Mountable position heaps

Exposé zur Studienarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

eingereicht von:	Hoang Tran duy
Betreuer:	Prof. Dr. Ulf Leser
	Dr. Sebastian Wandelt

1 Motivation and background

Day-to-day a huge amount of textual information is produced and consumed. Textual information is often available in databases and requires efficient indexing methods to support fast retrieval[HK04]. A typical search scenario is called *pattern matching*: Find all occurrences of the pattern P in a string T [KJP77].

Recently, there is an increasing number of applications dealing with highly-similar strings, such as multi genomes analysis of similar species [HK13] or document revision control [Rit13]. For searching in a collection of very large strings, there exist two approaches:

1. *Create a standard index for each string separately*: Standard index structures like suffix trees or suffix arrays are not well suited for collections of such highly-similar strings. The indexes consume a lot of space, since they do not exploit inter-string similarities.
2. *Create a combined index for the whole collection of strings*: Although new index structures for collections of highly-similar strings were proposed very recently [KN13,HPB13,WSBL13], these index structures either are still rather large [KN13,WSBL13] or searching them is slow [HPB13].

In this thesis, we present a new way for managing a single index over a string from a large collections of high similar strings. We envision a scenario, where a user needs to deal with only one (or few) of the highly-similar strings at a time; while frequently having to change the strings. An example for such scenario is the process of short read alignment with populations of genomes, where only one (or a few) genomes are targeted. Our idea is, for a given collection of highly-similar strings, rather than to create and store an index for each string, we use the index of only one string R of the collection as a base-index. For searching in another single string S of the collection, we transform the existing index of R into an index for S in main-memory and then apply the searching. This index transformation is what we envision as *mounting*[1]. Comparing to the two above mentioned methods, this approach has the following benefits: a smaller memory footprint, because only the index of one string is used at a time and a faster index construction by mounting from an existing index.

For example, consider two highly similar strings R and S and let R be the base-string. We denote an index for R with I_R and and an index for S with I_S. In order to transform I_R into I_S, there are two hurdles to overcome:

1. Find an appropriate index structure, which allows the indexed string to be edited, and
2. Identifying the similarity of strings and how to utilize them for the index transformation.

We envision a *mounting*-step of the index I_R as an in-memory-only and non-persistent edit-operation on the string R, such as *deleting* or *inserting* a block of characters. The benefit is, that our approach can be applied on any index structure, which supports the two operations. Standard index structures like suffix trees however are not suited for our approach. Not only because current implementations do not support edit-operations on a string. Even if an implementation would support these operations, it would be very inefficient to update the index because changing a single character can cause severe impact on the index (Figure 1).

Other existing index structures, which support edit-operations, like generalized suffix trees and dynamic extended suffix arrays do not support arbitrary edits [FGM98] or the theoretical worst-case bound is very high [SLLM10]. However, in 2011, Ehrenfeucht et al. introduce a new dynamic index structure called *position heaps*, which has a better theoretical worst-case bound [EMOW11] and is therefore suitable for our approach of frequently mounting an index.

Now we can assume, that I_R is available and mountable. The next question is, how to exploit the similarities of R and S. As our approach aims to be faster than the construction of I_S from the scratch, it is crucial to minimize the costs of transformation, i.e. the amount of *mounting*-steps. Since R and S are similar, the index I_R must contains parts, which represent the identical sub-strings of R and S and do not need to be transformed. In order to identify such sub-strings and the therefore the reusable index-parts, we use referential compression [WL13]. The referential compression of a string S against R is a set of referential match entries, which describe similarities and differences of S against R. Our hypothesis is that if S can be encoded with few referential match entries against R, then the transformation of I_R into I_S should only take a small amount of time, compared to creating I_S from the scratch.

[1] Mounting refers to the process of rewriting an index for a reference string from the collection into an index for a chosen string.

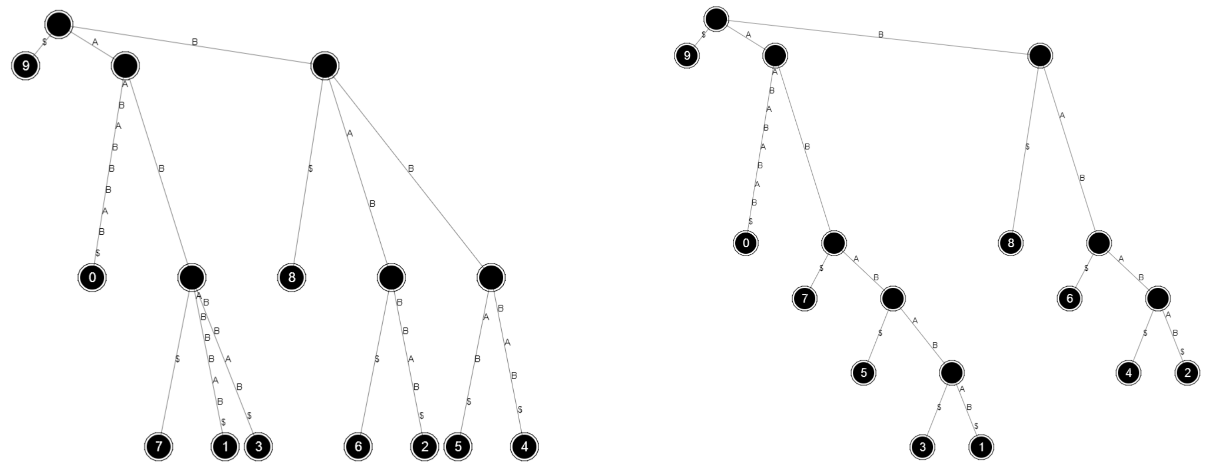

Fig. 1. Suffix tree of R = AABABBBAB$ (left) and S = AABABABAB$ (right).

2 Goal

For a given position heap ph_r for string R and a referential compression for string S against reference R, our approach is to convert the ph_r into a position heap ph_s for S. We expect, that this transformation is faster than creating the position heap or any other string index structure from the scratch. And so the goal of the thesis is to:

1. design an algorithm to *mount* a string index structure, by using the dynamic feature of position heaps with a given referential compression of the new string against the previously indexed string,
2. implement the algorithm in C++, and
3. evaluate whether our expectation (speed-up of index construction) is correct with respect to real world data.

3 Technical preliminaries

In the following we repeat two essential components for mounting index structure: position heaps and referential compression.

3.1 Position Heaps

Position heaps were introduced in [EMOW11] and are based on the data structure called *sequence hash trees* [CJE70]. Intuitively, position heaps represent the shortest unique prefixes of all suffixes of a string. In comparison to other index structures, the better theoretical worst-case bound for edit-operations of the position heaps is beneficial to the performance of the *mounting* process. A simplified definition of the (augmented) position heaps was given by Kucherov [Kuc13].

Assume, that a ordered set of strings $W = \{w_1 \dots w_n\}$ is given and no w_i is a prefix of w_j for any $j < i$. The sequence hash tree for W, denoted $SHT(W)$, is a trie defined by following iterative construction:

1. the tree $SHT_i(W)$ is the sequence hash tree for $\{w_1 \dots w_i\}$,
2. the tree $SHT_0(W)$ consists of only the single node *root* with label 0,
3. let $v = v'a$ the shortest prefix of w_{i+1}, which is not represented in $SHT_i(W)$, i.e v' is the longest prefix of w_{i+1} which is represented in $SHT_i(W)$. $SHT_{i+1}(W)$ is obtained from $SHT_i(W)$ by adding a new node as child of v' with the edge label a and the node label $i + 1$.

Let $T[1 \dots n]$ a string with the termination character $T[n] = \$$, a *position heap* of T is the sequence hash tree of the set of suffixes of T, where the suffixes are ordered in the descending order, i. e. from left to right. In order to support searching in linear time, there are three additional data structures associated with the *position heaps*:

- an array of pointers, such that for a given i, a node with label i can be found in $O(1)$ of time;
- a data structure, such that for given i and j, it can decide in $O(1)$, whether the node with label i is an ancestor of the node with label j.
- the node labeled i stores a pointer (called its maximal-reach pointer) to the deepest node whose path label is a prefix of $T[i \dots n]$.

In addition, there are two important constraints for the position heaps:

- *Heap property:* Each node carries a label from an ordered set, and for every internal node X, the labels of the children of X are greater than the label of X. As the suffixes are ordered in descending order, their correspondent nodes also have the labels equal to their positions within the string.
- *Uniqueness:* For each node U and a letter b $\in \Sigma$, there is at most one edge with label b from U to a child of U.

For example, consider $R = AABABBBAB\$$ and $S = AABABABAB\$$, the position heaps for both strings are shown in Figure 2.

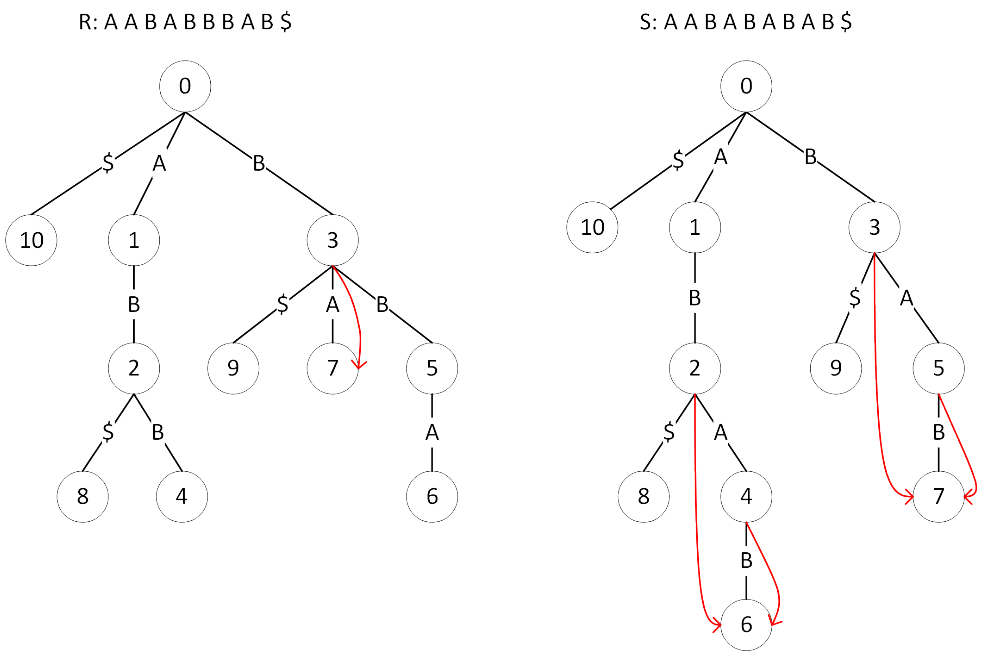

Fig. 2. The position heaps of R and S with maximal-reach pointers as red arrows.

Comparing Figure 1 with Figure 2, it can be seen that suffix trees encode rather a global view on the indexed strings, i.e. changing a single character can have a tremendous effect on the structure of the tree, or at least on the edge labels, since potentially many suffixes change. Position heaps, on the other hand, rather encode a local view on the indexed strings, where a local edit often only has a local effect on the position heaps. For long strings with only few edits, this local effect of position heaps is anticipated to be much stronger, i.e. the change of structure of the position heaps is very small.

In case of the index transformation, the use of suffix position as node label however is not sufficient, due the cost to update them. As solution for this problem, Ehrenfeucht et. al. introduced another data structure called *dynamic array abstract data type* (DAADT), which is based on the *splay trees*. This data structure enables the representation of the text and other lists in a dynamic way [EMOW11]. The data structures for the dynamic and static versions of the position heaps are identical, except that:

- the text is represented as DAADT, and
- each node of the position heap has a pointer to a node of the DAADT tree as node label and
- each node of the DAADT has a pointer to the corresponding node of the position heap(the first additional data structure of the augmented position heap).

The benefit of this data structure is, that there is no need for updating node labels after an edit-operation, since the position in memory of a splay tree - node never changes . The trade-off of this approach however is the slow down of the search operation from $O(m+k)$ to $O(m*\log n+k)$, with m the length of the search pattern and k the number of occurrences in the string. The time complexity for edit-operations(*delete, insert*) are $O((h(S)+b)*h(S)\log n)$, with $n=|S|$, b the length of the block of characters to be edited and $h(S)$ the length of the longest sub-string X of S that is repeated at least $|X|$ times in S. Furthermore, the depth of the position heaps is no more than $2h(S)$.

3.2 Referential Compression

As mentioned in the introduction, we use referential compression [WL13] to identify commonalities and differences between strings, in order to use them as input for our mounting algorithm.

> *A string S is a finite sequence over an alphabet Σ. The sub-string starting at position i with length n is denoted $S(i,n)$, with $i\geq 0$. A* referential match entry *(RME) is a triple $\langle start, length, mismatch\rangle$, where start is a number indicating the start of a match within the reference, length denotes the match length, and mismatch denotes a symbol. The length of a referential match entry RME, denoted $|RME|$, is length $+\,1$. Given strings S and ref, a* referential compression *of S with respect to ref, is a list of referential match entries,*
> $comp(S,ref)=[\langle start_1,length_1,mismatch_1\rangle\ldots\langle start_n,length_n,mismatch_n\rangle]$,
> *such that* $(ref(start_1,length_1)\circ mismatch_1)\circ(ref(start_2,length_2)\circ mismatch_2)\circ\cdots\circ(ref(start_n,length_n)\circ mismatch_n)=S$

In our example, the referential compression S against R has the following RMEs: $\{(0,5,A),(6,3,\$)\}$.

4 Index Mounting Algorithm

The starting point for the *mounting* process is the referential compression of S against the reference string R. Every RME has a description of an identical sub-string in form of a position-interval, which can also be referred to by other RMEs. To minimize the amount of *mounting* steps, the associated parts of the reference position heap for these identical sub-strings will be reused. In the following cases, such strategy can lead to an inconsistent state of the position heaps:

- *Incorrect/ non-linear order:* If a RME is in incorrect order or position, then there is a violation of the *heaps property* constraint of the position heaps. That is, that an internal node must have a smaller node label than its children nodes (Figure 3).
- *interval-overlapping:* If the intervals of two different RMEs are overlapped, there is a violation of the *uniqueness* constraint of the position heaps (Figure 4).

However, to identify these inconsistency problems, we have to analyse all RMEs first, before the *mounting* of the reference index can be applied. Our approach is therefore divided in two parts: analysing and *mounting* stage.

Analysing Stage

In the analysing stage we have to find a way to identify all cases of inconsistency. As for the overlapping problem, our idea is to sort all RMEs by their interval-start-values and then to compare the intervals of the consecutive entries. In addition, we envision a special case of interval-overlapping: *covering*, which is so far important, because our later to be described Algorithm 2 analyses only two consecutive RMEs at a time. As for example, let E_j and E_{j+1} two consecutive and not overlapped RMEs. But if there is a RME E_i, which is a predecessor of E_j and covers it entirely, E_i may overlaps with E_{j+1} (Figure 5). However, if there is no record for this covering-interval, especially where it ends, the overlapping between E_i and E_{j+1} will not be detected.

Definition 1. *Let E_i and E_j two augmented RME, with i and j their current position in the sorted RME-list , $i<j$ and $start(E_i)\leq start(E_j)$. E_i and E_j are* ***overlapped without covering****, iff $start(E_j)+length(E_j)>start(E_i)+length(E_i)$ and $start(E_j)\leq start(E_i)+length(E_i)$ applies.*

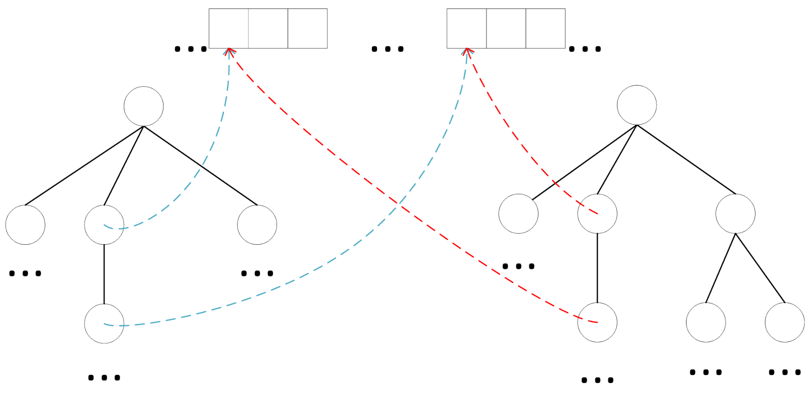
Fig. 3. Visualisation of the non-linear order problem. The original position heap is on the left side. It can be seen, that according to the heap property, the order of nodes corresponds with the position of the suffixes. On the right side however, the heap property is violated, due the non-linear order of the RMEs and the resulting nodes.

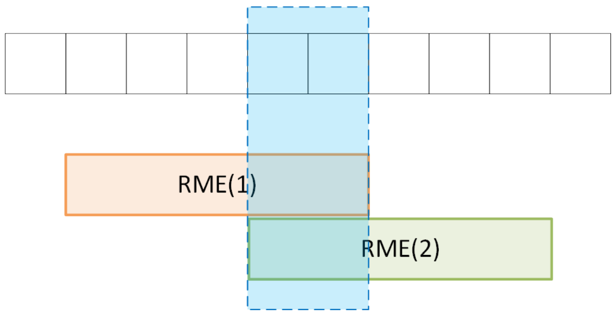

Fig. 4. Visualisation of the interval-overlapping problem.

Definition 2. *Let E_i and E_j two augmented RME, with i and j their current position in the sorted RME-list , $i < j$ and $start(E_i) \leq start(E_j)$. E_i is* ***covering*** *E_j , iff $start(E_i) + length(E_i) \geq start(E_j) + length(E_j)$ applies.*

And to check the order of the entries, we save their positions before and after the sorting to compare them later. We call such RME with additional informations the *augmented RME*. And therefore the first part of the analysing stage is to create the *augmented RME*-list, which is described in Algorithm 1.

Definition 3. *An* ***augmented RME*** *is a quintuple $\langle start, length, mismatch, start_t, pos\rangle$, where start, length and mismatch are the original components of the RME. $start_t$ denotes the new start position after the transformation and pos denotes the position of the RME within the original RME list.*

Definition 4. *Let E be an augmented RME, with $pos(E) = i$, its position within the RME-list before and j after the sorting. The entry E is in* ***non-linear order****, iff $i - j \neq 0$ applies.*

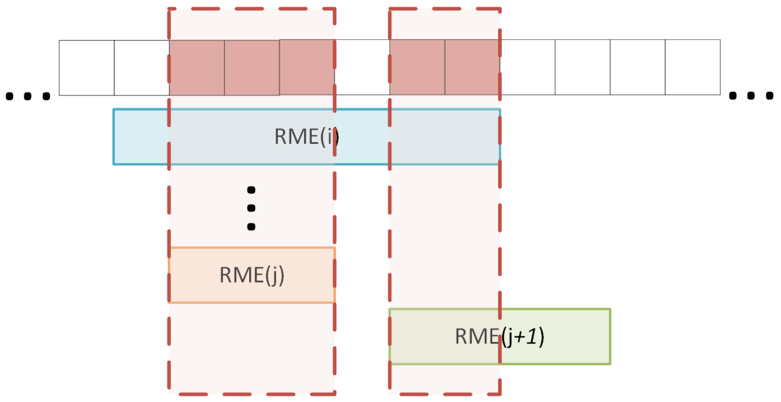

Fig. 5. Visualisation of the interval-covering problem.

Algorithm 1 *prepareList(rmeList)*

```
1: $pos := 1$
2: $start_t := 0$
3: List $augList := null$
4: RME $entry := null$
5: **while** $rmeList.hasNext$ **do**
6:     $entry\ \langle start, length, diff \rangle := getFirst(rmeList)$
7:     $augList.push(start, length, diff, start_t, pos)$
8:     $start_t := start_t + length + 1$
9:     $pos := pos + 1$
10: **end while**
11: sort the augList by start-values
```

In case of our example, $start_t$ of the first entry is 0 (Algorithm 1, line 2). As the match length of the first entry is 5, $start_t$ of the second entry is $0 + 5 + 1 = 6$ (line 8). The augmented RME list is finally: $\{(0, 5, A, 0, 1), (6, 3, \$, 6, 2)\}$.

In Algorithm 2, we describe the second part of the analysing stage, where the inconsistencies are identified and treated. The main idea of this algorithm is to analyse the left and right side of the RME-interval separately. If an entry is in an incorrect order, then the entire interval of this entry will be re-indexed. The basic principle for re-indexing is to remove and then re-insert the interval to the index. In case of overlapping, the corresponding sub-interval will also be re-inserted to the index(Figure 6).

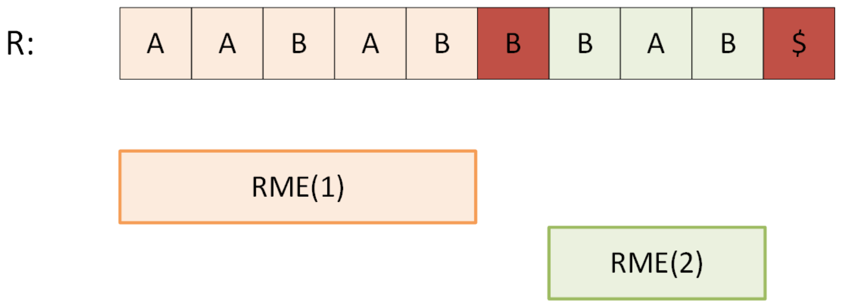

Fig. 6. Visualisation of Algorithm 2 with RME(1): $(0, 5, A, 0, 1)$ and RME(2): $(6, 3, \$, 6, 2)$. The red marked characters will be replaced with the *mismatch*-component of the RMEs.

However, it is possible, that some to be deleted or inserted parts of the index, are in the path of reference sub-string of other RMEs. Therefore, the position heap may enter an inconsistent state, if the insert or delete operations are applied immediately. As solution, all delete and insert assignments are saved in two extra lists. The insert list contains entries in form of a tuple (*block of characters, insert position*) and in case of the delete list (*interval*). In addition, the insert list is sorted by the insert positions and the delete list by the interval start values. The sorting ensures, that no delete or insert operation can be applied out of scope, for example like insert characters to a position, which is bigger then the size of the position heap at the moment. A simplified block diagram of this algorithm is shown in Figure 8.

Algorithm 2 $createTransformList(augList)$

```
 1: while not done do
 2:     get new entry from augList
 3:     add Diff to insert list
 4:     check for correct order
 5:     handle potential covering state of the left side
 6:     handle potential overlapping state of the right side
 7: end while
 8: sort delete list by interval-start
 9: sort insert list by insert positions
10: save delete list to disc
11: save insert list to disc
```

Index Mounting Stage

The actual *mounting* is described in the simple Algorithm 3. In the first step, the delete list will be loaded and applied on an in-memory reference index. In order to ensure, that no delete operation is applied to an incorrect position, the total number of current deleted characters is recorded and used to adjust the actual delete position. The final step is to load and apply the insert list.

Algorithm 3 $transformIndex$

```
1: read delL and execute
2: read insL and execute
```

We again take a look at our example:

- $R = AABABBBAB\$$,
- $S = AABABABAB\$$ and
- the augmented RME list as $\{(0, 5, A, 0, 1), (6, 3, \$, 6, 2)\}$.

In the first iteration we read (0,5,A,0,1). As the start position of this entry is 0 and the match length is 5, the insert position for the character A is therefore $0 + 5 = 5$ and we put (A,5) to the insert list. Since the original position (*pos*) of this entry in the list is 1, it is also in the correct order.

And now we read the second entry (6,3,\$,6,2). Since the matching block of the first entry ends at the position $0 + 5 - 1 = 4$, and the block of the second entry begins at position 6, there is a gap between the both intervals. To eliminate this gap, the interval (5,5) is added to the delete list.

The actual position of the current entry (6,3,\$,6,2) is 2. Because $pos - 2 = 2 - 2 = 0$, this entry is also in the correct order. Since the mismatch symbol is \$ and the current entry is also the last one on the list, there is nothing more to do.

And after processing all of the entries, we have:

- the delete list as $\{(5, 5)\}$, and
- the insert list as $\{(A, 5)\}$.

After applying the delete list, the modified string R' is $AABABBAB\$$ and $AABABABAB\$$ after the insert, which is identical to S (Figure 7).

In the optimal case, such that there is no overlapping and all entries are in correct order, the time complexity of our Algorithm 3 is $O((h(S)+|RMElist|)*h(S)logn$. Otherwise it is $O((h(S)+c)*h(S)logn$, with c is the total number of characters to deleted or inserted. Since all needed informations are on the delete and insert lists, the original RME-list can also be removed in order to reduce the space requirement.

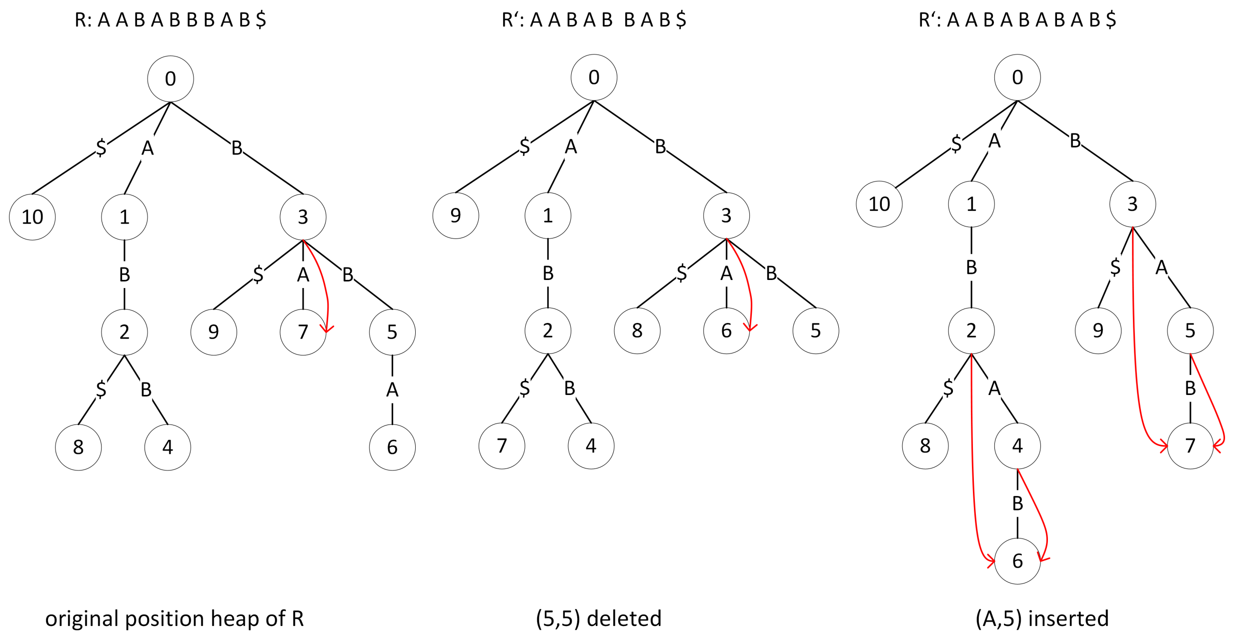

Fig. 7. Transformation steps of the position heap of R.

5 Implementation and evaluation

As our approach is intended to handle very large strings, C++ is therefore the programming language of choice, due to the support of manual memory management. As there is no implementation of (dynamic) position heaps available, the algorithm of this thesis will be applied with our own implementation of the position heaps. As for the realization of the approach of this thesis, we anticipate three major parts of the implementation:

– Augmentation of the RME list (offline)
– Generation of lists for delete and insert for a list of RMEs (offline)
– Index transformation (the actual mounting based on delete list and insert list)

As for the evaluation, we apply our implementation on a set of real world data: 1) human chromosomes and 2) textual data sets. We want to evaluate, whether our expectation is correct: For highly-similar strings, mounting is more fast than index construction from the scratch.

References

[CJE70] Edward G Coffman Jr and J Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, 1970.

[EMOW11] Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *J. of Discrete Algorithms*, 9(1):100–121, March 2011.

[FGM98] Paolo Ferragina, Roberta Grossi, and Manuela Montangero. On updating suffix tree labels. *Theoretical Computer Science*, 201(1):249–262, 1998.

[HK04] Khaled M Hammouda and Mohamed S Kamel. Efficient phrase-based document indexing for web document clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 16(10):1279–1296, 2004.

[HK13] Daniel Hupalo and Andrew D. Kern. Conservation and functional element discovery in 20 angiosperm plant genomes. *Molecular Biology and Evolution*, 30(7):1729–1744, 2013.

[HPB13] Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics (Oxford, England)*, 29(13):i361–i370, July 2013.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[KN13] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483(0):115 – 133, 2013.

[Kuc13] Gregory Kucherov. On-line construction of position heaps. *Journal of Discrete Algorithms*, 20(0):3 – 11, 2013. StringMasters 2011 Special Issue.

[Rit13] NWM Ritchie. The use of revision control to implement best practices for experimental microanalysis. *Microscopy and Microanalysis*, 19(S2):824–825, 2013.

[SLLM10] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8(2):241–257, 2010.

[WL13] Sebastian Wandelt and Ulf Leser. Fresco: Referential compression of highly-similar sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, page 1, 2013.

[WSBL13] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. RCSI: Scalable similarity search in thousand(s) of genomes. *PVLDB*, 6(13):1534–1545, 2013.

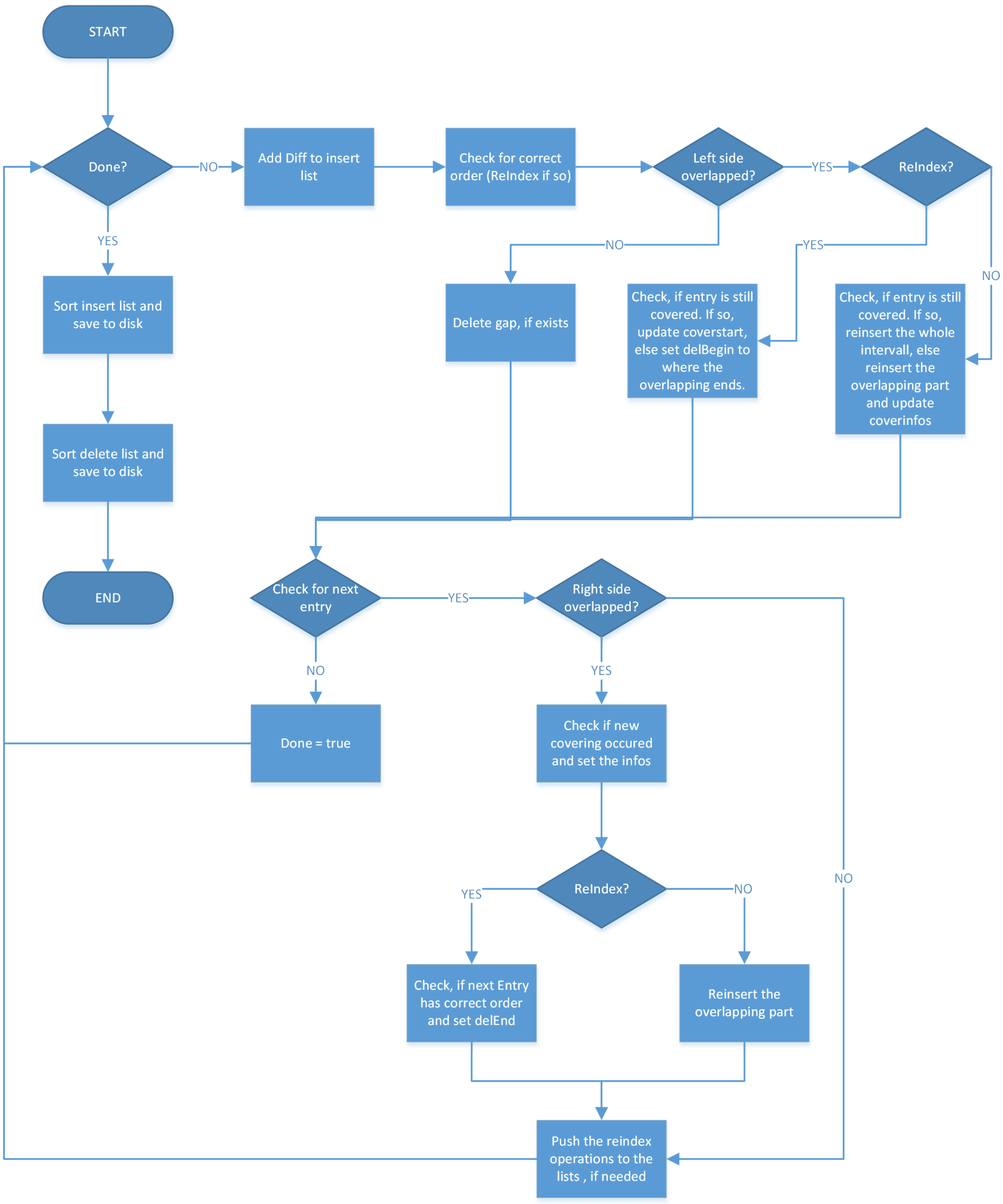
START
Done?
NO
Add Diff to insert list
Check for correct order (ReIndex if so)
Left side overlapped?
YES
ReIndex?
NO
YES
NO
YES
Sort insert list and save to disk
Delete gap, if exists
Check, if entry is still covered. If so, update coverstart, else set delBegin to where the overlapping ends.
Check, if entry is still covered. If so, reinsert the whole intervall, else reinsert the overlapping part and update coverinfos
Sort delete list and save to disk
END
Check for next entry
YES
Right side overlapped?
NO
YES
Done = true
Check if new covering occured and set the infos
YES
ReIndex?
NO
NO
Check, if next Entry has correct order and set delEnd
Reinsert the overlapping part
Push the reindex operations to the lists , if needed

Fig. 8. Simplified block diagram for Algorithm 2.