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 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 for example in finance, meteorology or sensor networks. All domains have applications that need to be fed continuously with the latest data e.g. the financial stock market or the weather information. But the processing of large volumes of time series data is impractical. Therefore, a system can only keep a limited size of data in main memory.
The data that is kept in main memory needs to be limited to just a portion of the streaming time series. Besides, in order to be practical for a 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.

2.1 TKCM

A streaming time series is not always gapless. E.g due to sensor failures or transmission error, values can get missing. Wellenzohn et al.[1] presents a two-dimensional query pattern over the most recent values of a set of time series to efficiently impute missing values. The two-dimensional query pattern $P_{\bar{t}}$ is defined with *l* reference time series on the spatial dimension and a time window of length *p* on the time dimension. The idea is to derive the missing value from the *k* most similar past pattern. Therefore, it determines for each *time series* a set of highly correlated *reference time series* which represent similar situations in the past e.g. similar weather situations. The value $\hat{s}(t)$ that is calculated as the average of the values $\{s(t)|t \in T_k\}$ will be imputed. Hence, TKCM is able to calculate an estimation of a missing value in streaming time series data.

TKCM must not only insert 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 the time window W. Since the time window has a limited, given size $|W|$, an old value has to be deleted for the new arriving value. Provided that, the oldest time *t* does no more fit into the time window because the window is already full.
Further, TKCM must be able to handle duplicate values. For example, if 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. Besides, the most similar base time values for a given value *v* should be efficiently found and returned.
These assumptions can be made for the implementation of the index structure for streaming time series data.

2.2 Access Methods

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 $P_{\bar{t}}$. Besides, TKCM initializes a set $T^* = \{\}$ that contains the *k* time points $t \in T$ that minimize the error $\delta(P_t, P_{\bar{t}})$. Therefore, $T^* \subseteq T$ is always true during execution.
TKCM uses two methods for accessing any time series $r \in S$, *random* and *sorted* access. 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 $r(t_s - o)$ is most similar to a given pattern cell $P_{\bar{t}}^{r,o}$. $t(s)$ is defined as:*

$$t_s = \underset{t_s \in W \setminus T}{\operatorname{argmin}} |r(t_s - o) - P_{\bar{t}}^{r,o}|$$

After T and T^* is initialized, TKCM iterates until set T^* contains the k time points t that minimize the difference $\delta(P_t, P_{\bar{t}})$.
Using the sorted access mode, the algorithm loops through the cells $P_{\bar{t}}^{r,o}$, 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 patter 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_s} is composed of. After each iteration a threshold τ is computed. The threshold τ is a lower-bound on the error $\delta(P_{t'}, P_{\bar{t}})$ for any time point t' that is yet unseen. Therefore, during the execution of the algorithm $\forall t' \in T : \tau \leq \delta(P_{t'}, P_{\bar{t}})$ is valid. Informally this significances that the lower-bound is always smaller or equal to the error between pattern $P_{t'}$ and query patter $P_{\bar{t}}$ for all time points t' that are elements of T. Once $\forall t \in T^* : \delta(P_t, P_{\bar{t}}) \leq \tau$ the algorithm terminates. At the end, $T^* = T_k$.

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.
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. Besides, consider 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 $s(t)$. 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.

3.1 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.

The $lookup$ operation is a random access method, while the $neighbor$ operation is a sorted access method.
Wellenzohn et al.[1] suggests 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 presented in in Chapter 4 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. 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. The data is assorted 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 number of measurements added to the array. The addition of a new measurement to the array is presented in Algorithm **??**.

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

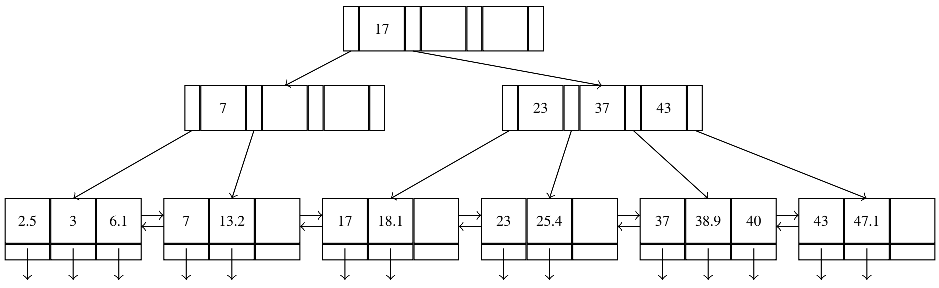

Figure 4.2: Example of a B^+ tree

4.2.1 The Structure of the used B^+ tree

The structure of the used B^+ tree is based on the book from Silberschatz et al.[4].
The difference to the traditional B^+ tree and our used B^+ tree 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 of our B^+ tree are presented in the following paragraph.

A B^+ tree is organized in blocks. All paths from the root to a leaf in a B^+ tree have the same length. This significances that the tree is always *balanced*. The balanced property ensures good performance for lookup, insertion and deletion that is why we use a B^+ tree. The $shift$ and the $neighbor$ operation are operations 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 search-keys and pointers in a node. The interior nodes can have maximum *n-1* search-keys and *n* pointers, pointing to its child nodes. Each interior node must have a minimum of $[n/2]$ pointers and can have maximum n pointers. A leaf node must have at least $[n/2]$ keys. And in our case at least $[n/2]$ pointers to doubly, circular linked lists containing the measurement time points. The root node is the only node that can contain less than $[n/2]$ pointers. The root node must have minimum one searchkey and two pointers to child nodes, unless the root node has no children and hence is a leaf node. A node containing m pointers $(m \leq n)$. For $i = 2, 3, ..., m - 1$, pointer P_i points to the subtree that contains searchkey values less than K_i and greater than or equal to K_{i-1}. Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1}. The searchkeys in the leaves are sorted from left to right.

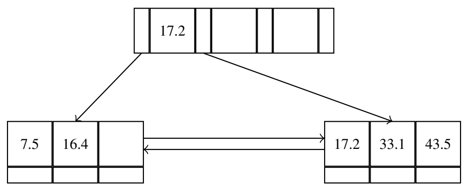

Figure 4.3: Left children keys < 17.2 and right children keys $\geq$ 17.2

A value in the time window W can occur multiple times. Hence, the values are not unique. Since the values are used as search-keys, 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 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. If a value key occurs multiple times, the new time point is added to the linked list. I 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 $firstlistvalue$. Hence, since a shift operation on the circular array leads to a deletion of the oldest measurement, is is always the $firstlistvalue$. 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 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 **??** and Algorithm **??**.

The leaf nodes in our B^+tree also have pointer. But the number of pointer is leaf nodes is always equal to the number of search-keys in the leaf. A pointer at position i points to the doubly, linked list associated with the key at position i.

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. Hence, In the worst case the entire 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.

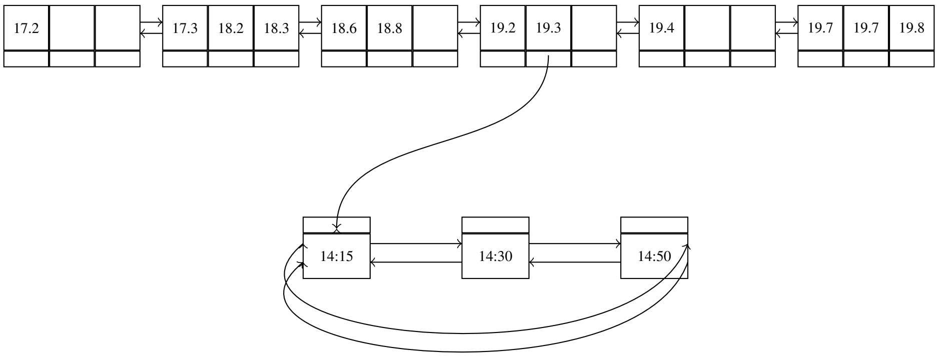

Figure 4.4: Doubly, circular linked list

Algorithm 1 Add a Value to the List

```
void addDuplicateToDoublyLinkedList(Node *node, timeStampT newTime, double
     duplicateKey){

   int i;
   //find right key with the associated list
   for(i = 0; i < node->numOfKeys; i++){
      double currentKey = node->keys[i];

      if(duplicateKey == currentKey){
         break;
      }
   }

   ListValue * newListValue = listValue_new(newTime);

   //first value in leaf
   ListValue * firstListValue = node->pointers[i];
   ListValue * lastListValue = firstListValue->prev;

   //update doubly linked pointers
   firstListValue->prev = newListValue;
   lastListValue->next = newListValue;

   //new Value is inseted to the last position in the list
   newListValue->prev = lastListValue;
   newListValue->next = firstListValue;

}
```

Algorithm 2 Delete a Value from the List

```
void deleteFirstListValue(Node * leaf, int index){

   //first value in leaf
   ListValue * firstListValue = leaf->pointers[index];

   ListValue * next = firstListValue->next;
   ListValue * prev = firstListValue->prev;

   //next is the second oldest key
   leaf->pointers[index] = next;
   prev->next = next;
   next->prev = prev;

   /*always the first list value must be the oldest list value and
    therefore the one that is deleted first*/
   free(firstListValue);
}
```

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. The Figure 4.5 illustrates the associated list.

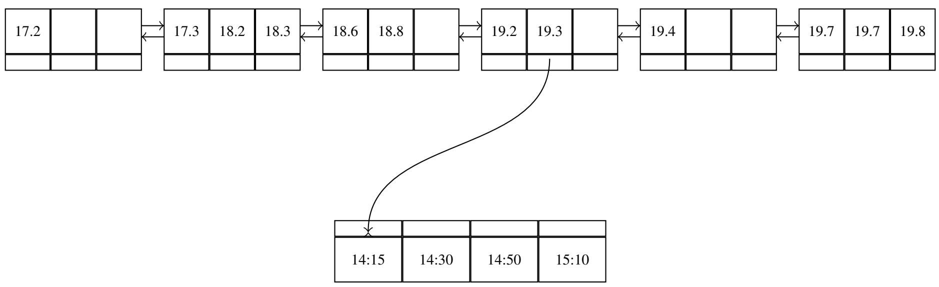

Figure 4.5: Associated List Approach.

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 extende since the array size must be reallocated with every additional time stamp.

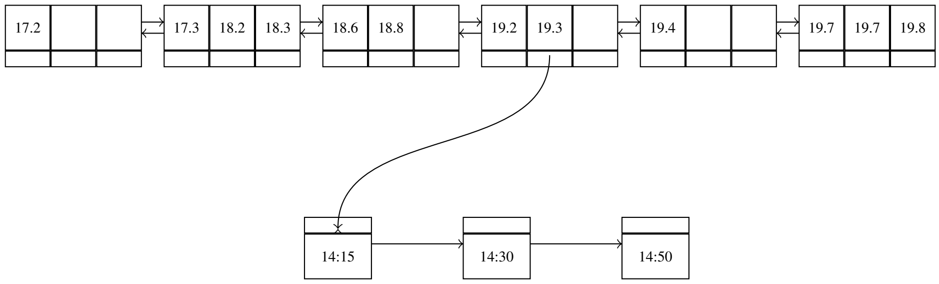

Figure 4.6: Singly linked list

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 4.7, 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.

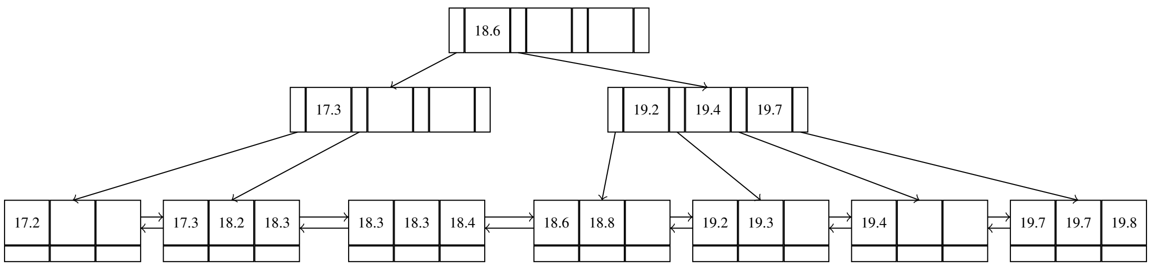

Figure 4.7: B^+ tree with an additional leaf without a parent.

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 search-keys. 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.

Example 4.3.1 *The example illustrated in Figure 4.8 illustrates the case, in which the interior node in the right sub-tree K_1 is set to null. The leaf node that pointer $(1+1)$ is associated*

with contains only duplicate key values which is indicated by setting the interior node K_1 to null. The search-key in the root node is set to 17 *because it is the lowest* new *key in the right sub-tree. Since* 13.2 *is already in the left sub-tree, the duplicate search-key in the right sub-tree cannot be the key in the root.*
If e.g. 14.2 *would have been added to the tree, the leaves must be reordered, since the* B^+ *tree property that all leaves are ordered from left to right would be hurt.*

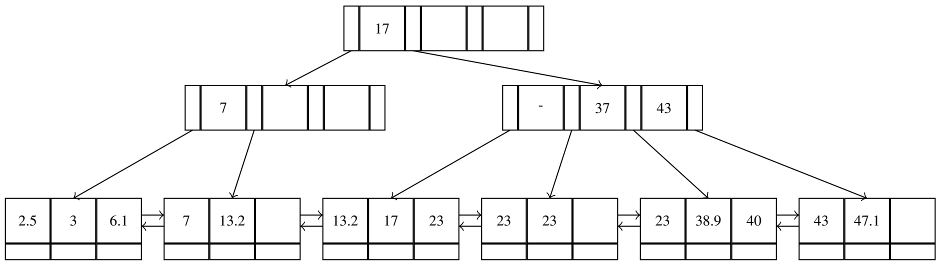

Figure 4.8: Duplicate handling proposed in [3].

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 **??**. 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 4.4.3.
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.

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

We assume we have the situation illustrated in Figure 4.12. 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.

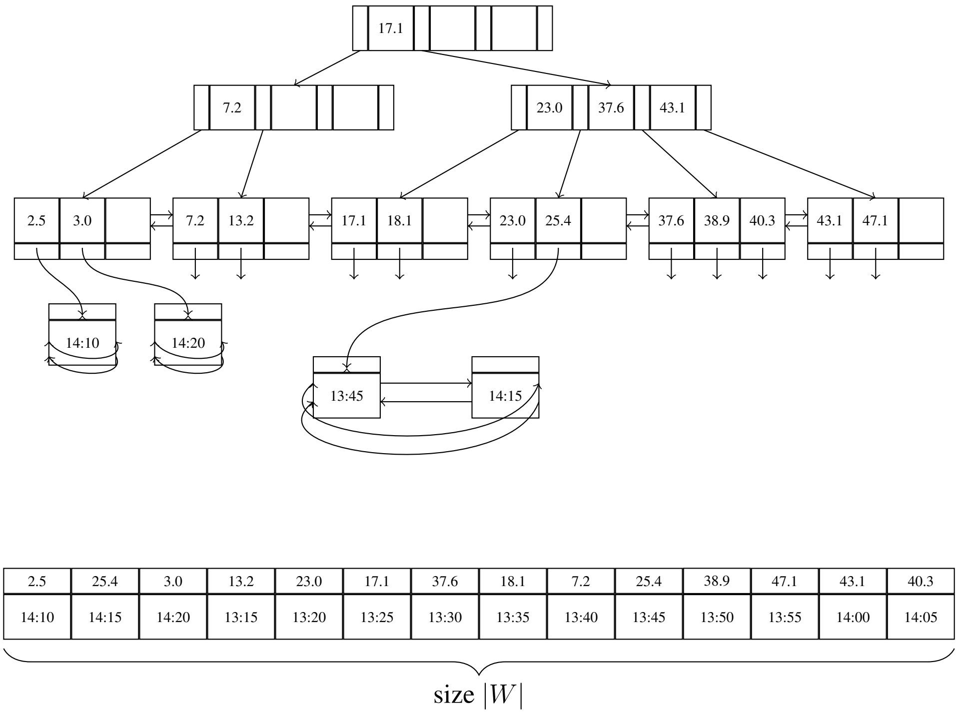

Figure 4.9: Start situation

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 Circular Array

```
void update_CircularArray(BPlusTree *tree, CircularArray *array,
    timeStampT newTime, double newValue){

   int newUpdatePosition = 0;
   //array is not full yet
   if(array->count < array->size){
      if(array->count != 0){
         newUpdatePosition = (array->lastUpdatePosition + 1) %array->size;
      }
      array->count++;
   }
   //array is full -> one value is inserted and one is deleted
   else{
      newUpdatePosition = (array->lastUpdatePosition + 1) %array->size;
      //delete measurement from tree
      delete(tree, array->data[newUpdatePosition].time, array->data[
         newUpdatePosition].value);
   }

   //update circularArray
   array->data[newUpdatePosition].time = newTime;
   array->data[newUpdatePosition].value = newValue;
   array->lastUpdatePosition = newUpdatePosition;

   addRecordToTree(tree, newTime, newValue);

}
```

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. Hence, 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 4.10: Circular Array after the insertion of 41.5

Delete 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. Therefore, we first present how the deletion works.

Algorithm 4 Delete

```
void delete(BPlusTree *tree, timeStampT time, double value){

   leaf = findLeaf(tree, value);
   int positionOfKey = -1;

   positionOfKey = findLeafKeyIndexAndSetboolIfMultipleListValues(leaf,
      value, &hasMultipleTimes);

   //key has duplicates -- delete first value ind doubly linked list
   if(hasMultipleTimes){
      deleteFirstListValue(leaf, positionOfKey);
   }
   else{
      //delete key form tree
      deleteEntry(tree, leaf, leaf->keys[positionOfKey], NULL);
   }
}
```

Algorithm 5 Delete Entry

```
void deleteEntry(BPlusTree *tree, Node *node, double toDelete, Node *pointer){

   // Remove key and pointer from this node
   node = removeEntryFromTheNode(tree, node, toDelete, pointer);

   if (node == tree->root){
      adjustTheRoot(tree);
      return;
   }
   minNumberofKeys = getMinNumberOfKeys(tree, node);

   //simple case - node has still enough keys
   if (node->numOfKeys >= minNumberofKeys){
      return;
   }
   //node falls below minimum - Either merge or redistribute.
   neighbourIndex = getNeighbourIndex(node);
   kIndex = getKindex(neighbourIndex);
   innerKeyPrime = node->parent->keys[kIndex];

   neighbor = getNeighborNode(node);
   capacityOfNode = setCapacity(tree, node);

   //Merge if both nodes together have enough space
   if ((neighbor->numOfKeys + node->numOfKeys ) < capacity){
      mergeNodes(tree, node, neighbor, neighbourIndex, innerKeyPrime);
   }
   //Redistribute
   else{
      redestributeNodes(tree, node, neighbor, neighbourIndex, kIndex, innerKeyPrime);
   }
}
```

node falls below minimum - Either merge or redistribute. Find the appropriate neighbor node with which to merge. Also finds the key in the parent between the pointer to the node and the pointer to the neighbor

Algorithm 6 Redestribute Nodes

```
void redestributeNodes(BPlusTree * tree, Node * node, Node * neighbor, int
    neighbourIndex, int kIndex, double kPrime){

   //node has a neighbor to the left. Pull the neighbor's last key-pointer
       pair over from the neighbor's right end to n's left end.
   if (neighbourIndex != -1) {
      if(!node->isLeaf){
         node->pointers[node->numOfKeys + 1] = node->pointers[node->
             numOfKeys];
      }
      for (i = node->numOfKeys; i > 0; i--) {
         node->keys[i] = node->keys[i - 1];
         node->pointers[i] = node->pointers[i - 1];
      }
      if (!node->isLeaf) {
         node->pointers[0] = neighbor->pointers[neighbor->numOfKeys];
         tmp = (Node *)node->pointers[0];
         tmp->parent = node;
         neighbor->pointers[neighbor->numOfKeys] = NULL;
         node->keys[0] = kPrime;

         //clear reason
         node->parent->keys[kIndex] = neighbor->keys[neighbor->numOfKeys
             -1];
      }
      else {
         node->keys[0] = neighbor->keys[neighbor->numOfKeys - 1];
         node->pointers[0] = neighbor->pointers[neighbor->numOfKeys - 1];
         node->parent->keys[kIndex] = node->keys[0];
      }
   }
   //node is the 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.
   else {
      if (node->isLeaf) {
         node->keys[node->numOfKeys] = neighbor->keys[0];
         node->pointers[node->numOfKeys] = neighbor->pointers[0];
         node->parent->keys[kIndex] = neighbor->keys[1];
      }
      else {
         node->keys[node->numOfKeys] = kPrime;
         node->pointers[node->numOfKeys + 1] = neighbor->pointers[0];
         tmp = (Node *)node->pointers[node->numOfKeys + 1];
         tmp->parent = node;
         node->parent->keys[kIndex] = neighbor->keys[0];
      }
      for (i = 0; i < neighbor->numOfKeys - 1; i++) {
         neighbor->keys[i] = neighbor->keys[i + 1];
         neighbor->pointers[i] = neighbor->pointers[i + 1];
         }
      }
      if (!node->isLeaf){
         neighbor->pointers[i] = neighbor->pointers[i + 1];
      }
   }
   //n now has one more key and one more pointer the neighbor has one
      fewer
   node->numOfKeys++;
   neighbor->numOfKeys--;
}
```

Algorithm 7 Merge Nodes

```
void mergeNodes(BPlusTree *tree, Node *node, Node *neighbor, int neighborIndex, double kPrime){

    //Swap neighbor with node if node is on the extreme left and neighbor is to its right.
    if (neighborIndex == -1) {
        tmp = node;
        node = neighbor;
        neighbor = tmp;
    }
    /*Starting point in the neighbor for copying keys and pointers from the other node.
    Recall that n and neighbor have swapped places
    in the special case of the node being a leftmost child */
    neighborInsertionIndex = neighbor->numOfKeys;

    //Append k_prime and the following pointer - append all pointers and keys from the neighbor
    if(!node->isLeaf)
    {
        neighbor->keys[neighborInsertionIndex] = kPrime;
        neighbor->numOfKeys++;

        int decreasingIndex = 0;
        int numOfKeysBefore = node->numOfKeys;

        for (i = neighborInsertionIndex + 1, j = 0; j < node->numOfKeys; i++, j++) {
            neighbor->keys[i] = node->keys[j];
            neighbor->pointers[i] = node->pointers[j];
            neighbor->numOfKeys++;
            //decreases numOfKeys
            decreasingIndex++;
        }
        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++) {
            tmp = (Node *)neighbor->pointers[i];
            tmp->parent = neighbor;
        }
    }
    // leaf node, 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.
    else
    {
        for (i = neighborInsertionIndex, j = 0; j < node->numOfKeys; i++, j++) {
            neighbor->keys[i] = node->keys[j];
            neighbor->pointers[i] = node->pointers[j];
            neighbor->numOfKeys++;
        }
        //relink leafs
        if(node->next != NULL){
            (node->next)->prev = neighbor;
        }
        neighbor->next = node->next;
    }
    deleteEntry(tree, node->parent, kPrime, node);
    destroyNode(node);
}
```

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. Hence, the tree after deleting* 13.2 *looks as follows:*

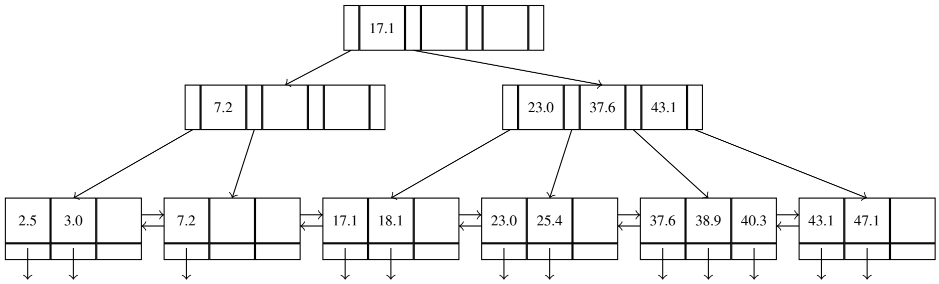

Figure 4.11: Start situation

Insert a Value to the B^+tree

With every circular array update a new measurement is added to the B^+ tree as shown in the Algorithm **??** at line 27.
We first find the leaf node in which the search-key value 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 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 stay equally ordered. If the search-key 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.

Algorithm 8 Add Measurement To Tree

```
void addRecordToTree(BPlusTree *tree, timeStampT time, double value){

    Node *leaf;

    //the tree does not exist yet --> create tree
    if (tree->root == NULL){
        newTree(tree, time, value);
        return;
    }

    //find the leaf to insert the record
    leaf = findLeaf(tree, value);

    //if duplicate -> insert to leaf as Doubly linked list value
    if(isDuplicateKey(leaf, time, value)){
        addDuplicateToDoublyLinkedList(leaf, time, value);
    }
    //new key
    else if(leaf->numOfKeys < tree->nodeSize){
        insertRecordIntoLeaf(tree, leaf, time, value);
    }
    else{
        //leaf must be split
        splitAndInsertIntoLeaves(tree, leaf, time, value);
    }
}
```

Algorithm 9 Find Leaf

```
Node * findLeaf(BPlusTree *tree, double newKey){
    int i = 0;

    Node * curNode = tree->root;

    if (curNode == NULL) {
        return curNode;
    }
    while (!curNode->isLeaf) {

        i = getInsertPoint(tree, curNode, newKey);
        //new lookup node
        curNode = (Node *)curNode->pointers[i];
    }
    //leaf found
    return curNode;
}
```

Algorithm 10 Split Leaf

```
void splitAndInsertIntoLeaves(BPlusTree *tree, Node *oldNode, timeStampT
    time, double firstValue){

    Node * newNode = Leaf_new(tree->nodeSize);
    int insertPoint, split, i, j;
    double *tempKeys = malloc((tree->nodeSize + 1)* sizeof(double));;
    void **tempPointers = malloc((tree->nodeSize +1)* sizeof(void *));

    insertPoint = 0;
    int nrOfTempKeys = 0;
    insertPoint = getInsertPoint(tree, oldNode, firstValue);

    //fills the keys and pointers
    for (i = 0, j = 0; i < oldNode->numOfKeys; i++, j++) {
        //pointers needs to be moved 1 position
        if (j == insertPoint){
            j++;
        }
        //the value where the new record is added is not been filled yet
        tempKeys[j] = oldNode->keys[i];
        tempPointers[j] = oldNode->pointers[i];
        nrOfTempKeys++;
    }
    //enter the record to the right position
    tempKeys[insertPoint] = firstValue;
    ListValue * newTime = listValue_new(time);
    tempPointers[insertPoint] = newTime;
    nrOfTempKeys++;

    newNode->numOfKeys = 0;
    oldNode->numOfKeys = 0;

    //find split point
    split = getSplitPoint(tree->nodeSize);

    //fill first leaf
    fillLeaf(oldNode, tempKeys, tempPointers, split);
    //fill second leaf
    fillLeaf(newNode, tempKeys, tempPointers, split + nrOfTempKeys);

    //update intermediate leaf pointers
    updateLeafPointers(oldNode, newNode);

    //the record to insert into the upper node
    double keyForParent;
    keyForParent = newNode->keys[0];

    //free allocated memory of pointers
    free(tempKeys);
    free(tempPointers);

    insertIntoParent(tree, oldNode, keyForParent, newNode);
}
```

Algorithm 11 Insert into Parent

```
void insertIntoParent(BPlusTree *tree, Node *oldChild, double newKey, Node
     *newChild){

   int pointerPositionToLeftNode;
   Node * parent = oldChild->parent;

   //new root
   if(parent == NULL){
      insertIntoANewRoot(tree, oldChild, newKey, newChild);
      return;
   }

   //Find the parents pointer to the old node
   pointerPositionToLeftNode = getLeftPointerPosition(parent, oldChild);

   //the new key fits into the node
   if (parent->numOfKeys < tree->nodeSize){
      insertIntoTheNode(parent, pointerPositionToLeftNode, newKey,
         newChild);
   }
   else{
      //split a node in order to preserve the B+ tree properties
      splitAndInsertIntoInnerNode(tree, parent, pointerPositionToLeftNode,
          newKey, newChild);
   }
}
```

Example 4.4.3 *We now consider an example of insertion in which a node must be split. The measurement which already has been inserted to the circular array belongs ot 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:*

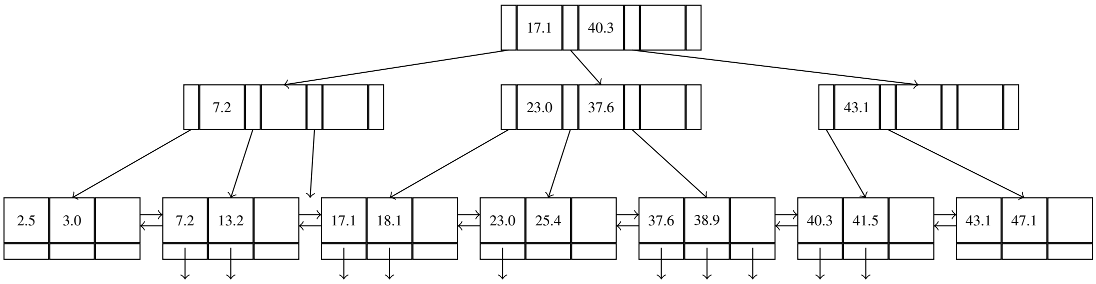

Figure 4.12: Start situation

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

```
bool lookup(CircularArray *array, timeStampT t, double *value){

   //array is empty
   if(array->count == 0){
      return false;
   }

   //steps from last timestamp to new timestamp
   int step = (int)(t - array->data[array->lastUpdatePosition].time)/
      TIMESTAMP_DIFF;

   //checks if array has enough values inserted for the necessary steps
   if(abs(step) < array->count){

      //chaining modulo for negative numbers
      int pos = (((array->lastUpdatePosition+step)%array->size)+array->
         size)%array->size;

      //value has been found
      if(array->data[pos].time == t){
         //set the pointer to the found value
         *value = array->data[pos].value;
         return true;
      }
   }
   //value was not found
   return false;
}
```

4.4.3 Sorted Access: Neighbor

Initialize Neighborhood

/* * 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 Initialize Neighborhood

```
Neighborhood *Neighborhood_new(BPlusTree *tree, Measurement *measurement,
    int patternLength, int offset){

   Node * leafNode;

   leafNode = findLeaf(tree, measurement->value);
   int pointerIndex = findLeafKeyIndex(leafNode, measurement->value);

   ListValue *listValueOnThatKey = leafNode->pointers[pointerIndex];

   //The value will be at most patternLength away in the doubly, linked
       list
   int maxSteps = patternLength;
   while(listValueOnThatKey->timestamp != measurement->timestamp &&
       maxSteps != 0){
      //go from newest value back towards oldest
      listValueOnThatKey = listValueOnThatKey->prev;
      maxSteps--;
   }

   setNeighborhoodPositions(newNeighborhood, leftNeighbourhoodPos,
       rightNeighbourhoodPos, pointerIndex, leafNode, listValueOnThatKey);

   return newNeighborhood;
}
```

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 Neighborhood Grow

```
bool Neighborhood_grow(Neighborhood *self, TimeSet *timeset, timeStampT *
    timestamp){

   bool neighborHoodHasGrown = true;
   //get the left neighbor of the measurement

   leftPos = getTMinusNeighborHoodPosition(leftPos);
   offsetMinusTime = getOffsetTime(self, leftPos);

   //get the right neighbor of the measurement
   rightPos = getTPlusNeighborHoodPosition(rightPos);
   offsetPlusTime = getOffsetTime(self, rightPos);

   //checks if offsetTime is in timestamp set
   while(leftPos.timeStampPosition != NULL && TimeSet_contains(timeset,
       offsetMinusTime)){
      leftPos = getTMinusNeighborHoodPosition(leftPos);
      offsetMinusTime = getOffsetTime(self, leftPos);

      //next time doesnt have to check this position again
      self->leftPosition = leftPos;
   }

   while(rightPos.timeStampPosition != NULL && TimeSet_contains(timeset,
       offsetPlusTime)){
      rightPos = getTPlusNeighborHoodPosition(rightPos);
      offsetPlusTime = getOffsetTime(self, rightPos);

      //next time doesnt have to check this position again
      self->rightPosition = rightPos;
   }

   if(leftPos.timeStampPosition != NULL && rightPos.timeStampPosition !=
       NULL){

      long tMinusdifference = fabs(leftPos.LeafPosition->keys[leftPos.
          indexPosition] - self->key);
      long tPlusdifference = fabs(rightPos.LeafPosition->keys[rightPos.
          indexPosition] - self->key);

      if(tMinusdifference <= tPlusdifference){
         self->leftPosition = leftPos;
         *timestamp = leftPos.timeStampPosition->timestamp;
      }
      else{
         self->rightPosition = rightPos;
         *timestamp = rightPos.timeStampPosition->timestamp;
      }
   }
   else if(leftPos.timeStampPosition != NULL){
      self->leftPosition = leftPos;
      *timestamp = leftPos.timeStampPosition->timestamp;

   }
   else if(rightPos.timeStampPosition != NULL){
      self->rightPosition= rightPos;
      *timestamp = rightPos.timeStampPosition->timestamp;
   }
   else{
      neighborHoodHasGrown = false;
   }
   return neighborHoodHasGrown;

}
```

4.5 Combination of the Data Structures

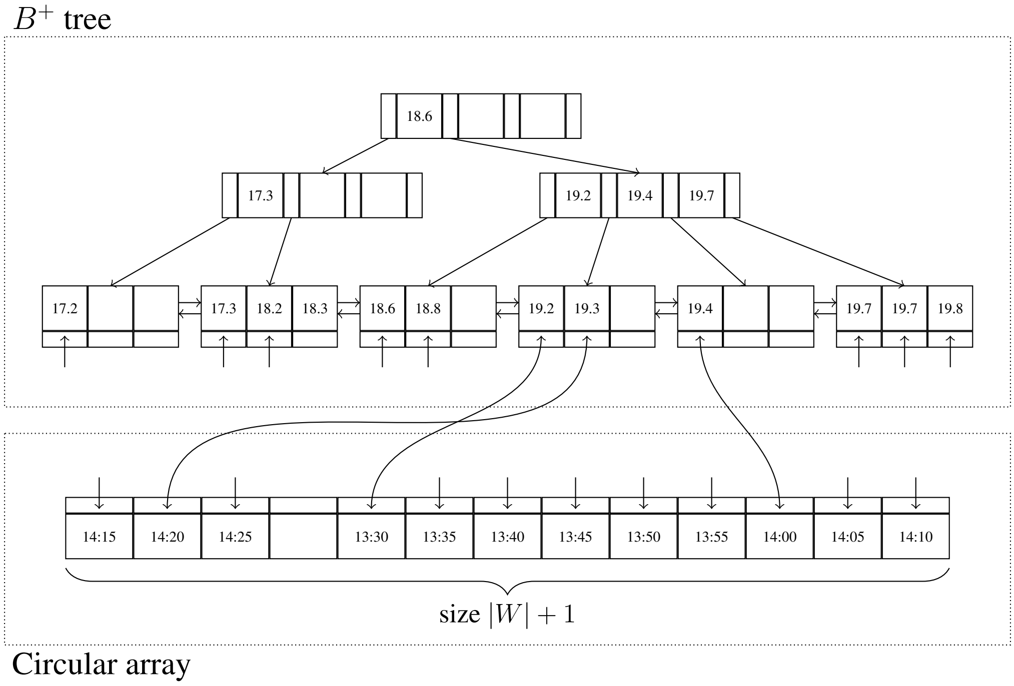

Figure 4.13: Proposed data structures in [1].

5 Complexity Analysis

5.0.1 Runtime Complexity

5.0.2 Space Complexity

6 Evaluation

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

"The conclusion (10 to 12 per cent of the whole research thesis) does not only summarize the whole research thesis, but it also evaluates the results of the scientific inquiry. Do the results confirm or reject previously formulated hypotheses? The conclusion draws both theoretical and practical lessons that could be used in future analyses. These lessons are to be embedded as 2 recommendations for the research community and for policy-makers (note: policy relevance instead of policy prescriptive). In addition, the conclusion gives insights for further research."

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-