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

D B T G

Acknowledgements

Abstract

Many domains have applications that need to be fed continuously with the latest data. The financial stock market need to be fed with the latest exchange rates. This time series data is continuously extended, potentially even forever.

We propose a system that keeps a portion of time series data in main memory and which allow an efficient random and sorted access to the data in a time series.

Zusammenfassung

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

A streaming time series s is an unbounded sequence of data points that is continuously extended, potentially forever. Streaming time series are relevant to applications in diverse domains e.g. in finance, meteorology or sensor networks. Many domains have applications that need to be fed continuously with the latest data, like the financial stock market or the weather information. A system can only keep a limited size of data in main memory. Also the processing of large volumes of time series data is impractical.

The data kept in main memory needs to be limited to a portion of the streaming time series. In order to be practical for an 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. 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 is not always gapless. Due to sensor failures or transmission errors, values can get missing. To efficiently impute missing values, Wellenzohn et al.[1] present the Top-k Case Matching algorithm (TKCM). The algorithm is introduced in Section 2.1.

2.1 TKCM

TKCM defines a two-dimensional query pattern over the most recent values of a set of time series. The idea is to derive the missing value in a time series s from the k most similar past pattern. Therefore, it determines for each *time series s* a set of highly correlated *reference time series* which exhibit similar behaviour to the base station e.g. similar weather situations. Hence, TKCM is able to calculate an estimation of a missing value in streaming time series data.

Definition 2.1.1 *Measurement. A measurement consists of a value v at a time point t.*

Definition 2.1.2 *Time window W. 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.*

Definition 2.1.3 *Streaming Time Series. Let S be 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 $s(t)$. For base time series s, let $R_s = \langle r_1, r_2, ...\rangle$ be an ordered sequence of the time series $r_i \in S \setminus \{s\}$. The set of reference time series for s, R_s^d, at the current time $\bar{t}$ are the first d time series in R_s for which $r(\bar{t}) \neq$ NIL. The time points in a streaming time series s are in time window W.*

Definition 2.1.4 *Pattern. Let $R_s^d = \{r_1, ..., r_d\}$ be the ordered set of reference time series for a time series s. A $pattern$ $P(t)$ of length $l > 0$ over R_s^d that is embedded at time t is defined as a $d \times l$ matrix $P(t) = [p_{ij}]_{d \times l}$.*

The two-dimensional query pattern $P(t)$ is anchored at a time point t and consists of the subsequence of length l spanning from $t - l + 1$ to t of each reference time series. Each row represents a subsequence of a reference time series and each column represents the values of the reference time series at a time point.
Every reference time series has associated reference time series as well, to keep the own data complete.
Only the values in the streaming time series with time points in 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.

TKCM must not only recover and impute 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 a streaming time series s. Since the time window has a limited, given size $|W|$, one value has to be deleted from the time series data for each new arriving value. Provided that, the time series data in window W is already completely filled.

Further, TKCM must be able to handle duplicate values. Assume the time window contains 100 temperature values from the same weather station and every 5 minutes a new value arrives. It is possible that the same temperature value arrives multiple times.

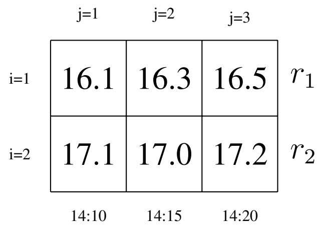
2.2 Access Methods

TKCM uses two methods for accessing any time series $r \in S$, *random* and *sorted* access. *Sorted* access is used for finding the most similar value to a given pattern cell and *random* access finds the values to fill the rest of the pattern cells. 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.*

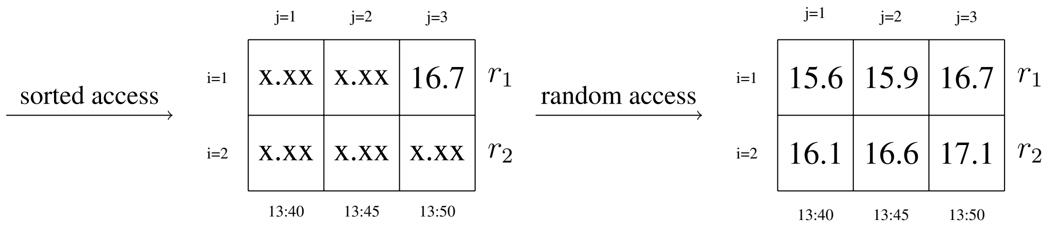
Definition 2.2.2 *Sorted Access. Sorted access returns the next yet unseen time point $t_s \notin T$ such that the value $r(t_s)$ is most similar to a given pattern cell P_{ij}. $t(s)$ is defined as:*

$$t_s = \underset{t_s \in W \setminus T}{\operatorname{argmin}} |r(t_s) - P_{ij}|$$



	j=1	j=2	j=3	
i=1	16.1	16.3	16.5	r_1
i=2	17.1	17.0	17.2	r_2
	14:10	14:15	14:20	

Figure 2.1: Query Pattern Q(t) of length $l = 3$ and $d = 2$ reference time series



	j=1	j=2	j=3	
i=1	x.xx	x.xx	16.7	r_1
i=2	x.xx	x.xx	x.xx	r_2
	13:40	13:45	13:50	

	j=1	j=2	j=3	
i=1	15.6	15.9	16.7	r_1
i=2	16.1	16.6	17.1	r_2
	13:40	13:45	13:50	

Figure 2.2: Pattern for query pattern cell q_{13}

TKCM initializes a set $T = \{\}$. The set is filled during execution with all time points *t* for which a pattern P(t) has been found. Using the sorted access mode, the algorithm finds the next yet unseen time point $t_s \notin T$ for which the value is most similar to a given value in a pattern cell p_{ij}. The time point $t_s \notin T$ is added to T. The time point t_s has a corresponding pattern $P(t_s)$ which is at least for one pattern cell similar to the query pattern cell p_{ij}.
The random access mode is used to look up the values that pattern $P(t)$ is composed of.

3 Problem Definition

The present thesis tries to introduce an efficient way to implement the $random$ and sorted access methods described in Section 2.2 for a streaming time series s.

3.1 Context

We make the following assumptions for our system:

- The values arrive in a fix interval. E.g. every five minutes.
- There are no gaps between the arriving values.
- There are no values arriving out-of-order.

3.2 Operations

The system 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.
- new_neighborhood(t, v, j, l): given a value *v* for the time point t, the pattern length l and the index j, return the new neighborhood N at t.

Wellenzohn et al.[1] suggest 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.
Additional reasons for the data structures are described in Chapter 4. Further, the implementation of the random and sorted access modes using the suggested data structures is presented and a solution for handling duplicate values is proposed.

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. We have a circular array and a B^{+}tree for every time series s we have new arriving measurements. E.g. for every sensor in a field that measures the weather temperature we update these two data structures. 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, sorted by time. Further, the time interval is predefined e.g. every 3 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 data structure consists of the following variables:

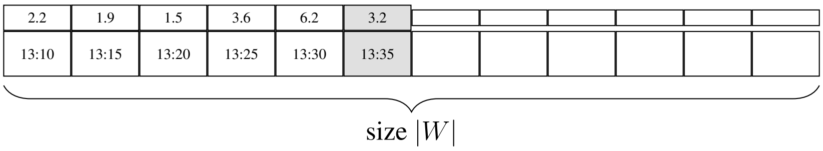
```
struct {
  timeStamp time;
  double value;
} Serie;

struct {
  Serie * data;
  int size;
  int lastUpdatePos;
  int count;
} Circular Array;
```

The circular array c has multiple variables: a counter, which counts the number of measurements in the array, a $lastUpdatePos$ to store the position that was last updated by a new arrived measurement, the size of the array which also represents the capacity for the number of measurements the array can hold and the data, which actually holds the time point and value of a measurement.

4.1.1 Phases

The circular array has two phases:

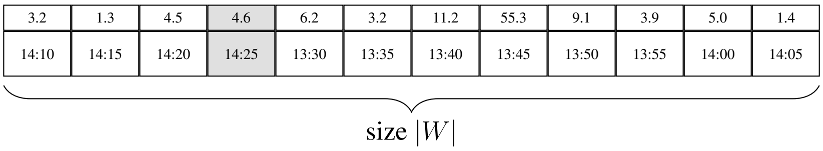
- First, the array is not full and is filled with every new arriving measurement until every position in the circular array is filled.



2.2	1.9	1.5	3.6	6.2	3.2						
13:10	13:15	13:20	13:25	13:30	13:35						

Figure 4.1: Unfilled circular array of size $|W|$.

- After the circular array is full the number of measurements stays constant



3.2	1.3	4.5	4.6	6.2	3.2	11.2	55.3	9.1	3.9	5.0	1.4
14:10	14:15	14:20	14:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05

Figure 4.2: Filled circular array of size $|W|$.

4.1.2 Observation

The measurements in a circular array are stored in a defined interval without any gaps in between. Therefore the value position and insertion can be easily calculated.

Example 1 *The value at time point 14.25 is the newest measurement. Hence, the last update position is at time point 14.25. A new measurement will be inserted at the next position in the circular array. So at the position of the oldest time point 13:30*
In order to lookup the value at time point 14:00 we can take advantage of the fixed interval. If the last update position is at time point 14.25, we can directly calculate the position for time point 14:00, using the last update position and the fixed interval.

The detailed insertion of a new measurement is presented in Algorithm 3 and the lookup of a value at time point t is presented in Algorithm 13.

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 3.2, the B^+tree we use has leaves linked in both directions. The Section 4.2.1 presents the structure of the B^+tree we used for our implementation.

4.2.1 The Structure of the used B^+tree

The used B^+tree and the implementation is based on the book of Silberschatz et al.[3]. We introduce the most important properties of a B^+tree, for further information please refer to the book.

The difference between the traditional B^+tree and the B^+tree we use is on the one hand, that the leaves are linked to the succeeding as well as the preceding leaf to efficiently perform the *neighbor*(v, T) operation and on the other hand, that our B^+tree is able to handle duplicate values. How our tree handles duplicate values is described in Section 4.3. The other properties of our B^+tree are presented in the following:

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 searchkeys and pointers in a node.

Leaf A leaf node must have at least $\lceil (n-1)/2 \rceil$ keys and may hold at most $n-1$ keys.

Interior Node The interior nodes can have at most *n-1* searchkeys and *n* pointers, pointing to its child nodes. The structure of nonleaf nodes like interior nodes and the root, is the same as for leaf nodes. Except for the pointers which points to tree nodes. An interior node must have at least $\lceil n/2 \rceil$ pointers. Hence, it must have at least of $\lceil n/2 \rceil - 1$ keys and can hold at most n pointers.

Root The root node is the only node that can contain less than $\lceil n/2 \rceil$ pointers. The root node must have at least one searchkey and two pointers to child nodes, unless the root is a leaf node and hence has no children.

Example 2 *If n is set to 7, an internal node may have between 4 and 7 children and therefore between 3 and 6 keys. The root may have between 2 and 7 children or if it is the only node in the tree it can have no children and just 1 key. A leaf node must have at least 3 keys and can have maximum 6 keys.*

A node contains the following attributes:

```
struct {
  struct Node *parent;
  void ** pointers;
  int numOfKeys;
  double * keys;
  bool isLeaf;
  struct Node *prev, *next;
} Node;
```

The node structure can be used for every type of node, since the structure of the root, the inner nodes and the leaf nodes is similar. A B+tree is built out of multiple nodes that include: A pointer to the parent, which is *NIL* if there is no parent, pointers to the child nodes or in leaves pointer to measurement time points, the number of keys that a node holds at the moment, the actual keys, a boolean which is true if the node is a leaf node and a node also has two pointers to the previous and the next node. These two pointers are only used if the node is a leaf node. All paths from the root to a leaf have the same length. This significances that the tree is always *balanced.*

The keys in an inner node and in a leaf node are always sorted from left to right. A node contains m non-null 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}.

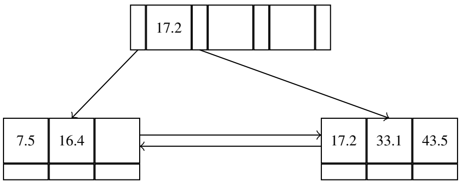


Figure 4.3: Left children keys < 17.2 and right children keys $\geq$ 17.2

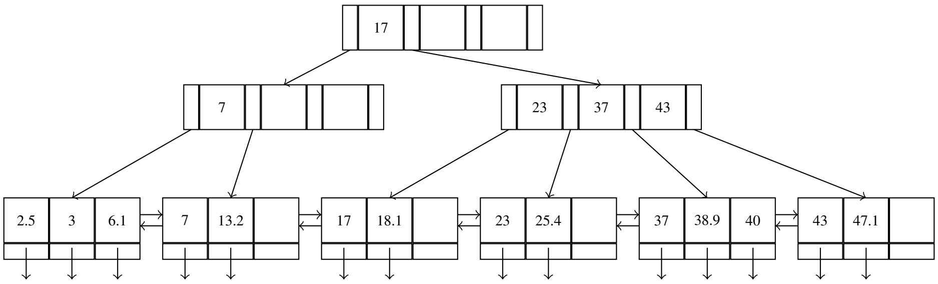


Figure 4.4: Example of a complete B^+tree

4.2.2 Application to our System

The B^+tree described above contains searchkeys. In our case these searchkeys are the measurement values in our circular array. A value in time series s can occur multiple times because the values are not unique. But for an efficient traversal of the tree the searchkeys should be comparable to each other. But, since the values are used as searchkeys, the B^+tree must be able to handle possible duplicates. Besides, the B^+tree needs to store the time point of a measurement because the time point is the amendment to the value of a measurement that makes it uniquely identifiable. Also the time point is returned by the neighbor(v, T) operation which is executed on the B^+tree. Section 4.3 proposes an approach that allow to use duplicate values in a B^+tree and it explains how the time points of the measurements are stored.

4.2.3 Observation

The next higher value hv of a given value v in a B^+tree is at the same time the right neighbor of v and the next lower value lv is also the left neighbor of v, since the leaves and their keys are sorted from left to right.

Example 3 *Assume we want to know the most similar value to the key* 40 *in the* B^+*tree illustrated in Figure 4.4. We just have to compare* 40 *to two values, namely the right neighbor* 43 *and the left neighbor* 38.9. *Hence, we find out that* 38.9 *is the most similar value in the entire* B^+*tree.*

Another property of the B^+tree we can take advantage is that the path from the root to a leaf node has always the same length, thus, the number of nodes to traverse to find a specific value in a leaf is always the same. Therefore, searching a value takes always the same number of node traversals. This leads to a more constant computation time.

4.3 Handling Duplicate Values

This Section presents a solution and its advantages to allow duplicate values in a B^+tree for our purposes.

4.3.1 Associated doubly, circular Linked List

The idea of this method is to associate a doubly, circular linked list to the each key in a leaf node. Cormen et al.[4] define linked lists as follows:

Definition 1 *Linked List. A linked list is a data structure arranged in a linear order. The order is determined by a pointer in each object. It can either be sorted in the linear order of the keys stored in elements or it can be unsorted. Given an element* e *in the list,* $e.next$ *points to the successor in the linked list* L*. The first element, or head, of the list has no predecessor and the last element, or tail, has no predecessor. A linked list may be either singly linked or doubly linked. A doubly linked list has an additional pointer in an element* e *which points to the predecessor, namely* $e.prev$*.*
Each element of a doubly, linked list L *is an object with an attribute* key *and two pointer attributes:* $next$ *and* $prev$*. In a circular list the* $prev$ *pointer of the head of the list points to the tail and the* $next$ *pointer ot the tail points to the* $head$*. If the element* e *is the only element in the doubly, linked list the pointers* $next$ *and* $prev$ *point to* e *itself.*

We use a doubly, circular linked list for our system, where time points are stored as the key of a list element. Every searchkey in a leaf node of our B^+tree has an associated linked list. If a measurement value occurs multiple times in the time series s, the time points of the measurements are simply added to the linked list associated to the searchkey representing the value of the measurement. So instead of inserting the key again and using another position 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 satisfying our requirements. The oldest value in a list, so the lowest time point, always is the element connected by a pointer from the leaf key to the list. Even though the doubly, circular linked list not really has an end and a beginning, we name the time point associated to the leaf the $head$ and we call the heads predecessor the $tail$.

The Figure 4.5 illustrates the leaf level of a B^+tree and the associated linked lists. It shows that the oldest time point, here 14:15, is connected to the tree and the newest time point, 14:50, the tail, is at the previous position. Also, the Figure illustrates that a single value in a doubly, circular linked list is linked to itself. The upper level of the B^+tree and the rest of the linked lists are left away for clarity.

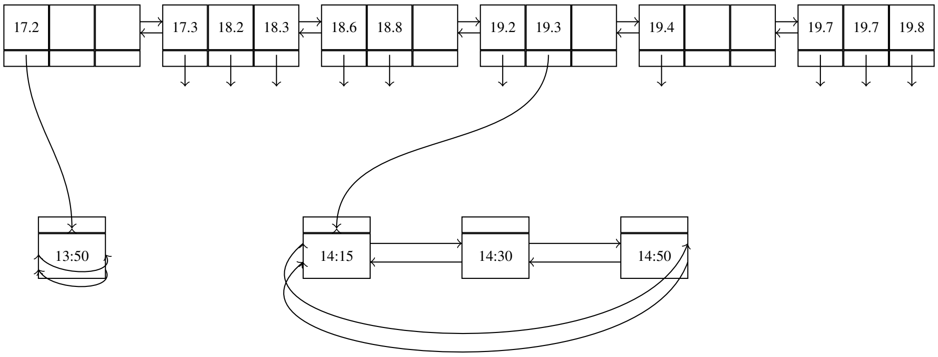


Figure 4.5: Doubly, circular linked list associated to a leaf node

As described above, the leaf nodes in our B^+tree also have pointers, namely pointers to the associated linked list. But the number of pointers in leaf nodes is always equal to the number of searchkeys in the leaf. A pointer at position i points to the doubly, linked list associated with the leaf key at position i.

```
struct {
  timeStamp timestamp;
  struct ListValue *prev, *next;
} ListValue;
```

A linked list element consist of the time point and the pointers to the predecessor $prev$ and to the successor $next$.

The insertion and the deletion of a list value from a linked list which contains multiple values is illustrated in Algorithm 1 and Algorithm 2, respectively.

```
Algorithm 1: AddNewTail($(node, i, t)$)
Data: Leaf $node$, the index position $i$ to the linked list $L$ and the time point to insert $t$
Result: Linked List $L$ such that $t \in L$
begin
    head ← node.pointers[i]
    tail ← head.prev
    insert $t$ between head and tail
    doubly link $t$ to head and tail
end
```

```
Algorithm 2: DeleteHead($(node, i)$)
Data: Leaf $node$ and index position $i$ for the position of the associated Linked List $L$
Result: Linked List $L$ such that $t \notin L$
begin
    elementToDelete ← node.pointers[i]
    nextElement ← elementToDelete.next
    prevElement ← elementToDelete.prev
    //next is the second oldest key
    leaf.pointers[i] ← next
    prevElement.next ← nextElement
    prevElement.prev ← prevElement
end
```

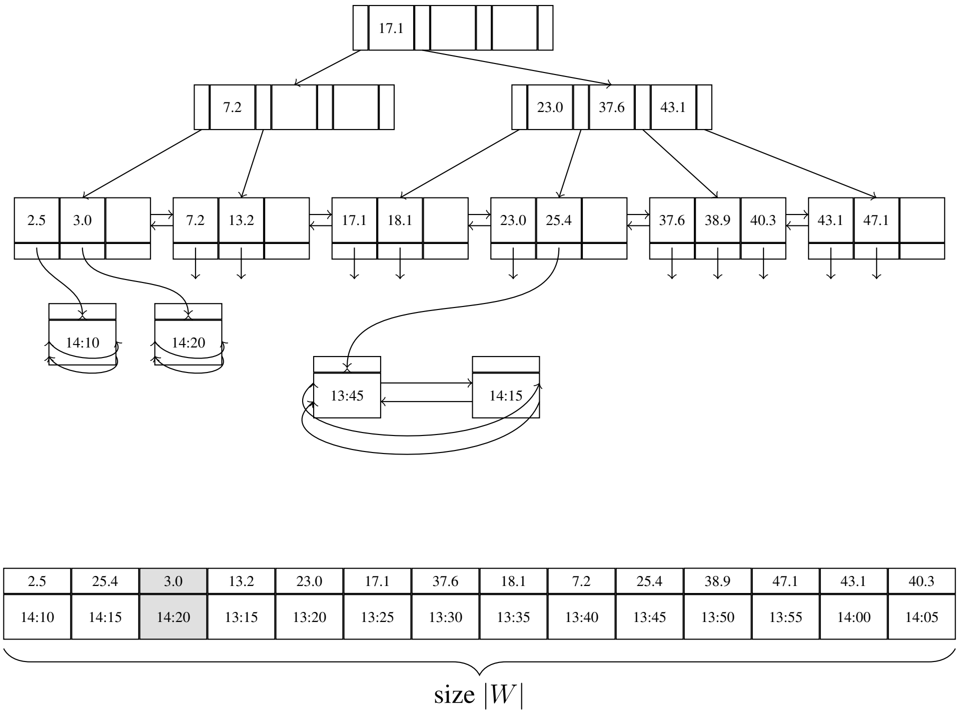
Observation

The linked list associated to the measurement key is used for storing the measurement time point. Every new time point that is added to an existing linked list always has a newer time point than all other time points in the list. Consequently, a new time point is always inserted at the tail position. Therefore, the linked list is always sorted by the time points from head to tail. A $shift$ operation on the circular array leads to a deletion of the oldest measurement in time series s, thus, the measurements time point, as explained, is always at the head position in a linked list. 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 tail position.
The new_neighborhood(t, v, j, l)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. In the worst case the entire linked list would be searched for the specific time point. We initialize $l \times d$ neighborhoods, one for each value in a query pattern $Q(t)$. Thus, l neighborhoods in every time series r in the pattern. Hence, the new_neighborhood(t, v, j, l) operation always is executed at the l newest measurements in a time series. Thus, we can give an upper bound, namely the pattern length l. Therefore, the worst case depends on the pattern length l.

4.4 Operations

The data in the circular array is updated with every new arriving measurement. Therefore, with every $shift$ execution the array is updated. The update method is illustrated in Algrithm 3. The $shift$ operation not only influences the array data but also the data in the B^+tree. Therefore, the deletion and insertion of a measurement in the tree is executed within the update of the circular array. The implementation of an insertion and deletion within a B^+tree is described in Section 4.4. The $lookup$ of a value is presented in Algorithm 13.
Further, the neighbor(v, T) method uses the B^+tree returning the time point $t \in T$ such that $|v - s(t)|$ is minimal, given a value *v*.

Example 4 *We assume we have the situation illustrated in Figure 4.6. 25.4 occurs two times, therefore two list values are part of the circular linked list associated to the key. Further, all keys in the leaves have an associated time point. Some associated linked lists are not illustrated to improve clarity. The size of the circular array is 14 and it is already full. This significances that for every new arriving measurement a value has to be added to and another one has to be deleted from the B^+tree.*



2.5	25.4	3.0	13.2	23.0	17.1	37.6	18.1	7.2	25.4	38.9	47.1	43.1	40.3
14:10	14:15	14:20	13:15	13:20	13:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05

Figure 4.6: Start situation

4.4.1 Shift$(\bar{t}, v)$

Add a Value to the Circular Array

The circular array has a $count$ attribute that is equal or smaller than the size of the array. It counts the number of measurements in the time series s. If the $count$ is equal to the size of the array one measurement has to be deleted for every new arriving measurement. If not, there is no need to delete a value from the B^+tree, since no value is overwritten in the circular array.

Algorithm 3: Update the Circular Array

```
Data: Tree $tree$, the circular array $array$, the new time point $t$ and the new value $v$
Result: The updated array
begin
    newPos ← 0
    if array.count < array.size then
        //the array has no value yet
        if array.count ≠ 0 then
            newPos ← (array.lastUpdatePosition + 1) %array.size
        end
        array.count++
    end
    else
        newUpdatePosition ← (array.lastUpdatePosition + 1) %array.size
        //delete measurement from tree
        delete(tree, array.data[newPos].time, array.data[newPos].value)
    end
    array.data[newPos].time ← newTime
    array.data[newPos].value ← newValue
    array.lastUpdatePosition ← newPos
    addRecordToTree(tree, newTime, newValue)
end
```

Example 4.4.1 *We assume that a new measurement arrives with time point 14:25 and key* 41.5*. The value* 13.2 *at time point 13:15 is overwritten by the new measurement. The new circular array looks as follows:*

2.5	25.4	3.0	41.5	23.0	17.1	37.6	18.1	7.2	25.4	38.9	47.1	43.1	40.3
14:10	14:15	14:20	13:15	13:20	13:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05

Figure 4.7: Circular Array before

2.5	25.4	3.0	13.2	23.0	17.1	37.6	18.1	7.2	25.4	38.9	47.1	43.1	40.3
14:10	14:15	14:20	14:25	13:20	13:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05

Figure 4.8: Circular Array after the insertion of 41.5

Searching in the B^+tree

Before we can delete or add a measurement to the B^+tree, we have to find the right leaf. Algorithm $FindLeaf$ presents the pseudo-code to find the appropriate leaf.

Algorithm 4: FindLeaf

```
Data: Tree $tree$ and the searchkey $k$
Result: The appropriate leaf for the searchkey $k$
begin
    curNode ← tree.root
    if curNode = NIL then
        return curNode
    end
    while curNode is no Leaf do
        Let $i$ ← smallest number such that $k \leq$ curNode.$K_i$
        if no $i$ then
            $m$ ← last non-null pointer in the node
            curNode ← curNode.$P_m$
        else
            curNode ← curNode.$P_i$
        end
    end
    return curNode
end
```

Example 5 *Assume we want to find the key value* 13.2 *in the* B^+*tree, since this is the one that has been overwritten by the newly arrived measurement in Figure 4.8. The function starts at the root of the tree, and goes through the tree until it reaches a leaf node that would contain the searched value. The current node is examined by looking for the smallest* i *for which the searchkey value* 13.2 *is greater or equal to. The Figure 4.6 illustrates the* B^+*tree. In this case the first pointer comes from the root at index position* 0*, since* 13.2 *is smaller than* 17.1*. Then the new current node is set to the child node at pointer position* 0*. Then the procedure is repeated until a leaf node is reached.*

Delete a Value from the B^+tree

The circular array update Algorithm first deletes a value if necessary and then adds the new value to the tree. Since the deletion is executed first, we first present the deletion.
At the beginning, the leaf for the measurement to delete is located. Since our B^+tree accepts

duplicates, it is afterwards checked if the associated linked list to the measurement key has multiple list values. If the list has multiple list values, the identified list value is deleted and the deletion is already finished.
But if the entries time point is the single value in the linked list the key and its belonging linked list is deleted. Therefore, $DeleteEntry$ is called.
After removing the entry from the node, there are three possible options:

- The node has still enough keys: The minimum number of keys depends on the node properties. If the node is a leaf node it contains at least $\lceil (n-1)/2 \rceil$ keys. If the node is an internal node the minimum number of keys is $\lceil n/2 \rceil - 1$ and the minimum number of pointers is $\lceil n/2 \rceil$.

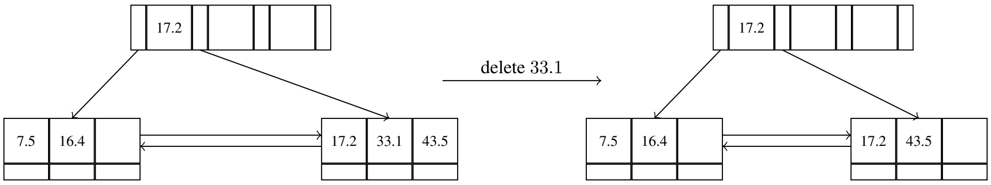


Figure 4.9: Option 1: The node has still enough keys

- The node needs to be merged with a sibling node: We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. If there is no left sibling the right sibling is selected. Once the node is deleted, we must also delete the entry in the parent node that pointed to the deleted node. Hence, we traverse the tree upwards recursively until $deleteEntry$ stops.

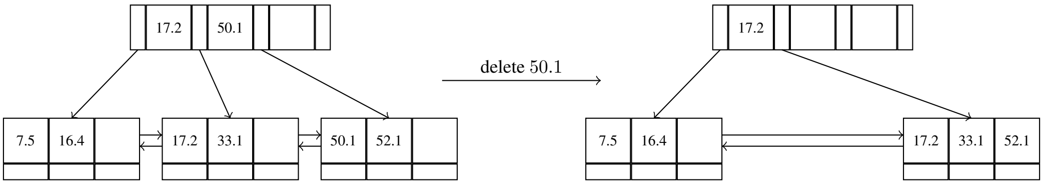


Figure 4.10: Option 2: The node can be merged with its sibling

- The values in the node have to be redistributed to ensure that each node is at least half-full and hence contains the minimum number of keys. Merging is not possible, if the sibling and node together have more than the allowed n pointers. Thus, the nodes have to be redistributed. We redistribute the keys, such that each node has at least $\lceil n/2 \rceil$ child pointers. Therefore, we move the rightmost pointer from the left sibling to the under-full right sibling. Hence, we also need to move a key so that the newly added pointer is separable. This pointer is neither present in the left sibling nor in the right sibling. So we take a key from the parent node.

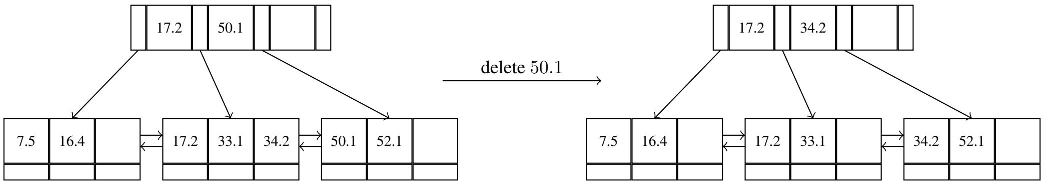


Figure 4.11: Option 3: The values are redistributed

As a result of deletion, a key value that is present in an interior node or in the root node of the $B{+}$tree may not be present at any leaf of the tree any more.

Algorithm 5: Delete$(tree, t, k)$

```
Data: Tree $tree$, the measurement with its time point $t$ and its key $k$
Result: Deletes measurement form $tree$
begin
    leaf ← findLeaf($tree$, $k$)
    i ← right position index in leaf
    if list on pointer at position i has multiple elements then
        deleteHead(leaf, keyPositionIndex)
    else
        deleteEntry(tree,leaf, key on keyPostionIndex)
    end
end
```

Example 4.4.2 *We again refer to the example in Figure 4.6. The measurement with time point 13:15 is overwritten by the new value. Therefore, it needs to be deleted from the B^+tree before the new value is added. First, the $findLeaf$ method finds the leaf where* 13.2 *is located. Then it deletes the value and its associated linked list. The leaf after has just 1 value left and therefore is smaller than the minimum allowed keys of* $\lceil(n-1)/2\rceil$. *Since* $1 < \lceil(4-3)/2\rceil$. *The left neighbor has still enough space for the only key left in the node, namely,* 7.2. *The node is merged with its left sibling. After the key in the parent is not right any more. Since* 7.2 *now is part of the left child. The key* 7.2 *in the parent is removed as well by calling* $DeleteEntry$ *at the end of* $MergeNodes$. *Then the* $DeleteEntry$ *procedure checks weather this node can be merged with its sibling. Since the node has one pointer left to its now only child and the sibling has already three keys,* $23.0, 37.6, 43.1$ *and hence 4 pointers. So* $1+4$ *is more than the allowed* 4 *pointers in an inner node. So the keys have to be redistributed. The node takes the root nodes key* 17.1 *as new key and gets the leftmost child of the sibling. This is the leaf node with the keys* 17.1 *and* 18.1. *The root node takes the leftmost key of its right children, so* 23.0. *The right children now has the keys* 37.6 *and* 43.1 *left. As a result, the tree after deleting* 13.2 *looks as follows:*

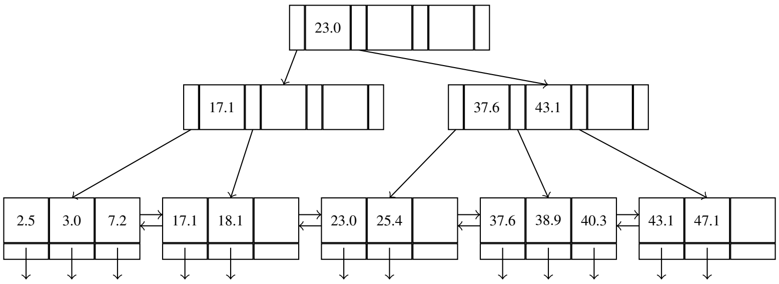


Figure 4.12: B^+tree after the deletion of 13.2

Algorithm 6: AdjustTheRoot($tree$)

```
Data: Tree $tree$
Result: The $tree$ with an adjusted root node
begin
    //enough keys in the root
    if 0 < tree.root.numOfKeys then
        return
    end
    //if the root has a child, promote the first (only) child as the new root
    if root is leaf then
        newRoot ← tree.root.pointers[0]
        newRoot.parent ← NIL
    else
        newRoot ← NIL
    end
    tree.root ← newRoot
end
```

Algorithm 7: DeleteEntry($tree, node, k, pointer$)

```
Data: Tree $tree$, the node $node$ where the deletion key belongs to, $k$ the key to delete
Result: $k \notin node$
begin
    node ← remove $k$ from $node$
    if node is the root then
        adjustTheRoot($tree$)
        return
    end
    if node is leaf then
        minNrOfKeys ← $\lceil (n-1)/2 \rceil$
    else
        minNrOfKeys ← $\lceil n/2 \rceil - 1$
    end
    if minNrOfKeys ≤ number of keys left in node then
        //node has still enough keys
        return
    end
    //node has not enough keys - merge or rearranges necessary
    neighborIndex ← get pointer position of left sibling in parent node
    if no left sibling then
        kIndex ← 0;
        neighbor ← node.parent.pointers[1]
    else
        kIndex ← neighborIndex;
        neighbor ← node.parent.pointers[neighborIndex]
    end
    //keyPrime is the value between pointers to $node$ and $neighbor$ in parent
    innerKeyPrime ← node.parent.pointers[kIndex]
    capacity ← treeNodeSize
    if node is a leaf then
        capacity ← treeNodeSize + 1
    end
    //Merge if both nodes together have enough space
    if (neighbor.numOfKeys + node.numOfKeys ) < capacity then
        mergeNodes(tree, node, neighbor, neighbourIndex, innerKeyPrime)
    else
        redestributeNodes(tree, node, neighbor, neighbourIndex, kIndex, innerKeyPrime)
    end
end
```

Algorithm 8: MergeNodes

```
Data: Tree $tree$, the node $node$ and its neighbor $neighbor$, the neighborIndex $nIndex$
      and the key $kPrime$
Result: $node$ and its $neighbor$ are merged to one node
begin
    //Swap neighbor with node if node is on the extreme left and neighbor is to its right
    if node is leftmost then
        swap neighbor with node
    end
    neighborInsertionIndex ← neighbor.numOfKeys
    //Append kPrime and the following pointers
    if node is no leaf then
        neighbor.keys[neighborInsertionIndex] ← kPrime
        neighbor.numOfKeys++
        decreasingIndex ← 0
        numOfKeysBefore ← node.numOfKeys
        for i ← neighborInsertionIndex + 1, j ← 0; j < node.numOfKeys; do
            neighbor.keys[i] ← node.keys[j]
            neighbor.pointers[i] ← node.pointers[j]
            neighbor.numOfKeys++
            decreasingIndex++
            i++, j++
        end
        node.numOfKeys ← numOfKeysBefore - decreasingIndex
        neighbor.pointers[i] ← node.pointers[j]
        //All children must now point up to the same parent
        for i ← 0; i < neighbor.numOfKeys + 1; i++ do
            tmp ← neighbor.pointers[i]
            tmp.parent ← neighbor
        end
    else
        // a leaf, append the keys and pointers of the node to the neighbor
        //Set the neighbor's last pointer to point to what had been the node's right
        neighbor
        for i ← neighborInsertionIndex, j ← 0; j < node.numOfKeys do
            neighbor.keys[i] ← node.keys[j]
            neighbor.pointers[i] = node.pointers[j]
            neighbor.numOfKeys++
            i++, j++
        end
        relink leaves
    end
    deleteEntry(tree, node.parent, kPrime, node)
end
```

Algorithm 9: Redistribute

```
Data: Tree $tree$, the node $node$ and its neighbor $neighbor$, the neighborIndex $nIndex$,
      the $kIndex$ and the the key $kPrime$
Result: The keys in the node and its neighbor, as well as the parents keys are
        redestributed
begin
    if node has neighbor to the left side then
        //Pull the neighbor's last key-pointer pair over from the neighbor's right end to n's
        if node is not a leaf then
            m ← neighbor.pointer[neighbor.numOfKeys]
            insert neighbor.pointers[m] and $kPrime$ to first position in node and shift
            other pointers and values right remove neighbor.key[m-1],
            neighbor.pointers[m] from neighbor
            replace $kPrime$ in node.parent by neighbor.keys[m-1]
        else
            //last value pointer pair in the node
            m ← neighbor.pointer[neighbor.numOfKeys -1]
            insert neighbor.pointers[m] and neighbor.keys[m] to first position in node and
            shift other pointers and values right remove neighbor.key[m],
            neighbor.pointers[m] from neighbor
            replace $kPrime$ in node.parent by node.keys[0]
        end
    else
        //node is leftmost child. Take a key-pointer pair from the neighbor to the right
        //Move the neighbor's leftmost key-pointer pair to n's rightmost position
        if node is not a leaf then
            node.keys[node.numOfKeys] ← $kPrime$
            node.pointers[node.numOfKeys +1] ← neighbor.pointers[0]
            replace $kPrime$ in node.parent by neighbor.keys[0]
            remove neighbor.keys[0], neighbor.pointers[0] from neighbor
        else
            node.keys[node.numOfKeys] ← neighbor.keys[0]
            node.pointers[node.numOfKeys +1] ← neighbor.pointers[0]
            node.parent.keys[kIndex] = neighbor.keys[1]
            remove neighbor.keys[0], neighbor.pointers[0] from neighbor
        end
    end
end
```

Insert a Value to the B^+tree

With every circular array update a new measurement is added to the B^+tree as shown in Algorithm 3 line 12.

We first find the leaf node where the key would appear by using $findLeaf()$. We then insert an entry, positioning it such that the search keys are still in order. Besides, a new doubly, circular linked list is allocated where the time point is inserted. If the searchkey already exists in the leaf node, the time point of the measurement is added to the already existing associated list and the searchkeys in the leaf remain unchanged. If the searchkey is new, it is inserted to the leaf.

The measurement is directly inserted to the leaf if there is still space left. The measurement is inserted, such that the leaf keys are still ordered from left to right.

The $splitAndInsertIntoInnerNode$ method works with the same principle as the $splitLeaves$ method. The difference is that if the inner nodes are split, one key is not included to the inner node but is given one level up to the parent of the splitted nodes. This searchkey separates the children.

Algorithm 10: AddMeasurement

```
Data: Tree $tree$, the new time point $time$ and the new value $value$
Result: The tree includes the new value $value$
begin
    //the tree does not exist yet - create tree
    if tree.root = NIL then
        newTree(tree, time, value)
        return
    end
    leaf ← findLeaf(tree, value)
    //insert to leaf as doubly linked list value
    if isDuplicateKey(leaf, time, value) then
        addDuplicateToDoublyLinkedList(leaf, time, value)
    else if leaf.numOfKeys < tree.nodeSize then
        //enough space for new key value pair
        insertRecordIntoLeaf(tree, leaf, time, value)
    else
        //leaf must be split
        splitAndInsertIntoLeaves(tree, leaf, time, value);
    end
end
```

```
Algorithm 11: SplitLeaves
Data: Tree $tree$, the node $node$ and its neighbor $neighbor$, the neighborIndex $nIndex$,
      the $kIndex$ and the the key $kPrime$
Result: The keys in the node and its neighbor, as well as the parents keys are
        redestributed
begin
    insertPoint ← 0
    int nrOfTempKeys ← 0
    insertPoint ← getInsertPoint(tree, oldNode, firstValue)
    //fills the keys and pointers
    for i = 0, j = 0; i < oldNode.numOfKeys; do
        //if value is entered in the first position: pointers needs to be moved 1 position
        if (j = insertPoint) j++
        tempKeys[j] = oldNode.keys[i]
        tempPointers[j] = oldNode.pointers[i]
        i++, j++
    end
    //enter the record to the right position
    tempKeys[insertPoint] ← firstValue
    newListValueTime ← new list value with time $time$
    tempPointers[insertPoint] ← newTime
    newNode.numOfKeys ← 0
    oldNode.numOfKeys ← 0
    //calculate splitpoint by $\lceil n/2 \rceil$
    split = getSplitPoint(tree.nodeSize)
    //fill first leaf
    for i = 0; i < split;) do
        oldNode.keys[i] ← tempKeys[i]
        oldNode.pointers[i] ← tempPointers[i]
        i++
    end
    //fill second leaf
    for j = 0, i = split; i < nrOfTempKeys; do
        newNode.keys[j] ← tempKeys[i]
        newNode.pointers[j] ← tempPointers[i]
        i++, j++
    end
    link leaves
    //the record to insert in upper node
    keyForParent
    keyForParent ← newNode.keys[0]
    insertIntoParent(tree, oldNode, keyForParent, newNode)
end
```

Algorithm 12: InsertIntoParent

```
Data: Tree $tree$, the newly created $node$ and the $oldnode$ and the key $k$ which is inserted
      to the parent
Result: The key $k$ is inserted to the parent or the parent is split
begin
    parent ← oldnode.parent
    if parent = NIL then
        insertIntoANewRoot(tree, oldnode, newKey, newChild)
        return
    end
    //Find the parents pointer from the old node
    pointerPos ← pointer position from parent to $oldnode$
    //the new key fits into the node
    if parent.numOfKeys < tree.nodeSize then
        insertIntoTheNode(parent, pointerPos, newKey, newChild)
    else
        splitAndInsertIntoInnerNode(tree, parent, pointerPos, newKey, newChild)
    end
end
```

Example 4.4.3 *We now consider the example from the beginning in Figure 4.6, where* 41.5 *has been inserted to the circular array. The measurement which already has been inserted to the circular array belongs to a leaf node which is already full. Hence, the* $InsertIntoParent()$ *method is executed and we find out that the parent is full as well. Therefore, the parent is split to a new node and the old node using the same principle as in the* $splitLeaf$ *method. Therefore, the algorithm is not shown again. After, we see that the parent of the inner node is not full yet. Also, the parent is at the same time the root node. The new new nodes leftmost key is used to insert into the root. We see that the insertion is recursive from the leaves till the root until a node with enough space is found. If the root node is already full the root node would be split as well and a new root node would be allocated. But in our case the root node just gets a new key. Its child nodes are split but the key is not inserted. The new key is just inserted to the root node. Therefore, the new tree after inserting the new measurement looks as follows:*

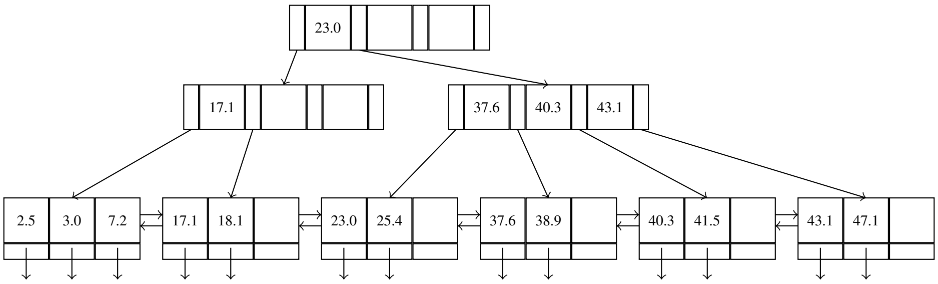


Figure 4.13: B^+tree after the insertion of 41.5

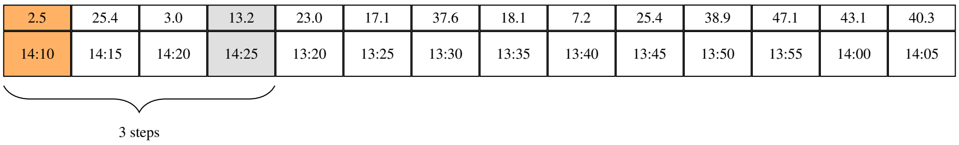
4.4.2 Random Access: Lookup a Value

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 interval between two consecutive measurements. The last update point can be used as reference time point for the calculation. The Lookup(t) is presented in Algorithm 13.

Algorithm 13: Lookup(t)

```
Data: The circular array $array$ and the time point $t$
Result: Returns the value $v$ at time point $t$
begin
    if array.count = 0 then
        //no values yet
        return NIL
    end
    step ← ($t$ - array.data[array.lastUpdatePosition].time)
    if $|step|$ < array.count then
        pos ← ((array.lastUpdatePosition + step)%array.size)
        if array.data[pos].time = $t$ then
            return array.data[pos].value
        end
    end
    return NIL
end
```

Example 4.4.4 *We assume we want to find the measurement value at time point 14:10. The last update position was at time point 14:20, thus the step is calculated as* -3 *and the position of time point 14:10 is* $0.2.5$ *is returned by the Lookup Algorithm. The circular array is illustrated in Figure 4.14.*



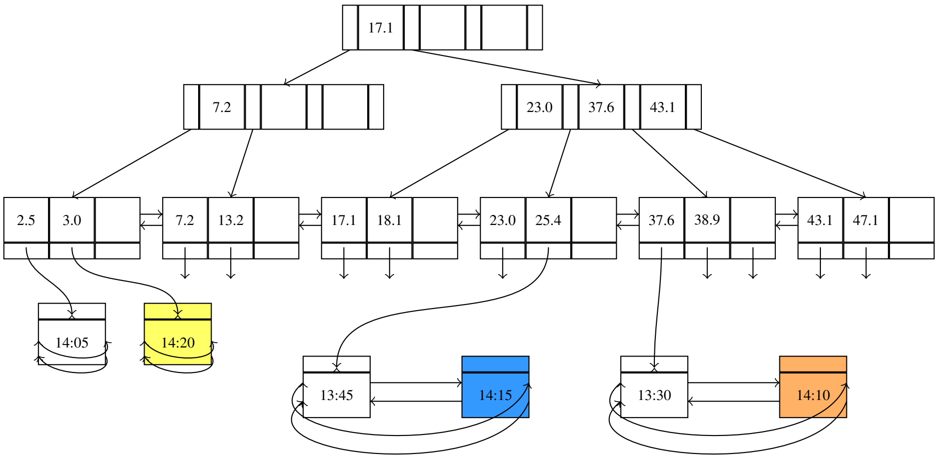
2.5	25.4	3.0	13.2	23.0	17.1	37.6	18.1	7.2	25.4	38.9	47.1	43.1	40.3
14:10	14:15	14:20	14:25	13:20	13:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05

Figure 4.14: 2.5 is the value at time point 14:10

4.4.3 Sorted Access: Neighbor

Initialize Neighborhood

```
struct {
  ListValue * timeStampPosition;
  Node * LeafPosition;
  int indexPosition;
} NeighborhoodPosition;

struct {
  int l;
  int j;
  double key;
  NeighborhoodPosition leftPosition;
  NeighborhoodPosition rightPosition;
} Neighborhood;
```



37.6	25.4	3.0	13.2	23.0	17.1	37.6	18.1	7.2	25.4	38.9	47.1	43.1	2.5
14:10	14:15	14:20	13:15	13:20	13:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05

Figure 4.15: Start situation

/* * Initializes a new neighborhood in the B+ tree. * * Parameters: * tree: The SBTree for this neighborhood * Serie: Serie that constitutes this pattern cell * patternlength: length of the query pattern * offset: position of the Serie within the query pattern * cell. offset=1 means the oldest time point in the * query pattern, offset=patternlength means the latest * time point in the query pattern. */

Algorithm 14: initializeNeighborhood

```
Data: Tree $tree$, the node $node$ and its neighbor $neighbor$, the neighborIndex $nIndex$,
      the $kIndex$ and the the key $kPrime$
Result: The keys in the node and its neighbor, as well as the parents keys are
        redestributed
begin
    neighboorhood.key ← measurement.value
    neighboorhood.offset ← offset
    neighboorhood.patternLength ← patternLength
    leafNode ← findLeaf(tree, measurement.value)
    pointerIndex ← getInsertionIndex(leafNode, measurement.value)
    listValueOnThatKey ← leafNode.pointers[pointerIndex]
    //Upper Bound: The value is at most patternLength away form first list value
    maxSteps ← patternLength
    while listValueOnThatKey.timestamp $\neq$ measurement.timestamp && maxSteps $\neq$ 0
    do
        //go from newest value back towards oldest
        listValueOnThatKey ← listValueOnThatKey.prev
        maxSteps–
    end
    neighboorhood.leftPosition ← set position to $listValueOnThatKey$
    neighboorhood.rightPosition ← set position to $listValueOnThatKey$
    return neighboorhood;
end
```

Grow Neighborhood

The *neighborhood grow* is then executed k times.
/* * Grows the neighborhood by one new value and returns its time point via the time point * parameter. The function returns true if there was a new unseen value and false otherwise * * Parameters * self: the neighborhood * timeset: a set of seen time points * time point: used as a return value, contains the time point of the * new still unseen value discovered by this function */

Algorithm 15: NeighborhoodGrow

```
Data:
Result:
begin
    leftPos ← self.leftPosition
    rightPos ← self.rightPosition
    while $t^- \neq$ NIL and TimeSetContains($t^-$ $-(j+l)$) do
        $leftPos^- \leftarrow leftPosition^-$ $-1$
    end
    while $t^- \neq$ NIL and TimeSetContains($t^-$ $-(j+l)$) do
        $rightPos^+ \leftarrow rightPosition^+$ $+1$
    end
    if $t^- \neq$ NIL and $t^+ \neq$ NIL then
        if $|r_i(t^-)$ - self.key$|$ $\leq$ $|r_i(t^+)$ - self.key$|$ then
            $leftPos^- \leftarrow leftPos^- - 1$
            $t \leftarrow t^-$
        end
        else
            $rightPos^+ \leftarrow rightPos^+ + 1$
            $t \leftarrow t^+$
        end
        else if $t^- \neq$ NIL then
            $leftPos^- \leftarrow leftPos^- - 1$
            $t \leftarrow t^-$
        else if $t^+ \neq$ NIL then
            $rightPos^+ \leftarrow rightPos^+ + 1$
            $t \leftarrow t^+$
        else
            return false
        end
    end
    return true
end
```

5 Complexity Analysis

5.0.1 Runtime Complexity

Circular Array Operations

Update $O(1)$

Lookup

B^+tree Operations

Although insertion and deletion operations on B+-trees are complicated, they are not very expensive. In the worst case for an insertion is proportional to $\log_{n\backslash 2}(|W|)$, where n is the maximum number of pointers in a node, and K is the number of keys in the leaf nodes. If there are no duplicate values the number of keys is the size of the time window $|W|$. Since in our case the insertion point of a new measurement to a doubly, linked list is always found in $O(1)$, duplicates have no influence to the complexity of an insertion.
The worst-case complexity of the deletion procedure is also proportional to $\log_{n\backslash 2}(|W|)$, even if there are duplicate values.

Neighborhood Operations

Neighborhood initialize The initialization of the neighborhood needs to search a specific measurement in the B^+tree. This is dependent on the number of values in the linked list associated to the key of the specific measurement. But the pattern length gives an upper bound to the maximum required value lookups in a doubly, linked list. In the worst-case the initialization needs to first find the appropriate leaf in $\log_{n\backslash 2}(K) + patternlength$

Neighborhood grow

5.0.2 Space Complexity

Circular Array

B^+tree

6 Evaluation

- runtime with different tree node sizes
- runtime with no duplicates
- runtime with duplicates
- increase size W
- - runtime neighborhood initialization - pattern length
- runtime neighborhood grow

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

+ wie macht man es effizient: bound erwähnen (neighborhood), leaves sortiert, neighborhood teuer. leaves sortiert ist gut, welche tree.nodesize

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

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

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein: *Introduction to Algorithms*; Massachusetts Institute of Technology, Massachusetts, USA, 2009.