Department of Informatics, University of Zürich

BSc Thesis

Implementing an Index Structure for Streaming Time Series Data

Melina Mast

Matrikelnummer: 13-762-588

Email: melina.mast@uzh.ch

August, 2016

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]

Department of Informatics

Acknowledgements

Abstract

...

Zusammenfassung

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The thesis presents a way to implement the described data structures after discussing the requirements. Furthermore, it documents the out coming experimental results. In the end of the thesis, in Chapter 8, the findings will be summarized and concluded.

1.1 Thesis Outline

2 Background

A streaming time series s is a unbounded sequence of data points that is continuously extended, potentially forever. Streaming time series are relevant to applications in diverse domains for example in finance, meteorology or sensor networks. All domains have applications that need to be fed continuously with the latest data e.g. the financial stock market or the weather information. But the processing of large volumes of time series data is impractical. Therefore, a system can only keep a limited size of data in main memory.
The data that is kept in main memory needs to be limited to just a portion of the streaming time series. Besides, in order to be practical for a application like the financial stock market, the data that arrives in a defined time interval (e.g. every 2 minutes) needs to be completely processed until the succeeding data arises.

2.1 TKCM

A streaming time series is not always gapless. E.g due to sensor failures or transmission error, values can get missing. Wellenzohn et al.[1] presents a two-dimensional query pattern over the most recent values of a set of time series to efficiently impute missing values. The two-dimensional query pattern P_t is defined with *l* reference time series on the spatial dimension and a time window of length *p* on the time dimension. The idea is to derive the missing value from the *k* most similar past pattern. Therefore, it determines for each *time series* a set of highly correlated *reference time series* which represent similar situations in the past e.g. similar weather situations. The value $\hat{s}(t)$ that is calculated as the average of the values $\{s(t)|t \in T_k\}$ will be imputed. Hence, TKCM is able to calculate an estimation of a missing value in streaming time series data.

TKCM must not only insert missing values, but also process the newest arriving values efficiently. In order to do that, TKCM must provide an insertion method for new arriving values to insert the new value into the time window W. Since the time window has a limited, given size $|W|$, an old value has to be deleted for the new arriving value. Provided that, the oldest time *t* does no more fit into the time window because the window is already full.
Further, TKCM must be able to handle duplicate values. For example, if the time window contains 100 temperature values from the same weather station and every 5 minutes a new value arrives. It is likely that the same temperature value arrives multiple times. Besides, the most similar base time values for a given value *v* should be efficiently found and returned.
These assumptions can be made for the implementation of the index structure for streaming time series data.

2.2 Access Methods

TKCM initializes a set $T = \{\}$. The set is filled during execution with all time points t for which pattern P_t has already been compared to the query pattern $P_{\bar{t}}$. Besides, TKCM initializes a set $T^* = \{\}$ that contains the k time points $t \in T$ that minimize the error $\delta(P_t, P_{\bar{t}})$. Therefore, $T^* \subseteq T$ is always true during execution.
TKCM uses two methods for accessing any time series $r \in S$, *random* and *sorted* access. The two methods are defined as follows:

Definition 2.2.1 *Random Access. Random access returns value r(t), given time series r and time point t.*

Definition 2.2.2 *Sorted Access. Sorted access returns the next yet unseen time point $t_s \notin T$ such that the value $r(t_s - o)$ is most similar to a given pattern cell $P^{r,o}_{\bar{t}}$. $t(s)$ is defined as:*

$$t_s = \underset{t_s \in W \setminus T}{\operatorname{argmin}} |r(t_s - o) - P^{r,o}_{\bar{t}}|$$

After T and T^* is initialized, TKCM iterates until set T^* contains the k time points t that minimize the difference $\delta(P_t, P_{\bar{t}})$.
Using the sorted access mode, the algorithm loops through the cells $P^{r,o}_{\bar{t}}$, reading the next potential time point $t_s \notin T$. The time point $t_s \notin T$ is added to T. The time point t_s has a corresponding patter P_{t_s} which is at least for one pattern cell similar to the query pattern $P_{\bar{t}}$.
The random access mode is used to look up the values that pattern P_{t_s} is composed of. After each iteration a threshold τ is computed. The threshold τ is a lower-bound on the error $\delta(P_{t'}, P_{\bar{t}})$ for any time point t' that is yet unseen. Therefore, during the execution of the algorithm $\forall t' \in T : \tau \leq \delta(P_{t'}, P_{\bar{t}})$ is valid. Informally this significances that the lower-bound is always smaller or equal to the error between pattern $P_{t'}$ and query patter $P_{\bar{t}}$ for all time points t' that are elements of T. Once $\forall t \in T^* : \delta(P_t, P_{\bar{t}}) \leq \tau$ the algorithm terminates. At the end, $T^* = T_k$.

3 Problem Definition

The present thesis tries to introduce an efficient way to implement the $random$ and $sortedaccess$ methods described in Section 2.2 for a streaming time series s.
Let $W = [\underline{t}, \bar{t}]$ be a sliding window of length $|W|$. Time $\underline{t}$ stands for the oldest time point that fits into the time window and $\bar{t}$ stands for the current time point for which the stream produced a new value. Besides, consider a set $S = \{s_1, s_2, ...\}$ of streaming time series. The value of time series $s \in S$ at time *t* is denoted as $s(t)$. Only the values in the time window W are kept in main memory. However, we assume that all the time points $t < \bar{t}$ have a time series *s* that is complete. Hence, $\forall t < \bar{t} : s(t) \neq NIL$ since *s* contains imputed values if the real ones were missing.

3.1 Operations

The system presented in the present thesis needs to efficiently perform on the streaming time series s in a sliding window $|W|$:

- shift($\bar{t}, v$): add value *v* for the new current time point $\bar{t}$ and remove value *v'* for the time point $\underline{t} - 1$ that just dropped out of time window W.
- lookup(t): return the value of time series *s* at time *t*, denoted by $s(t)$.
- neighbor(v, T): given a value *v* and a set of time points T, return the time point $t \in T$ such that $|v - s(t)|$ is minimal.

The $lookup$ operation is a random access method, while the $neighbor$ operation is a sorted access method.
Wellenzohn et al.[1] suggests a combination of two data structures: a B^+ tree and a circular array. The lookup operation can be performed by the circular array, while the neighbor operation takes advantage of the fact that the leaves of a B^+ tree are sorted.
The approach presented in in Chapter 4 the implementation of the random and sorted access modes using the suggested data structures. Further, it proposes a solution to handle duplicate values.

4 Approach

The lookup operation can be efficiently performed by the circular array, while the neighbor operation takes advantage of the fact that the leaves of a B^+ tree are sorted.
Each time series $s \in S$ can be implemented as a circular array. The circular array is kept in main memory. It uses random access to look up value $s(t)$ for a given time t. Further, for each time series s a B^+ tree is maintained that is also kept in main memory. The B^+ tree is ideal for sorted access by value and therefore for range queries. Both data structures are described in detail in Section 4.1 and Section 4.2.

4.1 Circular Array

A circular array is used to store the time series data. The data is assorted by time. Further, the time interval is predefined e.g. every 5 minutes a new value arrives.
The value and time are directly stored in the circular array. The last update position is stored in a variable and updated with every insertion. The circular array is shown in Figure 4.1.

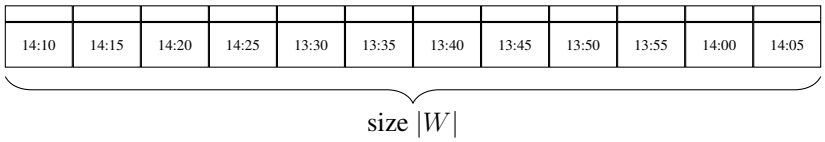

Figure 4.1: Circular array of size $|W|$.

The circular array stores the data, containing all measurement time stamps and values, the size, the last update Position and a counter, which counts the number of measurements added to the array. The addition of a new measurement to the array is presented in Algorithm 4.1.1.

4.1.1 Add a new Value to the Circular Array

If the counter of the array is equal or bigger than the size of the array, there is a measurement at the update position in the array that needs to be deleted. If not, there is no need to delete a value from the B^+ tree, since no value is overwritten in the circular array.

Algorithm 1 Update Circular Array

```
1  void update_CircularArray(BPlusTree *tree, CircularArray *array,
       timeStampT newTime, double newValue){
2
3      int newUpdatePosition = 0;
4
5
6      //array is not full yet
7      if(array->count < array->size){
8          if(array->count != 0){
9              newUpdatePosition = (array->lastUpdatePosition + 1) %array->size;
10         }
11         array->count++;
12     }
13
14     //array is full -> one value is inserted and one is deleted
15     else{
16         newUpdatePosition = (array->lastUpdatePosition + 1) %array->size;
17         //delete measurement from tree and circularArray
18         delete(tree, array->data[newUpdatePosition].time, array->data[
               newUpdatePosition].value);
19
20     }
21
22     //update circularArray
23     array->data[newUpdatePosition].time = newTime;
24     array->data[newUpdatePosition].value = newValue;
25     array->lastUpdatePosition = newUpdatePosition;
26
27     addRecordToTree(tree, newTime, newValue);
28
29 }
```

4.1.2 Random Access: Lookup a Value

Due to the properties of a circular array the lookup of a value at time t is very efficient. Since the position can be directly calculated without looping through the array by using the $TIMESTAMP\_DIFF$ representing the interval between two consecutive measurements. The last update point can be used as reference time point for the calculation.

Algorithm 2 Lookup

```
1  bool lookup(CircularArray *array, timeStampT t, double *value){
2
3     //array is empty
4     if(array->count == 0){
5        return false;
6     }
7
8     //steps from last timestamp to new timestamp
9     int step = (int)(t - array->data[array->lastUpdatePosition].time)/
          TIMESTAMP_DIFF;
10
11    //checks if array has enough values inserted for the necessary steps
12    if(abs(step) < array->count){
13
14       //chaining modulo for negative numbers
15       int pos = (((array->lastUpdatePosition+step)%array->size)+array->
             size)%array->size;
16
17       //value has been found
18       if(array->data[pos].time == t){
19          //set the pointer to the found value
20          *value = array->data[pos].value;
21          return true;
22       }
23    }
24    //value was not found
25    return false;
26 }
```

4.2 B^+ Tree

A B^+ tree is able to execute range queries very efficiently, since the leaves of a B^+ tree are ordered and linked. To perform the *neighbor*(v, T) operation described in Section 3.1, the B^+ tree for our requirements has leaves linked in both directions. The Section 4.2.1 presents the structure of the B^+ tree we used for the implementation.

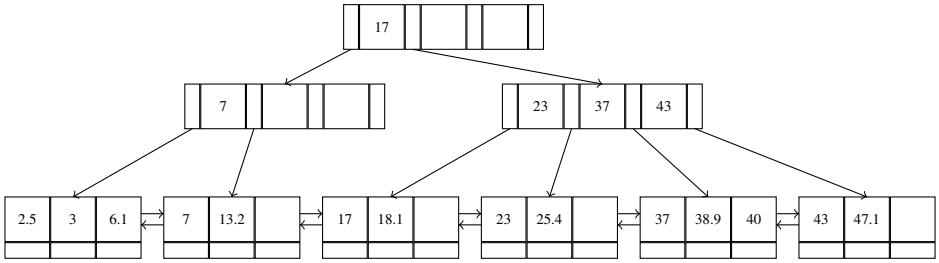

Figure 4.2: Example of a B^+ tree

4.2.1 The Structure of the used B^+ tree

The structure of the used B^+ tree is based on the book from Silberschatz et al. [4].
A B^+ tree is organized in blocks, as implied by its name. All paths from the root to a leaf have the same length which significances that the tree is always *balanced*.
There are three types of nodes that may exist in a B^+ tree: the root, interior nodes and leave nodes. The parameter *n* determines the number of search-keys and pointers in a node. The interior nodes can have maximum *n-1* search-keys and *n* pointers, pointing to its child nodes. Each interior node must have a minimum of $[n/2]$ pointers and can have maximum n pointers. The root node is the only node that can contain less than $[n/2]$ pointers. The root node must have minimum one searchkey and two pointers to child nodes, unless the root node has no children and therefore is a leaf node.
A node containing m pointers $(m \leq n)$. For $i = 2, 3, ..., m - 1$, pointer P_i points to the subtree that contains searchkey values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}.

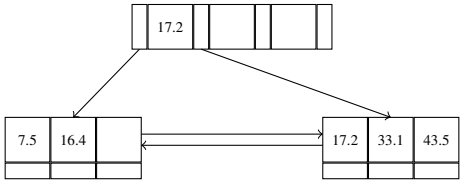

Figure 4.3: Left children keys < 17.2 and right children keys $\geq$ 17.2

What can appear in blocks is listed in the following:

- The keys in the leaves are sorted from left to right.
- All pointers in a node point to the level below.
- The interior nodes use at least $\lceil \frac{n+1}{2} \rceil$ of its $n + 1$ pointers. In the root at least 2 pointers must be used.
- The first pointer in a node where the first key is *K* points to a node and hence a part of the tree where the keys are less than *K*. The second pointer points to a node where the keys are greater than or equal to *K*, as shown in Figure **??**.

The B^+ tree used for the implementation of the streaming time series data is slightly different to the B^+ tree in [3]. The required properties are the following:

- The search-keys of the B^+ tree are the temperature values in the sliding window W.
- The leaves are linked in both directions to efficiently perform the *neighbor*(v, T) operation.
- The leaves are sorted by the temperature values.
- The interior nodes have a temperature search-key.
- The leaves store the record of the time series data. A record consists of the temperature value and the associated time value.

Due to weather conditions the temperature values in the time window W can occur several times. Therefore, the keys are not unique. Since the temperature values are used as search-keys, the B^+ tree must be able to handle duplicate values. Section 4.5 proposes different possibilities that allow to use duplicate values in a B^+ tree.
Unlike the traditional B^+tree, where the leaves are just linked to their succeeding leaf, like shown in Figure **??**, the leaves are linked to the succeeding as well as the preceding leaf as shown in Figure **??**.

4.3 Combination of the Data Structures

To efficiently implement the above mentioned operations the system combines two data structures: a circular array and a B^+ tree. In Figure 4.4 the originally proposed data structure in [1] is shown. The B^+ tree is connected with pointers to the circular array and vice versa. Further, the temperature values are used as keys in the B^+ tree. The size of the circular array is defined as $|W| + 1$ since an empty field is used to identify the current update position. The size $|W|$ and the order of the B^+ tree, which defines the size of the nodes, are parameters. Hence they can be changed.
The circular array and the B^+ tree are characterized in the following.

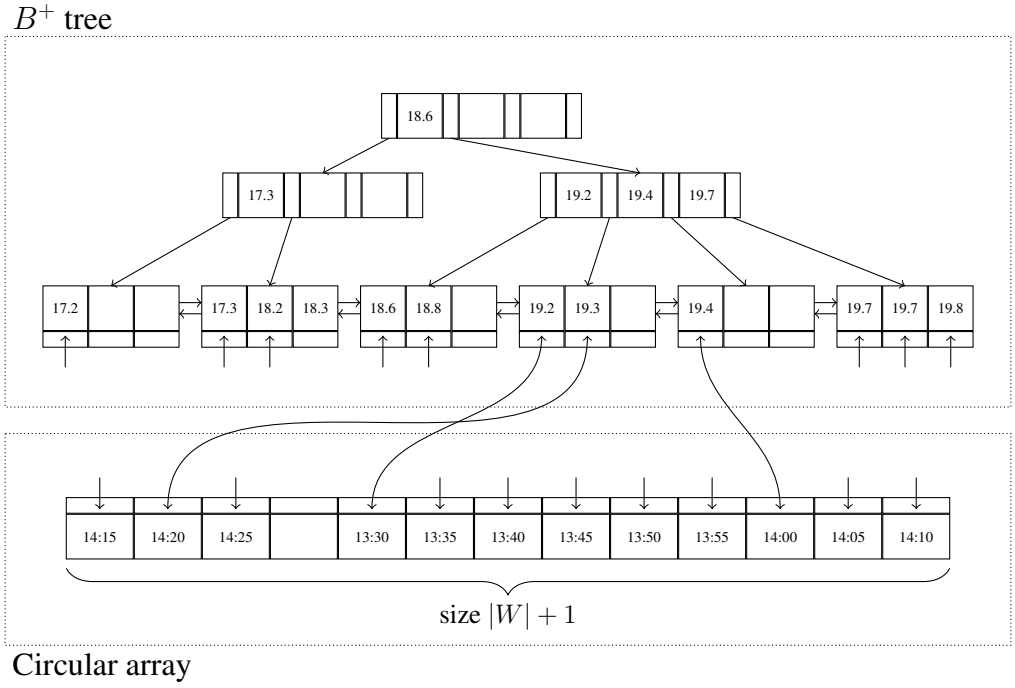

Figure 4.4: Proposed data structures in [1].

4.4 Sorted Access: Next Pattern Search

4.5 Handling Duplicate Values in B^+ trees

This Section presents a solution to allow duplicate values in a B^+ tree. Further, the advantages and disadvantages are discussed and some advantages over other approaches are illustrated.

4.5.1 Associated doubly, circular Linked List

The idea of this method is to associate a doubly, circular linked list to the each key. If a value key occurs multiple times, the new time point is added to the linked list. I So instead of inserting the key again and using another block in the leaf, the new time point is inserted as a linked list value.
Associating a doubly, circular linked list that is interconnected in both directions is ideal for our requirements. The oldest value in the list, so the lowest time point, always is the one connected to the leaf key. Hence, since a shift operation on the circular array leads to a deletion of the oldest measurement, is is always the time point connected to the leaf key that must be deleted. Also, a new measurement can be inserted without looping through the list. It is always added to the position before the oldest time point. The Figure 4.5 illustrates that the oldest time point, here $14:15$, is connected to the tree and the newest time point $14:50$ is at the previous position.
The *neighborhood grow* operation searches a specific time point in the doubly, linked list. Therefore, it cannot just take the oldest or newest time point position like with an insertion or deletion. Hence, In the worst case the entire list would be searched for the specific time point. But since the *neighborhood grow* operation always is executed at the newest measurements in the circular array, we can give an upper bound, namely the pattern length. Therefore the worst case depends on the pattern length and on the distribution of the measurements which are starting points of the $neighbor$ method.

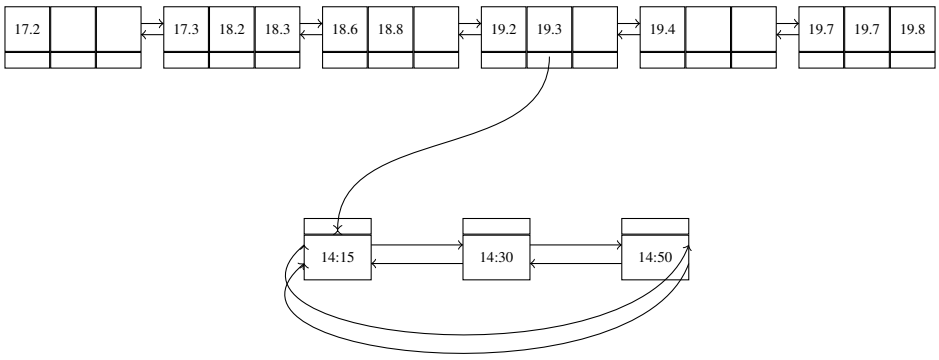

Figure 4.5: Doubly, circular linked list

4.5.2 Alternative Approaches

A similar idea as in our approach is to add a associate list to the each key that occurs multiple times. So instead of inserting the key again and using another block in the leaf, the new time point is just inserted to its associated list. The Figure 4.6 illustrates the associated list.

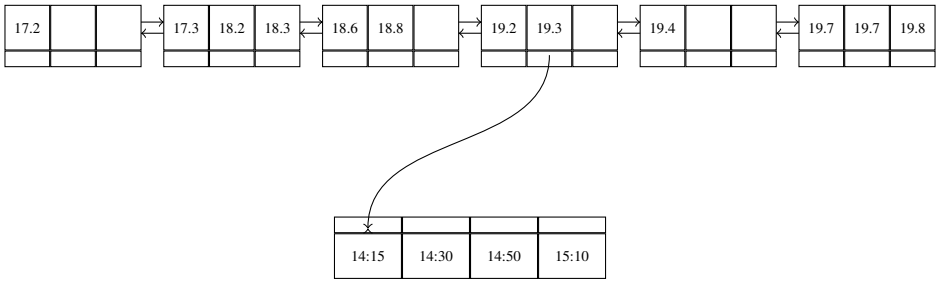

Figure 4.6: Associated List Approach.

A new time stamp can be inserted to the end of the list in $O(1)$ and since the time window W slides forward, the value that should be deleted first from the tree, normally, is at the first position in the list. Therefore, a value can be deleted in $O(1)$ from the list as well as with a doubly, circular linked list. But here the array cannot dynamically be extende since the array size must be reallocated with every additional time stamp.

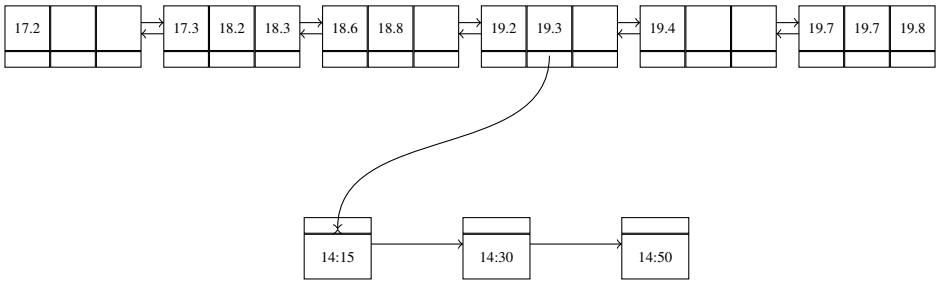

Figure 4.7: Singly linked list

A singly linked list uses less pointer than a doubly, circular list but since a insertion would cost $O(n)$ because a new value is always inserted to the end of the singly linked list and hence all older time points in the list need to be checked. Therefore, a circular, linked list is more suitable for our requirements than a singly, linked list.
Another idea to handle duplicate values is to add additional leaves to the tree that do not have a parent node. As shown in Figure 4.8, the node containing the temperature value *18.3* had been split, since the values did no longer fit into one leaf. The value *18.4* would belong into the same leaf as *18.3* but there is no more space. Instead of splitting the leaf, the additional leaf without a parent is filled up. If e.g. a value *18.5* must be inserted the leaf without a parent must be split. The new leaf would again receive a parent and the old leaf including the duplicate values would stay parent-less. But unlike the doubly, circluar linked list approach searching a specific record may take long, depending on the number of duplicate temperature values to the left side of the record.
The book *Database Systems - The Complete Book* [3] presents an additional approach to handle duplicate values. The definition of a key is slightly different when allowing duplicate

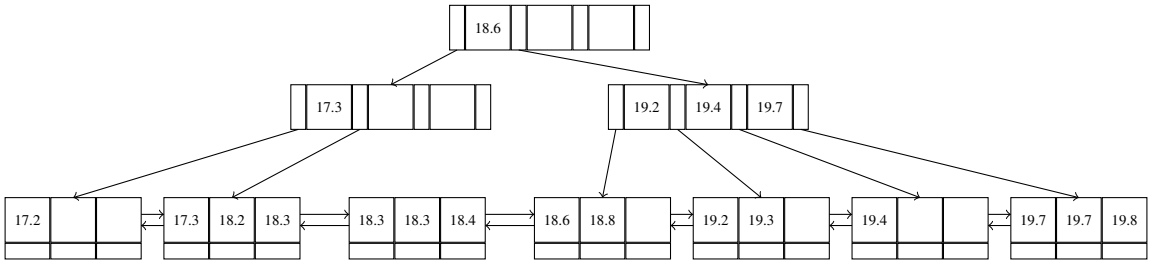

Figure 4.8: B^+ tree with an additional leaf without a parent.

search-keys. The keys the interior node $K_1, K_2, K_3, ..., K_n$ can be separated to *new* and old keys. K_i is the smallest new key that is part of the sub-tree linked with the $(i+1)$st pointer. If there is no new key associated with the $(i+1)$st pointer, K_i is set to null.

Example 4.5.1 *The example illustrated in Figure 4.9 illustrates the case, in which the interior node in the right sub-tree K_1 is set to null. The leaf node that pointer $(1+1)$ is associated with contains only duplicate key values which is indicated by setting the interior node K_1 to null. The search-key in the root node is set to 17 because it is the lowest new key in the right sub-tree. Since 13.2 is already in the left sub-tree, the duplicate search-key in the right sub-tree cannot be the key in the root.*
If e.g. 14.2 would have been added to the tree, the leaves must be reordered, since the B^+ tree property that all leaves are ordered from left to right would be hurt.

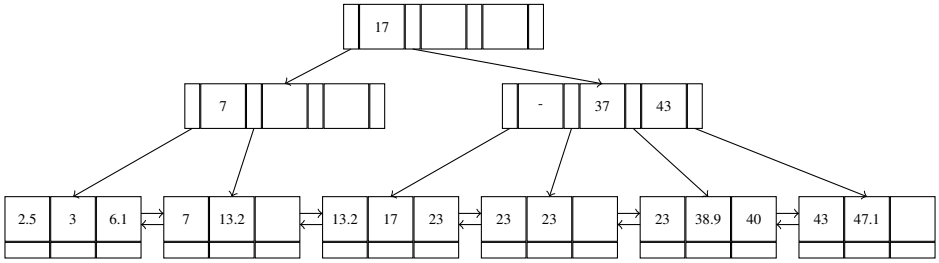

Figure 4.9: Duplicate handling proposed in [3].

Unlike our chosen approach the right sub-tree may also contain keys that are lower than the root key. Therefore the neighbour leaves must be checked as well when searching for a particular key. Besides, in some cases the leaves have to be reordered and in case of duplicate values the neighbour leaves has to be checked as well to find the insertion point for a new key.

5 Complexity Analysis

5.0.1 Runtime Complexity

5.0.2 Space Complexity

6 Evaluation

memory und runtime evaluation: nodesize, verteilung der daten Datenset erstellen

6.1 Experimental Setup

6.2 Results

6.3 Discussion

7 Related Works

8 Summary and Conclusion

"The conclusion (10 to 12 per cent of the whole research thesis) does not only summarize the whole research thesis, but it also evaluates the results of the scientific inquiry. Do the results confirm or reject previously formulated hypotheses? The conclusion draws both theoretical and practical lessons that could be used in future analyses. These lessons are to be embedded as 2 recommendations for the research community and for policy-makers (note: policy relevance instead of policy prescriptive). In addition, the conclusion gives insights for further research."

Bibliography

[1] K. Wellenzohn, M. Böhlen, A. Dignos, J. Gamper, and H. Mitterer: *Continuous imputation of missing values in highly correlated streams of time series data*; Unpublished, 2016.

[2] Themistoklis Palapanas, Michail Vlachos, Eamonn Keogh, Dimitrios Gunopulos, Wagner Truppel: *Online Amnesic Approximation of Streaming Time Series*; University of California, Riverside, USA, 2004. `http://www.cs.ucr.edu/~eamonn/ICDM_2004.pdf`

[3] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: *Database Systems - The Complete Book*; ISBN 0-13-031995-3, 2002 by Prentice Hall

[4] Abraham Silberschatz, Henry F. Korth, S. Sudarshan: *Database System Concepts*; ISBN 978-0-07-352332-3, 2011 by The McGraw-Hill Companies, Inc. p. 496-