Dynamic Sorting Algorithm Based on the B+ tree Data Structure

Weile TAN[1]

[1]The Hong Kong University of Science and Technology
[1]wtanae@connect.ust.hk

Abstract

Dynamic sorting is critical for applications requiring real-time data updates, such as stock trading, file management, and news pulication. This paper introduces the dynamic sorting problem, characterized by the need to maintain a sorted list as new data is continually inserted. We formally define the dynamic sorting problem as comprising two sub-problems: sorting an initial dataset and subsequently inserting new data while preserving order. Existing algorithms designed for static sorting are inadequate for the dynamic context due to inefficiencies in handling updates. To address these limitations, we propose a novel dynamic sorting algorithm, B+ Tree Sort, which utilizes a B+ tree data structure to efficiently manage sorted data and execute insertions in logarithmic time. Extensive experiments are conducted on both synthetic and real-world datasets. The results demonstrate that B+ Tree Sort significantly outperforms existing static sorting algorithms in terms of time efficiency for dynamic sorting problems.

Keywords: sorting algorithm, dynamic sorting, B+ tree, data structure.

1. Introduction

Ordered data is essential across various practical applications in different domains. In many scenarios, data updates dynamically, necessitating real-time adjustments to its order. For instance, in stock trading, prices fluctuate continuously, requiring the ranking of stocks to be updated in real-time. Similarly, in file management systems, files are frequently added, deleted, or modified, and the files need to be sorted based on criteria such as names, sizes, or creation dates. Additionally, in the news media, events occur every second, prompting editors to sort articles by time and popularity to determine their placement in publications. As a result, there is a significant demand for robust methodologies capable of efficiently sorting dynamically updating data into ordered sequences.

The dynamic sorting problem can be defined as follows. Given an order relation $\leq$ (e.g., numerically smaller or equal) and two list of data, the current data $L_{\text{current}}[0 : p-1]$ with length p, and the upcoming data $L_{\text{upcoming}}[0 : q-1]$ with length q, the dynamic sorting problem can be seperated into two sub-problems. Firstly, the current data L_{current} needs to be sorted into a list L'_{current} that the elements in L'_{current} are in monotonic order, and L'_{current} is a permutation of L_{current}. Secondly, the elements in the upcoming data L_{upcoming} need to be inserted into the sorted list L'_{current} one by one, while maintaining the monotonic order of the list. The final output of the dynamic sorting problem is the sorted list $L_{\text{final}}[0 : (p+q)-1]$ that contains all elements in L_{current} and L_{upcoming}, and the elements in L_{final} are in monotonic order. For instance, given the current data $L_{\text{current}} = [0, 8, 5, 1, 4, 10, 7]$ and the upcoming data $L_{\text{upcoming}} = [3, 6, 9, 2]$ with the numerical order relation $\leq$, the sorted list $L'_{\text{current}} = [0, 1, 4, 5, 7, 8, 10]$ is firstly obtained from the current data L_{current}. Secondly, the elements in the upcoming data L_{upcoming} are inserted into the sorted list L'_{current} one by one. After inserting element 3, the sorted list updated to $[0, 1, 3, 4, 5, 7, 8, 10]$. After inserting element 6, the sorted list updated to $[0, 1, 3, 4, 5, 6, 7, 8, 10]$. Similarly, after inserting element 9 and 2, the final output of the dynamic sorting problem is $L_{\text{final}} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. The first sub-problem of the dynamic sorting problem is same as the static sorting problem proposed in [1], but the second sub-problem that dynamically updates the sorted list is a novel definition that distinguishes the dynamic sorting problem from the static sorting problem.

As a sub-problem of the dynamic sorting problem, the static sorting problem is firstly defined in [1]. Along with defining the static sorting problem, an algorithm called *MaxSort* is proposed. The *MaxSort* algorithm is an intuitive and effective algorithm to solve the sorting problem, which iteratively searches for the greatest element in the unsorted list and inserts it to the leftmost of the sorted list. However, the efficiency of the *MaxSort* algorithm is suboptimal due to the redundancy of the comparisons between the elements. In [2], a new sorting algorithm called *HeapSort* is proposed to solve the static sorting problem. By leveraging a tree-based data structure called heap, the *HeapSort* algorithm significantly improves the scalability of the static sorting problem by reducing the time complexity of the sorting algorithm from $O(n^2)$ of the *MaxSort* algorithm to $O(n \log n)$.

While both the *MaxSort* and *HeapSort* algorithms are effective in solving the static sorting problem, they are not directly applicable to the dynamic sorting problem. Since both algorithms are only designed for the static sorting problem, they do not keep any addition information of the input data other than the sorted list after the static sorting is completed. For the *MaxSort* algorithm, the unsorted list is empty after the static sorting. Similarly, for the *HeapSort* algorithm, all the elements are removed from the heap after the static sorting. As a result, to maintain the monotonic order of the sorted list after inserting the upcoming data, the comparisons between the elements need to be repeated from scratch. To put the newly inserted element into the correct position, it is necessary to compare the new element with elements in the sorted list one by one. Such a linear search

process is repeated for every element in the upcoming data, resulting in the redundancy of the comparisons between the elements, and finally leading to the low efficiency of the dynamic sorting.

Motivated by this, we propose a novel dynamic sorting algorithm called *B+ Tree Sort*. To avoid the redundancy of the comparisons between the elements during the dynamic sorting process, we leverage a tree-based data structure called *B+ tree*, which can structurally store the sorted elements and update the sorted list efficiently. The B+ tree is a balanced tree data structure, where the internal nodes store the keys for navigation, and the leaf nodes store the data elements. The elements in the leaf nodes of a B+ tree are in the sorted order. Inserting a new element into a B+ tree and maintaining the sorted order of the elements in the leaf nodes can be done in $O(\log n)$ time, where n is the number of elements in the B+ tree. Based on the B+ tree data structure, we propose the *B+ Tree Sort* algorithm to solve the dynamic sorting problem. For the first sub-problem of the dynamic sorting problem, given a list of current data $L_{\text{current}}[0 : p-1]$, the *B+ Tree Sort* algorithm builds a B+ tree with the elements in L_{current}, and the sorted list L'_{current} is obtained by traversing the leaf nodes of the B+ tree. This process takes $O(p \log p)$ time, which has the same time complexity as the *HeapSort* algorithm. For the second sub-problem of the dynamic sorting problem, given a list of upcoming data $L_{\text{upcoming}}[0 : q-1]$, the *B+ Tree Sort* algorithm inserts the elements in L_{upcoming} into the B+ tree one by one. After inserting each element, the updated sorted list is obtained by traversing the leaf nodes of the B+ tree. The process of inserting each element into the B+ tree and updating the sorted list takes $O(\log(p+q))$ time, which is more efficient than the linear search process which takes $O(p+q)$ time. After inserting all elements in the upcoming data L_{upcoming}, the final output of the dynamic sorting problem L_{final} can be obtained from the final B+ tree. In general, when solving the dynamic sorting problem, the *B+ Tree Sort* algorithm is more efficient than existing solutions designed for static sorting problem. For the first sub-problem of the dynamic sorting problem, the *B+ Tree Sort* algorithm has the same time complexity as the *HeapSort* algorithm, where the *HeapSort* algorithm is shown to be the most efficient algorithm for the static sorting problem to the best of our knowledge. For the second sub-problem of the dynamic sorting problem, existing solutions designed for static sorting problem cannot be directly applied, so the additional linear search process is required, which is less efficient than the element insertion process in the *B+ Tree Sort* algorithm. We conduct experiments on both synthetic and real-world datasets to evaluate the effectiveness and efficiency of the *B+ Tree Sort* algorithm. The results show the superiority of the *B+ Tree Sort* algorithm over existing solutions on time efficiency in the dynamic sorting problem.

In summary, our contributions are shown as follows:

(1) We define the dynamic sorting problem, which is a novel problem that requires sorting the current data and inserting the upcoming data into the sorted list while maintaining the monotonic order of the list. To the best of our knowledge, we are

the first to define the dynamic sorting problem.

(2) We propose a novel dynamic sorting algorithm called *B+ Tree Sort*, which leverages the B+ tree data structure to efficiently solve the dynamic sorting problem. The *B+ Tree Sort* algorithm significantly improves the efficiency of the dynamic sorting problem by avoiding the redundancy of the comparisons between the elements. To the best of our knowledge, the *B+ Tree Sort* algorithm is the first algorithm proposed to solve the dynamic sorting problem.

(3) We conduct extensive experiments on both synthetic and real-world datasets to evaluate the effectiveness and efficiency of the *B+ Tree Sort* algorithm. The results show the effectiveness of the *B+ Tree Sort* algorithm, and the superiority of the *B+ Tree Sort* algorithm over existing solutions on time efficiency in the dynamic sorting problem.

The rest of the paper is organized as follows. In Section 2, we formally define the dynamic sorting problem. In Section 3, we review the related work of the dynamic sorting problem and the existing solutions. In Section 4, we introduce the *B+ Tree Sort* algorithm in detail. In Section 5, we conduct experiments to evaluate the effectiveness and efficiency of the *B+ Tree Sort* algorithm. In Section 6, we summarize the paper and discuss the future work.

2. Problem Definition

In this section, we formally define the dynamic sorting problem. The dynamic sorting problem consists of two sub-problems. The first sub-problem is to sort the current data into a sorted list, which is same as the static sorting problem proposed in [1], and the second sub-problem is to insert the upcoming data into the sorted list one by one while maintaining the monotonic order of the list.

Given an order relation $\leq$ and two lists of data, the current data $L_{\text{current}}[0:p-1]$ with length p, and the upcoming data $L_{\text{upcoming}}[0:q-1]$ with length q, the first sub-problem of the dynamic sorting problem is to sort the current data L_{current} into a sorted list L'_{current} that satisfies the following criteria:

(1) *Monotonic Order*: The output list L'_{current} is in monotonic order, i.e., each element in the output list is no smaller than the previous element. In formal language,

$$\forall i \in \{1, 2, \ldots, p-1\}, L'_{\text{current}}[i-1] \leq L'_{\text{current}}[i].$$

(2) *Permutation of input*: The output list L'_{current} is a permutation of the input list. Let $L_{\text{current}} = [e_0, e_1, \ldots, e_{p-1}]$, then there exists a bijection function (one-to-one

correspondence) $\sigma : \{0, 1, \ldots, p-1\} \mapsto \{0, 1, \ldots, p-1\}$ s.t.

$$L'_{\text{current}} = [e_{\sigma(0)}, e_{\sigma(1)}, \ldots, e_{\sigma(p-1)}].$$

The second sub-problem of the dynamic sorting problem is to insert the elements in the upcoming data L_{upcoming} into the sorted list L'_{current} one by one, while maintaining the monotonic order of the list. For the i-th insertion, where $i \in \{1, 2, \cdots, q\}$, denote the current sorted list before the i-th insertion as $L^{(i-1)}_{\text{sorted}}[0 : (p+i-2)]$. Accordingly, $L^{(0)}_{\text{sorted}} = L'_{\text{current}}$, and $L^{(i-1)}_{\text{sorted}}$ contains all p elements from the current data list L_{current} and the first $i-1$ elements in the upcoming data list L_{upcoming}. Denote the i-th element in the upcoming data as $e^{(i)}_{\text{upcoming}} = L_{\text{upcoming}}[i-1]$, the goal is to find the index $j \in \{0, 1, \cdots, (p+i-1)\}$ s.t.

$$L^{(i-1)}_{\text{sorted}}[j-1] \leq e^{(i)}_{\text{upcoming}} \leq L^{(i-1)}_{\text{sorted}}[j]$$

and insert the element $e^{(i)}_{\text{upcoming}}$ to the j-th position of the sorted list $L^{(i-1)}_{\text{sorted}}$, resulting in the updated sorted list $L^{(i)}_{\text{sorted}}[0 : (p+i-1)]$, where

$$L^{(i)}_{\text{sorted}} = \left[L^{(i-1)}_{\text{sorted}}[0], \cdots, L^{(i-1)}_{\text{sorted}}[j-1], e^{(i)}_{\text{upcoming}}, L^{(i-1)}_{\text{sorted}}[j], \cdots, L^{(i-1)}_{\text{sorted}}[p+i-2] \right].$$

After inserting all elements in the upcoming data L_{upcoming}, the final output of the dynamic sorting problem is the sorted list $L_{\text{final}}[0 : (p+q)-1]$ in monotonic order that contains all elements in the current data L_{current} and the upcoming data L_{upcoming}.

For instance, given the current data $L_{\text{current}} = [0, 8, 5, 1, 4, 10, 7]$ and the upcoming data $L_{\text{upcoming}} = [3, 6, 9, 2]$ with the numerical order relation $\leq$, the sorted current list $L'_{\text{current}} = [0, 1, 4, 5, 7, 8, 10]$ is firstly obtained from the current data L_{current}. Secondly, the elements in the upcoming data L_{upcoming} are inserted into the sorted list L'_{current} one by one. After inserting element 3, the sorted list updated to $L^{(1)}_{\text{sorted}} = [0, 1, \mathbf{3}, 4, 5, 7, 8, 10]$. After inserting element 6, the sorted list updated to $L^{(2)}_{\text{sorted}} = [0, 1, 3, 4, 5, \mathbf{6}, 7, 8, 10]$. Similarly, after inserting element 9 and 2, the final output of the dynamic sorting problem is $L_{\text{final}} = L^{(4)}_{\text{sorted}} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$.

3. Related Work

In this section, we review the related work of the dynamic sorting problem and the existing solutions.

The static sorting problem is firstly defined in [1], which aims to sort a given list of data into a sorted list that is a permutation of the input list and in monotonic order. The static sorting problem only considers the sorting of fixed existing data, but does not consider the dynamic update of the data. To better suimulate the real-world scenarios where the data updates dynamically, we define the dynamic sorting problem, which requires sorting

the current data and inserting the upcoming data into the sorted list while maintaining the monotonic order of the list.

Several methods have been proposed to solve the static sorting problem. The *MaxSort* algorithm is an intuitive algorithm proposed in [1] to solve the static sorting problem. The main idea of the *MaxSort* algorithm is to iteratively search for the greatest element in the unsorted list and insert it to the leftmost of the sorted list. However, the efficiency of the *MaxSort* algorithm is suboptimal due to the redundancy of the comparisons between the elements. In [2], a new sorting algorithm called *HeapSort* is proposed to solve the static sorting problem with improved efficiency. By leveraging a tree-based data structure called heap, the *HeapSort* algorithm significantly improves the scalability of the static sorting problem by reducing the time complexity of the sorting algorithm from $O(n^2)$ of the *MaxSort* algorithm to $O(n \log n)$.

While both the *MaxSort* and *HeapSort* algorithms are effective in solving the static sorting problem, they are not directly applicable to the dynamic sorting problem. Since both algorithms are only designed for the static sorting problem, other than the output sorted list, they do not retain any addition information of the input data once the static sorting is completed. For the *MaxSort* algorithm, the unsorted list is empty after the static sorting. Similarly, for the *HeapSort* algorithm, all the elements are removed from the heap after the static sorting. Consequently, to maintain the monotonic order of the sorted list after inserting the upcoming data, the comparisons between the elements need to be repeated from scratch. To put the newly inserted element into the correct position, it is necessary to compare the new element with elements in the sorted list one by one. Such a linear search process is repeated for every element in the upcoming data, resulting in the redundancy of the comparisons between the elements, and finally leading to the inefficient the dynamic sorting. Motivated by this, we propose a novel dynamic sorting algorithm called *B+ Tree Sort* to solve the dynamic sorting problem. Utilizing the B+ tree data structure, the *B+ Tree Sort* algorithm can structurally store the sorted elements after the static sorting process and dynamically update the sorted list efficiently.

4. Algorithm

In this section, we introduce the *B+ Tree Sort* algorithm in detail. The *B+ Tree Sort* algorithm is designed to solve the dynamic sorting problem by leveraging the B+ tree data structure. By constructing a B+ tree with the current data, the *B+ Tree Sort* algorithm can first sort the current data into a sorted list. After that, elements in the upcoming data are inserted into the B+ tree one by one, the sorted list is updated accordingly. Finally, after inserting all elements in the upcoming data, the final output of the dynamic sorting problem can be obtained from the B+ tree. In Section 4.1, we firstly introduce the B+ tree data structure. In Section 4.2, we describe the *B+ Tree Sort* algorithm in detail.

4.1 B+ Tree Data Structure

To avoid the redundancy of the comparisons between the elements during the dynamic sorting process, we propose a novel tree-based data structure called B+ tree. Intuitively, a B+ tree is a balanced tree data structure that structurally stores the elements in the given list and maintains the sorted order of the elements in the leaf nodes. In the following, we will first introduce the definition of the B+ tree, followed by the element insertion operation, and finally the theoretical analysis of the B+ tree data structure.

4.1.1 Definition

A B+ tree is a tree data structure that maintains sorted data and allows for efficient element insertion operations. A B+ tree is defined with the following properties:

(1) *Order*: The *order* of a B+ tree is the maximum number of pointers that a node can have, which is at least 3.

(2) *Node Structure*: A B+ tree consists of internal nodes and leaf nodes, both of which have the same structure. For a B+ tree of order m, each node contains at most m pointers $P_1, P_2, \ldots, P_m$ and $m-1$ search-key values $K_1, K_2, \ldots, K_{m-1}$. The pointers and search-key values are stored in the node in the following order: $P_1, K_1, P_2, K_2, \ldots, K_{m-1}, P_m$. The search-key values in a node are in monotonic order, i.e., $K_1 \leq K_2 \leq \ldots \leq K_{m-1}$.

(3) *Leaf Node*: In a leaf node, all the search-key values are the data elements. In a B+ tree of order m, each leaf node can hold at most $(m-1)$ search-key values and must hold at least $\lceil (m-1)/2 \rceil$ search-key values unless it is the root node. The last pointer P_m in a leaf node points to the next leaf node on the right, which is used to traverse the leaf nodes in the sorted order.

(4) *Nonleaf Node*: In a B+ tree of order m, each nonleaf node may hold at most m pointers and must hold at least $\lceil m/2 \rceil$ pointers. For $i \in \{2, 3, \cdots, m-1\}$, the i-th pointer P_i in a nonleaf node points to the subtree whose search-key values are greater than K_{i-1} and less than or equal to K_i. The first pointer P_1 points to the subtree whose search-key values are less than or equal to K_1, and the last pointer P_m points to the subtree whose search-key values are greater than or equal to K_{m-1}.

(5) *Balanced Tree*: A B+ tree is a balanced tree, where the length of every path from the root to the leaf nodes is the same.

An example of a B+ tree is shown in Figure 1. The B+ tree in Figure 1 is a B+ tree of order 3 which contains all the elements in the list $[0, 8, 5, 1, 4, 10, 7]$.

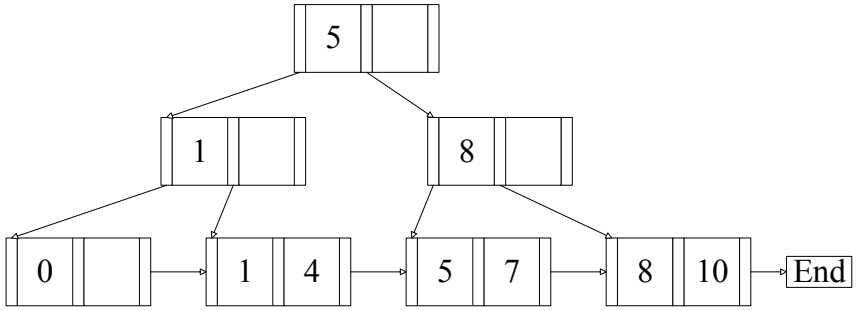


Figure 1: An B+ tree of order 3 containing all the elements in the list $[0, 8, 5, 1, 4, 10, 7]$.

4.1.2 Element Insertion

The element insertion operation is a fundamental procedure in maintaining a B+ tree. When a new element is to be added to the B+ tree, it must be placed in a position that preserves the sorted order of the elements in the leaf nodes, while maintaining the properties of the B+ tree. The process of inserting a new element into a B+ tree is shown as follows.

Algorithm 1 Element Insertion in B+ Tree

procedure InsertElement(B+ tree T, element K)
  if B+ tree is empty **then**
    Create a new leaf node with the element
    Set the new leaf node as the root of the B+ tree
  else
    Find the leaf node N that should contain the element
  end if
  if N has less than $(m-1)$ elements **then**
    InsertInLeaf(N, K)
  else
    Create a new leaf node $N^{'}$
    Copy $N.P_1 \cdots N.K_{n-1}$ to a block of memory T
      that can hold m (pointer, key-value) pairs
    InsertInLeaf(T, K)
    Set $N.P_m$ to point to $N^{'}$ and $N^{'}.P_n = N.P_n$
    Erase all the pointers and key-values in N
    Copy $[T.P_1, T.K_1, \cdots T.P_{\lceil m/2 \rceil}, T.K_{\lceil m/2 \rceil}]$ from T to N
    Copy $[T.P_{\lceil m/2 \rceil+1}, T.K_{\lceil m/2 \rceil+1}, \cdots T.P_m, T.K_m]$ from T to $N^{'}$
    $K_{\text{mid}} \leftarrow T.K_{\lceil m/2 \rceil+1}$
    InsertInParent(N, K_{mid}, $N^{'}$)
  end if
end procedure

Algorithm 2 Subsidiary Procedures for Element Insertion in B+ Tree

procedure InsertInLeaf(Leaf node N, element K)
  if $K < N.K_1$ **then**
    Insert K to the leftmost position of N
  else
    Find the position j s.t. $N.K_{j-1}$ is the largest element in N s.t. $N.K_{j-1} \leq K$
    Insert K to the j-th position of N
  end if
end procedure

procedure InsertInParent(Node N, element K, Node $N^{'}$)
  if N is the root of the B+ tree **then**
    Create a new root node R with K and pointers to N and $N^{'}$
    Set R as the root of the B+ tree
    return
  end if
  $P \leftarrow$ parent node of N
  if P has less than m pointers **then**
    Insert K and $N^{'}$ to P just after the pointer to N
  else
    Create a new node $P^{'}$
    Copy P to a block of memory T that can hold $m+1$ (pointer, key-value) pairs
    Insert K and $N^{'}$ to T just after the pointer to N
    Erase all the pointers and key-values in P
    Copy $[T.P_1, T.K_1, \cdots T.P_{\lceil (m+1)/2 \rceil}, T.K_{\lceil (m+1)/2 \rceil}]$ from T to P
    Copy $[T.P_{\lceil (m+1)/2 \rceil + 1}, T.K_{\lceil (m+1)/2 \rceil + 1}, \cdots T.P_m, T.K_m]$ from T to $P^{'}$
    $K_{\text{mid}} \leftarrow T.K_{\lceil (m+1)/2 \rceil}$
    InsertInParent(P, K_{mid}, $P^{'}$)
  end if
end procedure

As shown in Algorithm 1 and Algorithm 2, the general idea of inserting a new element into a B+ tree is to find the leaf node that should contain the inserting element. If there are too many elements in the leaf node after inserting the new element, the leaf node is split into two leaf nodes, and the pointer to the new node is inserted into the parent node. If the parent node has too many pointers after inserting the new pointer, a split operation is performed, and the split operation is recursively performed until either the root node is reached or an insertion does not cause an overflow.

An example of inserting a new element into a B+ tree is shown in Figure 2. The B+ tree in Figure 2 is the result of inserting the element 3 into the B+ tree shown in Figure 1. Since $3 < 5$ and $3 > 1$, the element 3 needs to be inserted to the leaf node containing the element 1 and 4. However, the leaf node has already contained 2 elements, so the leaf node is split into two leaf nodes, where the left leaf node contains the element 1, and the right leaf node contains the elements 3 and 4. The parent node of the leaf nodes is updated accordingly, and the B+ tree is updated.

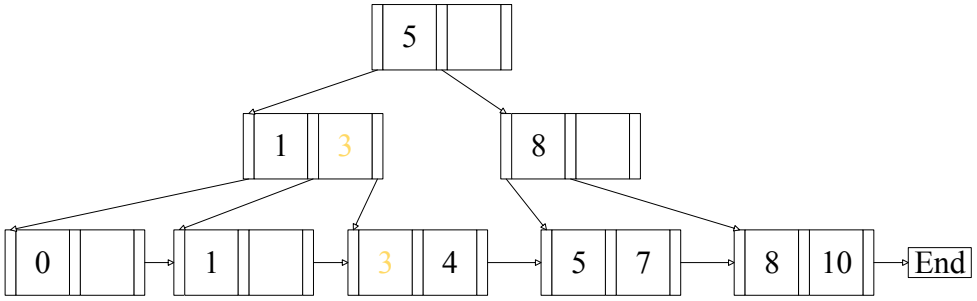


Figure 2: An example of inserting a new element into a B+ tree. The B+ tree in the figure is the result of inserting the element 3 into the B+ tree shown in Figure 1.

4.1.3 Theoretical Analysis

The time complexity of inserting a new element into a B+ tree is analyzed as follows. Since the B+ tree is a balanced tree, and each nonleaf node contains at least $\lceil m/2 \rceil$ pointers that point to child nodes, the height of the B+ tree is no larger than $\log_{\lceil m/2 \rceil} n$, i.e., $O(\log n)$, where n is the number of elements in the B+ tree. Since in the worst case, the insertion operation may cause a split operation from the leaf node to the root node, the time complexity of inserting a new element into a B+ tree is $O(\log n)$.

The space complexity of a B+ tree is analyzed as follows. Since all the elements in the B+ tree are stored in the leaf nodes as search-key values, and each leaf node contains at least $\lceil (m-1)/2 \rceil$ elements, the number of leaf nodes in a B+ tree is no larger than $n/(\lceil (m-1)/2 \rceil)$, i.e., $O(n)$. As for the nonleaf nodes, since each nonleaf node contains at least $\lceil m/2 \rceil$ pointers, the number of nonleaf nodes in the level just above the leaf nodes is no larger than $n/(\lceil m/2 \rceil)$, and the number of nonleaf nodes that are 2 levels above the leaf nodes is no larger than $n/(\lceil m/2 \rceil)^2$. Similarly, the number of nonleaf nodes that are i levels above the leaf nodes is no larger than $n/(\lceil m/2 \rceil)^i$. Since the height of the B+ tree is $O(\log n)$, and $\lceil m/2 \rceil \geq \lceil 3/2 \rceil = 2$, the total number of nonleaf nodes in a B+ tree is

$$\sum_{i=1}^{\log n} \frac{n}{(\lceil m/2 \rceil)^i} = O(n).$$

In general, for a B+ tree consists of n elements, it contains $O(n)$ leaf nodes and $O(n)$ nonleaf nodes, resulting in the space complexity of $O(n)$.

4.2 B+ Tree Sort Algorithm

Based on the B+ tree data structure, we propose the *B+ Tree Sort* algorithm to solve the dynamic sorting problem. The *B+ Tree Sort* algorithm consists of two main steps. The first step is to sort the current data into a sorted list by constructing a B+ tree with the elements in the current data, and the second step is to insert the elements in the upcoming data into the B+ tree one by one. The temporary sorted list can be obtained by traversing the leaf nodes of the B+ tree throughout the dynamic sorting process.

Algorithm 3 B+ Tree Sort Algorithm

procedure B+TreeSort(Current data L_{current}, Upcoming data L_{upcoming})
    Construct a B+ tree T with all the elements in L_{current}
    $L'_{\text{current}} \leftarrow$ TraverseLeafNodes(T) ▷ The sorted current data list
    for $i \leftarrow 0$ to $q-1$ **do**
        InsertElement(T, $L_{\text{upcoming}}[i]$)
        $L_{\text{sorted}} \leftarrow$ TraverseLeafNodes(T) ▷ The updated sorted list after insertion
    end for
    return L_{sorted}
end procedure

procedure TraverseLeafNodes(B+ tree T)
    $L \leftarrow []$
    $N \leftarrow$ leftmost leaf node of T
    while $N \neq$ End **do**
        $L \leftarrow L + [N.K_1, N.K_2, \ldots, N.K_{m-1}]$
        $N \leftarrow N.P_m$
    end while
end procedure

The *B+ Tree Sort* algorithm is shown in Algorithm 3. To solve the first sub-problem of the dynamic sorting problem, which is to sort the current data into a sorted list, the *B+ Tree Sort* algorithm firstly constructs a B+ tree with the current data L_{current}. After that, the sorted current data L'_{current} is obtained by traversing the leaf nodes of the B+ tree. For the second sub-problem of the dynamic sorting problem, which is to insert the elements in the upcoming data into the sorted list, the *B+ Tree Sort* algorithm iteratively inserts the elements in the upcoming data L_{upcoming} into the B+ tree one by one. After each insertion, the updated sorted list L_{sorted} is obtained by traversing the leaf nodes of the B+ tree. Finally, after inserting all elements in the upcoming data, the final output of the dynamic sorting problem L_{final} is the sorted list L_{sorted}.

An example of the *B+ Tree Sort* algorithm is shown in Figure 1, Figure 2, and Figure 3. Given the current data $L_{\text{current}} = [0, 8, 5, 1, 4, 10, 7]$ and the upcoming data $L_{\text{upcoming}} = [3, 6, 9, 2]$, the *B+ Tree Sort* algorithm firstly constructs a B+ tree with the current data L_{current} shown in Figure 1. By traversing the leaf nodes of the B+ tree, the sorted current data $L'_{\text{current}} = [0, 1, 4, 5, 7, 8, 10]$ is obtained. Next, the elements in the upcoming data L_{upcoming} are inserted into the B+ tree one by one. After inserting the element 3, the B+ tree is updated as shown in Figure 2. By traversing the leaf nodes of the updated B+ tree, the sorted list after inserting the element 3 is obtained as $[0, 1, 3, 4, 5, 7, 8, 10]$. After inserting the elements 6, 9, and 2 from the upcoming data list, the final updated B+ tree is shown in Figure 3. By traversing the leaf nodes of the final B+ tree, the final output of the dynamic sorting problem $L_{\text{final}} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ is obtained.

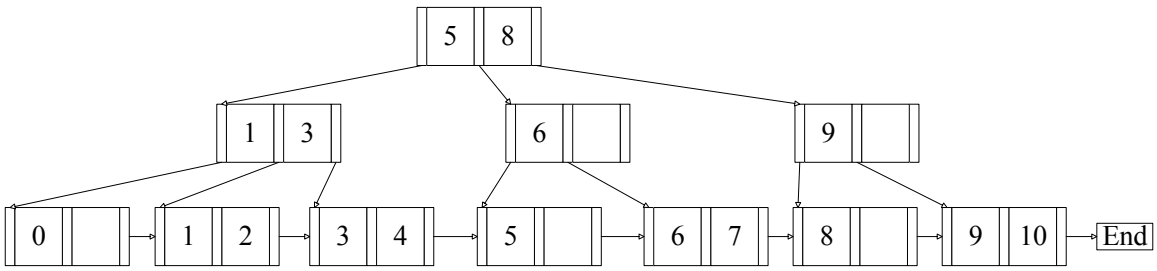


Figure 3: The B+ tree after inserting all the elements in $[3, 6, 9, 2]$ into the B+ tree shown in Figure 1. The B+ tree is the final B+ tree of *B+ Tree Sort* algorithm with the input current data as $[0, 8, 5, 1, 4, 10, 7]$ and the input upcoming data as $[3, 6, 9, 2]$.

5. Experiment

In this section, we present the experimental details of the *B+ tree sort* algorithm. We first introduce the experimental setup in Section 5.1. We then present the experimental results in Section 5.2. Finally, we summarize the experimental results in Section 5.3.

5.1 Experimental Setup

(1) **Algorithms:** To evaluate the performance of the *B+ Tree Sort* algorithm on the dynamic sorting problem, we compare the *B+ Tree Sort* algorithm with existing static sorting algorithms in the experiments. The *MaxSort* algorithm proposed in [1] and the *HeapSort* algorithm proposed in [2] are two existing solutions for the static sorting problem. However, as mentioned in Section 3, both algorithms are not directly applicable to the dynamic sorting problem. While handling the second sub-problem of the dynamic sorting problem, which is to insert the elements in the upcoming data into the sorted list, the *MaxSort* algorithm and the *HeapSort* algorithm need to linearly search for the correct position to insert the new element. We denote the combination of the *MaxSort* algorithm and the linear search process as the *MaxSort+* algorithm, and the combination of the *HeapSort* algorithm and the linear search process as the *HeapSort+* algorithm. In the experiments, we compare the *B+ Tree Sort* algorithm with the *MaxSort+* algorithm and the *HeapSort+* algorithm to evaluate the efficiency of the *B+ Tree Sort* algorithm.

(2) **Datasets:** To fully evaluate the performance of the *B+ Tree Sort* algorithm, we conduct experiments on both synthetic datasets and real-world datasets. The synthetic datasets denoted as RANDOM are generated by randomly permuting the list of integers $[0, 1, \ldots, n-1]$, where n is the size of the input list. The real-world datasets called REAL are obtained from the XXX repository [cite]. The REAL dataset contains 100,000 distinct integers ranging from 0 to 200,000. To simulate the dynamic sorting problem, we split the datasets into two parts, where the first 50% of the data is used as the current data and the second 50% of the data is used as the upcoming data. The statistics of the synthetic datasets and the real-world

Dataset	Total Size	Current Data Size	Upcoming Data Size
RANDOM	10,000 - 100,000	5000 - 50,000	5000 - 50,000
REAL	100,000	50,000	50,000

Table 1: Datasets statistics of the synthetic datasets and the real-world datasets.

datasets are shown in Table 1.

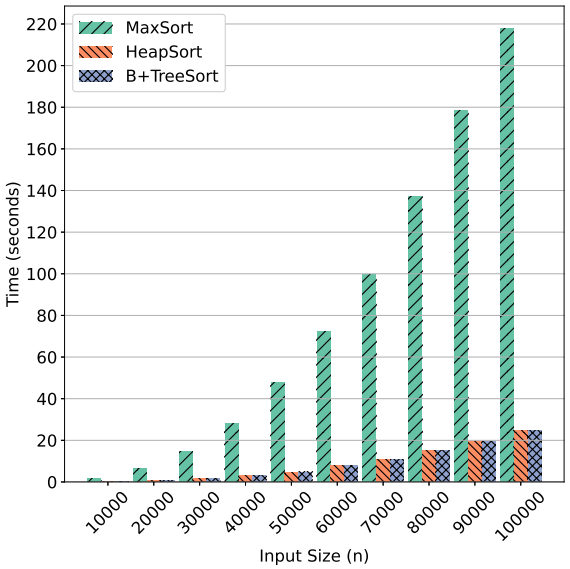
(3) **Factors:** The first factor of the experiment is the size of the synthetic datasets (i.e., n), which ranges from 10,000 to 100,000 with an interval of 10,000. The second factor of the experiment is the order of the B+ tree (i.e., m), which ranges from 3 to 7.

(4) **Evaluation Metrics:** To evaluate the time and space efficiency of the *B+ Tree Sort* algorithm, we measure the running time and the memory usuage of the algorithms on each dataset.

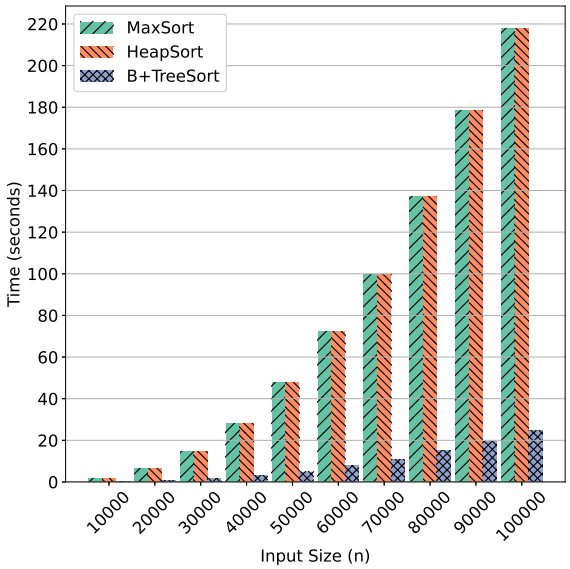
(5) **Implementation Details:** All the algorithms in the experiments are implemented in Python 3.8. The experiments are conducted on a Linux server with an Intel 1.9 GHz Intel Emerald Rapids CPU and 512GB of RAM.

5.2 Experimental Results

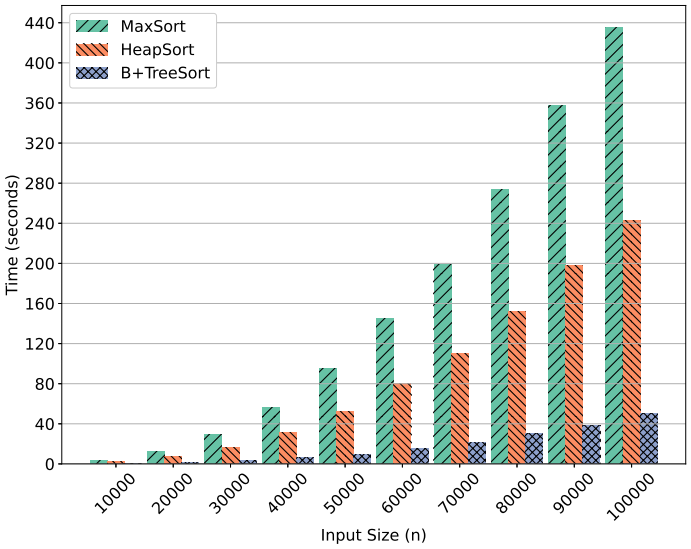
The experimental results of the time complexity of the *B+ Tree Sort* algorithm and the baseline algorithms *MaxSort+* and *HeapSort+* are shown in Figure 4. As shown in Figure 4a, the run time of the *B+ Tree Sort* algorithm on sorting the current data on the RANDOM dataset is generally the same as that of the HeapSort+ algorithm and significantly less than that of the *MaxSort+* algorithm. As for the run time of inserting the upcoming data, as shown in Figure 4b, the run time of the *B+ Tree Sort* algorithm is generally less than that of the *MaxSort+* algorithm and the *HeapSort+* algorithm. The reason for the superiority of the *B+ Tree Sort* algorithm is that the *B+ Tree Sort* algorithm can efficiently insert the elements in the upcoming data into the sorted list by leveraging the B+ tree data structure. As a result, as shown in Figure 4c, the total run time of solving the dynamic sorting problem of the *B+ Tree Sort* algorithm is generally less than that of the *MaxSort+* algorithm and the *HeapSort+* algorithm. When the size of the RANDOM dataset is set to be 100,000, which is the largest size in the experiments, the *B+ Tree Sort* algorithm takes around 40 seconds to solve the dynamic sorting problem, which is around 10 times faster than the *MaxSort+* algorithm and around 5 times faster than the *HeapSort+* algorithm. The experimental results on the REAL dataset are shown in Figure 4d. Consistent with the results on the synthetic datasets, the run time of the *B+ Tree Sort* algorithm on the REAL dataset is generally less than that of the *MaxSort+* algorithm and the *HeapSort+* algorithm. The superiority of the run time of the *B+ Tree Sort* algorithm is consistent with the theoretical analysis results, that the *B+ Tree Sort*



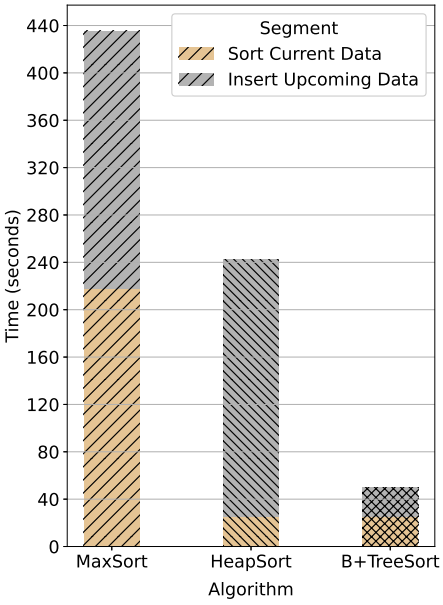
(a) Run time of the algorithms on sorting the current data on the RANDOM dataset.



(b) Run time of the algorithms on inserting the upcoming data on the RANDOM dataset.



(c) Total run time of the algorithms on the RANDOM dataset.



(d) Run time of the algorithms on the REAL dataset.

Figure 4: Run time of the algorithms on the RANDOM and REAL dataset. The experimental results are the average of 10 runs.

algorithm has the superiority of time efficiency in solving the dynamic sorting problem compared with the *MaxSort+* algorithm and the *HeapSort+* algorithm.

The experimental results of the space complexity of the *B+ Tree Sort* algorithm and the baseline algorithms are shown in Table 2. The memory usage of the *B+ Tree Sort* algorithm is generally slightly larger than that of the *HeapSort* algorithm and the *MaxSort* algorithm. When the size of the input list is set to be 100,000, the memory usage of the *B+ Tree Sort* algorithm is around 96MB, which is around 1MB larger than the *HeapSort*

Input Size	B+TreeSort	HeapSort	MaxSort
10,000	87.13	85.93	85.93
20,000	87.57	86.37	86.37
30,000	88.25	87.05	87.03
40,000	89.51	88.31	88.29
50,000	90.39	89.19	89.19
60,000	91.83	90.63	90.63
70,000	93.02	91.82	91.82
80,000	94.16	92.96	92.96
90,000	94.80	93.60	93.59
100,000	95.83	94.63	94.63

Table 2: Memory usage of the *B+ Tree Sort*, *HeapSort*, and *MaxSort* algorithms on the RANDOM dataset. The experimental results are the average of 10 runs, and the unit of the memory usage is MB.

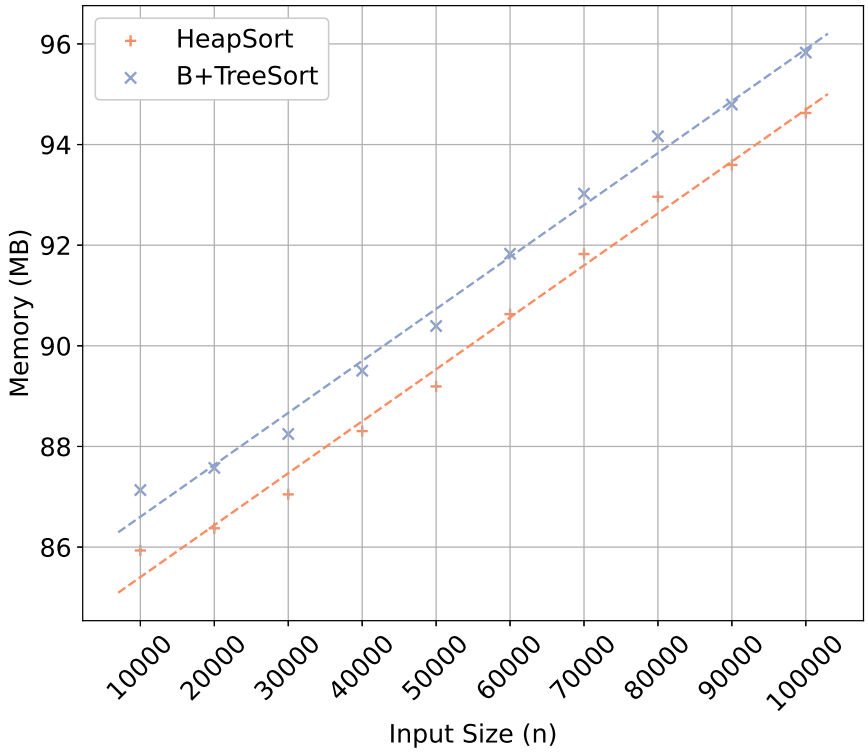


Figure 5: The plot of the memory usage of the *B+ Tree Sort* and the *HeapSort* algorithm on the RANDOM dataset.

algorithm and the *MaxSort* algorithm. As shown in Figure 5, the memory usage of the *B+ Tree Sort* algorithm is linearly increasing with the size of the input list. This is consistent with the theoretical analysis results that the space complexity of the *B+ Tree Sort* algorithm is $O(n)$.

5.3 Summary

The experiments demonstrate that the *B+ Tree Sort* algorithm is significantly more efficient than the baseline algorithms on both the synthetic datasets and the real-world datasets in term of time complexity, without significantly increasing the memory usage. When the size of the datasets are set to 100,000, which is the largest size in the experiments, the *B+ Tree Sort* algorithm solves the dynamic sorting problem in around 40 seconds on both the RANDOM and REAL datasets, which is around 10 times faster than the *MaxSort+* algorithm and around 5 times faster than the *HeapSort+* algorithm.

6. Conclusion

In this paper, we introduced the dynamic sorting problem, characterized by the need to efficiently manage and update a sorted list as new data is continuously inserted. We proposed the *B+ Tree Sort* algorithm, which leverages the B+ tree data structure to sort an initial dataset and perform logarithmic time insertions, thus avoiding the inefficiencies of existing static sorting algorithms in the dynamic sorting context. Our extensive experiments on synthetic and real-world datasets demonstrate that *B+ Tree Sort* significantly outperforms exsisting static sorting algorithms in terms of time complexity, while maintaining a space complexity of $O(n)$. Future work could explore the application of the *B+ Tree Sort* algorithm to other real-world problems, and investigate the potential for further optimization.

References

[1] W. Tan, 'MaxSort: A Method for Data Sorting through Interactively Moving the Largest Element,' en, Aug. 2024.

[2] W. Tan, 'HeapSort: An Efficient Sorting Algorithm Based on the Heap Data Structure,' en, Sep. 2024.