

Department of Informatics, University of Zurich

BSc Thesis

Implementing an Index Structure for Streaming Time Series Data

Melina Mast

Matrikelnummer: 13-762-588

Email: melina.mast@uzh.ch

August, 2016

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn



University of
Zurich ^{UZH}

Department of Informatics



Acknowledgements

I especially would like to thank my supervisor Kevin Wellenzohn for his advice, his amazing support and his endurance. I could learn a lot from this thesis at the Database Technology Group of the University of Zurich. Therefore, I would like to thank Prof. Dr. Michael Böhlen for this opportunity. Further, I would like to thank my boyfriend and my family for motivating and supporting me.

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).

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 amount 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

1. Introduction	10
2. Background	11
2.1. TKCM	11
2.2. Access Methods	12
3. Problem Definition	14
3.1. Context	14
3.2. Operations	14
4. Approach	15
4.1. Circular Array	15
4.1.1. States	16
4.1.2. $\text{Shift}(\bar{t}, v)$	16
4.1.3. Random Access	17
4.2. B^+ tree	18
4.2.1. The Structure of the used B^+ tree	19
4.2.2. Handling Duplicate Values: Associated Circular Doubly Linked List .	21
4.3. B^+ tree Operations	23
4.3.1. Search in a B^+ tree	24
4.3.2. Deletion in the B^+ tree	25
4.3.3. Insertion in the B^+ tree	29
4.4. Neighborhood	31
4.4.1. Initialize Neighborhood	31
4.4.2. Sorted Access	33
5. Complexity Analysis	35
5.1. Runtime Complexity	35
5.1.1. Space Complexity	37
6. Experimental Evaluation	38
6.1. Setup	38
6.2. Runtime	38
7. Summary and Conclusion	40
Appendices	43

A. Algorithms	43
A.1. Deletion	43
A.2. Insertion	46
B. Experimental Results	49

List of Figures

2.1. Example of a query pattern $Q(t)$ of length $l = 3$ and $d = 2$ reference time series.	12
2.2. Pattern for query pattern cell $q_{1,3}$.	13
4.1. A B^+ tree with the related circular array.	15
4.2. Shifted circular array.	16
4.3. Update of a circular array.	17
4.4. 2.5 is the value at time point 14:10.	18
4.5. Left children keys < 17.2 and right children keys ≥ 17.2 .	19
4.6. Example of a complete B^+ tree.	21
4.7. Circular doubly linked lists associated to leaf nodes.	22
4.8. Start situation	24
4.9. The node has still enough keys.	26
4.10. Some entries must be redistributed.	27
4.11. The node can be merged with its sibling.	27
4.12. Deleting 13.2 from the B^+ tree.	28
4.13. B^+ tree after the insertion of 41.5.	31
4.14. Query Pattern $Q(t)$ of length $l = 3$ and $d = 1$ reference time series.	32
4.15. The circular array and the B^+ tree for time series r_1 after initialization.	32
4.16. Status of $N(q_{1,2})$ after growing 3 times.	34

List of Tables

6.1.	Shift operation with increasing values for $n - 1$.	38
6.2.	Shift operation with increasing values for $ W $.	39

List of Algorithms

1.	NextPattern	13
2.	Shift(<i>tree</i> , <i>array</i> , \bar{t} , <i>v</i>)	17
3.	RandomAccess(<i>t</i>)	18
4.	AddNewTail(<i>node</i> , <i>i</i> , <i>t</i>)	23
5.	DeleteHead(<i>node</i> , <i>i</i>)	23
6.	FindLeaf(<i>tree</i> , <i>k</i>)	25
7.	Delete(<i>tree</i> , <i>t</i> , <i>k</i>)	26
8.	DeleteEntry(<i>tree</i> , <i>node</i> , <i>k</i> , <i>pointer</i>)	29
9.	AddMeasurement(<i>tree</i> , <i>t</i> , <i>v</i>)	30
10.	NewNeighborhood(<i>t</i> , <i>v</i> , <i>j</i> , <i>l</i>)	33
11.	SortedAccess($N(q_{i,j})$, <i>T</i>)	34
12.	AdjustRoot(<i>tree</i>)	43
13.	Redistribute(<i>tree</i> , <i>node</i> , <i>neighbor</i> , <i>nIndex</i> , <i>kIndex</i> , <i>kPrime</i>)	44
14.	MergeNodes(<i>tree</i> , <i>node</i> , <i>neighbor</i> , <i>nIndex</i> , <i>kPrime</i>)	45
15.	SplitLeaves(<i>tree</i> , <i>leaf</i> , <i>t</i> , <i>v</i>)	46
16.	SplitInnerNodes(<i>tree</i> , <i>oldInnerNode</i> , <i>index</i> , <i>key</i> , <i>childNode</i>)	47
17.	InsertIntoParent(<i>tree</i> , <i>oldChild</i> , <i>k</i> , <i>newChild</i>)	48

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). 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 proposes 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. The measurements are stored in the array, sorted by time. The B^+ tree as well contains the same measurements with time points in a predefined sliding time window. 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 slides forward. A value drops out of the window and a new value is inserted to the circular array and the B^+ tree. If a measurement's time point is not in the sliding window 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.[1] is introduced. 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 6. Finally, the thesis is summed up in chapter 7.

2. Background

A streaming time series is not always gapless. Due to sensor failures or transmission errors, values can get missing. To efficiently recover missing values, Wellenzohn et al.[1] 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 monitors a set of streaming time series and imputes missing measurement values if it detects any. It 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 time series s from the k most similar past pattern. For each *time series* s a set of highly correlated *reference time series* are determined. Such that TKCM is able to recover a missing value in streaming time series data.

Definition 2.1.1 (*Time Series*) Let S be a set $S = \{s_1, s_2, \dots\}$ of streaming time series. The measurement value of time series $s \in S$ at time t is denoted as $s(t)$. For base time series s , let $R_s = \langle r_1, r_2, \dots \rangle$ 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 $r(\bar{t}) \neq \text{NIL}$. The time points in a streaming time series s are in time window W . The measurement value of a time series $s \in S$ at time point t_i is denoted as $s_i(t)$. Hence, a measurement is represented by the tuple $(t, s_i(t))$.

To be able to recover a missing measurement value, TKCM retains a sliding window of the streaming time series in main memory.

Definition 2.1.2 (*Sliding 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 most recent time point for which the stream produced a new value.

To impute a missing value, TKCM defines a query pattern $Q(\bar{t})$ over the most recent values of the reference time series. Afterwards, TKCM looks for the k most similar patterns in the sliding window.

	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.: Example of a query pattern $Q(t)$ of length $l = 3$ and $d = 2$ reference time series.

Definition 2.1.3 (Pattern) Let $R_s^d = \{r_1, \dots, r_d\}$ 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 $P(t) = [p_{i,j}]_{d \times l}$ where $p_{i,j} = S_i(t - j + l)$. $P(t)$ is a set of pattern cells $p_{i,j}$, such that $p_{i,j} \in P(t)$.

The two-dimensional query pattern $Q(\bar{t})$ is anchored at a time point \bar{t} and consists of the subsequence of length l spanning from $(\bar{t} - l + 1$ to $\bar{t})$ 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.

Only the values in the streaming time series with time points in time window W are kept in main memory. $\forall t : \underline{t} \leq t < \bar{t} \rightarrow s(t) \neq \text{NIL}$ since s contains imputed values if the real ones are missing.

TKCM initializes one neighborhood $N(q_{i,j})$ around each $q_{i,j} \in Q(\bar{t})$. The algorithm continuously looks up new candidate pattern $P(t)$ in the time window $t \in W$ by *growing* the neighborhoods $N(q_{i,j})$ until it found the k most similar patterns to $Q(\bar{t})$. It remembers the time points t in a set T to avoid considering these patterns again.

Definition 2.1.4 (Next Pattern) The next pattern for neighborhood $N(q_{i,j})$ is pattern $P(t) = [p_{i,j}]_{d \times l}$ anchored at time point $t = \underset{t \in W \setminus T}{\operatorname{argmin}} |p_{i,j} - q_{i,j}|$

TKCM must not only 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.

2.2. Access Methods

The Next Pattern (Def. 2.1.4) is built on top of two access methods. *Sorted* access is used to find the most similar value to a given query pattern cell $q_{i,j}$ (Def. 2.1.3) and *random* access retrieves the values to fill the remaining pattern cells. The two methods are defined as follows:

Definition 2.2.1 (Sorted Access) Sorted access for a neighborhood $N(q_{i,j})$ returns the next yet unseen measurement $(t, s_i(t))$ where $t = \underset{t \in W \setminus T}{\operatorname{argmin}} |s_i(t) - q_{i,j}|$.

Definition 2.2.2 (*Random Access*) Random access returns value $r(t)$, given time series r and time point t .

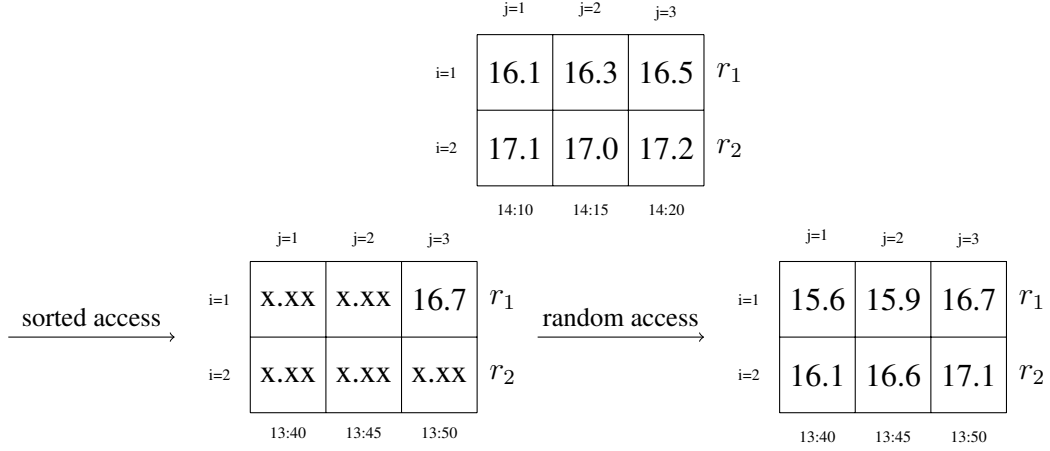


Figure 2.2.: Pattern for query pattern cell $q_{1,3}$.

The next pattern search is presented in algorithm 1. TKCM initializes a set $T = \{\}$. The set is filled during execution with all time points t for which a pattern $P(t)$ has been processed. Using the sorted access mode, the algorithm finds the next yet unseen time point $t \notin T$ for which the value is most similar to a given value in a query pattern cell $q_{i,j}$. The random access mode is used to look up the values that pattern $P(t)$ is composed of.

Algorithm 1: NextPattern

Input: Neighborhood $N(q_{i,j})$ and seen time points T

Output: Next pattern $P(t')$

- 1 $P \leftarrow$ matrix of size $d \times l$
 - 2 $(t, s_i(t)) \leftarrow \text{sortedAccess}(N(q_{i,j}), T)$
 - 3 $P[i,j] \leftarrow s_i(t)$
 - 4 **foreach** $q_{x,y} \in Q(\bar{t}) \setminus q_{i,j}$ **do**
 - 5 $P[x,y] \leftarrow \text{randomAccess}(s_x, t - y + j)$
 - 6 **end**
 - 7 **return** P anchored at time $t - j + l$
-

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 fixed interval (E.g. every five minutes).
- There are no gaps between the measurements since the gaps are filled before values are inserted in the data structure.
- 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 :

- $\text{shift}(\bar{t}, v)$: 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 .
- $\text{sortedAccess}(v, T)$: 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 (Def. 2.2.1).
- $\text{randomAccess}(t)$: return the value of time series s at time t , denoted by $s(t)$ (Def. 2.2.2).
- $\text{newNeighborhood}(t, v, j, l)$: 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.[1] suggest a combination of two data structures: a B^+ tree and a circular array. The random access operation can be performed by the circular array, while the sorted access 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. Figure 4.1 illustrates the data structures. All measurements in the circular array are represented in the B^+ tree as well.

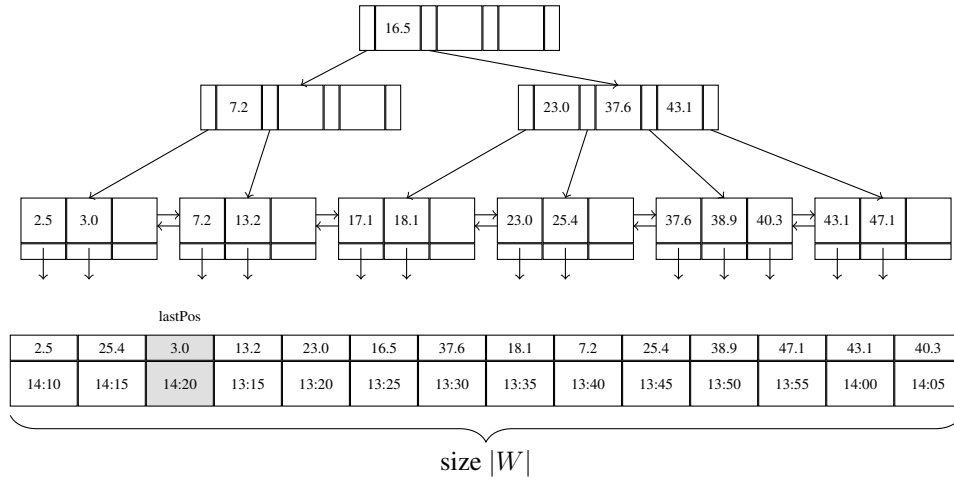


Figure 4.1.: A B^+ tree with the related circular array.

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 value and time of a measurement is directly stored in the circular array. The size of the array is determined by $|W|$ and represents the capacity of the number of measurements that the array may hold. The last update position $lastPos$ 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 $lastPos$ to store the position that was last updated with a measurement and the data, which actually holds the time points and values of all measurements.

```
struct Measurement{
    timeStamp time;
```

```

    double value;
};

struct CircularArray{
    Measurement * data;
    int lastPos;
    int count;
};

```

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 as illustrated in Figure 4.2, array 1.
2. Afterwards, if all spaces in the circular array are occupied, the number of measurements stays constant because a value drops out if a new value is inserted in Figure 4.2, array 2.

1.

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

2.

lastPos											
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.: Shifted circular array.

Example 1 *The value at time point 14:25 in array 2 in Figure 4.2 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.*

4.1.2. Shift(\bar{t}, v)

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. The circular array has a *count* attribute that is equal or smaller to $|W|$, which represents the number of measurements in the array. If the *count* is equal to $|W|$ one measurement has to be deleted for every arriving

measurement. Otherwise, there is no need to delete a value from the tree since no value dropped out of the sliding window W . The insertion of a new measurement is presented in Algorithm 2.

Algorithm 2: Shift($tree, array, \bar{t}, v$)

Input: Tree $tree$, the circular array $array$, the new time point \bar{t} and the new value v
Output: The $array$ such that $(\bar{t}, v) \in array$

```

1 newPos  $\leftarrow$  0
2 if array.count < |W| then
3   if array.count  $\neq$  0 then
4     newPos  $\leftarrow$  (array.lastPos + 1) % |W|
5   end
6   array.count++
7 else
8   newPos  $\leftarrow$  (array.lastPos + 1) % |W|
9   //delete measurement from tree
10  Delete(tree, array.data[newPos].time, array.data[newPos].value)
11 end
12 array.data[newPos].time  $\leftarrow$   $\bar{t}$ 
13 array.data[newPos].value  $\leftarrow$   $v$ 
14 array.lastPos  $\leftarrow$  newPos
15 AddMeasurement(tree,  $\bar{t}$ ,  $v$ )

```

Example 4.1.1 We assume a new measurement arrives with time point 14:25 and value 13.2. The value 41.5 at time point 13:15 is replaced by the new measurement in the circular array. Figure 4.3 show the circular array before and after the update.

lastPos													
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

lastPos													
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 4.3.: Update of a circular array.

4.1.3. Random Access

Due to the properties of a circular array the random access of a value at time t is very efficient. Since the position can be directly calculated without looping through the array, since the values arrive in a fixed interval (e.g. every 3 minutes). The *RandomAccess* operation is used to fill

a pattern $P(t)$ that is anchored at a time point t . The last update point can be used as reference time point for the calculation. The $\text{RandomAccess}(t)$ is presented in Algorithm 3.

Algorithm 3: $\text{RandomAccess}(t)$

Input: The circular array *array* and the time point t

Output: Returns the value v at time point t

```

1 if array.count = 0 then
2   //no values yet
3   return NIL
4 end
5 step  $\leftarrow (t - \text{array.data}[\text{array.lastPos}].\text{time})$ 
6 if |step| < array.count then
7   pos  $\leftarrow (\text{array.lastPos} + \text{step}) \% |W|$ 
8   if array.data[pos].time =  $t$  then
9     return array.data[pos].value
10  end
11 end
12 return NIL

```

Example 4.1.2 We want to find the measurement value at time point 14:10 in the circular array shown in Figure 4.4. The last update position was at time point 14:25, thus the step is calculated as -3 and the value at position 0 with time point 14:10 is 2.5. The value is returned by the *RandomAccess Algorithm*.

lastPos													
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 4.4.: 2.5 is the value at time point 14:10.

4.2. B^+ tree

A B^+ tree is organized in nodes which include searchkeys. The searchkey values in every node are kept in sorted order. A B^+ tree is a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree. The used B^+ tree and the implementation is based on the book of Silberschatz et al.[2]. A B^+ tree is able to execute range queries very efficiently, since the leaves of a B^+ tree are ordered from left to right and the leaf nodes are linked. Although insertion and deletion operations on B^+ trees are complicated, they require relatively few I/O operations. Therefore, the use of B^+ trees is popular in data base systems

since I/O operations are expensive. The speed of operations on B^+ trees makes it a frequently used index structure in database implementations.

The B^+ tree is a useful data structure to efficiently perform the $\text{sortedAccess}(v, T)$ operation described in Section 3.2. Because the B^+ tree we use has leaves linked in both directions. The Section 4.2.1 presents the structure of the B^+ tree for our implementation.

4.2.1. The Structure of the used B^+ tree

We introduce the most important properties of a B^+ tree, for further information please refer to the book [2].

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 $\text{sortedAccess}(v, T)$ operation. On the other hand, our B^+ tree is able to handle duplicate values. How our tree handles duplicate values is described in Section 4.2.2. The other properties of our used B^+ tree are presented in the following.

The parameter n determines the maximum number of pointers in a node and $n - 1$ represents the maximum number of searchkeys, hence the size of a tree node. The keys in a node are always sorted from left to right.

Example 2 If n is set to 7, an internal node may have between $\lceil 7/2 \rceil = 4$ and 7 children and between 3 and 6 keys. Because $\lceil n/2 \rceil - 1 = 3$ for $n = 7$. 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 m non-null pointers ($m \leq n$). For $i = 2, 3, \dots, m - 1$, 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} .

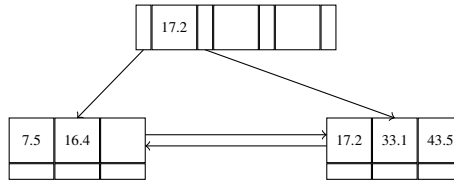


Figure 4.5.: Left children keys < 17.2 and right children keys ≥ 17.2 .

Example 3 Pointer P_0 in Figure 4.5 points to the left subtree, where all keys are smaller than 17.2 and pointer P_1 points to the subtree, where all keys greater than or equal to the root key 17.2.

There are three types of nodes that may exist in a B^+ tree: the root, interior nodes and leave nodes.

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

- (Inner Node) The inner nodes can have at most $n - 1$ searchkeys and n pointers, pointing to its child nodes. An inner node must have at least $\lceil n/2 \rceil$ pointers and can hold at most n pointers. Hence, it must have at least $\lceil n/2 \rceil - 1$ keys and at most $n - 1$ keys.
- (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.

A node contains the following attributes:

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

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 otherwise they are set to *NIL*.

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.

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.

Example 4 Assume we want to know the most similar value to the key 40 in the B^+ tree illustrated in Figure 4.6. 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.

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

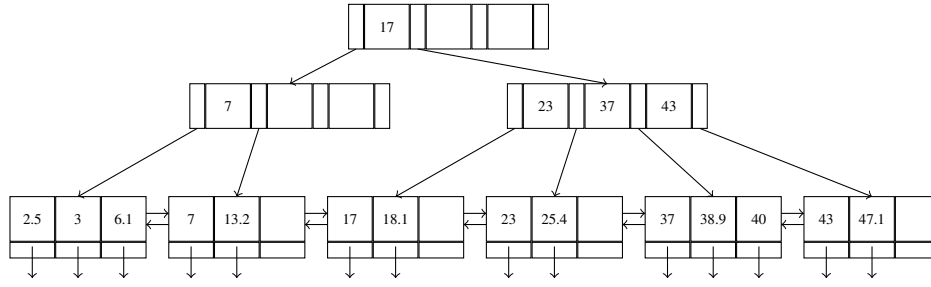


Figure 4.6.: Example of a complete B^+ tree.

the values are not unique and since the values are used as searchkeys, the B^+ tree must be able to handle possible duplicates. Section 4.2.2 proposes an approach that allows to use duplicate values in a B^+ tree and it explains how the time points of measurements are stored in the tree.

4.2.2. Handling Duplicate Values: Associated Circular Doubly Linked List

This Section presents our approach to handle duplicate values in our B^+ tree, in regard to our requirements. The idea is to associate a doubly, circular linked list to each key in leaf nodes. Cormen et al.[3] define three types of linked lists.

1. (Singly Linked List) A singly linked list can either be sorted in order of the keys or it can be unsorted. The order is determined by a pointer in every linked list element e . $e.next$ 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.
2. (Doubly Linked List) 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*.
3. (Circular Doubly Linked List) In a circular doubly linked list the *prev* pointer of the head of the list points to the tail and the *next* pointer of the tail points to the *head*. If the element e is the only element in the 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 doubly 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 the same measurement value is added multiple times to the time series s , the time points of the measurements are added to the linked list associated to the searchkey. 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.

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 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*.

Example 5 Figure 4.7 illustrates the leaf level of a B^+ tree and 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 linked list is linked to itself. The higher levels of the B^+ tree and the additional linked lists are left away for clarity.

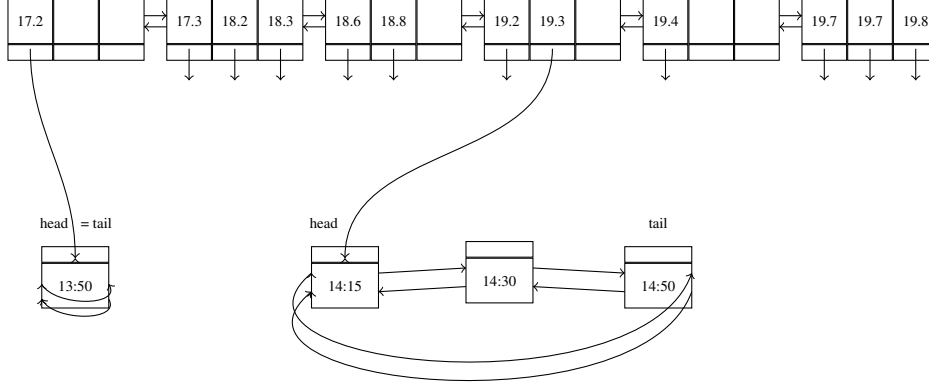


Figure 4.7.: Circular doubly linked lists associated to leaf nodes.

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 linked list associated with the leaf key at position i .

A 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 Element{
    timeStamp timestamp;
    struct Element *prev , *next;
};
```

Observation

The linked list associated to the measurement value stores the appropriate time point. The newest measurement with time point \bar{t} that is added to an existing linked list always has a newer time point compared to all other time points in the same list and also compared to the other measurements in the tree. Because $\forall t : \underline{t} \leq t < \bar{t}$ holds for all time points in the tree. 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 insertion and the deletion of a list value from a linked list that contains multiple values is illustrated in Algorithm 4 and Algorithm 5, respectively.

Algorithm 4: AddNewTail($node, i, t$)

Input: Leaf $node$, the index position i to the linked list L and the time point to insert t

Output: Linked List L such that $t \in L$

- 1 head \leftarrow node.pointers[i]
 - 2 tail \leftarrow head.prev
 - 3 insert t between head and tail
 - 4 doubly link t to head and tail
-

Algorithm 5: DeleteHead($node, i$)

1 ^a**Input:** Leaf $node$ and index position i for the position of the associated Linked List L

Output: Linked List L such that $t \notin L$

- 2 head \leftarrow node.pointers[i]
 - 3 nextElement \leftarrow head.next
 - 4 prevElement \leftarrow head.prev
 - 5 leaf.pointers[i] \leftarrow nextElement
 - 6 prevElement.next \leftarrow nextElement
 - 7 prevElement.prev \leftarrow prevElement
-

^aAlgorithm 5 is only used for linked lists that contain multiple elements.

4.3. B^+ tree Operations

The data in the circular array is updated with every arriving measurement. Therefore, every *shift* execution updates the array. 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 deletion and the insertion in a B^+ tree is described in Section 4.3.2 and Section 4.3.3, respectively.

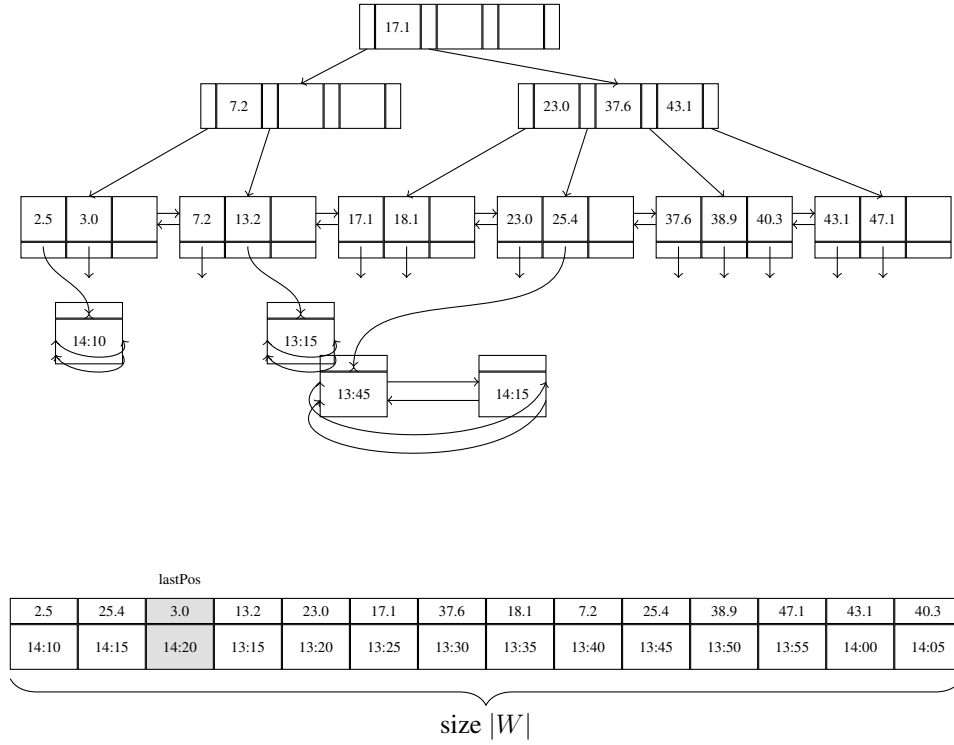


Figure 4.8.: Start situation

Example 6 We assume we have the situation illustrated in Figure 4.8. Value 25.4 occurs two times. Hence, the associated list contains two elements. Further, all leaf keys have an associated time point, but some linked lists are not illustrated to improve clarity. The size of window W and the circular array is $|W| = 14$. The array is already full, hence for every new arriving measurement a value has to be added to the circular array and the B^+ tree and another one has to be deleted from both data structures.

4.3.1. Search in a B^+ tree

Before we can delete or add a measurement, we have to find the right leaf. Algorithm 6 presents the pseudo-code to find the appropriate leaf. First, the method 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 k is greater or equal to. The current node is updated to the child node at pointer P_i . This procedure is repeated until a leaf node is reached.

Algorithm 6: FindLeaf($tree, k$)

Input: Tree $tree$ and the searchkey k
Output: The appropriate leaf for the searchkey k

```
1 curNode  $\leftarrow$  tree.root
2 if curNode = NIL then
3   | return NIL
4 end
5 while curNode is no leaf do
6   | Let  $i \leftarrow$  smallest number such that  $k \leq \text{curNode}.K_i$ 
7   | if no such  $i$  exists then
8   |   |  $m \leftarrow$  last non-null pointer in the node
9   |   | curNode  $\leftarrow$  curNode.pointers[m]
10  | else
11  |   | curNode  $\leftarrow$  curNode.pointers[i]
12  | end
13 end
14 return curNode
```

Example 7 We assume we want to find the key 13.2 in the B^+ tree illustrated in Figure 4.8 because this was the value that dropped out of the sliding window W .

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 which includes the searchkey 7.2. Afterwards, the current node is updated again to the node at pointer position $i = 1$, hence P_1 . Since the new current node is a leaf node, the algorithm returns the leaf node.

4.3.2. Deletion in the B^+ tree

In case there has been a measurement value that dropped out of the sliding window W , we first have to delete this measurement 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 with Algorithm 6. Since our B^+ tree accepts duplicates, it is afterwards checked as illustrated in Algorithm 7 if the associated linked list to the key has multiple linked list elements. If the linked list has multiple elements, the identified element is deleted and the deletion is already finished, as presented in Algorithm 5. If the entry time point is the single value in the linked list the leaf key and its belonging list is deleted.

Algorithm 7: Delete($tree, t, k$)

Input: Tree $tree$, the measurement time point t and key k

Output: Tree $tree$ such that $t, k \notin tree$

```
1 leaf  $\leftarrow$  findLeaf( $tree, k$ )
2  $i \leftarrow$  position index in leaf such that keys remain ordered
3 if list on pointer  $i$  has multiple elements then
4   | DeleteHead(leaf,  $i$ )
5 else
6   | DeleteEntry( $tree$ , leaf, leaf.keys[ $i$ ])
7 end
```

The *DeleteEntry* method illustrated in Algorithm 8 is called if a measurement is the only linked list element. A node has to be at least half-full to remain in the same state after a deletion. In detail, 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 $\lceil n/2 \rceil$.

Hence, all nodes have to be half-full after is removed from a node. There are three possible cases to achieve a state where the B^+ tree properties described in Section 4.2.1 are still satisfied. Hence, the tree must be still balanced and all nodes must be at least half-full after a deletion.

1. The node is at least half-full: The node has still enough keys. Hence, the measurement is deleted and afterwards the algorithm stops. Figure 4.9 illustrates the case where the leaf has still enough keys.

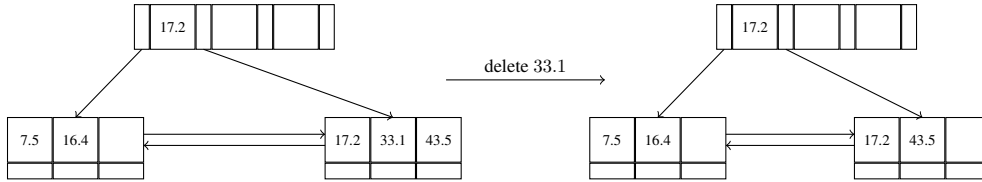


Figure 4.9.: The node has still enough keys.

2. The values in a node are redistributed: Provided that, the node is not half-full any more and the node cannot be merged with a sibling, the nodes 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 node size $n-1$. 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 interior node has at least $\lceil n/2 \rceil - 1$ keys and

each leaf node has at least $\lceil n - 1/2 \rceil$. 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 4.10 illustrates the case where some keys have to be redistributed. The Algorithm 13 for the redistribution can be found in Appendix Section A.1.

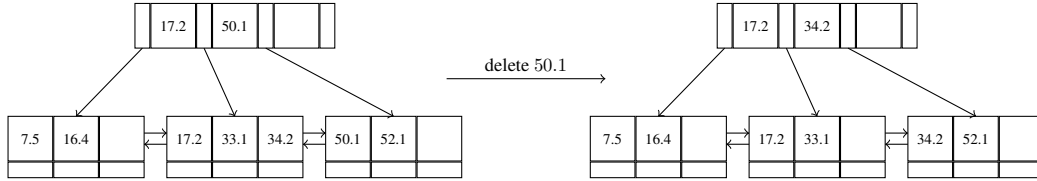


Figure 4.10.: Some entries must be redistributed.

3. The node is merged with a sibling: Provided that, the keys and pointers of the node and its sibling fit into a single node, we merge them. We move 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 4.11 illustrates the case where the node can be merged with a sibling. The Algorithm 14 for the redistribution can be found in Appendix Section A.1.

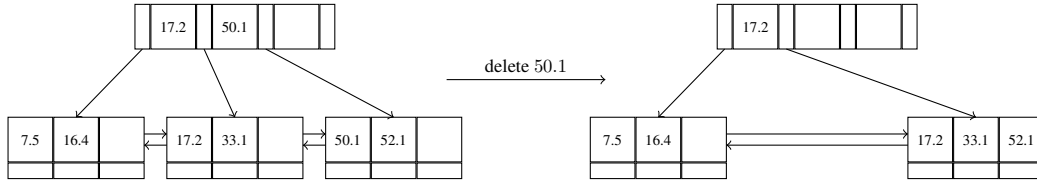


Figure 4.11.: The node can be merged with its sibling.

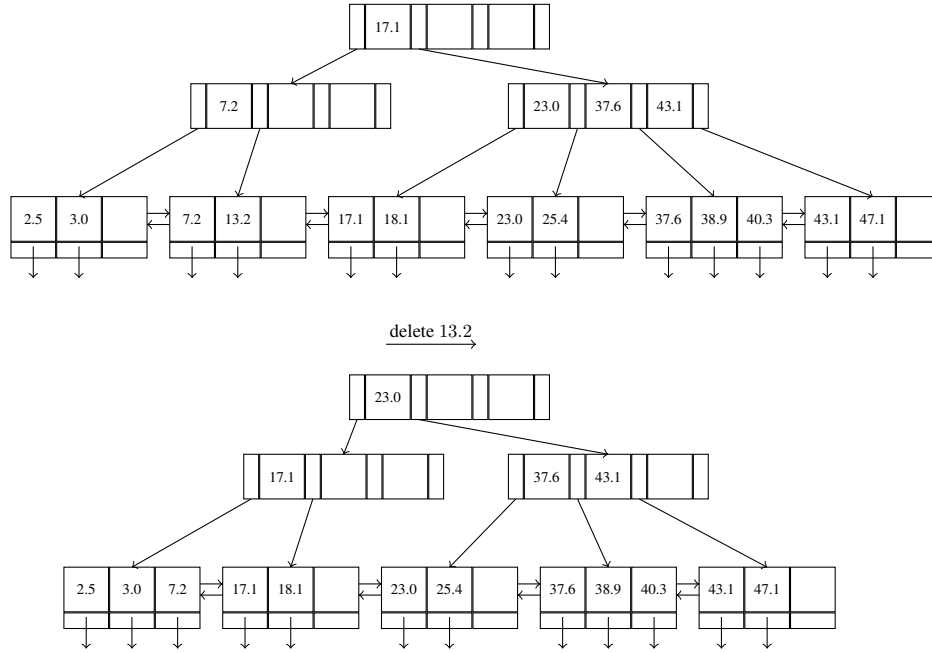


Figure 4.12.: Deleting 13.2 from the B^+ tree.

Example 4.3.1 We again refer to the example in Figure 4.8. The measurement with time point 13:15 dropped out of the sliding window W . 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 $2 = \lceil (n-1)/2 \rceil$. 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. Key 7.2 in the parent is removed as well by calling *DeleteEntry* recursively. Then the *DeleteEntry* procedure checks whether 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. The tree before and after deleting 13.2 is illustrated in Figure 4.12.

Algorithm 8: DeleteEntry(*tree*, *node*, *k*, *pointer*)

Input: Tree *tree*, the node *node* where the deletion key belongs to and *k* the key to delete

Output: The node *node* such that $k \notin \text{node}$

```
1 node ← remove k and associated linked list from node
2 if node is the root then
3   AdjustRoot(tree)
4   return
5 end
6 if node is leaf then
7   minNrOfKeys ←  $\lceil (n - 1)/2 \rceil$ 
8 else
9   minNrOfKeys ←  $\lceil n/2 \rceil - 1$ 
10 end
11 if minNrOfKeys ≤ number of keys left in node then
12   //node has still enough keys
13   return
14 end
15 //node has not enough keys - merge or rearranges necessary
16 neighborIndex ← get position of left sibling in parent or -1 if no left sibling
17 if neighborIndex = -1 then
18   kIndex ← 0
19   neighbor ← node.parent.pointers[1]
20 else
21   kIndex ← nIndex;
22   neighbor ← node.parent.pointers[nIndex]
23 end
24 //innerKeyPrime is the value between pointers to node and neighbor in parent
25 innerKeyPrime ← node.parent.pointers[kIndex]
26 capacity ← n-1
27 if node is a leaf then
28   capacity ← n
29 end
30 //Merge if both nodes together have enough space
31 if (neighbor.numOfKeys + node.numOfKeys) < capacity then
32   MergeNodes(tree, node, neighbor, nIndex, innerKeyPrime)
33 else
34   RedistributeNodes(tree, node, neighbor, nIndex, kIndex, innerKeyPrime)
35 end
```

4.3.3. Insertion in the B^+ tree

To insert a value to the tree, we first find the leaf node where the key would appear with Algorithm 6. 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 4.

The measurement is directly inserted to the leaf if there is an empty space, 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 9: AddMeasurement(*tree*, *t*, *v*)

Input: Tree *tree*, the new time point *t* and the new value *v*

Output: The *tree* such that $t, v \in \text{tree}$

```

1 //the tree does not exist yet - create tree
2 if tree.root = NIL then
3   |   newTree(tree, t, v)
4   |   return
5 end
6 leaf  $\leftarrow$  findLeaf(tree, v)
7 //insert to leaf as linked list value
8 if isDuplicateKey(leaf, t, v) then
9   |   i  $\leftarrow$  getInserPoint(tree, leaf, v)
10  |   AddNewTail(leaf, i, t)
11 else if leaf.numOfKeys < n-1 then
12  |   //enough space for new key value pair
13  |   InsertRecordIntoLeaf(tree, leaf, t, v)
14 else
15  |   SplitAndInsertIntoLeaves(tree, leaf, t, v);
16 end

```

Example 4.3.2 We again consider the example from the beginning in Figure 4.8, 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:

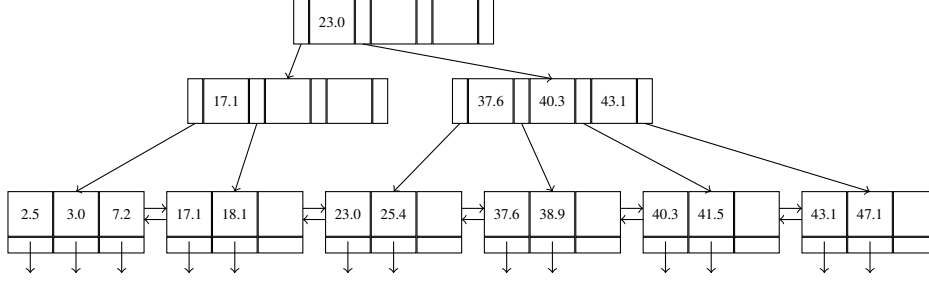


Figure 4.13.: B^+ tree after the insertion of 41.5.

4.4. Neighborhood

The $\text{sortedAccess}(v, T)$ method in Algorithm 11 uses the B^+ tree to return the time point $t \in T$ such that $|v - s(t)|$ is minimal, given a value v . But before a neighborhood can grow it has to be initialized. The neighborhood initialization is described in Section 10.

4.4.1. Initialize Neighborhood

The Algorithm 10 initializes a new neighborhood. A neighborhood consists of the following attributes:

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

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

A neighborhood $N(q_{ij})$ is defined around each value q_{ij} of a query pattern $Q(t)$, where leftPos^- and rightPos^+ are neighborhood positions of type *NeighborhoodPos* 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 neighborhoods. 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 8 If $d = 1$ just one time series is part of query pattern $Q(t)$. 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.

	j=1	j=2	j=3	
i=1	37.6	25.4	3.0	r_1
	14:10	14:15	14:20	

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

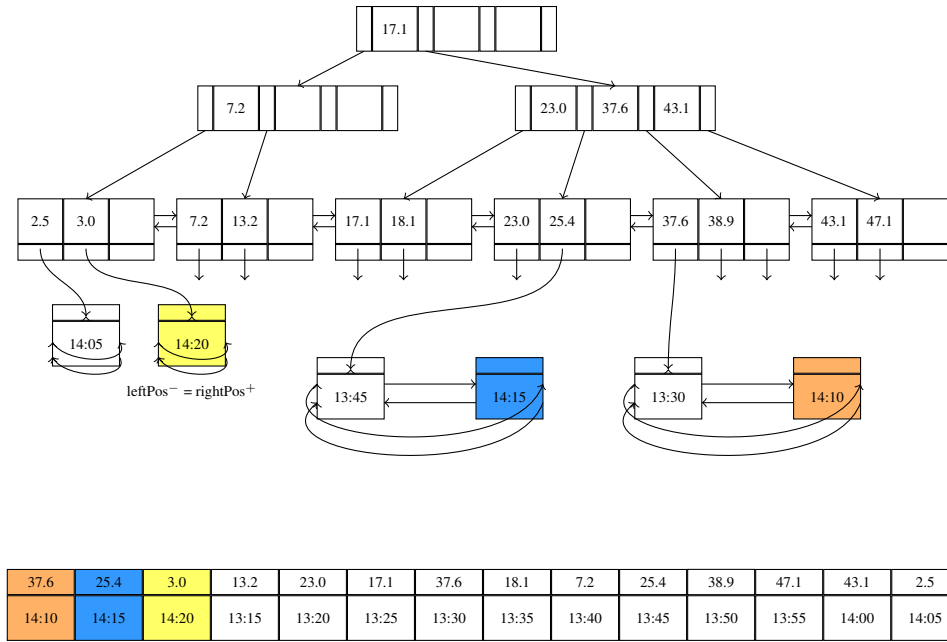


Figure 4.15.: The circular array and the B^+ tree for time series r_1 after initialization.

Algorithm 10: NewNeighborhood(t, v, j, l)

Input: Tree $tree$, the node $node$ and its neighbor $neighbor$, the neighborIndex $nIndex$, the $kIndex$ and the key $kPrime$

Output: The keys in the node and its neighbor, as well as the parents keys are redistributed

```
1 N.key  $\leftarrow$  measurement.value
2 N.j  $\leftarrow$  j
3 N.patternLength  $\leftarrow$  1
4 leafNode  $\leftarrow$  FindLeaf(tree, measurement.value)
5 pointerIndex  $\leftarrow$  GetInsertionIndex(leafNode, measurement.value)
6 element  $\leftarrow$  leafNode.pointers[pointerIndex]
7 //Upper Bound: The value is at most patternLength away form first list value
8 maxSteps  $\leftarrow$  1
9 while element.timestamp  $\neq$  measurement.timestamp && maxSteps  $\neq$  0 do
10   //go from newest value back towards oldest
11   element  $\leftarrow$  element.prev
12   maxSteps--
13 end
14 N.leftPos  $\leftarrow$  set position to element
15 N.rightPos  $\leftarrow$  set position to element
16 return neighborhood;
```

4.4.2. Sorted Access

After the initialization of the $l \times d$ neighborhoods the k non-overlapping patterns are calculated. Thus, the sortedAccess(v, T) operation is executed until the k patterns that contain the most similar values to q_{ij} are retrieved.

The sortedAccess(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 9 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:

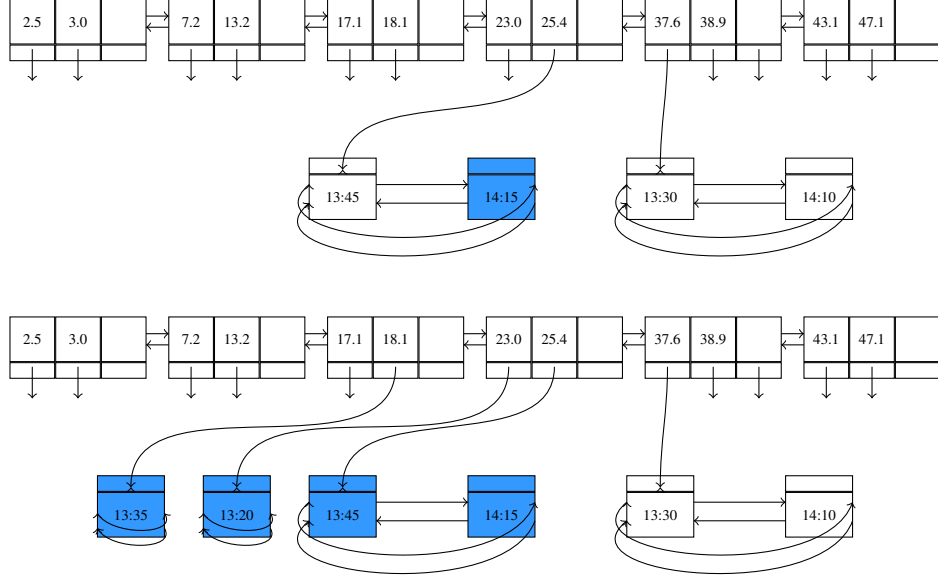


Figure 4.16.: Status of $N(q_{1,2})$ after growing 3 times.

Algorithm 11: SortedAccess($N(q_{i,j}), T$)

Input: The neighborhood N around query pattern value $q_{i,j}$ and the set of visited time points T

Output: Time point of the next most similar value to $q_{i,j}$

```

1 leftPos- ← N.leftPos
2 rightPos+ ← N.rightPos
3 while  $t^- \neq \text{NIL}$  and  $(t^- - (j + l)) \in T$  do
4   | leftPos- ← prev(leftPos-)
5 end
6 while  $t^- \neq \text{NIL}$  and TimeSetContains( $t^- - (j + l)$ ) do
7   | rightPos+ ← next(rightPos+)
8 end
9 if  $t^- \neq \text{NIL}$  and  $t^+ \neq \text{NIL}$  then
10  | if  $|r_i(t^-) - q_{i,j}| \leq |r_i(t^+) - q_{i,j}|$  then
11    | leftPos- ← prev(leftPos-);  $t \leftarrow t^-$ 
12  | else
13    | rightPos+ ← next(rightPos+);  $t \leftarrow t^+$ 
14  | end
15 else if  $t^- \neq \text{NIL}$  then
16  | leftPos- ← prev(leftPos-);  $t \leftarrow t^-$ 
17 else if  $t^+ \neq \text{NIL}$  then
18  | rightPos+ ← next(rightPos+);  $t \leftarrow t^+$ 
19 else
20  | return NIL
21 end
22 return t

```

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.

5.1. 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 $O(1)$ time. \square

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 $O(1)$ time. \square

Lemma 3 *The search of a specific time point in a linked list if $\text{newNeighborhood}(t, v, j, l)$ 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, starting from the tail and scanning backwards. \square

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 computed in $O(1)$ time, hence the update takes $O(1)$ time. \square

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

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

B^+ tree Operations

Normally, the complexity of operations on B^+ trees is dependent on the required I/O operations since I/O operations are expensive. The speed of operations on B^+ trees makes it a frequently used index structure in database implementations. But since we do not have I/O operations the complexity of our implementation is also dependent on the tree node size which usually is neglected.

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 `sortedAccess(v, T)` operation the pattern length l provides an upper bound for the maximum required value lookups in a 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 $O((\log_n(|W|) \times n))$. \square

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 linked list is always found in $O(1)$, duplicates have no influence to the complexity of an insertion. \square

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 linked list is always found in $O(1)$, duplicates as well have no influence to the complexity of a deletion. \square

Neighborhood Operations

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.

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

□

Lemma 10 *The $\text{sortedAccess}(v, T)$ 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 $\text{sortedAccess}(v, T)$ takes at most at most $O(|T|)$ time. □

5.1.1. Space Complexity

The B^+ tree structure imposes performance overhead on insertion and deletion, and adds space overhead. But the cost of reorganization of the measurements is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, is acceptable given the performance benefits of the B^+ tree structure.

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. □

6. Experimental Evaluation

This chapter describes our experimental setup and results. We evaluated the runtime of the *shift* algorithm with different values for the tree node size $n - 1$ and parameter W .

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 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 ($100 \times 365 \times 24 \times 20$). Assuming every of these 100 years has 365 days. We assume the data set contains values between 0 and 10000 and duplicates are possible. 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 increasing tree node size.

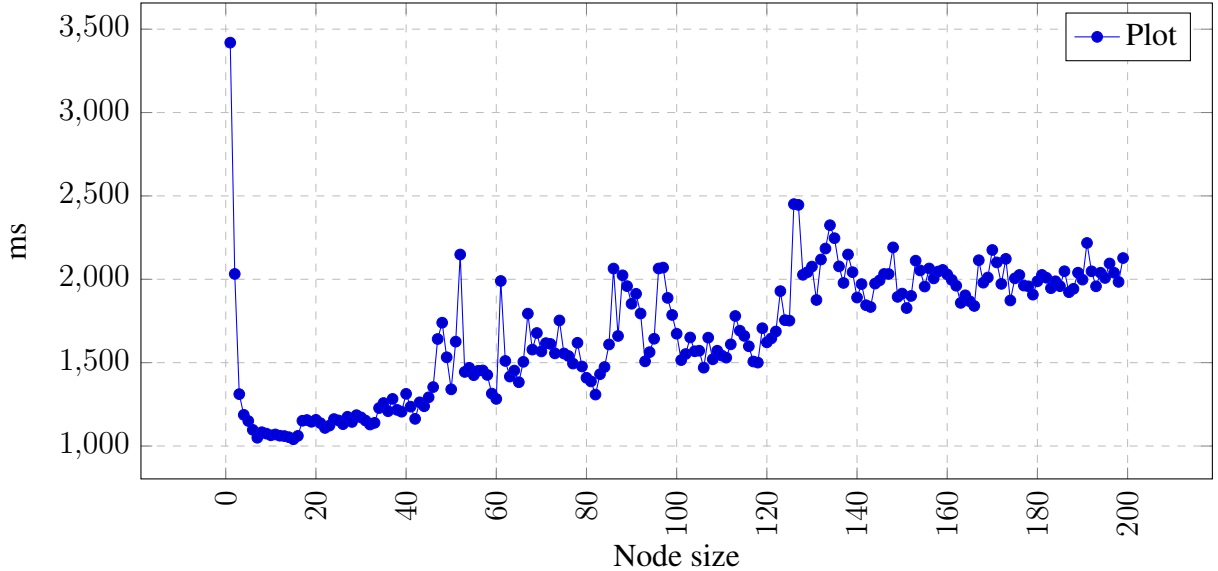


Table 6.1.: Shift operation with increasing values for $n - 1$.

Window Size. The window size $|W|$ determines the number of measurements stored in the circular array and the B^+tree . We show the effect of an increasing window size to the running time of the *shift* operation. The window size is between 60 days, hence 28800 measurements ($60 \times 24 \times 20$) and 3 years, 525600 measurements ($3 \times 365 \times 24 \times 20$). The tree node size is set to 40.

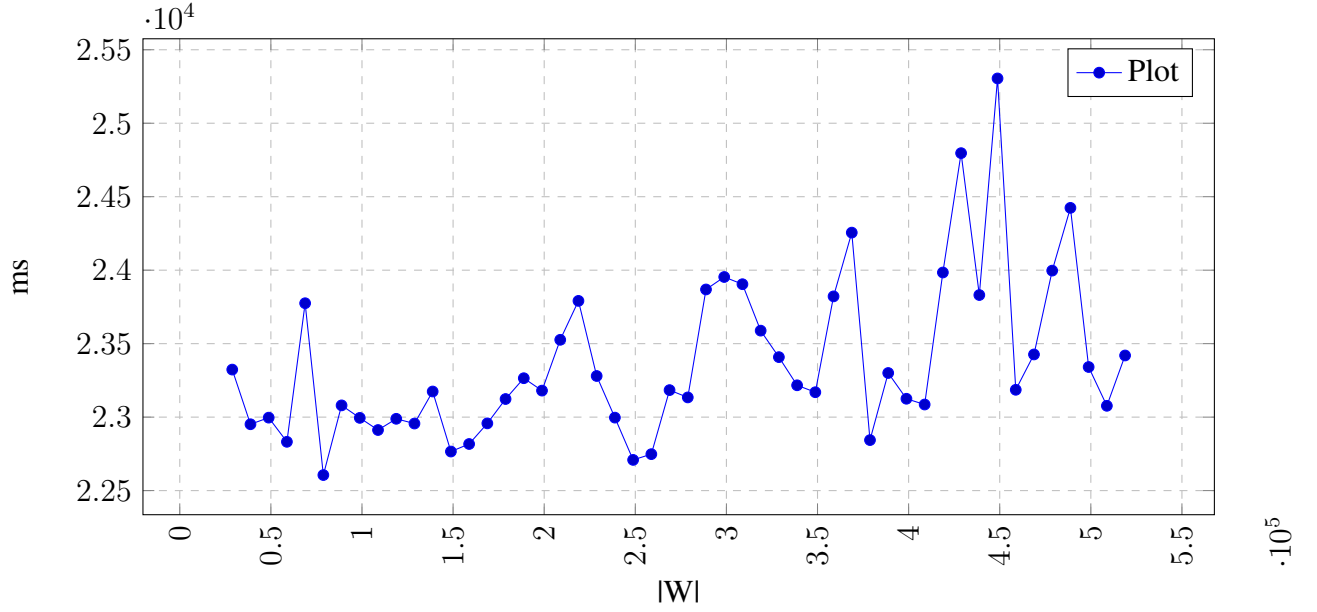


Table 6.2.: Shift operation with increasing values for $|W|$.

7. Summary and Conclusion

We studied the requirements for keeping a portion of a streaming time series in main memory and simultaneously 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 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. Dignös, 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.

Appendices

A. Algorithms

A.1. Deletion

Algorithm 12: AdjustRoot(*tree*)

Input: Tree *tree*

Output: The *tree* with an adjusted root node

```
1 //enough keys in the root
2 if  $0 < tree.root.numOfKeys$  then
3   | return
4 end
5 //if the root has a child, promote the first (only) child as the new root
6 if root is not a leaf then
7   | newRoot  $\leftarrow tree.root.pointers[0]$ 
8   | newRoot.parent  $\leftarrow NIL$ 
9 else
10  | newRoot  $\leftarrow NIL$ 
11 end
12 tree.root  $\leftarrow$  newRoot
```

Algorithm 13: *Redistribute(tree, node, neighbor, nIndex, kIndex, kPrime)*

Input: Tree *tree*, the node *node* and its neighbor *neighbor*, the neighborIndex *nIndex*, the *kIndex* and the the key *kPrime*

Output: The keys in the *node* and its *neighbor*, as well as the *parents* keys are redistributed

```
1 //node has neighbor to the left side
2 if nIndex != -1 then
3     //Pull neighbor's last key-pointer pair
4     over from the neighbor's right end to n
5     if node is not a leaf then
6         m ← neighbor.pointers[neighbor.numOfKeys]
7         insert neighbor.pointers[m] and kPrime to first position in node and shift other
            pointers and values right
8         remove neighbor.key[m-1], neighbor.pointers[m] from neighbor
9         replace kPrime in node.parent by neighbor.keys[m-1]
10    else
11        //last value pointer pair in the node
12        m ← neighbor.pointers[neighbor.numOfKeys - 1]
13        insert neighbor.pointers[m] and neighbor.keys[m] to first position in node and
            shift other pointers and values right
14        remove neighbor.key[m], neighbor.pointers[m] from neighbor
15        replace kPrime in node.parent by node.keys[0]
16    end
17 else
18     //node is leftmost child. Take a key-pointer pair from the neighbor to the right
19     //Move the neighbor's leftmost key-pointer pair to n's rightmost position
20     if node is not a leaf then
21         node.keys[node.numOfKeys] ← kPrime
22         node.pointers[node.numOfKeys + 1] ← neighbor.pointers[0]
23         replace kPrime in node.parent by neighbor.keys[0]
24         remove neighbor.keys[0], neighbor.pointers[0] from neighbor
25     else
26         node.keys[node.numOfKeys] ← neighbor.keys[0]
27         node.pointers[node.numOfKeys + 1] ← neighbor.pointers[0]
28         node.parent.keys[kIndex] = neighbor.keys[1]
29         remove neighbor.keys[0], neighbor.pointers[0] from neighbor
30     end
31 end
```

Algorithm 14: MergeNodes(*tree, node, neighbor, nIndex, kPrime*)

Input: Tree *tree*, the *node* and its neighbor *neighbor*, the neighborIndex *nIndex* and the key *kPrime*

Output: *node* and its *neighbor* are merged to one node

```
1 //Swap neighbor with node if node is on the extreme left and neighbor is to its right
2 if nIndex = -1 then
3   | swap neighbor with node
4 end
5 neighborInsertionIndex  $\leftarrow$  neighbor.numOfKeys
6 if node is no leaf then
7   | neighbor.keys[neighborInsertionIndex]  $\leftarrow$  kPrime
8   | neighbor.numOfKeys++
9   | decreasingIndex  $\leftarrow$  0
10  | numOfKeysBefore  $\leftarrow$  node.numOfKeys
11  | for i  $\leftarrow$  neighborInsertionIndex + 1, j  $\leftarrow$  0; j < node.numOfKeys do
12    | neighbor.keys[i]  $\leftarrow$  node.keys[j]
13    | neighbor.pointers[i]  $\leftarrow$  node.pointers[j]
14    | neighbor.numOfKeys++
15    | decreasingIndex++, i++, j++
16  | end
17  | node.numOfKeys  $\leftarrow$  numOfKeysBefore - decreasingIndex
18  | neighbor.pointers[i]  $\leftarrow$  node.pointers[j]
19  | //All children must now point up to the same parent
20  | for i  $\leftarrow$  0; i < neighbor.numOfKeys + 1; i++ do
21    | tmp  $\leftarrow$  neighbor.pointers[i]
22    | tmp.parent  $\leftarrow$  neighbor
23  | end
24 else
25   | // a leaf, append the keys and pointers of the node to the neighbor
26   | //Set the neighbor's last pointer to point to what had been the node's right neighbor
27   | for i  $\leftarrow$  neighborInsertionIndex, j  $\leftarrow$  0; j < node.numOfKeys do
28     | neighbor.keys[i]  $\leftarrow$  node.keys[j]
29     | neighbor.pointers[i] = node.pointers[j]
30     | neighbor.numOfKeys++, i++, j++
31   | end
32   | relink leaves
33 end
34 deleteEntry(tree, node.parent, kPrime, node)
```

A.2. Insertion

Algorithm 15: SplitLeaves(*tree*, *leaf*, *t*, *v*)

Input: Tree *tree*, the insertion node *leaf*, the time point *t* and the value *v*

Output: The leaf is split into two leaves

```
1 insertPoint  $\leftarrow$  0
2 nrOfTempKeys  $\leftarrow$  0
3 insertPoint  $\leftarrow$  getInsertPoint(tree, leaf, v)
4 //fills the keys and pointers
5 for i  $\leftarrow$  0, j  $\leftarrow$  0; i < oldNode.numOfKeys; do
6   if j = insertPoint then
7     | j++
8   end
9   tempKeys[j]  $\leftarrow$  oldNode.keys[i]
10  tempPointers[j]  $\leftarrow$  oldNode.pointers[i]
11  nrOfTempKeys++, i++, j++
12 end
13 //enter the record to the right position
14 tempKeys[insertPoint]  $\leftarrow$  v
15 newList  $\leftarrow$  create list and insert t
16 tempPointers[insertPoint]  $\leftarrow$  newList
17 nrOfTempKeys++
18 newNode.numOfKeys  $\leftarrow$  0
19 oldNode.numOfKeys  $\leftarrow$  0
20 //calculate splitpoint by  $\lceil n/2 \rceil$ 
21 split = GetSplitPoint(n-1)
22 //fill first leaf
23 for i  $\leftarrow$  0; i < split do
24   | oldNode.keys[i]  $\leftarrow$  tempKeys[i]
25   | oldNode.pointers[i]  $\leftarrow$  tempPointers[i]
26   | oldNode->numOfKeys++, i++
27 end
28 //fill second leaf
29 for j  $\leftarrow$  0, i  $\leftarrow$  split; i < nrOfTempKeys; do
30   | newNode.keys[j]  $\leftarrow$  tempKeys[i]
31   | newNode.pointers[j]  $\leftarrow$  tempPointers[i]
32   | newNode->numOfKeys++, i++, j++
33 end
34 link leaves
35 newNode.parent  $\leftarrow$  oldNode.parent
36 keyForParent  $\leftarrow$  newNode.keys[0]
37 insertIntoParent(tree, oldNode, keyForParent, newNode)
```

Algorithm 16: SplitInnerNodes(*tree*, *oldInnerNode*, *index*, *key*, *childNode*)

Input: Tree *tree*, the node *oldInnerNode* and the child node *childNode*, the index *index* and in addition the *key*

Output: The inner node is split into two nodes

```
1 nrOfTempKeys  $\leftarrow$  0, x  $\leftarrow$  0
2 for  $i \leftarrow 0, j \leftarrow 0; i < oldNode.numOfKeys; do$ 
3   if  $j = index$  then
4     |  $j++$ 
5   end
6   tempKeys[j]  $\leftarrow$  oldInnerNode.keys[i], nrOfTempKeys++, i++, j++
7 end
8 for  $i \leftarrow 0, j \leftarrow 0; i < oldInnerNode.numOfKeys + 1; do$ 
9   if  $j = index + 1$  then
10    |  $j++$ 
11  end
12  tempPointers[j]  $\leftarrow$  oldInnerNode.pointers[i], i++, j++
13 end
14 newInnerKey  $\leftarrow$  key
15 tempKeys[index]  $\leftarrow$  newInnerKey, tempPointers[index + 1]  $\leftarrow$  childNode
16 nrOfTempKeys++
17 newInnerNode.numOfKeys  $\leftarrow$  0, oldInnerNode.numOfKeys  $\leftarrow$  0
18 split  $\leftarrow$  getSplitPoint(n)
19 for  $x < split; do$ 
20   oldInnerNode.keys[x]  $\leftarrow$  tempKeys[x]
21   oldInnerNode.pointers[x]  $\leftarrow$  tempPointers[x]
22   oldInnerNode.numOfKeys++, x++
23 end
24 oldInnerNode.pointers[x]  $\leftarrow$  tempPointers[x]
25 leftMostKey  $\leftarrow$  tempKeys[x]
26 newInnerNode.parent  $\leftarrow$  oldInnerNode.parent
27 newInnerNode.numOfKeys  $\leftarrow$  (nrOfTempKeys - oldInnerNode.numOfKeys - 1)
28 for  $++x, j \leftarrow 0; j < newInnerNode.numOfKeys; do$ 
29   //first key is not inserted to this node - it is inserted to upper node
30   newInnerNode.pointers[j]  $\leftarrow$  tempPointers[x]
31   newInnerNode.keys[j]  $\leftarrow$  tempKeys[x], j++, x++
32 end
33 newInnerNode.pointers[j]  $\leftarrow$  tempPointers[x]
34 for  $i \leftarrow 0; i < newInnerNode.numOfKeys + 1; do$ 
35   childOfNewNode  $\leftarrow$  newInnerNode.pointers[i]
36   childOfNewNode.parent  $\leftarrow$  newInnerNode, i++
37 end
38 insertIntoParent(tree, oldInnerNode, leftMostKey, newInnerNode)
```

Algorithm 17: InsertIntoParent(*tree*, *oldChild*, *k*, *newChild*)

Input: Tree *tree*, the newly created *newChild* and the *oldChild* and the key *k*

Output: The key *k* is inserted to the parent or the parent is split

```
1 parent ← oldChild.parent
2 if parent = NIL then
3   | insertIntoANewRoot(tree, oldChild, k, newChild)
4   | return
5 end
6 pointerPos ← pointer position index from parent to oldChild
7 //the new key fits into the node
8 if parent.numOfKeys < n-1 then
9   | insertIntoTheNode(parent, pointerPos, k, newChild)
10 else
11   | splitAndInsertIntoInnerNode(tree, parent, pointerPos, k, newChild)
12 end
```

The entire source code can be found here:

<https://github.com/memast2/BA-TimeSeriesData>

B. Experimental Results

The entire source code can be found here:

https://github.com/memast2/BA-TimeSeriesData/tree/master/Experimental_Results