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 guidance. 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. Besides, I would like to thank my boyfriend and my family for 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 the 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 latest portion of past measurement data. We introduce an implementation using two data structures, a circular array and a B^+tree, to efficiently access the data of past measurements. The results of an experimental evaluation show the influence of the data structures on the system performance.

Zusammenfassung

Kontinuierliche Zeitreihen werden durch neu ankommende Daten stetig erweitert. Die Daten treffen dabei in einem vordefinierten Intervall ein. 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 der aktuellsten Daten im Arbeitsspeicher. Daraus resultiert, dass das System auf diese Daten zugreifen kann. Dazu stellen wir unsere Implementation vor, welche sich zwei Datenstrukturen zunutze macht: ein zirkuläres Array und ein B^+baum. Die beiden Datenstrukturen erlauben den effizienten Zugriff auf alle Werte der vergangenen Daten, welche sich in einem bestimmten Zeitfenster befinden. Die Resultate einer experimentellen Evaluation zeigen den Einfluss der Datenstrukturen auf die Laufzeit des Systems.

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). Such time series are relevant to applications in diverse domains. For example, imagine a meteorology station that sends a temperature measurement every 3 minutes or, to name a different domain, 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.
But not only the current data can be interesting for an application. A streaming time series contains historical data that can be exploited. For example to recover missing values since a streaming time series is not always gapless. Due to sensor failures or transmission errors, values can get missing. To recover missing values the historical data can be utilised. Ideally, the missing value should be recovered efficiently, such that it is imputed before another new value arrive. Therefore, the efficient access of past time series data is important.
We present a system that keeps a limited size of data in main memory. The measurements with time points in a sliding time window are kept.

Definition 1 *(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 point* $\bar{t}$ *represents the most recent time point for which the stream produced a new value.*

Because a portion of data is kept it is still accessible. The thesis proposes an implementation for an index structure for streaming time series data to efficiently access the time point and the value of measurements. In order to achieve efficient data access, the system uses two data structures: a circular array in combination with a B^+tree. The circular array has a size of length $|W|$. The measurements are stored in the array, sorted by time. The B^+tree as well contains the same measurements with time points in the 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 that the stream produces 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. Hence, if a measurement time point is not in the sliding window any more, it is removed from both data structures.
The described system is implemented and its advantages are outlined in this thesis. Furthermore, the system is analysed in terms of runtime and space complexity. Finally, an experimental evaluation tries to underpin the theoretical results.

At the beginning of this thesis, in Chapter 2, the TKCM algorithm is introduced. TKCM is designed by Wellenzohn et al.[1]. The background aims to improve the understanding for the system context and for the requirements our system must satisfy. Chapter 3 describes the

context and introduces the operations the system must be able to perform. In Chapter 4, our approach is presented and its advantages are discussed. Further, the pseudo-code to implement the system is outlined. After that, the time and space complexity of the system is described in Chapter 5. Followed by an experimental evaluation in Chapter 6 to examine the theoretical results. Finally, the thesis is summed up in Chapter 7.

2. Background

A streaming time series is not always gapless, but missing values can be recovered with the help of past data. To efficiently recover missing values, Wellenzohn et al.[1] present the Top-k Case Matching (TKCM) algorithm. The algorithm and its connection to this thesis is introduced in Section 2.1.

2.1. TKCM

TKCM recovers and imputes missing measurement values if it detects any. For each time series s a set of correlated reference time series is determined to recover a missing value. Therefore, TKCM monitors a set of streaming time series and once a value is missing 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.

Definition 2 *(Time Series) Let S be a set $S = \{s_1, s_2, ...\}$ of streaming time series. The measurement value of time series $s_i \in S$ at time t is denoted as $s_i(t)$ and is represented by the tuple $(t, s_i(t))$. For a 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.*

TKCM retains a sliding window of the streaming time series in main memory. 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 searches for the *k* most similar patterns in sliding window W.

	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 $Q(\bar{t})$ of length $l = 3$ and $d = 2$ reference time series at time point $\bar{t} =$ 14:20.

Definition 3 *(Pattern) Let $R_s^d = \langle r_1, ..., r_d\rangle$ be the sequence 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 $p_{i,j} = S_i(t - l + j)$. We write $p_{i,j} \in P(t)$ to denote that value $p_{i,j}$ is contained in $P(t)$.

The two-dimensional query pattern $Q(\bar{t})$ is anchored at time point $\bar{t}$ and consists 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 as illustrated in Figure 2.1.
For every time point t in W exists a measurement which means $\forall t : \underline{t} \leq t < \bar{t} \rightarrow s(t) \neq \text{NIL}$ 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 looks for a new pattern $P(t)$ in time window W, such that $t \in W$, by *growing* the neighborhood $N(q_{i,j})$ until the k most similar patterns to $Q(\bar{t})$ have been found. TKCM stores the seen time points t in a set T to avoid considering these patterns again.

Definition 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 $t = \underset{t \in W \setminus T}{\operatorname{argmin}} |p_{i,j} - q_{i,j}|$*

To recover a missing value in a time series, past measurements need to be accessed efficiently. Therefore, Wellenzohn et al.[1] suggest a combination of two data structures: a B^+tree and a circular array. The necessary access methods are described in Section 2.2.

2.2. Access Methods

The next pattern (Def. 4) is built on top of two access methods: sorted and random access.

Definition 5 *(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 6 *(Random Access) Random access returns value $r(t)$, given time series r and time point t.*

Sorted access finds the most similar value to a given query pattern cell $q_{i,j}$ (Def. 3), then, random access retrieves the values to fill the remaining pattern cells.
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 query pattern value $q_{i,j}$. The random access mode is used to look up the values that pattern $P(t)$ is composed of.

Time t	13:40	13:45	13:50	13:55	14:00	14:05	14:10	14:15	14:20
$s(t)$	15.9	16.0	16.3	15.9	16.2	15.8	15.9	16.1	x
$r_1(t)$	15.6	15.9	16.7	17.0	17.2	16.2	16.1	16.3	16.5
$r_2(t)$	16.1	16.6	17.1	15.2	16.0	15.4	17.1	17.0	17.2

Table 2.1.: Data of the streaming time series s and the reference time series r_1 and r_2

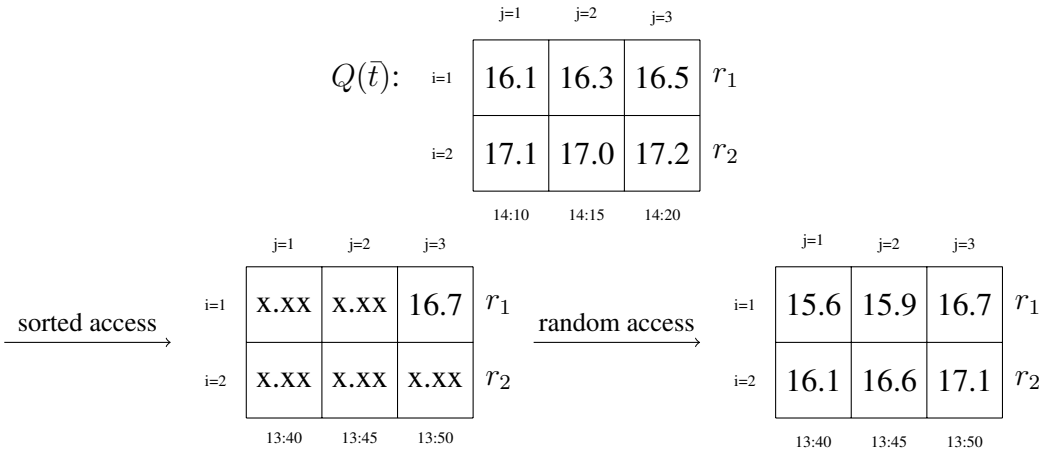


Figure 2.2.: Pattern for query pattern cell $q_{1,3}$.

Example 1 *As illustrated in Table 2.1 the value in time series s at time point 14:20 is missing. Therefore, a pattern is anchored at this time point. To find a pattern for pattern cell $q_{1,3}$ the next most similar value to 16.5 is searched using sorted access as shown in Figure 2.2. This means, the difference between 16.5 and 16.7 is the smallest compared to other values in reference time series r_1. Then, the missing pattern values are filled. Therefore, the values in r_1 at time point 13:40 and 13:45 and the values in r_2 at 13:40, 13:45 and 13:50 are inserted using random access.*

Algorithm 1: NextPattern($N(q_{i,j}), T$)

Input: Neighborhood $N(q_{i,j})$ and seen time points T
Output: Next pattern $P(t')$

```
P ← matrix of size d × l
(t, s_i(t)) ← sortedAccess(N(q_{i,j}),T)
P[i,j] ← s_i(t)
foreach q_{x,y} ∈ Q(t̄) do
    if x ≠ i and y ≠ j then
        P[x,y] ← randomAccess(s_x, t − j + y)
    end
end
return P anchored at time t − j + l
```

3. Problem Definition

The two access methods described in Section 2.2 are crucial for TKCM. However, they can be used in every application that needs access to past time series data. Therefore, the access methods should be usable for time series data in general. Hence, the data interval and the sliding window size should be variable. Furthermore, the access methods should be executable with duplicate values, since the measurements in time series data normally are not unique.
The present thesis introduces an implementation of the random and sorted access method for a streaming time series s. 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 predetermined interval (E.g. every five minutes).
- There are no gaps between the measurements since gaps are filled before additional values are inserted to the data structure.
- No measurements 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$(tree, array, \bar{t}, v)$: 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.
- sortedAccess$(N(q_{i,j}), T)$: given a neighborhood $N(q_{i,j})$ and a set of time points T, return the time point $t \notin T$, as described in Definition 5.
- randomAccess$(array, t)$: return the value of time series *s* at time *t*, denoted by $s(t)$ (Def. 6).
- newNeighborhood$(tree, q_{i,j}, t, j, l)$: given a value $q_{i,j} \in Q(\bar{t})$ for time point t, the pattern length l and the index j, return the new neighborhood $N(q_{i,j})$.

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. The thesis introduces an implementation of the random and sorted access method for a streaming time series s. The data structures and their advantages are described in Chapter 4. Further, the thesis proposes a solution for handling duplicate values.

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 shows the data structures. All measurements in the circular array are represented in the B^+tree as well.

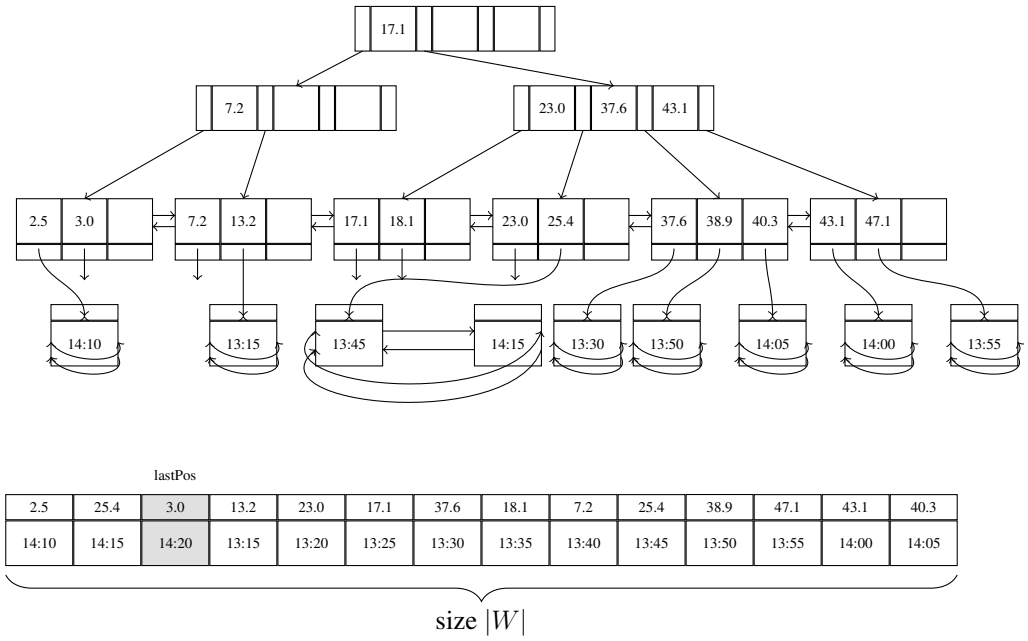


Figure 4.1.: A B^+tree with the related circular array.

4.1. Circular Array

A circular array is used to store the time series data sorted by time in the predefined window W. The value and the time point 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 array. Hence, an array with $|W| = 10$ can hold 10 measurements. The last update position $lastPos$ is stored in a variable and is updated with every insertion. In detail, a circular array contains the following attributes: a counter $count$ that counts the number of measurements in the array, a $lastPos$

to store the position that was last updated with a measurement and finally the data, which actually holds the time points and values of the 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 in array 1 in Figure 4.2,.

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 to array 2.

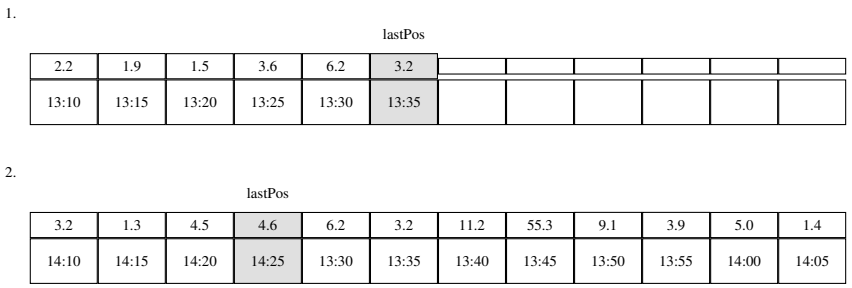


Figure 4.2.: Shifted circular array.

Example 2 *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 i.e. 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 fix 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 interval. The detailed calculation of a value position is described in Section 4.1.3.*

4.1.2. Shift

The measurements in a circular array are stored in a defined interval without any gaps in between. The measurement position in the array can be calculated with the $lastPos$ and the time difference between two consecutive measurements. 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 and simultaneous the conceivable deletion of a 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 ← 0
2  if array.count < |W| then
3  |   if array.count ≠ 0 then
4  |   |   newPos ← (array.lastPos + 1) % |W|
5  |   end
6  |   array.count++
7  else
8  |   newPos ← (array.lastPos + 1) % |W|
9  |   //delete the measurement that dropped out of the window Delete(tree,
   |   array.data[newPos].time, array.data[newPos].value)
10 end
11 array.data[newPos].time ← t̄
12 array.data[newPos].value ← v
13 array.lastPos ← newPos
14 AddMeasurement(tree, t̄, 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 shows 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. The position of a measurement in the array can be directly calculated without looping through the array, since the values arrive in a predefined interval (e.g. every 3 minutes). The random access operation is used to fill a pattern $P(t)$ that is anchored at a time point t. The last update position is used as reference time point for the calculation. The randomAccess$(array, t)$ is presented in Algorithm 3.

Algorithm 3: RandomAccess$(array, t)$

Input: The circular array $array$ and the time point t
Output: Returns the value $p_{i,j}$ at time point t

```
1  if array.count = 0 then
2  |   //empty array
3  |   return NIL
4  end
5  step ← (t - array.data[array.lastPos].time)
6  if |step| < array.count then
7  |   pos ← (array.lastPos + step)%|W|
8  |   if array.data[pos].time = t then
9  |   |   return array.data[pos].value
10 |   end
11 end
12 return NIL
```

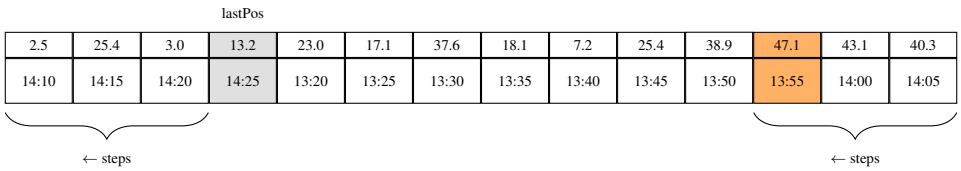


Figure 4.4.: Random access of the value at time point 13:55.

Example 4.1.2 *We want to find the measurement value at time point 13:55 in the circular array shown in Figure 4.4. The last update position was at time point 14:25 and therefore the step is* (-6)*. Since 13:55 −14:25* $= -30$ *which is* 6 *times the time difference of 5 minutes. Then the measurement position is calculated:* $(lastPos + (-6))\%14 = 11$*. the value is at position* 11*. Thus, the value with time point 13:55 is* 47.1*. The value is returned by the random access method.*

4.2. B^+tree

A B^+tree is organized with nodes that include search keys. The search key values in every node are kept in sorted order. A B^+tree is a balanced tree, hence every path from the root of the tree to a leaf of the tree has the same length. Each nonleaf node in the tree has between $\lceil n/2 \rceil$ and n children and n is predefined 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 because the values in the B^+tree leaves 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 disk 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.
But although we do not have disk I/O operations in our system the B^+tree has useful properties. The B^+tree can be used to efficiently perform the sortedAccess$(N(q_{i,j}), T)$ 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 we used 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 [2]. The differences between the traditional B^+tree in [2] 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 sortedAccess$(N(q_{i,j}), T)$ operation. On the other hand, our B^+tree is able to handle duplicate values. How the tree handles duplicate values is described in Section 4.2.2. The other properties of our 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 search keys, hence the size of a tree node. The keys in a node are always sorted from left to right.

Example 3 *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 for $n = 7$.*

A node contains m non-null pointers $(m \leq n)$ and $m-1$ non-null keys. For $i = 0, 1, 2, 3, ..., m$, pointer P_i points to the subtree that contains search key values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the subtree that contains those key values greater than or equal to K_{m-1}.

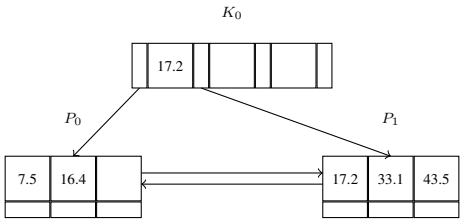


Figure 4.5.: Left children keys < 17.2 and right children keys $\geq$ 17.2.

Example 4 *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 are 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 $n-1$ search keys 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 search key and two pointers to child nodes, unless the root is a leaf node, thus, 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 node contains: 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 leaf node can have two pointers, one to the previous and one to the next sibling. 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 leaf of the 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.

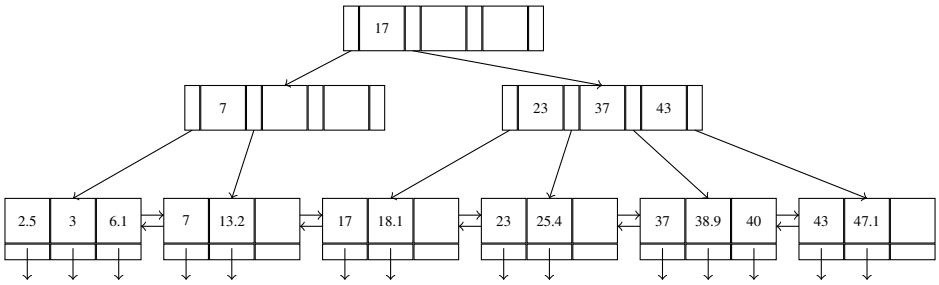


Figure 4.6.: Example of a complete B^+tree.

Example 5 *We want to know the most similar value to the key* 40 *in the* B^+*tree illustrated in Figure 4.6. We have to compare* 40 *to two values, namely the right neighbor* 43 *and the left neighbor* 38.9*. We find out that* 38.9 *is the most similar value in the entire* B^+*tree since* $|18.9 - 40| < |43 - 40|$.

Application to our System

The B^+tree described above contains search keys. In our case, these search keys are measurement values. A value in time series s can occur multiple times. Hence, the values are not unique and since the values are used as search keys, 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 measurement time points 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 successor.

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 linked 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 list the pointers $next$ and $prev$ point to e itself.

We use a doubly, circular linked list in our system. For clarity, we name the circular doubly linked list in the following only linked list. A measurement time point is stored in a linked list element. Every value 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 measurement time points are added to the linked list associated to the leaf value. So instead of inserting the value 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 with 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 $tail$.

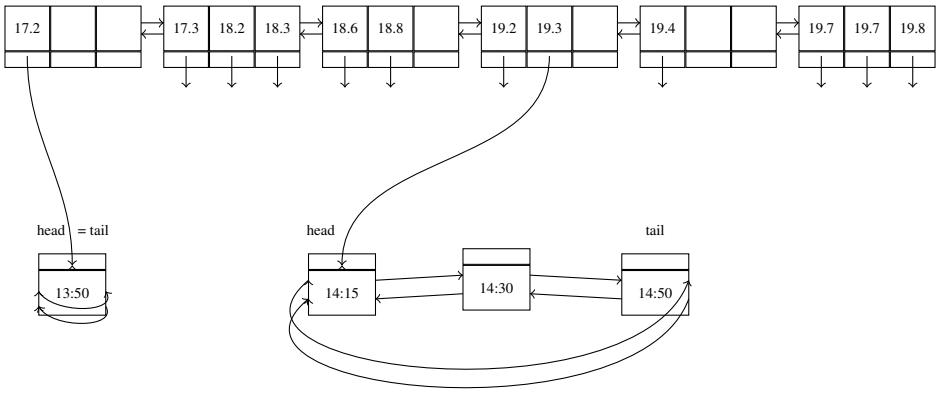


Figure 4.7.: Circular doubly linked lists associated to leaf nodes.

Example 6 *Figure 4.7 illustrates the leaf level of a B^+tree and the 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 of the head. 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.*

As described, 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 search keys in the leaf. Hence, a pointer P_i points to the linked list associated to the leaf value 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{
  timePoint time;
  struct Element *prev, *next;
};
```

Observation

The linked list associated to the measurement value stores the corresponding 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 on the circular array leads to a deletion of the oldest measurement in time series s, thus, the measurement time point, as explained, is always at the head position in a linked list. Also, a new measurement can be inserted without looping through the list. It is always added to the tail position.

The insertion and the deletion of a time point from a linked list that contains multiple values is illustrated in Algorithm 4 and Algorithm 5[1], respectively.

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

Input: Leaf $node$, the index position i to the linked list L and the time point t to insert
Output: Linked List L such that $t \in L$

```
1 head ← node.pointers[i]
2 tail ← head.prev
3 newElement = CreateElement(t)
4 //insert new linked list element between head and tail
5 head.prev = newElement
6 tail.next = newElement
7 newElement.prev = tail
8 newElement.next = head
```

Algorithm 5: DeleteHead$(node, i)$

Input: Leaf $node$ and index position i for the position of the associated Linked List L
Output: Linked List L such that $t \notin L$

```
1 head ← node.pointers[i]
2 nextElement ← head.next
3 prevElement ← head.prev
4 leaf.pointers[i] ← nextElement
5 prevElement.next ← nextElement
6 prevElement.prev ← prevElement
```

[1] Algorithm 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 of the sliding window W leads to an array update. The operation updates 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 deletion and the insertion in a B^+tree are described in Section 4.3.2 and Section 4.3.3, respectively.

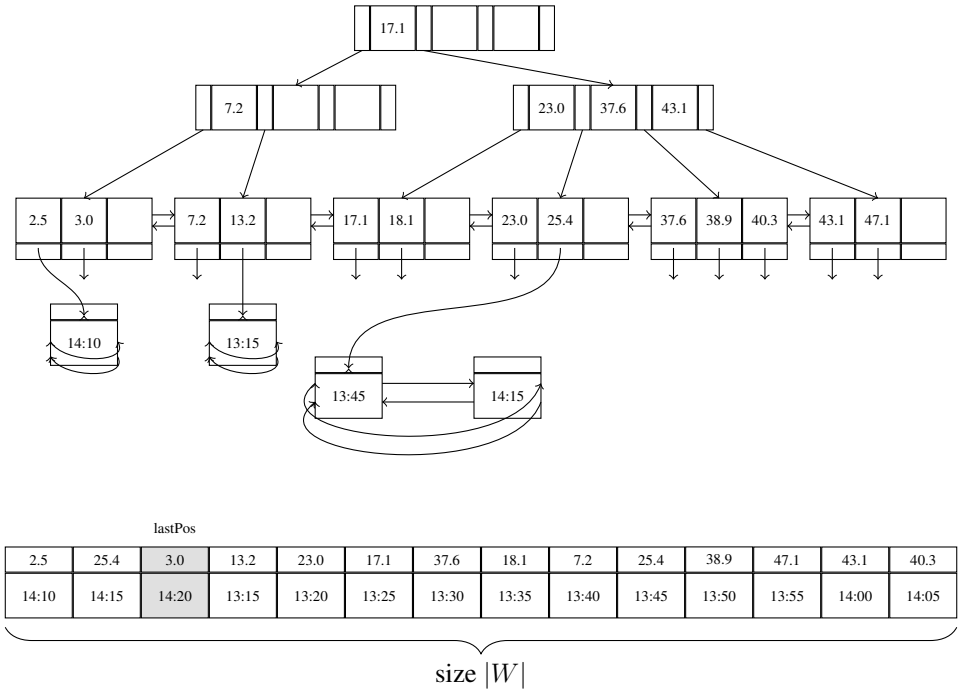


Figure 4.8.: Start situation

Example 7 *We assume we have the situation illustrated in Figure 4.8. Value* 25.4 *occurs two times. Hence, the associated linked list contains two elements. Further, all leaf values have an associated time point. 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 the 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. The method starts at the root of the tree and traverses the tree until it reaches the appropriate leaf node which would contain the value. The current node is examined by looking for the smallest i for which the search key 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 search key k
Output: The appropriate leaf for the search key k

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

Example 8 *We assume we want to find the key* 13.2 *in the* B^+*tree illustrated in Figure 4.8 because this is the value that dropped out of the sliding window* W*.*
The current node is examined by looking for the smallest i *for which the search key 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 search key* 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 leaf node is returned.*

4.3.2. Deletion in the B^+tree

In case a measurement value dropped out of the sliding window W, we first have to delete this measurement from the tree, before we can add the 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 value has multiple linked list elements. If the linked list has multiple elements, the identified element is deleted from the linked list using Algorithm 5 and the deletion is already finished. If the entry time point is the single value in the linked list the leaf value and its belonging linked 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 ← findLeaf(tree, k)
2 i ← smallest number such that k ≤ leaf.K_i
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($tree, node, k, pointer$) method illustrated in Algorithm 8 is called if the measurement to delete 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 must contain 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 the search key 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. The tree must still be 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 a leaf that has still enough keys.

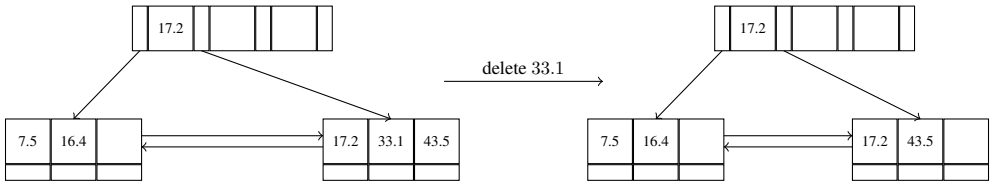


Figure 4.9.: The node has still enough keys.

2. (The node is merged with a sibling) Provided that, the keys and pointers of the node and its sibling fit into a single node and the node is not half-full after removing the entry, we merge them. We move the entries from the right sibling into the left sibling, and delete 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. This is done by calling deleteEntry($tree, node, k, pointer$) again. We traverse the tree recursively upwards until the *deleteEntry* stops. Figure 4.10 illustrates a node that is merged with a sibling. The Algorithm 13 for the redistribution can be found in Appendix Section A.1.

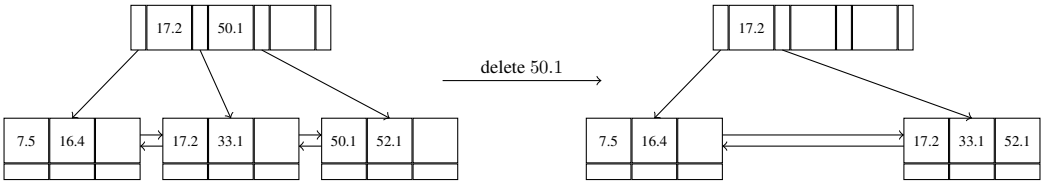


Figure 4.10.: The node can be merged with its sibling.

3. (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 are 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 $\lceil n - 1/2 \rceil$ keys. 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 illustrates the case where some keys have to be redistributed. The Algorithm 14 for a redistribution can be found in Appendix Section A.1.

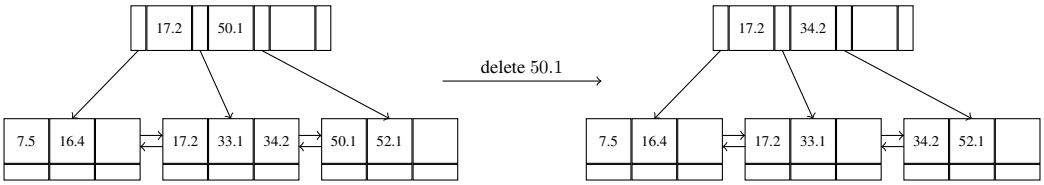


Figure 4.11.: Some entries must be redistributed.

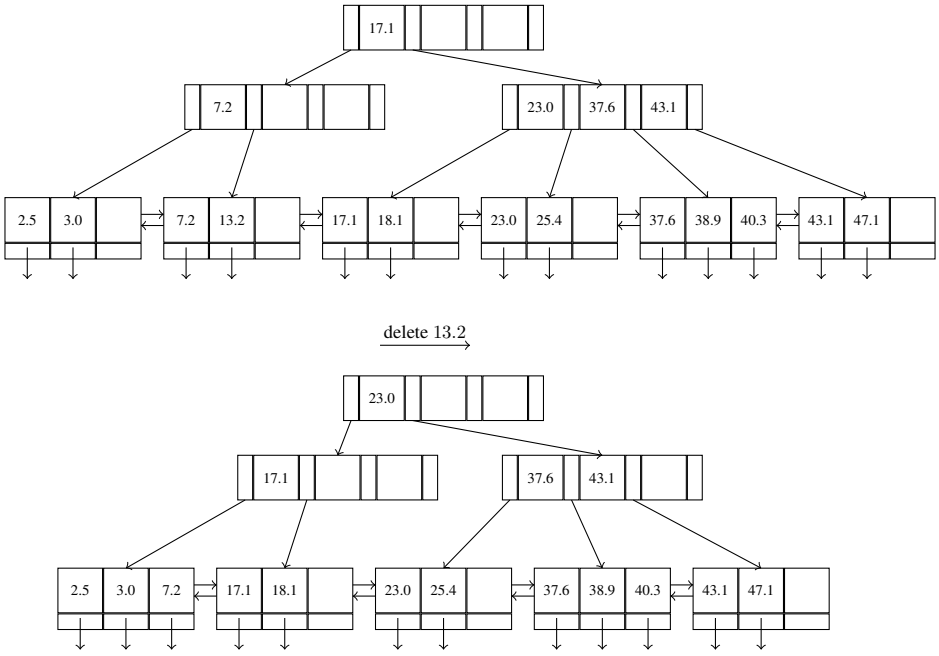


Figure 4.12.: Deleting 13.2 from the B^+tree.

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 now has only* 1 *key left and therefore is smaller than the minimum number of allowed keys of* $2 = \lceil (n-1)/2 \rceil$*, since* $1 < \lceil (4-1)/2 \rceil$*. 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* deleteEntry *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 right 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 $k \notin node$

```
node ← remove k and associated linked list from node
if node is the root then
    AdjustRoot(tree)
    return
end
if node is leaf then
    minNrOfKeys ← ⌈(n − 1)/2⌉
else
    minNrOfKeys ← ⌈n/2⌉ − 1
end
if minNrOfKeys ≤ number of keys left in node then
    //node has still enough keys
    return
end
//node has not enough keys - merge or rearranges necessary
neighborIndex ← get 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 ← nIndex;
    neighbor ← node.parent.pointers[nIndex]
end
//innerKeyPrime is the value between pointers to node and neighbor in parent
innerKeyPrime ← node.parent.pointers[kIndex]
capacity ← n-1
if node is a leaf then
    capacity ← n
end
//Merge if both nodes together have enough space
if (neighbor.numOfKeys + node.numOfKeys ) < capacity then
    MergeNodes(tree, node, neighbor, nIndex, innerKeyPrime)
else
    RedestributeNodes(tree, node, neighbor, nIndex, kIndex, innerKeyPrime)
end
```

4.3.3. Insertion in the B^+tree

We now describe the insertion of a measurement to the B^+tree. The general technique of an insertion into a B^+tree is to first determine the appropriate leaf node for the new measurement with the *findLeaf* method. After the right leaf has been found, there are three possible consequences for the B^+tree: The key already exists in the leaf, the key does not exist yet and the leaf has still enough space, and finally the key does not exist yet but the leaf is already full, since $numOfKey = n - 1$.

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, $v \in tree$

```
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 ← findLeaf(tree, v)
7  //insert to leaf as linked list value
8  if key already exists in leaf then
9  |   i ← smallest number such that k ≤ leaf.K_i
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
```

1. (Search key already exists in the leaf) If the search key already exists in the leaf node the time point of the measurement is added to the already existing associated list and the search keys in the leaf remain unchanged. In this case, refer to Algorithm 4.

2. (Search key does not exist yet and the leaf has enough space) We insert the entry to the node such that the search keys are still in order and a new linked list is allocated where the measurement time point is inserted.

3. (Search key does not exist yet but the leaf has not enough space) In general, we take n search key values (the values $n - 1$ in the leaf node plus the value being inserted to the right position such that the keys remain ordered) and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a newly created node. If a node is split, we consequently must insert the new leaf node into the B^+tree structure and re-link the leaves. We need to insert an entry with the leftmost key of the new leaf node, and a pointer to the new node, into the parent of the leaf node that was split. If there is no room, the parent must be split, requiring an key to be added to its parent. In the worst case,

all nodes in the path to the root must be split. If the root itself is split, the entire tree becomes one level deeper. Splitting of a nonleaf node is a different from splitting of a leaf node. If there is no space in the parent node to add a new key the parent node has to be split as well. When an nonleaf node is split, the child pointers are divided among the original and the newly created node. The key values are handled slightly differently if a inner node is split. The key values that lie between the pointers moved to the right node are moved along with the pointers, while those that lie between the pointers that stay on the left remain unchanged. However, the search key value that lies between the pointers that stay on the left, and the pointers that move to the right node is treated differently. This search key is not added to either of the two nodes. Instead, it is added to the parent node. The Algorithms for splitting a leaf node or an inner node can be found in Appendix Section A.2.

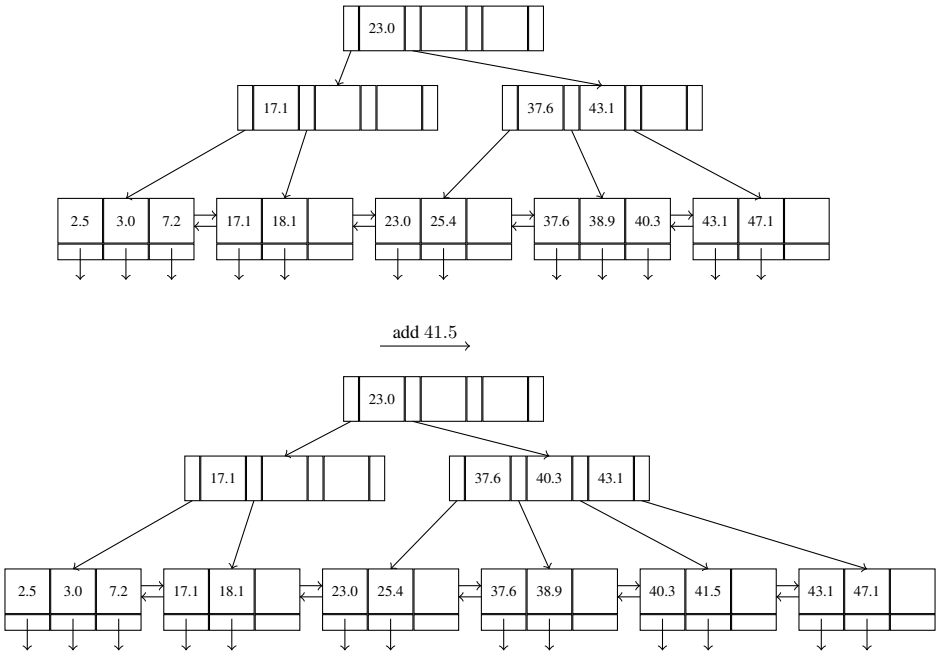


Figure 4.13.: B^+tree after the insertion of 41.5.

Example 4.3.2 *We again consider the example from the beginning after the deletion of* 13.2 *(Fig. 4.12). The value* 41.5 *has been inserted to the circular array and hence must be added to the* B^+*tree. The new tree after the insertion of the new measurement is illustrated in Figure 4.13. This measurement belongs to a leaf node which is already full. Hence, the leaf is split and the search key is passed to the parent and we find out that the parent is not full yet. Therefore, the leftmost key of the newly created leaf is inserted to the parent. Finally, the parent contains the search keys* 37.6*,* 40.3 *and* 43.1*.*

4.4. Neighborhood

The sortedAccess($N(q_{i,j}), T$) method in Algorithm 11 uses the B^+tree to return the time point $t \notin T$ such that $|q_{i,j} - s(t)|$ is minimal, given a neighborhood ($N(q_{i,j})$. But before a neighborhood can grow it has to be initialized. The neighborhood initialization is described in Section 4.4.1.

4.4.1. Initialize a Neighborhood

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

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

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

A neighborhood $N(q_{i,j})$ is defined around each value $q_{i,j}$ of a query pattern $Q(\bar{t})$, 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. In total $d \times l$ neighborhoods are initialized. Thus, for every time series r in query pattern $Q(\bar{t})$, l neighborhoods are initialized. 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 if j is equal to the pattern length, the latest time point in the query pattern is meant.
Initially the neighborhood position is set to the position of $q_{i,j}$ in the B^+tree. Therefore, we find the leaf that contains $q_{i,j}$ using Algorithm 6. Since $leftPos^-$ and $rightPos^+$ represent linked list elements we search qi, j in the linked list at the value position with the smallest number, such that $q_{i,j} \leq leaf.K_i$. The pattern length l is the upper bound for the search because the query pattern value qi, j is at least at position $\bar{t} - l$ in the circular array. Hence, it must be in the l newest time points in the linked list. We traverse the linked list backwards until the element time point is equal to the time point of $q_{i,j}$. After we place the $leftPos^-$ and $rightPos^+$ to the found linked list element and return neighborhood $N(qi, j)$

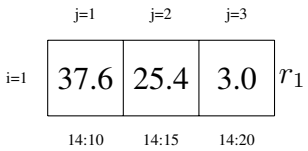
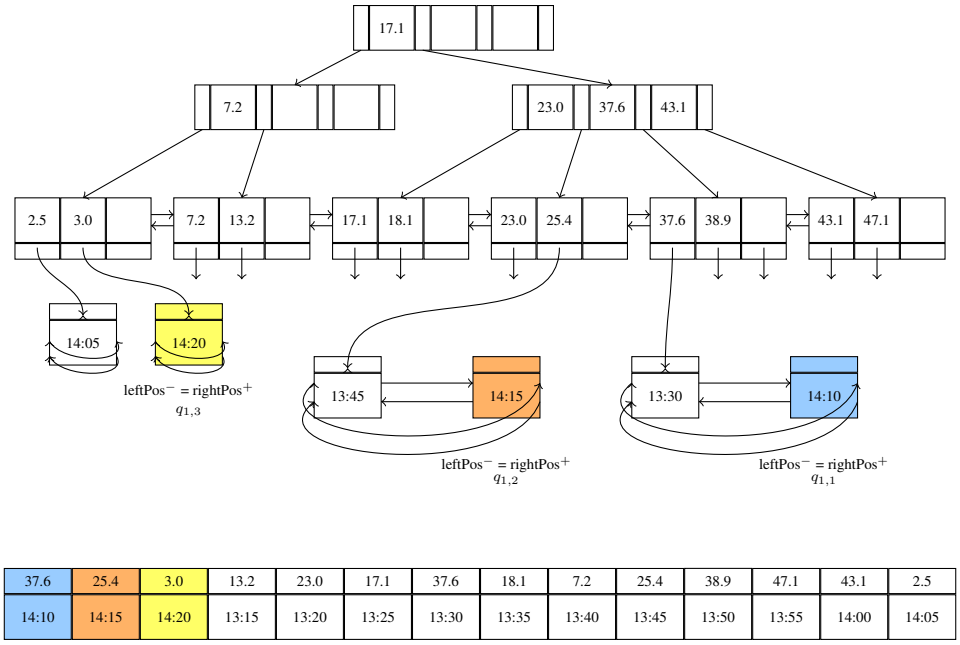


Figure 4.14.: Query Pattern $Q(\bar{t})$ of length $l = 3$ and $d = 1$ reference time series.



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

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

Example 9 *If $d = 1$ just one time series included in query pattern $Q(\bar{t})$. Figure 4.14 illustrates an example of a query pattern with $d = 1$. In total l neighborhoods are initialized for time series r_1. Figure 4.15 shows the circular array and the B^+tree for time series r_1 after the initialization. The coloured linked list elements represent the $leftPos^-$ and $rightPos^+$ of the newly initialized neighborhoods.*

Algorithm 10: NewNeighborhood($tree, q_{i,j}, t, j, l$)

Input: Tree $tree$, the query pattern value $q_{i,j}$ and its time point t and position j and the pattern length l

Output: The initialized neighborhood $N(q_{i,j})$

```
N.key ← q_{i,j}
N.j ← j
N.l ← l
leaf ← FindLeaf(tree, q_{i,j})
i ← smallest number such that q_{i,j} ≤ leaf.K_i
e ← leaf.pointers[i]
//Upper Bound: The value is at most pattern length away form first list value
maxSteps ← l
while e.time ≠ t and maxSteps ≠ 0 do
    //go from newest value back towards oldest
    e ← e.prev
    maxSteps−−
end
N.leftPos ← set position to e
N.rightPos ← set position to e
return N(q_{i,j});
```

4.4.2. Sorted Access

After the initialization of the $l \times d$ neighborhoods the k most non-overlapping patterns are calculated. Thus, the sortedAccess($N(q_{i,j}), T$) operation is executed until the k patterns are retrieved. Sorted access searches the time point for the next most similar value to a query pattern cell $q_{i,j}$ of query pattern $Q(\bar{t})$. The Algorithm 11 presents the sorted access method. The neighborhood $N(q_{i,j})$ is expanded until such a value is retrieved. Some time points may have to be skipped because the time point anchored a pattern already and therefore not need to be considered again. The next unseen most similar value to $q_{i,j}$ is either at position t^- or t^+, which represent the time points to the direct left or right of $leftPos^-$ and $rightPos^+$, respectively. If $|s_i(t^-) - q_{i,j}| \leq |s_i(t^+) - q_{i,j}|$ $leftPos^-$ is decremented to $prev(leftPos^-)$ and $t = t^-$. $prev(leftPos^-)$ searches for the next previous measurement time point ($next(rightPos^-)$ vice versa). If $leftPos^-$ has a predecessor in the linked list of a leaf value v, which is not the tail, $prev(leftPos^-)$ is set to the predecessor, thus $prev(leftPos^-)$ is still in the same linked list. If there is no appropriate predecessor in the linked list, $prev(leftPos^-)$ is placed to the tail of the linked list associated to the leaf value on the left side next to v. If there is no value to the left side in the same leaf, the rightmost value in the left leaf sibling is used.

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 $q_{i,j}$

leftPos$^-$ $\leftarrow$ N.leftPos
rightPos$^+$ $\leftarrow$ N.rightPos
while $t^- \neq$ *NIL* **and** *(*$t^- - j + l$*)* $\in T$ **do**
  leftPos$^-$ $\leftarrow$ prev(leftPos$^-$)
end
while $t^+ \neq$ *NIL* **and** *(*$t^+ - j + l$*)* $\in T$ **do**
  rightPos$^+$ $\leftarrow$ next(rightPos$^+$)
end
if $t^- \neq$ *NIL* **and** $t^+ \neq$ *NIL* **then**
  if $|r_i(t^-)$ - $q_{i,j}| \leq |r_i(t^+)$ - $q_{i,j}|$ **then**
    leftPos$^-$ $\leftarrow$ prev(leftPos$^-$); $t \leftarrow t^-$
  else
    rightPos$^+$ $\leftarrow$ next(rightPos$^+$); $t \leftarrow t^+$
  end
else if $t^- \neq$ *NIL* **then**
  leftPos$^-$ $\leftarrow$ prev(leftPos$^-$); $t \leftarrow t^-$
else if $t^+ \neq$ *NIL* **then**
  rightPos$^+$ $\leftarrow$ next(rightPos$^+$); $t \leftarrow t^+$
else
  return NIL
end
return t

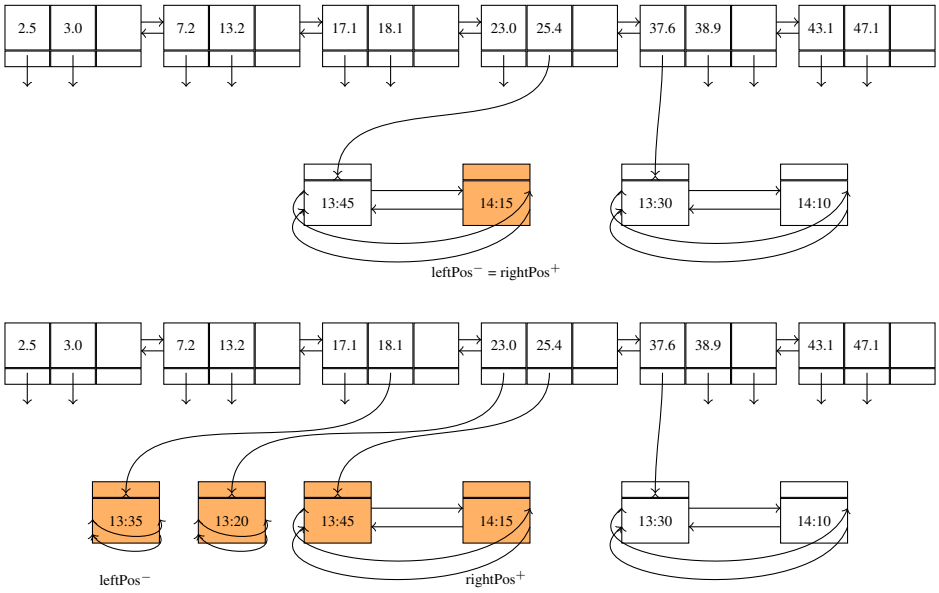


Figure 4.16.: Status of $N(q_{1,2})$ after growing 3 times.

Example 10 *Figure 4.16 shows how the neighborhood $N(q_{1,2})$ is expanded. We assume that the other neighborhoods $N(q_{1,1})$ and $N(q_{1,3})$ have 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 sortedAccess$(N(q_{i,j}), T)$ is executed 3 times the resulting neighborhood is illustrated in Figure 4.16.*

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. □

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. □

Circular Array Operations

Lemma 3 *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. □

Lemma 4 *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 calculated using the $lastPos$ and the time interval, it takes $O(1)$ time. □

B^+tree Operations

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

Since we have no disk I/O operation the parameter n has an influence on our complexity. For every node visited we pay $O(n)$ time to find the position of a value. The parameter $|W|$ has an influence as well since there are at most $|W|$ keys in B^+tree leaves (provided that there are no duplicate values).

Lemma 5 *Finding a leaf in the B^+tree takes at most $O(n \times \log_n |W|)$ time.*

Proof: The nodes are traversed recursively downwards from the root of the tree to its leaves. In the worst case, the number of nodes that must be traversed is $log_n|W|$. This leads to $log_n|W|$ complexity, where $|W|$ is the maximum total number of keys in the leaves and n is representing the maximum number of pointers in an inner node.
Additionally, you pay for every node that is visited $O(n)$ time to find the index position to the child node. Hence, the total complexity to find a leaf takes $O(n \times \log_n |W|)$ time. □

Lemma 6 *The insertion of a measurement to the B^+tree takes $O(n \times \log_n(|W|))$ time.*

Proof: Before a measurement can be inserted to the tree the appropriate leaf has to be found. Lemma 5 states that this takes $O(n \times \log_n(|W|))$ time. The addition of a measurement can cause three different cases as explained in Section 4.3.3.

- (The key already exists) Lemma 1 states that a measurement can be inserted in $O(1)$ time to the linked list.
- (The node has still room and the measurement is inserted to the node) This causes that the insertion place has to be found which takes at most $O(n)$ time, since the keys have to remain ordered.
- (The node is already full and the node has to split) To split the node it takes $O(n)$ time. Because the keys have to remain ordered, the insertion point is found by traversing the keys. After the keys are distributed to both nodes in $O(n)$ time. If the tree is recursively traversed upwards, at most $\log_n |W|$ nodes are split which results in a total time complexity of $n \times \log_n |W|$.

Since the time complexity to find a leaf is $O(n \times \log_n |W|)$ which is equal to the time complexity of a redistribution, the overall time complexity of an insertion is $O(n \times \log_n |W|)$. □

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

Proof: Before a measurement is deleted from the tree the appropriate leaf has to be found. Lemma 5 states that this takes $O(n \times \log_n |W|)$ time. The deletion of a measurement can cause four different cases as explained in Section 4.3.2.

- (The measurement time point is a linked list element which contains multiple elements) Lemma 2 states that a measurement can be deleted in $O(1)$ time from the linked list. Hence, duplicates have no influence on the complexity of a deletion.

- (The node is at least half-full) If the node has still enough keys the entry is searched in $O(n)$ time by traversing the node keys and then is deleted.
- (The values in a node are redistributed) The redistribution is done by moving keys and pointers. At most n keys have to be shifted which cost $O(n)$ time.
- (The node is merged with a sibling) To merge two nodes the keys and pointers have to be moved which costs at most $O(n)$ time. If the nodes are traversed upwards, there are at most $O(\log_n |W|)$ nodes to merge. Hence, the time complexity of the merge is $O(n \times \log_n |W|)$.

Since the time complexity to find a leaf is $O(n \times \log_n |W|)$ is equal to the time complexity of merge, the overall runtime complexity of deletion is $O(n \times \log_n |W|)$. □

Lemma 8 *The shift*$(tree, array, \bar{t}, v)$ *takes* $O(n \times \log_n |W|)$ *time.*

Proof: The total time complexity of a shift depends on the following operations:

- The update of the circular array takes $O(1)$ as stated in Lemma 3.
- The deletion of a measurement to the B^+tree takes $O(n \times \log_n |W|)$ time.
- The insertion of a measurement to the B^+tree takes $O(n \times \log_n |W|)$ time.

$O(n \times \log_n |W|)$ dominates the overall time complexity, this leads to a time complexity of $O(n \times \log_n |W|)$ for the shift operation. □

Neighborhood Operations

The newNeighborhood$(tree, q_{i,j}, t, j, l)$ operation searches a specific measurement. Therefore, it cannot just take the tail or head position in a linked list like the insertion or deletion method. We initialize $l \times d$ neighborhoods, one for each value in a query pattern $Q(\bar{t})$. Thus, l neighborhoods in every time series r which are part of the pattern. The *newNeighborhood* method always is executed at the l newest measurements in an involved time series because the query pattern is anchored at the newest time point $\bar{t}$. 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 search of a specific time point in a linked list for newNeighborhood*$(q_{i,j}, t, j, l)$ *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 the l newest measurements in the circular array for s have the same value and thus, are stored in the same linked list in the B^+tree. Therefore, l list elements have to be traversed to find the specific measurement time point, starting from the tail and scanning backwards. □

Lemma 10 *The initialization of a neighborhood in time series s takes $O(l + (n \times \log_n |W|))$ time.*

Proof: The initialization of the neighborhood needs to search a specific measurement in the B^+tree. Hence the complexity to find the measurement leaf is equal to the find leaf operation defined in Lemma 5. At most l linked list elements are traversed as stated in Lemma 9. This leads to a total complexity of $O(l + (n \times \log_n |W|))$. □

Lemma 11 *The sortedAccess$(N(q_{i,j}), T)$ takes at most $O(|T|)$ time.*

Proof: If the timeset contains $|T|$ time points and since at most $|T|$ time points have to be skipped the sortedAccess$(N(q_{i,j}), T)$ takes at most at most $O(|T|)$ time. □

5.2. Space Complexity

Lemma 12 *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 13 *The space complexity of a B^+tree is $O(|W|)$.*

Proof: Each measurement is stored once in the tree. In the worst case, all values are unique and $|W|$ values are stored in leaves which cause a space complexity of $O(|W|)$. The number of keys in inner nodes and the root is always smaller than the number of keys in the leaf level. Hence, the space complexity of the leaf level determines the overall space complexity. □

6. Experimental Evaluation

This Chapter describes our experimental setup, the experiments and their results. We evaluated the running time of the shift, the sorted access and the newNeighborhood operation with different parameters. We did not include random access, because its time complexity is constant.

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. The window size $|W|$ is set to 3 years ($525'600$). The data set contains values randomly chosen between 0 and $10'000$ and duplicates are possible.

6.2. Runtime

6.2.1. Shift

Increased Tree Node Size

Aim This experiment tries to illustrate the effect of an increased tree node size $n - 1$ on the shift operation. Besides, it wants to show that using a B^+tree was an appropriate choice.

Method To evaluate the influence of the tree node size on the runtime of the shift operation we executed the shift with values of the dataset introduced in Section 6.1. For each tree node size represented in Figure 6.1 we made $3 \times 1'000'000$ shifts to eliminate outliers.

Variables We measure the runtime of the algorithm with different tree node sizes $n - 1$. Therefore, the window size $|W|$ remains fixed to 3 years. The experiment starts with a tree node size $n - 1 = 1$.

Prediction We expect that the tree node size influences the runtime as we described in Lemma 8, which is nearly linear. Besides, we expect that the graph shows a minimum at a tree node size slightly larger than 1. Otherwise, e.g. a red-black tree might have been a better choice because red-black tree has a tree node size of 1[4].

Results Figure 6.1 shows that the parameter n indeed has an influence on the running time. We observe that the algorithm had the best running time with a tree node size around 11. A bigger or smaller tree node size decelerate the runtime. After the minimum is achieved the runtime is nearly linear with an increasing tree node size.

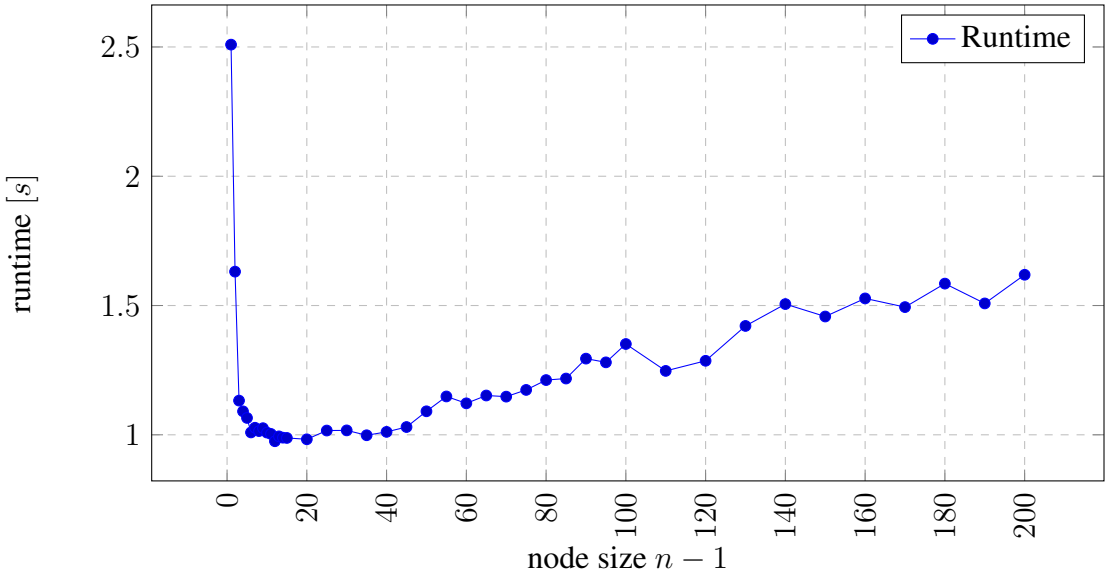


Figure 6.1.: Shift operation with increasing values for $n-1$.

Increased Window Size

Aim This experiment attempts to illustrate the effect of an increased window size $|W|$ on the performance of the shift operation.

Method To evaluate the influence of window size $|W|$, we executed the shift over a different dataset than in the first experiment. This dataset has the same size as the dataset described in Section 6.1 but contains values randomly chosen between 0 and $100'000$, thus 10 times more distinct values are possible. The dataset is modified because the insertion of a duplicate value does not enlarge the height of the B^+tree and therefore should have no effect on the runtime of the shift with an enlarged window size $|W|$. For each window size $|W|$ we made $3 \times 1'700'000$ shifts to keep the outliers as small as possible.

Variables We measure the runtime of the algorithm with different window sizes $|W|$. Therefore, the tree node size remains fix and is set to 11 because the runtime for this tree node size was the minimum in the experiment illustrated in Figure 6.1.

Prediction We expect that the window size influences the runtime as we described in Lemma 8. But we except that the window size $|W|$ only influences the runtime until no additional distinct values arrive. Hence, we expect an increasing runtime until the window has

a size of $100'000$. We predict that a larger window size also causes fewer deletes. The circular array only has cheap list operations instead of expensive structural changes like the B^+tree. This reduces the runtime for a larger window $|W|$ for the first updates which do not delete a measurement from the B^+tree. To reduce this effect we make $1'700'000$ shifts instead of $1'000'000$.

Results Figure 6.2 shows that the parameter $|W|$ influences the runtime of a shift. As expected, the runtime increases until the window size has achieved a size around $|W| = 100'000$. Afterwards, it decreases slightly. This can be explained with the increased window size which results in fewer measurement deletions. Hence, a larger window size has two different effects on the runtime. On the one hand, it increases the height of the B^+tree which increases the runtime for $|W|$ up to $100'000$ and on the other hand, a larger window size for $|W| > 100'000$ significances less deletions from the B^+tree which decreases the runtime.

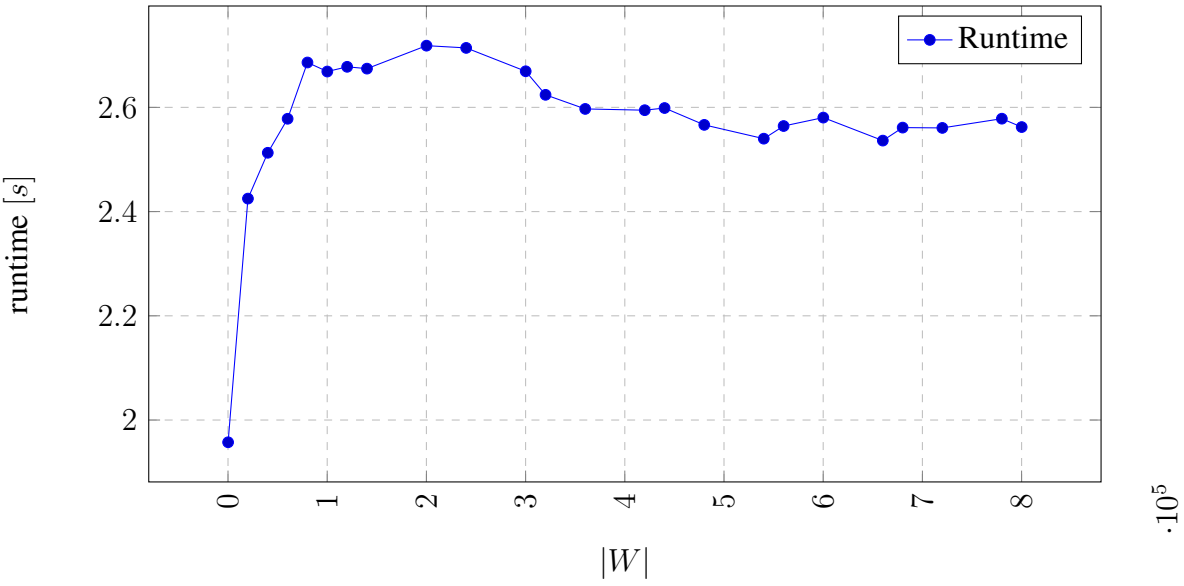


Figure 6.2.: Shift operation with increasing values for $|W|$.

6.2.2. New Neighborhood

Aim This experiment attempts to show the effect of an increased pattern length l on the new neighborhood operation. We try to show that the pattern length l has a very small influence to the runtime, which would support the usage of linked lists.

Method To evaluate the influence of the pattern length on the runtime of the new neighborhood operation, we executed the operation with values of a dataset with again $17'520'000$ values but only $1'000$ possible distinct values. This leads to many duplicate values. The window size $|W|$ is set to $17'520'000$ as well, to enhance the effect of duplicate values. We measure the runtime of the neighborhood initialization for a measurement at position $\bar{t} - l$ for

different pattern lengths. The B^+ tree is shifted $1'000'000$ times and then the initialization is executed.

Variables We set the tree node size to 11 and the window size to 6 years. This leads to more duplicate values which should increase the effect of l. We have a maximum pattern length of $9'000$.

Prediction We expect that the tree node size influences the runtime linearly as we described in Lemma 10. In our case, the influence is expected to be visible because the tree contains many duplicate values. The pattern length only has an influence if the value $q_{i,l}$ occurs multiple times in time window W.

Results As expected, the pattern length in our experiment has a visible influence on the runtime because the dataset contains many duplicate values. An increased pattern length leads to a worse runtime. However, we have many duplicates in our example. Since the values in our dataset are randomly chosen between 0 and $1'000$ there can be values that occur more often than others. This influences the runtime, since sometimes fewer linked list values have to be examined although l is higher.

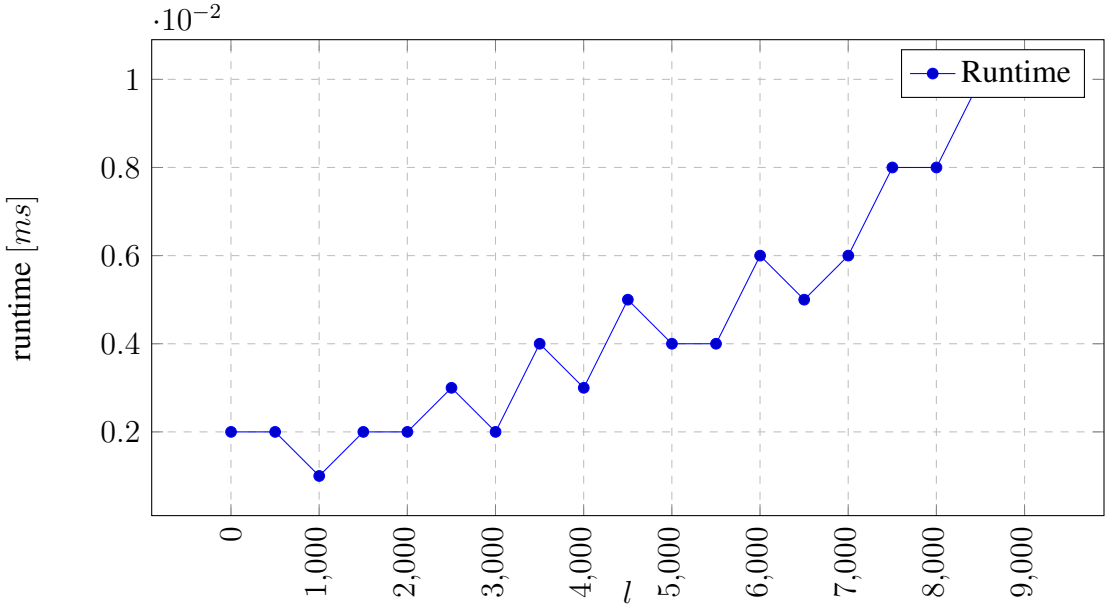


Figure 6.3.: Initialize neighborhood for a measurement at pattern cell $q_{i,l}$

6.2.3. Sorted Access

Aim This experiment attempts to illustrate the effect of timeset T on the runtime of the sorted access method.

Method To evaluate the influence of the timeset on the runtime of the sorted access operation, we execute the sorted access on a B^+tree filled with values of the dataset introduced in 6.1. The window size $|W|$ is set to 3 years and the tree node size to 11. The timeset is filled with time points that need to be skipped by the algorithm, hence $t^- - j + l \in T$. We have no time points $t^+ - j + l \in T$. The graph would show if there exists no more time points to the left side of $N(q_{i,j})$ and the condition would be renewed. At first, the timeset contains only five values that need to be skipped. After, every time 10 more values are added. We increase the number of time points $t^- - j + l \in T$, such that the while loop is executed every time $|T|$ times. The test is executed 3 times and the average runtime is calculated. The timeset for every execution is renewed, such that time points $t \in T$ are never skipped because they have been seen before. Furthermore, before each execution the neighborhood is initialized again.

Variables We measure the runtime of the sorted access algorithm with a fix tree node size and a fix window size $|W|$ but each time for a different timeset $|T|$.

Prediction We expect that the timeset T has a linear influence on the runtime as we described in Lemma 11, since for every skipped time point the sorted access is expected to be slower. Therefore, we expect a linear graph.

Results Figure 6.4 illustrates the results. We can see that the graph is nearly linear, which was expected. Although, the graph is not exactly linear this could be due to outliers.

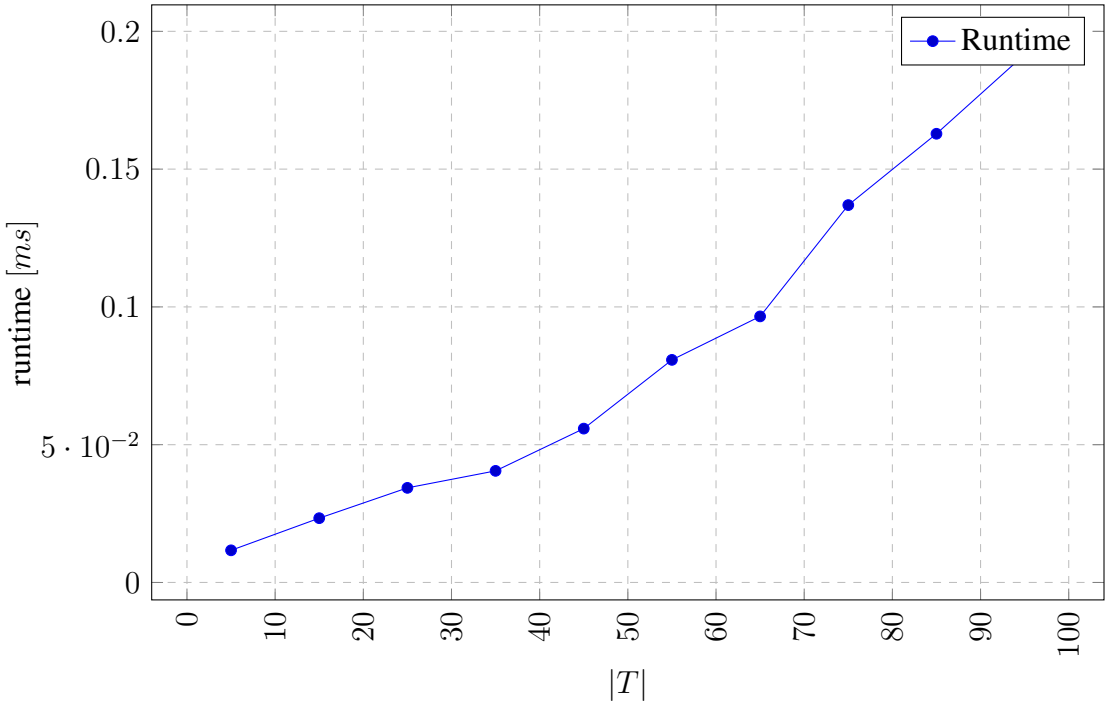


Figure 6.4.: Grow neighborhood with an increasing timeset size $|T|$.

7. Summary and Conclusion

We studied the requirements to keep a portion of a streaming time series in main memory and simultaneously provide efficient access possibilities to past time series data. The system we presented uses two different data structures to achieve efficient access: a B^+tree and a circular array. Random access is efficiently performed on a circular array and sorted access is efficiently performed on a B^+tree with leaves linked to the respective successor and predecessor. Furthermore, the thesis introduces a possibility to handle duplicate values in a B^+tree 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, in our context, retrieving a specific value in the linked list has an upper bound. We presented the algorithms to implement this system and analysed their runtime and space complexity. Finally, we evaluated the performance of the system to underpin our theoretical results.

Bibliography

[1] K. Wellenzohn, M. Böhlen, A. Dignös, J. Gamper, and H. Mitterer: *Continuous Imputation of Missing Values in Streams of Pattern-Determining Time Series*; Unpublished, 2016.

[2] Abraham Silberschatz, Henry F. Korth, S. Sudarshan: *Database System Concepts*; The McGraw-Hill Companies, Inc., New York, USA, pages 485-500, 2011.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein: *Introduction to Algorithms*; Massachusetts Institute of Technology, Massachusetts, USA, pages 236-241, 2009.

[4] Mark Allen Weiss: *Data Structures & Algorithm Analysis in Java*; Florida International University, Florida, USA, pages 460-468, 1999.

Appendices

A. Algorithms

A.1. B^+tree 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 ← tree.root.pointers[0]
8  |  newRoot.parent ← NIL
9  else
10 |  newRoot ← NIL
11 end
12 tree.root ← newRoot
```

Algorithm 13: 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 ← neighbor.numOfKeys
6  if node is no leaf then
7  |  neighbor.keys[neighborInsertionIndex] ← kPrime
8  |  neighbor.numOfKeys++
9  |  decreasingIndex ← 0
10 |  numOfKeysBefore ← node.numOfKeys
11 |  for i ← neighborInsertionIndex + 1, j ← 0; j < node.numOfKeys do
12 |  |  neighbor.keys[i] ← node.keys[j]
13 |  |  neighbor.pointers[i] ← node.pointers[j]
14 |  |  neighbor.numOfKeys++
15 |  |  decreasingIndex++, i++, j++
16 |  end
17 |  node.numOfKeys ← numOfKeysBefore - decreasingIndex
18 |  neighbor.pointers[i] ← node.pointers[j]
19 |  //All children must now point up to the same parent
20 |  for i ← 0; i < neighbor.numOfKeys + 1; i++ do
21 |  |  tmp ← neighbor.pointers[i]
22 |  |  tmp.parent ← 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 ← neighborInsertionIndex, j ← 0; j < node.numOfKeys do
28 |  |  neighbor.keys[i] ← 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)
```

Algorithm 14: 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 redestributed

```
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
```

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

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

Input: Tree $tree$, the node $oldInnderNode$ and the child node $childNode$, the index $index$ and in addition the key

Output: The inner node is split into two nodes

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

The CD-ROM contains the following content:

Abstract.txt The abstract of this thesis in English.

Zusfsg.txt The abstract of this thesis in German.

Thesis.pdf The digital copy of this thesis.

Experiments The experimental results in csv format and the datasets in text format.

B.1. CD-ROM