p-3,p-1}$ would also occur at least twice. Since $I_{p-3,p-1}$ is a prefix of S_{p-3} the lcp-value of S_{p-3} would then have be at least $|I_{p-3,p-1}| = 2$, but according to the definition of the sensitive interval it has to be ≤ 1. Therefore the position $p-2$ must be greater or equal to the positions defining the lexicographical order of all suffixes S_k and none of the S_k can ever be influenced by the operation at position p.

Pseudo-Code illustrating the approach for finding influenced suffixes is presented in **Algorithm 6**.

The average time consumption of this approach is $\mathcal{O}(q \cdot AVG(LCP))$ where q is the number of nodes in the journal tree and $AVG(LCP)$ is the average of the values in the lcp-table. For each node we have to check $AVG(LCP)$ suffixes on average to find the first non-influenced suffix. The average length of the sensitive intervals depends on the average lcp-values of the suffixes. We have q nodes, therefore the time consumption is $\mathcal{O}(q \cdot AVG(LCP))$.

For a text T of length $|T| = n$ the memory consumption for storing the inverse suffix array is an additional $4n$ bytes (on a 32-bit system) because for each position of the text a 32-bit integer containing the suffixes index in the suffix array has to be stored.

Alternatively an approach that works without using the inverse suffix array will be explored. It will become clear, that this alternative approach sacrifices speed for space efficiency and therefore it is important to consider for each problem instance which variation of the algorithm discussed here shall be used. This other algorithm for determining the influ-

Algorithm 6 The algorithm for finding influenced suffixes using sufinv

```
function $influenced\_suffixes(J)$
    $N \leftarrow J.nodes\_inorder()$
    $SA \leftarrow J.suffix\_array()$
    $LCP \leftarrow J.lcp\_table()$
    $SA^{-1} \leftarrow J.sufinv()$
    $In \leftarrow \{\}$ ▷ Set In will contain the positions of suffixes in the text that are influenced
    for all $n \in N$ do
        if $!n.isOriginal()$ then
            $i \leftarrow 1$
            while $i \leq n.pos$ do
                if $\max(LCP[SA^{-1}[n.pos - i]], LCP[SA^{-1}[n.pos - i] - 1]) \leq i - 1$ then
                    break                    ▷ the first non-influenced suffix is found
                else
                    $In \leftarrow \{n.pos - i\} \cup In$
                end if
                $i \leftarrow i + 1$
            end while
        end if
    end for
    return $In$
end function
```

enced suffixes uses only the suffix array, lcp-table and the journal tree.

The basic concept for this different approach is that of iterating through the suffix array and checking for each suffix $SA[i] : i \in 1 \dots n-1$ starting at position $SA[i] = k$ whether it has been influenced. To make sure the k-th suffix in T is not influenced by any operation we have to verify that none of the positions $k+1 \dots k + LCP[i]$ of the text are within an insertion or deletion block. This again uses the knowledge about the lexicographically defining positions that can be obtained from the lcp-table.

The most time intensive step of this algorithm is to search the journal tree for each of the positions $k+1 \dots k+LCP[i]$ and check if the resulting block is an insertion or deletion. Each of this searches costs $\mathcal{O}(\log q)$ comparisons in the best-case (the tree is balanced), where q is the number of nodes of the journal tree. We have to perform n of those searches since we have to search for each suffix and therefore for position of the text. For each position i we also have to do this search for each position of the sensitive interval of the suffix. We estimate the average length of the sensitive intervals of suffixes as $AVG(LCP)$. This amounts to a best-case running-time of $\mathcal{O}(n \cdot AVG(LCP) \cdot \log q)$ which is considerably worse than the $\mathcal{O}(q \cdot AVG(LCP))$ of the approach that uses the inverse suffix array.

A short description of the algorithm without the inverse suffix array in pseudo-code is given in **Algorithm 7**.

Algorithm 7 The algorithm for finding influenced suffixes without sufinv

```
1: function $influenced\_suffixes(J)$
2:     $SA \leftarrow J.suffix\_array()$
3:     $LCP \leftarrow J.lcp\_table()$
4:     $In \leftarrow \{\}$ ▷ Set In will contain the positions of suffixes in the text that are influenced
5:     for $i \leftarrow 1, length(SA)$ do
6:         for $j \leftarrow 1, LCP[i]$ do        ▷ iterate the position from $k+1 \ldots k+LCP[i]$
7:             $n \leftarrow J.get\_node\_at\_pos(SA[i]+j)$
8:             if $!n.isOriginal()$ then
9:                 $In \leftarrow \{SA[i]\} \cup In$        ▷ a position lies within a non-original block
10:                break
11:            end if
12:        end for
13:    end for
14:    return $In$
15: end function
```

After all the influenced suffixes have been found they are removed from the suffix array and added to the array of indices of inserted suffixes. As we described in **Section 2.2.3.1** the lcp-table must be modified to reflect the changes made to the suffix array. The array of inserted suffixes can easily be found by traversing the nodes of the journal tree inorder and checking for blocks of insertions.

A node n containing an insertion block then represents the suffixes starting in the text at positions from $n.pos$ to $n.pos + n.length - 1$. $n.pos$ returns the virtual position of the start of the block represented by n and $n.length$ returns the length of that block.

At this point we have the remaining old suffix array that contains all the shifted suffixes that have neither been deleted nor influenced. We also have the array of suffixes that have to be inserted combined with those suffixes that were influenced and have to have their positions in the suffix array recomputed. The contents of this array (the insertion array) must now be inserted into the old suffix array at the correct positions to yield a suffix array that is lexicographically ordered and synchronous with the text. As a first step the insertion array itself is sorted lexicographically. This can be done using standard sorting algorithms or even applying techniques used for suffix array construction. The idea is to have the old suffix array and the insertion array both sorted lexicographically so that the new suffix array can be computed by merging both arrays together.

2.2.3.3 Merging the Old Suffix Array and the Recomputed and Inserted Suffixes

This section deals with the process of merging the remains of the old suffix array with the already lexicographically sorted insertion array that we computed beforehand. Together both arrays contain all the suffixes from the modified text. The merging is performed in a way not unlike to the well known merge-sort algorithm described for example by Knuth (1998).

We iterate over the old suffix array, comparing the elements to the first element of the insertion array. When an entry in the old suffix array is found that is lexicographically larger than the first element of the insertion array we start removing suffixes from the beginning of the insertion array and insert them before the current entry in the old suffix array until the current entry is not lexicographically larger anymore. When the insertion array is empty or the end of the old suffix array is reached (and we appended the remaining insertion array) the algorithm terminates and the old suffix array now contains the new suffix array that is synchronous with the modified text.

When inserting into the suffix array the lcp-values must be recomputed accordingly, since we again change the successor of a suffix and therefore its common prefix with the new successor must be recomputed as well. **Figure 2.8** illustrates this.

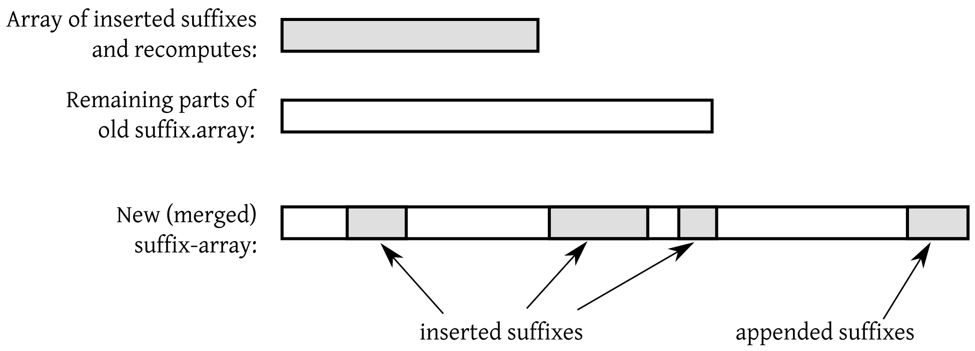

Array of inserted suffixes and recomputes:
Remaining parts of old suffix.array:
New (merged) suffix-array:
inserted suffixes
appended suffixes

Figure 2.8: Merging of the old suffix array and the array containing recomputed and inserted suffixes

Depending on whether the inverse suffix array was available and used it might have to be synchronize as well. When considering the relationship between the suffix array and the inverse suffix array it is apparent that every insertion or deletion in the suffix array translates to a shift in the inverse suffix array. This is because the inverse suffix array stores the suffixes positions in the suffix array and those positions are changed when we insert or delete suffixes in the suffix array.

We will now put together the pieces discussed so far to formulate the synchronization algorithm in its entirety.

2.2.3.4 The Algorithm – Complete Layout

After discussing the essential parts of the algorithm for synchronizing index and text, this section will provide a summary and complete description of the algorithm layout.

The essential steps performed during the synchronization are:

- traverse the journal tree inorder and determine blocks of deletions and shifts for the suffixes as described in **Algorithm 2** and **Algorithm 4**

- iterate over the suffix array deleting and shifting suffixes accordingly
- determine which suffixes from the suffix array are influenced by operations stored in the journal tree by means of either **Algorithm 6** or **Algorithm 7** depending on whether the inverse suffix array is available or not
- delete the influenced suffixes from the suffix array and store them in an additional array into which the suffixes starting in insertion-blocks are also inserted (i.e. generate the insertion array)
- sort the insertion array lexicographically
- merge the insertion array into the old suffix array to generate the new suffix array that is now synchronous with the text and modify the lcp-table accordingly
- synchronize the inverse suffix array if it was used

A complete overview of the algorithm is given in pseudo-code in **Algorithm 8**. Here we use the information from the inverse suffix array in combination with the inorder traversal of the journal tree to simultaneously apply the correct shifts and determine deleted, inserted and influenced suffixes. Finding the influenced suffixes is done in a similar manner to the one presented in **Algorithm 6**. The sorting and merging is not described in detail here. Those two functions must be implemented in such a way that the lcp-table and inverse suffix array are modified accordingly.

We have formalized the complete layout of the synchronization algorithm and will now start describing the tools needed to provide the implementation. As such we will begin by discussing the journal string class we want to implement in SeqAn (Döring *et al.* , 2008).

2.2.4 The Journal String Class

We discussed the general principles of journaling and the fundamentals of the synchronization algorithm, and will now describe the design of the journal string class that was developed as part of this thesis.

As this thesis relies an SeqAn (Döring *et al.* , 2008) as a basis for the implementation, the journaling functionality needed for this purpose was also designed with the goal of incorporating the results into the SeqAn library framework. Therefore, a new class of strings had to be designed that would work with the SeqAn framework, which will be called the journal string class.

This section will only cover the basic design principles used for the journal string class. It will not deal with implementation details since those will be provided in **Section 3**. There are a couple of things to consider when designing a string class with journaling capabilities that uses a journal tree to store the journaled operations. First of all it is clear that the advantage of saving computation time by not actually having to perform all the operations stored in the

Algorithm 8 The algorithm for synchronizing text and index - complete layout using sufinv

```
procedure $synchronize\_index(J, I)$
    $N \leftarrow J.nodes\_inorder()$
    $SA \leftarrow J.suffix\_array()$
    $LCP \leftarrow J.lcp\_table()$
    $SA^{-1} \leftarrow J.sufinv()$
    $cs \leftarrow 0$                                  ▷ cs will hold the current shift
    $Idx \leftarrow \{\}$                              ▷ Idx will hold the inserted and influenced suffixes
    for all $n \in N$ do
        if $n.isOriginal()$ then
            for $i \leftarrow n.pos, n.pos + n.length - 1$ do
                $SA[SA^{-1}[i]] \leftarrow SA[SA^{-1}[i]] + cs$          ▷ Shifts the current block
            end for
        else
            if $n.isDeletion()$ then
                for $i \leftarrow n.pos, n.pos + n.length - 1$ do
                    delete $SA[SA^{-1}[i]]$ and update $LCP$ and $SA^{-1}$ accordingly
                    $cs \leftarrow cs - n.length$                 ▷ Update the current shift
                end for
            else
                for $i \leftarrow n.pos, n.pos + n.length - 1$ do
                    $Idx \leftarrow \{SA[SA^{-1}[i]]\} \cup Idx$        ▷ Insert the suffixes in Idx
                    $cs \leftarrow cs + n.length$                 ▷ Update the current shift
                end for
            end if
            $i \leftarrow 1$                                   ▷ Loop variable
            while $i \leq n.pos$ do
                if max$(LCP[SA^{-1}[n.pos - i]], LCP[SA^{-1}[n.pos - i] - 1]) \leq i - 1$ then
                    break                               ▷ First non-influenced suffix found
                else
                    $Idx \leftarrow \{SA[SA^{-1}[n.pos - i]]\} \cup Idx$
                end if
                $i \leftarrow i + 1$
            end while
        end if
    end for
    sort $Idx$ lexicographically
    merge $SA$ and $Idx$
end procedure
```

journal will have a drawback. This drawback comes in the form of increased cost for querying data from the journal string class, since for each query the journal tree has to be traversed and the correct block of data must be found before returning the queried value. We already described this property of journaling data structures for the journal tree in **Section 2.1.4**.

To minimize this effect some methodologies may be applied, such as balancing the journal tree in order to guarantee $\mathcal{O}(\log q)$ for requests (in a tree with q nodes). Another possibility is optimizing the tree locally by merging deletion blocks that are in nodes succeeding each other when the tree is traversed inorder. The goal in doing this is to minimize the number of nodes in the tree. Deletions and insertions covering the same ranges of positions cancel each other out so that this effect can also be used to reduce the number of nodes in the journal tree. Possible optimizations are discussed in **Section 5.3.1**.

2.2.4.1 Elements of the Journal String Class

Before explaining the principles and designs of the journal string class in more detail, a short summary of all the necessary parts of the class will be given. The following elements will be part of an instance of the journal string class:

- a reference to the underlying string for which the changes are be journaled
- a structure storing the journal tree that contains the information about journaled operations
- a reference to the insertion string, a string that will store all the inserted values that are referenced by nodes in the journal tree

The class will consist of the three major elements that together provide the desired functionality. The reference to the underlying string is obviously necessary since we need a way to access the original string. This is important for data access on nodes in the journal tree representing blocks of the original string.

The reference to the underlying string is also important when we want to make the changes to the string persistent by actually applying the journaled operations to the underlying string. This process, which is called flattening the string, is explained in **Section 2.2.4.2**.

The existence of a reference to the insertion string shows that our design of the journal tree will differ slightly from the general concept described earlier. This difference is that all blocks of data that are inserted will be stored consecutively in the insertion-string. This makes it easier to maintain the data since we do not have to maintain pointers to external strings in the nodes that represent insertions. We thereby avoid potenial problems with invalid pointers.

For the nodes in the journal tree this means that we do not store pointers or references to the inserted blocks of data. Instead for each node we store its index within the string of which the node's block is a substring. Whether this is the underlying or the insertion string is determined by the type of node.

An insertion node will store an index of the insertion string and a node representing a block of original string will store an index of the underlying string. Aside from that, the nodes will need to store structural information about the tree topology, this includes references to the parent node and child nodes. The tree itself will be stored in a vector of nodes, using indices in that vector to reference parents and child nodes rather than using pointers. In **Section 3** we will describe this way of encoding the tree in a string in more detail. All in all, the journal tree will consist of nodes that contain the following information:

- the virtual position of the start of the block that is represented by the node
- a boolean attribute denoting whether the node represents an original or an insertion block (for deletion blocks this attribute is unnecessary since they do not point to any strings)
- a reference to the node's parent node in the tree (its position in the node-containig vector)
- references to the child nodes in the tree (positions within the node-containig vector)
- an operation object storing data about the position and length of the operation that is represented by the node

Figure 2.9 illustrates a typical instance of the journal string class complete with underlying string, insertion string and journal tree.

2.2.4.2 Functions of the Journal String Class

Now that the elements that will make up the journal string class are defined we will discuss the functionality it provides. Apart from the elements required for a string class to be used with the SeqAn interface there are some specific functions that we will describe in more detail. Those functions are:

1. data access through the journal tree
2. inserting and deleting blocks, i.e. journaling those operations correctly and modifying the journal tree accordingly
3. persisting journaled operations by applying them to the underlying string

Data Access for the journal string class is realized by traversal of the journal tree until the block containing the requested position is found.

The requested position is compared to each node n evaluating if it lies within the node's interval of positions $[n.pos \dots n.pos + n.length - 1]$. If that is the case the correct block has been found and the value can be returned.

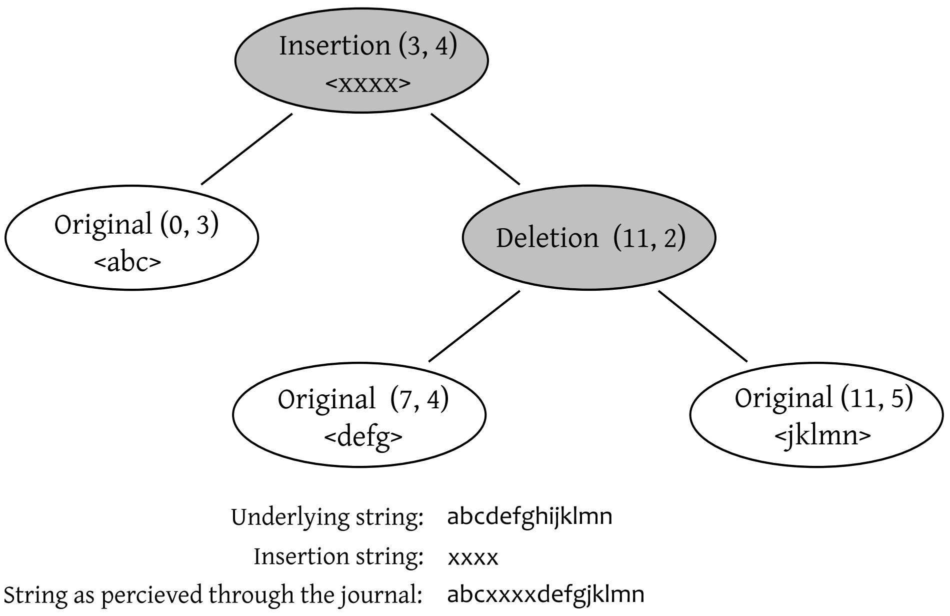


Figure 2.9: A more complex example of a journal tree.
This journal tree is the result of an insertion of length 4 at position 3 inserting the string "xxxx" into the underlying string "abcdefghijklmn" and a deletion of length 2 at position 11 of the modified string "abcxxxxdefghijklmn". Note that the two values in the nodes are the beginning and the length of the block of data represented by the node.

If the requested position is smaller than $n.pos$ the process is repeated with the predecessor of n in the sequence of nodes that is the inorder traversal of the tree, if it is larger than $n.pos + n.length - 1$ the successor will be used. This process is repeated until a node is found that contains the position. The tree traversal is realized via modification of an iterator on the vector containing the nodes. For this iterator the functions `goNext(it)` and `goPrev(it)` are implemented, that set the iterator to point to the inorder successor or predecessor of the current node by traversing the tree accordingly (inorder).

Figure 2.10 illustrates the tree traversal.

The process used for data access is described as pseudo-code in **Algorithm 9**.

At this point another possible journaling functionality comes into play. The algorithm for synchronization relies on deleting, inserting and shifting as its primary operations. The logical consequence is to journal not only the first two but also the last kind of operation.

Therefore, the journal-string class was modified to include a set of pairs containing shifts that are encoded in the same way as we described in **Section 2.2.3.1**. Those pairs store information about the shifts that need to be applied to values before they are returned from the string. Of course such a feature is only useful for journal strings that contain values that can be shifted, i.e. the `operator+` must be defined on the value type. We will only use this for the suffix array and inverse suffix array, which are strings of integer values.

In this way the time consumption for the synchronization algorithm can be further reduced, because the shifts do not need to be applied to the suffix array directly. They can rather be saved within the associated journal string and are applied on demand when data is

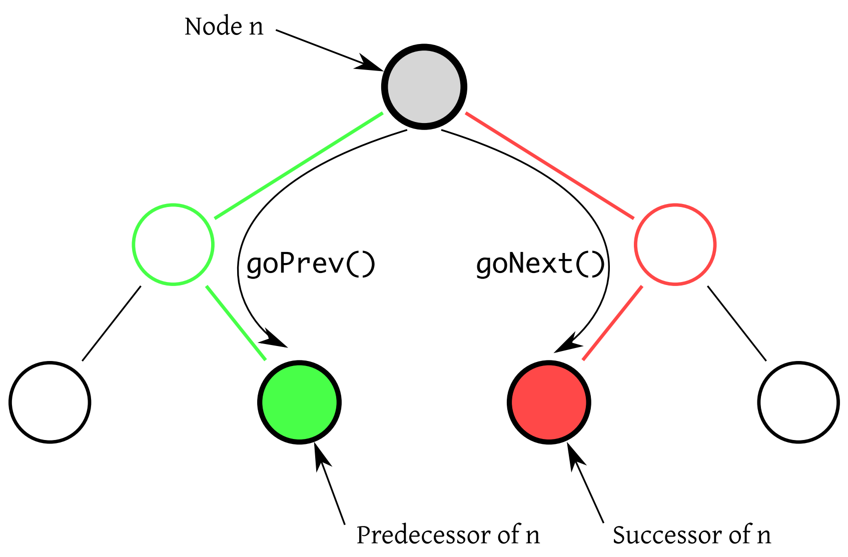


Figure 2.10: Traversal of the journal tree via the `goPrev()` and `goNext()` functions.

Algorithm 9 Getting data from the journal tree

```
1:  function $value$(J, pos)    ▷ returns the value stored at position pos in journal string J
2:      $n \leftarrow J.journal\_tree\_root()$    ▷ get iterator to root node of journal tree
3:      if $pos < n.pos$ then
4:          while $!n.inInterval(pos)$ do    ▷ if pos is not in block of n
5:              $goPrev(n)$    ▷ make n point to predecessor node
6:          end while
7:      else
8:          while $!n.inInterval(pos)$ do    ▷ if pos is not in block of n
9:              $goNext(n)$    ▷ make n point to successor node
10:         end while
11:     end if
12:     if $n.isInsertion$ then    ▷ inserted values are stored in the $insertion\_string$
13:         return $J.insertion\_string[n.pos + n.\text{offset}(pos)]$
14:     else
15:         return $J.underlying\_string[n.pos + n.\text{offset}(pos)]$
16:     end if    ▷ n.offset(pos) returns the index of pos within the block represented by n
17: end function
```

requested from that journal string.

When qerying a journal string for data and this journal string stores shifts for the values in its underlying string the function `getshift()` must be called for each access. `getshift()` uses binary search on the set of shifts to determine the shift by which the return value must be modified before it is returned. This increases the time consumption for data access by $\mathcal{O}(\log q)$ where q is the number of pairs in the set of shifts. When using the on-demand shifting functionality of the journal string class to speed up the synchronization algorithm we always have to consider the potential disadvantages, such as the slower data access. When we want to make many queries to the journal it is most likely better to apply the shifts directly. Initially this will be more time consuming than simply storing the shifts but the investment will likely be amortized through the reduced time consumption for data access.

The Time Consumption of finding the right block of data in the manner described in **Algorithm 9** is $\mathcal{O}(\log q)$ for a tree with q nodes in the average case. When the tree is unbalanced it is possible to have a structure resembling a linked list (degenerate tree) that contains the nodes inorder from the root node to the right most child node. In this worst case the time consumption is $\mathcal{O}(q)$ because the tree has height q. We described this kind of degeneration of the tree structure in **Section 2.1.4** and illustrated the effect in **Figure 2.5**.

In addition to the time consumption for finding the correct block we have to consider the time needed for requesting data from the underlying string or the insertion string. When they are continuously stored this is $\mathcal{O}(1)$ if we know their memory adress and the index we want to query for. Depending on whether the journal string contains a set of shifts Sh with $|Sh| = r$ another $\mathcal{O}(\log r)$ is added for determining the correct shift for the return value. This can lead to a time consumption of $\mathcal{O}(\log q + \log r) = \mathcal{O}(\log q \cdot r)$ for data access in the average case. Because of the above mentioned possible deterioration of the journal tree the time consumption can potentially be worse, depending on the topology of the journal tree.

Inserting and Deleting Blocks in the journal string represents the two central functionalities provided by the journal string class. To correctly perform this operations the journal tree must be adapted to include the operation that is performed and the relevant data must be copied to the insertion-string if the operation was an insertion.

The first step when performing an operation on a journal string is to find the node representing the block that the operation will take place in. For an operation o starting at position p the node containing p within its associated block must be found.

The process is similar to the traversal of the journal tree used for data access (see **Algorithm 9**), with the difference that the desired return value is the iterator to the node in question and not the position within the block represented by the node.

In principle each operation that is performed within the block of node n does not influence the topology of the two subtrees rooted in the left and right child node of n.

Inserting Blocks of Data is done by inserting a new node n_i representing the insertion in the place the containing node n was in, effectively replacing n. The containing node n will be split into two nodes at the insertion position and the two resulting nodes will become the left and right child of the insertion node. The left part of n will have the subtree formerly rooted in the left child of n as its left subtree and the right part will have the corresponding right subtree as its right child. This new insertion node n_i represents the insertion and contains the information about length, position and index in the insertion-string of the operation. The data that is inserted will be appended to the insertion-string.

After an insertion of length m at position p all the blocks starting behind the insertion block in the string must have their virtual positions updated since they are now shifted by m. (see **Section 2.2.3.1**) This can be done using the tree iterator described in paragraph Data Access in **Section 2.2.4.2**. We start at the new insertion node n_i and use the function `goNext()` to visit all nodes succeeding n_i in the inorder traversal of the journal tree, updating their virtual positions in the process. The concept of inserting a node into the journal tree is visualized in **Figure 2.11**.

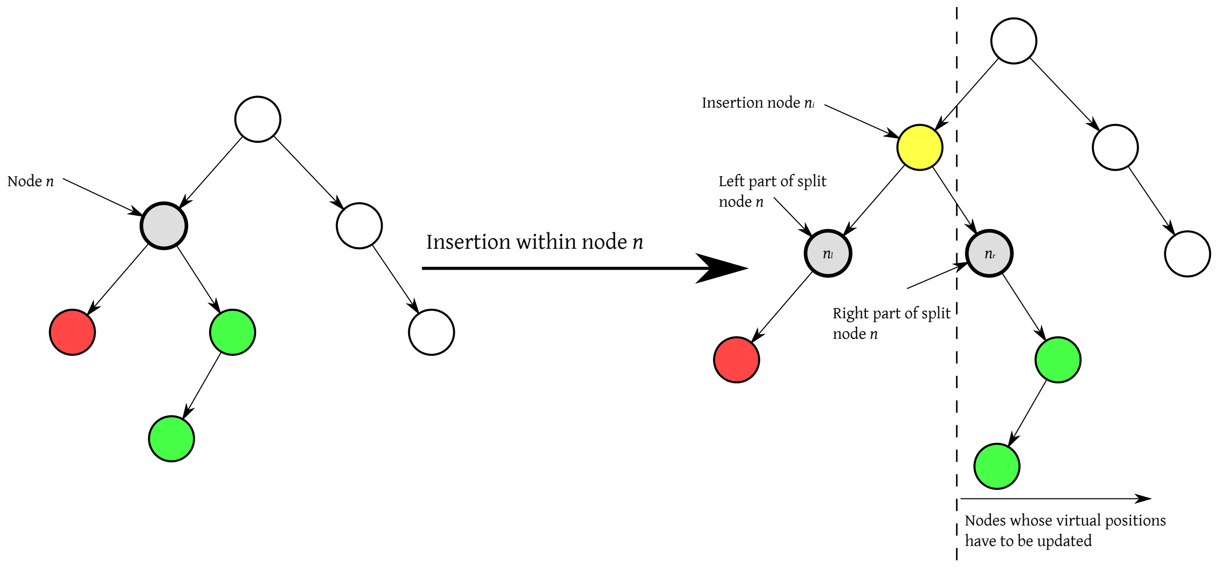

Node n
Insertion within node n
Insertion node nᵢ
Left part of split node n
nₗ
nᵣ
Right part of split node n
Nodes whose virtual positions have to be updated

Figure 2.11: Splitting of node n when inserting node n_i at a position within the block of n.

Deleting Blocks of Data is done by first finding the node containing the beginning of the block of data that is deleted, and then splitting that node in a way similar to the process for inserting a block of data. For the correct modification of the right and left subtree it is important to consider the length of the block that is deleted, because this block of data could span several nodes that all have to be modified accordingly.

If the deletion takes place within a single node, the process simply splits the node at the beginning and end of the deletion block and replaces it with a deletion node that has the two subtrees as its children. This is illustrated in **Figure 2.12**.

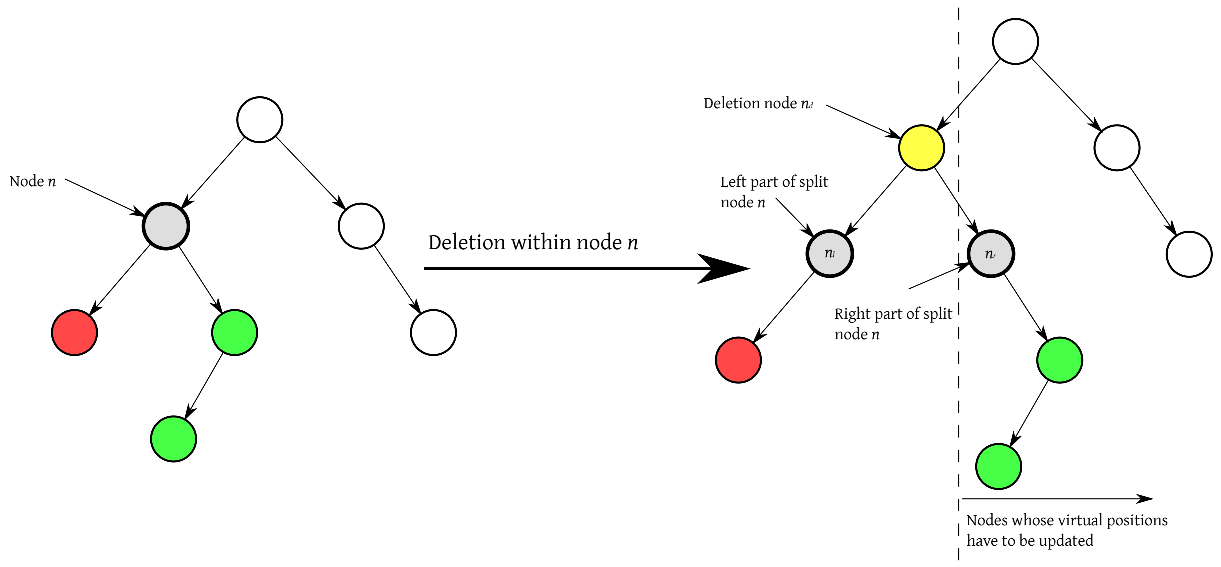

Node n
Deletion within node n
Deletion node n_d
Left part of split node n
n_l
n_r
Right part of split node n
Nodes whose virtual positions have to be updated

Figure 2.12: Splitting of node n when deleting a part of the block of n.

As an example we will consider a deletion taking place in node n from positions k_1 to k_2. The following holds for the positions:

$$n.\text{position} \leq k_1 < k_2 < n.\text{position} + n.\text{length}$$

which means that the deletion is completely contained in the block of node n. In this case the node n will be split into the two nodes n_l and n_r which represent the blocks $[n.\text{position} \ldots k_1]$ and $[k_2 \ldots n.\text{position} + n.\text{length} - 1]$. They will be the right and left children of the deletion node n_d that is used to replace n within the tree.

When the length of the block that is deleted makes it span more than one node, it is obvious that this cannot be implemented as a single operation. One possible way of solving this problem is dividing the deletion into a number of sub-deletions such that each of those sub-deletions can be applied in the way illustrated in **Figure 2.12**.

We will illustrate this process by an example: For a deletion from position k_s to k_e that spans the three nodes n_1, n_2 and n_3 in the journal tree the following holds:

$$\begin{aligned} n_1.\text{position} &< k_s \\ n_1.\text{position} + n_1.\text{length} - 1 &> k_s \\ n_2.\text{position} < n_3.\text{position} &< k_e \\ n_3.\text{position} + n_3.\text{length} - 1 &> k_e \end{aligned}$$

This large deletion is now split up into the three sub-deletions (encoded by pairs of begin and end positions). Those sub-deletions are $(k_s, n_2.\text{position})$, $(n_2.\text{position}, n_3.\text{position})$ and $(n_3.\text{position}, k_e)$ each of which can be applied using the process illustrated in **Figure 2.12**.

Figure 2.13 provides a visual example of such a deletion spanning multiple nodes in the journal tree..

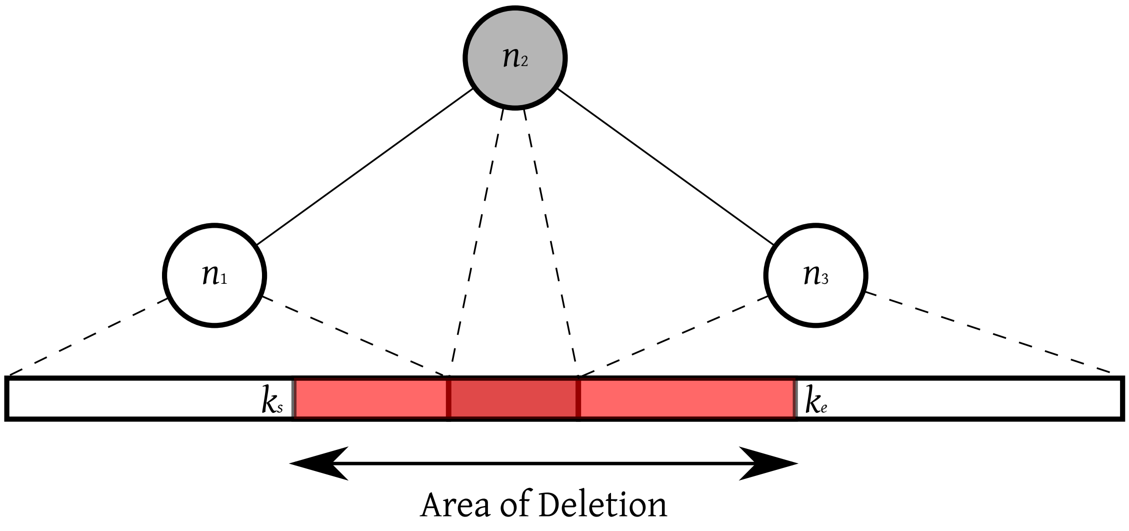


Figure 2.13: Deleting a block of data that spans multiple nodes from the journal tree.

Of course all the nodes that lie inorder after the node where the deletion occurs also have to have their virtual positions adapted in a similar way, as it is done when performing an insertion. Since the large deletion is split into consecutive sub-deletions the resulting shift will be computed as the sum of the lengths of those sub-deletions and will yield the length of the original large deletion.

Persisting the Changes is the process of applying all operations stored in the journal to the underlying string. We have already hinted at this process of flattening the journal string and will now explain this concept in more detail. The name flattening was chosen for this process because, as a result of persisting the changes, the journal tree will be reduced to its initial state, which contains only one node, thereby making the tree flat. The persisting of journaled changes can become necessary when the journal tree has grown too much and the resulting time consumption for data access becomes unfeasible (see **Section 2.2.4.2**). It might also be done prior to saving the string to the hard drive where it will be stored continuously.

When we start applying the changes to the underlying string by inserting and deleting data we will trigger the same time-costly move operation described in **Section 1.2.2**. If we would simply apply the changes in increasing order of their position in the string this could amount to a tremendous increase in time consumption for the flattening.

Consider the case where we want to apply two insertions of length 1 at positions k_1 and k_2 in a string S of length n ($0 \leq k_1 \leq k_2 < n$). If we first apply the insertion at k_1 we have to move the substring $S[k_1 \ldots n-1]$ by one position in memory because after the insertion it will correspond to the substring $S[k_1+1 \ldots n]$. When we then apply the second insertion the same effect triggers for the substring $S[k_2 \ldots n-1]$. In this way we have to move the substring $S[k_2 \ldots n-1]$ two times by one position (here n always stands for the length of the string before the operation in question).

Effectively we would have to invest all the computation time the journaling has saved us if we were to perform the operations in such a way. This could be remedied to a certain extend by copying the string blockwise into a new location because we can make use of the knowledge that the resulting move operations for a substring $S' = S[k \dots l]$ of S will be equal to the accumulated move operations resulting from all the operations that are applied at positions smaller than k. The correct position of a block is its virtual position, which we can easily retrieve from the journal tree.

We have chosen an approach where we perform the flattening in place with respect to the underlying string. We will first extend the string to make sure it can store all the blocks that have to be inserted and then reorganize the string block-wise starting from the end. **Figure 2.14** will help to clarify the process.

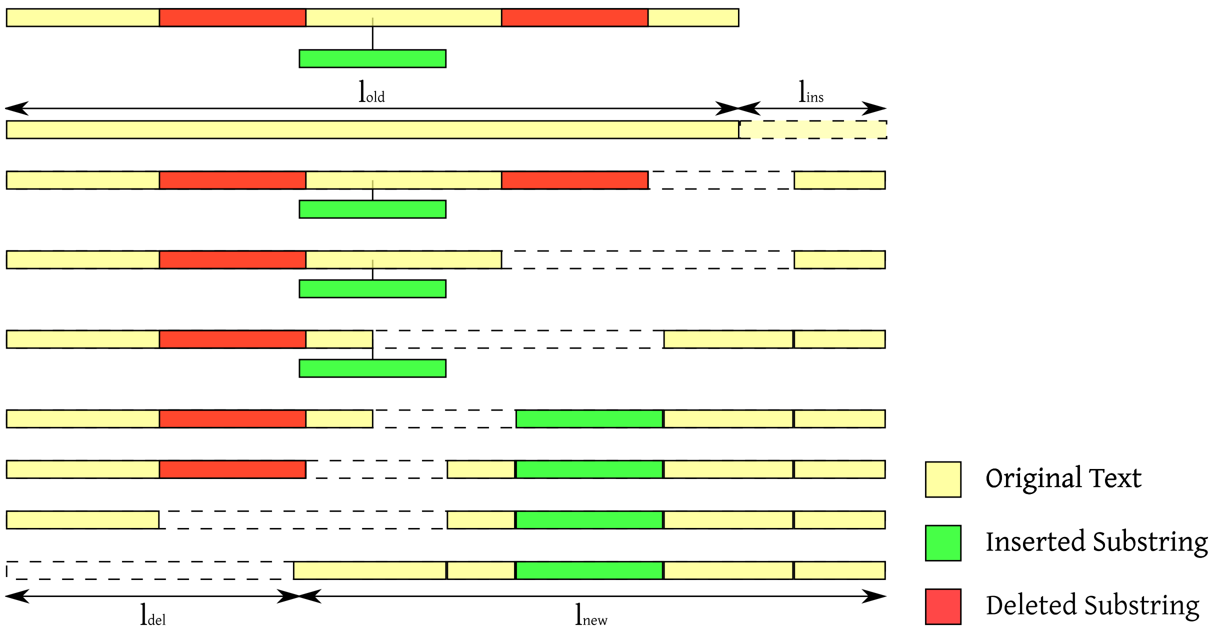


Figure 2.14: Flattening a string with two deletions and one insertion.
The string is handled block-wise and the whole process is performed in-place, after resizing the string.

The first step is to find the length of the modified string which we have to resize the original string to.

The journal stores the information about all operations that are to be persisted, this also gives us the new length of the string after the flattening. We will call this length l_{new} while the old length of the underlying string will be called l_{old}.

We will start by resizing the underlying string to make it sufficiently big to contain all modifications. For reasons that will be explained soon we will resize the underlying string to $l_{old} + l_{ins}$ where l_{ins} is the combined length of all insertions stored in the journal tree.

We will then start by filling this now resized string block-wise, beginning at the last position. We do this to guarantee that we will not overwrite parts of the string that should not be deleted. Because we have resized the string to $l_{old} + l_{ins}$ we have enough free space at the end to store all the insertions, even if they would all lie at the last position of the string.

Note that we cannot resize the string to l_{new} directly because then we could loose data that is still needed. A simple example to demonstrate this effect is that of a journal string that has a single deletion performed on it. When we have deleted exactly one value from the string we obviously have $l_{new} = l_{old} - 1$, but since the deletion might have occurred at a position different from the last position of the string we cannot simply resize the string to be one value shorter because we would lose the last value.

By constructing the persisted string starting at the end we guarantee that no information is lost. After we have moved all blocks to their correct positions relative to the end of the resized string we will have to remove the first l_{del} (the sum of the lengths of all deletions) positions from that string. This is done because we resized it to $l_{old} + l_{ins}$ but the actual length of the persisted string is $l_{new} = l_{old} + l_{ins} - l_{del}$. The new journal tree will now contain only one node representing the block from positions 0 to l_{new}. The deletion of the first l_{del} letters can be achieved easily by modifying the memory address of the begin of the string stored in the instance of the string class.

Another option is to modify the node in the journal tree ins such a way, that it represents the block from l_{del} to $l_{old} + l_{ins}$ in the underlying string. In this way the first l_{del} position of the string are not deleted, but simply ignored by the journal tree.

3. Implementation

In the above sections we have covered the general principles of the journal string class and the journaling functionality provided by it. We have discussed the necessary elements of the journal string and its interface functions for data access, insertion, deletion and persisting the journaled changes. We have also talked about the special requirements for large deletions that span multiple nodes in the journal tree. We will now consider some details of the implementation within the context of the SeqAn C++ Library.

3.1 The SeqAn C++ Library

The SeqAn C++ Library (Döring *et al.*, 2008) provides a lot of functionality for bioinformatical problem solving. To get a better overview of all the algorithms and data structures provided by SeqAn you can visit the SeqAn Homepage and have a look at the libraries documentation.

We will now have a short look at the parts of SeqAn that are of interest for this thesis. In particular we will concern ourselves with one of the core concepts of SeqAn, that is the use of templates and template specializations.

3.1.1 Templates and Template Specialization

C++ employs the concept of templates to implement strongly-typed generic classes. A container class is implemented for a generic type that is given as a template argument. Consider the standard `vector` class that holds values of a template-determined type in a contiguous block of memory. For example, `vector<char>` holds characters while `vector<int>` holds integer values. Both instances of the `vector` class template denote different types at compile time. Yet, both can be used with the same generic interface functions, independent of the type of the values contained in the vector.

For such a container class, functions can be implemented as function templates without knowledge of the specific type they will be used with. This allows for code reuse through generic programming. Of course, certain function templates have restrictions on the types they can work with. For example, a function that uses the less-than or equal operator $\leq$ on two generic objects will only work for objects of a type that defines comparability, i.e. that implement a matching `operator<=`.

SeqAn expands on this concept by using the fact that different implementations can be provided for different template parameterizations of a class. This allows for specialization of functions and classes based on the template arguments. The compiler will always use the most specialized implementation of a function it can find for the instance it is invoked on.

We can use the template parameters of an instance of a class template to define which implementation of a function is used, if there is more than one implementation available.

This provides a means of specializing a class through its template parameters and we can provide more optimized versions of a function for specific combinations of template parameters.

By using template specialization we implement a form of subclass polymorphism that is evaluated at compile time. Consider the example code from **Listing 3.1**. Here we have two classes `A` and `B` that are used as template parameters of the class template `Echo`.

```cpp
struct A {};
struct B {};

template< typename T > struct Echo;

template<> struct Echo<A> {
  void f() const{ std::cout << "Instance of A" << std::endl; }
}

template<> struct Echo<B> {
  void f() const{ std::cout << "Instance of B" << std::endl; }
}

template< typename T > void foo(){
  Echo<T>::f();
}

int main(){
  foo<A>();  // prints “Instance of A”
  foo<B>();  // prints “Instance of B”
}
```

Listing 3.1: Template specialization example

The implementations of the class template `Echo` is specialized for the two possible template parameters and the member function `f()` will yield different results based on the template parameters of the instance of echo.

The global function `foo()` is now generic for instances of `Echo` but is specialized with respect to the template parameter `T`. For `T==A` the corresponding implementation of `Echo<T>::f()` which is `Echo<A>::f()` will be used and for `T==B` the implementation of `Echo<B>::f()` will be used.

In this way the function `foo()` is specialized based on its template parameters and this polymorphism will be resolved at compile time, when the compiler decides which implementation of `Echo<T>::f()` to use.

This approach has an advantage over polymorphism using virtual functions and class inheritance because virtual functions provide runtime polymorphism which has to be evaluated every time a virtual function is called. When the decision which implementation to use is done at compile time we gain a performance boost when executing the program.

The concept of using template parameters to specialize global functions is used heavily within SeqAn, it allows for generic programming and makes it easy to implement a new class specialization simply by specializing all the necessary global function to perform their task in the way desired for the new class.

Often, some or most of the generic global functions do not need to be specialized and in this way a lot of code and work can be saved when implementing a new template specialization of a class.

3.1.2 Different Kinds of Strings

The central method for storing sequence information in SeqAn is the string class. SeqAn currently provides eight string classes that differ in various implementation details, for example in their method of storage. The string class developed for this thesis will be added to this set of strings provided by SeqAn.

Each of those string classes is a specialization by means of template parameters to the string class template. To facilitate the desired behavior in the different string classes several global functions have to be specialized.

Since this is not the main topic of this thesis we will not discuss the different string classes in more detail here. It is sufficient to know that each of those have their usage scenarios which they are best suited for. We will define the usage scenario for the journal string class in the following sections.

3.2 The Journal String Class

In this section we will now discuss implementation details of the journal string class. Before we begin, a short summary of the general design and capabilities of the journal string class will be given.

3.2.1 Elements of the Journal String

As was discussed before an instance of the journal string class will consist of the following parts:

- a reference to the underlying string
- the insertion string that holds all inserted values
- a string of nodes that encodes the journal tree

- optionally a string of pairs of positions and shifts that are applied to the values before they are returned

3.2.2 Template Arguments

Since templates and template specialization is an integral part of the SeqAn concept it is also used to implement the journal string class and to integrate it with the SeqAn programming library.

As we discussed in **Section 3.1.1** the journal string class will be designed to be an additional string class within SeqAn which will be specialized for its desired behavior by its template parameters. We will focus on the template arguments of the journal string class and how they affect the behavior of an instance of this class.

The source code shown in **Listing 3.2** illustrates the use of template parameters to specify the desired behavior of the journal string class.

```
template< typename TValue, typename TSloppySpec, typename TShiftSpec,
    typename TUnderlyingSpec, typename TInsertSpec>
struct JournalConfig{...};

template < typename TValue, typename TConfig >
class String< TValue, Journal< TConfig > > {...}
```

Listing 3.2: Template parameters of the journal string class

In SeqAn every type of string is defined by its template parameters. The general usage is `String< TValue, TSpec >` which has two template parameters defining the contents and the type of the string. The code `String< TValue, TSpec >` defines an instance of the string class that stores values of type `TValue` (e.g. `char` or `Dna<>`, etc.) and its behavior is defined by the parameter `TSpec` which will be used to chose which kind of string this instance shall have.

SeqAn provides different types of strings that are specified by using different template parameters for the `TSpec` to chose between strings stored for example in blocks, arrays or in a memory-mapped file on the hard disk. The default value for `TSpec` is `Alloc<>` which is a string stored continuously in memory.

To define an instance of string to be of the journal string class developed in this thesis, the template parameter `TSpec` must be set to `Journal< TConfig >`. This defines the string to be a journal string that itself has another template parameter `TConfig` which we use to specify the behavior of the journal string class further.

As can be seen in **Listing 3.2** a possible value for the template parameter `TConfig` could be `JournalConfig` which has 5 template parameters defining its behavior. Those parameters are explained in **Table 3.1**.

An example of how an actual instantiation of a journal string might look like is given in **Listing 3.3** where a journal string holding values of type `char`, is `Sloppy` and does not apply

TValue	the value type of the string, is identical to the TValue parameter of the string
TSloppySpec	can be Strict or Sloppy and specifies if references to values from the journal string can be modified or not
TShiftSpec	can be True or False and specifies whether the values must be shifted according to the shifts stored in the journal string prior to returning them
TUnderlyingSpec	contains the TSpec template parameter of the underlying string
TInsertSpec	defines the TSpec template parameter for the string containing the inserted values

Table 3.1: Template parameters of JournalConfig

shifts is instantiated.

```
String< char > underlying = "gattaccatac";
String< char, Journal< JournalConfig< char, Sloppy, False > > > journaled(
    underlying );
```

Listing 3.3: Example of instantiation of a journal string.
Some template parameters are not given and will be set to their respective default values by the compiler.

3.2.2.1 Data Access

The template parameters described in the above section define the behavior of the journal string concerning the return values for data access as well as the storage type for the insertion string. The defining parameters for the return value are `TShiftSpec` and `TSloppySpec`.

`TShiftSpec` is a parameter, that defines whether or not the return value is shifted using the shifts stored with the journal string before it is returned, it can have the two possible Values `True` and `False`. `TSloppySpec` defines whether the return value, if it is a reference, can be modified directly or not. It can have the value `Sloppy` or `Strict`, meaning that when a reference to a value in the string is returned that reference will be non-`const` (`Sloppy`) or `const` (`Strict`).

In this way it is possible to create a journal string that has journaled operations (the deletions and insertions) as well as direct modifications of values in the string (the non-const references returned by a `Sloppy` journal string).

When applying journal strings to the synchronization of an index to a changed text it is not favorable to have unjournaled modifications. The index synchronization algorithm relies on all the changes to the string being journaled in the journal tree and it is necessary to use a `Strict` journal string for the text we want to synchronize with its associated index.

When the return values have to be shifted before they are returned it is obvious that the `TSloppySpec` will have no influence on the behavior of the string. The data access function will not return a reference to a value in the string. Rather it will return a temporary object because a shift has to be applied to the value stored in teh string and the return value is computed within the data access function as a local variable.

Table 3.2 gives an overview of the different combinations of template parameters and the associated return values of the data access function.

	`Sloppy`	`Strict`
`True`	`TValue`	`const TValue`
`False`	`TValue &`	`const TValue &`

Table 3.2: Template parameters and the return value of data access.
This table shows the return types of the data access for different combinations of template parameters. Note that for the row `True` it does not really matter that the return value for the `Sloppy` version is not `const` since it is always a temporary object and any modifications applied to it will not influence the string.

Template specialization is used to implement this different behaviors for data access by overloading the `operator[]` or `value()` function of the string class to behave differently depending on the template parameters of the instance they are applied to. Those functions are the basic data access functions for string classes in SeqAn. The specialization is applied at compile time, when the compiler chooses the appropriate implementation of those functions depending on the instance's template parameters (see **Section 3.1.1**).

3.2.3 Encoding the Journal Tree

The journal tree is the most important part of the journal string, since it is essential for the string's journaling functionality.

As was described in **Section 2.1** the journal tree is a binary search tree consisting of nodes that represent blocks of the string. Those blocks can be parts of the original string, inserted substrings or deleted substrings. In the implementation of the journal string class we encoded the journal tree in a string of nodes. Instead of containing pointers to their child and parent nodes the nodes now contain the indices of their child and parent nodes within the string that encodes the tree. **Figure 3.1** illustrates the encoding of the journal tree as a string of nodes.

3.2.4 Implementing the Iterator

We will discuss some aspects of the iterator provided for the journal string class in this section. An iterator over a string is an object that references a specific position within the string, it can be incremented and decremented by arbitrary values and it can be used to access and eventually modify the value it references in the string. Since the journal string is not stored

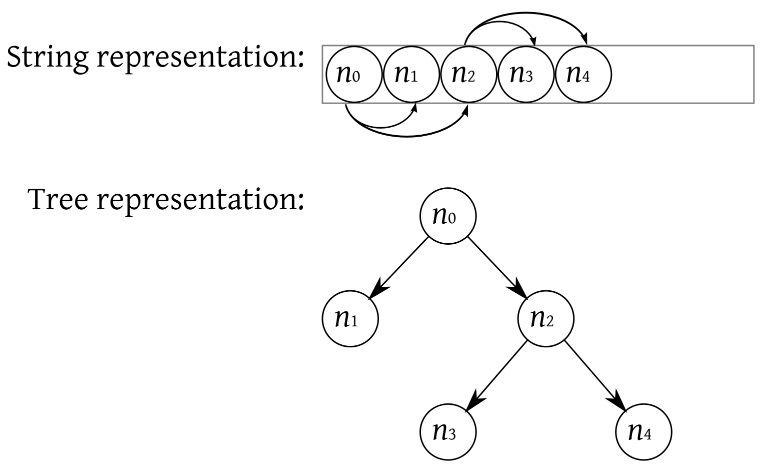


Figure 3.1: Encoding the journal tree as a string of nodes.

continuously within memory, the iterator, which is also used for sequential access, must perform some checks before incrementing or decrementing.

In a continuously stored string iterating is simple. All we need to do is an arithmetic operation where the iterator's memory address is shifted by the size of the increment times the length of one entry in the string. **Table 3.3** illustrates the general concept.

Memory address	00	04	08	0C	10	14	18
Value	**g**	a	t	t	a	c	a
	↑						

00	04	08	0C	10	14	18
g	a	t	t	**a**	c	a
				↑		

Table 3.3: Iterating over a continuous string of characters.
Note the hexadecimally encoded memory addresses. In this example, each value has a length of 32 bit, setting the values apart by 4 bytes. The situation shown in the right table is the result of incrementing the iterator by 4. The currently referenced character is marked in **bold**.

The iterator for the journal string must have information about the current block of data and the corresponding node in the journal tree. When we increment or decrement the iterator we have to check whether the beginning or end of the current block has been reached. If an increment or decrement operation would cause the iterator to reach the limits of the current block of data the predecessor or successor of the current node in the inorder traversal of the journal tree has to be found. The iterator must then be modified to point to the block of data that is represented by the new node. Basically the iterator for the journal string has to perform checks for each of its operations. It must determine whether the current block has to be changed and then find the next block and reposition itself to point to the right position in memory. See **Figure 3.2** for an illustration of this process.

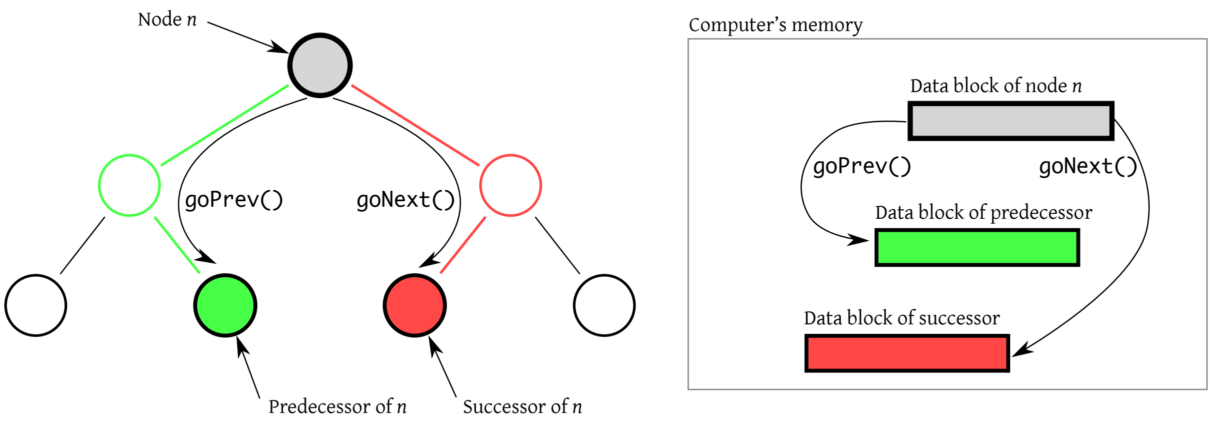


Figure 3.2: Iterating the non-continuously stored journal string.
Note how the blocks of data that appear in sequence in the journal string are stored non-continuously in the computers memory. When iterating over the journal string we have to check if the current node has been changed and the iterator must then be modified to point to the correct memory address of the new data block.

3.3 The Synchronization Algorithm

The synchronization algorithm described in **Section 2** can be implemented in different ways. As we already discussed beforehand there are two ways of achieving index synchronization. One option utilizes the information from the inverse suffix array and the other one works without the inverse suffix array. We implemented both versions and used preliminary benchmark results to decide which implementation to use. Of these two, the one using the inverse suffix array is favorable since it runs significantly faster than the one without.

This advantage in performance is mainly due to the fact that the inverse suffix array directly provides information about lexicographically adjacent suffixes given the position of an operation. Finding the influenced suffixes then becomes easy since all suffixes potentially influenced by an operation at a position k can be found in the inverse suffix array at the positions preceding k.

Without this information we have to iterate over the suffix array and check for each suffix if it is influenced by an operation. This check is done by inspecting the journal tree of the text which yields a significantly longer running time because for each suffix we have to perform a binary search on the journal tree. (see **Section 2.2.1**) This is an example where investing more computer memory can help reduce the time consumption of the algorithm.

While the decision whether to use the inverse suffix array or not is a principal one and determines the way the synchronization is performed, there is also another option that improves the performance of the synchronization algorithm. We can utilize the journal string class to store the modifications made to the suffix array, lcp-table and inverse suffix array (insertions, deletions and shifts). By using journaling within the synchronization algorithm the time consumption can be reduced. Of course when using journaling to track changes of the suffix array and the lcp-table we also have to deal with the drawbacks we mentioned in

Section 2.1.4. The searching within the index will be slower, since each access to the suffix array has to go through the corresponding journal tree first.

Whether or not it is still feasible to use this approach depends on the number of searches we want to perform on the index after synchronization and how their time consumption relates to the time saved by using journaling in the synchronization algorithm.

4. Performance

In this section we will deal with the general performance of the journal string class and compare it to different other string classes provided by SeqAn. We will compare the performance for deletion and insertion as well as for random and sequential data access. We will also compare the performance of the suffix array synchronization algorithm designed in this thesis with the suffix array creation algorithms provided by SeqAn to determine the applicability of the synchronization algorithm to different usage scenarios.

The data for time consumption of those operations was obtained on a server running a 4-Core Intel®Xeon™CPU at 3.20 GHz with 6 Gb of RAM.

If we do not specify otherwise in any of the following sections, all things "random" (e.g. positions, elements of an alphabet, etc.) can be considered to be uniformly distributed over their respective range.

4.1 Journal String Class Compared to Other String Classes

In this section we will compare the time consumption of insertions, deletions and data access of the journal string class with the SeqAn Default string class which is the alloc string.

4.1.1 The Alloc String

The alloc string, which is parameterized as `String< TValue, Alloc<> >` is the default string implementation of SeqAn. The values are stored in a continuous way in memory and it provides the basic functionality for modifying and accessing the data.

The important aspect of the alloc string's behavior for us, is the continuous storage of data in memory which leads to the effects described in **Section 1.2.2** when performing deletions or insertions on the string.

4.1.2 Deletions and Insertions

When considering the theoretical time consumption of a deletion or insertion operation on the journal string it is apparent that this will not be depending on the size of the string but rather on the size of the journal tree.

As was given in the introductory example of a string saved as an array (like the SeqAn alloc string) the time consumption is expected to be proportional to the string size. The string size directly influences the number of values that have to be copied in memory when deleting or inserting in the alloc string (see **Section 1.2.2**).

Figure 4.1 shows the time consumptions of the alloc string and the journal string for 100 random deletions and 100 random insertions in randomly generated strings of different lengths. The strings were defined over the alphabet $\Sigma := \{a, c, g, t\}$ and generated using a uniform distribution over Σ.

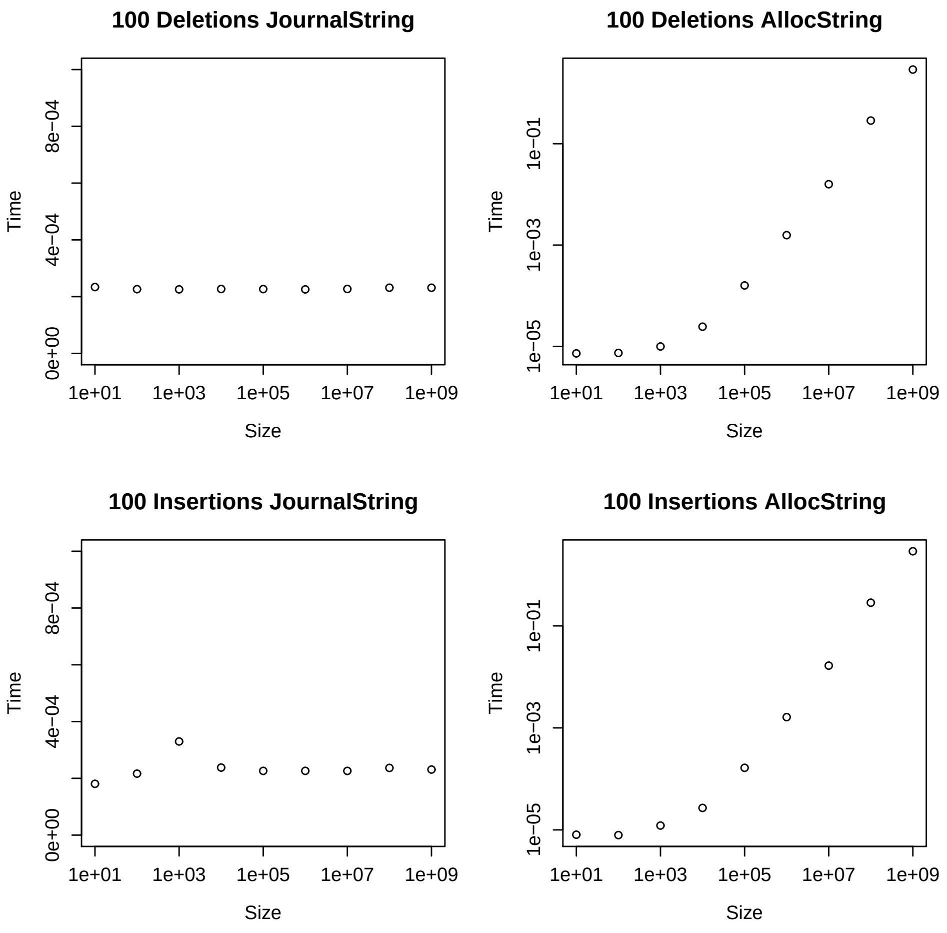


Figure 4.1: Deleting from and Inserting into the journal string and alloc string.
Consider the logarithmically scaled x-axes and the logarithmically scaled y-axes for the alloc string. Times are given in seconds. The linear development of access times for the alloc string has little influence for short sequences. This could be explained by memory allocation behavior.

For better readability the x-axes in **Figure 4.1** are logarithmically scaled. For the alloc string the y-axes are also logarithmically scaled because the values for the running time are relatively large when compared to those of the journal string.

The values were obtained by calculating the average time of 100 test runs. It can be seen that for the journal string the time consumption is more or less constant for different lengths of the string, while it scales linearly with the length of the alloc string. This can be easily explained with the way the journal string handles deletions and insertions, which are operations that primarily concern the restructuring of the journal tree. The time consumption of operations performed on the journal tree depends on the number of nodes in the tree, not the length of the string. Since the journal tree is a negligibly small data structure compared to the underlying string (we consider large sequences), the time consumption for performing operations on the journal is also small in comparison to operations on the underlying string.

4.1.3 Random Data Access

In **Figure 4.2** the time consumption of 100 random accesses on a journal string is compared to that for an alloc string. The unmodified versions of the strings have no operations performed on them, and consequently the journal tree consists of a single node. The time consumption of data access on the journal string should be comparable to that of the alloc string.

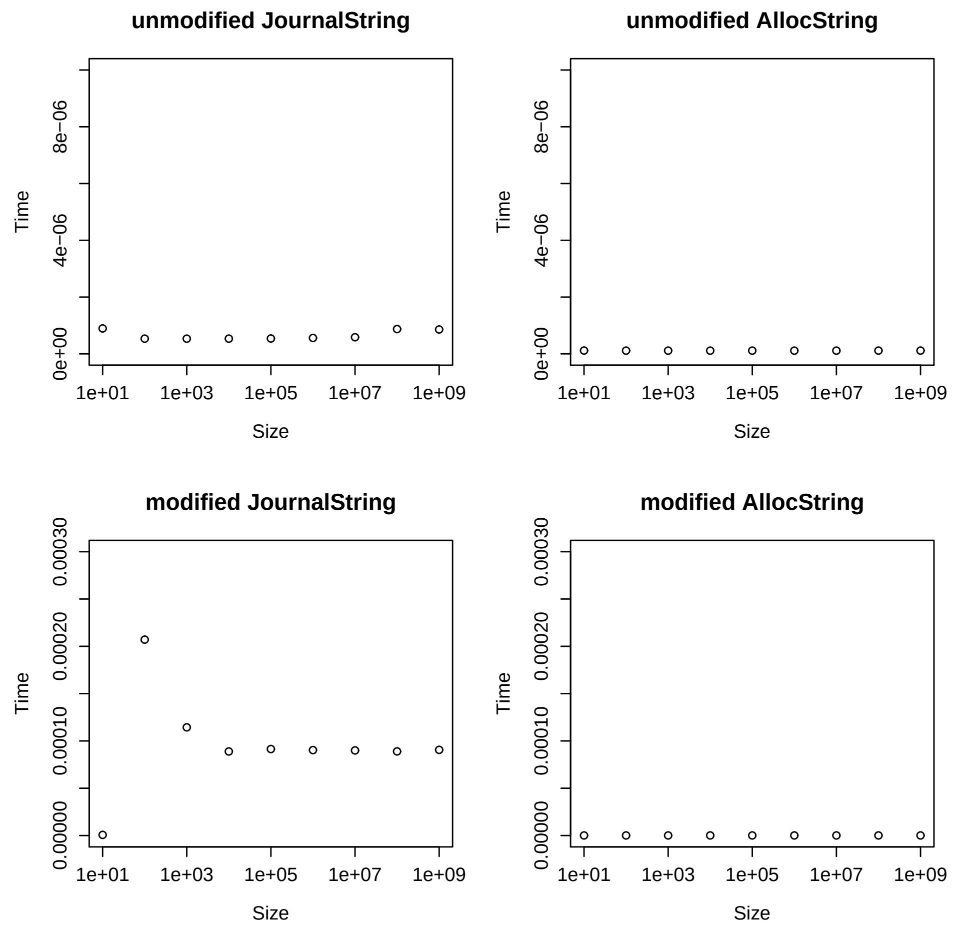


Figure 4.2: Accessing random positions within an unmodified and a modified journal and alloc string. Consider the logarithmically scaled x-axes. Times are given in seconds.

The modified strings have been subjected to 100 random insertions which should have little influence on the time consumption for accesses to the alloc string. For the journal string on the other hand, it is apparent that each insertion will generate two additional nodes in the journal tree. The node representing the block in which the insertion takes place is split and an insertion node is added to the tree. This generates three nodes from the one that was present before (see **Section 2.2.4.2**).

The journal tree now has approximately 200 nodes which leads to a best-case height of log 200 for the tree. See **Section 2.1.4** and **Figure 2.5** for a discussion and example of journal trees and their heights. The height of the journal tree is the worst-case length of a path that has to be taken through the tree to find the correct block. This happens when the corresponding node is in the lowest level of the journal tree.

Obviously the time consumption for data accesses will be higher than in the unmodified version of the journal string. Since the insertions and accesses are random some variation is to be expected which can also be seen in **Figure 4.2**.

This variation stems from the fact that depending on the positions requested for the random data access the journal tree has to be traversed up to different depths. While a node from the lowest level of the journal tree is the worst case for data access it is possible to find the correct block in the root node right at the beginning of the traversal. This is the best case for data access.

Also there seems to be some kind of initialization-effect that influences the measurements for the smallest strings. Since all the values were obtained by running each test 100 times and taking the average time consumption it can be assumed that the effect represents a systematic error. It apparently occurs in all 100 runs of the random access tests.

4.1.4 Sequential Data Access

We will now perform some benchmarking for the sequential data access of the journal string and the alloc string. We expect the alloc string to outperform the journal string because the iteration over the alloc string (which is saved as a continuous array) is a simple addition of the length of the value type to the current iterator.

Since the journal string is saved non-continuously in different blocks of data incrementing the iterator is not as simple as it is with the alloc string. See **Section 3.2.4** for details about the iterator implemented for the journal string class and its differences to iterators for continuously stored data structures. As we ahave lready discussed, some checks have to be performed when incrementing the iterator of the journal string. We have to determine whether the next value lies within the same block of data as the last, or not (see **Figure 3.2**). If we switch blocks, the memory address of the next block of data has to be computed and we have to reposition the iterator accordingly. This is done by finding the successor node of the current node in the inorder traversal of the journal tree, which holds all the information needed.

The running times for sequential data access are shown in **Figure 4.3**. The unmodified strings have no operations performed on them and the journal tree contains only one node corresponding to only one block of data (the complete underlying string).

The modified strings have had 100 random operations performed on them, yielding a journal tree of approximately 200 nodes. The setup is similar to that of the test for random data access (see **Section 4.1.3**).

Theoretically the 100 operations should not have any influence on the behavior of the alloc string. The increase or decrease in size by 100 values is insignificant when compared to the sizes of strings we considered here (lengths greater than 10000). The data shown in **Figure 4.3** supports the theoretical considerations we made.

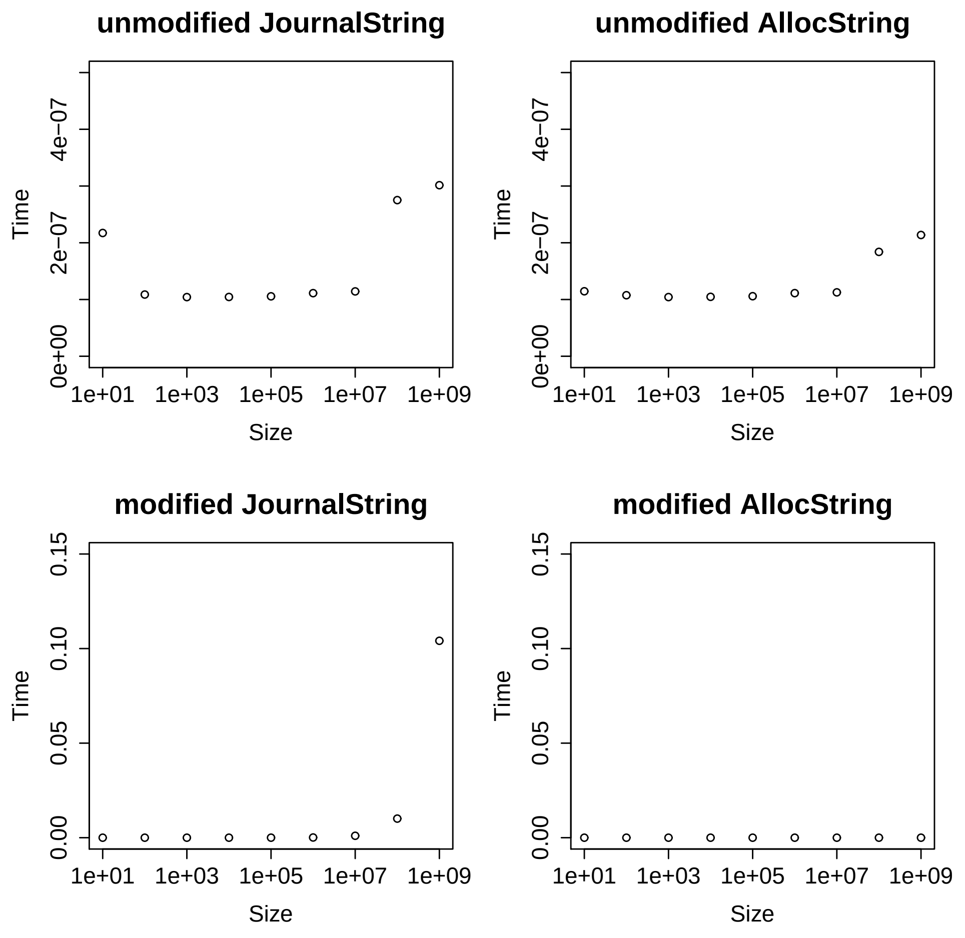


Figure 4.3: Sequential data access on the journal and alloc string.
Here the time consumption of sequential data access to modified and unmodified journal and alloc strings is compared. Times are given in seconds. We can see that the performance of sequential data access deteriorates for strings longer than 10^8 values. The journal string is influenced even more, if the journal is not empty, since the iterator has to switch between different non-contiguous blocks of memory.

For the unmodified strings there is no significant difference between alloc string and journal string. Since the journal tree contains only a single node, there are no transitions between blocks of data. The only overhead stems from the checking routine which tests whether the end of the current block has been reached. This is done before every increment or decrement of the iterator.

For the modified version the journal tree contains 200 nodes which leads to a measurably increased running time for sequential access to the journal string. It becomes more and more apparent as we consider larger strings. Another possible factor leading to the visible increase in running time for large strings could be the result of the memory consumption of those strings. For a string of length $10^8 = 100.000.000$ we need 100 Mb of memory for the sequence (of `char`) and about 400 Mb for each of the tables in the index (suffix array, lcp-table and inverse suffix array). This high memory footprint might cause the operating system to start swapping memory pages to the hard drive. Note that the lcp-table can be compressed since its values will likely be small enough. The full 4 byte required to store an `unsigned int`, which we use to store positions in the suffix array, will not be needed for the lcp-values.

4.2 Suffix Array Synchronization vs. Rebuild

In this section we will consider the index synchronization algorithm and compare the performance of the index synchronization with that of the rebuilding of the index. For the rebuild we use the SeqAn default suffix array creation algorithm which is Skew7 (Weese, 2006). Skew7 is the extension of the Skew algorithm by Kärkkäinen & Sanders (2003).

The testing was performed on randomly generated texts over the alphabet $\Sigma := \{a, c, g, t\}$ on which random operations were performed. For the generation of the text we used a uniform distribution over the alphabet.

The time consumption of rebuilding was assumed to be constant since the test cases involved strings of lengths greater than 10000 with less than 100 operations performed on them. This 100 values that are removed or added will not influence the time consumption of the index rebuilding significantly.

It was calculated for an increasing number of operations what fraction of the rebuild time was used for the synchronization and the process was stopped when the ratio

$$\frac{\text{synchronization time}}{\text{rebuild time}}$$

became larger than 1.0 i.e. the synchronization started to cost more time than the rebuilding. For this measurement only the number of operations was recorded. The influence of the length of the operations will be explored later on (see **Section 4.2.1**).

The synchronization algorithm used and maintained the inverse suffix array as an auxiliary table which was not provided by the normal suffix array construction algorithm. Therefore a certain amount of additional time was consumed by the synchronization algorithm for this maintenance. During the testing we initially used two different implementations of the synchronization algorithm. The fundamental algorithmic approach of both was the same and they both used the inverse suffix array to find the influenced suffixes but one used journaling on the index and the other applied the changes directly.

As we already talked about in **Section 3.3** the application of journaling within the synchronization algorithm improves the running time. Results of preliminary benchmarks comparing the two implementations of the synchronization algorithm (with and without journaling) showed that the version using journaling performs better. We therefore used the faster version (the one utilizing journaling) to perform the benchmarks within this section. Information on the different implementations of the synchronization algorithm can be found in **Section 3.3** and the general concept of the synchronization algorithm was described in **Section 2.2.3**. **Figure 4.4** shows the comparison of our synchronization algorithm with the Skew7-based rebuilding of the suffix array and lcp-table.

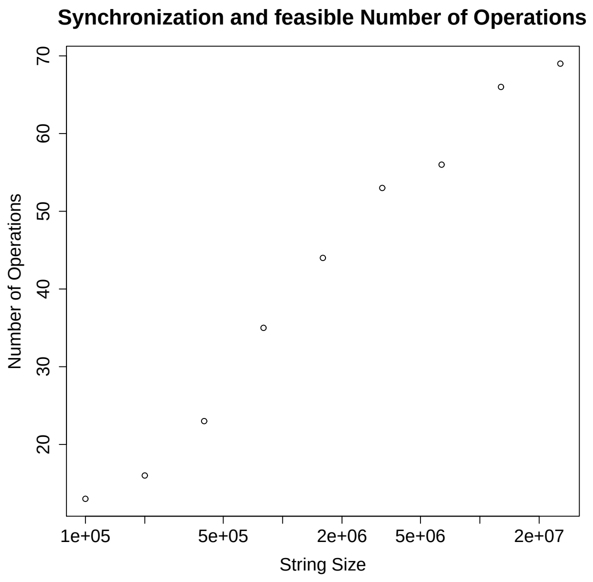


Figure 4.4: Comparison of conventional suffix array building and synchronization.
The number of operations for different string lengths where synchronization is faster than rebuilding is shown. The x-axis is scaled logarithmically.

We also performed tests with random sequences of different lengths where we computed the ratios between synchronization time and rebuild time for up to 50 insertions. In **Figure 4.5** we illustrate the results.

We can see that the runtime ratio gets better when the string length increases. For longer string lengths the advantages of our synchronization algorithm start to have more influence. Note that we always perform up to 50 insertions so the actual percentage of the string that has been modified decreases for longer strings.

4.2.1 Operations of Different Lengths

We will now consider the influence of the lengths of journaled operations on the synchronization time. The question that we want to answer is how the number of suffixes that have to be added or deleted relates to the number of suffixes that are influenced by the operations.

Because of the computational effort needed to calculate influenced suffixes we expect a single insertion or deletion of length m to be easier and faster so synchronize than m insertions or deletions of length 1. To examine the effects different lengths of operations have on the synchronization time, we performed a series of calculations. We tested a string of length 10000 into which we inserted 100 respectively 1000 values in packages of 1, 10 and 100 values

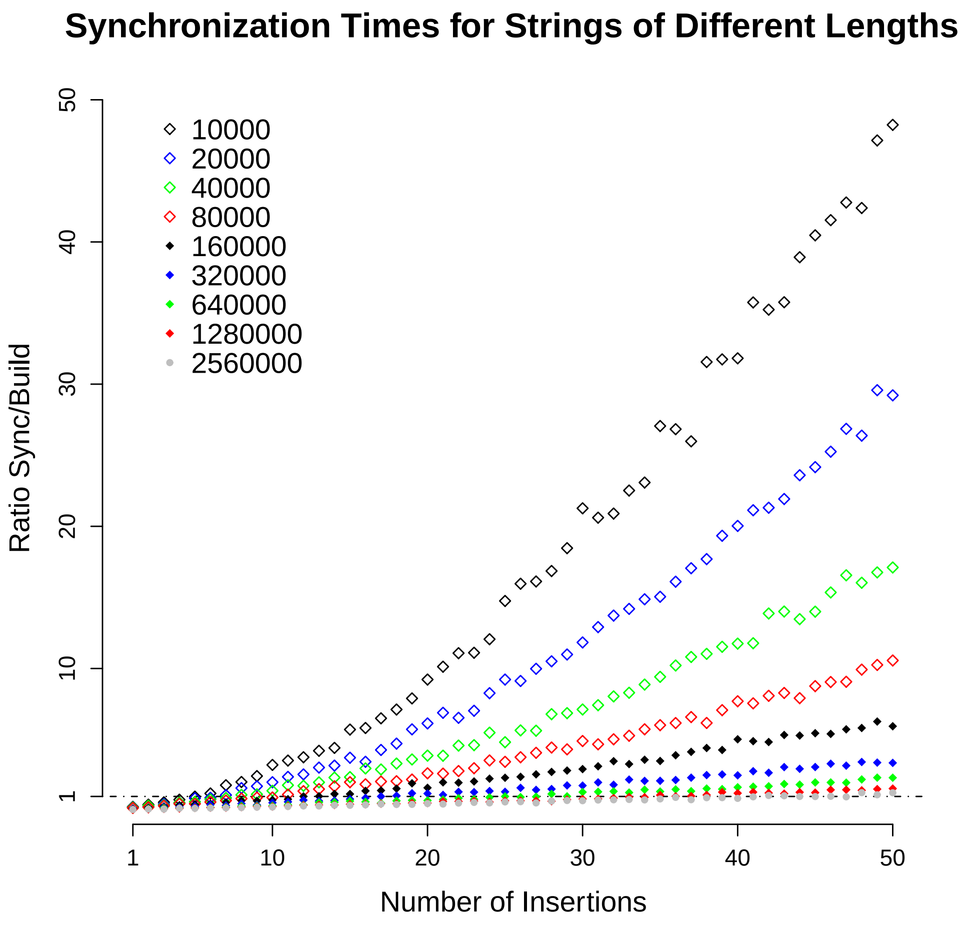


Figure 4.5: Comparison of synchronization and rebuild for strings of different lengths. The horizontal line at $y = 1.0$ represents the upper boundary for the time ratio where synchronization is still feasible.

(plus package size 1000 for the 1000 inserted values). **Figure 4.6** shows the resulting time consumption of the synchronization for the different package sizes.

Here we can see the influence the package size of the operation has on the synchronization algorithm. Synchronizing the inserted suffixes will be equally time consuming for arbitrary package sizes, since in any case 100 respectively 1000 suffixes have to be added. The number of influenced suffixes, however, differs greatly between the package sizes. Smaller package sizes lead to greater numbers of influenced suffixes because of the larger number of operations that are performed which in turn leads to a longer running time (see **Figure 4.6**).

The reason for the longer running time is that the number of influenced suffixes per operation is dependent on the average lcp-value and independent of the operation length. When splitting an insertion of length m into insertions of length 1 we dramatically increase the average number of influenced suffixes from $AVG(LCP)$ to $m \cdot AVG(LCP)$. This is because we increased the number of operations from 1 to m and each of those influences $AVG(LCP)$ many suffixes.

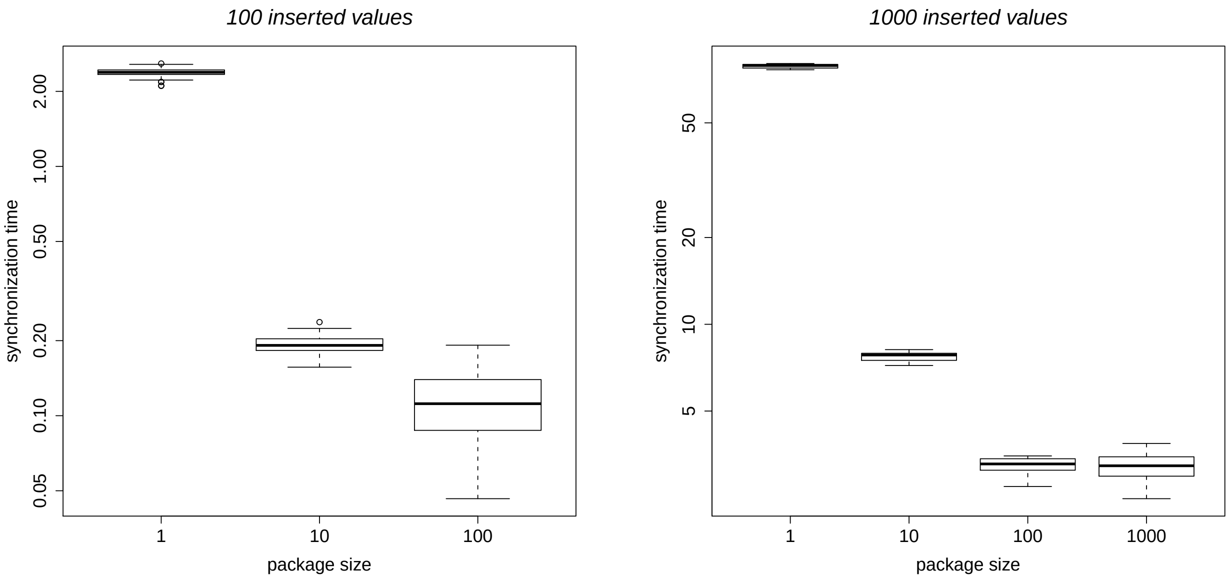


Figure 4.6: Inserting an equal number of values in different package sizes.
We inserted 100 respectively 1000 values into a string of length 10000 using the package sizes 1, 10 and 100 (and 1000). The distribution of synchronization times is shown via boxplot. Note the logarithmically scaled y-axis.

4.2.2 Alphabets of Different Size

For our randomly generated sequences the average lcp-value depends on the alphabet size. Larger alphabets result in fewer repetitions when considering a text of a given length and therefore reduce the average lcp-value. In **Section 2.2.3.2** we discussed the relation between the average lcp-value and the influenced suffixes. We performed test to determine how many suffixes are influenced per operation for different alphabets. The tests were done with the alphabets $\Sigma := \{a, c, g, t\}]$ and $\Sigma' := \{a, b, c, \ldots, y, z\}$ with $|\Sigma| = 4$ and $|\Sigma'| = 26$.

We can see the decrease in number of influenced suffixes per operation when we increase the alphabet size in the diagrams given in **Figure 4.7** and we can also see that for the same alphabet the number of influenced suffixes grows when the text gets larger. This is due to the fact that larger texts over the same alphabet have higher lcp-values for their suffixes.

For random texts over Σ and Σ' we calculated the average lcp-values of their suffixes using different lengths for the texts. The results are shown in **Figure 4.8** and we can see, that the number of influenced suffixes is always slightly higher than the average lcp-value. This is not unexpected, particularly when we consider the length of the sensitive interval of a suffix $S_k = SA[i]$ which is defined as max$(LCP[i], LCP[i-1])$.

The data was obtained by measuring the lcp-values and number of influenced suffixes for random texts of lengths 10000, 20000, …, 640000 over the alphabets Σ and Σ'. We inserted 50 values in packets of size 1, generating 50 operations in each string.

All of the properties concerning the running time of the synchronization algorithm show that generalized assertions about its applicability are not easily made. We have to consider

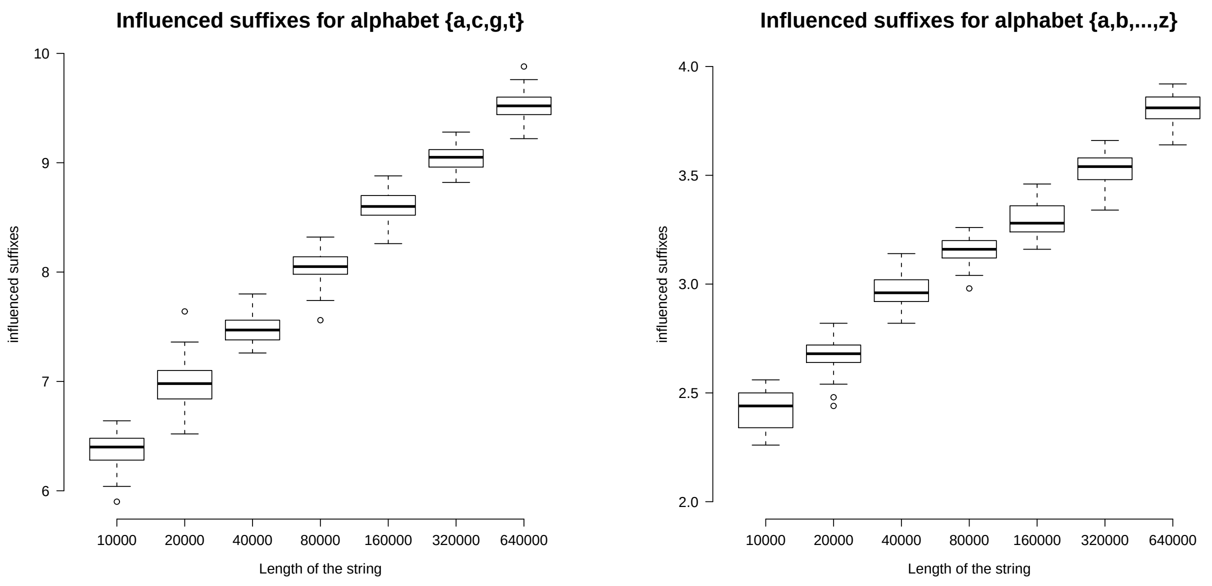


Figure 4.7: Average number of influenced suffixes per operation for alphabets Σ and Σ' of size 4 (nucleotides) and size 26 (English language), respectively. The measurements were taken for strings of increasing lengths. Note that the number of influenced suffixes per operation increases for larger strings, because the average lcp-value also increases.

the specific situation in which our algorithm will be used and examine the average lcp-value we expect, which depends on the alphabet size and the general repetitiveness of the text.

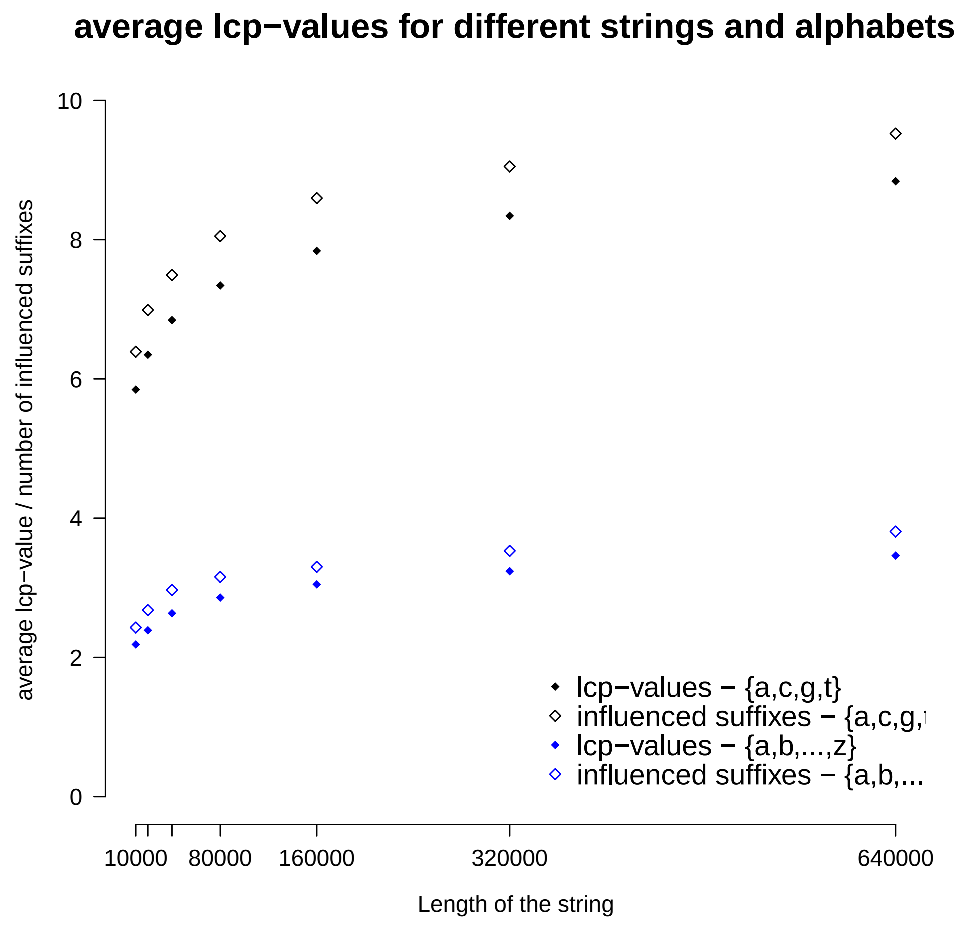


Figure 4.8: Influenced suffixes and mean lcp-values
Note how the number of influenced suffixes is always slightly higher than the mean lcp-value. This is due to the average length of the suffixes' sensitive intervals which corresponds to the average lcp-value $+1$. As a consequence of the definition of the sensitive interval (**Definition 14**) we expect the number of influenced suffixes per operation to be about $AVG(LCP) + 1$.

5. Conclusions

5.1 Advantages and Disadvantages of the Journal String Class

The journal string class has some advantages and some disadvantages that were discussed in theory in **Section 2.1.4** and the corresponding benchmarks were performed in **Section 4**. The main advantage of the journal string class is the constantly small time consumption of insertions and deletions and the main disadvantage is the increased time consumption for data access.

The decision whether to use the journal string class in a given scenario must be based on the relation of the number of operations to the number of data accesses that have to be performed. We always have to consider if the advantages make up for the disadvantages. It is highly dependent on the current usage scenario whether or not the time saved by journaling operations justifies the additional cost for accessing the data through the journal string.

5.2 Applicability for Index Synchronization

The results obtained in the context of this thesis show that in general the approach of synchronizing the index to a modified text is favorable to rebuilding the index from scratch.

With the current implementation the number of modifications for which the synchronization is more efficient than the rebuilding of the index is still quite small. This is a problem that to some extend will be remedied by implementing the optimizations discussed in **Section 5.3.1** and performing general optimizations on the code.

As with all things that improve on one aspect of a problem by accepting a trade off at another aspect the applicability of the synchronization algorithm is largely dependent on the usage scenario. In the case where we have large texts that are subject to relatively few modifications the synchronization will outperform a rebuilding of the index and is the favorable alternative. The exact boundaries for the number of modifications with respect to the length of the text are subject to changes when the implementation of the journal string and the synchronization algorithm are further optimized.

5.3 Possible Modifications and Improvements

In this section we deal with the possibilities to improve and modify the current implementation of the journal tree and the synchronization algorithm.

5.3.1 Optimizing the Journal Tree

As was already hinted at in previous sections there is some potential for optimization of the journal tree, since the current implementation uses and unbalanced journal tree. We could apply a balancing strategy similar to that of AVL-Trees to guarantee a worst case height of $\log k$ for a tree with k nodes. AVL-Trees a apply a balancing strategy after each operation performed on them. They keep the binary search tree property and keep the tree height as low as possible ($\log k$). See Knuth (1998) for a more detailed description.

Another option for improvement is reducing the number of nodes in the tree by merging adjacent nodes of the same type. This is trivial for nodes representing deletions. Consider two deletions at positions k_1 and k_2 of lengths m_1 and m_2 (we assume $k_1 < k_2$ without loss of generality). Since the nodes have to be adjacent in the tree (when traversing inorder) it is immediately clear that $k_2 = k_1 + m_1$ must hold. These nodes can easily be merged into a new deletion that starts at k_1 and has length $m_1 + m_2$ which is illustarted in **Figure 5.1**.

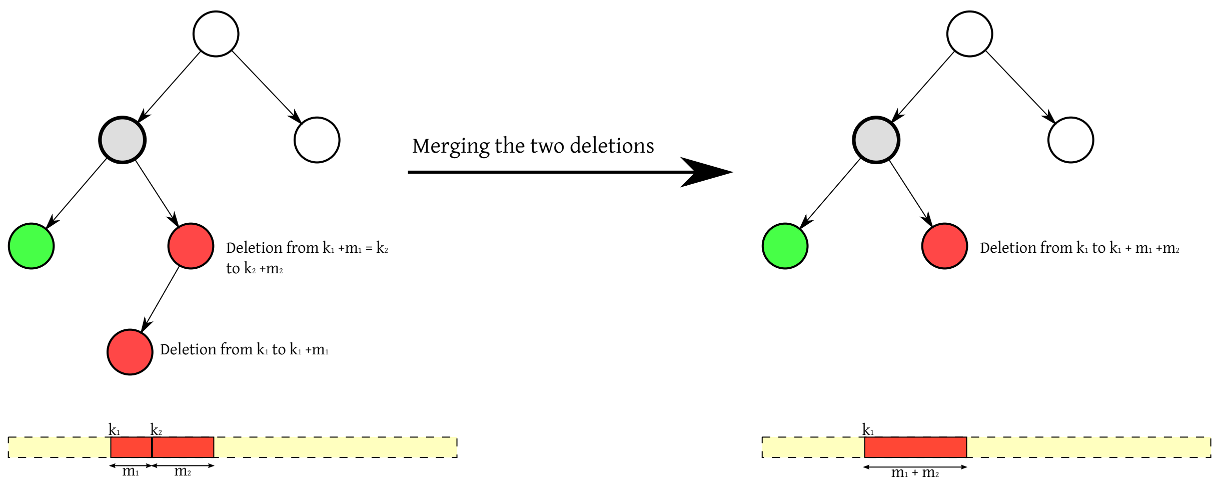

Merging the two deletions
Deletion from $k_1 + m_1 = k_2$ to $k_2 + m_2$
Deletion from k_1 to $k_1 + m_1$
Deletion from k_1 to $k_1 + m_1 + m_2$
k_1
k_2
m_1
m_2
k_1
$m_1 + m_2$

Figure 5.1: Merging of two adjacent deletions in the journal tree.
Note how both the node count and the tree height is reduced.

For adjacent insertions the process is more complicated. To merge two insertions some restructuring of the insertion string might be needed because adjacent insertion nodes in the journal tree do not necessarily correspond to adjacent blocks of data in the insertion string. Because of the way insertions are handled in our implementation, the newly created merged insertion node must point to a continuous block of data in the insertion string. We will likely have to reappend the two insertion blocks at the end of the insertion string and then we can create a merged insertion node representing this new merged block.

Another aspect of optimization of the journal tree is that of removing nodes that represent operations of length 0, which at the moment can occur when a node is split due to a deletion that is flushed with one side of the node's block of data. This effect is illustrated in **Figure 5.2**.

Also the addition of another node type representing a replacement of a block of data could reduce the number of nodes in the tree. At the moment a replacement is modeled as a consecutive deletion and insertion at the same position in the string. This creates an unnecessary

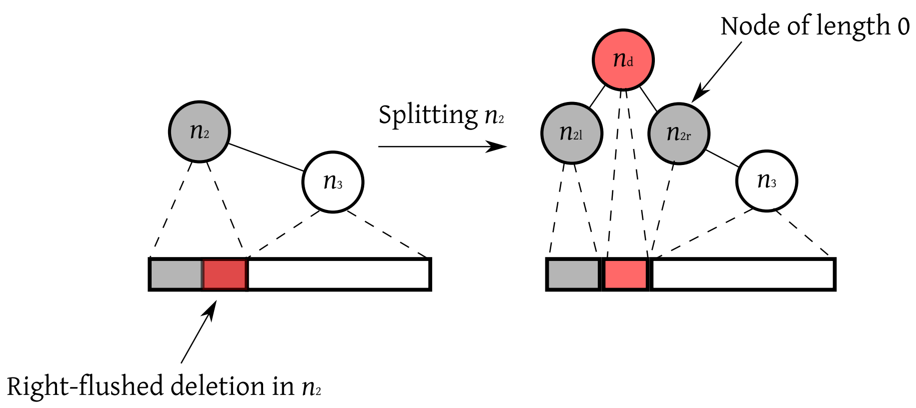


Figure 5.2: Splitting a node when journaling a deletion can yield nodes of length zero.

number of nodes (both a deletion and an insertion node).

As a general optimization strategy the usage of profiling software to find bottlenecks is a feasible approach. By using profiling software like the valgrind (Nethercote & Seward, 2007) profiling tool callgrind we can determine the functions called most often during the synchronization process. As we expected from our theoretical discussions in the previous sections the critical functionality is that of data access to the journal string. The tree traversal is crucial to the time consumption on almost any algorithm applied to a journal string. By making use of the valgrind toolbox in our development of the current implementation of the journal tree and synchronization algorithm we tried to optimize the performance critical functions that were identified. There are, however, certainly more bottlenecks to be found and improved by using software profiling.

In this section we have pointed out some possible optimizations of the journal tree that help to reduce the number of nodes and the tree height and will improve the time consumption for the data access further.

5.3.2 Compression, Lazy Data Structures and Localized Synchronization

In **Section 1.4** we already mentioned the concept of lazy data structures that save computation time for their creation and memory for storing them.

Applying those concepts to our journal string could improve running time of data access. We could also explore the possibilities of compressing the tables and making them lazy. Of course such an approach has the disadvantage of having additional computational cost for retrieving data from the compressed tables.

Since the journal string is fully integrated with SeqAn it should be relatively easy to use it with the compressed data structures provided by SeqAn. The PizzaChilli string is an example of a lazy compressed string that decompresses only when data is requested and only the requested data.

An interesting approach could also be a localized synchronization strategy that checks the queries made to the index. If an influenced suffix would be used in the search on the index it will automatically be synchronized with the operation that influenced the suffix. In this way the whole synchronization could be performed 'lazy' and only if we query for indices that have been influenced.

5.3.3 Improving the Synchronization Algorithm

In the current implementation the synchronization algorithm uses `std::sort` in combination with a custom designed compare functor to sort the array of indices that has to be merged with the remaining suffix array. `std::sort` is a very effective sorting algorithm but for sorting suffixes better approaches exists. By taking into account the lcp-values of the suffixes to minimize the number of character comparisons we can improve the sorting of the suffixes.

Another interesting point is the used merge-sort-like algorithm. At the moment we use two iterators over the suffix array and the array of suffixes that have to be inserted (the insertion array). We compare both suffixes and increment the iterator in the suffix array until it points to a lexicographically larger suffix than the iterator in the insertion array and then insert the suffix from the insertion array into the suffix array and increment the iterator of the insertion array.

This classical merge-sort approach has a running time of $\mathcal{O}(m+n)$ when counting suffix comparisons, where n is the suffix array size and m is the size of the insertion array. When counting character comparisons this gets worse since we potentially compare two suffixes that have a length of almos $m+n$ (the first and second for example).

Since we ideally will have the situation where $m \ll n$ we could inspect a different approach that would not be feasible for two equally sized arrays but may lead to runtime improvements in our situation. The idea is to use binary search on the suffix array to find the insertion positions for the first and last suffix in the insertion array. We now can determine the insertion positions for all other suffixes in the insertion array by performing binary search. Since the suffixes in both arrays are ordered lexicographically we know that the positions we look for will be between the insertion positions of the first and last suffix that we determined previously. In this way we can reduce the size of the binary search interval and thereby improve the running time of the insertions to be better than the trivial upper boundary $\mathcal{O}(m \cdot \log n)$ (m suffixes are inserted via binary search with $\log n$ comparisons each when counting suffix comparisons). When we assume the relation between m and n to be $m \ll n$ we will have a running time that is better than $\mathcal{O}(m \cdot \log n)$ which in turn could be better than $\mathcal{O}(m+n)$. This is an improvement that is obviously sensitive to the size of the insertion array which is determined by the number of inserted and influenced suffixes (see **Section 2.2.3**).

When we accumulate too many operations before we synchronize and therefore have $m \sim n$ the running time could become worse. We see this as a possibility to investigate the properties of the synchronization algorithm further but have not implemented the approach in this thesis.

5.4 Possible Usage Scenarios

Since the two main results of this thesis are the journal string class integrated with SeqAn and the index synchronization algorithm we will now discuss possible applications of this two features.

The journal string class in itself is useful for a lot of applications both bioinformatical and otherwise. The advantages and disadvantages which we discussed in this thesis are not restricted to bioinformatical applications and there are a wide range of usage scenarios for this kind of string class. Basically each usage scenario where large strings need to be modified and some additional time consumption for data access is acceptable can be a possible application of the journal string class.

Of course the decision to apply journaling in any application must be made specifically for the data that the application is working with. The problem structure is essential in determining the applicability of the journal string class to the given problem.

Depending on the amount of time one is wiling to sacrifice for data access some interesting concepts for saving memory on applications that work with multiple genomes arise. For example, when working with genetic sequences from several individuals that are closely related to each other, it could be interesting to encode genomes by their differences to one genomic sequence that serves as an underlying string (similar to the aproacch taken by (Christley *et al.* , 2009)). We could use available information about the differences between this base genome and the genomes of the other individuals to build journals containing all the operations necessary to convert the base genome into the genome of the individual the journal will correspond to.

In this way the information that normally would require the memory needed for one genome times the number of individuals we want to store genomes for can be saved in a comparatively small space. We could encode the genomes of many different persons in the memory used for one genome plus the size of a journal (which is considerably smaller than a genome) times the number of individuals.

Some work would be required concerning the average number of operations needed to transform one genome into another closely related one. This would serve to determine the feasibility of this application of journal strings. An illustration of the general idea behind this way of storing multiple sequences encoded as journals of their differences to a common reference sequence is shown in **Figure 5.3**.

The same concept that can be used to minimize the memory consumption of storing multiple genomes via the described method can be extended to indices as well. We would use the version of the synchronization algorithm where the changes to the different tables (suffix array, lcp-table and inverse suffix array) are stored in journals and not applied directly.

Because of the modular design applied in the journal string class it is possible to have multiple journals over the same text and thereby one can access different revisions of the text depending on which journal is used. We could then have different journals representing

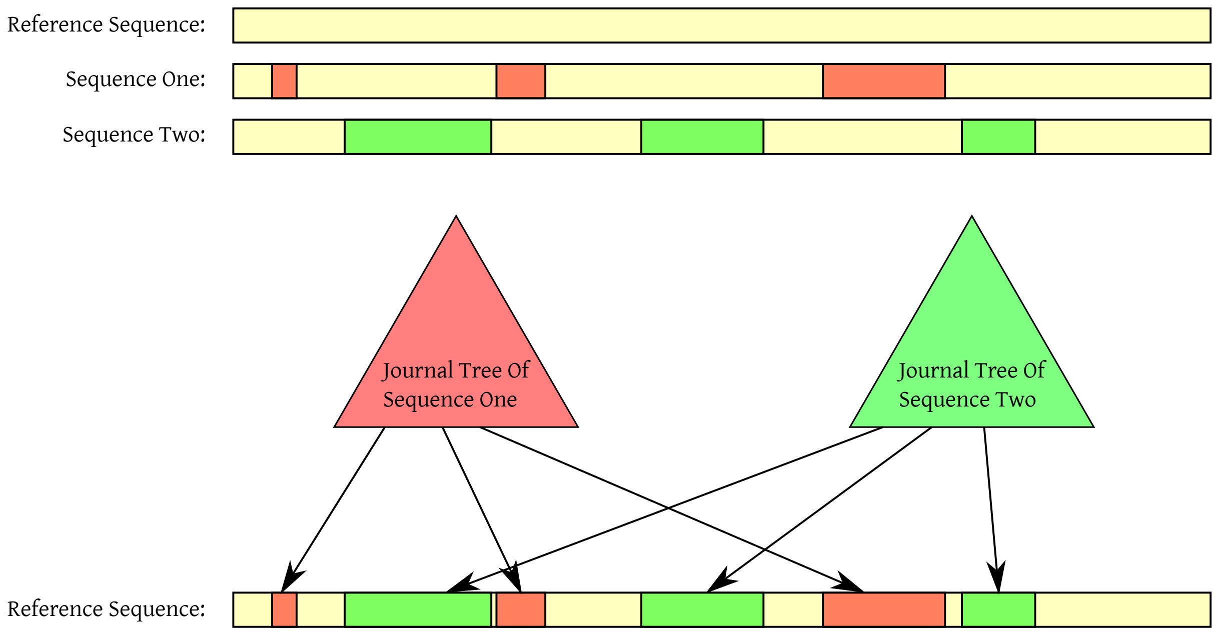


Figure 5.3: Encoding sequences by their differences to a common reference.
Here the principle design of a storage solution for multiple sequences that are defined by their differences to a common reference sequence and stored in the form of journals is shown. In this picture the red and green markings represent differences between the reference sequence and sequences one and two, respectively.

different revisions of the index tables allowing us to store the indices of multiple genomes in relatively small space. This approach demonstrates the possibilities of saving storage space that the journal string class provides and should be further explored.

As we already discussed in **Section 1.4** some ideas for storing genomes encoded as a set of modifications to a reference genome were discussed in the paper **Human genomes as email attachments** by Christley *et al.* (2009). For this approach we could explore the possibilities of serializing a journal tree into a file and determine the reduction in space consumption for whole genomes and their differences. We could also try to interface with the work done in Christley *et al.* (2009) and generate journal trees from the data that the authors produced.

5.5 Summary

In this thesis we have described the functionality required for journaling changes to a string. We also discussed the advantages and disadvantages of such a journal string. We have developed the journal string class as part of the SeqAn C++ library and used benchmarks to verify the theoretical considerations about its performance (**Section 4**).

Furthermore, we used the journal string class to implement the suffix array synchronization algorithm presented in this thesis. We provided a first usable implementation of the journal string class and tested it in a realistic usage scenario (the synchronization algorithm).

As evidenced by the benchmarks we performed, there is a cut-off threshold for the number or stored operation where the journal string class becomes so slow that the increased access times are no longer amortized by the time saved when performing the modifications.

We discussed possibilities to improve the journal string class and have given an outline of ideas to improve the access performance, thereby presenting opportunities to change this threshold. Such optimizations can better compensate for the slower data access that is inherent to the journal string when we compare it to other conventional string implementations.

This drawback is offset by the possibility to efficiently modify large sequences provided by the journal string class. Particularly its usage solving the index synchronization problem has a lot of practical benefits.

We see great potential for improving the current application of the journaling data structure by utilizing the discussed optimization techniques as well as for the above mentioned new application of journals in encoding genomes by their differences to a common reference genome. We consider it an important step towards achieving manageability of the large datasets that we expect to be made available for analysis in the coming years.

5.6 Acknowledgements

First and foremost I would like to thank everybody who helped and supported me in writing this thesis. I would expecially like to thank David Weese for providing ideas and insights into the theoretical aspects of my work and for helping with the usage of SeqAn and the debugging of my code. Of course my fellow students, Konrad Rudolph and Vipul Patel, with whom I shared many interesting discussions and long hours working in office 009 shall not be forgotten. Thank you for the creative input, helpful comments and tips.

Special thanks to Konrad for being such a great source on everything C++ and LaTeX.

I would also like to thank my supervisors Prof. Dr. Knut Reinert and Dr. Gunnar W. Klau for their effords in teaching and their support.

Bibliography

Burrows, M., & Wheeler, D. J. 1994. *A block-sorting lossless data compression algorithm.* Tech. rept. 124.

Christley, Scott, Lu, Yiming, Li, Chen, & Xie, Xiaohui. 2009. Human genomes as email attachments. *Bioinformatics*, **25**(2), 274–275.

Döring, Andreas, Weese, David, Rausch, Tobias, & Reinert, Knut. 2008. SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**(1), 11.

Giegerich, Robert, Kurtz, Stefan, & Stoye, Jens. 1999. Efficient Implementation of Lazy Suffix Trees. *Algorithm Engineering*, 30–42.

Illumina Inc. 2009. *Every Genome.* http://www.everygenome.com/.

Kärkkäinen, J., & Sanders, P. 2003. *Simple linear work suffix array construction.* Springer.

Kasai, Toru, Lee, Gunho, Arimura, Hiroki, Arikawa, Setsuo, & Park, Kunsoo. 2001. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. *Pages 181–192 of: CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching.* London, UK: Springer-Verlag.

Knuth, Donald E. 1998. *Art of Computer Programming, Volume 3: Sorting and Searching.* Second edn. Addison-Wesley Professional.

Manber, Udi, & Myers, Gene. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22**(5), 935–948.

Nethercote, Nicholas, & Seward, Julian. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, **42**(6), 89–100.

Salson, M., Lecroq, T., Léonard, M., & Mouchard, L. 2009a. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, March.

Salson, M., Lecroq, T., Léonard, M., & Mouchard, L. 2009b. A four-stage algorithm for updating a Burrows-Wheeler transform. *Theor. Comput. Sci.*, **410**(43), 4350–4359.

Venter, J. Craig, . . ., Reinert, Knut, *et al.* . 2001. The Sequence of the Human Genome. *Science*, **291**(5507), 1304–1351.

Weese, David. 2006. *Entwurf und Implementierung eines generischen Substring-Index.*