Department of Informatics, University of Zürich

**BSc Thesis**

# Implementing an Index Structure for Streaming Time Series Data

Melina Mast

Matrikelnummer: 13-762-588

Email: melina.mast@uzh.ch

August, 2016

supervised by Prof. Dr. Michael Böhlen and Kevin Wellenzohn

University of Zurich[UZH]

Department of Informatics

D
B
T G

# Acknowledgements

# Abstract

...

# Zusammenfassung

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# 1 Introduction

The data kept in main memory needs to be limited to a portion of the streaming time series. In order to be practical for an application like the financial stock market, the data that arrives in a defined time interval (e.g. every 2 minutes) needs to be completely processed until the succeeding data arises. The thesis presents a way to implement the described data structures after discussing the requirements. Furthermore, it documents the out coming experimental results. In the end of the thesis, in Chapter 8, the findings will be summarized and concluded.

## 1.1 Thesis Outline

# 2 Background

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

## 2.1 TKCM

A streaming time series is not always gapless. Due to sensor failures or transmission errors, values can get missing. Wellenzohn et al.[1] present an algorithm that defines a two-dimensional query pattern over the most recent values of a set of time series to efficiently impute missing values. The idea is to derive the missing value in a time series $s$ from the *k* most similar past pattern. Therefore, it determines for each *time series* $s$ a set of highly correlated *reference time series* which exhibit similar behaviour to the base station e.g. similar weather situations. Hence, TKCM is able to calculate an estimation of a missing value in streaming time series data.

**Definition 2.1.1** *Streaming Time Series. Let $S$ be a set $S = \{s_1, s_2, ...\}$ of streaming time series. The value of time series $s \in S$ at time* t *is denoted as $s(t)$. For base time series $s$, let $R_s = \langle r_1, r_2, ...\rangle$ be an ordered sequence of the time series $r_i \in S \setminus \{s\}$. The set of reference time series for $s$, $R_s^d$, at the current time $\bar{t}$ are the first $d$ time series in $R_s$ for which $r(\bar{t}) \neq$ NIL.*

**Definition 2.1.2** *Pattern. Let $R_s^d = \{r_1, ..., r_d\}$ be the ordered set of reference time series for a time series $s$. A $pattern$ $P(t)$ of length $l > 0$ over $R_s^d$ that is embedded at time $t$ is defined as a $d \times l$ matrix $P(t) = [p_{ij}]_{d \times l}$.*

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

**Definition 2.1.3** *Time window $W$. Let $W = [\underline{t}, \bar{t}]$ be a sliding window of length $|W|$. Time $\underline{t}$ stands for the oldest time point that fits into the time window and $\bar{t}$ stands for the current time point for which the stream produced a new value.*

Only the values in the time window $W$ are kept in main memory. However, we assume that all the time points $t < \bar{t}$ have a time series *s* that is complete. Hence, $\forall t < \bar{t} : s(t) \neq$ NIL since *s* contains imputed values if the real ones were missing.
TKCM must not only recover and impute missing values, but also process the newest arriving values efficiently. In order to do that, TKCM must provide an insertion method for new arriving values to insert the new value into a streaming time series $s$. Since the time window has a limited, given size $|W|$, an old value has to be deleted for each new arriving value. Provided that, the time series data in window $W$ is already completely filled.
TKCM must be able to handle duplicate values. Assume the time window contains 100 temperature values from the same weather station and every 5 minutes a new value arrives. It is likely that the same temperature value arrives multiple times.

## 2.2 Access Methods

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

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

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

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

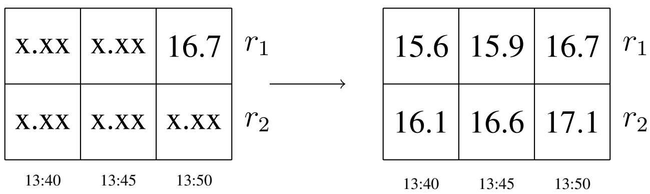



Figure 2.1: Pattern of length $l = 3$ and $d = 2$ reference time series.

TKCM initializes a set $T = \{\}$. The set is filled during execution with all time points *t* for which pattern $P(t)$ has already been compared to the query pattern $P(\bar{t})$. Using the sorted access mode, the algorithm loops through the cells $P_{ij}$, reading the next potential time point $t_s \notin T$. The time point $t_s \notin T$ is added to $T$. The time point $t_s$ has a corresponding pattern $P_{t_s}$ which is at least for one pattern cell similar to the query pattern $P_{\bar{t}}$.
The random access mode is used to look up the values that pattern $P(t)$ is composed of.

# 3 Problem Definition

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

## 3.1 Context

We make the following assumptions for our system:

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

## 3.2 Operations

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

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

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

# 4 Approach

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

## 4.1 Circular Array

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

size $|W|$

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

The circular array stores the data, containing all measurement time stamps and values, the size, the last update position and a counter, which counts the total number of measurements added to the array. The addition of a new measurement to the array is presented in Algorithm 3.

## 4.2 $B^+$ Tree

A $B^+$tree is able to execute range queries very efficiently, since the leaves of a $B^+$tree are ordered and linked. To perform the *neighbor*$(v, T)$ operation described in Section 3.2, the $B^+$tree we use has leaves linked in both directions. The Section 4.2.1 presents the structure of the $B^+$tree we used for the implementation.

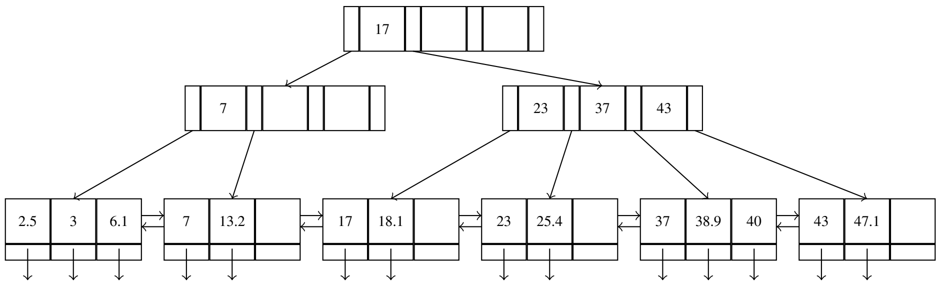



Figure 4.2: Example of a $B^+$tree

### 4.2.1 The Structure of the used $B^+$tree

The used $B^+$tree and the implementation is based on the book of Silberschatz et al.[4]. We introduce the most important properties of a $B^+$tree, for further information please refer to the book.
The difference between the traditional $B^+$tree and the $B^+$tree we use is on the one hand, that the leaves are linked to the succeeding as well as the preceding leaf to efficiently perform the *neighbor*$(v, T)$ operation and on the other hand, that the $B^+$tree is constructed to handle duplicate values. How our tree handles duplicate values is described in Section 4.3. The other properties the $B^+$tree are presented in the following.
All paths from the root to a leaf have the same length. This significances that the tree is always *balanced*. The balanced property ensures good performance for lookup, insertion and deletion. The $shift$ and the $neighbor$ operation are performed on the $B^+$tree.
There are three types of nodes that may exist in a $B^+$tree: the root, interior nodes and leave nodes. The parameter *n* determines the number of searchkeys and pointers in a node.

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

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

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

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

A node containing $m$ pointers $(m \leq n)$. For $i = 1, 2, 3, ..., m-1$, pointer $P_i$ points to the subtree that contains searchkey values less than $K_i$ and greater than or equal to $K_{i-1}$. Pointer $P_m$ points to the part of the subtree that contains those key values greater than or equal to $K_{m-1}$. The searchkeys in the leaves are sorted from left to right.

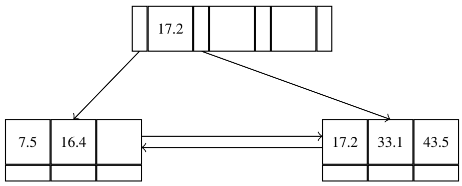



Figure 4.3: Left children keys < 17.2 and right children keys $\geq$ 17.2

A value in the time window $W$ can occur multiple times. Hence, the values are not unique. Since the values are used as searchkeys, the $B^+$tree must be able to handle duplicate values. Section 4.3 proposes our approach that allow to use duplicate values in a $B^+$tree.

## 4.3 Handling Duplicate Values

This Section presents a solution to allow duplicate values in a $B^+$tree. Further, the advantages of this approach are discussed and differences to other approaches are illustrated.

### 4.3.1 Associated doubly, circular Linked List

The idea of this method is to associate a doubly, circular linked list to the each key in a leaf node. Every key has an associated linked list, where the time points are stored. If a key occurs multiple times, the time points are simply added to the linked list. So instead of inserting the key again and using another block in the leaf, the new time point is inserted as a linked list value.

Associating a doubly, circular linked list that is interconnected in both directions is ideal for satisfying our requirements. The oldest value in the list, so the lowest time point, always is the one connected to the leaf key. Even though the doubly, circular linked list not really has an end and a beginning, we name the time point associated to the leaf the $firstlistvalue$. Since a shift operation on the circular array leads to a deletion of the oldest measurement in

the array, it is always the $firstlistvalue$ in the linked list. Also, a new measurement can be inserted without looping through the list. It is always added to the position before the oldest time point. We call the list value at this position the $lastlistvalue$. The Figure 4.4 illustrates that the oldest time point, here $14 : 15$, is connected to the tree and the newest time point $14 : 50$ is at the previous position. The addition and the deletion of a list value form a linked list containing multiple values is illustrated in Algorithm 1 and Algorithm 2.
The leaf nodes in our $B^+$tree also have pointers, the pointers to the associated linked list. But the number of pointers in leaf nodes is always equal to the number of searchkeys in the leaf. A pointer at position $i$ points to the doubly, linked list associated with the key at position $i$.

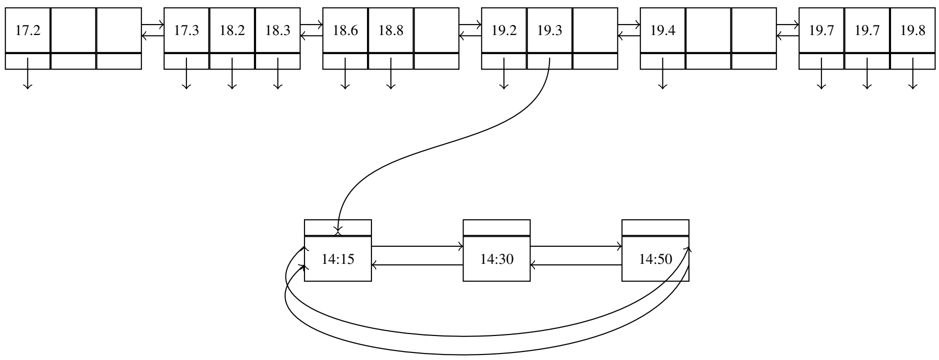



Figure 4.4: Doubly, circular linked list

The *neighborhood grow* operation searches a specific time point in the doubly, linked list. Therefore, it cannot just take the oldest or newest time point position like with an insertion or deletion. In the worst case the entire linked list would be searched for the specific time point. But since the *neighborhood grow* operation always is executed at the newest measurements in the circular array, we can give an upper bound, namely the pattern length. Therefore the worst case depends on the pattern length and on the distribution of the measurements which are starting points of the $neighbor$ method.

**Algorithm 1:** Add Time Point to Linked List

**Data**: Insertion node $node$, the time point to delete $t$ and the key associated to the Linked List $L$

**Result**: Linked List $L$ such that $t \in L$

```
begin
    for i ← 0 to numOfKeys in node do
        currentKey ← node->keys[i]
        if k == currentKey then
            break
        end
    end
    firstLV ← node->pointers[i]
    lastLV ← firstLV->prev
    insert t between firstLV and lastLV and link them
end
```

**Algorithm 2:** Delete Time Point from Linked List

**Data**: Node $node$ and index position $index$ for the position of the associated Linked List $L$

**Result**: Linked List $L$ such that $t \notin L$

```
begin
    //the list value linked to the node is always the oldest in the list
    timePointToDelete ← node->pointers[index]
    next ← timePointToDelete->next
    prev ← timePointToDelete->prev
    //next is the second oldest key
    leaf->pointers[index] ← next
    prev->next ← next
    next->prev ← prev
end
```

### 4.3.2 Alternative Approaches

A similar idea as in our approach is to add a associate list to the each key that occurs multiple times. So instead of inserting the key again and using another block in the leaf, the new time point is just inserted to its associated list. A new time stamp can be inserted to the end of the list in $O(1)$ and since the time window $W$ slides forward, the value that should be deleted first from the tree, normally, is at the first position in the list. Therefore, a value can be deleted in $O(1)$ from the list as well as with a doubly, circular linked list. But here the array cannot dynamically be extended since the array size must be reallocated with every additional time stamp.

A singly linked list uses less pointer than a doubly, circular list but since a insertion would cost $O(n)$ because a new value is always inserted to the end of the singly linked list and hence

all older time points in the list need to be checked. Therefore, a circular, linked list is more suitable for our requirements than a singly, linked list.
Another idea to handle duplicate values is to add additional leaves to the tree that do not have a parent node. As shown in Figure 4.5, the node containing the temperature value *18.3* had been split, since the values did no longer fit into one leaf. The value *18.4* would belong into the same leaf as *18.3* but there is no more space. Instead of splitting the leaf, the additional leaf without a parent is filled up. If e.g. a value *18.5* must be inserted the leaf without a parent must be split. The new leaf would again receive a parent and the old leaf including the duplicate values would stay parent-less. But unlike the doubly, circluar linked list approach searching a specific record may take long, depending on the number of duplicate temperature values to the left side of the record.

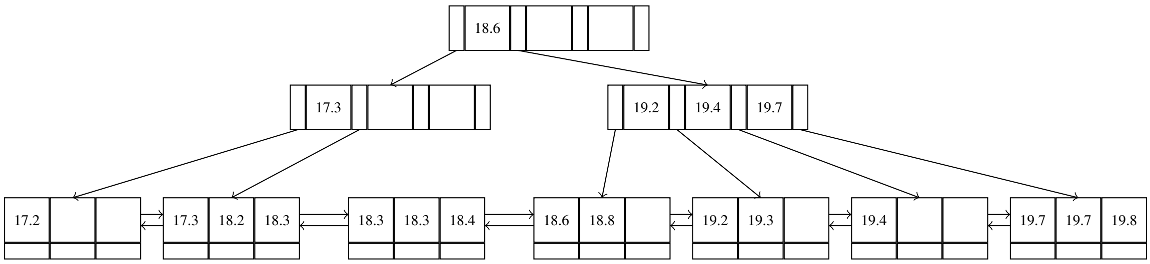



Figure 4.5: $B^+$tree with an additional leaf without a parent.

The book *Database Systems - The Complete Book* [3] presents an additional approach to handle duplicate values. The definition of a key is slightly different when allowing duplicate searchkeys. The keys the interior node $K_1, K_2, K_3, ..., K_n$ can be separated to *new* and old keys. $K_i$ is the smallest new key that is part of the sub-tree linked with the $(i+1)$st pointer. If there is no new key associated with the $(i+1)$st pointer, $K_i$ is set to null, as illustrated in Figure 4.6.

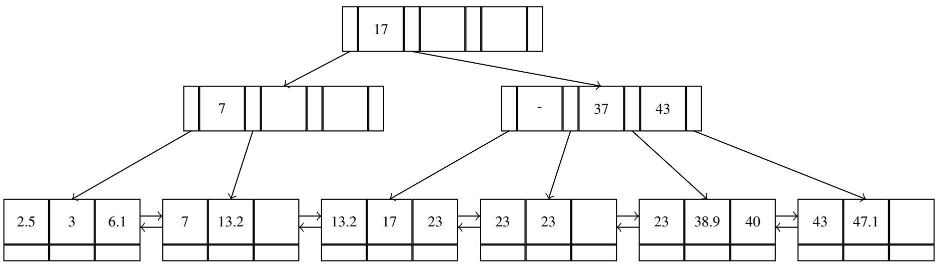



Figure 4.6: Duplicate handling proposed in [3].

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

# 4.4 Operations

The data in the circular array is updated with every new arriving measurement. Therefore, with every $shift$ execution the array is updated. The update method is illustrated in Algrithm 3. The $shift$ operation not only influences the array data but also the data in the $B^+$tree. Therefore, the deletion and insertion of a measurement in the tree is executed within the update of the circular array. The implementation of an insertion and deletion within a $B^+$tree is described in Section 4.4. The $lookup$ of a value is presented in Algorithm **??**.
Further, the neighbor$(v, T)$ method uses the $B^+$tree returning the time point $t \in T$ such that $|v - s(t)|$ is minimal, given a value *v*.
We assume we have the situation illustrated in Figure 4.11. 25.4 occurs two times, therefore two list values are part of the circular linked list associated to the key. Further, all keys in the leaves have an associated time point. Some associated linked lists are not illustrated to improve clarity. The size of the circular array is 14 and it is already full. This significates that for every new arriving measurement a value has to be added to and another one has to be deleted from the $B^+$tree.

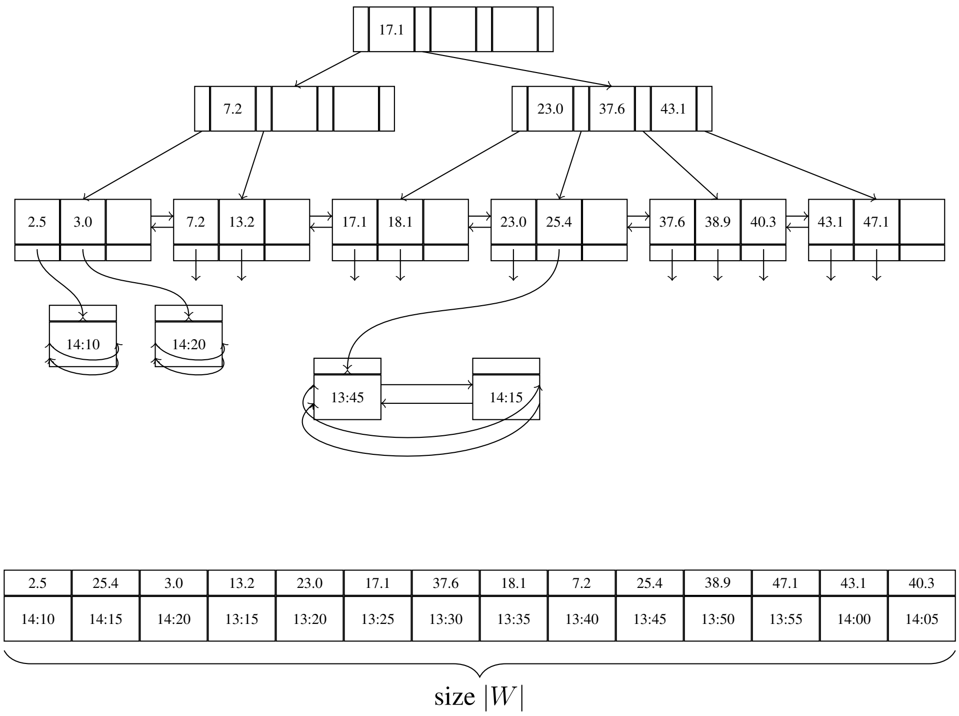



Figure 4.7: Start situation

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

#### Add a Value to the Circular Array

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

**Algorithm 3:** Update the Circular Array

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

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

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

Figure 4.8: Circular Array after the insertion of 41.5

### Searching in the $B^+$tree

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

**Algorithm 4:** Find Leaf

```
Data: Tree tree and the searchkey k
Result: The appropriate leaf for the searchkey k
begin
    curNode ← tree->root
    if curNode == NIL then
        return curNode
    end
    while curNode is not a Leaf do
        Let i = smallest number such that k ≤ curNode.K_i
        if no i then
            m ← last non-null pointer in the node
            curNode = curNode.P_m
        end
        else
            curNode = curNode.P_i
        end
    end
    return curNode
end
```

**Example 2** *Assume we want to find the key value* $13.2$*, since this is the one that has been overwritten by the newly arrived measurement. The function starts at the root of the tree, and goes through the tree until it reaches a leaf node that would contain the searched value. The current node is examined by looking for the smallest* $i$ *for which the searchkey value* $13.2$ *is greater or equal to. In this case the first pointer comes from the root at index position* $0$*, since* $13.2$ *is smaller than* $17.1$*. This is done by* $getInsertPoint$*. Then the new current node is set to the child node at pointer position* $0$*. Then the procedure is repeated until a leaf node is reached. This leaf node either contains* $13.2$*. Now after the leaf has been identified, the key can be deleted from the tree, which is explained in Section 5.*

### Delete a Value from the $B^+$tree

The circular array update Algorithm first deletes a value if necessary and then adds the new value to the tree. Since the deletion is executed first, we first present the deletion.
At the beginning, the entry for the measurement to delete is located. This is done by the $findLeaf$ method. Since our $B^+$tree accepts duplicates, it is afterwards checked if the associated linked list has multiple list values. If the list has multiple list values, the identified list value is deleted and the deletion is already finished.
But if the entries time point is the single value in the linked list the key and its belonging

linked list is deleted. Therefore, $DeleteEntry$ is called. After removing the entry, the node has either still enough keys or it needs to be merged with a sibling node or the values have to be redistributed to ensure that each node is at least half-full and hence have the minimum number of keys.
The minimum number of keys depends on the node properties. If the node is a leaf node the minimum number of keys in the node is $[(n-1)/2]$. If the node is an internal node the minimum number of keys is $[n/2]-1$ and the minimum number of pointers is $[n/2]$.
We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. If there is no left sibling the right sibling is selected. Once the node is deleted, we must also delete the entry in the parent node that pointed to the deleted node. Hence, we traverse the tree upwards until the $deleteEntry$ stops.
If merging is not possible, since the sibling and node together have more than the allowed $n$ pointers, the nodes have to be redistributed. We redistribute the keys, such that each node has at least $[n/2]$ child pointers. Therefore, we move the rightmost pointer from the left sibling to the under-full right sibling. Hence, we also need to move a key so that the newly added pointer is separable. This pointer is neither present in the left sibling nor in the right sibling. So we take a key from the parent node. As a result of deletion, a key value that is present in an interior node or in the root node of the $B+$tree may not be present at any leaf of the tree any more.

**Algorithm 5:** Delete

**Data**: Tree $tree$, measurement inforamtion: its time point $t$ and its key $k$
**Result**: Deletes measurement form $tree$

**begin**
  leaf ← findLeaf($tree$, $k$)
  keyPositionIndex ← right position index in leaf
  **if** *list on pointer at keyPositionIndex has multiple list values* **then**
    deleteFirstListValue(leaf, keyPositionIndex)
  **else**
    deleteEntry(tree,leaf, key on keyPostionIndex)
  **end**
**end**

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

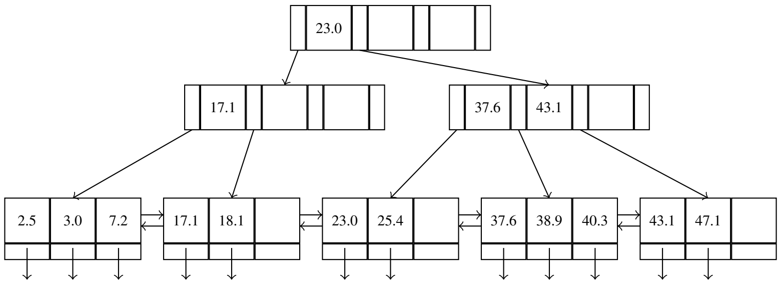



Figure 4.9: $B^+$tree after the deletion of $13.2$

**Algorithm 6:** DeleteEntry

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

**Result**: $k \notin node$

```
begin
    node ← remove k from node
    if node is the root then
        adjust the root
        return
    end
    if node is leaf then
        minNrOfKeys ← [(order − 1)/2]
    else
        minNrOfKeys ← [order/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 pointer position of left sibling in parent node
    if no left sibling then
        kIndex ← 0;
        neighbor ← node->parent->pointers[1]
    else
        kIndex ← neighborIndex;
        neighbor ← node->parent->pointers[neighborIndex]
    end
    //keyPrime is the value between pointers to node and neighbor in parent
    innerKeyPrime ← node->parent->pointers[kIndex]
    capacity = treeNodeSize
    if node is a leaf then
        capacity = treeNodeSize + 1
    end
    //Merge if both nodes together have enough space
    if (neighbor->numOfKeys + node->numOfKeys ) < capacity then
        mergeNodes(tree, node, neighbor, neighbourIndex, innerKeyPrime)
    else
        redestributeNodes(tree, node, neighbor, neighbourIndex, kIndex, innerKeyPrime)
    end
end
```

**Algorithm 7:** MergeNodes

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

**Result**: $node$ and its $neighbor$ are merged to one node

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

**Algorithm 8:** Redistribute

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

### Insert a Value to the $B^+$tree

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

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

The measurement is directly inserted to the leaf if the number of keys in the leaf is lower than the tree node size. The tree node size is determined by the parameter $n$. Hence, the tree node size is always $n - 1$. The measurement is inserted, so that the leaf keys are still ordered from left to right.

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

**Algorithm 9:** AddMeasurement

**Data**: Tree $tree$, the new timestamp $time$ and the new value $value$
**Result**: The tree includes the new value $value$

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

**Algorithm 10:** SplitLeaves

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

**Algorithm 11:** InsertIntoParent

**Data**: Tree $tree$, the newly created $node$ and the $oldnode$ and the key $k$ which is inserted to the parent

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

**begin**
    parent ← oldChild->parent;
    **if** *parent == NIL* **then**
        insertIntoANewRoot(tree, oldChild, newKey, newChild)
        return
    **end**
    //Find the parents pointer from the old node pointerPos ← pointer position from parent to $oldnode$
    //the new key fits into the node **if** *parent->numOfKeys < tree->nodeSize* **then**
        insertIntoTheNode(parent, pointerPos, newKey, newChild)
    **end**
    **else**
        splitAndInsertIntoInnerNode(tree, parent, pointerPos, newKey, newChild)
    **end**
**end**

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

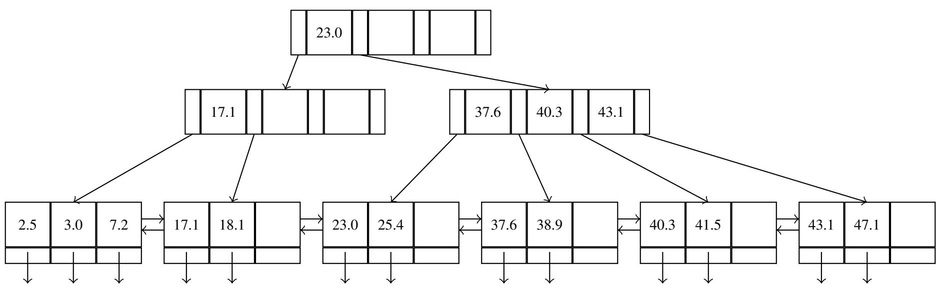



Figure 4.10: $B^+$tree after the insertion of $41.5$

### 4.4.2 Random Access: Lookup a Value

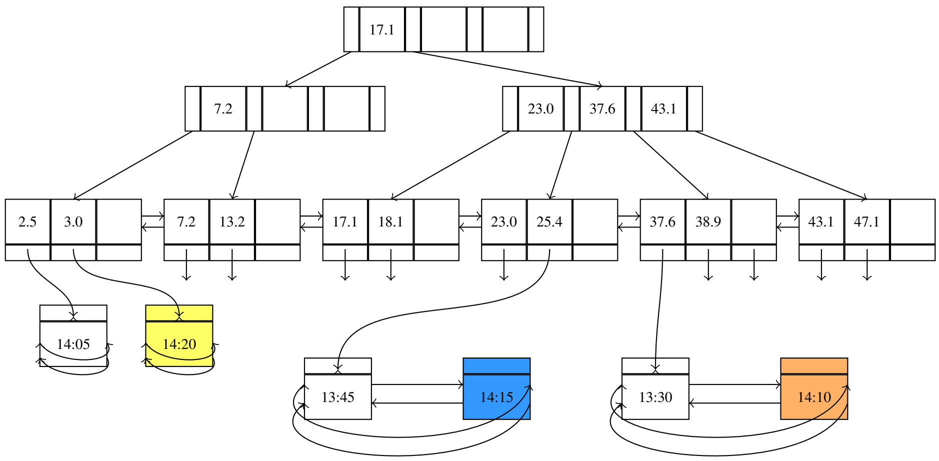
Due to the properties of a circular array the lookup of a value at time $t$ is very efficient. Since the position can be directly calculated without looping through the array by using the $TIMESTAMP\_DIFF$ representing the interval between two consecutive measurements. The last update point can be used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

**Algorithm 12:** Lookup

```
Data: The circular array array and the timestamp t
Result: Returns true if t ∈ array
begin
    if array->count == 0 then
        //no values yet
        return false
    end
    step ← (t - array->data[array->lastUpdatePosition].time)/TIMESTAMPDIFF
    if |step| < array->count then
        pos ← (((array->lastUpdatePosition +
        step)%array->size)+array->size)%array->size
        if array->data[pos].time == t then
            foundValue ← array->data[pos].value
            return true
        end
    end
    return false
end
```

### 4.4.3 Sorted Access: Neighbor

#### Initialize Neighborhood




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

Figure 4.11: Start situation

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

**Algorithm 13:** initializeNeighborhood

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

**Result**: The keys in the node and its neighbor, as well as the parents keys are redestributed

```
begin
    neighboorhood->key ← measurement->value
    neighboorhood->offset ← offset
    neighboorhood->patternLength ← patternLength

    leafNode ← findLeaf(tree, measurement->value)
    pointerIndex ← getInsertionIndex(leafNode, measurement->value)
    listValueOnThatKey ← leafNode->pointers[pointerIndex]

    //Upper Bound: The value is at most patternLength away form first list value
    maxSteps ← patternLength
    while listValueOnThatKey->timestamp ≠ measurement->timestamp && maxSteps ≠
    0 do
        //go from newest value back towards oldest
        listValueOnThatKey ← listValueOnThatKey->prev
        maxSteps–
    end
    neighboorhood->leftPosition ← set position to listValueOnThatKey
    neighboorhood->rightPosition ← set position to listValueOnThatKey

    return neighboorhood;
end
```

### Grow Neighborhood

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

**Algorithm 14:** NeighborhoodGrow

```
Data:
Result:
begin
    leftPos = self->leftPosition
    rightPos = self->rightPosition
    while t⁻ ≠ NIL && TimeSetContains(t⁻ −(offset + l) ∗ TIMEDIFF) do
        leftPos⁻ ← leftPosition⁻ −1
    end
    while t⁻ ≠ NIL && TimeSetContains(t⁻ −(offset + l) ∗ TIMEDIFF) do
        rightPos⁺ ← rightPosition⁺ +1
    end
    if t⁻ ≠ NIL && t⁺ ≠ NIL then
        if |r_i(t⁻) −self− > key| ≤ |r_i(t⁺) −self− > key| then
            leftPos⁻ ← leftPos⁻ − 1
            t ← t⁻
        end
        else
            rightPos⁺ ← rightPos⁺ + 1
            t ← t⁺
        end
        else if t⁻ ≠ NIL then
            leftPos⁻ ← leftPos⁻ − 1
            t ← t⁻
        else if t⁺ ≠ NIL then
            rightPos⁺ ← rightPos⁺ + 1
            t ← t⁺
        else
            return false
        end
    end
    return true
end
```

# 5 Complexity Analysis

### 5.0.1 Runtime Complexity

#### Circular Array Operations

**Update** $O(1)$

**Lookup**

#### $B^+$tree Operations

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

#### Neighborhood Operations

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

**Neighborhood grow**

### 5.0.2 Space Complexity

#### Circular Array

#### $B^+$tree

# 6 Evaluation

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

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

## 6.1 Experimental Setup

## 6.2 Results

## 6.3 Discussion

# 7 Related Works

# 8 Summary and Conclusion

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

# Bibliography

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

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

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

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