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

A streaming time series is an unbounded sequence of data points that is continuously extended. The data points arrive in a predefined interval, e.g. every 5 minutes a new value arrives.
Such time series are relevant to applications in diverse domains. Imagine a meteorology station that sends a temperature measurement every 3 minutes or imagine a trader in the financial stock market who receives updated pricing information every 5 minutes.
We present an implementation of an index structure for streaming time series data. The system keeps a limited size of time series data in main memory. As a result, it is able to access the information of past measurements in time window W. We introduce an implementation using two data structures, a circular array and a B^+tree, to efficiently access the data of past measurements.

Zusammenfassung

Kontinuierliche Zeitreihen werden durch neu ankommende Daten unbegrenzt erweitert. Die Daten werden in einem vordefinierten Intervall aktualisiert.
Derartige Zeitreihen sind relevant für diverse Bereiche. Beispielsweise in der Meteorologie, in welcher die Wetterinformationen kontinuierlich aktualisiert werden oder, um einen weiteren Bereich zu nennen, in Finanzmärkten, wo die Händler auf die neusten Preisinformationen angewiesen sind.
Wir präsentieren die Implementation einer Indexstruktur für kontinuierlich erweiterte Zeitreihen. Unser System behält eine limitierte Anzahl an vergangenen Daten im Arbeitsspeicher. Daraus resultiert, dass das System auf vergangenen Daten zugreifen kann. Dazu stellen wir unsere Implementation vor, welche von zwei Datenstrukturen Gebrauch macht: einem zirkulären Array und einem B^+baum. Die beiden Datenstrukturen erlauben den effizienten Zugriff auf alle Werte der vergangenen Daten, welche sich noch im Zeitfenster befinden.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

A streaming time series is an unbounded sequence of data points that is continuously extended, potentially forever. The data points arrive in a predefined interval, e.g. every 5 minutes a new value arrives. Such time series are relevant to applications in diverse domains. Imagine a meteorology station that sends a temperature measurement every 3 minutes or imagine a trader in the financial stock market who depends on updated pricing information. Thus, various applications need to be fed continuously with the latest data.
Our system neither forgets about and deletes all past measurements nor keeps all of them, since a system can only keep a limited size of data in main memory. But because a portion of the data is kept, it is still possible to access past data. The system provides data structures and operations to efficiently access the time point and the value of a measurement.
The thesis suggests an implementation for an index structure for streaming time series data. In order to achieve efficient access to the time points and values of measurements, the system uses two data structures: A circular array in combination with a B^+tree. The circular array has a limited size $|W|$. The measurements are stored in the array, sorted by time. The B^+tree as well contains the same measurements with time points in time window W. The leaves of the tree are sorted from left to right by the measurement value. Thus, range queries can be efficiently performed. For every new measurement the time window W slides forward and hence the circular array and the B^+tree are updated. If a measurements time point is not in the sliding window W any more, it is deleted from both data structures.
The introduced system is not only implemented but also analysed in terms of space and runtime complexity. Further, an experimental evaluation tries to underpin the theoretical results.

At the beginning of this thesis, in chapter 2, the TKCM algorithm designed by Wellenzohn et al.[**?**] is introduced. In order to achieve a better understanding for the system context and for the requirements our system must satisfy. Chapter 3 describes the context and introduces the operations that our system must be able to perform. In chapter 4, our approach is represented and its advantages are discussed. Further, the pseudo-code for implementing the system is outlined. After that, the runtime and space complexity of the implementation is described in chapter 5. Followed by an experimental evaluation of the implemented system in chapter **??**. Finally, the thesis is summed up in chapter **??**.

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.[**?**] present the Top-k Case Matching algorithm (TKCM). The algorithm and its connection to this thesis 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 a 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. Such that 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 and a time point t.*

Definition 2.1.2 *Time window. Let $W = [\underline{t}, \bar{t}]$ be a sliding window of length $|W|$. Time $\underline{t}$ represents the oldest time point that fits into the time window and time $\bar{t}$ represents 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 its data gapless.
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 are

missing.
TKCM must not only recover and impute missing values, but also process the new measurements efficiently. Therefore, TKCM must provide an insertion method for new arriving values to insert them 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.
Moreover, 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.

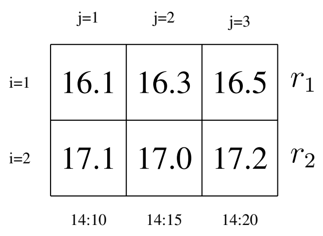
2.2 Access Methods

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 retrieves the values to fill the remaining 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.*

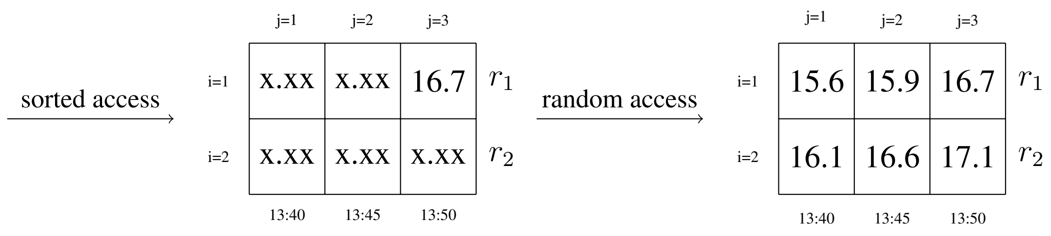
Definition 2.2.2 *Sorted Access. Sorted access returns the next yet unseen time point $t_s \notin T$ such that the value $r(t_s)$ is most similar to a given pattern cell p_{ij}. $t(s)$ is defined as:*

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



	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



sorted access $\longrightarrow$

	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	

random access $\longrightarrow$

	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 query pattern cell q_{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}. As a second step, the random access mode is used to look up the values that pattern $P(t)$ is composed of.

3 Problem Definition

The present thesis introduces an implementation of the $random$ and $sorted$ access method for a streaming time series s. The access modes are described in the previous Section 2.2. The required operations and the context for our system are presented in the following.

3.1 Context

We make the following assumptions for our system:

- The measurements arrive in a fix interval (E.g. every five minutes).
- There are no gaps between the measurements.
- There are no measurements that arrive 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 $t \notin T$ such that $|v - s(t)|$ is minimal.
- newNeighborhood(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.[**?**] 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 is executed on the leaves of a B^+tree.
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

Each time series $s \in S$ is represented by 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, respectively.

4.1 Circular Array

A circular array is used to store the time series data sorted by time and the time interval is predefined.
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 value and time are directly stored in the circular array. The last update position is stored in a variable and updated with every insertion. In detail, a circular array contains the following attributes: a counter, which counts the number of measurements in the array, a $lastUpdatePos$ to store the position that was last updated with a measurement, the size of the array which also represents the capacity of the number of measurements that the array may hold and the data, which actually holds the time point and value of all measurements.

4.1.1 States

The circular array can be in two different states:

1. First, the array is filled with every new arriving measurement until every position in the circular array is occupied.

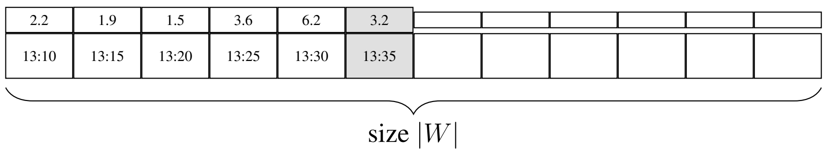


Figure 4.1: Unfilled circular array of size $|W|$

2. Afterwards, if all spaces in the circular array are occupied, the number of measurements stays constant.

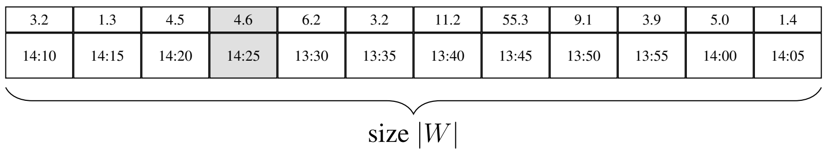


Figure 4.2: Filled circular array of size $|W|$

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 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. The lookup of a value at time point t is presented in Algorithm 16.

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.[**?**]. We introduce the most important properties of a B^+tree, for further information please refer to the book.
The differences between the traditional B^+tree and the B^+tree we use are the following: On the one hand, the leaves in our B^+tree are linked to the succeeding as well as the preceding leaf to efficiently perform the *neighbor*(v, T) operation. On the other hand, 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 used 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 pointers of nonleaf nodes point to tree nodes. An interior node must have at least $\lceil n/2 \rceil$ pointers. Hence, it must have at least $\lceil n/2 \rceil - 1$ 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 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 one 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. Further, pointers to child nodes or in leaves pointers to measurement time points. Besides, the number of keys that a node holds at the moment is stored and of course, the actual keys. Additionally, a boolean is used to represent if a node is a leaf node. A node can have two pointers to the previous and the next node. These two pointers are only used if the node is a leaf node. If not they are set to NIL. 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}.

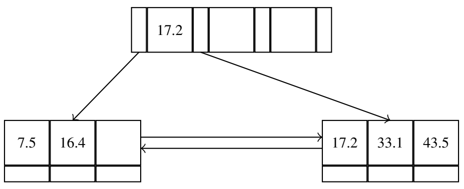


Figure 4.3: Left children keys < 17.2 and right children keys $\geq$ 17.2

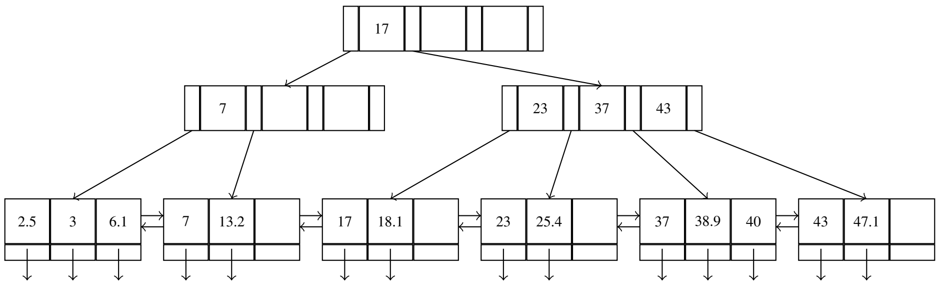


Figure 4.4: Example of a complete B^+tree

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. 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 of a given value v in a B^+tree is at the same time the right neighbor of v and the next lower value 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 exploit is that the path from the root to a leaf node always has the same length, thus, the number of nodes to traverse to find a specific value in a leaf is always the same.

4.3 Handling Duplicate Values

This Section presents our approach to handle duplicate values in our B^+tree, in regard to our requirements.

4.3.1 Associated doubly, circular Linked List

The idea is to associate a doubly, circular linked list to each key in leaf nodes. Cormen et al.[**?**] define linked lists as follows:

Definition 1 *Linked List. A linked list is a data structure arranged in linear order. The order is determined by a pointer in every linked list element. It can either be sorted in order of the keys or it can be unsorted. Given an element* e *in a linked list* L*,* $e.next$ *points to the successor element in the list. The first element, or head, of the list has no predecessor and the last element, or tail, has no predecessor. A linked list can either be singly or doubly linked. A doubly linked list element has an additional pointer which points to the predecessor, namely* $e.prev$*. Each element of a doubly, linked list 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. For clarity, we name the circular, linked list in the following only linked list. The measurement time points represent 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. As we know, the searchkey represents a value of a measurement. So instead of inserting the key again and using another position in the leaf, the new time point is inserted to the associated linked list.
Associating a doubly, circular linked list is ideal to satisfy 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 literally 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 the predecessor. Also, the Figure illustrates that a single element in a doubly, circular linked list is linked to itself. The higher levels of the B^+tree and the additional linked lists are left away for clarity.

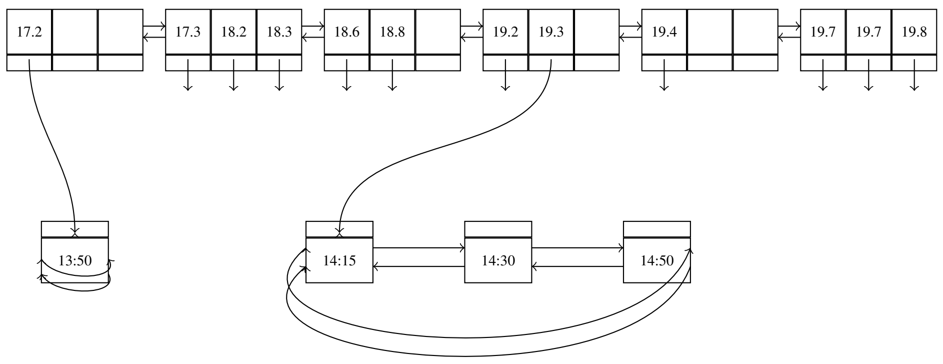


Figure 4.5: Doubly, circular linked lists associated to leaf nodes

As described above, the leaf nodes in our B^+tree also have pointers, namely pointers to the associated linked list. The number of pointers in a leaf node is always equal to the number of searchkeys in the leaf. Hence, a pointer at position i points to the doubly, linked list associated with the leaf key at position i.
A doubly, circular linked list element consists of the time point and the pointers to the predecessor $prev$ and to the successor $next$. This can be represented as follows:

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

The insertion and the deletion of a list value from a linked list that 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 $t \in L$
1 begin
2     head ← node.pointers[i]
3     tail ← head.prev
4     insert $t$ between head and tail
5     doubly link $t$ to head and tail
6 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 $t \notin L$
1 begin
2     head ← node.pointers[i]
3     nextElement ← head.next
4     prevElement ← head.prev
5     leaf.pointers[i] ← nextElement
6     prevElement.next ← nextElement
7     prevElement.prev ← prevElement
8 end
```

Observation

The linked list associated to the measurement value stores the appropriate time point. Every new time point added to an existing linked list always has the newer time point than all other time points in the same list. Consequently, a new time point is always inserted at the tail position. Therefore, the linked list is always sorted by the time 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. Also, a new measurement can be inserted without looping through the list. It is always added to the tail position. The newNeighborhood(t, v, j, l) operation searches a specific measurement. Therefore, it cannot just take the oldest or newest time point position like the insertion or deletion method. In the worst case the entire linked list must 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 which is part of the pattern. Hence, the newNeighborhood(t, v, j, l) operation always is executed at the l newest measurements in an involved time series. Thus, we can give an upper bound, namely the pattern length l. As a consequence, at most l element examinations in a linked list are necessary, where l represents the pattern length.

4.4 Operations

The data in the circular array is updated with every arriving measurement. Therefore, every $shift$ execution updates the array. The update method is illustrated in Algrithm 3. The operation not only influences the array data but also 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 the insertion and the deletion within a B^{+}tree is described in Section 4.4. The $lookup$ of a time point t is presented in Algorithm 16. Further, the neighbor(v, T) method in Algorithm 15 uses the B^{+}tree to return 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. Hence, the associated list contains two elements. Further, all leaf keys have an associated time point. Some associated linked lists are not illustrated to improve clarity. The size of the circular array is* 14*. The array is already filled. This significances that for every new arriving measurement a value has to be added to the* $B^{+}tree$ *and another one has to be deleted.*

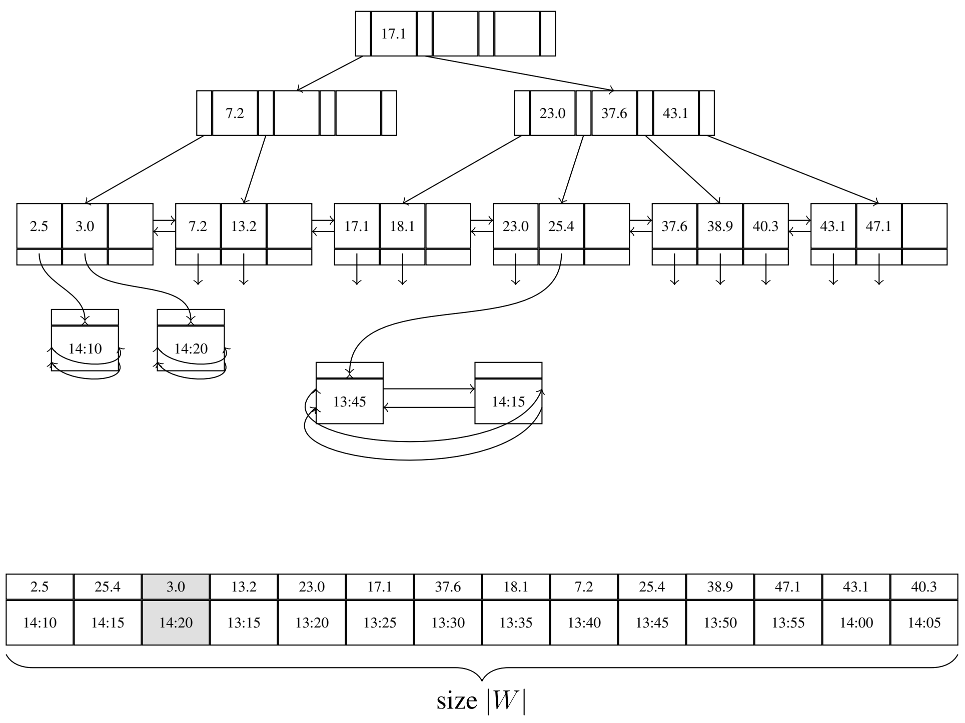


Figure 4.6: Start situation

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

Update the Circular Array

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

Algorithm 3: UpdateArray$(tree, array, t, v)$

```
Data: Tree $tree$, the circular array $array$, the new time point $t$ and the new value $v$
Result: The $array$ such that $t$,$v \in array$
begin
    newPos ← 0
    if array.count < array.size then
        //the array contains 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 ← t
    array.data[newPos].value ← v
    array.lastUpdatePosition ← newPos
    addRecordToTree(tree, t, v)
end
```

Example 4.4.1 *We assume a new measurement arrives with time point 14:25 and value* 41.5. *The value* 13.2 *at time point 13:15 is overwritten by the new measurement. This leads to the following change:*

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

Search in a B^+tree

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

Algorithm 4: FindLeaf($tree, k$)

```
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.pointers[m]
        else
            curNode ← curNode.pointers[i]
        end
    end
    return curNode
end
```

Example 5 *We assume we want to find the key* 13.2 *in the* B^+*tree illustrated in Figure 4.6 because this was the overwritten measurement value. The function starts at the root of the tree and traverses the tree until it reaches the appropriate leaf node that would contain the value. The current node is examined by looking for the smallest* i *for which the searchkey value* 13.2 *is greater or equal to. 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*. This procedure is repeated until a leaf node is reached.*

Deletion in the B^+tree

In case there has been an overwritten measurement, we first have to delete this measurment from the tree, before we can add a new one. We first present the deletion, because it is normally executed before the insertion.

First, the leaf containing the measurement to delete is located. Since our B^+tree accepts duplicates, it is afterwards checked if the associated linked list to the key has multiple list values. If the list has multiple list values, the identified list value is deleted and the deletion is already finished, as presented in Algorithm 2.
But if the entry time point is the single value in the list the key and its belonging list is deleted. Therefore, $DeleteEntry$ illustrated in Algorithm 7 is called. After removing the entry from the node, there are three possible consequences, the associated algorithms are attached to the end of this section:

1. 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 thus the minimum number of pointers is $\lceil n/2 \rceil$.

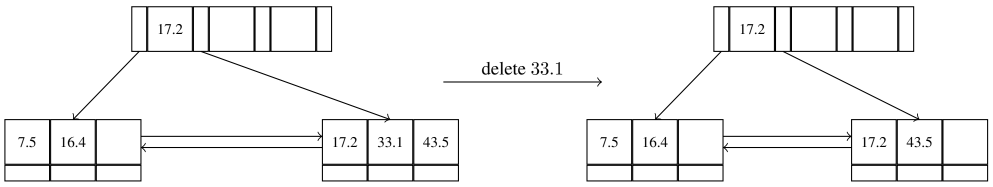


Figure 4.9: The node has still enough keys

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

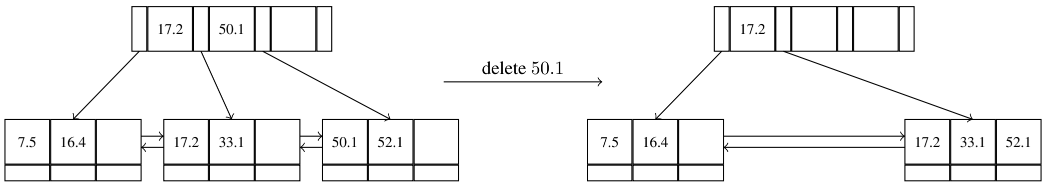


Figure 4.10: The node can be merged with its sibling

3. The values in a 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. Since inner nodes that have m keys have $m + 1$ pointers, the nodes can only be merged if the sum of all keys in both nodes is smaller than the tree nodesize $n - 1$. Leaf nodes can be merged if the sum of all keys in both leaves is smaller than n. Thus, if the keys and pointers do not fit into one

node, the keys 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. Consequently, we also need to replace the key in the parent to ensure the parent pointer to the new organised node is still separable from the other pointers.

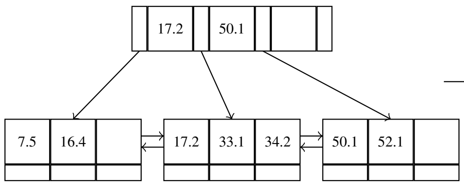


Figure 4.11: Some entries are redistributed

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. First, the* $findLeaf$ *method finds the leaf that contains* 13.2. *Then it deletes the value and its associated linked list. The leaf after has just* 1 *key 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. As a consequence, the key in the parent is not correct 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. As a consequence, the keys have to be redistributed. The node takes the root node key* 17.1 *as a new key and adds 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* 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:*

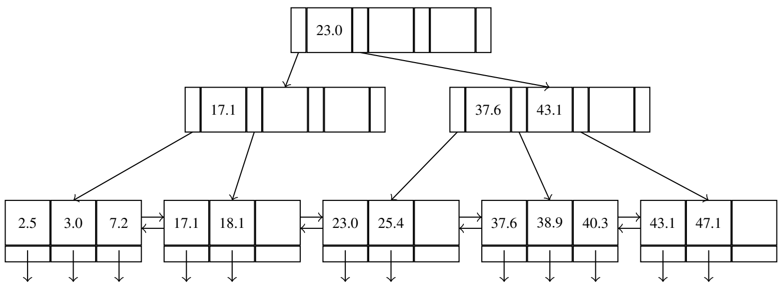


Figure 4.12: B^+tree after the deletion of 13.2

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

```
Data: Tree $tree$, the measurement $m$ time point $t$ and key $k$
Result: Tree $tree$ such that $m \notin tree$
begin
    leaf ← findLeaf($tree$, $k$)
    i ← position index in leaf such that keys remain ordered
    if list on pointer i has multiple elements then
        deleteHead(leaf, keyPositionIndex)
    else
        deleteEntry(tree, leaf, leaf.keys[i])
    end
end
```

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 not a 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 and $k$ the key to delete
Result: The node $node$ such that $k \notin node$
begin
    node ← remove $k$ and associated linked list 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 $\leq$ 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 position of left sibling in parent or -1 if no left sibling
    if neighborIndex = -1 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($tree, node, neighbor, nIndex, kPrime$)

```
Data: Tree $tree$, the $node$ 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 nIndex = -1 then
        swap neighbor with node
    end
    neighborInsertionIndex ← neighbor.numOfKeys
    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$(tree, node, neighbor, nIndex, kIndex, kPrime)$

```
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
    //node has neighbor to the left side if nIndex != -1 then
        //Pull neighbor's last key-pointer pair
        over from the neighbor's right end to n
        if node is not a leaf then
            m ← neighbor.pointers[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.pointers[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
```

Insertion in the B^+tree

To insert a value to the tree, we first find the leaf node where the key would appear by using $findLeaf$. After, we insert the entry such that the searchkeys are still in order and a 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. In this case, refer to Algorithm 1.
The measurement is directly inserted to the leaf if there is an empty space. It is inserted, such that the leaf keys are still ordered from left to right. The $splitAndInsertIntoInnerNode$ method has the same principle as the $splitLeaves$ method. But if the inner nodes are split, one key is removed from the node and is given one level upwards to the parent level. This searchkey separates the children.

Algorithm 10: AddMeasurement($tree, t, v$)

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

Algorithm 11: SplitLeaves($tree, leaf, t, v$)

```
Data: Tree $tree$, the insertion node $leaf$, the time point $t$ and the value $v$
Result: The leaf is split into two leaves
begin
    insertPoint $\leftarrow$ 0
    nrOfTempKeys $\leftarrow$ 0
    insertPoint $\leftarrow$ getInsertPoint(tree, leaf, v)
    //fills the keys and pointers
    for i $\leftarrow$ 0, j $\leftarrow$ 0; i < oldNode.numOfKeys; do
        if j = insertPoint then
            j++
        end
        tempKeys[j] $\leftarrow$ oldNode.keys[i]
        tempPointers[j] $\leftarrow$ oldNode.pointers[i]
        nrOfTempKeys++, i++, j++
    end
    //enter the record to the right position
    tempKeys[insertPoint] $\leftarrow$ v
    newList $\leftarrow$ create list and insert $t$
    tempPointers[insertPoint] $\leftarrow$ newList
    nrOfTempKeys++
    newNode.numOfKeys $\leftarrow$ 0
    oldNode.numOfKeys $\leftarrow$ 0
    //calculate splitpoint by $\lceil n/2 \rceil$
    split = getSplitPoint(tree.nodeSize)
    //fill first leaf
    for i $\leftarrow$ 0; i < split do
        oldNode.keys[i] $\leftarrow$ tempKeys[i]
        oldNode.pointers[i] $\leftarrow$ tempPointers[i]
        oldNode->numOfKeys++, i++
    end
    //fill second leaf
    for j $\leftarrow$ 0, i $\leftarrow$ split; i < nrOfTempKeys; do
        newNode.keys[j] $\leftarrow$ tempKeys[i]
        newNode.pointers[j] $\leftarrow$ tempPointers[i]
        newNode->numOfKeys++, i++, j++
    end
    link leaves
    newNode.parent $\leftarrow$ oldNode.parent
    keyForParent $\leftarrow$ newNode.keys[0]
    insertIntoParent(tree, oldNode, keyForParent, newNode)
end
```

Algorithm 12: SplitInnerNodes($tree, oldInnerNode, index, key, childNode$)

```
Data: Tree $tree$, the node $oldInnderNode$ and the child node $childNode$, the index
      $index$ and in addition the $key$
Result: The inner node is split into two nodes
begin
    nrOfTempKeys ← 0, x ← 0
    for i ← 0, j ← 0; i < oldNode.numOfKeys; do
        if j = index then
            j++
        end
        tempKeys[j] ← oldInnerNode.keys[i], nrOfTempKeys++, i++, j++
    end
    for i ← 0, j ← 0; i < oldInnerNode.numOfKeys + 1; do
        if j = index + 1 then
            j++
        end
        tempPointers[j] ← oldInnerNode.pointers[i], i++, j++
    end
    newInnerKey ← key
    tempKeys[index] ← newInnerKey, tempPointers[index + 1] ← childNode
    nrOfTempKeys++
    newInnerNode.numOfKeys ← 0, oldInnerNode.numOfKeys ← 0
    split ← getSplitPoint(tree.nodeSize + 1)
    for x < split; do
        oldInnerNode.keys[x] ← tempKeys[x]
        oldInnerNode.pointers[x] ← tempPointers[x]
        oldInnerNode.numOfKeys++, x++
    end
    oldInnerNode.pointers[x] ← tempPointers[x]
    leftMostKey ← tempKeys[x]
    newInnerNode.parent ← oldInnerNode.parent
    newInnerNode.numOfKeys ← (nrOfTempKeys - oldInnerNode.numOfKeys-1)
    for ++x, j ← 0; j < newInnerNode.numOfKeys; do
        //first key is not inserted to this node - it is inserted to upper node
        newInnerNode.pointers[j] ← tempPointers[x]
        newInnerNode.keys[j] ← tempKeys[x], j++, x++
    end
    newInnerNode.pointers[j] ← tempPointers[x]
    for i ← 0; i < newInnerNode.numOfKeys + 1; do
        childOfNewNode ← newInnerNode.pointers[i]
        childOfNewNode.parent ← newInnerNode, i++
    end
    insertIntoParent(tree, oldInnerNode, leftMostKey, newInnerNode)
end
```

Algorithm 13: InsertIntoParent($tree, oldChild, k, newChild$)

```
Data: Tree $tree$, the newly created $newChild$ and the $oldChild$ and the key $k$
Result: The key $k$ is inserted to the parent or the parent is split
1  begin
2      parent ← oldChild.parent
3      if parent = NIL then
4          insertIntoANewRoot(tree, oldChild, k, newChild)
5          return
6      end
7      pointerPos ← pointer position index from parent to $oldChild$
8      //the new key fits into the node
9      if parent.numOfKeys < tree.nodeSize then
10         insertIntoTheNode(parent, pointerPos, k, newChild)
11     else
12         splitAndInsertIntoInnerNode(tree, parent, pointerPos, k, newChild)
13     end
14 end
```

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

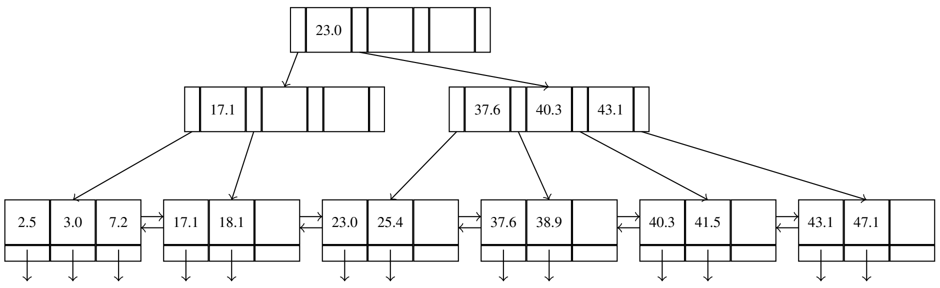


Figure 4.13: B^+tree after the insertion of 41.5

4.4.2 Sorted Access: Neighbor

Initialize Neighborhood

The Algorithm 14 initializes a new neighborhood. A neighborhood consists of the following attributes:

```
struct {
   ListValue * timeStampPos;
   Node * LeafPos;
   int indexPos;
} NeighborhoodPos;

struct {
   int l;
   int j;
   double key;
   NeighborhoodPos leftPosition;
   NeighborhoodPos rightPos;
} Neighborhood;
```

A neighborhood $N(q_{ij})$ is defined around each value q_{ij} of a query pattern $Q(t)$, where $leftPos^-$ and $rightPos^+$ are positions in leaves of the B^+tree. They represent the left and right border of the neighborhood. Initially the neighborhood positions are set to the position of q_{ij} in the B^+tree. In total $d \times l$ neighborhoods are initialized. Every time series r in query pattern $Q(t)$ initializes l patterns. The index j represents the position of the measurement within the query pattern. If $j = 1$ the oldest time point in the query pattern is meant and if $j = l$, thus j is equal to the pattern length, the latest time point in the query pattern is meant.

Example 6 *If $d = 1$ just one time series is part of pattern $P(t)$. Figure 4.14 illustrates an example of a query pattern with $d = 1$. In total l neighborhoods are initialized in time series r_1. The Figure 4.15 shows the circular array and the B^+tree for the time series r_1 after initialization.*

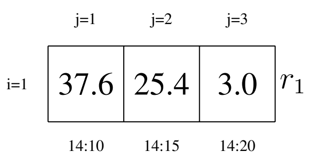


Figure 4.14: Query Pattern Q(t) of length $l = 3$ and $d = 1$ reference time series

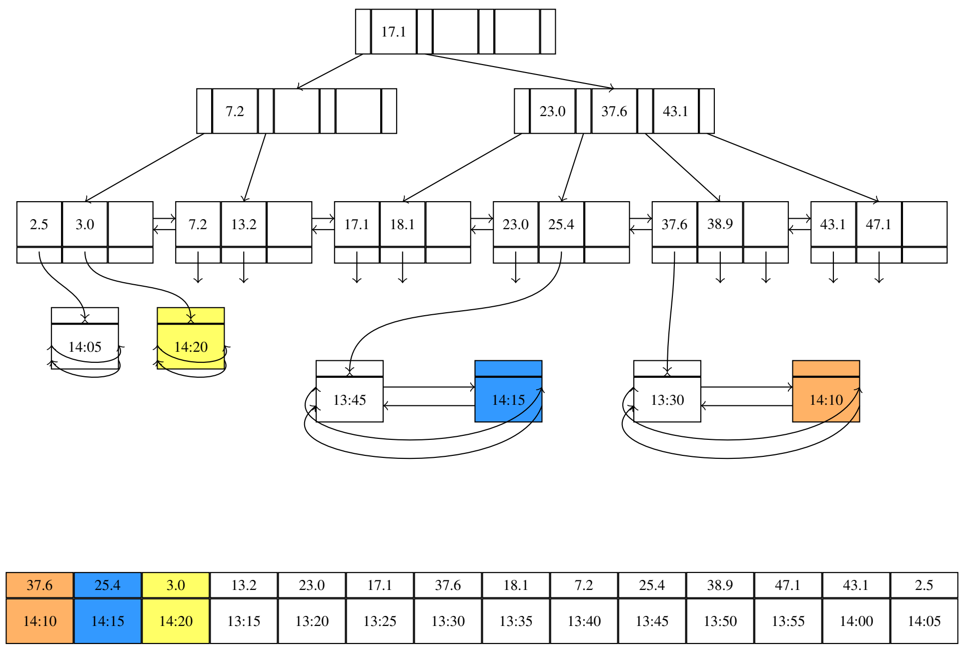


Figure 4.15: The circular array and the B^+tree for time series r_1 after initialization

Algorithm 14: NewNeighborhood(t, v, j, l)

```
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.j ← j
    neighboorhood.patternLength ← l

    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

After the initialization of the $l \times d$ neighborhoods the k non-overlapping patterns are calculated. Thus, the neighbor(v, T) operation is executed until the k patterns that contain the most similar values to q_{ij} are retrieved.
The neighbor(v, T) operation searches the time point for the next most similar value to a query pattern cell q_{ij} of the query pattern $Q(t)$. The neighborhood $N(q_{ij})$ is expanded until such a value is retrieved. Some time points may have to be skipped because the time point anchored a pattern previously. The next unseen most similar value to q_{ij} is either at position t^- or t^+, which represent the time points to the direct left or right of $leftPos^-$ and $rightPos^+$. If the t^- is more or equally similar to q_{ij} than t^+, the $leftPos^-$ is decremented and $t = t^-$. If t^+ is more similar $t = t^+$.

Example 7 *Figure 4.16 shows how the neighborhood $N(q_{12})$ is expanded. We assume that the other neighborhoods $N(q_{11})$ and $N(q_{13})$ has not been initialized yet. Hence, there are no additional time points anchored for a pattern previously and the timeset T is initially empty. If the Neighbor(v, T) is executed 3 times the resulting neighborhood will look as follows:*

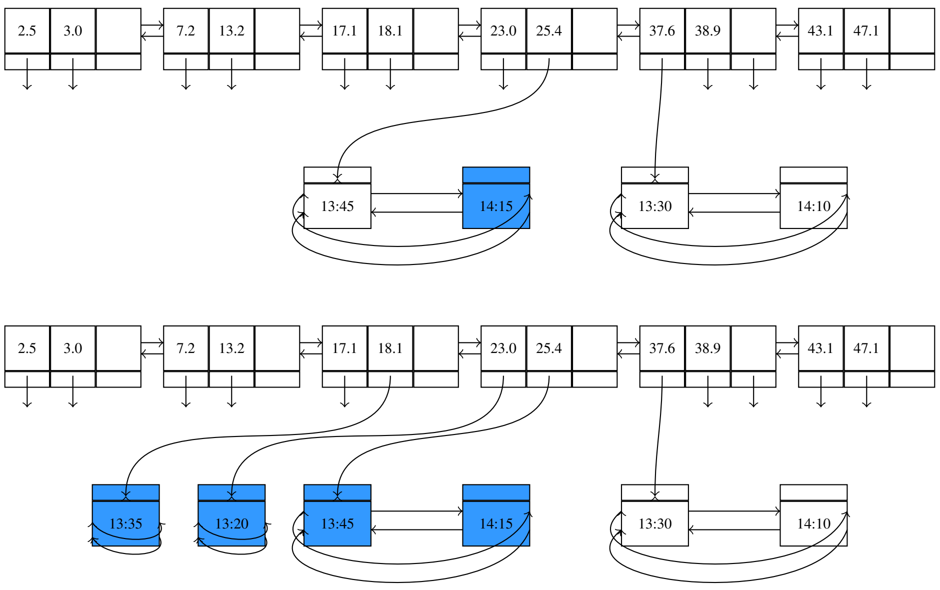


Figure 4.16: Status of $N(q_{12})$ after growing 3 times

Algorithm 15: Neighbor(v, T)

```
Data: Query pattern value $v$ and the set of visited time points $T$ and the neighborhood
      itself $self$
Result: Time point of the next most similar value to $v$
begin
    $leftPos^- \leftarrow$ self.leftPosition
    $rightPos^+ \leftarrow$ 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
            $leftPos^- \leftarrow leftPos^- - 1$
            $t \leftarrow t^-$
        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 NIL
    end
    return t
end
```

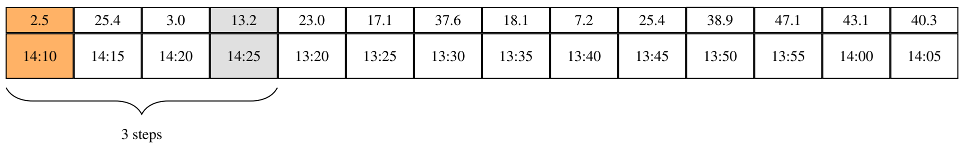
4.4.3 Random Access: Lookup a Value

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 *lookup* operation is used to fill a pattern $P(t)$ that is anchored at a time point t returned by executing $neighbor(v, T)$. The last update point can be used as reference time point for the calculation. The Lookup(t) is presented in Algorithm 16.

Algorithm 16: 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 4.17.*



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 4.17: 2.5 is the value at time point 14:10

5 Complexity Analysis

This analysis refers to a single time series s for which a B^+tree and a circular array are maintained and the complexity calculations are valid for our context, e.g. the linked list elements are traversed at most l times.

5.0.1 Runtime Complexity

Circular, Doubly Linked List

Lemma 1 *The insertion of a new time point to a linked list needs $O(1)$ time.*

Proof: For each new value produced by a time series s a linked list in the B^+tree is expanded. Since the new value is always inserted at the tail position it needs $O(1)$. □

Lemma 2 *The deletion of a time point from a linked list needs $O(1)$ time.*

Proof: For each deletion in a B^+tree a time point needs to be removed from a linked list. Since the oldest value is always at the head position, the deletion needs $O(1)$. □

Lemma 3 *The search of a specific time point in a linked list if newNeighborhood(t, v, j, l) is executed takes at most $O(l)$ time.*

Proof: A time series s in a pattern $P(t)$ initializes l neighborhoods. Hence in the worst case, if the l newest measurements in the circular array of s have the same value and thus are stored in the same linked list in the B^+tree, l list elements have to be traversed to find the specific measurement time point. □

Circular Array Operations

Lemma 4 *The update of the circular array takes $O(1)$ time.*

Proof: The next update position in a circular array is computet in $O(1)$ time, hence the update takes $O(1)$ time. □

Lemma 5 *The lookup of a time point t takes $O(1)$ time.*

Proof: The lookup algorithm needs to calculate the position for the time point t. Since the position can be directly calculated the calculation is done in $O(1)$. □

B^+tree Operations

Lemma 6 *The search of a measurement in the* B^+*tree takes at most* $O(l + (\log_n(|W|) \times n))$ *time.*

Proof: To search the insertion and deletion position in a B^+tree takes $O(\log_n(|W|) \times n)$. For the neighbor(v, T) operation the pattern length l provides an upper bound for the maximum required value lookups in a doubly, linked list.
The nodes are traversed, recursively and downwards from the root of the tree. Therefore, the problem of size $|W|$ is divided by the tree order n. This leads to $log_n(|W|)$ complexity, where $|W|$ is the maximum number of keys in the tree, at the same time the size of the circular array, and n is the order of the tree, representing the maximum number of pointers in a node.
The time taken to find the appropriate pointer index i in the $FindLeaf$ operation is at most $O(n)$. Hence, the total complexity to find a leaf key is $O((\log_n(|W|) \times n))$. □

Lemma 7 *The insertion of a measurement to the* B^+*tree takes* $O(\log_n(|W|) \times n)$ *time.*

Proof: In the worst case for an insertion is proportional to $\log_n(|W|)$, where n is the maximum number of pointers in a node. The number of keys in leaf nodes is at most $|W|$. If there are no duplicate values in the circular array. 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. □

Lemma 8 *The deletion of a measurement from the* B^+*tree takes* $O(\log_n(|W|) \times n)$ *time.*

Proof: If there are no duplicate values the number of leaf keys is the size of the time window $|W|$. Since the deletion point of a new measurement from a doubly, linked list is always found in $O(1)$, duplicates as well have no influence to the complexity of a deletion. □

Neighborhood Operations

Lemma 9 *The initialization of* l *neighborhoods in time series* s *takes* $O(l \times (l + (\log_n(|W|) \times n)))$ *time.*

Proof: The initialization of the neighborhood needs to search a specific measurement in the B^+tree. Hence the complexity is equal to the search operation in a B^+tree defined in 6. The pattern length gives an upper bound to the maximum required value lookups in a doubly, linked list. □

Lemma 10 *The neighbor*(v, T) *takes at most* $O(|T|)$ *time.*

Proof: If the time set contains $|T|$ time points and since at most $|T|$ time points have to be skipped the neighbor(v, T) takes at most at most $O(|T|)$ time. □

5.0.2 Space Complexity

Lemma 11 *The space complexity of a circular array is $O(|W|)$.*

Proof: Every circular array for a time series s has a size |W|. Hence the space complexity of one circular array is $O(|W|)$. □

Lemma 12 *The space complexity of a B^+tree is $O(|W|)$.*

Proof: Each measurement is stored once in the tree. Hence, the tree may have at most $|W|$ nodes. □

Lemma 13 *The space complexity of the neighborhoods in a time series s is at most $O(l \times |W|)$.*

Proof: In the worst case each neighborhood may span the whole sliding window of size $|W|$. Hence l neighborhoods of size $|W|$ are possible. □

6 Experimental Evaluation

This chapter describes our experimental setup and results. We evaluated our algorithms on running time and memory usage for different parameters.
Execution time
different window size different nodesize check
- runtime with different tree node sizes
- runtime with no duplicates
- runtime with duplicates
- increase size W
- - runtime neighborhood initialization - pattern length
- runtime neighborhood grow

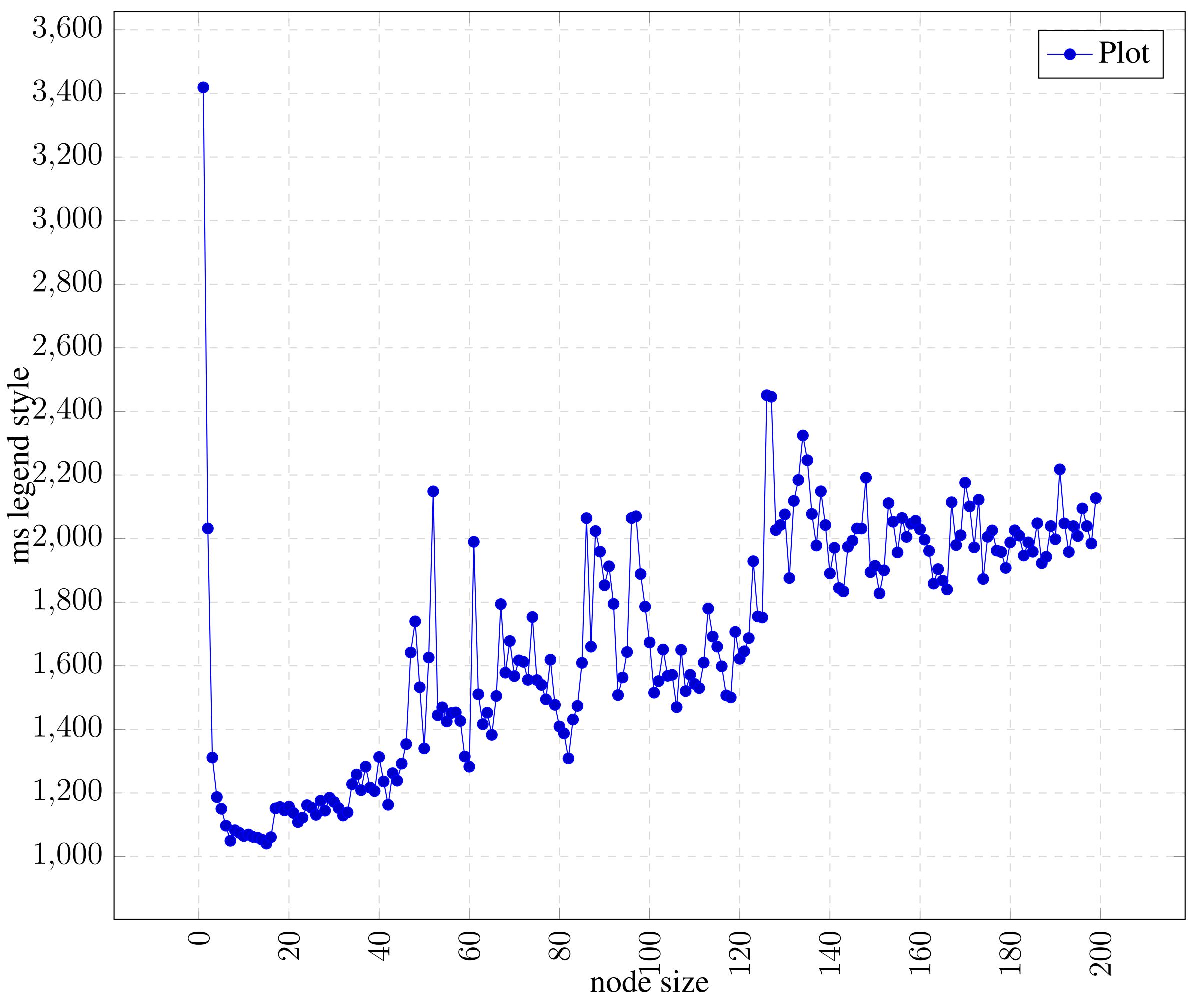
memory und runtime evaluation:
nodesize, verteilung der daten
Datenset erstellen

6.1 Setup

In the experiments we construct a data set with measurement values with a time span of 100 years between the newest and the oldest value. We use one reference time series r, thus $d = 1$. The interval between two values is set to 3 minutes. Since 20 measurements arrive per hour the data set contains in total 17'520'000 measurements in 100 years. Assuming every of these 100 years has 365 days. We assume the data set contains temperature values between -20 and 40 degrees. Hence duplicates are likely. The values are random but always between these bounds.

6.2 Runtime

Tree Node Size. The size of a node determines the number of values it can hold. We set the window size $|W|$ to 3 years for this experiment and show the effect of an encreasing treen node size.



Window Size. The window size $|W|$ determines the number of measurements stored in the circular array and the B^+tree.

6.3 Memory

7 Summary and Conclusion

We studied the requirements for keeping a portion of a streaming time series in main memory and simultaneous provide efficient access possibilities to the measurements in a time series. The system we presented uses two different data structures to achieve efficient access operations. Random access is efficiently performed on a circular array and sorted access is efficiently performed on a B^+tree with leaves linked in both directions, thus, to the respective successor and predecessor.
Furthermore, the system can handle duplicate values with a simple but powerful circular, doubly linked list. The duplicate handling is not only simple to implement but also effective in terms of update velocity. Moreover, retrieving a specific value in the linked list has an upper bound and therefore is efficiently performed in most cases.

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

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